Area Flexible $GF(2^k)$ Elliptic Curve Cryptography Coprocessor

ADNAN ABDUL-AZIZ GUTUB

Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, SAUDI ARABIA

gutub@kfupm.edu.sa

Abstract. Elliptic curve cryptography (ECC) is popularly defined either over GF(p) or $GF(2^k)$. This research modifies a GF(p) multiplication algorithm to make it applicable for $GF(2^k)$. Both algorithms, the GF(p) and $GF(2^k)$ one, are designed in hardware to be compared. The $GF(2^k)$ multiplier is found faster and small. This $GF(2^k)$ multiplier is further improved to benefit in speed, it gained more than 40% faster speed with the cost of 5% more area. This multiplier is adjusted to have the area flexibility feature, which is used as the basic block in modeling a complete projective coordinate ECC coprocessor.

Keywords. Elliptic Curve Cryptography, modular multiplication, area flexible multiplier, projective coordinates arithmetic

1 Introduction

Cryptography is a well-recognized technique for information protection. It is used effectively to protect sensitive data such as passwords that are stored in a computer, as well as information being transmitted through different communication media. Encryption is the transformation of data into a form, which is very hard to retransform back by anyone without a secret decryption key. Even if someone steals the encrypted information, he cannot benefit from it.

Depending on the encryption/decryption key, crypto-systems can be classified into two main categories: secret key cryptosystems and public key cryptosystems. The secret key cryptosystems uses one key for both encryption and decryption. Public key cryptosystems, however, use two different keys, one for encryption and the other for decryption. Secret key cryptosystems is used for encryption and decryption of messages, while, public key cryptosystems is used for digital signature and key exchange schemes. Although public key systems can be used for message encryption and decryption, it is found to be very slow. This made it practical to be used for digital signature and key exchange scheme more than encryption and decryption of messages.

In 1985, Koblitz and Miller independently proposed the Elliptic Curve Cryptosystem (ECC) [1,2,3,4,5,6,7,8,9], a method based on the Discrete Logarithm problem over the points on an elliptic curve. Since that time, ECC has received considerable attention from mathematicians around the world, and no significant breakthroughs have been made in determining weaknesses in the algorithm. Although critics are still skeptical as to the reliability of this method, several encryption techniques have been developed recently using these properties. The fact that the problem appears so difficult to crack means that key sizes can be reduced in size considerably, even exponentially [2,5,8], especially when compared to the key size used by other cryptosystems. This made ECC become a challenge to the RSA, one of the most popular public key methods known. ECC is showing to offer equal security to RSA but with much smaller key size. In addition to their simplicity and smaller size, ECC are more likely to be adopted in the future, especially in systems with limited processing and storage resources such as those incorporating mobile devices [2].

In order to use ECC, an elliptic curve must be defined over a specific finite field. Some finite field representations may lead to more efficient implementations than others, in hardware or in software. The elliptic curve arithmetic can be optimized depending on the type of finite field. The most popular finite fields used in ECC are Galois Fields, *GF(p)* and $GF(2^k)$ [1,2,3,5,10]. In this research, these two finite fields are compared. The number of arithmetic operations in *GF(p)* is found to be less than $GF(2^k)$, but each operation in *GF(p)* consumes much more time and area than $GF(2^k)$. This made up working with $GF(2^k)$ seams to be more efficient than *GF(p)*.

A basic operation in the ECC arithmetic is modular multiplication. This research adjust a *GF(p)* multiplication algorithm to make it applicable for $GF(2^k)$. Both algorithms are modeled in hardware to be compared and analyzed. The $GF(2^k)$ multiplier hardware is found to be faster and smaller. It is further modified and speeded up with a very small cost of area. The $GF(2^k)$ multiplier hardware is designed with a re-configurable area dependant way, such that the hardware is area flexible, it can be designed according to what area is available.

The flow of this paper is as follows; the following section will give some background on the elliptic curve theory and some efficient implementation study of ECC. Then, the *GF(p)* multiplication algorithm will be modified for $GF(2^k)$ which made up the basic multiplier hardware. After that, a complete ECC coprocessor model is proposed.

2 Elliptic Curves and Their Implementations

Elliptic curves are known so because they are described by cubic equations, similar to those used in ellipsis calculations. The general form for elliptic curve equation is: $y^2 + axy + by = x^3 + cx^2 + dx + e$. There is also a single element named the *point at infinity* or the *zero point* denoted 'φ'. The point at infinity is computed as the sum of any three points on an elliptic curve that lie on a straight line. If a point on the elliptic curve is to be added to another point on the curve or to itself, some special addition rules are applied depending on the finite field used. For more details on the elliptic curve theory, the reader is advised to look through [2,5,7,9].

A finite field is a set of elements that have a finite order (number of elements). The order of *Galois Field* (*GF*) is normally a prime number or a power of a prime number. There are many ways of representing the elements of the finite field. Some representations may lead to more efficient implementations of the field arithmetic in hardware or in software. The elliptic curve arithmetic is more or less complex depending on the finite field where the elliptic curve is applied. $GF(p)$ and $GF(2^k)$ are considered in this research because of their popularity in ECC [1,2,3,5,15,16,17].

The basic element of an elliptic curve cryptosystem is the calculation of the point kP, where $kP = P+P...+P$ *(k-times)*. Designing an efficient ECC hardware coprocessor depends tremendously on the type of finite field used. Most ECC hardware researches [4,6,8], show that the ECC implementations defined over $GF(2^k)$ are more suitable than $GF(p)$, especially when embedded into restricted area, such as smart cards. The simplicity of $GF(2^k)$ in hardware comes from the possibility of doing arithmetic without carry propagation. ECC, however, is not restricted to smart cards. There can be some hardware applications where $GF(p)$ can be more appropriate to use than $GF(2^k)$ [3].

