Embedded Reconfigurable Solutions for Cryptography

by

Chi Chun (Ambrose) Chu
B.Engr. University of Victoria 2005

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering
February 23, 2008

Embedded Reconfigurable Solutions for Cryptography

by

Chi Chun (Ambrose) Chu
B.Engr. University of Victoria 2005

Supervisory Committee

Dr. Mihai Sima (Department of Electrical and Computer Engineering)
Supervisor

Dr. Amirali Baniasadi (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Florin Diacu (Department of Mathematics and Statistics)
Outside Member

Supervisory Committee

Dr. Mihai Sima (Department of Electrical and Computer Engineering)
Supervisor

Dr. Dr. Amirali Baniasadi (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Florin Diacu (Department of Mathematics and Statistics)
Outside Member

Abstract

We present a cryptography-oriented reconfigurable array called *CryptoRA* that efficiently supports very long-integer addition/subtraction and comparator unit. We first describe the *CryptoRA* architecture and show that extending the dedicated carry chains of modern FPGAs over the orthogonal direction, followed by merging two FPGA columns to create computing tiles that support both *generate* and *propagate* signals of a carry-lookahead network, provides a reduction in operation latency. Then, we show that splitting a tile's Look-Up Table into two halves provides additional benefits in terms of latency and flexibility in using the dedicated *generate* and *propagate* chains. The result of a hardware-software co-design on MicroBlaze embedded system for computing public-key cryptosystem algorithm is also presented and compared to pure software solution. According to our estimations, long-integer addition and comparison widely used in cryptography is more than 4x faster on *CryptoRA* than on Virtex-II Pro FPGA. Moreover, the area consumption is also reduced by approx. 30% in 1-level Carry-lookahead adder implementation and 75% in comparator unit. These improvements have a large positive impact on implementing cryptography applications in embedded environments, which can further improve the performance of the co-design solution.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

The first honor of course goes to my awesome supervisor, Dr. Sima, whom not only I have great time working for, but also have gained many tips from on how to approach and solve problems. He treats all his students with repect and always takes our suggestions under consideration. I could not ask for anything more from him with what he provides and could not be any happier working for a such supervisor. He also provides sufficient guidance along the way to ensure that my journey in completing master degree is smooth. Not only so, he is also one of the hottest topic that my girl and I always have conversion, about which we enjoy talking. He is one of the best supervisors one could have, at least to my opinion.

The second honor, without a doubt, goes to my dear grandmom, who fed me with the best cooking in the world, raised me with many hearts and cares, and basically, provided me with everything she has. Among all the greatest thing she has done to me, I am most thankful to one, and that is, providing me with a new life. I was once abandoned by doctors as a very pre-mature baby, but was not given up by my dear grandmom. With many sleepless nights and intensive cares, she twisted the story by turning me from a to-be-buried baby into a 100% healthy and cute one. I have many thanks to say to her and will take care of her more than anything else. Here, I would like to say Thank you, grandmom.

The third honor goes to the rest of family, which also includes Andra and her mom, and Robert. Without my family's finicial support and Robert, Andra and her mom's mental support, this journey would go much roughly than it does. Because of my parents' great vision, I flighted to here, Canada by myself in the age of 15, where I virtually started a new life without any of my family and friends around. It was not easy at all. However, I was fortunate enough to meet great people, like my former roomate, Robert, and my current

girl friend, Andra, who provide me with family-like care so that they all basically become like my second family here.

Last, but not least, this honor goes to everyone I meet along this jorney, specially the colleages in the lab, including Eugene, Scott, Ehsan, Farshad, Hamed, Kaveh, and more. These are great people to work with. We have fun in the lab as well as outside the lab. They are all very intelligent and knowledged people and are willing to share their knowledge with me. Hence, I have also learned many priceless stuff from them. Thank you, guys.

Acronyms

RSA Rivest-Shamir-Adleman

ECC Elliptic Curve Cryptography

ASIC Application-Specific Integrated Circuits

ASIP Application-Specific Instruction set Processors

RC Reconfigurable Computing

FPGA Field-Programmable Gate Arrays

CryptoRA Cryptography-oriented Reconfigurable Array

IF Integer Factorization

DL Discrete Logarithms

ECDL Elliptic Curve Discrete Logarithm

DSA Digital Signature Algorithm

PGP Pretty Good Privacy

SSL Secure Socket Layer

ECDLP Elliptic Curve Discrete Logarithm Problem

NIST National Institute of Standards and Technology

SECG Standards for Efficient Cryptography Group

ECDH Elliptic Curve Diffie Hellman

FSM Finite State Machine

MMM Montgomery Modular Multiplication

MMM unit Montgomery Modular Multiplier unit

LUT Look-Up Table

RCA Ripple-Carry Adder

CLA Carry-Lookahead Adder

CSA Carry-Save Adder

CSeA Carry-Select Adder

CSkA Carry-Skip Adder

XPS Xilinx Platform Studio

FSL Fast Simplex Link

OPB On-chip Peripheral Bus

LMB Local Memory Block

BRAM Block RAM

AES Advanced Encryption Standard

DES Data Encryption Standard

CLB Configurable Logic Block

FCCM Field-Programmable Custom Computing Machine

Chapter 1

Introduction

With the advent of Internet banking and other data-sensitive activities, it becomes increasingly important to send information securely over insecure channels. For the wireless applications of greatest interest, this requires that information encryption and decryption is performed in real-time on mobile terminals. The *de facto* cryptography algorithms are Rivest-Shamir-Adleman (RSA) [38] and Elliptic Curve Cryptography (ECC) [32, 34]. The key operations employed by these algorithms are not directly supported by typical integer-oriented architectures used in embedded systems, like ARM [41], MicroBlaze [48], MIPS [**?**], and NIOS [2]. Therefore, the issues associated with the algorithms used in public-key cryptosystems have drawn attentions to many embedded engineers.

A common feature of traditional cryptographic schemes is the need to operate on long-integer data, e.g., 160 to 521 bits for ECC and 1024 to 2048 bits for RSA [46]. While executing typical cryptography operations, such as modular multiplication or exponentiation, on long-integer data does not overburden a workstation, the performance of such operations may overwhelm an embedded processor, especially for applications in wireless, handheld devices, and smart cards with small memory capacity and strict latency constraints.

Cryptography applications are computationally intensive [39,44]. Thus, software-based implementations are inherently slow. For this reason, cryptography applications have been traditionally implemented in Application-Specific Integrated Circuits (ASIC) [40], or in hardwired-assists in Application-Specific Instruction set Processors (ASIP) [26]. Other solutions rely on coprocessors to accelerate long-integer arithmetic operations. Due to the ASIC and ASIP's hardwired-assist lack of flexibility, a different full-custom circuit is needed for each particular task. Also, even a slight improvement or change to an existing

device requires that the custom circuit be redesigned, which translates to a large engineering effort. With today's rapidly evolving standards and functional requirements, these fixed-function devices are prone to rapid obsolence.

On the other hand, the Reconfigurable Computing (RC) paradigm provides hardware-like performance with software-like flexibility [5, 21]. In RC, application-specific computing units are defined and then instantiated onto a reconfigurable array. This way, a large number of customized computing units are emulated. The commercial Field-Programmable Gate Arrays (FPGA) [2, 53] are general-purposed reconfigurable devices. As such, they exhibit a large silicon area and power consumption overheads to support a broad range of applications. To reduce this overhead, we propose a reconfigurable solution, in which a RISC embedded processor is augmented in conjunction with Cryptography-oriented Reconfigurable Array, called *CryptoRA*. With such configuration, the bottleneck operations are supported by the hardware unit in the proposed FPGA while other less computational demanding operations are still executed by the embedded processor.

1.1 Contributions

In this thesis, we focus on public-key algorithms with the designing steps and approaches we took during my research period. We first present the hardware-software co-design implementation, then demonstrate the various adder structures on FPGA platform, and most importantly, we introduce a Cryptography-oriented Reconfigurable Array, called *CryptoRA*. Cryptography-oriented Reconfigurable Array (CryptoRA) supports very long-integer addition and subtraction without incurring the overhead of generic reconfigurable devices and the lack of flexibility of domain-specific devices. We first show that extending the dedicated carry chains of state-of-the-art FPGAs over the orthogonal direction, followed by merging two FPGA columns to create computing tiles that support both *generate* and *propagate* signals of a carry-lookahead network, leads to shorter operation latency.

Then, we propose to split a tile's Look-Up Table into two halves. This provides additional benefits in terms of latency and flexibility in using the dedicated *generate* and *propagate* chains. We also show that it is possible to implement an ALU supporting addition and subtraction in carry-lookahead arithmetic. According to our estimations, long-integer addition is more than 4$\times$ faster on *CryptoRA* than on Virtex-II Pro FPGA. This translates to more than 20% latency improvement for Montgomery Modular Multiplication (MMM) implemented in CryptoRA reconfigurable hardware. The penalty of our approach is a slightly larger silicon area. The paper contributions are as follows.

- Efficient Mapping techniques of various long-integer adder structures, and comparator unit onto FPGA platform.
- Description of the architecture of *CryptoRA*, with major features comprising of (i) an increased granularity of the logic tile, and (ii) the extention of the dedicated carry chain of standard FPGAs over the orthogonal direction.
- Transistor-level implementation of the logic tile, with major features comprising of (i) splitting the Look-Up Table (LUT) of standard FPGAs such that the dedicated carry chain is driven by a faster LUT output, and (ii) emulating extra functionality within the split LUT.
- Demonstration on the comparability of my C-level program for computing public-key cryptosystem algorithms to a well-known library. Description of steps on the hardware-software co-design implementation and illustration of its improvement.
- Estimation of the further improvement in terms of computing time for Montgomery modular multiplication.

1.2 Thesis Overview

In the second chapter, we cover briefly what cryptography standards are, but mainly focus on the public-key cryptography since that is the cryptographic class that uses algorithms which requires computational intensive operations, such as long-and-very-long integer modular operations. Within this class, we look into RSA and ECC algorithms; these are the representative algorithms that show all the operations needed in the public-key cryptosystem. Thus, we present the basis operations for these algorithms and also address their pros and cons.

In Chapter 3, we give a general overview on the computing machine design using different state-of-the-art technologies. One of which consists of a general-purpose RISC embeddded processor and a FPGA, called Field-Programmable Custom Computing Machine (FCCM). This solution is indeed what we use on public-key cryptography computation because we still like to have the flexibility of reconfiguring computing unit on-the-fly while accelerting the performance. Thus, we also review a number of papers that utilize this type of computing solutions for public-key cryptography application. Moreover, the architecture support for fast-adder addition from both Xilinx and Altera is discussed since our proposal Cryptography-oriented Reconfigurable Array (CryptoRA) is heavily based on fast-adder structure from Xilinx.

In Chapter 4, we present how the software implementation is built using the bottom up fashion level by level in order to have an efficient working framework for computing EC point multiplication. This includes introduction to the hardware platform configuration for MicroBlaze embedded system on which the C-level software program is download and ran. This also comprises software implementation for each level of operations, including modular operation level, EC point operations level, and EC point multiplication level. A description on data communication between the core processor and the hardware unit is also presented.

Chapter 5 is the main chapter in this thesis. It contains mapping techniques for several adder and fast-adder structures on Xilinx Virtex-II Pro FPGA, including Ripple-Carry Adder (RCA), Carry-Skip Adder (CSkA), Carry-Save Adder (CSA), Carry-Lookahead Adder (CLA), and a comparator unit. This is because that long-and-very-long integer addtion/subtraction and comparison become the core operation when Montgomery Modular Multiplication (MMM) algorithm is implemented in bit-level structure. After observing those implementation on Xilinx FPGA, some issues are raised, mainly global interconnect delays and more area consumption. Therefore, Cryptography-oriented Reconfigurable Array (CryptoRA) is proposed to alleviate these issues and its new features are introduced and described in great details. CLA and new comparator unit structure are able to take the advantage of CryptoRA and are emulated using CryptoRA. Descriptions on these mapping process are given in this chapter.

Chapter 6 is the showcase chapter that is used to demonstarte the simulated and estimated results of the implementation. It reaveal the comparability of our C-level software program to assembly-optimized library, such as MIRCL by comparing the performance in cycle count. The speedup for the computing time of the MMM from the hardware support is from 37$\times$ to 45$\times$ for bit length ranging from 160 to 224. This in turns allows a speedup from 15$\times$ to 22$\times$ for EC point multiplication. Due to the fact that by utilizing CryptoRA, instead of commercial FPGA, the overall critical path is reduced, resulting higher hardware clock frequency, the cycle count spent in the MMM unit is further reduced by XX%.

Chapter 7 concludes the thesis summarizing our findings, discussing the main contributations, and suggesting open area for future work.

Chapter 2

Cryptography Standards

Cryptography covers wide range of algorithms and theories to be used in many different applications and therefore, it is very difficult to explains everything in details. However, the cryptography basis is covered in this chapter and is organized as follows. The basic cryptography concept is introduced in Section 2.1. A number of the families of public-key algorithms are presented in Section 2.2, in particular, the Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC) families. This chapter is completed with some comparisons between these two public-key alogrithms.

2.1 Cryptography Introduction

Cryptography is the science of information security that utilizes mathematic algorithm to protect data transmitted in open communication networks, such as Internet. The four main purposes that it serves are confidentiality, integrity, authentication, and non-repudiation [46], shown in Table 2.1.

There are two major classes of cryptographic systems. The first class is called private-key cryptosystem, and includes as representative members Advanced Encryption Standard (AES) and Data Encryption Standard (DES). These algorithms use a single key, which both correspondents must know. They must keep it secret from a third correspondent, otherwise this third correspondent will be able to decrypt any messages encrypted using that key. The second class is called public-key cryptosystem and was first publicly suggested by Diffie and Hellman in their paper [13]. It includes as representative members RSA [38] and ECC [32, 34]. In the public-key systems both correspondents have a key pair, not just a single key. Every pair consists of a public key and a private key. The public key is used

to encrypt the message, so that anyone can encrypt. The message can be decrypted using only the private key, so only the owner can decrypt the message.

One of the main motivation behind the creation of public-key cryptosystem is to have a flexible mechanism that can be used to replace the key distribution and key management, which are expensive in terms of the hardware facility. In Table 2.1, it summarizes the basis of the cryptography standards.

Table 2.1: Comparisons between private-key and public-key cryptosystems

	Definition	Private-key Crypto	Public-key Crypto
Popular Algorithms		AES, Triple DES	RSA, DL, ECC
Advantages		Computationally fast	less overheads on key establishment
Disadvantages		More overheads on key establishment	Computationally slow
Confidentiality	Keep the data secret from other unitended receivers	X	X
Integrity (hash)	Keep the data unaltered	X	X
Authentication	Be certain where the data came from		X
Non-repudiation	Digital signature		X

It is noticed that not only the public-key cryptosystem serves more services than the secret-key cryptosystem, but also it resolves the main issues in the secret-key cryptosystem, which are the key distribution, and key management [20]. However, the computational

requirements of the private-key cryptography are much lower than those of the public-key cryptography. Therefore, both cryptosystems are often found to facilitate in conjunction in the cryptography protocols - public-key cryptosystem is used to exchange/establish the common key secretly between two parties, who later utilize that common key in the process of encrypting and decrypting the actual message using the secret-key cryptosystem.

Furthermore, the three basic types of cryptographic functions provided by the algorithms in public-key cryptography, standardized by the IEEE P1363 [23] are key agreement, digital signatures, and public key encryption. Due to the needs and the frequent usages of these public-key algorithms and because of the much slower in their computation, it becomes the motivation and the goal to find ways to speed up the performance in these public-key algorithms. Thus, only public-key cryptosystem is further considered in this thesis.

2.2 Public-Key Cryptosystem

The algorithms used in the public-key cryptosystem are classified based on the hard number theory problems upon which they are based and from which they derive their security. The three most common theory problems that define the the families of public-key algorithms are Integer Factorization (IF), Discrete Logarithms (DL), and Elliptic Curve Discrete Logarithm (ECDL). For the completeness reason, the definition for each number theory problems are given below [46].

1. IF problem:

   Given a positive integer n, finding its prime factorization is very difficult; that is, write $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$ where p_i are pairwise distinct primes and each $e_i > 0$.

2. DL problem:

   Fix a prime p. Let α and β be nonzero integers mod p and suppose

$\beta \equiv \alpha^x \pmod{p}$.

The problem of finding x is very difficult such that

$x = L_\alpha(\beta)$

3. ECDL problem:

   Suppose we have point Q, P on an elliptic curve E and we know that $Q = kP (= P + P + \ldots + P)$ for some integer k. The problem of finding k is very difficult.

Table 2.2 lists some of the most common protocols from each family for the public-key cryptographic functions. For a particular cryptographic functions, one protocol might be a better choice than the other, and therefore is used more frequent than the others; for example, Digital Signature Algorithm (DSA) from the DL family is used more often than the RSA from the IF family in the Digital Signature protocol.

Table 2.2: List of the most common Public-key protocols from each family

Key Agreement	EC Diffie-Hellman (ECDH), DH, and RSA
Digital Signature	EC Digital Signature Algorithm (ECDSA), DSA, and RSA
Public-key Encryption	EC Integrated Encryption Scheme (ECIES), ElGamal, and RSA

From the underlying operation point of view, all the public-key algorithms have something in common —that is, their underlying operations are modular operations. Because the core modular opeartion in algorithms from both the IF and the DL families is modualr exponentiation, it makes sense to just illustrate algorithms from one of these families. Due to the much more complex in desribing DL-family algorithms, algorithms from IF family is chosen. Thus, the focus of this thesis will again narrow down to the public-key algorithms from only the ECDL and the IF families. We will look particluarly at the RSA and ECC algorithms.

2.2.1 RSA Algorithm

RSA algorithm is named after Ron Rivest, Adi Shamir, and Leonard Adleman [38]. It is a commonly used cryptography algorithm that uses keys with the bit length ranging from 512 to 2048 bits, depending on the level of security that one desires. Because RSA is one of the early public-key algorithms used in place, it has been adpated widely in many applications, such as Pretty Good Privacy (PGP), a popular method for encrypting email, and in Secure Socket Layer (SSL) [45]. Essentially, RSA is based on two distinct odd prime numbers p and q, which are used to generate two so-called key-pair values: a public key-pair $\{e,n\}$, and a private key-pair $\{d,n\}$. Normally, the $\{e,n\}$ key-pair is used to encrypt data, while the $\{d,n\}$ key-pair is used to decrypt data. Assuming the string to be encrypted includes a block of data, $m < n$, and the encrypted string includes a block of data, c, the RSA algorithm can be described as follows.

Algorithm 2.1 RSA Algorithm [46]

1: Bob generates two odd primes p and q, and computes $n = pq$.
2: Bob computes e with $gcd(e,(p-1)(q-1)) = 1$.
3: Bob computes d with $de \equiv 1 \bmod ((p-1)(q-1))$.
4: Bob makes e and n public, and keeps p,q,d secret.
5: Alice encrypts m as $c \equiv m^e \pmod{n}$ and sends c to Bob.
6: Bob decrypts by computing $m \equiv c^d \pmod{n}$.

In Algorithm 2.1, m and c are unsigned integers with values less than n. If m is larger than n, then she breaks the message into blocks, each being less than n. The values of e and d can be ranged from $\{1,n-1\}$. However, A popular choice for e is $65537 = 2^{16}+1$ because it can be computed quite quickly. On the other hand, the decryption exponent d should be chosen large enough that brute force will not find it. Since these are extremely large numbers, the exponentiation operations m^e and c^d cannot be computed directly as it might possibly overflow the memory space. Fortunately, because of the modular operation

property, modular exponentiation can be computed by the recursive routine presented in Algorithm 2.2 using square-and-multiply technique[xx], where n is the wordlength, $e(i)$ denotes the bit i of E, P_i is the value of P at iteration i, and N is the modulus value. That is, modular exponentiation is reduced to a series of modular multiplication and square operations.

