Abstract

In this work a processor that supports elliptic curve cryptographic applications over $GF(2^m)$ is proposed. The proposed structure is capable of calculating point multiplication and addition using only one coordinate to contain the point information, allowing for a better usage of bandwidth resources. For the point multiplication procedure, any coordinate pre-calculation is completely avoided. This design was successful prototyped on a reconfigurable device for the particular field $GF(2^{163})$. Experimental results suggest that point multiplication can be performed in 144 $\mu$s and point affine addition in 1.02 $\mu$s. Comparing with the related work, a 5 times time performance increasing can be obtained for point addition and multiplication. The presented design offers a well balanced area-time performance comparing with elliptic curve point multiplication specific processors.

1. Introduction

Elliptic Curve Cryptography (ECC) was proposed in 1985 by N. Koblitz and V. Miller [5, 8]. Since then, it has been gathering increasing importance. Other algorithms, such as RSA, have reached a higher popularity and are now widely used. However, there is a new set of circumstances which motivates the need to research a different cryptosystem, namely the increasing use of portable devices, with constrained computing and power resources, and the increasingly more valuable bandwidth and memory availability, which demands lower sized keys. In fact, RSA is becoming a less efficient solution for some applications, such as portable devices and embedded systems, namely due to its mathematical properties. Our design differs from the related state of the art in the sense that only one coordinate of an elliptic point is used to compute point multiplication, achieving higher performance than the related art, also due to the use of projective coordinates and a careful operation scheduling. This procedure avoids the transmission of both elliptic curve points coordinates, and the calculation of a coordinate from the another one before the point multiplication. In the same architecture we also implement the point affine addition starting from only one coordinate, supporting the computation of digital signature algorithms and cipher/decipher procedures.

The presented design implementation suggest that ECC point multiplication and addition can be performed in 144 $\mu$s and 1.02 $\mu$s, respectively, for the field $GF(2^{163})$, with polynomial representation and a non-supersingular elliptic curve, achieving higher performance regarding the state of art for equivalent FPGA platforms.

The following sections are organized as follows. Section 2 gives an overview of ECC properties and operations. In Section 3, the proposed arithmetic units, over $GF(2^m)$, and the overall ECC processor structure are presented. In Section 4 implementation and experimental results are presented and compared with the related state of the art. Concluding remarks on the proposed ECC design over $GF(2^m)$ are presented in Section 5.

2. Elliptic Curve Cryptography

The ECC security is related to the computational hard Elliptic Curve Discrete Logarithm Problem (ECDLP), which consists in calculating the integer scalar $k$ knowing $P (= kG)$ and $G$ elliptic curve points. The steps taken by two individuals in order to communicate a secret through an unsecured channel can be found in [14]. In this protocol, they compute exactly the same thing except for the signal of the point multiplication. To encrypt the message $M$ is computed $C = M + k_1(k_2G)$ and to decrypt is computed $M = C - k_2(k_1G)$, with $k_i$ the private keys and $k_iG$ the public keys of the two individuals. To implement this and other protocols, like digital signatures, the elliptic curve
point adding is used.

Several implementations of ECC systems have been proposed, but all of them follow an hierarchy suggested in [12] to perform the ECC arithmetic. For hardware implementations, there are two major methods to approach this hierarchy: a complete dedicated structure to generate point multiplication [12, 15], or a co-processor scheme [4]. In this later case, the lower stages of the hierarchy are computed in dedicated co-processors that provide an instruction set to a host system in order to compute the top hierarchy operations.

The previously proposed designs, which refer to $GF(2^n)$, suggest some of the following approaches:

**Basis to Represent the Field Elements**: the most used basis are the polynomial base [12], the normal base and the optimal normal base [15], which allow the use of multipliers with minimum hardware interconnections.

**Multiplier Design**: the multiplier units supported by optimal normal basis use the properties of these bases to perform pre-calculated table look-up operations [15]. Other designs use the Montgomery field multiplication algorithms [7] and Karatsuba-Offman multipliers [12].

**Inverter**: the related units use Itoh and Tsujii inverters [4]; Fermat Little Theorem inspired inverters using various multiplications [12]; and Brunner [2] inverters, based on the Extended Euclidean Algorithm.

In the following sub-section the optimizations referring to the ECC stages hierarchy are presented, namely: i) the utilization of standard projective coordinates [12] to represent elliptic curve points, ii) the compact representation method for these points and iii) the related mathematical support [15].

