10

Elliptic Curve Cryptography

In this chapter we discuss several algorithms and their corresponding hardware architecture for performing the scalar multiplication operation on elliptic curves defined over binary extension fields $GF(2^m)$. By applying parallel strategies at every stage of the design, we are able to obtain high speed implementations at the price of increasing the hardware resource requirements. Specifically, we study the following four different schemes for performing elliptic curve scalar multiplications,

- Scalar multiplication applied on Hessian elliptic curves.
- Montgomery Scalar Multiplication applied on Weierstrass elliptic curves.
- Scalar multiplication applied on Koblitz elliptic curves.
- Scalar multiplication using the Half-and-Add Algorithm.

10.1 Introduction

Since its proposal in 1985 by [179, 236], many mathematical evidences have consistently shown that, bit by bit, Elliptic Curve Cryptography (ECC) offers more security than any other major public key cryptosystem.

From the perspective of elliptic curve cryptosystems, the most crucial mathematical operation is the *elliptic curve scalar multiplication*, which can be informally stated as follows. Let k be a positive integer and P a point on an elliptic curve. Then we define *elliptic curve scalar mutiplication* as the operation that computes the multiple $Q = kP$, defined as the point resulting of adding $P + P + \ldots + P$, k times. Algorithm 10.1 shows one of the most basic methods used for computing a scalar multiplication, which is based on a double-and-add algorithm isomorphic to the Horner's rule. As its name suggests, the two most prominent building blocks of this method are the *point*

doubling and *point addition* primitives. It can be verified that the computational cost of Algorithm 10.1 is given as $m-1$ point doublings plus an average of $\frac{m-1}{2}$ point additions.

The security of elliptic curve cryptosystems is based on the intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP) that can be formulated as follows. Given an elliptic curve E defined over a finite field $GF(p^m)$ and two points Q and P that belong to the curve, where P has order r, find a positive scalar $k \in [1, r-1]$ such that the equation $Q = kP$ holds. Solving the discrete logarithm problem over elliptic curves is believed to be an extremely hard mathematical problem, much harder than its analogous one defined over finite fields of the same size.

Scalar multiplication is the main building block used in all the three fundamental ECC primitives: *Key Generation*, *Signature* and *Verification* schemes[1].

Although elliptic curve cryptosystems can be defined over prime fields, for hardware and reconfigurable hardware platform implementations, binary extension finite fields are preferred. This is largely due to the carry-free binary nature exhibit by this type of fields, which is a valuable characteristic for hardware systems leading to both, higher performance and lesser area consumption.

Many implementations have been reported so far [128, 334, 261, 333, 20, 311, 327, 46], and most of them utilize a six-layer hierarchical scheme such as the one depicted in Figure 10.1. As a consequence, high performance implementations of elliptic curve cryptography directly depend on the efficiency in the computation of the three underlying layers of the model.

The main idea discussed throughout this chapter is that each one of the three bottom layers shown in Figure 10.1 can be implemented using parallel strategies. Parallel architectures offer an interesting potential for obtaining a high timing performance at the price of area, implementations in [333, 20, 339, 9] have explicitly attempted a parallel strategy for computing elliptic curve scalar multiplication. Furthermore, for the first time a pipeline strategy was essayed for computing scalar multiplication on a $GF(P)$ elliptic curve in [122].

In this Chapter we present the design of a generic parallel architecture especially tailored for obtaining fast computation of the elliptic curves scalar multiplication operation. The architecture presented here exploits the inherent parallelism of two elliptic curves forms defined over GF(2^m): The Hessian form and the Weierstrass non-supersingular form. In the case of the Weierstrass form we study three different methods, namely,

- Montgomery point multiplication algorithm;
- The τ operator applied on Koblitz elliptic curves and;
- Point multiplication using halving

[1] Elliptic curve cryptosystem primitives, namely, Key generation, Digital Signature and Verification were studied in §2.5

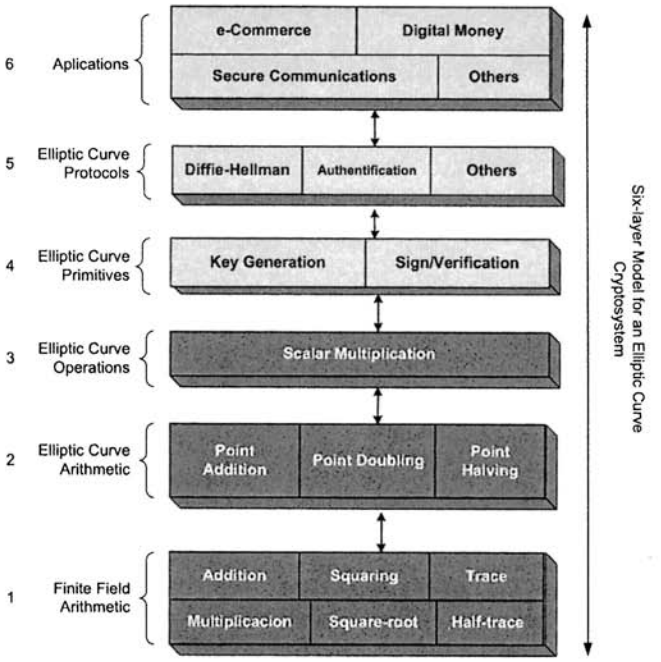

Fig. 10.1. Hierarchical Model for Elliptic Curve Cryptography

The rest of this Chapter is organized as follows. Section 10.2 briefly describe the Hessian form of an elliptic curve together with its corresponding group law. Then, in Section 10.3 we describe Weierstrass elliptic curve including a description of the Montgomery point multiplication algorithm. In Section 10.4 we present an analysis of how the ability of having more than one field multiplier unit can be exploited by designers for obtaining a high parallelism on the elliptic curve computations. Then, In Section 10.5 we describe the generic parallel architecture for elliptic curve scalar multiplication. Section 10.6 discusses some novels parallel formulations for the scalar multiplication on Koblitz curves. In Section 10.7 we give design details of a reconfigurable hardware architecture able to compute the scalar multiplication algorithm using halving. Section 10.8 includes a performance comparison of the design presented in this Chapter with other similar implementations previously reported. Finally, in Section 10.9 some concluding remarks are highlighted.

10.2 Hessian Form

Chudnvosky et al. presented in [53] a comprehensive study of formal group laws for reduced elliptic curves and Abelian varieties. In this section we discuss the Hessian form of elliptic curves and its corresponding group law followed by the Weierstrass elliptic curve form.

The original form for the law of addition on the general cubic was first developed by Cauchy and was later simplified by Sylvester-Desboves [316, 66]. Chudnovsky considered this particular elliptic curve form: "*By far the best and the prettiest*" [53]. In modern era, the Hessian form of Elliptic curves has been studied by Smart and Quisquater [335, 160].

Let $P(x)$ be a degree-m polynomial, irreducible over GF(2). Then $P(x)$ generates the finite field $\mathbb{F}_q = GF(2^m)$ of characteristic two. A Hessian elliptic curve $E(\mathbb{F}_q)$ is defined to be the set of points $(x, y, z) \in GF(2^m) \times GF(2^m)$ that satisfy the canonical homogeneous equation,

$$x^3 + y^3 + z^3 = Dxyz \tag{10.1}$$

Together with the point at infinity denoted by $\mathcal{O}$ and given by $(1, 0, -1)$.

Let $P = (x_1, y_1, z_1)$ and $Q = (x_2, y_2, z_2)$ be two points that belong to the plane cubic curve of Eq. 10.1. Then we define $-P = (y_1, x_1, z_1)$ and $P + Q = (x_3, y_3, z_3)$ where,

$$\begin{aligned} x_3 &= {y_1}^2 x_2 z_2 - {y_2}^2 x_1 z_1 \\ y_3 &= {x_1}^2 y_2 z_2 - {x_2}^2 y_1 z_1 \\ z_3 &= {z_1}^2 y_2 x_2 - {z_2}^2 y_1 x_1 \end{aligned} \tag{10.2}$$

Provided that $P \neq Q$. The addition formulae of Eq. (10.2) might be parallelized using 12 field multiplications as follows [335],

$$\begin{array}{lll} \lambda_1 = y_1 x_2 & \lambda_2 = x_1 y_2 & \lambda_3 = x_1 z_2 \\ \lambda_4 = z_1 x_2 & \lambda_5 = z_1 y_2 & \lambda_6 = z_2 y_1 \\ s_1 = \lambda_1 \lambda_6 & s_2 = \lambda_2 \lambda_3 & s_3 = \lambda_5 \lambda_4 \\ t_1 = \lambda_2 \lambda_5 & t_2 = \lambda_1 \lambda_4 & t_3 = \lambda_6 \lambda_3 \\ x_3 = s_1 - t_1 & y_3 = s_2 - t_2 & z_3 = s_3 - t_3 \end{array} \tag{10.3}$$

Whereas the formulae for point doubling are giving by

$$\begin{aligned} x_3 &= y_1 \left({z_1}^3 - {x_1}^3 \right); \\ y_3 &= x_1 \left({y_1}^3 - {z_1}^3 \right); \\ z_3 &= z_1 \left({x_1}^3 - {y_1}^3 \right). \end{aligned} \tag{10.4}$$

Where $2P = (x_3, y_3, z_3)$. The doubling formulae of Eq. (10.4) can be also parallelized requiring 6 field multiplications plus three field squarings for their computation. The resulting arrangement can be rewritten as [335],

$$\begin{array}{lll} \lambda_1 = {x_1}^2 & \lambda_2 = {y_1}^2 & \lambda_3 = {z_1}^2; \\ \lambda_4 = x_1 \lambda_1 & \lambda_5 = y_1 \lambda_2 & \lambda_6 = z_1 \lambda_3; \\ \lambda_7 = \lambda_5 - \lambda_6 & \lambda_8 = \lambda_6 - \lambda_4 & \lambda_9 = \lambda_4 - \lambda_5; \\ x_2 = y_1 \lambda_8 & y_2 = x_1 \lambda_7 & z_2 = z_1 \lambda_9; \end{array} \tag{10.5}$$

Algorithm 10.1 Doubling & Add algorithm for Scalar Multiplication: MSB-First

Require: $k = (k_{m-1}, k_{m-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(GF(2^m))$
Ensure: $Q = kP$
1: $Q = P$;
2: **for** $i = m - 2$ downto 0 **do**
3: $Q = 2 \cdot Q$; /*point doubling*/
4: **if** $k_i = 1$ **then**
5: $Q = Q + P$; /*point addition*/
6: **end if**
7: **end for**
8: **Return** Q

By implementing Eqs. (10.3) and (10.5), one can obtain the two building blocks needed for the implementation of the second layer shown in Figure 10.1. Hence, provided that those two blocks are available, one can compute the third layer of Figure 10.1 by using the well-known doubling and add Algorithm 10.1. That sequential algorithm needs an average of $\frac{m-1}{2}$ point additions plus m point doublings in order to complete one scalar multiplication computation.

Alternatively, we can use the algorithm of Figure 10.2 that can potentially be implemented in parallel since in this case the point addition and doubling operations do not show any dependencies between them. Therefore, if we assume that the algorithm of Figure 10.2 is implemented in parallel, its execution time in average will be of that of approximately $\frac{m}{2}$ point additions plus $\frac{m}{2}$ point doublings[2].

In Subsection 10.4 we discuss how to obtain an efficient parallel-sequential implementation of the second and third layers of the model of Figure 10.1.

Algorithm 10.2 Doubling & Add algorithm for Scalar Multiplication: LSB-First

Require: $k = (k_{m-1}, k_{m-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(GF(2^m))$
Ensure: $Q = kP$
1: $Q = 1$; $R = P$;
2: **for** $i = 0$ to $m - 1$ **do**
3: **if** $k_i = 1$ **then**
4: $Q = Q + R$; /*point addition*/
5: **end if**
6: $R = 2 \cdot R$; /*point doubling*/
7: **end for**
8: **Return** Q

[2] Because of the inherent parallelism of this algorithm, $\frac{m}{2}$ point doublings computations can be overlapped with the execution of about $\frac{m}{2}$ point additions.