Algorithm 2.2 Modular exponentiation by square-and-multiply

Ensure: $P = X^E \bmod N$, where $E = \sum_{i=0}^{n-1} e(i)2^i$, $e(i) \in \{0,1\}$

$P_0 = 1$, $Z_0 = X$

for $i = 0$ to $n-1$ **do**

$Z_{i+1} = Z_i^2 \bmod N$

if $e(i) = 1$ **then**

$P_{i+1} = P_i \cdot Z_i \bmod N$

end if

end for

2.2.2 ECC Algorithm

Elliptic Curve Cryptography (ECC), on the other hand, is a relatively new public-key algorithm. It is invented in 1987 by Neal Koblitz [34] and Victor Miller [32]. ECC becomes an attractive alternative solution for the next generation public-key cryptosystem [45] due to the same level of security that it can offer with smaller key size requirement compared to other public-key cryptosystems. Furthermore, not every elliptic curve offers strong security properties—for some curves, the Elliptic Curve Discrete Logarithm Problem (ECDLP) may be solved efficiently [43]; therefore, poor choice of the curve can compromise security. This is why National Institute of Standards and Technology (NIST) and Standards for Efficient Cryptography Group (SECG) have published a set of curves [16, 17] that possess the necessary security property [19].

In order to illustrate how heavily the performance of ECC is depended on the underlying modular operations, it is essential to describe briefly how ECC works. Assume a set of points having the property that they belong to an Elliptic Curve and let and $P(x_p, y_p)$ and $Q(x_Q, y_Q)$ such points in the set. The idea behind the ECC's security is that it is very difficult to find an large positive integer k, such that

$$Q = kP, \tag{2.1}$$

where kP is not a normal multiplication; it is a so called a Scalar (Point) Multiplication.

In Equation 2.1, k is a large random integer that is normally at least 160 bits long acting as a private key, while the result of multiplying the private k with the point P on the curve serves as the corresponding public key. The Scalar (Point) Multiplication is the main ECC operation that operates over a group of points on the elliptic curve defined over a finite field. Furthermore, a scalar multiplication is a combination of EC point addition and EC point doubling operations, as illustrated in the basic Double-and-Add Algorithm below. In the example of $11P$, it can be extracted as $((((2P)2)+P)2)+P$, which consists of 3 EC point doubling and 2 EC point addition if the Algorithm 2.3 is used.

Algorithm 2.3 Double-and Add Algorithm for EC Point Multiplication

Require: EC point $P = (x, y)$, integer $k, 0 < k < p, k = (k_{l-1}, k_{l-2}, \ldots, k_0)_2$

Ensure: $Q = k \cdot P$

1: $Q \leftarrow P$
2: **for** i from $l-2$ downto 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**

Furthermore, ECC is typically defined over two types of fields: binary and prime. The

operations over the binary field can be simply implemented since the long arithmetic addition is essentially a bit-wise *XOR* operation. Thus, the binary field operation does not pose significant computational requirements. Thus, ECC over binary field is not considered any further and only ECC over prime field is presented, whose underlying modular operaqtions are modolus of prime numbers. In other words, the same modular operations can be used in both RSA and ECC over prime field. ECC over the prime field can only be explained algebraically but not geometrically; this is because it contains discrete points that satisfy the defined equation with different parameters and that it is not clear how to be connected together to make their graph look like a curve. As a result, the algebraic rules for the arithmetic can be adapted from the elliptic curve over real numbers to that over both prime and binary fields. Thus, to show how the equations for computing points on a elliptic curve over prime field are derived geometrically, the elliptic curves over the real number field are used.

The general equation for elliptic curve E given a field F in affine coordinate is shown below and is called Weierstrass Equation.

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.2}$$

where $\{x,y\} \in F$ and $\{a_1,a_2,a_3,a_4,a_6\} \in F$.

The above equation can then be simplified to The Equation 2.2 below for elliptic curve over real number and prime fields.

$$E : y^2 = x^3 + ax + b, \tag{2.3}$$

where x,y,a,b are real numbers in the real number field and are unsigned integers in the prime field.

Different choices for the number a and b yield different elliptic curves. In addition, if $x^3 + ax + b$ contains no repeated factors, or equivalently if $4a^3 + 27b^2$ is not 0, then the elliptic curve in Equation 2.3 can be used to form a group. An elliptic curve group over real

numbers consists of the points on the corresponding elliptic curve, together with a special point O called point at infinity [8].

Figure 2.1 is used to illustrate how the EC point addition equations are derived geometrically. A line which is drawn through two distance points, P and Q are added will intersect the third point, called $-R$, the inverse of $R = P + Q$, on the elliptic curve. To get the point R from its inverse point $-R$ is to simply flip $-R$ alone the x-axis. This EC point addition operation is shown in Figure 2.1(a) assuming $a = -7$ and $b = 0$. One contraint on this operation is that P can not be $-Q$; otherwise, it will become $Q + (-Q) = O$ by definition, which is also depicted in 2.1(b) assuming $a = -6$ and $b = 6$. This O is not on the curve; however, all elliptic curves have this O, called additive identity (point at infinity) [8].

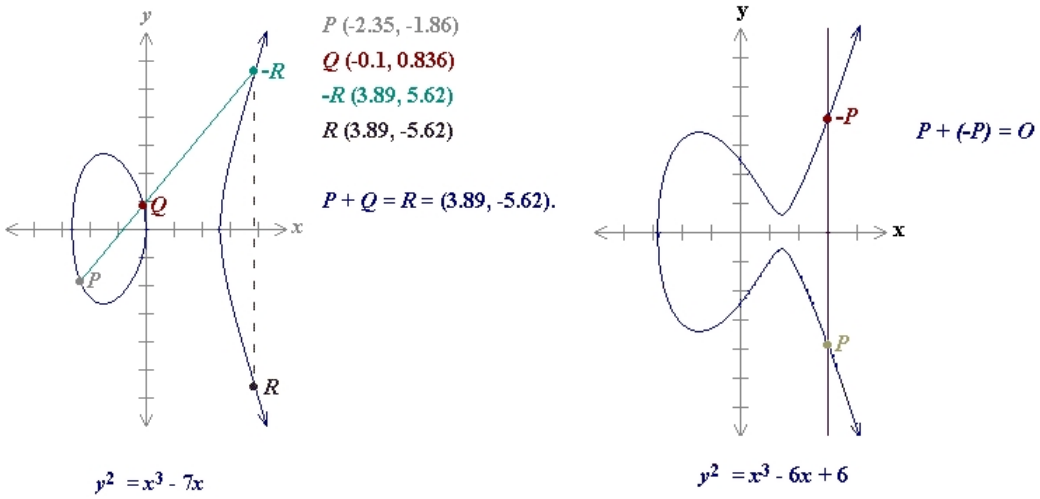


Figure 2.1: A Point Addition example over Real Number

A similar procedure is performed for EC point doubling operation in the point operation level – a line that is drawn through point P will intersect one point, called $-R$, the inverse of point $R = P + P$ on the curve. Again, to get the point R from its inverse point $-R$ is to simply flip $-R$ along the x-axis. This point doubling procedure is illustrated in Figure 2.2(a) assuming $a = -3$ and $b = 5$. One constraint on this operation is that if point $P(x_P, y_P)$and

y_P is on the x-axis, meaning $y_P = 0$, then the tangent line through P is vertical and will not intersect at any point on the curve. By definition, $2P = O$ for such a point P [8]. This phenomenon is illustrated in Figure 2.2(b) assuming $a = 5$ and $b = -7$.

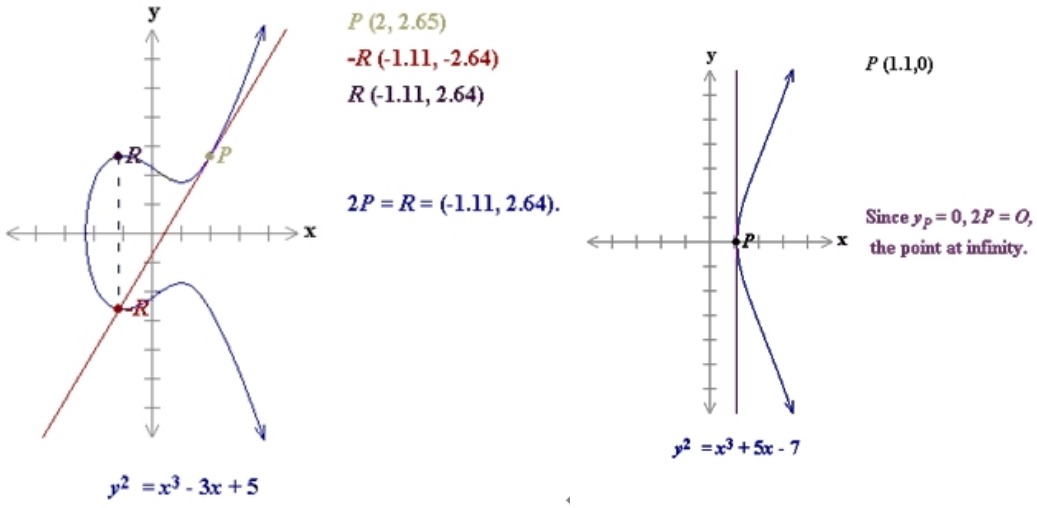


Figure 2.2: A Point Doubling example over Real Number

2.2.3 ECC over prime field

Table 2.3 shows the EC point addition and point doubling algebraically in the Affine coordinate, where one inversion exists. Different projective coordinates for computing point addition and doubling were proposed due to the slow computation in modular inversion. The use of these projective coordinates comes with the penalty of more numbers of modular multiplications and modular sqauring being generating, and requires more temporary variables as the result of eliminating the modular inversion operation. In sum, if the implementation of modular inversion is fast, meaning $T_{invert} < 4 \cdot T_{mult} + 6 \cdot T_{square}$, then affine point doubling is faster. And similarly, if $T_{invert} < 11 \cdot T_{mult} + 3 \cdot T_{square}$, then affine point addition is faster.

Table 2.4 depicts what operations, and the number of each operations executed in dif-

Table 2.3: Point Operations in Affine Coordinates

Point Addition	Point Doubling
Given:$P=(x_P,y_P)$, $Q=(x_Q,y_Q)$ and they are not negative to each other, and prime p	Given:$P=(x_P,y_P)$, $y_P \neq 0$, and prime p
Output: $P+Q=R$, where $R=(x_R,y_R)$	Output: $2P=R$, where $R=(x_R,y_R)$
$s=\left(\frac{y_P-y_Q}{x_P-x_Q}\right) \pmod p$, where s is the slope of the line through P and Q	$s=\left(\frac{3y_p^2+a}{2y_P}\right) \pmod p$, where s is the slope of the line through P and Q
$x_R=s^2-x_p-x_Q \pmod p$	$x_R=s^2-2x_p \pmod p$
$y_R=s(x_P-x_R)-y_P \pmod p$	$y_R=s(x_P-x_R)-y_P \pmod p$

ferent coordinates for both EC point doubling and point addition. For instance, the number of modular operations required in affine coordinate corresponds to the first entrance in first and second column. It is noticed that modular addition and modular subtraction are neglected because the time it takes to execute these two operations are insignificant when compared with modular multiplication and squaring. According to Table 2.4 [20], Jacobin projective is often used for point doubling, and mixed Jacobin and Affine coordinates is utilized for point addition because they yield the least numbers of operations.

Table 2.4: Operation counts for one point addition and one point doubling over $GF(p)$

Point Doubling		Point Addition		Mixed Coordinates	
$2A \rightarrow A$	$2M+2S+I$	$A+A \rightarrow A$	$2M+S+I$	$J+A \rightarrow J$	$8M+3S$
$2P \rightarrow P$	$7M+5S$	$P+P \rightarrow P$	$12M+2S$	$J+C \rightarrow J$	$11M+3S$
$2J \rightarrow J$	$4M+6S$	$J+J \rightarrow J$	$12M+4S$	$C+A \rightarrow C$	$8M+3S$
$2C \rightarrow C$	$5M+6S$	$C+C \rightarrow C$	$11M+3S$		

A-Affine, P- Standard Projective, J-Jacobin projective, C-Chudnovsky projective

M- Multiplication, S- Squaring, I- Inverse

To furhter illustrate how the EC point multiplication works in the application level, Figure 2.3 is provided as an example of a common-key establishment using Elliptic Curve Diffie Hellman (ECDH) protocol, a analogue to the popular Diffie Hellman key exchanged protocol from the DL family. This simplified protocol depicts that by computing 2 EC point multiplications in each party, a common secret key is generated, which can later used as the key for large-size encryption.

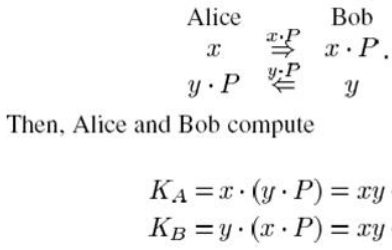


Figure 2.3: ECDH protocol example

In Table 2.5 [19], it shows the fact that ECC can provide the same level of security with much smaller key length than that in RSA, e.g., 224-bit ECC has the same security level as 2048-bit RSA. Due to the shorter key length used in ECC, less bandwidth, memory, and computing power is needed. This is the reason why ECC is a preferable option for public-key cryptosystem in embedded systems.

Table 2.5: Equivalent key sizes

ECC (in bits)	RSA (in bits)	MIPS Yrs to attack	Protection lifetime
160	1024	10^{12}	until 2010
224	2048	10^{24}	until 2030
256	3072	10^{28}	beyond 2031
384	8192	10^{47}	infinity at current technology
521	15360	10^{66}	infinity at current technology

Moreover, it also presents the predicting valid protection for different key size of ECC

and RSA. The growth in key length becomes even more the issue for RSA as the higher security level is needed for the future protection. However, it is eqaully important to support RSA cryptosystem as ECC cryptosystem since it is currently the most widely used public-key algorithms in many applications.

A Hierarchy of operations in RSA and ECC algorithms is shown in Figure 2.4. As noticed, the underlying finite field arithmetic are the same in ECC over prime field and RSA cryptosystems, regardless the difference in key length and modulus value. In other words, the same software routine or accelerated hardware unit can be shared in all public-key cryptosystems.

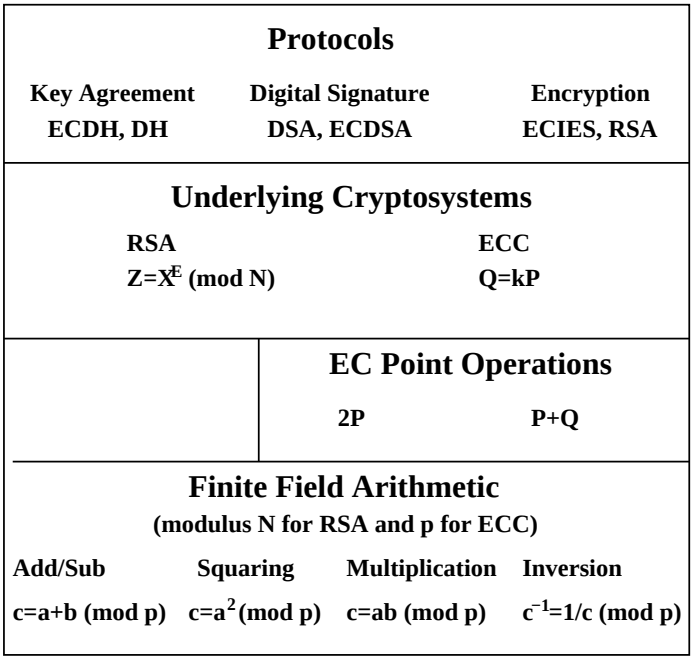


Figure 2.4: Hierarchy of operations in RSA and ECC schemes

It is also worth mentioning that there are different optimizations can be applied at each level of the hierarchy of the operations in public-key cryptography schemes to speed up the overall performance. Nevertheless, in our research, we mainly look at the possible optimizations on the underlying finite field arithmetic level. Furthermore, in our particluar

implementation, the modular inversion is computed based on the modular multiplication. A brute force solution, meaning computing regular multiplication and then modular operations of two long integers is extremely slow and maybe cause memory overflow. Thus, a high-performance modular multiplier is needed and Montgomery Modular Multiplication is used for such task as it is proved to be a very efficient algorithm. A number of results have been reported in the literature regarding its FPGA implementation. They will be outlined in the next chapter.

Chapter 3

State-of-the-art in Reconfigurable Solution for Cryptography

Since there are many state-of-the-art technologies out there that we can use for our implementation, this chapter starts out with brief review on the some of the options in the computing mahcine design. It is followed by the related works that have been done or currently ongoing using any type of reconfigurable computing solution on the public-key cryptography application. As the Field-Programmable Gate Arrays (FPGA) is the main platform that is used in our implementation, one particular FPGA from each of the two largest FPGA-chip maker, Xilinx and Altera are examed - Virtex-II Pro FPGA from Xilinx and Stratix FPGA from Altera. The architectural support for fast-adder structure from these FPGA chips are particularly to our interest. Finally, this chapter is completed with a brief summarization.

3.1 Computing Paradigm Review

In the design of computing machine, it is well-known that *General Purpose Processor* (GPP) provides the greatest flexibility at expense of performance while ASIC provides the greatest performance at expense of flxibility. Somewhere in the middle, there is the FPGA, which provides medium flexibility and performance when compared to the early-mentioned types of computing machines. FPGA is often used in conjunction with GPP to provide so-called RC [27], [28] an emerging computing paradigm for more than twenty years. Such a hybrid is referred to as a FCCM [5]. It works by defining custom computing resources on a per-application basis, and dynamically configuring them onto an FPGA so that a large number of application-geared computing unit can be emulated. Since FCCM provides RC

solution with hardware-like performance and software-like flexibility, this is the paradigm in which we focus and propose the solution to the public-key cryptography computation.

The work on reconfigurable cryptography has been focused in two main directions. The first direction involves the mapping of long-integer operations on general-purpose (commercial) FPGAs. The second direction that tries to go even faster. It entails the design of cryptography-oriented reconfigurable devices. The performance and limitations of these designs are presented next.

3.2 Related Work on Public-Key Cryptography

As the layers of operations exist in public-key cryptography algorithms, which is described in Chapter 2, it is up to the system designers to decide what to be supported in the hardware and what not. In the case where the entire RSA or ECC algorithms is augmented in hardware to offload the workload in the processor, there are two main methodologies of implementing the hardware co-processor in the literature. One is Finite State Machine (FSM), which can be found in [3], [6], [37] and [35]. Another is through definition of its own intstruction set, which can be found in [15], [18]. There are also papers that authors proposed hardware unit for computing only modular multiplicaton [14], [4], and [29]. In all cases, the modular multiplier hardware unit is implemented using the Montgomery Modular Multiplier unit (MMM unit), which can be implemented differently based on the criteria, such as the flexiblity, platform and area availabilities, performance ones desire to achieve, and target application. Furthermore, there are papers suggesting computing modular multiplication for only National Institute of Standards and Technology (NIST) recommended primes, which provides the fastest method, called fast reduction for computing modular multiplication in the literature. However, this hardware modular multiplier unit can not be used in RSA algorithms since RSA does not use those NIST recommended primes. Also, such hardware unit can not be used in EC point multiplicaiton over prime

field, in which the primes are not those NIST recommended primes. Therefore, due to the limitation on the fast reduction implementation, we consider to utilize another efficient algorithm called Montgomery Modular Multiplication (MMM) to compute modular multiplication operation.

Bit-level (radix-2) MMM unit in a pipelined structure can be found in [14], which utilizes processing unit (PU) as the computing tile, and in [4] using the systolic array. It can also be implemented using Carry-save adder structure, which can be found in [6], [18]. On the other hand, block-level (high-radix) MMM unit are implemented using dedicated 16×16 multipliers in FPGA [29] and $64X \times 64$ user-defined multiplier in ASIC [15]. The advantage in block-level MMM unit implementation is that the iteration of *loop* (refer to Algorithm 4.1) is reduced with the increase in block size. However, the algorithm become more complex and requires more silicon area. On the other hand, bit-level MMM unit implementation requires more iteration than block-level implmentation, but is much easy to be implemented and results less area consumption. However, in terms of the performance, there is no implication saying that the block-level MMM unit would perform better than the bit-level MMM unit. It would depend on one's implementation and platform.

