A high performance pseudo-multi-core elliptic curve cryptographic processor over GF(2^{163})

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon

By

Yu Zhang

Permission to Use

Abstract

Elliptic curve cryptosystem is one type of public-key system, and it can guarantee the same security level with Rivest, Shamir and Adleman (RSA) with a smaller key size. Therefore, the key of elliptic curve cryptography (ECC) can be more compact, and it brings many advantages such as circuit area, memory requirement, power consumption, performance and bandwidth. However, compared to private-key system, like Advanced Encryption Standard (AES), ECC is still much more complicated and computationally intensive. In some real applications, people usually combine private-key system with public-key system to achieve high performance. The ultimate goal of this research is to architect a high performance ECC processor for high performance applications such as network server and cellular sites.

In this thesis, a high performance processor for ECC over Galois field (GF)(2^{163}) by using polynomial presentation is proposed for high-performance applications. It has three finite field (FF) reduced instruction set computer (RISC) cores and a main controller to achieve instruction-level parallelism (ILP) with pipeline so that the largely parallelized algorithm for elliptic curve point multiplication (PM) can be well suited on this platform. Instructions for combined FF operation are proposed to decrease clock cycles in the instruction set. The interconnection among three FF cores and the main controller is obtained by analyzing the data dependency in the parallelized algorithm. Five-stage pipeline is employed in this architecture. Finally, the μ-code executed on these three FF cores is manually optimized to save clock cycles. The proposed design can reach 185 MHz with $20,807$ slices when implemented on Xilinx XC4VLX80 FPGA device and 263 MHz with 217,904 gates when synthesized with TSMC $.18\mu m$ CMOS technology. The implementation of the proposed architecture can complete one ECC PM in 1428 cycles, and is 1.3 times faster than the current fastest implementation over $GF(2^{163})$ reported in literature while consumes only 14.6% less area on the same FPGA device.

Acknowledgements

I would like to take this opportunity to express my sincere appreciation to my supervisor, Dr. Seok-Bum Ko. Without his guidance, advice and support throughout my research, this work could not be have been realized. Besides, he helped me a lot when I was hunting for jobs during the time of graduation, and gave me lots of valuable advices.

I would like to thank my supervisor Dr. Li Chen for his advice for my life, constant encouragement and support on my research and graduate studies.

I would also like to thank my lab member Dongdong Chen. Without the past two years discussion with him on research and algorithms, I could not have intensive understanding in research.

I would also like to thank my other friends in VLSI lab. Working with them is an invaluable and wonderful experience in my life.

I would like to thank my family for their understanding, constant support and encouragement throughout my study and life.

This is the dedication to my grandma

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application-specific integrated circuit
CMOS	Complementary Metal Oxide Semiconductor
DES	Data Encryption Standard
DL	Data Loading
EC	Elliptic Curve
ECADD	Elliptic Curve Point Addition
ECC	Elliptic Curve Cryptography
ECDBL	Elliptic Curve Point Doubling
ECP	Elliptic Curve Processor
EX	Instruction Executing
FF	Finite Field
FFA	Finite Field Addition
FFI	Finite Field Inversion
FFM	Finite Field Multiplication
FPGA	Field Programmable Gate Array
FFS	Finite field square
ILP	Instruction-level Parallelism
ID	Instruction Decoding
IF	Instruction Fetching
GNB	Gaussian Normal Basis
GF	Galois Field
LSB	Least Significant Bit
MQV	Menezes-Qu-Vanstone
NAF	Non-adjacent Form
NIST	National Institute of Standards and Technology
PB	Polynomial Basis
PM	Point Multiplication
RISC	Reduced Instruction Set Computer
RSA	Rivest, Shamir and Adleman
WB	Writing Back

Chapter 1

Introduction

1.1 Cryptography

Cryptography plays a very important role in modern communications as it can ensure the safety of the confidential data in the communication. Cryptography is composed with encryption and decryption. Encryption converts plain text (ordinary information) to cypher text (disordered information) by using a key, and decryption reverses the process. There are two types of cryptography, symmetric cryptography and asymmetric cryptography. In symmetric cryptography (also called as Private-Key cryptography) like AES [1], the sender and the receiver involved in the communication share a same key, so they have to negotiate the same key before conducting the communication. In asymmetric cryptography (also called as Public-key cryptography) such as ECC and RSA, different keys are used in the encryption and decryption, and both sides in the communication don't need to share their keys each other.

RSA, which stands for Rivest, Shamir and Adleman who first publicly described it in 1977 [2], is so far the most widely used public-key cryptography. It is based on the fact that it is much easier to create extreme large prime numbers while it is not practical in terms of time and money to factor the product of two primes of

Time to break in MIPS years	RSA/DSA key size	ECC key size	RSA/ECC key size ratio
10^{4}	512	106	5 : 1
10^{8}	768	132	6 : 1
10^{11}	1,024	160	7 : 1
10^{20}	2,048	210	10 : 1
10^{78}	21,000	600	35 : 1

Figure 1.1: Key size on equivalent strength between RSA and ECC [3]

this size. ECC was independently proposed by Miller [4] and Koblitz [5] in 1986 and 1987 respectively. ECC is based on the hardness of the elliptic curve discrete logarithm problem. ECC can guarantee the same security level with RSA with a smaller key size as shown in Fig. 1.1. Therefore, the key of ECC can be more compact, and it brings many advantages such as circuit area, memory requirement, power consumption, performance and bandwidth. ECC has also been included in IEEE 1363 [7] and NIST [6]. Consequently, ECC is said to be the next-generation cryptography, and a vast research has been done on its efficient implementation in software and hardware.

1.2 Previous Work

Compared to Private-key cryptography, ECC is computationally intensive, which is mainly caused by the computation of PM (involves arithmetic in FF of large order). The computation of PM is normally composed of point addition (ECADD) and doubling (ECDBL) operations, and these operations in turn rely on finite field (FF)

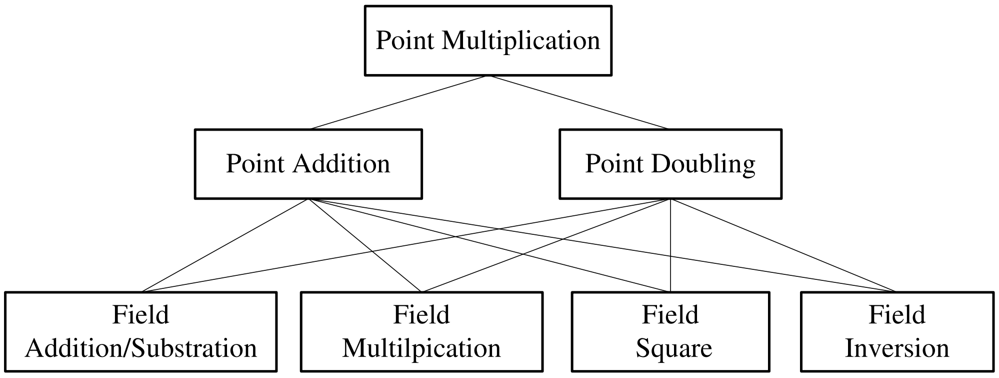

Figure 1.2: Conventional hierarchy for ECC operation

operations. The conventional operation hierarchy is shown in Fig.1.2. Normally, the complexity of these FF operations is FF inversion (FFI), FF multiplication (FFM), FF square (FFS), FF addition (FFA) in order from the most to the least. Thus many ECC arithmetic algorithms try to use different projective coordinates to lower FFI operations, such as Lopez-Dahab algorithm [14]. Also, there are many finite field arithmetic algorithms to implement FFI [17] [19], and it usually relies on FFM, FFS and FFA.

As we can see in Fig.1.2, the lowest level of PM operation is the FF operations, and there are different algorithms for them in either hardware or software implementations. Elliptic curve arithmetic means the algorithm to calculate PM by using ECADD and ECDBL. Therefore, before implementing the PM, several choices have to be made, and they include the selection of underlying finite field, field representation, elliptic curve, algorithms for finite field arithmetic, and algorithm for elliptic curve arithmetic. Different systems have different selections, which are determined by the system requirements (gate count, power consumption) and resources (availability of microprocessor, performance of microprocessor, ROM size and RAM size). These selections are tightly related to the implementation approaches, in turn, the implementation approaches (hardware, software, or hardware/software co-design) will rely on these selections, and it is very hard to make the "best" choice on these selection.

Normally, ECC in prime field is implemented in software due to that the operations in prime field are very similar with that in real number system except an extra modular operation needed in prime field. Algorithms for software implementation can be found in [8]. Hardware/software co-design implementation is an alternative approach to implement ECC when the hardware resource is tight [26]. In some high performance oriented applications like network servers and cellular sites, software ECC implementation will definitely cause a bottleneck of the entire system when the number of the services increases in a second. Therefore, the hardware PM core implementation on FPGA or ASIC is a solution for these applications. Many papers [10] [11] [12] [14] [15] [16] focus on the hardware implementation of PM on GF(2^{163}) defined in NIST curve over binary fields, and intensively compare their performance with others. In [10], the data dependency of Lopez-Dahab algorithm was analyzed in detail, and finally, a single FF multiplier in the elliptic curve processor (ECP) was employed to run with no rest, and other FF operations were finished in parallelism with the FF multiplier. As [10] did not introduce Lopez-Dahab algorithm with the largest parallelism, in [11], three 55-bit word level Gaussian normal basis (GNB) FF multipliers were employed to parallelize Lopez-Dahab algorithm to the largest extent. Besides, by using the GNB finite field representation, the operation of A^{2^s} in Itoh-Tsujii's FF inversion [17] can be simply accomplished by s-bit cyclic shift. The whole system is composed of two FF arithmetic atomic blocks, namely, point doubling&addition unit and coordinate conversion unit. Also, ECC hardware implementation on Koblitz curves can be found in [13], and they belong to a special class of binary curves, and the PM can be computed very efficiently. However, they are often vulnerable to side channel attacks [24].

1.3 Motivation

Efficient ECC hardware implementation depends on all the factors mentioned above. However, the above works only consider either elliptic curve arithmetic [14] or algorithms for finite field arithmetic [11], which is usually not the best case for hardware implementation. This work focuses on hardware implementation of PM on FPGA and ASIC on a NSIT proposed random curve over GF(2^{163}).

In ECC hardware implementations, there is usually no dedicated hardware for FFI, and it is implemented by a large number of other FF operations. FFM usually is the second time-consuming FF operation after FFI, and there are large numbers of FFMs involved in a PM operation. However, high speed FF multiplier by FF arithmetic algorithms doesn't necessarily mean high performance of the entire ECC system, which is determined by the following relationship, where the delay of the critical path is the period of the clock.

$$\textit{System performance} = \textit{Total clock cycles} \times \textit{Delay of the critical path}. \tag{1.1}$$

For example, if the computation of FFM only consumes one cycle, some other simple FF operations that also consume one cycle, such as FFA, will be as expensive as FFM. As a result, the total clock cycle reduced, the system performance may remain low because of the long system critical path in the system architecture caused by the FF multiplier. On the other hand, if the FF multiplier has a short critical path, and the FFM consumes several cycles, the system performance may still be low because of the large amount of clock cycles in total.

In order to improve the system performance, ECC arithmetic algorithm level, FF arithmetic level and system hardware architecture level should be considered simultaneously, and it requires to make a balance between total clock cycles and critical path, which usually refers to adding pipeline and increasing parallelism.

In [11], the critical path of the system is not analyzed in detail. The GNB based FF multiplier has a relative longer critical path than the polynomial based counterpart. Therefore, the performance may still be improved by analyzing the system critical path when using polynomial presentation. In order to increase the system performance, this thesis will consider the following three aspects simultaneously:

1. **ECC arithmetic algorithm level**: largely parallelize the Lopez-Dahab algorithm.

2. **FF arithmetic level**: make the FF multipliers run as often as possible, and other operations performed in parallel with FF multiplier.

3. **System hardware architecture level**: try to combine simple operations, and make the system critical path lie in the polynomial based FF multiplier.

1.4 Contribution

The main contributions of this thesis include:

1. Proposed a customized instruction set for parallel version of Lopez-Dahab algorithm;
2. Architected a three-FF-cores based architecture with five-stage pipeline, and analyzed the critical path in detail;
3. μ-code on three FF cores is given based on this proposed architecture;
4. Both FPGA and ASIC implementation results are provided. In FPGA implementation, this work is 1.3 times faster than the current fastest implementation over $GF(2^{163})$ reported in literature while consumes only 85.4% of their area on the same FPGA device.

1.5 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 gives the background knowledge in cryptography, and presents the ECC algorithms used in this work; Chapter 3 describes the architecture of the proposed ECC processor, and the critical path of system is analyzed in detail; Chapter 4 shows the implementation results on FPGA and ASIC of this work, and the comparison with some latest works as well. Some analysis is given on this result, and finally the conclusion and future work will be given.

Chapter 2
Elliptic Curve Cryptography

ECC system has different protocols, which contains ECC Diffie-Hellman key exchange protocol, Elliptic Curve Digital Signature Algorithm, and elliptic curve Menezes-Qu-Vanstone (MQV) [19] [21]. As this thesis is not mainly concerned with protocols, and it will only briefly introduce the ECC Diffie-Hellman key exchange protocol. Elliptic curve arithmetic algorithms are the main concern in this chapter. We will present how we propose an algorithm based on previous elliptic curve arithmetic algorithms in the literature [10] [11] [14] [18].

2.1 ECC Diffie-Hellman key exchange protocol

The ECC Diffie-Hellman key exchange protocol [19] [21] relies on the fact that the scalar of point P on elliptic curve (kP) is relatively easier while retrieving k knowing kP and P is a discrete logarithm problem. The mechanism of ECC Diffie-Hellman key exchange is shown in Fig. 2.1, where Alice and Bob are two persons who are involved in the communication, and Eva is a cracker. The key exchange mechanism procedure is described as follows:

1. Alice generates a random private key integer k_a , computes a public key $P_a =$

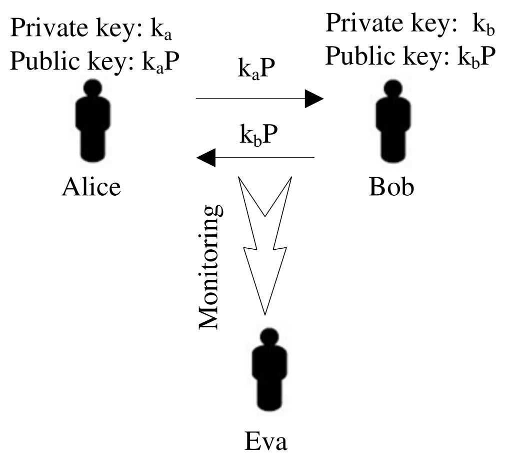

Figure 2.1: An example of ECC key exchange

k_aP, and sends P_a to Bob

2. Bob generates a random private key integer k_b , computes a public key $P_b = k_bP$, and sends P_b to Alice

3. Alice computes $k_aP_b = k_ak_bP$

4. Bob computes $k_bP_a = k_bk_aP$

5. Finally, Alice and Bob arrive at the same key k_ak_bP

In this communication, the information exposed to public (Eva) are only P_a, P_b, and P, and it is a discrete logarithm problem [21] to retrieve k_a and k_b.

2.2 Elliptic Curve Geometry

First, we will present the basic mathematics deduction of elliptic curve in real number system, describe how the ECADD and ECDBL are defined, and how the cryptography is defined on elliptic curve as well.

