5-2016

Low-Resource and Fast Elliptic Curve Implementations over Binary Edwards Curves

Brian Koziel
bck6520@rit.edu

Low-Resource and Fast Elliptic Curve Implementations over Binary Edwards Curves

by

Brian Koziel

A Thesis Proposal Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Reza Azarderakhsh
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology, Rochester New York, 14623
May 2016

Approved By:

Reza Azarderakhsh
Assistant Professor, Department of Computer Engineering
Primary Advisor

Amlan Ganguly
Assistant Professor, Department of Computer Engineering
Secondary Advisor

Mehran Mozaffari Kermani
Assistant Professor, Department of Electrical Engineering
Secondary Advisor

I would like to dedicate this thesis to my family: Karen, Mark, Eric, and Allison. Their love and support have been of the utmost value to my development and success.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text. This dissertation contains less than 65,000 words including appendices, bibliography, footnotes, tables and equations and has less than 150 figures.

Brian Koziel
2016

Acknowledgements

I would like to thank my supervisor Dr. Reza Azarderakhsh for his guidance throughout my research experience at RIT. I would like to thank my readers Dr. Marcin Łukowiak and Dr. Mehran Mozaffari Kermani for their helpful feedback.

Abstract

Elliptic curve cryptography (ECC) is an ideal choice for low-resource applications because it provides the same level of security with smaller key sizes than other existing public key encryption schemes. For low-resource applications, designing efficient functional units for elliptic curve computations over binary fields results in an effective platform for an embedded co-processor. This thesis investigates co-processor designs for area-constrained devices. Particularly, we discuss an implementation utilizing state of the art binary Edwards curve equations over mixed point addition and doubling. The binary Edwards curve offers the security advantage that it is complete and is, therefore, immune to the exceptional points attack. In conjunction with Montgomery ladder, such a curve is naturally immune to most types of simple power and timing attacks. The recently presented formulas for mixed point addition in K. Kim, C. Lee, and C. Negre [14] were found to be invalid, but were corrected such that the speed and register usage were maintained. We utilize corrected mixed point addition and doubling formulas to achieve a secure, but still fast implementation of a point multiplication on binary Edwards curves. Our synthesis results over NIST recommended fields for ECC indicate that the proposed co-processor requires about 50% fewer clock cycles for point multiplication and occupies a similar silicon area when compared to the most recent in literature.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An extremely small security coprocessor is necessary for applications such as the Internet of Things or smart cards to ensure that only valid users have access to the device. Today's computer market is dominated by embedded devices. These embedded devices are branching out and linking with other electronics wirelessly. Since the functionality and connectivity of these embedded system's consume most of the system's area and power, the security coprocessor must be tiny, fast, and power-efficient, but still provide the security required for the device.

Public key encryption uses a public and a private key to encrypt information. The public key is known to the general public, while the private key is stored securely. Two popular public key encryption schemes are Elliptic Curve Cryptography (ECC) and Rivest, Shamir, Adleman (RSA) encryption. ECC relies on point multiplications on an elliptic curve and RSA encryption relies on modular exponentiation of extremely large numbers. The National Institute of Standards and Technology (NIST) has recommended key sizes for ECC and RSA based on the equivalent security in private encryption schemes.

Private key encryption schemes, such Advanced Encryption Standard or Data Encryption Standard, are not a suitable solution to this problem because the key is stored on the device and the security implementation requires much more hardware per bit than public key encryption schemes.

From NIST's recommendation for key sizes, ECC uses a fraction of the amount of bits that RSA uses. One reason for this is that excellent factoring algorithms, such as the Index Calculus method exist for efficiently factoring extremely large primes to break RSA. There are more computations for point arithmetic on an elliptic curve, but the significantly fewer bits required for ECC makes it a prime choice for extremely constrained devices. ECC provides key exchange ECDH, authentication ECDSA, and encryption ECIES protocols.

An elliptic curve is composed of all points that satisfy an elliptic curve equation as well

as a point at infinity. This forms an Abelian group, E, over addition, where the point at infinity represents the zero element or identity of the group. The most basic operations over this Abelian group are point addition and point doubling. Using a double-and-add method, a point multiplication, $Q = kP$, where $k \in \mathbb{Z}$ and $Q, P \in E$, can be computed quickly and efficiently. Protocols implemented over ECC rely on the difficulty to solve the elliptic curve discrete logarithm problem (ECDLP), that given Q and P in $Q = kP$, it is infeasible to solve for k [11]. For the computations of ECC, several parameters should be considered including representation of field elements and underlying curve, choosing point addition and doubling method, selecting coordinate systems such as affine, projective, Jacobian, and mixed, and finally arithmetic (addition, inversion, multiplication, squaring) on finite field. Field multiplication determines the efficiency of point multiplication on elliptic curves as its computation is complex and point multiplication requires many field multiplications. IEEE and NIST recommended the usage of both binary and prime fields for the computation of ECC [12, 34]. However, in hardware implementations and more specifically for area-constrained applications, binary fields outperform prime fields, as shown in [7]. Therefore, a lot of research in the literature has been focused on investigating the efficiency of computing point multiplication on elliptic curves over binary fields. For instance, one can refer to [28], [32], [38] and [15] to name a few, covering a wide variety of cases including different curve forms, e.g., generic and Edwards, and different coordinate systems, e.g., affine, projective, and mixed. The formulas for point addition and point doubling can be determined by using geometric properties. In [4], binary Edwards curves are presented for the first time for ECC and their low-resource implementations appeared in [15]. It has been shown that a binary Edwards curves (BEC) is isomorphic to a general elliptic curve if the singularities are resolved [4]. Based on the implementations provided in [15], it has been observed that their implementations are not as efficient as other standardized curves. Recently, in [14], the authors revisited the original equations for point addition and doubling and provided competitive formulas. We observed that the revisited formulas for mixed point addition in [14] are invalid. After modifying their formulas, we employed them for the computation of point multiplication using a mixed coordinate system and proposed an efficient crypto-processor for low-resource devices. The main contributions of this thesis can be summarized as follows:

- We propose an efficient hardware architecture for point multiplication on binary Edwards curves. We employed Gaussian normal basis (GNB) for representing field elements and curves as the computation of squaring, inversion, and trace function can be done very efficiently over GNB in hardware.

- We modified and corrected the w-coordinates differential point addition formulas presented in [14]. We provide explicit formulas over binary Edwards curves that maintain the speed and register usage provided in [14] and employed the formulas in steps on the Montgomery Ladder [22]. This is the first time this double-and-add algorithm has appeared in literature. This implementation was competitive with many of the area-efficient elliptic curve crypto-processors found in literature, but adds the additional security benefit of completeness.
- We implemented and synthesized our proposed algorithms and architectures for the computation of point multiplication on binary Edwards curves and compared our results to the leading ones available in the literature.

This thesis is organized as follows. In Chapter 2, preliminaries necessary for elliptic curve cryptography are reviewed. In Chapter 3, the binary Edwards curve is introduced and proper mixed coordinate addition formulas are presented. Chapter 4 details the area-efficient architecture used for this ECC co-processor and compares this work to other ECC crypto-processors in terms of area, latency, computation time, and innate security. Chapter 5 concludes the thesis with takeaways and the future of area efficient implementations of point multiplication.

Chapter 2

Elliptic Curve Cryptography

In this chapter, we review elliptic curve cryptography principles. Cryptography is about transmitting messages between parties, in the threat of malicious third-parties. Public-key cryptography relies on the scenario where two parties relay information over an insecure public channel. There are public keys that are shared with the world and private keys that are known only to the party. Typically complex mathematical problems are used as a basis for public-key cryptography. ECC is one such problem based on the elliptic curve discrete log problem.

2.1 Elliptic Curve Theory

The most basic elliptic curves follow the formula in Equation 2.1. This is form is called the short Weierstrass form of elliptic curves. The curve is defined as the set of points, (x,y), that satisfy Equation 2.1 along with a special point $\mathbb{O}$ called the point at infinity. Elliptic curve cryptosystems follow a point addition operation that forms the set of all points into an abelian group over addition. The addition changes depending on the operands for four different addition cases.

$$y^2 = x^3 + ax + b \tag{2.1}$$

The first case deals with two points $P(x_1, y_1)$ and $Q(x_2, y_2)$ that have different x values. In this case, point addition is defined as the line that goes through P and Q. Since the curve has degree 3, the line will intersect the curve at a third point, -R $(x_3, -y_3)$. The solution is R, which flips the coordinate of the intersection of P and Q across the x-axis. The point addition is finished by inverting the y coordinate. Using the slope between the two points and the original equation for the elliptic curve,

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

The second case deals with two points that have the same x coordinate, but opposite y coordinates, so $P(x,y)$ and $-P(x,-y)$. The line through these two points is a vertical line that does not intersect at a third point, so the addition of these two points results in the point at infinity, $\mathbb{O}$.

The third case deals with the addition between the same point on a curve, $P(x_1, y_1)$. The line in this case is the tangent to at the x coordinate. This will intersect at only one other point since the degree is 3. Similar to the first case, the point where this tangent line intersects is $-2P(x_2, -y_2)$, so the sum of the double point is $2P(x_2, y_2)$. As The same formula to find the points can be used after finding the slope. Using the derivative to find the slope,

$$\lambda = \frac{3x_1^2 + a}{2y_1}$$

$$x_2 = \lambda^2 - x_1 - x_2$$

$$y_2 = \lambda(x_1 - x_3) - y_1$$

The final case deals with any addition with the point at infinity. The point at infinity is the identity for point addition, so anything added to the point at infinity is itself.

The two major point additions are point addition and point doubling. Fig. 2.1 demonstrates point addition and doubling.

These operations are used to perform point doubling and point addition for point multiplication. Multiplication uses a double-and-add formula to efficiently reach very high multiples of a point.

$$kP = P + P + \ldots + P$$

2.2 Binary Fields

These geometric examples work nicely since there are an infinite amount of points on the curve. However, cryptographic systems operate in binary Galois fields, GF(2^m), or large prime Galois Fields, GF(p). These fields have a finite number of elements, which is friendlier for

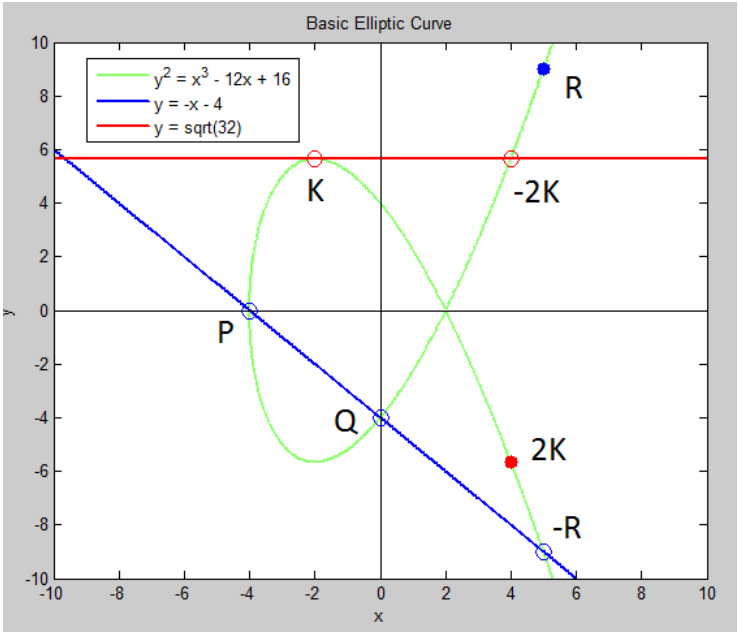

Figure 2.1: Point addition and doubling geometric example.

cryptographic applications. The equations used above can be applied to elliptic curves defined over Galois fields, even though the representation changes. The following equations show the new formulas for lambda. It is exactly the same, but the addition, multiplication, and inversion for the Galois field representation changes.

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

$$\lambda = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} & \text{if } P \neq Q \\ (3x_1^2 + a)(2y_1)^{-1} & \text{if } P = Q \end{cases}$$

Formally, a Galois field is defined as a field with a finite set of field elements that operate under the the two operations of addition and multiplication. Addition and multiplication between any two field elements produce an element in the field. We define a set G and a binary operation $\bullet$ to form a group $(G, \bullet)$ if they satisfy the following properties:

1. The operation $\bullet$ is closed (i.e., $a \bullet b \in G$ for all $a, b \in G$).

2. The operation $\bullet$ is associative (i.e., $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ for all $a, b, c \in G$).

3. The operation $\bullet$ is commutative (i.e., $a \bullet b = b \bullet a$ for all $a, b \in G$). In this case set $(G, \bullet)$

called Abelian.

4. There exists an identity element $i \in G$ such that $i \bullet a = a \bullet i = a$ for all $a \in G$.

5. For every $a \in G$, there exists an inverse element $b \in G$ such that $a \bullet b = i$.

We define the group $(G, \bullet)$ with the group operation as multiplication $\times$ to be the multiplicative group. It is easy to see that the identity element of the multiplicative group is 1 an inverse element is denoted as $a^{-1} \in G$. Similarly, the group with the group operation as addition $+$ is the additive group. The identity element of the additive group is 0 and an inverse element is denoted as $-a \in G$. The order of a group is the total number of all elements in the group. For groups of a finite size, the order of an element is the smallest positive integer, n, for which $a^n = i$. We define a group to be cyclic if all members of the group can be generated by applying group operations repeatedly to an element a. For this group, a is a generator of the group since all elements can be generated from a as a starting element.

A field $\mathbb{F}$ is a set of elements that operate under the operators addition and multiplication. A field demonstrates the following properties:

1. $\mathbb{F}$ is an abelian group with respect to addition.

2. $\mathbb{F}^*$, the field without the zero element, is an abelian group with respect to multiplication.

3. The multiplication operation is distributive (i.e., $a \times (b+c) = (a \times b) + (a \times c)$ and $(b+c) \times a = (b \times a) + (c \times a)$ for all $a, b, c \in \mathbb{F}$.).

We denote a Galois field, or finite field, with q elements as $\mathbb{F}_q$ or GF(q). If q is prime or q is a power of a prime, then the order of the group is the total number of elements in $\mathbb{F}_q$. Let $q = p^m$ for $m \geq 1$. A finite field is called a prime field if $m = 1$. A finite field is called an extension field if $m > 1$.

Fig. 2.2 represents the pyramid from which the ECC protocols are built. The Galois field operations are at the foundation.

Area-efficient ECC implementations require a highly efficient Galois field functional unit. Binary fields, $GF(2^m)$, are efficient primarily because addition is a simple XOR between bits of the two inputs and there is less logical expressions for operations. The main operations necessary for ECC in binary fields are addition, multiplication, squaring, and inversion.

The useful property from converting to Galois fields is that it creates a cyclic abelian group over addition. For an elliptic curve defined over GF(2^m), for instance, there are approximately 2^m points on the curve based on the Hasse bounds. A generator of the group cycles through all 2^m points when continuously adding itself to the point at infinity. The cycle is also seemingly

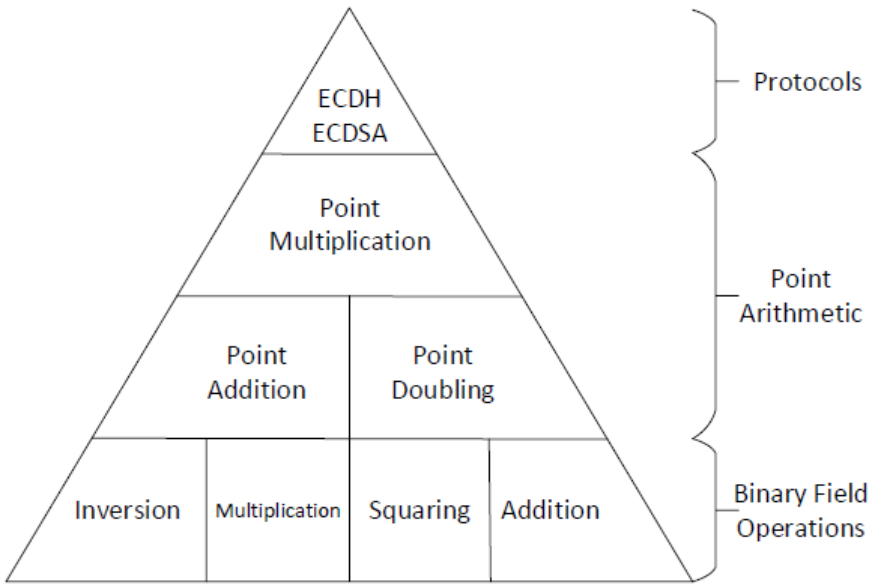

Figure 2.2: Elliptic curve cryptography computational pyramid.

random, which introduces the idea that for a generator point P and random integer k, and $Q = kP = P + P + \cdots + P$, it is infeasible to determine k without testing all possible combinations of k. For an extremely large underlying Galois field, it is infeasible to determine k. This is called the elliptic curve discrete log problem. The security of elliptic curve protocols are based on this property.

