CHAPTER 4

ECC OVER GF($2^n$) WITH DYNAMIC SCHEDULING

This chapter suggests another approach to compute the point multiplication of ECC over binary field GF($2^n$) for concurrent processing. It analyses the speed of crypto processor based on this approach.

4.1 INTRODUCTION

This chapter suggests a technique to implement the point multiplication of ECC over binary field (GF($2^n$)) for hardware applications. It is mainly going to use in wireless environments (Kazuo Sakiyama et al 2007). So the following details are needed for this computation as shown in Table 4.1. There are a lot of implementations available for this computation, but the most of them support only the linear computation model (Zhimin Chen et al 2011). There is no proper implementation for parallel computation.

Table 4.1 Different parameter values for ECC over GF($2^n$)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>ECC over GF($2^n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite Field</td>
<td>Based on the irreducible polynomial equation</td>
</tr>
<tr>
<td>Elliptic Curve</td>
<td>All type</td>
</tr>
<tr>
<td>Point Coordinate System</td>
<td>All type</td>
</tr>
<tr>
<td>Algorithms</td>
<td>All type</td>
</tr>
<tr>
<td>Protocol definition</td>
<td>Any type</td>
</tr>
<tr>
<td>Key Length</td>
<td>Any Size</td>
</tr>
<tr>
<td>Hardware/Software/both</td>
<td>Hardware</td>
</tr>
<tr>
<td>Way of Implementation</td>
<td>To support Parallel computation</td>
</tr>
<tr>
<td>Memory Space Usage</td>
<td>Optimum Level</td>
</tr>
<tr>
<td>Size of Code</td>
<td>Feasibility Level</td>
</tr>
</tbody>
</table>
Some of the different numbers within the prime ranges are unused for computation. So the ECC over GF(2^n) is defined with irreducible polynomial equation, because an irreducible polynomial equation generates all combinations of polynomial equations. An irreducible polynomial means that it cannot be divided by any of the polynomial equations. This type of irreducible polynomial equation is known as polynomial generator. So it is used to generate the finite field elements in the form of binary values.

Based on this, a simplified point multiplication with Tomasulo algorithm over binary field (2^n) is suggested for parallel computation. This approach is also used to schedule code for hardware execution known as dynamic scheduling, which avoids the dependent operations during the process. It will automatically reduce the number of clock pulses, latency time and repeated operations of the point computation (William N Chelton et al 2008). So the time complexity becomes better than other point multiplications, and also it saves power utilization in hardware (Jyu-Yhan Lai et al 2008).

For these reasons, this chapter is organized into seven sections. The section-4.2 describes the overview of ECC over GF(2^n) and the section-4.3 suggests an optimized technique for the point multiplication over binary field. Followed by, the section-4.4 demonstrates the result of proposed technique and subsequently these results are compared and analyzed with existing methodology in the section-4.5. Besides, the section-4.6 discusses some of the applications for this methodology. Finally, the section-4.7 concludes with its limitations and future enhancement.

4.2 OVERVIEW OF ECC OVER GF(2^n)

In this section, the ECC over GF(2^n) are described in detail with example. It is defined with the ElGamal concept, polynomial and binary field
(Kazuo Sakiyama et al 2007). The following subsections illustrate the concept of GF($2^n$), EC over GF($2^n$) and ECC over GF($2^n$) with examples.

### 4.2.1 Basics of Binary Field - GF($2^n$)

A polynomial equation is a combination of co-efficient and exponent values. Here, the exponents are used to define a set of elements. If the exponent value is available in the polynomial equation, then the binary value will be ‘1’. Similarly, if the exponent term is not in the polynomial equation, otherwise the binary value will become ‘0’ (Ray CC Cheung et al 2005).

A binary field is a finite field with the base value of two. It has two distinct digits: ‘0’ and ‘1’ used for defining the set elements based the irreducible and reducible polynomial equations. When a set has $2^n$ elements, the numbers of co-efficient are ‘0’ to $2^n-1$. So the value of ‘n’ denotes the size of any element in the set. Here, the maximum value of exponent is ‘n-1’ and the minimum value ‘0’.

It means that,

\[
f(x) = \sum \text{Co}_\text{eff}\exp x^\exp
\]

\[
\exp=\min
\]

\[
= \text{Co}_\text{eff}_\text{min} x^\text{min} + \text{Co}_\text{eff}_\text{min+1} x^{\text{min+1}} + \text{Co}_\text{eff}_\text{min+2} x^{\text{min+2}}
\]

\[
+ ... +\text{Co}_\text{eff}_\text{max-1} x^{\text{max-1}} + \text{Co}_\text{eff}_\text{max} x^\text{max}
\]

where the co-eff is a value of co-efficient, exp is a value of exponent, min=0 and max = $2^n$-1.
The operations over polynomial are defined with the coefficients and the exponent values of polynomials with modulo operation. In this case, the irreducible polynomial is a prime polynomial which cannot be factored by any polynomial. Here, the value of coefficients may be ‘0’ or ‘1’ and it is denoted by GF(2). If the polynomial is represented with the n-bit word length, it will be denoted by GF(2^n). A group of polynomials with the value of ‘n’ are the set of polynomials GF(2^n) (Hasan MA 1998).