### 2.1. ECC Coordinate Collapsing

An algorithm that allows to obtain the $x$ coordinate of a elliptic curve point multiplied by a scalar using only the $x$ coordinate of the starting point is proposed in [6]. A version of this algorithm was also proposed using projective coordinates, where an affine coordinate $x = \frac{x}{z}$, with $X$ and $Z$ the projective coordinates (the $Y$ projective coordinate is not used). The projective algorithm has the advantage of avoiding the inversion procedure, apart from the conversion to affine coordinates in the final step. To multiply a point by a scalar $k$ the algorithm is as follows [6]:

**Inputs**: $k = (k_{n-1}k_{n-2}...k_0)$ with $k_{n-1} = 1$

**Output**: $Q := kP$

$X_1 := x, Z_1 := 1, X_2 := x^4 + b, Z_2 := x^2$

for $(i = n-2$ downto 0) do
  if $(k_i = 1)$ then
    $(X_1, Z_1) \rightarrow \text{Madd}(X_1, Z_1, X_2, Z_2)$
    $(X_2, Z_2) \rightarrow \text{Mdouble}(X_2, Z_2)$
  else
    $(X_2, Z_2) \rightarrow \text{Madd}(X_2, Z_2, X_1, Z_1)$
    $(X_1, Z_1) \rightarrow \text{Mdouble}(X_1, Z_1)$
  end
end
return $Q := \text{Mxy}(X_1, Z_1, X_2, Z_2)$

The routine $\text{Madd}$ performs a projective addition:

$Z_R = (X_1Z_2 + X_2Z_1); X_R = xZ_R + (X_1Z_2)(X_2Z_1)$; (1)

while $\text{Mdouble}$ performs a projective doubling algorithm:

$Z_R = (XZ)^2; X_R = X^4 + bZ^4$. (2)

with $X_R$ and $Z_R$ being the result coordinates. The routine $\text{Mxy}$ performs the final conversion to affine coordinates. In [6] is suggested an expression for the $y$ coordinate that is adapted for projective coordinates in [11]:

$y_R = y + (x + X_1/Z_1)(xZ_1Z_2)^{-1}$ (3)

$[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)]$.

A manipulation of (3) in order to avoid the need to calculate the $y$ coordinate to obtain the result is proposed. By considering $x_R = \frac{X_1}{Z_1}$:

$y_R = \frac{(X_1 + xZ_1)(xZ_2 + xZ_1)}{Z_1Z_2} + (x^2 + y) + \frac{xR(X_1 + xZ_1)(X_2 + xZ_1)}{xZ_1Z_2} + \frac{xR(x^2 + y)}{x} + y$ (4)

$\Leftrightarrow \frac{y_R}{x_R} = (X_1 + xZ_1)(X_2 + xZ_1) + \frac{1}{x_RZ_1Z_2} + \frac{x^2}{x_R} + x + \frac{y}{x}$

In (4) only the term $\frac{y}{x}$ depends on $y$. This means that it is possible to perform $T\left(\frac{y_R}{x_R}\right)$ directly from $T\left(\frac{y}{x}\right)$ without knowing $y$. From (4) results:

$T\left(\frac{y_R}{x_R}\right) = T\left(\frac{y}{x}\right) + \frac{xR(X_1 + xZ_1)(xZ_1Z_2 + xZ_1Z_2 + X_1Z_2)}{xX_1Z_1Z_2}$ (5)

where $T(\beta)$ means the trace of $\beta$. The trace operator will be formally introduced later.
2.2 Mathematical Background

The following introduces Galois Fields $GF(2^m)$ properties in order to support ECC operations. The field $GF(2^m)$ has been important in cryptographic applications, leading to high performance hardware implementations. These fields have a finite number of elements, so it is called a finite or a Galois field. $GF(2^m)$ is formed by the expansion of the binary set $GF(2) \equiv \{0, 1\}$ in $m$ dimensions, so any element can be represented as a $m$ bit binary vector. In our design we choose a polynomial basis representation, so each $GF(2^m)$ element $A(x)$ refers to a polynomial with $m$ binary coefficients $a_k \in GF(2)$:

$$A(x) = \sum_{k=0}^{m-1} a_k x^k. \quad (6)$$

In $GF(2^m)$ the addition operation consists of an XOR bit-wise operation between two field elements. The additive inverse of a $GF(2^m)$ element is the element itself.