The following subsections are going to compare the elliptic curve point addition in the two finite fields: $GF(p)$ and $GF(2^k)$. The comparison is targeted towards finding out if $GF(p)$ can possibly be faster than $GF(2^k)$. This study will compare the number of the most-time-consuming operations in both finite fields. The costly arithmetic operations are assumed to be multiplication, inversion (division), and squaring. Addition, subtraction, and multiplication by small constants are not expensive [1,9], so their cost is neglected.

2.1 Comparing GF(p) and GF(2^k)

The addition of two different points on the elliptic curve is computed as shown in Table 1. The number of operations, as observed, is found to be the same in both fields (neglecting the addition, subtraction, and multiplication of small numbers [1,9]). Lambda requires one inversion and one multiplication in order to be calculated. Computing 'x_3' needs only one squaring of lambda. The value of 'y_3' is figured with one multiplication operation of lambda. The number of operations in both fields is the same: one inversion, one squaring, and two multiplication calculations.

$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$; where $x_1 \neq x_2$	
$GF(p)$	$GF(2^k)$
$\lambda = (y_2 - y_1)/(x_2 - x_1)$ $x_3 = \lambda^2 - x_1 - x_2$ $y_3 = \lambda(x_1 - x_3) - y_1$	$\lambda = (y_2 + y_1)/(x_2 + x_1)$ $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$

Table 1 addition of two different points on the elliptic curve

The addition of a point to itself (doubling a point) on the elliptic curve is computed as shown in Table 2. Computing lambda in $GF(p)$ requires an inversion, a multiplication, and a squaring of x_1, while it needs an inversion and a multiplication in $GF(2^k)$. Calculating 'x_3' in both fields want the same operation of squaring lambda. Computing 'y_3' in $GF(p)$ requires only one multiplication, while it needs a multiplication and a squaring operations in $GF(2^k)$. The number of operations is found to be the same in both fields: an inversion, two squaring, and two multiplication calculations [1,9].

In point operations of both finite fields $GF(p)$ and $GF(2^k)$ (adding two points and doubling a point), the number of operations is found to be the same. However, the time required to complete each operation differ much in each field.

$(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$; where $x_1 \neq 0$	
$GF(p)$	$GF(2^k)$
$\lambda = (3(x_1)^2 + a)/(2y_1)$ $x_3 = \lambda^2 - 2x_1$ $y_3 = \lambda(x_1 - x_3) - y_1$	$\lambda = x_1 + (y_1)/(x_1)$ $x_3 = \lambda^2 + \lambda + a$ $y_3 = (x_1)^2 + (\lambda + 1)x_3$

Table 2 doubling a point on the elliptic curve

Crutchley [5] in his master thesis have made a time comparison between $GF(p)$ and $GF(2^k)$. His result shows that the multiplication process in $GF(p)$ is the only faster operation than $GF(2^k)$. All other operations are reported to be slower in $GF(p)$ than in $GF(2^k)$. It seems that his conclusions are generated from a software implementation of the operations. Hardware multipliers are found to be faster in $GF(2^k)$ than in $GF(p)$, as studied in a following section.

Calculating the inverse is the most expensive operation. Designs replace the inversion by several multiplication operations by representing the elliptic curve points as projective coordinate points [1,4,7,9].

2.2 Projective Coordinates in GF(p)

The projective coordinates are used to eliminate the need for performing inversion. For elliptic curve defined over $GF(p)$, two different forms of formulas are found [1,9] for point addition and doubling. One form projections $(x,y)=(X/Z^2,Y/Z^3)$ [9], while the second projects $(x,y)=(X/Z,Y/Z)$ [1].

The two forms procedures for projective point addition of $P+Q$ (two elliptic curve points) is shown below:

$$P=(X_1,Y_1,Z_1);Q=(X_2,Y_2,Z_2);P+Q=(X_3,Y_3,Z_3);\ where\ P \neq \pm Q$$

$(x,y)=(X/Z^2,Y/Z^3) \Rightarrow (X,Y,Z)$		$(x,y)=(X/Z,Y/Z) \Rightarrow (X,Y,Z)$	
$\lambda_1 = X_1Z_2^2$	2M	$\lambda_1 = X_1Z_2$	1M
$\lambda_2 = X_2Z_1^2$	2M	$\lambda_2 = X_2Z_1$	1M
$\lambda_3 = \lambda_1 - \lambda_2$		$\lambda_3 = \lambda_2 - \lambda_1$	
$\lambda_4 = Y_1Z_2^3$	2M	$\lambda_4 = Y_1Z_2$	1M
$\lambda_5 = Y_2Z_1^3$	2M	$\lambda_5 = Y_2Z_1$	1M
$\lambda_6 = \lambda_4 - \lambda_5$		$\lambda_6 = \lambda_5 - \lambda_4$	
$\lambda_7 = \lambda_1 + \lambda_2$		$\lambda_7 = \lambda_1 + \lambda_2$	
$\lambda_8 = \lambda_4 + \lambda_5$		$\lambda_8 = \lambda_6^2 Z_1Z_2 - \lambda_3^2\lambda_7$	5M
$Z_3 = Z_1Z_2\lambda_3$	2M	$Z_3 = Z_1Z_2\lambda_3^3$	2M
$X_3 = \lambda_6^2 - \lambda_7\lambda_3^2$	3M	$X_3 = \lambda_8\lambda_3$	1M
$\lambda_9 = \lambda_7\lambda_3^2 - 2X_3$		$\lambda_9 = \lambda_3^2 X_1Z_2 - \lambda_8$	1M
$Y_3 = (\lambda_9\lambda_6 - \lambda_8\lambda_3^3)/2$	3M	$Y_3 = \lambda_9\lambda_6 - \lambda_3^3 Y_1Z_2$	2M
	-----		-----
	16 M		**15M**