According to our simulations on Amirix AP1000 FPGA Development Board [7] using an XC2VP100 Virtex-II Pro FPGA [51], and the software tool Xilinx ISE (Project Navigator) v9.1.03i [53], propagation through global interconnect takes at least 0.8 ns, while a LUT latency is $0.313ns$. This is contrast to the propagation through the dedicated carry chain that takes 0.04 ns per tile. This means that roughly 32-bit ripple-carry addition is as fast as 3-to-2 carry-save addition. This results is consistent witht he Xilinx figures: 64-bit addition has the latency of $10^3/114 = 8.8$ ns, while 16-bit addition has the latency of $10^3/1239 = 4.2$ ns. 8-bit addition has the latency of $10^3/292 = 3.4$ ns. This means fast adder structures show no improvement versus ripple-carry structure for 16-bit addition or less, and therefore they have to be considered only for long and very long-integer addition.

To reduce the addition latency in FPGA, Hauck *et al.* analyzed the impact of deploying

dedicated resources for a truly parallele carry-lookahead network as well as a complex carry-select network that comprises a dedicated carry-select multiplexor for each bit [22]. As mentioned, the carry-lookahead network suffers from increasing circuit complexity and fan-out toward high-order bits. This leads to a triangular layout. The carry-select network exhibits the same behavior. Therefore, both networks suffer from irregularity. In effect, the beginning of the carry chain is not tile independent. This is highly undesirable in an FPGA, since a circuit should ideally be mappable to any FPGA region.

Furthermore, Goodman and Chandrakasan [18] proposed a domain-specific reconfigurable cryptographic processor (DSRCP) in ASIC. They claimed that their implementation is reconfigurable even though it is in ASIC platform. This reconfigurability is essentially accomplished by adding multiplexers that are used to select the corresponding input to the output for performing the particular operation. Also, in order to support any arbitrary key size and to reduce the power consumption, the reconfigurable logic can be shutdown through the *SETLENGTH* instruction in 32-bit increment. The microcode is implemented for complex instructions (e.g., point addition and doubling) while reconfigurable hardware units are implemented for simple instructions (e.g., Montgomery modular multiplication). This reconfigurable array is very coarse-grain, in the sense that the array supports only three configurations. From this point of view, it cannot be classified as being truly reconfigurable. However, we are still interested in their hardware units, which include the reconfigurable logic, adder, and comparator unit. We use their designs of these hardware unit and implement them in Xilinx FPGA to build the Montgomery Modular Multiplication (MMM) hardware unit. Since these designs are originally implemented in ASIC platform, and may not be suitable in FPGA, further adjustments (e.g, on equations of the comparator unit) and proposals on new cryptography-oriented FPGA were made, resulting better mapping and performance in computing public-key cryptography operations.

As the proposed Cryptography-oriented Reconfigurable Array (CryptoRA) is based on the Xilinx FPGA Virtex-II Pro, in particular, the dedicated carry chain feature it provides,

we present Xilinx adder structure in the next section. We also present the Altera's FPGA support for addition in the following section.

3.3 Xilinx's adder structure

Ripple-carry addition is given architectural support in the form of a dedicated carry path in most mature FPGA families, such as, XC4000 from Xilinx [53] or FLEX 10K from Altera [2]. Since building carry-lookahead or carry-select networks requires the corresponding generate/propagate or select signals, respectively, go through the slow global interconnect, ripple-carry adder is generally preferred on these FPGAs. Ripple-carry addition support from Xilinx is rather simple and is preferred for addition of two operands in the range of tenth bits (e.g., 32-bit bits).

To quickly show how the dedicated carry chain is used to provide the support for ripple-carry addition, given the Equation 5.6, the Figure XX is used. While one operand signal is fed to the 0-input of the dedicated carry MUX, the carry signal from the previous position $(i-1)$ is fed to the 1-input. The output of the first *XOR* gate is used as the select signal for the MUX, as well as to generate the sum signal by *XORed* the in-coming carry (c_{in}). In order to generate the out-going carry (c_{out}) signal, it is little complex to explain. From the logic perspective, the carry is only generated if at least two of the three operands (x, y, c_{in}) are 1. Thus, having such configuration as shown in Figure XX does gurantee to have such function. This is in deed the resulting mapping on Xilinx FPGA when the + operator is used in HDL.

In modern FPGAs, fast-adder structures are also given architectural support on top of the ripple-carry addition support. This is needed for addition of long integers in the application, such as cryptography. The Virtex-II family from Xilinx provides dedicated hardware for a carry-lookahead network [52]. It is apparent in Equation 5.4 that the complexity of a carry-lookahead network increases toward high-order bits. The hardware resources of a

reconfigurable array are uniformly distributed across the die, such that a computing tile is replicated many times to generate an array of tiles. Therefore, due to the device uniformity, the Xilinx' carry-lookahead signals are emulated serially, along dedicated chains, as shown in Figure 3.1.

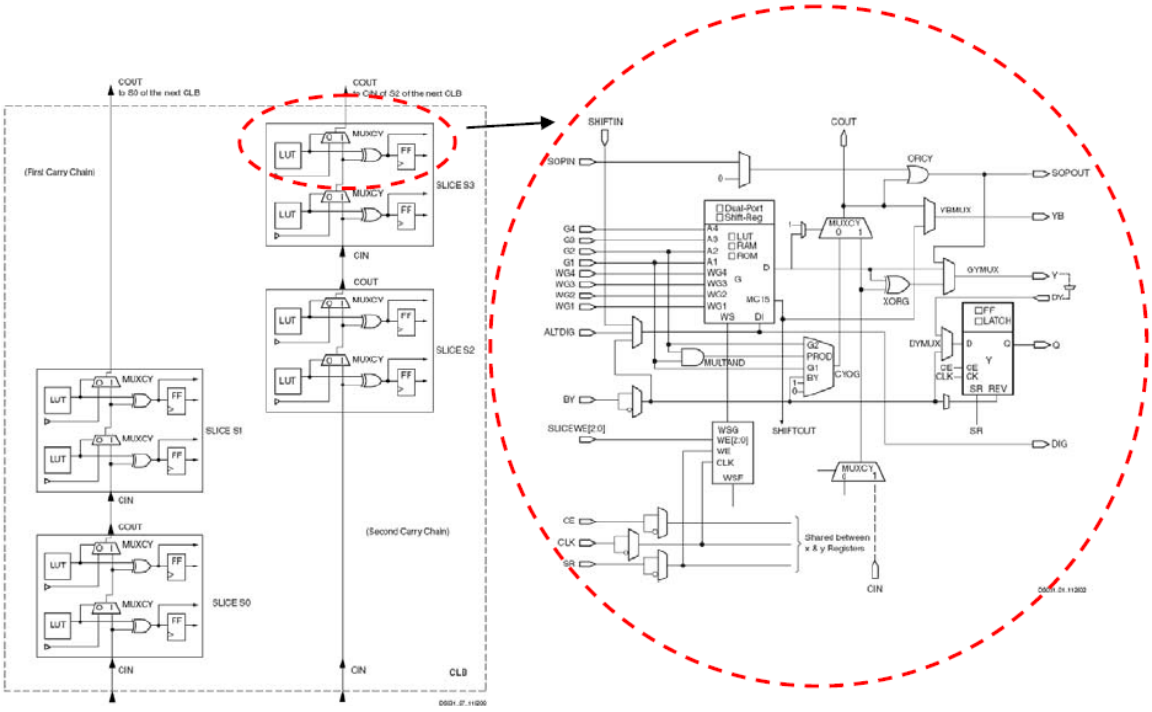


Figure 3.1: Fast carry chain in Xilinx Virtex-II Pro FPGA

As seen, a simplified Configurable Logic Block (CLB) shows the hardware support for the RCA through dedicated carry Muxes in the carry chain and the sum bit is generated by the dedicated *XOR* gate. A closer look at the half of the slice in CLB shows more details in the slice. To be able to support CLA, mainly block-level *generate* and *propagate* signals, the dedicated *AND* gate, named *MULTAND* is used. Details on mapping CLA onto Xilinx FPGA is presented in Chapter 5. A different fast adder architecture supported from Altera is presented in the next section.

3.4 Altera's adder structure

Altera has different architecture support for fast-adder. For instance, the Stratix family from Altera provides architectural support for carry-select addition [1]. Carry-select adder works as the follows. Basically, the adder is split into groups. For each group, two ripple-carry additions are performed in parallel assuming group-level incoming carry bits of 0 and 1. Then based on the correct incoming carry bit, the appropriate result is selected by means of a dedicated multiplexor. The major drawback of Carry-Select Adder (CSeA) is

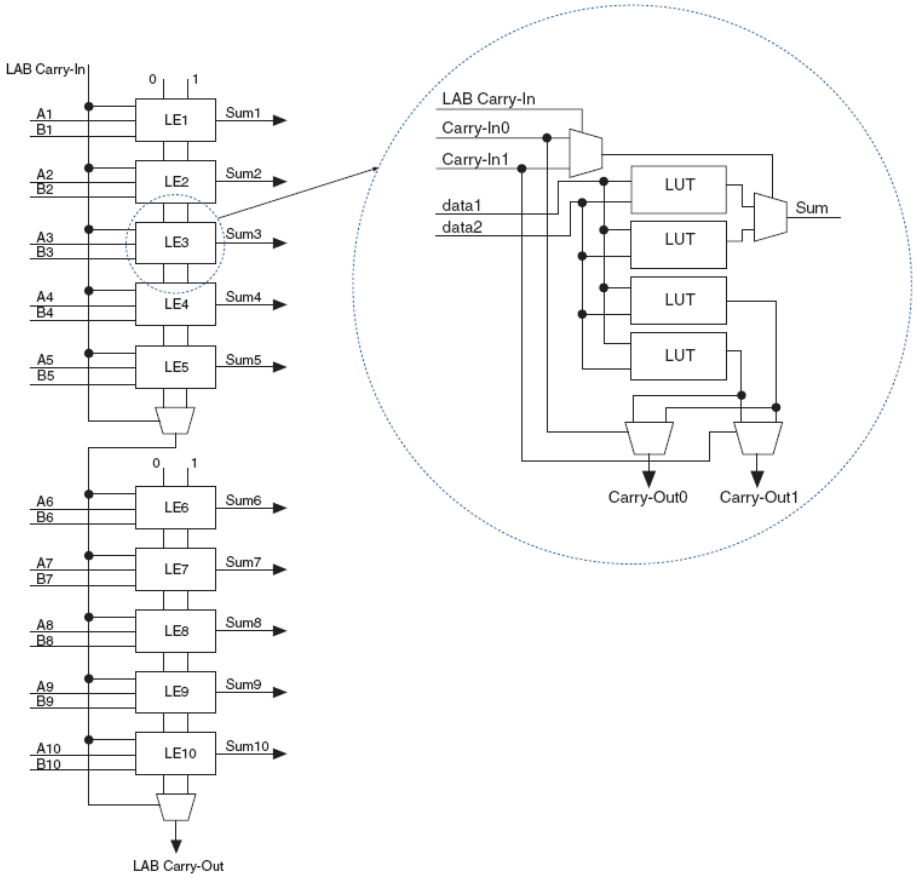


Figure 3.2: Carry Select Chain from Altera Straix FPGA [2]

that it utilizes almost twice the resources compared to a ripple-carry adder. This is due to the duplication of RCA in each group and the need for the multiplexors to select correct

group sum bits. This can be clearly seen in Figure 3.2, in which every Logic Element (LE) consists of four 2-input LUT, one pair being used to calculate the sum and another pair being used to calculate the out-going carry. The critical path of this particular Altera Stratix FPGA is the initial group plus the number of intermediate MUXes at the end of each group of five LEs. Since the work is being done on the Xilinx FPGA, this section is more for the reference and comparison purpose. Thus, this configuration from Altera is no longer considered.

3.5 Summarization

As we have seen, there are papers proposing new implementations on existing platform (e.g., FPGA) to improve the performance of public-key cryptography computation, and others proposing new platform architectures (e.g., domain-specific). Our approach to this issue is to propose a Reconfigurable Computing (RC) environment, where a general-purpose processor (GPP) is augmented with a FPGA, creating a computing engine that acts like an ASIC but has the GPP flexibility.

Moreover, as discussed, the FPGA chip that we use throughout this research is the Xilinx Virtex-II Pro FPGA, in which both ripple-carry and carry-lookahead additions are given architecture support. While ripple-carry addition provides best results for adding two operands with bit length ranging in tenth, carry-lookahead addition might be a better option for long-and-very-long integer addition (e.g., bit length ranging hundreds) in the application, such as public-key cryptography. However, using carry-lookahead addition in FPGA would introduce delays from the expensive global interconnects, as well as from the Look-Up Table (LUT). On top of that, carry-lookahead adder requires much more area and thus, increases the power consumption. Another issue is that it is not straightforward and more expensive to construct a subtraction out of a carry-lookahead adder in this particular Xilinx FPGA. For these reasons, we take one step further than just providing a RC solution;

we propose a Cryptography-oriented Reconfigurable Array (CryptoRA) to minimize the issues that general-purpose (commercial) FPGA might introduce when it is used for public-key cryptography implementation.

Thus, a software implementation in EC point multiplication is presented in the next chapter. It is followed by a description of the Montgomery Modular Multiplication (MMM) hardware unit implemented on Xilinx FPGA, as well as the MMM unit on the proposed CryptoRA. The estimated results from the CryptoRA is presented and compared to the simulated results from the Xilinx FPGA to provide the evidences on performance and area consumption that CryptoRA is able to improve.

Chapter 4

Public-key Cryptosystem Software Implementation

As our approach to the issue of computing public-key cryptosystem in embedded devices is to provide a reconfigurable solution, which is a RISC embedded processor augmented with FPGA, we utilize the IBM workstation with PCI-bus-interfaced Aimirix AP 1000 FPGA Development Board [7], provided by CMC to develop the required platform. This chapter is organized as follows. The hardware platform with a chosen embedded core processor is described in Section 4.1. The software implementation for computing EC point multiplication is presented in Section 4.2. In particular, the Montgomery Modular Multiplication algorithm is discussed in details with its block-level and bit-level examples in subsequent Section 4.2.2. This chapter is completed with the discussion on the communication bridge between the core processor and to-be-included Montgomery Modular Multiplication hardware unit in the next chapter.

4.1 Hardware Platform Configuration

Before the C-level program can be ran on the XC2VP100 Virtex-II Pro FPGA package FF1704 [51] sitting on Amirix AP1000 FPGA Development Board [7], the FPGA embedded processor must be defined. Inside this particular Xilinx FPGA chip, it has dual hard processors (PowerPC 405) and a soft procesor (MicroBlaze) [48]. A hard embedded processor refers to a processor built from dedicated silicon while a soft embedded procesor refers to a processor built using the FPGA general-purpose logic and is typically described in a Hardware Description Language (HDL) or netlist, and therefore, must be synthesized and fit into the FPGA fabric. In both soft and hard processor system, the following Intellectual Property (IP) are needed to be customized and defined to build a complete hardware

platform, in which includes local memory, processor busses, internal peripherals, peripheral controllers, and memory controllers.

In particular, we choose MicroBlaze as the core processor of the system because the Fast Simplex Link (FSL) interface for the data communication between the core processor and the accelerated hardware unit can only be supported if MicroBlaze is used. To define a complete hardware platform on which the C-level program can be ran, the *Base System Builder* wizard is used in the Xilinx Platform Studio (XPS). XPS is an embedded-system developing tool package provided by Xilinx, that is used to create a hardware platform on which a user program can be ran. The IPs included in this embedded processor design are listed below:

- MicroBlaze
  - 100 MHz processor-bus clock frequency
  - disable cache
  - On-chip H/W debug module
  - 64KB local memory
- RS232 UART - 9600 Baudrate, 8 Data-bits
- 32-bit width OPB Timer

The corresponding block digram of the embedded processor system is depicted in Figure 4.1. One of the advantages of the FPGA embedded processor is its reconfigurability. The system can be configured to perform the best in an application. There are two main bus systems supported - Local Memory Block (LMB) Bus and On-chip Peripheral Bus (OPB) Buses. LMB Bus is used specifically to transfer data with single-cycle access in and out the Block RAM (BRAM), where the program instruction and data are stored. On the other hand, OPT Bus interface provides a connection to both on-and off-chip peripherals and

memory. As noticed from the configuration, the system is kept minimal in order to achieve the best performing result in this case. It is worth mentioning that since the size of our entire program (mainly the text and data) is just over 33 KB, the 64KB LMB is selected, which is made up with BRAMs. In addition, since the entire program is stored in the local memory, where it has the same read/write performance as cache, the cache is disable. Cache should only be enable when large section of the program does not fit in the local memory, but instead stored in the external memory, which has large latency in access and is attached to the OPB.

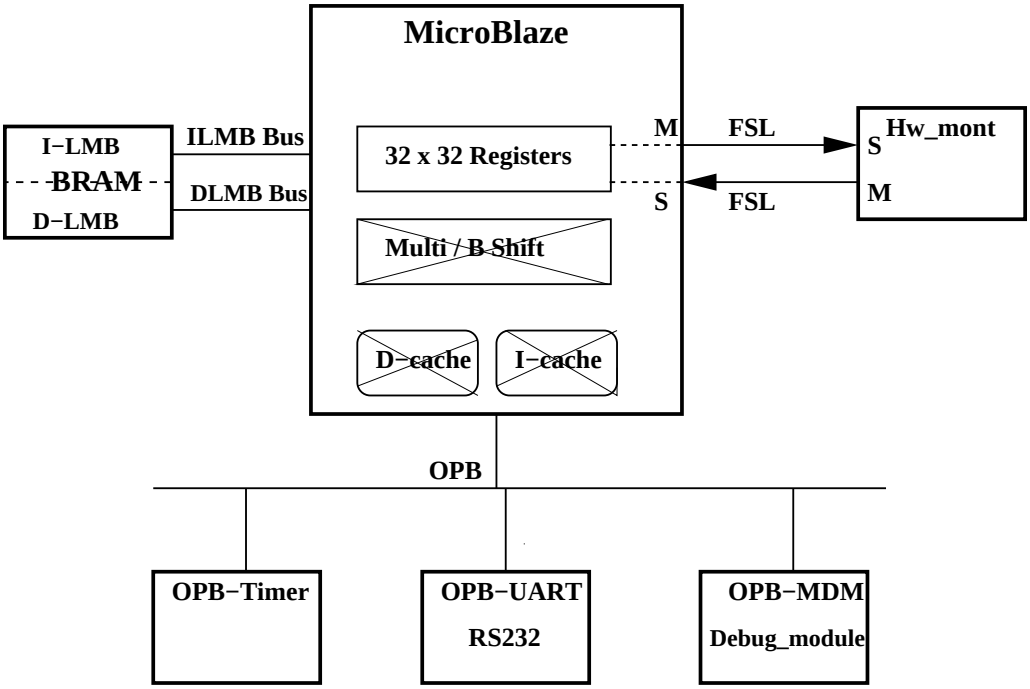


Figure 4.1: Block Diagram of the embedded processor system.

Once the soft MicroBlaze embedded processor is completely defined, synthesized, and mapped to the Virtex-II Pro FPGA, the C-level program is downloaded to and is then run on the development board. The output of the program is viewed on the HyperTerminal. In the next section, a process of the software development for computing the EC point

multiplication is described. In particular, the Montgomery Modular Multiplication (MMM) algorithm is fully discussed.