10.3 Weierstrass Non-Singular Form

As it was already studied in Section 4.3, a Weierstrass non-supersingular elliptic curve $E(\mathbb{F}_q)$ is defined to be the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the affine equation,

$$y^2 + xy = x^3 + ax^2 + b, \tag{10.6}$$

Where a and $b \in \mathbb{F}_q, b \neq 0$, together with the point at infinity denoted by $\mathcal{O}$. The Weierstrass elliptic curve group law for affine coordinates is given as follows.

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve 10.6 then $-P = (x_1, x_1 + y_1)$. For all P on the curve $P + \mathcal{O} = \mathcal{O} + \mathcal{P} = \mathcal{P}$. If $Q \neq -P$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a & P \neq Q \\ x_1^2 + \frac{b}{x_1^2} & \mathrm{P = Q} \end{cases} \tag{10.7}$$

$$y_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})(x_1 + x_3) + x_3 + y_1 & P \neq Q \\ x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3 & \mathrm{P = Q} \end{cases} \tag{10.8}$$

From Eqns. (10.7) and (10.8) it can be seen that for both of them, point addition (when $P \neq -Q$) and point doubling (when $P = Q$), the computations for (x_3, y_3) require one field inversion and two field multiplications[3].

Notice also (a clever observation first made by Montgomery) that the x-coordinate of $2P$ does not involve the y-coordinate of P.

10.3.1 Projective Coordinates

Compared with field multiplication in affine coordinates, inversion is by far the most expensive basic arithmetic operation in $GF(2^m)$. Inversion can be avoided by means of projective coordinate representation. A point P in projective coordinates is represented using three coordinates X, Y, and Z. This representation greatly helps to reduce internal computational operations[4]. It is customary to convert the point P back from projective to affine coordinates in the final step. This is due to the fact that affine coordinate representation involves the usage of only two coordinates and therefore is more useful for external communication saving some valuable bandwidth.

In *standard* projective coordinates the projective point (X:Y:Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z$. The projective equation of the elliptic curve is given as:

$$Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \tag{10.9}$$

[3] The computational costs of field additions and squarings are usually neglected.

[4] Projective Coordinates were studied in more detail in §4.5

10.3.2 The Montgomery Method

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve of Equation 10.6. Then $P + Q = (x_3, y_3)$ and $P - Q = (x_4, y_4)$, also belong to the curve and it can be shown that x_3 is given as [128],

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2; \tag{10.10}$$

Hence we only need the x coordinates of P, Q and $P - Q$ to exactly determine the value of the x-coordinate of the point $P + Q$. Let the x coordinate of P be represented by X/Z. Then, when the point $2P = (X_2, -, Z_2)$ is converted to projective coordinate representation, it becomes [211],

$$\begin{aligned} X_2 &= X^4 + b \cdot Z^4; \\ Z_2 &= X^2 \cdot Z^2; \end{aligned} \tag{10.11}$$

The computation of Eq. 10.11 requires one general multiplication, one multiplication by the constant b, five squarings and one addition. Fig. 10.3 is the sequence of instructions needed to compute a single point doubling operation $Mdouble(X_1, Z_1)$ at a cost of two field multiplications.

Algorithm 10.3 Montgomery Point Doubling

Require: $P = (X_1, -, Z_1) \in E(GF(2^m)), c$ such that $c^2 = b$
Ensure: $P = 2 \cdot P$ /* Mdouble(X_1, Z_1)*/
1: $T = X_1^2$;
2: $M = c \cdot Z_1^2$;
3: $Z_2 = T \cdot Z_1^2$;
4: $M = M^2$;
5: $T = T^2$;
6: $X_2 = T + M$;
7: **Return** (X_2, Z_2)

In a similar way, the coordinates of $P + Q$ in projective coordinates can be computed as the fraction X_3/Z_3 and are given as:

$$\begin{aligned} Z_3 &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2; \\ X_3 &= x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1); \end{aligned} \tag{10.12}$$

The required field operations for point addition of Eq. 10.12 are three general multiplications, one multiplication by x, one squaring and two additions. This operation can be efficiently implemented as shown in Fig. 10.4.

Algorithm 10.4 Montgomery Point Addition

Require: $P = (X_1, -, Z_1), Q = (X_2, -, Z_2) \in E(GF2^m)$
Ensure: $P = P + Q$ /* Madd(X_1, Z_1, X_2, Z_2)*/
1: $M = (X_1 \cdot Z_2) + (Z_1 \cdot X_2)$;
2: $Z_3 = M^2$;
3: $N = (X_1 \cdot Z_2) \cdot (Z_1 \cdot X_2)$;
4: $M = x \cdot Z_3$;
5: $X_3 = M + N$;
6: **Return** (X_3, Z_3)

Montgomery Point Multiplication

A method based on the formulas for doubling (from Eq. 10.11) and for addition (from Eq. 10.12) is shown in Fig. 10.5 [211]. Notice that steps 2.2 and 2.3 are formulae for point doubling (*Mdouble*) and point addition (*Madd*) from Figs. 10.3 and 10.4 respectively. In fact both *Mdouble* and *Madd* operations are executed in each iteration of the algorithm. If the test bit k_i is '1', the manipulations are made for $Madd(X_1, Z_1, X_2, Z_2)$ and $Mdouble(X_2, Z_2)$ (steps 5-6) else $Madd(X_2, Z_2, X_1, Z_1)$ and $Mdouble(X_1, Z_1)$, i.e., *Mdouble* and *Madd* with reversed arguments (step 8-9).

The approximate running time of the algorithm shown in Fig. 10.5 is $6m$M + ($1I$ + 10M) where M represents a field multiplication operation, m stands for the number of bits and I corresponds to inversion. It is to be noted that the factor ($1I$ + 10M) represents time needed to convert from standard projective to affine coordinates. In the next Subsection we explain the conversion from SP to affine coordinates and then in Subsection 10.4, we discuss how to obtain an efficient parallel implementation of the above algorithm.

Conversion from Standard Projective (SP) to Affine Coordinates

Both, point addition and point doubling algorithms are presented in standard projective coordinates. A conversion process is therefore needed from SP to affine coordinates. Referring to the algorithm of Fig. 10.5, the corresponding affine x-coordinate is obtained in step 3:

$$x_3 = X_1/Z_1.$$

Whereas the affine representation for the y-coordinate is computed by step 4:

$$y_3 = (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y.$$

Notice also that both expressions for x_3 and y_3 in affine coordinates include one inversion operation. Although this conversion procedure must be performed only once in the final step, still it would be useful to minimize the number of inversion operations as much as possible. Fortunately it is possible to reduce one inversion operation by using the common operations from

Algorithm 10.5 Montgomery Point Multiplication

Require: $k = (k_{n-1}, k_{n-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(GF2^m)$
Ensure: $Q = kP$
1: $X_1 = x$;, $Z_1 = 1$;
2: $X_2 = x^4 + b$;, $Z_2 = x^2$;
3: **for** $i = n - 2$ downto 0 **do**
4: **if** $k_i = 1$ **then**
5: $Madd(X_1, Z_1, X_2, Z_2)$;
6: $Mdouble(X_2, Z_2)$;
7: **else**
8: $Madd(X_2, Z_2, X_1, Z_1)$;
9: $Mdouble(X_1, Z_1)$;
10: **end if**
11: **end for**
12: $x_3 = X_1/Z_1$;
13: $y_3 = (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$;
14: **Return** (x_3, y_3)

the conversion formulae for both x and y-coordinates. A possible sequence of the instructions from SP to affine coordinates is given by the algorithm in Fig. 10.6.

Algorithm 10.6 Standard Projective to Affine Coordinates

Require: $P = (X_1, Z_1)$, $Q = (X_2, Z_2)$, $P(x, y) \in E(GF2^m)$
Ensure: (x_3, y_3) /* affine coordinates */
1: $\lambda_1 = Z_1 \times Z_2$;
2: $\lambda_2 = Z_1 \times x$;
3: $\lambda_3 = \lambda_2 + X_1$;
4: $\lambda_4 = Z_2 \times x$;
5: $\lambda_5 = \lambda_4 + X_1$;
6: $\lambda_6 = \lambda_4 + X_2$;
7: $\lambda_7 = \lambda_3 \times \lambda_6$;
8: $\lambda_8 = x^2 + y$;
9: $\lambda_9 = \lambda_1 \times \lambda_8$;
10: $\lambda_{10} = \lambda_7 + \lambda_9$;
11: $\lambda_{11} = x \times \lambda_1$;
12: $\lambda_{12} = inverse(\lambda_{11})$;
13: $\lambda_{13} = \lambda_{12} \times \lambda_{10}$;
14: $x_3 = \lambda_{14} = \lambda_5 \times \lambda_{12}$;
15: $\lambda_{15} = \lambda_{14} + x$;
16: $\lambda_{16} = \lambda_{15} \times \lambda_{13}$;
17: $y_3 = \lambda_{16} + y$;
18: **Return** (x_3, y_3)

The coordinate conversion process makes use of 10 multiplications and only 1 inversion ignoring addition and squaring operations.

The algorithm in Fig. 10.6 includes one inversion operation which can be performed using Extended Euclidean Algorithm or Fermat's Little Theorem (FLT)[5].

10.4 Parallel Strategies for Scalar Point Multiplication

As it was mentioned in the introduction Section, parallel implementations of the three underlying layers depicted in Figure 10.1 constitutes the main interest of this Chapter. Several parallel techniques for performing field arithmetic, i.e. the first Layer of the model, were discussed in Chapter 5. However, hardware resource limitations restrict us from attempting a fully parallel implementation of second and third layers. Thus, a compromising strategy must be adopted to exploit parallelism at second and third layers.

Let us suppose that our hardware resources allow us to accommodate up to two field multiplier blocks. Under this scenario, the Hessian form point addition primitive $(x_3 : y_3 : z_3) = (x_1 : y_1 : z_1) + (x_2 : y_2 : z_2)$ studied in Section 10.2 can be accomplished in just six clock cycles as[6],

$$
\begin{array}{lll}
\textbf{Cycle 1}: & \lambda_1 = y_1 \cdot x_2; & \lambda_2 = x_1 \cdot y_2; \\
\textbf{Cycle 2}: & \lambda_3 = x_1 \cdot z_2; & \lambda_4 = z_1 \cdot x_2; \\
\textbf{Cycle 3}: & \lambda_5 = z_1 \cdot y_2; & \lambda_6 = z_2 \cdot y_1; \\
\textbf{Cycle 4}: & s_1 = \lambda_1 \cdot \lambda_6; & s_2 = \lambda_2 \cdot \lambda_3; \\
\textbf{Cycle 5}: & s_3 = \lambda_5 \cdot \lambda_4; & t_1 = \lambda_2 \cdot \lambda_5; \\
\textbf{Cycle 6}: & t_2 = \lambda_1 \cdot \lambda_4; & t_3 = \lambda_6 \cdot \lambda_3; \\
\textbf{Cycle 6.a}: & x_3 = s_1 - t_1; & y_3 = s_2 - t_2;\ z_3 = s_3 - t_3;
\end{array}
$$

Similarly, the Hessian point doubling primitive, namely, $2(x_1 : y_1 : z_1) = (x_2 : y_2 : z_2)$ can be performed in just 3 cycles as[7],

$$
\begin{array}{lll}
\textbf{Cycle 1}: & \lambda_1 = {x_1}^2;\ \lambda_2 = {y_1}^2; & \lambda_3 = {z_1}^2; \\
\textbf{Cycle 1.a}: & \lambda_4 = x_1 \cdot \lambda_1; & \lambda_5 = y_1 \cdot \lambda_2; \\
\textbf{Cycle 2}: & \lambda_6 = z_1 \cdot \lambda_3; & z_2 = z_1 \cdot (\lambda_4 - \lambda_5); \\
\textbf{Cycle 2.a}: & \lambda_7 = \lambda_5 - \lambda_6; & \lambda_8 = \lambda_6 - \lambda_4; \\
\textbf{Cycle 3}: & x_2 = y_1 \cdot \lambda_8; & y_2 = x_1 \cdot \lambda_7;
\end{array}
$$

The same analysis can be carried out for the Montgomery point multiplication primitives. The Montgomery point doubling primitive $2(X_1 : - : Z_1) =$

[5] Efficient multiplicative inverse algorithms were studied in §6.3.