Similarly, the two forms of formulas for projective point doubling is shown below:

$$P = (X_1,Y_1,Z_1);\ P+P = (X_3,Y_3,Z_3)$$

$(x,y)=(X/Z^2, Y/Z^3) \rightarrow (X,Y,Z)$		**$(x, y) = (X/Z, Y/Z) \rightarrow (X,Y,Z)$**	
$\lambda_1 = 3X_1^2 + aZ_1^4$	4M	$\lambda_1 = 3X_1^2 + aZ_1^2$	2M
$Z_3 = 2Y_1Z_1$	1M	$\lambda_2 = Y_1Z_1$	1M
$\lambda_2 = 4X_1 Y_1^2$	2M	$\lambda_3 = X_1Y_1\lambda_2$	2M
$X_3 = \lambda_1^2 - 2\lambda_2$	1M	$\lambda_4 = \lambda_1^2 - 8\lambda_3$	1M
$\lambda_3 = 8Y_1^4$	1M	$X_3 = 2\lambda_4\lambda_2$	1M
$\lambda_4 = \lambda_2 - 2X_3$		$Y_3=\lambda_1(4\lambda_3-\lambda_4)-8(Y_1\lambda_2)^2$	3M
$Y_3 = \lambda_1\lambda_4 - \lambda_3$	1M	$Z_3 = 8\lambda_2^3$	2M
	------		-----
	10M		**12M**

The squaring calculation over *GF(p)* is very similar to the multiplication computation. They both are noted as *M* (multiplication). It is worth noting that any EC crypto processor must implement the procedures of projective coordinates efficiently since they are the core steps of the point operation algorithm of ECC.

2.3 Projective Coordinates in GF(2^k)

For elliptic curve defined over $GF(2^k)$, to eliminate the need for performing inversion, its coordinates (x, y) are to be projected to (X, Y, Z). Similar in principle to *GF(p)* projective coordinates, two different forms of formulas are found [9,18] for point addition and doubling. One form projects $(x,y)=(X/Z^2,Y/Z^3)$ [9], while the second projects $(x,y)=(X/Z,Y/Z)$ [18].

The two forms procedures for projective point addition of $P+Q$ (two elliptic curve points) is shown below:

$$P=(X_1,Y_1,Z_1);Q=(X_2,Y_2,Z_2);P+Q=(X_3,Y_3,Z_3);\ where\ P \neq \pm Q$$

$(x,y)=(X/Z^2,Y/Z^3) \rightarrow (X,Y,Z)$		**$(x,y)=(X/Z,Y/Z) \rightarrow (X,Y,Z)$**	
$A = X_1Z_2^2$	2M	$A = X_1Z_2$	1M
$B = X_2Z_1^2$	2M	$B = X_2Z_1$	1M
$C = A+B$		$C = A+B$	
$D = Y_1Z_2^3$	2M	$D = Y_1Z_2$	1M
$E = Y_2Z_1^3$	2M	$E = Y_2Z_1$	1M
$F = D+E$		$F = D+E$	
$G = Z_1C$	1M	$G= C+F$	
$H = FX_2+GY_2$	2M	$H= Z_1Z_2$	1M
$Z_3 = GZ_2$	1M	$I=C^3+aHC^2+HFG$	6M
$I = F+Z_3$		$X_3 = CI$	1M
$X_3= aZ_3^2+IF+C^3$	5M	$Z_3 = HC^3$	1M
$Y_3= IX_3+HG^2$	3M	$Y_3=GI+C^2[FX_1+CY_1]$	5M
	-----		-----
	20 M		**17M**

Similarly, the two forms of formulas for projective point doubling are shown below:

$$P = (X_1,Y_1,Z_1);\ P+P = (X_3,Y_3,Z_3)$$

$(x,y)=(X/Z^2, Y/Z^3) \rightarrow (X,Y,Z)$		**$(x, y) = (X/Z, Y/Z) \rightarrow (X,Y,Z)$**	
$Z_3=X_1Z_1^2$	2M	$A=X_1Z_1$	1M
$A = bZ_1^2$	1M	$B= bZ_1^4+X_1^4$	5M
$B = X_1+A$		$C= AX_1^4$	1M
$X_3 = B^4$	2M	$D=Y_1Z_1$	1M
$C = Z_1Y_1$	1M	$E=X_1^2+D+A$	
$D=Z_3+X_1^2+C$	1M	$Z_3=A^3$	2M
$E = DX_3$	1M	$X_3=AB$	1M
$Y_3 = X_1^4Z_3+E$	2M	$Y_3= C+BE$	1M
	------		-----
	10M		**12M**

The squaring calculation over $GF(2^k)$ is assumed very similar to the multiplication computation. They are both denoted as *M* (multiplication) in the above. Since the number of additions is taken to be, on the average, half the number of bits, it can be clearly seen form the above tables that the projective coordinate $(x,y) = (X/Z^2,Y/Z^3)$ has on the average 20 multiplication iteration, while the projection $(x,y) = (X/Z,Y/Z)$ has on the average 20.5 multiplications. Clearly, the former would be the projection of choice for sequential implementation. However, as will be discussed later, the projection $(x,y) = (X/Z,Y/Z)$ has an advantage for parallel implementation.

2.4 Remarks

As can be observed from before, the number of multiplication processes for adding and doubling two EC points in *GF(p)* is found to be different than $GF(2^k)$. Furthermore, this number is different depending on the specific procedure used in the field, i.e. $(X/Z,Y/Z)$ or $(X/Z^2,Y/Z^3)$. Comparison of the number of operations to chose the proper finite field is not accurate because operations in *GF(p)* require completely different calculation time than $GF(2^k)$.

The following section will implement two modulo multiplication hardware for both fields *GF(p)* and $GF(2^k)$. The two hardware designs will be compared depending on speed and area to compute ECC arithmetic computations.

3 Modular Multiplication Hardware

The straightforward approach to compute modular multiplication is by performing multiplication followed by reduction [2,7]. The multiplication can be computed through several addition operations. Then, the reduction is performed through several subtractions, by subtracting the modulus several times until the result is less than the modulus. This approach is inefficient and suffers from very low speed. It can, however, be improved by merging modulo subtraction with the multiplication-add operations [11], as Algorithm 1.