2.2.1 Polynomial Basis

One possible basis for binary fields is the polynomial basis. Consider an element x to be a root of the primitive polynomial of degree m. We define the polynomial basis as the set $\{1, x, x^2, \cdots, x^{m-1}\}$. An element A is defined as the linear combination of the elements in the polynomial basis, $A = \sum_{i=0}^{m-1} a_i x^i$, where $a_i \in \{0, 1\}$. Thus, each part of the linear combination is defined over $GF(2)$. Using bits as digits in today's language of computers, we require a single bit for each a_i. A vector of m bits represents an entire element in $GF(2^m)$. The identity element of addition is 0 and the identity element of multiplication is 0.

2.2.2 Gaussian Normal Basis

Another possible basis for binary fields is the normal basis, for which Gaussian Normal Basis (GNB) is a special form. Similar to polynomial basis, there is a normal basis for all binary

fields. The normal basis is defined over a normal element $\beta \in GF(2^m)$, where β is a root of an irreducible polynomial of degree m. The set $N = \{\beta, \beta^2, \cdots, \beta^{2^{m-1}}\}$ is a basis for $GF(2^m)$ and each of its elements are linearly independent. Similar to polynomial basis, we represent the basis with linearly independent elements in $GF(2^m)$. Thus, an element A is defined in normal basis as $A = \sum_{i=0}^{m-1} a_i \beta^{2^i}$, where $a_i \in GF(2)$. The identity element of the normal basis is again 0, but the identity element of multiplication is $1 = \beta + \beta^2 + \beta^{2^2} + \cdots + \beta^{2^{m-1}}$.

Definition 1. [2, 20] Let m and T be positive integers such that $p = mT + 1$ be a prime number and $\gcd\left(\frac{mT}{k}, m\right) = 1$, where k is the multiplication order of 2 modulo p. Let α be a primitive $mT + 1$-th root of unity in $\mathbb{F}_{2^{Tm}}$. Then, for any primitive T-th root of unity τ in $\mathbb{Z}_p$, $\beta = \sum_{i=0}^{T-1} \alpha^{\tau^i}$ generates a normal basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$ given by $N = \{\beta, \beta^2, \cdots, \beta^{2^{m-1}}\}$, which is called a Gaussian normal basis of type T.

GNB is a special case of a normal basis. Specifically, GNB exists whenever m is not divisible by 8. This is the case for the NIST standardized binary fields [34], which is explained in Section 4.1.2. Moreover, all standardized binary fields are odd, so that ensures that the GNB type is even. Notably, if the binary field is GNB type 1 or 2, then it is considered an optimal normal basis [23]. Generally, smaller type GNB's have simpler multiplication complexity, which will be mentioned in Section 2.3.2.

2.3 Binary Field Arithmetic

Here, this thesis reviews the basic field operations in a binary field. Specifically, the operations of addition, multiplication, squaring, inversion, and the half-trace are reviewed with their implementation in polynomial basis or GNB.

2.3.1 Finite-Field Addition

Finite-field addition performs $A + B = C$, where $A, B, C \in \mathbb{F}_{2^m}$. Thus, this is a simple addition of each linearly independent digit. Each digit is represented as a single bit and there are no carries. Therefore, addition can be represented as a bit-wise XOR, as illustrated by Eq. 2.2. In both polynomial basis and GNB, addition is identical.

$$C = A + B$$

$$C = \sum_{i=0}^{m-1} (a_i + b_i) x^i \quad c_i \in 0, 1$$

$$c_i = a_i \oplus b_i \tag{2.2}$$

2.3.2 Finite-Field Multiplication

Finite-field multiplication performs $A \times B = C$, where $A, B, C \in \mathbb{F}_p$. This equates to a regular multiplication of A and B to produce a third element C. However, if elements A and B are both m-bits, then the result, C, is $2m$-bits. A reduction must be made so that the result is still within the field.

For polynomial basis, multiplication is the sum of bit by bit multiplications, similar to a Schoolbook form of multiplication. Since some multiplication terms go beyond the max digit in the polynomial basis, the product is brought back down by computing the modulus, F(x). This modulus is a is irreducible in $GF(2^m)$. The smaller multiplications can be done in a bit-serial ($d = 1$) fashion and a digit-serial ($d > 1$) fashion. The bit-serial form is shown in Eq. 2.3. Bit-serial multipliers require many fewer gates to represent than digit-serial multipliers, but at the cost of many more iterations for a single multiplication.

$$C = A \times B \bmod \mathrm{F}(x)$$

$$C = A \times (\sum_{i=0}^{m-1} b_i x^i) \bmod \mathrm{F}(x)$$

$$C = a_0 \times (\sum_{i=0}^{m-1} b_i x^i) + a_1 \times x \times (\sum_{i=0}^{m-1} b_i x^i) + \ldots + a_{m-1} \times x^{m-1} \times (\sum_{i=0}^{m-1} b_i x^i)) \bmod \mathrm{F}(x) \tag{2.3}$$

Multiplication in Gaussian normal basis is much more complex. Consider

$$A = (a_0, a_1, \cdots, a_{m-1}) = \sum_{i=0}^{m-1} a_i \beta^{2^i}$$

$$B = (b_0, b_1, \cdots, b_{m-1}) = \sum_{j=0}^{m-1} b_j \beta^{2^j}$$

$$C = AB = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{2^i + 2^j}$$

Each product of elements in GNB produce a term $\beta^{2^i+2^j} = \sum_{l=0}^{m-1} u_{i,j}^l \beta^{2^l}$, where i is the term in the first operand and j is the term in the second operand. The l-th digit of C can be recovered

by utilizing the relationship $c_l = aM^{(l)}b^{tr}$, where M is a multiplication matrix and tr is the transpose operation. The first row of the multiplication table always has a single '1' on the second column of the first row. After that, each following row will have as many '1's as the type of the basis. Generally, resolving the products with the multiplication table involves cyclic left shifts of inputs A and B. The explicit formulas given in [12, 29] are shown in Eq. 2.4 and Eq. 2.5.

$$C = (A \odot (B \ll 1)) \oplus \sum_{i=1}^{m-1} (A \ll m-i) \odot S(i,B), \tag{2.4}$$

$$S(i,B) = ((B \ll R(i,1)) \oplus (B \ll R(i,2)) \oplus \cdots \oplus (B \ll R(i,T))), 1 \leq i \leq m-1. \tag{2.5}$$

In these equations, $(X \ll i)$ is the i-fold left cyclic shift of $X \in GF(2^m)$. $\odot$ and $\oplus$ denote bit-wise AND and XOR operations between coordinates of X and Y, respectively.

2.3.3 Finite-Field Squaring

Finite-field squaring performs $A \cdot A = C$, where $A, C \in \mathbb{F}_p$. This is a simpler example of multiplication. Since the input operands are the same, the arithmetic can be optimized further over the multiplication case.

For a binary field defined over polynomial basis, the product of an element squared is the same as the original element, with '0's inserted between each element. For instance, 1011×1011 = 1000101. Thus, efficient squaring can be broken down into expansion and reduce stages. The expansion stage involves inserting the '0' between each bit and the reduction involves using the irreducible modulus to keep the result in the basis. Otherwise, bit-parallel squaring units can be implemented efficiently based on the irreducible modulus.

For GNB, squaring an element involves a right cyclic shift, as shown in Eq. 2.6. In hardware, squaring an element comes only at the cost of rewiring.

$$A^2 = (\sum_{i=0}^{m-1} a_i \beta^{2^i})^2 = A \gg 1 \tag{2.6}$$

2.3.4 Finite-Field Inversion

Finite-field inversion finds some A^{-1} such that $A \cdot A^{-1} = 1$, where $A, A^{-1} \in \mathbb{F}_q$. There are many schemes to perform this efficiently. Fermat's little theorem exponentiates $A^{-1} = A^{q-2}$. This requires many multiplications and squarings, but is a constant set of operations. The Extended Euclidean Algorithm (EEA) has a significantly lower time complexity of $O(\log^2 n)$ compared to $O(\log^3 n)$ for Fermat's little theorem. EEA uses a greatest common divisor algorithm to compute the modular inverse of elements a and b with respect to each other, $ax + by = \gcd(a,b)$. Unfortunately, EEA utilizes a non-constant set of operations that risks leaking important information related to timing and power. Such information can be used by an outside third party to discover bits of the key and break the cryptosystem. Further, the use of multiple addition and subtraction units in EEA require more hardware.

Inversion over binary fields utilize the exponentiation A^{2^m-2}. Addition chains are a method to efficiently exponentiate these large values as a base. Using multiplications and squarings, the large exponential can be broken into a chain of operations with $log_2(m)$ complexity. The most basic exponentiation is the binary method, which executes a square-and-multiply method. For an m-bit element in $\mathbb{F}_{2^m}$, there are approximately $m-1$ squarings and $m-2$ multiplications.

The complexity of the exponentiation can be further reduced by using the Itoh-Tsujii method [13]. This method exponentiates along $2^k - 1$ chains, where k is an integer, to $2^{m-1} - 1$, and then a final squaring to $2^{m-2} - 2$. An example of the Itoh-Tsujii method for $2^{283} - 2$ is shown in Table 2.1 (adapted from [31]). Itoh-Tsujii reduces the complexity of the exponentiation to $m-1$ squarings and $H(m-1)$ multiplications, , where H represents the Hamming weight. $H(m-1)$ is much smaller than $m-2$ because there are $\log_2 m - 1$ bits in the representation for $m-1$. Even if all bits are set in $m-1$, there are still significantly fewer multiplications. The only caveat to using the Itoh-Tsujii method is that it requires an additional temporary value to perform the exponentiation.

2.3.5 Finite-Field Half-Trace

Finite-field half-trace solves the quadratic equation $X^2 + X = A$, for $X = (x_0, x_1, \cdots, x_{m-1}) \in GF(2^m)$. This only has a solution if the trace function, $\mathrm{Tr}(A) = 0$. Further, if a solution exists, then both X and $X+1$ are solutions. For normal basis, when m is odd, the trace of element A can be computed as $\mathrm{Tr}(A) = \sum_{i=0}^{m-1} a_i$, which is bit-wise XOR operation of all bits of vector A. The solution X can be found from the bit-wise XOR algorithm shown in Algorithm 2.1. However, in polynomial basis, $m-1$ squarings and $(m-1)/2$ additions are required.

Table 2.1: Itoh-Tsujii inversion for GF(2^{283})

i	u_i	Binary	Rule	Multiplication	Current
0	1	1		A	A^{2^1-1}
1	2	10	$2u_0$	$(A^{2^1-1})^{2^1}A^{2^1-1}$	A^{2^2-1}
2	4	100	$2u_1$	$(A^{2^2-1})^{2^2}A^{2^2-1}$	A^{2^4-1}
3	8	1000	$2u_2$	$(A^{2^4-1})^{2^4}A^{2^4-1}$	A^{2^8-1}
4	16	1000	$2u_3$	$(A^{2^8-1})^{2^8}A^{2^8-1}$	$A^{2^{16}-1}$
5	17	10001	u_4+u_0	$(A^{2^{16}-1})^{2^1}A^{2^{16}-1}$	$A^{2^{17}-1}$
6	34	100010	$2u_5$	$(A^{2^{17}-1})^{2^{17}}A^{2^{17}-1}$	$A^{2^{34}-1}$
7	35	100011	u_6+u_0	$(A^{2^{34}-1})^{2^1}A^{2^{34}-1}$	$A^{2^{35}-1}$
8	70	1000110	$2u_7$	$(A^{2^{35}-1})^{2^{35}}A^{2^{35}-1}$	$A^{2^{70}-1}$
9	140	10001100	$2u_8$	$(A^{2^{70}-1})^{2^{70}}A^{2^{70}-1}$	$A^{2^{140}-1}$
10	141	10001101	u_9+u_0	$(A^{2^{140}-1})^{2^1}A^{2^{140}-1}$	$A^{2^{141}-1}$
11	282	100011010	$2u_{10}$	$(A^{2^{141}-1})^{2^{141}}A^{2^{141}-1}$	$A^{2^{282}-1}$

Algorithm 2.1 Solving a quadratic equation $X^2+X=A$ using GNB [11].

Input: $A=X^2+X$, where $A, X^2+X \in GF(2^m)$.
Output: $X \in GF(2^m)$, iff $\mathrm{Tr}(A)=0$.
1. $x_0 \leftarrow a_0$.
2. **For** i from 1 to $m-2$ **do**
3. $x_i \leftarrow a_i \oplus x_{i-1}$.
4. **end for**
5. $x_{m-1} \leftarrow 0$.
6. **Return** X.

Algorithm 2.2 demonstrates how the half-trace can be computed in polynomial basis, which also works in GNB.

2.4 Applications

Two major cryptographic protocols that are based on point multiplication in ECC are the elliptic curve Diffie-Hellman key-exchange and the elliptic curve digital signature algorithm.

Algorithm 2.2 Solving a quadratic equation $X^2+X=A$ using polynomial basis [15].

Input: $A=X^2+X$, where $A, X^2+X \in GF(2^m)$.
Output: $X \in GF(2^m)$, iff $\mathrm{Tr}(A)=0$.
1. $X=0$
2. **For** i from 1 to $\lfloor \frac{m}{2} \rfloor - 1$ **do**
   3. $X = X + A^{2^{2i+1}}$.
4. **end for**
5. **Return** X.

Algorithm 2.3 Elliptic curve Diffie-Hellman key exchange

Input: Prime p,
Elliptic curve $E: y^2 \equiv x^3+ax+b \bmod p$
Primitive point on E, $P=(x_p, y_p)$
Alice's private key k_1
Bob's private key k_2
Output: Shared secret point $Q=k_1k_2P$
1. Alice performs the point multiplication $R=k_1P$
2. Bob performs the point multiplication $S=k_2P$
3. Alice and Bob exchange their point multiplication results R and S over a public channel
4. Alice performs $Q=k_1S=k_1k_2P$
5. Bob performs $Q=k_2R=k_1k_2P$
6. Alice and Bob acquire the secret shared point Q

2.4.1 Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman (ECDH) uses two different multiplications to give two parties the same unique key. Let Alice and Bob be the two parties. Each have their own special keys, k_1 and k_2, respectively. They exchange knowledge of an elliptic curve system over a public channel and decide on a point P. Alice performs $Q=k_1P$ and Bob performs $R=k_2P$. Alice sends her point Q over the public channel and Bob sends his point$R$ over the public channel. Alice performs$K=k_1R$ and Bob performs $K=k_2Q$. Both determine the same point on the elliptic curve which is k_1k_2P. Because of the ECDLP, it is infeasible for an outside party to determine Alice or Bob's private key given point P, Q, and R. The x-coordinate of the resulting point is typically used as the shared key between Alice and Bob. Algorithm 2.3 illustrates the key exchange [24].

Algorithm 2.4 Elliptic curve digital signature algorithm

Input: Prime q,
Elliptic curve $E : y^2 \equiv x^3 + ax + b \bmod q$
Primitive point on E, $A = (x_A, y)$
Output: If the signature is verified.
Key Generation.
1. Choose a random integer d with $0 < d < q$
2. Compute $B = dA$
ECDSA Signature Generation.
3. Choose a random ephemeral key k with $0 < k < q$
4. Compute $R = kA = (u, v)$
5. Let $r = u \bmod q$
6. Compute $s = k^{-1}(SHA - 1(x) + mr) \bmod q$
ECDSA Signature Verification.
7. Compute $w = s^{-1} \bmod q$
8. Compute $i = wSHA - 1(x) \bmod q$
9. Compute $j = wr \bmod q$
10. Compute $(u, v) = iA + jB$
11. Verify $ver_K(x, (r, s)) =$ true iff $u \bmod q \equiv r$

2.4.2 Elliptic Curve Digital Signature Algorithm

The second protocol is Elliptic Curve digital signature algorithm (ECDSA) . This algorithm verifies that a message is valid. This uses an elliptic curve system over a Galois Field, p, and public point, A. There is also a private key, m, and secret random number, k. The algorithm verifies using a variety of computations, as shown in Algorithm 2.4. The protocol can be broken down into key generation, signature generation, and signature verification [24]. $SHA - 1$ is a hashing algorithm that can be changed out for any other hasing algorihtm, such as $SHA - 3$.

