2005

Design of a novel hybrid cryptographic processor

Li, Jianzhou

Lethbridge, Alta. : University of Lethbridge, Faculty of Arts and Science, 2005

DESIGN OF A NOVEL HYBRID CRYPTOGRAPHIC PROCESSOR

Jianzhou Li

ME., Hunan University, 2002

A Thesis

Submitted to the School of Graduate Studies

of the University of Lethbridge

in Partial Fulfilment of the

Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science

University of Lethbridge

LETHBRIDGE, ALBERTA, CANADA

Abstract

A new multiplier that supports fields $GF(p)$ and $GF(2^n)$ for the public-key cryptography, and fields $GF(2^8)$ for the secret-key cryptography is proposed in this thesis. Based on the core multiplier and other extracted common operations, a novel hybrid crypto-processor is built which processes both public-key and secret-key cryptosystems. The corresponding instruction set is also presented. Three cryptographic algorithms: the Elliptic Curve Cryptography (ECC), AES and RC5 are focused to run in the processor.

To compute scalar multiplication kP efficiently, a blend of efficient algorithms on elliptic curves and coordinates selections and of hardware architecture that supports arithmetic operations on finite fields is required. The Nonadjacent Form (NAF) of k is used in Jacobian projective coordinates over $GF(p)$; Montgomery scalar multiplication is utilized in projective coordinates over $GF(2^n)$. The dual-field multiplier is used to support multiplications over $GF(p)$ and $GF(2^n)$ according to multiple-precision Montgomery multiplication algorithms. The design ideas for AES and RC5 are also described.

The proposed hybrid crypto-processor increases the flexibility of security schemes and reduces the total cost of cryptosystems.

Acknowledgments

I would like to express many thanks to my supervisor Dr. Hua Li, for his invaluable advice and ideas on the research and also for his devotion of time to me in the past two years. His support and expertise resolved many hurdles that I encountered throughout the research. I would also like to thank my co-supervisor, Dr. Jim Liu, for his kind encouragement and guidance.

I am also grateful to other committee members Dr. Jackie Rice and Dr. Rob Sutherland for their advice.

Finally, I would like thank my parents for their support of me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The electronic world is increasingly influencing our lives. Every day hundreds of thousands of people interact electronically through e-mail, e-commerce (business conducted over the Internet), ATM machines, or cellular phones. This has led to an increased reliance on the security of information transmitted electronically. By far the most effective ways to ensure network and communication security are related to cryptography. Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather a set of techniques [30]. Two types of cryptographic tools are commonly used: secret-key cryptography and public-key cryptography. For secret-key cryptography, RC5 and AES are two widely used important algorithms, while for public-key cryptography, the vast majority of the products and standards use RSA algorithm based on the integer factorization. Elliptic curve cryptography (ECC) is another approach to public-key cryptography based on the mathematics of elliptic curves. The primary advantage of elliptic curve cryptosystems over RSA is the absence of a sub-exponential-time algorithm that could solve the discrete logarithm problem (DLP) in the elliptic curve groups

[5]. Consequently, ECC can maintain the same level of security with a far smaller key size, therefore reducing processing overhead.

It is necessary to implement cryptographic algorithms in hardware due to the fact that software implementations are too slow to satisfy the real-time requirement. For efficiency reasons, usually hybrid encryption systems are used in practice; a key is exchanged using a public-key cipher, and the rest of the communication is encrypted using a symmetric-key algorithm (which is typically much faster). So it is necessary to design a chip that performs hybrid encryption systems for the user's convenience. Many hardware implementations of cryptosystems have been proposed to speed up the throughput while keeping the circuit area as small as possible. These designs can utilize the hardware resources and customize the architecture to maximize the efficiency of the implementations. However, most of the designs are dedicated to specific cryptographic algorithms. For example, many chips that only can perform AES algorithms have been discussed for secret-key cryptosystems, while others are proposed to speed up the public-key cryptosystems. Very little of the literature deals with hardware designs to implement both secret-key and public-key cryptosystems. A crypto-processor in [18] can perform secret-key algorithms AES and triple-DES, and public-key algorithms RSA and ECC. However, it uses dedicated coprocessor blocks for each algorithm, which consumes lots of hardware area. Also, it can only perform ECC over specific binary field $GF(2^{146})$. In this thesis, a processor that can flexibly deal with both secret-key algorithms for AES and RC5 and public-key algorithms for ECC over fields $GF(p)$ and $GF(2^n)$ with variable parameters is proposed.

1.2 Literature Review

Many hardware implementations on Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) have been presented for elliptic curve

cryptosystems and for AES and RC5, respectively. A review of the previous work is given in this section.

1.2.1 Hardware Speed-up of Secret-key Algorithms

Due to the simple computation compared with public-key cryptography and dedication to specific secret-key algorithm, chip designs used in the hardware speed-up of secret-key cryptography are much simpler. Since the invention of AES, many efficient hardware implementations on FPGA or ASIC are presented [50, 53, 29, 6, 24, 25]. References [26, 46] propose hardware architectures for the RC5 block cipher.

However there is little literature that deals with hardware implementations on several secret-key algorithms in one chip. A bulk encryption crypto-processor dedicated to smart cards was designed which could perform DES and 3DES algorithms [47].

1.2.2 Processor for Elliptic Curve Cryptography

Many hardware implementations on elliptic curve cryptography have been proposed [2, 23, 10, 38, 36, 11, 39, 37, 4, 44, 12, 43, 9]. Binary field $GF(2^n)$ arithmetic is more suitable for fast and compact hardware than a prime field $GF(p)$, because elements over $GF(2^n)$ are unsigned binary numbers and there is no need for carry propagation. However, conventional implementations for ECC over $GF(2^n)$ have little flexibility due to dedicated field parameters. Some designs [2, 23, 10] are based on the fixed size Massey-Omura multiplier [37] using optimal normal bases. Some [38, 36] are based on the specific polynomial bases. On the other hand, conventional ECC hardware designs over $GF(p)$ [38] support only the specific prime numbers. So the restrictions of the conventional approaches reduce the flexibility of hardware implementations and limit the application areas.

The Montgomery multiplication algorithm [32, 21] proposed by P.L Montgomery

in 1985 was for modular multiplication over $GF(p)$ to avoid expensive division computation. The algorithm was then extended to binary field $GF(2^n)$ [22]. This provides the guidance to unify the fields $GF(p)$ and $GF(2^n)$ into one computation component. Several contributions based on dual-field multipliers have been made to support field arithmetic on both $GF(p)$ and $GF(2^n)$. One hardware architecture that uses carry-save adders to perform on dual-field operations is introduced in [44]. Processors based on a fully parallel multiplier that supports dual-field operations are presented in [43, 9], where n-bit operands need to be divided into m w-bit (word size) words to perform multiple-precision operations. This further provides the flexibility to accommodate elliptic curves with different key lengths.

1.2.3 Hardware Implementations for Both Public-key and Secret-key Cryptosystems

To our knowledge, very few hardware implementations of both public-key and secret-key cryptosystems exist. Reference [18] presents a crypto-processor that can perform secret-key algorithms AES and triple-DES, and public-key algorithms RSA and ECC. However, its design uses dedicated coprocessor blocks for each algorithm, which consumes lots of hardware area. Also, it can only perform ECC over specific binary field $GF(2^{146})$.

This section summarizes the previous hardware implementations of secret-key and public-key cryptosystems. They primarily focus on the specific algorithm or specific elliptic curve, thereby lacking certain flexibility. The processor based on a parallel dual-field multiplier makes it possible to perform elliptic curve cryptography on both $GF(p)$ and $GF(2^n)$ with variable parameters. However, the author is aware of very few hardware designs that implement algorithms for both secret-key and public-key systems.

1.3 Contributions

The thesis presents a novel hybrid crypto-processor that can process not only public-key cryptography, such as ECC over $GF(p)$ and $GF(2^n)$ with random key length, but also secret-key algorithms, such as AES and RC5. The main contributions of my thesis are as follows.

1. The multiplier designed in this thesis is a novel 32-bit by 32-bit multiplication-accumulator that integrates multiplications over $GF(p)$, $GF(2^n)$ used in public-key cryptosystem and $GF(2^8)$ used in secret-key cryptosystem.

2. Unlike previous work, the hybrid crypto-processor in this thesis has more flexibility that can perform not only public-key algorithms such ECC over fields $GF(p)$ and $GF(2^n)$, but also secret-key algorithms AES and RC5.

3. The prime number p over $GF(p)$ and the irreducible polynomial over $GF(2^n)$ can easily be changed to improve the security of the processor.

1.4 Thesis Outline

Chapter 2 introduces basic mathematical background which serves as the basis for discussions in later chapters.

In chapter 3, an introduction to cryptography is presented which includes secret-key and public-key cryptosystems.

Chapter 4 describes the architecture of the proposed processor and the implementation details of ECC, AES and RC5. The core arithmetic component such as multi-function multiplier and barrel shifter are presented. The related algorithms used for the ECC design are discussed. Finally, the performance analysis and comparison with previous work are made.

Conclusions and possible future work are given in Chapter 5.

Chapter 2

Mathematical Background

Cryptographic algorithms rely heavily on the base of mathematics. For example, the Advanced Encryption Standard (AES) and Elliptic Curve Cryptography (ECC) are built on properties of finite fields. In this chapter, we review some necessary mathematical background in abstract algebra, in particular finite fields, which are relevant to the work in this thesis.

2.1 Groups, Rings, and Fields

A group $(G, *)$ is a set together with a binary operation $*$ (called the multiplication) on G such that

1. Closure: $g_1 * g_2 \in G$ for all $g_1, g_2 \in G$;
2. Associative: $g_1 * (g_2 * g_3) = (g_1 * g_2) * g_3$ for all $g_1, g_2, g_3 \in G$;
3. Identity element: there is an element $e \in G$ such that $g * e = e * g$ for all $g \in G$;
4. Inverse element: for each element $g \in G$, there exists an element g' such that $g * g' = g' * g = e$.

The element e is called the identity of the group G. The element g' is called the inverse of g, usually denoted by g^{-1}.

A group G is said to be abelian or commutative if $g_1 * g_2 = g_2 * g_1$ for all g_1, $g_2 \in G$. When G is an abelian group, the operation $*$ is denoted by $+$, which is called the addition. The identity element e is denoted by 0, which is called the zero element. The inverse of $g \in G$ is denoted by $-g$, which is called the negative of g.

A ring $(R, +, \times)$ is an nonempty set R with two binary operations $+$ and $\times$, called addition and multiplication, such that

1. $(R, +)$ is an abelian group.
2. Closure under multiplication: $r_1 \times r_2 \in R$ for all r_1, $r_2 \in R$;
3. Associative under multiplication: $r_1 \times (r_2 \times r_3) = (r_1 \times r_2) \times r_3$ for all r_1, r_2, $r_3 \in R$;
4. Multiplication is distributive over addition: $r_1 \times (r_2 + r_3) = r_1 \times r_2 + r_1 \times r_3$ and $(r_1 + r_2) \times r_3 = r_1 \times r_3 + r_2 \times r_3$ for all r_1, r_2, $r_3 \in R$.

A ring R is said to be a commutative ring if $r_1 \times r_2 = r_2 \times r_1$ for all r_1, $r_2 \in R$.

A field $(F, +, \times)$ is a set with two binary operations, called addition and multiplication, such that

1. $(F, +, \times)$ is a commutative ring
2. Multiplicative identity: There exists an element 1 such that $a \times 1 = 1 \times a = a$ for all $a \in F$
3. No zero divisors: If a, $b \in F$ and $a \times b = 0$, then $a = 0$ or $b = 0$;
4. Multiplicative inverse: For $a \in F$ and $a \neq 0$, there exists an element $a^{-1} \in F$ such that $a \times a^{-1} = a^{-1} \times a = 1$

A ring $(R, +, \times)$ is said to be an integral domain if it is a commutative ring with multiplicative identity and it contains no zero-divisors. A field is a set on which one can perform addition, subtraction, multiplication, and division. For this work, we

are only interested in fields with finite numbers of elements, which are called finite fields. More details about abstract algebra can be found in [13].

2.2 Finite Fields

A finite field $(F, +, \times)$ [49, 14] consists of a finite set of elements. The number of elements q in the field is called the order of F. It can be shown that there exists a finite field of order q if and only if q is a prime power p^n, where n is a positive integer and p is a prime number. There is essentially only one finite field of order $q = p^n$, denoted by $GF(q) = GF(p^n)$. Here p is called the characteristics of $GF(p^n)$ and n is called the extension degree. We are especially interested in two cases. For $n = 1$, and $q = p$ $(p \neq 2)$, we have $GF(p)$; for $p = 2$, and $q = 2^n$, we have $GF(2^n)$.

2.2.1 The Finite Field $GF(p)$

The finite field $GF(p)$, where p is an odd prime, also called a prime finite field, is defined as the set of integers $\{0, 1, \ldots, p-1\}$, together with the modular arithmetic operations. Since $GF(p)$ is a field, it should satisfy the conditions mentioned in Section 2.1. Table 2.1 lists the properties of modular arithmetic operations in $GF(p)$.

2.2.2 The Finite Field $GF(2^n)$

The finite field $GF(2^n)$, also called a binary finite field, can be viewed as a vector space of dimension n over $GF(2)$. There exist n elements $\{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$ such that for each element $a \in GF(2^n)$, a can be written uniquely as :
$a = a_0\alpha_0 + a_1\alpha_1 + \ldots + a_{n-1}\alpha_{n-1}$, where $a_i \in \{0, 1\}$
The set $\{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$ is called a basis of $GF(2^n)$ over $GF(2)$. Given such a basis, an element a can also be represented as the bit string $(a_0, a_1, \ldots, a_{n-1})$.

Polynomial basis and normal basis are two kinds of commonly used bases (see Johnson et al [15] for more details).

Commutative laws	$(w + x) \pmod p = (x + w) \pmod p$ $(w \times x) \pmod p = (x \times w) \pmod p$
Associative laws	$[(w + x) + y] \pmod p = [w + (x + y)] \pmod p$ $[(w \times x) \times y] \pmod p = [w \times (x \times y)] \pmod p$
Distributive laws	$[w \times (x + y)] \pmod p = [(w \times x) + (w \times y)] \pmod p$ $[(x + y) \times w] \pmod p = [(x \times w) + (y \times w)] \pmod p$
Identities	$(0 + w) \pmod p = \mathrm{w} \pmod p$ $(1 \times w) \pmod p = w \pmod p$
Additive inverse $(-w)$	For each $w \in GF(p)$, there exists a z such that $w + z = 0 \pmod p$
Multiplicative inverse w^{-1}	For each non-zero $w \in GF(p)$, there exists a value a such that $a \times w = 1 \pmod p$

Table 2.1: Properties of modular arithmetic operations in $GF(p)$

Polynomial Basis

The finite field $GF(2^n)$ can also be viewed as a set of polynomials over $GF(2)$, together with polynomial arithmetics. The polynomials are defined modulo an irreducible polynomial $f(x)$ whose highest power is integer $n-1$. A polynomial $f(x)$ over field $GF(2)$ is called irreducible if and only if $f(x)$ cannot be factored as a product of polynomials in $GF(2)$, with each highest degree less than n. An irreducible polynomial $f(x)$ is also called a prime polynomial by analogy to primes in $GF(p)$. Each $f(x)$ defines a polynomial basis representation of $GF(2^n)$. The $GF(2^n)$ can be expressed in the following form after the irreducible polynomial is determined:
$GF(2^n) = \{a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0 |\ a_i \in \{0,1\}\}$,
or in the bit string form:
$GF(2^n) = \{(a_{n-1}a_{n-2}\ldots a_1a_0) |\ a_i \in \{0,1\}\}$.
The arithmetic operations on the elements in $GF(2^n)$ are as follows.

- Additive identity is represented as $(00\ldots00)$.
- Multiplicative identity is represented as $(00\ldots01)$.
- Addition: Due to the coefficient over $GF(2)$, the addition operation is bitwise

exclusive OR. Suppose $a = (a_{n-1}a_{n-2}\ldots a_0)$ and $b = (b_{n-1}b_{n-2}\ldots b_0)$ are elements of $GF(2^n)$, then $a + b = c = (c_{n-1}c_{n-2}\ldots c_0)$, where $c_i = a_i \oplus b_i$, $i = 0, \ldots, n-1$.

- Multiplication: Suppose $a = (a_{n-1}a_{n-2}\ldots a_0)$ and $b = (b_{n-1}b_{n-2}\ldots b_0)$ are elements in $GF(2^n)$ and $f(x) = f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + \cdots + f_1x + f_0$ is the irreducible polynomial, then the product $r = (r_{n-1}r_{n-2}\ldots r_0) = a \times b \pmod{f(x)}$.

- Inversion: Suppose $a = (a_{n-1}a_{n-2}\ldots a_0)$ is a nonzero element in $GF(2^n)$, then there exits a unique element $c = (c_{n-1}c_{n-2}\ldots c_0)$ to satisfy $c \times a = 1 \bmod f(x)$.

Normal Basis

A normal basis of $GF(2^n)$ is a basis of the form $\{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{n-1}}\}$, where $\beta \in GF(2^n)$. The $GF(2^n)$ can be expressed in the following form after a normal basis is determined:

$GF(2^n) = \{a_0\beta + a_1\beta^1 + \cdots + a_{n-2}\beta^{2^{n-2}} + a_{n-1}\beta^{2^{n-1}} \mid a_i \in \{0,1\}\}$,

or in the bit string form:

$GF(2^n) = \{(a_0a_1\ldots a_{n-2}a_{n-1}) \mid a_i \in \{0,1\}\}$.

The arithmetic operations on the elements of $GF(2^n)$ are described in the following.

- Additive identity is represented as $(00\ldots00)$.

- Multiplicative identity is represented as $(11\ldots11)$.

- Addition: Due to the coefficients over GF(2), the addition operation is bitwise exclusive OR. Suppose $a = (a_0a_1\ldots a_{n-1})$ and $b = (b_0b_1\ldots b_{n-1})$ are elements of $GF(2^n)$, then $a + b = c = (c_0\ c_1\ldots c_{n-1})$, $c_i = a_i \oplus b_i$, $i = 0, \ldots, n-1$.

- Multiplication: Suppose $a = (a_0a_1\ldots a_{n-1})$ and $b = (b_0b_1\ldots b_{n-1})$ are elements of $GF(2^n)$, then

$c = a \times b = (\sum_{i=0}^{n-1} a_i \beta^{2^i}) \times (\sum_{i=0}^{n-1} b_i \beta^{2^i})$
$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j \beta^{2^i} \beta^{2^j} = \sum_{k=0}^{n-1} c_k \beta^{2^k}$.

We can write

$$\beta^{2^i} \beta^{2^j} = \sum_{k=0}^{n-1} \lambda_{i,j}^{(k)} \beta^{2^k}$$

Substitution yields

$$c_k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j \lambda_{i,j}^{(k)} \tag{2.1}$$

- Squaring: Suppose $a = (a_0 a_1 \ldots a_{n-1})$ is an element of $GF(2^n)$, then
  $a^2 = (\sum_{i=0}^{n-1} a_i \beta^{2^i})^2 = \sum_{i=0}^{n-1} a_i (\beta^{2^i})^2 = \sum_{i=0}^{n-1} a_i \beta^{2^{i+1}} = \sum_{i=0}^{n-1} a_{i-1} \beta^{2^i}$.
  The squaring is also represented in the string binary form:
  $(a_0 a_1 a_2 \ldots a_{n-1})^2 = (a_{n-1} a_0 a_1 \ldots a_{n-2})$.

- Inversion: Suppose $a = (a_0 a_1 \ldots a_{n-1})$ is a nonzero element of $GF(2^n)$, then there exists a unique element $c = (c_0 c_1 \ldots c_{n-1})$ to satisfy $c \times a = 1$.