The two operations of the polynomials are the polynomial addition and polynomial multiplication. The polynomial addition over GF(2^n) cannot be extended to the degree of (n-1), whereas, the polynomial multiplication over GF(2^n) can be extended (Harrison K et al 2002). When it is extended, it can be reduced through the modulus polynomial. The addition of GF(2) is shown as follows:

\[
\begin{align*}
0+0 &= 0 \mod 2 = 0 \\
0+1 &= 1 \mod 2 = 1 \\
1+0 &= 0 \mod 2 = 1 \\
1+1 &= 2 \mod 2 = 0
\end{align*}
\]

Here, the value ‘0’ is an additive identity of GF(2). Similarly the subtraction is performed as follows:

\[
\begin{align*}
0-0 &= 0 \mod 2 = 0 \\
0-1 &= -1 \mod 2 = 1 \\
1-0 &= 1 \mod 2 = 1 \\
1-1 &= 0 \mod 2 = 0
\end{align*}
\]
From these two operations, it is concluded that the addition and subtraction of GF(2) are the same. It is similar to the X-OR operation ($\oplus$) between the two values. Besides, the addition of GF($2^n$) is also performed based on the X-OR operation between two n-bit values (Chiou CW et al 2009).

The second operation of GF(2) is a polynomial multiplication and it is symbolically denoted by $\odot$. Here, each term of the first polynomial is multiplied with the entire second polynomial equation. Finally, it is summed up based on its corresponding exponents. Mostly, based on the second method, the algorithm is designed for point multiplication as follows:

1. If the most significant bit of previous result is ‘1’, it will shift a bit from right to the left. Then the X-OR operation is performed with modulo value. It means that the modulus operation is applied for reduction.

2. If the most significant bit of previous result is ‘0’, it will shift a bit from right to the left. But there is no X-OR operation. That is, the modulus operation is not applied for reduction.

An irreducible polynomial over GF($2^n$) generates the other elements in the set and it is denoted by $g$. It means that \(\{0, g^0, g^1, g^2, g^3, ..., g^N\}\), where $N=2^n - 2$. The power term denotes the repeated number of polynomial generator where $g$ is a polynomial generator (Meher PK 2009). Here, the identity of polynomial multiplication is ‘1’ and the inverse of polynomial multiplication is defined, based on modulo operation manipulation within $2^n-1$ (Akashi Satoh et al 2003). Besides, the polynomial division is defined by the product of the element and the inverse element of polynomials.
A binary field is a combination of an Abelian Group, Commutative Ring and Field rules based on modulo operation and they are defined as follows.

**Definition 1:** An Abelian Group \((G, \oplus)\) is a set of elements over operation \(\oplus\) with the following Group laws and Commutative law (Yongfei Han et al 1999).

1. \(a, b \in G\) then \(a \oplus b \in G\)
2. \(a, b, c \in G\) then \((a \oplus b) \oplus c = a \oplus (b \oplus c) \in G\)
3. \(a, -a \in G\) then \(a \oplus (-a) = (-a) \oplus a = 0 \in G\)
4. \(a, -a \in G\) then \(a \oplus 0 = 0 \oplus a = a \in G\)
5. \(a, b \in G\) then \(a \oplus b = b \oplus a \in G\) where \(a, b, c, 0, -a, -b, -c \in G\)

**Definition 2:** A Commutative Ring \((R, \oplus, \otimes)\) is an Abelian Group over two operations known as the polynomial addition and multiplication, and its rules are as follows (Rajendra Katti et al 2003):

1. \(a, b \in G\) then \(a \otimes b \in G\)
2. \(a, b, c \in G\) then \((a \otimes b) \otimes c = a \otimes (b \otimes c) \in G\)
3. \(a, b, c \in G\) then \((a \otimes b) \otimes c = (a \otimes c) \oplus (b \otimes c) \in G\)
4. \(a, b \in G\) then \(a \otimes b = b \otimes a \in G\)

**Definition 3:** A Field is defined by using a Commutative Ring under the following three rules.

1. \(a \in G, a \otimes 1 = 1 \otimes a = a \in G\)
2. \(a, b \in G, a \otimes b = 0 \in G\) then either \(a = 0\) or \(b = 0\)
3. \(a \otimes a^{-1} = a^{-1} \otimes a = 1\) and \(a, a^{-1} \in G\)
**Definition 4:** Finally, a Finite Field GF($2^n$) consists of $2^n$ elements with the polynomial addition and multiplication. They are determined based on reducible and irreducible polynomial equations. It means that GF ($2^n$) contains $2^n$ elements. They are irreducible polynomials which are identified from reducible polynomial.