Algorithm 1:

Define k: number of bits in x; x_i: the i^{th} bit of x
Input: x,y, and n; where x,y < n;
Output: P = xy mod n

```
1.  P := 0;
2.  For i = k-1 down to 0;
3.      {
4.      P:= 2P;
5.      If P ≥ n Then P:= P – n ;
6.      If xi = 1 Then
7.          {P := P + y;
8.          If P ≥ n Then P:= P – n};
9.      }
10. End;
```

Algorithm 1 is developed for *GF(p)*. In order to use this algorithm for $GF(2^k)$, the carry propagation is not needed any more. All the addition and subtraction operations are replaced by exclusive-OR (XOR) computations. The ‘if’ statement in Step 8 of Algorithm 1 is not required, because the result of XOR-ing P with y can not be more than the modulus. Algorithm 1 can be modified to be used for $GF(2^k)$ as shown in Algorithm 2.

Algorithm 2:

Define k: number of bits in x; x_i: the i^{th} bit of x
Input: x,y, and f(x); where $x,y,f(x) \in GF(2^k)$
Output: P = xy mod f(x)

1. *P := 0;*
2. *For i = k-1 down to 0;*
3. *{*
4. *P:= 2P;*
5. *If P_k =1 Then P:= P⊕f(x) ;*
6. *If x_i = 1 Then P := P⊕y;*
7. *}*
8. *End;*

These two algorithms are implemented in hardware in the following subsections. The purpose of the implementations is for proper multiplication comparison between *GF(p)* and *GF(2^k)* finite fields.

3.1 GF(p) Modular Multiplication Hardware

Algorithm 1 for *GF(p)* modulo multiplication is found to be very suitable for VLSI implementation [11]. It has a bounding 'for' loop, which includes iterative modulo multiplication reduction operations. The bounding loop can be designed in hardware as a controller that will control the number and processes of the iterations. The modulo multiplication reduction is implemented in hardware with three adders and three multiplexors connected as shown in Figure 1. There are no registers in the design, the small boxes shown are only to show the correct mapping of bit-flow. The adder can function as a subtractor if one of its inputs is inverted. The complete process of *x.y mod n* will need *k* clock cycles, if each modulo reduction iteration is performed in one clock cycle.

The multiplication of *P* by two (as in step 4 of Algorithm 1) is performed by a shift to the bits of *P* toward the left. The multiplexors: Mux-1, and Mux-3, are controlled by the subtractor's output-carry-bit. Therefore, the complete subtractions are to be made for the Mux to give the output. Assume '*k*' is the number of bits we are working with. The simplest adder, carry ripple adder, is constructed from *k-1* full adders (FA) and one half adder (HA) [12,13]. Each FA is built of two XOR gates, two AND gates, and one OR gate [12]. The HA is constructed of an XOR gate, and an AND gate. The subtractor is different than the adder with the addition of *k* NOT gates. If we use the NOT gate as our gate reference, as described in [14], each XOR gate is equivalent to three gates, and each AND and OR gate is the same as two gates [14]. The Adder's area will be equivalent to *12k-7* gates. The subtractor will have an area similar to *13k-7* gates.

Each multiplexor is made of *2k* AND gates, *k* OR gates and one NOT gate [13]. The area of the multiplexor is equivalent to *6k+1* NOT gates. The complete hardware shown in Figure 1 is constructed of the following gates:

(6k-3) XOR+(12k-3)AND+(6k-3)OR+(2k+3)NOT,

which is equivalent to *56k-18* NOT gates. For the reason of comparison, assume the delay is constant for different gates. The longest path in the adder and the subtractor is found to be through *2k* gates, because of the carry propagation. The multiplexor's delay is through three gates. The complete *GF(p)* hardware longest path is found to be through three adders and three multeplexors, which made it to be through *6k+9* gates.

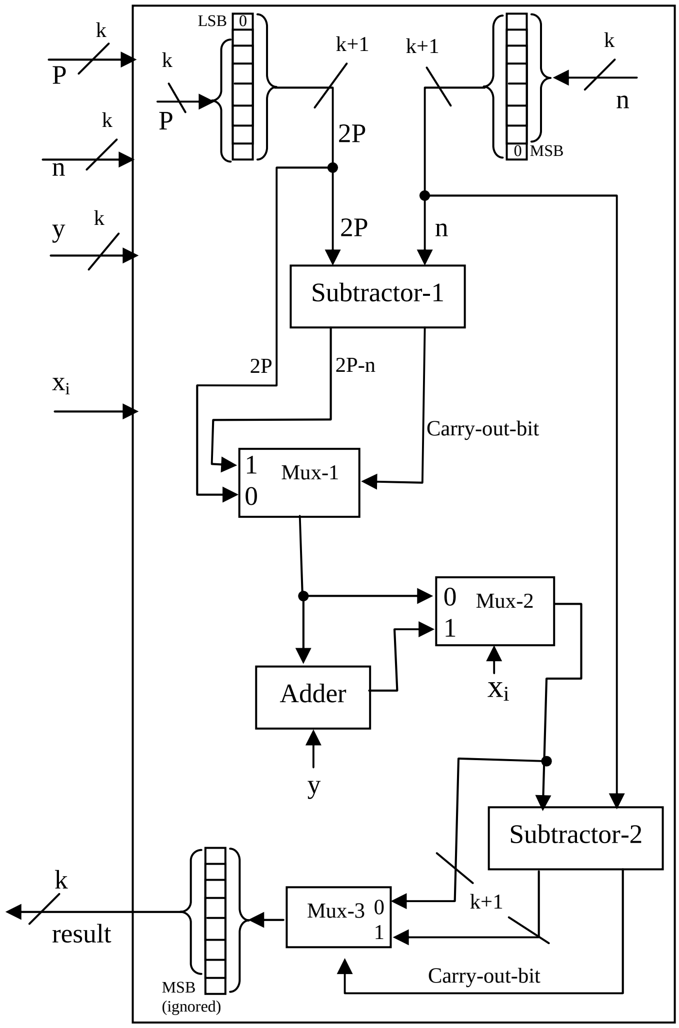


Figure 1: *GF(p)* modulo reduction hardware

3.2 GF(2^k) Modular Multiplication Hardware

Algorithm 2, is a modification of Algorithm 1 to make it suitable for *GF(2^k)*. It has a similar bounding 'for' loop the same as Algorithm 1, which make the controller of *GF(2^k)* the same as *GF(p)*. The *GF(2^k)* modulo multiplication reduction hardware is shown in Figure 2. It requires only two multiplexors and two *k*-parallel XOR gates. Mux-1 and Mux-2 are to perform the 'if' comparisons of step 5 and 6 in Algorithm 2. The number of gates used in this hardware is *2k XOR+4k AND+2k OR+2 NOT*, which is equivalent to *18k-2* NOT gates.