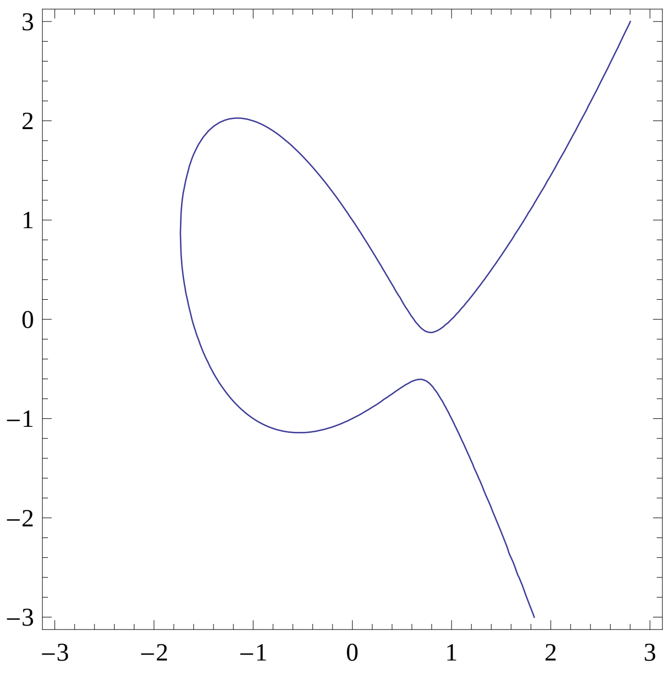

Figure 2.2: An example of an elliptic curve in real numbers system

NIST recommends the elliptic curve $\mathcal{E}$ in Eq. 2.1 , where a, b, x and y are all finite field numbers [6] [19] [20] . The prime field and binary finite field are two finite fields used in ECC.

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

In real number system, this type of equation stands for an elliptic curve in geometry as the example shown in Fig. 2.2, where the equation is

$$y^2 + xy = x^3 - 2x^2 + 1 \tag{2.2}$$

In the following, we will analyze the elliptic curve geometry in real number system.

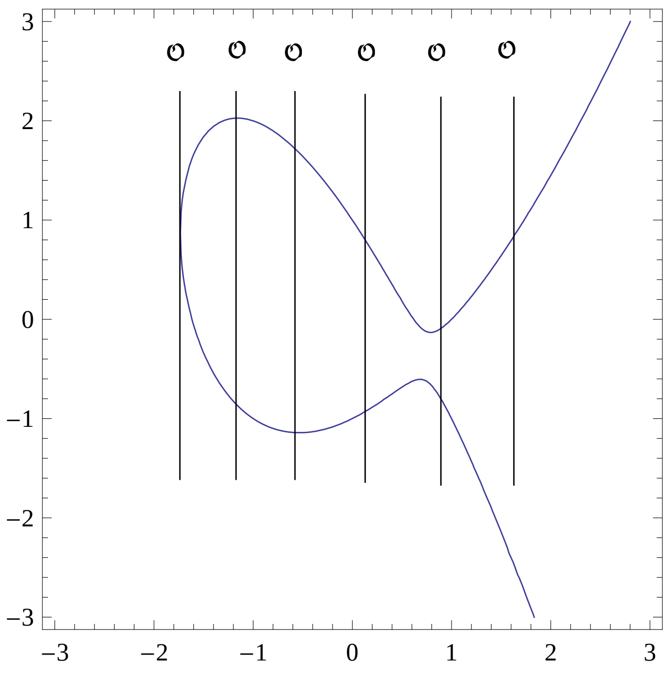

Figure 2.3: Special point on elliptic curve

2.2.1 A line through Two Distinct Points On Elliptic Curve

Given two distinct points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the elliptic curve $\mathcal{E}$, and $\mathcal{L}$ is the line through them, we can obtain the third intersection point $T = (T_x, T_y)$ on the $\mathcal{E}$. As the slope of $\mathcal{L}$ may not exist if x_1 equals to x_2 as shown in Fig. 2.3, the third intersection does not physically exist on the $\mathcal{E}$. Therefore, a special point $\mathcal{O}$ is defined as the third intersection point for this situation, and it is located at infinity as shown in Fig. 2.3.

If $x_1 \neq x_2$ as shown in Fig. 2.4, we can get the slope of line $\mathcal{L}$,

$$\lambda_1 = \frac{y_2 - y_1}{x_2 - x_1}, \tag{2.3}$$

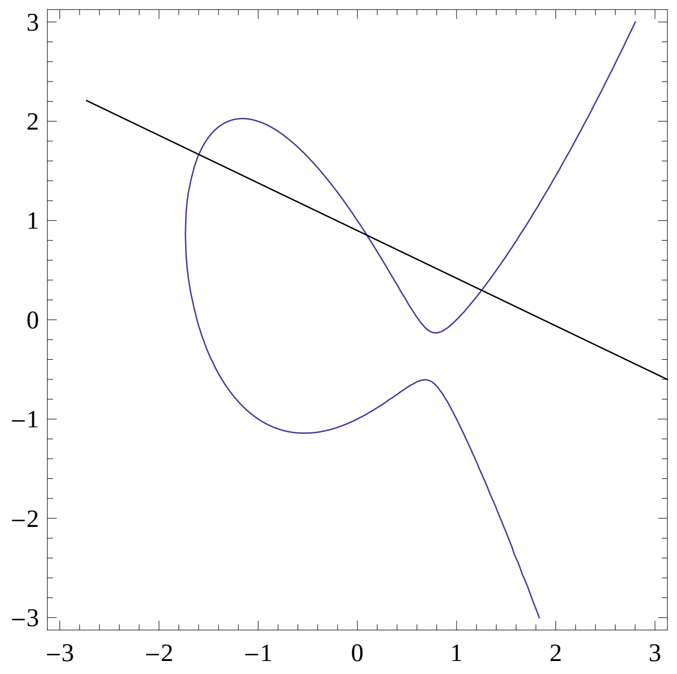

Figure 2.4: Intersection points on Elliptic Curve

then, the line $\mathcal{L}$ is

$$y = \lambda_1(x - x_1) + y_1. \tag{2.4}$$

By substituting Eq. 2.4 into Eq. 2.1,

$$(\lambda_1(x - x_1) + y_1)^2 + x(\lambda_1(x - x_1) + y_1) = x^3 + ax^2 + b. \tag{2.5}$$

At the first glance, we need to solve a quadratic and cubic equation. However, knowing x_1 and x_2, we can compare the parameters of the following equation with Eq. 2.5,

$$(x - x_1)(x - x_2)(x - T_x) = 0, \tag{2.6}$$

and after the normalization, we have,

$$x^3 - (x_1 + x_2 + T_x)x^2 + (T_x x_1 + T_x x_2 + x_1 x_2)x - x_1 x_2 T_x = 0. \tag{2.7}$$

Then, we can compare the parameter of x^2 between Eq. 2.7 and Eq. 2.5,

$$a - \lambda_1 - {\lambda_1}^2 = -(x_1 + x_2 + T_x) \tag{2.8}$$

$$T_x = {\lambda_1}^2 + \lambda_1 - (x_2 + x_1) - a. \tag{2.9}$$

Finally, from Eq. 2.9 and Eq. 2.4,

$$T_y = \lambda_1(T_x - x_1) + y_1. \tag{2.10}$$

2.2.2 A Tangent Line of Elliptic Curve

If P_1 and P_2 are the same, and the line $\mathcal{L}$ is a tangent line through it as shown in Fig. 2.5, the third intersection point T can be obtained similarly. It can also be the infinite point $\mathcal{O}$ as shown in the Fig. 2.3 if the slope of the tangent line $\mathcal{L}$ does not exist. If the slope λ of $\mathcal{L}$ exists, it is the derivative of $\mathcal{E}$ in P_1,

$$\begin{aligned} det(y^2 + xy) &= det(x^3 + ax^2 + b) \\ 2yy' + y + xy' &= 3x^2 + 2ax \\ y' &= \frac{3x^2 + 2ax - y}{2y + x} \end{aligned} \tag{2.11}$$

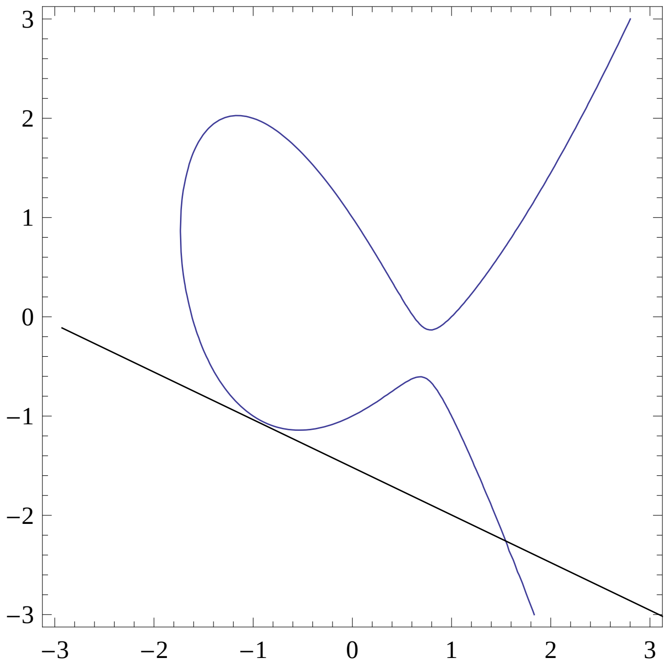

Figure 2.5: Tangent line on Elliptic Curve

Then,

$$\lambda_2 = \frac{3x_1^2 + 2ax_1 - y_1}{2y_1 + x_1} \tag{2.12}$$

and the line $\mathcal{L}$ is

$$y = \lambda_2(x - x_1) + y_1. \tag{2.13}$$

By substituting Eq. 2.13 into Eq. 2.1, we have

$$(\lambda_2(x - x_1) + y_1)^2 + x(\lambda_2(x - x_1) + y_1) = x^3 + ax^2 + b. \tag{2.14}$$

As we have two same points, then

$$(x - x_1)(x - x_1)(x - T_x) = 0, \tag{2.15}$$

and after the normalization, it is

$$x^3 - (2x_1 + T_x)x^2 + (2T_x x_1 + x_1^2)x - x_1^2 T_x = 0 \tag{2.16}$$

By comparing the parameters between Eq. 2.16 and Eq. 2.14, we can get,

$$T_x = {\lambda_2}^2 + \lambda_2 - 2x_1 - a. \tag{2.17}$$

$$T_y = \lambda_2(T_x - x_1) + y_1. \tag{2.18}$$

2.2.3 Point Addition on Elliptic Curve

In the above sections, we have described some mathematic background of elliptic curve in geometry in real number system, now we are going to know how the PM is defined on the elliptic curve in binary finite field number system. The reason why the number used in ECC is in finite field is that the operations over real number system on elliptic curve refer to rounding, and the result will be inaccurate. Besides, in binary finite field using polynomial representation, the addition operation is a simple exclusive OR operation of each corresponding bits between two operands, and this work also chooses this number system. In the following, all the numbers are in binary finite field.

As shown in Fig. 2.6, ECADD is defined by taking two distinct points (P_1 and P_2) on the elliptic curve and drawing a straight line connecting them. Using the third point (T) at which the straight line intersects the elliptic curve, take a reflection on

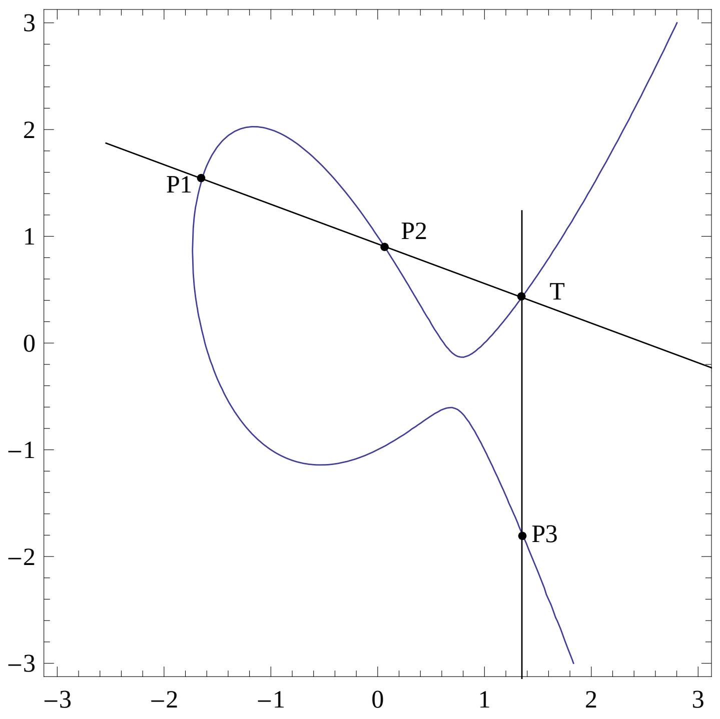

Figure 2.6: Point Addition on Elliptic Curve

the x-axis and the resulting point (P_3) is the definition of the ECADD. We defined a special point $\mathcal{O}$ previously, and it is also used here for point calculating operations. Now, let's first see the rules of ECADD.

1. $P_1 + P_2 = P_2 + P_1$

2. $P_1 + (-P_1) = \mathcal{O}$

3. $P_1 + P_2 = \mathcal{O}$ if $P_1 = -P_2$

As P_3 equals to $-T$, it has the same x-coordinate with T.

$$\begin{aligned} y^2 + xy = x^3 + ax^2 + b \\ y^2 + xy - x^3 - ax^2 - b = 0 \end{aligned} \tag{2.19}$$

From Eq. 2.19, we can get the relation of the y-coordinate between T_y and y_3,

$$\begin{aligned} T_y + y_3 &= -T_x \\ P_3 &= (T_x, -T_x - T_y) \end{aligned} \tag{2.20}$$

Based on the previous sections, given P_1 and P_2, we already know how to get the third intersection T in real number system. In binary finite field, the operations are slightly different, and some equations can be simplified. The addition and subtraction are the same, and are all bitwise XOR operations. Finally, we have,

$$\begin{aligned} x_3 &= (\frac{y_1 + y_2}{x_1 + x_2})^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + a \\ y_3 &= (\frac{y_1 + y_2}{x_1 + x_2})(x_1 + x_3) + x_3 + y_1 \end{aligned} \tag{2.21}$$

2.2.4 Point Doubling on Elliptic Curve

Point doubling is defined similarly with ECADD, except instead of using two distinct points to draw the straight line, it uses the tangent line of a single point (P_1) as shown in Fig. 2.7.

