Design of elliptic curve cryptoprocessors over GF(2^{163}) using the Gaussian normal basis

Diseño de criptoprocesadores de curva elíptica sobre GF(2^{163}) usando bases normales Gaussianas

P.C. Realpe-Muñoz¹, V. Trujillo-Olaya² and J. Velasco-Medina³

ABSTRACT
This paper presents an efficient hardware implementation of cryptoprocesors that perform the scalar multiplication kP over a finite field GF(2^{163}) using two digit-level multipliers. The finite field arithmetic operations were implemented using the Gaussian normal basis (GNB) representation, and the scalar multiplication kP was implemented using the Lopez-Dahab algorithm, the 2-non-adjacent form (2-NAF) halve-and-add algorithm and the w-τNAF method for Koblitz curves. The processors were designed using a VHDL description, synthesized on the Stratix-IV FPGA using Quartus II 12.0 and verified using SignalTAP II and Matlab. The simulation results show that the cryptoprocesors provide a very good performance when performing the scalar multiplication kP. In this case, the computation times of the multiplication kP using the Lopez-Dahab algorithm, 2-NAF halve-and-add algorithm and 16-τNAF method for Koblitz curves were 13.37 µs, 16.90 µs and 5.05 µs, respectively.

Keywords: elliptic curve cryptography, Gaussian normal basis, digit-level multiplier, scalar multiplication.

RESUMEN
En este trabajo se presenta la implementación eficiente en hardware de criptoprocesadores que permiten llevar a cabo la multiplicación escalar kP sobre el campo finito GF(2^{163}) usando dos multiplicadores a nivel de digito. Las operaciones aritméticas de campo finito fueron implementadas usando la representación de bases normales Gaussianas (GNB), y la multiplicación escalar kP fue implementada usando el algoritmo de López-Dahab, el algoritmo de bisección de punto 2-NAF y el método w-τNAF para curvas de Koblitz. Los criptoprocesadores fueron diseñados usando descripción VHDL sintetizados en el FPGA Stratix-IV usando Quartus II 12.0 y verificados usando SignalTAP II y Matlab. Los resultados de simulación muestran que los criptoprocesadores presentan un muy buen desempeño para llevar a cabo la multiplicación escalar kP. En este caso, los tiempos de computo de la multiplicación kP usando Lopez-Dahab, bisección de punto 2-NAF y 16-τNAF para curvas de Koblitz fueron 13.37 µs, 16.90 µs and 5.05 µs, respectivamente.

Palabras clave: criptografía de curva elíptica, bases normales Gaussianas, multiplicador a nivel de digito, multiplicación escalar.

Received: October 29th 2013
Accepted: February 25th 2014

Introduction
The use of computer networks and the steady increase in the number of users of these systems have driven the need to improve security for the storage and transmission of information. There are many applications that must ensure the privacy, integrity or authentication of the information stored or transmitted. The security of the applications has been resolved by using different cryptographic algorithms, which are used in private- or public-key cryptosystems.

The security of public-key cryptosystems is based on mathematical problems that are computationally difficult to resolve, i.e., problems for which there are no known algorithms to resolve them in a practical time. Because of the high volume of information processed, electronic systems are required to perform the encryption and decryption processes in the shortest time possible without compromising the security. In this regard, hardware implementations of cryptographic algorithms have advantages, such as high speed, high security levels and low cost.

One of the most important cryptosystems is the elliptic curve cryptosystem (ECC), proposed independently by Koblitz (Kobliz, 1987) and Miller (Miller, 1986). There have been several investigations of the theory and practice of this cryptosystem. The results of the investigations demonstrated the ability of these systems to encrypt information and concluded that this cryptosystem offers

¹ P. C. Realpe-Muñoz. B.S in Physics Engineering, Universidad del Cauca, Colombia. M.Sc. in Electronics Engineering, Universidad del Valle, Colombia. Affiliation: Universidad del Valle, Colombia. E-mail: paulo.realpe@correounivalle.edu.co
² Vladimir Trujillo Olaya. B.S in Electronic Engineering, Universidad del Valle, Colombia. M. Sc. in Electronics Engineering, Universidad del Valle, Colombia. Affiliation: Universidad del Valle, Colombia. E-mail: vladimir.trujillo@correounivalle.edu.co
³ Jaime Velasco Medina. B.S in Electrical Engineering, University del Valle, Colombia. Ph.D in Microelectronics, TIMA-INP, France. Universidad del Valle, Colombia. E-mail: jaime.velasco@correounivalle.edu.co

better security, efficiency and memory usage. The hardware implementations of ECCs have many advantages and are used in equipment such as ATMs, smart cards, telephones, and cell phones.

In elliptic curve cryptography, it is known that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point, that is, the elliptic curve discrete logarithm problem or ECDLP, has high hardness. The entire security of the ECC depends on the ability to compute the scalar multiplication and the inability to compute the multiplicand given the original and product points. Furthermore, the finite-field size of the elliptic curve determines the computational complexity of the above problem.

Several works regarding scalar multiplication over a finite field \(GF(2^m)\) have been proposed and implemented efficiently in hardware.

C. Rebeiro and D. Mukhopadhyay (Rebeiro and Mukhopadhyay, 2008) presented a cryptoprocessor with novel multiplication and inversion algorithms. J.Y. Lai, T.Y. Hung, K.H. Yang and C.T. Huang (Lai et al., 2010) proposed an architecture for elliptic curves along with the operation scheduling for the Montgomery scalar multiplication algorithm. B. Muthukumar and S. Jeevananthan (Muthukumar and Jeevananthan, 2010) implemented an elliptic curve coprocessor, which is a dual-field processor with a projective coordinate. A.K. Rahuman and G. Athiha (Rahuman and Athiha, 2010) presented an architecture using the Lopez-Dahab algorithm for the elliptic curve point multiplication and Gaussian normal basis (GNB) for field arithmetic over \(GF(2^{163})\). M. Amara and A. Siad (Amara and Siad, 2011) proposed an EC point multiplication processor intended for cryptographic applications such as digital signatures and key agreement protocols. X. Cui and J. Yang (Cui and Yang, 2012) implemented a processor that parallelizes the computations of the ECC at the bit-level and gains a considerable speed-up. The processor is fully implemented in hardware and supports key lengths of 113 bits, 163 bits and 193 bits.

In this context, we present in this work efficient hardware implementations of cryptoprocessors over \(GF(2^{m})\) using a GNB representation and the Lopez-Dahab algorithm, 2-NAF halve-and-add algorithm and w-\(r\)NAF method for Koblit curves (Anomalous Binary Curves or ABC) with window sizes of 2, 4, 8 and 16 to perform the scalar multiplication \(kP\).

The main contributions of this work are: (i) the hardware design of cryptoprocessors using the GNB over \(GF(2^{163})\) and three scalar multiplication algorithms (Lopez-Dahab, halve-and-add and w-\(r\)NAF method for Koblit curves) to determine the best cryptoprocessor for embedded cryptographic applications. (ii) an efficient hardware implementation of cryptoprocessors based on the w-\(r\)NAF method with different window sizes for the Koblit curves. They present the best trade-off between the computation time and area, obtaining a higher performance than the other cryptoprocessors reported in the literature. Additionally, they are very suitable for hardware cryptosystems.

