ARCHITECTURE EXPLORATIONS FOR ELLIPTIC CURVE CRYPTOGRAPHY ON FPGAS

A THESIS

submitted by

CHESTER REBEIRO

for the award of the degree

of

MASTER OF SCIENCE

(by Research)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

FEBRUARY 2009

THESIS CERTIFICATE

This is to certify that the thesis titled **Architecture Explorations for Elliptic Curve Cryptography on FPGAs, IIT Madras**, submitted by **Chester Rebeiro**, to the Indian Institute of Technology Madras, for the award of the degree of **Master of Science**, is a bonafide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Debdeep Mukhopadhyay
Research Guide
Professor
Dept. of CS and Engineering
IIT Madras, 600 036

Date: 14th January 2009

ACKNOWLEDGEMENTS

Foremost, I would like to thank my guide Dr. Debdeep Mukhopadhyay who shared a lot of his experience and ideas with me. I appreciate his professionalism, planning, and constant involvement in my research. I cherish the time we spent in discussions and in the laboratory pouring over problems. Working under him has sharpened my research skills and increased my appetite to work in cryptography.

I am grateful to Dr. Kamakoti and Dr. Shankar Balachandran for their encouragement, advice, and help whenever needed. I am indebted to the RISE lab and the Computer Science Department for offering me a fabulous environment to work and study.

I would like to take this opportunity to acknowledge several friends and lab mates who made my stay at IIT Madras exciting and unforgettable. I acknowledge the help received from Noor on innumerable occasions. I would especially like to thank him for helping me out with various tool flows. Shoaib, for the discussions that we had on technical as well as non technical topics. Rajesh, for being so easy to connect to, and Venkat among all things for letting me know the best Idly joints in Chennai. I thank Pavan, Shyam, Sadgopan, Parthasarthy, and Lalit for working along with me on several courses and assignments.

I am grateful to the Centre for Development of Advanced Computing for giving me this opportunity to further my studies. I would like to acknowledge the help received from my colleagues Hari Babu, Ramana Rao, and Alok Singh who took care of things while I was away.

I would like to thank my wife Sharon and my parents for the love and encouragement I received. Without their support this thesis would not have been possible. I would like to thank my grandmother for her prayers and for being my role model for hardwork. I would like to dedicate this thesis to her.

Chester Rebeiro

ABSTRACT

The current era has seen an explosive growth in communications. Applications like online banking, personal digital assistants, mobile communication, smartcards, etc. have emphasized the need for security in resource constrained environments. Elliptic curve cryptography (ECC) serves as a perfect cryptographic tool because of its short key sizes and security comparable to that of other standard public key algorithms. However, to match the ever increasing requirement for speed in today's applications, hardware acceleration of the cryptographic algorithms is a necessity. As a further challenge, the designs have to be robust against side channel attacks.

This thesis explores efficient hardware architectures for elliptic curve cryptography over binary Galois fields. The efficiency is largely affected by the underlying arithmetic primitives. The thesis therefore explores FPGA designs for two of the most important field primitives namely multiplication and inversion. FPGAs are reconfigurable hardware platforms offering flexibility and lower costs like software programs. However, designing on FPGA platforms is challenging because of the large granularity, limited resources, and large routing delay. The smallest programmable entity in an FPGA is the look up table. The arithmetic algorithms proposed in this thesis maximizes the utilization of LUTs on the FPGA.

A novel finite field multiplier based on the recursive Karatsuba algorithm is proposed. The proposed multiplier combines two variants of Karatsuba, namely the general and the simple Karatsuba multipliers. The general Karatsuba multiplier has a large gate count but for small sized multiplications is compact because it utilizes LUT resources efficiently. For large sized multiplications, the simple Karatsuba is efficient as it requires lesser gates. The proposed hybrid multiplier does the initial recursion using *the simple algorithm while final small sized multiplications is done using the general algorithm. The multiplier thus obtained has the best area time product compared to reported literature.*

The Itoh-Tsujii multiplicative inverse algorithm is based on Fermat's little theorem and requires $m - 1$ *squarings and* $O(log_2(m))$ *multiplications. The proposed inverse algorithm called quad-Itoh Tsujii, is based on the fact that on an FPGA, using quad circuits is more efficient than using squarers due to a better LUT utilization. The quad-Itoh Tsujii requires* $(m-1)/2$ *quad circuits and has the best computation time compared to any inverse algorithm reported.*

The proposed primitives are organized as an elliptic curve crypto processor (ECCP) and has one of the best timings and area time product compared to reported works. We conclude that the performance of an ECCP is significantly enhanced if the underlying primitives are carefully designed. Further, a side channel attack based on simple timing and power analysis is demonstrated on the ECCP. The construction of the ECCP is then modified to mitigate such attacks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AU	Arithmetic Unit
ASIC	Application Specific Integrated Circuit
DPA	Differential Power Analysis
ECC	Elliptic Curve Cryptography
ECCP	Elliptic Curve Crypto Processor
ECDLP	Elliptic Curve Discrete Logarithm Problem
EEA	Extended Euclid’s Algorithm
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GF	Galois Field
ITA	Itoh-Tsujii Algorithm
LD	Lopez-Dahab
LUT	Look Up Table
RSA	Rivest Shamir Adleman
SPA	Simple Power Analysis
SR-ECCP	SPA Resistant Elliptic Curve Crypto Processor
VCD	Value Change Dump

CHAPTER 1

Introduction

This era has seen an astronomical increase in communications over the wired and wireless networks. Everyday thousands of transactions take place over the world wide web. Several of these transactions have critical data which need to be confidential, transactions that need to be validated, and users authenticated. These requirements need a rugged security framework to be in force.

