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

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

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

Abstract

We first propose a reconfigurable processor, which consists of a MicroBlaze processor augmented with a Field-Programmable Gate Array (FPGA) to mitigate the computing time for public-key cryptography algorithms. We first consider Virtex-II Pro from Xilinx to analyze the potential solution of a Field-Programmable Custom Computing Machine (FCCM), which is composed of MicroBlaze augmented with a Virtex-II FPGA. We then propose a cryptography-oriented reconfigurable array, called *CryptoRA*, that efficiently supports long-word integer addition, subtraction and comparison. As a result, RISC processor can potentially be augmented with the *CryptoRA* rather than Virtex-II. The three main features that *CryptoRA* has are: (i) an increased granularity of the logic tile, (ii) the extension of the dedicated carry chain over the horizontal direction, and (iii) the incremental splitting Look-Up Table. According to our simulations, the CryptoRA-based FCCM provides a significant performance improvement over an optimized pure-software solution at an acceptable cost.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

The first honor, of course, went to my awesome supervisor, Dr. Sima, whom not only I had great time working for, but also have gained many tips from on how to approach and solve problems. He treated all his students with respects and always took our suggestions into consideration. I could not ask for anything more from him with what he had provided and could not be any happier working for a such supervisor. He also provided sufficient guidance along the way to ensure that my journey in completing my Master's degree had no surprises.

The second honor, without a doubt, went 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. Here, I would like to say Thank You, grandmom.

The third honor went to the rest of the family members, which also includes Andra and her mom, Robert, and the Sherwood family. Without my family's finicial support, and my brothers', Robert's, Andra's and her mom's support, this journey would have been more difficult. Because of my parents' great visions, I have received superior education here in Canada. Also, I was fortunate enough to meet great people, like Robert, Andra, and the Sherwood family who have provided me with family-like cares such that they all basically become like my second family here.

Last, but not least, this honor went to everyone I meet along this jorney, specially the colleages from the internship and the lab, including Jay Lu, Eugene, Scott, Ehsan, Farshad,

Hamed, Kaveh A., and many more. These are great people to work with. We had fun in the lab as well as outside the lab. They are all very intelligent and knowledgable people and are willing to share their knowledge with me. Hence, I have also learned much priceless knowledge from them. Thank you, guys.

Acronyms

AES Advanced Encryption Standard

ASIC Application-Specific Integrated Circuits

ASIP Application-Specific Instruction set Processors

BRAM Block RAM

CLA Carry-Lookahead Adder

CLB Configurable Logic Block

CSA Carry-Save Adder

CSeA Carry-Select Adder

CSkA Carry-Skip Adder

CryptoRA Cryptography-oriented Reconfigurable Array

DES Data Encryption Standard

DL Discrete Logarithms

DSA Digital Signature Algorithm

EC Elliptic Curve

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie Hellman

ECDL Elliptic Curve Discrete Logarithm

ECDLP Elliptic Curve Discrete Logarithm Problem

FCCM Field-Programmable Custom Computing Machine

FPGA Field-Programmable Gate Array

FSL Fast Simplex Link

IF Integer Factorization

LMB Local Memory Block

LUT Look-Up Table

MMM Montgomery Modular Multiplication

MMM unit Montgomery Modular Multiplier unit

NIST National Institute of Standards and Technology

OPB On-chip Peripheral Bus

PGP Pretty Good Privacy

RC Reconfigurable Computing

RCA Ripple-Carry Adder

RSA Rivest-Shamir-Adleman

SECG Standards for Efficient Cryptography Group

SSL Secure Socket Layer

XPS Xilinx Platform Studio

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 are performed in real-time on mobile terminals. There are two classes of cryptographic systems. The frist class is called private-key cryptosystem, which is computationally cheap but requires a secure way to obtain the private key among the communicating parties. The second class is called public-key cryptosystem, which is computationally demanding, mainly due to the long-integer and complex operations, but solves the limitation of private-key cryptosystem. The problem we focus on is to improve the performance in running public-key cryptography tasks on embedded systems.

The *de facto* public-key cryptography algorithms are Rivest-Shamir-Adleman (RSA) [44] and Elliptic Curve Cryptography (ECC) [37, 39]. The common operations employed by these algorithms are not directly supported by the integer-oriented architectures typically used in embedded systems, such as ARM [50], MicroBlaze [59], MIPS [25], and NIOS [2]. Therefore, the issues associated with the algorithms used in public-key cryptosystems have drawn the attention of many embedded engineers.

A common feature of traditional cryptographic schemes is the operation on long-integer data, e.g., 160 to 521 bits for ECC, and 1024 to 2048 bits for RSA [57]. While executing typical cryptography operations, such as modular multiplication or exponentiation, on long-integer data does not overburden a workstation with extensive resources, the performance of such operations may overwhelm an embedded processor, and especially wireless, handheld devices, and smart cards that have small memory capacity and strict latency constraints.

Cryptography applications are computationally intensive [45, 53]. Thus, a software-based implementation is inherently slow. For this reason, cryptography applications have traditionally been implemented in Application-Specific Integrated Circuits (ASIC) [5, 46, 47], or in hardwired-assists in Application-Specific Instruction set Processors (ASIP) [15, 19, 30]. Due to the ASICs and ASIP's hardwired-assists 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 [6, 22]. 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. A common reconfigurable array is the Field-Programmable Gate Array (FPGA), a general purpose fine-grain array, which allows the designer to implement any computing units subject to the FPGA architecture and logic capacity. Furthermore, a typical reconfigurable processor is called a Field-Programmable Custom Computing Machine (FCCM), which consists of a general purpose processor augmented with a reconfigurable array. The processor used in our experiements is the 32-bit RISC softcore processor, called *MicroBlaze* [59], that is to be mapped on a Xilinx's Virtex-II Pro FPGA. Even though this particular FCCM provides speedup over the pure-software solution, it exhibits a long critical path delay and high slice usage overheads since the Virtex-II Pro FPGA is general purpose, being designed to support a broad range of applications. To reduce these overheads, we propose a Cryptography-oriented Reconfigurable Array, called *CryptoRA* so that MicroBlaze could be augmented with the *CryptoRA*, rather than with Xilinx FPGAs. With this new configuration, the *CryptoRA*-based FCCM is expected to provide further improvement on the computing speed and to reduce the slice usage with respect to the Virtex-II-based counterpart.

1.1 Problem overview and thesis scope

Since the pure-software solution is slow in performance and the ASIC/ASIP solution is expensive in cost, the Reconfigurable Computing (RC) is an attractive solution for improving the performance in running public-key cryptography tasks on embedded systems at an acceptable cost. Furthermore, unlike in the server environment, the embedded system only requires the choice of one key length per secure session. Only when the times that key length is chosen differently from the previous, the re-programming process of the FPGA chip is taken place, which in general requires ranging from 20ms to 100ms to complete. To be able to evaluate the effectiveness of an RC solution, a good pure-software solution as a reference implementation is needed. Assembly-level software gives the best performance; nevertheless, it has the worst time-to-market and development cost. An alternative is to write the program in a high-level language, e.g., C/C++ [27], and optimize the high-level code. Since in embedded systems domains, time-to-market is of paramount importance, only the latter approach is further considered.

Given an embedded FCCM composed of a RISC-like processor and a fine-grain FPGA, the issues of providing a reconfigurable solution to public-key cryptography are as follows:

1. Profiling the public-key cryptography domain and choosing the appropriate FPGA architecture.

2. Performing hardware-software co-design to partition the public-key cryptography task into a software component and hardware component.

3. Mapping the hardware component onto FPGA.

4. Incorporating the FPGA-based hardware unit into the host embedded processor.

As far as the item 4 is concerned, MicroBlaze already provides a good solution for it: the Fast Simplex Link (FSL) [58] is a very efficient and easy-to-use interface to transfer

data between MicroBlaze and the FPGA-based computing units. Therefore, only items 1, 2, and 3 are considered throughout the thesis.

As mentioned, a referenced pure-software implementation is needed in order to effectively evaluate the benefits and drawbacks of an RC solution. Although many open-source cryptographic packages/libraries provide assembly-optimized routines/functions for long integer operations, porting them to the MicroBlaze-based embedded system [59] may create problems due to instruction incompatibility. Thus, developing a C-level program that runs on MicroBlaze is a good alternative. Nevertheless, those assembly-optimized software/libraries can be regarded as an optimal pure-software implementation, and therefore, they can be used to evaluate the C-level program's performance. In general, the main idea of performing hardware-software co-design is to provide hardware support for the computationally demanding operations/functions, which can be determined through program profiling. The hardware unit is first specified using a hardware description language (HDL), then synthesized, and finally placed and routed onto Xilinx Virtex-II Pro FPGA. The initial requirements and freedom degrees of our research activity can be summarized as follows:

1. Develop a C-level implementation of public-key cryptography tasks on MicroBlaze processor, profile it, optimize it, and assess its performance.

2. Use MicroBlaze + Xilinx Virtex-II Pro fine-grain FPGA as an experimental embedded FCCM.

3. Assess the performance of a Virtex-II-based Reconfigurable Computing solution in implementing public-key cryptography and assess the order of magnitude implementation versus pure-software solution.

4. Assess the appropriateness of commercial FPGAs in implementing public-key cryptography, and investigate FPGA architectures to better support cryptography.

Based on these requirements and the available development tools [35, 60, 64] for MicroBlaze and Virtex-II Pro FPGA, we restrict our thesis scope as follows.

- In order to complete a hardware-software solution for computing Elliptic Curve (EC) point multiplication, a fully-functional and relatively high-performance software-based framwork is needed. The function and the performance of this framework is verified by comparing it against an assembly-optimized library, which is modified such that it also computes EC point multiplication for various bit length, mainly 160, 192, and 224 bits. As the rule of thumb for hardware and software partitioning, determining the most computationally demanding portion in a software implementation helps to make the decision of what part should be implemented in hardware and what part should remain in the software.

- As mentioned, the primary goal is to augment the MicroBlaze processor with a reconfigurable functional unit. This particular experimental plaform can be configured using the Xilinx Platform Studio (XPS) developing tools [60]. The FPGA-based functional unit can then be incorporated into the processor using the Xilinx's primaritory wrapping interface, FSL [61], which provides the fastest data transferring between the core processor and other functional units.

- As another goal of ours is to improve the timing and slice usage on commercial FPGAs, we propose a novel Cryptography-oriented Reconfigurable Array (CryptoRA), which uses a similar fast-addition structure to that of modern Xilinx FPGAs (Xilinx Virtex-II FPGAs [62]). This proposed architecture should provide improvement on the performance and slice usage for public-key cryptography applications that require long-integer operations, mainly addition, subtraction and comparison.

- The evaluation of the FPGA-augmented MicroBlaze performance is carried out within the public-key cryptography domain. The estimated performance and slice usage fig-

ures from the MicroBlaze + *CryptoRA* hybrid is compared against the MicroBlaze + Xilinx hybrid figures in order to show the *CryptoRA* performance.

1.2 Open questions

In this section, the main design questions, which are needed to be cleared up along the design process, are given as follows. The answers to these questions are outlined as well. Detailed explainations can be found within this thesis.

1. Why is a new pure-software solution needed?

   As the domain application to be tested with our Reconfigurable Computing implementation is cryptography, Elliptic Curve Cryptography (ECC) algorithm is primarily considered. In particular, we focus the ECC over the prime field. This is because ECC is more attractive to the embedded system environment. Also, by implementing ECC over the prime field, the modular exponentiation, the most computationally demanding operation in RSA, can also be explicitly verified. Since MicroBlaze is a very specific embedded processor, none of the existing software/libraries that supports cryptographic functions can be easily ported on MicroBlaze. This is because most of these software/libraries are assembly-optimized for general purpose instruction set architectures (e.g., RISC Pentium). Hence, a possible approach to this is to develop a reliable and relatively high-performance C-level software that computes EC point multiplication as the reference implementation.

2. What is it that we want to have hardware support for?

   To answer this question, the bottleneck function in software must be determined. This can be accomplished by the profiling feature in the developing tools. In our case, the modular multiplication is found to be the most time-consuming operation. Therefore, we provide reconfigurable hardware support for it. Consequently, it requires algorithm research and VHDL design to build this FPGA-based modular multiplier.

3. What are the features that a *CryptoRA* should include?

   The primary idea of commercial FPGAs (e.g., Xilinx and Altera) is to provide the designer with the maximum flexibility such that virtually any logic function can be implemented. As a result, the commercial FPGA performance may not be optimal in terms of speed and area. Thus, it is possible to modify the FPGA architecture catering towards more specific application, and to still maintain the original flexibility level that commercial FPGAs have. This is the main idea of proposing the Cryptography-oriented Reconfigurable Array (*CryptoRA*). The improvements of the architecture aspect come from investigating the results after mapping some of the fast-addition techniques on Xilinx Virtex-II Pro FPGA. *CryptoRA* comprises the major features of (i) an increased granularity of the logic tile, (ii) the extention of the dedicated carry chain of standard FPGAs over the horizontal direction, and (iii) the split Look-Up Table (LUT). With the inclusion of these features in *CryptoRA*, improvements on speed and area performance for long-integer operations required in public-key cryptography tasks can be expected.

1.3 Thesis overview

This thesis is organized as follows. In the second chapter, we briefly cover the existing cryptography standards. We, however, mainly focus on the public-key cryptography since that is the cryptographic class that uses algorithms which requires computationally intensive operations, such as long-integer modular operations. Within this class, we look into the Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC) algorithms; these are the representative algorithms that contain all the computationally demanding operations needed in the public-key cryptosystem. Thus, we present the basic operations for these algorithms and also address their pros and cons.

In Chapter 3, the architectural support for addition and fast addition on both Xilinx and

Altera is discussed. For instance, Xilinx's FPGAs, such as Spartan and Virtex families, provide architectural support for carry-lookahead addition while Altera's FPGAs, such as Stratix families, provide architectural support for carry-select addition. Thus, it is important to show how their dedicated carry chains operate to support these fast-adder additions.

In Chapter 4, we give a general overview on the state-of-the-art in the computing machine design for cryptography applications. One of these machines is an FCCM that consists of a RISC embeddded processor augmented with a very coarse-grain reconfigurable array. This solution is indeed what we use for public-key cryptography computation because we still like to have the flexibility of reconfiguring a computing unit on-the-fly while accelerating the performance. A digest of the papers which use different type of computing solutions for public-key cryptography application is presented.

In Chapter 5, we cover the software implementation for each level of operations, including modular operation level, EC point-operation level, and EC point-multiplication level. Modular multiplication, which is determined to be the most time-consuming operation in both RSA and ECC algorithms, is explicitly presented using Montgomery Modular Multiplication (MMM) algorithm.

Chapter 6 contains the detailed description of the hardware components in the VirtexII-based Montgomery Modular Multiplier unit, including Carry-Save Adder (CSA), Ripple-Carry Adder (RCA), and a comparator unit. This is because that the long-integer addition, subtraction and comparison are the core operations in public-key cryptography. After analysing those implementations on Xilinx FPGA, some issues are raised. Mainly the critical path delay and the high slice usage are determined. Then, Cryptography-oriented Reconfigurable Array (CryptoRA) is proposed to alleviate these issues and its new features are introduced and described in detail. Carry-Lookahead Adder and the new comparator unit structures are able to take the advantage of *CryptoRA*. The mapping process on *CryptoRA* is also presented.

Chapter 7 presents the experimental platform configurations, including the MicroBlaze processor and its surrounding peripherials. The data transferring interface between MicroBlaze processor and FPGA-based functional unit is among one of the important peripherial IP cores, and hence, its usage and functionality are covered extensively. Additionally, this chapter showcases the simulated and estimated results of the hardware-software implementation. It reaveals the performance expressed in cycle count for our C-level software program and for the *MIRACL*, an assembly-optimized library. The speedup of the FPGA-based Montgomery modular multiplier (MMM) versus software MMM ranges from 37$\times$ to 45$\times$ for bit lengths between 160 to 224. This in turns allows a speedup ranging from 11$\times$ to 22$\times$ for EC point multiplication. Due to the fact that by utilizing *CryptoRA* instead of a commercial FPGA, the critical path is reduced, and therefore, results in a better performance.

Chapter 8 concludes the thesis summarizing our findings, discussing our main contributations, and suggesting an area for future work.

Chapter 2

Cryptography domain and standards

The cryptography domain covers a wide range of algorithms and theories to be used in many different protocols. The cryptography basics are covered in this chapter and are organized as follows. The basic cryptography concept is first introduced. Then a number of the public-key algorithms are presented. Finally, the common operations shared between the public-key cryptography algorithms are shown through the operational hierarchical diagram.

2.1 Cryptography introduction

Cryptography is the science of information security that utilizes mathematical algorithms to protect data transmitted in open communication networks, such as the Internet. The four main purposes that it serves are data confidentiality, integrity, authentication, and non-repudiation.

In Table 2.1, a brief comparison between the cryptography standards is provided. There are two major classes of cryptographic systems. The first class is called private-key cryptosystem, and includes Advanced Encryption Standard (AES) and Data Encryption Standard (DES) as representative members. 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 [14]. It includes Rivest-Shamir-Adleman (RSA) [44] and Elliptic Curve Cryptography (ECC) [37, 39] as representative members. In the public-key cryptosystem, 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.

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 cheap	less overheads on key establishment
Disadvantages		More overheads on key establishment	Computationally demanding
Confidentiality	Keep the data secret from other unitended receivers	supported	supported
Integrity (hash)	Keep the data unaltered	supported	supported
Authentication	Be certain where the data came from	not supported	supported
Non-repudiation	Digital signature	not supported	supported

It is noticed that not only the public-key cryptosystem provides more services than the private-key cryptosystem, but also it resolves the main issues in the secret-key cryptosystem, which are the key distribution, and key management [21]. This is one of the main motivation behind the creation of public-key cryptosystem. However, the computational requirements of private-key cryptography are much lower than those of public-key cryptography. Therefore, both cryptosystems are often used in conjunction in 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 [24] 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 for 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 the Integer Factorization (IF), Discrete Logarithms (DL), and Elliptic Curve Discrete Logarithm (ECDL). The definition for each number theory problems are given below [57].

1. IF problem:

   Given a positive large 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

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

   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.

Although there are not yet any efficient algorithms existing to compute the corresponding value to crack these public-key algorithms, the operations required in these algorithms are also computational demanding. As seen, it is clear that the algorithms from the DL family require so-called modular exponentiation operation. However, it is not so clear on the underlying operations required in the algorithms from both the IF and ECDL families; however, they will be covered in the subsequent section.

From the underlying operation point of view, all the public-key algorithms have something in common —that is, their underlying operations are modular operations. Since the core modular operation in both IF and DL families is the modular exponentiation, it is redundant to cover both families. Therefore, we will utilize the Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC) algorithms to demonstrate the underlying operations required in public-key cryptosystem.

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: A 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

2.2.1 RSA algorithm

RSA algorithm is named after Ron Rivest, Adi Shamir, and Leonard Adleman [44]. 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 adapted widely in many applications, such as Pretty Good Privacy (PGP), a popular method for encrypting email, and in Secure Socket Layer (SSL) [54]. 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 [57]

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 Alice 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 easily computed. 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, 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/square operations. According to the Algorithm 2.2, it requires $1.5 \cdot n$ (where n is the bit-length of N) long-word modular multiplication operations on average, assuming modular squaring is computed the same way as modular multiplication. In the example of 1024-bit RSA, the number of calls to modular multiplication is $1,536 \equiv 1.5 \cdot 1024$, and these operations are operated on 1024-bit long variables.

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 [39] and Victor Miller [37]. ECC becomes an attractive alternative solution for the next generation public-key algorithm [54] due to the same level of security that it can offer with smaller key size requirement compared to other public-key algorithms. However, not every elliptic curve offers strong security properties—for some curves, the Elliptic Curve Discrete Logarithm Problem (ECDLP) may be

solved efficiently [52]; 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 [17, 18] that possess the necessary security property [20].

