A Comparison of Double Point Multiplication Algorithms and their Implementation over Binary Elliptic Curves

Reza Azarderakhsh and Koray Karabina

Abstract

Efficient implementation of double point multiplication is crucial for elliptic curve cryptographic systems. We revisit three recently proposed simultaneous double point multiplication algorithms. We propose hardware architectures for these algorithms, and provide a comparative analysis of their performance. We implement the proposed architectures on Xilinx Virtex-4 FPGA, and report on the area and time results . Our results indicate that differential addition chain based algorithms are better suited to compute double point multiplication over binary elliptic curves for both high performance and resource constrained applications.

Index Terms

Elliptic curve cryptography (ECC), differential addition chains, binary fields, double point multiplication, Field Programmable Gate Array (FPGA).

I. Introduction

Elliptic curves have been extensively used in public key cryptography especially in embedded, resource-constrained, and high-performance applications. Point multiplication is a major operation in many elliptic curve based cryptosystems. For example, if a subgroup $\langle P \rangle$ of an elliptic curve E is deployed in a Diffie-Hellman type key exchange protocol, then a party A chooses a secret random integer a, computes aP, and sends it to the other party with whom A wants to share a secret key. For another example, if a prime order cyclic subgroup $\mathbb{G}$ of an elliptic curve E is deployed in a Cramer-Shoup encryption scheme, then in the key generation phase a party A computes $aP + bQ$ as a part of her public key. Here, P, Q are two random generators of $\mathbb{G}$, and a, b are two random integers all chosen by the party A. Even though A announces P and Q as a part of her public key, she has to keep a and b private as a part of her secret key. Similarly, in the decryption phase, A computes $\tilde{a}P_1 + \tilde{b}P_2$, where $P_1, P_2 \in \mathbb{G}$ are parts of a ciphertext, and the integers $\tilde{a}, \tilde{b}$ have to be kept secret by A. The security of such cryptosystems relies heavily on the difficulty of the *discrete logarithm problem* (DLP) in $\mathbb{G}$ (i.e., given $P, aP \in \mathbb{G}$, compute a). A generic way to solve DLP in $\mathbb{G}$ is to use the Pollard's rho method that runs in time $O(\sqrt{G})$ [20]. *Side channel analysis* includes a class of other methods to recover the secret a by making use of *side channel information* extracted from the computation of aP. A conventional method for computing aP is to use a variant of double-and-add type algorithms based on the binary representation of the secret exponent a. Such an algorithm would suffer from *power analysis* attacks when doubling and addition operations are distinguishable [8]. One method to provide Diffie-Hellman type protocols with some level of protection against side channel attacks is to split the scalar $a = r + (a - r)$ for some secret random integer r, and to compute $aP = rP + (a - r)P$ [7].

For the sake of generality, let $\mathbb{G}$ be an additive abelian group. Given an integer a and a point $P \in \mathbb{G}$, a *(single) point multiplication* algorithm computes $aP \in \mathbb{G}$. Given two integers a, b and two points P, $Q \in \mathbb{G}$, a *double point multiplication* algorithm computes $aP + bQ \in \mathbb{G}$. As we see in the above examples, having an *efficient* and *secure*[1] double point multiplication algorithm is crucial for many cryptographic schemes. Another scenario where one needs efficient and secure double point multiplication is to speed up single point multiplication over elliptic curves with endomorphisms, see [11],[10],[12].

A naive way to perform double point multiplication is to perform two single point multiplications. A more efficient method is to compute $aP + bQ$ simultaneously. Straus-Shamir's trick (see Algorithm 14.88 in [17]) and interleaving [18] are two such methods. Straus-Shamir's type simultaneous double point multiplication algorithms are vulnerable to side-channel analysis because double and add instructions are not performed in a regular fashion. Fortunately, *recoding* the scalars a and b allows us to adapt Straus-Shamir's type algorithms in such a way that the same instructions are executed in the same order. Joye and Tunstall [15] proposed several methods of regular recoding of scalars for regular point multiplication algorithms, which can immediately be adapted to yield regular simultaneous double point multiplication algorithms. In particular, their signed-digit recoding method with the digit set $\{\pm 1, \pm 3\}$ yields a regular double point multiplication algorithm, that we call the JT-$\{\pm 1, \pm 3\}$ algorithm. JT-$\{\pm 1, \pm 3\}$ costs half addition and one doubling per scalar bit. Using differential addition chains (DAC) is another method to perform simultaneous double point multiplication; see for instance [19], [2], and [6]. DAC-method

R. Azarderakhsh is with the Department of Combinatorics and Optimization, Center for Applied Cryptographic Research (CACR), University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. E-mail address: razarder@uwaterloo.ca.

K. Karabina is with the Department of Mathematics, Bilkent University, Bilkent, Ankara, Turkey, 06800. E-mail address: karabina@fen.bilkent.edu.tr.

[1] *Here, "secure" implies resistance against side channel analysis atacks.*

Table I. A comparison of three simultaneous double point multiplication algorithms JT-$\{\pm 1\}$, B-NBC, and AK-DAC. Double and add operations are denoted by D and A, respectively.

Algorithm	Per-bit cost	Regular	DAC-based	Parallelizable
JT-$\{\pm 1, \pm 3\}$	$0.5A + 1D$	Yes	No	No
B-NBC	$2A + 1D$	Yes	Yes	Yes
AK-DAC	$1.4A + 1.4D$	Yes	Yes	Yes

is attractive because it yields potentially simple power analysis resistant algorithms due to the uniform pattern of operations executed; and it is especially efficient in elliptic curves setting because double and add operations can be performed using x-coordinates only. Bernstein [6] proposed a double point multiplication algorithm based on the *new binary chain*, that we call the B-NBC algorithm. B-NBC has a uniform structure, and costs two additions and one doubling per scalar bit. More recently, Azarderakhsh and Karabina [4] proposed a simultaneous double point multiplication algorithm based on DAC, that we call the AK-DAC algorithm. AK-DAC has a uniform structure, and costs 1.4 additions and 1.4 doublings per scalar bit.