### Mathematical background

#### GNB representation

ANSI X9.62 (ANSI, 1999) describes the detailed specifications of the ECC protocols and uses the GNB to represent the finite field elements (NIST, 2000). An element over \(GF(2^m)\) has the computational advantage of performing squaring very efficiently. However, multiplying distinct elements can be cumbersome. In this case, there are multiplication algorithms that make this operation both simpler and more efficient.

A normal basis over \(GF(2^m)\) is as follows:

\[
\{\beta, \beta^2, \beta^3, \ldots, \beta^{m-1}\}
\]

where \(\beta \in GF(2^m)\) and any element \(A \in GF(2^m)\) can be written as:

\[
A = \sum_{i=0}^{m-1} a_i \beta^i, \quad a_i \in \{0,1\}
\]

The type \(T\) of a GNB is a positive integer and measures the complexity of the multiplication operation with respect to that basis. Generally, the type \(T\) of a smaller value provides more efficient multiplication. For a given \(m\) and \(T\), the field \(GF(2^m)\) can have at most one GNB of type \(T\). A GNB exists whenever \(m\) is not divisible by 8. Let \(m\) and \(T\) be two positive integers. Then, the type \(T\) of a GNB over \(GF(2^m)\) exists if and only if \(p = Tm + 1\) is prime.

If \(\{\beta, \beta^2, \beta^3, \ldots, \beta^{m-1}\}\) is a GNB over \(GF(2^m)\), then the element \(A = \sum_{i=0}^{m-1} a_i \beta^i\) is represented by the binary string \((a_1a_2\ldots a_{m-1})\), where \(a_i \in \{0,1\}\). In this case, the multiplicative identity element is represented by the bit string of all ones.

The additive identity element is represented by the bit string of all zeros. An important result for the GNB arithmetic is Fermat’s Theorem. For all \(\beta \in GF(2^m)\), then

\[
\beta^{2^m} = \beta
\]

This theorem is important for performing the squaring of an element over \(GF(2^m)\).

#### Finite field arithmetic operations

The following arithmetic operations can be performed over \(GF(2^m)\) when using a normal basis of type \(T\).

**Addition:** If \(A = (a_0a_1a_2\ldots a_{m-1})\) and \(B = (b_0b_1b_2\ldots b_{m-1})\) are elements over \(GF(2^m)\), then \(A + B = C = (c_0c_1c_2\ldots c_{m-1})\), where \(c_i = (a_i + b_i) \mod 2\).

**Squaring:** Let \(A = (a_0a_1a_2\ldots a_{m-1}) \in GF(2^m)\), then

\[
A^2 = \left(\sum_{i=0}^{m-1} a_i \beta^i\right)^2 = \sum_{i=0}^{m-1} a_i \beta^{2i} = \sum_{i=0}^{m-1} a_{m-i} \beta^{2i}
\]

Based on Fermat’s Theorem, \(\beta^{2^m} = \beta\), then

\[
A^2 = \left(\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} b_{R(i,j)} \beta^{i+j}\right)
\]

In this case, squaring is a simple rotation of the vector representation.

**Multiplication:** The multiplication \(C = AB\) is based on the multiplication matrix \(R_{m-1}\times T\) (Masoleh, 2006). If \(A = (a_0a_1a_2\ldots a_{m-1})\) and \(B = (b_0b_1b_2\ldots b_{m-1})\) are elements over \(GF(2^m)\) and are represented using a GNB, then \(AB = C = (c_0c_1c_2\ldots c_{m-1})\), where the coefficient \(c_0\) is given by equation (6)

\[
c_0 = a_0b_1 + \sum_{i=0}^{m-1} a_i \left(\sum_{j=0}^{m-1} b_{R(i,j)}\right)
\]

and \(R(i,j), 0 \leq R(i,j) \leq m - 1, 1 \leq i \leq m - 1, 1 \leq j \leq T\) denotes the \((i,j)\)th element of the \(R_{m-1}\times T\) matrix. To obtain the \(i\)th coefficient of \(C\), i.e., \(c_i\), add “i mod m” to all indices in (6).
**Inversion:** If $A \neq 0$ and $A \in \text{GF}(2^m)$, the inverse of $A$ is $C \in \text{GF}(2^m)$, and $C$ is the only element of $\text{GF}(2^m)$ such that $A \cdot C = 1$, i.e., $C = A^{-1}$. The algorithm used to calculate the inversion is based on equation (7):

$$A^{-1} = A^{2^m - 2} = (A^{2^{m - 1}})^2$$

Itoh and Tsujii (Itoh and Tsujii, 1998) proposed a method that reduces the number of multiplications to calculate the inversion, and it is based on the following:

$$A^{2^m - 2} = \begin{cases} A \cdot (A^{2^{m - 1}})^2 & \text{if } m \text{ even} \\ (A^{2^{m - 1}})^2 \cdot A^{2^{m - 1}} & \text{if } m \text{ odd} \end{cases}$$

**Trace:** If $A$ is an element over $\text{GF}(2^m)$, the trace of $A$ is:

$$Tr(A) = A + A^2 + A^4 + \ldots + A^{2^m - 1}$$

If $A = (a_0a_1a_2\ldots a_{m - 1})$ and it is represented in a normal basis, then the trace can be computed efficiently as follows:

$$Tr(A) = a_0 \oplus a_1 \oplus \ldots \oplus a_{m - 1}$$

The trace of the element $A$ has two possible values (0 or 1). Quadratic equation solving over $\text{GF}(2^m)$: If $A$ is an element of $\text{GF}(2^m)$ represented in a normal basis, then the quadratic equation:

$$z^2 + z = A$$

has $2 - 2T$ solutions over $\text{GF}(2^m)$, where $T = Tr(A)$. Therefore, if $T = 1$, there is no solution, and if $T = 0$, there are two solutions. If $z$ is one solution, then the other solution is $z + 1$. For example, if $A = 0$, the solutions are $z = 0$ and $z = 1$ (IEEE std 1363, 2000). The algorithm 1 calculates the quadratic equation over $\text{GF}(2^m)$ for a normal basis representation.

**Algorithm 1: Quadratic equation solving over $\text{GF}(2^m)$**

*Input:* An element $A \neq 0$

*Output:* An element $z$ for which $z^2 + z = A$

1. Let $(a_0a_1a_2\ldots a_{m - 1})$ be the representation of $A$
2. Set $z_0 \leftarrow 0$
3. For $i$ from 0 to $m - 1$
   1.1 Set $z_i \leftarrow z_i \oplus a_i$
4. Return $z \leftarrow (z_{2i}z_{2i+1})$

**Square root:** Let $A = (a_0a_1a_2\ldots a_{m - 1}) \in \text{GF}(2^m)$, then

$$\sqrt{A} = (a_0a_2\ldots a_{m - 2}a_0)$$

In this case, the square root in a normal basis is a simple rotation of the vector representation (IEEE std 1363, 2000).