Chapter 3

Edwards Curve

In this Chapter, we review the Edwards curve. ECC cryptosystems can be implemented over a variety of curves. Some curves have more inherent properties than others. Table 3.1 contains a comparison of point addition and doubling formulas presented in literature. In comparing different coordinate schemes over this addition law, we utilize the following notation for the complexity. I refers to finite-field inversions. M refers to finite-field multiplications. D refers to a multiplication by a constant, i.e. curve parameter. S refers to finite-field squarings. A refers to finite-field additions, but are primarily excluded since the complexity of other finite-field arithmetic is much greater. Primarily, reducing the total number of multiplications and multiplications by a constant will have the greatest impact on complexity of a point multiplication. Completeness means that there are no exceptional cases to addition or doubling (e.g., adding the neutral point). From this table, the choice was to apply the new mixed coordinate addition and doubling formulas over new binary Edwards curves presented in [14].

The Edwards curve was originally proposed by Harold M. Edwards in [8] in 2007. Geometrically, the curve appears as a circle that caves in or outward, as shown in Figure 3.1. In this figure, the standard Edwards curve is defined in Eq. 3.1. We alter the value of d to produce several different curves, as shown in Figure 3.1.

$$x^2 + y^2 = 1 + dx^2y^2 \tag{3.1}$$

3.1 Binary Edwards Curve

The binary Edwards curve is an Edwards curve defined over a binary field. The original Edwards form is not elliptic over binary fields. Consider a finite field of characteristic two, K. Let

Table 3.1: Cost of point operations on binary generic curves (BGCs) [17], binary Edwards curves (BECs) [4], binary Edwards curves revisited [14], and generalized Hessian curves (GHC) [9] over $GF(2^m)$.

Curve	Coordinate System	Differential PA and PD	Completeness
BGC	Projective	$6M+1D+5S$	$\times$
	Mixed	$5M+1D+4S$	$\times$
BEC ($d_1 = d_2$)	Projective	$7M+2D+4S$	✓
	Mixed	$5M+2D+4S$	✓
BEC-R ($d_1 = d_2$)	Projective	$7M+2D+4S$	✓
	Mixed	$5M+1D+4S$	✓
GHC	Projective	$7M+2D+4S$	✓
	Mixed	$5M+2D+4S$	✓

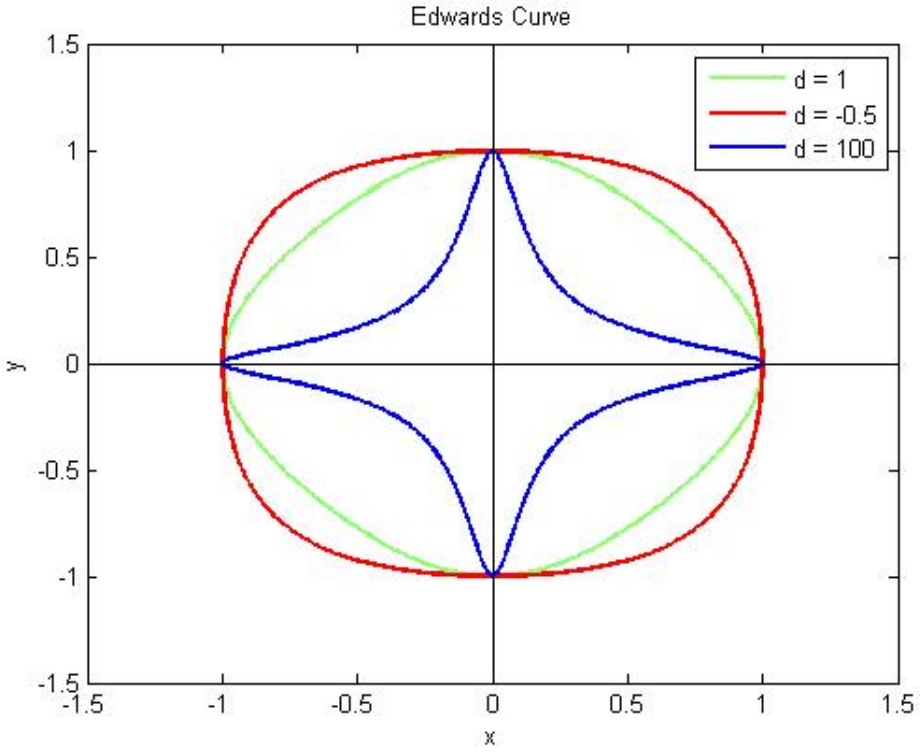

Figure 3.1: Edwards curve geometric representation.

$d_1, d_2 \in K$ such that $d_1 \neq 0$ and $d_2 \neq d_1^2 + d_1$.Then the binary Edwards curve with coefficients $d_1$and d_2 is the affine curve [4]:

$$E_{\mathbb{F}_{2^m},d_1,d_2} : d_1(x+y) + d_2(x^2+y^2) = xy + xy(x+y) + x^2y^2 \tag{3.2}$$

It can be shown that the above form of the curve is birationally equivalent to the standard Weierstrass form, e.g. $v^2 + uv = u^3 + a_2u^2 + a_6$. We utilize the map $(x,y) \longrightarrow (u,v)$ defined as [5]

$$u = d_1(d_1^2 + d_1 + d_2)(x+y)/(xy + d_1(x+y)),$$

$$v = d_1(d_1^2 + d_1 + d_2)(x/(xy + d_1(x+y)) + d_1 + 1)$$

Thus, the short Weierstrass curve is equivalent to the Edwards curve with the following relation:

$$v^2 + uv = u^3 + (d_1^2 + d_2)u^2 + d_1^4(d_1^4 + d_1^2 + d_2^2)$$

The above equivalence works for all but the point at infinity. Thus, we define the point $(0,0)$ on the binary Edwards curve to be isomorphic to the point at infinity in a binary generic curve. This point represents the neutral point in the binary Edwards curve. This curve is symmetric in that if (x,y) is on the curve, then (y,x) is also on the curve. In fact, these points are additive inverses over the Edwards addition law. The point (1,1) is also on every binary Edwards curve, and has order 2. The curve is complete if there is no element $t \in K$ that satisfies the relation $t^2 + t + d_2 = 0$ [4]. Alternatively, this means that if $\mathrm{Tr}(d_2) = 1$, then the curve is complete [26].

3.2 Edwards Addition Law

Here, we review the Edwards addition law, which defines the point arithmetic for Edwards curves.

Point addition and point doubling do not have the same representation or equations as standard generic curves. The Edwards addition law, derived in [4], is presented below in Eq. 3.3. The sum of any two points $(x_1,y_1),(x_2,y_2)$ on the curve defined by $E_{\mathbb{F}_{2^m},d_1,d_2}$ to (x_3,y_3) is

defined as

$$x_3 = \frac{d_1(x_1+x_2)+d_2(x_1+y_1)(x_2+y_2)+(x_1+x_1^2)(x_2(y_1+y_2+1)+y_1y_2)}{d_1+(x_1+x_1^2)(x_2+y_2)} \tag{3.3}$$

$$y_3 = \frac{d_1(y_1+y_2)+d_2(x_1+y_1)(x_2+y_2)+(y_1+y_1^2)(y_2(x_1+x_2+1)+x_1x_2)}{d_1+(y_1+y_1^2)(x_2+y_2)}$$

Since we have already shown that the binary Edwards curve is birationally equivalent to a short Weierstrass curve, we can utilize points on a binary Edwards curve instead of a short Weierstrass curve and, thus, gain benefits of speed and security by using the arithmetic over binary Edwards curves.

As we explore the arithmetic in other coordinate systems on the binary Edwards curve, we will make the assumption that $d_1 = d_2$. This assumption does not change the security in using points on the curve, but makes the arithmetic require less overall operations.

3.2.1 Montgomery Ladder

The Montgomery powering ladder [22] is an alternative to a standard binary approach that uses a double-and-add method. Instead of requiring an add only if the bit of scalar multiple is '1', there is a double-and-add at each step. Although this requires more time to compute a scalar point multiplication, the fact that it uses a constant set of point doubling and point additions means that it is resistant to timing and simple power analysis attacks that try to retrieve bits of the secret key k.

The Montgomery ladder ensures that the difference between two points on the ladder is always P, the starting point. To do this, the first step is to calculate $2P$. From there, each step of the ladder contains registers holding the values for mP and $mP+1$. The algorithm for the Montgomery powering ladder [22] is shown in Algorithm 3.1. The final result is mP after the last step.

For the rest of this thesis, the Montgomery ladder will be used as it provides essential timing and simple power analysis defense. All interactions with the secret key must be done in such a way that the key is not exposed.

Algorithm 3.1 Montgomery powering ladder for scalar point multiplication [22].

Inputs: A point $P = (x_0, y_0) \in E(\mathbb{F}_{2^m})$ on a binary curve and an integer $k = (k_{l-1}, \cdots, k_1, k_0)_2$.
Output: $Q = kP \in E(\mathbb{F}_{2^m})$.
1: set: $A = P$, $B = 2P$ and initialize
2: **for** i from $i-1$ downto 0 **do**
 3: **if** $k_i = 0$ **then**
  4: $A = 2A$, $B = A + B$
 5: **else**
  6: $A = A + B$, $B = 2B$
 7: **end if**
8: **end for**
9: **return** $Q = A$

3.2.2 Affine Coordinates

The formula for affine point addition is presented in Eq. 3.3. Actually, the assumption $d_1 = d_2$ does not speed up point addition. Although the denominators are different in the formula, a simultaneous inversion trick can be used to perform both at the same time. Consider D_1 and D_2 to be denominators 1 and 2, respectively. Also consider I_1 and I_2 to be the inverse of denominators 1 and 2, respectively. Then these inversions can be generated with a single inversion and 3 extra multiplications as shown:

$$A = \frac{1}{D_1 D_2}$$

$$I_1 = AD_2$$

$$I_2 = AD_1$$

Avoiding a single inversion is necessary for efficient implementations of ECC, since it could cost as many as $m-1$ squarings and $H(m-2)$ multiplications. Careful planning and reuse of values for the rest of the formula can implement affine addition in $I + 10M + 1D + 4S$, as shown in [14].

3.2.3 Projective Coordinates

Along the same lines of minimizing the number of inversions in a point multiplication, projective coordinates are a way to only require a single inversion over the course of a scalar point multiplication. Instead of using the standard coordinate system $P = (x, y)$, we represent coordinates as $P = (X : Y : Z)$, where $x = X/Z$, and $y = Y/Z$. Thus, instead of performing the division at the end of the affine coordinate formula, we adjust the Z-coordinate so that the division at the end of the Montgomery ladder is all that is needed to recover x and y. In projective coordinates, the formula for point addition, $(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2)$ requires $14M + 2D + S$ if $d_1 = d_2$, as shown in [14]. There are many terms for X_3, Y_3, and Z_3, so the formula will not be reiterated here.

3.2.4 Differential Coordinates

Differential coordinates reduce the number of registers that need to be stored to hold a point from 2 to 1 over standard affine coordinates. Notably, a point is represented as $w = x + y$. Thus, only a single coordinate $(x, y) \longrightarrow (w)$ is needed between point additions and doublings. These differential coordinates also simplify some of the logic for point addition and doubling formulas, since the value $w_i = x_i + y_i$ is reused heavily in the affine equation for point addition. It should also be noted that the point doubling and point addition formulas are different for this case as well. For computing a point multiplication, let P be a point on a binary Edwards curve $E_{\mathbb{F}_{2^m},d_1,d_2}$ and let us assume $w(nP)$ and $w((n+1)P)$, $0 < n < k$ are known. Therefore, one can use the w-coordinate differential addition and doubling formulas to compute their sum as $w((2n+1)P)$ and double of $w(nP)$ as $w(2nP)$[4].

In [14], the authors present faster equations for w-coordinates and mixed coordinates addition than those presented in [4]. This equation makes the assumption that $d_1 = d_2$ as well as that curve parameters. An analysis of the formula, however, shows that they do not properly produce the correct w-coordinates. The authors correctly identify the relation, $\frac{w_3 w_0}{d_1(w_1^2+w_2^2)} = \frac{w_3+w_0+1}{d_1}$, but incorrectly solve for w_3. We observe that the final equation for differential point addition that is presented in subsection (3.19) of [14] is faulty. Therefore, we wrote a sage script to verify this claim, which is listed in the Appendix. This algebra was performed correctly and here we present the revised formulas. The incorrect formula presented in [14] is in Eq. 3.4 and the corrected formula is shown in Eq. 3.5. This formula defines the addition of $w_1 + w_2 = w_3$, given that $w_i = x_i + y_i$ and $w_0 = w_2 - w_1$. The curve parameter $\frac{1}{w_0}$ does not change when utilized throughout the Montgomery ladder. As such, it can be precomputed and utilized to speed up the arithmetic.

Algorithm 3.2 Montgomery algorithm [22] for point multiplication using w-coordinates.

Inputs: A point $P=(x_0,y_0)\in E(\mathbb{F}_{2^m})$ on a binary curve and an integer $k=(k_{l-1},\cdots,k_1,k_0)_2$.
Output: $w(Q)=w(kP)\in E(\mathbb{F}_{2^m})$.
1: `set` : $w_0\leftarrow x_0+y_0$ and initialize
a: $W_1\leftarrow w_0$ and $Z_1\leftarrow 1$ and $c=\frac{1}{w_0}$ (inversion)
b: $(W_2,Z_2)=\texttt{DiffDBL}(W_1,Z_1)$
2: **for** i **from** $l-2$ down to 0 **do**
a: **if** $k_i=1$ **then**
i): $(W_1,Z_1)=\texttt{MDiffADD}(W_1,Z_1,W_2,Z_2,c)$
ii): $(W_2,Z_2)=\texttt{DiffDBL}(W_2,Z_2)$
b: **else**
i): $(W_1,Z_1)=\texttt{DiffDBL}(W_1,Z_1)$
ii): $(W_2,Z_2)=\texttt{MDiffADD}(W_1,Z_1,W_2,Z_2,c)$
end if
end for
3: **return** $w(kP)\leftarrow(W_1,Z_1)$ and $w((k+1)P)\leftarrow(W_2,Z_2)$

The w-coordinate differential addition formula over binary Edwards curves with $d_1=d_2$ proposed in [14] does not provide correct formulation based on the following equation:

$$w_3=1+\frac{\frac{1}{w_0}(w_1^2+w_2^2)}{\frac{1}{w_0}(w_1^2+w_2^2)+1} \tag{3.4}$$

In the following equations, the correct w-coordinate differential addition formula over binary Edwards curves with $d_1=d_2$ is discovered from the starting relation in [14].

$$\frac{w_3w_0}{d_1(w_1^2+w_2^2)}=\frac{w_3+w_0+1}{d_1}$$

$$\frac{w_3w_0}{(w_1^2+w_2^2)}=w_3+w_0+1$$

$$\frac{w_3w_0}{(w_1^2+w_2^2)}+w_3=w_0+1$$

$$w_3\left(\frac{w_0}{(w_1^2+w_2^2)}+1\right)=w_0+1$$

$$w_3(w_0+w_1^2+w_2^2)=(w_0+1)(w_1^2+w_2^2)$$

$$w_3 = \frac{(w_0+1)(w_1^2+w_2^2)}{w_0+w_1^2+w_2^2}$$

$$w_3 = \frac{(1+\frac{1}{w_0})(w_1^2+w_2^2)}{\frac{1}{w_0}(w_1^2+w_2^2)+1}$$

Corrected w-Coordinate Differential Addition

$$w_3 = \frac{w_1^2+w_2^2+\frac{1}{w_0}(w_1^2+w_2^2)}{\frac{1}{w_0}(w_1^2+w_2^2)+1}. \tag{3.5}$$

The explicit affine w-coordinate differential addition is

$$\begin{aligned} A = (w_1+w_2)^2, \quad B = A\cdot\tfrac{1}{w_0}, \quad N = A+B, \\ D = B+1, \quad E = \tfrac{1}{D}, \quad w_3 = N\cdot E. \end{aligned} \tag{3.6}$$

The total cost of this corrected formula is still $1I+1M+1D+1S$, but now the differential addition functions as intended. Assuming that inversion requires at least two registers, a total of three registers are required. $\frac{1}{w_0}$ is the inverse of the difference between the points and will not be updated in each step of the point multiplication algorithm. For the application in Montgomery Ladder [22], the difference between the two points is always P (specifically $w(P)$). Therefore, this value can be determined at the start of the ladder and used throughout to cut down on each step.