The slope λ can be further simplified in binary finite field

$$\begin{aligned} \lambda &= \frac{3x_1^2 + 2ax_1 + y_1}{2y_1 + x_1} \\ \lambda &= \frac{x_1^2 + y_1}{x_1} \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned} \tag{2.22}$$

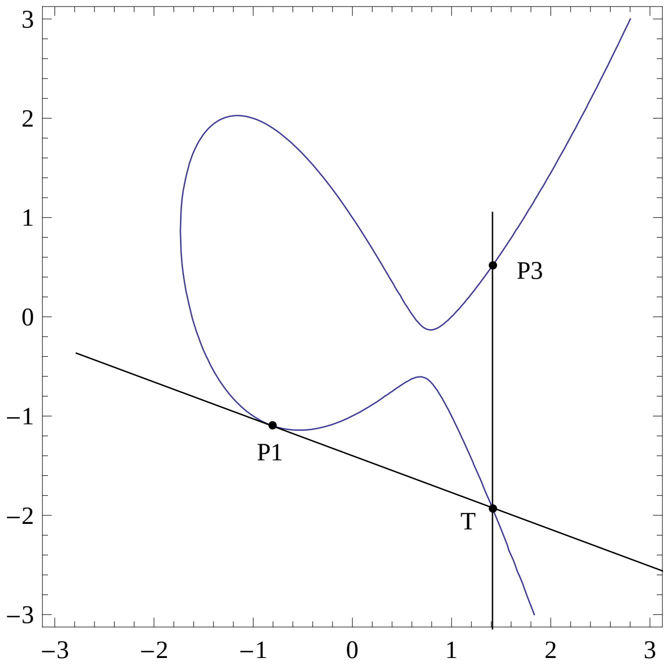

Figure 2.7: Point Doubling on Elliptic Curve

By substituting Eq. 2.22 into Eq. 2.17 and Eq. 2.18 respectively, we have,

$$
\begin{aligned}
T_x &= (x_1 + \frac{y_1}{x_1})^2 + x_1 + \frac{y_1}{x_1} + a \\
T_x &= x_1^2 + \frac{y_1^2}{x_1^2} + x_1 + \frac{y_1}{x_1} + a \\
T_x &= x_1^2 + x_1 + \frac{y_1 + x_1 y_1}{x_1^2} + a \\
T_x &= x_1^2 + x_1 + \frac{x_1^3 + ax^2 + b}{x_1^2} + a \\
T_x &= x_1^2 + x_1 + x_1 + a + \frac{b}{x_1^2} + a \\
T_x &= x_1^2 + \frac{b}{x_1^2}
\end{aligned}
\tag{2.23}
$$

$$
\begin{aligned}
T_y &= (x_1 + \frac{y_1}{x_1})(T_x + x_1) + y_1 \\
T_y &= (x_1 + \frac{y_1}{x_1})T_x + x_1^2.
\end{aligned}
\tag{2.24}
$$

Finally, the ECDBL result P_3 is obtained as follows,

$$\begin{aligned} x_3 &= x_1^2 + \frac{b}{x_1^2} \\ y_3 &= x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3. \end{aligned} \tag{2.25}$$

2.2.5 NIST-recommended random elliptic curves over binary fields

Table 2.1: NIST-recommended random elliptic curves over binary fields

B-163: $m = 163$, $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$, $a = 1$, $h = 2$					
b = 0x	00000002	0A601907	B8C953CA	1481EB10	512F7874
	4A3205FD				
B-233: $m = 233$, $f(z) = z^{233} + z^{74} + 1$, $a = 1$, $h = 2$					
b = 0x	00000066	647EDE6C	332C7F8C	0923BB58	213B333B
		20E9CE42	81FE115F	7D8F90AD	
B-283: $m = 283$, $f(z) = z^{283} + z^{12} + z^7 + z^5 + 1$, $a = 1$, $h = 2$					
b = 0x	027B680A	C8B8596D	A5A4AF8A	19A0303F	CA97FD76
	45309FA2	A581485A	F6263E31	3B79A2F5	
B-409: $m = 409$, $f(z) = z^{409} + z^{87} + 1$, $a = 1$, $h = 2$					
b = 0x	0021A5C2	C8EE9FEB	5C4B9A75	3B7B476B	7FD6422E
	F1F3DD67	4761FA99	D6AC27C8	A9A197B2	72822F6C
	D57A55AA	4F50AE31			
B-571: $m = 571$, $f(z) = z^{571} + z^{10} + z^5 + z^2 + 1$, $a = 1$, $h = 2$					
b = 0x	02F40E7E	2221F295	DE297117	B7F3D62F	5C6A97FF
	CB8CEFF1	CD6BA8CE	4A9A18AD	84FFABBD	8EFA5933
	2BE7AD67	56A66E29	4AFD185A	78FF12AA	520E4DE7
	39BACA0C	7FFEFF7F	2955727A		

NIST recommends five random elliptic curves on binary fields [6] [19], and they are listed in Table 2.1. The $f(z)$ is the reduction polynomial of degree m, and as the key size (m) increases from 163 to 571, security level is increased correspondingly,

Algorithm 1 Binary NAF method [19]

```
Input: a point over $E(GF(2^m))$, a positive $l$-bit integer $k = \sum_{i=0}^{l-1} k_i 2^i, k_i \in \{-1, 0, 1\}$.
Output: $Q = kP$.
//*Initialization*//
$P_1 \leftarrow P, P_2 \leftarrow \mathcal{O}$.
//*PM Loop Process*//
for $i = l - 1$ down to $0$ do
  $P_2 \leftarrow 2P_2$
  if $k_i = 1$ then
    $P_2 \leftarrow P_2 - P_1$
  end if
  if $k_i = -1$ then
    $P_2 \leftarrow P_2 - P_1$
  end if
end for
return $(Q = P_2)$
```

and the computation time and complexity of the system increase as well. NIST also recommends five random elliptic curves on prime fields, and it is more suitable to software implementations as its numbers are rational numbers in finite field, and can employ general microprocessor to do the rational number arithmetic operations. In hardware implementation, the binary finite field is more popular as its finite field arithmetic can be very simple. For example, the addition and subtraction is only exclusive operation on two operands' corresponding bits. Therefore, there is no carry propagation delay, which is usually the bottleneck of the critical path in prime field number system in hardware implementation.

2.3 Elliptic Curve Arithmetic

The algorithms for calculating PM on elliptic curve are also called elliptic curve

Algorithm 2 Montgomery Method [18]

```
Input: a point $P(x,y)$ over $E(GF(2^m))$, a positive $l$-bit integer $k = (k_{l-1}, \cdots, k_1, k_0)_2$ .
Output: $Q = kP$.
//*Initialization*//
$P_1 \leftarrow P, P_2 \leftarrow 2P$.
//*PM Loop Process*//
for $i = l-2$ down to $0$ do
  if $k_i = 1$ then
    $P_1 \leftarrow P_1 + P_2, P_2 \leftarrow 2P_2$.
  else
    $P_2 \leftarrow P_1 + P_2, P_1 \leftarrow 2P_1$.
  end if
end for
return $(Q(x_0, y_0) = P_1)$
```

arithmetic, and it has a conventional hierarchy as shown in Fig. 1.2. There are many methods for calculating kP. The most common one is the binary method, which is the same with the method when we do the multiplication in binary numbers (e.g. $11P = 2(2(2P) + P) + P$). Similar with the multiplier design in binary hardware, there exist many methods to recode the k so that the number of operations of ECADD and ECDBL can be reduced. Signed-digit (SD) representation [19] of k is, $k = \sum_{i=0}^{l-1} k_i 2^i$, and $k_i \in \{-1, 0, 1\}$. If there is no adjacent non-zero digits, the SD form is called as non-adjacent form (NAF) [19]. NAF form is the least weight of any SD representation of k, and it is also unique for every integer k.

2.3.1 Montgomery PM

Montgomery method is a variant of binary method, and it is based on keeping an invariant relationship during the whole kP calculation process, that is, $P2 - P1 = P$. By using this relationship, only $2P$ x-coordinate is needed in the kP loop calculation process, and the final result point's y-coordinate can be obtained from its x-coordinate.

From Eq. 2.21, if $P_1 \neq P_2$ we have

$$\begin{aligned} x_3 &= (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a \\ &= \frac{x_1y_2 + x_2y_1 + x_1x_2^2 + x_2x_1^2}{(x_1+x_2)^2} \end{aligned} \tag{2.26}$$

Here $P = (x_0, y_0)$, and as $P = P_2 - P_1$, then,

$$x = \frac{x_1y_2 + x_2(x_1+y_1) + x_1x_2^2 + x_2x_1^2}{(x_1+x_2)^2}. \tag{2.27}$$

By combining Eq. 2.26 and Eq. 2.27, the x-coordinate of P_3 is

$$x_3 = x + (\frac{x_1}{x_1+x_2})^2 + \frac{x_1}{x_1+x_2} \tag{2.28}$$

Finally, we have

$$x_3 = \begin{cases} x + (\frac{x_1}{x_1+x_2})^2 + \frac{x_1}{x_1+x_2} & if P_1 \neq P_2 \\ x_1^2 + \frac{b}{x_1^2} & if P_1 = P_2 \end{cases} \tag{2.29}$$

Therefore, y-coordinate is not involved in the calculation during the LOOP process. In the following, the y-coordinate can be calculated based on x-coordinate in the final stage. As $P_2 = P_1 + P$, then from Eq. 2.26,

$$x_2 = \frac{x_1y + xy_1 + x_1x^2 + xx_1^2}{(x_1+x)^2} \tag{2.30}$$

$$y_1 = (\frac{x_1}{x} + 1)\{(x_1 + x)(x_2 + x) + x^2 + y\} + x \tag{2.31}$$

Therefore, we only need to calculate the x-coordinate in the loop process, and it can significantly decrease the number of finite field operations.

2.3.2 Projective Coordinates

So far, all the calculations are defined in conventional affine coordinates, and the problem with the Montgomery method in affine coordinates is that there are lots of finite field inversion operations involved in the loop process. As the iteration number of the LOOP is 162 in this work, it means by using Montgomery method, it will consumes nearly 2×162 finite field inversions in total. As the finite field inversion is usually the most time-consuming operation in finite field, many works tried to use projective coordinate [22][25] to reduce the number of inversions.

In standard projective coordinates [8], the projective point on the elliptic curve has the following relationship with its corresponding point on elliptic curve, where $Z \neq 0$.

$$(X, Y, Z) \Longleftrightarrow (X/Z, Y/Z). \tag{2.32}$$

By substituting $(X/Z, Y/Z)$ into Eq. 2.32, the projective form of the EC is

$$ZY^2 + XYZ = X^3 + aX^2Z + bZ^3 \tag{2.33}$$

Algorithm 3 Lopez-Dahab algorithm [14]

```
**Input**: a point $P(x, y)$ over $E(GF(2^m))$, a positive $l$-bit integer $k = (k_{l-1}, \cdots, k_1, k_0)_2$ with $k_{l-1} = 1$.
**Output**: $Q = kP$.
//*Affine To Projective Coordinate Initialization*//
$(X_1, Z_1) \leftarrow (x, 1), (X_2, Z_2) \leftarrow (x^4 + b, x^2)$.
//*PM Loop Process*//
for $i = l - 2$ down to $0$ do
  if $k_i = 1$ then
    $(X_1, Z_1) \leftarrow Madd(X_1, Z_1, X_2, Z_2, x)$,
    $(X_2, Z_2) \leftarrow Mdouble(X_2, Z_2, b)$.
  else
    $(X_2, Z_2) \leftarrow Madd(X_1, Z_1, X_2, Z_2, x)$,
    $(X_1, Z_1) \leftarrow Mdouble(X_1, Z_1, b)$.
  end if
end for
//*Projective To Affine Coordinate Conversion*//
$Q \leftarrow M_{xy}(X_1, Z_1, X_2, Z_2, x, y)$.
return $Q(x_0, y_0)$.
```

Similarly, there are also other projective coordinates such as Jacobian projective coordinates [23], Chudnovsky Jacobian coordinates [23], and Lopez-Dahab projective coordinates. For example, in Jacobian projective coordinate, the projective point $(X, Y, Z), Z \neq 0$ corresponds to $(X/Z^2, Y/Z^3)$ in affine coordinate, and its projective form of elliptic curve is $Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6$.

2.3.3 Lopez-Dahab Algorithm

Lopez-Dahab projective coordinates is one of the most effective way to lower the number of the FF inversion operations. In Lopez-Dahab projective coordinates, each point on elliptic curve is also in the form of (X, Y, Z). However, they have different relation in affine coordinates as $(X, Y, Z) \Longleftrightarrow (X/Z, Y/Z^2)$. Then, its projective form of the elliptic curve is

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6 \tag{2.34}$$

After we have obtained the Lopez-Dahab projective coordinates, let's see how Lopez-Dahab algorithm optimizes the Montgomery method by using its projective coordinates. From Eq. 2.29, we can rewrite it in Lopez-Dahab projective coordinates. If $P_1 = P_2$,

$$\begin{aligned} X_3/Z_3 &= (X_1/Z_1)^2 + b(Z_1/X_1)^2 \\ X_3/Z_3 &= x_1^4 + bZ_1^4/Z_1^2X_1^2 \end{aligned} \tag{2.35}$$

Then,

$$\begin{aligned} X_3 &= X_1^2 + bZ_1^4 \\ Z_3 &= Z_1^2X_1^2. \end{aligned} \tag{2.36}$$

Similarly, if $P_1 \neq P_2$,

$$\begin{aligned} X_3 &= xZ_3 + (X_1Z_2)(X_2Z_1) \\ Z_3 &= (X_1Z_2 + X_2Z_1)^2. \end{aligned} \tag{2.37}$$

Finally, we can get the Lopez-Dahab method in Algorithm 3, where the *Madd* stands for ECADD as in Eq. 2.37, and *Mdoubling* stands for ECDBL as in Eq. 2.36. As we can see, there are no FF inversion operation involved in the LOOP process, and the FF inversions are only need in the final stage in M_{xy}. Therefore, Lopez-Dahab method can significantly improve the performance if the FF inversion operation is very expensive. To summarize, $Madd$, $Mdoubling$ and M_{xy} are defined as follows,

```
$Madd(X_1, Z_1, X_2, Z_2, x_0)$
{
$X \leftarrow X_1Z_2X_2Z1 + x(X_1Z_2 + X_2Z_1)^2;$
$Z \leftarrow (X_1Z_2 + X_2Z_1)^2;$
$Return(X, Z);$
}
$Mdouble(X_1, Z_1, b)$
{
$X \leftarrow X_1^4 + bZ_1^4;$
$Z \leftarrow X_1^2Z_1^2;$
$Return(X, Z);$
}
$M_{xy}(X_1, Z_1, X_2, Z_2, x, y)$
{
$x_k = X_1/Z_1;$
$y_k = [x^2 + y + (x + X_1/Z_1)(x + X_2/Z_2)](x + x_k)/x + y;$
$Return(x_k, y_k);$
}
```

In [10], a slight modification was made to move the condition evaluation to the end of the LOOP as shown in Algorithm 4. By doing so, $Madd()$ and $Mdouble()$

Algorithm 4 Modified Lopez-Dahab algorithm [10]

```
**Input**: a point $P(x,y)$ over $E(GF(2^m))$, a positive $l$-bit integer $k = (k_{l-1}, \cdots, k_1, k_0)_2$ with $k_{l-1} = 1$.
**Output**: $Q = kP$.
//*Affine To Projective Coordinate Initialization*//
$(X_1, Z_1) \leftarrow (x, 1), (X_2, Z_2) \leftarrow (x^4 + b, x^2)$.
if $k_{l-2} = 1$ then
   $Swap(X_1, X_2), Swap(Z_1, Z_2)$
end if
//*PM Loop Process*//
for $i = l - 2$ down to $0$ do
   $(X_2, Z_2) \leftarrow Madd(X_1, Z_1, X_2, Z_2, x)$,
   $(X_1, Z_1) \leftarrow Mdouble(X_1, Z_1, b)$.
   if $(i \neq 0$ and $k_i \neq k_{i-1})$ or $(i = 0$ and $k_i = 1)$ then
      $Swap(X_1, X_2), Swap(Z_1, Z_2)$
   end if
end for
//*Projective To Affine Coordinate Conversion*//
$Q \leftarrow M_{xy}(X_1, Z_1, X_2, Z_2, x, y)$.
return $Q(x_0, y_0)$.
```

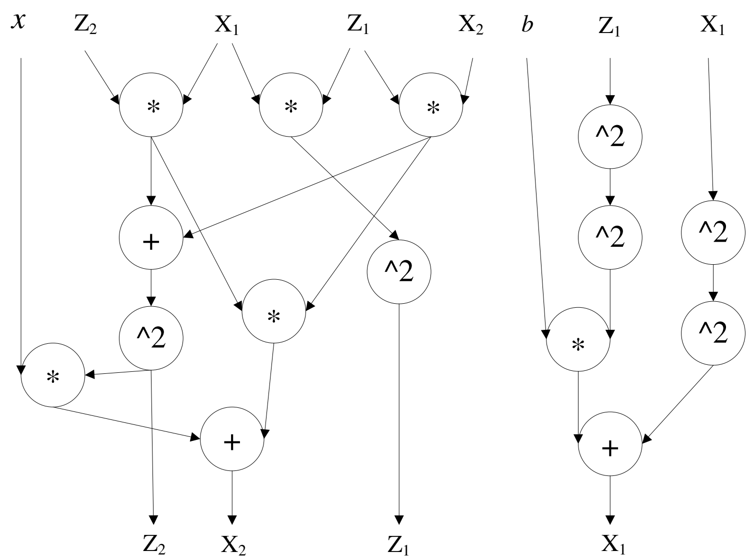

Figure 2.8: Data dependency inside the LOOP

have uniform inputs, and we can easily analyze the data dependency before the swap operation inside the LOOP as shown in Fig. 2.8.