Cryptology is the science concerned with providing secure communications. The goal of cryptology is to construct schemes which allow only authorized access to information. All malicious attempts to access information is prevented. An authorized access is identified by a *cryptographic key*. A user having the right key will be able to access the hidden information, while all other users will not have access to the information. Cryptology consists of *cryptography* and *cryptanalysis*. The former involves the study and application of various techniques through which information may be rendered unintelligible to all but the intended receiver. On the other hand cryptanalysis is the science of breaking crypto systems and recovering the secret information.

There are two types of cryptographic algorithms, these are symmetric key and asymmetric key algorithms. *Symmetric key* cryptographic algorithms have a single key for both encryption and decryption. These are the most widely used schemes. They are preferred for their high speed and simplicity. However they can be used only when the two communicating parties have agreed on the secret key. This could be a hurdle when used in practical cases as it is not always easy for users to exchange keys. In *asymmetric key* cryptographic algorithms two keys are involved. A private key and a public key. The private key is kept secret while the public key is known to everyone.

The encryption is done with the public key, and the encrypted message can be only decrypted by the corresponding private key. Security of these algorithms depends on the hardness of deriving the private key from the public key. Although slow and highly complex, asymmetric key cryptography has immense advantages. The main advantage is that the underlying primitives used are based on well known problems such as integer factorization and discrete logarithm problems. These problems have been studied extensively and their hardness has not been contradicted after years of research. This is unlike symmetric key cryptography where the strength of the algorithm relies on combinatorial techniques. The security of such algorithms is not proven and does not rely on well researched problems in literature. The most used asymmetric key crypto algorithm is RSA [2]. Of late asymmetric crypto algorithms based on elliptic curves have been rapidly gaining popularity due to the higher level of security offered at lower key sizes. Several security standards have emerged which use elliptic curves for the underlying security algorithm.

There are several methods to cryptanalyze modern cryptographic algorithms. Conventional cryptanalysis techniques exploit algorithm weaknesses. They cannot be applied in practice due to the large number of data that is required. In addition most techniques require huge amount of computation time making them very expensive. However, the most serious threat to modern cryptographic algorithms are attacks based on information gathered from *side channels*. These attacks [3, 4] target the implementation rather than the algorithm. Sources of side channel include power consumption of the device, timing, acoustics, and radiation characteristics, thus an attacker monitoring one or more side channels of a device performing an encryption (or decryption) can gather information about the secret key. Optimized cryptographic implementations are more susceptible to side channel attacks, therefore high performing cryptographic hardware must consider this class of attacks during implementation.

1.1 Motivation

Though asymmetric key cryptography is indispensable for communication, there is a penalty on the application's performance. Most of the pubic key cryptographic algorithms have several complex mathematical computations making the penalty dear. It is therefore important to have efficient implementations of the algorithms.

There are two schemes for developing efficient cryptographic implementations. The first focuses on implementing and optimizing the cryptographic algorithms in software platforms. This has the advantage of being low cost as no additional hardware is required. However, benefits obtained by this method are restricted by the architecture limitations of the microprocessor. For example, arithmetic on large numbers cannot be as efficiently done on microprocessors as it can be performed on dedicated hardware. Such arithmetic is a norm in public key cryptographic algorithms. Besides, software can very easily be tampered thus compromising the security of the application.

Even if software implementations are tailored to exploit the processor's architecture [5–8] they are no match to dedicated hardware implementations. The inherent parallelism, flexibility, and custom design of hardware significantly speed up execution. Also, hardware devices can be made more tamper resistant compared to software. This is beneficial for cryptographic applications. However, hardware is more expensive than software and the amount of resources available is limited. Design cycles for hardware is also more involved and complex. Memory is yet another constraint for such designs. It is therefore vital to have compact, scalable and modular hardware designs which are fine tuned to the specific application. *Field programmable gate arrays* (FPGAs) are reconfigurable platforms to build hardware. They offer advantages of hardware platforms as well as software platforms. While on one hand they offer more programmability and lower costs like a software platform, they also offer better performance than a software implementation. However designing for FPGAs is tricky. What works for an *application specific integrated circuit* (ASIC) library does not always work for an FPGA. The main differences occur because of the inherent difference in the libraries and the architectures. FPGAs have fixed resources, a *look up table* (LUT) based architecture, and larger interconnect delays. Hence a design on FPGA must be carefully built to utilize the resources well and satisfy the timing constraints of the FPGA library. In this work we design and implement a *side channel attack* (SCA) resistant elliptic curve processor on an FPGA platform.

1.2 Contribution of the Thesis

In this thesis, architectures for a public key crypto algorithm based on elliptic curves[9–11] are explored. The architectural explorations are targeted for reconfigurable platforms. The contribution of this thesis is as follows.

- The thesis presents an architecture for efficient implementations of finite field multiplication. The proposed multiplier is called *hybrid Karatsuba multiplier* and is based on the Karatsuba-Ofman multiplication algorithm [12]. Detailed analysis has been carried out on how existing multiplication algorithms utilize FPGA resources. Based on the observations, the work develops a hybrid technique which has a better area delay product compared to known algorithms. Results have been practically demonstrated through a large number of experiments.

- The most complex finite field operation in elliptic curve cryptography (ECC) is the multiplicative inverse. The thesis proposes a novel inversion algorithm for FPGA platforms. The proposed algorithm is a generalization of the Itoh-Tsujii inversion algorithm [13]. Evidence has been furnished and supported with experimental results to show that the proposed inversion algorithm outperforms existing results. The proposed method is demonstrated to be scalable with respect to field sizes.

- The work presents the design of a high performance elliptic curve crypto processor (ECCP) for an elliptic curve over the finite field $GF(2^{233})$. The chosen elliptic curve is one of the selected curves for the digital signature standard [14]. The high performance is obtained by efficient implementations of the underlying finite field arithmetic. The processor is synthesized for Xilinx's FPGA [15] platform and is shown to be one of fastest reported implementations on FPGA.