**4.2.2 EC over GF($2^n$)**

EC over GF($2^n$) is an algebraic structure of Elliptic Curve equations with binary field. The following equation is considered as for EC over GF($2^n$) which is derived from Equation (1.1) (Darrel Hankerson et al 2000).

$$y^2 + xy = x^3 + ax^2 + b$$  \hspace{1cm} (4.1)

where $a_1=1$, $a_2=0$, $a_3=a$, $a_4=0$, $a_5=b$, $x$, $y$ are variables and $a$, $b$ are coefficients.

The set of $(x,y)$ values are substituted in Equation (4.1) to find out the points on $E_{2^n}(a,b)$. These points are manipulated based on the point addition and multiplication for ECC over GF($2^n$), and the following rules are used to define the point addition (William Stallings 2006).

Rule 1: $P_1=(x_1,y_1), P_2=(0,0)$ additive identity and $P_1+P_2$

$$P_1+P_2=(x_1,y_1)+(0,0) = P_1$$  \hspace{1cm} (4.2)

Rule 2: $P_1=(x_1,y_1)$ and $P_2=(x_1,x_1+y_1)$ (inverse of P)

$$P_1+P_2=(x_1,y_1)+(x_1,x_1+y_1)=O$$  \hspace{1cm} (4.3)

Rule 3: $P_1=(x_1,y_1), P_2=(x_1,y_1)$ and $P_1=P_2$

$$P_3=P_1+P_2=P_1+P_1=P_3$$  \hspace{1cm} (4.4)

where $(x_3 = \lambda^2 + a, y_3 = (x_1^2 + (\lambda + 1)x_3)$,

$$\lambda = x_1 + y_1/x_1.$$
It is also known as the point doubling.

Rule 4: $P_1=(x_1,y_1)$, $P_2=(x_2,y_2)$ and $P_1 \neq P_2$

$$P_3=P_1+P_2=P_3$$ (4.5)

where

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

$$\lambda = (y_2 + y_1)/(x_2 + x_1)$$

It is known as the point addition.

The number of elements is equal to the number of points in binary field $GF(2^n)$.

4.2.3 ECC over $GF(2^n)$

Three procedures of Elliptic Curve Cryptography are manipulated with the help of EC over $GF(2^n)$, which are defined as follows (Levent Ertaul et al 2007).

4.2.3.1 Key generation

The procedure of key generation for encryption or decryption is described as follows (William Stallings 2006):

**Common parameters**

1. Define $E_2^n(a,b)$ where $a$, $b$—elliptic curve parameters, $2^n$—prime number.
2. Find out $n \cdot G = 0$ where $G$ is point on EC whose order is within $2^n-1$.

**A pair of key generation on sender side**

1. Select a private key $n_A$ where $n_A < n$
2. Calculate a public key $P_A$ where $P_A = n_A \times G$

**A pair of key generation on receiver side**

1. Select a private key $n_B$ where $n_B < n$
2. Calculate a public key $P_B$ where $P_B = n_B \times G$

**Finding a secret key on sender side**

Calculate of a secret key on sender side $K = n_A \times P_B$

**Finding a secret key on receiver side**

Calculate of secret key on receiver side $K = n_B \times P_A$

**Verification and Validation of key pairs**

$$K = n_A \times P_B = n_A \times n_B \times G = n_B \times n_A \times G = n_B \times n_A \times G = n_B \times P_A = K$$

### 4.2.3.2 Encryption

The procedure of encryption explains the conversion of the plain text into the cipher text based on the sender’s private key and the receiver’s public key. These keys are used to compute the secret key for this operation.

Cipher Text = $(C_1, C_2)$ where $C_1 = kG$, and $C_2 =$ Plain Text $+ kP_B$

Here, the value of $k$ is a large random positive integer.
4.2.3.3 Decryption

The procedure of decryption describes the conversion of the cipher text into the plain text based on secret key. The secret key is generated based on the sender’s public key and a receiver’s private key.

\[
\text{Plain Text } = (C_2 - dC_1) = (C_2 + dC_1) \\
= (\text{Plain Text} + kP_B - n_B G) = (\text{Plain Text} + k(n_B \times G) - n_B (kG)) \\
= (\text{Plain Text} + k(n_B G) - n_B (kG)) = (\text{Plain Text} + \text{‘O’}) \\
= \text{Plain Text}
\]

The following example demonstrates the encryption, decryption and key generation of ECC over binary field (Forouzan BA 2008).