**Elliptic curve arithmetic**

A non-supersingular elliptic curve $E(F_q)$ is defined as a set of points $(x, y) \in \text{GF}(2^m) \times \text{GF}(2^m)$ that satisfies the affine coordinates equation,

$$y^2 + xy = x^3 + ax^2 + b$$

where $a$ and $b \in F_q$ and are constants with $b \neq 0$ together with the point at infinity denoted by $O$. The group operations for the elliptic curve arithmetic in affine coordinates are defined as follows. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve, and let the addition inverse of $P$ be defined as $-P = (x_1, x_1 + y_1)$. Then, if $Q \neq -P$, the point $P + Q = (x_3, y_3)$ can be computed as:

$$x_3 = \begin{cases} \frac{y_1 + y_2}{x_1 + x_2} & P \neq Q \\ x_1 & P = Q \end{cases}$$

$$y_3 = \begin{cases} \frac{y_1 + y_2}{x_1 + x_2} (x_3 + y_3) & P \neq Q \\ x_1 y_1 & P = Q \end{cases}$$

Using the group operations above, the elliptic curve scalar multiplication can be defined as follows. Let $E$ be an elliptic curve over $\text{GF}(2^m)$, let $Q$ and $P \in E$ be two arbitrary elliptic points satisfying equation (13), and let $k$ be an arbitrary positive integer. Then, the elliptic curve scalar multiplication $Q = kP$ is defined as:

$$kP = P + P + \ldots + P \text{ (k times)}$$

Considering the group operations described in equations (14) and (15) using the finite field arithmetic in affine coordinates, three main elliptic curve operations can be defined: point addition, point doubling and point halving. In the group operations, the inversion is the arithmetic operation that is most expensive over $\text{GF}(2^m)$, and this operation can be avoided with a projective coordinate representation. In this case, the inversion is avoided by using the finite field multiplication.

A point $P$ in the projective coordinates is represented using three coordinates $(X, Y$ and $Z)$. For the Lopez-Dahab (LD) projective coordinates (Lopez and Dahab, 1999), the projective point $(X : Y : Z)$ with $Z \neq 0$ corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z$. Then, equation (13) can be mapped from the affine coordinates to the LD projective coordinates as:

$$Y^2 + XYZ = X^4Z + aX^2Z^2 + bZ^4$$

The three group operations for the elliptic curve arithmetic in the projective and affine coordinates can be computed as (Menezes et al., 2003):

1. **Point doubling** $Q = 2P$, where $Q = (X_3 : Y_3 : Z_3)$ and $P = (X_1 : Y_1 : Z_1)$ in the projective coordinates, can be performed using 4 finite field multiplications, such as

   $$Z_3 = X_1^2Z_1^3 \quad X_3 = X_1^4 + bZ_1^4$$

   $$Y_3 = bZ_1^2Z_3 + X_3(aZ_1 + Y_1^2 + bZ_1^2)$$

2. **Point addition** $Q + P$, where $Q = (X_1 : Y_1 : Z_1)$ in the projective coordinates and $P = (x_2, y_2)$ in the affine coordinates, can be performed using 8 finite field multiplications, such as

   $$A = y_2Z_1^2 + Y_1 \quad B = x_2Z_1 + X_1$$

   $$C = Z_1B \quad D = B^2(C + aZ_1^2)$$

   $$Z_3 = C^2 \quad E = AC$$

   $$X_3 = A^2 + D + E \quad F = X_3 + X_2Z_3$$

   $$G = (x_2 + y_2)Z_3^2 \quad Y_3 = (E + Z_3)F + G$$

3. **Point halving** $Q/2$ is the inverse operation of point doubling. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be the points over the curve (13) in the affine coordinates. The point halving operation is performed by computing $P$ such that $Q = 2P$ by solving the following equations:
\[ \lambda^2 + \lambda = x_1 + a \]  
(20)
\[ x_1 = \sqrt{y_2 + x_2(\lambda + 1)} \]  
(21)
\[ y_1 = \lambda x_1 + x_1^2 \]  
(22)

Let the \( \lambda \)-representation of a point \( Q = (x_2, y_2) \) be \( Q = (x, \lambda_0) \), where
\[ \lambda_Q = x_2 + y_2 \]  
(23)

If \( Q \) in the \( \lambda \)-representation is the input to the point halving algorithm, then it is possible to compute point halving without using the affine coordinates. In scalar multiplication, repeated point halving operations can be performed directly on the \( \lambda \)-representation. However, when a point addition is required, a conversion to the affine coordinates must be performed. Algorithm 2 computes the point halving operation.

**Algorithm 2: Point Halving**

Input: \( \lambda \)-representation \((x_1, \lambda_0)\) or affine representation \((x_2, y_2)\) of \( Q \)

Output: \( \lambda \)-representation \((x, \lambda)\) of \( P = (x_1, y_1) \)

1. Find a solution \( \lambda \) of \( \lambda^2 + \lambda = x_2 + a \) 
   - Else \( \lambda = x_2 + x_1 \)
2. If \( \text{Tr}(\lambda) = 0 \) \[ \lambda_0 = \lambda, \quad y_2 = \sqrt{\lambda + x_1} \]  
   - Else \[ \lambda_0 = \lambda + 1, \quad x_1 = \sqrt{\lambda} \]
3. Return \((x_1, \lambda_0)\)

**Koblitz Curves**

Koblitz curves, or anomalous binary curves, are elliptic curves defined over \( \text{GF}(2^n) \). The main advantage of these curves is that the scalar multiplication operation can be performed without the use of point doubling operations.

An algorithm for scalar multiplication on Koblitz curves is presented by Solinas (Solinas, 2000). The Solinas algorithm or the \( \tau \)-adic window method computes a special \( \tau \)-adic expansion of an integer number in \( \mathbb{Z}[\tau] \). For example, a special \( \tau \)-adic expansion is the window \( \tau \)-adic non-adjacent form (\( \tau \)-NAF).

The Koblitz curves are defined over \( \text{GF}(2^n) \) by:
\[ E_a = y^2 + xy = x^3 + ax + 1 \]  
(24)

where \( a \in \{0, 1\} \), that is, curves \( E_0 \) and \( E_1 \).

These curves present the following property: If \( P(x, y) \) is a point on the curve \( E_a \), then the point \((x^2, y)\) is also a point on \( E_0 \). In addition, they satisfy \( (x^2, y^2) + 2(x, y) = \mu(x^2, y^2) \) for each point \((x, y)\) on \( E_a \), where \( \mu = (-1)^{a} \). In \( \text{GF}(2^n) \), the Frobenius map \( \tau \) is an endomorphism that raises every element to its power of two, i.e., \( \tau : x \to x^2 \). Then, the Frobenius endomorphism is performed efficiently (cost-free) when the elements of the finite field are represented in a normal basis (Cui and Yang, 2012). Koblitz shows that the point doubling operation can be performed efficiently by using the Frobenius endomorphism, if the binary curve is defined over \( \text{GF}(2^n) \) and \( a \in \{0, 1\} \). Then, the Frobenius map can be defined as \( \tau : (x, y) \to (x^2, y^2) \). In this case, if the scalar \( k \) is represented in \( \tau \)-NAF, then
\[ \sum_{i=0}^{l-1} k_i \tau^i \quad \text{for } k_i \in \{0, 1, -1\} \]  
(25)

The \( \tau \)-adic representation can be obtained by repeatedly dividing \( k \) by \( \tau \), where the remainders of each division step are named digits \( u \). This procedure is also used to obtain the representation’s NAF of the scalar \( k \), namely, \( k \) is repeatedly divided by \( 2 \). To decrease the number of point additions for the scalar multiplication, it is necessary to obtain a \( \tau \)-NAF representation of \( k \) that achieves a smaller number of nonzero digits. The scalar multiplication can be computed as:
\[ kP = \sum_{i=0}^{l-1} k_i \tau^i (P) \]  
(26)

The result corresponds to the Hamming weight of the \( \tau \)-NAF, and it is equal to the binary NAF representation, i.e., the Hamming weight \( = (\log_2 k) / 3 \), and the length of the \( \tau \)-adic representation of \( k \) is approximately \( 2m \), which is twice the length of the binary NAF representation. However, Solinas presents a method that reduces the length of the \( \tau \)-adic representation to approximately \( m \). Thus, the Koblitz curves’ arithmetic is based on the point addition and Frobenius map \( \tau \).