2.3.4 Parallelized Lopez-Dahab Algorithm

In hardware implementations, parallelism is always a good way to improve the performance at the cost of circuit area and power consumption. In [11], the Algorithm 5 is fully parallelized based on its data dependency. The two uniform steps are used to support the data dependency as shown in Fig. 2.9, and as the GNB is used, the square operation is simply a shift operation. Therefore, a uniform block with a addition appended to the output of multipliers is architected in [11].

Algorithm 5 Parallelized version of Lopez-Dahab algorithm with uniform addressing [11]

```
**Input**: $P = (x, y) \in E(GF(2^m))$, an $l$-bit integer k, $k \leftarrow (k_{l-1}, \cdots, k_1, k_0)_2$.
**Output**: $kP = (x_0, y_0)$.
//*Affine To Projective Coordinate Initialization*//
$(X_1, Z_1) \leftarrow (x, 1), (X_2, Z_2) \leftarrow (x^4 + b, x^2)$.
//*PM LOOP Process*//
**if** $k_{l-2} = 1$ **then**
  **Swap**$(X_1, X_2)$, **Swap**$(Z_1, Z_2)$,
**end if**
**for** $i = l - 2$ **down to** $0$ **do**
  1. $T_1 \leftarrow (X_1 Z_2), T_2 \leftarrow (X_2 Z_1), T_3 \leftarrow (X_1 Z_1)^2, Z_3 \leftarrow (T_1 + T_2)^2, Z_2 \leftarrow Z_3$;
  2. $X_2 \leftarrow T_1 T_2 + x Z_3, X_1 \leftarrow b Z_1^4 + X_1^4, Z_1 \leftarrow T_3$
  **if** $(i \neq 0$ **and** $k_i \neq k_{i-1})$ **or** $(i = 0$ **and** $k_i = 1)$ **then**
    **Swap**$(X_1, X_2)$, **Swap**$(Z_1, Z_2)$
  **end if**
**end for**
//*Projective To Affine Coordinate Conversion*//
$x_0 \leftarrow \frac{X_1}{Z_1}$,
$y_0 \leftarrow \frac{1}{x}(x + \frac{X_1}{Z_1})\{(x + \frac{X_1}{Z_1})(x + \frac{X_2}{Z_2}) + x^2 + y\} + y$.
**return** $kP = (x_0, y_0)$.
```

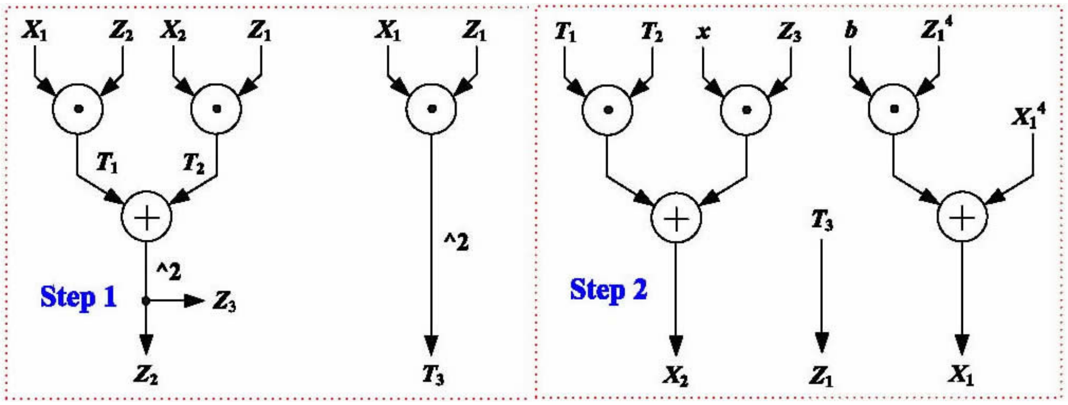

Figure 2.9: Data dependency inside the LOOP in [11]

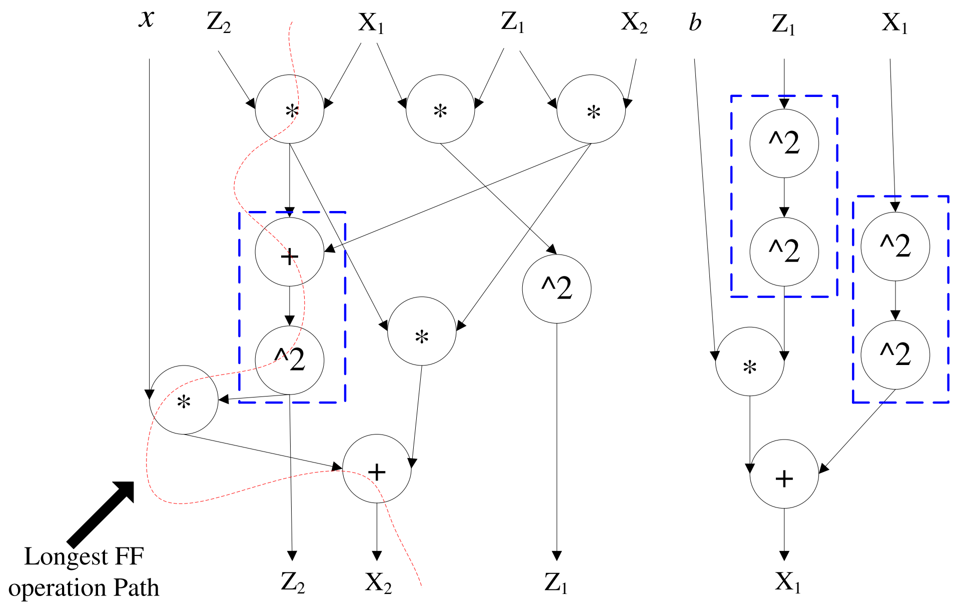

Figure 2.10: Proposed instruction set based on the data dependency

2.3.5 Proposed Instruction-level Parallelism for Parallelized Lopez-Dahab Algorithm

In this work, a processor based architecture is employed to improve the system performance. The advantages of using processor based architecture are that it can be flexible, also it would be easier for us to control the system critical path as the control path and data path can be easily separated and pipelined.

As the program executed on cryptoprocessor is fixed for a certain type of elliptic curve arithmetic algorithm, we can generate a customized instruction set for the algorithm to accelerate the system performance. First, let's review the data dependency as shown in Fig. 2.10. Conventionally, there are only FFM, FFA, FFS instruction available. If so, the longest FF operation path will include two FFM operations, two FFA, and a FFS operation. As we know, these operations are all inside of the LOOP, the total number of the clock cycles to finish the longest FF operation path has significant impact on the system's performance. In this work, we combined FFA and FFS, and instruction $(A + B)^2$ is proposed to finish them in one clock cycle. Therefore, we can save one clock cycle in one iteration, and save 162 in total of PM calculation. An instruction A^4 is also proposed to finish two square operations in one clock cycle, and it can significantly decrease clock cycles by nearly half in FFI operation as it will be describe in Chapter 3.

In this work, the ILP for the parallelized Lopez-Dahab algorithm is achieved on three FF cores as in Algorithm 6. There are three columns of FF operations to execute on each FF core by using the customized FF arithmetic instruction set AB, $A + B$, $(A + B)^2$ and A^4. The NOP in the algorithm stands for empty operation. Apparently, we can replace this customized instruction set by AB, $A + B$ and A^2, but when calculating $(A + B)^2$ and A^4, it would cost two clocks for both. Therefore, instructions $(A+B)^2$ and A^4, which are both done in one clock cycle in the customized instruction set, can decrease the clock cycles. The customized instruction set can also meet the hardware level aspect mentioned above, and it will be described later in critical path analysis. As a FFM costs several clock cycles, the operation A^4 in the loop is computed in parallel with FF multiplication to meet the instruction level aspect mentioned above. The interconnections among three cores are needed for data dependency in the operations in each core. For instance, when arriving step 2 in the loop, core 1 needs the data $V2$ from core 2, which is generated in step 1. Thus an interconnection between core 1 and core 2 is needed to support such data dependency. Similarly, other necessary interconnections can be also obtained.

Algorithm 6 Proposed ILP of parallelized Lopez-Dahab algorithm on three FF cores

```
Input: $P = (x, y) \in E(GF(2^m))$, an $l$-bit integer k, $k \leftarrow (k_{l-1}, \cdots, k_1, k_0)_2$.
Output: $kP = (x_0, y_0)$.
//Affine To Projective Coordinate Initialization
//                  core 1                      core 2          core 3

        1. $X_1 \leftarrow (x + 0)$;            $NOP$;          $Z_2 \leftarrow (x + 0)^2$;
        2. $Z_1 \leftarrow (1 + 0)$;            $NOP$;          $X_2 \leftarrow x^4$;
        3. $NOP$;                              $NOP$;          $X_2 \leftarrow X_2 + b$;

//PM LOOP Process
for $i = l - 2$ down to $0$ do
  //                core 1                      core 2                  core 3

        1. $V_1 \leftarrow X_1Z_2$;             $V_2 \leftarrow X_2Z_1$;    $V_3 \leftarrow X_1Z_1$;
                                                                     $R_3 \leftarrow Z_1^4$;
        2. $Z_2 \leftarrow (V_1 + V_2)^2$;      $NOP$;                  $Z_1 \leftarrow (V_3 + 0)^2$;
        3. $V_1 \leftarrow V_1V_2$;             $V_2 \leftarrow xZ_2$;      $V_3 \leftarrow bR_3$;
                                                                     $R_3 \leftarrow X_1^4$;
        4. $X_2 \leftarrow V_1 + V_2$;          $NOP$;                  $X_1 \leftarrow V_3 + R_3$;

  if $(i \neq 0$ and $k_i \neq k_{i-1})$ or $(i = 0$ and $k_i = 1)$ then
    Swap$(X_1, X_2)$, Swap$(Z_1, Z_2)$
  end if
end for
//Projective To Affine Coordinate Conversion
//                  core 1                      core 2                  core 3

        1. $V_1 \leftarrow Inv(Z_1)$;   $V_2 \leftarrow Inv(Z_2)$;   $V_3 \leftarrow Inv(x)$;
        2. $R_1 \leftarrow X_1V_1$;     $V_2 \leftarrow X_2V_2$;     $R_3 \leftarrow (x + 0)^2$;
                                                           $R_3 \leftarrow R_3 + y$;
        3. $V_1 \leftarrow x + R_1$;    $V_2 \leftarrow x + V_2$;    $NOP$;
        4. $V_1 \leftarrow V_1V_3$;     $V_2 \leftarrow V_2V_1$;     $NOP$;
        5. $NOP$;                      $V_2 \leftarrow V_2 + R_3$;  $NOP$;
        6. $NOP$;                      $V_2 \leftarrow V_1V_2$;     $NOP$;
        7. $NOP$;                      $R_2 \leftarrow V_2 + y$;    $NOP$;

return $kP = (x_0, y_0) = (R_1, R_2)$.
```

Chapter 3
Elliptic curve cryptographic processor

3.1 Finite Field Arithmetic Operations

In this section, we present the algorithms used in this design to implement FF arithmetic instructions. The corresponding critical paths of each FF arithmetic operations are also given for analysis. Before looking into the implementation each FF operations, we will introduce the basic operations in binary finite field.

3.1.1 Basic Finite Field Operations

Binary finite field has two representations: Gaussian Normal Basis (GNB) and Polynomial Basis (PB) representation. As this work focuses on PB representation, here we only describe the number in PB representation. Each binary number has its corresponding PB representation, and an example is shown as follows. In the following, all the operations are based on PB representation.

$$11010010 \rightarrow x^7 + x^6 + x^4 + x \tag{3.1}$$

FF addition and subtraction are the same operation in binary finite field, it only

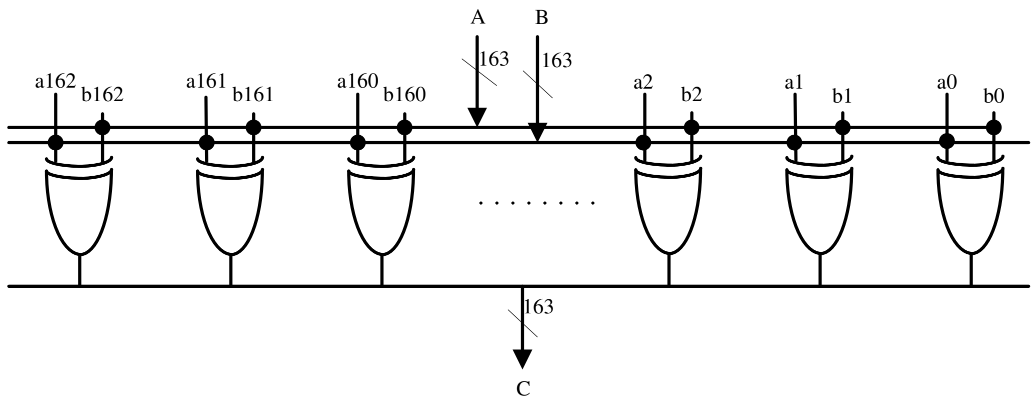

Figure 3.1: Architecture of finite field adder

needs to do the bitwise XOR on two operands. Therefore, the FF adder can be finished in one clock cycle with only a delay of T_{Xor} as shown in Fig. 3.1.

FF multiplication is similar with normal multiplication in real number system except a modular operation is involved in the calculation, and the addition has no propagation delay. Also, an example of FF multiplication is shown in Fig 3.2, where the f(x) is the irreducible polynomial.

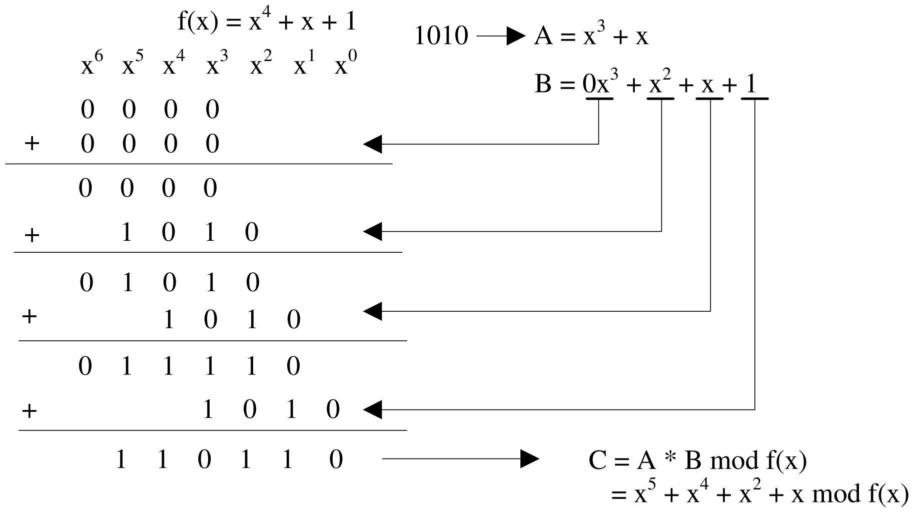

Figure 3.2: An example of FF multiplication

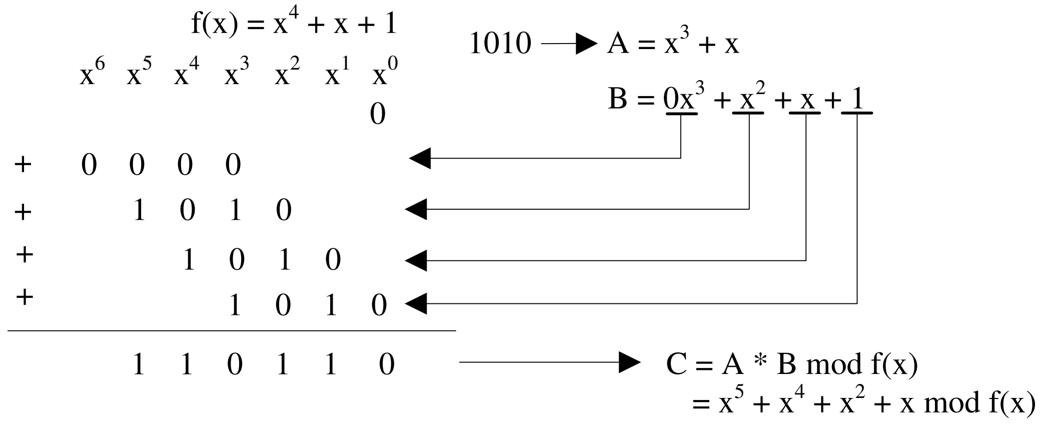

Figure 3.3: An example of pure parallel FF multiplication

From this example, we can easily figure out that there can be bit-serial or full parallel in the hardware implementation of FF multiplier. In bit-serial FF multiplication, it takes one clock cycle to calculate each intermediate result. Therefore, it will take four clock cycles to finish the above calculation (the modular operation is not considered here). In the pure parallel FF multiplier, all the operations are finished in one clock cycle as shown in the Fig. 3.3.