I. Key Generation

Common parameters

1. Define \( E_2^n(a,b) = E_2^3(g^3,1) \) where \( a=g^3, b=1 \) and \( p=2^3-1 \)
2. Assume \( G=(g^3,g^2) \)

A pair of key generation on sender side

1. private key \( n_A = g \pmod{2^3} = g(010) \)
2. public key \( P_A = g \pmod{2^3} = g^3 \pmod{2^3} = (g^5, g^4) \)

A pair of key generation on receiver side

1. private key \( n_B = g^2 \)
2. public key \( P_B = n_B \times G \pmod{2^3} = g^2 \times (g^3, g^2) \pmod{2^3} = (g^6, g) \)
II. Encryption

Assume the plain text P is equal to \((g^5,1)\) which is encoded from the value of symbol with \(k=g\).

\[
C_1 = kG = g(g^5, g^2) = (g^5, g^4)
\]

\[
C_2 = \text{Plain Text} + kP_B = g(g^6, g) + (g^5, 1) = (g^3, g^5)
\]

Therefore, the cipher text = \((C_1, C_2) = ((g^5, g^4), (g^3, g^5))\)

III. Decryption

The cipher text is decrypted as follows to produce the plain text \(P = (g^5, 1)\).

\[
\text{Plain text} = (C_2 - dC_1) \mod 2^3 - 1 = (g^3, g^5) + g(g^5, g^4)
\]

\[
= (g^3, g^5) + g^2((g^5, g^4) = (g^5, 1)
\]

ECC over \(GF(2^n)\) does not have the point subtraction, because the point addition and point subtraction are the same in binary field. It has only the point addition, point multiplication and point inversion. The point multiplication is a major part of ECC over \(GF(2^n)\). It is computed through the repeated number of the point additions and point doublings (Sining Liu et al 2007). The linear point multiplication is shown in Equation (4.6) and it is diagrammatically shown in Figure 4.1.
Figure 4.1  Linear point multiplication of kp, where the value of k denotes number of times and p is a point on ECC over GF($2^n$)

\[ kP = (1)_2P + (10)_2P + (11)_2P + (100)_2P + \ldots + (k-1)_2P + k_2P \]  

(4.6)

(k times of p)

There are a lot of implantations available for the point multiplications which are mostly implemented in the linear fashion. But they cannot be suitable for parallel computation, because they have a lot of dependencies which exist in its computations. The example for point multiplications are the linear point multiplication, double and add point multiplication, montgomery point multiplication and Jacobian point multiplication (Darrel Hanker Son et al 2000). The studies of these methodologies are already discussed in the chapter 2.

In the Jacobian point multiplication, the affine co-ordinate system is converted into the projective co-ordinate system over binary field for removing the inverse operations. Then, it is implemented with double and add or montgomery methodology (Siddika Berna Ors et al 2008).

4.2.4  Dynamic Code Scheduling

There are many types of dependences existing in the point scalar multiplication such as instruction dependencies, register value dependencies,
register name dependencies, predictor dependencies, loop carried dependencies and structural dependencies which affect the speed of computation (Kaihara ME 2005).

In this case, an instruction dependencies means that it depends upon another instruction during execution. The register name dependencies shares the same register for two more point computation. The predictor dependencies determine the way of testing the points based on Equations (4.2), (4.3), (4.4) and (4.5) to compute the point addition.

The analysis of loop-level parallelism checks the relationship between the recent computation and the previous / next computation. Finally, the structural dependency means that two or more instructions require the hardware for the same time. So the linear multiplication is not suitable for parallel computation. Hence, the following methodology is suggested for concurrent computation with the help of dynamic scheduling (Mark Allen Weiss 2007). This code scheduling is used to identify and avoid hazards and stalls based on the different dependencies.

4.2.5 Tomasulo Approach

The basic structure of Tomasulo algorithm is useful for avoiding hazards and stalls during the parallel processing. The point addition of Tomasulo approach performs the point addition and the point doubling separately through the different units.
Figure 4.2 Block diagram for performing point multiplication using Tomasulo with simplified double and add algorithms

At the time of computation, a set of registers are used for storing intermediate results (P, P₁, P₂, P₃ and SUM). Then, the common data bus updates the point registers in each time during iteration with its corresponding results. Here, the instructions are in a queue to issue sequentially for arithmetic units to perform point multiplication concurrently shown in Figure 4.2. The Tomasulo algorithm is also known as the operand forwarding technique for avoiding the different dependencies hazards and stalls (Hennessy JL et al 2007).