In Table I, we present a brief comparison of these three simultaneous double point multiplication algorithms JT-$\{\pm 1, \pm 3\}$, B-NBC, and AK-DAC. All of these three algorithms are regular, and so they are potentially resistant against power analysis attacks. However, comparing these algorithms from the efficiency point of view is not straightforward. Even though JT-$\{\pm 1, \pm 3\}$ has the best per-bit cost, B-NBC and AK-DAC have the advantage of being based on DAC. For example, in elliptic curves setting, one can implement B-NBC and AK-DAC using the addition formulas that use only the x-coordinates of the points, and that are much more efficient than their traditional counterparts. Moreover, JT-$\{\pm 1, \pm 3\}$ is not parallelizable in the sense that the double and add operations cannot be executed in parallel because an addition operation should always follow after two consecutive doubling operations. Double and add operations can be totally parallelized in both B-NBC and AK-DAC. If one deploys two parallel addition/doubling units, then the per-bit costs of B-NBC and AK-DAC becomes $1A+1D$ and $1.4A$, respectively. Similarly, if one deploys three parallel addition/doubling units, then the per-bit cost of B-NBC becomes $1A$.

In this paper, we realize hardware implementations of JT-$\{\pm 1, \pm 3\}$, B-NBC, and AK-DAC using standard Weierstrass binary elliptic curve groups, and present detailed performance comparisons with several area and time results. To the best of our knowledge, these three algorithms are some of the most promising regular algorithms with low precomputation and storage requirements, and their relative performance comparisons have not been analyzed.

The rest of the paper is organized as follows. In Section II, we review the naive method and the three algorithms JT-$\{\pm 1, \pm 3\}$, B-NBC, and AK-DAC for computing double point multiplication. In Section III, we provide hardware architectures and implement them of FPGA and compare the implementation results. Finally, we conclude the paper in Section IV.

II. A Review of Double Point Multiplication Algorithms

In this section, we review the three algorithms JT-$\{\pm 1, \pm 3\}$, B-NBC, AK-DAC, and the naive method for computing double point multiplication. We also introduce the elliptic curve equation over which we realize our implementation, and introduce some notation that we refer throughout the paper.

Let $E_{W,a,b}$ be a non-supersingular binary generic elliptic curve (short Weierstrass) defined as

$$E_{W,a,b}: \; y^2 + xy = x^3 + ax^2 + b, \tag{1}$$

where $a, b \in \mathbb{F}_{2^\ell}$, and $b \neq 0$. The set of points (x, y), $x, y \in \mathbb{F}_{2^\ell}$, that satisfy (1) together with a special point at infinity $\mathcal{O}$ (group identity) form a finite additive abelian group that we denote by $E_{W,a,b}(\mathbb{F}_{2^\ell})$. The group operation can be performed using the *chord-and-tangent* rule [13]. We have, $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E_{W,a,b}(\mathbb{F}_{2^\ell})$, and the inverse of the point $P = (x, y)$ is $-P = (x, x + y)$.

A. Traditional Scheme

The traditional scheme (in Hardware) for fast computation of double point multiplication is to employ two parallel (point multiplication) circuits to compute aP and bQ separately and add the final results together. The latency of computing double point multiplication based on this scheme is one point multiplication and a point addition (using explicit addition formulas) which requires to duplicate the hardware.

B. The JT-$\{\pm 1, \pm 3\}$ algorithm

Joye and Tunstall [15] proposed several methods of regular recoding of scalars for regular point multiplication algorithms. One of these algorithms is so called the *signed-digit recoding* algorithm that allows regular implementation of m-ary point multiplication algorithms. We represent their recoding algorithm for $m = 4$ in Algorithm 1. We should note that a typical choice

Algorithm 1 JT-$\{\pm 1, \pm 3\}$ scalar recoding algorithm

Inputs: a odd, $m = 4$
Output: $a = (a_{\ell-1}, \ldots, a_0)$ with odd $a_i \in \{\pm 1, \pm 3\}$
1: $i \leftarrow 0$
2: **While** $a > m$ **do**
3: $\quad a_i \leftarrow (a \mod 2m) - m$
4: $\quad a \leftarrow (a - a_i)/m$
5: $\quad i \leftarrow i + 1$
6: **end While**
7: $a_i \leftarrow a$

Table II. An example to compute $71P + 93Q$ using JT-$\{\pm 1, \pm 3\}$

a	1	1	-3	3
b	1	1	3	1
Point	$P + Q$	$5P + 5Q$	$17P + 23Q$	$71P + 93Q$

for m is $m = 2^k$ for some positive integer k, and the choice $k = 2$ seems to be optimal to get a competitive exponentiation algorithm with reasonable storage requirements for resource constrained applications. For example, the choice of $k = 2$ requires to store 8 group elements, whereas with the choice of $k = 3$, one has to store 32 group elements.

This recoding algorithm immediately yields a regular double point multiplication algorithm to compute $aP + bQ$, and we call this algorithm JT-$\{\pm 1, \pm 3\}$. If a and b are ℓ-bit integers, then JT-$\{\pm 1, \pm 3\}$ requires about $\ell/2$ iterations, and at each iteration a point X is updated to a point $4X + R$ for some $R \in \{\pm(P + Q), \pm(P - Q)\}$. Therefore, the per-bit cost of JT-$\{\pm 1, \pm 3\}$ is $0.5A + 1D$. For example, Algorithm 1 recodes $a = 71 = (1, 1, -3, 3)$ and $b = (1, 1, 3, 1)$, and $aP + bQ = 71P + 93Q$ is computed as in Table II.

The cost of point addition and doubling in $E_{W,a,b}(\mathbb{F}_{2^\ell})$ are $13\mathbf{M} + 4\mathbf{S} + 9\mathbf{A}$ and $5\mathbf{M} + 4\mathbf{S} + 5\mathbf{A}$, respectively [13]. Here, $\mathbf{M}$, $\mathbf{S}$, and $\mathbf{A}$, are the costs of multiplication, squaring, and addition in $\mathbb{F}_{2^\ell}$, respectively. In Lopez-Dahap coordinates [16] where one of the points is represented in affine, the cost of mixed projective point addition, i.e., $(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (x_2, y_2)$, reduces to $9\mathbf{M} + 5\mathbf{S} + 9\mathbf{A}$ [3]. The explicit formulas for point addition (PA) and point doubling (PD) are as follows [3]:

$$\text{PA}: \begin{cases} A = Y_1 + y_2 Z_1^2, B = X_1 + x_2 Z_1, C = BZ_1, \\ Z_3 = C^2, D = x_2 Z_3, X_3 = A^2 + C(A + B^2 + aC), \\ Y_3 = (D + X_3)(AC + Z_3) + (y_2 + x_2) Z_3^2 \end{cases}$$

$$\text{PD}: \begin{cases} A = X_1 Z_1, B = X_1^2, C = B + Y_1, D = AC \\ Z_3 = A^2, X_3 = C^2 + D + aZ_3 \\ Y_3 = (Z_3 + D) X_3 + B^2 Z_3. \end{cases}$$

In Fig. 1, the data dependency graph for computing point addition and point doubling are illustrated employing four and two parallel multipliers, respectively. As one can see, the total cost (latency) of computing point addition and point doubling is $3M + 13$ and $3M + 10$ clock cycles, respectively. Therefore, the cost of computing double point multiplication using JT-$\{\pm 1, \pm 3\}$ is $\approx 0.5 \times (l - 1) \times (3M + 13) + (l - 1) \times (3M + 10)$. As we noted earlier, the latency of this scheme cannot be reduced further.