3.1.2 Parallel Finite Field Reduction

So far, we have not considered the modular operation in Fig.s 3.2 and 3.3. The modular operation is also called as FF reduction. It can be calculated after each intermediate step of other FF operations or at the end of them. For example, if a FFM consumes 10 cycles, and we calculate a modular operation after each intermediate cycle of FFM, there will be 10 modular operations in total. On the contrary, we can also calculate only one modular operation when the FFM is finished after 10 cycles, but its hardware complexity is much higher as illustrated in this section.

FF reduction is performed after every FF operation, and the maximum size of the polynomial result that needs FF reduction operation is 325, which is caused by

$$\begin{aligned} C(x) &= [A(x) \times B(x)] \, mod \, f(x) \\ &= [\sum_{i=0}^{162} \sum_{j=0}^{162} a_i b_j x^{i+j}] \, mod \, f(x), \end{aligned} \tag{3.2}$$

where the irreducible polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$, and the $C(x) = c_{325}.x^m + \cdots + c_1.x + c_0, c_i \in \{0, 1\}$. $C(x)$ can be decomposed as

$$C(x) = H(x)f(x) + R(x), \tag{3.3}$$

where the maximum size of $H(x)$ is 162, and $R(x)$ is the reduced result. In this equation, it is easy to find that the parameters c_i with $i > 162$ in $C(x)$ are not determined by reduced variable $R(x)$ but $H(x)f(x)$, then we have

$$h_i = \begin{cases} c_{i+163} & 7 \leq i \leq 161; \\ c_{i+163} \oplus h_{i+156} & i = 6; \\ c_{i+163} \oplus h_{i+157} \oplus h_{i+156} & 3 \leq i \leq 5; \\ c_{i+163} \oplus h_{i+160} \oplus h_{i+157} \oplus h_{i+156} & 0 \leq i \leq 2. \end{cases} \tag{3.4}$$

from Eq. 3.3, the following equations can be obtained,

$$R(x) = C(x) + H(x)f(x). \tag{3.5}$$

$$r_i = \begin{cases} c_i \oplus h_{i-3} \oplus h_{i-6} \oplus h_{i-7} & i = 162; \\ c_i \oplus h_i \oplus h_{i-3} \oplus h_{i-6} \oplus h_{i-7} & 7 \leq i \leq 161; \\ c_i \oplus h_i \oplus h_{i-3} \oplus h_{i-6} & i = 6; \\ c_i \oplus h_i \oplus h_{i-3} & 3 \leq i \leq 5; \\ c_i \oplus h_i & 0 \leq i \leq 2. \end{cases} \tag{3.6}$$

Finally, by combining Eq. 3.4 and Eq. 3.6, the final reduced result can be obtained,

$$W = c_i \oplus c_{i+157} \oplus c_{i+160},$$

$$M = c_i \oplus c_{i+163} \oplus c_{i+319},$$

$$r_i = \begin{cases} W \oplus c_{i+156} & i = 162; \\ W \oplus c_{i+156} \oplus c_{i+163} & 13 \leq i \leq 161; \\ W \oplus c_{i+156} \oplus c_{i+163} \oplus c_{i+312} & \\ & 11 \leq i \leq 12; \\ W \oplus c_{i+156} \oplus c_{i+163} \oplus c_{i+312} \oplus c_{i+314} & \\ & 7 \leq i \leq 10; \\ W \oplus c_{i+163} \oplus c_{i+313} \oplus c_{i+314} \oplus c_{i+316} & i = 6; \\ M \oplus c_{i+160} \oplus c_{i+316} \oplus c_{i+317}; & 3 \leq i \leq 5; \\ M \oplus c_{i+320} & i = 2; \\ M \oplus c_{i+320} \oplus c_{i+323} & 0 \leq i \leq 1. \end{cases} \tag{3.7}$$

Generally, the delay of FF reduction presented above is $\lceil \log_2 7 \rceil T_{Xor}$. However, in some specific FF operations, some parameters of $C(x)$ are zero, thus the FF reduction can be further simplified. For example, if $c_i = 0$, $c_i \oplus c_j = c_j$. The specific delay of FF reduction in each FF operation will be analyzed respectively later in the following.

3.1.3 Word-level finite field multiplier

Algorithm 7 82-bit word-level FF multiplier

```
Input: $A(x), B(x) \in E(GF(2^m))$, and $B(x) = \left[B_3(x)x^{41} + B_2(x)\right] x^{82} + B_1(x)x^{41} + B_0(x)$.
Output: $C(x) \in E(GF(2^m))$.
     $R(x) = 0$;        //Initialize
for $i = 1$ down to $0$ do
  1.     $T_1 = A(x)B_{2i}(x); T_2 = A(x)B_{2i+1}(x)$;
  2.     $C(x) = T_1 + x^{41} * T_2 + x^{82} * R(x)$;
         $R(x) = Reduction(C(x))$;
end for
return $R(x)$.
```

The algorithm of FF multiplier used in this thesis is from [10]. In our design, an 82×163 word-level FF multiplier is used, where two 41×163 FF multipliers are employed in the first level, and the two sub products are summed up in the second level as in Fig. 3.4. In order to support the data loading stage in five-stage pipeline, a 2-input multiplexer is added to select input data registers F and S. The delay of path 1 in the FF multiplier is $T_{Mux} + T_{And} + \lceil \log_2 41 \rceil T_{Xor}$. In path 2, the length of the summed result is 245, therefore, the reduction can be simplified with a delay of $\lceil \log_2 5 \rceil T_{Xor}$ in this FF multiplier. As the summarization and reduction unit are synthesized together, the delay of path 2 is $\lceil \log_2 (3 * 5) \rceil T_{Xor}$.

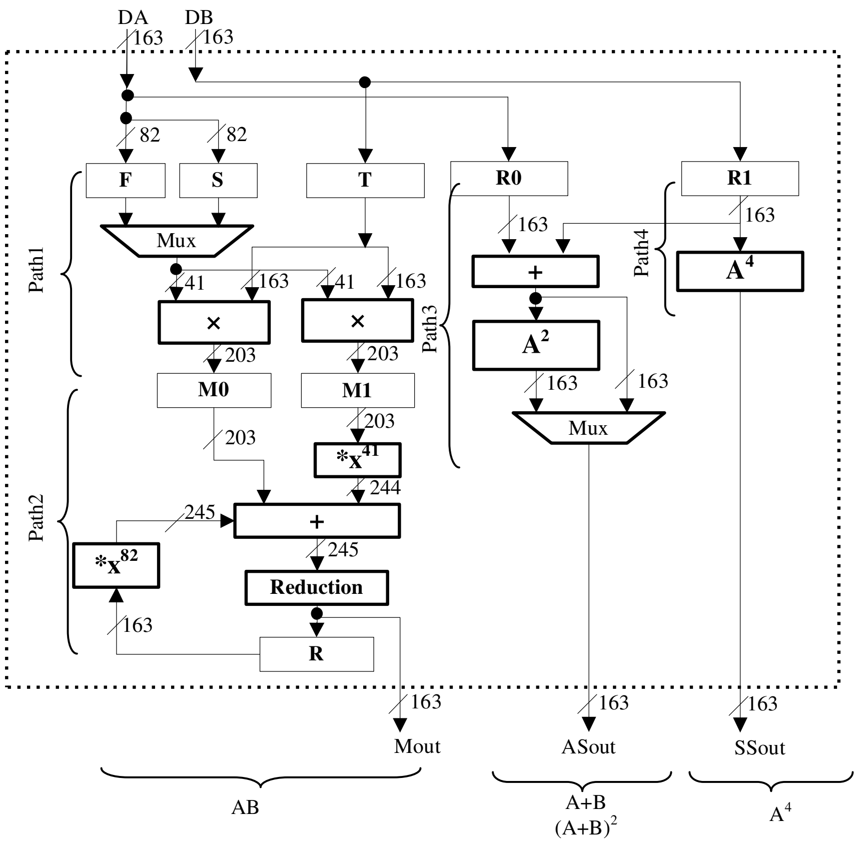

Figure 3.4: The architecture of finite field ALU

3.1.4 FF square and double square

As in Eq. 3.8, $A(x)$ is a binary FF number in PB presentation, and its FF square can be performed by FF multiplication. However, due to its special property (i.e. $A \times A$), it can be simplified by inserting zeros in-between the bits of A as shown in Eq. 3.9, and then do the FF reduction.

$$A(x) = a_{162}x^{162} + a_{161}x^{161} + \cdots + a_i x^i + \cdots + a_0 x^0; \qquad (3.8)$$

$$A^2(x) = a_{162}x^{324} + 0 + a_{161}x^{322} + \cdots + a_ix^{2i}$$
$$+ \cdots + a_2x^4 + 0 + a_1x^2 + 0 + a_0x^0; \tag{3.9}$$

As there are many zeros in $A^2(x)$, the FF reduction of FF square is further simplified by eliminating XOR operations with zeros. Finally the FF square operation has a delay of $\lceil \log_2 5 \rceil T_{Xor}$. By combining A^2 with $A + B$ as in Fig. 3.4, path 3 for $(A + B)^2$ has a delay of $\lceil \log_2(2 \times 5) \rceil T_{Xor} + T_{Mux}$.

In order to accelerate FF inverse operation, we propose a new operation, $A^4(x)$. It can be obtained by combining two $A^2(x)$ together, and its simplification refers to $a_i \oplus 0 = a_i$ and $a_i \oplus a_i = 0$. Finally, the FF operation $C(x) = \mathit{Reduction}(A^4(x))$ is performed in one cycle with a delay of $\lceil \log_2 12 \rceil T_{xor}$.

3.1.5 FF inversion

Table 3.1: Itoh-Tsujii Algorithm for $GF(2^{163})$

i	μ_i	$[\beta_{\mu_{i_1}}(a)]^{2^{\mu_{i_2}}} \times \beta_{\mu_{i_2}}(a)$	$\beta_{\mu_i}(a) = a^{2^{\mu_i}-1}$
0	1	-	$\beta_{\mu_0}(a) = a^{2^1-1}$
1	2	$[\beta_{\mu_0}(a)]^{2^{\mu_0}} \times \beta_{\mu_0}(a)$	$\beta_{\mu_1}(a) = a^{2^2-1}$
2	3	$[\beta_{\mu_1}(a)]^{2^{\mu_0}} \times \beta_{\mu_0}(a)$	$\beta_{\mu_2}(a) = a^{2^3-1}$
3	5	$[\beta_{\mu_2}(a)]^{2^{\mu_1}} \times \beta_{\mu_1}(a)$	$\beta_{\mu_3}(a) = a^{2^5-1}$
4	10	$[\beta_{\mu_3}(a)]^{2^{\mu_3}} \times \beta_{\mu_3}(a)$	$\beta_{\mu_4}(a) = a^{2^{10}-1}$
5	20	$[\beta_{\mu_4}(a)]^{2^{\mu_4}} \times \beta_{\mu_4}(a)$	$\beta_{\mu_5}(a) = a^{2^{20}-1}$
6	40	$[\beta_{\mu_5}(a)]^{2^{\mu_5}} \times \beta_{\mu_5}(a)$	$\beta_{\mu_6}(a) = a^{2^{40}-1}$
7	41	$[\beta_{\mu_6}(a)]^{2^{\mu_0}} \times \beta_{\mu_0}(a)$	$\beta_{\mu_7}(a) = a^{2^{41}-1}$
8	81	$[\beta_{\mu_7}(a)]^{2^{\mu_6}} \times \beta_{\mu_6}(a)$	$\beta_{\mu_8}(a) = a^{2^{81}-1}$
9	162	$[\beta_{\mu_8}(a)]^{2^{\mu_8}} \times \beta_{\mu_8}(a)$	$\beta_{\mu_9}(a) = a^{2^{162}-1}$

Compared to other FF operations, FF inversion is the most time-consuming operation. In this thesis, we adopt Itoh-Tsujii algorithm [17] for FF inversion and in the following we briefly describe the Itoh-Tsujii Algorithm.

In $GF(2^{163})$, any nonzero element a has a cyclic order of $2^{163}-1$. Therefore, the inverse of a can be obtained by $a^{-1} = a^{2^{163}-2}$. Here, we define $\beta_k(a) = a^{2^k-1}, k \in N$, and it has the following property.

$$\begin{aligned} \beta_{k+j}(a) &= a^{2^{k+j}-1} \\ &= \left(\frac{a^{2^k}}{a}\right)^{2^j} \frac{a^{2^j}}{a} = \left(a^{2^k-1}\right)^{2^j} a^{2^j-1} \\ &= \beta_k(a)^{2^j} \beta_j(a) \end{aligned} \tag{3.10}$$

Now, we can decompose $a^{2^{163}-2} = (a^{2^{162}-1})^2$ by a sequence of FF operations as listed in Table 3.1.

As a^{2^s} is frequently performed, by employing A^4, we can decrease the clock cycles needed in FF inversion by nearly half. For example, when calculating $a^{2^{16}}$, we can calculate it with eight successive A^4 instructions in 8 cycles while 16 cycles are needed when using A^2.

3.2 Architecture and implementation

The proposed architecture consists of a main controller and three FF cores as shown in Fig. 3.5. The five-stage pipeline, instruction fetching (IF), instruction decoding (ID), data loading (DL), instruction executing (EX) and writing back (WB), is employed in each FF core. These three FF cores are almost same except some

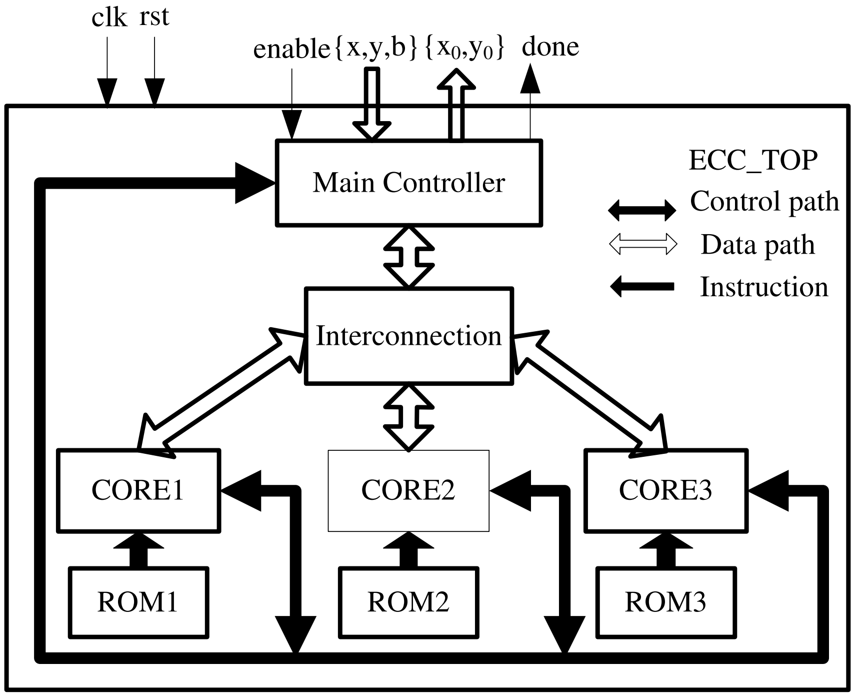

Figure 3.5: The structure of pseudo-multi-core ECC processor

differences in arrangement of register files and interconnection as shown in Fig. 3.6. As there are only several variables involved in the fixed program in Algorithm 6 executed on three cores, the register files of each core in Fig. 3.6 are enough to store the middle results. The instruction set in three finite cores are very similar except some minor differences as shown in Table 3.3, Table 3.4, and Table 3.5.