For the particular case $m = 163$, the multiplication $Z(x) = A(x)B(x)$, with $A(x)$ and $B(x)$ two polynomials in $GF(2^m)$, may be obtained as:

$$Z(x) = b_0 A(x) + b_1 x A(x) + \ldots + b_{162} x^{162} A(x). \quad (7)$$

The multiplication in (7) is supported on individual multiplications by powers of the polynomial $x$. For an irreducible polynomial $P(x)$, with order $m$ [14]:

$$x^m \text{ mod } P(x) = P(x) - x^m. \quad (8)$$

$$A(x) \times x = a_{m-1} (P(x) - x^m) + \ldots + a_3 x^2 + a_0 x \quad (9)$$

(9) shows that two cases can be considered to perform a multiplication by $x$: if $a_{m-1} = 0$, the multiplication operation is equivalent to a one position left shift; if $a_{m-1} = 1$, beside the shift, we have also to sum the irreducible polynomial that supports the field. To divide by $x$ we can perform the inverse: if $a_0 = 0$ we perform a one position right shift; if $a_0 = 1$, before the shift, we subtract the irreducible polynomial without the $m$ order term. The squaring is a particular case of multiplication:

$$Z(x) = A(x)^2 = \sum_{k=0}^{m-1} \sum_{k=0}^{m-1} a_k x^k a_k x^k = \sum_{k=0}^{m-1} a_k x^{2k} \quad (10)$$

To perform efficiently (10), one shall pre-compute the dependencies of each result coefficient $z_k$ from the $a_k$ input coefficients. Observing (10), when $k < \lfloor m/2 \rfloor$ the $z_k$ coefficient depends directly from the $a_{2k}$ coefficients. For $k \ge \lfloor m/2 \rfloor$ the dependencies are calculated through the irreducible polynomial by observing the property in (8). For these values of $k$, one shall compute $x^{2k}$ using (8) and (7). The dependencies arise from the result’s coefficients which value different from zero. Since the irreducible polynomial has at most five non zero coefficients (it usually is a pentanomial or a trinomial [10]), there the few dependencies in (10) can be efficiently computed.

The adopted inversion algorithm [2] is based on the Extended Euclidean Algorithm and can it be adapted to invert a polynomial $B(x)$ or to directly perform a division $A(x)/B(x)$, differing only on an initialization step.

The trace operator is useful in some operations related with elliptic curves. The trace of a polynomial $A$ is given by [3]:

$$T(A) = \sum_{i=0}^{m-1} A^{2^i}. \quad (11)$$

The trace operator is known to assume two possible values, 0 or 1 [3]. The trace operator is linear, thus can be calculated as an inner product:

$$T(A) = \sum_{i=0}^{m-1} t_i a_i, \quad (12)$$

where $t_i$ is obtained through:

$$t_i = T(a_i), \quad (13)$$

with $a_i$ as the various elements of the basis used to construct the field $GF(2^m)$. In the case of polynomial basis $a_i = x^i$. The trace operator has the properties $T(A + B) = T(A) + T(B)$ and $T(A) = T(A^2)$ [13].

To compute the roots from second order equations is important for calculating one elliptic curve coordinate from the other one. In $GF(2^m)$, the equation $z^2 + z + A = 0$ has a solution if and only if $T(A) = 0$ and, if $z_1$ is a solution, then $z_2 = z_1 + 1$ is also a solution [3]. Furthermore, the following equations hold [3]:

$$\sum_{i \in E} A^{2^i} = T(z) + z, \quad E = \{0, 2, 4, \ldots, m - 1\}, \quad (14)$$

$$\sum_{i \in O} A^{2^i} = T(z) + z, \quad O = \{1, 3, \ldots, m - 2\}. \quad (15)$$

From (14) and (15) one can conclude that the value $T(z)$ allows to distinguish between the two solutions of second order equations.

**ECC Arithmetic:** We will work over a non-supersingular elliptic curve group $E(GF(2^m))$, which is defined as a point at infinity $O = (0, 1)$ and the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that verify the following equation:

$$y^2 + xy = x^3 + ax^2 + b, \quad a, b \in GF(2^m). \quad (16)$$

The set defined in (16) forms an additive commutative group. These groups may contain subgroups inside, particularly cyclic subgroups. A cyclic subgroup $G_s$ is the set of points that can be calculated by the recurrent use of the operation $+ \text{ over a generator } g$, as $g + g + \ldots + g + g = ng$, with
A cyclic group has an order $p$ that consists of the smallest $n$ such that $n \cdot g = O$. In practice, $p$ is the number of elements in the subgroup. For $p$ prime, it holds that for all $g_1 \in G$, exists an element $g_2 \in G$ such that $g_1 = g_2 + g_2 = 2g_2$ [13]. To define addition over $E(GF(2^m))$, let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points that obey to (16). If $x_1 \neq 0$, $x_2 \neq 0$ and $x_1 \neq x_2$, we have a point $P_3 = P_1 + P_2 = (x_3, y_3)$ that obeys to (16) with:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a; \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

$$\lambda = \frac{(y_1 + y_2)}{(x_1 + x_2)}$$

(17)