In normal basis representation, the squaring operation is only the simple left circular shift. However, multiplication can be cumbersome in general. For some special finite fields of $GF(2^n)$, there exist Optimal Normal Bases (ONB) to simplify the multiplication. An ONB [34] is one with the minimum number of nonzero terms in Equation 2.1.

2.3 Summary

In this chapter, the concept of finite fields is introduced. Two kinds of important finite fields $GF(p)$ and $GF(2^n)$ are discussed, which are the mathematical fundamental related to both public-key and secret-key cryptographic algorithms. Furthermore, the two bases representations of $GF(2^n)$, polynomial basis and normal basis, are explained. Even though the squaring in normal basis can be performed very efficiently, the arithmetic component for squaring is dedicated to a specific finite

field, which leads to lack of flexibility. Polynomial basis is chosen in this processor design, because finite fields with different parameters need to be accommodated.

Chapter 3

Cryptography

In this chapter, some basic terms used in cryptography are given. Two kinds of cryptosystems: the secret-key and public-key cryptosystems are discussed. The secret-key cryptographic algorithms: Advanced Encryption Standard (AES) and RC5, are introduced. Finally, details of public-key cryptographic algorithms focusing on Elliptic Curve Cryptography (ECC) are presented.

3.1 Terminology

Cryptography is the technique of converting data into a secret code for transmission over a public network. The original message, known as the *plaintext*, is converted into a coded message, called *ciphertext*. The process of converting from plaintext to ciphertext is called *encryption*, while the process of converting from ciphertext into plaintext is *decryption*. The theme used for encryption is called the *cryptographic system* or *cipher*. Encryption algorithms use a key, which is a binary secret number typically ranging from 40 to 256 bits in length, to control how the ciphertext is produced. At the receiving end, a key is also used to restore the plaintext. Cryptographic systems can be divided into two types according to the number of keys used. If both sender and receiver share the same key, the system is known as *secret-key, symmetric-key, single-key*, or *conventional* cryptography. Conversely, if the sender and receiver use different keys, the system is known as *public-key, asymmetric-key,*

or *two-key* cryptography. The secret-key cryptosystem can further be distinguished as block cipher and stream cipher. The block cipher processes the plaintext one block of elements at a time and produces the corresponding block of ciphertext elements. The stream cipher processes the elements of plaintext continuously and produces ciphertext one element at a time. The secret-key algorithms used in this thesis are all block ciphers.

3.2 Secret-key System

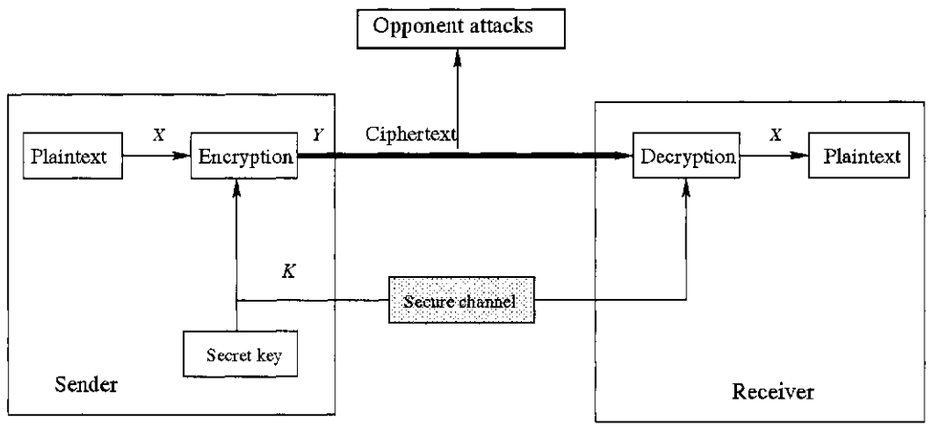

Figure 3.1: Model of secret-key cryptosystem

The model of secret-key cryptosystem shown in Fig. 3.1 gives the processes for encryption and decryption. The encryption algorithm can be written in the following form with plaintext X and secret key K as input and ciphertext Y as output.

$$Y = E_K(X) \tag{3.1}$$

From this formula we can see that the ciphertext Y is determined by both the encryption algorithm E and the key K. Similarly, the decryption inverts the transformation using the same secret key K and the corresponding decryption algorithm.

$$X = D_K(Y) \tag{3.2}$$

The security of the secret-key encryption depends on a strong encryption algorithm and the key size. The key is kept secret while the encryption algorithm is open. So the design of algorithm requires that the opponents cannot be able to decrypt ciphertext without knowing the secret key and to figure out the key even if they know a number of pairs of plaintext and ciphertext.

DES, AES and RC5 are three commonly used secret-key algorithms.

The main challenge in secret-key cryptosystems lies in how to have the sender and receiver share the secret key while keeping it secret without anyone else finding out. If the secret keys are in separate physical locations, a trusted third-party such as a courier, phone system, or some other transmission medium must be responsible for distributing the keys. The generation, transmission and storage of keys are called *key management.*

3.3 Public-key System

Whitfield Diffie and Martin Hellman introduced the concept of public-key cryptography in 1976 in order to solve the key management problem [8]. Public-key cryptosystems have two primary themes, encryption and authentication (digital signatures) as illustrated in Fig. 3.2. In the two systems, each participant gets a pair of keys, one referred to as the public key KU and the other referred to as the private key KR. The public key is published, while the private key is kept secret. The need for the sender and receiver to share secret information is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. In the encryption theme (Fig. 3.2a), the sender A uses receiver B's public key information KU_b to send confidential messages which can only be decrypted with B's private key, KR_b. This process can be expressed as follows for the sender A:

$$Y = E_{KU_b}(X) \tag{3.3}$$

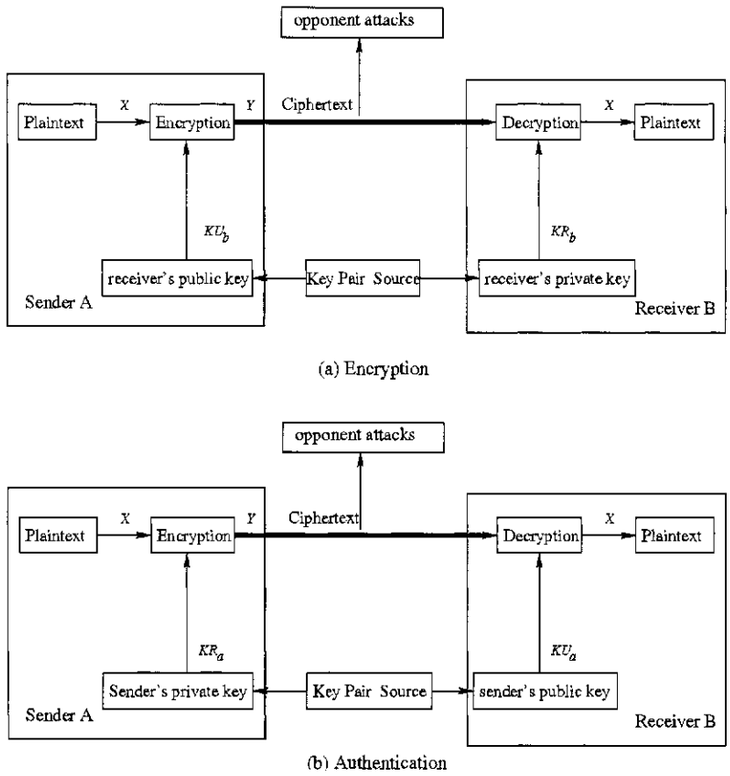

Figure 3.2: Model of public-key cryptosystem

And for the receiver B, the inverted transformation is in the following formula.

$$X = D_{KR_b}(Y) \tag{3.4}$$

In the authentication theme (Fig. 3.2b), the sender A uses his or her private key KR_a to encrypt the message sent to B and B decrypts using A's public key KU_a. Because only sender A is in possession of the private key to encrypt the message X, the encrypted message Y serves as the digital signature of A. This process can be expressed as follows for the sender A:

$$Y = E_{KR_a}(X) \tag{3.5}$$

For the receiver B, the inverted transformation is given in the following formula.

$$X = D_{KU_a}(Y) \tag{3.6}$$

In a public-key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. Typically, the defense against this is to make the problem of deriving the private key from the public key as difficult as possible. For instance, some public-key cryptosystems are designed such that deriving the private key from the public key requires the attacker to factor a large number. In this case it is computationally infeasible to perform the derivation. This is the idea behind the RSA public-key cryptosystem.

3.4 Elliptic Curve Cryptography (ECC)

The security of the public-key cryptography depends on the trap-door one-way function. A one-way function maps a domain into a range such that every function has a unique inverse with the property that it is easy to calculate in one direction and infeasible to calculate in the other direction unless certain additional information is known. This can be summarized as the following three formulas:

1. $Y = f_K(X)$ easy to calculate, if key K and X are known;
2. $X = f_K^{-1}(Y)$ easy to calculate, if key K and Y are known;
3. $X = f_K^{-1}(Y)$ infeasible, if Y is known but K is not known.

Here the private key K is the trap-door.

Since the introduction of the public-key cryptography concept, only two cryptosystems have been invented, RSA and ECC. The RSA was published in 1978 by Rivest, Shamir and Adleman [41]. It uses exponentiation modulo a product of two large primes to encrypt and decrypt. Its security is based on the difficulty of factoring large integers. The introduction of Elliptic Curve Cryptography (ECC) independently by Neal Koblitz [19] and Victor Miller [31] in the mid' 80s has yielded a new family of analogous public-key algorithms. Although mathematically more

complex, elliptic curves appear to provide a more efficient way to leverage the discrete logarithm problem, particularly with respect to the key size.

Because a large number of elliptic curves are in use, it is necessary for an ECC processor to be able to handle different elliptic curves and the underlying fields.

The security of ECC relies on the discrete logarithm problem for the group of points on an elliptic curve defined over a finite field. The main advantage of ECC over systems based on the multiplicative group is the absence of a sub-exponential-time algorithm for solving the underlying hard mathematical problem in ECC, i.e. the Elliptic Curve Discrete Logarithm Problem (ECDLP). Consequently, a significantly smaller parameter can be used in ECC while maintaining the equivalent levels of security. It results in a smaller key size, bandwidth, and electrical power, and is especially attractive in applications where computational power and space are constrained, such as smart cards and wireless devices.

Good overviews of elliptic curve cryptography can be further found in [28, 20]

3.5 Elliptic Curves over Finite Fields

Elliptic curve cryptography is based on elliptic curves over finite fields. Two types of finite fields have been introduced in Section 2.2, Chapter 2. The Weierstrass equations for elliptic curves defined on these two fields $GF(p)$ and $GF(2^n)$ are described in the following subsections, respectively.

3.5.1 Elliptic Curves over $GF(p)$

The elliptic curves over $GF(p)$ (where p is an odd prime and $p > 3$) is defined by the equation

$$y^2 = x^3 + ax + b \tag{3.7}$$

where the parameters $a, b \in GF(p)$ and $4a^3 + 27b^2 \neq 0 \pmod{p}$. The set of solutions (or points) $P = (x_p, y_p)$ where $x_p, y_p \in GF(p)$, together with a special point O (called

the point at infinity) constitute the set $E_p(a,b)$. A finite abelian group $(E_p(a,b), +)$ is defined on the set $E_p(a,b)$ with the O acting as its additive identity. According to the rule of abelian group, the addition operation in $E_P(a,b)$ for all points P, $Q \in E_p(a,b)$ in affine coordinate is as follows.

1. $P + O = O + P = P$

2. If $P = (x_p, y_p)$, then $P + (x_p, -y_p) = O$. The point $-P = (x_p, -y_p)$ is called the negative of P

3. If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ and $P \neq \pm Q$, then $R = P + Q = (x_R, y_R)$, where

$$x_R = (\lambda^2 - x_p - x_Q) \ (mod\ p) \tag{3.8}$$

$$y_R = (\lambda(x_p - x_R) - y_P) \ (mod\ p) \tag{3.9}$$

   where $\lambda = (\frac{y_Q - y_P}{x_Q - x_P}) \ (mod\ p)$.

4. If $P = (x_P, y_P)$, then $R = P + P = 2P = (x_R, y_R)$, where

$$x_R = (\lambda^2 - x_p - x_Q) \ (mod\ p) \tag{3.10}$$

$$y_R = (\lambda(x_p - x_R) - y_P) \ (mod\ p) \tag{3.11}$$

   where $\lambda = (\frac{3y_Q^2 + a}{2y_P}) \ (mod\ p)$. This operation is referred to as the doubling of a point.

5. Scalar multiplication (or point multiplication) kP is defined as repeated addition of P to itself k times.

3.5.2 Elliptic Curves over $GF(2^n)$

Elliptic curves over $GF(2^n)$ are defined by the equation

$$y^2 + xy = x^3 + ax^2 + b \tag{3.12}$$

where the parameters $a, b \in GF(2^n)$. The set of solutions (or points) $P = (x_p, y_p)$ where $x_p, y_p \in GF(2^n)$ together with a special point O called the point at infinity constitute the set $E_{2^n}(a, b)$. A finite abelian group $(E_{2^n}(a, b), +)$ is defined on the set $E_{2^n}(a, b)$ with the a point at infinity O acting as its additive identity. According to the rule of abelian group, the addition operation in $E_{2^n}(a, b)$ for all points $P, Q \in E_{2^n}(a, b)$ in affine coordinate is as follows.

1. $P + O = O + P = P$.

2. If $P = (x_p, y_p)$, then $P + (x_p, x_p + y_p) = O$. The point $-P = (x_p, x_p + y_p)$ is called the negative of P.

3. If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ and $P \neq \pm Q$, then $R = P + Q = (x_R, y_R)$, where

$$x_R = \lambda^2 + \lambda + x_p + x_Q + a \tag{3.13}$$

$$y_R = \lambda(x_p + x_R) + x_R + y_P \tag{3.14}$$

   where $\lambda = (\frac{y_Q + y_P}{x_Q + x_P})$.

4. If $P = (x_P, y_P)$, then $R = P + P = 2P = (x_R, y_R)$, where

$$x_R = \lambda^2 + \lambda + a \tag{3.15}$$

$$y_R = \lambda(x_P + x_R) + x_R + y_p \tag{3.16}$$

   where $\lambda = (x_P + \frac{y_P}{x_P})$. This operation is referred to as the doubling of a point.

5. Scalar multiplication (or point multiplication) kP is defined as repeated addition of P to itself k times.

3.6 ECC Domain Parameters

Some definitions are required before the introduction of ECC domain parameters. The *order* of a point P on an elliptic curve is the smallest positive integer r such

that $rP = O$. $kP = lP$ if and only if $k = l \ (mod\ r)$ and $k, l \in Z_n$. The number of points of $E(GF(q))$, denoted by $\#E(GF(q))$, is known as the *curve order* of the curve. Hasse's theorem states that $\#E(GF(q)) = q + 1 - t$, where $|t| \leq 2\sqrt{q}$. ECC domain parameters over $GF(q)$ are a septuple:

$$T = (q, FR, a, b, G, n, h) \tag{3.17}$$

where q specifies a prime power ($q = p$ or $q = 2^m$, here m is denoted to distinguish the power of m from the ECC domain parameter n); FR (field representation) indicates the method used for representing field elements in $GF(q)$; two field elements a and $b \in GF(q)$ specify the equation of the elliptic curve E over $GF(q)$; $G = (x_G, y_G) \in E_q(a, b)$ is the base point on $E(GF(q))$; the prime number n is the order of G and the integer h is the cofactor $h = \#E(GF(q))/n$. The ECC key length is defined to be the bit-length of n, because n is the primary security parameter.

3.7 Key Generation

An entity A's public and private key pair is associated with a particular set of elliptic curve domain parameters (q, FR, a, b, G, n, h). To generate a key pair, entity A does the following:

1. Select a random or pseudo-random integer d in the interval [1, n-1].
2. Compute $Q = dG$.
3. A's public key is a point Q in $E_q(a, b)$ and A's private key is an integer d.

3.8 Elliptic Curve Protocols

Elliptic curve Diffie-Hellman (ECDH), the Elliptic Curve Digital Signature Algorithm (ECDSA) and the Elliptic Curve Authenticated Encryption Scheme (ECAES)

are three fundamental protocols based on elliptic curves. The ECDH is the elliptic curve analog of Diffie-Hellman key exchange; the ECDSA is the elliptic version of the DSA proposed by Scott Vanstone [52] in 1992; and the ECAES is a variant of the ElGamal public-key encryption theme proposed by Abdalla, Bellare and Rogaway [1] in 1999. Only the simple ECDH is described here and relevant references can be referred to for other protocols. Assume participants A and B share the same domain parameters $D = (q, FR, a, b, G, n, h)$. The key exchange between A and B can be accomplished as follows:

1. A selects its private key d_A and generates a public key Q_A according to key generation procedure.

2. B selects its private key d_B and generates a public key Q_B according to key generation procedure.

3. A computes $P_A = d_A Q_B = (x_A, y_A)$ and B computes $P_B = d_B Q_A = (x_B, y_B)$.

4. Check that $P_A \neq O$, $P_B \neq O$.

5. The shared secret value is $k = x_A = x_B$.

3.9 AES Algorithm

The Advanced Encryption Standard (AES) [49, 35] is the new information protection standard defined by the US National Institute for Security Technologies (NIST) to replace the previous Data Encryption Standard (DES) to protect certain levels of federal information and communications. In 1997, NIST called for a new AES algorithm. The Rijndael algorithm was selected as the finalist and published in 2001.

AES performs four operations on a 128-bit block of data for a certain number of repetitions. All the intermediate results of the 128-bit block as well as the input

and the output block are called states. The most intuitive way to understand AES operation is to picture each state as a 4×4 matrix of bytes which are filled in the matrix column by column from the most significant byte (indexed as 0) to the least significant byte (indexed as F). At each stage of the transformation between plaintext and ciphertext, the block of data is transformed from its current state to the next new state, depending on which operation is used. The four operations are SubBytes, ShiftRows, MixColumns and AddRoundKey.

The SubBytes operation is a nonlinear substitution that performs on each byte of the state using a substitution table (S-box) which contains a permutation of all possible 256 8-bit values. The inverse of this operation, InvSubBytes, consists of applying the inverse of the affine transformation followed by taking the same multiplicative inverse in $GF(2^8)$.

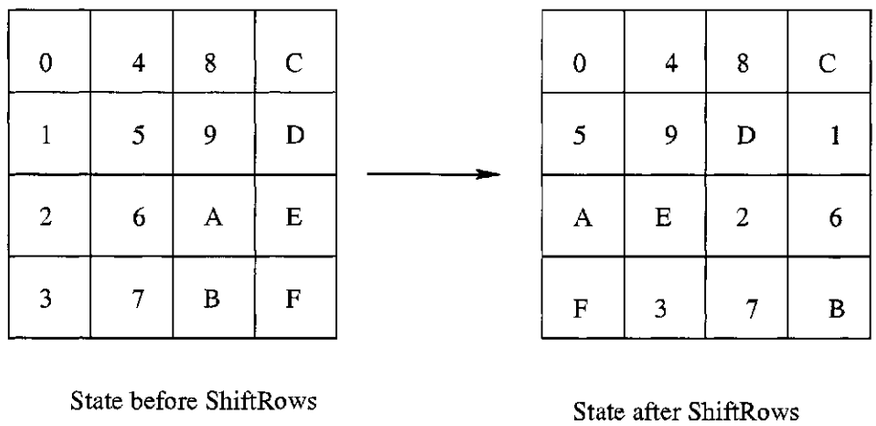

Figure 3.3: ShiftRows operation for encryption in AES algorithm

The ShiftRows operation (Fig. 3.3) is the cyclic shifting of each row of the state to the left over different numbers of bytes (offsets) on encryption, while the InvShiftRows shifts to the right on decryption.