Unlike the RSA algorithm, the most time-consuming operation in ECC algorithm is so-called the Scalar (Point) Multiplication. It works as follows. 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 the 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 point multiplication is a combination of EC point addition and EC point doubling operations, as illustrated in the Double-and-Add algorithm. On average, it needs n EC point squaring and $0.5 \cdot n$ EC point addition. 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.

ECC is typically defined over two types of fields: binary and prime. The operations over the binary field can be simply implemented since the field addition/subtraction is essentially a bit-wise *XOR* operation. Also, the binary-field squaring is much simpler than the binary-field multiplication; only hard-wired shift is required and can be done in a single cycle, resulting a significant savings on the computation time. The binary-field operation does not pose significant computational requirements. Thus, ECC over the binary field is not

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**

considered any further and only ECC over the prime field is presented, whose underlying modular operations are modulus of prime numbers. In other words, both RSA and ECC over the prime field require modular operations.

2.2.3 ECC over the prime field $(GF(p))$

Prior to the calculation for the EC point multiplication, an elliptic curve over the prime field, whose general equation is give in Equation 2.2, is needed to be defiend. Different sets of parameters would yield different points, and the number of points on the curve. The base point P, intermediate points Rs, and the resulting point from the EC point multiplication Q, must be the points that is part of the defined elliptic curve.

$$E : y^2 = x^3 + ax + b, \tag{2.2}$$

where x, y, a, b are large unsigned integers, ranging from 160-bit to 521-bits.

To algebraically show the EC point addition and point doubling in the Affine coordinate, where one inversion exists, Table 2.3 is presented. Different projective coordinates for computing point addition and doubling were proposed in order to minimize the number of modular inversion [8, 9]. The use of those projective coordinates comes with the penalty

of more modular multiplications and modular sqauring operations, and requires more temporary variables as the result of eliminating the modular inversion operation. Nevertheless, a variant of the projective coordiates is used in our EC point multiplication implementation. This will be further discussed in Chapter 5. We would like to remind that the x and y are the coordinates of a point on the curve, and their values are ranging from 160-bit to 521-bit in bit length, which corresponding to 2^{160} up to 2^{521} in the actual value. Therefore, all the operations performed on these are long-word integers.

Table 2.3: Point operations in Affine coordinates [21]

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
$x_R = s^2 - x_p - x_Q \pmod p$	$x_R = s^2 - 2x_p \pmod p$
$y_R = s(x_P - x_R) - y_P \pmod p$	$y_R = s(x_P - x_R) - y_P \pmod p$

To further emphasize on what modular operations are required and the operation counts per EC point doubling and point addition needed in different coordinates, Table 2.4 is presented. As noticed, we only consider the modular operations that are computationally intensive, such as modular multiplication, squaring, and inversion. This is because the modular addition/subtraction operations are relatively fast, and thus, can be neglected in comparison with the other operations. According to the first entry in Table 2.4, a point doubling in Affine coordinate requires two modular multiplications, two modular squarings, and one modular inversion. Other entries can be interpreted the same way. This table also reveals that using Jacobian projective, and mixed Jacobian and Affine coordinates are

good coordinates for EC point doubling and point addition, respectively because they yield the least numbers of modular operations. In the example of Jacobin, mixed Jacobin and Affine coordinates are used for the 160-bit ECC over prime field $(GF(p))$, it yeilds, on average, a total number of $2480 \equiv (160 \times 10 + 80 \times 11)$ modular multiplication, assuming modular multiplication is also used for modular squaring. This reveals that the number of calls to modular multiplication are in the range of thousands for one EC point multiplication.

Table 2.4: Operation counts for one point addition and one point doubling over $GF(p)$ [21]

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, the cryptographic protocol presented serves as an example for a common-key establishment using Elliptic Curve Diffie Hellman (ECDH) protocol, which is an analogue to the popular Diffie Hellman key exchanged protocol from the DL family. This simplified protocol depicts that by computing two EC point multiplications in each party, a common secret key is generated, and can later be used as the key for large-size encryption.

$$\begin{array}{cc} Alice & Bob \\ x & \xrightarrow{x\cdot P} x\cdot P \\ y\cdot P & \xleftarrow{y\cdot P} y \end{array}$$

Then, Alice and Bob compute

$$K_a = x\cdot(y\cdot P) = xy\cdot P$$

$$K_b = y\cdot(x\cdot P) = xy\cdot P$$

In Table 2.5, 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 are needed. This is the reason why ECC is a preferable option for public-key cryptography algorithm in embedded systems.

Table 2.5: Equivalent key sizes [20]

ECC (in bits)	RSA (in bits)	Protection lifetime
160	1024	until 2010
224	2048	until 2030
256	3072	beyond 2031
384	8192	infinity at the current level of technology
521	15360	infinity at the current level of technology

.

Moreover, the table also presents the valid protection level 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 equally important to support both RSA and ECC standards since they are currently the most widely used public-key algorithms today.

A hierarchy of operations in RSA and ECC algorithms is shown in Figure 2.1. As noticed, the underlying finite field arithmetic are the same in ECC over prime field $(GF(p))$ and RSA, regardless the disparity in bit length and modulus value. In other words, it is possible to utilize the same software routine or accelerated hardware unit to compute the finite field arithmetic operations required in all the public-key algorithms.

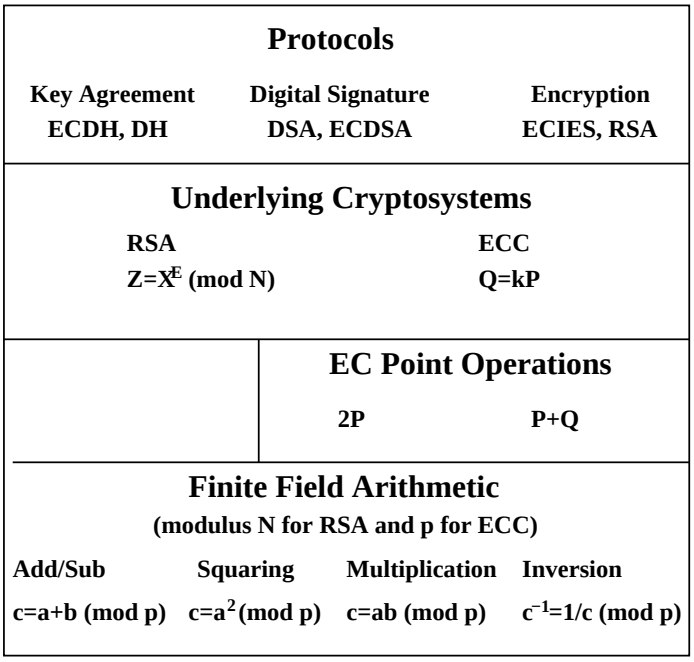


Figure 2.1: Hierarchy of operations in RSA and ECC schemes

We would like to emphasize that because we deal with long-word integer operations, all these operations can only be computed using routines that consist of basic 32-bit instruction set architecture on the 32-bit processor. To speed up the overall performance, different optimizations can be applied at each level of the hierarchy of the operations in public-key cryptography schemes. Nevertheless, in our research, we mainly look at the possible optimizations at the underlying finite-field arithmetic level. Furthermore, in our particular implementation, the modular inversion computation is based on the modular multiplication. A brute force solution, meaning computing regular multiplication and then modular

operations of two long integers is extremely slow. Thus, a high-performance modular multiplier is needed and Montgomery Modular Multiplication (MMM) [38] is used for such a task as it is proved to be a very efficient algorithm. Wired-hardware support for MMM is expensive. Thus, providing reconfigurable MMM hardware support is worth investigating.

In the next chapter, we look at what the architectural supports for addition and/or fast addition that Xilinx and Altera have on some of their existing FPGA devices.

Chapter 3

Xilinx and Altera FPGA architectures

We have established that public-key cryptography requires long-integer arithmetic. Also, as the Field-Programmable Gate Array (FPGA) is the main platform used in our research, the appropreciateness of Xilinx's Virtex-II Pro and Altera's Stratix FPGAs in implementing cryptography are analyzed. Thus, the architectural support for addition/fast addition from these FPGA chips are particularly to our interest. This chapter is organized as follows. It discusses the architectural support for Ripple-Carry Adder (RCA) in Xilinx's FPGA family. Subsequently, the architectural support for Carry-Lookahead Adder (CLA) is presented. It is then followed by the Altera's FPGA support for addition.

3.1 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 [64] or FLEX 10K from Altera [2]. Since building fast-adder structures requires the deployment of carry-lookahead or carry-select networks, the corresponding generate/propagate or select signals, respectively, need to go through the slow global interconnect. Therefore, 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-bits).

According to our simulations on Amirix AP1000 FPGA Development Board [7] using an XC2VP100 Virtex-II Pro FPGA [62], and the software tool Xilinx ISE (Project Navigator) v9.1.03i [64], propagation through global interconnect takes at least 0.8 ns, while a LUT latency is 0.4 ns. This is in contrast to the propagation through the dedicated carry chain that takes 0.0313 ns per tile. This means that roughly 32-bit ripple-carry addition

is as fast as 3-to-2 carry-save addition. This result is consistent with the Xilinx figures: 64-bit addition has the latency of $10^3/114 = 8.8$ ns, while 16-bit addition has the latency of $10^3/239 = 4.2$ 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 quickly show how the dedicated carry chain is used to provide the support for ripple-carry addition, the equation for the basic addition element, the Full Adder (FA), in a Ripple-Carry Adder (RCA) is presented in Equation 3.1. Assuming two input arguments, x and y, and their sum, s, the full-adder identities for bit i are shown in Equation 3.1, where $c_{in}(i)$ and $c_{out}(i)$ are the input and output carry bits, respectively. An RCA [43] is built with a series of these full-adder blocks with c_{out} at position i being connected to c_{in} at position $i+1$. The $s(i)$ is only set if odd number of the input bits $(x(i), y(i), c_{in}(i))$ are set. The *carry* bit is only set if at least one of the following scenarios is true - (i) both $x(i)$ and $y(i)$ are set, or (ii) one of the $x(i)$ and $y(i)$ is set, and incoming carry bit $(c_{in}(i))$ is set.

$$\begin{aligned} s(i) &= x(i) \oplus y(i) \oplus c_{in}(i) \\ c_{out}(i) &= x(i)y(i) + x(i)c_{in}(i) + y(i)c_{in}(i) \end{aligned} \tag{3.1}$$

To emulate the RCA in some Xilinx FPGAs, the FA element, shown in Figure 3.1, is connected through the dedicated carry chain. Having such configuration as shown in Figure 3.1 does guarantee the deployment of such function. The dedicated carry chain is composed of a number of such elements. This is in fact the resulting mapping on Xilinx FPGA when the '+' operator is used in HDL.

The basic block in Xilinx Virtex-II Pro family FPGA is called a Configurable Logic Block (CLB). It is shown in Figure 3.2. As seen, there are four slices in a CLB. Each slice has two four-input Look-up Tables (LUTs) and two flip-flops. The LUTs may be configured as either combinational logic or as RAM. In the case of RCA, the LUT is configured as an *XOR*. Additionally, it contains dedicated hardware, such as two dedicated carry MUXes

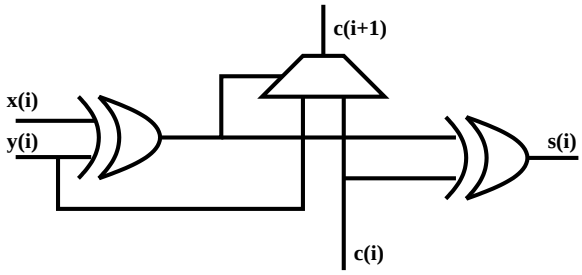


Figure 3.1: Dedicated Carry Chain Element

and two dedicated *XOR* gates, to implement fast carry operations for arithmetic circuits. This particular Xilinx FPGA device also has architectural support for Carry-Lookahead Adder (CLA), which is discussed next.

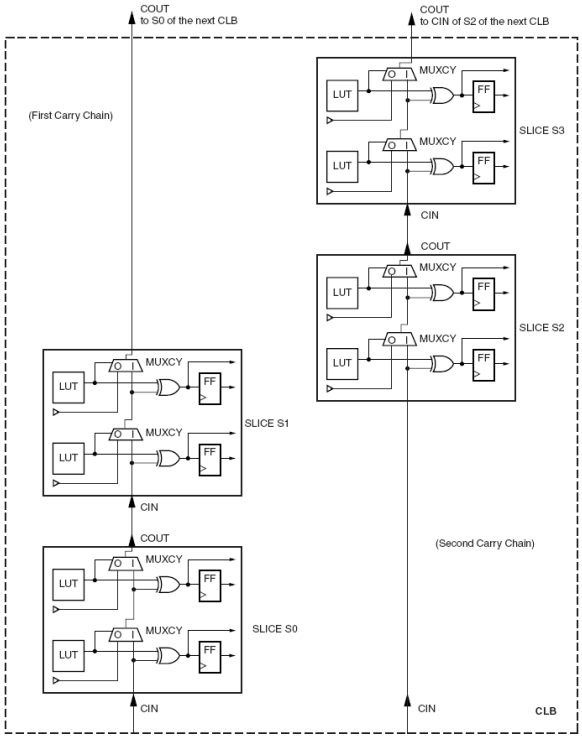


Figure 3.2: Fast carry chain in Xilinx Virtex-II Pro FPGA [63]

3.1.1 Carry-Lookahead Adder in Xilinx

In modern FPGAs, fast-adder structures are also given architectural support in addition to the ripple-carry addition. These structures can be used for long-integer addition in cryptography. The Virtex-II family provides dedicated hardware for a carry-lookahead network [63]. It is apparent in Equation 3.4 and 3.5 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 3.2.

As the limitation with RCA is the time it takes for the carry to be propagated through the entire length of the adder. The RCA latency depends linearly with the adder length. Many fast adder techniques were 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, $c_{in}(i)$. Recall that Xilinx Virtex-II Pro FPGA does have architectural support for Carry-Lookahead Adder (CLA). A closer look at the slice in CLB is presented in Figure 3.3. To be able to support CLA, namely block-level *generate* and *propagate* signals, the dedicated *AND* gate, named *MULTAND* is required.

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 3.3 and 3.2, respectively. Based on bit-level signals, block-level *generate* ($g_b(j)$) and *propagate* ($p_b(j)$) signals, can be defined for each block j, as shown in Equation 3.5 and 3.4, 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.

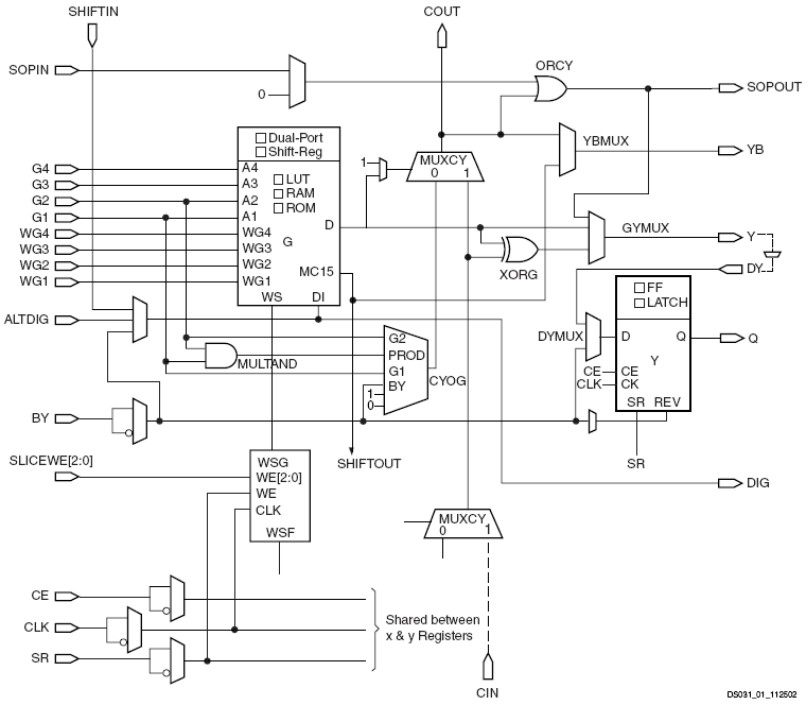


Figure 3.3: Xilinx's Slice Structure in Detail [63].

$$p(i) = x(i) \oplus y(i) \tag{3.2}$$

$$g(i) = x(i) \cdot y(i) \tag{3.3}$$

$$p_b(j) = \prod_{i=I}^{I+K-1} p(i) \tag{3.4}$$

$$g_b(j) = g(i) + g(i-1)p(i) + g(i-2)p(i-1)p(i) + g(i-3)p(i-2)p(i-1)p(i) + \ldots \tag{3.5}$$

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 3.6.

$$\begin{aligned} c_{out}(i) &= g(i) + c_{in}(i)p(i) \\ c_{out}(j) &= g_b(j) + c_{in}(j)p_b(i) \end{aligned} \tag{3.6}$$

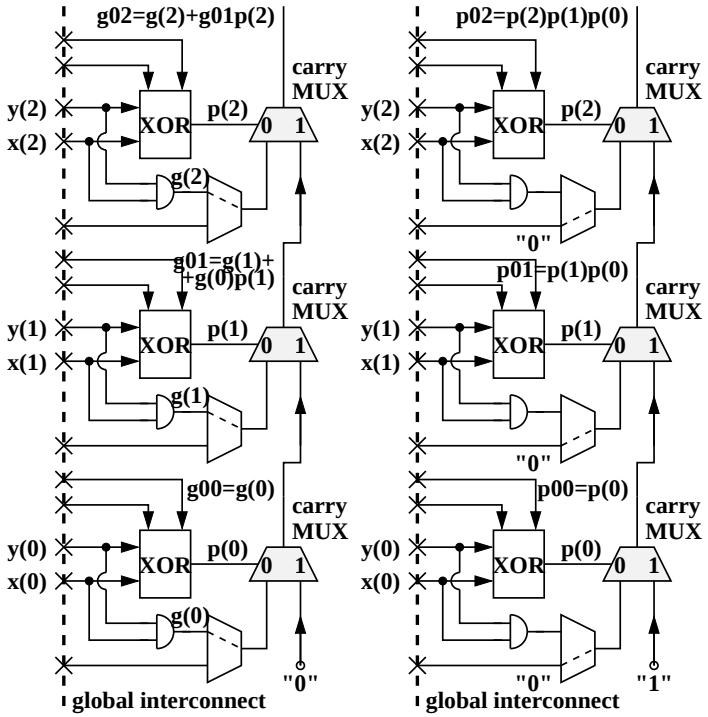
In carry-lookahead adders, the block-level *generate* and *propagate* signals in the hierarchy are first computed in parallel. Based on the these signals, the block-level carry signals

used as inputs for each sum-bit block are then computed serially. Finally, all the sum bits, $s(i)$ are computed according to Equation 3.1. It is well-known that CLA has a latency of $O(log(n))$, where n is the wordlength [43].

From the point of view of a reconfigurable array, CLA has an advantage over Carry-Skip Adder (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 decide to 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.*'s result that a Brent-Kung adder, which is essentially a carry-lookahead adder, achieves a very good latency performance on FPGAs [23].