[14] uses the faulty formula (3.4) for determining explicit formulas in mixed w-coordinates, but also gives a faster and correct formula for affine w-coordinate differential addition which requires $1I+1M+2S$, so long as the values $\frac{1}{w_0+w_0^2}$ and w_0 are known. This formula is shown below.

$$w_3 = w_0+1+\frac{1}{\frac{1}{w_0+w_0^2}(w_1^2+w_2^2+w_0)} \tag{3.7}$$

The explicit affine w-coordinate differential addition is

$$\begin{aligned} A = (w_1+w_2)^2, \quad B = A+w_0, \quad D = B\cdot\frac{1}{w_0+w_0^2}, \\ E = \frac{1}{D}, \quad w_3 = E+1+w_0 \end{aligned} \tag{3.8}$$

Assuming that w_0 and $\frac{1}{w_0+w_0^2}$ are known, the actual cost for w-coordinate differential addition can be reduced down to $1I+1D+1S$. This method requires two registers for w_1 and w_2, and the storage of w_0 and $\frac{1}{w_0+w_0^2}$.

3.2.4.1 Retrieving x and y from w-Coordinates

The formula to retrieve the x-coordinate from w-coordinates is presented in [4], which is based on the point doubling formula. This formula requires P, $w(kP)$, and $w(kP+1)$. Again, relating back to the application of Montgomery Ladder [22], each consecutive step produces $w(mP)$ and $w(mP+1)$, where m represents the scalar multiplication over each steps. The formula to solve for the x-coordinate of mP is shown below [4]. In this formula, $P=(x_1,y_1)$,$w_0=x_1+y_1$, $w_2=w(kP)$, and $w_3=w(kP+1)$.

$$x_2^2+x_2=\frac{w_3(d_1+w_0w_2(1+w_0+w_2)+\frac{d_2}{d_1}w_0^2w_2^2)+d_1(w_0+w_2)+(y_1^2+y_1)(w_0^2+w_2)}{w_0^2+w_0} \tag{3.9}$$

This formula requires $1I+4M+4S$ if $d_2=d_1$. After solving for $x_2^2+x_2=A$, if $\mathrm{Tr}(A)=0$, then the value of x_2 or x_2+1 can be recovered by using the half-trace.

After the value of x_2 has been found, y_2 can be retrieved by solving the curve equation for $y_2^2+y_2$, Eq. 3.10, and also using the half-trace to solve for y_2 or y_2+1.

$$y_2^2+y_2=\frac{d(x_2+x_2^2)}{d+x_2+x_2^2} \tag{3.10}$$

Therefore, recovering y_2 requires $1I+2M+S$, and the total cost of recovering points from w-coordinates is $2I+6M+5S$. Even though the point $(x_2+1,\ y_2+1)$ is not the same as $(x_2,\ y_2)$, both points will produce the same value in standard ECC applications. Algorithm 3.3 summarizes how to retrieve the x and y-coordinates.

3.2.5 Mixed Coordinates

Eq. 3.5 can be applied to mixed w-coordinate differential addition and doubling. The general formula and explicit formula are shown below. This formula defines the addition of $\frac{W_1}{Z_1}+\frac{W_2}{Z_2}=\frac{W_3}{Z_3}$, given that $w_0=w_2-w_1$.

$$\frac{W_3}{Z_3}=\frac{(W_1Z_2+W_2Z_1)^2+\frac{1}{w_0}(W_1Z_2+W_2Z_1)^2}{Z_1^2Z_2^2+\frac{1}{w_0}(W_1Z_2+W_2Z_1)^2} \tag{3.11}$$

Algorithm 3.3 Retrieving x and y from w-coordinates

Inputs: A point $P = (x_0, y_0) \in E(\mathbb{F}_{2^m})$ on a binary curve and an integer $k = (k_{l-1}, \cdots, k_1, k_0)_2$.
Output: $Q = kP \in E(\mathbb{F}_{2^m})$.
1: set: $w_0 \leftarrow x_0 + y_0$ and initialize
2: compute: $w_2 \leftarrow w(kP)$, $w_3 \leftarrow w(kP+1)$
3: solve (3.9) for $x_2 + x_2^2$
4: **if** $\mathrm{Tr}(x_2 + x_2^2) = 0$ **then**
a: $x_2 =$**half-trace**$(x_2 + x_2^2)$
end if
5: solve (3.10) for $y_2 + y_2^2$
6: **if** $\mathrm{Tr}(y_2 + y_2^2) = 0$ **then**
a: $y_2 =$**half-trace**$(y_2 + y_2^2)$
end if
7: **return** $Q = (x_2, y_2) = kP \in E(\mathbb{F}_{2^m})$

$$C = (W_1 Z_2 + W_2 Z_1)^2, \quad D = (Z_1 Z_2)^2, \quad E = \frac{1}{w_0} \cdot C, \qquad (3.12)$$
$$W_3 = E + C, \quad Z_3 = E + D$$

Thus, mixed w-coordinate differential addition requires $3M + 1D + 2S$. From a simple analysis of the formula, four registers are needed.

For mixed w-coordinate differential addition and doubling, the doubling formula from [4] can be used in conjunction with this corrected differential addition formula, with the assumption that $d_1 = d_2$. This formula defines the addition of $\frac{W_1}{Z_1} + \frac{W_2}{Z_2} = \frac{W_3}{Z_3}$ and doubling of $2 \times \frac{W_1}{Z_1} = \frac{W_4}{Z_4}$ given that $w_0 = w_2 - w_1$.

$$\frac{W_4}{Z_4} = \frac{(W_1(W_1 + Z_1))^2}{d_1 . Z_1^4 + (W_1(W_1 + Z_1))^2} \qquad (3.13)$$

$$C = (W_1 Z_2 + W_2 Z_1)^2, \quad D = (Z_1 Z_2)^2, \quad E = \frac{1}{w_0} \cdot C, \qquad (3.14)$$
$$W_3 = E + C, \quad Z_3 = E + D \quad W_4 = (W_1(W_1 + Z_1))^2,$$
$$Z_4 = W_4 + d_1 \cdot Z_1^4$$

Thus, mixed w-coordinate differential addition and doubling requires $5M + 1D + 5S$. From

an analysis of the formula, five registers are needed.

3.2.5.1 Mixed w-Coordinate Differential Addition and Doubling with the Co-Z Trick

We note that in [39] the common-Z trick is proposed. This method to reduces the number of registers required per step of the Montgomery Ladder [22] and simplifies the number of operations per step. Each step of the Montgomery Ladder is a point doubling and addition. By using a common-Z coordinate system, one less register is required for a step on the ladder, and the method becomes more efficient, requiring one less squaring operation. The doubling formula was obtained from [4] and it is assumed that $d_1 = d_2$. The general formula and explicit formulas are shown below. This formula defines the addition of $\frac{W_1}{Z} + \frac{W_2}{Z} = \frac{W_3}{Z'}$ and doubling of $2 \times \frac{W_1}{Z} = \frac{W_4}{Z'}$ given that $w_0 = w_2 - w_1$.

$$\frac{W_3}{Z'} = \frac{(W_1+W_2)^2 + \frac{1}{w_0}(W_1+W_2)^2}{Z^2 + \frac{1}{w_0}(W_1+W_2)^2} \tag{3.15}$$

$$\frac{W_4}{Z'} = \frac{(W_1(W_1+Z))^2}{d_1 \cdot Z^4 + (W_1(W_1+Z))^2} \tag{3.16}$$

$$\begin{aligned} C = (W_1+W_2)^2, \quad D = Z^2, \quad E = \frac{1}{w_0} \cdot C, \\ U = E+C, \quad V = E+D, \quad S = (W_1(W_1+Z))^2, \\ T = S + d_1 \cdot D^2, \quad W_3 = U \cdot T, \quad W_4 = V \cdot S, \\ Z' = V \cdot T \end{aligned} \tag{3.17}$$

Thus, the mixed w-coordinate differential addition and doubling formula requires $5M + 1D + 4S$. An analysis of this formula shows that it requires only four registers. As will be discussed later, this implementation incorporates shifting for the multiplication within the register file, forcing the need for an additional register. This formula requires one less squaring than that provided in [4], and also uses registers much more efficiently. Table 2 shows a comparison of differential point addition schemes for BEC with $d_1 = d_2$.

3.3 Resistance Against Side-Channel Attacks

The binary Edwards curve features the unique properties that its addition formula is unified and complete. Unified implies that the addition and doubling formulas are the same. This

Table 3.2: Comparison of differential point addition schemes for BEC with $d_1 = d_2$.

Operation	Eq.	Complexity	#Reg
Affine w-coordinate Differential Addition	3.6	$1I + 1M + 1D + 2S$	3
Affine w-coordinate Differential Addition	3.8	$1I + 1D + 1S$	2
Mixed Differential Addition	3.12	$3M + 1D + 1S$	4
Mixed Differential Addition and Doubling	3.14	$5M + 1D + 5S$	5
Mixed Differential Addition and Doubling w/ Co-Z	3.17	$5M + 1D + 4S$	4

gives the advantage that no checking is required for the points to differentiate if an addition or doubling needs to take place. Complete implies that the addition formula works for any two input points, including the neutral point. Therefore, as long as two points are on the curve, no checking is needed for the addition formula, as it will always produce a point for a complete binary Edwards Curve [4].

One common attack to reveal bits of an ECC system's key is to use the exceptional points attack [33]. This attacks the common projective coordinate system. For the point at infinity in a non-binary Edwards curve system, the point is often represented as $(X_k, Y_k, 0)$. Hence, a conversion back to the (x_k, y_k) coordinate system would attempt to divide by zero, causing an error or revealing a point that is not on the curve [33]. In either case, an adversary could detect that the point at infinity was attempted to be retrieved. The attack relies on picking different base points, which after multiplied by the hidden key, reveal that the point at infinity was retrieved. If the point at infinity is reached, then that reveals critical bits of the secret key, which could be used to break the cryptosystem.

The binary Edwards curve's completeness property and coordinate system make the curve immune to this form of attack. For a complete binary Edwards curve, the projective coordinate system representation for the neutral point, which is isomorphic to the point at infinity of other curves, is $(X_k, Y_k, 1)$. Furthermore, the completeness also ensures that no other sets of points can be used to break the system and reveal critical information about the key. The mixed w-coordinates that are used for their speed in the binary Edwards curve are also invulnerable to this attack as long as $w_0 \neq 0,1$, since the denominator will never be 0 [14]. With the Montgomery Ladder [22], a proper curve and starting point will never violate this condition.

Montgomery Ladder [22] is a secure way to perform repeated point addition and point doublings to thwart side channel attacks. The ladder provides a point addition and point doubling for each step, with each step taking the same amount of time. Therefore, this application provides an extremely powerful defense against power analysis attacks and timing attacks. Power analysis attacks identify characteristics of the power consumption of a device to reveal bits of the key and timing attacks identify characteristics of the timing as the point multiplica-

tion is performed. By application of the binary Edwards curve with Montgomery Ladder, the binary Edwards curve features an innate defense against many of the most common attacks on ECC systems today.

It should be noted that there are many other types of attacks, such as differential power analysis (DPA) [16] or electromagnetic (EM) radiation leaks. These require a much closer look at key management and register usage, and are thus, more dependent on the architecture rather than the curve coordinate system.

Chapter 4

Lightweight Implementation of ECC

This chapter details the lightweight implementation of a point multiplier over the revised formulas for binary Edwards curves.

4.1 Design Methodology

This section details the methodology to create a small embedded crypto coprocessor.

4.1.1 Motivation: Embedded Crypto Coprocessor

With the increasing popularity of Radio Frequency Identification (RFID) technologies in areas such as the Internet of Things [19] and medical implants, there is a need to design a small and efficient security coprocessor. Embedded devices have generally dominated the computer market. With the transition to the Internet of Things, all tools and electronics will be linked wirelessly. With so many devices connected for potentially sensitive applications, security must also be done correctly. However, functionality and connectivity monopolize an embedded system's area and power, so such a security coprocessor must be tiny, fast, and power-efficient, illustrated in Fig. 4.1.

Such a coprocessor will be able to efficiently perform all of the security computations. This could include generating a secret key through ECDH, signing a message to a host machine with ECDSA, or many others. ECC is the ideal implementation for these security needs because it provides a secure application for far fewer bits than RSA and other public key encryption schemes. Overall, the goal is to implement a coprocessor that achieves a tiny footprint, while still performing the calculations in a short amount of time. Thus, the main emphasis is on size, but the latency is also minimized so that power and energy remain relatively

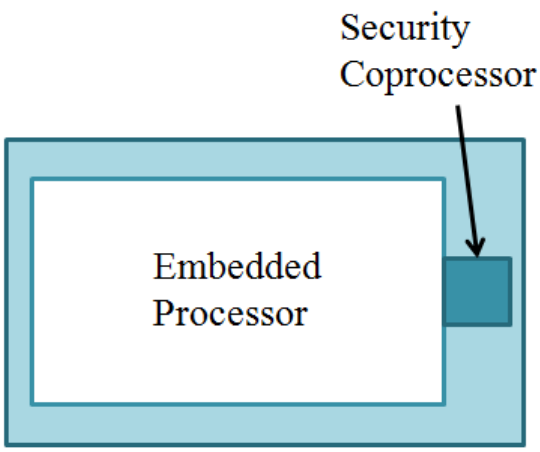

Figure 4.1: Security coprocessor design.

low.

4.1.2 NIST Binary Field Standards

In 2000, NIST standardized several elliptic curves over binary fields and prime fields [34]. In particular, NIST standardized binary generic curves over the binary fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$. These are not necessarily over binary Edwards curves, but the field sizes determine the approximate security of the implementation. For an elliptic curve defined over the binary field $\mathbb{F}_{2^{283}}$, there are approximately 2^{283} points on the curve. Based on the progress of factoring algorithms to solve the ECDL problem, this corresponds to a private key cryptography key size of 128-bits. Thus, it requires approximately the complexity 2^{128} brute force attempts to potentially determine the secret key k in $Q = kP$. We summarize NIST's security of standardized curves in Table 4.1. Advanced Encryption Standard (AES) is the current standard for private key cryptography, or symmetric key cryptography, at multiple security levels. The security size of RSA, another popular public-key cryptography scheme is also included for comparison. We note that $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$ correspond to a low, medium, and high level of security. Typically, the convention is to implement a design over a single standardized curve, but this paper implements point multiplications for the low, medium, and high level of security to compare the scalability and security of the approaches.

Table 4.1: NIST recommended curves and security [34].

Key Size (bits)			
Symmetric key	RSA	ECC (prime field)	ECC (binary field)
80	1,024	192	163
112	2,048	224	233
128	3,072	256	283
192	7,680	384	409
256	15,360	521	571

Table 4.2: NIST recommended curve binary field parameters [34].

Key Size (bits)		Irreducible	GNB
Symmetric key	ECC (binary field)	Polynomial	Type
80	163	$x^{163}+x^7+x^6+x^3+1$	4
112	233	$x^{233}+x^{74}+1$	2
128	283	$x^{283}+x^{12}+x^7+x^5+1$	6
192	409	$x^{409}+x^{87}+1$	4
256	571	$x^{571}+x^{10}+x^5+x^2+1$	10

4.1.3 GNB or Polynomial Basis

This thesis will focus on the binary fields $\mathbb{F}_{2^{163}}, \mathbb{F}_{2^{233}}$, and $\mathbb{F}_{2^{283}}$ as they are standardized by NIST and provide a low, medium, and high level of security [34]. The complexity of the multiplier is dependent on the irreducible modulus for polynomial basis and type of GNB. For polynomial basis, the smallest irreducible modulus for $\mathbb{F}_{2^{163}}$ is $F(x) = 2^{163} + 2^7 + 2^6 + 2^3$, for $\mathbb{F}_{2^{233}}$ is $F(x) = 2^{233} + 2^{74}$, for $\mathbb{F}_{2^{283}}$ is $F(x) = 2^{283} + 2^{12} + 2^7 + 2^5$. For GNB, $\mathbb{F}_{2^{163}}$ has a type 4 GNB, $\mathbb{F}_{2^{233}}$ has a type 2 GNB, and $\mathbb{F}_{2^{283}}$ has a type 6 GNB. These factors are summarized with the size of each NIST standardized curve in Table 4.2.