- The thesis demonstrates that a naive implementation of an elliptic curve crypto processor is vulnerable to simple power attacks. The attack is demonstrated using XPower[1]; a power simulation tool from Xilinx. The power traces are shown to leak information about the key and internal activities of the state machine of the processor. A side channel resistant processor is also designed and demonstrated to be resistant to similar attacks.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows.

- Chapter 2 contains a brief introduction of ECC and covers aspects about engineering an elliptic curve processor. A survey is made on existing elliptic curve crypto processors reported in literature. The chapter also contains a brief introduction on FPGA architecture and side channel attacks.

- Chapter 3 contains the mathematical background required to understand ECC. The first part of the chapter outlines the required concepts in abstract algebra. It also presents some of the basic arithmetic circuits such as adders, squarers, and modular operators. The second part of the chapter discusses elliptic curve cryptography.

- Finite field multiplication is discussed in detail in Chapter 4. The Karatsuba multiplier is chosen as the multiplier in the elliptic curve crypto processor. A

[1]http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

hybrid Karatsuba multiplier is proposed for FPGA platforms and shown to have the best area time product compared to existing works.

- Chapter 5 discusses finite field inversion. A generalization of the Itoh-Tsujii inversion algorithm is proposed. A specific form of the generalized Itoh Tsujii algorithm known as the *quad Itoh Tsujii* is shown to be more efficient for FPGA platforms. A processor based on the quad Itoh Tsujii is constructed and shown to be the fastest inversion algorithm reported.

- Chapter 6 integrates the various finite field arithmetic primitives into an elliptic curve crypto processor. The efficient underlying primitives result in one of the fastest reported elliptic curve crypto processors.

- Chapter 7 uses Xilinx tools to demonstrate that the naive implementation of an elliptic curve crypto processor is vulnerable to side channel attacks. The chapter then proposes a modification to the architecture which makes the processor less prone to side channel attacks.

- Chapter 8 has the conclusion of the thesis and future directions of research in this area of work.

- Appendix A has details of how the correctness of the ECCP was verified and the testing of the ECCP on a FPGA hardware platform. Appendix B has a list of the finite fields that were used to test the scalability of the proposed inverse algorithm. Appendix C has instructions to use XPower to obtain the power trace of an FPGA. Appendix D has derivations for the elliptic curve arithmetic equations. Appendix E has derivations for the gate requirements for the simple Karatsuba multiplier.

CHAPTER 2

A Survey

Definition 2.0.1 *A symmetric key cryptosystem can be defined by the tuple* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ *[16], where*

- $\mathcal{P}$ *represents the finite set of possible plaintexts.*
- $\mathcal{C}$ *represents the finite set of possible ciphertexts.*
- $\mathcal{K}$ *represents the finite set of possible keys.*
- *For each* $k \in \mathcal{K}$ *there is an encryption rule* $e_k \in \mathcal{E}$ *and a corresponding decryption rule* $d_k \in \mathcal{D}$*. Each* $e_k : \mathcal{P} \to \mathcal{C}$ *and* $d_k : \mathcal{C} \to \mathcal{P}$ *are functions such that* $d_k(e_k(x)) = x$ *for every plaintext* $x \in \mathcal{P}$*.*

The keys for both encryption and decryption are the same and must be kept secret. This leads to problems related to *key distribution* and *key management*. In 1976, Diffie and Hellman [17] invented asymmetric key cryptography which solved the problem of key distribution and management. Asymmetric algorithms use a pair of keys for encryption

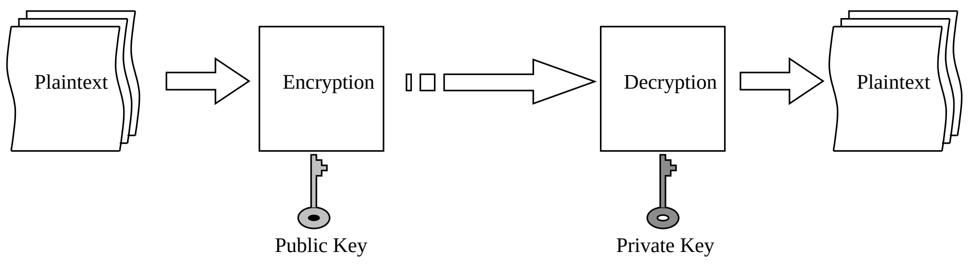

Fig. 2.1: Public Key Cryptosystem

and decryption (Figure 2.1). Encryption is done by a public key which is known to everyone. Decryption can be only done using the corresponding private key. *Given the private key, the corresponding public key can easily be derived. However, the private key cannot be efficiently derived from the public key.* An asymmetric key cryptosystem is constructed by means of *trapdoor one-way functions* which are defined as follows [11].

Definition 2.0.2 *A function $f(x)$ from a set X to a set Y is called a one-way function if $f(x)$ can efficiently be computed but the computation of $f^{-1}(x)$ is computationally intractable.*

Definition 2.0.3 *A trapdoor one-way function is a one-way function $f(x)$ if and only if there exists some supplementary information (usually the secret key) with which it becomes feasible to compute $f^{-1}(x)$.*

Thus, trapdoor one way functions rely on intractable problems in computer science. An example of an intractable problem is the *integer factorization problem*, which states that given an integer n, one has to obtain its prime factorization, i.e find $n = p_1^{e^1} p_2^{e^2} p_3^{e^3} \cdots p_k^{ek}$, where p_i is a prime number and $e^i \geq 1$. Solving the problem of factoring the product of prime numbers is considered computationally difficult for properly selected primes of size at least 1024 bits. This forms the basic security assumption of the famous RSA algorithm [2]. Another intractable problem, the *elliptic curve discrete logarithm problem* (ECDLP) has given rise to new asymmetric cryptosystems based on elliptic curves.