As mentioned before, the addition operation of *GF(p)* is replaced by bit-wise XOR-ing which does not have any carry propagation. This made the area and speed of *GF(2^k)* multiplication more practical to be used. Referring to the same assumption of constant delay for different gates, the longest path for the *GF(2^k)* hardware shown in Figure 2 is found to be of only *7* gates.

More elaboration on the hardware shown in Figure 2 is performed to make it modified to be even faster. This modified design is shown in Figure 3. Its speed is accomplished with the cost of some more hardware area. The longest path is made shorter and is found to be through only *4* gates instead of *7*. This modified scheme requires three *k*-parallel XOR gates and a 4×1-multiplexor. The 4×1-multiplexor, as mentioned in [12], is made up of *4k AND + k OR + 2 NOT* gates. The *GF(*2^k*)* modified hardware area is found to be equivalent to *19k+2* NOT gates.

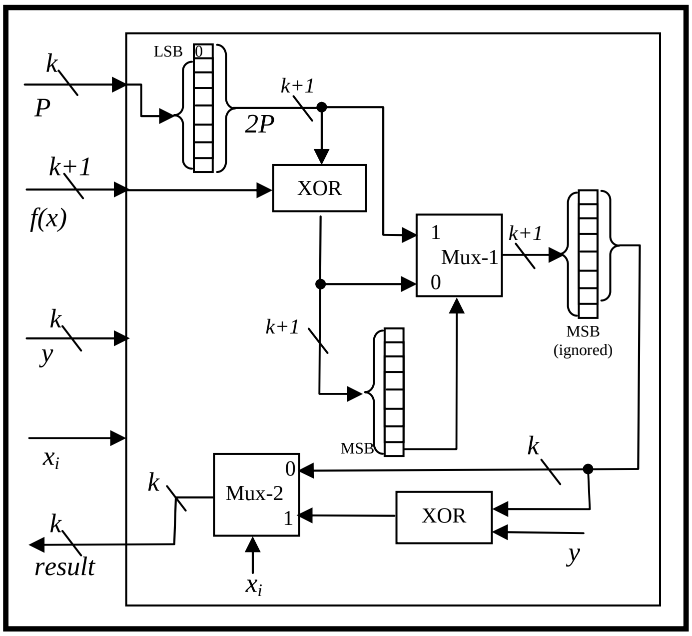


Figure 2: *GF(*2^k*)* modulo reduction hardware

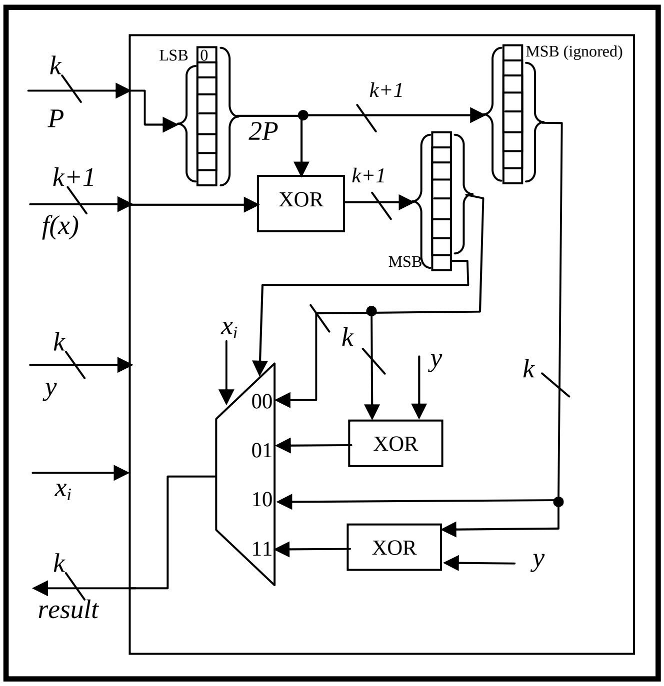


Figure 3: modified *GF(*2^k*)* modulo reduction hardware

3.3 Discussion

In order to compare the different hardware designs described before, their areas and delays are listed in Table 5. The table is expressed into the number of bits '*k*' used. Note that the multiplication in *GF(*2^k*)* requires a fixed amount of time. While multiplication in *GF(p)* depends on the number of bits used. If the numbers, for example, are expressed into 300-bits, the longest path in *GF(p)* is found to be through *1809* gates. This long path makes the design very slow when compared to the fixed longest path of *GF(*2^k*)*. The hardware area of *GF(p)* is found to be around three times larger than *GF(*2^k*)*.

Modulo Multiplication Hardware Design	Area (equivalent to NOT gates)	Delay (longest path # gates)
GF(p) (Figure 1)	*56k-18*	*6k+9*
GF(2^k*) (Figure 2)*	*18k-2*	*7*
GF(2^k*) (Figure 3)*	*19k+2*	*4*

Table 5 comparing the three multipliers

The main problem of *GF(p)* hardware designs, compared with *GF(*2^k*)*, is due to the carry propagation issue. This problem have been tried out to be solved by several methods. However, none of theses techniques can have the speed nor the area of *GF(*2^k*)* [12]. These reasons made the motivation to choose working with *GF(*2^k*)* instead of *GF(p)* for ECC hardware design.