The complexity of a bit-serial multiplier over polynomial basis is relatively simple. The Least-Significant Bit (LSB) method is shown in Algorithm and illustrated in Fig. 4.2. For this type of multiplier, there are m AND gates, $m+\omega$ XOR gates, 2 registers, and a shift register. ω represents the number of non-zero terms in the irreducible modulus. Thus, there are 2 additional XOR gates in the modulus term in a polynomial basis over $\mathbb{F}_{2^{283}}$ as compared to a modulus term in a polynomial basis over $\mathbb{F}_{2^{233}}$.

Squaring over polynomial basis is much more difficult. A bit-parallel squaring unit effectively performs all reduction operations at once, which is more efficient since the original value can be extended by inserting '0's between each bit and then reducing. Trinomials are much more efficient for this purpose. As noted in [36], the general complexity of a bit-parallel

Algorithm 4.1 The LSB-first bit-serial polynomial basis multiplier [1].

Input: $A, B \in GF(2^m)$, F(x)
Output: $C = A \times B$ mod F(x).
1. $A' = A$, $Y = 0$
2. **For** i from 0 to $m-1$ **do**
   3. $Y = b_i A' + Y$
   4. $A' = A'x$ mod F(x)
5. **end for**
6. **Return** $C = Y$

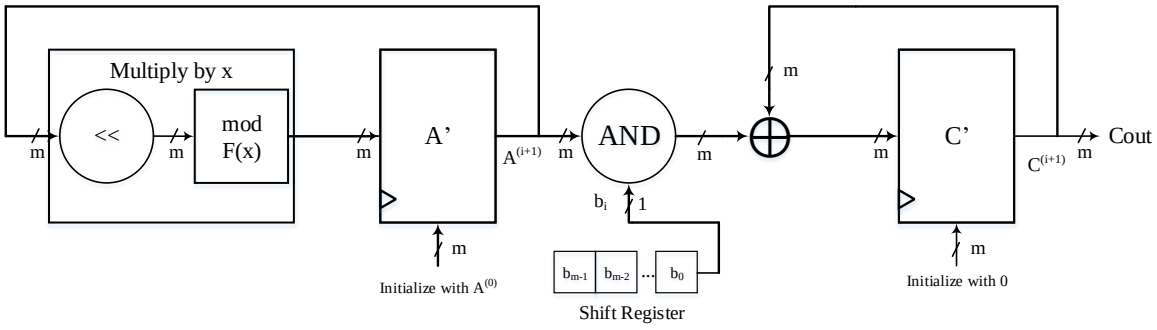

Figure 4.2: Polynomial basis bit-serial multiplier [1].

Table 4.3: Comparison among bit-level multipliers for type T GNB over $GF(2^m)$ with $2m-1 \leq C_N \leq Tm-T+1$.

Bit-level	#	# XOR				#	Critical
Multipliers	AND	$\mathbb{F}_{2^m}$	$\mathbb{F}_{2^{283}}$	$\mathbb{F}_{2^{233}}$	$\mathbb{F}_{2^{163}}$	FFs	path delay
MO [18]	C_N	C_N-1	1692	464	648	$2m$	$T_A+\lceil \log_2 C_N \rceil T_X$
RH [30]	$\frac{m+1}{2}$	$\leq \frac{C_N+2m-1}{2}$	1129	465	487	$3m$	$T_A+(1+\lceil \log_2(T+1) \rceil)T_X$
Feng[10]	$2m$-1	C_N+3m-1	2541	1163	1173	$3m$	$T_A+(2+\lceil \log_2 T \rceil)T_X$
AR[3], [29]	m	$\leq \frac{C_N+m}{2}$	964	349	401	$3m$	$T_A+(1+\lceil \log_2 T \rceil)T_X$
AJL [27]	m	$\leq \frac{C_N+m}{2}$	817	349	401	$3m$	$T_A+(1+\lceil \log_2 T \rceil)T_X$

squaring unit over is $(m+k-1)/2$ for the case that m is odd and k is even. For $\mathbb{F}_{2^{233}}$, that means that there are 153 XOR gates. The forms of the irreducible pentanomials do not demonstrate form suitable for efficient bit-parallel squaring in shifted polynomial basis [37]. Thus, a bit-parallel squaring can be done by unrolling all of the different possibilities of the irreducible, which is as much as 3 XOR's per bit. Bit-parallel squaring is applicable for $\mathbb{F}_{2^{233}}$, but not for $\mathbb{F}_{2^{163}}$ or $\mathbb{F}_{2^{283}}$.

Table 4.3 compares several different GNB multiplication schemes. C_N denotes the complexity of normal basis and it is measured by the number of entries of multiplication matrix R. For more details about its values, one can refer to [23] and [2]. It is not explicitly stated, but these multipliers can be Serial-in-Serial-out (SISO), Serial-in-Parallel-out (SIPO), Parallel-in-Serial-out (PISO), or Parallel-in-Parallel-out (PIPO). For the purposes of using the multiplier with a register file, PIPO is preferred. Even if PISO is used, there will still be a shift register accumulator to accumulate the full result. The last bit-level multiplier, in [27] provides the smallest complexity in terms of XOR gates, and a reasonable complexity of AND gates and registers. Further, it is PIPO multiplier. Therefore, this multiplier is a suitable choice for a lightweight implementation of ECC. We note the critical path delay of each of the multipliers as well, but this is not necessarily an important metric for a security coprocessor, as the critical path of the device will most likely be the limiting factor, and not the security coprocessor. We illustrate this multiplier in Fig. 4.3.

As a comparison between GNB and polynomial basis, it is noted that squaring is free in GNB. Bit-serial squaring is expensive in polynomial basis, unless the irreducible modulus is a trinomial. In terms of multipliers, both multipliers require m AND gates. Polynomial basis requires one fewer fewer shift register than GNB. However, squaring is perhaps the main distinguishing factor. Squaring is found numerous times throughout the point arithmetic formulas, and approximately m times for inversion. If a squaring unit was left out, in the case of

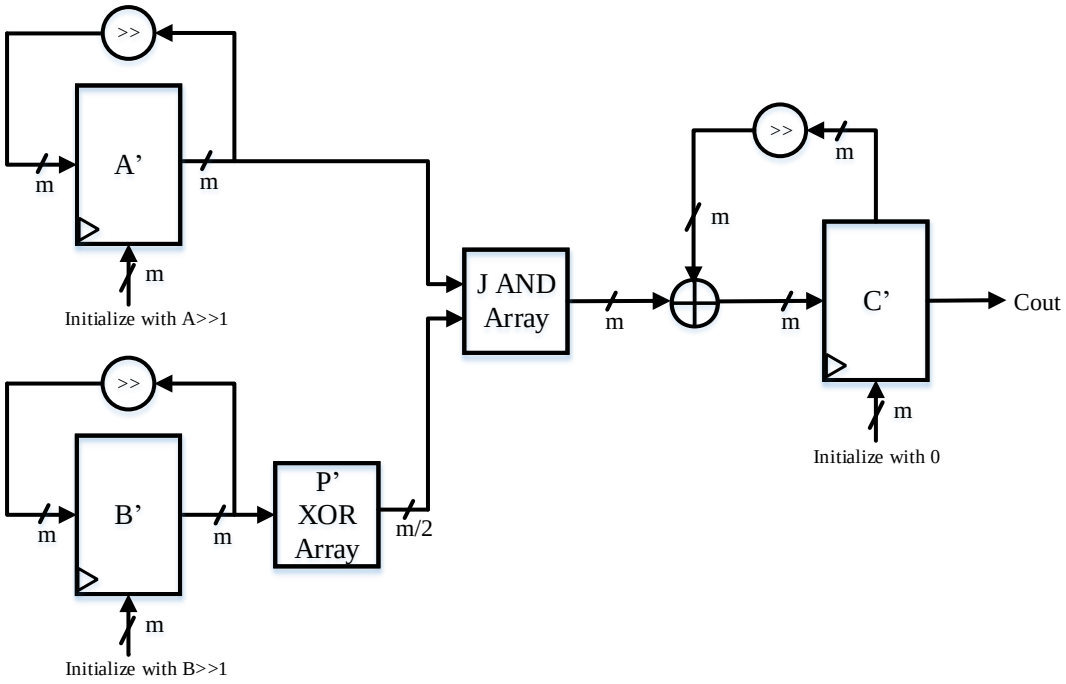

Figure 4.3: Gaussian normal basis bit-serial PIPO multiplier [27]

a polynomial basis architecture, then the increase in latency for each step of the Montgomery ladder and inversion would be massive. Bit-parallel squaring for $\mathbb{F}_{2^{233}}$ is relatively cheap as it only requires 153 XOR gates. However, when compared with 349 XOR gates for a squaring unit and a multiplication unit, the polynomial basis system requires 388 XOR gates. Therefore, GNB appears to be a clear winner in reduction of multiplication and squaring units. Polynomial basis could be considered in the case of using the standard expand and reduce methodology for squaring, but this requires more control logic, i.e. multiplexers. Thus, GNB multiplication was chosen for the arithmetic involved in this architecture.

4.1.4 Curve Selection

The implementation of the revised binary Edwards curves point arithmetic require a binary Edwards curve with $d_1 = d_2$. The standardized NIST curves over binary generic curves [34] could be converted to binary Edwards curves. However, there is no guarantee that these isomorphic binary Edwards curves would satisfy $d_1 = d_2$. Therefore, values for x and d were randomly picked and used in conjunction with Eq. 3.10 to solve for y. If the point (x,y) was on the curve, then the point and corresponding binary Edwards curve were valid and could be used with the above algorithms. It can also be noted that there are no restrictions on d, so

it could be chosen to be small for faster arithmetic. This design uses a full register to store values, so a smaller value is not helpful in this case.

We provide a sample point and curve for $\mathbb{F}_{2^{283}}$ in Eq. 4.1.

$$d(x+y)+d(x^2+y^2)=(x+x^2)(y+y^2)$$

$$\mathbb{F}_{2^{283}}\begin{cases} d = & 59da32313070cf9c9296a984782aa0aaa33747b9bc82b29018f8ca1f6668735b52c267f \\ x_0 = & 4aad94334c0ba0131029aa662f4e2dc19759ea7ab1430f3cfa0096b214f82e6edfec025 \\ y_0 = & 45be47b3a58d7dfd1a6bcb2edb1f5b4d9ab7fbb94529e6e72f810663a42ac094381d7ec \\ \frac{1}{w_0} = & 116ab9e363fddcd697da67caad49a7f0f7365d09207a82d762fdcf6715b602791a3c42e \end{cases} \tag{4.1}$$

4.2 Architecture

The architecture of the ECC co-processor that was implemented resembles that of [28]. However, there are several major differences. An analysis of the explicit formula presented for mixed w-coordinate addition and doubling revealed that five registers (T_0,T_1,R_0,R_1,R_2) and four constants $(\frac{1}{w_0},d_1,x_1,y_1)$ were required. Additionally, it was deemed that the neutral element in GNB multiplication (all '1's) was not required for any part of the multiplication, which reduced the size of the 4:2 output multiplexer to a 3:2 multiplexer. These following sections will explain the design in more detail. The architecture for the field arithmetic unit is shown in Fig. 4.4. The architecture of the register file is shown in Fig. 4.5. The top-level design of the point multiplier is shown in Fig. 4.6.

4.2.1 Field Arithmetic Unit

The field arithmetic unit is designed to incorporate the critical finite field operations in as small of a place as possible. In particular, this requires multiplication, squaring, and addition. The XOR gate to add two elements was reused in the multiplication and addition to reduce the total size of the FAU. Since the neutral element was not necessary for this point multiplier, the neutral element select from the output multiplexer in [28] was removed to save area. Swap functionality was added to incorporate quick register file swap operations. The field arithmetic unit incorporates the GNB multiplier from [27]. The operations are as follows:

- **Addition** $C=A+B$: Addition is a simple XOR of two inputs. The first input is loaded to Z by selecting the first input in the register file, and setting $s_1=$"01" and $s_2=$"00".

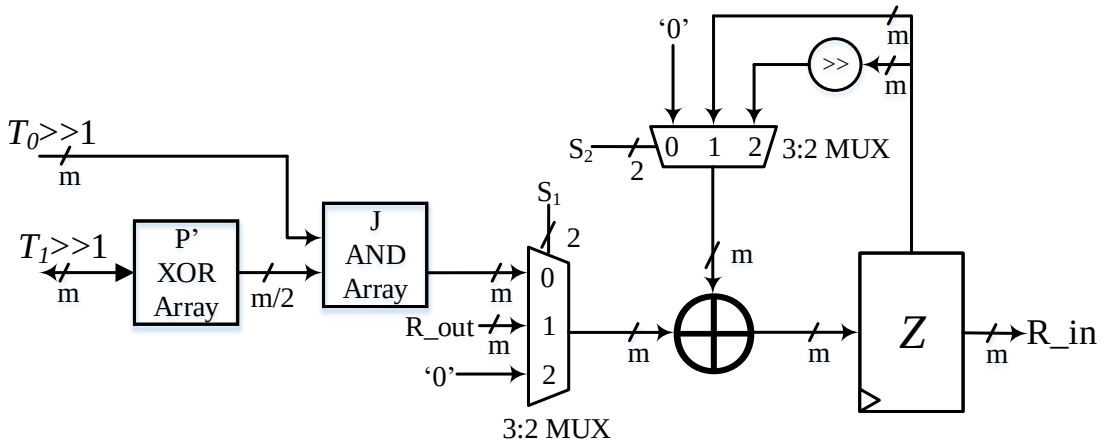

Figure 4.4: Proposed field arithmetic unit.

The next cycle, the second operand is selected from the register file, and $s_2 =$ "01" so that the output register has the addition of the two input elements. The output is written on the third cycle. This operation requires three clock cycles.

- **Squaring** $C = A \gg 1$: Squaring is a right circular shift of the input. The input is loaded to Z by selecting the input in the register file, and setting $s_1 =$ "01" and $s_2 =$ "00". The next cycle, $s_1 =$ "10" and $s_2 =$ "10" so that the output register has been shifted. The output is written on the third cycle. This operation requires three clock cycles.

- **Multiplication** $C = T_0 \times T_1$: Multiplication is a series of shifted additions. For the first cycle, $s_1 =$ "00",$s_2 =$ "00", and $s_{T_0} = s_{T_1} =$ "1". The next cycle, $s_2 =$ "01". After m cycles of shifts and addition, $s_{T_0} = s_{T_1} =$ "0",and the output is ready. The output is written on the mth cycle. This operation requires m clock cycles.

- **Swapping** $A,B = B,A$: Swapping is a switch of two registers within the register file. The first register is loaded to Z by selecting the input in the register file, and setting $s_1 =$ "01" and $s_2 =$ "00". The next cycle, the first register is written to the second register's location as it is being loaded to Z. The second register's value is written to the first register's place on the third cycle. This operation requires three clock cycles.

4.2.2 Register File

Similar to [28] and [15], the register file was designed to contain registers, with two particular registers that perform special shifting for the finite field multiplication. An analysis of the formulas used in this ECC unit revealed that four registers and four constants were required.