2.1 Elliptic Curve Cryptography

Elliptic curves have been studied for over hundred years and have been used to solve a diverse range of problems. For example, elliptic curves are used in proving Fermat's last theorem, which states that $x^n + y^n = z^n$ has non zero integer solutions for x, y, and z when $n > 2$ [18].

The use of elliptic curves in public key cryptography was first proposed independently by Koblitz [19] and Miller [10] in the 1980s. Since then, there has been an abundance of research on the security of ECC. In the 1990's ECC began to get accepted by several accredited organizations, and several security protocols based on ECC [14, 20, 21] were standardized. The main advantage of ECC over conventional asymmetric crypto systems [2] is the increased security offered with smaller key sizes. For example, a 256 bit key in ECC produces the same level of security as a 3072 bit RSA key[1]. The smaller key sizes leads to compact implementations and increased performance. This makes ECC suited for low power resource constrained devices.

An elliptic curve is the set of solutions (x, y) to Equation 2.1 together with the point at infinity ($\mathcal{O}$). This equation is known as the *Weierstraß* equation [18].

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

For cryptography, the points on the elliptic curve are chosen from a large finite field. The set of points on the elliptic curve form a *group* under the addition rule. The point $\mathcal{O}$ is the identity element of the group. The operations on the elliptic curve, i.e. the group operations are *point addition*, *point doubling* and *point inverse*. Given a point $P = (x, y)$ on the elliptic curve, and a positive integer n, *scalar multiplication* is defined as

$$nP = P + P + P + \cdots P(n\,\mathrm{times}) \tag{2.2}$$

The *order* of the point P is the smallest positive integer n such that $nP = \mathcal{O}$. The points $\{\mathcal{O}, P, 2P, 3P, \cdots (n-1)P\}$ form a group generated by P. The group is denoted as $< P >$.

[1]NIST Sources

The security of ECC is provided by the *elliptic curve discrete logarithm problem* (ECDLP), which is defined as follows : *Given a point P on the elliptic curve and another point $Q \in < P >$, determine an integer k ($0 \leq k \leq n$) such that $Q = kP$.* The difficulty of ECDLP is to calculate the value of the scalar k given the points P and Q. k is called the discrete logarithm of Q to the base P. P is the generator of the elliptic curve and is called the basepoint.

The ECDLP forms the base on which asymmetric key algorithms are built. These algorithms include the elliptic curve Diffie-Hellman key exchange, elliptic curve ElGamal public key encryption, and the elliptic curve digital signature algorithm.

2.2 Engineering an Elliptic Curve Crypto Processor

The implementation of elliptic curve crypto systems constitutes a complex interdisciplinary research field involving mathematics, computer science, and electrical engineering [22]. Elliptic curve crypto systems have a layered hierarchy as shown in Figure 2.2. The bottom layer constituting the arithmetic on the underlying finite field most prominently influences the area and critical delay of the overall implementation. The group

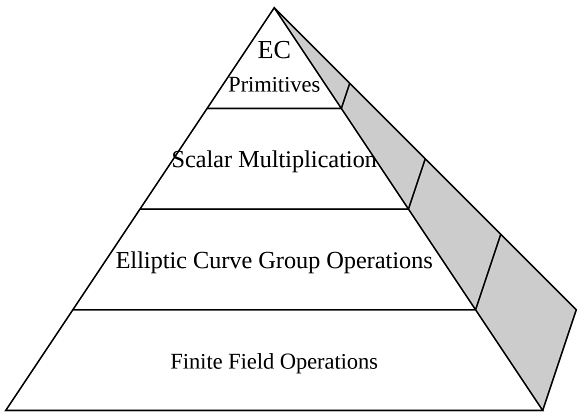

Fig. 2.2: Elliptic Curve Pyramid

operations on the elliptic curve and the scalar multiplication influences the number of clock cycles required for encryption.

To be usable in real world applications, implementations of the crypto system must be efficient, scalable, and reusable. Applications such as smart cards and mobile phones require implementations where the amount of resources used and the power consumed is critical. Such implementations should be compact and designed for low power. Computation speed is a secondary criteria. Also, the degree of reconfigurability of the device can be kept minimum [23]. This is because such devices have a short lifetime and are generally configured only once. On the other side of the spectrum, high performance systems such as network servers, data base systems etc. require high speed implementations of ECC. The crypto algorithm should not be the bottleneck on the application's performance. These implementations must also be highly flexible. Operating parameters such as algorithm constants, etc. should be reconfigurable. Reconfiguration can easily be done in software, however software implementations do not always scale to the performance demanded by the application. Such systems require to use dedicated hardware to speedup computations. When using such hardware accelerators, the clock cycles required, frequency of operation, and area are important design criteria. The clock cycles should be low and frequency high so that the overall latency of the hardware is less. The area is important because smaller area implies more parallelism can be implemented on the same hardware, thus increasing the device's throughput.

2.3 Hardware Accelerators for ECCP

There are two platforms on which hardware accelerators are built: *application specific integrated circuits* (ASICs) and *field programmable gate arrays* (FPGAs). ASICs are one time programmable and are best suited for high volume production. ASICs can reach high frequency of operation, and algorithms implemented on these devices have high performance. Also, ASICs are best when data protection is concerned. Once data is written into an ASIC it is extremely difficult to read back. However, ASICs suffer from high development costs and lack flexibility with respect to modifying algorithms and reconfiguring parameters [24]. Besides, production of an ASIC requires to be done in fabrication units. These fabrication units are generally owned by a third party. This is not suited for cryptographic applications where minimum number of parties must be involved.