The MixColumns operation treats each column of the state as a four-term polynomial over $GF(2^8)$ and transforms each column to a new one by multiplying it with a constant polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ modulo

$x^4 + 1$. The inverse MixColumns operation is a multiplication of each column with $b(x) = a^{-1}(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}$ modulo $x^4 + 1$. The transformation can also be written in the following matrix multiplication given a 32-bit input word $w = w_3w_2w_1w_0$ where each w_i has eight bits.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} w_3 \\ w_2 \\ w_1 \\ w_0 \end{bmatrix} = \begin{bmatrix} w'_3 \\ w'_2 \\ w'_1 \\ w'_0 \end{bmatrix} \tag{3.18}$$

The AddRoundKey operation is a simple logical XOR of the current state with a round key that is generated by the key expansion. The XOR operation is its own inverse.

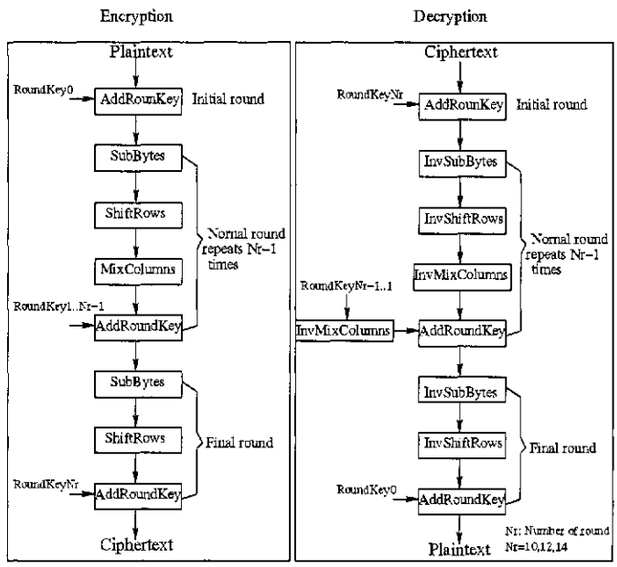

Figure 3.4: AES algorithm flow for encryption/decryption

In the key expansion algorithm, The initial Nk-word key corresponds to the cipher key and all subsequent Nk-word keys are derived recursively from their respective predecessors [49, 35]. Nk is the number of 32-bit words comprising the cipher key, which can be 4, 6 or 8. The same serial Nk-word keys are used in reversed order for decryption and all these keys can be derived from the last round of

Nk-word key using the inverse operations.

3.10 RC5 Algorithm

RC5 [49, 42, 17] is a fast block cipher designed by Ronald Rivest in 1994. The easy implementation and the security of heavily using data-dependent rotations and the mixture of different operations make it widely adopted in the area requiring high level strength for bulk encryption, such as wireless communications. A particular RC5 is exactly designated as RC5-$w/r/b$, where the variable parameters w, r, b denote the word size (in bits), the number of rounds and the length of secret key (in bytes), respectively. The allowable values of w are 16, 32 and 64; the allowable values of r and b range from 0 to 255. RC5-32/12/16 is commonly chosen.

There are three routines in RC5: key expansion, encryption, and decryption. These routines use three primitive operations (and their inverses): words addition modulo 2^w (w is the word size parameter), bitwise XOR, and data-dependent left rotation of x by y denoted by $x <<< y$. Note that only the $log_2(w)$ low-order bits of y affect this rotation. In the key-expansion routine, the user-provided secret key is expanded to fill a key table whose size depends on the number of rounds. The key table is then used in both encryption and decryption. The decryption follows the same scheme as encryption except that it requires words subtraction and rotation to the right. The description of the encryption algorithm is given in the following [42].

Input: Plaintext $\{A, B\}$, secret key array $(S[0], \ldots, S[2r], S[2r+1])$

Output: Ciphertext $\{A, B\}$

$A = A + S[0]$

$B = B + S[1]$

for $i = 1$ to r do

$A = ((A \ominus B) <<< B) + S[2i]$

$B = ((B \oplus A) <<< A) + S[2i+1]$

Algorithm 1: RC5 encryption algorithm

RC5 subkey generation is quite complex which generates a subkey array S of $t = 2r + 2$ words from b-byte secret key K. This includes three algorithm steps. First, the secret key array $K[0, \ldots, b-1]$ in byte is copied into an array $L[0, \ldots, c-1]$ in length of $c = \lceil b/u \rceil$ words where $u = w/8$ is the number of bytes/word. Second, array S is initialized using an arithmetic progress modulo 2^w determined by the predefined magic constants P_w and Q_w. At last, a mix in the secret key in three passes over the arrays S and L is performed as follows [42].

$i = j = X = Y = 0$

Do $3* \ max\{2r + 2, c\}$ times:

$X = S[i] = (S[i] + X + Y) <<< 3$

$i = (i + 1) \ (mod \ t)$

$Y = L[j] = (L[j] + X + Y) <<< (X + Y)$

$j = (j + 1) \ (mod \ c)$

3.11 Comparison between Public-key and Secret-key Cryptosystems

In this section, the advantages and disadvantages between public-key and secret-key cryptosystems are summarized. The advantages of public-key cryptosystem over secret-key cryptosystem are as follows:

1. Public-key cryptography has increased security and convenience since private keys never need to be transmitted or revealed to anyone. In a secret-key cryptosystem, by contrast, the secret keys must be transmitted.

2. Public-key cryptosystems can provide digital signatures that cannot be repudiated.

The disadvantages of public-key cryptosystem are listed as follows:

1. Many secret-key encryption algorithms are significantly faster than any currently available public-key encryption algorithm.

2. Public-key cryptography may be vulnerable to impersonation, even if users' private keys are not available.

In general, public-key cryptography is best suited for an open multi-user environment. It is not meant to replace secret-key cryptography, but rather to supplement it and to make it more secure. For efficiency reasons, a hybrid cryptosystem is used in practice; a key is exchanged using a public-key cipher, and the rest of the communication is encrypted using a secret-key algorithm.

3.12 Summary

In this chapter, an overview of public-key and secret-key cryptography is presented. The public-key Elliptic Curve Cryptography (ECC) and secret-key algorithms of

AES and RC5 are discussed. These three algorithms are implemented in the proposed crypto-processor in this thesis. In the final section, comparisons between public-key and secret-key cryptography are given.

Chapter 4

The Cryptographic Processor Architecture

In this chapter, the algorithms used in the ECC are introduced, and the architecture of the processor with corresponding instruction set is described. The performance evaluation and comparison are given finally.

The implementation of ECC is more complicated than secret-key cryptosystems such as AES and RC5. So a majority of the work in this thesis focuses on the ECC design. In ECC, the computation of scalar multiplication kP involves three different levels:

- Selection of scalar multiplication algorithms. These algorithms include the double-and-add method using binary representation of k, addition-subtraction method using nonadjacent form of k, and Montgomery scalar multiplication. Montgomery scalar multiplication is a fast algorithm only for $GF(2^n)$.

- Elliptic arithmetics in different coordinate representations. These coordinates include affine coordinates, projective coordinates, Jacobian coordinates used in both $GF(p)$ and $GF(2^n)$ and López-Dahab [28] projective coordinates used only in $GF(2^n)$, etc. Different coordinate representations lead to different formulae for point addition and point doubling.

- Field arithmetics. These include the basis selection, multiplier and squaring design, etc. In this thesis, the dual-field multipliers based on Montgomery multiplication are designed, which can perform multiplications both in $GF(p)$ and $GF(2^n)$ using polynomial bases. Furthermore, this multiplier design has been extended to implement four independent multiplications over $GF(2^8)$ used in the secret-key cryptosystems such as AES.

In this thesis, algorithm combinations between selection of fast scalar multiplication algorithms and selection of coordinate representations are made in order to arrive at the best solution. Due to the different characteristics of $GF(p)$ and $GF(2^n)$, the combinations are chosen separately. For $GF(p)$, an addition-subtraction method using the NonAdjacent Form (NAF) of k in Jacobian projectives is chosen. For $GF(2^n)$, Montgomery scalar multiplication algorithm in projective coordinates is selected. In the finite field arithmetic level, the multiplication over $GF(p)$ and $GF(2^n)$ are unified using Montgomery multiplication algorithm.

In the following first two sections, the elliptic arithmetic over $GF(p)$ of point addition and point doubling in modified Jacobian coordinates, and scalar multiplication using NAF are introduced. The Montgomery scalar multiplication algorithm over $GF(2^n)$ in projective coordinates is described in the third section. Due to simple formulae for point addition and point doubling, the two parts are described in one section. In Section 4.4, the details of Montgomery multiplication algorithm over both $GF(p)$ and $GF(2^n)$ are introduced. In Section 4.5, the overall architecture of the processor is proposed and the details of important common components are described. Finally, the instruction set is given and the corresponding programs for AES, RC5, and multiplication in $GF(p)$ are listed respectively. In Section 4.6 and Section 4.7, the performance evaluation and comparison are given.

Coordinate Representation	Addition	Doubling
Projective	12 $\mathcal{M}$+2 $\mathcal{S}$	7$\mathcal{M}$+5$\mathcal{S}$
Jacobian	12 $\mathcal{M}$+4 $\mathcal{S}$	4$\mathcal{M}$+6$\mathcal{S}$
Chudnovshky Jacobian	11 $\mathcal{M}$+3 $\mathcal{S}$	5$\mathcal{M}$+6$\mathcal{S}$
Modified Jacobian	13 $\mathcal{M}$+6 $\mathcal{S}$	4$\mathcal{M}$+4$\mathcal{S}$

Table 4.1: Projective coordinate representations over $GF(p)$

4.1 Coordinate Representation of Elliptic Curves over $GF(p)$

Since inversions are more expensive relative to multiplications, it is more efficient to represent points in projective coordinates. The inversion operation is traded for more multiplications and other less expensive finite field operations.

Several projective themes in $GF(p)$ are described in [7]. Table 4.1 compares the themes for addition and doubling with respect to the number of multiplications $\mathcal{M}$ and squarings $\mathcal{S}$ in the underlying finite field. The inexpensive field addition operation is omitted. In this thesis, the modified Jacobian projective coordinates are chosen due to their fastest doubling operation. They are represented internally as the quadruple $(X,\ Y,\ Z,\ aZ^4)$. The formulae of addition and doubling in the modified Jacobian projectives are given as follows [7].

Let $P = (X_1,\ Y_1,\ Z_1,\ aZ_1^4)$, $Q = (X_2,\ Y_2,\ Z_2,\ aZ_2^4)$, and $R = P + Q = (X_3,\ Y_3,\ Z_3,\ aZ_3^4)$. The addition formulae of $R = P + Q$ $(P \neq \pm Q)$ are the following:

$$
\begin{aligned}
&U_1 = X_1 \cdot Z_2^2, \quad S_2 = Y_2 \cdot Z_1^3, && X_3 = -H^3 - 2U_1 \cdot H^2 + r^2,\\
&U_2 = X_2 \cdot Z_1^2, \quad H = U_2 - U_1, && Y_3 = -S_1 \cdot H^3 + r(U_1 \cdot H^2 - X_3),\\
&S_1 = Y_1 \cdot Z_2^3, \quad r = S_2 - S_1, && Z_3 = Z_1 \cdot Z_2 \cdot H,\ aZ_3^4 = aZ_3^4.
\end{aligned}
$$

The doubling formulae of $R = 2P$ are the following:

$$S = 4X_1 \cdot Y_1^2, \quad X_3 = -2S + M^2,$$
$$U = 8Y_1^4, \qquad Y_3 = M \cdot (S - X_3) - U,$$
$$M = 3X_1^2 + (aZ_1^4), \quad Z_3 = 2Y_1 \cdot Z_1, \; aZ_3^4 = 2U \cdot (aZ_1^4).$$

4.2 Elliptic Scalar Multiplication over $GF(p)$

There are several methods known to compute kP. The basic method is the binary double-and-add method [30, 49] which requires k doublings and $k/2$ addition on average. The addition-subtraction method requires only $k/3$ additions on average with the same number of doubling [48]. It is based on NonAdjacent Form (NAF or sparse signed-digit representation) of the coefficient k where the redundant binary representation using {-1, 0, 1} is allowed. It is known that every integer has a unique NAF with the property that no two consecutive coefficients are nonzero. Also, the NAF has the minimum number of nonzero coefficients among all signed-digit representations, $k/3$ on average. Here gives an example of NAF:

$$NAF(29) = (1, 0, 0, -1, 0, 1), and\ 29 = 32 - 4 + 1. \tag{4.1}$$

The NAF of an integer is at most one bit longer than its binary expansion. The addition-subtraction method requires bit conversion left-to-right. An optimal NAF algorithm converted from the most significant bit is listed in Algorithm 2 [16].

Input: $(e_t, e_{t-1}, \ldots, e_1, e_0)$

Output: $(d_t, d_{t-1}, \ldots, d_0)$

$b_t \leftarrow 0$; $e_t \leftarrow 0$; $e_{-1} \leftarrow 0$; $e_{-2} \leftarrow 0$;

for *i=t down to 0* **do**

$b_{i-1} \leftarrow \lfloor (b_i + e_{i-1} + e_{i-2})/2 \rfloor$;

$d_i \leftarrow e_i + b_{i-1} - 2b_i$;

end

Algorithm 2: Left-to-right NAF

This algorithm can also be expressed as in Table 4.2 (X stands for don't care) [16]. Based on this table, a simple hardware speeding-up of NAF is proposed in [16]. d_i ($d_i \in \{0, 1, \bar{1}\}$) can be encoded with two one-bit variables $\{d_H, d_L\}$. Let $\{0 \rightarrow (X,0)_2$; $1 \rightarrow (0,1)_2$; -1 $\rightarrow (1,1)_2\}$, and after simple logic reduction, b_{i-1}, d_H, and d_L can be expressed as follows.

$$\begin{cases} b_{i-1} = \overline{b_i} \cdot r_{i-1} \cdot r_{i-2} + b_i \cdot r_{i-1} + b_i \cdot r_{i-2} \\ d_H = b_i \\ d_L = \overline{b_i} \cdot r_{i-1} \cdot r_{i-2} + \overline{b_i} \cdot r_i + b_i \cdot \overline{r_i} + b_i \cdot \overline{r_{i-1}} \cdot \overline{r_{i-2}} \end{cases}$$

The logic diagram based on this equation is illustrated in Fig. 4.1. Initially, the shift-left registers $\{r_i, r_{i-1}, r_{i-2}\}$ are loaded with $\{0, r_{m-1}, r_{m-2}\}$ and the latch is reset to logic "0". At the end of each iteration, the output d_H and d_L are used to decide the value of the encoded d_i.

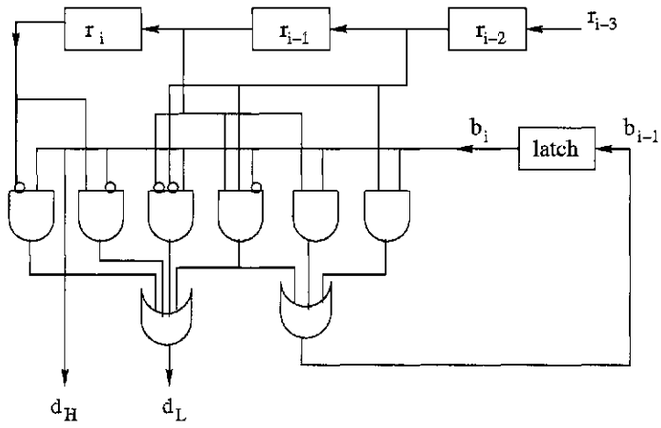

Figure 4.1: Hardware speeding-up of NAF

Given the NAF of $n = \sum_{i=0}^{t} d_i 2^i$, the elliptic scalar multiplication $Q = nP$ is performed as follows [28, 48].

b_i	r_i	r_{i-1}	r_{i-2}	b_{i-1}	d_i
0	0	0	X	0	0
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	X	0	1
1	0	1	X	1	$\bar{1}$
1	1	0	0	0	$\bar{1}$
1	1	0	1	1	0
1	1	1	X	1	0

Table 4.2: Left-to-right NAF

Input: An integer $k = (d_t, d_{t-1}, \ldots, d_0)$, and a point $P \in E(GF(q))$

Output: $Q = kP \in E(GF(q))$

Set $Q \leftarrow O$;

for *i=t-1 down to 0* **do**
  Set $Q \leftarrow 2Q$;

  if $e_i = 1$ **then**
    Set $Q \leftarrow Q + P$;

  end

  if $e_i = -1$ **then**
    Set $Q \leftarrow Q - P$;

  end

end

Algorithm 3: Addition-subtraction method using NAF

4.3 Elliptic Scalar Multiplication over $GF(2^n)$

A new scalar multiplication algorithm that was first proposed by Montgomery [33] and then modified by López and Dahab [27] proves to be the best algorithm for elliptic scalar multiplication in $GF(2^n)$. It is based on the binary expansion of k and the observation that the x-coordinate of the sum of two points whose difference is known can be computed in terms of the x-coordinates of the involved points. It

can be expressed in Algorithm 4.

Input: An integer $k > 0$, and a point $P \in E(GF(2^n))$

Output: $Q = kP \in E(GF(2^n))$

Set $k \leftarrow (k_{t-1}, k_{t-2}, \ldots, k_1, k_0)_2$;

Set $P_1 \leftarrow P$, $P_2 \leftarrow 2P$;

for *i=t-2 down to 0* **do**

  if $k_i = 1$ **then**

    Set $P_1 \leftarrow P_1 + P_2$, $P_2 \leftarrow 2P_2$;

  else

    Set $P_2 \leftarrow P_1 + P_2$, $P_1 \leftarrow 2P_1$;

  end

end

$Q = P_1$;

Algorithm 4: The basic Montgomery scalar multiplication

As in $GF(p)$, projective coordinates (X, Y, Z) are represented in order to avoid expensive inversions by x=X/Z and y=Y/Z from affine coordinates (x, y). The Montgomery scalar multiplication in projective coordinates is listed in Algorithm 5.

Input: An integer $k \geq 0$, and a point $P \in E(GF(2^n))$

Output: $Q = kP \in E(GF(2^n))$

if *k=0 or x=0* **then**
output (0,0) and stop ;

end

Set $k \leftarrow (k_{t-1}, k_{t-2}, \ldots, k_1, k_0)$;

Set $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x_4 + b$, $Z_2 \leftarrow x^2$;

for *i=t-2 down to 0* **do**

if $k_i = 1$ **then**

$Z_3 \leftarrow (x_1 \cdot Z_2 + X_2 \cdot Z_1)^2$, $X_3 \leftarrow x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1)$;

$Z_4 \leftarrow Z_2^2 \cdot X_2^2$, $X_4 \leftarrow X_2^4 + b \cdot Z_2^4$;

$Z_1 \leftarrow Z_3$, $X_1 \leftarrow X_3$, $Z_2 \leftarrow Z_4$, $X_2 \leftarrow X_4$;

else

$Z_3 \leftarrow (x_1 \cdot Z_2 + X_2 \cdot Z_1)^2$, $X_3 \leftarrow x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1)$;

$Z_4 \leftarrow Z_1^2 \cdot X_1^2$, $X_4 \leftarrow X_1^4 + b \cdot Z_1^4$;

$Z_1 \leftarrow Z_4$, $X_1 \leftarrow X_4$, $Z_2 \leftarrow Z_3$, $X_2 \leftarrow X_3$;

end

end

return(The affine coordinates converted from projective coordinates of Q) ;

Algorithm 5: The Montgomery scalar multiplication algorithm using projective coordinates

The y-coordinates of kP can be computed from the x-coordinates of points involved in the last iteration in Algorithm 5, which is shown as follows.

$$if\ Z_1 = 0\ then\ output\ (0,0)$$

$$if\ Z_2 = 0\ then\ output\ (x, x+y)$$

$$x_k = \frac{X_1}{Z_1}$$

$$y_k = (\frac{X_1}{Z_1} + x) \cdot \frac{(\frac{X_1}{Z_1} + x)(\frac{X_2}{Z_2} + x) + x^2 + y}{x} + y$$

Using projective coordinates, Montgomery scalar multiplication requires $6\lfloor log_2(k) \rfloor +$

9 multiplications, $5\lfloor log_2(k) \rfloor + 3$ squarings, $3\lfloor log_2(k) \rfloor + 7$ additions and 1 multiplicative inverse.