4.3 PROPOSED METHODOLOGY FOR ECC OVER GF(2ⁿ)

The proposed methodology of point multiplication for ECC over GF(2ⁿ) simplifies the double and add of point multiplication through the code scheduling for hardware. It is executed concurrently through the Tomasulo approach. The procedure of this code scheduling is explained briefly as follows.
Figure 4.3  Point multiplication of $kP$, where the value of ‘$k$’ is 7 and $P$ is a point in ECC over $GF(2^n)$

The value of ‘$k$’ in Equation (4.6) is used to create Huffman Tree for point multiplication with the help of divide and conquer methodology shown in Figure 4.3. The quotient and remainder values are obtained each time, when the value of ‘$k$’ is divided by 2. The points are substituted in Equation (4.4) or (4.5) to obtain the point multiplication. Finally these values are summed by through Equations (4.2), (4.3), (4.4) or (4.5) to compute the point multiplication of $kP$.

It means that the value of ‘$k$’ is used to form the binary trees for point doubling and skew trees for point addition with the help of divide and conquer in each computation. This divide and conquer strategy is meant to divide the point multiplication into two or more sub point multiplications. Finally, these are combined together as shown in Figure 4.4.
Figure 4.4 Dividing processes of k in the point multiplication of kP, where ‘k’ is the number of times and P is a point on ECC over GF(2^n)

Figure 4.5 Conquering processes of k in the point multiplication of kP, where k is number of times and P is a point on ECC over GF(2^n)
Each node in the binary tree has a point value (P) that can be computed through the point addition or point doubling. Finally, the summing of these trees gives the value of point multiplication shown in Figure 4.5.

For example, the value of ‘k’ is assumed as \((1111)_2\). When it is divided by 2 in each time, the quotient value becomes \((111)_2, (11)_2, (1)_2\) and the remainder values are 1, 1, 1. The binary tree of point doubling is computed by using a quotient value and a point on the node. It creates a complete binary tree as shown in Figure 4.6. These are used to perform point doubling based on Equation (4.4). Subsequently, the skew tree of point addition is also calculated by using a reminder value and a point on the node. It also creates an incomplete binary tree known as skew tree as shown in Figure 4.7. These are used to perform point addition based on Equation (4.5).

![Figure 4.6 Point doubling based on quotient value of k and a point on ECC over GF(2^n).](image1)

![Figure 4.7 Point addition based on the quotient value of k and a point on ECC over GF(2^n).](image2)
Finally, these two tree computations are summed up by using Equations (4.2), (4.3), (4.4) or (4.5) to compute point multiplication of kP. The procedures of these computations are as follows.

**Procedure for PointCompute()**

Input : scalar value k bits (k₀, k₁, … ,kₙ),
Point P : co-ordinates values in two registers
Output kP : co-ordinate values in two registers
Point P₁, P₂ : are pair registers, Pₐ is a parity bit - kᵢth bit

1. if k←1 then
   1.1 No computation.
   1.2 P ← P
2. P₁←(0,0)
3. P₂←(0,0)
4. if k>1 then
   4.1 Pₐ←right shift the value of K
   4.2 if(Pₐ=0 and k=1) and ((Pₐ=0 and k=0)  
      4.2.1 No operation and exit
   4.3 if(Pₐ≠0 and k=0) then
      4.3.1 P₁←call PointDoublingBinary(Point P)
      4.3.2 P←P₁
   4.4 if(Pₐ≠1 and k=1) then
      4.4.1 P₂←call PointDoublingSkew(Point P)
      4.4.2 P←P₂
5. \( k \leftarrow \text{left shift the value of } k \) and goto step 2

6. \( P \leftarrow P_1 + P_2 \)

**Procedure for PointDoublingBinary()**

Point \( P \) : co-ordinates values in two registers,

Quotient \( Q \) : in a register

1. Point Sum: \( \leftarrow (0,0) \)

2. \( \text{SUM} \leftarrow P + \text{SUM} \) based on Equations (4.2), (4.3), (4.4) or (4.5)

3. return SUM value

**Procedure for PointDoublingSkew(Point \( P \), Remainder \( R \))**

Point \( P \) : co-ordinates values in two registers

Quotient \( Q \) : in a register

1. Point Sum: \( \leftarrow (0,0) \)

2. \( \text{SUM} \leftarrow P + \text{SUM} \) based on Equations (4.2), (4.3), (4.4) or (4.5)

3. return SUM value

**4.4 RESULT**

The structural language ‘C’ is chosen for implementing the dynamic scheduling of ECC point multiplication. The point multiplication of code scheduling is simulated for measuring the number of clock cycles based on the following processor configuration.
Processor name: AMD Zacate E-350
Processor type: dual core with a die size of 75mm²
Processor speed: 1.60 GHz
Processor memory: 2 GB RAM
Word length of operating system: 64 bits