FPGAs are reconfigurable devices offering parallelism and flexibility on one hand while being low cost and easy to use on the other. Moreover they have much shorter design cycle times compared to ASICs. FPGAs were initially used as prototyping devices and in high performance scientific applications but the short time-to-market and on-site reconfigurability features have expanded their application space. These devices can now be found in various consumer electronic devices, high performance networking applications, medical electronics and space applications. The reconfigurability aspect of FPGAs also makes them suited for cryptographic applications. Reconfigurability results in flexible implementations allowing operating modes, encryption algorithms, and curve constants etc. to be configured. FPGA's do not require sophisticated equipment for production, they can be programmed in house. This is beneficial for cryptography as no untrusted party is involved in the production cycle.

2.3.1 FPGA Architecture

There are two main parts of the FPGA chip [25] : the input/output (I/O) blocks and the core. The I/O blocks are located around the periphery of the chip and are used to provide programmable connectivity to the chip. The core of the chip consists of programmable logic blocks and programmable routing architectures. A popular architecture for the core, called *island style* architecture, is shown in Figure 2.3. *Logic blocks*, also called *configurable logic blocks* (CLB), consists of logic circuitry for implementing logic. Each CLB is surrounded by routing channels connected through switch blocks

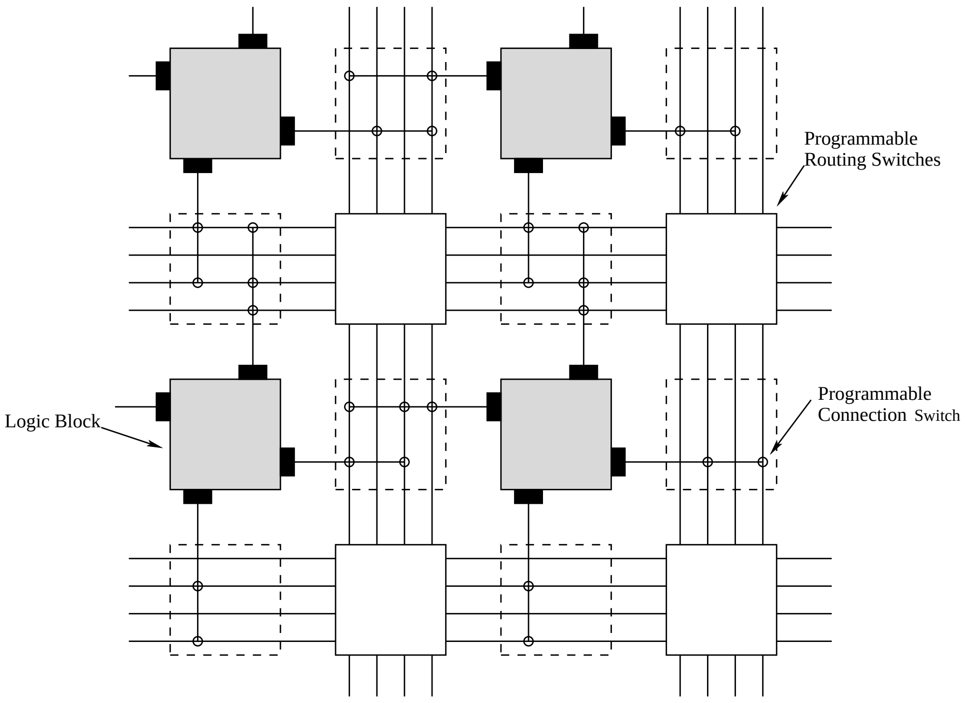

Fig. 2.3: FPGA Island Style Architecture

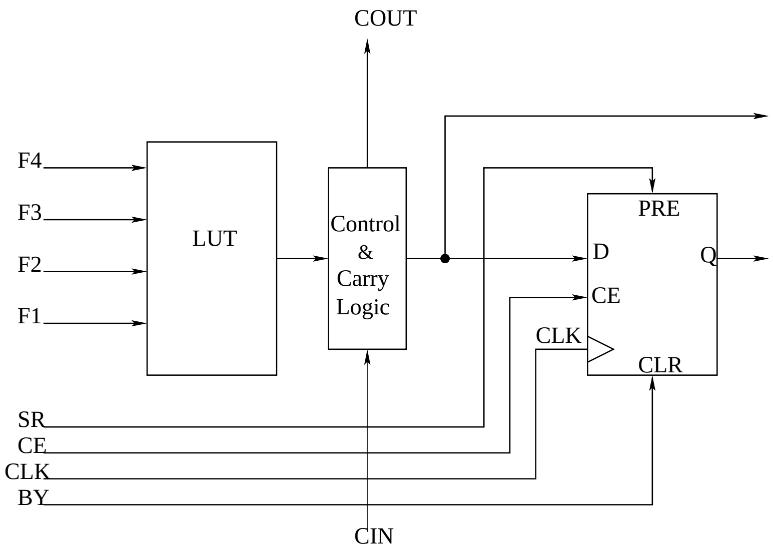

Fig. 2.4: FPGA Logic Block

and connection blocks. A *switch block* connects wires in adjacent channels through programmable switches. A *connection block* connects the wire segments around a logic block to its inputs and outputs. Each logic block further contains a group of *basic logic elements* (BLE). Each BLE has a *look up table* (LUT), a storage element, and combinational logic as shown in Figure 2.4. The storage element can be configured as an edge triggered D-flip flop or as level sensitive latches. The combinational logic generally contains logic for carry and control signal generation.