In order to implement and map the CLA using the dedicated carry chain on Xilinx Virtex II Pro Chip, The VHDL code is written in a way that exposes the *Slice* components to VHDL compiler. Since dedicated carry chains are used for signals, such as block-level *generate* ($g_b(j)$), block-level *propagate* ($p_b(j)$), block-level *carry* ($c_b(j)$), and sum bit ($s(i)$), 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 *XOR/AND/OR* logics in the Equation 3.1, 3.4 to 3.6 for generating the $g_b(j)$, $p_b(j)$, $c_b(j)$, $s(i)$ signals are now emulated using the dedicated carry chains and the internal logic gates. The emulations using these dedicated hardware are shown in Figure 3.4(a), Figure 3.4(b), Figure 3.5(a), and Figure 3.5(b).

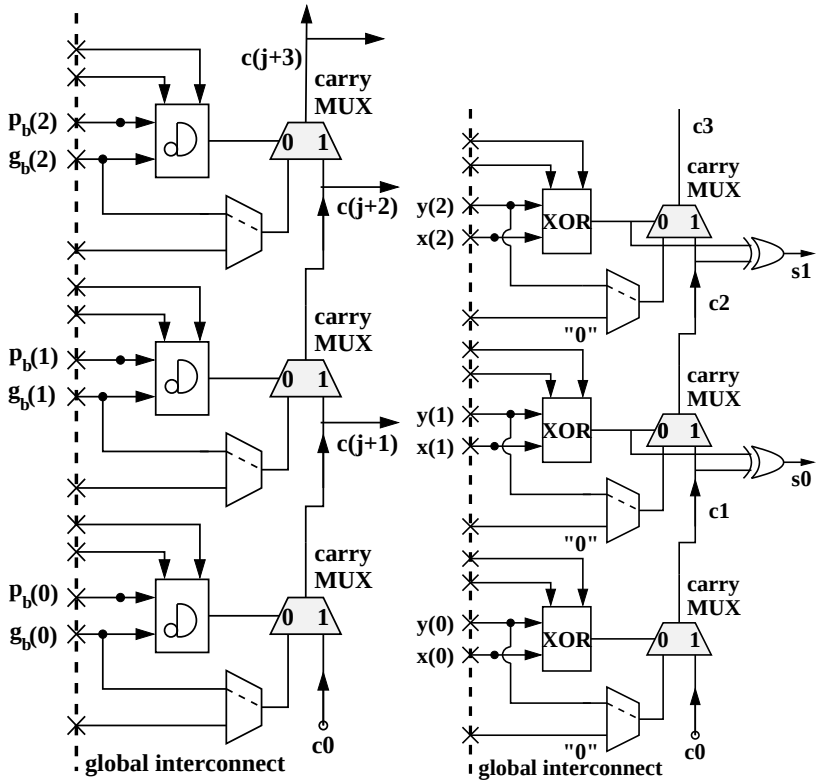
To begin with, the block-level *propagate* signal ($p_b(j)$) in Equation 3.4 is essentially a *wide-AND* function, and therefore, it can be implemented by means of the carry chain [62]. Also, it is noticed that block-level *generate* signal ($g_b(j)$) in Equation 3.5 is indeed the *carry* signal that 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 on Xilinx Virtex II Pro Chip [62], to implement the $g_b(j)$ signal using the carry chain would require



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

Figure 3.4: Carry-lookahead network on Xilinx Part-I

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 devices support CLA [55]. In addition, the reason why Equation 3.4 and 3.5 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 inherent in the block-level signals. Block-level signals, therefore, can be implemented using the dedicated carry chain. For the block-level *carry*, because the inputs to the block are $g_b(j)$ and $p_b(j)$ signals, LUT is configured as *AND-with-one-input-inverted* gate, and the schematic diagram is shown in Fig 3.5(a) for Equation 3.6.



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

Figure 3.5: Carry-lookahead network on Xilinx Part-II

In Figure 3.5(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 first MUX in each sum-bit block 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. In other words, as the dedicated carry chain contiunes with the sum-bit block, which is unwanted, it means that in order to continue propagating the *carry* bits, it is first forced to go through global interconnects and then continuing with a block of dedicated carry chain. As a result, this mapping leads to a longer critical path in the design as the block-level *carry* signals now take longer to be propagted, which is

not desirable. This unwanted mapping result is due to the dedicated carry MUX having its output connected to the input of two separate dedicated MUXes.

The solution to this problem is that we emulate the first element in each sum-bit block in Figure 3.5(b) using LUT. This emulation for generating the carry and sum bits in the first element of each sum-bit blocks is based on the original carry and sum equations (Equation 3.1) and is shown in Figure 3.6. 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 with no choice but continuing the dedicated path in block-level *carry*. The penalty is that an extra LUT is needed to generate the sum bit of the first element in each sum-bit blocks. In addition, if CLAs are implemented in commercial FPGAs, the number of carry-lookahead stages should be kept minimal and only increase the block size when longer bit length is needed. This is because the connection between the stages corresponding to the global interconnects on the FPGA, which is expensive. On the other hand, the increase in elements in the carry chain means increase in the dedicated MUX usage, which is relatively cheap compared to the global interconnects. A different fast adder architecture supported from Altera is presented in the next section.

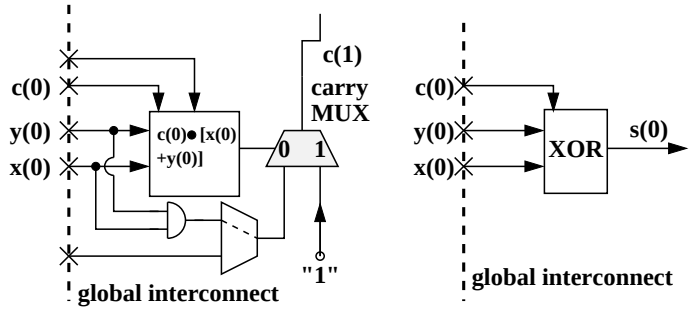


Figure 3.6: Modification of 1st element in each sum-bit block.

3.2 Altera's adder structure

Altera has different architectural support for fast addition. 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.

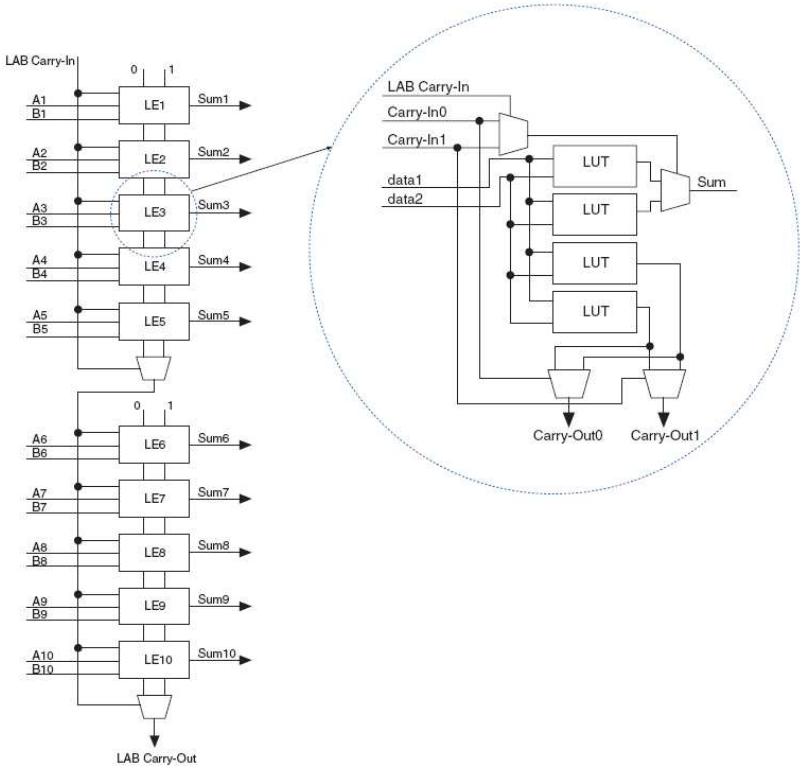


Figure 3.7: Carry Select Chain from Altera Straix FPGA [2]

The major drawback of Carry-Select Adder (CSeA) is that it utilizes 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 3.7, 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 outgoing 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 using the Xilinx FPGA, this section is more for the reference and comparison purpose. Thus, this configuration from Altera is no longer considered.

3.3 Conclusions

As discussed, the FPGA that we use throughout this research is the Xilinx Virtex-II Pro FPGA, in which both ripple-carry and carry-lookahead additions are given architectural support. While ripple-carry addition provides best results for adding two operands with bit length ranging in tenth, carry-lookahead addition is a better option for long-integer addition (e.g., bit length ranging hundreds) in 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 more expensive to construct a subtraction out of a carry-lookahead adder in this particular Xilinx FPGA since it requires additional LUT to invert one of the operands. For these reasons, we take one step further than just providing an Reconfigurable Computing (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.

As we know, providing reconfigurable solution using a processor augmented with a FPGA is only one of many methodologies that one can adopt. We will, in the next chapter, present a number of different implementations towards public-key cryptography that have been reported in the literature.

Chapter 4

State-of-the-art solutions for public-key cryptography

Since designing a processor for a given application family, e.g., cryptography, requires essentially solving an optimization problem in a multidimensional space, this chapter starts out with brief review on the some of the options in the computing machine design for public-key cryptography. In particular, we focus on the Reconfigurable Computing paradigm. It is followed by the related work that have been done or are currently ongoing in providing computing solution for the Montgomery Modular Multiplication algorithm in the public-key cryptography domain. Finally, this chapter is completed with some conclusions and closing remarks.

4.1 Reconfigurable computing paradigm review

In the design of computing machine, it is well-known that General Purpose Processor (GPP) provides the flexibility at expense of performance while Application-Specific Integrated Circuits (ASIC) provides the performance at expense of flexibility. Somewhere in the middle, there is the reconfigurable arrays, which provide acceptable flexibility and performance when compared to the afore-mentioned types of computing machines. 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. This type of computing engine is so-called Reconfigurable Computing (RC) [31, 32], an emerging computing paradigm. FPGA is often used in conjunction with GPP to become a hybrid referred to as a Field-Programmable Custom Computing Machine (FCCM) [6].

Concerning cryptography, which is the main domain of this thesis, it requires operations

on various bit-length, ranging from 160 to 521−bits for ECC and 1024 to 2048−bits for RSA. This bit-length required in the cryptographical protocol depends on the algorithms used and the security level one needs. Having said that, with ASIC/ASIP approach, it can only support up to the maximum bit-length that it is decided at the implementation stage and can not be changed if a bit-length longer than the maximum bit-length is required in the future. However, this would not be an issue with FPGAs approach since it can be re-configured at the running time as long as the configuration file exists. Consequently, since an FCCM provides a solution with hardware-like performance and software-like flexibility, it is the RC paradigm that we focus on and propose the solution to the public-key cryptography computation.

4.2 Related work: hardware modular multiplier

As the layers of operations exist in public-key cryptography algorithms, which is described in Chapter 2, it is the responsibility of the system designers to decide what layers are to be supported in the hardware. Regardless of how deepth the public-key cryptography algorithms are deployed in the hardware, modular multiplication is always given hardware support due to its well-known computational complexity in software. It is worth reminding that we are only interested in the modular multiplication in the prime field, which is the case in all public-key cryptography algorithms, besides ECC over the binary field.

One of the most frequently used algorithms for both software and hardware implementations of modular mulitplication is the Montgomery Modular Multiplication (MMM) algorithm [38]. It is especially suitable in the hardware since it replaces the trial division operation with a series of additions and shifts, and is possible to trade-off the number of iterations (computing time) with silicon area. There are many different optimized hardware implementations on AISC/ASIP and FPGA platforms [5,10,15,16,33,34,40,42,48,49,56]. Furthermore, MMM algorithm is flexible in terms of the implementation - it can be imple-

mented in both block-wise and bit-wise methods and details on the MMM algorithm will be covered in the next chapter.

Ors and Batina proposed a systolic array architecture that implements a variant of the MMM algorithm [42]. This variant MMM algorithm does not require the comparison and the final subtraction operations at the end, which is different from the original MMM algorithm shown in Algorithm 5.2 in Chapter 5. It does, however, require one extra iteration and more area to hold larger variables. Their systolic array is implemented in Xilinx Virtex E FPGA using four different systolic array cells structures. In order to support any arbitrary bit length, this multiplier can be modified by means of reconfiguration or firmware update. Its computation time is $3L+4$, where L is the bit length. For the 160-bit modular multiplication, it would take $3 \times 160 + 4$, which is 480 clock cycles running at system frequency of 91.308 MHz. This FPGA-based Montgomery multiplier is used in their other papers on implementing procesor for RSA and ECC [4, 41]. Along the same line, a semi-systolic array, by means of Processing Element (PE), is proposed by Tenca and Koc [49]. Their Montogmery multiplier implemented in ASIC, running at 80 MHz, supports both ECC over binary ($GF(2^m)$) and prime ($GF(p)$) fields. With 32-bit PEs width, for 160-bit modular multiplication, they were able to obtain the run time of 4.1 us, which corresponds to 328 clock cycles. Furthermore, the run time of 18.3 us on ARM with assembly was also provided.

Eberle and Gura proposed a 64×64-bit multiplier that implements the MMM algorithm and has dual-field support on the field multiplication. They prototyped their implementation on Xilinx Virtex-II FPGA. Their multiplier structure is as follows: the product of two 64-bit operands is computed by two-dimentional arrays of *AND* gates. The resulting partial products are then added using the modified Carry-Save Adder (CSA) tree. Using this 64-bit multiplier, high performance is achieved in terms of the cycle count. However, the system frequency is expected to be quite slow due to the long critical path created by the two-dimentional array of *AND* gates and CSA tree. Even though this approach is proto-

typed in Xilinx FPGA, it is not appropriate for implementation on an FPGA platform due to the existence of expensive global interconnects; instead, it is more appropriate for this implementation on an ASIC. In their paper, the system frequency was not revealed and the computing time for RSA and ECC with varying bit length were given assuming on the system frequency of 1.5 GHz.

By the same token, a word-wise multiplier implementing MMM algorithm is also presented by McIvor and McLoone [34] using Xilinx Virtex-II Pro FPGA. They take the advantage of the 18×18-bit full-fledged multiplier embedded on the FPGA fabrics and the architectural support for fast carry lookahead logic, to construct their wide-word Montgomery multiplier on this particular FPGA device. The performance result on their 128-bit Montgomery multiplier reveals that it requires 26 clock cycle to complete one modular multiplication running at frequency of 75.63 MHz, and consumes $3,468$ slices. The drawback in this implementation is that it consumes a large amount of slices and operates on relatively low frequency due to the long critical path created by the multiplier. For instance, the number of slices used increases to $11,992$ and the clock frequency decreases down to 45.68 MHz for the 256-bit Montgomery multiplier. Also, even though this implementation offers high throughput rate, it is lack of flexibility in comparison to their another implementation in the same paper. A similar idea of implementing word-wise Montogmery multiplier on ASIC was proposed by Satoh and Takano [48]. Their Montgomery multiplier has a dual-field support. For the 160-bit ECC over $(GF(p))$ using word-length of 16-bit and of 32-bit, the operation cycles required for one modular multiplication are 276 and 91, respectively. Using the same bit-length multiplier for the 160-bit ECC over $GF(2^m)$, the corresponding operation cycles are 224 and 72, respectively. This lower number of cycle counts is obtained for ECC over $GF(2^m)$ because of the obsence of the carry propagation, which leads to simplier and faster long-word addition.

In [10], Crowe and Daly proposed a different methodology to implement the dual-mode Montgomery multiplier on Xilinx Virtex-II FPGA - two-level Ripple-Carry Adder (RCA),

which leads to a low operation frequency. In order to handle the disparity of bit length in RSA and ECC over $(GF(p))$, the Montgomery multiplier is made scalable, meaning that the fixed-area multiplier module can handle operands of any size. The performance results they provide are from synthesis rather then from placement and routing. Thus, the resulting frequency after placement and routing is expected to be much lower than the synthesis one, which is 44.91 MHz. This leads to higher computation time than they claim (which is 5.75 us) for one modular multiplication. However, the slice usage should remain the same, which is $5,267$.

Another approach in implementing Montgomery multiplier is to make use of Carry-Save Adders (CSA). This is exactly what McIvor and McLoone suggested [33]. In general, CSA is a good approach in implementing long-word addition because it can significantly shorten the carry propagation delay incurring in the ordinary ripple carry addition by separating the *sum* and *carry* bits. They realize that long-word addition is the core operation in MMM algorithm, having shorter critical path on the addition can also benefit to plaform other than FPGAs, such as ASIC. They implemented 5-to-2 and 4-to-2 CSA on Xilinx Virtex-II family FPGA, whose cycle counts are $k+1$ and $k+2$, respectively, where k is the bit length. Their implementation is able to retain relatively high operating frequency because it does not include the addition needed for converting the the results in the redundant form back to the non-redundant form. This process is often taken care by means of ripple carry addition. Thus, we do not consider that they build a complete Montogmery multiplier.

Along the same line, Goodman and Chandrakasan [19] proposed a domain-specific reconfigurable cryptographic processor (DSRCP) in ASIC. The main hardware components in this reconfigurable datapath are the reconfigurable logic, wide adder unit, wide comparator unit, and register files. The reconfigurability of this ASIC implementation comes from the controlled multiplexers added in the reconfigurable logic block such that it can be reconfigured on the fly to perform the required operation. The possible configurations

are Montgomery multiplication, $GF(2^m)$ multiplication, and $GF(2^m)$ inversion. Thus, 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. The reconfigurable logic is made of two-level of 3-to-2 Carry-Save Adder (CSA) and a number of MUXes. This is actually the implementation that we reference to for our Virtex-II based MMM functional unit; this design results relatively short critical path due to only two-level of Look-Up Table (LUT) needed to go through in Xilinx FPGA device. In their design, the resulting *sum* and *carry* bits from the CSA are added by the wide adder unit (fast Carry-Skip Adder (CSkA)), which requires three clock cycles to complete one long-word addition. We also implemented this fast CSkA on the Virtex-II FPGA for performance comparison purpose, but did not use it in our Virtex-II based design. Its implementation on FPGA is covered extensively in the subsequent section. We utilized the Ripple-Carry Adder (RCA) instead of the CSkA for the Montgomery modular multiplier on Virtex-II Pro FPGA. It is, however, later suggested to use Carry-Lookahead Adder (CLA). Moreover, we also reference to their tree-based wide comparator unit and implement it on Xilinx FPGA, which is also covered extensively in Chapter 6. Even though this comparator unit does not provide optimal performance result due to the $O(log_2 n)$ levels of global interconnects it has to go through, we still use it as part of the MMM hardware unit. It is, however, later modified such that it can be emulated serially using the dedicated carry chain on our proposed FPGA architecture, Cryptography-oriented Reconfigurable Array (CryptoRA).

On the other hand, Ananyi and Rakhmatov proposed a multiplier architecture [3] to compute modular multiplication for only the National Institute of Standards and Technology (NIST) recommended primes [18], which leads to the fastest method, called fast reduction. The reduction step for the product of two operands in the modular multiplication is instead computed using a series of modular addition and modular subtraction operations. An example of 192-bit in length is given in Algorithm 4.1 for further demonstration. Even though this approach provides the fastest computational time for modular multiplication operation,

one drawback is that this hardware unit is only useful for NIST recommended curves. In other words, 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 ECC over $GF(p)$, in which the primes are not those NIST recommended primes. Another drawback is that the multiplication product need to be first computed prior to this fast reduction. This means larger menory storage are required. Therefore, due to these limitation on the fast reduction implementation, we do not consider this approach to implement our hardware modular multiplier in FPGA. However, it is noted that the core operation in this approach is still the long-word addition/subtraction.