[6] Because of their simplicity, the arithmetic operations of Cycle 6.a can be computed during the execution of Cycle 6.

[7] Due to the simplicity of the arithmetic operations included in cycles 1 and 2.a above, those operations can be merged with the operations performed in cycles 1.a and 2, respectively.

$(X_2 : - : Z_2)$ when using two multiplier blocks can be accomplished in just one clock cycle as,

$$\begin{array}{ll} \textbf{Cycle 1}: & T = X_1^2; \quad M = c \cdot Z_1^2; \; Z_2 = T \cdot Z_1^2; \\ \textbf{Cycle 1.a}: & X_2 = T^2 + M^2; \end{array} \tag{10.13}$$

Whereas, the Montgomery point addition primitive $(X_1 : - : Z_1) = (X_1 : - : Z_1) + (X_2 : - : Z_2)$ when using two multiplier blocks can be accomplished in just two clock cycles as,

$$\begin{array}{ll} \textbf{Cycle 1}: & t_1 = (X_1 \cdot Z_2); \; t_2 = (Z_1 \cdot X_2); \\ \textbf{Cycle 1.a}: & M = t_1 + t_2; \quad Z_1 = M^2; \\ \textbf{Cycle 2}: & N = t_1 \cdot t_2; \quad M = x \cdot Z_1; \\ \textbf{Cycle 2.a}: & X_1 = M + N; \end{array} \tag{10.14}$$

If two multiplier blocks are available, we can choose whether we want to parallelize the second or the third Layer of the model shown in Fig.10.1.

Algorithm 10.5, i.e. the third Layer of Fig. 10.1, can be executed in parallel by assigning one of our two multiplier blocks to compute the Montgomery point addition of Algorithm 10.4, and the other to perform the Montgomery point doubling of Algorithm 10.3. Then, the corresponding computational cost of point addition and point doubling primitives become of four and two field multiplications, respectively. In exchange, steps 5-6 and 8-9 of Algorithm 10.5 can be performed in parallel. Since those steps can be performed concurrently their associated execution time reduces to about 4 field multiplications. Therefore, the execution time associated to Algorithm 10.5 would be equivalent to $4m$ field multiplications[8].

Alternatively, the second layer can be executed in parallel by using our two multiplier blocks for computing point addition and point doubling in just 2 and 1 cycles, as it was shown in Eqs.(10.14) and (10.13), respectively. However, this decision will force us to implement Algorithm 10.5 (corresponding to the third layer of Fig.10.1) in a sequential manner. Therefore, the execution time associated to Algorithm 10.5 would be equivalent to $3m$ field multiplications.

If our hardware resources allow us to implement up to four field multiplier blocks, then we can execute both, the second and third Layers of Fig.10.1 in parallel. In that case the execution time of Algorithm10.5 reduces to just $2m$ field multiplications.

It is noticed that this high parallelism achieved by the Montgomery point multiplication method cannot be achieved by the Hessian form of the Elliptic curve.

Table 10.1 presents four of the many options that we can follow in order to parallelize the computation of scalar point multiplication. The computational costs shown in Table 10.1 are normalized with respect to the required number

[8] Since we can execute concurrently the procedures *Mdouble* and *Madd* the execution time of the former is completely overlapped by the latter.

Table 10.1. $GF(2^m)$ Elliptic Curve Point Multiplication Computational Costs

Strategy		Req. No.	EC Operation Cost		Equivalent	EC Operation Cost		Equivalent
2nd	3rd	of Field	Hessian Form		Time	Montgomery Algorithm		Time
Layer	Layer	Mults.	Doubling	Addition	Costs	Doubling	Addition	Costs
Sequential	Sequential	1	$6M$	$12M$	$12mM$	$2M$	$4M$	$6mM$
Sequential	Parallel	2	$6M$	$12M$	$9mM$	$2M$	$4M$	$4mM$
Parallel	Sequential	2	$3M$	$6M$	$6mM$	$1M$	$2M$	$3mM$
Parallel	Parallel	4	$3M$	$6M$	$\frac{3}{2}mM$	M	$2M$	$2mM$

of field multiplication operations (since the computation time of squaring operations is usually neglected in arithmetic over GF(2^m)).

Notice that the computation times of the Hessian form has been estimated assuming that the scalar multiplication has been accomplished by executing Algorithm 10.2. For instance, the execution time of the Hessian form in the fourth row of Table 10.1 has been estimated as follows,

$$\text{Time Cost} = \frac{m}{2}PD + \frac{m}{2}PA = \frac{3m}{2}M + \frac{6m}{2}M = \frac{9m}{2}M.$$

Due to area restrictions we can afford to accommodate up to two fully parallel field multipliers in our design. Thus, we can afford both, second and third options of Table 10.1. However, third option is definitely more attractive as it demonstrates better timing performance at the same area cost. Therefore, and as it is indicated in the third row of Table 10.1, the estimated computational cost of our elliptic curve Point multiplication implementation will be of $6m$ field multiplications in Hessian form. It costs only $3m$ field multiplications using the Montgomery algorithm for the Weierstrass form.

In the next Section we discuss how this approach can be carried out on hardware platforms.

10.5 Implementing scalar multiplication on Reconfigurable Hardware

Figure 10.2 shows a generic structure for the implementation of elliptic curve scalar multiplication on hardware platforms. That structure is able to implement the parallel-sequential approach listed in the third row of Table 10.1, assuming the availability of two GF(2^m) multiplier blocks.

In the rest of this Section, it is presupposed that two fully-parallel GF(2^{191}) Karatsuba-Ofman field multipliers can be accommodated on the target FPGA device.

The architecture in Figure 10.2 is comprised of four classes of blocks: field multipliers, Combinational logic blocks and/or finite field arithmetic (i.e. squaring, etc.), Blocks for intermediate results storage and selection (i.e. registers, multiplexers, etc.), and a Control unit (CU).

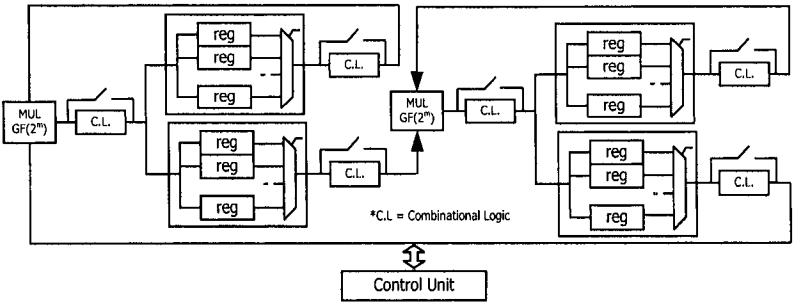

Fig. 10.2. Basic Organization of Elliptic Curve Scalar Implementation

A Control Unit is present in virtually every hardware design. Its main responsibility is to control the dataflow among the different design's modules. Design's main architecture, on the other hand, is responsible of computing all required arithmetic/logic operations. It is frequently called Arithmetic-Logic Unit (ALU).

10.5.1 Arithmetic-Logic Unit for Scalar Multiplication

Figure 10.3 shows the arithmetic-logic unit designed for computing the scalar multiplication algorithms discussed in the preceding Sections. It is a generic FPGA architecture based on the parallel-sequential approach for kP computations discussed before.

In order to implement the memory blocks of Figure 10.2, fast access FPGA's read/write memories BlockRAMs (BRAMs) were used. As it was studied in Chapter 3, a dual port BRAM can be configured as a two single port BRAMs with independent data access. This special feature allows us to save a considerable number of multiplexer operations as the required data is independently accessible from any of the two available input ports. Hence, two similar BRAMs blocks (each one composed by 12 BRAMs) provide four operands to the two multiplier blocks simultaneously. Since each BRAM contains 4k memory cells, two BRAM blocks are sufficient. The combination of 12 BRAMs provides access to a 191-bit bus length. All control signals (read/write, address signals to the BRAMs and multiplexer enable signals) are generated by the control unit (CU). A master clock is directly fed to the BRAM block which is afterwards divided by two, serving as a master clock for the rest of the circuitry. The external multiplexers apply pre and post computations (squaring, XOR, etc.) on the inputs of the multipliers whenever they are required.

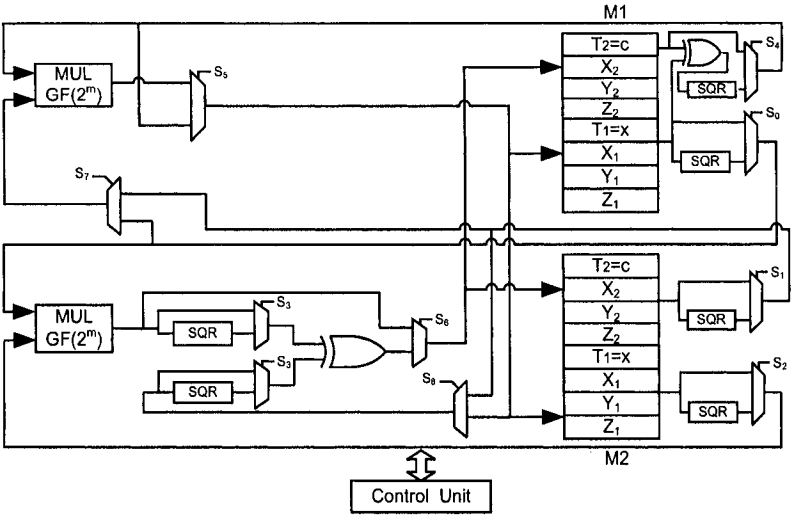

Fig. 10.3. Arithmetic-Logic Unit for Scalar Multiplication on FPGA Platforms

Let us recall that we need to perform an inversion operation in order to convert from standard projective coordinates to affine coordinates [9]. A squarer block "SqrInv" is especially included for the sole purpose of performing that inversion. As it was explained in Section 6.3.2, the Itoh-Tsujii multiplicative inverse algorithm requires the computation of m field squarings. This can be accomplished by cascading several squarer blocks so that several squaring operations can be executed within a single clock cycle (See Fig. 6.11 for more details).

In the next Subsection we discuss how the arithmetic logic unit of Figure 10.2 can be utilized for computing a Hessian scalar multiplication.

10.5.2 Scalar multiplication in Hessian Form

According to Eq. (10.3) of Section 10.2 we know that the addition of two points in Hessian form consists of 12 multiplications, 3 squarings and 3 addition operations. Implementing squaring over GF(2^m) is simple, so we can neglect it. Using the parallel architecture proposed in Figure 10.3, point addition can be performed in 6 clock cycles using two GF(2^{191}) multiplier blocks. The Hessian curve point addition sequence using two multiplier units is specified in Eq. (10.13). Table 10.2 shows that sequence in terms of read/write cycles.

[9] This conversion is required when executing a Montgomery point multiplication in Standard Projective coordinates

Referring to the architecture of Figure 10.3, $M1$ and $M2$ are two memory (BRAMs) blocks, each one composed of two independent ports $PT1$ and $PT2$.

It is noticed that the inputs/outputs of the multipliers are different from those read/write values at the memory blocks. This is due to pre or post computations required during the next clock cycle. Table 10.2 lists computed values during/after multiplications for both, the read and write cycles.

Table 10.2. Point addition in Hessian Form

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	Y_1	X_1	X_2	Y_2	λ_1	λ_2
2	X_1	Z_1	Z_2	X_2	λ_3	λ_4
3	Z_1	Z_2	Y_2	Y_1	λ_5	λ_6
4	λ_1	λ_2	λ_6	λ_5	x_3	–
5	λ_2	λ_1	λ_3	λ_4	y_3	–
6	λ_5	λ_6	λ_4	λ_3	z_3	–

Similarly, Hessian point doubling implementation of Eq. (10.13) consists of 6 multiplications, 3 squarings and 3 additions. Table 10.3 describes the algorithm flow implemented using the same architecture (Figure 10.3).