**Hardware architectures for elliptic curve cryptoprocessors**

In this section, we present the hardware architectures for elliptic curve cryptoprocessors over \( \text{GF}(2^{163}) \) using a Gaussian normal basis. Each cryptoprocessor is designed using one algorithm for the scalar multiplication, namely, the Lopez-Dahab algorithm (Lopez and Dahab, 1999), the halve-and-add 2-NAF algorithm (Menezes et al., 2000) and the \( w \)-\( \tau \)-NAF method for Koblitz curves with \( w = 2, 4, 8, 16 \) (Solinas, 2000).

**Digit-level multiplier**

The finite field multiplication over \( \text{GF}(2^n) \) is an operation that is important for performing the scalar multiplication. Thus, this operation must be implemented efficiently in hardware. There are several algorithms for performing the finite field multiplication that are presented in Azarderakhsh and Masoleh (2010), Huang et al. (2011), Wang and Fan (2012) and Lee and Chiu (2012).

Azarderakhsh and Masoleh (Azarderakhsh and Masoleh, 2010) proposed a serial or parallel digit-level multiplier with a digit-size \( d \), where \( 1 \leq d \leq m \). In this case, if \( d = m \), the multiplier is parallel and if \( d < m \), it is serial and requires \( M = \lceil m / d \rceil \leq m \leq m \), clock cycles to generate all the \( m \) coefficients of \( C = AB = (c_0, c_1, \ldots, c_{m-1}) \), where \( A = (a_0 a_1 \ldots a_{m-1}) \) and \( B = (b_0 b_1 \ldots b_{m-1}) \) are elements represented in a GNB over \( \text{GF}(2^n) \). Figure 1 shows the digit-level \( \text{GF}(2^n) \) multiplier for \( T = 4 \), where \( A, B \) and \( C \) are registers for storing the input and output elements.

![Figure 1. Digit-level Multiplier](image-url)
block CS is a d-fold cyclic shift and an adder GF \((2^{163})\), which is a set of two-input XOR gates.

The block \(p_1\) is an optimal set of XOR gates that are obtained using (27), and \(p_2\) is a set of XOR gates that are obtained from the main matrix \(p\):

\[
P_1(B) = (b_{i-1} + s_i(B < k), s_i(2, B < k), \ldots, s_i((m-1)/2, B < k), 0 \leq k \leq d - 1
\]

\[
s_i(k, B) = \sum_{j=0}^{b-1} b_{i-k}
\]

The time complexity of the digit-level multiplier is \(T_x + (2 + \lceil \log m \rceil)T_s\), where \(T_x\) and \(T_s\) are the delay time of a two-input XOR gate and a two-input AND gate, respectively. The area complexity of this multiplier is \(m^2\) ANDs and \(\leq 2m^2 - 2m\) XORs (Azarderakhsh and Masoleh, 2010).

To implement the digit-level multiplier with a digit-size \(d = 55\) in hardware, that is \(M = 3\) clock cycles, a Matlab code is written to generate the equations of the blocks \(p_1\) and \(p_2\), which are synthesized using VHDL.

**Hardware architecture using the Lopez-Dahab algorithm**

The scalar multiplication \(kP\) for non-supersingular elliptic curves over binary fields using the Lopez-Dahab algorithm is shown in Algorithm 3, which is a modified version of the Montgomery algorithm, where the same operations are performed during each iteration of the main loop (D. Hankerson et al., 2003).

**Algorithm 3: Montgomery point multiplication**

Input: \(k = (k_0, k_1, \ldots, k_{163})\) with \(k_1 = 1, P = (x, y) \in GF(2^{163})\)

Output: \(kP\)

1. \(X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2\)
2. For \(i\) from \(t = 2\) downto \(0\) do
   3. if \(k_i = 1\) then
      3.1 \(T \leftarrow Z_2, Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2, X_1 \leftarrow xZ_1 + X_1X_1TZ_2\)
      3.2 \(T \leftarrow X_2, X_1 \leftarrow T^2Z_1^2, Z_2 \leftarrow TZ_1^2\)
   4. else
      4.1 \(T \leftarrow Z_2, Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2, X_1 \leftarrow xZ_1 + X_1X_1TZ_2\)
      4.2 \(T \leftarrow X_2, X_1 \leftarrow T^2Z_1^2, Z_2 \leftarrow TZ_1^2\)
5. \(x_3 \leftarrow X_1/Z_1, y_3 \leftarrow (x + x_3/z_1)(x_1 + xZ_1)(x_2 + xZ_2) + (x^2 + y)(Z_2Z_1)\)
6. Return \((x_3, y_3)\)

In this case, the scalar multiplication is performed in three steps: 1) conversion of \(P\) from affine to projective coordinates; 2) compute \(Q = kP\) by addition and doubling; and 3) conversion of \(Q\) from projective to affine coordinates.

To implement the above algorithm in hardware, we initially define three functions: \(M_{add()}\) performs the point addition, \(M_{double()}\) performs the point doubling and \(M_{inv()}\) performs the conversion from projective to affine coordinates. These functions are defined as follows:

\[
M_{add}(X_1, Z_1, X_2, Z_2)
\]

\[
X_1 \leftarrow X_1Z_2X_2Z_1 + x(X_2Z_2 + X_2Z_1)^2, \quad Z_2 \leftarrow X_2^2Z_2^2
\]

\[
M_{double}(X_1, Z_1)
\]

\[
X_2 \leftarrow X_2^2 + bZ_2^2
\]

\[
Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2
\]

Return \((X_1, Z_1)\) and \((X_2, Z_2)\)

where, \((x, y)\) and \((x_1, y_1)\) are the coordinates of points \(P\) and \(Q = kP\), respectively.

Point addition and point doubling are implemented in hardware using the data dependence graph shown in Figure 2, and the conversion from the projective to affine coordinates is implemented using two digit-level multipliers for the data dependence graph shown in Figure 3. The inversion operation is implemented using the Itoh-Tsujii algorithm (Itoh and Tsujii, 1998).

**Figure 2. Data dependence graph for \(M_{add}\) and \(M_{double}\)**

According to Figures 2 and 3, the latencies for \(M_{add}\) and \(M_{double}\) and the projective to affine conversion are \(3M\) and \(15M + 1\), respectively, where \(M\) is the latency for a finite field multiplication.