The point $O = (0, 1)$ is the additive identity of the group $E(GF(2^m))$. Hence, we can define the addition inverse of a point $P_1 = (x_1, y_1)$ as a point $P_2 = (x_2, y_2)$ such that $P_1 + P_2 = O$. The coordinates of the addition inverse can be obtained with:

$$x_2 = x_1; \quad y_2 = x_1 + y_1$$

(18)

When $x_1 = x_2$, an expression for point doubling holds, but it is not relevant for the work herein presented. From (18) it is possible to observe that the inverse of a point has the same $x$ coordinate of the point itself. Note that (16) can be rewritten as:

$$\left(\frac{y}{x}\right)^2 + \left(\frac{y}{x}\right) + a + \frac{b}{x^2} = \left(\frac{y}{x}\right)^2 + \left(\frac{y}{x}\right) + w(x) = 0$$

(19)

From (19) we conclude that knowing $x$, we can compute $w(x)$. Using (15) with the value of $w(x)$, it is possible to obtain $T \left(\frac{x}{2}\right) = \frac{y}{x}$. This means that it is possible to transmit an elliptic curve point only with the coordinate $x$ and one more bit coding $T \left(\frac{x}{2}\right)$. This last bit allows the distinction between a point and its inverse.

For all $x$ coordinates of a point that belongs to a cyclic subgroup, the condition $T(x) = T(a)$ holds [13]. Furthermore, for the vector obtained from (13), when $m$ is odd then $t_0 = 1$ [13]. With this information, we can compress the point into a single $m$ length coordinate, using the $x_0$ bit of a coordinate to code the bit $T \left(\frac{x}{2}\right)$. Then, to recover the point from the single coordinate, we read the bit $T \left(\frac{x}{2}\right)$ stored in $x_0$ and correct the coordinate $x$ with the bit $x_0$ such that the condition $T(x) = T(a)$ holds. To compute the $y$ coordinate, we compute first $y/x$ using (15) with the bit $T \left(\frac{x}{2}\right)$ and the corrected $x$ coordinate, and then we perform a final multiplication by $x$. This procedure fits most ECC protocols, which are supported over cyclic subgroups.

### 3 Proposed Field and ECC Structures

In this section a structure capable of processing elliptic curve cryptography over the finite field $GF(2^m)$ is proposed. In the presented prototype a $GF(2^m)$ finite field with $m = 163$ is used, however, the proposed design is capable of handling any odd $m$. The used field size and the irreducible polynomial $(P(x) = x^{163} + x^7 + x^6 + x^3 + 1)$ is one of the sets recommended by NIST [10]. The structure herein proposed is capable of performing elliptic curve multiplication and point addition, and has been oriented towards the Xilinx VIRTEX 4 reconfigurable technology.

An overview of our design is depicted in Figure 1.

![Figure 1. Proposed design overview.](image)

The developed structure is able to perform the point multiplication with only one coordinate, as described in Section 2.1, and the point affine addition in (17). The most important operation in ECC is the point multiplication. Thus, to improve the performance of the point multiplication, this computation was parallelized in order to perform three multiplications at once, which is achieved by using three field multipliers units.

A division unit is also present in the proposed structure, in order to perform the final computation described in (5), and the affine addition, which involves the computation of the input $y$. The root calculation unit computes (15). We choose this last equation since comparing with (14), it requires one less field addition. The trace of multiplication unit computes the trace of a multiplication without calculating the multiplication result. In the design, parameters $a$ and $b$ from the elliptic curve are used, described as in (16), and a pre-computed parameter $d$ such that $d^4 = b$. This last parameter can be computed using the Fermat Little Theorem, which states that for any field ele-
ment β, it holds that \( β = β^2 \) [2]. Thus, \( d = b^{2m} - 4 \), with 
\( 2^m = 4 = 2^1 + 2^1 + \ldots + 2^{m-1} \). In the design we have oth-
mered the construction of the input \( x \) coordinate from the co-
ordinate with the embedded trace, explained in Section 2.2, 
which consists in reading the value \( T(y/x) \) and correcting 
the coordinate regarding the property \( T(x) = T(a) \). We 
consider the scalar to have the fixed length of the used cyclic 
subgroup generator order, with the most significant bit not 
equal to 0. The second most significant bit of the register 
Scalar is designated by \( k \). The scheduling used in the de-
ign is described in Tab. 1, from the top to the bottom.