Table 10.3. Point doubling in Hessian Form

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_1	Y_1	X_1	Y_1	λ_4	λ_9
2	λ_9	λ_4	Z_1	Z_1	z_2	λ_8
3	$\lambda 8$	λ_9	Y_1	X_1	x_2	y_2

Let m represents the number of bits and M denotes a single finite field multiplication. Then the number of multiplications for one point addition and point doubling are $6M$ and $3M$, respectively. Referring to the algorithm in Figure 10.1, average of $(\frac{m}{2})6M$ and $3mM$ multiplications are needed for computing all m bits of the vector k. Thus, 6mM are the total multiplication operations required for computing kP scalar multiplication.

In the case of $m = 191$ bits, the total number of field multiplications required by the algorithm are 1146. Let T be the minimum clock period allowed by the synthesis tool. Then, $1146 \times T$ is the total time required for completing one Hessian elliptic curve scalar multiplication.

10.5.3 Montgomery Point Multiplication

Let us consider now Algorithm 10.5, where each bit of the scalar k are scanned from left to right (i.e., MSB-First).

At every iteration (regardless if the bit scanned is zero or one), both point addition ($Madd$) and point doubling ($Mdouble$) operations must be performed. However, notice that the order of the arguments is reversed: if the tested bit is '1', $Mdouble(X_2, Z_2)$, $Madd(X_1, Z_1, X_2, Z_2)$ are computed and $Mdouble(X_1, Z_1)$, $Madd(X_2, Z_2, X_1, Z_1)$ otherwise. Algorithms 10.4 and 10.3 describe the sequence of instructions for $Madd$ and $Mdouble$ operations, respectively, whereas Eqs. (10.14) and (10.13) specify how those primitives can be accomplished in 2 and 1 cycles, respectively[10].

Tables 10.4 and 10.5 describe the multiplications performed for both point addition and point doubling operations in three normal clock cycles when the scanned bit is '1' or '0' respectively. We kept the same notations used in algorithms 10.4 and 10.3 for point addition and point doubling, respectively. M1 and M2 represent two memory blocks (BRAMs) each one with two independent ports $PT1$ and $PT2$. Some required arithmetic operations (squaring etc.) need to be performed during read/write cycles at the memories before and after the multiplication operations.

Table 10.4. kP Computation, if Test-Bit is '1'

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_1	Z_2	Z_1	X_2	P	Q
2	X_2	Z_2	Z_2	T_1	Z_2=Z_3	X_2=X_3
3	P	Q	Q	T_2	X_1=X'	Z_1=Z'

The resulting vectors X_1,Z_1,X_2,Z_2, are updated at the memories after the completion of point addition and doubling operations using 3 clock cycles per each bit. Therefore, the total time for the whole 191-bit scalar is $191 \times 3 \times T$, where T represents design's maximum allowed frequency.

10.5.4 Implementation Summary

All finite field arithmetic blocks and then the kP computational architecture were implemented on a VirtexE XCV3200e-8bg560 device by using Xilinx Foundation Tool F4.1i for design entry, synthesis, testing, implementation and verification of results. Table 10.6 lists timing performances and occupied resources by the said architectures.

[10] Provided that two multiplier units are available.

Table 10.5. kP Computation, If Test-Bit is '0'

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_2	Z_1	Z_2	X_1	P	Q
2	X_1	Z_1	Z_1	T_1	$Z_1{=}Z_3$	$X_1{=}X_3$
3	P	Q	Q	T_2	$X_2{=}X'$	$Z_2{=}Z'$

Elliptic curve point addition and point doubling do not participate directly as a single computational unit in this design; however parallel computations for both point addition and point doubling are designed together as it was shown in Algorithm 10.1.

Both point addition and point doubling occupy 18300 (56.39 %) CLB slices and it takes $100.1\eta s$ (at a clock speed of 9.99 MHz) to complete one execution cycle. As it was mentioned in Section 10.2, when using two field multiplier units, six and three clock cycles are needed for computing point addition and point doubling in Hessian form, respectively.

The total consumed time for computing each iteration of the algorithm of Figure 10.1 is 900.9η if the corresponding bit is one and $300.3\eta s$ otherwise. Therefore, scalar point multiplication in Hessian form is the time needed to complete m/2 point additions (in average) and m point doublings. For our case m=191, the total time is therefore $(191/2) \cdot (600.6\eta) + 191 \cdot (300.3\eta) = 114.71\mu s$[11].

Similarly, two and one clock cycles are needed to perform Montgomery point addition and point doubling, respectively. The associated executing time is thus, $200.1\eta s$ and $100.2\eta s$ for point addition and point doubling respectively. Each iteration of the algorithm thus consumes $300.3\eta s$ for 3 clock cycles. In the case of $m = 191$, the total time needed for computing a scalar multiplication is $191(300.3) = 57\mu s$.

Inversion is performed at the end of the main loop of Algorithm 10.5. It takes 28 clock cycles to perform one inversion in $GF(2^{191})$ occupying 1312 CLB slices. The CLB slices for inversion in fact are the FPGA resources occupied for squaring operations only and the multiplier blocks are the same used for point addition and point doubling. The total conversion time (See Algorithm 10.6) is therefore $28 \cdot 100.1\eta + 10 \cdot 100.1\eta = 3.8\mu s$. Therefore, the execution time for algorithm 10.5 is given as the sum of the time for computing the scalar multiplication and the time to perform coordinate conversion namely,

$$57.36 + 3.8 = 61.16\mu s.$$

[11] It is noted that we did not include a conversion from projective to affine coordinates in the case of the Hessian form.

The architecture for elliptic curve scalar multiplication in both cases (Hessian form & Montgomery point multiplication) occupies 19626 (60 %) CLB slices, 24 (11%) BRAMs and performs at the rate of $100.1\eta s$ (9.99 MHz). The design for GF(2^{191}) Karatsuba-Ofman Multiplier occupies 8721 (26.87%) CLB slices, where one field multiplication is performed in $43.1\eta s$. Table 10.6 summarizes the design statistics.

Table 10.6. Design Implementation Summary

Design	Device (XCV)	CLB slices	Timings
Inversion in $GF(2^{191})$	3200E	1312	$2.8\eta s$
Binary Karatsuba Multiplier	3200E	8721	$43.1\eta s$
1 Field Multiplication			$100.1\eta s$
Point addition + Point doubling in Hessian Form	3200E	18300	$300.3\eta s$ (if bit = '0') $900.9\eta s$ (if bit = '1')
Point Multiplication in Hessian form	3200E	19626 & 24 BRAMs	$114.71\mu s$
Point addition + Point doubling (Montgomery Point Multiplication)	3200E	18300	$300.3\eta s$ (3 Multiplications)
Point Multiplication (Montgomery Point Multiplication)	3200E	19626 & 24 BRAMs	$61.16\mu s$

10.6 Koblitz Curves

First proposed in 1991 by N. Koblitz [180], *Koblitz Elliptic Curves* have been object of analysis and study since then, due to their superb usage of endomorphism via the Frobenius map for increasing the elliptic curve arithmetic computational performance [180, 133]. Across the years, several efforts for speeding up elliptic curve scalar multiplication on Koblitz curves have been reported both, in hardware and software platforms [13, 384, 216, 133, 132, 339, 340].

Let $P(x)$ be a degree-m polynomial, irreducible over $GF(2)$. Then $P(x)$ generates the finite field $\mathbb{F}_q = GF(2^m)$ of characteristic two. A Koblitz elliptic curve $E_a(\mathbb{F}_q)$, also known as Anomalous Binary Curve (ABC) [180], is defined as the set of points $(x, y) \in GF(2^m) \times GF(2^m)$, that satisfy the Koblitz equation,

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \tag{10.15}$$

together with the point at infinity denoted by $\mathcal{O}$. It is customary to use the notation E_a where $a \in \{0, 1\}$. It is known that E_a forms an *addition Abelian group* with respect to the elliptic point addition operation[12].

[12] Notice that since Eq. (10.15) assumes $a \in \{0, 1\}$, then Koblitz curves are also defined over $GF(2)$.

So far, most works have strived for reducing the cost associated to the double-and-add method by following two main strategies: Reducing the computational complexity of both, point addition and point doubling primitives and; reducing the number of times that the point addition primitive is invoked during the algorithm execution. Recently, the idea of representing the scalar k in mixed base rather than the traditional binary form has been proposed. This way, point doublings can be partially substituted with advantage by tripling, quadrupling and even halving a point [171, 69, 12, 13, 385, 176].

In this Section we discuss yet another approach for speeding up the computational cost of scalar multiplication on Koblitz curves: the usage of parallel strategies. In concrete, we show that the usage of the τ^{-1} Frobenius operator can be successfully applied in the domain of Koblitz elliptic curves giving an extra flexibility and potential speedup to known elliptic curve scalar multiplication procedures.

The rest of this Section is organized as follows. In Subsection 10.6.1 some relevant mathematical concepts are briefly outlined. Then, in Subsection 10.6.2 several parallel formulations of the scalar multiplication on Koblitz curves are presented. Subsection 10.6.3 discusses relevant implementation aspects of the proposed parallel algorithms for hardware platforms.

10.6.1 The τ and τ^{-1} Frobenius Operators

In a field of characteristic two, the map between an element x and its square x^2 is called the Frobenius map. It can be defined on elliptic points as:

$$\tau(x, y) := (x^2, y^2).$$

Similarly, we can define the τ^{-1} Frobenius operator as,

$$\tau^{-1}(x, y) := (\sqrt{x}, \sqrt{y}).$$

In binary extension fields, the Lagrange theorem[13] dictates that $A^{2^m} = A$ for any arbitrary element $A \in GF(2^m)$, which in turn implies that for any $i \in \mathbb{Z}$, $A^{2^i} = A^{2^{i \bmod m}}$. Notice also that by applying the square root operator in both sides of Fermat little theorem identity, we obtain, $\sqrt{A} = A^{2^{-1}} = A^{2^{m-1}}$, which can be generalized as, $A^{2^{-i}} = A^{2^{m-i}}$ for $i = 0, 1, \ldots, m$.

Using above identities, it is easy to show that the Frobenius operator satisfies the properties enumerated in the next theorem.

Theorem 10.6.1 *The Frobenius operator satisfies the following properties,*

1. $\tau\tau^{-1} = \tau^{-1}\tau = 1$
2. $\tau^i = \tau^{i \bmod m}$, *for* $i \in \mathbb{Z}$
3. $\tau^{-i} = \tau^{m-i}$, *for* $i = 1, 2, \cdots, m-1$
4. $\tau^i = \tau^{-(m-i)}$, *for* $i = 1, 2, \cdots, m-1$

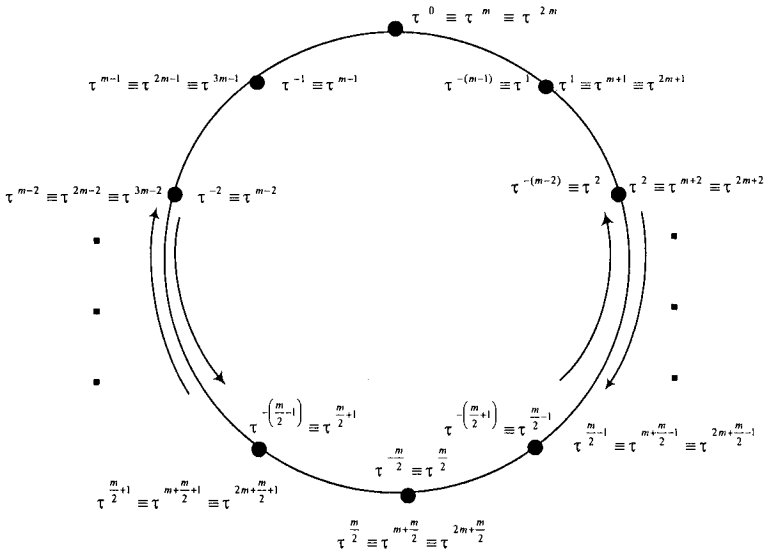

Fig. 10.4. An illustration of the τ and τ^{-1} Abelian Groups (with m an Even Number)

In other words, the τ and the τ^{-1} operators generate an *Abelian group* of order m as is depicted in Fig. 10.4. Considering an arbitrary element $A \in GF(2^m)$, with m even, Fig. 10.4 illustrates, in the clockwise direction, all the m elliptic curve points that can be generated by repeatedly computing the τ operator, i.e., $\tau^i P$ for $i = 0, 1, \cdots, m-1$. On the other hand, in the counter-clockwise direction, Fig. 10.4 illustrates all the m points that can be generated by repeatedly computing the τ^{-1} operator, i.e., $\tau^{-i} P$ for $i = 0, 1, \cdots, m-1$.