Algorithm 4.1 Fast reduction with NIST primes for 192-bit

Require: Integers $c = (c5, c4, c3, c2, c1, c0)$, where each c_i is a 64-bit word, and $0 \leq c < p^2$, and a modulus p

Ensure: $t = c \bmod p$

1: Define $192-$bit integers: $s1 = (c2, c1, c0), s2 = (0, c3, c3), s3 = (c4, c4, 0), s4 = (c5, c5, c5)$

2: **return** $s1 + s2 + s3 + s4 \bmod p$

To reduce the addition latency in FPGA, Hauck *et al.* analyzed the impact of deploying dedicated resources for a truly parallel carry-lookahead network as well as a complex carry-select network that comprises a dedicated carry-select multiplexor for each bit [23]. As mentioned, the carry-lookahead network suffers from increasing circuit complexity and fan-out toward high-order bits. This basically 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.

4.2.1 High-speed carry-skip adder in Xilinx

An ASIC-based Carry-Skip Adder (CSkA) functional unit was used for the purpose of adding two long-word integers in [19]. In our implementation, their high-speed CSkA can not be implemented efficiently for subtraction on Xilinx Virtex-II Pro FPGA, and therefore, the sum and carry bit of the Carry-Save Adder result in the Montgomery Modular Multiplier unit are added using the simple Ripple-Carry Adder (RCA) instead. However, the CSkA was still implemented as a stand-alone functional unit for comparison purpose. The top-level block diagram of this CSkA is presented in Figure 4.1 using 9-bit addition as an example. Each of the full-adder (*FA*) block in each column basically emulates the following equations, except for Equation 4.4.

$$p(i) = (x(i) \oplus y(i)) \cdot p(i-1) \tag{4.1}$$

$$g(i) = x(i) \cdot y(i) + y(i) \cdot g(i-1) + x(i) \cdot g(i-1) \tag{4.2}$$

$$s(i) = (x(i) \oplus y(i) \oplus g(i-1)) \oplus (p(i-1) \cdot c(j)) \tag{4.3}$$

$$c(j+1) = G(j) \oplus (P(j) \cdot c(j)) \tag{4.4}$$

where p, g, P, G, s, and c are the *propagate*, *generate*, the block-level *propagate*, *generate*, *sum*, and *carry* signals, respectively.

According to Equation 4.1 and 4.2, these p and g signals are independent of the incoming carry, and thus, can be computed in parallel in all the columns shown in Figure 4.1, resulting in shorter carry propagation delay. On the other hand, the value of *sum* can only be calculated once the incoming carry ($c(j)$) from the previous column on the left is ready. Equation 4.4 represents the carry-skip logic in the adder and determines the value of the outgoing carry ($c(j+1)$) to the next block on the right. In order to obtain the optimal critical path delay, the block-internal and carry-propagation delays should be balanced. This means that the propagation delay in the last column should be equal or close to that in the first column plus the delay disparity in the first column and the second column and so on.

The key to balance the propagation delay is to find out the delay of the carry-skip logic. For instance, in their ASIC-based implementation, the carry-skip logic has the same delay as one *FA*, and therefore, the number of *FA* in the next column is incremented by one *FA* as shown in Figure 4.4. However, in the FPGA-based implementation, each *FA* is emulated in LUT and connected through the dedicated carry chain. The carry-skip logic is emulated in LUT and has the delay of approx. 20 MUXes in the carry chain. Hence, every higher column is approx. 20-bit longer than the column directly below.

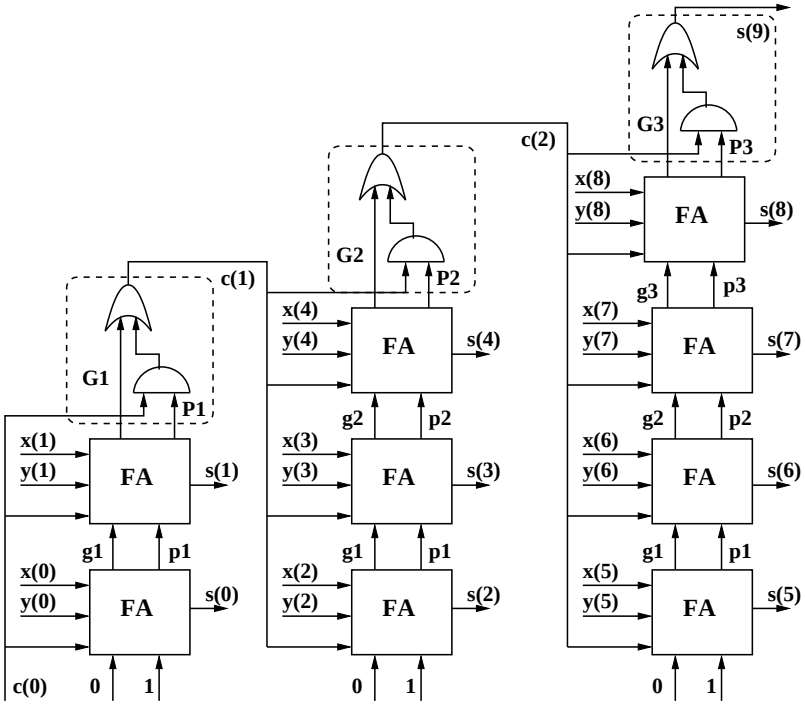


Figure 4.1: 9-bit carry skip adder. [19]

As discussed, even though the high-speed CSkA structure is similar to the CLA and therefore, it can be emulated using the dedicated carry chain in Xilinx FPGAs, it suffers from the irregularity in nature. This is due to two factors:

1. The carry-skip logic (*AND-OR* gates) at the end of each column.

2. The large number of fanout created by the output of the carry-skip logic.

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. As for the second concern, it is impossible to make these signals dedicated pathes in FPGA platform as the number of fanouts can not be fixed because the required number of elements in a column 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 Chapter 7.

4.3 MIRACL library

Since a portion of this research involves software implementation, we feel it is necessary to discuss briefly the library that we use as our checking reference. As mentioned, we use MIRACL [51], which stands for Multiprecision Integer and Rational Arithmetic C/C++ Library, to check the effectiveness as well as the correctness of our C-level software implementation. MIRACL provides low-level routines (e.g., *divide*, *add*, and etc...) and Montgomery arithmetic routines (e.g., *nres_modmult*, *nres_moddiv*, and etc...) through a collection of assembly-optimized functions/methods. It also provides high-level routines (e.g., Elliptic curve routines, encryption routines, and etc) in C/C++, which calls the low-level or Montgomery arithmetic routines for computing the computationally demanding operations. We are not only looking for the efficiency of computing one EC point multiplication, but also that of modular multiplication. As it will be presented in Chapter 7, in MIRACL the modular multiplication operations do occupy 62% to 71% of the

entire computing time of one EC point multiplication. This stresses again that the modular multiplication is indeed the most computationally demanding operation. The assembly-optimized *nres_modmult* routine is in fact used for computing modular multiplication and invokes either Karatsuba-Comba-Montgomery (KCM) or Comba-Montgomery Modular Multiplier [28].

Here, the Montgomery algorithm is used for the reduction purpose while the Karatusba-Comba is used for the purpose of computing the product of two multi-precision integers. The combination of these algorithms used to speedup the computation is quite common in software implementation since Karatsuba-Comba algorithm allows relatively fast product of two multi-precision integers and Montgomery algorithm allows relatively fast reduction without trial division. Nevertheless, this algorithm is not well suited for hardware, especially in FPGAs. This is because constructing a 160×160 multiplier from the 18×18 hardware multiplier that modern FPGAs have support for would result layers of partial products that would need tree-base Carry-Save Adder (CSA) to complete the computation. This would result in large number of registers and large number of global interconnects. On the other hand, the Montgomery Modular Multiplication (MMM) algorithm can be implemented in block-wise or bit-wise, which allows us to implement it efficiently in both software and hardware; software is generally good for the block-wise processing while fine-grain FPGAs are generally good for the bit-wise processing Thus, we consider Montgomery Modular Multiplication (MMM) algorithm for computing modular multiplication so that both software and reconfigurable solutions can be compared fairly.

4.4 Conclusions

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 has ASIC-like computing power and the GPP flexibility.

We have also reviewed a number of MMM algorithm implementations for public-key cryptography. At the end, we settle with the ASIC-based Montgomery Modular Multiplication functional unit by Goodman and Chandrakasan [19], which we believe has the greatest potential to be mapped onto FPGA platform and improve upon. In addition, we are able to narrow down which fast-adder is most suitable and feasible in the FPGA platform by implementing their high-speed carry-skip adder in Xilinx FPGA and analyzing the pros and cons of such adders. MIRACL, the library that we use for the purpose of the effectiveness and correctness of our C-level software, is also discussed. At the same time, the advantages and disadvantages of using Karatsuba-Comba or Montgomery modular multipliers are discussed.

As mentioned, having a working and relatively high-performance C-level pure-software framework is essential for a fair comparison with the reconfigurable solution. Thus, in the next chapter, the software implementation for public-key algorithms is presented. In particular, we pay more attention to the modular multiplication operation as it is the bottleneck in software. Finding an efficient algorithm for computing modular multiplication is no easy task. However, Montgomery Modular Multiplication (MMM) algorithm is proven to be quite efficient, and thus, is discussed in great detail.

Chapter 5

Public-key algorithms software implementation

Essentially, what we are really interested in seeing is how much speed-up that the reconfigurable solution is able to provide in comparison with pure-software solution. However, in order to obtain a fair comparison, we need a relatively high-performance C-level software implementation running on MicroBlaze. This chapter provides a detailed software implementation and is organized as follows. The software implementation for modular operations are first presented, including modular addition/subtraction, multiplication, and inversion. In particular, the Montgomery Modular Multiplication algorithm is discussed in details with its block-wise and bit-wise examples. It is then followed by the EC point operation implementation.

5.1 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, we begin to build a software solution with the implementation of the finite field arithmetic with operations including modular addition/subtraction, multiplication, sqauring, exponentiation, and inversion for long-word integers. In our implementation, modular inversion operation is computed using Fermat's Little Theorem, and modular multiplication is described using the Montgomery modular multiplication algorithm.

5.1.1 Finite-field arithmetic implementation on prime

The finite field on prime, $GF(p)$, 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)).

To start with the simplest modular operation, Algorithm 5.1 presents the pseudo code for modular addition. While addition is one of the most fundamental operations that an ALU supports, it is not as straighforward to add two numbers in a crypto application since it exceeds the processor's operating width (e.g., 32 bits). Thus, carry bit is generated from each 32-bit block of data. Thus, long-word operation needs to be emulated in software and a routine is required to accomplish such task; it includes a *LOOP* going through the appropriate number of blocks in the array and the detection of the carry bit from each block. This is exactly what the *LongAdd* and *LongSub* functions in line 1 and 3 do. There is no easy way of accessing the carry bit of a word in C. One way to detect the carry bit is to check if the resulting 32-bit addition is less than the current 32-bit word of the second

operand - if it is true, then the carry bit is set; otherwise, it is not set. And we need to do this carry-bit detection for every single 32-bit addition. To complete the modular addition, we check if the long-adding resulting value is greater than the modulous p. This is done by comparing the 32-bit word of c against that of p from the most significant block to the least significant block in *checkReduct* function in line 2. If it is true, an additional subtraction is required, as shown in the modular addition example. Similar approach can be applied on the modular subtraction with minor modifications in the code.

Algorithm 5.1 Modular Addition

Require: Integers $a, b \in [0, p-1]$, and a modulus p

Ensure: $c = (a+b) \bmod p$

1: $carry \leftarrow \text{LongAdd}(a, b, c)$
2: **if** checkReduct(c, p) **then**
3: LongSub(c, p, c)
4: **end if**
5: **return** c

As mentioned earlier, modular inversion can be eliminated from the point operation level (namely, EC point addition and doubling) by carrying out the point operations in projective 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 5.1

$$a^{-1} = a^{p-2} (\text{mod } p), \text{if } gcd(a, p) = 1 \tag{5.1}$$

The main contribution using this theorem is that the 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. This is exactly what is also being performed in RSA. Due to the use of this theorem, the efficiency of the EC point multiplication relies

on modular multiplication implementation even more. Hence, it is crucial to implement an efficient modular multiplier.

As a result, a various standard key lengths of point multiplication operations in ECC are used as the testing cases since the modular exponentiation operation required in RSA is also tested and verified by the modular inversion required in the ECC implementation. Therefore, due to the modular exponentiation and modular inversion are now heavily based on modular multiplication operation, modular multiplication is hereafter the most important and the most timing consuming operation in the finite filed arithmetic level operation.

To further illustrate modular multiplication is indeed the bottleneck in software, we present one of our experimental results, which reveals the number of cycle counts that modular multiplication operations in one EC point multiplication are required with respect to that of one EC point multiplicaiton. This numerical figure shown in Figure 5.1 is for the 192-bit length. The first group represents the cycle count from the MIRACL library running on a Pentium processor. The second group represents that from our C-level software also on a Pentium processor. The third group represents the cycle count from the same C-level software running on a MicroBlaze-based platform.

It is apparent that a large percentage of the computing time in an EC point multiplication is spent on performing modular multiplication. It is worth reminding that in RSA modular multiplication is also heavily used to implement modular exponentiation, as shown in Algorithm 2.2. Therefore, in order to increase the computing speed, providing reconfigurable hardware support for modular multiplication is a good approach.

Thus, we have proved that modular multiplication is the most computationally demanding in the finite field operations as modular exponentiation and inversion are now computed using 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 the 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

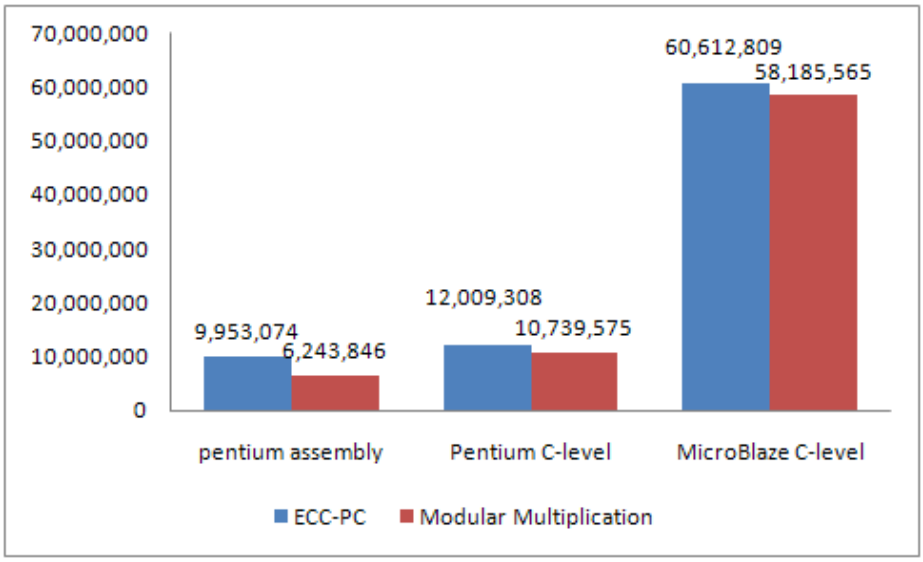


Figure 5.1: Cycle count of modular multiplication vs that of one ECC point multiplication in 192-bit keylength.

software and hardware implementations. Some of the most popular algorithms for modular multiplication and reduction are listed in Table 5.1.

Moreover, if the prime that is used in EC point multiplication is one of the NIST recommended primes, then the fast reduction can be applied, which by far is the fastest method for 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 alogrithms. Due to the fact that a processor can operate better on data that has the same bit-width as the processing bandwidth, the Karatusuba-Ofman multiplier [26] in conjunction with Montgomery reduction [38] is a popular algorithm for computing modular multiplication in software implementation as mentioned in Chapter 4. The Montgomery reduction is a simplified version of Montgomery Modular Multiplication algorithm that is introduced in the next subsection - it performs reduction modulo-p utilizing simple shift operations, which can

Table 5.1: A list of possible algorithms for modular multiplication and reduction

Methods	Description
NIST recommended curves (fast reduction) [18]	is the fastest reduction method. Only for Mersenne primes and polynomials
Montgomery reduction [38]	is used in conjunction with other multiplication algorithms.
Montgomery Multiplier [38]	does the modular multiplication. Dual field support - prime and binary fields.
Shift-and-add Multiplier [21]	performs only regular multiplication → need to be combined with reduction algorithm to complete modular multiplication.
Karatsuba-Ofman Multiplier [26]	performs only regular multiplication → need to be combined with reduction algorithm to complete modular multiplication.

be implemented very efficiently even on conventional processors. However, this approach is not well suited to hardware, especially in FPGAs due to FPGAs are better suited for bit-wise processing, as opposed to word-wise processing. On the other hand, Montgomery Modular Multiplication (MMM) is a good choice for both software and hardware implementations since it can be implemented in either word-wise or bit-wise. Consequently, MMM algorithm is used for computing the modular multiplication operation.

5.1.2 Montgomery Modular Multiplication

Among all the proposed algorithms, Montgomery Modular Multiplication (MMM) algorithm [38] is proved to be a very efficient algorithm - it replaces the division operation with a series of multiplications, additions, and shifts to perform the reduction. In addition, MMM was also shown 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 a polynomial [29]. Therefore, we 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 5.2.

Algorithm 5.2 Montgomery modular multiplication with final 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 $x, y \in [0, p-1]$, $R = (2^\alpha)^l$, $gcd(p, 2^\alpha) = 1$ 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)

The basic idea behind Montgomery's algorithm is that, given an odd n-bit modulus, p, and two multi-precision operands, x and $y \in [0, p-1]$, whose product can be reduced to a n-bit value by a right-shift of n positions. Among all the required parameters, R is the scaler factor and p' is the value needed to be pre-computed if $\alpha > 1$, and is guaranteed to exist as long as p is relatively prime to 2^n, which is always true given that p is odd for all public-key algorithms. As noticed in Algorithm 5.2, the output of the MMM is (xyR^{-1} mod p), instead of (xy mod p) that we are looking for. Basically, the R term is equivalent to 2^n, where n is the bit-length of the modulus, p. In order to obtain the correct value of (xy

mod p) result, an additional MMM operation is required. The operands of this additional MMM are the result from the last MMM operation and the value of (R^2 mod p). By doing so, the (R mod p) term is to compensate the (R^{-1} mod p) introduced in the MMM itself while another (R mod p) term is to compensate the (R^{-1} mod p) from the last MMM. This is essentially the overhead that MMM creates. Hence, the benefit of the efficiency of MMM is diminished because of the overhead it creates when only one or a few modular multiplication operation is required. In other words, the efficiency will only be obtained from performing MMM if a series of modular multiplications are need, such as 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 of the EC point multiplication, such that $\bar{x} = Mont(x, R^2) \equiv (x \cdot R) \bmod p$. 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 $x = Mont(\bar{x}, 1) \equiv (x) \bmod p$.

In Algorithm 5.2, it is apparent that the core of this algorithm lies in Line 3 and Line 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., $56 \bmod 11 \equiv 56 + (3 \times 11) \bmod 11 \equiv 1$). Thus, this technique is used in Line 4 of Algorithm 5.2 to zero the least significant block.

Implementing Algorithm 5.2 is not an easy task as it might seem to be, mainly because we are dealing with long-integer operations. Its pseudo code is presented in Algorithm 5.3. Since the resulting value from Line 5 is mod 2^{32}, meaning it is always less than $2^{32} - 1$, we can just extract the lower 32-bit word from the result of the 32-bit multiplication. The

LongMulti function is implemented to multiple a 32-bit word of one operand to the 2nd operand. The number of blocks of the resulting array is always $l+1$, where l is the number of blocks in the array. To implement the *LongMulti* function in Line 6 and Line 8, $32 \times 32bits$ multiplication and intermediate 32-bit addition for the partial products are utilized inside a *LOOP* statement. The *LongShift* function in Line 10 is to shift the entire resulting variable array, T, by one 32-bit word to the right after the least significant word is zeroed out. Furthermore, A reduction of modulus p is performed if T_l is greater than p to ensure the modular property is met, which is in Line 12 and 13.