<table>
<thead>
<tr>
<th>Table 1. Architecture scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Point multiplication</strong></td>
</tr>
<tr>
<td>Point multiplication initialization</td>
</tr>
<tr>
<td>( X_0 = x; \ Z_0 = 1; \ X_1 = x^2 + b; \ Z_1 = x^2; )</td>
</tr>
<tr>
<td>Point multiplication main cycle</td>
</tr>
<tr>
<td>( Z_k = ((X_k = X_1Z_0) + (A_{11} = X_1Z_2)); \ A_{22} = (dZ_2)^2; )</td>
</tr>
<tr>
<td>Obtaining point multiplication result coordinate and trace</td>
</tr>
<tr>
<td>( A_{22} = x_{20}; \ X_1Z_0 + (A_{11} = X_1Z_2) )</td>
</tr>
<tr>
<td>( A_{22} = A_{11}Z_1; \ X_1Z_0 + (A_{11} = X_1Z_2) )</td>
</tr>
<tr>
<td>( A_{22} = A_{11}Z_1 ) (cont.); ( X_0 = A_{22}X_1^2 )</td>
</tr>
<tr>
<td>( A_{22} = A_{11}Z_1 ) (cont.);</td>
</tr>
<tr>
<td>( x_R = X_0A_{22}; \ T \left( \frac{m}{R} \right) = T(A_{11}Z_1) + T (\frac{a}{2}) )</td>
</tr>
<tr>
<td><strong>Point addition</strong></td>
</tr>
<tr>
<td>Point addition initialization</td>
</tr>
<tr>
<td>( XR0 = xR + a + \frac{b}{a} )</td>
</tr>
<tr>
<td>( S = \sum_{O} (XOR2)^2 + T \left( \frac{m}{R} \right) + decrypt )</td>
</tr>
<tr>
<td>( Z_0 = xR ) (Zo become the y coordinate of the last point multiplication result)</td>
</tr>
<tr>
<td>Point addition main procedure</td>
</tr>
<tr>
<td>( XOR2 = xR + a + \frac{b}{a} )</td>
</tr>
<tr>
<td>( A_{22} = \frac{1}{xR} ); ( S = \sum_{O} (XOR2)^2 + T \left( \frac{a}{2} \right) )</td>
</tr>
<tr>
<td>( A_{22} = \frac{1}{xR} ) (cont.); ( Z_1 = xS ) (Z1 becomes the y input coordinate)</td>
</tr>
<tr>
<td>( A_{22} = \frac{1}{xR} ) (cont.);</td>
</tr>
<tr>
<td>( A_{22} = (A_{11} = A_{11}Z_0) + (A_{11} = A_{11}Z_0) + xR + x + a )</td>
</tr>
<tr>
<td>( (A_{11} = A_{11}Z_0) + xR ) becomes the result’s x coordinate)</td>
</tr>
<tr>
<td>inv = ( A_{22}^{-1} ); ( A_{22} = A_{22}A_{22}x + xR ) + ( A_{22}Z_1 )</td>
</tr>
<tr>
<td>(Output trace) = ( T(invA_{22}) )</td>
</tr>
</tbody>
</table>

Note that when we want to transmit/receive a secret that is 
mapped in more than one elliptic curve point, the Tab. 1 
point addition initialization steps are performed only once, 
after the multiplication of the local private key with the 
other part public key, as presented in Section 2.2. There is a 
bit that controls the complementation of this last operation 
trace \( T(yR/xR) \), hence, also controls the calculus of the 
gy coordinate of a point or the y coordinate of its inverse, 
this latter used in the decryption process. The following 
presents each individual unit in detail.

**Multiplication and division by polynomial \( x \):** These 
units are referred as \( x^i \). These units’ design follows (9). 
Given that the irreducible polynomial only has 5 non zero 
bits, a multiplication by polynomial \( x \) unit can be imple-
mented with one hardwired left shift and 3 XOR gates (5-2), 
resulting in a very compact unit. To obtain a full multipli-
cation by a polynomial \( x^i \), one can serialize \( i \) multipliers 
by polynomial \( x \) circuits. To obtain a division by polynomial \( x \), 
the circuit is identical, performing the operation discussed 
in Section 2.2.