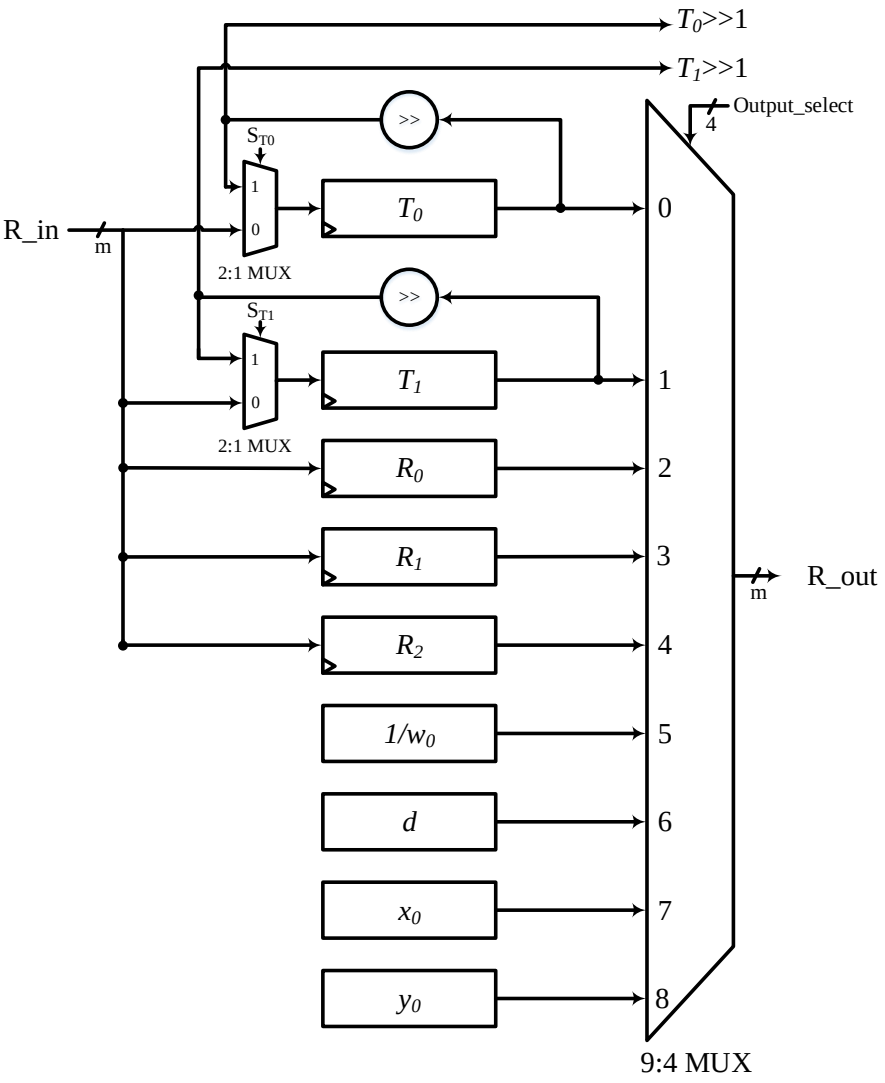

Figure 4.5: Proposed register file for GNB.

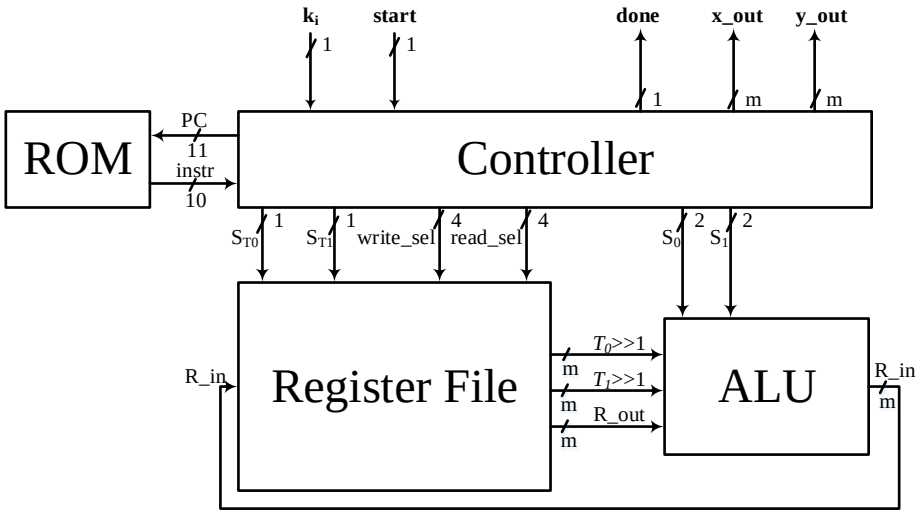

Figure 4.6: GNB top-level control unit.

However, with two registers being designated as multiplication registers, an extra register is needed for swapping in the value of d_1 for a multiplication with D^2. The other three registers would be holding (U,V,S). Thus, the formulas require five registers with the Co-Z trick implementation.

For unified access to constants and not impact the retrieval, the registers and constants are co-located in the register file. However, since this implementation targets a future standardization of a binary Edwards curve, the idea was that a starting point and curve parameters would be strictly defined. Therefore, there is no reason to add flexibility to the parameters of the base point or d_1. Hardwiring these coordinates to the register file provides the advantage that they can be used on-the-fly and that no extra control is necessary to bring these into the register file. For instance, [15] uses a small and external RAM chip to hold these constants. Such a design requires extra interfacing and extra cycles to load the value into the register file. After NIST standards for ECC are revised, hardwiring the constants in a place close to the register file is the best solution to save power and area.

The register file is random access to values including the constants. A register is written to when write is enabled and the multiplexer for writing selects that register.

4.2.3 Control Unit

The control unit handles the multiplexers for reading, writing, and performing operations. The four operations are ADD, SQ, MULT, and SWAP. The control unit uses a Finite State Machine to switch between these operations. A program counter is sent to an external ROM device that

feeds in the current instruction. Instructions are ten bits long. The first two bits indicate which instruction is being used. The next four bits indicate the input register. This value does not matter for multiplication. The last four bits indicate the output register.

The key is never stored in the control unit, such as how it was in [28]. The controller signals the master device to provide the next bit as the Montgomery Ladder [22] is being performed. Special SWAP instructions that depend on the key were left inside the controller to handle each step of the ladder, depending on the provided bit of the key. The subroutine for a step on the Montgomery Ladder with the corresponding register usage is shown below in Table 4.4. Table 4.4 shows the registers after each instruction. Six multiplications are required for each step.

Table 4.4: Point addition and doubling register usage.

#	Op	T_0	T_1	R_0	R_1	R_2
1	ADD T0 T1	W_1	W_1+W_2	Z		
2	SQ T1 R1	W_1	W_1+W_2	Z	C	
3	SWAP T1 R0	W_1	Z	W_1+W_2	C	
4	SQ T1 R0	W_1	Z	D	C	
5	ADD T0 T1	W_1	W_1+Z	D	C	
6	MULT T1 T0	$W_1 \cdot (W_1+Z)$	W_1+Z	D	C	
7	SQ T0 R2	$W_1 \cdot (W_1+Z)$	W_1+Z	D	C	S
8	SWAP R1 T1	$W_1 \cdot (W_1+Z)$	C	D	W_1+Z	S
9	SWAP R3 T0	$\frac{1}{w_0}$	C	D	W_1+Z	S
10	MULT T0 R1	$\frac{1}{w_0}$	C	D	E	S
11	ADD R1 T1	$\frac{1}{w_0}$	U	D	E	S
12	ADD R0 R1	$\frac{1}{w_0}$	U	D	E	S
13	SQ R0 R0	$\frac{1}{w_0}$	U	D^2	V	S
14	SWAP R0 T1	$\frac{1}{w_0}$	D^2	U	V	S
15	SWAP R4 T0	d	D^2	U	V	S
16	MULT T1 T0	$d \cdot D^2$	D^2	U	V	S
17	ADD R2 T0	T	D^2	U	V	S
18	SWAP R0 T1	T	U	D^2	V	S
19	MULT T0 T1	T	W_3	D^2	V	S
20	SWAP T1 R1	T	V	D^2	W_3	S
21	MULT T0 T0	Z'	V	D^2	W_3	S
22	SWAP T0 R2	S	V	D^2	W_3	Z'
23	MULT T0 T1	S	W_4	D^2	W_3	Z'
24	SWAP R0 R2	S	W_4	Z'	W_3	D^2
25	SWAP T0 R1	W_3	W_4	Z'	S	D^2

To save area, the half-trace functionality was left as a series of squarings and additions. Adding additional area to handle the half-trace saves a relatively small fraction of instructions but adds an additional multiplexer select in the FAU.

Inversion and the half-trace were implemented as subroutines within the ROM for instructions. The half traces uses a repetitive combination of double SQ then ADD. This was used to recover the x and y-coordinates of the final point. The special property of the half-trace as

Initialization
1: $T_0 \leftarrow x_1$
2: $T_0 \leftarrow T_0 + y_1$
3: $T_1 \leftarrow T_0^2$
4: $T_1 \leftarrow T_1 + T_0$
5: $R_0 \leftarrow T_1^2$
6: $T_1 \leftarrow d$
7: $T_1 \leftarrow T_1 + R_0$
8: $T_0 \leftarrow T_0 \times T_1$
9: $T_1, R_0 \leftarrow R_0, T_1$
Reg: $W_1, W_2, Z, 0, 0$
Mont. Ladder
Reg: $W_3, W_4, Z', 0, 0$
Recover w_2 **and** w_3
1: $R_1, T_0 \leftarrow T_0, R_1$
2: $T_1, R_2 \leftarrow R_2, T_1$
3: $T_0, R_0 \leftarrow R_0, T_0$
4: $T_0 \leftarrow T_0^{-1}$
5: $T_1, R_2 \leftarrow R_2, T_1$
6: $R_2 \leftarrow T_0 \times T_1$
7: $T_1, R_1 \leftarrow R_1, T_1$
8: $T_1 \leftarrow T_0 \times T_1$
9: $T_0 \leftarrow x_0$
10: $T_0 \leftarrow T_0 + y_0$
Reg: $w_0, w_2, 0, 0, w_3$
Recover x_2
1: $R_0 \leftarrow T_0 \times T_1$
2: $T_0 \leftarrow T_0 + R_0$
3: $T_0 \leftarrow T_0 + T_1$
4: $T_1, R_0 \leftarrow R_0, T_1$
5: $T_0 \leftarrow T_0 \times T_1$
6: $T_0 \leftarrow T_0 + d$
7: $T_1, R_2 \leftarrow R_2, T_1$
8: $R_2 \leftarrow T_0 \times T_1$
9: $T_0 \leftarrow x_0$
10: $T_0 \leftarrow T_0 + y_0$
11: $T_0 \leftarrow T_0 + R_0$
12: $T_0 \leftarrow T_0 + R_0$
13: $T_1 \leftarrow d$
14: $T_0 \leftarrow T_0 \times T_1$
15: $R_2 \leftarrow R_2 + T_0$
16: $T_0 \leftarrow y_1^2$
17: $T_0 \leftarrow T_0 + y_1$
18: $T_1 \leftarrow R_0^2$
19: $T_1 \leftarrow T_1 + R_0$
20: $T_0 \leftarrow T_0 \times T_1$
21: $R_2 \leftarrow R_2 + T_0$
22: $T_0 \leftarrow x_0$
23: $T_0 \leftarrow T_0 + y_0$
24: $T_1 \leftarrow T_0^2$
25: $T_0 \leftarrow T_0 + T_1$
26: $T_0 \leftarrow T_0^{-1}$
27: $T_1, R_2 \leftarrow R_2, T_1$
28: $T_0 \leftarrow T_0 \times T_1$
29: $T_0 \leftarrow halfTr(T_0)$
Reg: $x_2, 0, 0, 0, 0$
Recover y_2
1: $R_2 \leftarrow T_0^2$
2: $R_2 \leftarrow R_2 + T_0$
3: $R_1, T_0 \leftarrow T_0, R_1$
5: $T_0 \leftarrow T_0 + R_2$
6: $T_0 \leftarrow T_0^{-1}$
7: $T_1, R_2 \leftarrow R_2, T_1$
8: $T_0 \leftarrow T_0 \times T_1$
9: $T_1 \leftarrow d$
10: $T_0 \leftarrow T_0 \times T_1$
11: $T_0 \leftarrow \text{halfTr}(T_0)$
12: $T_2 \leftarrow Z$
13: $T_0, T_1 \leftarrow T_1, T_0$
14: $T_0, R_1 \leftarrow R_1, T_0$
Reg: $x_2, y_2, 0, 0, 0$

Figure 4.7: Main program listing for point multiplication using binary Edwards curves.

a bit-wise XOR was not selected for this application because that would add more complexity to the FAU and the half-trace requires relatively little time as it does not use a multiplication. Inversion was used to obtain $w_i = \frac{W_i}{Z_i}$, recover the x-coordinate, and recover the y-coordinate. Itoh-Tsujii inversion algorithm [13] was used to reduce the number of multiplications. For $\mathbb{F}_{2^{283}}$, the addition chain (1,2,4,8,16,17,34,35,70,140,141,282) was used. By implementing these repeated functionalities as subroutines, the number of instructions in the ROM is dramatically reduced. The main program is shown in Fig. 4.7. The subroutines for inversion in $\mathbb{F}_{2^{283}}$ and the half-trace are shown in Fig. 4.8. The total instruction count of the point multiplier for $\mathbb{F}_{2^{283}}$ is shown in Table 4.5. Approximately 132, 10-bit instructions were needed.

Inversion

1: $T_1 \leftarrow T_0^2$
2: $T_1 \leftarrow T_0 \times T_1$
3: $T_0, R_0 \leftarrow R_0, T_0$
4: $T_0 \leftarrow T_1^2$
5: $T_0 \leftarrow T_0^2$
6: $T_0 \leftarrow T_0 \times T_1$
7: $T_0 \leftarrow T_1^2$
8: $T_0 \leftarrow T_0^{2^3}$
9: $T_0 \leftarrow T_0 \times T_1$
10: $T_0 \leftarrow T_1^2$
11: $T_0 \leftarrow T_0^{2^7}$
12: $T_0 \leftarrow T_0 \times T_1$
13: $T_1, R_0 \leftarrow R_0, T_1$
14: $T_0 \leftarrow T_0^2$
15: $T_0 \leftarrow T_0 \times T_1$
16: $T_1, R_0 \leftarrow R_0, T_1$
17: $T_0 \leftarrow T_1^2$
18: $T_0 \leftarrow T_0^{2^{16}}$
19: $T_0 \leftarrow T_0 \times T_1$
20: $T_1, R_0 \leftarrow R_0, T_1$
21: $T_0 \leftarrow T_0^2$
22: $T_0 \leftarrow T_0 \times T_1$
23: $T_1, R_0 \leftarrow R_0, T_1$
24: $T_0 \leftarrow T_1^2$
25: $T_0 \leftarrow T_0^{2^{34}}$
26: $T_0 \leftarrow T_0 \times T_1$
27: $T_0 \leftarrow T_1^2$
28: $T_0 \leftarrow T_0^{2^{69}}$
29: $T_0 \leftarrow T_0 \times T_1$
30: $T_1, R_0 \leftarrow R_0, T_1$
31: $T_0 \leftarrow T_0^2$
32: $T_0 \leftarrow T_0 \times T_1$
33: $T_0 \leftarrow T_1^2$
34: $T_0 \leftarrow T_0^{2^{140}}$
35: $T_0 \leftarrow T_0 \times T_1$
36: $T_0 \leftarrow T_0^2$

Half-Trace

1: $T_1 \leftarrow T_0^2$
2: $T_1 \leftarrow T_1^2$
3: $T_0 \leftarrow T_0 + T_1$
4: $T_1 \leftarrow T_1^2$
5: $T_1 \leftarrow T_1^2$
6: $T_0 \leftarrow T_0 + T_1$

*Repeat steps 4-6 $\frac{m-2}{2}$ times

Figure 4.8: Itoh-Tsujii [13] inversion ($\mathbb{F}_{2^{283}}$) and half-trace subroutines.

Table 4.5: Necessary subroutines for point multiplication.

Subroutine	Iterations	#ADD	#SQ	#MULT	#SWAP	Latency (cc)
Init	1	3	2	1	4	310
Step	281	5	4	6	10+2[1]	494,841
x Recovery	1	16	5	9	13	2,649
y Recovery	1	3	2	3	6	882
Half Trace	2×141	1	2	0	0	2×1,269
Inversion	3×1	0	282	11	6	3×3,977
Total		1,705	2,540	1,730	3,410	512,555

1. Special SWAP's that the controller handles.

4.3 Comparison and Discussion

This design was synthesized using Synopsys Design Compiler in $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{233}}$, and $\mathbb{F}_{2^{163}}$, each a different standardized binary field size by NIST [34]. The TSMC 65-*n*m CMOS standard technology and CORE65LPSVT standard cell library were used for results. This implementation was optimized for area.

The area was converted to Gate Equivalent (GE), where the size of a single NAND gate is considered 1 GE. For our particular technology library, the size of a synthesized NAND gate was 1.4 μm^2, so this was used as the conversion factor. Latency reports the total number of cycles to compute the final coordinates of a point multiplication. Parameters such as the type of curve used and if Montgomery Ladder were used to indicate some innate security properties of the curve. Power and energy results were not included as a comparison because they are dependent on the underlying technology, frequency of the processor, and testing methodology. The comparison results are shown in Table 4.6.