Algorithm 5.3 Pseudo code for the 32-bit word-wise MMM with final subtraction

Require: Integers $x, y \in [0, p-1]$, a modulus p, and $p' = -p^{-1} \bmod 2^{32}$

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

1: **for** i from 0 to $(l-1)$ **do**
2: $\quad T(i) \leftarrow 0$
3: **end for**
4: **for** i from 0 to $(l-1)$ **do**
5: $\quad m_i \leftarrow \text{getMi}(x(i), y(0), T(0), p')$
6: $\quad tmpT \leftarrow \text{LongMulti}(x(i), y)$
7: $\quad carry \leftarrow \text{LongAdd}(T_i, tmpT, T_i)$
8: $\quad tmpT \leftarrow \text{LongMulti}(m_i, p)$
9: $\quad carry \leftarrow \text{LongAdd}(T_i, tmpT, T_i) + carry$
10: $\quad T_{i+1} \leftarrow \text{LongShift}(T_i)$
11: **end for**
12: **if** checkReduct(T_l, p) **then**
13: $\quad \text{LongSub}(T_l, p, T_l)$
14: **end if**
15: **return** (T_l)

As it can be seen, implementing the Montgomery Modular Multiplication (MMM) algorithm to compute modular multiplication operation requires many *LOOPs* to emulate the long-integer operation in the 32-bit processing bandwidth environment. Even though MMM algorithm provide an efficient method for computing modular multiplication, all these calls have high demand on execution cycles. Thus, its software implementation is still considered to be slow in performance.

Furthermore, while word-wise MMM implementation is preferable in the software approach, bit-wise MMM implementation is often favored in the hardware approach, particularly for the FPGAs. Word-wise MMM means that the block size, α, is greater than 1, usually equals the processing bandwidth, which is normally 16 bits or 32 bits so that it can be efficiently executed. On the other hand, bit-wise MMM means that the block size, α equals 1, leading to one of the operands being scanned bit by bit from the least significant bit to the most significant bit. Even though the bit-wise implementation yields the largest number of iterations, l, it does simplify Algorithm 5.2, such that Line 3 and Line 4 in Algorithm 5.2 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 Line 4 in Algorithm 5.2 disappear, which also makes the Montgomery Modular Multiplication easier mapped onto FPGAs. Below is an example to illustrate how MMMA works in bit-wise 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).

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

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-wise MMM. In this case, the number of iteration is 5, which equals to the bit-length of the modulus. Due to α equals to 1 in the bit-wise MMM, $p^{'}$ also equals to 1. Table 5.1.2 reveals the results from line 3 and 4 of Algorithm 5.2 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 first 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-wise MMM. This demonstration also reveals that the bit-wise MMM does require more iterations than the block-wise to complete the computation. However, the operations are much simplier (e.g., addition and shift).

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

The next level of operation to build is the EC point operation, which are EC point addition and doubling required in EC Point Multiplication stated in Algorithm 2.3 and is presented in the next subsection.

5.1.3 EC point operation implementation

As mentioned earlier, different coordinates in which the EC point multiplication is operated yields different operations in EC point addition and doubling. The coordinate that

we decided to use is the modified Jacobian (J^m) coordinates proposed by Cohen *et al.* [9]. This is because that this coordinate 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 [41]. In the (J^m) coordinate, the total number 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-Add 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 2.2. In addition, the EC point variables in normal representation are needed to be converted to Montgomery representation. Hereafter, all the arithmetic are carried out in the Montgomery representation. At the end of each EC point multiplication, converting the J^m coordinate representation back to the Affine coordinate is needed. In other words, we want the 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 was described in the previous subsection.

5.2 The usage of MIRACL

Since we deal with long-and-very-long integer operations, the computing results can not be simply verified by hand, nor by any ordinary high-level language tool, such as MATLAB. Fortunately, a Multi-precision Integer and Rational Arithmetic C/C++ Library called

MIRACL, licensed through Shamus Software Ltd [51], is found and imported to Microsoft Visual Studio. This library has been used in many embedded application projects and is recommended in the book titled *"Guide to Elliptic Curve Cryptography"*. MIRACL is programmed to be assembly-optimized for Pentium processor compiled in Microsoft Virtual Studio. By comparing our C-level program performance in cycle count against with MIRACL, it provides us some numerical figures on the effectiveness of the C-level program. As we will see in Chapter 7, this library's performance in cycle count is within the scale of $2\times$ when compared to the C-level program for computing one EC point multiplication. This result in turn ensures us that our C-level program is relatively comparable.

As the software implementation for computing EC point multiplication is developed and ran on the customized hardware platform with a MicroBlaze core-processor, the next step is to present the MMM hardware unit on Xilinx FPGA and our new proposal, Cryptography-oriented Reconfigurable Array (CryptoRA).

Chapter 6

Montgomery Modular Multiplier in FPGA

Since Montgomery Modular Multiplication (MMM) algorithm is chosen to be implemented bit-wise in hardware, the long-integer addition/subtraction and comparison become the core operations, as discussed in the previous chapter. Thus, several fast adders' implementations on Xilinx Virtex-II Pro FPGA are covered in this chapter. First, we present the MMM implementation on Xilinx Virtex-II Pro FPGA and highlight the limitations in such a FPGA device. This is followed by the implementation of a new comparator unit structure for building a FPGA-based MMM hardware in *CryptoRA*. This chapter is completed with the *CryptoRA*'s features which is expected to reduce the critical path and slice usage when long-integer operations are mapped onto *CryptoRA* FPGAs.

6.1 Montgomery modular multiplier in Xilinx

Since the Montgomery modular multiplication is recursion, its two additions can be implemented using a redundant carry-save representation, as proposed by Goodman and Chandrakasan [19]. 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.

6.1.1 Carry-Save Adder in Xilinx

To avoid the carry propagation, carry-save addition uses a redundant representation for the result comprising separate sum and carry bit words [43]. Carry-save addition is especially appropriate 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 adder structure over another depends on the platform on which the adder is implemented. For instance, in ASIC, fast adder structures are much faster than ripple-carry adders; in FPGAs, the performance is dependent on whether if the particular fast adder structure is given support (e.g., Xilinx provides dedicated resources to support carry lookahead, whereas Altera supports carry select in hardware). Even though carry-save addition is fast, the extra usage on the Flip-Flop for storing the carry bit and the conversion to the non-redundant representation at the end are the downside using such approach. This is an issue in both ASICs and FPGAs, especially in FPGAs.

While the MMM is often implemented in processor-word length (e.g., 32-bit) in software, it is chosen to be implemented in bit-wise in our hardware implementation. It is worth reminding that the disparity in block-wise and bit-wise processing of the MMM algorithm. That is, in the block-wise implementation, the number of bits that is scanned at each *FOR* loop must be greater than 1-bit; in the bit-wise implementation, bits are scanned only one at a time. Thus, bit-wise MMM implementation does deteriorate the performance in software because of more iterative loops. However, it has the opposite effect on the hardware implementation due to the following:

- Possible shorter critical path delay of MMM functional unit is obtained.
- Hardware parallelism is taken advantage of as long as the silicon area permits.

Our hardware MMM functional unit is based on the Goodman's ASIC implementation [19]. As line 3 and 4 in Algorithm 5.2 are simplified in the bit-wise implementation as described in Chapter 5, all the operations in line 3 and 4 can be emulated using two levels of 3-to-2 Carry-Save Adder (CSA). It is detailed in the form of bit-slice configuration, shown in Figure 6.1. In this figure, one of the inputs to the *AND* gate at the top is the current scanned bit (1 or 0) of the operand x, which is broadcast throughout the array. If $x(i)$ is 1, then the operand y is added, otherwise 0 is added. Similarly, the 2nd *AND* gate is used to detect the varaible m_i bit, which is also broadcast 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.

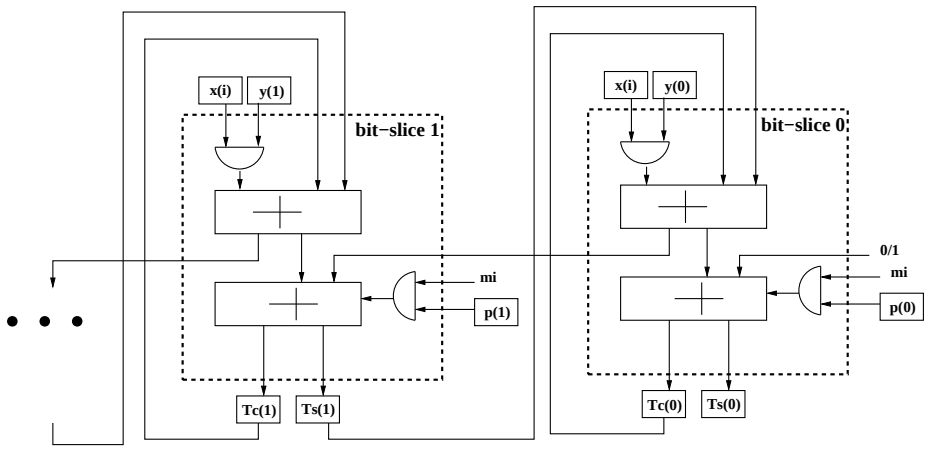


Figure 6.1: Bit-slice of CSA implementation.

Furthermore, the shift-to-the-right by 1 bit operation is emulated by feeding the resulting sum bit at the output of the second CSA one position back (e.g., $i-1$), and feeding the carry bit back at the same position i for the next iteration as illustrated in Figure 6.1. Therefore, this specific configuration allows all the operations in line 3 and line 4 of Algorithm 5.2 to be executed in one hardware clock cycle. This results in a very low clock cycle count per MMM operation, which can be generalized as n+3, where n is the word-length. These extra 3 cycles come from the ripple carry addition, comparing, and selecting appropriate results. However, these cycle counts exclude the cycle count for uploading and downloading the inputs and outputs. For instance, in 192-bit ECC, it would take 195 hardware clock cycle to complete one modular multiplication.

Although this carry-save adder implementation is very simple, it has three major drawbacks:

- Expensive in terms of the slice usages and the global interconnects.

- Large number of fan-out from signals, $x(i)$ and m_i.
- Still need to convert back to the non-redundant form.

First, this is an expensive technique since two tiles are used per bit to produce in parallel the sum and the carry bits. More importantly, 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 broadcasted to $2 \cdot n$ computing tiles, which leads to a fan-out of $2 \cdot n$ (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 actual 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 (CLA), can be considered to be utilized for further performance improvement and are the possible solutions to the third issue.

6.1.2 Ripple-Carry Adder in Xilinx

To convert the adding result from the Carry-Save Adder (CSA) in the redundant form to the non-redundant form, other adder techniques are needed. We choose Ripple-Carry Adder (RCA) for such task even though its carry propagation delay has exponential relation with the increase in bit length. It is apparent that the critical path includes the carry propagation. This translates to a large latency in cryptography, where long-word addition is required. Therefore, fast addition techniques are worth being investigated, such as the Carry-Lookahead Adder (CLA). However, although CLA is given hardware support in the Xilinx Virtex-II Pro FPGA, performing a subtraction operation requires extra LUTs and

thus, increases its critical path and slice usage. Also, because this Xilinx chip has hardware support for RCA by means of the dedicated carry path as shown in Chapter 4, RCA is chosen for both the redundant conversion and the subtraction operation required in MMM algorithm to be implemented in Xilinx FPGA.

6.1.3 N-bit comparator unit in Xilinx

As seen in Algorithm 5.2, the MMM algorithm consists of a comparison of two long-unsigned integers. J. Goodman and A. Chandrakasan proposed an implementation of the compare unit in ASIC [19]. The equations for the fast tree-based comparator are as follows.

$$EQ(i) = \overline{x(i) \oplus y(i)} \tag{6.1}$$

$$GT(i) = x(i) \cdot \overline{y(i)} \tag{6.2}$$

$$EQ(j+1,i) = EQ(j,2i+1) \cdot EQ(j,2i) \tag{6.3}$$

$$GT(j+1,i) = GT(j,2i+1) + GT(j,2i) \cdot EQ(j,2i+1) \tag{6.4}$$

Equations 6.1 and 6.2 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 $y(i)$ is 0, meaning x is greater than y at bit position i. Two bits adjacent to each other in the array are then used as the inputs to compute the next level of the comparator tree using Equation 6.3 and Equation 6.4. The Equation 6.3 can be interpreted as the following: the output from this equation is only set to 1 if the compared bits are equal. Equation 6.4 can be interpreted as the following: the output is set to 1 in two scenarios - one scenario is when the higher order bit of which two compared bits is 1, implying $x(i+1)$ is greater than $y(i+1)$, (ie: $GT(1,1) = 1$). Another scenario is when 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, $x(0) > y(0)$ and $x(1) = y(1)$. In other cases, the output in Equation 6.4 is 0.

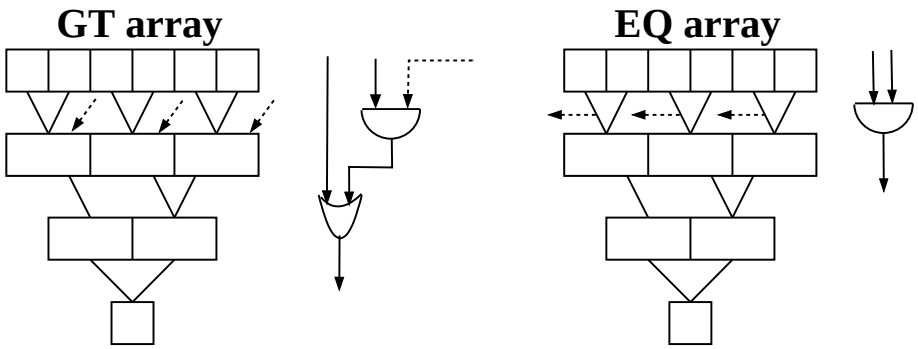


Figure 6.2: Original GT and EQ flags in parallel structure

Figure 6.2 is the graphical representation of the Equation 6.1 to Equation 6.4. The topmost arrays are computed using the Equation 6.1 and Equation 6.2 for *EQ* and *GT* arrays, respectively. The array in the rest of the layers are computed using Equation 6.3 and Equation 6.4.

Moreover, in order for the comparator unit to make use of the dedicated carry chain on *CryptoRA*, some modifications on the original equations, mainly Equation 6.1 to Equation 6.4, are required. Those equations become the Equation 6.5 to Equation 6.8 accordingly after the changes are made. The outcome from these modified equations remains the same as long as an inverter is present at the end of the final EQ signal. These equations are indeed used in the Virtex-II based comparator unit to ensure that its functionality is unaltered after the changes. With these changes, the equations can be further modified to better use the features in *CryptoRA*-based FPGA, which will be covered in the later section.

$$EQ(i) = x(i) \oplus y(i) \tag{6.5}$$

$$GT(i) = x(i) \cdot \overline{y(i)} \tag{6.6}$$

$$EQ(j+1,i) = EQ(j,2i+1) + EQ(j,2i) \tag{6.7}$$

$$GT(j+1,i) = GT(j,2i+1) + GT(j,2i) \cdot \overline{EQ(j,2i+1)} \tag{6.8}$$

Furthermore, the latency of the fast tree-based comparator unit for comparing two n-bit operands is $O(log(n))$ gate delay in ASIC [19]. However, this claim does not apply in 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 is implemented on an FPGA, $O(log(n))$ is the number of interconnect that it requires. In order to fit this comparator unit better in *CryptoRA* FPGA by means of the dedicated carry path, we propose a new sequence of comparing the bits. That is, changing it from its original parallel structure to a serial structure, which is emulated using the dedicated chain. The detail on this proposal is presented in the next section.

6.2 Architectural supports from *CryptoRA*

Even though our Montgomery Modular Multiplier, which is initially based on the ASIC hardware, has been successfully implemented on Xilinx Virtex-II Pro FPGA, it also creates some drawbacks on the FPGAs. The main drawbacks are:

- The extra silicon area required for storing the *sum* and *carry* bits in CSA.
- The long critical path created by CSA, comparator unit, and especially by RCA.

In order to alleviate these issues, we propose a new Cryptography-oriented Reconfigurable Array, called *CryptoRA*. Its features include an increased granularity of the logic tile, the extension of the dedicated carry chain over the horizontal direction, and the split LUT, which will be discussed extensively starting from Section 6.3 to 6.5.

6.2.1 Carry-Lookahead Adder in *CryptoRA*

As covered in Chapter 3, Carry-Lookahead Adder (CLA) is given fully architectural support on Xilinx Virtex-II by means of the dedicated carry chain, the internal *AND*, and *XOR* gates. However, Virtex-II Pro FPGA has a limitation in its carry-chain: it can only

perform fast addition - not subtraction. The dedicated *AND* gate only allows for addition to be performed. In order to perform subtraction, an additional stage of computing tiles would have to invert the subtrahend is required. This technique would introduce a very large penalty on the operation, as it would require an additional computing tile delay and, worse, another global interconnect delay. Therefore, this architecture are not well suited to long-word subtraction. This is illustrated in Figure 6.3 using one bit element. The inversion of the subtrahend is accomplished by the *XOR* logic, which is configured in LUT. The output of this computing tile is then propagated through global interconnect and is used as one of the inputs for performing the fast addition.

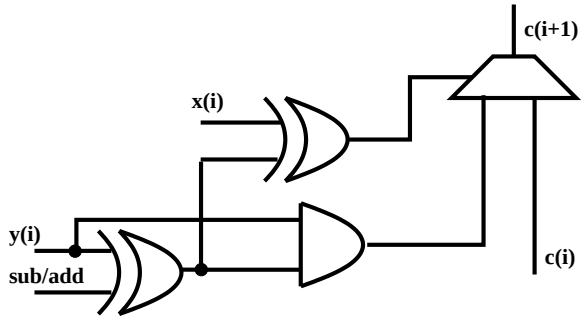


Figure 6.3: Sub. operation in one of the elements in CLA

This issue is addressed and can be solved using the split LUT feature of *CryptoRA*, whose functionality is discussed later in this chapter. The idea of split LUT feature regarding this issue is that the dedicated *AND* gate is now replaced by the lower part of the four-input LUT. This in turn gives the freedom of having any independent three-input logic being configured in that lower LUT. Thus the *XOR-AND* logic, which is needed for subtraction in CLA, can be configured in the lower LUT.

6.2.2 Comparator unit in serial structure

In order to have architectural support for a comparator unit in *CryptoRA*, the original comparator equations are needed to be modified to a CLA-like form so that it can then be

emulated using the dedicated carry chain. We realize that the results from the comparator are not altered when a different sequence of comparing bits is applied. Thus, instead of comparing bits in pairs in parallel, the bits are now compared serially. The modified comparator equations (mainly Equation 6.5 to 6.8) are became Equation 6.9 to 6.12, which now look similar with the CLA equations.