In step 4 of Figure 3, two multipliers are used, and one of them with the block of rotation performs the inversion of an element \(A \in GF(2^{163})\). In this case, the latency of the inversion is \(10M\) because it needs 10 finite field multiplications for \(m = 163\). In step 6, a multiplier is only used because the last operation of the coordinate conversion requires a multiplication.
The architecture of the cryptoprocessor over \(GF(2^{163})\) using the Lopez-Dahab algorithm is shown in Figure 4. It uses two register files, two parallel digit-level multipliers, one inversion block, several squaring and adder blocks, a main control and an FSM to perform the point addition, point double and conversion from the projective to affine coordinates.

![Figure 4. Elliptic curve cryptoprocessor using the Lopez-Dahab algorithm](image)

The functional blocks that perform the finite field arithmetic operations over \(GF(2^{163})\) for the Lopez-Dahab cryptoprocessor are shown in Figure 5. It is important to mention that the performance of any cryptoprocessor depends on the efficient implementation of the hardware for the finite field arithmetic.

![Figure 5. Functional blocks of the finite field arithmetic](image)

The main control is an FSM that generates the control signals to perform the scalar multiplication, process the key, initialize the cryptoprocessor and control the I/O registers. The second FSM performs the point addition, point doubling and conversion from the projective to the affine coordinates.

In Figure 6, the ASM chart of the main control is shown, where the variables \(X_i, Z_i, X_0\) and \(Z_0\) are initialized and stored in the register files. Each bit of the scalar \(k\) is evaluated from left to right to perform the operations \(M_{\text{add}}\) and \(M_{\text{double}}\) using the data dependence graph shown in Figure 2. If the bit \(k_i\) is ‘1’, then \(M_{\text{add}}(X_i, Z_i, X_{i+1}, Z_{i+1})\) or \(M_{\text{double}}(X_i, Z_i, X_{i+1}, Z_{i+1})\) are computed. Else, \(M_{\text{add}}(X_i, Z_i, X_{i+1}, Z_{i+1})\) or \(M_{\text{double}}(X_i, Z_i, X_{i+1}, Z_{i+1})\). When all bits of the scalar \(k\) are evaluated, the conversion from the projective to affine coordinates is executed using the data dependence graph shown in Figure 3, and \(kP\) in the affine coordinates is stored in the output register.

Algorithm 3 is more resistant against simple power analysis and timing attacks. This is because the computation cost does not depend on the specific bit of the scalar \(k\). For each bit of the scalar \(k\), one point addition and one point doubling are performed. The proposed scheme has two different execution paths depending on the current bit of the scalar \(k\). Both execution paths have the same complexity and require the same number of clock cycles.

### Hardware architecture using the halve-and-add algorithm

Schroeppel (Schroeppel, 2000) and Knudsen (Knudsen, 1999) independently proposed the halve-and-add algorithm to accelerate the scalar multiplication on the elliptic curves defined over the binary extension fields. This algorithm uses an elliptic curve primitive called point halving as shown in Algorithm 2.

Because, theoretically, the point halving operation is three times faster than the point doubling operation, it is possible to accelerate the scalar multiplication \(Q = kP\) by replacing the double-and-add algorithm with the halve-and-add algorithm, which uses an expansion of the scalar \(k\) in terms of negative powers of 2 (Mercurio et al., 2006).

In the halve-and-add algorithm, it is necessary to transform the integer \(k = (k_m \ldots k_0)\) into a window-NAF. The NAFw of a positive integer \(k\) and \(w \geq 2\) is represented by the expression

\[
\text{NAF}_w(k) = \sum_{i=0}^{\lfloor \log_2 (|k|) \rfloor} \lfloor \frac{k}{2^w} \rfloor 2^w
\]

where \(n\) represents the order of the base point, then

\[
kP = \sum_{i=0}^{\lfloor \log_2 (|k|) \rfloor} k_i / 2^w P
\]

Equation (29) can be generalized to an window-NAF. The NAFw of a positive integer \(k\) and \(w \geq 2\) is represented by the expression

\[
k = \sum_{i=0}^{\lfloor \log_2 (|k|) \rfloor} k_i 2^w
\]

In this work, a Maple code is written to obtain the expansion coefficients NAFw with \(w = 2\), namely, the coefficients NAFw(2^-1 k mod n), which are represented by 2-bits.

The halve-and-add algorithm is shown in algorithm 5. Step 3 of the algorithm performs the point addition \(P_i + P\) in the Lopez-Dahab mixed coordinates \((Q, P)\) are represented in LD projective and affine coordinates, respectively, using equation (14) and the halving point \(P/2\) in the affine coordinates or \(\lambda\)-representation, if bit \(k_i\) is odd and at most, one of any \(w\) consecutive digits is nonzero. In this case, the NAFw of \(k\) can be computed using algorithm 4.

![Figure 6. ASM chart of the main control](image)
in the LD projective coordinates using equation (18) with \( X_i = x_i \)
\( Y_i = y_i \) and \( Z_i = 1 \).

**Algorithm 5: Halve-and-add w-NAF point multiplication**

**Input:** Window width \( w \), NAF\(_w\) \((2^{t-1} \cdot \text{k mod } n) = \sum_{i=t}^{0} k_i 2^i \) \( P \in \text{GF}(2^m) \)

**Output:** \( kP \)

1. Set \( Q_i \leftarrow \infty \) for \( i \in \{1, 3, 5, \ldots, 2^{w-1} − 1\} \)
2. if \( k_i = 1 \) then \( Q_i = 2P \)
3. For \( i \) from \( t \) to \( 1 \) do
   3.1 if \( k_i > 0 \) then \( Q_i' = Q_i' + P \)
   3.2 if \( k_i < 0 \) then \( Q_i' = Q_i' - P \)
   3.3 \( P \leftarrow P/2 \)
4. \( Q \leftarrow \sum_{i=0}^{t} iQ_i \)
5. Return \( Q \)

**Figure 7. Data dependence graph for point addition**

**Figure 8. Data dependence graph for point doubling**

The point addition in the LD mixed coordinates and the point doubling in the LD projective coordinates are implemented in hardware using the data dependence graphs shown in Figure 7 and Figure 8, respectively. According to Figures 7 and 8, the latencies for the point addition and point doubling are \( 5M \) and \( M + 3 \), respectively.

The architecture of the cryptoprocessor over \( \text{GF}(2^{163}) \) using the halve-and-add algorithm is shown in Figure 9, and it uses two register files, two digit-level finite multipliers, one solving quadratic equation block, one point halving block, several squaring andadder blocks, a main control and an FSM to perform the point addition, point doubling and point halving.

**Figure 9. Elliptic curve cryptoprocessor using the halve-and-add algorithm**

The functional blocks that perform the finite field arithmetic operations over \( \text{GF}(2^{163}) \) for the halve-and-add cryptoprocessor are shown in Figure 10. In this case, finite field arithmetic operations are the addition, squarer, square root, trace, half trace (quadratic equation solving in a normal basis) and multiplication.

**Figure 10. Blocks of the finite field arithmetic**

The main control is an FSM that generates the control signals to perform the scalar multiplication, process the key, initialize the cryptoprocessor and control the I/O registers. The second FSM performs the point addition, point doubling and point halving.