C. The B-NBC algorithm

We briefly explain Bernstein's double point multiplication algorithm based on the new binary chain [6]. Let a and b be two positive integers. The new binary chain for (a, b) is computed as follows. Let $(M, N) = (a, b)$ and $D = a \mod 2$. $C_D(0, 0)$ is defined as $(0, 0), (1, 0), (0, 1), (1, -1)$. For $(M, N) \neq (0, 0)$, $C_D(M, N)$ is defined recursively:

$$\begin{aligned} C_D(M, N) = & C_d(m, n), \\ & (M + (M + 1 \text{mod} 2), N + (N + 1 \text{mod} 2)), \\ & (M + (M \text{mod} 2), N + (N \text{mod} 2)), \\ & (M + (M + D \text{mod} 2), N + (N + D + 1 \text{mod} 2)), \end{aligned}$$

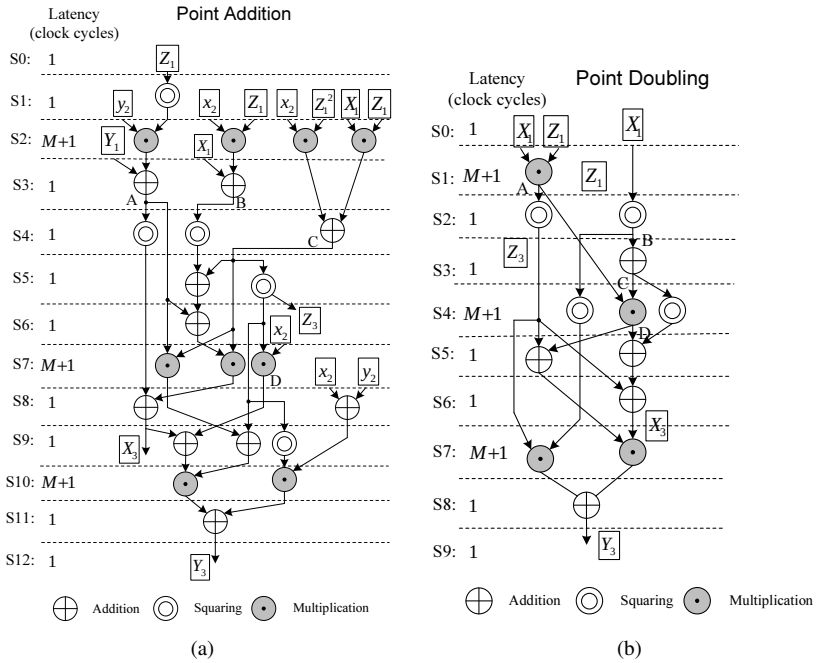


Figure 1. Data dependency graph for computing (a) point addition [5] and (b) doubling employing parallel multipliers.

where $m = \lfloor M/2 \rfloor$, $n = \lfloor N/2 \rfloor$, and

$$d = \begin{cases} 0 & \text{if } (m+M, n+N) \mod 2 = (0,1) \\ 1 & \text{if } (m+M, n+N) \mod 2 = (1,0) \\ D & \text{if } (m+M, n+N) \mod 2 = (0,0) \\ 1-D & \text{if } (m+M, n+N) \mod 2 = (1,1). \end{cases}$$

Building the new binary chain for (a, b) requires $\max(\lceil \log_2 a \rceil, \lceil \log_2 b \rceil)$ iterations, and at the each iteration three vectors are added to the sequence. Let $V_0, V_1, V_2, \ldots, V_\ell$ be the new binary chain for (a, b), where $V_0 = C_D(0, 0)$ and $V_k = v_k^{(1)}, v_k^{(2)}, v_k^{(3)}$ for $i = 1, \ldots, \ell$. Because of the correspondence between the tuple (i, j) and the group element $iP + jQ$, it will be convenient for us to call V_0 the *input,* and call V_1 the *initial state* (IS). By construction, there are six possibilities for V_1: $v_1^{(1)}$ is always $(1, 1)$, and $(v_1^{(2)}, v_1^{(3)}) \in \{((2,0),(2,1)), ((2,0),(1,0)), ((0,2),(0,1)), ((0,2),(1,2)), ((2,2),(2,1)), ((2,2),(1,2))\}$ In any case, initial state V_1 can be obtained from the input V_0 at a cost of at most 2 additions and 1 doubling. Furthermore, V_k can be obtained from V_{k-1} at a cost of 2 additions and 1 doubling for all $2 \leq k \leq \ell$. In particular, we have $v_k^{(1)} = v_{k-1}^{(1)} + v_{k-1}^{(2)}$, $v_k^{(2)} = 2v_{k-1}^{(i_k)}$, and $v_k^{(3)} = v_{k-1}^{(j_k)} + v_{k-1}^{(3)}$ for some $i_k \in \{1, 2, 3\}$ and $j_k \in \{1, 2\}$. The values of i_k and j_k can be determined easily while computing the new binary chain as follows. By construction, the parities of the vectors $v_i^{(1)}, v_i^{(2)}, v_i^{(3)}$ must be either (odd,odd), (even,even), (odd, even) or (odd,odd), (even,even), (even, odd), respectively, for all $i = 1, \ldots, \ell$. This already shows that $v_k^{(1)} = v_{k-1}^{(1)} + v_{k-1}^{(2)}$. Moreover, if the value of $v_k^{(2)}$ modulo 4 is $(0, 0)$ then $v_k^{(2)} = 2v_{k-1}^{(1)}$; if it is $(4, 4)$ then $v_k^{(2)} = 2v_{k-1}^{(2)}$; and if it is $(2, 4)$ or $(4, 2)$ then $v_k^{(2)} = 2v_{k-1}^{(3)}$. Finally, if the parities of $v_k^{(3)}$ and $v_{k-1}^{(3)}$ are the same then $v_k^{(3)} = v_{k-1}^{(2)} + v_{k-1}^{(3)}$; otherwise $v_k^{(3)} = v_{k-1}^{(1)} + v_{k-1}^{(3)}$. Therefore, the new binary chain $\{V_k\}_{k=0}^{\ell}$ can be associated with what we call the *chain sequence*

$$\text{CS} = \{(i_k, j_k)\}_{k=1}^{\ell}, i_k \in \{1, 2, 3\}, j_k \in \{1, 2\}. \tag{2}$$