This ECC implementation over BEC does make a few assumptions that not necessarily each of these other implementations make. This architecture's area does not include the ROM to hold the instructions. The ROM was not synthesized, but approximately 165 bytes of ROM were required. By the estimate that 1,426 bytes is equivalent to 2,635 GE in [6], 165 bytes of ROM is roughly equivalent to 274 GE. This architecture assumes that each bit of the key will be fed into the co-processor. These assumptions are explained in previous sections. The areas of the implementation for $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, and $\mathbb{F}_{2^{283}}$ excluding the register file and program ROM are 3,248 GE, 3,788 GE, and 5,566 GE, respectively.

Looking at timing for these implementation, the number of clock cycles appears to rise quadratically when comparing $\mathbb{F}_{2^{163}}$ to $\mathbb{F}_{2^{283}}$. This is to be expected, as the Montgomery Ladder performs 6 multiplications each step. A multiplication takes m clock cycles and there are $m-2$ steps.

The area appears to have a linear relationship. This is also to be expected, as the register file's size increases linearly. The area of the FAU depends on the underlying finite field and the area of the controller is fairly constant. The area of the FAU and controller for $\mathbb{F}_{2^{233}}$ is only a slight increase over the area of the FAU and controller for $\mathbb{F}_{2^{163}}$ because the $\mathbb{F}_{2^{233}}$ is type II GNB, in contrast to $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{283}}$ are type IV GNB. Therefore the p' block in $\mathbb{F}_{2^{233}}$ requires much fewer XOR gates.

The underlying architecture of this implementation was similar to [28]. This implementation uses more area because an additional register and two additional constants were used in the register file. However, one less multiplexer was required in the FAU since the neutral

Table 4.6: Comparison of different point bit-level multiplications targeted for ASIC.

Work	Curve	Ladder?	Field Size	Tech (nm)	Mult.	# of clock Cycles	Coord.	Area (GE)
[39], 2007	BGC	✓	$\mathbb{F}_{2^{163}}$	180	Bit-serial	313,901	Projective	13,182
[38], 2008	BGC	✓	$\mathbb{F}_{2^{163}}$	130	Bit-serial	275,816	Mixed	12,506
[15], 2010	BEC	✓	$\mathbb{F}_{2^{163}}$	130	Bit-serial	219,148	Mixed	11,720
[35], 2011	BGC	✓	$\mathbb{F}_{2^{163}}$	130	Comb-serial	286,000	Projective	8,958
[28], 2014	BKC	×	$\mathbb{F}_{2^{163}}$	65	Bit-serial	106,700	Affine	10,299
[25], 2014	BGC	×	$\mathbb{F}_{p^{160}}$	130	Comb-serial	139,930	Projective	12,448[1]
[32], 2015	BKC	×	$\mathbb{F}_{2^{283}}$	130	Comb-serial	1,566,000	Projective	10,204[2] (4,323)[3]
This work	BEC	✓	$\mathbb{F}_{2^{163}}$	65	Bit-serial	177,707	Mixed	10,945[4]
		✓	$\mathbb{F}_{2^{233}}$			351,856		14,903[4]
		✓	$\mathbb{F}_{2^{283}}$			512,555		19,058[4]

1. Includes a Keccak module to perform ECDSA
2. RAM results were not synthesized, but extrapolated from a different implementation.
3. Area excluding RAM
4. Area excluding ROM. Approximately 274 GE more with ROM.

element in GNB was not required in any formulas. Other than that, the implementation in [28] does not use the Montgomery Ladder and performs over Koblitz curves, which speeds up the point multiplication at the cost of some security.

The only other light-weight implementation of BEC point multiplication is found in [15]. Many of the internals of our point multiplier are different. For instance, this implementation uses a circular register structure, and also a different bit-serial multiplier in Polynomial Basis. A Polynomial Basis parallel squaring unit was used in this implementation, which is costly when compared to the GNB. This implementation uses Common-Z differential coordinate system for the Montgomery Ladder, but each step requires 8 multiplications. Our implementation requires only 6 multiplications, representing a reduction of latency in the Montgomery Ladder by approximately 25%. Lastly, this implementation requires a register file to hold 6 registers, whereas our register file only requires 5 registers. Hence, our implementation features a smaller and faster point multiplication scheme than that in [15].

The introduction of extremely area-efficient crypto-processors with comb-serial multiplication schemes [21] like the one proposed in [32] indicates that there is a need for new trade-off for future implementations of these ECC targeted at RFID chips. Bit-parallel multiplication architectures are among the fastest approaches to perform finite field multiplications, but this requires a tremendous amount of area. Digit-serial schemes require a factor more of cycles, but use less area. The most popular scheme for RFID chip point multiplication is bit-serial, which requires a fraction of the area of digit-serial and requires m cycles to perform a multiplication. Comb-serial multiplication takes this a step further by performing small multiplications over many small combs. Depending on the multiplication scheme, this could require more than m cycles but holds new records for area-efficiency. The work presented in [32] is among the smallest ECC co-processors, even in $\mathbb{F}_{2^{283}}$. It was designed as a drop-in concept, such that the co-processor can share RAM blocks with a microcontroller. This implementation utilizes a comb-serial multiplicationscheme in polynomial basis over Koblitz curves. As such, the latency of each operation is larger than that of this work. Field addition, squaring, and multiplication require 60, 200, and 829 cycles, respectively. This implementation needs space to hold 14 intermediate elements throughout the point multiplication operation. Including the constants, our implementation requires 9 intermediate values. The area of the co-processor without the RAM for the register file is 4,323 GE. Moreover, in [32], the RAM results that were included were extrapolated from a different implementation of ECC appeared in [6]. With these extrapolated results, the total area of the co-processor would be 10,204 GE. Our crypto-processor with the register file uses 87% more area, but performs the point multiplication approximately three times faster, reducing the need to run at higher speeds to meet timing

requirements in a device. Further, [32] utilizes zero-free tau-adic expansion to enforce a constant pattern of operations, similar to the Montgomery ladder [22], to protect against timing and power analysis attacks. However, this new technique has not been thoroughly explored like the Montgomery ladder. Furthermore, the co-processor does not have any protection against exceptional points attacks such as the ones presented in [33]. In summary, for higher levels of security as was implemented in [32], the time complexity was several factors higher, but the area was comparable to an implementation of a smaller finite field. As there is a push for larger field sizes for higher security levels, the time complexity of the comb-serial method of multiplication and other operations becomes inefficient.

Chapter 5

Conclusion

In this thesis, it is shown that new mixed w-coordinate differential addition and doubling formulas for binary Edwards curve produce a fast, small, and secure implementation of point multiplication. Corrected formulas for addition in this coordinate system have been provided and proven. Binary Edwards curves feature a complete and unified addition formula. The future of point multipliers targeted at RFID technology depends on the trade-offs among area, latency, and security. The binary Edwards curves implementation presented in this thesis has demonstrated that BEC is highly-competitive with the dominant elliptic curve systems standardized by NIST and IEEE. As such, new standardizations that include binary Edwards curves are necessary for the future of elliptic curve cryptography. The detailed analysis in this thesis also suggests that binary Edwards curves are among the fastest and most secure curves for point multiplication targeting resource-constrained devices.

References

[1] Hariri Arash. Arithmetic units for the elliptic curve cryptography with concurrent error detection capability. *Electronic Thesis and Dissertation Repository. Paper*, (75), 2010.

[2] D. W. Ash, I. F. Blake, and S. A. Vanstone. Low Complexity Normal Bases. *Discrete Applied Mathematics*, 25(3):191–210, 1989. ISSN 0166-218X.

[3] Reza Azarderakhsh and Arash Reyhani-Masoleh. A Modified Low Complexity Digit-Level Gaussian Normal Basis Multiplier. In *Proceedings of Third International Workshop on Arithmetic of Finite Fields (WAIFI 2010)*, volume 6087, pages 25–40, 2010.

[4] D. J. Bernstein, T. Lange, and R. R. Farashahi. Binary Edwards Curves. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008)*, volume 5154, pages 244–265, 2008.

[5] DanielJ. Bernstein, Tanja Lange, and Reza Rezaeian Farashahi. Binary edwards curves. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems, CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 244–265. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-85052-6. doi: 10.1007/978-3-540-85053-3_16. URL http://dx.doi.org/10.1007/978-3-540-85053-3_16.

[6] E. Wenger. Hardware architectures for MSP430-based wireless sensor nodes performing elliptic curve cryptography. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 290–306. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38979-5. doi: 10.1007/978-3-642-38980-1_18. URL http://dx.doi.org/10.1007/978-3-642-38980-1_18.

[7] E. Wenger and M. Hutter. Exploring the design space of prime field vs. binary field ecc-hardware implementations. In Peeter Laud, editor, *Information Security Technology for Applications*, volume 7161 of *Lecture Notes in Computer Science*, pages 256–

271. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29614-7. doi: 10.1007/978-3-642-29615-4_18. URL http://dx.doi.org/10.1007/978-3-642-29615-4_18.

[8] Harold M. Edwards. A normal form for elliptic curves. In *Bulletin of the American Mathematical Society*, pages 393–422.

[9] R. Farashahi and M. Joye. Efficient Arithmetic on Hessian Curves. In *Proceedings of The 13th International Conference on Practice and Theory of Public Key Cryptography (PKC 2010)*, pages 243–260, 2010.

[10] G.-L. Feng. A vlsi architecture for fast inversion in gf(2m). *IEEE Transactions onComputers*, 38(10):1383–1386, Oct 1989.

[11] D. R. Hankerson, S. A. Vanstone, and A. J. Menezes. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York Inc, 2004. ISBN 038795273X.

[12] IEEE Std. 1363-3/D1. "Draft Standard for Identity-based public key cryptography using pairings". January 2008.

[13] 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.

[14] K. Kim, C. Lee, and C. Negre. Binary edwards curves revisited. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology – INDOCRYPT 2014*, pages 393–408. Springer International Publishing, 2014. ISBN 978-3-319-13038-5. doi: 10.1007/978-3-319-13039-2_23. URL http://dx.doi.org/10.1007/978-3-319-13039-2_23.

[15] U. Kocabas, J. Fan, and I. Verbauwhede. Implementation of Binary Edwards Curves for Very-Constrained Devices. In *Proceedings of 21st International Conference on Application-specific Systems Architectures and Processors (ASAP 2010)*, pages 185–191, 2010.

[16] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology - CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66347-8. doi: 10.1007/3-540-48405-1_25. URL http://dx.doi.org/10.1007/3-540-48405-1_25.

[17] J. Lopez 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.

[18] J.L. Massey and J.K. Omura. Computational Method and Apparatus for Finite Arithmetic. *US Patent*, (4587627), 1986.

[19] Friedemann Mattern and Christian Floerkemeier. From active data management to event-based systems and more. chapter From the Internet of Computers to the Internet of Things, pages 242–259. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-17225-3, 978-3-642-17225-0. URL http://dl.acm.org/citation.cfm?id=1985625.1985645.

[20] A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullin, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. Kluwer Academic Publishers, 1993.

[21] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

[22] P. L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of computation*, pages 243–264, 1987.

[23] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. Optimal Normal Bases in $GF(p^n)$. *Discrete Appl. Math.*, 22(2):149–161, 1989. ISSN 0166-218X.

[24] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 3642041000, 9783642041006.

[25] Peter Pessl and Michael Hutter. Curved tags - a low-resource ecdsa implementation tailored for rfid. In Nitesh Saxena and Ahmad-Reza Sadeghi, editors, *Radio Frequency Identification: Security and Privacy Issues*, volume 8651 of *Lecture Notes in Computer Science*, pages 156–172. Springer International Publishing, 2014. ISBN 978-3-319-13065-1. doi: 10.1007/978-3-319-13066-8_10. URL http://dx.doi.org/10.1007/978-3-319-13066-8_10.

[26] R. Azarderakhsh A. Reyhani-Masoleh. Efficient FPGA implementations of point multiplication on binary Edwards and generalized Hessian curves using Gaussian normal basis. *IEEE Trans. Very Large Scale Integr. Syst.*, 20(8):1453–1466, August 2012. ISSN 1063-8210. doi: 10.1109/TVLSI.2011.2158595. URL http://dx.doi.org/10.1109/TVLSI.2011.2158595.

[27] R. Azarderakhsh, D. Jao, and H. Lee. Common subexpression algorithms for space-complexity reduction of Gaussian normal basis multiplication. *IEEE Transactions on Information Theory*, 61(5):2357–2369, 2015.

[28] R. Azarderakhsh, K. U. Jarvinen, and M. Mozaffari Kermani. Efficient algorithm and architecture for elliptic curve cryptography for extremely constrained secure applications. *IEEE Trans. on Circuits and Systems*, 61-I(4):1144–1155, 2014.

[29] A. Reyhani-Masoleh. Efficient Algorithms and Architectures for Field Multiplication Using Gaussian Normal Bases. *IEEE Transactions on Computers*, 55(1):34–47, 2006.

[30] Arash Reyhani-Masoleh and M. Anwarul Hasan. A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$. *IEEE Transactions on Computers*, 51(5):511–520, 2002.

[31] Francisco Rodríguez-Henríquez, N. A. Saqib, A. Díaz-Pèrez, and Cetin Kaya Koc. *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387338837.

[32] S. S. Roy, K. Jarvinen, and I. Verbauwhede. Lightweight coprocessor for Koblitz curves: 283-bit ECC including scalar conversion with only 4300 gates. Cryptology ePrint Archive, Report 2015/556, 2015. http://eprint.iacr.org/.

[33] T. Izu and T. Takagi. Exceptional procedure attack on elliptic curve cryptosystems. In YvoG. Desmedt, editor, *Public Key Cryptography, PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 224–239. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-00324-3. doi: 10.1007/3-540-36288-6_17. URL http://dx.doi.org/10.1007/3-540-36288-6_17.

[34] U.S. Department of Commerce/NIST. National Institute of Standards and Technology. *Digital Signature Standard, FIPS Publications 186-2*, January 2000.

[35] Erich Wenger and Michael Hutter. A hardware processor supporting elliptic curve cryptography for less than 9 kges. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 182–198. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-27256-1. doi: 10.1007/978-3-642-27257-8_12. URL http://dx.doi.org/10.1007/978-3-642-27257-8_12.

[36] Huapeng Wu. Bit-parallel finite field multiplier and squarer using polynomial basis. *IEEE Trans. Comput.*, 51(7):750–758, July 2002. ISSN 0018-9340. doi: 10.1109/TC.2002.1017695. URL http://dx.doi.org/10.1109/TC.2002.1017695.

[37] Xi Xiong and Haining Fan. $gf(2^n)$ bit-parallel squarer using generalized polynomial basis for a new class of irreducible pentanomials. Cryptology ePrint Archive, Report 2014/003, 2014. http://eprint.iacr.org/.

[38] Y. K. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. Elliptic-Curve-Based Security Processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, 2008.

[39] Y. Lee and I. Verbauwhede. A compact architecture for montgomery elliptic curve scalar multiplication processor. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications*, volume 4867 of *Lecture Notes in Computer Science*, pages 115–127. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-77534-8. doi: 10.1007/978-3-540-77535-5_9. URL http://dx.doi.org/10.1007/978-3-540-77535-5_9.

Appendix A

Implementation Code Listing

A.1 Sage Scripts

These scripts are run with Sage math. Sage math is based on Python, but has been designed to work well with many number theoretic functions, such as finite-field arithmetic. The setup script, in Algorithms A.1 and A.2, contains most of the preliminaries for the setup of BEC, and only the a few sets of equations need to be added where <add code here> is.