**Figure 11. ASM chart of the main control**
In Figure 11, the ASM chart of the main control is shown, where the sequence processing is as follows: initialize coordinate $Q$ according to the sign of the bit $k'_{t-1}$; perform the point halving operation on $P$; evaluate the bit $k'_i$ for $i > t-1$; compute the point addition in the LD mixed coordinates and point halving on $P$ if $k'_i \neq 0$, else compute point halving; and perform the conversion of the point $P$ in the $\lambda$-representation to the affine coordinates only when a point addition is required. Finally, $Q = kP$ is obtained in the LD projective coordinates.

Algorithm 7 performs the rounding of a complex number $\lambda_0 + \lambda_1 \tau$ with $\lambda_0$ and $\lambda_1 \in Q$ to obtain an element in $\mathbb{Z}[\tau]$.

**Hardware architecture using the $w$-tNAF algorithm**

The length of the $\tau$-adic representation for $k = d_0 + d_1 \tau \in \mathbb{Z}[\tau]$ is roughly twice $\log_2(\max(d_0, d_1))$. Solinas (Solinas, 2000) presents a method that reduces the length of the $\tau$-adic representation. The objective is to find $p \in \mathbb{Z}[\tau]$ of small norm with $p = k \mod \delta$, where $\delta = (\tau^{n-1} - 1)/(\tau - 1)$, and use tNAF($p$) to calculate $pP$.

Algorithm 6 calculates an element $p^* = k \mod \delta$, which is also written as $p = k \mod \delta$. Solinas proved that $l(p) \leq m + a$ and if $C \geq 2$, then $l(p^*) \leq m + a + 3$.

**Algorithm 6: Partial reduction modulo $\delta=(\tau^{n-1} - 1)/(\tau - 1)$**

Input: $k \in [n-1]$, $C > 2$, $d_i = d_0 - \mu_i$, $V_m, s_i = -d_i$.
Output: $p^* = k \mod \delta$.
1. $k' = [k/2^{\lfloor \log_2 C \rfloor n/2}]$. 
2. For $i$ from 0 to $1$ do 
   2.1 $g_i \leftarrow s_i k'$; 
   2.2 $\lambda_i \leftarrow (g_i + 1)/2^{i/2} + i/2$. 
3. Use Algorithm 7 to compute $(q_k, q_i)$ \rightarrow Round($\lambda_0, \lambda_1$).
4. $r \leftarrow -k \cdot (d_0 + (s_0 + 1/2)q_1 + 2q_2 + q_3)$. 
5. Return $r \tau^i r^*$.

**Algorithm 7: Rounding off**

Input: Rational numbers $\lambda_0$ and $\lambda_1$.
Output: Integers $q_k$ and $q_i$.
1. For $i$ from 0 to $1$ do 
   1.1 $\beta_i \leftarrow |\lambda_i + 1/2|$. 
   1.2 $\alpha_i \leftarrow -\beta_i$; $h_i \leftarrow 0$.
2. $n \leftarrow 2n_0 + \mu_0 i$. 
3. If $n \geq 1$ then 
   3.1 If $n - 3 \mu_0 i < 1$ then $h_i \leftarrow \mu$; else $h_i \leftarrow -1$.
   3.2 If $n - 4 \mu_0 i \geq 2$ then $h_i \leftarrow -\mu$.
4. If $n < -1$ then 
   4.1 If $n_0 - 3 \mu_0 i \geq 1$ then $h_0 \leftarrow -\mu$; else $h_0 \leftarrow -1$.
   4.2 If $n_0 - 4 \mu_0 i < 2$ then $h_0 \leftarrow -\mu$.
5. $q_i \leftarrow \beta_i h_i$, $q_k \leftarrow \beta_i$, $h_i \leftarrow h_i$.
6. Return $(q_k, q_i)$.

Let $w \geq 2$ be a positive integer, and $a_i = i \mod t$ for $i \in \{1, 3, 5, ..., 2^{m-1} - 1\}$. A $w$-tNAF expansion of an nonzero element $\kappa \in \mathbb{Z}[\tau]$ is an expression:

$$\kappa = \sum_{i=1}^{t} u_i \tau^i$$

where $u_i \in \{0, \pm a_1, \pm a_2, ..., \pm a_{m-1}\}$, $u_{i-1} \neq 0$ and at most, one of any $w$ consecutive digits is nonzero. Then, $kP = a_0 P + a_1 \tau \alpha P + ... + a_{m-1} \tau^{m-1} P$, when the scalar $k$ is represented in $w$-tNAF.

The $w$-tNAF expansion can be efficiently computed using algorithm 8, which can be viewed as an approach similar to the general NAF algorithm. In this work, a Maple code is written to obtain the expansion $w$-tNAF of the scalar $k$ with $w = 2, 4$ and 8, generating 8-bit expansion coefficients and $w = 16$, generating 16-bit expansion coefficients.

**Algorithm 8: Computing a $w$-tNAF of an element in $\mathbb{Z}[\tau]$**

Input: $w, \tau, \kappa = kP \in \mathbb{Z}[\tau], a_i = \beta_i + \gamma_i$, for $u \in \{1, 3, 5, ..., 2^{m-1} - 1\}$
Output: $w$-tNAF($\kappa$)
1. $i \leftarrow 0$
2. While $n \not= 0$ and $r_i \not= 0$ do 
   2.1 If $n_0$ is odd then $u \leftarrow n_0 + r_0 \tau$, $\mod 2^w$.
   2.2 Else: $u_i \leftarrow 0$
3. $i \leftarrow i + 1$.
4. If $u > 0$ then $s \leftarrow 1$, else $s \leftarrow -1$, $u \leftarrow -u$
5. $r_0 \leftarrow r_0 - s \beta_0$, $r_i \leftarrow r_i - s \tau \alpha_0$.

Solinas proposed algorithms to compute $kP$ using the window tNAF method for the scalar $k$, namely, $kP$ is calculated using the $w$-tNAF method and Horner’s rule (Solinas, 2000). An efficient scalar multiplication algorithm that uses the $w$-tNAF method is presented in algorithm 9, where step 1 calculates the $w$-tNAF of the scalar $k$ with the partial reduction modulo $\delta = (\tau^{n-1} - 1)/(\tau - 1)$, namely, $w$-tNAF($p = k \mod \delta$), where $p = k \mod \delta$ is obtained from algorithms 6 and 7; step 2 generates the multiples of the point $P$ and step 4.2 performs the point addition $Q + P_u$, when the bit $u \neq 0$, and point doubling $Q_2$, when the results of the two first operations $A$ and $B$ of equation (19) are equal to zero.

**Algorithm 9: $w$-tNAF point multiplication method for Koblitz curves**

Input: Window width $w$, integer $k \in \{1, n - 1\}, P \in GF(2^w)$ of order $n$. 
Output: $kP$.
1. Compute $w$-tNAF($\rho = k \mod \delta$) $= \sum_{i=1}^{w} u_i \tau^i$.
2. Compute $P_u = a_i P, u \in \{1, 3, 5, ..., 2^{m-1} - 1\}$
3. $Q \leftarrow \omega_0$.
4. For $i$ from $l - 1$ downto $0$ do 
   4.1 $Q \leftarrow \tau^iQ$.
   4.2 If $u_i \not= 0$ then 
      Let $u$ be such that $a_u = u_i$ or $a_{-u} = -u_i$
      If $u > 0$ then $Q \leftarrow Q + P_u$.
      Else $Q \leftarrow Q - P_u$.
5. Return $Q$.