LUTs can be configured to be used in logic circuitry. If the LUT has m inputs, then any m variable boolean function can be implemented. The LUT mainly contains memory to store truth tables of boolean functions and multiplexers to select the values of memories. There have been several studies on the best configuration for the LUT. A larger LUT would result in more logic fitted into a single LUT and hence lesser critical delay. However, a larger LUT would also indicate larger memory and bigger multiplexers, hence larger area. Most studies show that a 4 input LUT provides the best area-time product, though there have been few applications where a 3 input LUT [26] and 6 input LUT [27] have been found beneficial. Most FPGA manufacturers, including Xilinx[2] and Altera[3], use 4 input LUTs.

2.4 Side Channel Attacks

From the mid 90's, a new research area that has gained focus is *side channel cryptanalysis*. This is becoming the biggest threat to modern day cryptosystems with many of the algorithms successfully attacked. These attacks analyze unintended information leakage that is provided by naive implementations of a crypto algorithm.

Side channel attacks are broadly classified into *passive* and *active* attacks. In a passive attack, the functioning of the cryptographic device is not tampered. The secret key is revealed by observing physical properties of the device, such as timing characteristics, power consumption traces, etc. In an active attack, the inputs and environment are

[2]http://www.xilinx.com
[3]http://www.altera.com

manipulated to force the device to behave abnormally. The secret key is then revealed by exploiting the abnormal behavior of the device [28].

The two most extensively exploited side channels are power consumption and timing analysis. An attack based on timing analysis[3] first identifies and then monitors certain operations in the device. The time required to complete these operations leaks information about the secret key. Power consumption attacks [4] reveal the secret key by monitoring the power consumed by the device. The power consumption of a device has dependencies on the data being manipulated and the operation being performed. There are essentially two forms of power attacks : *simple power analysis* and *differential power analysis*. An attacker using a simple power analysis (SPA) requires just a single power trace. Features of the power trace are used to directly interpret the secret key. A stronger form of power attack called differential power attack (DPA) was first introduced by Kocher in [4]. This is a statistical technique and requires several power traces to be analyzed before the key is revealed. This class of attacks is based on the power consumption dependence of a device, which is dependent on the key.

2.5 Related Work

There have been several reported high performance FPGA processors for elliptic curve cryptography. Various acceleration techniques have been used ranging from efficient implementations to parallel and pipelined architectures. In [29] the Montgomery multiplier [30] is used for scalar multiplication. The finite field multiplication is performed using a digit-serial multiplier proposed in [31]. The Itoh-Tsujii algorithm is used for finite field inversion. A point multiplication over the field $GF(2^{167})$ is performed in 0.21ms.

In [32] a fully parameterizable ABC processor is introduced, which can be used with any field and irreducible polynomial without need for reconfiguration. This implementation although highly flexible is slow and does not reach required speeds for high bandwidth applications. A 239 bit point multiplication requires 12.8ms, clearly this is extremely high compared to other reported implementations.

In [33], the ECC processor designed has squarers, adders ,and multipliers in the data path. The authors have used a hybrid coordinate representation in affine, Jacobian, and López-Dahab form.

In [34] an end-to-end system for ECC is developed, which has a hardware implementation for ECC on an FPGA. The high performance is obtained with an optimized field multiplier. A digit-serial shift-and-add multiplier is used for the purpose. Inversion is done with a dedicated division circuit.

The processor presented in [35] achieves point multiplication in 0.074ms over the field $GF(2^{163})$. However, the implementation is for a specific form of elliptic curves called Koblitz curves. On these curves, several acceleration techniques based on precomputation [36] are possible. However our work focuses on generic curves where such accelerations do not work.

In [37] a high speed elliptic curve processor is presented for the field $GF(2^{191})$, where point multiplication is done in 0.056ms. A binary Karatsuba multiplier is used for the field multiplication. However, no inverse algorithm seems to be specified in the paper, making the implementation incomplete.

In [38] a microcoded approach is followed for ECC making it easy to modify, change, and optimize. The microcode is stored in the block RAM [39] and does not require additional resources.

In [40], the finite field multiplier in the processor is prevented from becoming idle. The finite field multiplier is the bottle neck of the design therefore preventing it from becoming idle improves the overall performance. Our design of the ECCP is on similar lines where the operations required for point addition and point doubling are scheduled so that the finite field multiplier is always utilized.

In [1], a pipelined ECC processor is developed which uses a combined algorithm to perform point doubling and point addition. This computes the scalar product in 0.019ms for an elliptic curve over $GF(2^{163})$. This is the fastest reported in literature. However, the seven stage pipeline used has huge area requirements.

In this thesis, high performance is attained by focusing on efficient implementations of the finite field primitives. The algorithms used for the critical finite field operations are tuned for the FPGA platform. Our novel finite field multiplier is a combinationl circuit and produces the output in one clock cycle. This has tremendous performance benefits. The proposed inversion algorithm is the fastest reported in literature. These efficient underlying primitives result in one of the fastest elliptic curve processors even though no pipelining is used.

2.6 Conclusion

In this chapter a brief introduction of elliptic curve cryptography was made, and the hierarchy in an elliptic curve processor was presented. A review of the existing literature on elliptic curve crypto processors was made. Hardware platforms used for elliptic curve cryptography were discussed, with special focus on FPGA architectures. The vulnerability of crypto processors to side channel attacks was also presented.

CHAPTER 3

Mathematical Background

Understanding *Elliptic Curve Cryptography* (ECC) requires a good understanding of the underlying mathematics. ECC relies heavily on abstract algebra for its construction. This chapter therefore starts with a brief overview of the primitive algebraic structures, namely groups, rings, and fields. The second part of this chapter is dedicated to the mathematics behind elliptic curves. In specific, elliptic curves over finite fields of the form $GF(2^m)$ are considered. The operations on this form of elliptic curve are discussed.