4.2 EC Point Multiplication Software Implementation

Due to the code portability issue with any of the existing library that supports arbitrary-precision arithemetic routines, a software program that ultimately performs EC Point Multiplication is coded from scratch. With the hierarchy of operations in RSA and ECC public-key cryptographys in mind, the software implementation was began with the implementation of the finite field arithmetic, which are modular addition/subtraction, multiplication, sqauring, exponnetiation, and inversion for long and very long unsigned integers. In the next subsection, modular multiplication is described using the block-level Montgomery modular multiplication algorithm, and modular inversion operation is presented using Fermat's Little Theorem.

4.2.1 Finite Field Arithmetic Implementation

The finite field $GF(p)$, where p is a prime number. $GF(p)$ consists of the integers from $[0, p-1]$, with addition and multiplication performed modulo p. For any integer a, a (mod p) shall denote the unique integer remainder $r, 0 < r < p-1$ obtained upon dividing a by p; this operation is called reduction modulo p. The following examples are the modular operations over a prime field of a prime number $23, GF(23)$.

1. Modular addition:$15+22$ (mod 23) = 37 (mod 23) = 8 (mod 23) (37 is outside the range of $[0, p-1]$; hence a subtraction from the modulus p is required).

2. Modular subtraction:$15-22$ (mod 23) = -7 (mod 23) = 16 (mod 23) (again, -7 is outside the range of $[0, p-1]$; hence an addition to the modulus p is required).

3. Modular Multiplication:15×22 (mod 23) = 330 (mod 23) = 8 (mod 23) (again, 330

is outside the range of $[0, p-1]$; thus, reduction is required, which we can use either division or a series of subtraction until the result is within the range).

4. Modular Inversion:8^{-1} (mod 23) = 3 (mod 23) (using the inverse identity, we know $8 \times 3 = 1$ (mod 23).

As mentioned earlier, modular inversion can be eliminated from the Point operation level (mainly, EC point addition and doubling) by carrying out the Point operations in different coordinates. However, this modular inversion operation is still required once per EC point multiplication. We use so-called Fermat's Little Theorem to perform modular inversion, which is stated in Equation 4.1

$$a^{-1} = a^{p-2} (\text{mod } p), \text{if } gcd(a, p) = 1 \tag{4.1}$$

The main contribution from this theorem is that a modular inversion can now be implemented using modular exponentiation, which is just a series of modular multiplication and squaring as shown in Algorithm 2.2, which is also performed in RSA. By using this theorem, EC point multiplication is even more dependent on the modular multiplication. It is crucial to implement an efficient modular multiplier. Furthermore, this then requires no separate dedicated hardware unit for modular inversion, saving silicon area.

As a result, a various standard key lengths of Point Multiplication operations of ECC public-key cryptosystem is used as the testing cases since RSA cryptosystem is indirected tested by the modular inversion required in the ECC implementation. Therefore, due to the modular exponentiation and modular inversion are based modular multiplication operation, modular multiplication is hereafter the most important and the most timing consuming operation in the finite filed arithmetic level operation. Thus, modular multiplication is implemented using the Montgomery Modular Multiplication algorithm to achieve better computing efficiency.

Thus, it is well known that modular multiplication is the most computational complexity in the finite field operations because of the reduction operation in modular multiplication. The result of a reduction operation is essentially the remainder of a division, which is in the range of $[1, p-1]$. Since division is a most time consuming operation, it also makes the modular multiplication one of the most expensive operation. Thus, there exist many proposed algorithms to speed up the performance in both software and hardware implementations. Some of the most popular algorithms for modular multiplication and reduction are listed in Table 4.1.

Table 4.1: List of Possible Algorithms for modular multiplication and reduction

Methods	Description
NIST recommended curves (fast reduction) [17]	Is the fastest reduction method. Only for Mersenne primes and polynomials
Barrett reduction [20]	Only for prime field.
Montgomery Multiplier [33]	Does the modular multiplication. Dual field support - prime and binary fields.
Shift-and-add Multiplier [20]	Performs only regular multiplication $\rightarrow$ need to be combined with reduction algorithm to complete modular multiplication.
Karatsuba-Ofman Multiplier [24]	Performs only regular multiplication $\rightarrow$ need to be combined with reduction algorithm to complete modular multiplication.

Moreover, if the prime that is used in EC point multiplication is one of the NIST recommended prime, then the fast reduction can be applied, which by far is the fastest method of computing modular multiplication. In other words, if the prime used in EC point multiplication is not the NIST recommended prime, then this fast reduction can not be applied.

Hence, the hardware unit can not be shared between RSA and ECC alorithms. Due to the processor can operate better on the data that has the same bit-width (block-wise operation), the Karatusuba-OFman Multiplier in conjunction with Montgomery reduction is a popular algorithm for computing modular multiplication in software. Nevertheless, MMM is a good choice for both software and hardware implementation since it can be implemented in either word-level or bit-level.Thus, MMM is our choice for modular multiplication operation.

4.2.2 Montgomery Modular Multiplication

Among all the proposed algorithms, Montgomery Modular Multiplication (MMM) Algorithm [33] is proved to be a very efficient algorithm - it replaces the division operation with a series of multiplications and shifts to perform the reduction. In addition, MMM was also verified that can it not only be used in the prime field, it can also be used in the binary field if the inputs are in the form of polynomial [25]. In addition, it has been shown that MMM is the bottleneck for software solution. Therefore, we will only consider the MMM Algorithm for the modular multiplication implementation in reconfigurable hardware. While there are many variants of the MMM exist, the original proposed MMM by Montgomery is shown in Algorithm 4.1.

Among all the required parameters, R is the scaler factor and p' is a value needed to be pre-computed if $\alpha > 1$. As noticed from the Algorithm 4.1, the output of the MMM is $xyR^{-1} \bmod p$, instead of $xy \bmod p$ that we are looking for. The term R is created by the total number of shift-to-the-right operation in the MMM, which simply equals to the bit-length of the modulus, p. In order to obtain the correct value of $xy \bmod p$ result, an additional MMMA operation is required. The operands of this additional MMM are the result of first MMM and the value of $R^2 \bmod p$. By doing so, the ($R \bmod p$) term is to compensate the $R^{-1} \bmod p$ introduced in this MMM itself while another ($R \bmod p$) term is to compensate the $R^{-1} \bmod p$ in the last MMM. This is essentially the overhead that MMM

Algorithm 4.1 Montgomery modular multiplication with fianl subtraction

Require: Integers $p = (p(l-1)\dots p(1)p(0))_{2^\alpha}$, $x = (x(l-1)\dots x(1)x(0))_{2^\alpha}$, $y = (y(l-1)\dots y(1)y(0))_{2^\alpha}$ with $x,y\varepsilon[0,p-1]$, $R = (2^\alpha)^l$, $gcd(p,2^\alpha) = 1$ and $p' = -p^{-1} \bmod 2^\alpha$

Ensure: $xyR^{-1} \bmod p$

1: $T \leftarrow 0$
2: **for** i from 0 to $(l-1)$ **do**
3: $\quad m_i \leftarrow (T_i(0) + x(i)\cdot y(0))p' \pmod{2^\alpha}$
4: $\quad T_{i+1} \leftarrow (T_i + x(i)\cdot y + m_i \cdot p)/2^\alpha$
5: **end for**
6: **if** $T_l \geq p$ **then**
7: $\quad T_l \leftarrow T_l - p$
8: **end if**
9: **return** (T_l)

creates. Hence, the benefit of the efficiency of MMM will be diminished by the overhead it creates if only one or a few modular multiplication operation is required. In other words, the efficiency will be benefited from performing MMM if a series of modular multiplications are need, such in the case of RSA and ECC. A more efficient way when performing MMM is to convert all the operands from the normal representation to Montgomery representation in the beginning, such that $\overline{x} = Mont(x,R^2) \equiv (x\cdot R) \bmod p$.

Again, the R mod p term is absorbed by the MMM itself. Continue with all other modular operations in this representation. At the end, to covert back from the Montgomery representation to normal representation, such that $x = Mont(\overline{x},1) \equiv (x) \bmod p$.

It is apparent that the core of this algorithm lies in the line 3 and 4, in which the most time are spent. While line 3 is used to determine the coefficent value representing the multiple of modulus later added to the summation of $T_i + x(i)y$, line 4 is to zero the least

significant block of the temporary result T_i ($T_i(0)$) so that when T_i is shifted to the right by the block size of α bits, the temporary result does not loose any digit. It is worth mentioning that when performing modular addition, adding any value to a multiple of the modulus does not alter the modular addition result (e.g., 56 mod 11 == $56+3\cdot 11$ mod 11 == 1). Thus, this technique is used in line 4 of Algorithm 4.1 to zero the least significant block.

Furthermore, while word-level MMM implementation is preferable in the software approach, bit-level MMM implementation is often favored in the hardware approach, particularly for the FPGAs. Word-level MMM means that the block size, α, is greater than 1, usually is the wordlength of the system, which is normally 16 bits or 32 bits so that it can be efficiently executed. On the other hand, bit-level MMM means that the block size, α equals to 1, leading one of the operands is scanned bit by bit from the least significant bit to the most significant bit. Even though the bit-level implementation yeilds the largest number of iterations, l, it in turn simplifies the Algorithm 4.1, such that line 3 and line 4 become $m_i = T_i(0) + x(i)y(0)$ and $T_{i+1} = (T_i + x(i)\cdot y + m_i \cdot p)/2$, respectively. Furthermore, m_i can only be either 0 or 1, and so can x_i. This in turn makes multiplicaitons in line 3 and 4 disappear, which also makes the Montgomery modular multiplication easiler to be mapped onto FPGAs, and thus, saves some hardware area. Below is an example to illustrate how MMMA works in bit-level implementation.

Inputs: x, y, p, p', where x is the multiplier, y is the multiplicand, p is the modulus, and p' is the reduction factor.

Outputs: xyR^{-1} (mod p), where $R = 2^n$, and $n =$ bit length of p.

Assuming: $x = [17]_{10} = [10001]_2, y = [22]_{10} = [10110]_2, p = [23]_{10} = [10111]_2$

The straightforward hand calculation is as the following:

R^{-1} is the inverse of R and can be computed using Extended Euclidean Algorithm (EEA).

$RR^{-1} = 1$ (mod23) $\Rightarrow 2^5 R^{-1} = 1$(mod23) $\Rightarrow R^{-1} = -5$

xyR^{-1} (mod p) $\Rightarrow 17(22)(-5)$ (mod 23)=16

Here, we have computed that the result is 16. And now, we will illustrate how to

produce the same result by performing a bit-level MMM. In this case, the number of iteration is 5, which equals to the bitlength of the modulus. Due to α equals to 1 in the bit-level MMM, $p^{'}$ also equals to 1. Table 4.2.2 reveals the results from line 3 and 4 of Algorithm 4.1 for each iteration. In the example that we use, since the bit 0 of x is 1, the operand y is then added to the temporary result, which is initially set to 0. Since the least significant bit of this summation result (m_i) is 0, it can then be directly shifted to the right by one bit, resulting 11 in the frist iteration. Same procedure is taken for each every iteration until the $l-1$-bit position is reached. As seen, in the last entrance, the result is calculated to be 16, which is the same value as it was calculated using block-level MMM. This demonstration also reveals that the bit-level MMM requires more iterations than the block-level to complete the computation. However, the operations are much simplier (e.g., addition and shift). The next level of operation to build is the EC Point operation, which are

i	x_i	$m_i = T_0 + x_i y_0$	$T = (T + x_i y + m_i p)/2$
0	1	0	(22+0(23))/2=11
1	0	1	(11+1(23))/2 = 17
2	0	1	(17+1(23))/2 = 20
3	0	0	20/2 = 10
4	1	0	32 /2 = 16

EC point addition and doubling required in EC Point Multiplicatoin stated in Algorithm 2.3 and is presented in the next subsection.

4.2.3 EC Point Operation Implementation

As mentioned earlier, different coordinates in which the EC point multiplication is operated yields different implementation of EC point addition and doubling. The coordianate that we decided to use is the modified Jacobian (J^m) coordinates proposed by Cohen *et al.* [9]. This is because that this coordinates yields the fastest EC point doubling, meaning

the least number of modular multiplicaton required. EC point addition and doubling in the (J^m) coordinate can be found in [35]. In the (J^m) coordinate, the total number count of modular multiplication are 14 and 8 per EC point addition and per point doubling operation, respectively. The next level to build is of course the EC point multiplication. For the simplicity, the Double-and-Andd Algorithm described in Algorthm 2.3 is used to perform EC point multiplication.

All the points on the elliptic curve start in the Affine Coordinate (x,y) representation. In order to perform in the J^m coordinate, the representation of the points needs to be converted to (X,Y,Z,aZ), (e.g., a point $P(x,y) \rightarrow P(x,y,1,a)$), where x, y, 1, and a are mapped to X, Y, Z, and Za, respectively. In particular, a is the variable a in the elliptic curve equation in Equation 2.3 (same curve equation for prime field). In addition, the EC point variables in normal representation are needed to be converted to Montgomery representation as described in Chapter 5. Hereafter, all the arithmetic are carried out in the Montgomery representation. At the end of each Point Multiplication, converting the J^m coordinate representation back to the Affine coordinate is needed. In other words, we want this following conversion: $Q(X,Y,Z,aZ) \rightarrow Q(x,y)$, where $x = X \cdot Z^{-2}$ and $y = Y \cdot Z^{-3}$. To do so, we need to perform one modular inversion of Z using the Fermat's Little Theorem to obtain Z^{-1}. And then perform $Mont(Z^{-1},Z^{-1})$ to obtain Z^{-2} and so on. The last step is to convert Montgomery representaiton back to Normal representation, which is also described in Section 4.2.2.

Furthermore, a Multi-precision Integer and Rational Arithmetic C/C++ Library called *MIRACL*, licensed through Shamus Software Ltd [42] is imported to the Microsoft Visual Studio. For the comparison purpose, this library is then used to perform a EC point multiplication of various key length with its embedded function routines optimized in assembly for the GNU C compiler in the Virtual Studio development environment. MIRACL is used to ensure not only the correctness of our program, but also the performance, which can be seen in Chapter6. As it is impossible to perform EC point multiplication by hand, it is nec-

essary to utilize other software/library to verify the result. At the same time, the computing performance can be compared using the profiling feature.

4.2.4 Commuication Between Core Processor and Hardware Unit Interface

In order to merge the accelerated Montgomery Modular Multiplication hardware unit to our C-level software program, the Xilinx Fast Simplex Link (FSL) is used for the co-processor interface [47]. FSL is a very fast dedicated connection that Xilinx provides for data transfering between MicroBlaze and the user-defined hardware unit because it brings data in and out directly from the MicroBlaze internal register files. There are up to 16 FSL ports available on the MicroBlaze (8 master / 8 slave), in each of which is first in, first out (FIFO) link, which can be seen in Figure 4.2. Also, data are transferred in and out by the *get* and *put* assembly instructions (available in blocking and non-blocking instructions).

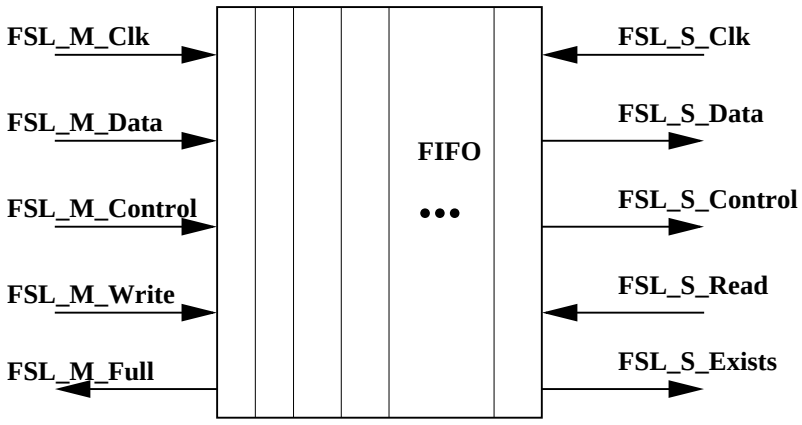


Figure 4.2: Fast Simplex Link (FSL) Bus [50]

Even though FSL is capable of transfering 32-bit data every two clock cycle, we deal with integers ranging from 160-bit to 2048-bit, which is enterpreted to 6 to 64 blocks of 32-bit data. In our implementation, we transfer 5 variables per modular multiplication operation to the hardware for the computation, and transfer 2 variables back to the core processor at end of each modular multiplication operation. As a result, this creates large

overheads on the communication as the cycle count for the EC point multiplication is substantially reduced with the MMM hardware unit support. One of the many techniques that we apply to mitigate the time it spends on the communication is the loop-unrolling. This eliminates the additional cycle counts created by the *FOR* loop (e.g, branch, comparison, and addition instructions).

As the software implementation for computing EC point multiplication is developed and running on the defined hardware platform with the core-processor being MicroBlaze, the next step is to present the Montgomery Modular Multiplication hardware unit on both Xilinx FPGA and our new proposal, Cryptography-oriented Reconfigurable Array (CryptoRA).

Chapter 5

Montgomery Modular Multiplier Hardware Unit in Xilinx and *CryptoRA*

Since Montgomery Modular Multiplication algorithm is implemented bit-level in hardware, the long-and-very long integer addition/subtraction and comparison become the core operations. Thus, several fast adders implementations on Xilinx Virtex-II Pro FPGA are discussed. This chapter is organized as follows. A discussion on the Carry-Skip Adder and Carry-Lookahead Adder implementations is taken place in Section 5.1. The implementation of the adder/subtractor component and comparator unit component needed to build a Xilinx FPGA-based MMM hardware unit is covered in Section 5.2. This is followed by the implementation of the new components and structures in the adder/subtractor and comparator unit required to build a *CryptoRA* FPGA-based MMM hardware. This chapter is completed with the *CryptoRA*'s features, which include: an increased granularity of the logic tile, the extension of the dedicated carry chain over the horizontal direction, and the splitting LUT.

5.1 Other Fast Adders Implementation in Xilinx

Both Carry-Skip Adder (CSkA) and Carry-Lookahead Adder (CLA) implementations are attemped on Xilinx Virtex-II Pro FPGA and the pros and cons of their structures are discussed.

5.1.1 High-Speed Carry-Skip Adder in Xilinx

Since their high-speed CSkA [18] can not be implemented efficiently for subtraction in current FPGA platform, the sum and carry bit of the result after the CSA are added using

simple RCA, which obviously is not the best solution for long and very long unsigne-integer addition. According to the high-level structure of the CSkA in Figure 5.1, even though the majority of the high-speed CSkA structure is similar to the one in CLA and can be implemented in similar way, e.g., *generate* and *propagate* signals, it contains inregular structure in nature. These are: 1. the *AND OR* gates at the end of each column – it would be a waste in area to implement this logic as dedicated gate in every slice since it is only needed at the end of each column. On the other hand, if only a dedicated logic gate is included at the end of each column, it will limit FPGA structure in the sense that each column must end at the place where the dedicated logic gate is implemented. This *AND OR* logic function is emulated using LUT since there is no dedicated logic for it in the current FPGA. Also, to balance block-internal and carry-propagation delays, every higher block is approx. 20-bit longer than the block directly below (LUT delay equal approx. 20 carry MUXes delay). 2. the large number of fanout created by the output of the *AND OR* logic gate —it is impossible to make these signals dedicated path in FPGA platform as the number of fanouts can not be fixed because the required column size might be different. Nevertheless, the fast CSkA is still implemented in Xilinx FPGA for the comparison purpose with other type of fast adders and results are shown in the next chapter.