4.4 Montgomery Multiplication

Montgomery multiplication was first proposed by Montgomery in 1985 [32], as an efficient method for doing modular multiplication in prime fields $GF(p)$. This method was extended to the binary finite field $GF(2^m)$ by Koç and Acar in [21]. Several papers [12, 43, 9] have explored the similarity of hardware design between Montgomery multiplications in $GF(p)$ and $GF(2^m)$ and proposed unified or dual-field multipliers based on the Montgomery multiplication algorithm.

4.4.1 Montgomery Multiplication over $GF(p)$

Given two integers A and B, and the prime number p, the Montgomery multiplication algorithm in $GF(p)$ computes

$$C = MonMul(A, B) = A \cdot B \cdot R^{-1} \ (mod\ p) \tag{4.2}$$

where $R = 2^m$ and $0 \leq A,\ B < p < R$ and p is an m-bit number. Before using Montgomery multiplication, the field element should be transformed into Montgomery domain by using the formula $\bar{A} = A \cdot R \ (mod\ p)$ for $A \in GF(p)$. For any two elements in the Montgomery domain $\bar{A}$ and $\bar{B}$, the result using equation 4.2 is still in the Montgomery domain.

$$\overline{C} = \overline{A} \cdot \overline{B} \cdot R^{-1} \ (mod\ p) = (A \cdot R) \cdot (B \cdot R) \cdot R^{-1} = C \cdot R \ (mod\ p) \tag{4.3}$$

The transformation operations between the two domains can be performed using the MonMul function as follows.

$$\bar{A} = MonMul(A, R^2) = A \cdot R^2 \cdot R^{-1} = A \cdot R \ (mod\ p) \tag{4.4}$$

$$C = MonMul(\bar{C}, 1) = C \cdot R \cdot R^{-1} = C \ (mod\ p) \tag{4.5}$$

The key idea of the Montgomery multiplication algorithm is to add an appropriate multiple of p to make the lowest m bits of $A \cdot B$ equal to 0. The addition operation does not influence the equation 4.2 due to its modulo p arithmetic. In the following, we assume that the operation numbers have already been transformed to Montgomery domain and the overlines are omitted. For Montgomery reduction algorithm, an additional value N' is needed which satisfies the property $R \cdot R^{-1} - N \cdot N' = 1$. The integer R^{-1} and N' can be computed by the extended Euclidean algorithm. The Montgomery multiplication algorithm is given below:

Input: A, B, p $(0 \leq A, B \leq p)$
Output: $C = AB2^{-m} mod\ p$
1. $T = A \cdot B$;
2. $M = T \cdot N'\ mod\ 2^m$;
3. $C = (T + M \cdot p)/2^m$;
4. if $C \geq p$ then $C = C - p$;

For multiple-precision multiplication, the operands of m-bit length should be divided into s blocks of words $(m = s \times w)$ in the form as $A = (a_{s-1}, \ldots, a_1, a_0)$, where w is the word size. This can also be expressed as follows:

$$A = a_{s-1}2^{w(s-1)} + \ldots + a_1 2^w + a_0 \tag{4.6}$$

If one operand is in multiple-precision, while other operands are still in full-precision, the Montgomery multiplication algorithm is shown as Algorithm 6. In this algorithm, the division of 2^w is trivial because the least significant word of the dividend is zero. This can be proved by the following argument:

$$\begin{aligned} & c_0 + a_i b_0 + t_i p\ mod\ 2^w \\ &= c_0 + a_i b_0 + (c_0 + a_i b_0)(-p^{-1})p\ mod\ 2^w \\ &= c_0 + a_i b_0 - (c_0 + a_i b_0)\ mod\ 2^w \\ &= 0 \end{aligned}$$

Input: $A = (a_{s-1}, \ldots, a_1, a_0)$, B, p, $q = -p^{-1}\ mod\ 2^w = -p_0^{-1}\ mod\ 2^w$
Output: $C = AB2^{-m}$ mod p
$C = 0$;
for *(i=0 to s-1)* **do**
  $t_i = (c_0 + a_i b_0)q \mod 2^w$;
  $C = (C + a_i B + t_i p)/2^w$;
end
if $(C \geq p)$ **then**
  C=C-p ;
end

Algorithm 6: Word level full precision Montgomery multiplication

If all the operands are in multiple-precision form, the Montgomery multiplication is listed in Algorithm 7 based on the Coarsely Integrated Operand Scanning (CIOS) Method [22].

Input: $A = (a_{s-1}, \ldots, a_1, a_0)$, $B = (b_{s-1}, \ldots, b_1, b_0)$, $p = (p_{s-1}, \ldots, p_1, p_0)$,
$q = -p_0^{-1}\ mod\ 2^w$

Output: $t = AB2^{-m} \bmod p$

for *i=0 to s-1* **do**
  $C = 0$;
  for *j=0 to s-1* **do**
    $(C, S) = t[j] + a[j] \times b[i] + C$;
    $t[j] = S$;
  end
  $(C, S) = t[s] + C$; $t[s] = S$; ;
  $m = t[0] \times q \bmod 2^w$;
  for *j=0 to s-1* **do**
    $(C, S) = t[j] + m \times p[j] + C$;
    if $j \neq 0$ **then**
      $t[j-1] = S$;
    end
  end
  $(C, S) = t[s] + C$;
  $t[s-1] = S$; $t[s] = C$;
  if $t > p$ **then**
    $t = t - p$;
  end
end

Algorithm 7: Word level word level Montgomery multiplication over $GF(p)$

Input: $A = (a_{s-1}, \ldots, a_1, a_0)$, $p = (p_{s-1}, \ldots, p_1, p_0)$, $q = -p_0^{-1} \bmod 2^w$

Output: $t = AB2^{-m} \bmod p$

for *i=0 to s-1* **do**
  $C = 0$;
  $(C, S) = t[i] + a[i] \times a[i] + C$; $t[i] = S$;
  for *j=i+1 to s-1* **do**
    $(C, S) = t[j] + 2 \times a[j] \times a[i] + C$; $t[j] = S$;
  end
  $(C, S) = t[s] + C$;
  $m = t[0] \times q \bmod 2^w$;
  for *j=0 to s-1* **do**
    $(C, S) = t[j] + m \times p[j] + C$;
    if $j \neq 0$ **then**
      $t[j-1] = S$;
    end
  end
  $(C, S) = t[s] + C$;
  $t[s-1] = S$; $t[s] = C$;
  if $t > p$ **then**
    $t = t - p$;
  end
end

Algorithm 8: Montgomery squaring over $GF(p)$

4.4.2 Montgomery Multiplication over $GF(2^n)$

The Montgomery multiplication in $GF(2^n)$ using the polynomial basis is very similar to the one in $GF(p)$ [22]. The Montgomery multiplication of $A(x)$ and $B(x)$ with product $C(x)$ is given as

$$C(x) = A(x) \cdot B(x) \cdot R(x)^{-1} \ (mod\ p(x)) \tag{4.7}$$

where $R(x) = x^m$ instead of $R = 2^m$ as compared with equation 4.2. Furthermore, the representation of the elements of $GF(p)$ and $GF(2^n)$ are the same. An example in [44] is given as follows.

The elements of $GF(7)$ for $p = 7$ and the elements of $GF(2^3)$ for $p(x) = x^3 + x + 1$ are represented as $GF(7) = \{000, 001, 010, 011, 100, 101, 110\}$ and $GF(2^3) = \{000, 001, 010, 011, 100, 101, 110, 111\}$, respectively.

Similarly, the domain transformations are required: before Montgomery multiplication, the operands should be transformed into Montgomery domain and the computing result should be transformed back. The transformations are accomplished as follows:

$$\bar{A} = MolMul(A, R^2) = A(x) \cdot R^2(x) \cdot R^{-1}(x) = A(x) \cdot B(x) \ (mod\ p(x))$$
$$\bar{B} = MolMul(B, R^2) = B(x) \cdot R^2(x) \cdot R^{-1}(x) = A(x) \cdot B(x) \ (mod\ p(x))$$
$$\bar{C} = MolMul(\bar{C}, 1) = C(x) \cdot R(x) \cdot R^{-1}(x) = C(x) \ (mod\ p(x))$$

However, the operations in $GF(2^m)$ are much simpler without considering carry propagations.

The multiple-precision Montgomery multiplication algorithm for multiplication and squaring can be written based on the Coarsely Integrated Operand Scanning (CIOS) Method [22] as follows:

Input: $A(x) = (a_{s-1}, \cdots, a_1, a_0)$, $B(x) = (b_{s-1}, \cdots, b_1, b_0)$,
$p(x) = (p_{s-1}, \cdots, p_1, p_0)$, $q(x) = p(x)^{-1} \bmod x^w = p_0(x)^{-1} \bmod x^w$

Output: $t(x) = A(x)B(x) \bmod p(x)$

for *i=0 to s-1* **do**
  $C = 0$;
  for *j=0 to s-1* **do**
    $(C, S) = t[j] + a[j] \times b[i] + C$; $t[j] = S$;
  end
  $(C, S) = t[s] + C$;
  $m = t[0] \times n'[0] \bmod x^w$;
  for *j=0 to s-1* **do**
    $(C, S) = t[j] + m \times n[j] + C$;
    if $j \neq 0$ **then**
      $t[j-1] = S$;
    end
  end
  $(C, S) = t[s] + C$;
  $t[s-1] = S$; $t[s] = C$;
end

Algorithm 9: Montgomery multiplication over $GF(2^n)$

Input: $A(x) = a_{s-1}, \ldots, a_1, a_0)$, $p(x) = (p_{s-1}, \ldots, p_1, p_0)$, $q = p(x)^{-1} \bmod x^w = p_0(x)^{-1} \bmod x^w$

Output: $t(x) = A(x)B(x) \bmod p(x)$

for *i=0 to s-1* **do**
  $C = 0$;
  $(C, S) = t[i] + a[i] \times a[i] + C$;
  $t[i] = S$;
  for *j=i+1 to s-1* **do**
    $(C, S) = t[j] + 2 \times a[j] \times a[i] + C$;
    $t[j] = S$;
  end
  $(C, S) = t[s] + C$;
  $m = t[0] \times p'[0] \bmod x^w$;
  for *j=0 to s-1* **do**
    $(C, S) = t[j] + m \times n[j] + C$;
    if $j \neq 0$ **then**
      $t[j-1] = S$;
    end
  end
  $(C, S) = t[s] + C$;
  $t[s-1] = S$; $t[s] = C$;
end

Algorithm 10: Montgomery squaring over $GF(2^n)$

4.5 Architecture of the Hybrid Processor

The common arithmetic components in the three algorithms: ECC, AES, and RC5 are extracted as shown in Table 4.3. As the most important arithmetic component, a novel multiplier of 32 by 32 bits is proposed to perform multiplication over $GF(p)$ and $GF(2^n)$ in the ECC, and four independent multiplications over $GF(2^8)$ in the

components	AES	RC5	ECC over $GF(p)$	ECC over $GF(2^n)$
multiplication	√		√	√
barrel shifter		√	√	√
adder/subtracter		√	√	
XOR	√	√		√

Table 4.3: The core components in different cryptographic algorithms

MixColumns operation of AES. An adder/subtracter is used for the modular addition/subtraction in RC5 and over $GF(p)$ in ECC. In addition, a barrel shifter is embedded for data-dependent rotation in RC5. This barrel shifter can also be used for bit-shifts of parameter k in scalar multiplication kP over $GF(p)$ and $GF(2^n)$.

In the following, the data path of the processor is described and followed by the details of core arithmetic components and the instruction set.

4.5.1 Data Path

The main data path is 32-bit wide as shown in Fig. 4.2. The long operands of the ECC algorithm need to be broken up into multiple smaller words, and the arithmetic operations such as addition, subtraction, and multiplication, are implemented as multiple-precision operations [21]. The 32-bit data path is chosen considering that the word length in RC5 is 32-bit, and AES can also be processed in 32-bit processors. The two dual-port memories, which both have the capacity of 256×32 bits, are used not only for passing parameters between the cryptographic processor and the host, but also for operands. The parameters for ECC or the expanded subkey array for secret-key cryptography are loaded into the dual-port RAMs from I/O ports. The operands of many instructions come from the dual-port RAMs. The formula of multiplication-accumulator operation is shown as $(CA, SUM) = src0 \times src1 + src2 + CA$, where CA is the carry vector, SUM is the sum vector, and $src0$, $src1$,

src2 are the three source operands which are accessed directly from the two dual-port RAMs with the ports *douta1*, *doutb1* or *douta2*. The data in the dual-port RAMs can also be routed to register A through ports *douta1* or *doutb2*. The results of barrel shifter, XOR and addition/subtraction operations are stored to register A, whereas the results of multiplications for ECC are written back to the dual-port RAMs.

In order to accommodate the eight-bit operations used in secret-key cryptographic algorithms, an eight-bit data path is also provided as shown in Fig. 4.3. The register A comprises of four eight-bit registers, i.e. A0, A1, A2, and A3. The sufficient Block RAMs provided by Xilinx FPGA devices are utilized to store the constants of the S-box and inverse S-box lookup tables operation used in AES algorithm. The ROM of 512×8 bits is implemented by Block RAMs. The register file is composed of 32 eight-bit registers, of which four registers hold the indirect address for the dual-port RAMs and seven registers are used for the polynomial reduction for multiplication over $GF(2^8)$. The result of the multiplication over $GF(2^8)$ is transmitted into A passing through multipliers as a 32-bit operation (Fig. 4.2) or the four independent bytes are XORed and the result is then stored into one of A3, A2, A1, A0 as an eight-bit operation (Fig. 4.3). For the MixColumns operation in AES, each element in the right column of the matrix in Equation 3.18 needs eight-bit XOR operations between the results of four eight-bit multiplications. For example, $w_3' = (\{02\} \cdot w_3) \oplus (\{03\} \cdot w_2) \oplus (\{01\} \cdot w_1) \oplus (\{01\} \cdot w_0)$. This special operation for MixColumns is supported directly in the design.

4.5.2 Multiplier Design

Based on the multiple-precision multipliers for ECC over $GF(p)$ and $GF(2^n)$ [43, 9], a novel multiplier is proposed which is extended to perform not only the multiplications over $GF(p)$ and $GF(2^n)$, but also four parallel eight-bit multiplications over $GF(2^8)$ that are used in the secret-key cryptosystems such as AES. The block

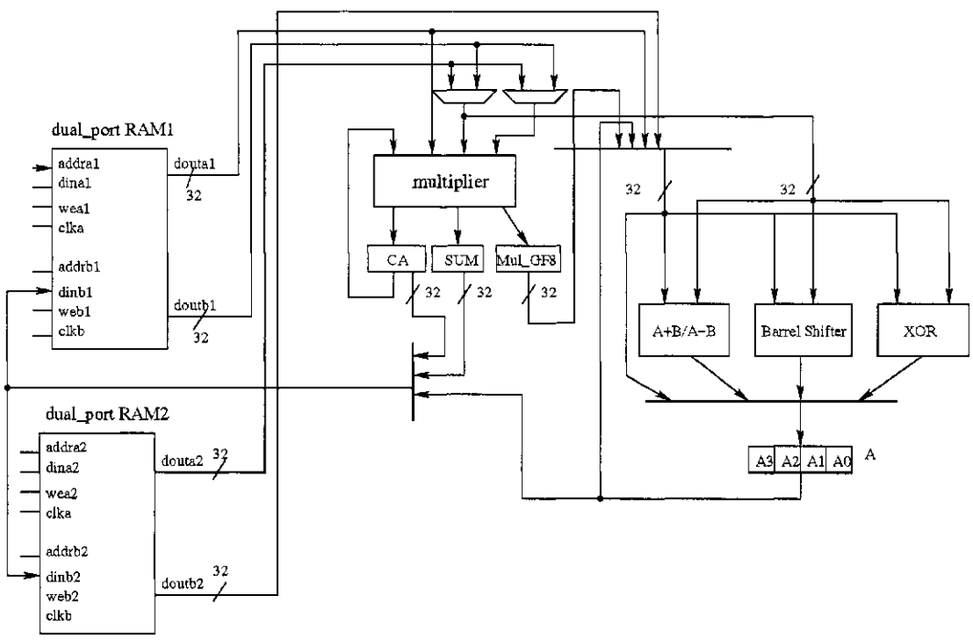

Figure 4.2: The data path of the architecture: 32-bit part

diagram of the multiplier is illustrated in Fig. 4.4. It can perform operation $(CA, SUM) = src0 \times src1 + src2 + CA$ for ECC over $GF(p)$ and $GF(2^n)$, and operation $Mul\_GF8 = src0 \times src1$ with 32-bit operands. The presented multiplier uses parallel tree multiplier which is based on the integer multiplier design. Multiplications presented in this thesis consist of three main stages [51]:

- Partial product generation. It uses 32×32 AND gates to generate the partial product bits.
- Partial product reduction. It uses a Carry-Save Adder (CSA) tree to reduce the partial products in a redundant carry/sum representation.
- Final carry-propagate addition. It uses Carry-Propagate Adders (CPA) to add the sum and carry vectors and produces the final product.

Carry-Save Adders are implemented by Half Adders (HA) and Full Adders (FA). The formulae of the HA and FA used in the tree multiplier are shown in Equation

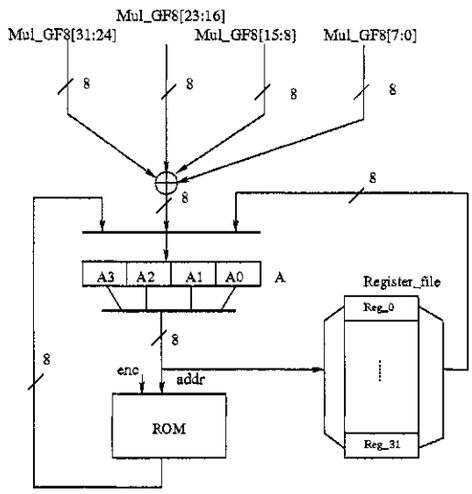

Figure 4.3: The data path of the architecture: eight-bit part

4.8 and 4.9. We can see that the sums are only XOR operation.

$$\begin{cases} s_k = a_k \oplus b_k \\ c_{k+1} = a_k \cdot b_k \end{cases} \tag{4.8}$$

and

$$\begin{cases} s_k = a_k \oplus b_k \oplus c_k \\ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \end{cases} \tag{4.9}$$

The multiplication over $GF(2^n)$ involves only two stages that are partial product generation and partial product reduction, since no carry-propagations are considered. The partial product reduction performs only XOR operations between the same digit position. From the equations for the HA and FA, it can be seen that the sum is exactly the XOR operation required by multiplication over $GF(2^n)$. So it is possible to combine the multiplication over $GF(2^n)$ with the one over $GF(p)$ in such a way that the sum bits s_k are added first and the carry bits c_{k+1} are added as late as possible in the partial product reduction. The multiplication over $GF(2^n)$ is just a special case of the one over $GF(p)$ without considering the carry-propagation. Finally, the product for $GF(2^n)$ is summed up with the reduced carry-bits in the final stages and the product for $GF(p)$ is obtained by the carry-lookahead adder.