3.1 Abstract Algebra

3.1.1 Groups, Rings and Fields

Definition 3.1.1 *A* group *denoted by* $\{\mathbb{G}, \cdot\}$*, is a set of elements* $\mathbb{G}$ *with a binary operation '*$\cdot$*', such that for each ordered pair* (a, b) *of elements in* $\mathbb{G}$*, the following axioms are obeyed [41, 42]:*

- Closure : *If* $a, b \in \mathbb{G}$*, then* $a \cdot b \in \mathbb{G}$*.*
- Associative : $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ *for all* $a, b, c \in \mathbb{G}$*.*
- Identity element : *There is a unique element* $e \in \mathbb{G}$ *such that* $a \cdot e = e \cdot a = a$ *for all* $a \in \mathbb{G}$*.*
- Inverse element : *For each* $a \in \mathbb{G}$*, there is an element* $a' \in \mathbb{G}$ *such that* $a \cdot a' = a' \cdot a = e$

If the group also satisfies $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{G}$ then it is known as a *commutative* or an *abelian group*.

Definition 3.1.2 *A* ring *denoted by* $\{\mathbb{R}, +, \times\}$ *or simply* $\mathbb{R}$ *is a set of elements with two binary operations called addition and multiplication, such that for all* $a, b, c \in \mathbb{R}$ *the following are satisfied:*

- $\mathbb{R}$ *is an abelian group under addition.*
- *The closure property of* $\mathbb{R}$ *is satisfied under multiplication.*
- *The associativity property of* $\mathbb{R}$ *is satisfied under multiplication.*
- Distributive Law : *For all* $a, b, c \in \mathbb{R}$*,* $a \cdot (b+c) = ab+ac$ *and* $(a+b) \cdot c = ac+bc$*.*

The set of integers, rational numbers, real numbers, and complex numbers are all rings. A ring is said to be commutative if the commutative property under multiplication holds. That is, for all $a, b \in \mathbb{R}, a \cdot b = b \cdot a$.

Definition 3.1.3 *A* field *denoted by* $\{\mathbb{F}, +, \times\}$ *or simply* $\mathbb{F}$ *is a commutative ring which satisfies the following properties*

- *There exists a multiplicative identity element denoted by* 1 *such that for every* $a \in \mathbb{F}$*,* $a \cdot 1 = 1 \cdot a = 1$*.*
- Multiplicative inverse : *For every element* $a \in \mathbb{F}$ *except* 0*, there exists a unique element* a^{-1} *such that* $a \cdot (a^{-1}) = (a^{-1}) \cdot a = 1$*.* a^{-1} *is called the multiplicative inverse of the element* a*.*
- No zero divisors : *If* $a, b \in \mathbb{F}$ *and* $a \cdot b = 0$*, then either* $a = 0$ *or* $b = 0$*.*

The set of rational numbers, real numbers and complex number are examples of fields, while the set of integers is not. This is because the multiplicative inverse property does not hold in the case of integers.

The above examples of fields have infinite elements. However in cryptography *finite fields* play an important role. A finite field is also known as *Galois field* and is denoted by $GF(p^m)$. Here, p is a prime called the *characteristic* of the field, while m is a positive integer. The *order* of the finite field, that is, the number of elements in the field is p^m. When $m = 1$, the resulting field is called a *prime field* and contains the *residue classes* modulo p[41].

In cryptography two of the most studied fields are finite fields of characteristic two and prime fields. Finite fields of characteristic two, denoted by $GF(2^m)$, is also known as *binary extension finite fields* or simply *binary finite fields*. They have several advantages when compared to prime fields. Most important is the fact that modern computer systems are built on the binary number system. With m bits all possible elements of $GF(2^m)$ can be represented. This is not possible with prime fields (with $p \neq 2$). For example a $GF(2^2)$ field would require 2 bits for representation and use all possible numbers generated by the two bits. A $GF(3)$ field would also require 2 bits for representing the three elements in the field. This leaves one of the four possible numbers generated by two bits unused leading to an inefficient representation. Another advantage of binary extension fields is the simple hardware required for computation of some of the commonly used arithmetic operations such as addition and squaring. Addition in binary extension fields can be easily performed by a simple XOR. There is no carry generated. Squaring in this field is a linear operation and can also be done using XOR circuits. These circuits are much more simple than the addition and squaring circuits of a $GF(p)$ field.

3.1.2 Binary Finite Fields

A polynomial of the form $a(x) = a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0$ is said to be a *polynomial over* $GF(2)$ if the coefficients a_m, a_{m-1}, $\cdots, a_1$, a_0 are in $GF(2)$. Further, the polynomial is said to be *irreducible* over $GF(2)$ if $a(x)$ is divisible only by

c or by $c \cdot a(x)$ where $c \in GF(2)$ [43]. An irreducible polynomial of degree m with coefficients in $GF(2)$ can be used to construct the extension field $G(2^m)$. All elements of the extension field can be represented by polynomials of degree $m-1$ over $GF(2)$.

Binary finite fields are generally represented using two types of bases. These are the *polynomial* and *normal base* representations.

Definition 3.1.4 *Let* $p(x)$ *be an irreducible polynomial over* $GF(2^m)$ *and let* α *be the root of* $p(x)$*. Then the set*

$$\{1,\, \alpha,\, \alpha^2,\, \cdots, \alpha^{m-1}\}$$

is called the polynomial base.

Definition 3.1.5 *Let* $p(x)$ *be an irreducible polynomial over* $GF(2^m)$*, and let* α *be the root of* $p(x)$*, then the set*

$$\{\alpha,\, \alpha^{2^m},\, \alpha^{2^{2m}},\, \cdots, \alpha^{2^{m(m-1))}}\}$$

is called the normal base *if the* m *elements are linearly independent.*

Any element in the field $GF(2^m)$ can be represented in terms of its bases as shown below.

$$a(x) = a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0$$

Alternatively, the element $a(x)$ can be represented as a binary string $(a_{m-1},\, \cdots, a_1,\, a_0)$ making it suited for representation on computer systems. For example, the polynomial $x^4 + x^3 + x + 1$ in the field $GF(2^8)$ is represented as $(00011011)_2$.