The Equation (4.1) is assumed to generate a set of points for experimentation. Besides, the equation $E_2^n(a,b)$ is assumed with:

$$E_2^4(g^4,g^2) \quad (4.7)$$

The point $(x=g^5,y=g^3)$ is taken from the set to compute point multiplication for both linear and the proposed point multiplications. It's parameter are listed in Table 4.2. The point multiplications are implemented through the bitwise operators, and the clock pulses are measured. These clock pulses are used to analyze the time complexity of the proposed point multiplications in three different cases with linear point multiplication. They are best, worst and average cases.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_2^n$</td>
<td>Input</td>
<td>$E_2^4$</td>
</tr>
<tr>
<td>A</td>
<td>Input</td>
<td>$g^4$</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>$g^2$</td>
</tr>
<tr>
<td>X</td>
<td>Input</td>
<td>$g^5$</td>
</tr>
<tr>
<td>Y</td>
<td>Input</td>
<td>$g^3$</td>
</tr>
<tr>
<td>P</td>
<td>Input</td>
<td>$(x,y)$</td>
</tr>
<tr>
<td>K</td>
<td>Input</td>
<td>Number of times(N)</td>
</tr>
<tr>
<td>$k_i$</td>
<td>Input</td>
<td>$0 &lt; N &lt; 2^i$</td>
</tr>
<tr>
<td>kP</td>
<td>Output</td>
<td>Point multiplication</td>
</tr>
<tr>
<td>kP execution time</td>
<td>Output</td>
<td>Number of Clock Pulses</td>
</tr>
</tbody>
</table>
Table 4.3 Comparison of Liner point multiplication and Huffman point multiplication of worst cases based on clock pulses

<table>
<thead>
<tr>
<th>i (1 to k)</th>
<th>$2^i$</th>
<th>Number of Clock pulses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Worst</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>11111</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>111111</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1111111</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>11111111</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>111111111</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>1111111111</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11111111111</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>111111111111</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>1111111111111</td>
<td>14</td>
</tr>
</tbody>
</table>

The best case time complexity is an optimal condition. When it considers the proposed point multiplication, there is no remainder for the value of ‘k’ in each time of iteration. It takes only $\log_2 N$ times to compute $kP$ based on the quotient value, where $k=N$ and $P$ is a point. There is no need to compute skew tree point multiplication, because its remainder value is zero in each case. But the linear point multiplication takes $2^n - 1$ times to compute $kP$. The comparisons between these two cases are shown in Table 4.3.
Table 4.4 Comparison of Liner point multiplication and Huffman point multiplication of best case based on clock pulses

<table>
<thead>
<tr>
<th>Compute kP</th>
<th>Number of Clock pulses</th>
</tr>
</thead>
<tbody>
<tr>
<td>i (1 to k)</td>
<td>2^i</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
</tr>
<tr>
<td>5</td>
<td>100000</td>
</tr>
<tr>
<td>6</td>
<td>1000000</td>
</tr>
<tr>
<td>7</td>
<td>10000000</td>
</tr>
<tr>
<td>8</td>
<td>100000000</td>
</tr>
<tr>
<td>9</td>
<td>1000000000</td>
</tr>
<tr>
<td>10</td>
<td>10000000000</td>
</tr>
<tr>
<td>11</td>
<td>100000000000</td>
</tr>
<tr>
<td>12</td>
<td>1000000000000</td>
</tr>
<tr>
<td>13</td>
<td>100000000000000</td>
</tr>
</tbody>
</table>

The second is a worst case which defines the way of behavior of algorithm in all possible conditions. In the proposed point multiplication, there is a remainder and quotient values for ‘k’ in each time. The value of ‘k’ should be 2^n-1 number. It takes log₂N times to compute point doubling based on the quotient value with log₂N times to compute the point addition.

Then, the time complexity of this case is measured as by O(log₂N) for point doubling and O(log₂N) for point addition. Hence, the time complexity of point multiplication is defined as O(log₂N+log₂N). In linear
point multiplication, it takes $2^n-1$ times of point additions to compute $kP$. So there is no change in linear time complexity. The comparisons between these two cases are shown in Table 4.4.

The third case is an average case in which the algorithm acts under the probability of execution. Here, there is a remainder for the value of ‘$k$’ in some cases and no remainder for some other cases. Therefore, the computation time is measured as $\log_2 N$ of point doubling and $\log_2 N$ times of point addition based on the probability. So it is denoted by $O(\log_2 N)+O(\text{Probability of } \log_2 N)$. All these time complexities are redefined by $\log_2 N+1$, $2\log_2 N+1$ and $\log_2 N+\text{Prob}\{\log_2 N\}+1$, where the value ‘1’ denotes the final computation of $kP$.