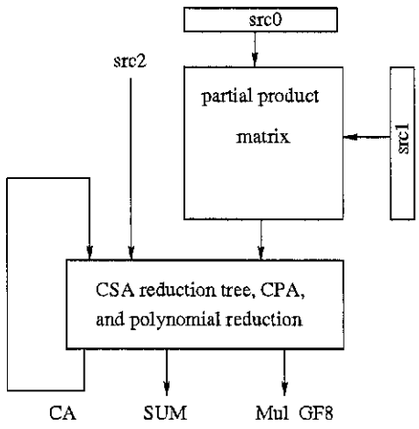

Figure 4.4: The block diagram of multiple-precision multifunction multiplier

The procedure is illustrated in Fig. 4.5.

In the secret-key cryptography, the multiplications over $GF(2^8)$ are required by the MixColumns operation in AES. The proposed multiplier is further extended to perform four independent multiplications over $GF(2^8)$. The 32-bit inputs $src0$ and $src1$ are referred to as the concatenation of four eight-bit elements over $GF(2^8)$, respectively. The four multiplications are $src0[31\ldots24]\times src1[31\ldots24]$, $src0[23\ldots16]\times src1[23\ldots16]$, $src0[15\ldots8]\times src1[15\ldots8]$, and $src0[7\ldots0]\times src1[7\ldots0]$. The dot diagram of partial product generation for the four eight-bit multiplications is illustrated as hollow dots in Fig. 4.6. The multiplication over $GF(2^8)$ can be viewed as a special case of multiplication over $GF(2^n)$. The four multiplications of eight-bit by eight-bit over $GF(2^8)$ are performed first. This result is used in the next XOR operations in the same digit positions in the computation of $GF(2^n)$. The results of the four 15-bit products over $GF(2^8)$ are ready after the third stage, and the product over $GF(2^n)$ is ready after the fifth stage. The four 15-bit results obtained from the partial product reduction need modulo the irreducible polynomial $f(x)$ to reduce to elements over $GF(2^8)$ as shown in Fig. 4.5. For the AES algorithm, this irreducible polynomial is $f(x) = x^8 + x^4 + x^3 + x + 1$. The reduction method is illustrated as follows. Suppose the product is $a(x) = a_{14}x^{14} + a_{13}x^{13} \ldots + a_8x^8 + a_7x^7 + \ldots + a_0x^0$,

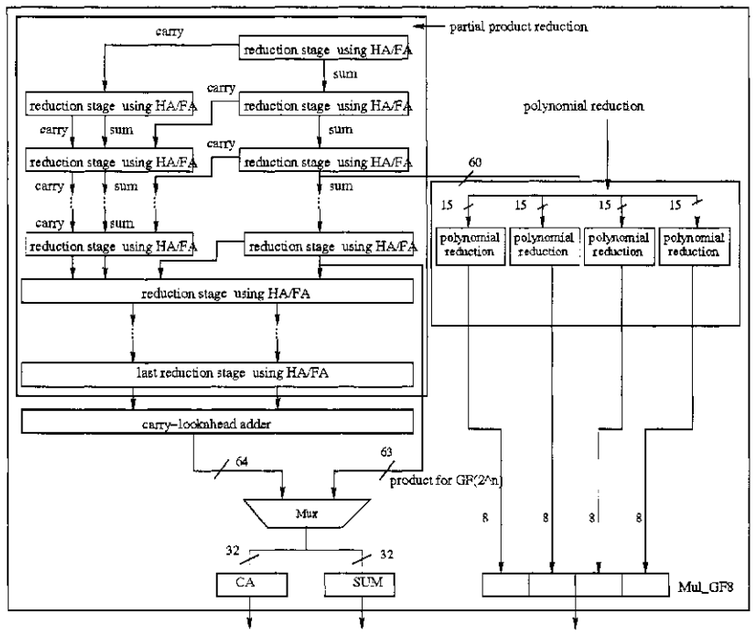

Figure 4.5: Partial product reduction, CPA and polynomial reduction

and let

$$b_1(x) = x^8 \ mod \ f(x) = x^4 + x^3 + x + 1$$

$$b_2(x) = x^9 \ mod \ f(x) = (x^4 + x^3 + x + 1)x = x^5 + x^4 + x^2 + x$$

$$b_3(x) = x^{10} \ mod \ f(x) = (x^4 + x^3 + x + 1)x^2 = x^6 + x^5 + x^3 + x^2$$

$$b_4(x) = x^{11} \ mod \ f(x) = (x^4 + x^3 + x + 1)x^3 = x^7 + x^6 + x^4 + x^3$$

$$b_5(x) = x^{12} \ mod \ f(x) = (x^4 + x^3 + x + 1)x^4 = x^7 + x^5 + x^3 + x + 1$$

$$b_6(x) = x^{13} \ mod \ f(x) = (x^4 + x^3 + x + 1)x^5 = x^5 + x^4 + x^3 + x$$

$$b_7(x) = x^{14} \ mod \ f(x) = (x^4 + x^3 + x + 1)x^6 = x^7 + x^4 + x^3 + x$$

Then $a(x)$ is reduced by $a(x)' = a(x) \ mod \ f(x) = \sum_{i=1}^{7} a_{7+i} \cdot b_i(x) + \sum_{i=0}^{7} a_i x^i$. Parameters b_1 to b_7 are kept in registers since they are constants once the irreducible polynomial is fixed. It is flexible in that the parameters of b_1 to b_7 can be changed according to different applications.

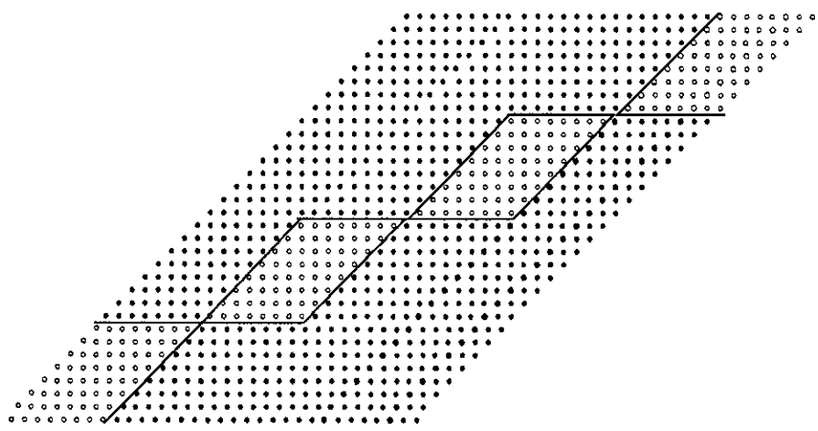

Figure 4.6: The dot diagram for partial product matrix of $32 - bit \times 32 - bit$ multiplication

The CSA utilizes tree topologies of adders so that the carry-out from one adder is not connected to the carry-in of the next one. The Reduced Area (RA) multiplier is used in the design instead of Wallace tree [43]. It can obtain the maximum reduction in the number of partial product bits in each reduction stage and minimize the total hardware area. Its reduction scheme differs from Wallace and Dadda's methods in that the maximum number of FAs is utilized as early as possible, and HAs are used only in two cases: (1) to reduce the number of bits in a column to the number of bits specified by Dadda sequence; (2) to reduce the rightmost column containing exactly two bits [51]. The RA multiplier is more flexible to be used in the partial product reductions for multiplications over $GF(p)$, $GF(2^n)$, and $GF(2^8)$. This design requires only nine reduction stages, the same as the 32-bit Wallace tree and Dadda tree multipliers that perform only multiplication over $GF(p)$. It means that the the implementation of multiplications over $GF(2^n)$ and $GF(2^8)$ with integer multiplication over $GF(p)$ does not produce additional delay.

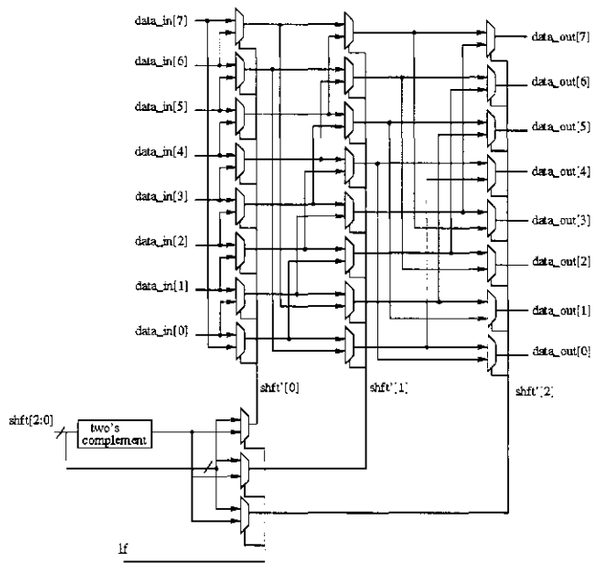

Figure 4.7: Example of an 8-bit barrel shifter

4.5.3 Barrel Shifter

The proposed multiplexer-based barrel shifter performs only circular shift operation. It is a combinational logic circuit design with 32-bit data inputs, 32-bit data outputs, and a five-bit control input to specify the amount of circular shift (0 to 31 bits), and a one-bit control signal to indicate the direction of circular shift (left or right). The circuit consists of five stages of 2:1 multiplexers, with one multiplexer per bit of the input data. The selection signals of all multiplexers in each stage are connected together. When shft[0]=1, the first stage of multiplexers performs a shift by one bit; when shft[0]=0, it does not perform any shift. Similarly, the second, third, fourth, and fifth stage perform a shift by 2, 4, 8, and 16 bits respectively. Due to the cascade, all shift amounts (0 to 31) can be performed. For right shift, the signal shft[4:0] is complemented, which means the shifter performs left shift with a complementary amount of bits used for right shift. An example of an eight-bit barrel shifter is shown in Fig. 4.7.

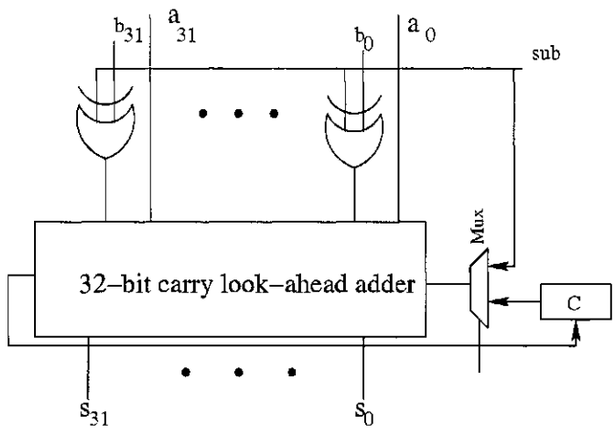

Figure 4.8: Adder/subtracter

4.5.4 Adder/Subtracter

The adder/subtracter illustrated in 4.8 is a 32-bit carry look-ahead adder with data inputs A and B, one output S, and a carry bit C. To perform subtraction, the adder is utilized to add the two's complement of the input B, i.e. A-B=A+(-B). The exclusive OR gates together with the signal sub are used to perform the two's complement when sub=1. For multiple-precision addition, the signal sub is set to "0"; it passes through the multiplexor in the first iteration and the carry bit C passes in the rest of iterations. Similarly, for multiple-precision subtraction, the signal sub is set to "1"; it passes through the multiplexor in the first iteration and C in the rest of iterations.

4.5.5 Instruction Set

Table 4.4 contains the complete instruction set. Instructions fall into three categories: arithmetic instructions, memory instructions, and control instructions.

The implementation of RC5 is quite simple, just following the Algorithm 1 in Chapter 3. Table 4.5 lists the source codes for RC5. The implementation of AES is not so straightforward. The S-box operations are implemented by 16 look-up table. The MixColumns operation is performed by 16 multiplication according to Equation

instruction	explanation	clock cycles
Mul_GFP *src*0, *src*1, *src*2, *dst*	multiplication over $GF(p)$ $dst = src0 \times src1 + src2 + C$	5
Mul_GF2N *src*0, *src*1, *src*2, *dst*	multiplication over $GF(2^n)$ $dst = src0 \times src1 \oplus src2 \oplus C$	5
Mul_GF8 *src*0, *src*1, *A*	four multiplications over $GF(2^8)$	4
MixColumns *src*0, *src*1, *dst*	multiplications over $GF(2^8)$ and the result is XORed between the four bytes	4
SQUR_GFP *src*0, *src*1, *dst*	squaring over $GF(p)$ $dst = src0 \times src0 + src1 + C$	5
SQUR_GF2N *src*0, *src*1, *dst*	squaring over $GF(2^n)$ $dst = src0 \times src0 \oplus src1 \oplus C$	5
SHFT	barrel shift	4
XOR	exclusive OR	4
ADD *src*0, *src*1, *dst*	addition	5
SUB *src*0, *src*1, *dst*	subtraction	5
LDA	load into A (32-bit)	4
	or load into A0, A1, A2, A3 (eight-bit)	3
STR	store constant or A0,A1,A2,A3	3
	or store A or CA to dual-port RAM	4
LKUP	lookup table	4
BRNZ bit, addr	branch to addr if test bit is set	3/4
BRLE #const, addr	branch to addr if index $\leq$ #const	3/4
CLR	clear the carry bit of adder/subtracter or clear the carry register of multiplier	3
NOP		3

Table 4.4: Instruction set

3.18. The ShiftRows operation spans throughout the entire 128-bit. The 16 bytes are first put into 16 eight-bit general registers, and read back in the order as shown in Fig. 3.3. The codes for one round of AES are shown in Table 4.15. Table 4.6 gives the source codes for Montgomery multiplication over $GF(p)$ which is based on the Algorithm 7 in Section 4.4.1.

	STR #0, Reg_index1
LE_0=A+S[0]	ADD *AR0, *AR2+, *BR0
RE_0=B+S[1]	ADD *AR1, *AR2+, *BR1
for i=1 to r do	J1:
$temp = LE_{i-1} \ominus RE_{i-1}$	XOR *BR0, *BR1, A
$temp = temp <<< RE_{i-1}$	SHFT A, *BR1, A
$LE_i = temp + S[2 \times i]$	ADD A, *AR2, *BR0
$temp = LE_i \ominus RE_{i-1}$	XOR A, *BR0, A
$temp = temp <<< LE_i$	SHFT A, *BR0, A
$RE_i = temp + S[2 \times i + 1]$	ADD A, *AR2+, *BR1
	BRLE #11, J1

Table 4.5: Codes for RC5

4.6 Performance Evaluation

The cryptographic processor was prototyped in a Xilinx Virtex-II xc2v3000-5bf957 FPGA using Verilog Hardware Description Language (Verilog HDL). It works at the maximum frequency of 60.15MHz.

Table 4.7 lists the resources used by different operation components. The resources contain 4-input look-up tables (LUTs), slices and Block RAMs. From the table, it can be seen that the multiplier occupies most of the hardware resources. The implementation of ECC relies heavily on the multiplication, so consuming more resources on the design of parallel multiplier is worthwhile. Block RAMs are utilized to implement the look-up tables for S-box operation in AES, and the two dual-port RAMs.

The performance of secret-key and public-key cryptosystems is illustrated in Table 4.8. When evaluating the time used for scalar multiplication kP in ECC over $GF(2^n)$, let $k \approx 2^n$; for kP over $GF(p)$, let $k \approx l$, where l is the order of the field $GF(p)$.

For(i=0 to s-1)
  C=0;
  For(j=0 to s-1)
  (C,S)=t[j]+a[j] × b[i]+C;
  t[j]=S;
  (C,S)=t[s]+C; t[s]=S;
  m=t[0] × q mod W;
  For(j=0 to s-1)
  (C,S)=t[j]+m × p[j]+C;
  If(j ≠ 0)
  t[j-1]=S;
  (C,S)=t[s]+C;
  t[s-1]=S;
  t[s]=C;
	STR #0, Reg_index1
	J1: CLR C
	J2:
	Mul_GFP *AR0+, *AR1, *AR2+
	BRLE s-1, J2
	Mul_GFP *AR0+, *AR1+, *AR2+
	Mul_GFP *AR0, *AR2, *AR1
	
	Mul_GFP *AR0+, *AR1, *AR2+
	STR #1, Reg_index2
	J3: Mul_GFP *AR0+, *AR1, *AR2+
	BRLE #s-1, J3
	STR C, *AR2
	BRLE #s-1, J1

If($t > p$) t=t-p;	STR #0, Reg_index1 CLR Add_C J4: SUBC *AR0+, *AR1+, *AR2+ BRLE #s-1, J4

Table 4.6: Codes for Montgomery multiplication over $GF(p)$

Components	Slices	BRAM
Multiplier	2005	-
Barrel shifter	95	-
Adder/subtracter	91	-
NAF	2	
Full design	2479	67

Table 4.7: Usage of FPGA resources

4.7 Performance Comparison

The comparison with other work for RC5 is shown in Table 4.9. The throughput of RC5-32/12/16 in this design achieves 10.7 Mbit/s on encryption/decryption. The work in [40] implements RC5-32/12/16 for prototype of small sensor devices using an eight-bit processor with its throughput of only 4.8 Kbit/s. The throughput of a software implementation of RC5-32/12/8 in [45] on a Pentium 266MMX can reach 11.4 Mbit/s. This means that the performance of the proposed hybrid processor competes with that of the Pentium 266MMX which uses an advanced superscalar

Algorithms	RC5	AES	ECC over $GF(2^{146})$	ECC over $GF(p)$ 192-bit
Throughput (Mbit/s)	10.7	2.0	-	-
scalar multiplication (ms)	-	-	12	28.4

Table 4.8: The performance for different cryptosystems

Performance parameters	Perrig et al. [40]	Sesstions [45]	This work
Throughput (Mbit/s)	4.8 Kbit/s	11.4 Mbit/s	10.7 Mbit/s

Table 4.9: The performance comparison for RC5

architecture.

Performance parameters		Atasu et al. [3]	This work
Throughput (Mbit/s)	Encryption	0.5	2.0
	Decryption	0.29	2.0

Table 4.10: The performance comparison for AES

Atasu et al. [3] implements AES on ARM-based platforms, which are 32-bit embedded Reduced Instruction Set Computer (RISC) microprocessors. They also use the look-up table method for S-box operations. The throughputs are much lower than those in this design for both encryption and decryption.

Weimerskirch et al. [54] implements the elliptic curve over $GF(2^{163})$ on a Palm OS device with a 32-bit processor. The time for point multiplication using Montgomery scalar multiplication is 2.73s. The work in [27] proposed the new Montgomery scalar multiplication algorithms and reported results obtained from a software implementation using a Sun UltraSPARC 300 MHz. From the Table 4.11, it can be seen that the scalar multiplication in this thesis is better than [54, 27]. Kim and Lee [18] also propose a secret-key and public-key crypto-processor. It can perform AES, SEED and ECC. The design implements each cryptographic algorithm separately using each dedicated cryptographic block, which uses 4725 slices for AES and ECC. However, the common arithmetic components are extracted and shared in this design, and only 2479 slices are required in total, which is only the 1/2 of that used in [18]. In addition, the work in [18] can only perform on the fixed elliptic

Implementations	Scalar multiplication	Field	Platform	Frequency (MHz)
This work	12 (ms)	$GF(2^{146})$	Xilinx xc2v3000-5bf957	60.15
Weimerskirch et al. [54]	2.73 (s)	$GF(2^{163})$	Palm OS	16
López et al. [27]	13.5 (ms)	$GF(2^{163})$	UltraSPARC	300 (software)
Kim et al. [18]	7.28 (ms)	$GF(2^{146})$	FPGA	50

Table 4.11: The scalar multiplication comparison for ECC over $GF(2^n)$

Logic size (slices)	Implementations	AES	ECC	Total
	This work	2479		
	Kim et al.[18]	1689	3036	4725

Table 4.12: The hardware comparison for ECC over $GF(2^n)$

curve $GF(2^{146})$, whereas the design in this thesis is much flexible, which performs ECC not only over $GF(2^n)$, but also over $GF(p)$. The prime number of p of $GF(p)$ and the irreducible polynomial of $GF(2^n)$ can be changed easily to meet different requirements. It increases the security and reduces the users' cost to easily adopt more secure elliptic curves without changing the hardware.