3.2.1 Instruction Set Design of FF Cores

In order to reduce the complexity caused by the instruction set design, all the instructions in each core have the same length as in Fig. 3.7. As the double square SS need only one source operator, *source* 2 is used in this design.

The *LOOP* instruction is used to control the program counter in each core. It

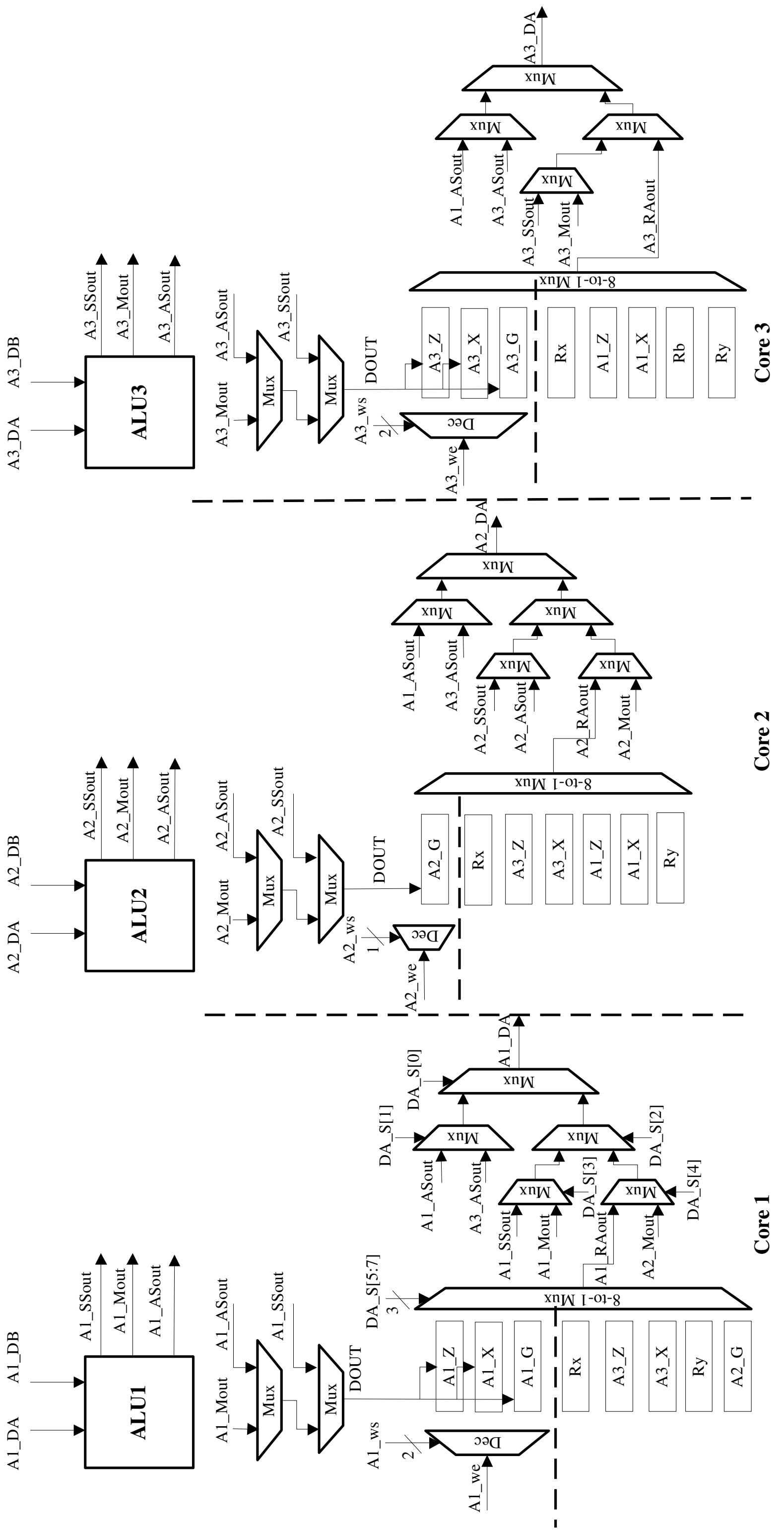

Figure 3.6: Interconnection and register files for FF cores

Table 3.2: Instruction description

Operation	Clock cycles	Description
MUL	2	FF multiplication. The bit width is 163 x 163
SMUL	2	Special FF multiplication, and it is used for swap purpose
SQA	1	FF Square and addition. It is a combined operation
ADD	1	FF addition
SS	1	Double square (finish two square operations in one cycle)
LOOP	1	For iteration purpose
NOP	1	IDLE status for one cycle

Table 3.3: Instruction Set Design of FF Core 1

Operation		Destination		Source	
3'b111	MUL	2'b11	DA1_S	8'b00000_111	SA1_Z
3'b110	SMUL	2'b10	DA1_X	8'b00000_110	SA3_Z
3'b101	SQA	2'b01	DA1_Z	8'b00000_101	SA1_X
3'b100	ADD	2'b00	Res.	8'b00000_100	SA3_X
3'b011	SS	-	-	8'b00000_011	Rx
3'b010	LOOP	-	-	8'b00000_010	Ry
3'b001	Res.	-	-	8'b00000_001	SA1_S
3'b000	NOP	-	-	8'b00000_000	SA2_S
-	-	-	-	8'b10000_xxx	A3_BP_OUT2
-	-	-	-	8'b01000_xxx	A1_BP_OUT2
-	-	-	-	8'b00100_xxx	A1_SS_OUT
-	-	-	-	8'b00010_xxx	A1_BP_OUT1
-	-	-	-	8'b00001_xxx	A2_BP_OUT1

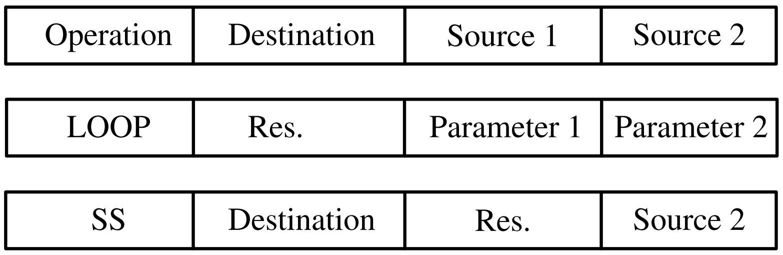

Figure 3.7: Instruction formats in three cores

Table 3.4: Instruction Set Design of FF Core 2

Operation		Destination		Source	
3'b111	MUL	1'b0	Res.	8'b00000_111	SA1_Z
3'b110	SMUL	1'b1	DA2_S	8'b00000_110	SA3_Z
3'b101	SQA	-	-	8'b00000_101	SA1_X
3'b100	ADD	-	-	8'b00000_100	SA3_X
3'b011	SS	-	-	8'b00000_011	Rx
3'b010	LOOP	-	-	8'b00000_010	SA2_S
3'b001	Res.	-	-	8'b00000_001	Ry
3'b000	NOP	-	-	8'b00000_000	Res.
-	-	-	-	8'b10000_xxx	A3_BP_OUT2
-	-	-	-	8'b01000_xxx	A1_BP_OUT2
-	-	-	-	8'b00100_xxx	A2_SS_OUT
-	-	-	-	8'b00010_xxx	A2_BP_OUT2
-	-	-	-	8'b00001_xxx	A2_BP_OUT1

Table 3.5: Instruction Set Design of FF Core 3

Operation		Destination		Source	
3'b111	MUL	2'b11	DA3_S	7'b0000_111	SA1_Z
3'b110	SMUL	2'b10	DA3_X	7'b0000_110	SA3_Z
3'b101	SQA	2'b01	DA3_Z	7'b0000_101	SA1_X
3'b100	ADD	2'b00	Res.	7'b0000_100	SA3_X
3'b011	SS	-	-	7'b0000_011	Rx
3'b010	LOOP	-	-	7'b0000_010	SA3_S
3'b001	Res.	-	-	7'b0000_001	Rb
3'b000	NOP	-	-	7'b0000_000	Ry
-	-	-	-	7'b1000_xxx	A3_BP_OUT2
-	-	-	-	7'b0100_xxx	A1_BP_OUT2
-	-	-	-	7'b0010_xxx	A3_SS_OUT
-	-	-	-	7'b0001_xxx	A3_BP_OUT1

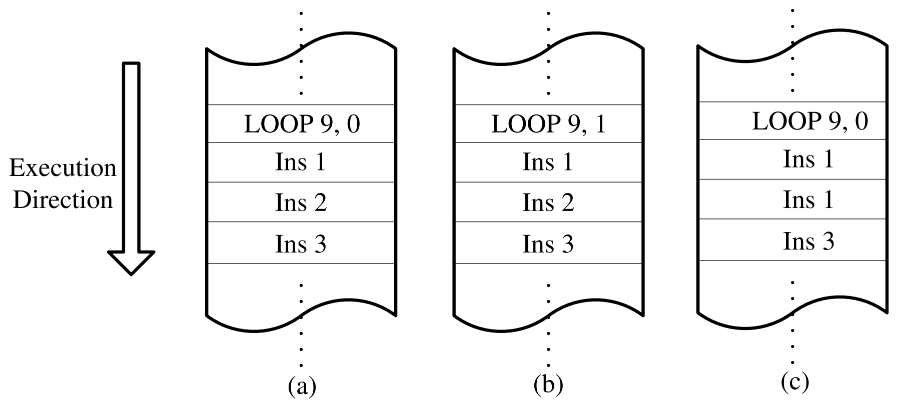

Figure 3.8: Examples of LOOP instruction

has two parameters, the first one is the number of the loops, and the other one is the offset address. The bit length of the *parameter* 1 and *parameter* 2 in core 1 and core 2 just follow the bit length of *source* 1 and *source* 2 correspondingly. As the maximum iterations are this design is 162, and the bit length of *source* 1 in core 3 is 7 bits (maximum value is 127), one bit is borrowed from the ref. field to accomplish maximum number of LOOPs. Due to the instruction fetching and instruction decoding used in the 5-stage pipeline, the *LOOP* instruction has a limit that it can not control the immediately followed instruction. For example, in Fig. 3.8(a), the instruction *LOOP* 9, 0 set the number of loops (9) of the instruction $Ins2$, and it can never include the $Ins1$ in the LOOP although its offset address is 0. However, sometimes we need the LOOP operation on $Ins1$ due to the tight data dependency and limited register files, we can simply accomplish it by the implementation in Fig. 3.8(c), where the $Ins1$ will run 10 times. In Fig. 3.8(b), the offset address is 1, and the *LOOP* operation will affect the instructions from $Ins2$ to $Ins3$.

Registers& Interconnection	8-bit address
A1_Z	00000_111
A3_Z	00000_110
A1_X	00000_101
A3_X	00000_100
Rx	00000_011
Ry	00000_010
A1_G	00000_001
A2_G	00000_000
A3_ASout	1xxxx_xxx
A1_ASout	01xxx_xxx
A1_SSout	001xx_xxx
A1_Mout	0001x_xxx
A2_Mout	00001_xxx

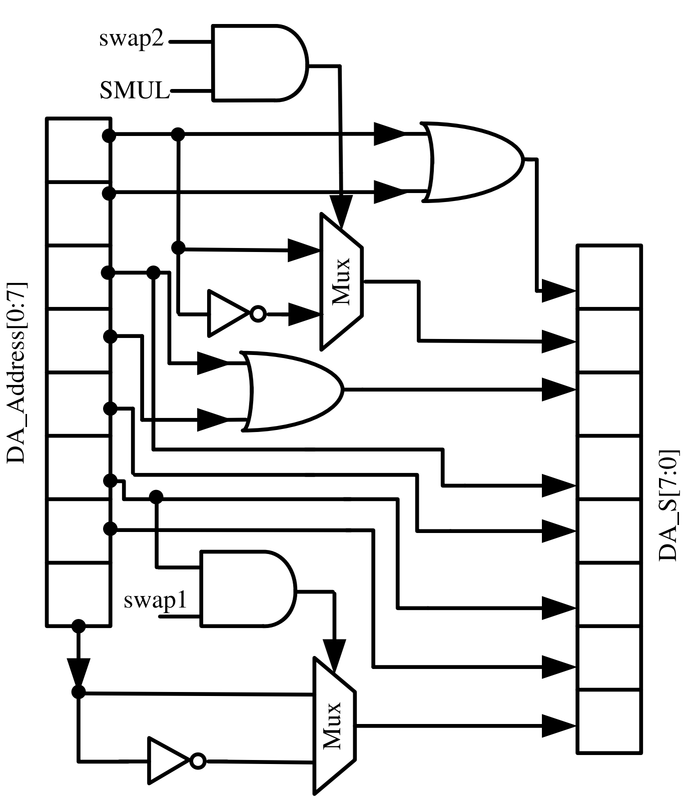

Figure 3.9: Swap logic and address decoding unit of DA in core 1

3.2.2 Register files, interconnection and swap logic

As register files, interconnection and swap logic are tightly related, we analyze them together in this section. In Fig. 3.6, we present the architecture of register files and interconnection for *DA*. Above the dashed line in each core are the local registers, which can only be written by local core. A_i_Z and A_i_X are special registers to store Z_i and X_i in Algorithm 6 (refer to p. 32). A_i_G is the only general register. Under the dashed line in each core are the interconnection and by-pass results from local FF ALU result, which are determined by the ILP data dependency in Algorithm 6. Rx, Ry and Rb are interconnections from the main controller for accessing x, y and *b*.

The address of data path is divided into two levels as shown in Fig. 3.9. The first level has a higher priority, and uses five most-significant bits to distinguish five non-register data paths. The second level uses the rest three least-significant bits to decode register data paths. If the second level is valid, and the third bit from least-significant bit (lsb) of the address is “1”, it represents a special register. By using the arrangement in Fig. 3.9, swap operation between special registers can be easily performed by changing the lsb of the address. In our design, $A1\_X$ and $A1\_Z$ are chosen as the default source for accessing X_1 and Z_1 while X_2 and Z_2 are stored in them at the end of each loop in core 1, which is similar with core 3. Therefore, a swap operation exists by default. The swap logic is composed with two swap signals, $swap1$ and $swap2$, which are generated from the main controller. $swap1$ is used to swap special registers by changing the LSB of the address, and $swap2$ is used to swap by-pass data $A1\_ASout$ and $A3\_ASout$, which are the result of X_i at the end of each loop in Algorithm 6. The first FFM in the loop is defined as swap multiplication (SMUL) to differentiate itself from the common FFM, and with $swap2$, swap operation between $A1\_ASout$ and $A3\_ASout$ can be done. Then, one cycle is saved in one loop when we load the data directly from ALU by-pass output A_Sout for SMUL.

The address unit of DB in core 1 is nearly the same with DA except no need of $swap2$ and $SMUL$. Similarly, the address units in core 2 and core 3 can be obtained.