4.5 PERFORMANCE ANALYSES

In this case, the computational complexity of point multiplication is analyzed through the clock pulses with the different bar graphs. Here the x axis denotes the number of clock pulses and the y axis denotes the value of $k$ in point multiplication. The following graph shows that the numbers of clock pulses needed for linear point multiplication based on the different value of ‘$k$’ as shown in Figure 4.8.
Figure 4.8 Numbers of clock pulses needed to compute $kP$ by using linear scalar multiplication, where $P$ is a point on ECC over $2^n$ and $k$ is the number of times in polynomial generator.

Figure 4.9 Numbers of clock pulses needed to compute $kP$ by using Huffman computation for best and worst cases, where $P$ is a point on ECC over $2^n$ and $k$ is the number of times in polynomial generator.
In the proposed methodology, the point multiplication is analyzed in two different cases. One is the best case time complexity and another is the worst case time complexity as shown in Figure 4.9. The blue lines in the graph show the best case time complexity and the green lines shows the worst case time complexity for the various value of ‘k’ in the point multiplication.

The first case is known as best case. It is identified by the way of behaving in an optimal condition. Then, it is compared with the linear point multiplication. It is shown in the graph as Figure 4.10.

![Comparison between the linear scalar and the best case of Huffman computation, where P is a point on ECC over $2^n$ and k is the number of times in polynomial generator](image)

The second case is known as worst case. It is identified by the way of behaving in all cases. Then it is compared with the linear point multiplication. It is shown in the graph as Figure 4.11.
Figure 4.11  Comparison between the linear scalar and the worst case of Huffman computation, where P is a point on ECC over $2^n$ and $k$ is the number of times in polynomial generator

Figure 4.12  Comparison between the linear scalar and the worst case of Huffman computation, where P is a point on ECC over $2^n$ and $k$ is the number of times in polynomial generator
The third case is known as average case. The algorithm is analyzed based on the probability of execution. There is a remainder for k values in some time and there is no remainder for some other time in iteration. So the value of ‘k’ is $2^N$ or $2^N-1$. Again, it is compared with linear multiplication. The final graph in Figure 4.12 shows the conclusion of the linear and proposed point multiplications based on the different value of ‘k’.

When it is considering the number of clock cycles from Tables 4.3 and 4.4, the proposed methodology is better than linear point multiplication shown in Table 4.5.

<table>
<thead>
<tr>
<th>The value of ‘k’</th>
<th>Linear Point Multiplication (clock cycles)</th>
<th>Proposed point Multiplication (clock cycles)</th>
<th>The percentage of improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 (best)</td>
<td>6</td>
<td>3</td>
<td>50 %</td>
</tr>
<tr>
<td>15 (worst)</td>
<td>6</td>
<td>5</td>
<td>16.67%</td>
</tr>
<tr>
<td>256 (best)</td>
<td>91</td>
<td>6</td>
<td>93.41%</td>
</tr>
<tr>
<td>255 (worst)</td>
<td>91</td>
<td>10</td>
<td>88.89 %</td>
</tr>
<tr>
<td>4095 (best)</td>
<td>1503</td>
<td>9</td>
<td>99.4 %</td>
</tr>
<tr>
<td>4096 (worst)</td>
<td>1503</td>
<td>13</td>
<td>99.14 %</td>
</tr>
</tbody>
</table>

The same improvement is achieved for Double and Add, Montgomery and Joabosin point multiplications. But these methodologies have the different amount of dependencies, which affect parallel processing. The proposed technique is also used for avoiding the different dependencies through the code scheduling. The Table 4.6 shows the comparison among
different point multiplication with proposed point multiplication. It will also avoids the more utility of hardware.

Table 4.6 Comparison of different point multiplication with the proposed computation

<table>
<thead>
<tr>
<th>Methodologies</th>
<th>Linear</th>
<th>Montgomery</th>
<th>Jacobian</th>
<th>Tree Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary length of k value in kP denoted by n</td>
<td>Best Case</td>
<td>2×n times of Point addition</td>
<td>n times of point doubling</td>
<td>n times of point computation without inversion operation</td>
</tr>
<tr>
<td></td>
<td>worst case</td>
<td>n times of point doubling + n times of point addition</td>
<td>n times of point computation without inversion operation</td>
<td>n times of point computation</td>
</tr>
<tr>
<td>Number of Dependences</td>
<td>operand</td>
<td>More</td>
<td>More</td>
<td>more</td>
</tr>
<tr>
<td></td>
<td>predictor</td>
<td>More</td>
<td>More</td>
<td>less</td>
</tr>
<tr>
<td></td>
<td>loop carried</td>
<td>More</td>
<td>More</td>
<td>less</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>Less</td>
<td>Less</td>
<td>more</td>
</tr>
<tr>
<td>Occurrence of Hazards and Stalls</td>
<td>RAW</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>WAR</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>WAW</td>
<td>No</td>
<td>No</td>
<td>YES</td>
</tr>
<tr>
<td>Parallel Processing</td>
<td>not possible</td>
<td>not possible</td>
<td>not possible</td>
<td>possible</td>
</tr>
</tbody>
</table>

The operand dependencies in the Double and Add, Montgomery or Jacobian cases are similar to data dependencies, and also the predictor dependencies are similar to control dependencies based on hardware decision. There is no change in the hardware part of loop carried dependencies. The
register dependencies are identified through the point addition and point doubling which maintains the results in the same register. It is a same number of register dependencies for all point multiplications except the proposed point multiplication. It includes an additional register to maintain intermediate results.