Implementations	Scalar multiplication	Field	Platform	Frequency (MHz)
This work	28.4 (ms)	$GF(p)$ 192-bit	FPGA	60.15
Xu et al. [55]	30 ms	$GF(2^{192} - 2^{64} - 1)$	ASIC	50

Table 4.13: The performance comparison for ECC over $GF(p)$

The performance comparison with other work of ECC over $GF(p)$ is shown in Table 4.13. Note that the prime field in [55] is fixed for $GF(2^{192} - 2^{64} - 1)$ and for specific elliptic curve with $a = p - 3$, which lacks flexibility.

4.8 Future Improvement

In the future, efforts will be made to speed up the proposed processor by the pipeline technique. The number of clock cycles of each instruction will be reduced (Table

4.14), and the maximum frequency could reach around 90~120 MHz. It is estimated that the time for the scalar multiplication over $GF(2^{146})$ would be 3.4~4.6 ms.

instruction	explanation	clock cycles
Mul_GFP *src*0, *src*1, *src*2, *dst*	multiplication over $GF(p)$ $dst = src0 \times src1 + src2 + C$	3
Mul_GF2N *src*0, *src*1, *src*2, *dst*	multiplication over $GF(2^n)$ $dst = src0 \times src1 \oplus src2 \oplus C$	3
Mul_GF8 *src*0, *src*1, *A*	four multiplications over $GF(2^8)$	2
MixColumns *src*0, *src*1, *dst*	multiplications over $GF(2^8)$ and the result is XORed between the four bytes	2
SQUR_GFP *src*0, *src*1, *dst*	squaring over $GF(p)$ $dst = src0 \times src0 + src1 + C$	3
SQUR_GF2N *src*0, *src*1, *dst*	squaring over $GF(2^n)$ $dst = src0 \times src0 \oplus src1 \oplus C$	3
SHFT	barrel shift	2
XOR	exclusive OR	2
ADD *src*0, *src*1, *dst*	addition	3
SUB *src*0, *src*1, *dst*	subtraction	3
LDA	load into A (32-bit)	2
	or load into A0, A1, A2, A3 (eight-bit)	1
STR	store constant or A0,A1,A2,A3	1
	or store A or CA to dual-port RAM	2
LKUP	lookup table	2
BRNZ bit, addr	branch to addr if test bit is set	1/2
BRLE #const, addr	branch to addr if index $\leq$ #const	1/2
CLR	clear the carry bit of adder/subtracter or clear the carry register of multiplier	1
NOP		1

Table 4.14: Instruction set using pipeline technique

// S-box	// ShiftRows	// MixColumns
LDA *AR0+	LDA Reg0, A3	MixColumns *AR0, *AR2+, A3
LKUP A3	LDA Reg5, A2	MixColumns *AR0, *AR2+, A2
LKUP A2	LDA Reg10, A1	MixColumns *AR0, *AR2+, A1
LKUP A1	LDA Reg15, A0	MixColumns *AR0, *AR2+, A0
LKUP A0	STR A, *AR0+	STR A, *AR0+
STR A3, Reg0	LDA Reg4, A3	MixColumns *AR0, *AR2+, A3
STR A2, Reg1	LDA Reg9, A2	MixColumns *AR0, *AR2+, A2
STR A1, Reg2	LDA Reg14, A1	MixColumns *AR0, *AR2+, A1
STR A0, Reg3	LDA Reg3, A0	MixColumns *AR0, *AR2+, A0
LDA *AR0+	STR A, *AR0+	STR A, *AR0+
LKUP A3	LDA Reg8, A3	MixColumns *AR0, *AR2+, A3
LKUP A2	LDA Reg13, A2	MixColumns *AR0, *AR2+, A2
LKUP A1	LDA Reg2, A1	MixColumns *AR0, *AR2+, A1
LKUP A0	LDA Reg7, A0	MixColumns *AR0, *AR2+, A0
STR A3, Reg4	STR A, *AR0+	STR A, *AR0+
STR A2, Reg5	LDA Reg12, A3	MixColumns *AR0, *AR2+, A3
STR A1, Reg6	LDA Reg1, A2	MixColumns *AR0, *AR2+, A2
STR A0, Reg7	LDA Reg6, A1	MixColumns *AR0, *AR2+, A1
LDA *AR0+	LDA Reg11, A0	MixColumns *AR0, *AR2+, A0
LKUP A3	STR A, *AR0+	STR A, *AR0+
LKUP A2		//AddRoundKey
LKUP A1		XOR *AR0,*AR1+, A
LKUP A0		STR A, *AR0+
STR A3, Reg8		XOR *AR0,*AR1+, A
STR A2, Reg9		STR A, *AR0+
STR A1, Reg10		XOR *AR0,*AR1+, A
STR A0, Reg11		STR A, *AR0+
LDA *AR0+		
LKUP A3		
LKUP A2		
LKUP A1		
LKUP A0		
STR A2, Reg12		
STR A1, Reg13		
STR A0, Reg14		
STR A3, Reg15		

Table 4.15: Codes for AES

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The architecture of a novel cryptographic processor that supports both public-key cryptosystem and secret-key cryptographic algorithms is proposed in this thesis. It provides flexibility for applications where public-key cryptography is first required for secure key exchange, and then secret-key cryptosystem is used for secure data transportation. It is suitable for resource limited devices such as smart cards and cellular phones where secure communication can be provided.

In the design, the common components used by ECC, AES and RC5 are extracted. A novel multifunction multiplier is proposed which can perform multiplications over $GF(p)$ and $GF(2^n)$ in ECC and multiplications over $GF(2^8)$ used in MixColumns operation in AES. The Reduced Area tree is utilized in the design of the multiplier, which provides flexibility to deal with the three cases. The number of reduction stages is the same as the one of only multiplications over $GF(p)$ without affecting the critical path of the multiplier.

The implementation of ECC is the most complex part in the design, which involves selections of algorithms for the scalar multiplication kP, the coordinates for point in ECC, and the basis for finite fields over $GF(p)$ and $GF(2^n)$. The Nonadjacent Form (NAF) is chosen for the scalar multiplication over $GF(p)$ in modified

Jacobian projective coordinates; the Montgomery scalar multiplication is selected for the scalar multiplication over $GF(2^n)$ in projective coordinates. The polynomial basis is used for both finite fields, which flexibly converts the long operands into multiple-precision operations. This makes it possible that different elliptic curves can be implemented without changing the hardware. It increases the security of the processor considering that new elliptic curves should be adopted when the old one is no longer secure.

The performance comparison indicates that the proposed hybrid crypto-processor can achieve better performance than previous work.

5.2 Future Work

The proposed design can support other cryptographic algorithms. For example, the RSA algorithm uses modular exponentiations which can be implemented through repeated multiplications and squarings. In addition, many other cryptosystems can be implemented after analyzing the algorithms, extracting common parts, and adding new components to the processor.

The speed of the proposed processor can be much faster if the pipeline technique is applied and the instruction set is further optimized.

Bibliography

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHAED: An encryption scheme on the Diffe-Hellman problem. *Meeting by IEEE P1363: Standard for Public-Key Cryptography*, 1998.

[2] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal On Selected Areas In Communications*, 11(5):804–813, June 1993.

[3] Kubilary Atasu, Luca Breveglieri, and Marco Macchetti. Efficient AES implementations for ARM based platforms. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 841–845, Nicosia, Cyprus, March 2004.

[4] M. Bednara, M. Daldrup, J. Teich, J. von zur Gathen, and J. Shokrollahi. Tradeoff analysis of FPGA based elliptic curve cryptography. In *IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, pages V–797–V–800, May 2002.

[5] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

[6] Pawel Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. In *Cryptographic Hardware and Embedded Systems-CHES 2003, LNCS 2779*, pages 319–333. Springer-Verlag, 2003.

[7] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinate. In *Advances in Cryptology - ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security, LNCS 1514*, pages 51–65. Springer-Verlag, 1998.

[8] Whitfield Diffie and Martin Hellman. New direction in cryptography. *IEEE Transactions on Information theory*, 22:644–654, 1976.

[9] Hans Eberle, Nils Gura, Sheueling Chang Shantz, Vipul Gupta, and Leonard Rarick. A public-key cryptographic processor for RSA and ECC. In *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)*, pages 202–219, September 2004.

[10] Lijun Gao, Sarvesh Shrivastava, and Gerald E. Sobelman. Elliptic curve scalar multiplier design using FPGAs. In *Cryptographic Hardware and Embedded Systems: First International Workshop, CHES'99, LNCS 1717*, pages 257–268. Springer-Verlag, August 1999.

[11] James Goodman and Anantha P. Chandrakasan. An energy efficient reconfigurable public-key cryptography processor architecture. *IEEE Journal of Solid-state Circuits*, 36(11):1808–1820, November 2001.

[12] Johann Großschädl. A bit-serial unified multiplier architecture for finite field $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems CHES 2001, LNCS 2162*, pages 202–219. Springer-Verlag, May 2001.

[13] I. N. Herstein. *Abstract Algebra.* Macmillan Publishing Company, 1990.

[14] IEEE 1363-2000: Standard specifications for public-key cryptography, January 2000. available: http://grouper.ieee.org/groups/1363/P1363.

[15] Don Johnson and Alfred Menezes. The elliptic curve digital signature algorithm (ECDSA). Technical Report CORR99-34, Deptment of C&O, University of Waterloo, Canada, February 2000.

[16] Marc Joye and Sung-Ming Yen. Optimal left-to-right binary signed-digit recoding. *IEEE Transactions on Computers*, 49:740–748, 2000.

[17] B. S. Kalinski and Y. L. Yin. On the security of the RC5 encryption algorithm. *CryptoBytes*, 1(2):13–14, 1995.

[18] Ho Won Kim and Sunggu Lee. Design and implementation of a private and public key crypto processor and its application to a security system. *IEEE Transaction on Consumer Electronics*, 50(1):214–224, February 2004.

[19] Neal Koblitz. Elliptic curve cryptosystems. *Mathematic of Computation*, 48: 203–209, 1987.

[20] Neal Koblitz, Alfred Menezes, and Scott Vanstone. The state of elliptic curve cryptography. *Designs, Codes and Cryptography*, 19:173–193, 2000.

[21] Cetin Kaya Koç, Tolga Acar, and Burton S. Kaliski. Analyzing and computing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[22] Cetin Kaya Koç, Tolga Acar, and Burton S. Kaliski. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[23] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong. FPGA implementation of a microcoded elliptic curve cryptographic processor. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 68–76, April 2000.

[24] Hua Li and Zac Friggstad. An efficient architecture for the AES mix columns operation. In *IEEE International Symposium on Circuits and Systems (ISCAS05)*, pages 4637–4840, May 2005.

[25] Hua Li and Jianzhou Li. A high performance sub-pipelined architecture for AES. In *2005 IEEE International Conference on Computer Design (ICCD05)*, pages 491–496, October 2005.

[26] Hua Li, Jianzhou Li, and Jing Yang. An efficient and reconfigurable architecture for RC5. In *18th Annual Canadian Conference on Electrical and Computer Engineering (CCECE05)*, pages 1652–1655, May 2005.

[27] Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Cryptographic Hardware and Embedded Systems: First International Workshop, CHES'99, LNCS 1717*, pages 316–327. Springer-Verlag, August 1999.

[28] Julio López and Ricardo Dahab. An overview of elliptic curve cryptography. Technical report, Institute of Computing, State University of Campinas, Brazil, May 2000.

[29] Stefan Mangard, Manfred Aigner, and Sandra Dominikus. A highly regular and scalable AES hardware architecture. *IEEE Transactions on Computers*, 52:483–491, 2003.

[30] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[31] Victor Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptology:Proceedings of CRYPTO'85, LNCS 218*, pages 417–426, New York, 1985. Springer-Verlag.

[32] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[33] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[34] R C Mullin, I M Onyszchuk, S A Vanstone, and R M Wilson. Optimal normal bases in $GF(p^n)$. *Discret Applied Mathematics*, 22(2):149–161, February 1989.

[35] NIST. Federal information processing standard 197: The Advanced Encryption Standard (AES), 2001. available: http://csrc.nist.gov/publications/fips/fips197/fips197.pdf.

[36] Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka. Implementation of elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA. In *Cryptographic Hardware and Embedded Systems-CHES 2000, LNCS 1965*, pages 25–40. Springer-Verlag, August 2000.

[37] J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. *US Patent Number 4,587,627*, May 1986.

[38] Gerardo Orlando and Christof Paar. A high-performance reconfigurable elliptic curve processor for $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems-CHES 2000, LNCS 1965*, pages 41–56. Springer-Verlag, August 2000.

[39] Gerardo Orlando and Christof Paar. A scalable $GF(p)$ elliptic curve processor architecture for programmable hardware. In *Cryptographic Hardware and Embedded Systems CHES 2001, LNCS 2162*, pages 348–363. Springer-Verlag, May 2001.

[40] Adrian Perrig, Robert Szewczyk, Victor Wen, David E. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *Proceedings of the 7th annual international conference on mobile computing and networking*, pages 189–199, Rome, Italy, 2001. ACM Press.

[41] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[42] R. L. Rivest. The RC5 encryption algorithm. In *Proceedings of the 1994 Leuven Workshop on Fast Software Encryption*, pages 86–96. Springer-Verlag, 1995.

[43] Akashi Satoh and Kohji Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Transactions on Computers*, 52(4):449–460, April 2003.

[44] Erkay Savas, Alexandre F. Tenca, and Cetin Kaya Koç. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems-CHES 2000, LNCS 1965*, pages 277–292. Springer-Verlag, August 2000.

[45] Julian Brently Sessions. Fast software implementations of block ciphers. Master's thesis, Oregon State University, Department of Electrical & Computer Engineering, November 1998.

[46] N. Sklavos, C. Machas, and O. Koufopavlou. Area optimized architecture and VLSI implementation of RC5 encryption algorithm. In *Proceedings of 2003 10th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2003)*, pages 172–175 Vol.1, December 2003.

[47] N. Sklavos, G. Selimis, and O. Koufoupavlou. Bulk encryption crypto-processor for smart cards: design and implementation. In *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and systems (ICECS 2004)*, pages 579–582, 2004.

[48] Jerome A. Solinas. An improved algorithm for arithmetic on a family of elliptic curves. In *Advances in Cryptology - CRYPTO'97: 17th Annual International Cryptology Conference, LNCS 1294*, pages 357–371. Springer-Verlag, 1997.

[49] William Stallings. *Cryptography and Network Security.* Prentice Hall, third edition, 2003.

[50] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. Efficient implementation of Rijndael encryption in reconfigurable hardware:improvements and design tradeoffs. In *Cryptographic Hardware and Embedded Systems-CHES 2003, LNCS 2779*, pages 334–350. Springer-Verlag, 2003.

[51] James E. Stine. *Digital Computer Arithmetic Datapath Design Using Verilog HDL*. Kluwer Academic Publishers, 2004.

[52] Scott Vanstone. Responses to NIST's proposal. *Communications of the ACM*, 35(7):50–52, 1992.

[53] Ingrid Verbauwhede, Patrick Schaumont, and Henry Kuo. Design and performance testing of a 2.29 Gb/s Rijndael processor. *IEEE Journal of Solid-State Circuits*, 38(3):569–572, March 2003.

[54] André Weimerskirch, Christof Paar, and Sheueling Chang Shantz. Elliptic curve cryptography on a palm OS device. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy, LNCS 2119*, pages 502–513, Sydney, Australia, July 2001. Springer-Verlag.

[55] Sheng-Bo Xu and Lejla Batina. Efficient implementation of elliptic curve cryptosystems on an ARM7 with hardware accelerator. In *Proceedings of the 4th International Conference on Information Security*, pages 266–279, October 2001.