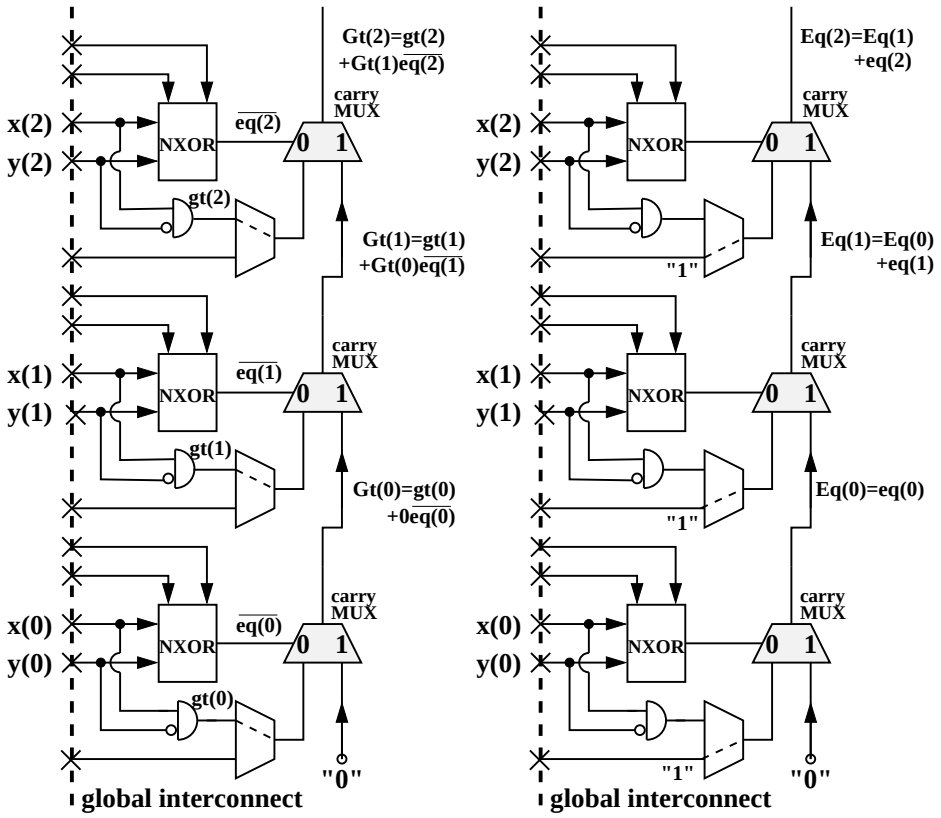
$$eq(i) = x(i) \oplus y(i) \tag{6.9}$$

$$gt(i) = x(i) \cdot \overline{y(i)} \tag{6.10}$$

$$Eq(i) = eq(i) + Eq(i-1) \tag{6.11}$$

$$Gt(i) = gt(i) + Gt(i-1) \cdot \overline{eq(i)} \tag{6.12}$$

As a result, the Gt and Eq signals can also be emulated using the dedicated carry chain in the similar way as the g_b and p_b signals in CLA. These mapping results at the component level are presented in Figure 6.4. While Figure 6.4(a) shows the dedicated carry chain for emulating the Gt signal, Figure 6.4(b) shows the chain for emulating the Eq signal. Because of the $\overline{eq(i)}$ term in Equation 6.12, the LUT is configured as the *NXOR* logic. The output of each LUT in the chain is used as the selecting signal of carry MUX. To map the Gt signal, "0" is fed to the 1-input of the first MUX, and the output of the *AND-with-one-inverted-input* logic, emulating the gt signal, is fed to the 0-input of every MUXes. For instance, if $\overline{eq(i)}$ is 0, 0-input of the MUX is selected; otherwise, 1-input is selected, which emulates the Equation 6.12. As noticed, we need *AND-with-one-inverted-input* logic to be able to implement the comparator unit using the dedicated carry chain and this can not accomplished in Xilinx Virtex-II Pro FPGA. However, *CryptoRA* can again accommodate this need with its split LUT feature, which will be presented in the later section. Similarly, to map the Eq signal, "0" is fed to the 1-input of the first MUX, "1" is fed to the 0-input of every MUXes, and the MUX is also controlled by the output of LUT, which is configured



(a) Gt signal using dedicated carry chain (b) Eq signal using dedicated carry chain

Figure 6.4: The generation of GT and EQ signals in comparator unit using dedicated carry chain

as *NXOR* logic. For this reason, sharing-one-LUT feature in *CryptoRA*, which is also discussed later in this chapter, can be applied to implement the comparator unit.

According to the Equation 6.9 and 6.10 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, meaning $x(i)$ 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.

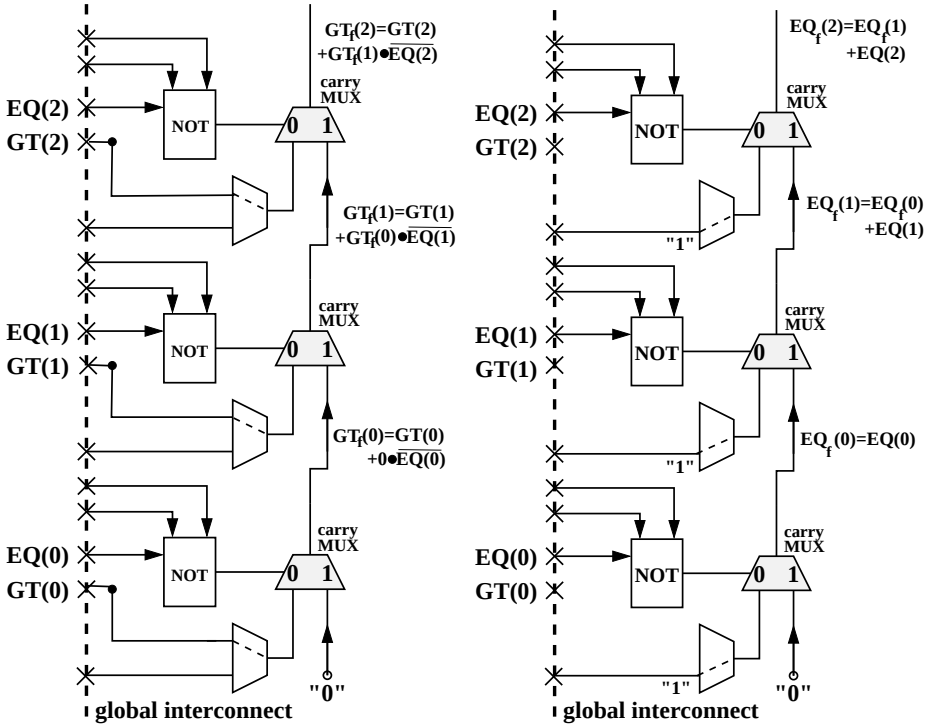
Moreover, if the comparator unit were implemented on *CryptoRA*, the critical path would be the LUT delay plus the delay from n number of MUXes in the dedicated chain, which essentially equals to a ripple carry delay for adding two n-bit integers. To reduce the delay, a n-bit block can be divided into j blocks, each having length of i-bits since there is no dependency on the sequence of the comparison on which 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 ($GT_f(m)$) signals. The equations for the final EQ and GT signals are shown in Equation 6.13 and 6.14, respectively.

$$EQ_f(m) = EQ(j) + EQ_f(m-1) \tag{6.13}$$

$$GT_f(m) = GT(j) + (GT_f(m-1) \cdot \overline{EQ(j)}) \tag{6.14}$$

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. Again, the GT_f signal is emulated using the dedicated carry chain and is shown in the Figure 6.5(a). Its carry chain is configured with "0" fed to the 1-input of the first MUX, the previous GT_f signal fed is to the 1-input of every other MUX, and the $GT(j)$ signal fed to the 0-input of every MUX. The LUTs are configured as *NOT* logic, to invert the $EQ(j)$ signals, which are used as the selection signals of the carry MUXes, providing the logic function for GT_f. Similarly, the EQ_f signal is also emulated using the dedicated carry chain and is shown in Figure 6.5(b). Its carry chain is configured with "0" fed to the 1-input of first MUX, previous EQ_f signal fed to the 1-input of every other MUX, and "1" fed to the 0-input. The LUTs are configured as *NOT* logic, to invert the $EQ(j)$ signals, which are used as the selection signals of the carry MUXes, providing the logic function for EQ_f.

From the equation aspect, if $EQ(j)$ is 1, the second term of the Equation 6.14 is in-



(a) GT_f signal using dedicated carry chain (b) EQ_f signal using dedicated carry chain

Figure 6.5: The generation of final GT and EQ flags in comparator unit using dedicated carry chain

significant 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, the same logic function from LUT (NOT) is used, with 0-input of MUX connected to $EQ(j)$ and 1-input of the MUX connected to 0, which is shown in Figure 6.4(b). Such connection ensures that the second term of the Equation 6.13 is the dominant term to determine the output when $EQ(j)$ is 0.

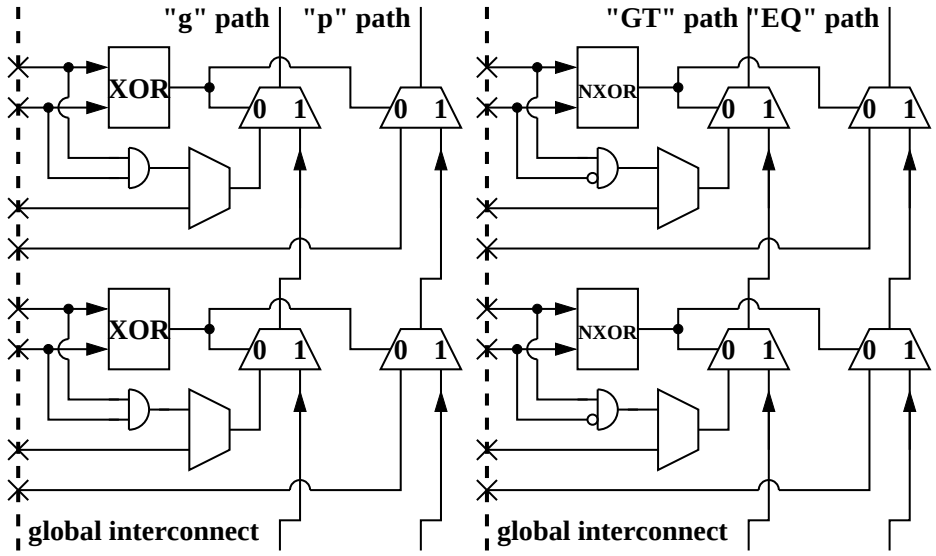
We have, thus far, seen how to make use of the dedicated carry chains to emulate both the Carry-Lookahead Adder and comparator unit. In the next few sections, we will present

the three features that we include in *CryptoRA* FPGA in details. We will also show how much of the impacts that this novel FPGA structure would have in terms of the critical path and slice usage when it is used for mapping long-integer CLA and comparator unit.

6.3 Two dedicated carry MUXes driven by 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 [11], [12], [13]. Thus, we propose to increase the FPGA granularity such that a LUT drives two dedicated MUXes, as shown in Figure 6.6(a) and in Figure 6.6(b) for carry-lookahead adders and fast comparator unit, respectively. In Figure 6.6(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 6.6(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.

Such connection was realized by the fact that in both CLA and comparator unit implementations, the 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 slice usages. For example, the block-level *generate* and *propagate* signals can now be implemented within one slice, instead of two, resulting 50% saving on the slice usage. Similar result applies on the comparator unit implementation as well.



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

Figure 6.6: LUT for 2 MUXes

6.4 Horizontal dedicated path

As mentioned, although carry-lookahead addition is given architectural support in Xilinx FPGA, the block-level *generate* and *propagate* signals are still routed through the global interconnect. To improve the propagation delay 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 presented in Figure 6.7. This way, the *generate* and *propagate* signals can use a dedicated, thus fast, interconnect.

To demonstrate this idea on CLA, single-stage CLA is used as an example and is presented in Figure 6.8. It shows that having the block-level *generate* ($g_b(j)$) and *propagate* ($p_b(j)$) signals on the horizontal dedicated path, which are then selected to generate the block-level *carry* signals can bypass the LUT, and creates (theoretical) 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. As far as this example goes, the 2nd dedicated carry path is idle because to generate the sum bits, it requires different logic function.

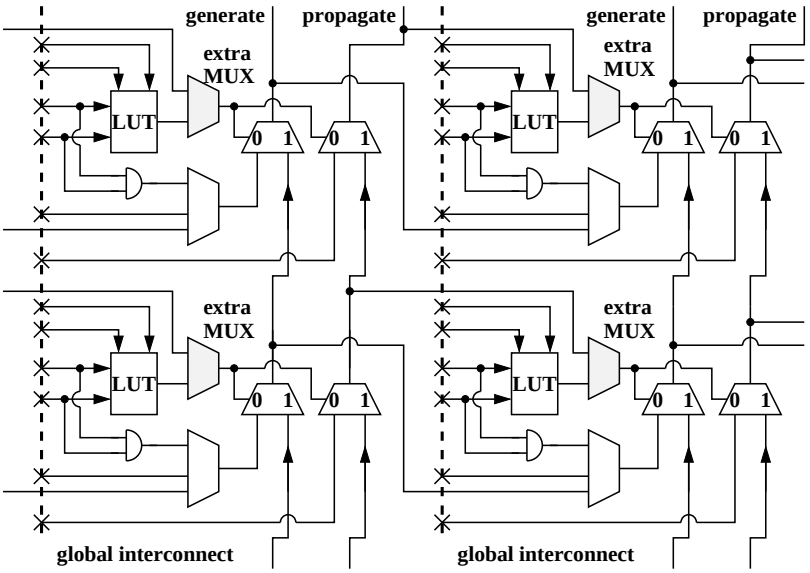


Figure 6.7: Carry network extended horizontally.

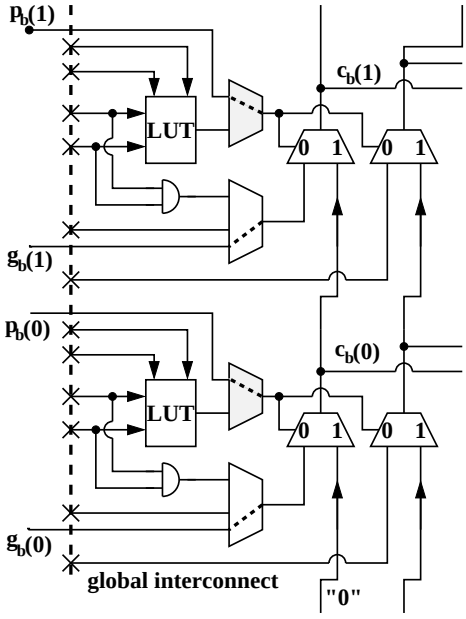


Figure 6.8: Single-stage CLA using horizontal dedicated path.

Moreover, to fully take advantage of this new proposal of extending the dedicated carry chain over the horizontal direction for a 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 6.14, 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 dedicated input for the inverted *propagation* signal to the *extra MUX* in Figure 6.7 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 benefits more from this feature as both GT_f and EQ_f signals require the 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.

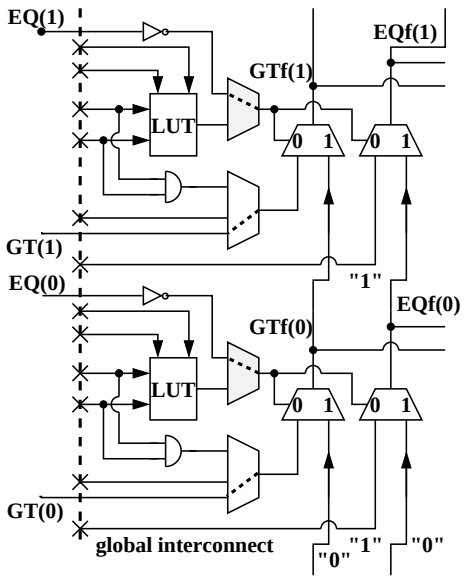


Figure 6.9: Comparator unit using horizontal dedicated path.

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 [63], so the latency will not be significantly affected on such FPGAs. With respect to Figure 6.10, there are two ways to add a MUX: (i) extending the carry MUX (Fig. 6.10-(a)), or (ii) additional MUX per se (Figure 6.10-(b)). By using the unfolded-MUX Level-Restoring Buffer (UMUX-LRB) described in Miller and Sima's paper [36] to improve the switching characteristics, they determined that the later approach shown in Figure 4-(b) is 40% faster. Therefore, extending the carry MUX is not considered any longer.

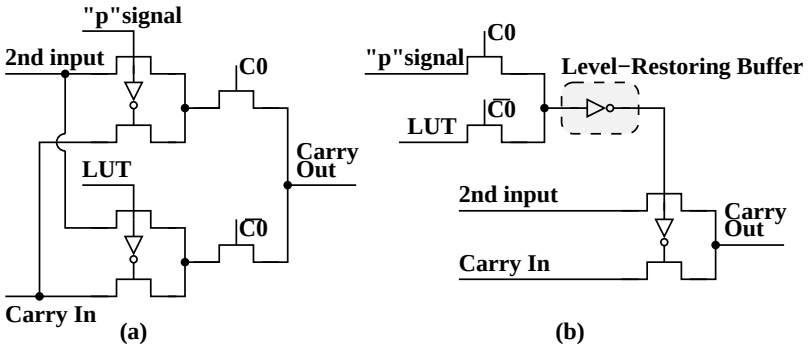


Figure 6.10: Solutions in adding an extra MUX.

6.5 Split LUT structure

There are two major limitations on the fast carry-lookahead network presented in Figure 3.4. 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 6.11 and 6.12. 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 split-

ting 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.

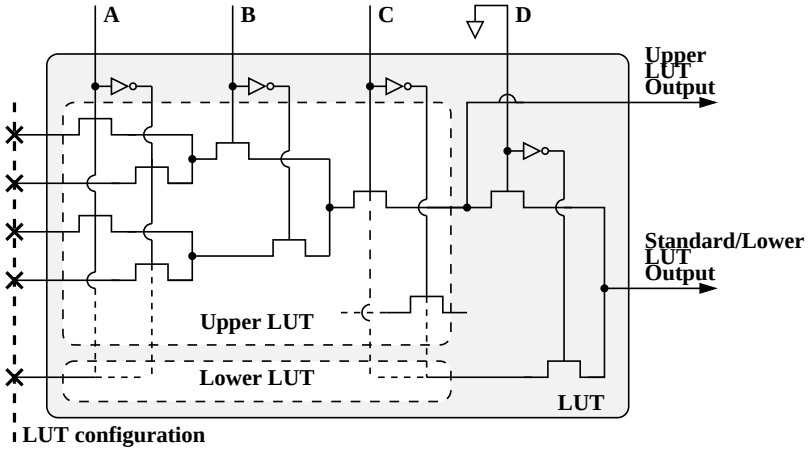


Figure 6.11: Split LUT - transistor level.

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 to the MMM functional unit. With the additional computation capability, it is possible to build an ALU using carry-lookahead arithmetic. In summary, as the current Xilinx Virtex-II Pro FPGA only support carry-lookahead addition using dedicated *AND* gate, having this split LUT feature, we can now support more effectively, but are not limited to, the carry-lookahead subtraction, and the *AND-with-one-inverted-input* gate.

Up to this point, we have presented our C-level software implementation in the last chapter, and the long-integer adders and comparator for MMM hardware unit on both current Xilinx and *CryptoRA* FPGAs from this chapter. To complete the Field-Programmable

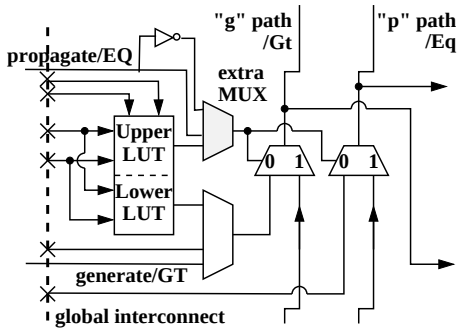


Figure 6.12: Split LUT

Custom Computing Machine (FCCM) solution, we need to connect the FPGA-based MMM functional unit to the core processor- MicroBlaze, which is tailored to our requirement. This is covered in the next chapter.