It also follows from the construction of $\{V_k\}_{k=0}^{\ell}$ that when two vectors in V_{k-1} are added to obtain a vector in V_k, the difference of the vectors $v_{k-1}^{(1)} - v_{k-1}^{(2)}$ and $v_{k-1}^{(j_k)} - v_{k-1}^{(3)}$ must belong to the set $\{\pm(P+Q), \pm(P-Q)\}$ and $\{\pm P, \pm Q\}$, respectively. Therefore, the new binary chain $\{V_k\}_{k=0}^{\ell}$ can be associated with what we call the *differences sequence*

$$\text{DS} = \{((a_k, b_k), (c_k, d_k))\}_{k=1}^{\ell}, \tag{3}$$

where $a_k, b_k, c_k, d_k \in \{-1, 0, 1\}$, a_k and b_k are nonzero, exactly one of c_k and d_k is zero, and $((a_k, b_k), (c_k, d_k))$ represents $a_k P + b_k Q$ and $c_k P + d_k Q$.

Table III. An example to compute $71P + 93Q$ using $B\text{-}NBC$.

k	CS	DS	$v_{k+1}^{(1)}$	$v_{k+1}^{(2)}$	$v_{k+1}^{(3)}$
0			$P+Q$	$2P+2Q$	$P+2Q$
1	$(1,1)$	$(-1,-1),(0,-1)$	$3P+3Q$	$2P+2Q$	$2P+3Q$
2	$(3,1)$	$(1,1),(1,0)$	$5P+5Q$	$4P+6Q$	$5P+6Q$
3	$(2,2)$	$(1,-1),(-1,0)$	$9P+11Q$	$8P+12Q$	$9P+12Q$
4	$(3,1)$	$(1,-1),(0,-1)$	$17P+23Q$	$18P+24Q$	$18P+23Q$
5	$(3,2)$	$(-1,-1),(0,1)$	$35P+47Q$	$36P+46Q$	$36P+47Q$
6	$(3,1)$	$(-1,1),(-1,0)$	$71P+93Q$	$72P+94Q$	$71P+94Q$

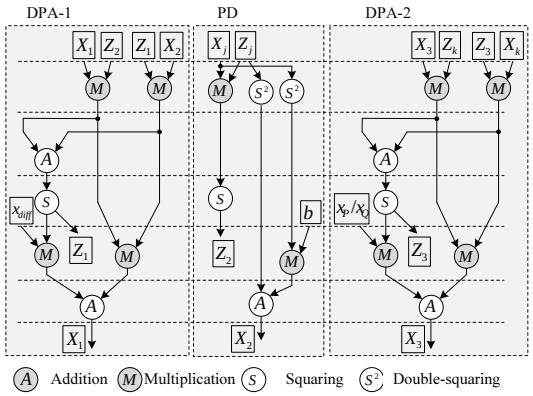


Figure 2. The data dependency graph for computing double point multiplication.

To summarize, given two positive integers a,$b \in \mathbb{Z}$ and two group elements $P, Q \in \mathbb{G}$, the new binary chain for (a, b) allows us to generate the chain sequence CS and the differences sequence DS as described in the previous paragraph. We can then compute $aP + bQ$ at a cost of $(2A + 1D) \cdot \max(\lceil \log_2 a \rceil, \lceil \log_2 b \rceil)$, where A and D represent the cost of addition and doubling in $\mathbb{G}$, respectively. The chain sequence CS specifies the input to the doubling and addition operations at each iteration. The differences sequence DS encodes the differences of the points that are the input points to the addition operations at each iteration. Note that if P and $-P$ can be identified with a same string S_P that only depends on P for all $P \in \mathbb{G}$, then the differences of the points encoded by the differences sequence during the computation of $mP + nQ$ can be identified only with S_P, S_Q, S_{P+Q}, and S_{P-Q}. Table III presents an example for computing $71P + 93Q$.

The computation of double point multiplication in $E_{W,a,b}(\mathbb{F}_{2^\ell})$ (see (1)) can be performed using differential point addition and doubling formulas. As mentioned earlier, its per-bit cost is $2A + 1D$, and one can employ two point addition circuits and one point doubling circuit in parallel to reduce the latency. The mixed projective differential point addition and doubling formulas for generic elliptic curves over $\mathbb{F}_{2^\ell}$ are defined as [16]

$$\begin{aligned} Z_{Add} &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2, \\ X_{Add} &= x \cdot Z_{Add} + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1) \end{aligned} \tag{4}$$

and

$$\begin{aligned} Z_{Dbl} &= (X_2 \cdot Z_2)^2, \\ X_{Dbl} &= X_2^4 + b \cdot Z_2^4, \end{aligned} \tag{5}$$

where $P_i = [X_i, Y_i, Z_i]$, $P_1 + P_2 = [X_{Add}, Y_{Add}, Z_{Add}]$, $2P_2 = [X_{Dbl}, Y_{Dbl}, Z_{Dbl}]$ in projective coordinates, and $P_1 - P_2 = (x, y)$. The combined point addition and doubling requires [16] $6\mathbf{M} + 5\mathbf{S} + 3\mathbf{A}$ over $\mathbb{F}_{2^\ell}$, where $\mathbf{M}$, $\mathbf{S}$, and $\mathbf{A}$, are the costs of multiplication, squaring and addition in $\mathbb{F}_{2^\ell}$, respectively. Note that in hardware the fastest possible implementation of combined point addition and doubling utilizes 3 parallel multipliers, and its latency is two multiplications.