3.4 Different Implementations of Algorithm 2

Referring to Algorithm 2 for *GF(*2^k*)* multiplication, the loop can be unfolded differently to design different multipliers. It can use any of the two modules of the modulo reduction designs shown in Figures 2 or 3. The area difference between the designs of Figure 2 and 3 is not much, Figure 3 is *5%* larger than Figure 2. However, the speed of the hardware shown in Figure 3 is *43%* faster than the one shown in Figure 2. The speed difference makes the choice of the hardware of Figure 3 to be the practical module to be used.

Algorithm 2 can be unfolded and implemented in several ways. A fully parallel way, where the number of reduction modules is the same as the number of bits, is described in the next subsection. Partially parallel methods will be introduced later.

Fully Parallel Hardware

Completely unfolding Algorithm 2 will give a fully parallel modulo multiplier. It is assumed that it performs its computation in one clock cycle. This parallel hardware is shown in Figure 4.

This *GF(*2^k*)* parallel multiplier is made of four registers and *k*-modulo reduction modules, assuming *k* is the number of bits of *x* and *y*. All registers are for holding *k*-bits except one, which is for *f(x)*, holding *k+1* bits. The register holding the value of *x* is mapping each bit to a different modulo reduction unit. The registers holding the bits of *y* and *f(x)* broadcast their bits to all modulo reduction modules. The longest path in this *GF(*2^k*)* parallel multiplier is found to be through *4k* gates. This longest path will define the length of the clock cycle needed.

The area of this design includes the register's area. The *k*-bit register is constructed of *k* D-Flip-Flops (DFF), where each DFF is made of six NAND gates as described in [14]. This makes all four registers to be made of *24k+6* NAND gates. Each NAND gate is equivalent to a NOT gate [13], which makes the registers area of $24k+6$ NOT gates. All the modulo reduction hardware area is equivalent to $19k^2+2k$ NOT gates. The complete $GF(2^k)$ fully parallel multiplier hardware is found to have an area equivalent to $19k^2+26k+6$ NOT gates. This area is very huge to implement. In order to design a feasible implementation a partially parallel hardware is developed.

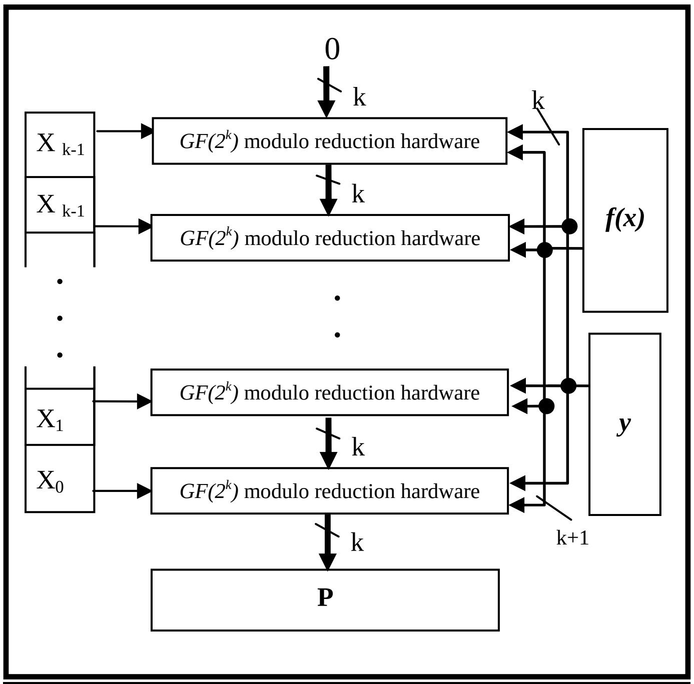


Figure 4: unfolded $GF(2^k)$ parallel multiplier

Partially Parallel Hardware

The data flow graph shown in Figure 4 can be implemented in hardware in several ways, depending on the size of the hardware to be implemented. For example, if the area available for the modular multiplier is equivalent to $50k$ NOT gates, only two reduction units can fit in this area. This means that this hardware will need $k/2$ clock cycles to complete a modulo multiplication process. For this research, the ECC coprocessor is investigated, and depending on it the proper partial parallel hardware is designed.

4 ECC Coprocessor

The $GF(2^k)$ partially parallel modular multiplier described before is used as the basic unit in an ECC coprocessor. Assume P is an elliptic curve point used for ECC. The ECC algorithm used for calculating nP from P is the binary method, since it is known to be efficient and practical to implement in hardware [2,5,7,9,10]. This binary method algorithm is shown below as Algorithm 3.

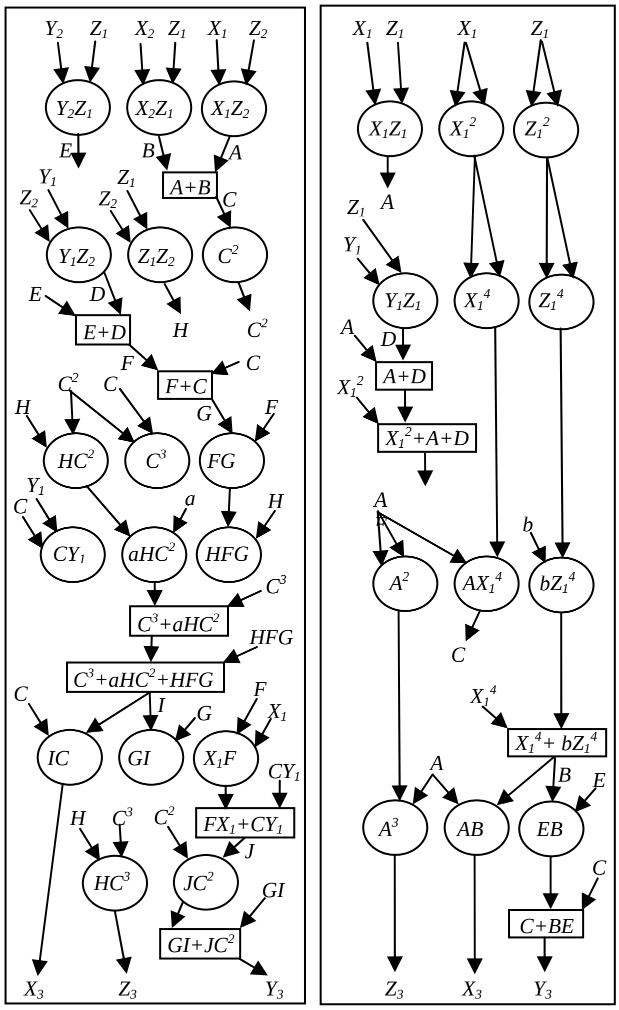
Algorithm 3:

Define k: number of bits in n; n_i: the i^{th} bit of n
Input: P (a point on the elliptic curve).
Output: $Q = nP$ (another point on the elliptic curve).