5.1.2 Carry-Lookahead Adder in Xilinx

As the issue with RCA is well know to us, which is the time it takes for the carry to be propagated through the entire length of the adder that has linear relation with the adder length, there are variations of fast adder techniques proposed in order to reduce such the latency created by the carry propagation. One way to decrease the ripple-carry critical path is to reduce the dependency of the outgoing carry, $c_{out}(i)$, on the incoming carry, $c_{in}(i)$. In CLA, this is achieved by defining two bit-level *generate* ($g(i)$) and *propagate* ($p(i)$) signals, for each position i, as shown in Equation 5.2 and 5.1, respectively. Based on bit-level signals, block-level *generate* ($g_b(j)$) and *propagate* ($p_b(j)$) signals, can be defined for each

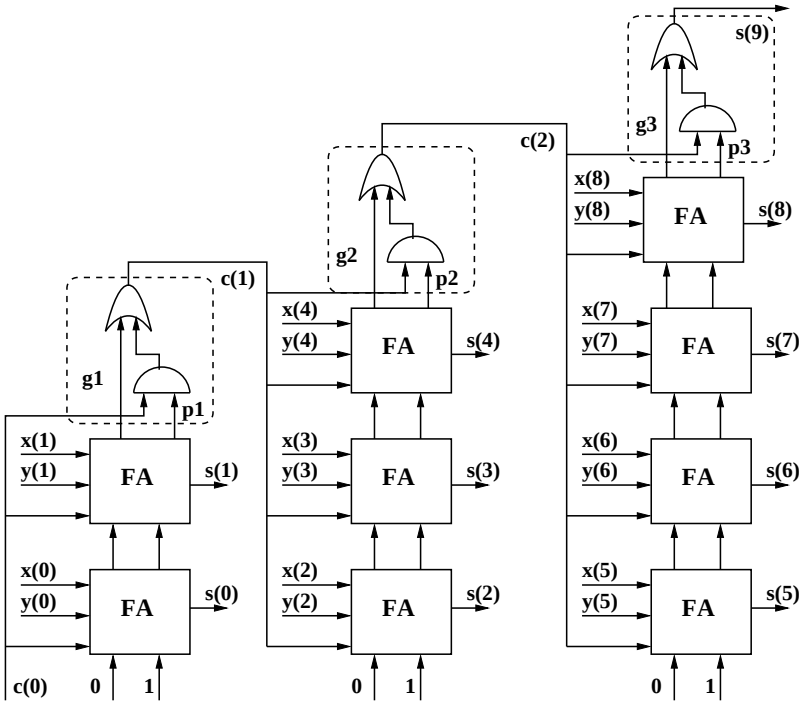


Figure 5.1: 9-bit carry skip adder.

block j, as shown in Equation 5.4 and 5.3, respectively. The grouping process can continue recursively, where blocks can be combined into a next level block to form a hierarchy of block-level generate and propagate signals. Since these *generate* and *propagate* signals do not depend on the incoming carry bits, $c_in(i)$ and $c_in(j)$, they can be calculated in parallel.

$$p(i) = x(i) \oplus y(i) \tag{5.1}$$

$$g(i) = x(i) \cdot y(i) \tag{5.2}$$

$$p_b(j) = \prod_{i=I}^{I+K-1} p(i) \tag{5.3}$$

$$g_b(j) = g(i) + g(i-1)p(i) + g(i-2)p(i-1)p(i) + g(i-3)p(i-2)p(i-1)p(i) + \ldots \tag{5.4}$$

Based on bit-level and/or block-level generate and propagate signals, the input and

output carries for position i and block j are defined by Equation 5.5.

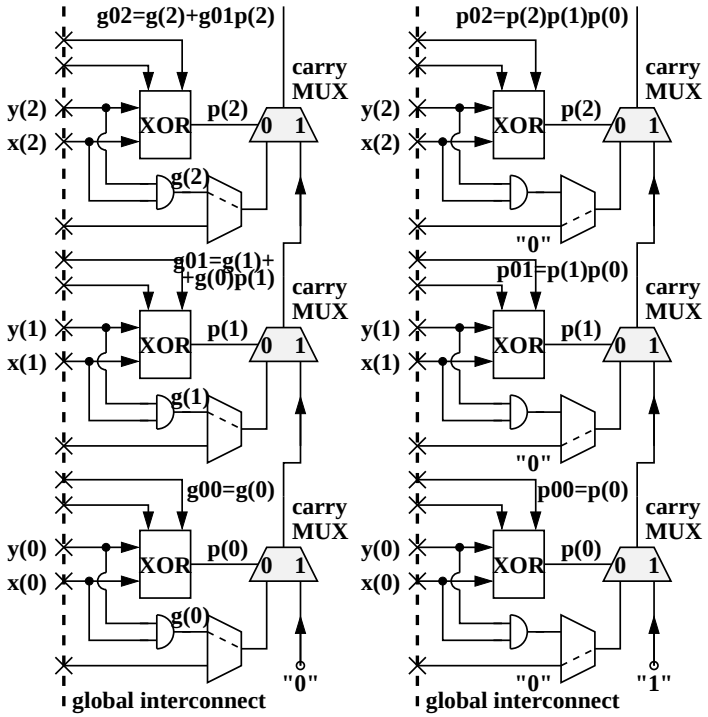
$$\begin{aligned} c_{out}(i) &= g(i) + c_{in}(i)p(i) \\ c_{out}(j) &= g_b(j) + c_{in}(j)p_b(i) \end{aligned} \tag{5.5}$$

In carry-lookahead adders, all the generate and propagate signals in the hierarchy are initially calculated. Then, based on the incoming carry into the first position, $c_{in}(0)$, all carries are calculated in parallel. Finally, all the sum bits, $s(i)$ are calculated according to Equation 5.6. It is well-known that CLA has a latency of $O(log(n))$, where n is the wordlength [36]. (maybe include carry-lookahead hierarchy diagram here).

From the point of view of a reconfigurable array, CLA has an advantage against CSkA. Specifically, *generate* and *propagate* networks of Xilinx are intrinsically more flexible in implementing other wide-input logic functions (e.g., *OR* or *AND*) than the carry-select networks of Altera. For this reason, we build our cryptography-oriented FPGA starting from a Xilinx-style architecture, on which carry-lookahead is architecturally supported. It is worth mentioning that our decision is consistent with Hauck *et al.* result that a Brent-Kung adder, which a carry-lookahead adder achieves a very good latency performance [22].

In order to implement and map the CLA using the dedicated carry chain on Xilinx Virtex II Pro Chip, VHDL code must be written in the logic level to specify the internal components to be used inside the *Slices*. Since the dedicated carry chain is used to for signals, such as block-level *generate* ($g_b(j)$), block-level *propagate* ($p_b(j)$), block-level *carry* ($c_b(j)$), and intermediate *carry* ($c(i)$), the block size is no longer bound to four bits in length; it can now be any arbitrary numbers, such as 8, 16, or 32 and so on. This is because the *AND/OR* gates that are used to generate the $g_b(j)$, $p_b(j)$, $c_b(j)$, $c(i)$ signals are now emulated using the dedicated carry chain and the internal logic gates, which are shown in Figure 5.2(a), Figure 5.2(b), Figure 5.3(a), and Figure 5.3(b), respectively.

To begin with, the block-level *propagate* signal ($p_b(j)$) in Equation 5.3 is essentially a *wide-OR* function, and therefore, it can be implemented by means of the carry chain

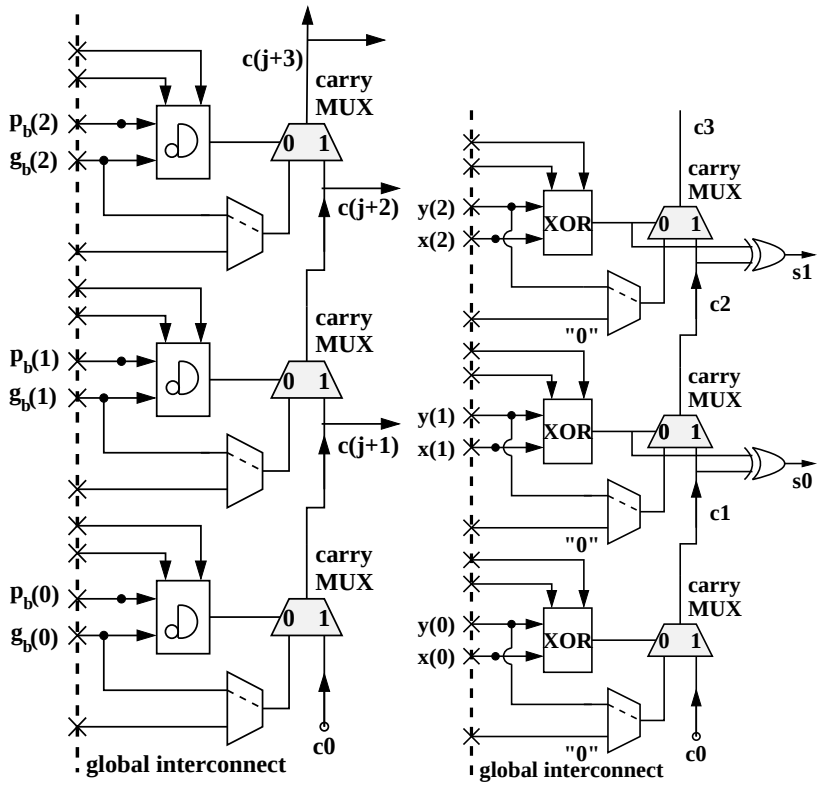


(a) Block generate signal (b) Block propagate signal

Figure 5.2: Carry-lookahead network on Xilinx Part-I

[51]. Also, it is noticed that block-level *generate* signal, $g_b(j)$, in Equation 5.4 is indeed the *carry* signal that does not include the incoming carry. Thus, $g_b(j)$ signal also can be implemented by utilizing the carry chain. Without the internal *AND* gates inside each Slice of Xilinx Virtex II Pro Chip [51], to implement the $g_b(j)$ signal using the carry chain would require 2 LUTs and 2 dedicated MUXes in each element of the $g_b(j)$ chain. By making the use of the internal *AND* gates, each elements of the $g_b(j)$ chain can be implemented using only 1 LUT and 1 dedicated MUX. This is why Xilinx claims that some of their chips support CLA [Xilinx support carry lookahead paper]. In addition, the reason why Equation 5.3 and 5.4 can be mapped onto dedicated carry chain is because the mutually exclusive property that is held between the bit-level *generate* and *propagate.* This mutually exclusive property ensures that the condition of both $g(i)$ and $p(i)$ signals being true can

never occur. This property is also held true in the block-level signals. Thus, block-level signals can be implemented using the dedicated carry chain. For the block-level *carry*, because the inputs to this block are $g_b(j)$ and $p_b(j)$ signals, the logic function using LUT is modified to $p_b(j)\overline{g_b(j)}$ and the schematic diagram is shown in Fig 5.3(a) for Equation 5.5.



(a) Block-level carry generator (b) Sum bit signals

Figure 5.3: Carry-lookahead network on Xilinx Part-II

In Figure 5.3(a), the output from each dedicated MUX in j_{th} element is then connected to the input of the next dedicated MUX in $j+1$ element, and also to the input of the MUX for generating the sum bits. When tracing the signals on the chip floorplan, it was observed that the physical dedicated carry chain for the block-level *carry* generator block is being discontinued. Instead, the physical dedicated carry chain is continued with the sum-bit

block. This mapping results in a longer critical path in the design, which is undesired. This unwanted mapping result is due to the dedicated carry MUX having its output connected to the input of two separate dedicated MUXes. When the VHDL code is been sythesized, it sees that the block for generating the sum bits is longer than the *carry generator* block. It is not awared of the overall design for the best optimization. Therefore, it tries to optimize the design by choosing to continue the dedicated carry path to the *sum-bit* block.

The solution to this issue is that we emulate the first element in each sum-bit block in Figure 5.3(b) using LUT. This emulation for generating the carry and sum bits of the first element is based on the original carry and sum equations and is depicted in Figure 5.4. This technique forces the incoming carry in each sum-bit block to go through interconnection because the incoming carry is now one of the inputs of LUT; this leaves the output of the dedicated carry MUX in block-level *carry* with one choice to continuing the dedicated path. The penalty is that an additional LUT is needed to generate the sum bit of the first element. In addition, if CLAs are implemented in commercial FPGAs, the number of carry-lookahead level should be kept minimal and only increase the block size when longer bit length is needed. This is because the connection between the level means the global interconnects on FPGA, which is expensive. On the other hand, the increse in number of elements means increase in the dedicated MUX usage, which is quite cheap compared to the global interconnects.

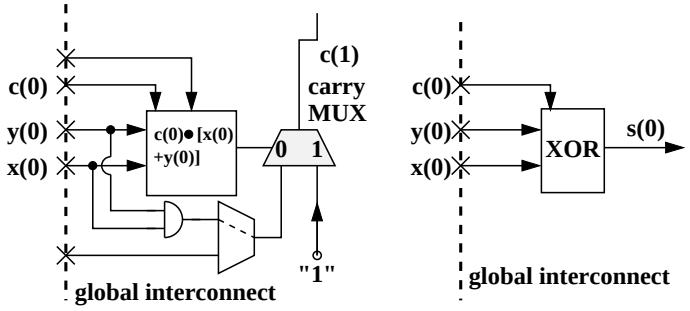


Figure 5.4: Modification of 1st element in sum-bit block.

5.2 Montgomery Modular Multiplier Hardware in Xilinx

Since the Montgomery modular multiplication is a recursion, its two additions can be implementated using a redundant carry-save representation, as proposed by Goodman and Chandrakasan [18]. This means that the addition result is represented as a carry and a sum pair. Therefore, a reduction from four operands (previous carry and sum, and operands y and p) to two operands (carry and sum) is performed each iteration.

5.2.1 Carry-Save Adder Structure In MMM

To avoid the time-comsuming carry propagation, carry-save addition uses a redundant representation for the result comprising separate sum and carry bit words [36]. Carry-save addition is espeically approriate for multi-operand addition, when only last two-operand addition needs to reduce the representation (thus propagate the carry) to a single word of sum bits.

The advantage of utilizing one implementation over another depends on the platform on which the adder will be implemented. For example, carry lookahead may be cheaper than carry-select on an ASIC. For FPGAs, Xilinx provides dedicated resources to supoort carry lookahead, while Altera supports carry select in hardware. Even though Carry-save addition is fast, the twice usage on the Flip-Flop for storing the carry bit and the conversion two the non-redundant representation at the end are the downside using such approach.

While the MMM is implemented in 32-bit block-level in software, it is implemented in bit-level in hardware. Our initial MMM is based on the Goodman's ASIC implementation [18]. As line 3 and 4 in Algorithm 4.1 are simplified in bit-level implementation as described in Chapter 4, all the operations executing in line 3 and 4 can be emulated using two levels of 3-to-2 CSA in the form of bit-slice configuration, shown in Figure 5.5. The top *AND* gate in each bit-slice configuration is used to detect the value of the i_{th}-bit of x (1 or 0), which is broadcasted throughout the array. If $x(i)$ is 1, then the y operand is added, otherwise 0 is added. Similarly, the 2nd *AND* gate is used to detect the varaible m_i bit,

which is also broadcasted throughout the array. If m_i bit is 1, then the p is added, otherwise 0 is added. Even though the m_i signal needs extra logic to compute, it does not pose any addition delay because it is computed in parallel with the first 3-to-2 CSA, which is the top half of the bit-slice.

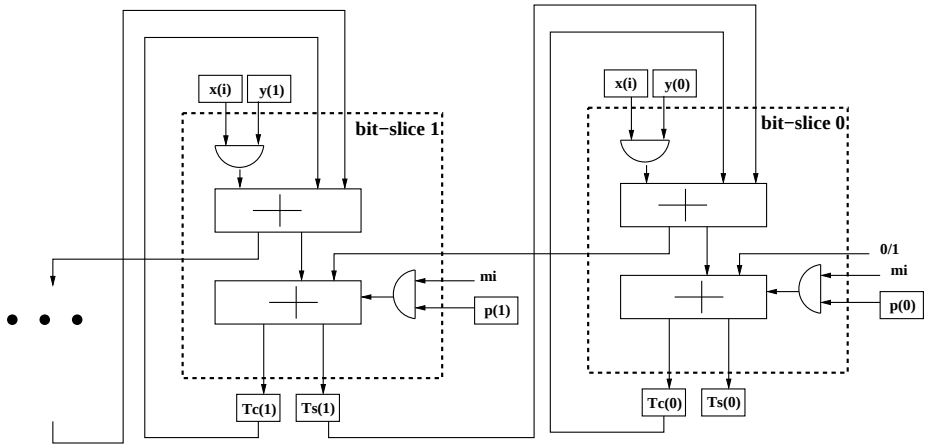


Figure 5.5: Bit-slice of CSA implementation.

Furthermore, the shift-to-the-right by 1 bit operation is emulated by feeding the sum bit at position i one position back, while feeding the carry bit back at the same position i for the next iteration as illustrated in Figure 5.5. In sum, this specific configuration allows all the operations executing in line 3 and line 4 in one hardware clock cycle. This results in a very low clock cycle count per MMM operation (e.g., for 192-bit ECC, it would take around 195 hardware clock cycle to complete one MMM).

Although the carry-save technique is very simple, it has two major drawbacks. First, this is an expensive technique since two tiles are used per bit to calculate in parallel the sum and the carry bits. More important, all the connections are made with the global interconnect network. Therefore, a rich, thus expensive, global interconnect network is required to support the carry-save technique. Second, to implement the products $x(i) \cdot y$ and $m_i \cdot p$, the bit signals $x(i)$ and m_i need to be broadcast to n computing tiles, which leads to a fan-out of

n (where n is the operands word-length). Due to the propagation over a long global interconnect wire and the large fan-out, a large propagation delay is encountered. This means that the acutal delay introduced by the computing tiles is much less than the propagation delay. Rather than having the computing tiles idle while waiting for $x(i)$ and m_i to propagate, it is better to perform some other useful tasks, e.g., perform ripple-carry computation or calculate block-level *generate* and *propagate* signals with the dedicated carry chains. Hence, other fast-adder techniques, such as Carry-Lookahead Adder are considered to be utilized for further performance improvement.

5.2.2 Ripple-Carry Adder in MMM

To convert the adding result using CSA from the redundant form to the non-redundant form, other form of adder or fast adder techniques is needed. We choose the Ripple-Carry Adder for such task even though it poses substantial increase in delay due to the nature of its structure, which is described as the follows. Assume two input arguments x and y, and their sum, s. The full-adder identities for bit i are shown in Equation 5.6, where $c_{in}(i)$ and $c_{out}(i)$ are input and output carry bits, respectively. A RCA [36] is built with a series of full-adder blocks with c_{out} at position i being connected to c_{in} at position $i+1$. It is apparent that the critical path includes the carry propagation. This translates to a large latency in cryptography, where long and very long addition are required. Therefore, fast addition techniques are worth being investigated.

$$\begin{aligned} s(i) &= x(i) \oplus y(i) \oplus c_{in}(i) \\ c_{out}(i) &= x(i)y(i) + x(i)c_{in}(i) + y(i)c_{in}(i) \end{aligned} \tag{5.6}$$

However, Even though the Carry-Lookahead Adder is given hardware support in the Xilinx FPGA Virtex-II Pro, performaning subtraction operation, which is required in MMM algorithm, requires extra Look-Up Table and thus, increases its critical path and area consumption. Also, because this Xilinx chip has hardware support for RCA by means of the

dedicated carry path as shown in Chapter 3, RCA is chosen for the one addition and one subtraction operation required in MMM algorithm implemented in Xilinx FPGA.