In Fig. 2, the data dependency graph for computing two differential point additions and one point doubling in parallel is illustrated. As one can see, it requires five parallel finite field multipliers, two circuits to perform double squaring, one circuit to perform single squaring, and three adders to operate in parallel. The critical path has two field multipliers, two field adders, and one field squarer. As one can see in Fig. 2 we achieved 100% multiplier utilization employing 5 field multipliers. The differential input of DPA-1 is denoted by x_{diff} which is either $x_{(P+Q)}$ or $x_{(-P+Q)}$. Also, the differential input of DPA-2 is denoted by x_P/x_Q which could be x_P or x_Q based on the given DS sequence. As one can see, the latency of computing double point multiplication without considering coordinate conversion is about $\approx (l-1) \times (2M + 5)$ clock cycles.

D. The AK-DAC algorithm

Let a and b be two positive integers. In order to compute $aP + bQ$, $AK\text{-}DAC$ starts with the initial values $d = a$, $e = b$, $\vec{R} = (P, Q)$, $\vec{u} = (1, 0)$, $\vec{v} = (0, 1)$, and $\vec{\Delta} = (1, -1)$. We also define $R_u = \vec{u} \cdot \vec{R}$, $R_v = \vec{v} \cdot \vec{R}$, and $R_\Delta = \vec{\Delta} \cdot \vec{R}$. The initial

Algorithm 2 $AK\text{-}DAC$ double point multiplication algorithm

Inputs: $a > 0, b > 0, P, Q$
Output: $aP + bQ$
1: $d \leftarrow a$, $e \leftarrow b$, $\vec{u} \leftarrow (1, 0)$, $\vec{v} \leftarrow (0, 1)$, $\vec{\Delta} \leftarrow (1, -1)$
2: $R_u \leftarrow P$, $R_v \leftarrow Q$, $R_\Delta \leftarrow P - Q$
3: **While** $d \neq e$ **do**
4: Execute the first applicable rule in Table IV
5: **end While**
6: Using single point multiplication with input d and $(R_u + R_v)$, compute and return $d(R_u + R_v)$

Table IV. Update rules for double point multiplication

Rule	Condition	d	e	$\vec{u}$	$\vec{v}$	$\vec{\Delta}$	R_u	R_v	R_Δ
R1	$d \equiv e \pmod 2$ and $d > e$	$(d-e)/2$	e	$2\vec{u}$	$\vec{u}+\vec{v}$	$\vec{\Delta}$	$2R_u$	$R_u + R_v$	R_Δ
R1′	$d \equiv e \pmod 2$ and $d < e$	d	$(e-d)/2$	$\vec{u}+\vec{v}$	$2\vec{v}$	$\vec{\Delta}$	$R_u + R_v$	$2R_v$	R_Δ
R2	$d \equiv 0 \pmod 2$	$d/2$	e	$2\vec{u}$	$\vec{v}$	$\vec{u}+\vec{\Delta}$	$2R_u$	R_v	$R_u + R_\Delta$
R2′	$e \equiv 0 \pmod 2$	d	$e/2$	$\vec{u}$	$2\vec{v}$	$\vec{\Delta}+(-\vec{v})$	R_u	$2R_v$	$R_\Delta + (-R_v)$

values yield $R_u = P$, $R_v = Q$, $R_\Delta = R_u - R_v = P - Q$, and $dR_u + eR_v = aP + bQ$, and the values $d, e, \vec{u}, \vec{v}, \vec{\Delta}, R_u, R_v, R_\Delta$ are updated so that $dR_u + eR_v = aP + bQ$ and $R_\Delta = R_u - R_v$ hold, $d, e > 0$, and $(d+e)$ decreases until $d = e$. When $d = e$, we will have $aP + bQ = dR_u + eR_v = d(R_u + R_v)$ which can be computed using a single point multiplication algorithm with base $R_u + R_v$ and scalar d. Note that when $\gcd(a, b) = 1$, $(d + e)$ in the algorithm will decrease until $d = e = 1$ and we have $aP + bQ = d(R_u + R_v) = R_u + R_v$.

It is discussed in [4] that, if a and b are ℓ-bit integers, then $aP + bQ$ can on average be computed in about 1.4ℓ additions and 1.4ℓ doublings. Moreover addition and doubling operations can be performed using differential addition and differential doubling formulas as the difference of the group elements to be added are known by construction. We give an example in Table V to show intermediate values of Algorithm 2 with input $a = 71, b = 93, P, Q$ and $P - Q$. Note that in step 6 of Algorithm 2, we have $d = 1$, $R_u = 31P + 37Q$, $R_v = 40P + 56Q$, and the output is $R_u + R_v = 71P + 93Q$, as required.

The data dependency graph for computing double point multiplication employing four parallel multipliers, three squarers, and two adders is illustrated in Fig. 3 based on differential point addition and doubling formulae given in [22]. One should note that the difference of two points is given in projective coordinates as we need to update them at each iteration based on the conditions given in Algorithm 2. As one can see, we first perform data-flow analysis for ECC computations to understand how data has to move between the different logic and computational elements such as field multipliers, adders, and squarers. Then, we perform a latency analysis to determine where potential bottlenecks may occur and then find a balance between desired performance and the cost of implementing the design. Therefore, the latency of computing double point multiplication on binary generic curves is $\approx 1.4 \times (l - 1) \times (2M + 9)$, without considering the cost of conversion from mixed projective coordinates to affine coordinates, where M is the cost of a field multiplication.