3.2.3 Main controller

The main controller has two main tasks: provides three data paths Rx, Ry and Rb for the data x, y, b, 0, and 1 to three FF cores, and generates two swap signals $swap1$ and $swap2$. As we have described the swap operation in the previous section, here we only refer to the first task. In order to decrease complexity of the FF cores and interconnection, in our instruction set, there is no data move operation as shown in Algorithm 6. In the initialization stage, we move the data x to $A1\_X$ with the help of the main controller by setting Ry to constant 0, and performing a FF addition between x and 0. Then, the data x is moved into $A1\_X$. Similarly, x^2 is moved into $A3\_Z$, and this is also the way we perform FF square operation in this design. As the data y is only needed in the final coordination conversion stage, Ry is always set with 0 till y is needed in the calculation. The main controller is implemented by using finite state machine.

3.2.4 Critical path analysis

The address unit and interconnection described above is carefully designed by considering the hardware level aspect. In Fig. 3.4, the critical path, path 1 in FF multiplier is $T_{Mux} + T_{And} + \lceil \log_2 41 \rceil T_{Xor}$, and the three by-pass delays path2, path3 and path4 from the FF ALU in Fig. 3.4 are $\lceil \log_2 15 \rceil T_{Xor}$, $\lceil \log_2 10 \rceil T_{Xor} + T_{Mux}$ and $\lceil \log_2 12 \rceil T_{Xor}$ respectively. As the data path DA is similar with DB in core 1, core 2 and core 3, we only need to consider the critical path in DA. As the three by-pass delays in ALU finally go to Ai_DA, then, by adding the previous delay in

Table 3.6: Long paths and comparison

No.	Logic delay[a]	Description
1	$T_{Mux}+T_{And}+\lceil\log_2 41\rceil T_{Xor}$	path1[b]
2	$\lceil\log_2 10\rceil T_{Xor} + 4T_{Mux}$	path3[b] + $\{A2\_ASout$ to $A2\_DA\}$
3	$\lceil\log_2 12\rceil T_{Xor} + 3T_{Mux}$	path4[b] + $\{A1\_SSout$ to $A1\_DA\}$
4	$\lceil\log_2 15\rceil T_{Xor} + 3T_{Mux}$	path2[b] +$\{A1\_Mout$ to $A1\_DA\}$
5	$6T_{Mux}$	{registers to $A1\_DA$}
6	$2T_{And} + 9T_{Xor}$	critical path in [11]

[a] In TSMC18 standard cell library [27], T_{And} is 0.101ns, T_{Xor} is 0.183ns and T_{Mux} is 0.153ns.
[b] Refer to Fig. 3.4 (refer to p. 39)

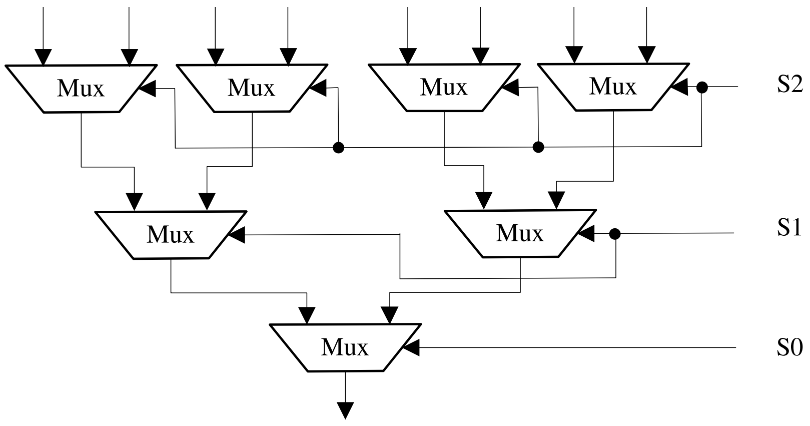

Figure 3.10: Architecture of 8-to-1 multiplexer in Fig. 3.6 (refer p. 43)

ALU with the delay between the by-pass output and Ai_DA, we can get the total delay of three paths. For instance, $A1\_ASout$ goes through two 2-to-1 multiplexers to arrive $A1\_DA$, the total delay is $\lceil\log_2 10\rceil T_{Xor} + 3T_{Mux}$. Similarly, other cases are all obtained in the Table 3.6. The 5th long path is from the second part address decoder unit, which is an 8-to-1 multiplexer and can be implemented by seven 2-to-1 multiplexers as shown in Fig. 3.10. Also, we compare these long paths with the critical path of [11] in the Table 3.6. The longest logic delay path is determined by the ratio of T_{And}, T_{Xor}, and T_{Mux}. Based on the delay parameters provided in TSMC18 technology [27], we can easily get the longest logic delay path in the proposed architecture is path1 in the FF multiplier, and it is approximately $3T_{Xor}$ shorter than the critical path in [11].

3.2.5 Pipeline and timing

Five-stage pipeline (IF, ID, DL, EX and WB) with ILP is employed to increase the performance. In the IF stage, instruction is fetched from each ROM for corresponding core, and stored in the instruction register. During the ID stage, instruction is decoded to generate control signals to FF ALU, and the swap operation is also accomplished in this stage by using the address decoding unit in Fig. 3.9. In the DL stage, data needed for calculation is loaded to the input registers (F, S, T, R_0 and R_1) of FF ALU. In the EX stage, instructions are executed, and the corresponding by-pass results M_{out}, $ASout$ and $SSout$ are generated. In the final stage WB, the result is written into the register.

In the following, we present the timing of loop process of Algorithm 6 in Fig. 3.11 to show how the μ-code is optimized and executed based on the given architecture and pipeline. In this design, FF multiplication AB costs two clock cycles, and other FF operations need only one cycle. There are three features to generate the μ-code:

1. The special register Ai_X is only used to store Xi, and Ai_Z is for Zi so that other cores can access them. Ai_G is used to store other intermediate data.

2. If the next FF operation needs the result from the present FF operation, it can load the by-pass result at the end of the EX stage in the present FF operation

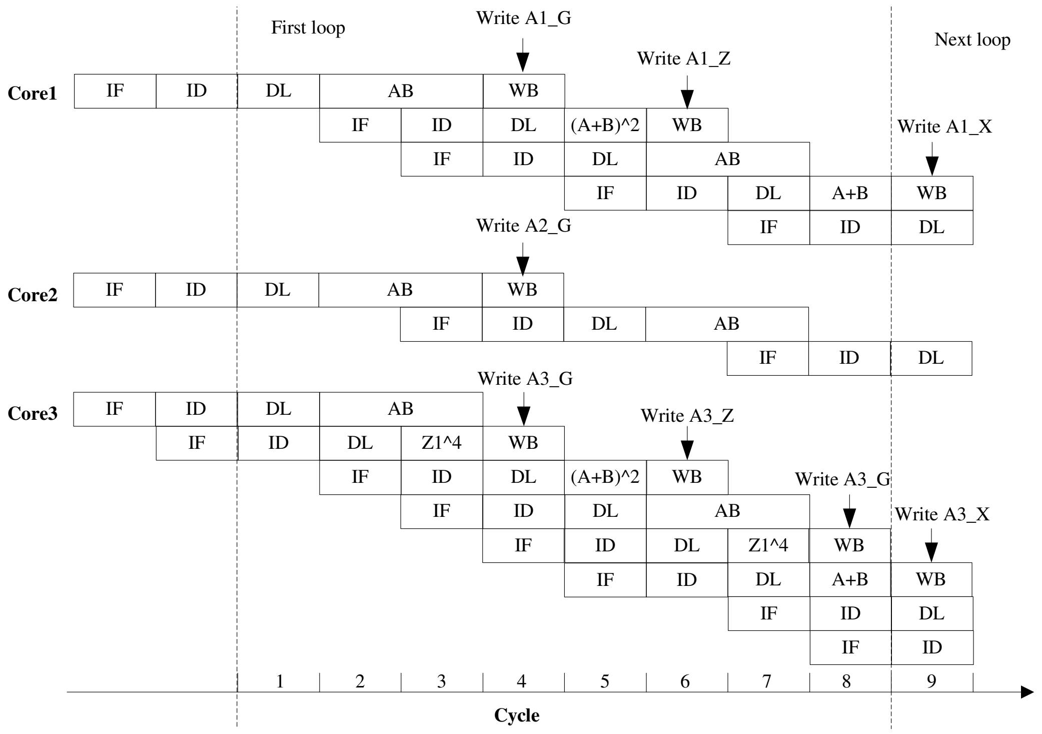

Figure 3.11: Timing in the loop of Algorithm 6

to save one clock cycle. For example, the operation $A + B$ in core 1 can load the by-pass result AB from FF ALUs in core 1 and core 2 at the end of EX stage.

3. If the result from the present FF operation is needed in the next FF operation, and is not needed in the future, it does not need to be written into register file. Thus, we can decrease the number of registers to the given number shown in Fig. 3.6. For instance, as the second multiplication result AB in core 1 is only needed in the next $A+B$ operation and not needed anymore in the future, the WB stage of this instruction can be omitted.

As the loop instruction is used in our instruction set to control the program counter directly, it is apparent that the clock cycles needed for one loop in Algorithm 6 is 8.

Similarly, the μ-code of the initialization stage and the projective to affine coordinate conversion stage in Algorithm 6 can be obtained except the following requirements. During the step 2 in the projective to affine coordinate conversion stage, $R3 \leftarrow R3 + y$ in core 3 needs main controller to change the value in Ry from 0 to y. Besides, as the result $R3$ is needed in core 2, $R3$ can not be stored in general register as there is no interconnection $A3\_G$ to core 2, and it must be stored in a special register. As Xi and Zi are not needed after step 2, special registers can be used for general use afterwards, which can be done by disabling $swap1$ ($swap1 \leftarrow 0$) by the main controller. Therefore, $R3$ can be stored in a special register in core 3. For the same reason, in the step 4 of core 1, $V1$ is stored in a special register to meet the data dependency of core 2. Finally, the PM results (x_0, y_0) are obtained in $A1\_X$ and $A2\_G$ in Fig. 3.6 respectively when the calculation ends.

Chapter 4
Experiment Results

The μ-code in each ROM is presented in Appendix section. When the FF cores are not enabled, they keep fetching the NOP at the address 0 in the ROM, and they remain IDLE. When the input enable signal is set, three FF cores start fetching the instruction in the ROM at the same time. The size of ROM1, ROM2 and ROM2 are 128×21 bits, 128×20 bits, 128×19 bits respectively. All the ROMS are implemented by using combinational logic block. As three ROMs have different length of the valid μ-code, three cores will stop their works at different time, and FF core2 ends its work after the end of FF core1 and FF core3. After each FF core finish their work, they have to remain IDLE to ensure the result not to be altered. Therefore, the μ-code of ROM1 after address 69 are all NOP, the μ-code of ROM2 after address 76 are all NOP, and the μ-code of ROM3 after address 66 are all NOP.

The total clock cycles are composed with three parts. First, 5 clock cycles are needed in the affine to projective coordinate initialization. Then, 8×162 clocks are consumed in the PM loop process. The final coordinate conversion stage consumes 123 cycles, where the FF inversion costs 111 cycles. Therefore, the total clock cycles required for one PM are $5 + 8 \times 162 + 123 + 4 = 1428$, in which the last 4 cycles results from the five-stage pipeline.

Table 4.1: Area information of different blocks

Block	ROM1	ROM2	ROM3	AB	$(A+B)^2$	A^4
Occupied slices	54	50	55	5,037	232	181
Block	FF ALU	core 1	core 2	core 3	Main controller	Total
Occupied slices	5,437	6,993	6,830	6,994	616	20,847

The proposed architecture is coded using verilog HDL, and the μ-code in each ROM is translated to machine code by using Perl script. We firstly verified each submodules in our design in Modelsim by using some direct testcases, and then, we verified the top level of our design by using some direct testcases. All these direct testcaes are elaborately chosen to consider both the corner cases and the ordinary cases. The whole system is simulated in Modelsim. Finally, we implement it on both Xilinx XC4VLX80 FPGA device and TSMC18 technology. We use Xilinx ISE 11.1 to do the synthesis, place and route, and the highest frequency it can reach is 185 MHz with $20,807$ slices. In order to compare the area among different blocks, we synthesize them separately, and the areas of each block are obtained from synthesis report of ISE except the total area, which is obtained after place and route. As we can see, the total area is not equal to the summed area of its components, which is caused by two aspects: there are some global optimizations when synthesizing the whole design together, and some slices are used for routing through. In this table, we can see three FF multipliers occupy nearly three quarter of the total area. When synthesized by Synopsys Design Compiler in TSMC18 CMOS technology [27], the highest frequency it can reach is 263 MHz, and occupies $217,904$ gates. Both implementation results show the critical path lies in the path1 FF multiplier.

Table 4.2: Performance comparison

Work	Technology/ area	Clk(MHz)/ #clk cycles	Time for kP/ Remarks
Kazuo [12] (2007)	$0.13 \mu m$ CMOS	555.6	$12 \mu s$
$GF(2^{163})$	$154K$ gates	-	TNAF method
Kim [11] (2008)	XC4VLX80	143	$10 \mu s$
$GF(2^{163})$	24,363 slices	1446	Three 55-bit GNB mul.
Bijan [10] (2008)	XC2V2000	100	$41 \mu s$
$GF(2^{163})$	3,416 slices	4050	One 41-bit Karatsuba mul.
Kimmo [16] (2008)	Stratix II	-	$49 \mu s$
$GF(2^{163})$	-	-	-
This work	XC4VLX80	185	7.7 μs
$GF(2^{163})$	20,807 slices	1428	Three FF cores
This work	TSMC18	263	5.4 μs
$GF(2^{163})$	217,904 gates	1428	-

We compare our work with several recent works in Table 4.2. To our best knowledge, our work is the fastest implementation over $GF(2^{163})$ in the literature reported. When implementing on Xilinx XC4VLX80 FPGA, our work consumes 77% total time of the work of [11] while the area is only 85.4% of their design. The performance of our work is better than the result in [11] mainly attributes to our higher frequency, which is determined by the short critical path in FF multiplier. Besides, the total clock cycles of our work (1428) is less than that in [11] (1446). Our work uses the same number of FF multiplier (3) with [11]. As the most area-consuming block is the FF multiplier, and the complexity of the FF multiplier using PB presentation is smaller than its counterpart using GNB presentation, the area of our work is sightly smaller than [11].

As the area of our design can not be suited to the FPGA device used in [10], it is hard to accurately compare our result using XC4LX80 with the result in [10] using XC2V2000. Therefore, we only briefly compare them. In [10], only one 42-bit FF multiplier is used, this is why their area is much smaller (6 times) than ours. In turn, the performance gain of our work (5 times faster) mainly result from highly parallel architecture using 3 FF multipliers.

As many papers [12] only use the synthesized result from Synopsys Design Compiler for comparison, we also compare our work in the same way. The TSMC18 ASIC result of our work is faster than that in [12], which uses $0.13\mu m$ CMOS technology. Therefore, our work can be faster than [12] when using a same technology.

From the above comparison, we can see our performance gain mainly results from three factors: high frequency caused by the short critical path of the whole system, highly parallel architecture using three FF multipliers, and small clock cycles caused by using ILP and proposed instruction set, especially A^4.

Chapter 5
Conclusion and Future Work

In this thesis, we proposed a FF arithmetic instruction set AB, $A+B$, $(A+B)^2$ and A^4 for parallelized algorithm for ECC PM, where the $(A+B)^2$ and A^4 are proposed to decrease clock cycles needed in the loop of algorithm and Itoh-Tsujii's finite field inversion respectively while not affecting the system critical path. Then, the register files and interconnection of three FF cores are carefully designed to minimize the critical path and support the data dependency in the proposed algorithm. Finally, a pseudo-multi-core architecture with five-stage pipeline (IF, ID, DL, EX and WB) in each core is obtained to finish the ECC PM.

The implementation of the proposed architecture can finish one ECC PM in 1428 cycles, and is 1.3 times faster than the current fastest implementation over $GF(2^{163})$ reported in literature while consumes only 85.4% of their area on the same FPGA device. Therefore, the proposed architecture and algorithm can be well suited to high performance applications.