Frobenius Operator Applied on Koblitz Curves

Koblitz curves exhibit the property that, if $P = (x, y)$ is a point in E_a then so is the point (x^2, y^2) [338]. Moreover, it has been shown that, $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ for every (x, y) on E_a, where $\mu = (-1)^{1-a}$. Therefore, using the Frobenius notation, we can write the relation,

$$\tau(\tau P) + 2P = (\tau^2 + 2)P = \mu\tau P. \tag{10.16}$$

Notice that last equation implies that a point doubling can be computed by applying twice the τ Frobenius operator to the point P followed by a point

[13] Lagrange theorem can be used to prove the Fermat's little theorem and its generalization Euler's theorem studied in Chapter 4

addition of the points $\mu\tau P$ and $\tau^2 P$. Let us recall that the Frobenius operator is an inexpensive operation since field squaring is a linear operation in binary extension fields.

By solving the quadratic Eq. 10.16 for τ, we can find an equivalence between a squaring map and the scalar multiplication with the complex number $\tau = \frac{-1+\sqrt{-7}}{2}$. It can be shown that any positive integer k can be reduced modulo $\tau^m - 1$. Hence, a τ-adic non-adjacent form (τNAF) of the scalar k can be produced as,

$$k = \sum_{i=0}^{l-1} u_i \tau^I,$$

where each $u_i \in \{0, \pm 1\}$ and l is the expansion's length. The scalar multiplication kP can then be computed with an equivalent non-adjacent form (NAF) addition-subtraction method.

Standard (NAF) addition-subtraction method computes a scalar multiplication in about m doubles and $m/3$ additions [129]. Likewise, the τNAF method implies the computation of l τ mappings (field squarings) and $l/3$ additions.

On the other hand, it is possible to process ω digits of the scalar k at a time. Let $\omega \geq 2$ be a positive integer. Let us define $\alpha_i = i \bmod \tau^\omega$ for $i \in [1, 3, 5, \ldots, 2^{\omega-1} - 1]$. A width-$\omega$ τNAF of a nonzero element k is an expression $k = \sum_{i=0}^{l-1} u_i \tau^i$ where each $u_i \in [0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{\omega-1}-1}]$ and $u_{l-1} \neq 0$. It is also guaranteed that at most one of any consecutive ω coefficients is nonzero. Therefore, the $\omega\tau$NAF expansion of k represents an equivalence relation between the scalar multiplication kP and the expression,

$$u_0 P + \tau u_1 P + \tau^2 u_2 P + \ldots + \tau^{l-1} u_{l-1} P \tag{10.17}$$

In [338, 337, 26] it was proved that for a Koblitz elliptic curve $E_a[GF(2^m)]$, the length l of a τNAF expansion, is always less or equal than $m + a + 3$,

$$\ell_{NAF} \leq m + a + 3$$

Using the properties enounced in Theorem 10.6.1, Equation (10.17) can be *reduced* even further whenever $l \geq m$.

Indeed, given the fact that $\tau^{m+i} = \tau^i$ for $i = 0, 1, \cdots, m-1$, we can reduce all the expansion coefficients u_i greater than m as follows,

$$k = \sum_{i=0}^{m+a+2} u_i \tau^i = \sum_{i=0}^{m-1} u_i \tau^i + \sum_{i=m}^{m+a+2} u_i \tau^i = \sum_{i=0}^{a+2} (u_i + u_{m+i})\, \tau^i + \sum_{i=a+3}^{m-1} u_i \tau^i \tag{10.18}$$

Furthermore, using property 4 of Theorem 10.6.1, it is always possible to express a length m $\omega\tau$NAF expansion in terms of the τ^{-1} operator as follows,

$$k = \sum_{i=0}^{m-1} u_i \tau^i = \left(u_0 + u_1\tau^1 + u_2\tau^2 + \ldots + u_{m-1}\tau^{m-1}\right) \tag{10.19}$$

$$= \left(u_0 + u_1\tau^{-(m-1)} + u_2\tau^{-(m-2)} + \ldots + u_{m-1}\tau^{-1}\right) = \sum_{i=0}^{m-1} u_i \tau^{-(m-i)}$$

Summarizing, Koblitz elliptic curve scalar multiplication can be accomplished by processing elliptic point additions and τ and/or τ^{-1} mappings. Hence, a Koblitz multiplication algorithm is usually divided into two main phases: a ω-TNAF expansion of the scalar k; and the scalar multiplication itself based on the τ Frobenius operator and elliptic curve addition sequences.

10.6.2 $\omega\tau$NAF Scalar Multiplication in Two Phases

Algorithm 10.7 $\omega\tau$NAF Expansion[133, 132]

Require: *Curve Parameters; representative elements:* $\alpha_u = \beta_u + \gamma_u\tau$ *for* $u = 1, 3, \ldots, 2^{w-1} - 1$;$\delta$; *Scalar* k.
Ensure: $\omega\tau NAF(k)$
1: Compute $(r_0, r_1) \leftarrow k \bmod \delta$;
2: **for** $\{i = 0;\ (r_0 \neq 0)$ OR $(r_1 \neq 0);\ i = i + 1\}$ **do**
3: **if** r_0 is odd **then**
4: $u \leftarrow r_0 + r_1 t_w \bmod 2^w$;
5: **if** $u > 0$ **then**
6: $\xi \leftarrow 1$;
7: **else**
8: $\xi \leftarrow -1$; $u \leftarrow -u$;
9: **end if**
10: $r_0 \leftarrow r_0 - \xi\beta_u$; $r_1 \leftarrow r_1 - \xi\gamma_u$; $u_i \leftarrow \xi\alpha_u$;
11: **else**
12: $u_i \leftarrow 0$;
13: **end if**
14: $(r_0, r_1) \leftarrow (r_1 + \frac{\mu r_0}{2}, \frac{-r_0}{2})$;
15: **end for**
16: $l = i$;
17: **Return** $l, (u_{l-1}, u_{l-2}, \cdots, u_1, u_0)$;

Algorithms 10.7 and 10.8 show the adaptations of Solinas procedures as they were reported in [132, 133].

It should be noticed that Algorithm 10.7 produces the $\omega\tau$NAF expansion coefficients from right to left, i.e., the least significant coefficient u_0 is first produced, then u_1 and so on, until the most significant coefficient, namely, u_{l-1}, is obtained. Algorithm 10.8 on the contrary, computes the expression 10.17 from left to right, i.e., it starts processing u_{l-1} first, then u_{l-2} until it ends with the coefficient u_0.

Algorithm 10.8 $\omega\tau$NAF Scalar Multiplication [133, 132]

Require: $\omega\tau NAF(k) = \sum_{i=0}^{l-1} u_i \tau^i$, $P \in E_a(F_{2^m})$.
Ensure: kP
1: Precompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
2: $Q \leftarrow \mathcal{O}$;
3: **for** i from $l - 1$ downto 0 **do**
4: $Q \leftarrow \tau Q$;
5: **if** $u_i \neq 0$ **then**
6: Find u such that $\alpha_u = \pm u_i$;
7: **if** $u > 0$ **then**
8: $Q \leftarrow Q + P_u$;
9: **else**
10: $Q \leftarrow Q - P_{-u}$;
11: **end if**
12: **end if**
13: **end for**
14: **Return** Q;

The combination of those two characteristics is unfortunate as it forces us to work in a strictly sequential manner: First Algorithm 10.7 must be executed and only when it finishes, Algorithm 10.8 can start the computation of the Koblitz curve scalar multiplication operation. However, invoking Eq. (10.19), we can formulate a parallel version of Algorithm 10.8 as is shown in Algorithm 10.9. If two separated point addition units are available, the expected computational speedup of the parallel version in Algorithm 10.9 is of about 50 % when compared with its sequential version.

10.6.3 Hardware Implementation Considerations

In an effort to minimize the number of clock cycles required by Algorithm 10.8 when implemented in a hardware platform, we first proceed to pre-process the width-$\omega\tau$NAF expansion of coefficient k as described below.

Firstly, without loss of generality we will assume that the length of the expansion is m[14]. Secondly, let us recall that it is guaranteed that at most one of any consecutive ω coefficients of an $\omega\tau$NAF expansion is nonzero. Let $w_i \in [1, 3, 5, \ldots, 2^{w-1} - 1]$ denote each one of the up to $N_w = \lceil \frac{m}{\omega+1} \rceil$ nonzero $\omega\tau NAF$ expansion coefficients. Then, the expansion would have the following structure:

$$w_0, 0 \ldots 0, w_1, 0 \ldots 0, w_2, 0, \ldots, 0, w_{i-1}, 0 \ldots 0, w_{N_w - 1}$$

Above runs of up to $2w - 2$ consecutive *zeroes* [340], can be counted and stored. Let $z_i \in [\omega - 1, 2\omega - 2]$ denote the length of each of the at most

[14] Otherwise, if $l > m$, we can use Eq. (10.18) in order to reduce the expansion length back to m.

Algorithm 10.9 $\omega\tau$NAF Scalar Multiplication: Parallel Version

Require: $\omega\tau NAF(k) = \sum_{i=0}^{m-1} u_i \tau^i$, $P \in E_a(F_{2^m})$.
Ensure: kP

PreCompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
$Q = R = \mathcal{O}$;
$N = \lfloor \frac{m}{2} \rfloor$; $u_m = 0$;
for i from N downto 0 **do**
  $Q \leftarrow \tau Q$;
  if $u_i \neq 0$ **then**
    Find u such that $\alpha_{\pm u} = \pm u_i$;
    if $u > 0$ **then**
      $Q \leftarrow Q + P_u$;
    else
      $Q \leftarrow Q - P_{-u}$;
    end if
  end if
end for
$Q \leftarrow Q + R$;
Return Q;

for $j = N + 1$ **to** m **do**
  $R \leftarrow \tau^{-1} R$;
  if $u_j \neq 0$ **then**
    Find u such that $\alpha_{\pm u} = \pm u_j$;
    if $u > 0$ **then**
      $R \leftarrow R + P_u$;
    else
      $R \leftarrow R - P_{-u}$;
    end if
  end if
end for

Algorithm 10.10 $\omega\tau$NAF Scalar Multiplication: Hardware Version

Require: $\tau NAF_\omega(k)$ in the format: $w_0, z_1, w_2, z_3, \ldots, z_{N_w-2}, w_{N_w-1}$, $N_w = 2\lceil \frac{m}{w+1} \rceil$. Where $w_i \in [1, 3, 5, \ldots, 2^{w-1} - 1]$ and $z_i \in [w - 1, 2w - 2]$
Ensure: kP

Precompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
$Q \leftarrow \mathcal{O}$
for i from $N - 1$ downto 0 **do**
  if i is odd **then** {/*processing a zero coefficient z_i*/}
    $Q \leftarrow \tau^{w-1} Q$
    $z_i \leftarrow z_i - (w - 1)$
    if $z_i \neq 0$ **then**
      $Q \leftarrow \tau^{z_i} Q$
    end if
  else {/*processing a nonzero coefficient w_i*/}
    Find u such that $\alpha_u = \pm \omega_i$;
    if $u > 0$ **then**
      $Q \leftarrow Q + P_u$;
    else
      $Q \leftarrow Q - P_{-u}$;
    end if
  end if
end for
Return Q;

$N_w = \lfloor \frac{m}{\omega+1} \rfloor$ *zero* runs. Then, the proposed compact version of the expansion has the following form,

$$w_0, z_0, w_1, z_2, \ldots, z_{N_w-1}, w_{N_w-1} \tag{10.20}$$

In this new format we just need to store in memory at most $2\lceil \frac{(m)}{\omega+1} \rceil$ expansion coefficients. Algorithm 10.10 shows how to take advantage of the compact representation just described. Given the relatively cheap cost of the field squaring operation, steps 5-8 of Algorithm 10.10 can compute up to $\omega-1$ applications of the τ Frobenius operator[15]. This will render a valuable saving of system clock cycles. Moreover, using the same idea already employed in Algorithm 10.9, we can parallelize Algorithm 10.10 using the τ and τ^{-1} operators concurrently. The resulting procedure is shown in Algorithm 10.11.