III. Implementations of Double Point Multiplication Algorithms

A. Hardware Architectures

In this section, we propose hardware architectures for computing double point multiplication algorithms reviewed in Section II, and implement them. The hardware architectures are depicted in the Figs 4a and 4b for $B\text{-}NBC$ algorithm and $AK\text{-}DAC$

Table V. An example to compute $71P + 93Q$ using $AK\text{-}DAC$

Rule	d	e	$\vec{u}$	$\vec{v}$	$\vec{\Delta}$	R_u	R_v	R_Δ
	71	93	$(1, 0)$	$(0, 1)$	$(1, -1)$	P	Q	$P - Q$
R1′	71	11	$(1, 1)$	$(0, 2)$	$(1, -1)$	$P + Q$	$2Q$	$P - Q$
R1	30	11	$(2, 2)$	$(1, 3)$	$(1, -1)$	$2P + 2Q$	$P + 3Q$	$P - Q$
R2	15	11	$(4, 4)$	$(1, 3)$	$(3, 1)$	$4P + 4Q$	$P + 3Q$	$3P + Q$
R1	2	11	$(8, 8)$	$(5, 7)$	$(3, 1)$	$8P + 8Q$	$5P + 7Q$	$3P + Q$
R2	1	11	$(16, 16)$	$(5, 7)$	$(11, 9)$	$16P + 16Q$	$5P + 7Q$	$11P + 9Q$
R1′	1	5	$(21, 23)$	$(10, 14)$	$(11, 9)$	$21P + 23Q$	$10P + 14Q$	$11P + 9Q$
R1′	1	2	$(31, 37)$	$(20, 28)$	$(11, 9)$	$31P + 37Q$	$20P + 28Q$	$11P + 9Q$
R2′	1	1	$(31, 37)$	$(40, 56)$	$(-9, -19)$	$31P + 37Q$	$40P + 56Q$	$-9P - 19Q$

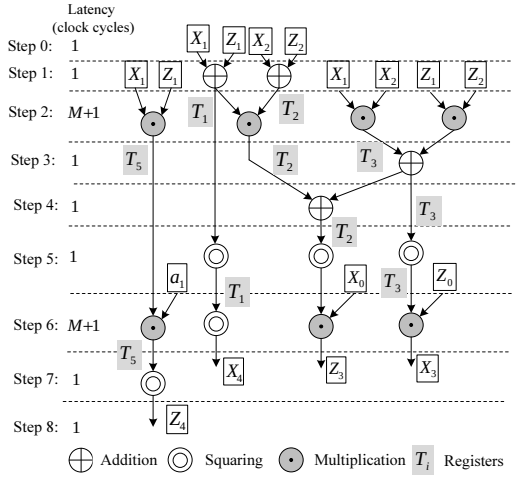


Figure 3. Data dependency graph for differential point addition and doubling on binary elliptic curves using four parallel multipliers [23].

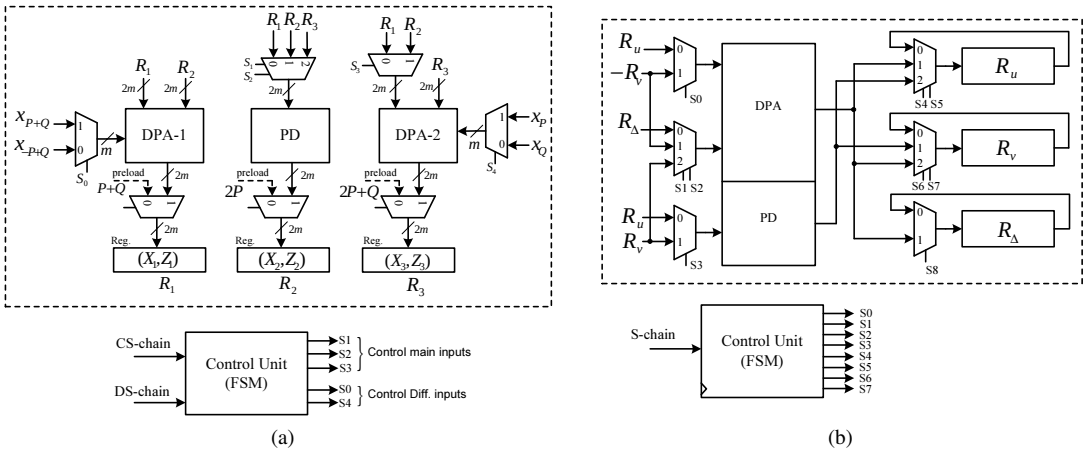


Figure 4. Hardware Architecture for computing double point multiplication based on (a) B-NBC algorithm (b) AK-DAC algorithm.

algorithm, respectively.Since the architectures of the naive method and the JT-$\{\pm 1, \pm 3\}$ algorithm are rather straightforward, we do not include them in the following.

B. Arithmetic Unit

The arithmetic unit is the main part for each architecture which is composed of $\mathbb{F}_{2^\ell}$ adders, squarers, and finite field multipliers as described in the following.

1) Addition and Squaring : Addition of two field elements, say, $A = \sum_{i=0}^{l-1} a_i \alpha^i = (a_{l-1}, \cdots, a_1, a_0)$ and $B = \sum_{i=0}^{l-1} b_i \alpha^i = (b_{m-1}, \cdots, b_1, b_0)$ in $\mathbb{F}_{2^\ell}$ represented by polynomial basis is $C = A + B$ and can be obtained by pair-wise addition of the coordinates of A and B over $\mathbb{F}_2$ (i.e., modulo 2 addition) as $c_i = a_i \oplus b_i$. Addition requires only one clock cycle to store the results in the registers.

For squaring an element $A \in \mathbb{F}_{2^\ell}$, we first simply insert zeros between each bit in the bit-vector representing A which must be followed by a reduction operation as $A^2 = \sum_{i=0}^{l-1} a_i \alpha^{2i} \bmod f(x)$. The reduction, mod $f(x)$ ($f(x)$ is a degree-l irreducible polynomial) is computed using XOR and shift operations only. Squaring is a simply hardwired permutations (inserting zeros) and requires only one clock cycle over $\mathbb{F}_{2^\ell}$ (note that the irreducible polynomial is fixed in this work).

2) Multiplication: Finite field multipliers are available in bit-level (with area complexity of $O(m)$ and time complexity of $O(m)$), digit-level (with area complexity of $O(m\delta)$ and time complexity of $O(m/\delta)$), and bit-parallel (with area complexity of $O(m^2)$ for quadratic and $O(m^{log_2 3})$ for subquadratic with time complexity of $O(1)$) architectures depending on the available resources. The digit-level polynomial basis multiplier architecture proposed in [21] is used in this work. For the binary extension field $\mathbb{F}_{2^{233}}$, recommended by NIST [1], the irreducible polynomial is $\mathbb{F}_{2^\ell}$: $\mathbb{F}_{2^{233}} = \mathbb{F}_2[x]/x^{233} + x^{74} + 1$ is a trinomial. In Fig. 5, the polynomial basis digit-level multiplier with serial-in parallel-out (SIPO) architecture is depicted. As one can see, in