The point addition in the LD mixed coordinates and the point doubling in the LD projective coordinates with $b = 1$ are implemented in hardware using the data dependence graphs shown in Figure 7 and Figure 8, respectively.

---

**Figure 12. Elliptic curve cryptoprocessor for Koblitz curves**

The architecture of the cryptoprocessor over GF(2^{163}) using the $w$-tNAF algorithm for Koblitz curves is shown in Figure 12, and it uses two register files, two digit-level finite multipliers, one Frobenius map block, one RAM that stores the expansion coefficients $w$-tNAF of the scalar $k$, two ROMs that store the pre-computed
points \( P_i \) in the affine coordinates, which were obtained from Matlab for \( w = 2, 4, 8 \) and 16, several squaring and adder blocks, a main control and an FSM to perform the point addition, point doubling and \( tQ \).

The functional blocks that perform the finite field arithmetic operations over \( GF(2^{163}) \) for the \( w-\)tNAF cryptoprocessor for Koblitz curves are shown in Figure 13.

![Figure 13. Blocks of the finite field arithmetic](image)

The main control is an FSM that generates the control signals to perform the scalar multiplication, process the key, initialize the cryptoprocessor and control the I/O registers. The second FSM performs the point addition, point doubling and \( tQ \).

In Figure 14, the ASM chart of the main control is shown, where the sequence processing is as follows: initialize the \( Q \) coordinate according to the sign of the bit \( u_i \) of the \( w-\)tNAF expansion; evaluate the bits \( u_i \) for \( i > t-1 \); and compute the point addition in the LD mixed coordinates and the Frobenius map \( \tau \) on \( Q \), if \( u_i \neq 0 \). Else, compute \( tQ \). Finally, \( Q = kP \) is obtained in the LD projective coordinates. In Figure 14, the ASM of the FSM is shown. One important remark is that the Koblitz curves are resistant to simple power analysis and to all the known special attacks (T. Juhas, 2007).

![Figure 14. ASM chart of the main FSM](image)

### Hardware verification and synthesis results

The López-Dahab, halve-and-add and \( w-\)tNAF cryptoprocessors are described using generic VHDL, which are synthesized for a digit-size of \( d = 55 \) on the Stratix-IV FPGA (EP4SGX180F35C2) using the Altera Quartus II version 12 design software for the implementation and are verified using SignalTap II and Matlab.

### Hardware verification of the cryptoprocessors

To verify the synthesis and simulation results of the cryptoprocessors, the following parameters for a pseudo-random elliptic curve are used according to the National Institute of Standards and Technology (NIST, 2000):

1. Random elliptic curves \( B-163 \):
   - The form of the curve is: \( y^2 + xy = x^3 + x + b \)
   - \( Gx = 3F0EBA16286A2D57E909D168D9946378343E6 \)
   - \( Gy = 0DS1FBC6C71A009FA2CD545B1CC5C0C7937344FI \)
   - \( b = 20A61907BB953CA1481EB10512F7844A3205FD \)

2. Koblitz elliptic curves \( K-163 \):
   - The form of the curve is: \( y^2 + xy = x^3 + x + 1 \)
   - \( Gx = 2FE13C0537BCC1ACAA07D793DE46D55C094EE8 \)
   - \( Gy = 28907F0B05388F5821F2E800536D38CCDA3D9 \)
   - \( n = 40000000000000000000000000000000020108A2E00C099F8A5EF \)

In Figures 15 through 17, the simulation results for the cryptoprocessors over \( GF(2^{163}) \) in a GNB using SignalTap II and Matlab are shown.

![Figure 15. Simulation results for the López-Dahab cryptoprocessor. (a) Results from SignalTAP II (b) Results from Matlab](image)

![Figure 16. Simulation results for the halve-and-add cryptoprocessor. (a) Results from SignalTAP II (b) Results from Matlab](image)

![Figure 17. Simulation results for the Koblitz curves cryptoprocessor. (a) Results from SignalTAP II (b) Results from Matlab](image)

From Figures 15 through 17, we can see that the results obtained from Matlab are the same as the results from SignalTap II. Then, the hardware verification results verify the correct functionality of the designed cryptoprocessors.

### Synthesis results for the cryptoprocessors

The synthesis results of the cryptoprocessors over \( GF(2^{163}) \) are shown in Table 1. Additionally, some of the data presented in Table 1 are plotted in Figure 18.
mercury et al. (2006) computes $kP$ by using the half-add algorithm, $m=163$, polynomial bases representation and one parallel multiplier. Our processor requires more area than the mentioned processor because it uses two digit-level multipliers, and our design requires less time to perform the scalar multiplication. However, the first processor requires more area than our processor. Mercurio et al (2006) computes $kP$ by using the half-add algorithm, $m=163$, polynomial bases representation and one parallel multiplier. Our processor requires more area than the mentioned processor because it uses two digit-level multipliers, but our design requires less time to perform the scalar multiplication, and the latency to compute the point addition is $5\mu s$. Finally, our processor is based on the Koblitz curves and has a higher performance (area and time) than the processor presented in Azarderakhsh (2013) because our design has a latency of $5M$ to compute the point addition, and it uses two digit-level multipliers and a window method that allows us to reduce the amount of point addition operations.

From Table 2, it is possible to observe that the GF($2^{163}$) crypto-processor presented in Mahadizadeh et al (2013) requires less time to perform the scalar multiplication than our processor based on the Lopez-Dahab algorithm because the first processor uses three digit-level multipliers, and our design uses two digit-level multipliers, and the latency to compute $M_{\text{add}}$ and $M_{\text{double}}$ is $3M$. However, the first processor requires more area than our processor. Mercurio et al (2006) computes $kP$ by using the half-add algorithm, $m=163$, polynomial bases representation and one parallel multiplier. Our processor requires more area than the mentioned processor because it uses two digit-level multipliers, but our design requires less time to perform the scalar multiplication, and the latency to compute the point addition is $5\mu s$. Finally, our processor is based on the Koblitz curves and has a higher performance (area and time) than the processor presented in Azarderakhsh (2013) because our design has a latency of $5M$ to compute the point addition, and it uses two digit-level multipliers and a window method that allows us to reduce the amount of point addition operations.