1. if $n_{k-1} = 1$, then $Q:=P$ else $Q:=0$;
2. for $i = k-2$ down to 0;
3. { $Q := Q+Q$;
4. if $n_i = 1$ then $Q:= Q+P$; }
5. return Q;

The binary method algorithm scans the binary bits of n and doubles the point Q k-times. Whenever, a particular bit of n is found to be one, an extra operation is needed. This extra operation is $Q+P$. Adding two elliptic curve points and doubling a point are to be performed in the projective coordinates system to avoid the inversion operation. The data flow graph for doubling a point is shown in Figure 5. It is made of ten multipliers and four k-bit XOR gates. Figure 6 shows the data flow graph for adding two elliptic curve points. The hardware of this design if implemented as shown in Figure 6 is made of twenty multipliers and seven k-bit XOR gates.

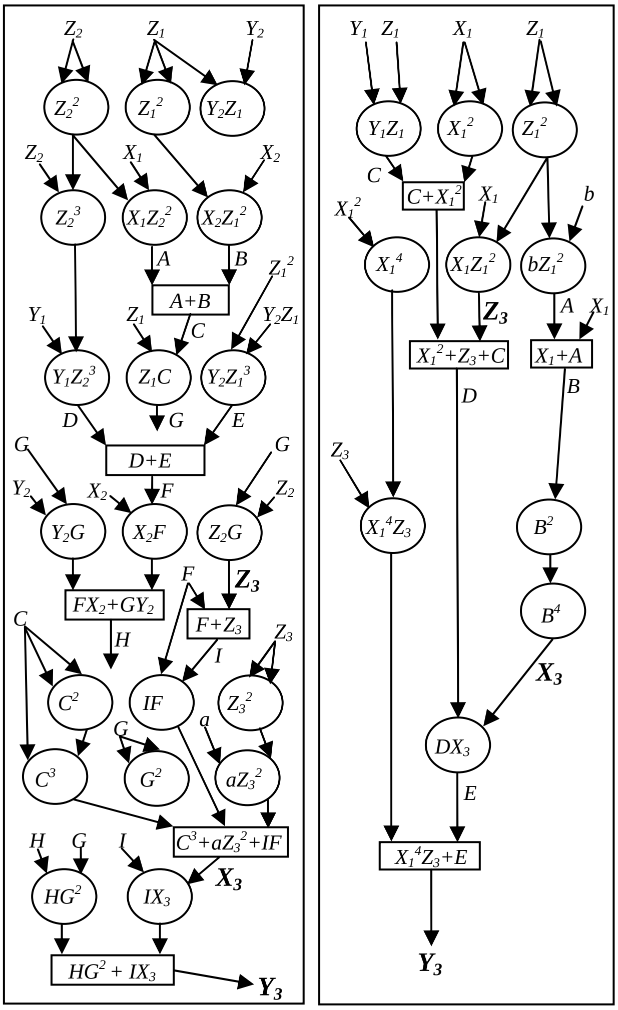
It is found to be unpractical to implement the elliptic curve point operations as shown in the Figures 5 or 6 for the different $GF(2^k)$ projective coordinate procedure forms.



(a) adding two points **(b) doubling a point**

Figure 5: data flow graphs for the elliptic curve point operations of projecting *(x,y)* to *(X/Z,Y/Z)*

The area of both figures is very large and the clock cycle will be inefficient. This made the idea of designing hardware models with a less number of small multipliers more suitable to do as will be clarified in the following subsections.



(a) adding two points **(b) doubling a point**

Figure 6: data flow graphs for the elliptic curve point operations of projecting *(x,y)* to *(X/Z^2,Y/Z^3)*

4.1 General Elliptic Curve Point Operation Hardware

Reconsider Algorithm 3 described before, the two operations to be repeated for k-iterations are doubling an elliptic curve point and adding a point to another. It is found that they cannot be performed in parallel (each of these computations is to be performed separately). This made the idea of designing a general point operation hardware.

The general elliptic curve point operation hardware is outlined in Figure 7. It is informed about the type of computation to be done through a signal ‘Slct’. If ‘Slct’ is high, the hardware is to do an elliptic curve doubling operation. If ‘Slct’ is low, the hardware is to do addition of two elliptic curve points. The hardware begins its operation when ‘Start’ signal is raised. The ‘Start’ signal also indicates to the hardware that all the input data values are available so the controller will command the registers to load them

The controller will then command the data flow to be routed to perform the proper elliptic curve computation. When the computation is complete a ‘Done’ flag is raised indicating that the results are ready at the data output pins.

The controller guides the hardware to perform one of the two operations, each at its appropriate time. The routing of data is performed through multiplexors to direct the data flow between the registers and the multipliers and XOR operation module. The controller is to influence the multiplexors selection signal and the registers loading signal.