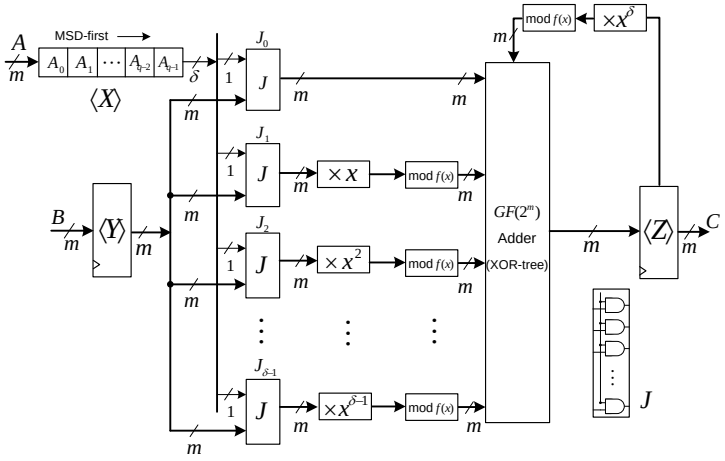


Figure 5. Then architecture of employed digit-level SIPO polynomial basis multiplier for trinomial irreducible polynomials [21].

Table VI. The FPGA implementation results of different double point multiplication algorithms over $GF(2^{233})$ on Xilinx Virtex-4.

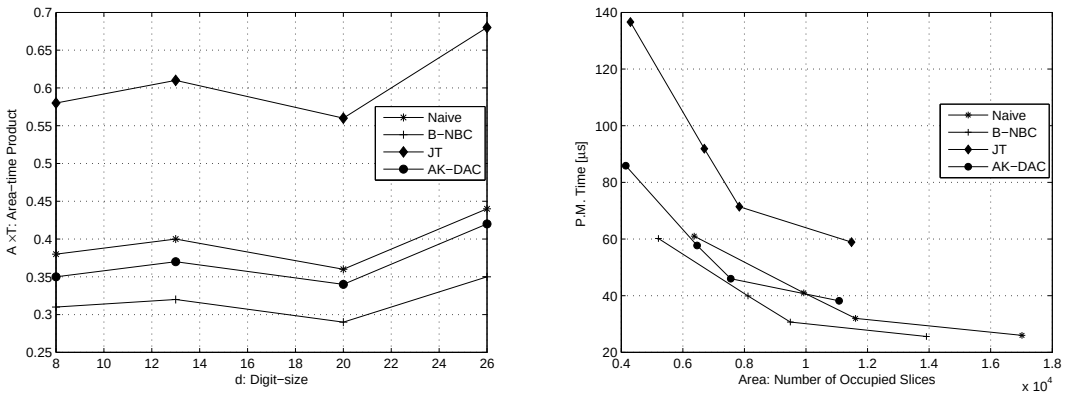
Naive Method 6 Mults. (Section II-A)							B-NBC 5 Mults. (Section II-C) [6]				
d	q	Latency [# Clock cycles]	CPD [ns]	Time [μs]	Area [# Slices]	AT Area × Time	Latency [# Clock cycles]	CPD [ns]	Time [μs]	Area [# Slices]	AT Area × Time
7	34	17,937	3.40	60.9	6,218	0.38	17,828	3.38	60.2	5,207	**0.31**
13	18	10,305	3.93	40.4	9,693	0.39	10,244	3.90	39.9	8,117	**0.32**
18	13	7,920	3.97	31.4	11,335	0.35	7,874	3.91	30.7	9,492	**0.29**
26	9	6012	4.31	25.9	16,612	0.43	5,978	4.29	25.7	13,911	**0.35**
JT-$\{\pm1, \pm3\}$ 4 Mults. (Section II-B) [15]							AK-DAC 4 Mults. (Section II-D) [4]				
7	34	40,057	3.42	136.9	4,196	0.57	25,437	3.38	85.9	**4,146**	0.35
13	18	23,145	3.98	92.1	6,541	0.60	14,884	3.88	57.7	**6,462**	0.37
18	13	17,860	4.01	71.6	7,649	0.54	11,586	3.97	45.9	**7,557**	0.34
26	9	13,632	4.33	59.1	11,210	0.66	8,947	4.28	38.2	**11,075**	0.42

each clock cycle δ coefficients of the operand A are processed having all bits of operand B available through multiplication process. In this architecture, the J blocks perform bit-wise AND operation as $a_i \odot B$. The $\times x^i$ blocks perform corresponding shift operations and are only wiring. Once the δ partial products are computed at the output of J blocks, they are multiplied by x^i, $1 \leq i \leq \delta$ and then reduced using mod $f(x)$ blocks. The $\mathbb{F}_{2^\ell}$ adder block performs addition (XOR) over $\delta+1$ l-bit field elements. Therefore, the critical-path delay of the $\mathbb{F}_{2^\ell}$ adder is $(\lceil \log_2(\delta+1) \rceil) T_X$. For multiplier operation, first the registers $\langle Y \rangle$ and $\langle Z \rangle$ are preloaded with the operand B and zero ($0 \in \mathbb{F}_{2^\ell}$), respectively. The register $\langle X \rangle$ provides in each clock cycle d bits of operand A. Then, the results of the multiplication are available after $M_q = \lceil \frac{l}{\delta} \rceil$, $1 \leq \delta \leq m$ clock cycles in the register $\langle Z \rangle$. The main advantage of this multiplier is that it operates in higher clock frequencies in comparison to the counterparts available in the literature such as Karatsuba multiplier.

3) Inversion: Inversion is the most expensive operation and can be computed using the Extended Euclidean Algorithm (EEA) or Fermat's Little Theorem (FLT) [9]. Base on FLT, one can write $A^{2^l-2} = A^{-1}$ whose computation requires $l-1$ squarings and $l-2$ multiplications as $2^l - 2 = (11 \cdots 110)_2$. However, Itoh and Tsujii (IT) [14] proposed an efficient algorithm for computing inversion over $\mathbb{F}_{2^\ell}$. The IT scheme requires $\lfloor \log_2(l-1) \rfloor + HW(l-1) - 1$ multiplications and $l-1$ squarings, where $HW(l-1)$ is the Hamming weight (number of ones) of the binary representation of $l-1$. Inversion over $\mathbb{F}_{2^{233}}$ using Itoh-Tsujii scheme [14] requires $\lfloor \log_2(l-1) \rfloor + h(l-1) - 1 = 10$ multiplications and $l-1 = 232$ squarings.