Algorithm 10.11 $\omega\tau$NAF Scalar Multiplication: Parallel HW Version

Require: $\tau NAF_\omega(k)$ in the format: $w_0, z_1, w_2, z_3, \ldots, z_{N_w-2}, w_{N_w-1}$, $N_w = 2\lceil \frac{m}{w+1} \rceil$. Where $w_i \in [1, 3, 5, \ldots, 2^{w-1} - 1]$ and $z_i \in [w-1, 2w-2]$
Ensure: kP

1: PreCompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
2: $Q = R = \mathcal{O}$;
3: $N = \lfloor \frac{N_w}{2} \rfloor$;
4: **for** i from N downto 0 **do**
5: **if** i is odd **then**
6: $Q \leftarrow \tau^{\omega-1} Q$;
7: $z_i \leftarrow z_i - (w-1)$;
8: **if** $z_i \neq 0$ **then**
9: $Q \leftarrow \tau^{z_i} Q$;
10: **end if**
11: **else**
12: Find u such that $\alpha_{\pm u} = \pm u_i$;
13: **if** $u > 0$ **then**
14: $Q \leftarrow Q + P_u$;
15: **else**
16: $Q \leftarrow Q - P_{-u}$;
17: **end if**
18: **end if**
19: **end for**
20: $Q \leftarrow Q + R$;
21: **Return** Q;

for $j = N + 1$ **to** m **do**
if i is odd **then**
$R \leftarrow \tau^{-(\omega-1)} R$;
$z_j \leftarrow z_j - (w-1)$;
if $z_i \neq 0$ **then**
$R \leftarrow \tau^{z_j} R$;
end if
else
Find u such that $\alpha_{\pm u} = \pm u_j$;
if $u > 0$ **then**
$R \leftarrow R + P_u$;
else
$R \leftarrow R - P_{-u}$;
end if
end if
end for

[15] Let us recall that applying i times the τ Frobenius operator over an elliptic point Q consists of squaring each coordinate of Q i times. See §6.2 for details about how to compute efficiently squaring and other field arithmetic operations

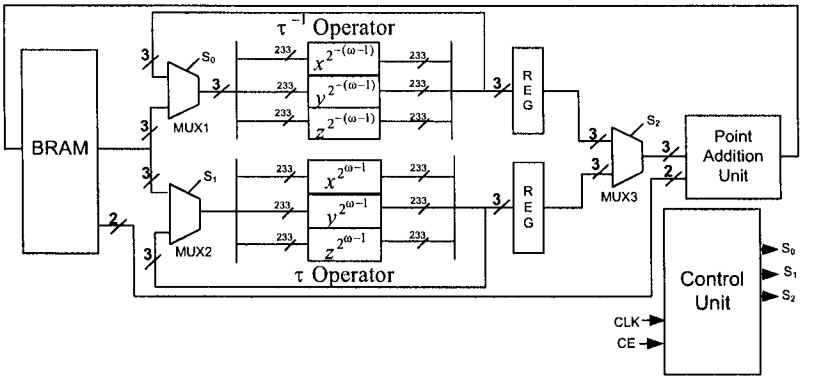

Fig. 10.5. A Hardware Architecture for Scalar Multiplication on the NIST Koblitz Curve K-233

Proposed Hardware Architecture

According to Algorithm 10.11, one can accomplish a scalar multiplication operation by computing two sequences, namely, τ operator-then-add and; τ^{-1} operator-then-add. Both sequences are independent and therefore, they can be processed concurrently provided that hardware resources meet up design requirements. An aggressive approach would be to use two point addition units with τ and τ^{-1} blocks operating separately. That, however, could be unaffordable as the point addition block consumes a vast amount of hardware resources. A more conservative approach consisting of a single point addition unit is shown in Fig. 10.5. The main idea used there is to keep the τ and τ^{-1} computations in parallel while a multiplexer block allows the control unit to decide which result will be processed next by the point addition unit. Intermediate results required for next stages of the algorithm are read/written in a Block select RAM (BRAM).

The inputs/output of the point addition unit read/write data from/to the BRAM block according to an address scheme orchestrated by the control unit. Data paths for the τ and τ^{-1} operators and then point addition are adjusted by providing selection bits for the three multiplexers MUX1, MUX2, and MUX3. Notice that all three multiplexers handle three 233-bit inputs/outputs. This is the required size for a three-coordinate LD projective point as it was described in Subsection 4.5.2. The τ and τ^{-1} operators were designed using the formulae described in §6.2. The Point Addition Unit (PAU) performs the point addition operation using the LD-affine mixed coordinates algorithm to be explained in the next Section. PAU has two inputs. One input comes from (via MUX3) the output of either τ or τ^{-1} blocks in the form of a three-coordinate LD projective point. The other input comes directly from the BRAM block and corresponds to one of the pre-computed multiples of P, namely, $P_{u_i} =$

$\alpha_{u_i} P$. Those multiples have been pre-computed in affine coordinates. A 4- bit counter and a ROM constitute the control unit block. The ROM block is filled with *control words*, which are used at each clock cycle for the orchestration and synchronization of algorithm's dataflow. The ROM block address bits are timely incremented by a 4-bit counter. A total of 11 bits (8 bits for each port of the BRAM, 1 bit for MUX1, 1 bit for MUX2 and 1 bit for MUX3) are used for controlling and synchronizing the whole circuitry. The 11-bit control word for each clock cycle is filled in the BRAM block, and then they are extracted at the rising edge of each clock cycle.

The expected performance of the architecture shown in Fig. 10.5 can be estimated as follows. As it has been mentioned, in a $\omega\tau$ NAF expansion there exists a total of $N_w = \lceil \frac{m}{\omega+1} \rceil$ nonzero coefficients. Let ξ be the number of cycles required for computing an elliptic point addition operation. Knowing that the Frobenius operators depicted in Fig. 10.5 are each able to compute $\omega - 1$ τ or τ^{-1} operators in one cycle, it seems fair to say that our architecture can process a coefficient zero in $\frac{1}{\omega-1}$ cycles. Therefore, the total number of system clock cycles required by Algorithm 10.10 for computing a scalar multiplication can be estimated as,

$$\#\text{Number of Clock Cycles} = \xi \frac{m}{\omega+1} + \frac{1}{\omega-1}\frac{\omega m}{\omega+1}. \tag{10.21}$$

In the case of Algorithm 10.11 since the τ and τ^{-1} operations are computed at the same time that the point addition processing is taking place, the total number of clock cycles can be estimated as just,

$$\#\text{Number of Clock Cycles} = \xi \frac{m}{\omega+1}. \tag{10.22}$$

As a way of illustration, let us assume that the architecture shown in Fig. 10.5 has been implemented using the arithmetic building blocks for the NIST recommended K-233 Koblitz curve. Then using $m = 233$ and $\xi = 8$ and equations (10.21) and (10.22), a saving of 14.28%,13.51% and 13.04% can be obtained when using $\omega = 4, 5, 6$, respectively.

10.7 Half-and-Add Algorithm for Scalar Multiplication

Schroeppel [322] and Knudsen [176] independently proposed in 1999 a method to speedup scalar multiplication on elliptic curves defined over binary extension fields. Their method is based on a novel elliptic curve primitive called *point halving*, which can be defined as follows.

Given a point Q of odd order, compute P such that $Q = 2P$. The point P is denoted as $\frac{1}{2}Q$. Since theoretically, point halving is up to three times as fast as point doubling, it is possible to improve the performance of scalar multiplication computation $Q = nP$ by replacing the double-and-add algorithm

with a half-and-add method based on an expansion of the scalar n in terms of negative powers of 2.

As it was discussed in Chapter 2, the efficiency of ECDSA depends on the arithmetic involving the points of the curve. For this reason it becomes necessary to implement efficient curve operations in order to obtain high performances. In this Section we describe an architecture that employs a parallelized version of the half-and-add method and its associated building blocks.

The rest of this Section is organized as follows. Subsection 10.7.1, describes the algorithms utilized for implementing elliptic curve arithmetic. In Subsection 10.7.2, the proposed hardware architecture is explained in detail.

10.7.1 Efficient Elliptic Curve Arithmetic

With the help of the arithmetic operators described in Chapter 6, we can efficiently construct the three main elliptic curve operations, namely, point addition, point doubling and point halving.

As a means of avoiding the expensive field inversion operation, it results convenient to work with *López-Dahab (LD)* projective coordinates[16]. For convenience, here we will repeat some of the main characteristics of those coordinates.

In LD projective coordinates, the projective point (X:Y:Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z^2$. The elliptic curve Equation (10.6) mapped to LD projective coordinates is given as,

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{10.23}$$

The point at infinity is represented as $\mathcal{O} = (1 : 0 : 0)$. Let $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : 1)$ be an arbitrary point belonging to the curve 4.19. Then the point $-P = (X_1 : X_1 + Y_1 : Z)$ is the addition inverse of the point P.

Point Doubling

The point doubling primitive $2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3)$ can be performed as,

$$\begin{aligned} &Z_3 = {X_1}^2 \cdot {Z_1}^2; X_3 = {X_1}^4 + b \cdot {Z_1}^4; \\ &Y_3 = b{Z_1}^4 Z_3 + X_3 \cdot (aZ_3 + {Y_1}^2 + b{Z_1}^4) \end{aligned} \tag{10.24}$$

Assuming that only one field multiplier block is available, it is possible to compute above Equations in just three clock cycles as shown in Table 10.7.

[16] LD projective coordinates were already studied in Section 4.5.

Table 10.7. Parallel López-Dahab Point Doubling Algorithm

A Parallel approach of point doubling, LD-affine coordinates.		
Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates		
on $E/K : y^2 + xy = x^3 + ax^2 + b, a \in \{0, 1\}$.		
Output: $2P = (X_3 : Y_3 : Z_3)$ in LD coordinates		
# cycle	C_0	C_1
1. cycle:	$Z_3 = X_1^2 \cdot Z_1^2$	$T_1 = b \cdot Z_1^4$
2. cycle:	$T_2 = (X_1^4 + T_1) \cdot (Z_3 + Y_1^2 + T_1)$	$X_3 = X_1^4 + T_1$
3: cycle:	$Y_3 = T_1 \cdot Z_3 + T_2$	

Point Addition

If $Q \neq -P$, the point addition primitive $(X_1 : Y_1 : Z_1) + (X_2 : Y_2) = (X_3 : Y_3 : Z_3)$ can be performed at a computational cost of 8 field multiplications as,

$$\begin{array}{ll} A = Y_2 \cdot Z_1^2 + Y_1; & B = X_2 \cdot Z_1 + X_1; \\ C = Z_1 \cdot B; & D = B^2 \cdot (C + aZ_1^2); \\ Z_3 = C^2; & E = A \cdot C; \\ X_3 = A^2 + D + E; & F = X_3 + X_2 \cdot Z_3; \\ G = (X_2 + Y_2) \cdot Z_3^2; & Y_3 = (E + Z_3) \cdot F + G \end{array} \tag{10.25}$$

Table 10.8. Parallel López-Dahab Point Addition Algorithm

A parallel approach of point addition, LD-affine coordinates.		
Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates,		
$Q = (x_2, y_2)$ in affine coordinates		
on $E/K : y^2 + xy = x^3 + ax^2 + b$.		
Output: $P + Q = (X_3 : Y_3 : Z_3)$ in LD coordinates		
# cycle	C_0	C_1
1. cycle:	$Y_3 = y_2 \cdot Z_1^2 + Y_1$	
2. cycle:	$X_3 = x_2 \cdot Z_1 + X_1$	
3. cycle:	$T_1 = X_3 \cdot Z_1$	
4. cycle:	$X_3 = X_3^2 \cdot (a \cdot Z_1^2 + T_1)$	$Z_3 = T_1^2$
5. cycle:	$X_3 = Y_3 \cdot T_1 + X_3 + Y_3^2$	$T_1 = Y_3 \cdot T1$
6. cycle:	$T_1 = x_2 \cdot Z_3 + X_3$	
7. cycle:	$Y_3 = (x_2 + y_2) \cdot Z_3^2$	$T_2 = T_3$
8. cycle:	$Y_3 = (T_2 + Z_3) \cdot T_1 + Y_3$	

Once again, we point out that field multiplication is by far the most time consuming arithmetic operation. Field addition can be time neglected in a hardware implementation.