5.2.3 N-bit Comparator Unit in MMM

As it can be seen in Algorithm 4.1, it consists of a comparison of two long-unsigned integers. J. Goodman and A. Chandrakasan proposed an implementation of the compare unit in ASIC [18]. The equations for the fast tree-based comparator are as follows.

$$EQ(i) = \overline{x(i) \oplus y(i)} \tag{5.7}$$

$$GT(i) = x(i) \cdot \overline{y(i)} \tag{5.8}$$

$$EQ(j+1,i) = EQ(j,2i+1) \cdot EQ(j,2i) \tag{5.9}$$

$$GT(j+1,i) = GT(j,2i+1) + GT(j,2i) \cdot EQ(j,2i+1) \tag{5.10}$$

Equation 5.7 and Equation 5.8 are used to compute an array of equal bits and greater bits, respectively. $EQ(i)$ is set to 1 if both $x(i)$ and $y(i)$ are equal while $GT(i)$ is only set to 1 when $x(i)$ is 1 and $x(i)$ is 0, meaning x is greater than y at bit position i. Two bits adjcent to each other in the array are then used as the inputs to compute the next level of the comparator tree using Equation 5.9 and Equation 5.10. The Equation 5.9 can be interpreted as the following: the output from this equation is only set to 1 if the compared bits are equal. Equation 5.10 can be interpreted as the following: the output is set to 1 when the higher order bit of which two compared bits is 1, implying $x(i+1)$ is greater than $y(i+1)$, (ie: $GT(1,1) = 1$). The output is also true if the lower order bit of which two compared bits is 1 (ie: $GT(1,0) = 1$) and the equal bit of the higher order bit is 1 (ie: $EQ(1,1) = 1$); this implies that, for example, $x(1) > y(1)$ and $x(2) = y(2)$. In other cases, the output in Equation 5.10 is false.

Figure 5.6 is the graphical representation of the Equation 5.7 to Equation 5.10. The topmost arrays are computed using the Equation 5.7 and Equation 5.8 for EQ and GT

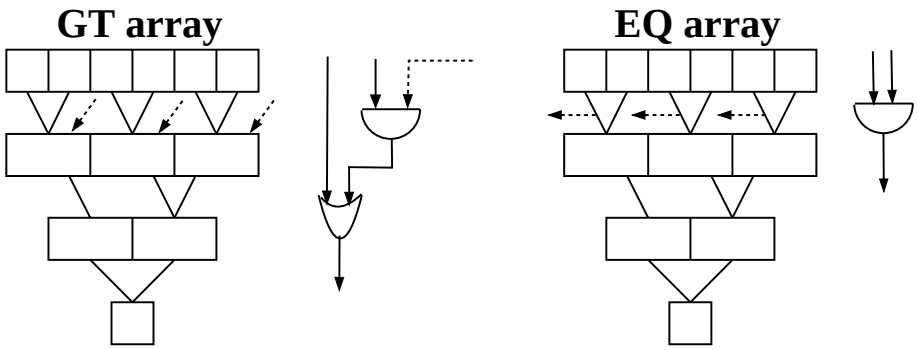


Figure 5.6: Original GT and EQ flags in parallel structure

arrays, respectively. The array in the rest of the layers are computed using Equation 5.9 and Equation 5.10.

Moreover, the equations for the comparator unit given in [18] can be modified by changing the *AND* logic to *OR* logic in Equation 5.9 and invert the final EQ bit while keeping other equations provided in that literature the same. These equations are indeed what is been used for the FPGA implementation (Equation 5.11 to Equation 5.14 due to a better mapping from the equation to the logic, (e.g.: GT array and EQ array can use the same logic function).

$$EQ(i) = x(i) \oplus y(i) \tag{5.11}$$

$$GT(i) = x(i) \cdot \overline{y(i)} \tag{5.12}$$

$$EQ(j+1,i) = EQ(j,2i+1) + EQ(j,2i) \tag{5.13}$$

$$GT(j+1,i) = GT(j,2i+1) + GT(j,2i) \cdot \overline{EQ(j,2i+1)} \tag{5.14}$$

Furthermore, the latency of the fast tree-based comparator unit for comparing two n-bit operands is $O(log(n))$ gate delay [18]. However, this claim does not apply on the FPGA platform since the connection between each level is now global interconnection, which is expensive in terms of the latency. Thus, if the same structure was implemented in FPGA, $O(log(n))$ is the number of interconnect that it requires. In order to fit this comparator unit

better in FPGA by means of making use of the dedicated carry path, we propose a new sequence of comparing the bits. That is, changing it from its original parallel structure to the new-proposed serial structure using the dedicated chain. This new serial structure of comparator unit is also presented in the next section with other improvement and features using our proposed Cryptography-oriented Reconfigurable Array (CryptoRA).

5.3 Montogmery Modular Multiplier in *CryptoRA*

As we have discussed, with our initial MMM unit implementation on Xilinx Virtex-II Pro FPGA, the issues that it creates are the additional area required for storing sum and carry signals in the CSA, and the long critical path created by the CSA, the comparator unit, and especially by the RCA. In order to alleviate these issues, we propose a new Cryptography-oriented Reconfigurable Array, called CryptoRA.

As we have seen how to map the Carry-Lookahead Adder (CLA) equations, which are proposed to be used to replace the CSA in MMM hardware unit, onto the current Xilinx Virtex-II Pro FPGA in the earlier section, we present a better mapping result using *CryptoRA*. Also, since the current Xilinx FPGA does not have hardware support for the new-proposed comparator unit structure, we now show how to map it onto *CryptoRA*.

5.3.1 New Serial structure of Comparator Unit

We realize that the comparison result does not change with different order of comparing bits being applied. Thus, instead of comparing bits in pairs in parallel, the bits are now compared in serial to the previous compared result. The original comparator equations (mainly Equation 5.11 ~5.14) are became Equation 5.15 ~5.18, which now look similar with the CLA equations.

$$eq(i) = x(i) \oplus y(i) \tag{5.15}$$

$$gt(i) = x(i) \cdot \overline{y(i)} \tag{5.16}$$

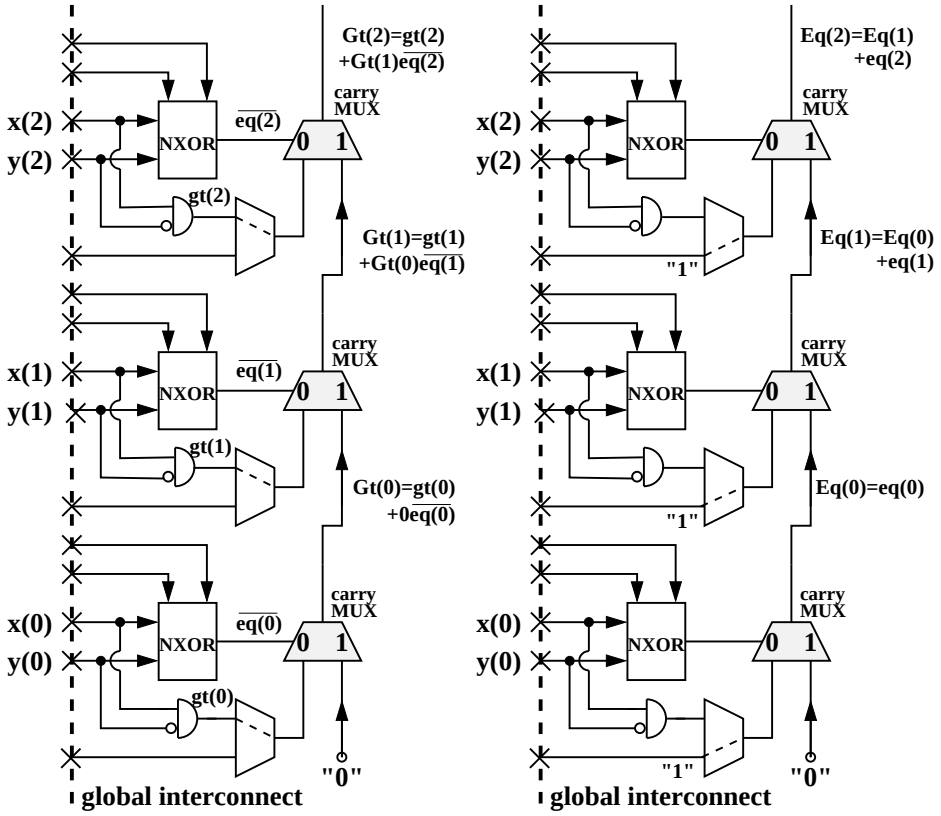
$$Eq(i) = eq(i) + Eq(i-1) \tag{5.17}$$

$$Gt(i) = gt(i) + Gt(i-1) \cdot \overline{eq(i)} \tag{5.18}$$

As a result, the Gt and Eq signals can also be emulated using the dedicated carry chain in the similar way as g_b and p_b signals. These mapping results at the component level are presented in Figure 5.7. While Figure 5.7(a) shows the dedicated carry chain used to emulate the Gt signal Figure 5.7(b) shows the chain emulating the Eq signal. Because of the $\overline{eq(i)}$ term in Equation 5.18, *NXOR* logic function is needed. To map Gt signal, "0" is fed to the 1-input of the first MUX, and the output of the dedicated *AND-with-one-inverted-input* gate is fed to the 0-input of every MUX. For instance, if $\overline{eq(i)}$ is 0, 0-input of the MUX is selected; otherwise, 1-input is selected, which emulates the Equation 5.18. As noticed, we need *AND-with-one-inverted-input* gate to be able to implement the comparator unit using the dedicated carry chain and this can not accomplished in current Xilinx FPGA. However, *CryptoRA* can accommodate this need, as will be presented in the later section. Similarly, to map Eq signal, "0" is fed to the 1-input of the first MUX, "1" is fed to the 0-input of every MUX, and the MUX is also controlled by *NXOR* logic function. For this reason, sharing-one-LUT feature in *CryptoRA* can be applied on implementing comparator unit, which is discussed later in this chapter.

According to the Equation 5.15 and the Equation 5.16 for the $eq(i)$ and $gt(i)$ signals, respectively, there is a condition that $eq(i)$ and $gt(i)$ never occured. That is, $eq(i)$ is 0, meaning $x(i)$ and $y(i)$ at bit-i_{th} position is the same, and $gt(i)$ is 1, in which $x(i)$ must be 1 and $y(i)$ must be 0. The later case contradicts the former case. Therefore, a mutually exclusive property between the $EQ(i)$ and $GT(i)$ signals is held true. Because of such property, it makes the equations for the EQ and GT signals mapped to Xilinx dedicated carry chain possible.

Moreover, when comparing two n-bit unsigned integers, the critical path would be the LUT delay plus the delay from n number of MUXes in the dedicated chain, which essen-



(a) GT flag mapped onto the dedicated carry chain

(b) EQ flag mapped onto the dedicated carry chain

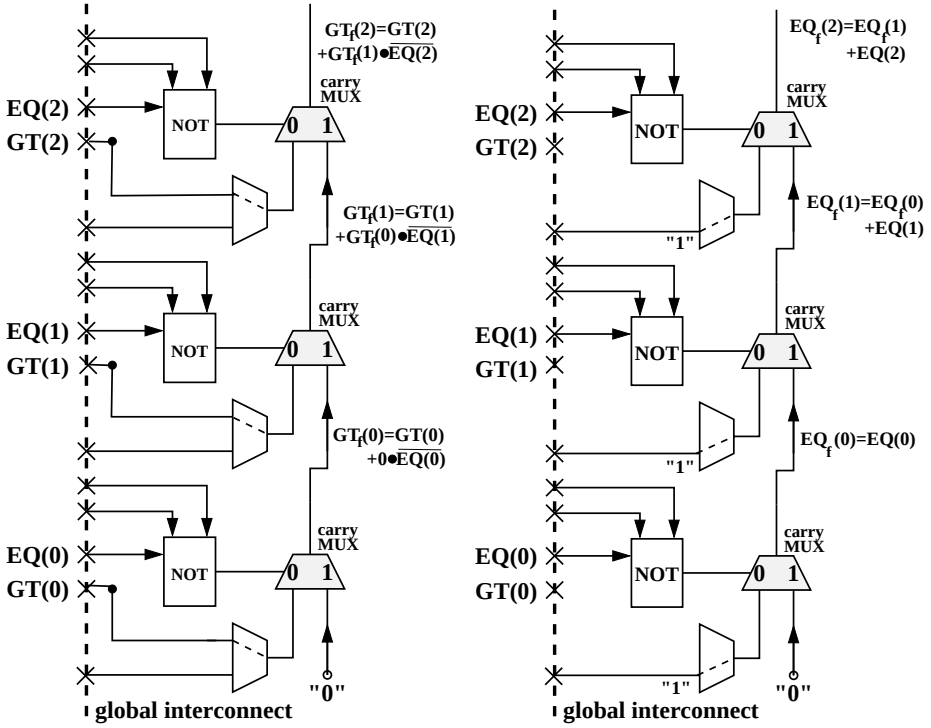
Figure 5.7: The generation of GT and EQ flags in comparator unit using dedicated carry chain

tially a ripple carry delay for adding two *n*-bit unsigned integers. To decrease the delay, a *n*-bit block can be divided into j blocks, each has length of *i*-bits, since there is no dependency on the sequence of the comparison is operated. Therefore, the $EQ(j)$ and $GT(j)$ signals from the j_{th}-block can be calculated in parallel and are then passed as the inputs to generate the final EQ ($EQ_f(m)$) and GT ($GT_f(m)$) flags. The equation for the final EQ and GT signal are shown in Equation 5.19 and 5.20.

$$EQ_f(m) = EQ(j) + EQ_f(m-1) \tag{5.19}$$

$$GT_f(m) = GT(j) + (GT_f(m-1) \cdot \overline{EQ(j)}) \tag{5.20}$$

With no change in the equations structure, the logic that we need to generate the final EQ and GT signals is different from that in the j-blocks since the inputs are no longer the $x(i)$ and $y(i)$. Instead, the inputs are now the $EQ(j)$ and $GT(j)$ signals generated from each j_{th}-block. The logic function in LUT needed for the selection signal of the each MUX in both chains is now $\overline{EQ(j)}$. This is why $EQ(j)$ signal is inverted. Again, the GT_f signal is emulated in the Figure 5.8(a), having the carry chain with "0" fed to the 1-input of first MUX and $GT(j)$ fed to the 0-input, and the EQ_f signal is emulated in Figure 5.8(b), having the carry chain with "0" fed to the 1-input of first MUX and "1" fed to the 0-input.



(a) final GT signal mapped onto the dedicated carry chain

(b) final EQ signal mapped onto the dedicated carry chain

Figure 5.8: The generation of final GT and EQ flags in comparator unit using dedicated carry chain

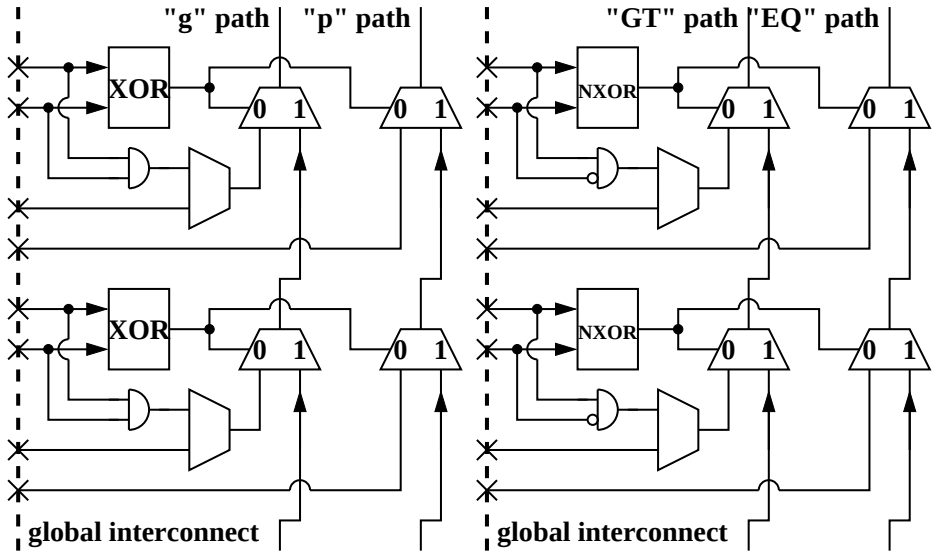
From the equation aspect, if $EQ(j)$ is 1, the second term of the Equation 5.20 is insignificant and the first term then determines the output. This is when the 0-input of MUX is connectetd to the output of MUX in position *m*. In the case when $EQ(j)$ is 0, $GT(j)$ will also be 0. Therefore, the second become the dominant term. This is when the 1-input of MUX is connected to the output of MUX in position *m*. As for the final EQ signal, same logic function from LUT ($\overline{EQ(j)}$) is used, with 0-input of MUX connected to $EQ(j)$ and 1-input of MUX connected to 0, which is shown in Figure 5.7(b). Such connection ensures that the second term of the Equation 5.19 is the dominant term to determine the output when $EQ(j)$ is 0.

As now, we have seen how to make use of the dedicated carry chain to compute both Carry-Lookahead Adder and comparator unit. In the next few sections, we show how the improvements in terms of critical path, area consumption can be made with the features provided in *CryptoRA*.

5.4 Two Dedicated Carry Muxes Using One LUT

To build a cryptography-oriented FPGA, we first observe that heterogenous LUT-based logic functions are of second importace. Long addition is much more important for cryptography. Thus, a major goal would be to improve the addition performance at the expense of heterogenous logic function implementation performance. Second, FPGA active logic uses 1%, configuration memory uses 9%, and interconnection network uses 90% of the die area [10], [11], [12]. Thus, we propose to increase the FPGA granularity such that a LUT drives two dedicated MUXes, as shown in Figure 5.9(a) and in Figure 5.9(b) for carry-lookahead adders and fast tree-based comparator unit, respectively. In Figure 5.9(a), the first dedicated carry chain is used to emulate the block-level *generate* signal ($g_b(j)$) while the 2nd chain is to emulate the block-level *propagate* signal ($p_b(j)$). Also, in Figure 5.9(b), the first dedicated carry chain is used to emulate the *Gt* signal and the 2nd chain

is to emulate the Eq signal for comparator unit.



(a) LUT for 2 MUXes in CLA (b) LUT for 2 MUXes in comparator

Figure 5.9: LUT for 2 MUXes

Such connection was realized by the fact that in both CLA and tree-based comparator unit implementations same LUT logic function is used, (e.g., *XOR* in CLA and *NXOR* in comparator unit). Thus, it is clear that having such connection will have major benefit on the silicon area savings. For example, the block-level *generate* and *propagate* signals can now be implemented within one slice, instead of two, resulting 50% saving on the area consumption. Similar result applies on the comparator unit implementation as well.

5.5 Horizontal Dedicated Path

As mentioned, although carry-lookahead addition is given architectural support in Xilinx architecutres, the block-level *generate* and *propagate* signals are still routed through the global interconnect. To improve the propagation of these signals as well as of other carry and/or carry-related signals, we also propose to extend the current dedicated carry chains

running vertically over the horizontal direction as well, as presented in Figure 5.10. This way, the *generate* and *propagate* signals can use a dedicated, thus fast, interconnect.

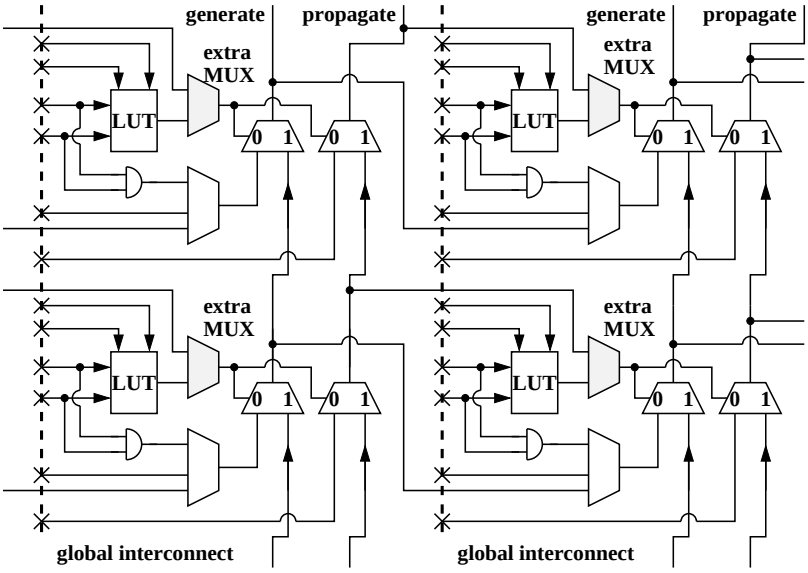


Figure 5.10: Carry network extended horizontally.