**Squaring unit:** The squaring unit \( w^2 \) is constructed by 
following the pre-calculus of each output bit dependencies 
from the input vector bits, as referred in Section 2.2. As 
explained, these dependencies arise directly from the less 
significant half of the input vector, and from the irreducible 
polynomial for the other half. The irreducible polynomial 
has only five non zero bits, which results in few dependen-
cies per bit. After analysing the dependencies, we con-
cluded that the maximum dependencies for one result bit 
is five only for two bits, and four or less for the other ones, 
leading to an efficient implementation. Hence, a squaring 
unit has approximately the same complexity as the addition 
operation. A \( w^4 \) unit is obtained with the serialization of 
two \( w^2 \) units.

**Field multiplication unit:** The field multiplication (\( multi \ i \) unit, depicted in Figure 2, is obtained from the 
parallelization of the equation (7) in four subtasks [9]: 

\[
Z_0 (x) = b_0A (x) + b_4x^4A (x) + \ldots + b_{160}x^{160}A (x); \\
Z_1 (x) = x \left[ b_1A (x) + b_5x^4A (x) + \ldots + b_{160}x^{160}A (x) \right]; \\
Z_2 (x) = x^2 \left[ b_2A (x) + b_6x^4A (x) + \ldots + b_{162}x^{160}A (x) \right]; \\
Z_3 (x) = x^3 \left[ b_3A (x) + b_7x^4A (x) + \ldots + b_{165}x^{156}A (x) \right]; \\
(20)
\]

with \( Z(x) = Z_0 (x) + Z_1 (x) + Z_2 (x) + Z_3 (x) \).

In order to achieve a good compromise between the 
available area and performance, 4 level parallelization is 
used. This structure can be further parallelized in order
to achieve a higher performance. No degradation of the critical path is expected when increasing the parallelization level [9]. In the last iteration the register $Z_3$ is not updated, which follows directly from (20).

**Field division unit:** The division unit (div) is implemented directly from the algorithm discussed in Section 2.2. The resulting structure is depicted in Figure 3.

![Figure 3. Field divider.](image)

This unit performs both division and inversion, according to the initialization of the register $U$: if there is a division $A/B$ to be performed the $U$ register loads the input $A$; in the case of an inversion, the register $U$ is set with value 1. The counter controls the number of iterations required, i.e. $2m = 326$. Note that the multiplications by polynomial $x$, which involves the $R$ and $S$ registers, are not performed $modP(x)$, with $P(x)$ the irreducible polynomial. This last operation corresponds to one bit hardware left shift operation.

**Trace of multiplication unit:** This unit calculates the trace of a multiplication from the input operands, without calculating the product itself. The trace vector calculated with (12) is sparse, only having two non zero bits for the field parameters. Thus, this unit only needs to calculate this two bits of the multiplication result, which correspond to the non zero trace vector’s index. The computation of the dependencies of those two bits is performed taking into account:

$$Z(x) = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} a_k x^k b_l x^l = \sum_{k=0}^{m-1} a_k b_l x^{k+l}$$

(21)

Particularly in this implementation, the non zero index of the trace vector are 0 and 157. Thus the dependencies for these two bits are the values $k, l$ such that $k + l = 0$, for the index 0, and $k + l = 157$ for index 157. If the trace vector is not sparse, this unit result may be alternatively calculated with a multiplier followed by a trace calculation unit.

**Root calculation unit:** This unit allows to calculate the value of $y/x$ from $w(x) = x + a + b/x^2$ and the trace $T(y/x)$ using (15). After calculating $y/x$, the value of the $y$ coordinate can be obtained by multiplying $y/x$ with the $x$ coordinate. The resulting structure is depicted in Figure 4.

In order to improve the performance of this structure, in each iteration the eighth power of the previous result is computed, which means that in each iteration the previous iteration result is added with two new values. The control signal implements a conditional addition operation with the associated term. This is useful if we implement the design for fields $GF(2^m)$ such that $m \equiv 3 \ mod \ 4$, which means that there is an odd number of terms in (15). For our implementation with $m = 163$ we have to use this conditional addition.

4 Implementation Results

In this section we present the implementation results of the proposed ECC design and compare them with the related state of the art. The proposed ECC core was successfully implemented and thoroughly tested on a Xilinx VIRTEX 4 (XC4VSX35) prototyping platform. The implemented core implements the required point multiplication and addition for $GF(2^m)$, supporting the ECC protocols based on this field. Table 2 depicts the area occupation and performance for the implementation results, after Place&Route, considering the field size $m = 163$. The area values for the individual units of the proposed ECC design include the required input and output registers. These results were obtained from a VHDL description of the design, using Synplify Premier (version 8.6.2) as the synthesis tool and Xilinx ISE (version 9.2i) as the Place&Route tool.