Therefore we can parallelize some operations in such a way that we can perform two operations at a time. As it is shown in Table 10.8, by rearranging the set of Equations 10.25 we can manage for computing a point addition operation in LD projective coordinates in just eight clock cycles.

Point Halving

Point halving can be seen as the reverse operation of point doubling [96]. We can define the elliptic curve point halving as follows. Let $Q = (x_2, y_2)$ be an arbitrary point that belongs to the curve of Eq. (10.6). Our problem in hand is to find a second point $P = (x_1, y_1)$, such that $Q = 2P$: This can be accomplished by solving the following set of equations,

$$\begin{aligned} \lambda^2 + \lambda &= x_2 + a \\ x_1 &= \sqrt{y_2 + x_2(\lambda + 1)} \\ y_1 &= \lambda x_1 + x_1^2 \end{aligned}$$

Algorithm 10.12 Point Halving Algorithm

Require: $2P = (x_2, y_2)$
Ensure: $P = (x_1, y_1)$
1: Solve $\lambda^2 + \lambda = x_2 + a$ for λ.
2: $t = y_2 + x_2 \cdot \lambda$;
3: **if** $Tr(t) = 0$ **then**
4: $x_1 = \sqrt{t + x_2}$;
5: **else**
6: $\lambda = \lambda + 1; x_1 = \sqrt{t}$;
7: **end if**
8: $y_1 = \lambda \cdot x_1 + x_1^2$;
9: **Return** (x_1, y_1)

Algorithm 10.12 was proposed in [96] for computing an elliptic point halving. However, it results more convenient in practice to define the *λ-representation* of a point as follows. Given $Q = (x, y) \in E(GF(2^m))$, let us define (x, λ_Q), where

$$\lambda_Q = x + \frac{y}{x}$$

Given the λ-representation of Q, we may compute a point halving without converting back to affine coordinates. In this way, repeated halvings can be performed directly on λ-representation.

Half-and-Add Scalar Multiplication Algorithm

In Chapter 6 several algorithms addressing the problem of how to perform efficient finite field arithmetic were studied. Notice that Algorithm 10.12 requires the following $GF(2^m)$ arithmetic main building blocks,

1. Computing field square root (studied in §6.2).
2. Computing the trace (studied in §6.4.1).
3. Solving quadratic equations (studied in §6.4.2).

Above operations constitute the building blocks for performing elliptic curve scalar multiplication using the half-and-add method shown in Algorithms 10.12 and 10.13.

Algorithm 10.13 Half-and-Add LSB-First Point Multiplication Algorithm

Require: $P \in E(GF(2^m))$, $k = k'_0/2^{m-1} + \cdots + k'_{m-1} + 2k'_m \bmod n$, with $k_i \in \{-1, 0, 1\}$ for $i =, 1, \ldots, m$.
Ensure: kP
1: $Q = \mathcal{O}$;
2: **if** $k'_m = 1$ **then**
3: $Q = 2P$;
4: **end if**
5: **for** i from $m - 1$ downto 0 **do**
6: **if** $k'_i > 0$ **then**
7: $Q = Q + P$;
8: **else if** $k'_i < 0$ **then**
9: $Q = Q - P$;
10: **end if**
11: $P = P/2$;
12: **end for**
13: **Return** (Q)

10.7.2 Implementation

The proposed architecture for achieving elliptic curve scalar multiplication is shown in Figure 10.6. The architecture consists of two main units, namely, an Arithmetic Logic Unit (ALU) block (responsible of performing field arithmetic and elliptic curve arithmetic), and a control unit (that manages and controls the dataflow of the whole circuit).

Control Unit

Table 10.9 shows the operations that can be performed by the circuit per clock cycle. In the first column the operations that the ALU can perform are listed. The first eight rows specify the sequence of operations needed for computing an elliptic curve point addition. The next three rows specify the operations needed for computing a point doubling primitive. The last three rows show the necessary operations for computing a point halving (either in λ-representation or in affine coordinates).

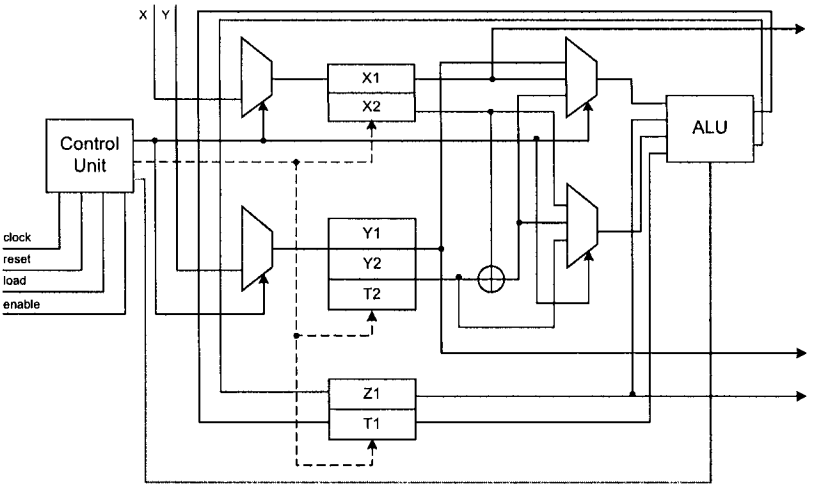

Fig. 10.6. Point Halving Scalar Multiplication Architecture

The second column represents the inputs given to the ALU circuit, whereas the fourth column shows the ALU circuit output being written to memory.

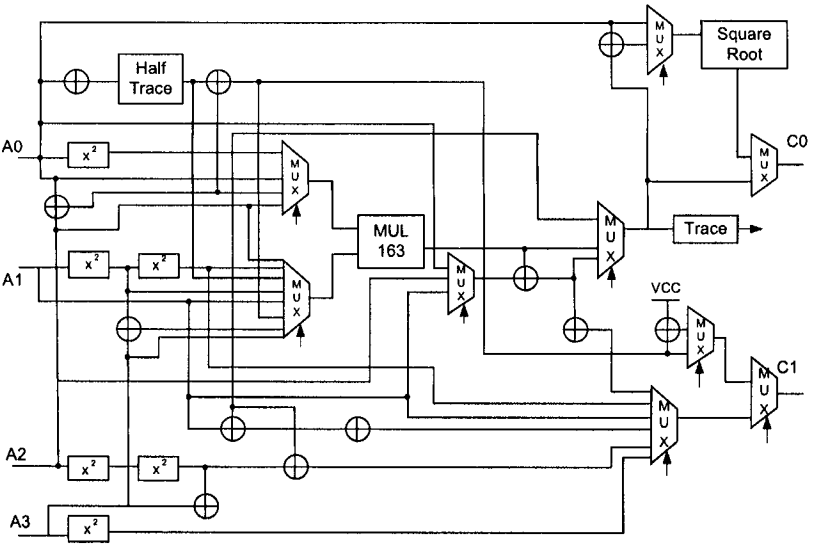

Fig. 10.7. Point Halving Arithmetic Logic Unit

Finally, the third column includes a twenty-six bit control word that stipulates which parts of the Arithmetic Logic Unit must be activated by the Control Unit. The control word format is explained below.

Table 10.9. Operations Supported by the ALU Module

operation	input	control word	output
	$a_0a_1a_2a_3$	$s_{25}\cdots s_0$	c_0c_1
$Y_1 = y_2 \cdot Z_1^2 + Y_1$	$y_2Z_1Y_1-$	1xx01000xx11010000110xxx1x	Y_1x
$X_1 = x_2 \cdot Z_1 + X_1$	$x_2Z_1X_1-$	110xxxx0xx00010010110xxx1x	X_1x
$T_1 = X_1 \cdot Z_1$	X_1Z_1--	10xxxxx0x0xx01001xx00xxx1x	T_1x
$X_1 = X_1^2 \cdot (Z_1^2 + T_1)$	$X_1Z_1-T_1$	00xxxxx010xx00100xx0000111	X_1Z_1
$X_1 = Y_1 \cdot T_1 + X_1 + Y_1^2$	$y_2Z_1Y_1-$	0xx01000xx11010000110xxx1x	T_1X_1
$T_2 = x_2 \cdot Z_1 + X_1$	$x_2Z_1X_1-$	110xxxx0xx00010010110xxx1x	T_2x
$Y_1 = (x_2 + y_2) \cdot Z_1^2$	$x_2Z_1y_2-$	01xxx010xx0111000xx00xxx1x	Y_1x
$Y_1 = (T_1 + Z_1) \cdot T_2 + Y_1$	$Y_1T_1T_2Z_1$	0xx0010x1011100110010xxx1x	Y_1x
$Z_1 = X_1^2 \cdot Z_1^2$	X_1Z_1--	00xxxxx0x0xx00000xx0000011	Z_1T_2
$X_1 = (X_1^4 + T_1) \cdot (Y_1^2 + Z_1 + T1)$	$Y_1Z_1X_1T_1$	0x010xxxx10xxxxxxxx0101011	T_2X_1
$Y_1 = Z_1 \cdot T_1 + T_2$	$T_2Z_1-T_1$	00xxx101xx01010010110xxx1x	Y_1x
Point Halving (affines)	x_2-y_2-	101xxx01xx01011010110xxx00	x_2y_2
Point Halving (λ-representation)	x_2-y_2-	101xxx01xx0101110xx00xxx00	$x_2\lambda$
$y_2 = \lambda x_2 + x_2^2$	x_2-y_2-	101xxx01xx01010011010xxx1x	$-y_2$

Each control word consists of a string of 26 bits organized as follows:

$$\underbrace{XX001010}_{direction}\ \underbrace{1100}_{MUX}\ \underbrace{100110010XXX1X}_{ALU}$$

The first eight bits designate the addresses to be read by the memory block, the next four bits designate which operand will be loaded to the ALU unit, and finally the last fourteen bits designate which operations will be performed by the ALU unit according to the list of supported operations shown in Table 10.9.

As an example, consider point halving computation in affine coordinates of Algorithm 10.12. The datapath for this computation is illustrated in Fig. 10.8. First, it is necessary to load x_2, y_2 into the input registers A_0, A_2, respectively. Additionally, a copy of x_2 is stored in A_1. Then, the operations for loading $HT(A_0 + 1)$ and A_1 on the finite field multiplier are commanded by the Control Unit. Next, we multiply $A_1 \cdot HT(A_0 + 1)$ and immediately after A_2 is added to that product obtaining $A_2 + A_1 \cdot HT(A_0 + 1)$. Thereafter, the result obtained by the multiplication operation is computed into the trace unit, in order to choose the appropriate operand for the square-root unit, and to send the corresponding outputs C_0, C_1. The dataflow just described is highlighted in Figure 10.8.