C. Control Unit and Memory

The control unit is designed with a finite state machine (FSM) based on the double point multiplication algorithms given in the previous sections. It schedules the computation tasks by generating the signals and switching the operands for arithmetic units. The intermediate results are stored in the register files. We note that the control unit is simpler and requires smaller area than the other units in the data path. Since it is implemented as a FSM, it can easily mapped into the FPGA by the synthesis tools. To store the input and output points and the intermediate results we employed a register file using flip-flops of the FPGA. Also, several multiplexers are employed to chose appropriate registers and connect to the arithmetic unit.



(a) (b)

Figure 6. Implementation results of different double point multiplication algorithms and their comparison to the counterparts in terms of (a) digit-size and area-time products and (b) area and computation time over $GF(2^{233})$ on Xilinx Virtex-4 FPGA.

D. Implementation Results and Comparisons

In this section, we implement the proposed architecture for double point multiplication in the previous sections to evaluate its area and time requirements. We have selected the Xilinx® Virtex™-4 xc4vlx200 device as the target FPGA. The proposed architecture is modeled in VHDL and synthesized for different digit sizes using XST™ of Xilinx® ISE™ version 12.1 design software. The results of implementations of double point multiplication algorithms based on the proposed hardware architectures are reported in Table VI for $l = 233$ and different digit sizes $d = \{7, 13, 18, 26\}$. As shown in this table, we provided the latency (number of clock cycles), total time of computation, critical path delay (CPD), occupied area (number of slices) and area-time products. The Naive method requires largest area in comparison to the other schemes and AK-DAC scheme requires the smallest area. The B-NBC scheme provides fastest results and best area-time trade-offs in comparison to the counterparts. The JT-$\{\pm 1, \pm 3\}$ is the slowest method and is not efficient in terms of time-area trade-offs. In Fig. 6, we plotted the area, time, and area-time results in terms of the digit-size of different scheme for more clarification.

In Fig. 6, implementation results of different double point multiplication algorithms and its comparison to the traditional method is illustrated. In Fig. 6a, we plot area-time products in terms of the digit-size and in Fig. 6b the area given by number of occupied slices is plotted in terms of time of computing double point multiplication.

IV. Conclusion

In this paper, efficient implementation of double point multiplication over binary elliptic curves is presented. We provide a comprehensive analysis and comparison of double point multiplication algorithms based on differential addition chains, binary double and add method, and naive method. We investigate the performance and efficiency of these schemes based on the required area and time of computation. Our results indicate that the differential addition chain based schemes are the most suitable schemes for computing double point multiplication. For instance, we show that the scheme proposed in [6] provides the fastest double point multiplication, and the one presented in [4] requires the smallest silicon area for simultaneous computation.

References

[1] National Institute of Standards and Technology. *Digital Signature Standard*, 186-2, January 2000.

[2] T. Akishita. Fast Simultaneous Scalar Multiplication on Elliptic Curve with Montgomery Form. *Selected Areas in Computer Science SAC 2001, LNCS*, 2259:225–267, 2001.

[3] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman. A New Addition Formula for Elliptic Curves Over $GF(2^m)$. *IEEE Transactions on Computers*, 51(8):972–975, 2002.

[4] R. Azarderakhsh and K. Karabina. A New Double Point Multiplication Algorithm and its Application to Binary Elliptic Curves with Endomorphisms. *IEEE Transactions on Computers*, to appear:pp, 2013.

[5] Reza Azarderakhsh and Arash Reyhani-Masoleh. High-Performance Implementation of Point Multiplication on Koblitz Curves. *IEEE Trans. on Circuits and Systems*, 60-II(1):41–45, 2013.

[6] D. Bernstein. Differential addition chains. Technical report, 2006. Available at http://cr.yp.to/ecdh/diffchain-20060219.pdf.

[7] C. Clavier and M. Joye. Universal Exponentiation Algorithm – A First Step towards *Provable* SPA-Resistance. *Lecture Notes in Computer Science, CHES 2001*, 2162:300–308, 2001.

[8] J-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. *Lecture Notes in Computer Science, CHES 1999*, 1717:292–302, 1999.

[9] K. Fong, D. Hankerson, J. López, and A. Menezes. Field inversion and point halving revisited. *IEEE Transactions on Computers*, pages 1047–1059, 2004.

[10] D. Galbraith, X. Lin, and M. Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. *Journal of Cryptology*, 24:446–469, 2011.

[11] R. Gallant, R. Lambert, and S. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. *Advances in Cryptology - CRYPTO 2011, LNCS*, 2139:190–200, 2001.
[12] D. Hankerson, K. Karabina, and A. Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. *IEEE Transactions on Computers*, 58:1411–1420, 2009.
[13] D.R. Hankerson, S.A. Vanstone, and A.J. Menezes. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York Inc, 2004.
[14] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Information Computing*, 78(3):171–177, 1988.
[15] M. Joye and M. Tunstall. Exponent recoding and regular exponentiation algorithms. *Lecture Notes in Computer Science, AFRICACRYPT 2009*, 5580:334–349, 2009.
[16] J. López and R. Dahab. Fast Multiplication on Elliptic Curves Over $GF(2^m)$ Without Precomputation. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*, pages 316–327, 1999.
[17] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. New York, 1996.
[18] B. Möller. Algorithms for Multi-exponentiation. *Selected Areas in Computer Science SAC 2001, LNCS*, 2259:165–180, 2001.
[19] P. Montgomery. Evaluating recurrences of form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas chains. www.cwi.nl/ftp/pmontgom/Lucas.ps.gz, December 13, 1983; Revised March, 1991 and January, 1992.
[20] J.M. Pollard. Monte Carlo Methods for Index Computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978.
[21] Chang Shu, Soonhak Kwon, and Kris Gaj. Reconfigurable Computing Approach for Tate Pairing Cryptosystems over Binary Fields. *IEEE Transactions on Computers*, 58(9):1221–1237, 2009.
[22] M. Stam. On Montgomery-Like Representations for Elliptic Curves over $GF(2^k)$. *(PKC 2003)*, pages 240–253.
[23] M. Stam. On Montgomery-like Representations for Elliptic Curves Over $GF(2^k)$. In *Proceedings of The 3rd International Conference on Practice and Theory of Public Key Cryptography (PKC 2003)*, pages 240–254, 2003.