From Table 2 it can be observed that the unit with the critical path is the division unit. Since the complete ECC unit is constructed over these units, we would expect the complete ECC core to have an identical critical path as the division unit. However, a difference of 20.3 ns can be observed from the full core and the division unit. This suggests that the interconnection logic, routing and multiplex-
Table 2. Implementation Results for the Xilinx Virtex 4

<table>
<thead>
<tr>
<th>Unit</th>
<th>Slices</th>
<th>Freq. [MHz]</th>
<th>Clock cycles</th>
<th>Total time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>squaring (w^2)</td>
<td>295 (1%)</td>
<td>467</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>squaring (w^4)</td>
<td>352 (2%)</td>
<td>317</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>multi</td>
<td>1475 (9%)</td>
<td>248</td>
<td>43</td>
<td>173</td>
</tr>
<tr>
<td>div</td>
<td>1169 (7%)</td>
<td>147</td>
<td>327</td>
<td>2,221</td>
</tr>
<tr>
<td>root calc.</td>
<td>1066 (6%)</td>
<td>210</td>
<td>40</td>
<td>190</td>
</tr>
<tr>
<td>trace of mult.</td>
<td>458 (2%)</td>
<td>216</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Complete ECC unit</td>
<td>10488 (68%)</td>
<td>99</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(mult.)</td>
<td></td>
<td>14,303</td>
<td>144,374</td>
<td></td>
</tr>
<tr>
<td>(1st add.)</td>
<td></td>
<td>1,434</td>
<td>14,474</td>
<td></td>
</tr>
<tr>
<td>(add.)</td>
<td></td>
<td>1,024</td>
<td>10,336</td>
<td></td>
</tr>
</tbody>
</table>

In regard to the related state of the art, the comparison between designs is not straightforward, since different field sizes, different reconfigurable devices, and different design objectives are considered. It is not obvious what will be the influence of expanding or reducing a field size in each implementation. Nevertheless, in order to compare the related state of the art with the work herein proposed, we consider time and area inversely proportional towards the field size. For this reason, we introduce a compensation factor of \(163/m\) to obtain compensated area and time values, with \(m\) the compared field size. The comparison metrics were obtained using the compensated values. In Table 3 the point multiplication characteristics of the related work are presented and compared with the results obtained for the proposed structure. Even though the proposed structure has been optimized for the Virtex 4 technology, Virtex-E and Spartan 3 implementation were also realized, in order to properly compare with the related work. Many of the proposed ECC structures use Lopez-Dahab coordinates [15, 4, 7, 1], and require 10 and 4 multiplications to compute point addition point doubling, respectively. To perform simultaneously point addition and doubling our design requires 6 multiplications.

The use of NAF scalar recoding is proposed in [15], in order to reduce the required point additions in the traditional double and add algorithms. Because our design has 5 times speedup, we expect that only if this design would avoid 1 addition in every 5 additions, it would achieve the same time performance as the design herein proposed, considering identical field multiplication performance. Furthermore, this design does not implement affine point addition or projective to affine point conversion.

To improve the performance of projective to affine conversion, a mixed coordinate representation is proposed in [1]. However, this conversion procedure has far less weight in the complete ECC multiplication procedure. Jacobian coordinates applied on the Montgomery ECC multiplication are also used in [1], resulting in a better performance using the Massey-Omura field multiplier supported over an optimal normal base. Nevertheless, performance comes at the higher expense of circuit area. Moreover, no parallelization of the ECC multiplication algorithm can be exploited. Our design suggests better performance, since parallelization techniques can be used in order to obtain a more area efficient solution, hence this has a \((\text{Slice} \times \text{gain})^{-1}\) of 0.47. In our presented implementation 3 parallel point multipliers are employed.

In [4] an Itoh-Tsujii inverter is implemented using multiplications recurrently. In our design, a inversion takes approximately the same time as 8 multiplications. Since, in [4] more than 8 multiplications are needed to perform the inversion, our design can achieve higher performance. Furthermore, with a dedicated divider architecture it is possible to parallelize the division and multiplication procedures, which is not possible in the approach used in [4].

The inversion procedure used in [7] is based on the Fermat Little Theorem, allowing for a more compact design. However, the costs in performance are very high, resulting in a lower \((\text{Slice} \times \text{Time})^{-1}\) efficiency metric of 0.16.