Appendix A

Part of Verliog HDL codes

A.1 Verilog HDL codes for multiplier

```
module RA(S,MUL8,A,B,C,D,sig_sel,A8,A9,A10,A11,A12,A13,A14);
input [31:0] B,D;
input [32:0] A,C;
input sig_sel; input [7:0] A8,A9,A10,A11,A12,A13,A14;
output [7:0] MUL8;
output [64:0] S;
wire [32:0] P31,P30, P29, P28, P27, P26, P25, P24,
P23,P22,P21,P20,P19,P18,P17,P16,P15,P14,P13,P12,P11,P10,P9,
P8,P7,P6,P5,P4,P3,P2,P1,P0;
wire [58:0] Sum;
wire [64:0] mul_P_64,mul_2n_64;
// Partial Product Generation
PP PP1(P31,P30, P29, P28, P27, P26, P25, P24,
P23,P22,P21,P20,P19,P18,P17,P16,P15,P14,P13,P12,P11,
P10,P9,P8,P7,P6,P5,P4,P3,P2,P1,P0,A, B);
//Partial Product Reduction
//stage2
```

```
//first 8*8
//////////////// ha (Sum, Cout,A, B);
ha HA1(NN2_0_1,NN2_1_2,P0[1],P1[0]);
fa FA1(NN2_0_2,NN2_1_3,P0[2],P1[1],P2[0]);
fa FA2(NN2_0_3,NN2_1_4,P0[3],P1[2],P2[1]);
fa FA3(NN2_0_4,NN2_1_5,P0[4],P1[3],P2[2]);
fa FA4(NN2_0_5,NN2_1_6,P0[5],P1[4],P2[3]);
fa FA5(NN2_2_5,NN2_3_6,P3[2],P4[1],P5[0]);
fa FA6(NN2_0_6,NN2_1_7,P0[6],P1[5],P2[4]);
fa FA7(NN2_2_6,NN2_3_7,P3[3],P4[2],P5[1]);
fa FA8(NN2_0_7,NN2_1_8,P0[7],P1[6],P2[5]);
fa FA9(NN2_2_7,NN2_3_8,P3[4],P4[3],P5[2]);
ha HA2(NN2_4_7,NN2_5_8,P6[1],P7[0]);
fa FA10(NN2_0_8,NN2_1_9,P1[7],P2[6],P3[5]);
fa FA11(NN2_2_8,NN2_3_9,P4[4],P5[3],P6[2]);
fa FA12(NN2_0_9,NN2_1_10,P2[7],P3[6],P4[5]);
fa FA13(NN2_2_9,NN2_3_10,P5[4],P6[3],P7[2]);
fa FA14(NN2_0_10,NN2_1_11,P3[7],P4[6],P5[5]);
fa FA15(NN2_0_11,NN2_1_12,P4[7],P5[6],P6[5]);
fa FA16(NN2_0_12,NN2_1_13,P5[7],P6[6],P7[5]);
//second 8*8
fa FA17(NN2_0_18,NN2_1_19,P8[10],P9[9],P10[8]);
fa FA18(NN2_0_19,NN2_1_20,P8[11],P9[10],P10[9]);
fa FA19(NN2_0_20,NN2_1_21,P8[12],P9[11],P10[10]);
fa FA20(NN2_0_21,NN2_1_22,P8[13],P9[12],P10[11]);
fa FA21(NN2_2_21,NN2_3_22,P11[10],P12[9],P13[8]);
fa FA22(NN2_0_22,NN2_1_23,P8[14],P9[13],P10[12]);
fa FA23(NN2_2_22,NN2_3_23,P11[11],P12[10],P13[9]);
```

```
fa FA24(NN2_0_23,NN2_1_24,P8[15],P9[14],P10[13]);
fa FA25(NN2_2_23,NN2_3_24,P11[12],P12[11],P13[10]);
ha HA4(NN2_4_23,NN2_5_24,P14[9],P15[8]);
fa FA26(NN2_0_24,NN2_1_25,P9[15],P10[14],P11[13]);
fa FA27(NN2_2_24,NN2_3_25,P12[12],P13[11],P14[10]);
fa FA28(NN2_0_25,NN2_1_26,P10[15],P11[14],P12[13]);
fa FA29(NN2_2_25,NN2_3_26,P13[12],P14[11],P15[10]);
fa FA30(NN2_0_26,NN2_1_27,P11[15],P12[14],P13[13]);
fa FA31(NN2_0_27,NN2_1_28,P12[15],P13[14],P14[13]);
fa FA32(NN2_0_28,NN2_1_29,P13[15],P14[14],P15[13]);
//third 8*8
ha HA5(NN2_0_33,NN2_1_34,P16[17],P17[16]);
fa FA33(NN2_0_34,NN2_1_35,P16[18],P17[17],P18[16]);
fa FA34(NN2_0_35,NN2_1_36,P16[19],P17[18],P18[17]);
fa FA35(NN2_0_36,NN2_1_37,P16[20],P17[19],P18[18]);
fa FA36(NN2_0_37,NN2_1_38,P16[21],P17[20],P18[19]);
fa FA37(NN2_2_37,NN2_3_38,P19[18],P20[17],P21[16]);
fa FA38(NN2_0_38,NN2_1_39,P16[22],P17[21],P18[20]);
fa FA39(NN2_2_38,NN2_3_39,P19[19],P20[18],P21[17]);
fa FA40(NN2_0_39,NN2_1_40,P16[23],P17[22],P18[21]);
fa FA41(NN2_2_39,NN2_3_40,P19[20],P20[19],P21[18]);
ha HA94(NN2_4_39,NN2_5_40,P22[17],P23[16]);
fa FA42(NN2_0_40,NN2_1_41,P17[23],P18[22],P19[21]);
fa FA43(NN2_2_40,NN2_3_41,P20[20],P21[19],P22[18]);
fa FA44(NN2_0_41,NN2_1_42,P18[23],P19[22],P20[21]);
fa FA45(NN2_2_41,NN2_3_42,P21[20],P22[19],P23[18]);
fa FA46(NN2_0_42,NN2_1_43,P19[23],P20[22],P21[21]);
fa FA47(NN2_0_43,NN2_1_44,P20[23],P21[22],P22[21]);
```

```
fa FA48(NN2_0_44,NN2_1_45,P21[23],P22[22],P23[21]);
ha HA93(NN2_0_45,NN2_1_46,P22[23],P23[22]);

    //fourth 8*8
ha HA6(NN2_0_49,NN2_1_50,P24[25],P25[24]);
fa FA49(NN2_0_50,NN2_1_51,P24[26],P25[25],P26[24]);
fa FA50(NN2_0_51,NN2_1_52,P24[27],P25[26],P26[25]);
fa FA51(NN2_0_52,NN2_1_53,P24[28],P25[27],P26[26]);
fa FA52(NN2_0_53,NN2_1_54,P24[29],P25[28],P26[27]);
fa FA53(NN2_2_53,NN2_3_54,P27[26],P28[25],P29[24]);
fa FA54(NN2_0_54,NN2_1_55,P24[30],P25[29],P26[28]);
fa FA55(NN2_2_54,NN2_3_55,P27[27],P28[26],P29[25]);
fa FA56(NN2_0_55,NN2_1_56,P24[31],P25[30],P26[29]);
fa FA57(NN2_2_55,NN2_3_56,P27[28],P28[27],P29[26]);
ha HA7(NN2_4_55,NN2_5_56,P30[25],P31[24]);
fa FA58(NN2_0_56,NN2_1_57,P25[31],P26[30],P27[29]);
fa FA59(NN2_2_56,NN2_3_57,P28[28],P29[27],P30[26]);
fa FA60(NN2_0_57,NN2_1_58,P26[31],P27[30],P28[29]);
fa FA61(NN2_2_57,NN2_3_58,P29[28],P30[27],P31[26]);
fa FA62(NN2_0_58,NN2_1_59,P27[31],P28[30],P29[29]);
fa FA63(NN2_0_59,NN2_1_60,P28[31],P29[30],P30[29]);
fa FA64(NN2_0_60,NN2_1_61,P29[31],P30[30],P31[29]);
////////////////////////////////
fa FA65(N2_0_8,N2_1_9,P0[8],P8[0],C[8]); //col 9
fa FA66(N2_0_9,N2_1_10,P0[9],P1[8],P8[1]);
fa FA67(N2_2_9,N2_3_10,P9[0],C[9], D[9]); //col 10
fa FA68(N2_0_10,N2_1_11,P0[10],P1[9],P2[8]);
fa FA69(N2_2_10,N2_3_11,P8[2],P9[1],P10[0]); //col 11
```

```
fa FA70(N2_0_11,N2_1_12,P0[11],P1[10],P2[9]);
fa FA71(N2_2_11,N2_3_12,P3[8],P8[3],P9[2]);
fa FA72(N2_4_11,N2_5_12,P10[1],P11[0],C[11]); //col 12
fa FA73(N2_0_12,N2_1_13,P0[12],P1[11],P2[10]);
fa FA74(N2_2_12,N2_3_13,P3[9],P4[8],P8[4]);
fa FA75(N2_4_12,N2_5_13,P9[3],P10[2],P11[1]);
fa FA76(N2_6_12,N2_7_13,P12[0],C[12],D[12]); //col 13
fa FA77(N2_0_13,N2_1_14,P0[13],P1[12],P2[11]);
fa FA78(N2_2_13,N2_3_14,P3[10],P4[9],P5[8]);
fa FA79(N2_4_13,N2_5_14,P8[5],P9[4],P10[3]);
fa FA80(N2_6_13,N2_7_14,P11[2],P12[1],P13[0]); //col 14
fa FA81(N2_0_14,N2_1_15,P0[14],P1[13],P2[12]);
fa FA82(N2_2_14,N2_3_15,P3[11],P4[10],P5[9]);
fa FA83(N2_4_14,N2_5_15,P6[8],P8[6],P9[5]);
fa FA84(N2_6_14,N2_7_15,P10[4],P11[3],P12[2]);
fa FA85(N2_8_14,N2_9_15,P13[1],P14[0],C[14]); //col 15
fa FA86(N2_0_15,N2_1_16,P0[15],P1[14],P2[13]);
fa FA87(N2_2_15,N2_3_16,P3[12],P4[11],P5[10]);
fa FA88(N2_4_15,N2_5_16,P6[9],P7[8],P8[7]);
fa FA89(N2_6_15,N2_7_16,P9[6],P10[5],P11[4]);
fa FA90(N2_8_15,N2_9_16,P12[3],P13[2],P14[1]);
fa FA91(N2_10_15,N2_11_16,P15[0],C[15],D[15]);
fa FA92(N2_0_16,N2_1_17,P0[16],P1[15],P2[14]);
fa FA93(N2_2_16,N2_3_17,P3[13],P4[12],P5[11]);
fa FA94(N2_4_16,N2_5_17,P6[10],P7[9],P9[7]);
fa FA95(N2_6_16,N2_7_17,P10[6],P11[5],P12[4]);
fa FA96(N2_8_16,N2_9_17,P13[3],P14[2],P15[1]);
fa FA97(N2_10_16,N2_11_17,P16[0],C[16],D[16]);
```

fa FA98(N2_0_17,N2_1_18,P0[17],P1[16],P2[15]);

fa FA99(N2_2_17,N2_3_18,P3[14],P4[13],P5[12]);

fa FA100(N2_4_17,N2_5_18,P6[11],P7[10],P10[7]);

fa FA101(N2_6_17,N2_7_18,P11[6],P12[5],P13[4]);

fa FA102(N2_8_17,N2_9_18,P14[3],P15[2],P16[1]);

fa FA103(N2_10_17,N2_11_18,P17[0],C[17],D[17]);

fa FA104(N2_0_18,N2_1_19,P0[18],P1[17],P2[16]);

fa FA105(N2_2_18,N2_3_19,P3[15],P4[14],P5[13]);

fa FA106(N2_4_18,N2_5_19,P6[12],P7[11],P11[7]);

fa FA107(N2_6_18,N2_7_19,P12[6],P13[5],P14[4]);

fa FA108(N2_8_18,N2_9_19,P15[3],P16[2],P17[1]);

fa FA109(N2_10_18,N2_11_19,P18[0],C[18],D[18]);

fa FA110(N2_0_19,N2_1_20,P0[19],P1[18],P2[17]);

fa FA111(N2_2_19,N2_3_20,P3[16],P4[15],P5[14]);

fa FA112(N2_4_19,N2_5_20,P6[13],P7[12],P12[7]);

fa FA113(N2_6_19,N2_7_20,P13[6],P14[5],P15[4]);

fa FA114(N2_8_19,N2_9_20,P16[3],P17[2],P18[1]);

fa FA115(N2_10_19,N2_11_20,P19[0],C[19],D[19]);

fa FA116(N2_0_20,N2_1_21,P0[20],P1[19],P2[18]);

fa FA117(N2_2_20,N2_3_21,P3[17],P4[16],P5[15]);

fa FA118(N2_4_20,N2_5_21,P6[14],P7[13],P13[7]);

fa FA119(N2_6_20,N2_7_21,P14[6],P15[5],P16[4]);

fa FA120(N2_8_20,N2_9_21,P17[3],P18[2],P19[1]);

fa FA121(N2_10_20,N2_11_21,P20[0],C[20],D[20]);

fa FA122(N2_0_21,N2_1_22,P0[21],P1[20],P2[19]);

fa FA123(N2_2_21,N2_3_22,P3[18],P4[17],P5[16]);

fa FA124(N2_4_21,N2_5_22,P6[15],P7[14],P14[7]);

fa FA125(N2_6_21,N2_7_22,P15[6],P16[5],P17[4]);

fa FA126(N2_8_21,N2_9_22,P18[3],P19[2],P20[1]);

fa FA127(N2_10_21,N2_11_22,P21[0],C[21],D[21]);

fa FA128(N2_0_22,N2_1_23,P0[22],P1[21],P2[20]);

fa FA129(N2_2_22,N2_3_23,P3[19],P4[18],P5[17]);

fa FA130(N2_4_22,N2_5_23,P6[16],P7[15],P15[7]);

fa FA131(N2_6_22,N2_7_23,P16[6],P17[5],P18[4]);

fa FA132(N2_8_22,N2_9_23,P19[3],P20[2],P21[1]);

fa FA133(N2_10_22,N2_11_23,P22[0],C[22],D[22]);

fa FA134(N2_0_23,N2_1_24,P0[23],P1[22],P2[21]);

fa FA135(N2_2_23,N2_3_24,P3[20],P4[19],P5[18]);

fa FA136(N2_4_23,N2_5_24,P6[17],P7[16],P16[7]);

fa FA137(N2_6_23,N2_7_24,P17[6],P18[5],P19[4]);

fa FA138(N2_8_23,N2_9_24,P20[3],P21[2],P22[1]);

fa FA139(N2_10_23,N2_11_24,P23[0],C[23],D[23]);

fa FA140(N2_0_24,N2_1_25,P0[24],P1[23],P2[22]);

fa FA141(N2_2_24,N2_3_25,P3[21],P4[20],P5[19]);

fa FA142(N2_4_24,N2_5_25,P6[18],P7[17],P8[16]);

fa FA143(N2_6_24,N2_7_25,P16[8],P17[7],P18[6]);

fa FA144(N2_8_24,N2_9_25,P19[5],P20[4],P21[3]);

fa FA145(N2_10_24,N2_11_25,P22[2],P23[1],P24[0]);

fa FA146(N2_0_25,N2_1_26,P0[25],P1[24],P2[23]);

fa FA147(N2_2_25,N2_3_26,P3[22],P4[21],P5[20]);

fa FA148(N2_4_25,N2_5_26,P6[19],P7[18],P8[17]);

fa FA149(N2_6_25,N2_7_26,P9[16],P16[9],P17[8]);

fa FA150(N2_8_25,N2_9_26,P18[7],P19[6],P20[5]);

fa FA151(N2_10_25,N2_11_26,P21[4],P22[3],P23[2]);

fa FA152(N2_12_25,N2_13_26,P24[1],P25[0],C[25]);

fa FA153(N2_0_26,N2_1_27,P0[26],P1[25],P2[24]);

fa FA154(N2_2_26,N2_3_27,P3[23],P4[22],P5[21]);

fa FA155(N2_4_26,N2_5_27,P6[20],P7[19],P8[18]);

fa FA156(N2_6_26,N2_7_27,P9[17],P10[16],P16[10]);

fa FA157(N2_8_26,N2_9_27,P17[9],P18[8],P19[7]);

fa FA158(N2_10_26,N2_11_27,P20[6],P21[5],P22[4]);

fa FA159(N2_12_26,N2_13_27,P23[3],P24[2],P25[1]);

fa FA160(N2_14_26,N2_15_27,P26[0],C[26],D[26]);

fa FA161(N2_0_27,N2_1_28,P0[27],P1[26],P2[25]);

fa FA162(N2_2_27,N2_3_28,P3[24],P4[23],P5[22]);

fa FA163(N2_4_27,N2_5_28,P6[21],P7[20],P8[19]);

fa FA164(N2_6_27,N2_7_28,P9[18],P10[17],P11[16]);

fa FA165(N2_8_27,N2_9_28,P16[11],P17[10],P18[9]);

fa FA166(N2_10_27,N2_11_28,P19[8],P20[7],P21[6]);

fa FA167(N2_12_27,N2_13_28,P22[5],P23[4],P24[3]);

fa FA168(N2_14_27,N2_15_28,P25[2],P26[1],P27[0]);

fa FA169(N2_0_28,N2_1_29,P0[28],P1[27],P2[26]);

fa FA170(N2_2_28,N2_3_29,P3[25],P4[24],P5[23]);

fa FA171(N2_4_28,N2_5_29,P6[22],P7[21],P8[20]);

fa FA172(N2_6_28,N2_7_29,P9[19],P10[18],P11[17]);

fa FA173(N2_8_28,N2_9_29,P12[16],P16[12],P17[11]);

fa FA174(N2_10_28,N2_11_29,P18[10],P19[9],P20[8]);

fa FA175(N2_12_28,N2_13_29,P21[7],P22[6],P23[5]);

fa FA176(N2_14_28,N2_15_29,P24[4],P25[3],P26[2]);

fa FA177(N2_16_28,N2_17_29,P27[1],P28[0],C[28]);

fa FA178(N2_0_29,N2_1_30,P0[29],P1[28],P2[27]);

fa FA179(N2_2_29,N2_3_30,P3[26],P4[25],P5[24]);

fa FA180(N2_4_29,N2_5_30,P6[23],P7[22],P8[21]);

fa FA181(N2_6_29,N2_7_30,P9[20],P10[19],P11[18]);

fa FA182(N2_8_29,N2_9_30,P12[17],P13[16],P16[13]);

fa FA183(N2_10_29,N2_11_30,P17[12],P18[11],P19[10]);

fa FA184(N2_12_29,N2_13_30,P20[9],P21[8],P22[7]);

fa FA185(N2_14_29,N2_15_30,P23[6],P24[5],P25[4]);

fa FA186(N2_16_29,N2_17_30,P26[3],P27[2],P28[1]);

fa FA187(N2_18_29,N2_19_30,P29[0],C[29],D[29]);

fa FA188(N2_0_30,N2_1_31,P0[30],P1[29],P2[28]);

fa FA189(N2_2_30,N2_3_31,P3[27],P4[26],P5[25]);

fa FA190(N2_4_30,N2_5_31,P6[24],P7[23],P8[22]);

fa FA191(N2_6_30,N2_7_31,P9[21],P10[20],P11[19]);

fa FA192(N2_8_30,N2_9_31,P12[18],P13[17],P14[16]);

fa FA193(N2_10_30,N2_11_31,P16[14],P17[13],P18[12]);

fa FA194(N2_12_30,N2_13_31,P19[11],P20[10],P21[9]);

fa FA195(N2_14_30,N2_15_31,P22[8],P23[7],P24[6]);

fa FA196(N2_16_30,N2_17_31,P25[5],P26[4],P27[3]);

fa FA197(N2_18_30,N2_19_31,P28[2],P29[1],P30[0]);

fa FA198(N2_20_30,N2_21_31,P15[15],C[30],D[30]);

fa FA199(N2_0_31,N2_1_32,P0[31],P1[30],P2[29]);

fa FA200(N2_2_31,N2_3_32,P3[28],P4[27],P5[26]);

fa FA201(N2_4_31,N2_5_32,P6[25],P7[24],P8[23]);

fa FA202(N2_6_31,N2_7_32,P9[22],P10[21],P11[20]);

fa FA203(N2_8_31,N2_9_32,P12[19],P13[18],P14[17]);

fa FA204(N2_10_31,N2_11_32,P15[16],P16[15],P17[14]);

fa FA205(N2_12_31,N2_13_32,P18[13],P19[12],P20[11]);

fa FA206(N2_14_31,N2_15_32,P21[10],P22[9],P23[8]);

fa FA207(N2_16_31,N2_17_32,P24[7],P25[6],P26[5]);

fa FA208(N2_18_31,N2_19_32,P27[4],P28[3],P29[2]);

fa FA209(N2_20_31,N2_21_32,P30[1],P31[0],C[31]);

```
fa FA210(N2_0_32,N2_1_33,P1[31],P2[30],P3[29]);
fa FA211(N2_2_32,N2_3_33,P4[28],P5[27],P6[26]);
fa FA212(N2_4_32,N2_5_33,P7[25],P8[24],P9[23]);
fa FA213(N2_6_32,N2_7_33,P10[22],P11[21],P12[20]);
fa FA214(N2_8_32,N2_9_33,P13[19],P14[18],P15[17]);
fa FA215(N2_10_32,N2_11_33,P17[15],P18[14],P19[13]);
fa FA216(N2_12_32,N2_13_33,P20[12],P21[11],P22[10]);
fa FA217(N2_14_32,N2_15_33,P23[9],P24[8],P25[7]);
fa FA218(N2_16_32,N2_17_33,P26[6],P27[5],P28[4]);
fa FA219(N2_18_32,N2_19_33,P29[3],P30[2],P31[1]);
fa FA220(N2_20_32,N2_21_33,P16[16],P0[32],C[32]);
fa FA221(N2_0_33,N2_1_34,P2[31],P3[30],P4[29]);
fa FA222(N2_2_33,N2_3_34,P5[28],P6[27],P7[26]);
fa FA223(N2_4_33,N2_5_34,P8[25],P9[24],P10[23]);
fa FA224(N2_6_33,N2_7_34,P11[22],P12[21],P13[20]);
fa FA225(N2_8_33,N2_9_34,P14[19],P15[18],P18[15]);
fa FA226(N2_10_33,N2_11_34,P19[14],P20[13],P21[12]);
fa FA227(N2_12_33,N2_13_34,P22[11],P23[10],P24[9]);
fa FA228(N2_14_33,N2_15_34,P25[8],P26[7],P27[6]);
fa FA229(N2_16_33,N2_17_34,P28[5],P29[4],P30[3]); //col 34
fa FA230(N2_0_34,N2_1_35,P3[31],P4[30],P5[29]);
fa FA231(N2_2_34,N2_3_35,P6[28],P7[27],P8[26]);
fa FA232(N2_4_34,N2_5_35,P9[25],P10[24],P11[23]);
fa FA233(N2_6_34,N2_7_35,P12[22],P13[21],P14[20]);
fa FA234(N2_8_34,N2_9_35,P15[19],P19[15],P20[14]);
fa FA235(N2_10_34,N2_11_35,P21[13],P22[12],P23[11]);
fa FA236(N2_12_34,N2_13_35,P24[10],P25[9],P26[8]);
fa FA237(N2_14_34,N2_15_35,P27[7],P28[6],P29[5]);
```

fa FA238(N2_16_34,N2_17_35,P30[4],P31[3],P2[32]);