To demonstrate this idea on CLA, 1-level CLA is used as an example and is presented in Figure 5.11. It shows that having The block-level *generate* ($g_b(j)$) and *propagate* ($p_b(j)$) signals on the horizontal dedicated path to generate block-level *carry* signals can bypass the LUT and creates zero delays on the propagation. As seen, ”0” is fed to the 1-input of the first MUX and $g_b(j)$ is fed to the 0-input. In this case, the 2nd dedicated carry path is idle because to generate the sum bits, it requires different logic function.

Moreover, to fully take advantage of this new proposal on extending the dedicated carry chain over the horizontal direction in comparator unit, the $GT(j)$ and $EQ(j)$ signals from each j_{th} block can be propagated using these horizontal dedicated carry path to the $GT_f(m)$ and $EQ_f(m)$ chain. However, to recall the $GT_f(m)$ equation in Equation 5.20, the $EQ(j)$ signal is inverted. Therefore, each incoming $EQ(j)$ signal to the $GT_f(m)$ and $EQ_f(m)$ chain would need to be inverted in order to produce the correct result. Thus, an additional

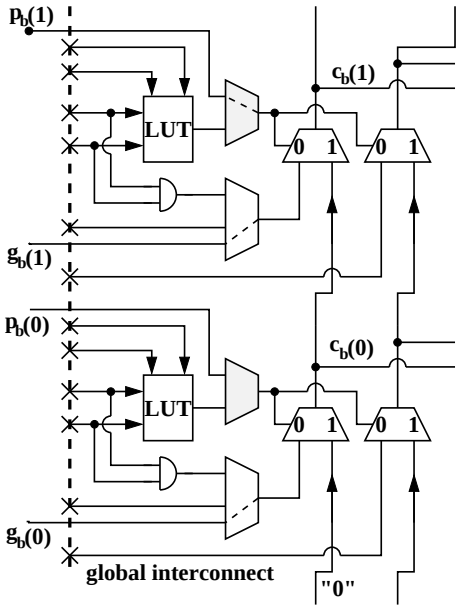


Figure 5.11: 1-level CLA using Horizontal dedicated path demostration.

dedicated input of the inverted *propagation* signal to the *extra MUX* in Figure 5.10 is required. Again, the GT_f signal is emulated in the 1st carry chain with ”0” fed to the 1-input of first MUX and $GT(j)$ fed to the 0-input, and the EQ_f signal is emulated in the 2nd carry chain with ”0” fed to the 1-input of first MUX and ”1” fed to the 0-input. As noticed, comparator unit is benefited more from this feature as both GT_f and EQ_f signals require same selection signal. Since the *extra MUX* is set at boot time, adding an additional input on this MUX does not increase the propagation delay. On the other hand, adding the an inverter does introduce very small delay on the path, but only when that input is used.

To extend the chain horizontally, one extra MUX is needed. It is worth mentioning that some FPGAs already have that MUX, see for example Virtex-II [52], so the latency will not be significantly affected on such FPGAs. With respect to Figure 5.13, there are two ways to add a MUX: (i) extending the carry MUX (Fig. 5.13-(a)), or (ii) additional MUX per se (Figure 5.13-(b)). By using the unfolded-MUX Level-Restoring Buffer (UMUX-LRB) de-

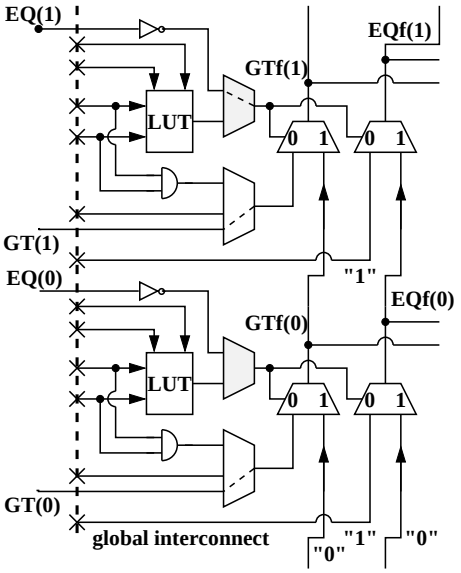


Figure 5.12: Comparator unit using Horizontal dedicated path demostration.

scribed in our previous paper [31] to improve the switching characteristics, we determined that the later approach shown in Figure 4-(b) is 40% faster. Therefore, extending the carry MUX is not considered any longer.

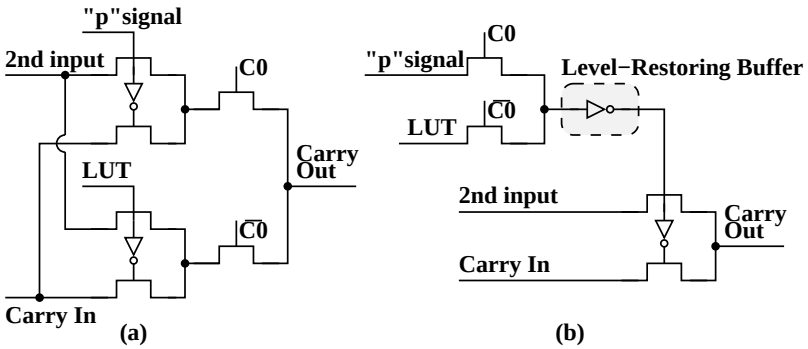


Figure 5.13: Solutions in adding an extra MUX.

5.6 Splitting LUT Structure

There are two major limitations of the fast carry-lookahead network presented in Figure 5.2. First, the propagation through the LUT to the carry MUX is much slower than the propagation through the *AND* gate and the subsequent multiplexor. This means that toggling from the propagate chain to the generate chain is slow, which limits usefulness of the dedicated chain. Second, the dedicated *AND* gate can only be used for carry-lookahead addition. To reduce the toggling time, we propose to split the LUT, as shown in Figure 5.14 and 5.15. The upper half of the LUT is configured to use only three out of the four inputs, and its output is connected between the third and fourth stages of the intial LUT. The splitting is performed by forcing the fourth LUT input to zero, such that only the lower half of the initial LUT is propagated to the initial output. According to our transistor-level simulations, the critical path is reduced such that the toggling delay is greatly reduced, and so is the critical path for the dedicated carry chain.

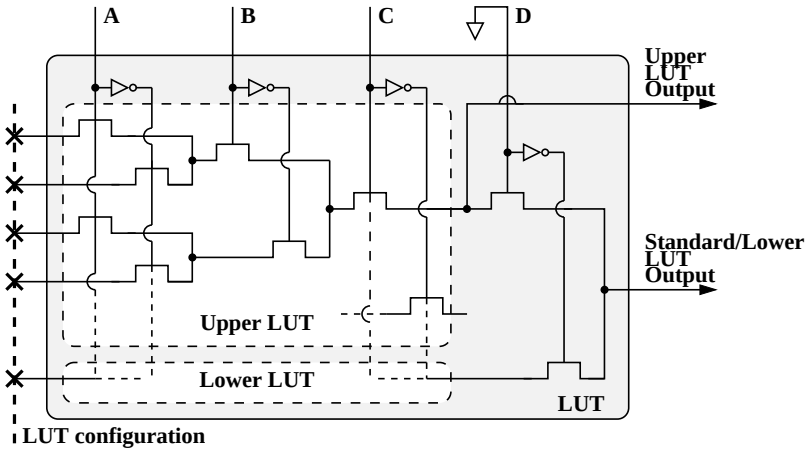


Figure 5.14: Split LUT - transistor level.

It is important to mention that this has a significant advantage in terms of flexibility and silicon area, since the *AND* gate has been emulated by the lower (unused) LUT half. For example, carry-lookahead subtraction can be performed by emulating an *XOR-AND*

gate in the lower half of the LUT. Having such flexibility is crucial in MMM since it is impossible to implement subtraction function using CLA structure in current Xilinx FPGA even though the addition function is supported. With the additional computation capability, it is possible to build an ALU using carry-lookahead arithmetic. In sum, as the current Xilinx Virtex-II Pro FPGA only support carry-lookahead addition using dedicated *AND* gate, having this splitting LUT feature, we can now support, but not limited to, carry-lookahead subtraction, and the *AND-with-one-inverted-input* gate.

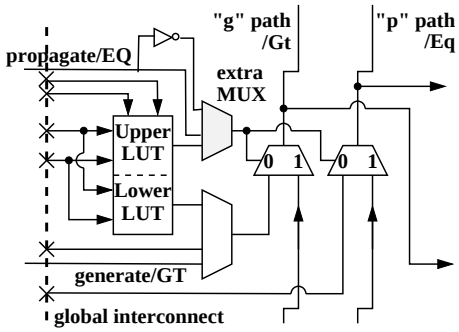


Figure 5.15: Splitting LUT

Up to this point, our RISC-augmented-with-FPGA reconfigurable solution for computing EC point multiplication has been completely presented. This includes the EC point multiplication software implementation in the last chapter, and MMM hardware unit on both current Xilinx and *CryptoRA* FPGA from this chapter. The measured and estimated results in terms of the performance and the area consumption are presented in the next chapter.

Chapter 6

Reconfigurable Solution Implementation and Simulation results for Cryptography

This chapter is the showcase to present both simulated and estimated results for all the implementations that have been covered in this thesis. It starts with the developing tools used in the designing, implementing, and testing phases. It is followed by the simulation and estimation result in Section 6.2, which includes many subsequent sections showing results for different implementations.

It begins with the pure software simulation results from both MIRCL library and the developed C-level program running on PC in Subsection 6.2.1, followed by the simulation result for the C-level program running on MicroBlaze embedded system in Subsection 6.2.2. The simulation result and the resulting speedup for the hardware Montgomery Modular Multiplication (MMM) unit on FPGA controlled by the core MicroBlaze embedded processor is presented in Subsection 6.2.3. The simulation result for the Carry-Save Adder (CSA) used in the MMM unit on Xilinx FPGA is presented in Subsection 6.2.4. Since the *CryptoRA*'s architecture at this point is only the prove of concept, the results on the performance and area consumption for *CryptoRA* can only be estimated and then compared to the simulated results of implemented units on Xilinx FPGA. These simulated and estimated results of the various adders and the comparator unit are shown in Subsection 6.2.5, and 6.2.6, respectively. Finally, the estimated performance and area consumption results using *CryptoRA* is presented and compared again with that using Xilinx FPGA.

6.1 Developing Tools

The software is first programmed, compiled and ran in the Microsoft Visual Studio 2005 Team Suite Edition, only in which the profiling is supported. Then it is ported to run on the MicroBlaze embedded system after the hardware platform is configured using the Xilinx Platform Studio (XPS) SDK version 9.1 [49]. The output of the program is then viewed on the HyerTerminal.

The accelerated hardware unit is first coded by VHDL in the Xilinx ISE (Project Navigator) v9.1.03i [53]. Each module of the hardware unit is verified functionally as well as post-place-and-route by waveform simulations in Mentor Graphics ModelSim XE [30]. To augment the software program with an accelerated hardware unit, a custom IP wizard in the XPS is used. The mechanism for the data transferring between MicroBlaze and the hardware unit is the proprietary connection called, MicroBlaze Fast Simplex Link (FSL) [50]. The performances of various implementations are measured using the OPB Hardware Timer [53] in cycle count.

6.2 Simulation and Estimation Results

6.2.1 Software Simulation Result on Pentium

To show how our C-level program is performed in comparison with an assembly-optimized library, called MIRCEL, the meansurement from profiling in cycle count for key length of 160, 192, and 224-bit EC point multiplication is presented in Figure 6.1.

The Pentium setting is the 64-bit Intel processor running at 1.8 GHz. The result from the MIRCL is presented on the left-side of the histogram pair, and that from C-level program is on the right-side of the pair. With the performance of computing one EC point multiplication within $2\times$ scale slower, which is accepted when compared with MIRCL, it reveals the comparability of our C-level software implementation. Another important piece of information it also shows is the consistancy on the cycle count of the modular multipli-

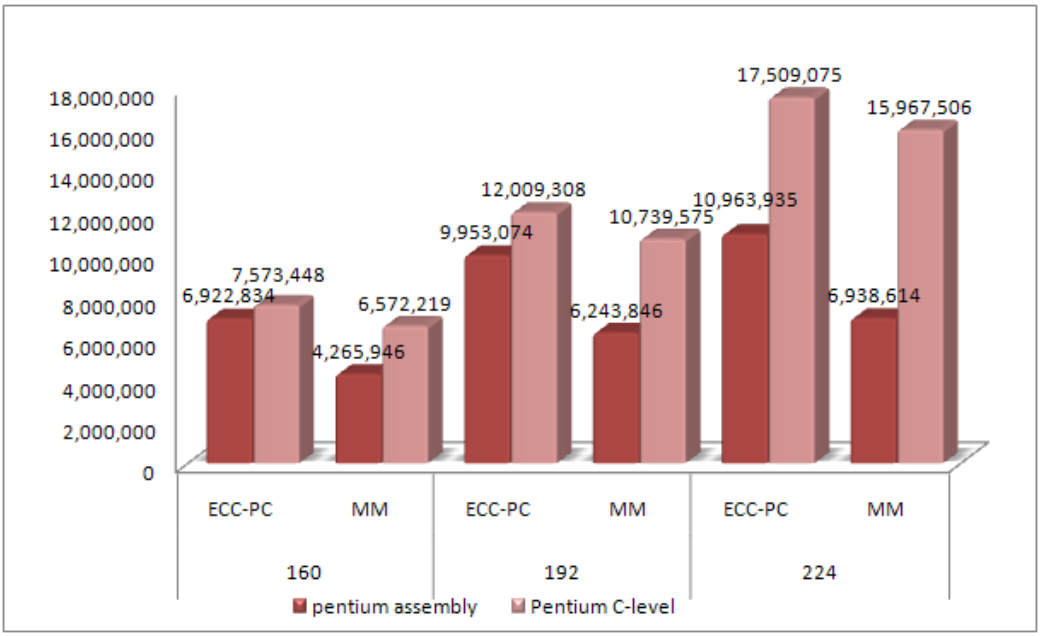


Figure 6.1: Cycle count of MIRCLE and my program Running on Pentium processor.

cation operations relative to that of the EC point multiplication. In other words, over 60% of the computing time spent in one EC point mulitplication is in modular multiplication operations in MICRL while over 90% in our C-level software. The difference on these percentages is because of the differences in the modular multiplication implementation. However, these percentages still show why modular multiplication is the core operation in public-key cryptogrpahy.

6.2.2 Software Simulation Result on MicroBlaze

In this subsequent section, we look at what happen to the performance of our C-level software program after it is ported and ran on the MicroBlaze embedded system and is then compared to that of the same code running on Pentium. Figure 6.2 shows the pure-software performance in cycle count.

As seen, the cycle count required in MicroBlaze embedded system increases by approx.

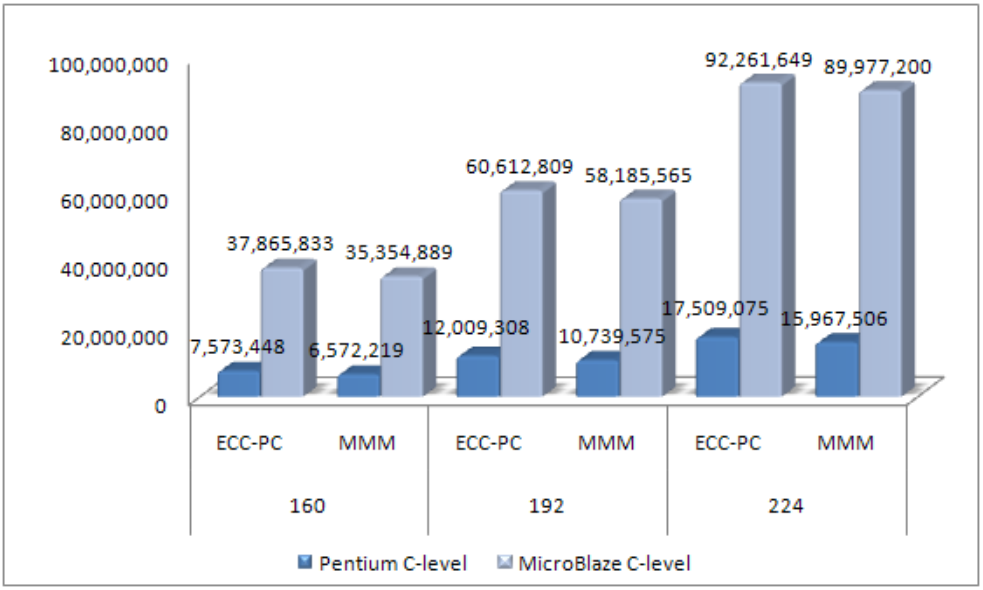


Figure 6.2: Cycle Count of C-level Program Running on Pentium and MicroBlaze Processor.

4× ˜ 6×. This result is expected and is mainly due to the combination of the less powerful processor and compiler compared to Pentium. Moreover, because of the increase in the cycle count, the relative computing time of modular multiplication is increased to approx. 95% ˜ 97%, which tells that it is crucial to have high-performance modular multiplier in order to reduce the overall EC point multiplication computing time. Thus, this is why the modular multiplication is given a hardware support with Montgomery Modular Multiplier unit. One thing worth mentioning is that while the cycle count obtained in Pentium is from the profiling feature in WVS, that obtained in MicroBlaze is from the OPB-Hardware Timer. This is because there is a software bug in the XPS developing tool such that we are not able to use the profiling feature.

6.2.3 Hardware Montgomery Modular Mutiplier Simulation Result on MicroBlaze

After successfully connecting the FPGA-based MMM hardware unit to the FPGA-based RISC MicroBlaze embedded processor via the FSL interface, the reconfigurable solution for computing EC point multiplication on Xilinx FPGA Virtex-II Pro is completed. The performance in cycle count is given in Table 6.1 and is compared with the pure-software solution.

Table 6.1: Cycle count for Montgomery modular multiplication.

Key length	MicroBlaze C-level	FPGA Virtex-II Pro	Speedup
160	35,354,889	953,903	37X
192	58,185,565	1,312,530	44X
224	89,977,200	1,979,888	45X

According to this table, the speedup obtained for one Montgomery Modular Multiplication operation in key length of 160, 192, and 224 bits are $37\times$, $44\times$, and $45\times$, respectively. This translates to a speedup of $15\times$ to $22\times$ in one EC point multiplication as the cycle count difference between the EC point multiplication and MMM remains. It is worth mentioning that with the substantial reduction in cycle count due to the MMM hardware support, the time spent on the communication become more noticeable as the number of calls to MMM unit per EC point multiplication is in the range of thousands on top of the issue, which is the large number of blocks needed to be tranferred as well. In the future work chapter, this issue will be covered.

Even though by deploying hardware support for MMM results in substantial speedup, this MMM hardware unit is not the optimal implementation for FPGA, mainly because: 1. the computing tiles are idle while waiting for the $x(i)$ and q_i signals to propagate throughout the array in the CSA implementation. 2. RCA becomes the bottleneck on the system frequency as the key length increases. 3. the fast tree-based comparator unit has $O(log(n))+1$

levels of global interconnects in FPGA platform. In the next few sections, we prove with simulation results showing that the above-mentioned facts are issues to the FPGA-based MMM hardware unit. In addition, the simulated and estimated results of the solutions to these issues are presented.