The design proposed in [12], using two field Karatsuba-Offman multipliers, is able to achieve the computation of point multiplication about 6 times faster than the structure herein proposed. Part of this improvement is obtained from a more aggressive parallelization, which results in an area 94% higher. It is important to highlight that this design does not support the computation of point addition or projective to affine coordinates conversion. Moreover, in [12] embedded memories (BRAM) are used to perform part of the data routing. The 24 BRAMs used are not considered in the area metric.

As a concluding remark, it should be noted that none of these designs implements coordinate \(y\) collapsing, as is the case of the proposed structure. This coordinate collapse allows for a reduction by half of the required bandwidth.

5 Conclusions

In this work a complete elliptic curve processor capable of computing point multiplication and addition is pro-
posed. The proposed structure is also able of collapsing both point coordinates into a single coordinate and performs scalar multiplications without having to calculate both input/output coordinates. Since both point multiplication and point addition are implemented, it supports most elliptic curve protocols, including digital signing. This is an advantage in the sense that we have only to transmit half the data for the same information without compromising the performance. The proposed ECC processor design supports operations over field $GF(2^m)$, and was implemented for a field with $m = 163$, on a Xilinx Virtex4. It requires 10,488 slices and computes each point multiplication in 144.374 µs and each point addition in 1.024 µs. Even though the comparison with the related work is not direct because different FPGA technologies and field parameters are used, the implementation results suggest that significant performance gains can be achieved. In comparison with most of the related state of the art performance gains in the order of 5 times faster point multiplication and addition can be achieved. When compared with the related work with better results, there is a proposed structure which performs the point multiplication 6 times faster at the expense of 94% more area. However, this structure is only capable of computing point multiplication while the structure proposed in this work computes the full ECC operations. Furthermore, the design herein proposed also computes coordinate collapsing, allowing for a better usage of the available transmission bandwidth.

References


Table 3. ECC point multiplication state of the art comparison table

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Device</th>
<th>Field $GF(2^m)$</th>
<th>Slices</th>
<th>Max. Freq. [MHz]</th>
<th>Total time $\mu$s</th>
<th>area $\times 163^2$</th>
<th>speedup $(Slices \times time)^{-1}$</th>
<th>(Slices $\times$ time)$^{-1}$ improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4]</td>
<td>XC4085XLA</td>
<td>$GF(2^{218})$</td>
<td>2,634</td>
<td>32</td>
<td>1,610</td>
<td>28%</td>
<td>0.21</td>
<td>0.32</td>
</tr>
<tr>
<td>[1]</td>
<td>XCV1000</td>
<td>$GF(2^{163})$</td>
<td>29,766</td>
<td>36</td>
<td>270</td>
<td>316%</td>
<td>1.27</td>
<td>0.17</td>
</tr>
<tr>
<td>[7]</td>
<td>XCV800</td>
<td>$GF(2^{163})$</td>
<td>1,596</td>
<td>47</td>
<td>3,801</td>
<td>17%</td>
<td>0.08</td>
<td>0.16</td>
</tr>
<tr>
<td>[12]</td>
<td>XCV3200E</td>
<td>$GF(2^{163})$</td>
<td>18,314</td>
<td>9.99</td>
<td>56</td>
<td>194%</td>
<td>6.11</td>
<td>1.34</td>
</tr>
<tr>
<td>Ours</td>
<td>XCV3200E</td>
<td>$GF(2^{163})$</td>
<td>9,432</td>
<td>49</td>
<td>292</td>
<td>100%</td>
<td>0.36</td>
<td>1</td>
</tr>
<tr>
<td>[15]</td>
<td>XC3S1000</td>
<td>$GF(2^{233})$</td>
<td>n.a</td>
<td>80</td>
<td>2280</td>
<td>-</td>
<td>0.20</td>
<td>-</td>
</tr>
<tr>
<td>Ours</td>
<td>XC3S2000</td>
<td>$GF(2^{163})$</td>
<td>10,379</td>
<td>44</td>
<td>325</td>
<td>100%</td>
<td>0.30</td>
<td>1</td>
</tr>
<tr>
<td>Ours</td>
<td>XC4VSX35</td>
<td>$GF(2^{233})$</td>
<td>10,488</td>
<td>99</td>
<td>144</td>
<td>100%</td>
<td>1</td>
<td>0.66</td>
</tr>
</tbody>
</table>

1 Our Virtex-E implementation ratio 0.616 Slice/LUT was used to convert LUTs to slices.

2 The Xilinx Virtex datasheet information was used to convert logic gates to slices.