Chapter 7

Reconfigurable solution simulation and estimation results

This chapter describes the MicroBlaze-based experimental platform and the data-transfer mechansim for completing the hardware-software co-design for public-key cryptography task, in particular, Elliptic Curve Cryptography (ECC). Additionally, this chapter presents both the simulation and estimated performance figures for all the implementations covered in this thesis. Specifically, it includes the performance figures from the pure-software solution and the reconfigurable solution for the EC point multiplication, and pure-hardware solution for modular multiplication using Montgomery Modular Multiplier. Moreover, since the *CryptoRA*'s architecture at this point is not yet a mature design, the performance figures of *CryptoRA* are only estimates rather than back annotations, originating from a silicon-level implementation. The estimated performance figures of *CryptoRA* are then compared against with the simulated performance figures of a Virtex-II FPGA.

7.1 Hardware platform configuration

Prior to running C-level program on the XC2VP100 Virtex-II Pro FPGA package FF1704 [62] sitting on Amirix AP1000 FPGA Development Board [7], the RISC embedded processor must firstly be defined and instantiated onto the FPGA. Inside this particular FPGA device, there are dual hard processors (PowerPC 405) and a soft procesor (MicroBlaze) [59]. A hard embedded processor refers to a processor built from dedicated silicon. On the other hand, 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 a netlist, and therefore, must be synthesized and fit into the FPGA device. In both the soft and hard processor system, the following Intellectual Property (IP) requrie customization and to be

defined to build a complete hardware platform, which includes local memory, processor busses, internal peripherals, peripheral controllers, and memory controllers.

In particular, we choose MicroBlaze as the core processor in the system. The reasoning behind our choice is that the Fast Simplex Link (FSL) interface used for the data communication between the core processor and the accelerated hardware unit can only be utilized in conjunction with a MicroBlaze processor [59]. To define a complete hardware platform on which the C-level program can be run, the *Base System Builder* (BSB) wizard is used in the Xilinx Platform Studio (XPS) developing environment [60]. XPS is an embedded-system developing tool package provided by Xilinx used to create a hardware platform on which a user program can be run. The Intellectual Properties (IPs) included in this embedded processor design are listed below:

- MicroBlaze
  - 100 MHz processor-bus clock frequency
  - 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 this embedded processor system is depicted in Figure 7.1. There are two main bus systems supported - Local Memory Block (LMB) Bus and On-chip Peripheral Bus (OPB). LMB Bus is used specifically to transfer data with single-cycle access in and out the Block RAM (BRAM), in which the program instruction and data are stored separately. On the other hand, OPB Bus interface provides a connection to both on-chip and off-chip peripherals and memory. As noticed from Figure 7.1, the system

is kept minimal in order to achieve the best performance result. 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 BRAM. In addition, since the entire program is stored in the local memory, it has the same read/write performance as cache. Therefore, the cache is disabled. Cache should only be enabled when large portions of the program do not fit in the local memory, and needs to reside in the external memory (e.g., SDRAM), which has a large access latency and is being connected to the OPB.

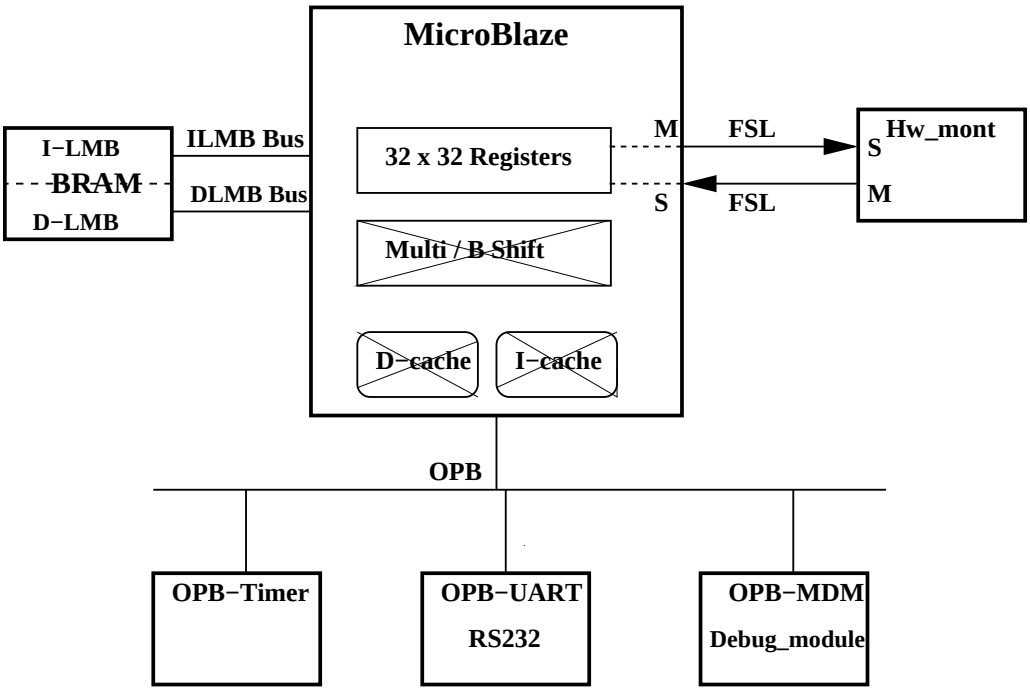


Figure 7.1: Block diagram of the embedded processor system.

Once the soft MicroBlaze embedded processor is completely defined, synthesized, and instantiated onto the Virtex-II Pro FPGA, the executable program is downloaded to and then run on the development board. The output of the program is sent to a terminal application, i.e., HyperTerminal. In the next section, the Fast Simplex Link (FSL) interface for the data communication between the MicroBlaze and the MMM unit is described.

7.2 Interface between MicroBlaze and MMM unit

Instead of computing the Montgomery Modular Multiplication in software, the MMM hardware unit is called to perform the computation. In order to call the MMM hardware unit from within our C-level software program, the Xilinx Fast Simplex Link (FSL) is used for the co-processor interface [58]. FSL is a very fast dedicated connection that Xilinx provides for data transferring 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). Each link is based on the first in, first out (FIFO) mechanism, which can be seen in Figure 7.2. Also, data are transferred in and out by the *get* and *put* assembly instructions.

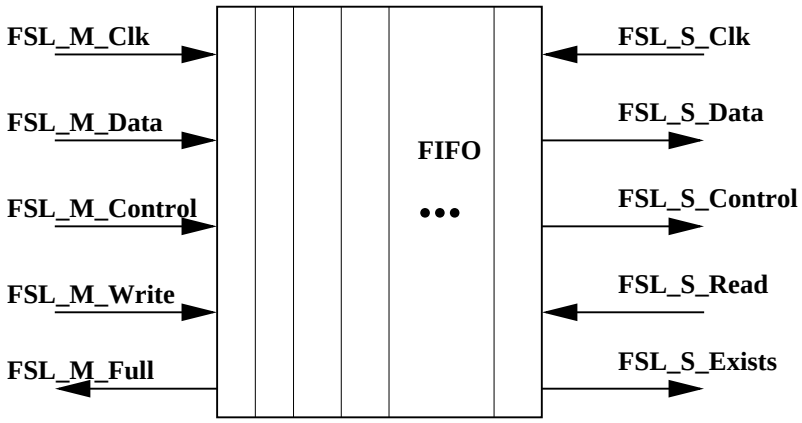


Figure 7.2: Fast Simplex Link (FSL) Bus [61]

To further illustrate this communication between MicroBlaze embedded processor and the MMM hardware unit on Xilinx Virtex-II Pro FPGA, the pseudo code for the FSL calls is provided in Algorithm 7.1. Data is written to the FSL using *putfsl* call in C (Lines 2, 3, and 4), which then calls *put* instruction in assembly. A *putfsl* function reads in the data and the *id* variable, which is assigned at the time that the FSL is instantiated. This *id* variable is used to distinguish between different hardware units instantiated on the FPGA. Setting the controlling signals shown in Figure 7.2 are vital for having the hardware unit

working correctly and efficiently with MicroBlaze processor. Some of these signals are set internally and others are set externally. For instance, the *FSL_S_Exists* is set automatically by the FSL itself when it detects data in the link. This signal is then used in the hardware unit to set *FSL_S_Read* signal, which is to tell the hardware unit if it should start reading data from the link. Furthermore, data can be read and written simultaneously from and to the FSL by both MicroBlaze and the MMM unit. Similarly, once the computed result is ready and is written onto a separate FSL, the *FSL_M_Write* signal must be set manually at the hardware side in HDL. This is when MicroBlaze knows and uses *getfsl* in C, which calls *get* in assembly to get data from the FSL. It is worth mentioning that MicroBlaze is actually idle while the MMM unit is doing computations.

Algorithm 7.1 Pseudo code for FSL calls

Require: Integers $x, y \in [0, p-1]$, and a modulus p

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

```
for i from (l − 2) downto 0 do
    putfsl( x(i), id)
    putfsl( y(i), id)
    putfsl( p(i), id)
end for
for i from 0 to (l − 1) do
    getfsl( T(i), id)
end for
return T
```

Even though FSL is capable of transferring 32-bit data unidirectional point-to-point at high speed, we deal with integers ranging from 160-bit to 2048-bit, which can be regarded as 5 to 64 blocks of 32-bit data, respectively. In our implementation, we transfer 3 long-integer variables per modular multiplication operation to the hardware for the computation,

and transfer the long-integer resulting data back to the core processor at end of each modular multiplication operation. As a result, this creates large overheads on the communication between the MicroBlaze and the MMM functional unit. One of the many techniques that we apply to mitigate the time it spends on the communication is loop-unrolling. This eliminates the additional cycle counts created by the *FOR* loop (e.g., branch, comparison, and addition instructions).

7.3 Developing tools

The C-level program is first compiled and run in the Microsoft Visual Studio 2005 Team Suite Edition with profiling enabled. Then it is ported to the MicroBlaze processor after the hardware platform is configured using the Xilinx Platform Studio (XPS) SDK version 9.1 [60]. The output of the program is sent to Window's HyperTerminal.

The MMM hardware unit is first coded in VHDL using the Xilinx ISE (Project Navigator) v9.1.03i [64]. Each module of the hardware unit is functionally verified through both behavior and post-place-and-route timing simulations using Mentor Graphics ModelSim XE [35]. To augment the software program with the MMM hardware unit, a custom IP wizard in the XPS is used. As mentioned, the mechanism for the data transfer between the MicroBlaze and the hardware unit is the proprietary connection called, Fast Simplex Link (FSL) [61]. The cycle count for various implementations on Virtex-II Pro FPGA are measured using the on-board OPB Hardware Timer [64].

7.4 Simulation and estimation of performance figures

In this section, we show both the simulated and estimated performance results. It begins with the pure-software simulation results from both the MIRACL library and our C-level program running on Pentiumn. It is followed by the simulation results for our C-level program running on MicroBlaze embedded system. The simulation results and its corre-

sponding speedups for the Montgomery Modular Multiplier on FPGA-augmented MicroBlaze embedded processor is next presented. Moreover, the critical path for the Carry-Save Adder (CSA) in the Virtex-II-based MMM is presented. Also, the simulated performance results on Xilinx FPGA and estimated performance results on *CryptoRA* for various adders and the comparator unit are shown.

7.4.1 Software experimental figures on Pentium

To show how our C-level program performs in comparison with the MIRACL, an assembly-optimized library, the profiling figures expressed in cycle count for a key length of 160-, 192-, and 224-bit EC point multiplication is presented in Figure 7.3.

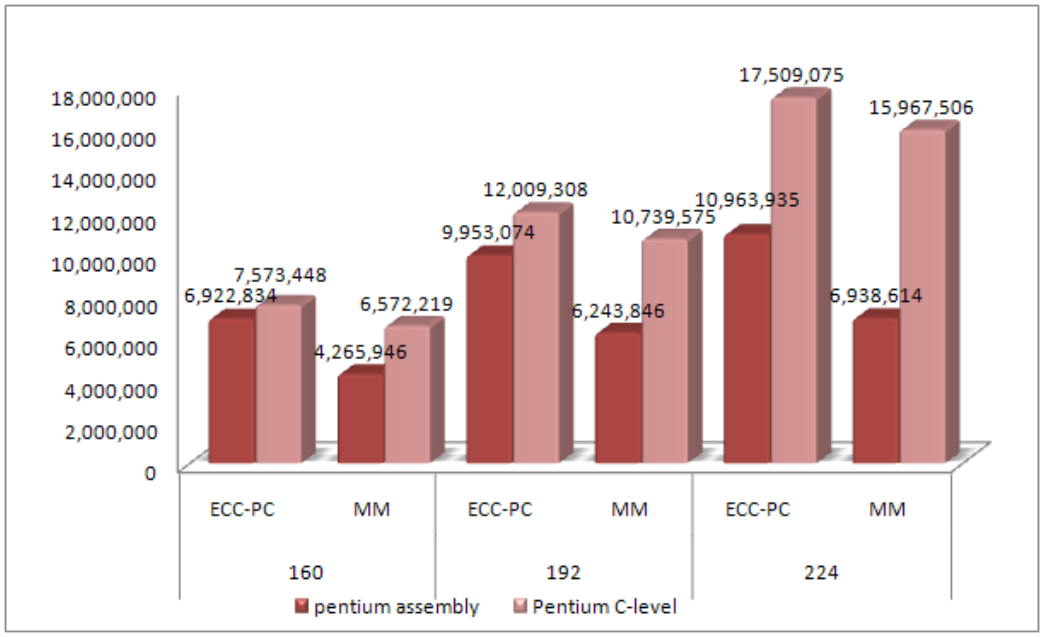


Figure 7.3: Cycle count for MIRACL and C-level program on Pentium.

The setup is a 64-bit Pentium processor running at 1.8 GHz. The performance figures from MIRACL is presented on the left side of the histogram pair, and that from C-level program is on the right side of the pair. Figure 7.3 illustrates the effectiveness of our pro-

gram against with MIRACL library - it shows the speedup for one EC point multiplication performed using MIRACL is within 2× in comparison with our program. These performance figures are acceptable because our pure-software solution is programmed in C while MIRACL is optimized in assembly. Equally important, it also shows the consistancy of the cycle count in modular multiplication operations in respect to that of one EC point multiplication. For instance, in MIRACL, over 60% of the computing time spent in one EC point mulitplication is in modular multiplication operations and over 90% in our C-level software. Despite of the difference, these percentages still show a strong relation between the modular multiplication operations and the overall performance in public-key cryptography.

7.4.2 Software profiling figures on MicroBlaze

In this subsequent section, we analyze the performance for our C-level software program on the MicroBlaze embedded system. Figure 7.4 shows the pure-software performance in cycle count.

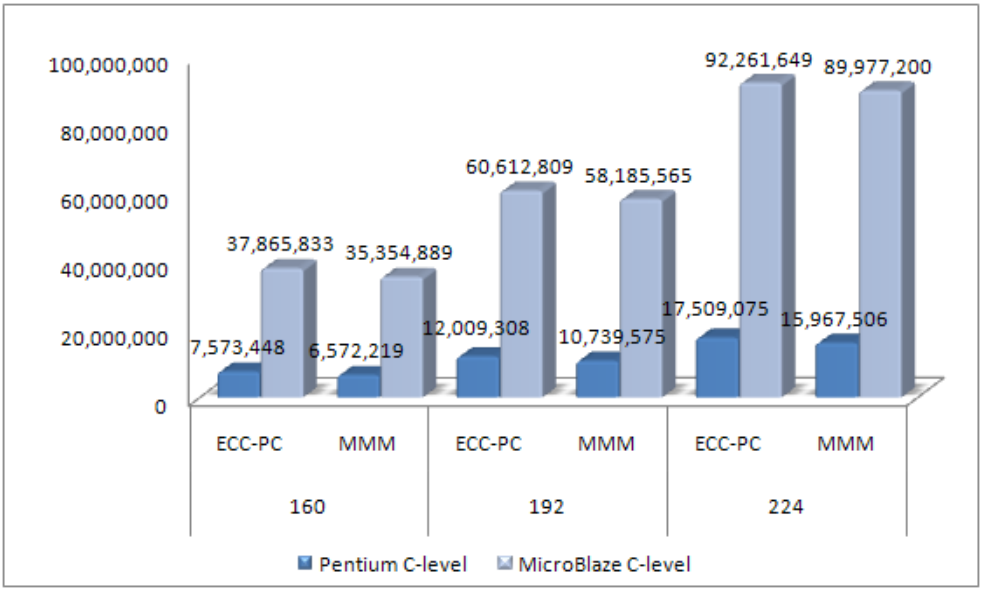


Figure 7.4: Cycle count for C-level program on Pentium and MicroBlaze processors.

As seen, the cycle count required in MicroBlaze embedded system increases by approximately 4× to 6× in comparison with that in Pentium system. This result is expected and is mainly due to the combination of the differences in the processor architecture and compiler. Such increase in cycle count required in the embedded system will be even more obvious if the difference in the system frequency of the Pentium and MicroBlaze is taken into account. Moreover, because of this increase in the cycle count, the relative computing time for modular multiplication is increased to approximately 95% to 97%, which tells us that it is more crucial to have high-performance modular multiplier in order to reduce the overall EC point multiplication computing time in the embedded system environment. Thus, this is the main reason why the modular multiplication is given the reconfigurable hardware support with Montgomery Modular Multiplier unit. One thing worth mentioning is that the cycle count in the Pentium platform is obtained by the profiling feature in Microsoft Visual Studio while the ones obtained in MicroBlaze are reported by the OPB-Hardware Timer. This is because a software bug existing in the XPS developing tool, which does not allow the developer to utilize the profiling feature.

7.4.3 Hardware Montgomery Modular Mutiplier profiling figures on MicroBlaze

After successfully connecting the Virtex-II based MMM to the 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 7.1 and is compared with the pure-software solution. According to this table, the speedup obtained for Montgomery Modular Multiplication operations in one EC point multiplication for various key length of 160, 192, and 224 bits are 37×, 44×, and 45×, respectively. This translates to a speedup of 11× to 22× in one EC point multiplication since the cycle count does not change for other operations in the EC point multiplication.

Although deploying hardware support for MMM results in substantial speedup, this MMM hardware unit does not provide optimal performance for FPGA implementation,

Table 7.1: Cycle count for MMM operations in one EC point 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

mainly because: (i) the computing tiles are idle while waiting for the $x(i)$ and q_i signals to propagate throughout the array in the Carry-Save Adder (CSA) implementation. (ii) Ripple-Carry Adder (RCA) becomes the bottleneck for the system frequency as the key length increases. (iii) 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 aforementioned facts are the issues in the Virtex-II based MMM hardware unit. In addition, the experimental and estimated results of the solutions to these issues are then presented.

7.4.4 Two 3-to-2 Carry-Save Adder simulation results on Xilinx FPGA

To map the two-level Carry-Save Adder (CSA) design shown in Figure 6.1 for the MMM unit on FPGA, two 4-to-1 LUTs for carry and sum bits in the 1st level, and similarly, two 4-to-1 LUTs in the 2nd level are required. This approximates the critical path for the CSA, which is two LUTs, connected through global interconnects and the result is again fed back through global interconnects. This theoretical delay is generalized and can be expressed as the following:

$$\text{delay}_{\text{TOL}} = \text{delay}_{\text{LUT}} + \text{delay}_{\text{INTER}} + \text{delay}_{\text{LUT}} + \text{delay}_{\text{INTER}}, \tag{7.1}$$

where $delay_{\text{LUT}}$ is a constant, which is 0.313ns [62], and $delay_{\text{INTER}}$ is the global interconnect delay, which is ranging from 0.8 to 7ns, depending how close the blocks are routed.

Thus, the theoretical critical path for this CSA combination logic is determined by

the delay of two 4-to-1 LUTs and two times of the delay from the global interconnects. Figure 7.5 reveals the critical path in *ns* measured from the timing waveform for the 2-level CSA of 160 to 256-bit length.