As the elliptic curve arithmetic algorithm is largely parallelized based on the data dependency, and three FF cores are employed to support this data dependency in the proposed architecture, the circuit area is still very large. Some potential approaches can still make an optimum trade-off between performance and area can be analyzed to meet different applications as follows:

1. If one FF core, we need to re-analyze the data dependency based on one FF core. The FFM needs to be re-designed to make a balance between the critical path and total clock cycles. The elliptic curve arithmetic algorithm needs to be modified so that it only consumes one FFI as there is only one core available. Otherwise, several FFI operations need to be calculated serially, and hence consume many cycles.

2. If we use two FF cores, the data dependency also needs to be re-analyzed, and FFM needs to be re-designed to make a balance between the critical path and total clock cycles. The elliptic curve arithmetic algorithm needs to be modified to include either one FFI or two FFIs.

3. Also, some FF arithmetic algorithms in [19] can be employed, especially algorithm for FFI, such as extended Euclidean algorithm, to decrease clock cycles.

4. As we only consider the architecture for GF(2^{163}), similar analysis can be used on other curves recommended by [6], especially Kobliz curves as the value of b in the Eq. 2.1 equals to 1 for Kobliz curves. Therefore, some special methods can be employed to simplify the PM calculation based on this feature.

References

[1] Joan Daemen, Vincent Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard". *Springer*, 2002.

[2] Rivest RL, Shamir A, Adleman L (1978) "A method for obtaining digital signatures and public-key cryptosystems". *Commun ACM* pp. 120-126.

[3] The Elliptic Curve Cryptosustem for smart cards, A Certicom White Paper Published: May 1998.

[4] V.S. Miller, "Use of elliptic curves in cryptography", *CRYPTO85: Proceedings of the Advances in Cryptology, Springer-Verlag*, pp. 417-426, 1986.

[5] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, vol. 48, no.177, pp. 203-209, 1987.

[6] NIST, "Recommended elliptic curves for federal government use", May 1999.

[7] IEEE 1363, *Standard Specifications for Publickey Cryptography*, 2000.

[8] D. Hankerson, J. Hernandez, A. Menezes, "Software implementation of elliptic curve cryptography over binary fields", *in: Proceedings of the CHES 2000, Lecture Notes in Computer Science*, vol. 1965, 2000, pp. 1 - 24.

[9] J. Huang, H. Li, and P. Sweany, "An FPGA Implementation of Elliptic Curve Cryptography for Future Secure Web Transaction", *International Conference on Parallel and Distributed Computing Systems*, pp. 296-301, Sept. 2007.

[10] B.Ansari and M.Anwar, "High-Performance Architecture of Elliptic Curve Scalar Multiplication", *IEEE Trans. on Computers*, vol. 57, no. 11, pp. 1443-1452, Nov. 2008.

[11] C.H. Kim, S. Kwon and C.P. Hong, "FPGA implementation of high performance elliptic curve cryptographic processor over $GF(2^{163})$", *Journal of Systems Architecture*, vol. 54, no. 10, pp. 893-900, Apr. 2008.

[12] K. Sakiyama, L. Batina and B. Preneel, "High-performance Public-key Cryptoprocessor for Wireless Mobile Applications", *Mobile Networks and Applications*, vol. 12, no. 4, pp. 245-258, Oct. 2007.

[13] Kimmo J., Jorma S., “Fast point multiplication on Koblitz curves: Parallelization method and implementations”, *Journal of Microprocessors and Microsystems, Elsevier*, 2009.

[14] J. Lopez and R. Dahab, “Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation”, *Proc. First Int’l Workshop Cryptographic Hardware and Embedded Systems*, C.K. Koc and C. Paar, eds., pp. 316-327, 1999.

[15] Louis Dupont, Sebastien Roy and Jean-Yves Chouinard, “A FPGA Implementation of an Elliptic Curve Cryptosystem”, *IEEE International Symposium on Circuits and Systems*, 2006.

[16] K. Jarvinen, and J. Skytta, *“On Parallelization of High-Speed Processors for Elliptic Curve Cryptography”*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, no 9, pp. 1162-1175, Sept. 2008.

[17] T. Itoh and S. Tsujii, “A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases”, *Information and Computation*, vol. 78, no. 3, pp. 171-177, 1988.

[18] P.L. Montgomery, “Speeding the Pollard and Elliptic Curve Methods of Factorization”, *Math of Computation*, vol. 48, pp. 243-264, 1987.

[19] Hankerson D, Menezes A, Vanstone S, Guide to elliptic curves cryptography. Springer, 2004.

[20] Henk C.A. van Tilborg, Fundamentals of Cryptology, Eindhoven University of Technology, Kluwer Academic Publishers, 2000.

[21] Roberto M. Avanzi, Henri Cohen, Christophe Doche, Gerhard Frey, Tanja Lange, Kim Nguyen, Frederik Vercauteren, Handbook of Elliptic and Hyperelliptic Curve Cryptography, Published by Chapman & Hall/CRC, 2006.

[22] J. Lopez and R. Dahab, “Algorithms for Elliptic Curve Arithmetic in GF(2n)”, *SAC’98, LNCS* Springer Verlag, 1998.

[23] D. Chudnovsky and G. Chudnovsky, “Sequences of numbers generated by addition in formal groups and new primality and factoring tests”, *Advances in Applied Mathematics*, 7 (1987), 385-434.

[24] Okeya, K., Takagi, T., Vuillaume, C.: “Efficient representations on Koblitz curves with resistance to side channel attacks”. In: Boyd, C., Gonzalez Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 218-229. Springer, Heidelberg, 2005

[25] A. Menezes, Elliptic curve public key cryptosystems, Kluwer Academic Publishers, 1993.

[26] M. Koschuch, J. Lechner, A. Weitzer, J. Grobschadl, A. Szekely, S.Tillich, and J. Wolkerstorfer, "Hardware/Software Co-Design of Elliptic Curve Cryptography on an 8051 Microcontroller", *Cryptographic Hardware and Embedded Systems*, vol. 4249, pp. 430–444. Springer Verlag, 2006.

[27] TSMC 0.18μm Process 1.8-Volt SAGE-XTM Standard Cell Library Databook, 2001.

Appendix A

μ-code on FF cores

A.1 μ-code in ROM1

```
//Stay idle when the enable signal for point mulitplication is not set
0  NOP
//Begin initialization
1  ADD DA1_X, Rx, Ry
2  NOP
3  ADD DA1_Z, Rx, Ry
// Start the Loop, and it begins from address 6 to 13
// The number of Loop is 161
4  LOOP RES, 161, 7
5  MUL DA1_S, SA1_X, SA3_Z
6  NOP
7  NOP
8  SQA DA1_Z, A1_BP_OUT1, A2_BP_OUT1
9  MUL RES, SA1_S, SA2_S
10 NOP
11 NOP
12 ADD DA1_X, A1_BP_OUT1, A2_BP_OUT1
13 SMUL DA1_S, A1_BP_OUT2, SA3_Z
// The last Loop is from address 13 to 20
14 NOP
15 NOP
16 SQA DA1_Z, A1_BP_OUT1, A2_BP_OUT1
17 MUL RES, SA1_S, SA2_S
18 NOP
19 NOP
20 ADD DA1_X, A1_BP_OUT1, A2_BP_OUT1
// FF inversion calculation is from address 21 to 64
21 SQA RES, SA1_Z, Ry
22 MUL DA1_S, A1_BP_OUT2, SA1_Z
23 NOP
24 NOP
25 SQA RES, A1_BP_OUT1, Ry
26 MUL RES, A1_BP_OUT2, SA1_Z
27 NOP
28 NOP
29 SS RES, XXXX, A1_BP_OUT1
```

```
30 MUL DA1_S, A1_SS_OUT, SA1_S
31 NOP
32 NOP
33 SS RES, XXXX, A1_BP_OUT1
34 SS RES, XXXX, A1_SS_OUT
35 SQA RES, Ry, A1_SS_OUT
36 MUL DA1_S, A1_BP_OUT2, SA1_S
37 NOP
38 LOOP RES, 4, 0
39 SS RES, XXXX, A1_BP_OUT1
40 SS RES, XXXX, A1_SS_OUT
41 MUL DA1_S, A1_SS_OUT, SA1_S
42 NOP
43 LOOP RES, 9, 0
44 SS RES, XXXX, A1_BP_OUT1
45 SS RES, XXXX, A1_SS_OUT
46 MUL DA1_S, A1_SS_OUT, SA1_S
47 NOP
48 NOP
49 SQA  RES, A1_BP_OUT1, Ry
50 MUL RES, A1_BP_OUT2, SA1_Z
51 NOP
52 LOOP RES, 19, 0
53 SS RES, XXXX, A1_BP_OUT1
54 SS RES, XXXX, A1_SS_OUT
55 MUL DA1_S, A1_SS_OUT, SA1_S
56 NOP
57 LOOP RES, 39, 0
58 SS RES, XXXX, A1_BP_OUT1
59 SS RES, XXXX, A1_SS_OUT
60 SQA RES, A1_SS_OUT, Ry
61 MUL RES, A1_BP_OUT2, SA1_S
62 NOP
63 NOP
64 SQA RES, A1_BP_OUT1, Ry
// End of FF inversion
65 MUL DA1_X, A1_BP_OUT2, SA1_X
66 NOP
67 NOP
68 ADD RES, A1_BP_OUT1, Rx
69 MUL DA1_Z, A1_BP_OUT2, SA3_X
```

A.2 μ-code in ROM2

```
//Stay idle when the enable signal for point mulitplication is not set
0 NOP
//Begin initialization
1 NOP
2 NOP
3 NOP
// Start the Loop, and it begins from address 6 to 13
// The number of Loop is 161
4 LOOP RES, 161, 7
5 MUL DA2_S, SA3_X, SA1_Z
6 NOP
7 NOP
8 NOP
9 MUL RES, Rx, A1_BP_OUT2
10 NOP
11 NOP
12 NOP
13 SMUL DA2_S, A3_BP_OUT2,SA1_Z
// The last Loop is from address 13 to 20
14 NOP
15 NOP
16 NOP
17 MUL RES, Rx, A1_BP_OUT2
18 NOP
19 NOP
20 NOP
// FF inversion calculation is from address 21 to 64
21 SQA RES, SA3_Z, Ry
22 MUL DA2_S, A2_BP_OUT2, SA3_Z
23 NOP
24 NOP
25 SQA RES, A2_BP_OUT1, Ry
26 MUL RES, A2_BP_OUT2, SA3_Z
27 NOP
28 NOP
29 SS RES, XXXX, A2_BP_OUT1
30 MUL DA2_S, A2_SS_OUT, SA2_S
31 NOP
32 NOP
33 SS RES, XXXX, A2_BP_OUT1
34 SS RES, XXXX, A2_SS_OUT
35 SQA RES, Ry, A2_SS_OUT
36 MUL DA2_S, A2_BP_OUT2, SA2_S
```

```
37 NOP
38 LOOP RES, 4, 0
39 SS RES, XXXX, A2_BP_OUT1
40 SS RES, XXXX, A2_SS_OUT
41 MUL DA2_S, A2_SS_OUT, SA2_S
42 NOP
43 LOOP RES, 9, 0
44 SS RES, XXXX, A2_BP_OUT1
45 SS RES, XXXX, A2_SS_OUT
46 MUL DA2_S, A2_SS_OUT, SA2_S
47 NOP
48 NOP
49 SQA  RES, A2_BP_OUT1, Ry
50 MUL RES, A2_BP_OUT2, SA3_Z
51 NOP
52 LOOP RES, 19, 0
53 SS RES, XXXX, A2_BP_OUT1
54 SS RES, XXXX, A2_SS_OUT
55 MUL DA2_S, A2_SS_OUT, SA2_S
56 NOP
57 LOOP RES, 39, 0
58 SS RES, XXXX, A2_BP_OUT1
59 SS RES, XXXX, A2_SS_OUT
60 SQA RES, A2_SS_OUT, Ry
61 MUL RES, A2_BP_OUT2, SA2_S
62 NOP
63 NOP
64 SQA RES, A2_BP_OUT1, Ry
// End of FF inversion
65 MUL RES, A2_BP_OUT2, SA3_X
66 NOP
67 NOP
68 ADD RES, A2_BP_OUT1, Rx
69 MUL RES, A1_BP_OUT2, A2_BP_OUT2
70 NOP
71 NOP
72 ADD DA2_S, A2_BP_OUT1, SA3_Z
73 MUL RES, A2_BP_OUT2, SA1_Z
74 NOP
75 NOP
76 ADD DA2_S, A2_BP_OUT1, Ry
```

A.3 μ-code in ROM3

```
//Stay idle when the enable signal for point mulitplication is not set
0 NOP
//Begin initialization
1 SQA DA3_Z, Rx, Ry
2 SS RES, Ry, Rx
3 ADD DA3_X, A3_SS_OUT, Rb
// Start the Loop, and it begins from address 6 to 13
// The number of Loop is 161
4 LOOP RES, 161, 7
5 MUL RES, SA1_X, SA1_Z
6 SS DA3_S, XXXX, SA1_Z
7 NOP
8 SQA DA3_Z, Ry, A3_BP_OUT1
// The last Loop is from address 13 to 20
9 MUL RES, SA3_S, Rb
10 SS DA3_S, XXXX, SA1_X
11 NOP
12 ADD DA3_X, A3_BP_OUT1, SA3_S
13 SMUL RES, A1_BP_OUT2(X1), SA1_Z
14 SS DA3_S, XXXX, SA1_Z
15 NOP
// FF inversion calculation is from address 21 to 64
16 SQA DA3_Z, Ry, A3_BP_OUT1
17 MUL RES, SA3_S, Rb
18 SS DA3_S, XXXX, SA1_X
19 NOP
20 ADD DA3_X, A3_BP_OUT1, SA3_S
21 SQA RES, Rx, Ry
22 MUL DA3_S, A3_BP_OUT2, Rx
23 NOP
24 NOP
25 SQA RES, A3_BP_OUT1, Ry
26 MUL RES, A3_BP_OUT2, Rx
27 NOP
28 NOP
29 SS RES, XXXX, A3_BP_OUT1
30 MUL DA3_S, A3_SS_OUT, SA3_S
31 NOP
32 NOP
33 SS RES, XXXX, A3_BP_OUT1
34 SS RES, XXXX, A3_SS_OUT
35 SQA RES, Ry, A3_SS_OUT
36 MUL DA3_S, A3_BP_OUT2, SA3_S
```

```
37 NOP
38 LOOP RES, 4, 0
39 SS RES, XXXX, A3_BP_OUT1
40 SS RES, XXXX, A3_SS_OUT
41 MUL DA3_S, A3_SS_OUT, SA3_S
42 NOP
43 LOOP RES, 9, 0
44 SS RES, XXXX, A3_BP_OUT1
45 SS RES, XXXX, A3_SS_OUT
46 MUL DA3_S, A3_SS_OUT, SA3_S
47 NOP
48 NOP
49 SQA  RES, A3_BP_OUT1, Ry
50 MUL RES, A3_BP_OUT2, Rx
51 NOP
52 LOOP RES, 19, 0
53 SS RES, XXXX, A3_BP_OUT1
54 SS RES, XXXX, A3_SS_OUT
55 MUL DA3_S, A3_SS_OUT, SA3_S
56 NOP
57 LOOP RES, 39, 0
58 SS RES, XXXX, A3_BP_OUT1
59 SS RES, XXXX, A3_SS_OUT
60 SQA RES, A3_SS_OUT, Ry
61 MUL RES, A3_BP_OUT2, SA3_S
62 NOP
63 NOP
64 SQA DA3_X, A3_BP_OUT1, Ry
// End of FF inversion
65 SQA RES, Rx,  Ry
66 ADD DA3_Z, A3_BP_OUT2, Ry
```