6.2.4 2 3-to-2 Carry-Save Adder Simulation Results on Xilinx FPGA

To map the two-level CSA design shown in Figure 5.5 for theMMM unit in FPGA platform, it requries two 4-to-1 LUTs for carry and sum bits in the 1st level, and similarly in the 2nd level. This leads to the approximation of the critical path of the CSA, which is twoLUTs, connected through interconnect and the result is fed back through interconnects. This theoretical delay generalization can be expressed in the following equation:

$$\text{delay}_{\text{TOL}} = \text{delay}_{\text{LUT}} + \text{delay}_{\text{INTER}} + \text{delay}_{\text{LUT}} + \text{delay}_{\text{INTER}}, \tag{6.1}$$

where $delay_{LUT}$ is constant and is $0.4ns$ [51]. $delay_{INTER}$ is the global interconnect delay and can be ranged from 0.8 to 7 ns, depending how close the blocks are routed to each other.

Thus, the critical path of this CSA combination logic is determined by the delay of two 4-to-1 LUTs and two times the delay of the global interconnects. Figure 6.3 reveals the critical path in *ns* measured from the timing waveform for the 2-level CSA of 160 to 256-bit key length.

It is noticed that the critical path increases with the increase in key length. This is because the number of fanouts of the $x(i)$ and m_i signals in Figure 5.5 has the proportional relation with the key length. It takes longer to charge all the capacitors as the number of fanout increases. As a result, the critical path of the combinational logic increases. Furthermore, These timing waveform measurements do not match the synthesized result. For instance, the critical path of 192-bit 2-level CSA is $2.596ns$ in synthesis report while it is measured to be $8.631ns$ in the waveform. This mismatch is due to that the net delays

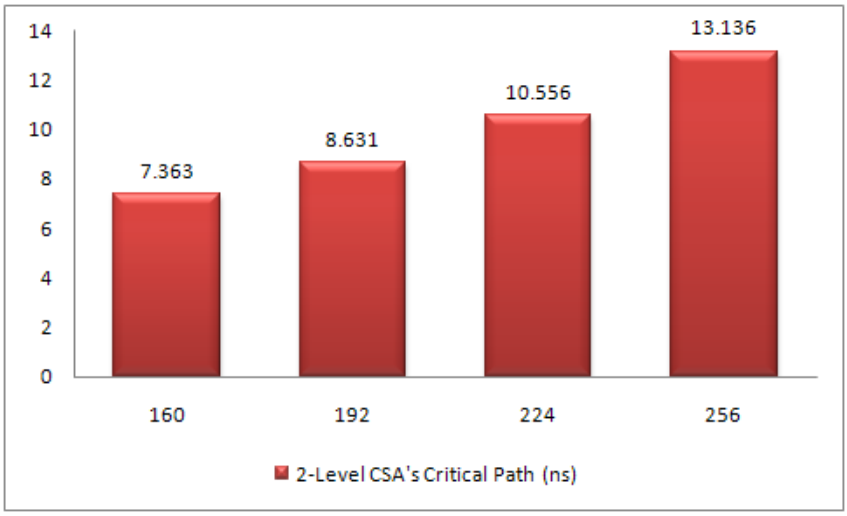


Figure 6.3: Critical path for 2-level CSA of various key length.

created by the large fanouts are much longer on the FPGA fabric after place and route than the net delay given in the synthesis report. As discussed, we rather to have the computing tiles to perform some useful tasks while waiting for $x(i)$ and q_i to propagate. Hence, we look into the CLA structure and compare it with the fast carry skip and RCA in terms of the performance and the area consumption. This solution can also be applied on the RCA used for the subtraction at the end of the Algorithm 4.1.

6.2.5 Various Adder Simulated and Estimated Results on Xilinx and *CryptoRA*

To overcome the issues created by the FPGA-based MMM hardware unit, the conservative estimations on the performance and area consumption of mapping required operations on *CryptoRA* are presented, which are then used to compare to the Xilinx FPGA-based implementations. Table 6.2 shows the measured critical path of long-integer addition in *ns* for key length from 192-bit to 1024-bits in three different adder implementations on Xilinx Virtex-II Pro FPGA. They are high-speed carry-skip adder, 1-level Carry-Lookahead Adder (CLA) and Ripple-Carry Adder (RCA), in which are implemented using the dedi-

cated carry path supported in the Xilinx FPGA chip. It also has the estimated critical path of CLA in *CryptoRA*.

Table 6.2: Critical path in *ns* for CLA and RCA

key length	CSkA on Xilinx	CLA on Xilinx	CLA on CryptoRA	RCA on Xilinx
192	7.36 ns	8.54 ns	2.07 ns	10.73 ns
224	7.99 ns	9.32 ns	2.15 ns	11.09 ns
256	7.91 ns	9.72 ns	2.23 ns	13.82 ns
384	8.18 ns	9.47 ns	2.55 ns	21.31 ns
521	12.10 ns	10.32 ns	2.95 ns	36.80 ns
1024	19.10 ns	11.73 ns	4.15 ns	69.47 ns

As expected, a ripple carry addition is still an issue for computing long integer addition; the critical path increases linearly with the increase in key length. Thus, if RCA is used as an adder/subtractor as we used in our initial MMM hardware unit, the system frequecy goes down drastically as the key length increases. On the other hand, the growth in the critical path from the fast adder structures are much slower. Thus, considering fast-adder structure is a good approach for constant to little decrease in the system frequency for the same key length. According to Figure **??**, even though carry-skip adder show shorter critical path up to 400 bit, due to the irregularity that it suffers from, CLA is a better option for the long integer addition/subtraction.

Even though the delay is much reduced in comparison with RCA, because of the CLA's parallelism property, the delays from LUT, dedicated carry Muxes, and the global interconnects still exist and can be generalized in Equation 6.2.

$$\begin{aligned} \text{delay}_{\text{TOL}} = &[\text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize})] \\ &+ \text{delay}_{\text{INTER}} + [\text{delay}_{\text{LUT}} + \text{delay}_{\text{MUX}} * \textit{numBlk})] \\ &+ \text{delay}_{\text{INTER}} + [\text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize})], \end{aligned} \tag{6.2}$$

where $\text{delay}_{\text{MUX}}$ is a constant and is $0.313ns$ [51]. While the *blkSize* variable represents the size of a block, the *numBlk* variable represents the number of blocks, which depends on the key length and how the blocks are being divided. The frist square bracket represnts the delay created by the block-level *generate* and *propagate* blocks. The second square bracket represents the delay created by the block-level *carry* blocks. And finally, the last square bracket represents the delay created by the sum-bit blocks.

Also, the results for CLA on Xilinx FPGA are obtained after the modification described in Chapter 5 is applied to prevent our dedicated carry path from being cut. Our proposal on extending the dedicated path over the horizontal direction will decrease the the major delay from the global interconnects. This is because that these signals (e.g., block-level *generate*, *propagate*, and *carry*) are now travelling on the dedicated path, which has zero delay. Moreover, the delay from the LUTs can also be cut; the number of LUTs is reduced from 3 to 1 as the signals riding on the horizontal dedicated path do not go through LUT to generate the outputs. The issue of the path being discontinued described in Chapter 5 is solved as the 2nd path has been modified to dedicated path, like the vertical dedicated carry path. Both the simulated and estimated critical path in 1-level CLA for 192-bit to 1024-bit key length is shown in Figure 6.2. As explained, the global interconnects delay and 2 out of 3 LUT delay can be eliminated in the CryptoRA FPGA architecture. Hence, the estimated total delay is modified and generalized in Equation 6.3.

$$\begin{aligned}\text{delay}_{\text{TOL}} = &[\text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize})] \\ &+ (\text{delay}_{\text{MUX}} * \textit{numBlk}) + (\text{delay}_{\text{MUX}} * \textit{blkSize})\end{aligned} \tag{6.3}$$

To further demonstrate how the numbers are estimated, we use 192-bit key length as an example. In this example, we set *blksize* to be 16 and *numBlk* is then 12 $(192/16)$. This would yield the total estimated delay of 2.073, which is just about $4\times$ shorter in critical path than the CLA implemented on the current Xilinx FPGA.

However, according to Table 6.3, the better performance in CLA comes with the penalty

of 4× more on the area consumption than that of RCA. This issue can be alleviated by another proposal of ours: increasing in the FPGA granularity. This allows the *generate* and *propagate* signals to be computing using the same LUT, resulting estimated 30% reduction on the area consumption of 1-level CLA implementation, shown in column 4.

Table 6.3: Area consumption in slice for CLA and RCA

key length	CSkA on Xilinx	CLA on Xilinx	CLA on CryptoRA	RCA on Xilinx
192	416	423	296	96
224	480	493	345	112
256	549	564	394	128
384	824	819	573	192
521	1168	1092	764	272
1024	2200	2183	1528	512

6.2.6 Comparator Unit Simulated and Estimated Results on Xilinx and *CryptoRA*

Two versions of the fast tree-based comparator unit are desribed in Chapter 5. The orignial version is implemented and measured from the timing waveform while the result of the proposed version is estimated since the current Xilinx FPGA does not support the architecture. The resulting critical path, and the area consumption for both versions are presented and compared in Table 6.4. The theoretical delay generalization for the original comparator unit is expressed in Equation 6.4.

To estimate the comparator's critical path in *CryptoRA*, its delay is derived from the theoretical delay of the original comparator, which is expressed in Equation 6.4.

$$\text{delay}_{\text{TOL}} = \text{delay}_{\text{LUT}} + [\text{delay}_{\text{INTER}} + log(n) * (\text{delay}_{\text{LUT}} + \text{delay}_{\text{INTER}})] \qquad (6.4)$$

The first $delay_{LUT}$ is created by the generation of the EQ and GT arrays. The rest are the delays created from each level of the comparator unit of $log(n)$ and the intermediate

Table 6.4: Critical path in *ns* and area consumption in slice for comparator unit.

Key length	Virtex-II Pro	*CryptoRA*	Virtex-II Pro	*CryptoRA*
192	7.42 ns	1.51 ns	442	102
256	8.29 ns	1.76 ns	590	136
521	9.17 ns	2.31 ns	1178	281

global interconnects. Due to the growth rate is $O(log(n))$, the delays for key-length 192-bit to 255-bit are the same. Thus, the results are shown based on every growth in level, not in key length. As seen, the delay is substantially reduced in *CryptoRA*. This is because the comparison function has been emulated using the dedicated carry chain, the number of levels is reduced from the original $O(log(n))+1$ to just 1, in which is connected through the horizontal dedicated path. As the signals on the horizontal dedicated path are selected as the inputs, the second-level LUT is bypassed, resulting in saving one LUT delay ($0.313ns$). However, each signal would need to go through a dedicated inverter on the dedicated path, which is simulated to be a delay of $0.08ns$. This new delay for comparator unit on CryptoRA can be generalized in Equation 6.5.

$$\begin{aligned} \text{delay}_{\text{TOL}} = \text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize}) \\ + \text{delay}_{\text{INVERTER}} + (\text{delay}_{\text{LUT}} * \textit{numBlk}) \end{aligned} \tag{6.5}$$

As noticed, the majority of the delay are now contributed by the dedicated carry Muxes, which is very cheap in comparison with that of global interconnects.

In addition, sharing the same LUT feature benefits even more on the area consumption for the comparator unit because it utilizes 100% both dedicated carry paths in all its computation. As seen in Table 6.4, a significant reduction on the area consumption is the result of this feature. For instance, the number of slice usage is reduced from 590 in original implementation on Xilinx Virtex-II Pro FPGA to 136 in proposed implementation on CryptoRA architecture. The estimated value is calculated as the following: the 256-bit

comparator unit is divided into 16 16-bit blocks, occupying 8 slices per block. As a result, this layer would require (8×16) slices and the block in the next layer to generate the final GT and EQ signals would require another 8 slices, resulting total of 136 slices. It is worth mentioning that since it is an optimization problem to find the optimal configuration of the comparator unit, the result shown in the Table 6.4 may not be optimal, rather illustrative.

Chapter 7

Conclusions

Bibliography

[1] Altera Corp., San Joes, CA. *Stratix Device Handbook*, 2006.

[2] Altera Incorporation. http://www.altera.com/.

[3] Lejla Batina, Geeke Bruin-Muurling, and Siddika Berna Ors. Flexible hardware design for rsa and elliptic curve cryptosystems. In *CT-RSA*, pages 250–263, 2004.

[4] Lejla Batina and Geeke Muurling. Montgomery in practice: How to do it more efficiently in hardware. In *CT-RSA '02: Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, pages 40–52, London, UK, 2002. Springer-Verlag.

[5] Duncan A. Buell and Kenneth L. Pocek. Custom computing machines: an introduction. *J. Supercomput.*, 9(3):219–229, 1995.

[6] M. McLoone C. McIvor and J.V. McCanny. Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures. In *in: Proceedings of the 37th Asilomar Conference on Signals, Systems, and Computers*, pages 379–384, January 2003.

[7] Canadian Micro-electronics Corporation (CMC Microsystems). http://www.cmc.ca/.

[8] Certicom Inc. Online Elliptic Curve Cryptography Tutorial.

[9] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT '98: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security*, pages 51–65, London, UK, 1998. Springer-Verlag.

[10] Andre DeHon. Reconfigurable architectures for general-purpose computing. Technical Report AITR-1586, Massachusetts Inst. of Technology, 1996.

[11] Andre DeHon. Balancing interconnect and computation in a reconfiguable computing array (or, why you don't really want 100 In *FPGA*, pages 69–78, 1999.

[12] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[13] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[14] M. E. Ciftgibasi E. Savag, A.F. Tenca and C.K. Koc. Novel multiplier architectures for gf(p) and gf(2n). In *IEE Proceedings - Computers and Digital Techniques*, pages 147–160, March 2004.

[15] Hans Eberle, Nils Gura, Sheueling Chang Shantz, Vipul Gupta, Leonard Rarick, and Shreyas Sundaram. A public-key cryptographic processor for rsa and ecc. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pages 98–110, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Standards for Efficient Cryptography Group. Sec2 - recommended elliptic curve domain parameters, September 2000.

[17] National Institute for Standards and Technology. Recommended Elliptic Curves For Federal Government Use, (1999). Available at http://crsr.nist.gov/encryption/.

[18] James Goodman and Anantha Chandrakasan. An energy efficient reconfigurable public-key cryptograhpy processor architecture. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 175–190, London, UK, 2000. Springer-Verlag.

[19] Vipul Gupta, Douglas Stebila, Stephen Fung, Sheueling Chang, Nils Gura, and Hans Eberle. Speeding up secure web transactions using elliptic curve cryptography. In *11th Ann. Symp. on Network and Distributed System Security – NDSS 2004*. Internet Society, February 2004.

[20] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[21] S. Hauck. The roles of fpgas in reprogrammable systems, 1998.

[22] Scott Hauck, Matthew M. Hosler, and Thomas W. Fry. High-performance carry chains for FPGAs. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 223–233, New York, NY, USA, February 1998. ACM Press.

[23] IEEE P1363 Draft Standard. Standard Specifications for Public Key Cryptography, 1998.

[24] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet Physics-Koklady*, volume 7, pages 595–596, 1963.

[25] Cetin K. Koc and Tolga Acar. Montgomery multiplication in GF(2K). *Des. Codes Cryptography*, 14(1):57–69, 1998.

[26] Ruby B. Lee, Zhijie Shi, and Xiao Yang. Cryptography efficient permutation instructions for fast software. *IEEE Micro*, 21(6):56–69, 2001.

[27] William Mangione-Smith and Brad Hutchings. Configurable computing: The road ahead. In *Reconfigurable Architectures: High Performance by Configware*, pages 81–96, Chicago, 1997. IT Press.

[28] William H. Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner Hartenstein, Oskar Mencer, John Morris, Krishna Palem, Viktor K. Prasanna, and Henk A. E. Spaanenburg. Seeking solutions in configurable computing. *Computer*, 30(12):38–43, 1997.

[29] Ciaran McIvor, Máire McLoone, and John V. McCanny. Fpga montgomery modular multiplication architectures suitable for eccs over gf(p). In *ISCAS (3)*, pages 509–512, 2004.

[30] Mentor Graphics Incorporation. http://www.model.com/.

[31] Scott MILLER, Mihai SIMA, and Michael McGUIRE. Alternatives in designing level-restoring buffers for interconnection networks in field-programmable gate arrays. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 138–146, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Victor S. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, pages 417–426, 1985.

[33] Peter L. Montgomery. Modular multiplication without trial division. In *Mathematics of Computation*, pages 519–521, April 1985.

[34] Koblitz N. Elliptic curve cryptosystem. In *Mathematics of Computation*, pages 203–209, April 1987.

[35] S. Ors, L. Batina, and B. Preneel. Hardware implementation of elliptic curve processor over GF(p), 2002.

[36] Behrooz Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Oxford, UK, 2000.

[37] Jin Park and Jeong-Tae Hwang. Fpga and asic implementation of ecc processor for security on medical embedded system. In *ICITA '05: Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, pages 547–551, Washington, DC, USA, 2005. IEEE Computer Society.

[38] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[39] Michael Rosing. *Implementing Elliptic Curve Cryptography*. Manning Publications Co., Greenwich, CT, USA, 1999.

[40] Akashi Satoh, Y. Kobayashi, H. Niijima, Nobuyuki Ooba, Seiji Munetoh, and S. Sone. A high-speed small rsa encryption lsi with low power dissipation. In *ISW '97: Proceedings of the First International Workshop on Information Security*, pages 174–187, London, UK, 1998. Springer-Verlag.

[41] David Seal. *ARM Architecture Reference Manual*. ARM, 2000.

[42] Shamus Software Ltd. Multi-precision Integer and Rational Arithmetic C/C++ Library (MIRACL), 2005.

[43] Nigel P. Smart. How secure are elliptic curves over composite extension fields? *Lecture Notes in Computer Science*, 2045:30–??, 2001.

[44] Douglas R. Stinson. *Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, November 2005.

[45] Sun Microsystems Inc. Elliptic Curve Cryptography: The Next Generation of Internet Security, 2002.

[46] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[47] Xilinx Corp., San Joes, CA. *Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel*, May 2004.

[48] Xilinx Corp., San Joes, CA. *MicroBlaze Processor Reference Guide*, 2005.

[49] Xilinx Corp., San Joes, CA. *Platform Studio User Guide*, February 2005.

[50] Xilinx Corp., San Joes, CA. *Fast Simplex Link (FSL) Bus (v2.10a)*, November 2006.

[51] Xilinx Corp., San Joes, CA. *Virtex-II Pro and Virtex-II Pro X Platform FPGA User Guide*, 9.1i edition, 2007.

[52] Xilinx Corp., San Joes, CA. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, March 2007.

[53] Xilinx Incorporation. http://www.xilinx.com/.

Curriculum Vitae

Chi-Chun (Ambrose) Chu was borne in I-Lan, Taiwan on March 17th, 1981. He started his education journey in Canada when he was in the age of fifteen. He spent one year in ESL school, and then attended regular school in North Vancouver, where he recevied an award in ESL student of the year. He then moved with the host family to Osoyoos,B.C., where he completed his secondary education with three scholarships in 2000.

He obtained a bachelor degree in computer engineering with specialization in computer system in 2005. During those five years, he was with an IT company in Taiwan for three Co-ops, where he gained experiences in maintaining and troubleshooting systems at both user-end and server-end. He also spent one Co-op as a GUI optimization enginner in a lumber machine manufacturer in Salmon Arm, B.C.. His primiary work was to develop, improve and maintain the Graphical User Interface (GUI) of an automation software for a system using C/C++ programming language in MS Visual Studio environment.

He then continued pursuing his education in a master's degree program in computer engineering under the supervision of Dr. Mihai Sima, where he further refined his knowledge and skills in the technical aspect. His research focus was on the performance of public-key cryptography in embedded systems, in which the computing power, memory, and bandwidth are limited. In addition to his research, he was appointed as Teacher Assistant mnay times for different engineering labs, where he have received many positive feedback from his students which prove his teaching ability.

His research interests include software development and optimization for embedded system, reconfigurable computing, and field programmable gate array.

Partial Copyright License

Title of Thesis:

Embedded Reconfigurable Solutions for Cryptography

Author: _____

Chi Chun (Ambrose) Chu

Signed: August 31, 2006