As mentioned previously, our architecture allows us to perform three main elliptic curve operations, namely, point addition, point doubling and point

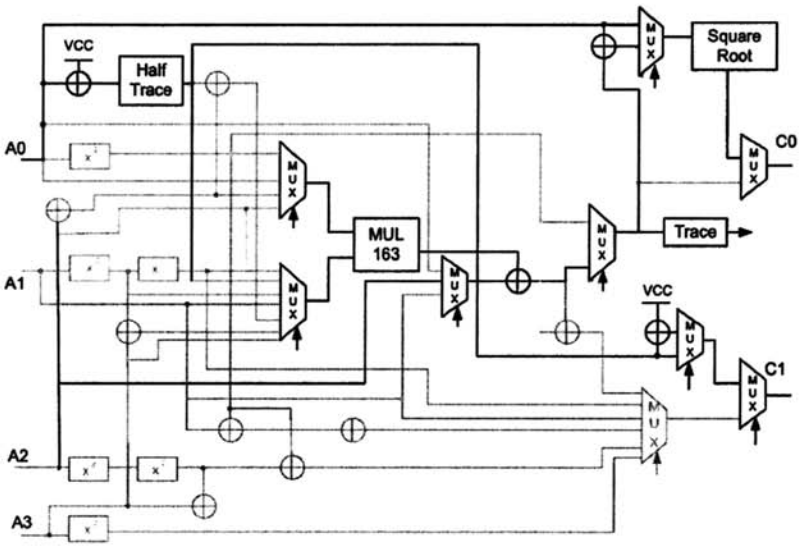

Fig. 10.8. Point Halving Execution

halving, Table 10.10 lists the number of cycles required in order to perform such operations. Furthermore, Figures 10.9 and 10.10 show the time diagram corresponding to the execution of the point addition and point doubling primitives, respectively.

Table 10.10. Cycles per Operation

Elliptic curve operations	# cycles
Point Halving (affine coordinates)	1
Point Halving (λ-representation)	2
Point Doubling	3
Point Addition	8

10.7.3 Performance Estimation

We estimate the running time of the circuit of Fig. 10.6 as follows. We need eight cycles and one cycle for performing a Point Addition (PA) in mixed LD coordinates and a Point Halving (PH) operation, respectively. On the other hand, the computational cost of Algorithm 10.13 is approximately,

$$\frac{m}{3}PA + mPH.$$

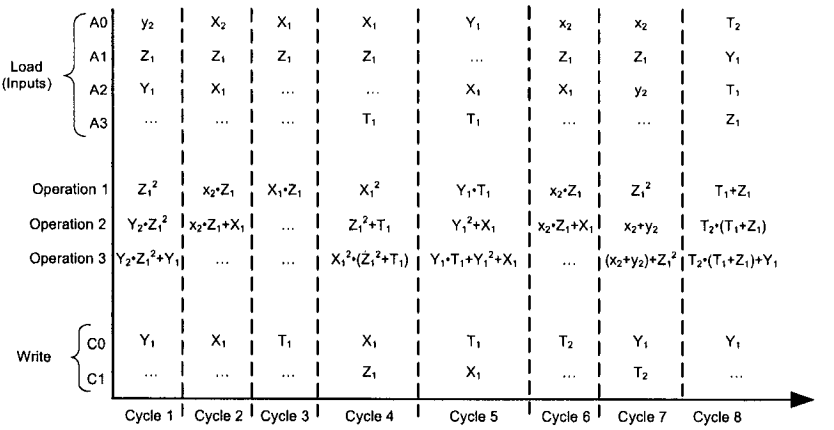

Fig. 10.9. Point Addition Execution

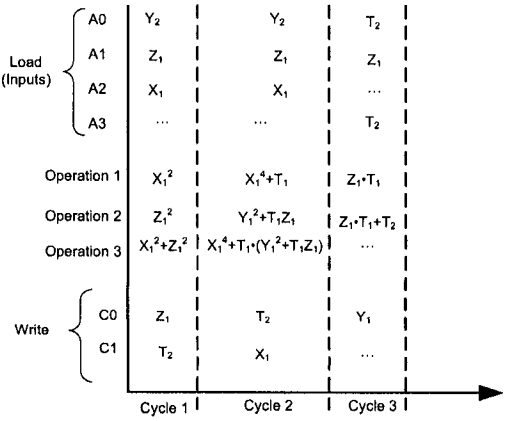

Fig. 10.10. Point Doubling Execution

Translating above equation to clock cycles, we get,

$$\frac{m}{3}(8) + mPH(1) = \frac{11}{3}m \text{ Clock Cycles.}$$

In other words, the architecture presented in this Section (see Figures 10.6 and 10.7) needs approximately $\frac{11}{3}m$ clock cycles for performing an elliptic curve point multiplication using the Half-and-Add Algorithm 10.13.

Table 10.11. Fastest Elliptic Curve Scalar Multiplication Hardware Designs

Author	year	platform	m	clock MHz	time (μS)	Cost LUTs	$\frac{m}{T \cdot LUT}$
Cruz-A. et al.[54]	2006	Virtex II	233	27.58	17.64	39762(11)	332.19
Hernández-R et al.[137]	2005	Virtex II	163	23.94	25.0	22665	287.67
Cheung et al. [50]	2005	Virtex 4	113	65	30	13922 (est)	270.55
Shu et al.[329]	2005	Virtex II	163	68.9	48	25763	131.81
Saqib et al.[310]	2006	Virtex II	191	9.99	61.16	39252(24)	79.56
Lutz [216]	2004	Virtex II	163	66.0	75	10017	216.95
Jarvinen et al.[155]	2004	Virtex II	163	90.2	106	36158(est)	42.53
Gura et al. [125]	2002	Virtex II	163	66.4	143	22665	36.14
Satoh et al. [313]	2003	0.13μm CMOS	160	510.2	190	–	–
Orlando et al.[261]	2000	Virtex	167	76.7	210	3002	265.03
Bednara et al. [20]	2002	Virtex	191	50	270	–	–
Sozzani et al. [341]	2005	0.13μm CMOS	163	417	270	–	–
Ernst et al. [313]	2002	Atmel	113	12	1400	–	–
Schroeppel et al. [322]	2003	0.13μm CMOS	178	227	4400	143K gates	–

10.8 Performance Comparison

In this Section we compare some of the most representative elliptic curve designs reported during this decade. In our survey we considered three metrics: speed, compactness and efficiency. Our study tries to sum up the state-of-the-art of scalar multiplication hardware implementations.

Table 10.11 shows the fastest designs reported to date for elliptic scalar multiplication over $GF(2^m)$[17]. It can be observed that the design of [54] which features a specialized design on Koblitz curves shows the highest speed of all designs considered.

Table 10.12. Most Compact Elliptic Curve Scalar Multiplication Hardware Designs

Author	year	platform	m	clock MHz	time (mS)	Cost	$\frac{m}{T \cdot Gates}$
Kim et al. [172]	2002	0.35μm CMOS	192 binary	10	36.2 (est)	16.84K gates	0.315
Öztürk et al. [265]	2004	0.13μm CMOS	167 prime	20	31.9	30.3K gates	0.1727
			167 prime	200	3.1	34.4K gates	1.56
Aigner et al. [2]	2004	0.13μm CMOS	191 binary	10	46.9	25K NANDs	0.163
Schroeppel et al. [322]	2003	0.13μm CMOS	178 binary	227	4.4	143K gates	0.283
Shuhua et al. [330]	2005	Virtex II	192 prime	50	6	4729 LUTs	–

[17] Whenever the number of LUTs utilized by the design is not available, an estimation based on the reported number of CLBs has been made. The number in parenthesis in the seventh column represents the total number of BRAMs.

In Table 6.4 we show a selection of some of the most compact reconfigurable hardware elliptic curve designs reported to date. It is noted that this category is dominated by those designs implemented in VLSI working with elliptic curves defined over $GF(2^m)$. Indeed, the most compact $GF(P)$ elliptic curve design in [265] has a hardware cost 1.8 times greater than that of the smallest $GF(2^m)$ elliptic curve design in [172].

We measure efficiency by taking the ratio of number of bits processed over slices multiplied by the time delay achieved by the design, namely,

$$\frac{bits}{Slices \times timings}$$

For instance, consider the Koblitz design presented in [54]. As is shown in Table 10.11, working over $GF(2^{233})$, that design achieved a time delay of just 17.64μS at a cost of 39762 Look Up Tables (LUTs) and 11 Block RAMs. Therefore its efficiency is calculated as,

$$\frac{bits}{Slices \times timings} = \frac{233}{39762 \times 17.64\mu} = 332.19$$

When comparing the designs featured in Tables 10.11 and 10.13, it is noticed that the fastest and most efficient multiplier designs are the Koblitz elliptic curve designs as well as the half-and-add scalar multiplication design studied in this Chapter.

Table 10.13. Most Efficient Elliptic Curve Scalar Multiplication Hardware Designs

Author	year	platform	m	clock MHz	time (μS)	Cost LUTs	$\frac{m}{T \cdot LUT}$
Cruz-A. et al.[54]	2006	Virtex II	233	27.58	17.64	39762(11)	332.19
Hernández-R et al.[137]	2005	Virtex II	163	23.94	25.0	22665	287.67
Cheung et al. [50]	2005	Virtex 4	113	65	30	13922 (est)	270.55
			163	35	50	20047 (est)	162.61
Orlando et al.[261]	2000	Virtex	167	76.7	210	3002	265.03
Lutz [216]	2004	Virtex II	163	66.0	75	10017	216.95
Shu et al.[329]	2005	Virtex II	163	68.9	48	25763	131.81
			233	67.9	89	35800	73.13
Saqib et al.[310]	2006	Virtex II	191	9.99	61.16	39252(24)	79.56
			191	9.99	114.71	39252(24)	42.41
Jarvinen et al.[155]	2004	Virtex II	163	90.2	106	36158(est)	42.53
			193	90.2	139	38500(est)	36.06
			233	73.6	227	46040(est)	22.29
Gura et al. [125]	2002	Virtex II	163	66.4	143	22665	36.14
Leung et al. [205]	2002	Virtex	113	31	750	17506	8.61

10.9 Conclusions

Two major factors contribute for achieving high performances in the architectures presented throughout this chapter. Firstly, the usage of parallel strategies applied at every stage of the design. Secondly, efficient elliptic curve algorithms such as the Montgomery point multiplication, scalar multiplication on Koblitz curves, the half-and-add method, etc, along with their efficient implementations on reconfigurable hardware. Furthermore, it resulted also crucial to take advantage of the lower-grained characteristic of reconfigurable hardware devices and their associated functionality (in the form of BRAMs and other resources).

In §10.5 we studied a generic architecture able to compute the scalar multiplication in Hessian form as well as the Montgomery point multiplication algorithm. It is noticed that theoretically (see Table 10.1), the Weierstrass form utilizing the Montgomery point multiplication formulation can be computed in about half the execution time consumed by the Hessian form. This prediction was confirmed in practice in [310] for elliptic curves defined over $GF(2^{191})$, as is shown in Table 10.13.

Then, we presented in §10.6 parallel formulations of the scalar multiplication operation on Koblitz curves. The main idea proposed in that Section consisted on the concurrent usage of the τ and τ^{-1} Frobenius operators, which allowed us to parallelize the computation of scalar multiplication on elliptic curves. On the other hand, we described a compact format of the $\omega\tau$NAF expansion which was especially tailored for hardware implementations. In this new format at most $2\lceil \frac{(m)}{\omega+1} \rceil$ expansion coefficients need to be stored and processed, provided that the arithmetic unit can compute up to $\omega - 1$ subsequent applications of the τ Frobenius operator in one single clock cycle. Furthermore, it was shown that by using as building blocks the τ and τ^{-1} Frobenius operators along with a single point addition unit, a parallel version of the classical double-and-add scalar multiplication algorithm can be obtained, with an estimated speedup of up to 14% percent when compared with the traditional sequential version.

In §10.7 we presented an architecture that is able to compute the elliptic curve scalar multiplication using the half-and-add method. Additionally, we presented optimizations strategies for computing a point addition and a point doubling using LD projective coordinates in just eight and three clock cycles, respectively.

Finally, in §10.8 we compared some of the most representative elliptic curve designs reported during this decade. In our survey we considered three metrics: speed, compactness and efficiency. Our study tries to sum up the state-of-the-art of scalar multiplication hardware implementations.