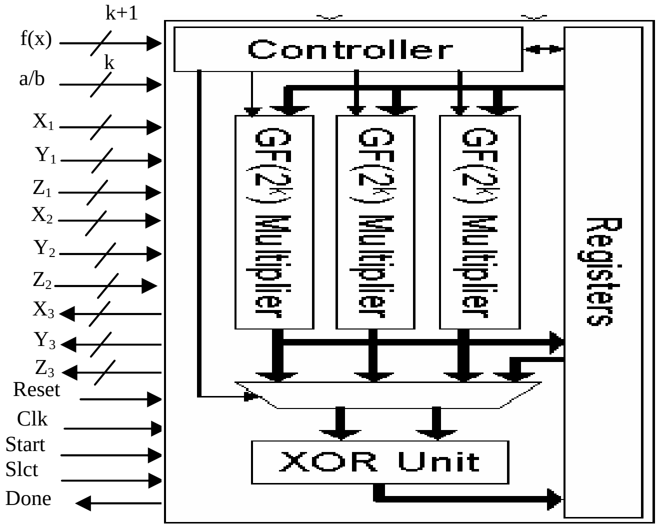


Figure 7: general elliptic curve point hardware outline

4.2 The ECC Coprocessor

Algorithm 3 is the main procedure to be implemented in hardware for designing the ECC coprocessor. This hardware requires nine *k*-bit registers and one *k+1* bit register for storing the modulus f(x). It contains, other than the registers, three multiplexors, a counter, a state-machine, and the general point operation hardware. The computations are performed in the general point operation hardware detailed in Figure 7. All components are controlled by the main state machine. This state machine resets its states at the beginning. It loads all the input data values, then, the state machine checks if the most significant bit of n (n_{k-1}) has the value one it loads P into register Q. If not, register Q keeps its original values of zeros. Then, the elliptic curve point operations begin. It starts with iteration number 1 and proceed until iteration k-1. Iteration k-1 is the last one to process, where the result should be ready after it.

4.3 Area Flexibility

The ECC coprocessor is designed to deal with numbers that are in the order of k-bits. However, The size of the hardware can be designed depending on the area available. In other words, the ECC coprocessor is built of two types of components, fixed size modules and flexible size ones. All modules are fixed size ones except the multiplier, which is completely flexible to the area available. Depending on the area available the number of modulo multiplication reduction modules are chosen. This flexible size multiplier is described in depth before.

5 Conclusion

We modified a *GF(p)* multiplication algorithm to make suitable for *GF(2^k)*. Then, both multiplication algorithms where modeled as hardware designs to compare them thoroughly. The modified *GF(2^k)* algorithm showed fixed fast speed and smaller area relative to the *GF(p)* multiplier. A further hardware improvement have been accomplished to the *GF(2^k)* multiplier to make it *40%* faster with an area cost of *5%*.

The *GF(2^k)* multiplier was used to build an ECC coprocessor. The point operation was processed in a projective coordinate manner to avoid the lengthy inversion computation. In fact, the inversion is needed only at the beginning and at the end, two times only, which made the assumption to compute it in software. This design is attractive because of its simplicity and suitability to be implemented in VLSI with today's technology.

6 Acknowledgements

Thanks to Professor *Mohammad K. Ibrahim* for his valuable comments and suggestions. Thanks also to KFUPM for all the support to conduct research.

References

[1] Miyaji A., "*Elliptic Curves over F_P Suitable for Cryptosystems*", Advances in cryptology-AUSCRUPT'92, Australia, December 1992.

[2] Stallings, W. "*Cryptography and Network Security: Principles and Practice*", Second Edition, Prentice Hall Inc., New Jersey, 1999.

[3] Chung, Sim, and Lee, "*Fast Implementation of Elliptic Curve Defined over GF(p^m) on CalmRISC with MAC2424 Coprocessor*", Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, Massachusetts, August 2000.

[4] Okada, Torii, Itoh, and Takenaka, "*Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2^m) on an FPGA*", Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, Massachusetts, August 2000.

[5] Crutchley, D. A., "*Cryptography And Elliptic Curves*", MS Thesis Supervised by Gareth Jones, Faculty of Math, University of Southampton, England, May 1999.

[6] Orlando, and Paar, "*A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)*", Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, Massachusetts, August 2000.

[7] Stinson, D. R., "*Cryptography: Theory and Practice*", CRC Press, Boca Raton, Florida, 1995.

[8] Paar, Fleischmann, and Soria-Rodriguez, "*Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents*", IEEE Transactions on Computers, Vol. 48, No. 10, October 1999.

[9] Blake, Seroussi, and Smart, "*Elliptic Curves in Cryptography*", Cambridge Univ. Press: NY, 1999.

[10] Hankerson, Hernandez, and Menezes, "*Software Implementation of Elliptic Curve Cryptography Over Binary Fields*", Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, Massachusetts, August 2000.

[11] Orton, Roy, Scott, Peppard, and Tavares. "VLSI implementation of public-key encryption algorithms", *Advances in Cryptology -- CRYPTO '86*, pages 277-301, 11-15 August 1986. Springer-Verlag, 1987.

[12] Scott, Norman R., "*Computer Number Systems and Arithmetic*", Prentice-Hall Inc., New Jersey, 1985.

[13] Tocci, and Widmer, "*Digital Systems: Principles and Applications*", Prentice-Hall Inc., New Jersey, 2001.

[14] Ercegovac, Lang, and Moreno, "*Introduction to Digital System*", John Wiley & Sons, Inc., New York, 1999.

[15] Gutub and Tenca, "Efficient Scalable VLSI Architecture for Montgomery Inversion in GF(p)", *Integration, the VLSI Journal*, Vol. 37, No. 2, 2004.

[16] Adnan Gutub, "Fast Elliptic Curve Cryptographic Processor Architecture Based On Three Parallel GF(2^k) Bit Level Pipelined Digit Serial Multipliers", *IEEE 10th International Conference on Electronics, Circuits and Systems (ICECS 2003)*, pages 72-75, University of Sharjah, United Arab Emirates, December 14-17, 2003.

[17] Adnan A. Gutub, "VLSI Core Architecture For GF(P) Elliptic Curve Crypto Processor", *IEEE 10th International Conference on Electronics, Circuits and Systems (ICECS 2003)*, pages 84-87, University of Sharjah, United Arab Emirates, December 14-17, 2003.

[18] Ernst, Klupsch, Hauck, and Huss,"Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems", *IEEE 12th Inter. Workshop on Rapid System Prototyping*, Monterey, CL, June 25-27, 2001.