For example, the following samples are taken from Tables 4.4 and 4.5 to compare the proposed methodology of worst case with Double and Add, Montgomery and Jacobian methodologies shown in Table 4.7.

**Table 4.7 Comparison among the different point multiplication with the proposed tree computation**

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Name of Dependency</th>
<th>Double and Add</th>
<th>Montgomery</th>
<th>Jacobian</th>
<th>Proposed Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 8, 12</td>
<td>operand</td>
<td>7, 15, 23</td>
<td>7, 15, 23</td>
<td>7, 15, 23</td>
<td>4, 8, 12</td>
</tr>
<tr>
<td></td>
<td>predictor</td>
<td>4, 8, 12</td>
<td>8, 16, 24</td>
<td>(4, 8, 12) or (8, 16, 24)</td>
<td>3, 7, 11</td>
</tr>
<tr>
<td></td>
<td>loop carried</td>
<td>3, 7, 11</td>
<td>3, 7, 11</td>
<td>3, 7, 11</td>
<td>2, 6, 10</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

From Table 4.8, when the proposed point multiplication is compared to other point multiplication, the following percentages of dependencies are reduced as shown in Figure 4.13.
1. 48% to 43% amount of operand dependencies are reduced
2. 25% to 8% amount of predictor dependencies are reduced
3. 33% to 9% amount of loop carried dependencies are reduced

![Pie chart showing dependencies reduction](image)

**Figure 4.13** The number of dependencies are reduced in this proposed point multiplication

### 4.6 DISCUSSION AND APPLICATION

The crypto processor is designed, based on the different parameters such as processor speed, size of processor, a level of secrecy (Top Secret, Secret, Confidential and Commercial) and architecture design issues for solving computational problems (Bartolini S et al 2008). A computational problem is a mathematical object representing the mathematical logics. The mathematical logic is a part of the foundations for computer science to develop an algorithm for problems (Ricardo Dahab et al 2006).

An algorithm is analyzed based on the different versions of a computational complexity. This complexity helps to estimate the resources needed for any task, which solves a computational problem theoretically (Dimitrov VS et al 2008). The main resources of these algorithms are measured as a function, which is going to relate the space complexity with the
time complexity. In theoretical analysis of algorithms, these complexities are known as the asymptotic complexities such as the big, omega and theta notations. Normally, the speed of processor is more important than the size of the processor (Sakiyama K et al 2007). So a time-constructible function is a function ‘f’ with the property f(n), and it can be constructed from ‘n’ steps for processor and time hierarchy theorems, which are important for time-bounded computation.

In addition to, the descriptive complexity is used to characterize the complexity classes, based on a computational complexity and finite model. So the arithmetic circuit of processor is designed to take inputs either variables or numbers from the already computed values. Then, they are allowed to perform either add or multiply operations. These intermediate results are a set of decision problems, and the problems may be decidable through the period of poly logarithmic time for parallel processing. So the suitable algorithm is selected for designing crypto processor to provide top security with rapid speed. It also utilizes hardware in limited level (Shoufan A et al 2010).

Besides, the crypto processor is designed based on the space, time and circuit complexities, strength of security and lifetime. In these, the lifetime is an important parameter which decides the performance of processor (Jarvihen K et al 2008). Amdahl's Law is also stated, that the lifetime of the processor mainly depends upon the performance of the processor, which is briefly discussed in appendixes 1 and 2. So, it is necessary to design a rapid speed of crypto processor which utilizes the hardware units in minimum level. This chapter suggests a methodology for ECC over \(2^n\) to provide a greater level security with minimum amount of usage.
4.7 SUMMARY

ECC over binary field are more secure to carry out cryptographic operations in the hardware such as a crypto processor or crypto chip. The Tomasulo algorithm with this approach is a hardware approach to track instruction dependences, and it allows execution as soon as data available on the common bus. This algorithm also avoids hazards and stalls, based on the dynamic scheduling. So, the possibilities of overlapping operations are created for parallel computation. In future, the proposed point multiplication of ECC over GF(2^n) is combined with the Tomasulo architecture for various applications to save the power. It automatically avoids the more usage of the hardware. So the life time of hardware is increased.