Algorithm A.1 Sage setup script for *w*-coordinate differential addition (1)

```
def mynumerator(x):
  if parent(x) == R:
    return x
  return numerator(x)
class fastfrac:
  def __init__(self,top,bot=1):
    if parent(top) == ZZ or parent(top) == R:
      self.top = R(top)
      self.bot = R(bot)
    elif top.__class__ == fastfrac:
      self.top = top.top
      self.bot = top.bot * bot
    else:
      self.top = R(numerator(top))
      self.bot = R(denominator(top)) * bot
  def reduce(self):
    return fastfrac(self.top / self.bot)
  def sreduce(self):
    return fastfrac(I.reduce(self.top),I.reduce(self.bot))
  def iszero(self):
    return self.top in I and not (self.bot in I)
  def isdoublingzero(self):
    return self.top in J and not (self.bot in J)
  def __add__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top + self.bot * other,self.bot)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.bot + self.bot * other.top,self.bot * other.bot)
    return NotImplemented    def __sub__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top - self.bot * other,self.bot)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.bot - self.bot * other.top,self.bot * other.bot)
    return NotImplemented
  def __neg__(self):
    return fastfrac(-self.top,self.bot)
  def __mul__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top * other,self.bot)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.top,self.bot * other.bot)
    return NotImplemented    def __rmul__(self,other):
    return self.__mul__(other)    def __div__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top,self.bot * other)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.bot,self.bot * other.top)
    return NotImplemented    def __pow__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top ^ other,self.bot ^ other)
    return NotImplemented
```

Algorithm A.2 Sage setup script for *w*-coordinate differential addition (2)

```
def isidentity(x):
  return x.iszero()
def isdoublingidentity(x):
  return x.isdoublingzero()
R.<ud,ud2overd1plus1,ux3,uy3,ux2,uy2,uw1,uw2,uw3> = PolynomialRing(GF(2),9,order='invlex')
ux_2 = (uy2)
uy_2 = (ux2)
ux1 = (((ud*(ux3+ux_2)+ud*(ux3+uy3)*(ux_2+uy_2)+(ux3+ux3^2)*(ux_2*(uy3+uy_2+1)+uy3*uy_2))
/(ud+(ux3+ux3^2)*(ux_2+uy_2)))
uy1 = (((ud*(uy3+uy_2)+ud*(ux3+uy3)*(ux_2+uy_2)+(uy3+uy3^2)*(uy_2*(ux3+ux_2+1)+ux3*ux_2))
/(ud+(uy3+uy3^2)*(ux_2+uy_2)))
I = R.ideal([    mynumerator((ud*(ux1+uy1)+ud*(ux1^2+uy1^2))-((ux1+ux1^2)*(uy1+uy1^2)))
, mynumerator((ux1+uy1)-(uw1))
,mynumerator((ud*(ux2+uy2)+ud*(ux2^2+uy2^2))-((ux2+ux2^2)*(uy2+uy2^2)))
, mynumerator((ux2+uy2)-(uw2))
, mynumerator((ud*(ux3+uy3)+ud*(ux3^2+uy3^2))-((ux3+ux3^2)*(uy3+uy3^2)))
, mynumerator((ux3+uy3)-(uw3)) , mynumerator((ud2overd1plus1)-(ud/ud+1)) ])
ud = fastfrac(ud)
ud2overd1plus1 = fastfrac(ud2overd1plus1)
ux3 = fastfrac(ux3)
uy3 = fastfrac(uy3)
ux2 = fastfrac(ux2)
uy2 = fastfrac(uy2)
uw1 = fastfrac(uw1)
uw2 = fastfrac(uw2)
uw3 = fastfrac(uw3)
ux_2 = fastfrac(ux_2)
uy_2 = fastfrac(uy_2)
ux1 = fastfrac(ux1)
uy1 = fastfrac(uy1)


<INSERT TEST CODE HERE FOR DIFFERENTIAL ADDITION>


ux5 = (((ud*(ux3+ux2)+ud*(ux3+uy3)*(ux2+uy2)+(ux3+ux3^2)*(ux2*(uy3+uy2+fastfrac(1))+uy3*uy2))
/(ud+(ux3+ux3^2)*(ux2+uy2)))).reduce()
uy5 = (((ud*(uy3+uy2)+ud*(ux3+uy3)*(ux2+uy2)+(uy3+uy3^2)*(uy2*(ux3+ux2+fastfrac(1))+ux3*ux2))
/(ud+(uy3+uy3^2)*(ux2+uy2)))).reduce()
print isidentity((ud*(ux5+uy5)+ud*(ux5^2+uy5^2))
-((ux5+ux5^2)*(uy5+uy5^2))) or isidentity(uy1*uy1*uy2*uy3*ux1*ux1*ux2*ux3
*((ud*(ux5+uy5)+ud*(ux5^2+uy5^2))-((ux5+ux5^2)*(uy5+uy5^2))))
print isidentity((ux5+uy5)-(uw5)) or isidentity(uy1*uy1*uy2*uy3*ux1*ux1*ux2*ux3*((ux5+uy5)-(uw5)))
```

The code in Algorithm A.3 demonstrates that the proposed equations in K. Kim, C. Lee, and C. Negre [14] were incorrect, most likely caused by an algebra mistake.

Algorithm A.3 Sage script disproving *w*-coordinate differential addition presented in K. Kim, C. Lee, and C. Negre [14]

```
uN = (uw2^2+uw3^2)
uD = (uw2^2+uw3^2)/uw1 + fastfrac(1)
uw5 = fastfrac(1)+uN/uD
```

The code in Algorithm A.4 proves that the *w*-coordinate addition formulas proposed in this thesis are valid.

Algorithm A.4 Sage script proving explicit formula 3.7 for *w*-coordinate differential addition

```
uA = (uw2+uw3)^2
uB = uA/uw1
uN = uA+uB
uD = uB + fastfrac(1)
uE = fastfrac(1)/uD
uw5 = uN*uE
```

The next Sage scripts prove the formulas in the mixed *w*-coordinate Differential Addition system. Again, this sets up the curve and automatically converts back to affine coordinates at the end of the program to verify the formulas. Algorithms A.5 and A.6 demonstrate the setup code.

Algorithm A.5 Sage setup script for WZ coordinate system (1)

```
def mynumerator(x):
  if parent(x) == R:
    return x
  return numerator(x)

class fastfrac:
  def __init__(self,top,bot=1):
    if parent(top) == ZZ or parent(top) == R:
      self.top = R(top)
      self.bot = R(bot)
    elif top.__class__ == fastfrac:
      self.top = top.top
      self.bot = top.bot * bot
    else:
      self.top = R(numerator(top))
      self.bot = R(denominator(top)) * bot
  def reduce(self):
    return fastfrac(self.top / self.bot)
  def sreduce(self):
    return fastfrac(I.reduce(self.top),I.reduce(self.bot))
  def iszero(self):
    return self.top in I and not (self.bot in I)
  def isdoublingzero(self):
    return self.top in J and not (self.bot in J)
  def __add__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top + self.bot * other,self.bot)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.bot + self.bot * other.top,self.bot * other.bot)
    return NotImplemented
  def __sub__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top - self.bot * other,self.bot)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.bot - self.bot * other.top,self.bot * other.bot)
    return NotImplemented
  def __neg__(self):
    return fastfrac(-self.top,self.bot)
  def __mul__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top * other,self.bot)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.top,self.bot * other.bot)
    return NotImplemented
def __rmul__(self,other):
    return self.__mul__(other)
def __div__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top,self.bot * other)
    if other.__class__ == fastfrac:
      return fastfrac(self.top * other.bot,self.bot * other.top)
    return NotImplemented
  def __pow__(self,other):
    if parent(other) == ZZ:
      return fastfrac(self.top ^ other,self.bot ^ other)
    return NotImplemented
```

Algorithm A.6 Sage setup script for WZ coordinate system (2)

```
def isidentity(x):
  return x.iszero()
def isdoublingidentity(x):
  return x.isdoublingzero()

R.<ud1,ud2,ux3,uy3,ux2,uy2,uW1,uZ1,uW2,uZ2,uW3,uZ3> = PolynomialRing(GF(2),12,order='invlex')
ux_2 = (uy2)
uy_2 = (ux2)
ux1 = ((ud1*(ux3+ux_2)+ud2*(ux3+uy3)*(ux_2+uy_2)+(ux3+ux3^2)*(ux_2*(uy3+uy_2+1)+uy3*uy_2))
/(ud1+(ux3+ux3^2)*(ux_2+uy_2)))
uy1 = ((ud1*(uy3+uy_2)+ud2*(ux3+uy3)*(ux_2+uy_2)+(uy3+uy3^2)*(uy_2*(ux3+ux_2+1)+ux3*ux_2))
/(ud1+(uy3+uy3^2)*(ux_2+uy_2)))
I = R.ideal([    mynumerator((ud1*(ux1+uy1)+ud2*(ux1^2+uy1^2))-((ux1+ux1^2)*(uy1+uy1^2)))
, mynumerator((ux1+uy1)-(uW1/uZ1))
, mynumerator((ud1*(ux2+uy2)+ud2*(ux2^2+uy2^2))-((ux2+ux2^2)*(uy2+uy2^2)))
, mynumerator((ux2+uy2)-(uW2/uZ2))
, mynumerator((ud1*(ux3+uy3)+ud2*(ux3^2+uy3^2))-((ux3+ux3^2)*(uy3+uy3^2)))
, mynumerator((ux3+uy3)-(uW3/uZ3))
, mynumerator((ud1)-(ud2))
, mynumerator((uZ2)-(uZ3))
, mynumerator((uZ1)-(1)) ])
ud1 = fastfrac(ud1)
ud2 = fastfrac(ud2)
ux3 = fastfrac(ux3)
uy3 = fastfrac(uy3)
ux2 = fastfrac(ux2)
uy2 = fastfrac(uy2)
uW1 = fastfrac(uW1)
uZ1 = fastfrac(uZ1)
uW2 = fastfrac(uW2)
uZ2 = fastfrac(uZ2)
uW3 = fastfrac(uW3)
uZ3 = fastfrac(uZ3)
ux_2 = fastfrac(ux_2)
uy_2 = fastfrac(uy_2)
ux1 = fastfrac(ux1)
uy1 = fastfrac(uy1)


<INSERT CODE HERE FOR WZ DIFFERENTIAL ADDITION>


ux5 = (((ud1*(ux3+ux2)+ud2*(ux3+uy3)*(ux2+uy2)+(ux3+ux3^2)*(ux2*(uy3+uy2+fastfrac(1))+uy3*uy2))
/(ud1+(ux3+ux3^2)*(ux2+uy2)))).reduce()
uy5 = (((ud1*(uy3+uy2)+ud2*(ux3+uy3)*(ux2+uy2)+(uy3+uy3^2)*(uy2*(ux3+ux2+fastfrac(1))+ux3*ux2))
/(ud1+(uy3+uy3^2)*(ux2+uy2)))).reduce()
print isidentity((ud1*(ux5+uy5)+ud2*(ux5^2+uy5^2))
-((ux5+ux5^2)*(uy5+uy5^2))) or isidentity(uy1*uy1*uy2*uy3*ux1*ux1*ux2*ux3*
((ud1*(ux5+uy5)+ud2*(ux5^2+uy5^2))-((ux5+ux5^2)*(uy5+uy5^2))))
print isidentity((ux5+uy5)-(uW5/uZ5)) or isidentity(uy1*uy1*uy2*uy3*ux1*ux1*ux2*ux3
*((ux5+uy5)-(uW5/uZ5)))
```

The code in Algorithm A.7 proves the mixed *w*-coordinate addition formula in the WZ coordinate system.

Algorithm A.7 Sage script proving explicit formula 3.12 for mixed *w*-coordinate differential addition

```
uC  =  (uW2*uZ3+uW3*uZ2)^2
uD  =  (uZ2*uZ3)^2
uE  =  uC/uW1
uW5=  uE  +  uC
uZ5  =  uE  +  uD
```

The code in Algorithm A.8 proves that the mixed *w*-coordinate differential addition works with the Co-Z trick and is more efficient.

Algorithm A.8 Sage script proving explicit formula 3.17 for mixed *w*-coordinate differential addition with the co-Z trick

```
uC  =  (uW2+uW3)^2
uD  =  uZ2^2
uE  =  uC/uW1
uW5  =  uE  +  uC
uZ5  =  uE  +  uD
```

A.2 Subroutines

This contains a code listing of the program in assembly.

Algorithm A.9 shows the Itoh-Tsujii Itoh and Tsujii [13] inversion subroutine for $\mathbb{F}_{2^{283}}$. This follows the addition chain (1,2,4,8,16,17,34,35,70,140,141,282). Eleven multiplications are required for this binary field. A similar approach was done for $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$.

Algorithm A.9 Itoh-Tsujii Itoh and Tsujii [13] inversion subroutine for $GF(2^{283})$

```
SQ T0 T1
MULT T1 T1 –2^2-1
SWAP T0 R0
SQ T1 T0
SQ T0 T0
MULT T1 T0 –2^4-1
SQ T0 T1
SQ T1 T1 3 Times
MULT T1 T0 –2^8-1
SQ T0 T1
SQ T1 T1 7 Times
MULT T1 T0 –2^16-1
SWAP T1 R0
SQ T0 T0
MULT T1 T0 –2^17-1
SWAP T1 R0
SQ T0 T1
SQ T1 T1 16 Times
MULT T1 T0 –2^34-1
SWAP T1 R0
SQ T0 T0
MULT T1 T0 –2^35-1
SWAP T1 R0
SQ T0 T1
SQ T1 T1 34 Times
MULT T1 T0 –2^70-1
SQ T0 T1
SQ T1 T1 69 Times
MULT T1 T0 –2^140-1
SWAP T1 R0
SQ T0 T0
MULT T1 T0 –2^141-1
SQ T0 T1
SQ T1 T1 140 Times
MULT T1 T0 –2^282-1
SQ T0 T0
```

Algorithm A.10 shows the half-trace subroutine. This is a simple double square and add routine that produces the result after $\frac{m-1}{2}$ iterations.

Algorithm A.10 Half-trace subroutine

SQ T0 T1
SQ T1 T1
ADD T1 T0
{SQ T1 T1
SQ T1 T1
ADD T1 T0} loop for $\frac{m-2}{2}$ times

Algorithm A.11 shows the beginning of the main program that was used. This includes the initialization of the point and the repeated step of the Montgomery ladder Montgomery [22].

Algorithm A.11 General program flow (1)

INIT
SWAP R5 T0
ADD R6 T0
SQ T0 T1
ADD T0 T1
SQ T1 R0 –W4
SWAP R4 T1
ADD R0 T1 –Z4
MULT T1 T0 –W1 revised
SWAP T1 R0 –W1 W4 Z4
STEP
SWAP T0 T0 –OUTPUT register selected by k bit
ADD T0 T1
SQ T1 R1
SWAP T1 R0
SQ T1 R0
ADD T0 T1
MULT T1 T0
SQ T0 R2 –S
SWAP R1 T1
SWAP R3 T0 –1/w0
MULT T0 R1 –E
ADD R1 T1 –U
ADD R0 R1 –V
SQ R0 R0
SWAP R0 T1
SWAP R4 T0 –d1
MULT T1 T0
ADD R2 T0 –T
SWAP R0 T1
MULT T0 T1 –W3
SWAP T1 R1
MULT T0 T0 –Z'
SWAP T0 R2
MULT T0 T1 –W4
SWAP R0 R2
SWAP T0 R1
SWAP T0 T0 –Output register selected by k bit. Repeat for every step

Algorithms A.12 and A.13 show the end of the main program that was used. This includes the recovery of w_2, w_3, x_2, y_2.

Algorithm A.12 General program flow (2)

RECOVER X
SWAP T0 R1
SWAP T1 R2
SWAP R0 T0
Invert T0 –1/Z
SWAP R2 T1
MULT T1 R2 –w3
SWAP R1 T1
MULT T1 T1 –w2
SWAP R5 T0
ADD R6 T0
MULT T1 R0
ADD R0 T0
ADD T1 T0
SWAP R0 T1 –(w1w2+w1+w2) w1w2 w2 0 w3
MULT T1 T0
ADD T1 T0
ADD R4 T0 –d1+w1w2+w1w2*(w1+w2+w1w2) w1w2 w2 0 w3
SWAP T1 R2
MULT T1 R2 –1st part of the numerator – 0 0 w2 0 1st
SWAP R5 T0
ADD R6 T0
ADD R0 T0 –w1+w2
SWAP R4 T1
MULT T1 T0
ADD T0 R2 –0 0 w2 0 1st+2nd
SQ R6 T0
ADD R6 T0
SQ R0 T1
ADD R0 T1
MULT T1 T0
ADD T0 R2 –Numerator complete, compute inversion now
SWAP R5 T0
ADD R6 T0 –w1 0 0 0 Numerator
SQ T0 T1
ADD T1 T0 –w1^2+w1 0 0 0 Numerator, now inversion
Invert T0 –1/(w0^2+w0)
SWAP R2 T1
MULT T1 T0
T0 = HalfTrace(T0) –x2 or x2+1

Algorithm A.13 General Program Flow (3)

RECOVER Y
SQ T0 R2
ADD T0 R2
SWAP T0 R1
SWAP R4 T0
ADD R2 T0
Invert T0 –1/(d+x+x^2)
SWAP T1 R2
MULT T1 T0
SWAP R4 T1
MULT T1 T0
T0 = HalfTrace(T0) –y2 or y2+1
SWAP T0 T1
SWAP R1 T0 –Solution is x, y in T0 T1