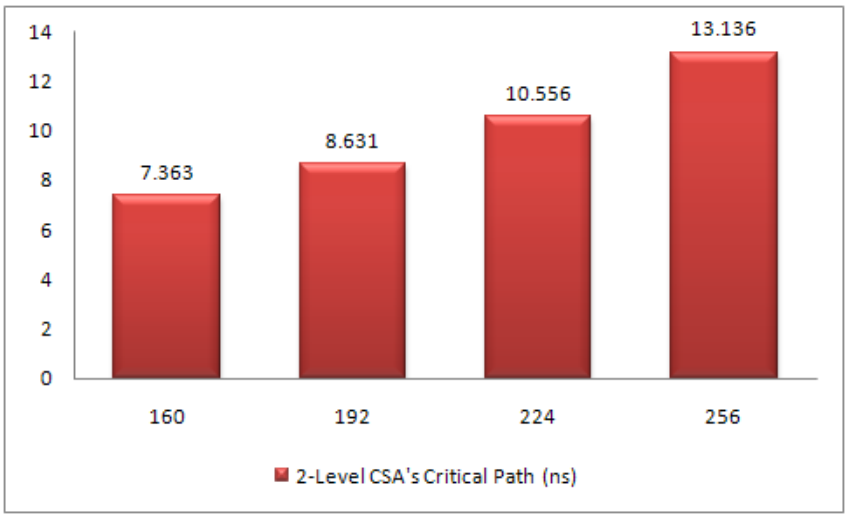


Figure 7.5: Critical path for 2-level CSA in various bit length.

It should be noted that the critical path increases with the bit length. This is because fanout of the $x(i)$ and m_i signals in Figure 6.1 is proportional to the bit length. It takes longer to charge more input capacitance as fanout increases. As a result, the critical path of the combinational logic increases. It should be noted that these timing waveform measurements do not match the figures reported from the synthesis. For instance, the critical path of 192-bit 2-level CSA is reported to be 2.596ns by the synthesis while it is measured to be 8.631ns in the waveform. This mismatch is due to that the net delays created by the large fanouts are much longer on the FPGA fabric after the place and route than the net delay given in the synthesis report. As discussed, we would rather 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 slice usages. CLA can also be used to replace the RCA for the subtraction

in the MMM algorithm (Algorithm 5.2).

7.4.5 Various adders' experimental and estimated figures on Xilinx and *CryptoRA*

To address the issues created by the Virtex-II based MMM unit, Table 7.2 is used to show the critical path for long-integer addition in *ns* for bit-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, single-stage Carry-Lookahead Adder (CLA) and Ripple-Carry Adder (RCA), and are implemented using the dedicated carry path supported in the Xilinx FPGA chip. The same table also shows the estimated critical path of CLA on *CryptoRA*.

Table 7.2: Critical path in *ns* for CSkA, CLA and RCA

key length	CSkA on Virtex-II	CLA on Virtex-II	CLA on *CryptoRA*	RCA on Virtex-II
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 still has an issue for computing long-integer addition - the critical path increases exponentially with the increase in bit length. As a result, the system frequency for the MMM unit decreases drastically as the bit length increases due to the use of the RCA in the Virtex-II based MMM unit. On the other hand, the growth in the critical path of the fast-adder structures are much slower. Thus, considering fast-adder structure is a good approach for keeping the system frequency high.

Even though the CLA's critical path is significantly reduced in comparison with RCA because of the CLA's parallelism property, the delays from LUTs, dedicated carry MUXes,

and global interconnects still exist and can be generalized as the following.

$$\begin{aligned} \text{delay}_{\text{TOL}} = & [\text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize})] \\ & + \text{delay}_{\text{INTER}} + [\text{delay}_{\text{LUT}} + \text{delay}_{\text{MUX}} * \textit{numBlk})] \\ & + \text{delay}_{\text{INTER}} + [\text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize})], \end{aligned} \tag{7.2}$$

where $\text{delay}_{\text{MUX}}$ is a constant and is $0.0313ns$ [62]. While the *blkSize* variable represents the size of a block, the *numBlk* variable represents the number of blocks, which depends on the bit length and *blkSize* variable. The first square bracket represents 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 6 is applied to prevent our dedicated carry path from being cut. Our proposal on extending the dedicated path over the horizontal direction is expected to help decreasing the delays 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, as opposed to the global interconnects. Moreover, the delay from the LUTs can also be eliminated - 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 6 is also solved by the dedicated carry path over the horizontal direction. Both the simulated and estimated critical paths in single-stage CLA for 192-bit to 1024-bit bit length are shown in Table 7.2. As explained, the global interconnects delay and 2 out of 3 LUT delay can be eliminated on *CryptoRA* FPGA. Hence, the estimated total delay for single-stage CLA is modified and generalized in Equation 7.3.

$$\begin{aligned} \text{delay}_{\text{TOL}} = & [\text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * \textit{blkSize})] \\ & + (\text{delay}_{\text{MUX}} * \textit{numBlk}) + (\text{delay}_{\text{MUX}} * \textit{blkSize}) \end{aligned} \tag{7.3}$$

To further demonstrate on how these numerical figures are estimated, we use a key length of 192-bit as an example. In this example, we set *blksize* to be 16 and *numBlk* is then $12 \equiv (192/16)$. This would yield the total estimated critical path of 2.07ns, which is about $4\times$ shorter in critical path than the CLA implemented on the Xilinx Virtex-II Pro FPGA.

However, according to Table 7.3, the tradeoff for a shorter critical path in CLA than in RCA is the higher slice usage. This issue can be alleviated by another proposal of ours - increasing in the FPGA granularity. This allows the block-level *generate* and *propagate* signals to be computed using the same LUT, resulting estimated 30% reduction on the slice usages for single-stage CLA implementation on *CryptoRA* when compared against the Virtex-II-based CLA.

Table 7.3: Slice usgae in CSkA, CLA and RCA

key length	CSkA on Virtex-II	CLA on Virtex-II	CLA on *CryptoRA*	RCA on Virtex-II
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

7.4.6 Comparator unit experimental and estimated figures on Xilinx and *CryptoRA*

Two versions of the comparator unit are described in Chapter 6. The orignial version is implemented using the tree-based structure and its performance is determined by the measurement of the timing waveform. The performance of the proposed version, which is assumed to be emulated using the dedicated carry chain on *CryptoRA*, is estimated. The

resulting critical path, and the slice usage for both versions are presented and compared in Table 7.4.

Table 7.4: Critical path in *ns* and slice usage 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

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 7.4.

$$\text{delay}_{\text{TOL}} = \text{delay}_{\text{LUT}} + [\text{delay}_{\text{INTER}} + log(n) * (\text{delay}_{\text{LUT}} + \text{delay}_{\text{INTER}})] \tag{7.4}$$

The first $delay_{\text{LUT}}$ is created by the generation of the EQ and GT arrays. The rest are the delays created from the $log(n)$ levels in the comparator unit. 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 the growth in level, not in key length. As seen, the delay is substantially reduced in *CryptoRA* and 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 an inverter on the dedicated path, which in simulation it has a delay of $0.008ns$ in 90 nm technology. This new delay for comparator unit on *CryptoRA* can be generalized in Equation 7.5.

$$\begin{aligned} \text{delay}_{\text{TOL}} = {} & \text{delay}_{\text{LUT}} + (\text{delay}_{\text{MUX}} * blkSize) \\ & + \text{delay}_{\text{INVERTER}} + (\text{delay}_{\text{MUX}} * numBlk) \end{aligned} \tag{7.5}$$

Equation 7.5 is used to estimate the critical path for the proposed serial comparator unit mapped on *CryptoRA*. In the example of 192-bit comparator unit, the estimated critical path is calculated as follows. The same value for *blkSize* and *numBlk* as in the single-stage CLA example are used, which are 16 and 12, respectively. With other variables' values being constant, the estimated critical path is calculated to be 1.51ns, which is about 5$\times$ shorter than the critical path of the Virtex-II based comparator unit. As noticed, the majority of the delay is now contributed to 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 slice usage for the comparator unit because it utilizes 100% both dedicated carry paths in all its computation. As seen in Table 7.4, a significant reduction on the slice usage 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 7.4 may not be optimal, rather illustrative.

Chapter 8
Conclusions

We have addressed the issues associated with public-key cryptography algorithms, and have presented our reconfigurable solution towards those computationally demanding algorithms with a Field-Programmable Custom Computing Machine (FCCM), which consists of a softcore MicroBlaze embedded processor augmented with a Xilinx Virtex-II based Montgomery Modular Multiplication (MMM) functional unit. We have also presented the new FPGA architecture, which is based on Xilinx Virtex-II Pro FPGA's dedicated carry chain, called Cryptography-oriented Reconfigurable Array (CryptoRA). This would then be used to agument with MicroBlaze to further reduce the critical path and slice usage, potentially resulting in faster computing time due to the possibility of achieving higher system frequency.

This chapter summarizes our overall investigations and achievements. It is organized in three sections. Section 8.1 discusses the overall conclusions. Section 8.2 presents the major contributions. Section 8.3 proposes future research directions.

8.1 Summary

In this thesis, we have considered and solved a number of issues associated with reconfigurable computing technology. Our overall achievements can be summarized by the following:

In Chapter 2, we first describe the basis of publc-key cryptography algorithms and introduce their common underlying finite field arithmetic, including modular addition/subtraction, multiplication/squaring, exponentiation, and inversion operations. We also state that modular exponentiation and EC point multiplication are the most time-consuming operations

in Rivest-Shamir-Adleman (RSA) and in Elliptic Curve Cryptography (ECC) algorithms, respectively. This is due to the fact that we deal with long-word integers, ranging from 160- to 2048-bits, and modular operations, which in general are computationally demanding.

In Chapter 3, we evaluate different FPGAs' architectures, including Xilinx's Virtex-II Pro and Altera's Stratix. Specifically, we are interested in their architectural support for addition/fast-adder addition. Thus, we look at their dedicated carry chains in detail to gain a better understanding of how they work. Beside the support for Ripple-Carry Adder (RCA), we discover that it also has architectural support for Carry-Lookahead Adder (CLA) in Xilinx and Carry-Select Adder (CSeA) in Altera.

In Chapter 4, we have given a brief review on different computing paradigms, including general purpose processor (GPP), Application-Specific Integrated Circuits (ASIC), and Field-Programmable Custom Computing Machine (FCCM). We also provide an overview on the proposed research papers in the literature with respect to different Montgomery Modular Multiplication implementations. In particular, Goodman's domain-specific reconfigurable cryptographic processor (DSRCP) [19] is specially to our interest, and therefore, we reference their ASIC-based Montgomery Modular Multiplication (MMM) functional unit for developing our Virtex-II based MMM functional unit.

In Chapter 5, we have presented the C-level software implementation for EC point multiplication over prime fields, which is capable of computing various bit-length, ranging from 160- to 521-bit. The RSA algorithm is not explicitly implemented because its required modular exponentiation is indirectly utilized in our ECC implementation because the modular inversion is computed using modular exponentiation. Here, the hardware-software partitioning is determined through the program profiling indicating modular multiplication is in fact the most expensive operation in EC point multiplication. Furthermore, the MIRACL, an assembly-optimized library, is used to verify the computing results and the quality of our C-level software implementation.

In Chapter 6, we have shown the Virtex-II based MMM functional unit whose components are comprised of a Carry-Save Adder (CSA), Ripple-Carry Adder (RCA), and a comparator unit. As it was discovered that this implementation results in long critical path and high slice usage, a Cryptography-oriented Reconfigurable Array, called *CryptoRA*, based on dedicated carry chain in Xilinx Virtex-II Pro, is therefore proposed. The main three features it includes are an increased granularity of the logic tile, the extension of the dedicated carry chain over the horizontal direction, and the split Look-Up Table. Due to *CryptoRA*'s features, Carry-Lookahead Adder (CLA) and the new comparator structures can be emulated more efficiently on *CryptoRA*. Thus, the benefits from the FCCM using *CryptoRA* would be reductions on critial path and slice usage, resulting in shorter computing time and less power consumption.

In Chapter 7, we have presented the cutomized Intellectual Property of the hardware platform used in our implementation, including the softcore MicroBlaze embedded processor and all the surrounding peripherals components. This hardware platform is kept minimal with respect to the number of hardware components in order to achieve the best performance. Fast Simplex Link interface is among the important IP required because it makes the instantiation of the data transferring link easy-to-use. We also presented all the findings in regards to the performance speedup and/or slice usage using different solutions on different platforms. As we have proved that our C-level software is a well-written program and thus, is able to provide a fair comparison for the reconfigurable solution. As far as the performance on modular multiplier is concerned, the Virtex-II based MMM unit has speedup of $37\times$ to $45\times$ over the word-wise MMM pure-software implementation. As a result, the FCCM using Xilinx FPGA-based unit has speedup of $11\times$ to $22\times$ over the pure-software solution per 160- to 224-bit EC point multiplication. As this verified that the critical delay increases drastically with the increase in bit-length, CLA is assumed to be used to replace the CSA and RCA in MMM unit on *CryptoRA* FPGA. This way, the critical delay is estimated to have only a small increase in comparison with the same or

other adder structures implemented in Xilinx Virtex-II Pro FPGA. Hence, the MMM unit on *CryptoRA* FPGA is expected to run at a higher system frequnecy, and as a result, reduces the computing time and slice usage.

8.2 Contributions

The major contributions of this research can be summarized as follows.

- We have addressed two main issues associated with public-key cryptography algorithms, which are the long-interger arithmetic and expensive operations.
- We have shown that it is possible to achieve a software-like flexibility and ASIC-like computing power with the reconfigurable solution using Field-Programmable Custom Computing Machine (FCCM) for public-key cryptography task.
- Hardware-software partitioning can be determined by locating the most time-consuming operation/function in the program through data profiling.
- Having hardware support for Montgomery Modular Multiplication (MMM) algorithm is beneficial to all public-key cryptography algorithms as they all have common underlying finite field arithmetic, expect for the ECC over binary field.
- Description of the architecture of *CryptoRA*, with major features comprising of (i) an increased granularity of the logic tile, (ii) the extention of the dedicated carry chain of standard FPGAs over the orthogonal direction, and (iii) the split Look-Up Table (LUT).
- A reconfigurable solution is a feasible solution to applications requiring long-word integer operations, as proved by the experimental platform comprising MicroBlaze and Xilinx Virtex-II Pro FPGA.

- Further improvements in terms of the critical path and slice usage are possible by augmenting MicroBlaze with our new proposed Cryptography-oriented Reconfigurable Array (*CryptoRA*).

8.3 Proposed research directions

As a continuation of the research, we suggest the following:

- As pointed out, the time spent on data communication become apparent. To reduce such overheads, one approach is to move the entire EC point multiplication for ECC and modular exponentiation for RSA to hardware through microcoded implementation. This way, not only the number of data transfers is reduced, also other operations are supported in hardware, which will result further improvement in performance.

- We assume that dedicated carry chain over the horizontal direction are made possible for any hierarchical carry-lookahead adder signals without penalty. In reality, these signals that ride on horizontal dedicated carry chain can not be perfectly allied with each other, resulting idle computing tiles and long propagation delay on the dedicated carry chain. One possible solution is to apply a snail structure, which is not maturely thought through at present.

- Since this reconfigurable solution implemented on the Xilinx Virtex-II Pro shows promising results in comparison with a pure-software solution, our *CryptoRA* is therefore proposed to provide further improvements. To make *CryptoRA* possible in use, a package and synthesis tools should be developed at the system-level to provide to system designers who would like to utilize *CryptoRA* FPGA in their applications.

Bibliography

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

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

[3] Kendall Ananyi and Daler Rakhmatov. Design of a Reconfigurable Processor for NIST Prime Field ECC. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 333–334, Washington, DC, USA, 2006. IEEE Computer Society.

[4] 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.

[5] Lejla Batina and Geeke Muurling. Montgomery in Practice: How to Do It More Efficiently in Hardware. In Bart Preneel, editor, *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology (CT-RSA 2002)*, volume 2271 of *Lecture Notes in Computer Science (LNCS)*, pages 40–52, San Jose, California, 2002. Springer-Verlag.

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

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

[8] D V Chudnovsky and G V Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Adv. Appl. Math.*, 7(4):385–434, 1986.

[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] Francis Crowe, Alan Daly, and William Marnane. A scalable dual mode arithmetic unit for public key cryptosystems. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, pages 568–573, Washington, DC, USA, 2005. IEEE Computer Society.

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

[12] Andre DeHon. Balancing Interconnect and Computation in Reconfigurable Computing Array. In *FPGA*, pages 69–78, 1999.

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

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

[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] C. Paar A. J. Elbirt. Towards an FPGA architecture optimized for public-key algorithms. In *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pages 33–42, Bellingham, WA, 1999. SPIE – The International Society for Optical Engineering.

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

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

[19] 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.

[20] 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.

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

[22] S. Hauck. The roles of FPGAs in Reprogrammable Systems, 1998.

[23] 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.

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

[25] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

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

[27] Brian W. Kernighan, Dennis Ritchie, and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988.

[28] Donald Ervin Knuth. *The Art of Computer Programming, 3rd Ed.*, volume 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

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

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

[31] 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.

[32] 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.

[33] C. McIvor, M. McLoone, 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.

[34] 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.

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

[36] 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.

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

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

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

[40] Gerardo Orlando and Christof Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 348–363, London, UK, 2001. Springer-Verlag.

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

[42] Siddika Berna Ors, Lejla Batina, Bart Preneel, and Joos Vandewalle. Hardware implementation of a montgomery modular multiplier in a systolic array. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 184.2, Washington, DC, USA, 2003. IEEE Computer Society.

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

[44] 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.

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

[46] Erkay Savaş, Alexandre F. Tenca, M. E. ¸Cift¸ibasi, and ¸Cetin K. Koc. Novel multiplier architectures for $GF(p)$ and $GF(2^n)$. In *IEE Proceedings – Computers and Digital Techniques*, volume 151, pages 147–160, March 2004.

[47] 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.

[48] Akashi Satoh and Kohji Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Trans. Comput.*, 52(4):449–460, 2003.

[49] Erkay Savas, Alexandre F. Tenca, and Çetin Kaya Koç. A scalable and unified multiplier architecture for finite fields gf(p) and gf(2m). In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 277–292, London, UK, 2000. Springer-Verlag.

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

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

[52] Nigel P. Smart. How secure are elliptic curves over composite extension fields? In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 30–39, London, UK, 2001. Springer-Verlag.

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

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

[55] T. Pham, C., M. Kondapalli, S. Young. Programmable logic block with carry chains providing lookahead functions of different lengths, 2007.

[56] Alexandre F. Tenca and Çetin Kaya Koç. A scalable architecture for montgomery multiplication. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 94–108, London, UK, 1999. Springer-Verlag.

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

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

[59] Xilinx Inc., San Joes, CA. *MicroBlaze Processor Reference Guide*, 2005.

[60] Xilinx Inc., San Joes, CA. *Platform Studio User Guide*, February 2005.

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

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

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

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

摘要

在这篇论文里，我们提出一个reconfigurable计算的翻案用来减少公众钥匙密码术算法计算时间，这个reconfigurable平台主要包括了可编程序领域的门阵列(FPGA)增大的一台MicroBlaze 处理器。我们首先考虑以Xilinx的Virtex-II组成的分析计算器器来分析一个可编程序领域(FCCM)的潜在解决办法。然后，我们提出一个cryptography-oriented reconfigurable array，叫CryptoRA，用来有效地支持长字整数增加，减和比较然而，CryptoRA就能有效的与RISC 处理器合并来形成CryptoRA-based的FCCM。CryptoRA的3 个主要特徵是：(i)逻辑瓦片的增加的颗粒性，(ii)dedicated carry chain的横向的一个沿伸(iii)Look-up table的分裂以增加LUT运算的灵活性。根据我们的估计，相较于最优化的软件解法，CryptoRA-based的FCCM是一个能 能提供显着的效能提高方法。