From Table 2, it is possible to observe that the GF($2^{163}$) crypto-processor presented in Mahadizadeh et al (2013) requires less time to perform the scalar multiplication than our processor based on the Lopez-Dahab algorithm because the first processor uses three digit-level multipliers, and our design uses two digit-level multipliers, and the latency to compute $M_{\text{add}}$ and $M_{\text{double}}$ is $3M$. However, the first processor requires more area than our processor. Mercurio et al (2006) computes $kP$ by using the half-add algorithm, $m=163$, polynomial bases representation and one parallel multiplier. Our processor requires more area than the mentioned processor because it uses two digit-level multipliers, but our design requires less time to perform the scalar multiplication, and the latency to compute the point addition is $5\mu s$. Finally,

### Table 1. Synthesis results for the cryptoprocessors

<table>
<thead>
<tr>
<th>Cryptoprocessor</th>
<th>Area (ALUTs)</th>
<th>$F_{\text{max}}$ (MHz)</th>
<th>Registers</th>
<th>$kP$ ($\mu s$)</th>
<th>$T_{\times A}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lopez-Dahab</td>
<td>24882</td>
<td>215.3</td>
<td>2608</td>
<td>13.37</td>
<td>0.33</td>
</tr>
<tr>
<td>Halving 2-NAF</td>
<td>22670</td>
<td>158.2</td>
<td>2572</td>
<td>16.90</td>
<td>0.38</td>
</tr>
<tr>
<td>Koblitz 2-NAF</td>
<td>24223</td>
<td>226.6</td>
<td>2046</td>
<td>9.88</td>
<td>0.23</td>
</tr>
<tr>
<td>Koblitz 4-NAF</td>
<td>24257</td>
<td>226.7</td>
<td>2050</td>
<td>7.37</td>
<td>0.17</td>
</tr>
<tr>
<td>Koblitz 8-NAF</td>
<td>24249</td>
<td>211.6</td>
<td>2108</td>
<td>6.17</td>
<td>0.14</td>
</tr>
<tr>
<td>Koblitz 16-NAF</td>
<td>24270</td>
<td>177.1</td>
<td>2135</td>
<td>5.05</td>
<td>0.12</td>
</tr>
</tbody>
</table>

### Figure 18

(a) Area resources. (b) Frequency. (c) Registers resources. (d) Time to perform the scalar multiplication of each cryptoprocessor.

From Figure 18, we can see that the $w$-NAF cryptoprocessor with $w=16$ performs the scalar multiplication at a faster time (5.05 $\mu s$), and the halve-and-add processor with $w=2$ uses fewer area resources than the other processors.

### Comparison of the results with other works

To compare the performance of the designed cryptoprocessors with respect to the cryptoprocessors presented in the literature, Table 2 shows several design parameters and processing times, such as area resources, frequency, $kP$ time and time-area product. However, it is important to mention that performing a fair comparison in hardware design is very difficult because there are other technical considerations, including the technologies, hardware platforms, software tools, scalar multiplication algorithms, finite field representations, and size of the fields.

From Table 2, it is possible to observe that the GF($2^{163}$) crypto-processor presented in Mahadizadeh et al (2013) requires less time to perform the scalar multiplication than our processor based on the Lopez-Dahab algorithm because the first processor uses three digit-level multipliers, and our design uses two digit-level multipliers, and the latency to compute $M_{\text{add}}$ and $M_{\text{double}}$ is $3M$. However, the first processor requires more area than our processor. Mercurio et al (2006) computes $kP$ by using the half-add algorithm, $m=163$, polynomial bases representation and one parallel multiplier. Our processor requires more area than the mentioned processor because it uses two digit-level multipliers, but our design requires less time to perform the scalar multiplication, and the latency to compute the point addition is $5\mu s$. Finally, our processor is based on the Koblitz curves and has a higher performance (area and time) than the processor presented in Azarderakhsh (2013) because our design has a latency of $5M$ to compute the point addition, and it uses two digit-level multipliers and a window method that allows us to reduce the amount of point addition operations.

### Table 2. Performance comparison results

<table>
<thead>
<tr>
<th>Design</th>
<th>FPGA</th>
<th>Area</th>
<th>$F_{\text{max}}$ (MHz)</th>
<th>$kP$ ($\mu s$)</th>
<th>$T_{\times A}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Trujillo and Velasco, 2010)</td>
<td>Stratix III</td>
<td>18567 ALUTs</td>
<td>97.51</td>
<td>60</td>
<td>1.11</td>
</tr>
<tr>
<td>(Malik, 2010)</td>
<td>Stratix IV</td>
<td>24882 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>(Sandolal et al., 2011)</td>
<td>Stratix IV</td>
<td>24249 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>(Nabil et al., 2012)</td>
<td>Stratix IV</td>
<td>24223 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>(Mercurio et al., 2006)</td>
<td>Stratix IV</td>
<td>24257 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>(Azarderakhsh et al., 2013)</td>
<td>Stratix IV</td>
<td>24270 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>This work 16-NAF</td>
<td>Stratix IV</td>
<td>24223 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>This work 2-NAF</td>
<td>Stratix IV</td>
<td>24257 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
<tr>
<td>This work 8-NAF</td>
<td>Stratix IV</td>
<td>24270 ALUTs</td>
<td>158.2</td>
<td>158.2</td>
<td>1.49</td>
</tr>
</tbody>
</table>

From Table 2, it is possible to observe that the GF($2^{163}$) crypto-processor presented in Mahadizadeh et al (2013) requires less time to perform the scalar multiplication than our processor based on the Lopez-Dahab algorithm because the first processor uses three digit-level multipliers, and our design uses two digit-level multipliers, and the latency to compute $M_{\text{add}}$ and $M_{\text{double}}$ is $3M$. However, the first processor requires more area than our processor. Mercurio et al (2006) computes $kP$ by using the half-add algorithm, $m=163$, polynomial bases representation and one parallel multiplier. Our processor requires more area than the mentioned processor because it uses two digit-level multipliers, but our design requires less time to perform the scalar multiplication, and the latency to compute the point addition is $5\mu s$. Finally,
our processor is based on the Kobitz curves and has a higher performance (area and time) than the processor presented in Azarderakhsh (2013) because our design has a latency of SM to compute the point addition, and it uses two digit-level multipliers and a window method that allows us to reduce the amount of point addition operations.

Conclusions

This work presents the design of elliptic curve cryptoprocessors to compute the scalar multiplication over $\mathbb{G}F(2^{163})$ using the GNB. The Lopez-Dahab, halve-and-add and $w$-tNAF algorithms are used to design the cryptoprocessors, which are described using generic structural VHDL, synthesized on the Stratix IV FPGA (EP4SGX180F35C2).

Considering the hardware verification results, the 16-tNAF cryptoprocessor performs the scalar multiplication in less time (5.05 $\mu$s), and the 2-NAF halve-and-add cryptoprocessor uses fewer area resources than the other processors, in this case, 22670 ALUTs. All the cryptoprocessors use roughly 17% of the ALUTs of the FPGA.

Additionally, it is important to mention that the algorithms are synthesized on the same hardware platform using Quartus II, and are verified using SignalTap and Matlab; the cryptoprocessors use two digit-level finite field multipliers over $\mathbb{G}F(2^{163})$ in the GN; the expansion coefficients for the private key $k$ are obtained using the software Maple; and the FSMs use a data dependence graph to perform $kP$ to achieve the minimal states.

Future work will be oriented to increase the performance of the designed cryptoprocessors and the hardware implementation of the $\mathbb{G}F(2^{233})$ processors. Additionally, new cryptoprocessors will be designed based on elliptic curves that are not included in the National Institute of Standards and Technology (NIST), such as the Hessian and Edwards curves that perform the scalar multiplication $kP$.

References

Amara, M., & Siad, A. (2011). Hardware implementation of arithmetic for elliptic curve cryptosystems over $\mathbb{G}F(2^{163})$. In World Congress on Internet Security [WorldCIS] (pp. 73-78). London: IEEE.