fa FA239(N2_0_35,N2_1_36,P4[31],P5[30],P6[29]);

fa FA240(N2_2_35,N2_3_36,P7[28],P8[27],P9[26]);

fa FA241(N2_4_35,N2_5_36,P10[25],P11[24],P12[23]);

fa FA242(N2_6_35,N2_7_36,P13[22],P14[21],P15[20]);

fa FA243(N2_8_35,N2_9_36,P20[15],P21[14],P22[13]);

fa FA244(N2_10_35,N2_11_36,P23[12],P24[11],P25[10]);

fa FA245(N2_12_35,N2_13_36,P26[9],P27[8],P28[7]);

fa FA246(N2_14_35,N2_15_36,P29[6],P30[5],P31[4]);

fa FA247(N2_0_36,N2_1_37,P5[31],P6[30],P7[29]);

fa FA248(N2_2_36,N2_3_37,P8[28],P9[27],P10[26]);

fa FA249(N2_4_36,N2_5_37,P11[25],P12[24],P13[23]);

fa FA250(N2_6_36,N2_7_37,P14[22],P15[21],P21[15]);

fa FA251(N2_8_36,N2_9_37,P22[14],P23[13],P24[12]);

fa FA252(N2_10_36,N2_11_37,P25[11],P26[10],P27[9]);

fa FA253(N2_12_36,N2_13_37,P28[8],P29[7],P30[6]);

fa FA254(N2_0_37,N2_1_38,P6[31],P7[30],P8[29]);

fa FA255(N2_2_37,N2_3_38,P9[28],P10[27],P11[26]);

fa FA256(N2_4_37,N2_5_38,P12[25],P13[24],P14[23]);

fa FA257(N2_6_37,N2_7_38,P15[22],P22[15],P23[14]);

fa FA258(N2_8_37,N2_9_38,P24[13],P25[12],P26[11]);

fa FA259(N2_10_37,N2_11_38,P27[10],P28[9],P29[8]);

fa FA260(N2_12_37,N2_13_38,P30[7],P31[6],P5[32]);

fa FA261(N2_0_38,N2_1_39,P7[31],P8[30],P9[29]);

fa FA262(N2_2_38,N2_3_39,P10[28],P11[27],P12[26]);

fa FA263(N2_4_38,N2_5_39,P13[25],P14[24],P15[23]);

fa FA264(N2_6_38,N2_7_39,P23[15],P24[14],P25[13]);

fa FA265(N2_8_38,N2_9_39,P26[12],P27[11],P28[10]);

fa FA266(N2_10_38,N2_11_39,P29[9],P30[8],P31[7]);

fa FA267(N2_0_39,N2_1_40,P8[31],P9[30],P10[29]);

fa FA268(N2_2_39,N2_3_40,P11[28],P12[27],P13[26]);

fa FA269(N2_4_39,N2_5_40,P14[25],P15[24],P24[15]);

fa FA270(N2_6_39,N2_7_40,P25[14],P26[13],P27[12]);

fa FA271(N2_8_39,N2_9_40,P28[11],P29[10],P30[9]);

fa FA272(N2_0_40,N2_1_41,P9[31],P10[30],P11[29]);

fa FA273(N2_2_40,N2_3_41,P12[28],P13[27],P14[26]);

fa FA274(N2_4_40,N2_5_41,P15[25],P16[24],P24[16]);

fa FA275(N2_6_40,N2_7_41,P25[15],P26[14],P27[13]);

fa FA276(N2_8_40,N2_9_41,P28[12],P29[11],P30[10]);

fa FA277(N2_0_41,N2_1_42,P10[31],P11[30],P12[29]);

fa FA278(N2_2_41,N2_3_42,P13[28],P14[27],P15[26]);

fa FA279(N2_4_41,N2_5_42,P16[25],P17[24],P24[17]);

fa FA280(N2_6_41,N2_7_42,P25[16],P26[15],P27[14]);

fa FA281(N2_8_41,N2_9_42,P28[13],P29[12],P30[11]);

fa FA282(N2_0_42,N2_1_43,P11[31],P12[30],P13[29]);

fa FA283(N2_2_42,N2_3_43,P14[28],P15[27],P16[26]);

fa FA284(N2_4_42,N2_5_43,P17[25],P18[24],P24[18]);

fa FA285(N2_6_42,N2_7_43,P25[17],P26[16],P27[15]);

fa FA286(N2_8_42,N2_9_43,P28[14],P29[13],P30[12]);

fa FA287(N2_0_43,N2_1_44,P12[31],P13[30],P14[29]);

fa FA288(N2_2_43,N2_3_44,P15[28],P16[27],P17[26]);

fa FA289(N2_4_43,N2_5_44,P18[25],P19[24],P24[19]);

fa FA290(N2_6_43,N2_7_44,P25[18],P26[17],P27[16]);

fa FA291(N2_8_43,N2_9_44,P28[15],P29[14],P30[13]);

fa FA292(N2_0_44,N2_1_45,P13[31],P14[30],P15[29]);

fa FA293(N2_2_44,N2_3_45,P16[28],P17[27],P18[26]);

fa FA294(N2_4_44,N2_5_45,P19[25],P20[24],P24[20]);

fa FA295(N2_6_44,N2_7_45,P25[19],P26[18],P27[17]);

fa FA296(N2_8_44,N2_9_45,P28[16],P29[15],P30[14]);

fa FA297(N2_0_45,N2_1_46,P14[31],P15[30],P16[29]);

fa FA298(N2_2_45,N2_3_46,P17[28],P18[27],P19[26]);

fa FA299(N2_4_45,N2_5_46,P20[25],P21[24],P24[21]);

fa FA300(N2_6_45,N2_7_46,P25[20],P26[19],P27[18]);

fa FA301(N2_8_45,N2_9_46,P28[17],P29[16],P30[15]);

fa FA302(N2_0_46,N2_1_47,P15[31],P16[30],P17[29]);

fa FA303(N2_2_46,N2_3_47,P18[28],P19[27],P20[26]);

fa FA304(N2_4_46,N2_5_47,P21[25],P22[24],P24[22]);

fa FA305(N2_6_46,N2_7_47,P25[21],P26[20],P27[19]);

fa FA306(N2_8_46,N2_9_47,P28[18],P29[17],P30[16]);

fa FA307(N2_10_46,N2_11_47,P31[15],P23[23],P14[32]);//col 47

fa FA308(N2_0_47,N2_1_48,P16[31],P17[30],P18[29]);

fa FA309(N2_2_47,N2_3_48,P19[28],P20[27],P21[26]);

fa FA310(N2_4_47,N2_5_48,P22[25],P23[24],P24[23]);

fa FA311(N2_6_47,N2_7_48,P25[22],P26[21],P27[20]);

fa FA312(N2_8_47,N2_9_48,P28[19],P29[18],P30[17]);

fa FA313(N2_0_48,N2_1_49,P17[31],P18[30],P19[29]);

fa FA314(N2_2_48,N2_3_49,P20[28],P21[27],P22[26]);

fa FA315(N2_4_48,N2_5_49,P23[25],P25[23],P26[22]);

fa FA316(N2_6_48,N2_7_49,P27[21],P28[20],P29[19]);

fa FA317(N2_8_48,N2_9_49,P30[18],P31[17],P24[24]);

fa FA318(N2_0_49,N2_1_50,P18[31],P19[30],P20[29]);

fa FA319(N2_2_49,N2_3_50,P21[28],P22[27],P23[26]);

fa FA320(N2_4_49,N2_5_50,P26[23],P27[22],P28[21]);

fa FA321(N2_6_49,N2_7_50,P29[20],P30[19],P31[18]);

```
fa FA322(N2_0_50,N2_1_51,P19[31],P20[30],P21[29]);
fa FA323(N2_2_50,N2_3_51,P22[28],P23[27],P27[23]);
fa FA324(N2_4_50,N2_5_51,P28[22],P29[21],P30[20]);
fa FA325(N2_0_51,N2_1_52,P20[31],P21[30],P22[29]);
fa FA326(N2_2_51,N2_3_52,P23[28],P28[23],P29[22]);
fa FA327(N2_4_51,N2_5_52,P30[21],P31[20],P19[32]);
fa FA328(N2_0_52,N2_1_53,P21[31],P22[30],P23[29]);
fa FA329(N2_2_52,N2_3_53,P29[23],P30[22],P31[21]);
fa FA330(N2_0_53,N2_1_54,P22[31],P23[30],P30[23]);
fa FA331(N2_0_54,N2_1_55,P23[31],P31[23],P22[32]);
...
my_multiplexier m1(S,mul_P_64,mul_2n_64,sig_sel);
////////////////////// mul 8 bit ////////////////
wire [14:0] mul_8_1,mul_8_2,mul_8_3,mul_8_4;
wire [7:0] P8_1,P8_2,P8_3,P8_4;
assign mul_8_1={ NN3_0_14,NN3_0_13,NN3_0_12,NN3_0_11,NN3_0_10,
NN3_0_9,NN3_0_8,NN3_0_7,NN3_0_6,NN3_0_5,NN3_0_4,
NN3_0_3,NN3_0_2,NN3_0_1,NN3_0_0 },
mul_8_2={P15[15],NN3_0_29,NN3_0_28,NN3_0_27,NN3_0_26,NN3_0_25,
NN3_0_24,NN3_0_23,NN3_0_22,NN3_0_21,NN3_0_20,
NN3_0_19,NN3_0_18,NN3_0_17,NN3_0_16},
mul_8_3=NN3_0_46,NN3_0_45,NN3_0_44,
NN3_0_43,NN3_0_42,NN3_0_41,NN3_0_40,NN3_0_39,NN3_0_38,NN3_0_37,
NN3_0_36,NN3_0_35,NN3_0_34,NN3_0_33,NN3_0_32,
mul_8_4=NN3_0_62,NN3_0_61,NN3_0_60,
NN3_0_59,NN3_0_58,NN3_0_57,NN3_0_56,NN3_0_55,NN3_0_54,NN3_0_53,
NN3_0_52,NN3_0_51,NN3_0_50,NN3_0_49,P24[24]};
///PPR(A,P,A8,A9,A10,A11,A12,A13,A14);
```

PPR ppr_1(P8_1,mul_8_1,A8,A9,A10,A11,A12,A13,A14);

PPR ppr_2(P8_2,mul_8_2,A8,A9,A10,A11,A12,A13,A14);

PPR ppr_3(P8_3,mul_8_3,A8,A9,A10,A11,A12,A13,A14);

PPR ppr_4(P8_4,mul_8_4,A8,A9,A10,A11,A12,A13,A14);

assign MUL8=P8_1 ∧ P8_2 ∧ P8_3 ∧ P8_4;

endmodule

A.2 Verilog HDL codes for the data path of the processor

module myalu(addra1,addrb1,addra2,addrb2,in_RAM,Zflag,

index1_flag,index2_flag,cmp_flag1,cmp_flag2,data_in1,data_in2,data_in3,

data_in4,instr,tmp,mul_squ_sel,px_sel, carry_clr,sel_1,load_Sum,

load_Carry,lr,sub,first, sel_regout,sel_A,sel_RAM,

load_Cbit,load_B, load_A0,load_A1,load_A2,load_A3,

load_addrR0,load_addrR1,load_addrR2,load_addrR3,

inc_Index1,inc_Index2,dec_Index1,dec_Index2,

inc_R0,inc_R1,inc_R2,inc_R3,reg_en,enc,read_en,clk,rst);

output [31:0] in_RAM;

output Zflag;

output index1_flag,index2_flag,cmp_flag1,cmp_flag2;

output [7:0] addra1,addrb1,addra2,addrb2;

input [31:0] data_in1,data_in2,data_in3,data_in4;

input mul_squ_sel,carry_clr,sel_1,clk,rst;

input load_Sum,load_Carry,px_sel; //px_sel=1, GF(P), px_sel=0,$GF(2^n)$

input lr,sub,first; //control the barrel-shifter,lr=1, left, lr=0, right sub=1, subtraction

input [1:0] sel_RAM;

```
input [2:0] sel_A;
input load_Cbit,load_B,load_A0,load_A1,load_A2,load_A3;
input [7:0] instr;
input [7:0] tmp;
input [3:0] sel_regout;
input load_addrR0,load_addrR1,load_addrR2,load_addrR3,inc_R0,inc_R1;
input inc_R2, inc_R3, dec_Index1 ,dec_Index2,reg_en,enc,read_en;
input inc_Index1,inc_Index2;
reg index1_flag,index2_flag,cmp_flag1,cmp_flag2;
wire [31:0] sum;
wire [32:0] carry, m1_out,Carr_out;//,data_C;
wire [31:0] m2_out,B_out,rst_S,mm2_out;
wire [7:0] A0_out,A1_out,A2_out,A3_out;
wire [31:0] rst_xor,rst_shft,rst_addsub;
wire [7:0] m3_out,m4_out,m5_out,m6_out;
wire [7:0] mult8;
wire [7:0] A8_out,A9_out,A10_out,A11_out,A12_out,A13_out,A14_out;
wire [7:0] regout,index1,index2,rom_table;
multi32 mym2(m2_out,data_in2,data_in3,sel_1);
multi32 mymm2(mm2_out,data_in3,data_in2,sel_1);
multi33 mym1(m1_out,1'b0,data_in1,data_in1,1'b0,mul_squ_sel);

RA mul1(carry,sum, mult8, m1_out, m2_out,Carr_out,mm2_out,px_sel,
A8_out,A9_out,A10_out,A11_out,A12_out,A13_out,A14_out);
Register_32 Reg_S(rst_S, sum,load_Sum,clk,rst);
data_cell33_clr Reg_Carry(Carr_out,carry,load_Carry,carry_clr,clk,rst);
multi7_8 mym3(m3_out, mult8,data_in1[31:24], rst_shft[31:24],
rst_xor[31:24], regout,data_in4[31:24],rom_table,sel_A),
```

```
mym4(m4_out, mult8,data_in1[23:16], rst_shft[23:16],
rst_xor[23:16], regout,data_in4[23:16],rom_table,sel_A),
mym5(m5_out, mult8,data_in1[15:8], rst_shft[15:8],
rst_xor[15:8], regout,data_in4[15:8], rom_table,sel_A),
mym6(m6_out, mult8,data_in1[7:0], rst_shft[7:0],
rst_xor[7:0], regout,data_in4[7:0], rom_table,sel_A);
Register_32 Reg_B(B_out, m2_out,load_B,clk,rst);
Register_8 Reg_A0(A0_out,m3_out,load_A0,clk,rst);
Register_8 Reg_A1(A1_out,m4_out,load_A1,clk,rst);
Register_8 Reg_A2(A2_out,m5_out,load_A2,clk,rst);
Register_8 Reg_A3(A3_out,m6_out,load_A3,clk,rst);
wire [7:0] reginput;
wire [31:0] En_sig;
wire [7:0] Reg_0_out,Reg_1_out,Reg_2_out,Reg_3_out,Reg_4_out,Reg_5_out,Reg_6_out,
Reg_7_out,Reg_8_out,Reg_9_out,Reg_10_out,Reg_11_out,Reg_12_out,Reg_13_out,
Reg_14_out,Reg_15_out;
wire [7:0] R0_out,R1_out,R2_out,R3_out;
wire [3:0] sel_regout;
multi5_8 mx1(reginput ,A0_out,A1_out,A2_out,A3_out, tmp,instr[7:5]);
demulti8 dmx(En_sig,instr[4:0],reg_en);
multi16_8 mx3(regout,Reg_0_out,Reg_1_out,Reg_2_out,Reg_3_out,
Reg_4_out,Reg_5_out,Reg_6_out,Reg_7_out,Reg_8_out,
Reg_9_out,Reg_10_out,Reg_11_out,Reg_12_out,
Reg_13_out,Reg_14_out,Reg_15_out,sel_regout);
// 16 8-bit register used for shiftrows
Register_8 Reg_0(Reg_0_out, reginput,En_sig[0],clk,rst);
Register_8 Reg_1(Reg_1_out, reginput,En_sig[1],clk,rst);
Register_8 Reg_2(Reg_2_out, reginput,En_sig[2],clk,rst);
```

```
Register_8 Reg_3(Reg_3_out, reginput,En_sig[3],clk,rst);
Register_8 Reg_4(Reg_4_out, reginput,En_sig[4],clk,rst);
Register_8 Reg_5(Reg_5_out, reginput,En_sig[5],clk,rst);
Register_8 Reg_6(Reg_6_out, reginput,En_sig[6],clk,rst);
Register_8 Reg_7(Reg_7_out, reginput,En_sig[7],clk,rst);
Register_8 Reg_8(Reg_8_out, reginput,En_sig[8],clk,rst);
Register_8 Reg_9(Reg_9_out, reginput,En_sig[9],clk,rst);
Register_8 Reg_10(Reg_10_out, reginput,En_sig[10],clk,rst);
Register_8 Reg_11(Reg_11_out, reginput,En_sig[11],clk,rst);
Register_8 Reg_12(Reg_12_out, reginput,En_sig[12],clk,rst);
Register_8 Reg_13(Reg_13_out, reginput,En_sig[13],clk,rst);
Register_8 Reg_14(Reg_14_out, reginput,En_sig[14],clk,rst);
Register_8 Reg_15(Reg_15_out, reginput,En_sig[15],clk,rst);
// used for dual-port RAM
Register_Inc_8 R0(R0_out, reginput, En_sig[16],inc_R0,clk,rst);
Register_Inc_8 R1(R1_out, reginput, En_sig[17],inc_R1,clk,rst);
Register_Inc_8 R2(R2_out, reginput, En_sig[18],inc_R2,clk,rst);
Register_Inc_8 R3(R3_out, reginput, En_sig[19],inc_R3,clk,rst);
// used for reduction Register_8 Reg_A8(A8_out, reginput,En_sig[20],clk,rst);
Register_8 Reg_A9(A9_out, reginput,En_sig[21],clk,rst);
Register_8 Reg_A10(A10_out, reginput,En_sig[22],clk,rst);
Register_8 Reg_A11(A11_out, reginput,En_sig[23],clk,rst);
Register_8 Reg_A12(A12_out, reginput,En_sig[24],clk,rst);
Register_8 Reg_A13(A13_out, reginput,En_sig[25],clk,rst);
Register_8 Reg_A14(A14_out, reginput,En_sig[26],clk,rst);
// used for counter
Register_8s Reg_counter1(index1, reginput,En_sig[27],inc_Index1,dec_Index1,clk,rst);
Register_8s Reg_counter2(index2, reginput,En_sig[28],inc_Index2,dec_Index2,clk,rst);
```

```
always @( index1 or index2 or instr[7:0]) begin
index1_flag = —index1; //
index2_flag = —index2;
cmp_flag1=index1¡=instr[7:0]?1'b1:1'b0; //comparator
cmp_flag2=index2¡=instr[7:0]?1'b1:1'b0;
end
   // address register
Register_8 addrR0(addra1,R0_out,load_addrR0,clk,rst);
Register_8 addrR1(addrb1,R1_out,load_addrR1,clk,rst);
Register_8 addrR2(addra2,R2_out,load_addrR2,clk,rst);
Register_8 addrR3(addrb2,R3_out,load_addrR3,clk,rst);
mod_addsub mas1(rst_addsub, Zflag,{A3_out,A2_out,A1_out,A0_out,B_out,
sub,first,load_Cbit,clk,rst);
mybarrel_shifter bshft1(rst_shft,{A3_out,A2_out,A1_out,A0_out},B_out[4:0],lr);
oper_XOR myxor(rst_xor, {A3_out,A2_out,A1_out,A0_out},B_out);
multi32_3 m7(in_RAM,rst_S, A3_out,A2_out,A1_out,A0_out,rst_addsub,sel_RAM);
rominfr rom1(rom_table, enc,reginput, read_en,clk); endmodule
```