Various arithmetic operations such as addition, subtraction, multiplication, squaring and inversion are carried out on binary fields. *Addition* and *subtraction* operations are identical and are performed by XOR operations.

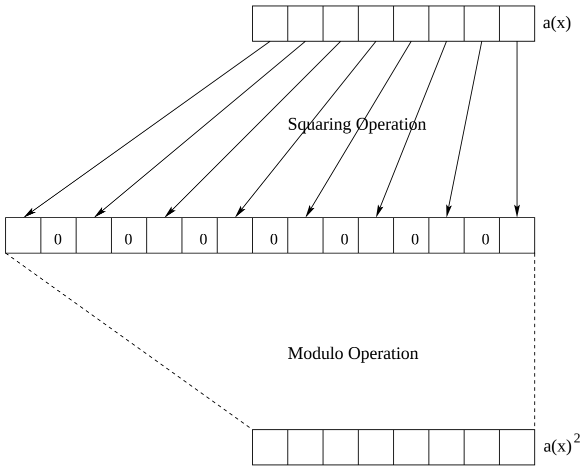

Fig. 3.1: Squaring Circuit

Let $a(x)$, $b(x) \in GF(2^m)$ be denoted by

$$a(x) = \sum_{i=0}^{m-1} a_i x^i \qquad b(x) = \sum_{i=0}^{m-1} b_i x^i$$

then the *addition* (or *subtraction*) of $a(x)$ and $b(x)$ is given by

$$a(x) + b(x) = \sum_{i=0}^{m-1} (a_i + b_i) x^i \tag{3.1}$$

here the $+$ between a_i and b_i denotes a XOR operation.

The *squaring* operation on binary finite fields is as easy as addition. The square of the polynomial $a(x) \in GF(2^m)$ is given by

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} \bmod p(x) \tag{3.2}$$

The squaring essentially spreads out the input bits by inserting zeroes in between two bits as shown in Figure 3.1.

Multiplication is not as trivial as addition or squaring. The product of the two polynomials $a(x)$ and $b(x)$ is given by

$$a(x) \cdot b(x) = \Big(\sum_{i=0}^{n-1} b(x) a_i x^i \Big) \bmod p(x) \tag{3.3}$$

Most multiplication algorithms are of order $O(n^2)$.

Inversion is the most complex of all field operations. Even the best technique to implement inversion is several times more complex than multiplication. Hence, algorithms which use finite field arithmetic generally try to reduce the number of inversions at the cost of increasing the number of multiplications.

The multiplication and squaring operation require a *modular operation* to be done. The modular operation is the remainder produced when divided by the field's irreducible polynomial. If a certain class of irreducible polynomials is used, the modular operation can be easily done. Consider the irreducible trinomial $x^m + x^n + 1$, having a

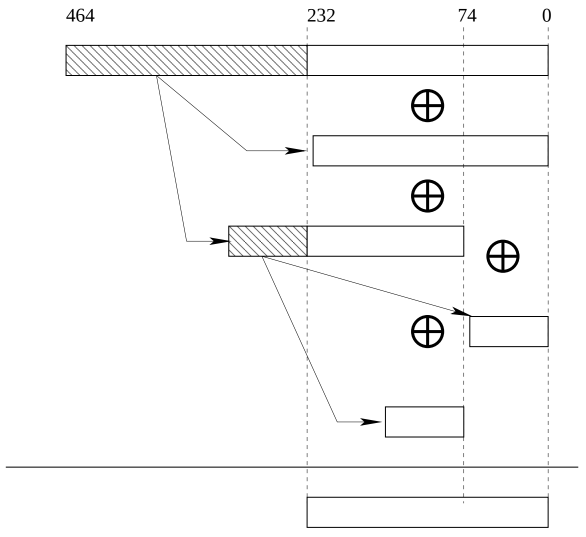

Fig. 3.2: Modular Reduction with Trinomial $x^{233} + x^{74} + 1$

root α and $1 < n < m/2$. Therefore $\alpha^m + \alpha^n + 1 = 0$. Therefore,

$$\begin{aligned}
\alpha^{m} &= 1 + \alpha^{n} \\
\alpha^{m+1} &= \alpha + \alpha^{n+1} \\
&\vdots \\
\alpha^{2m-3} &= \alpha^{m-3} + \alpha^{m+n-3} \\
\alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2}
\end{aligned} \tag{3.4}$$

For example, consider the irreducible trinomial $x^{233} + x^{74} + 1$. The multiplication or squaring of the polynomial results in a polynomial of degree at most 464. This can be reduced as shown in Figure 3.2. The higher order terms 233 to 464 are reduced by using Equation 3.4.

3.2 Elliptic Curves

Definition 3.2.1 *An* elliptic curve E over the field $GF(2^m)$ *is given by the simplified form of the Weierstraß equation mentioned in Equation 2.1. The simplified Weierstraß equation is :*

$$y^2 + xy = x^3 + ax^2 + b \tag{3.5}$$

with the coefficients a and b in $GF(2^m)$ and $b \neq 0$.

If $b \neq 0$, then the curve in Equation 3.5 is a *non-singular curve*. A point on the curve is said to be *singular* if its partial derivatives vanish.

The set of points on the elliptic curve along with a special point $\mathcal{O}$, called the *point at infinity*, form a group under addition. The identity element of the group is the point at infinity ($\mathcal{O}$). The arithmetic operations permitted on the group are point inversion, point addition and point doubling which are described as follows.

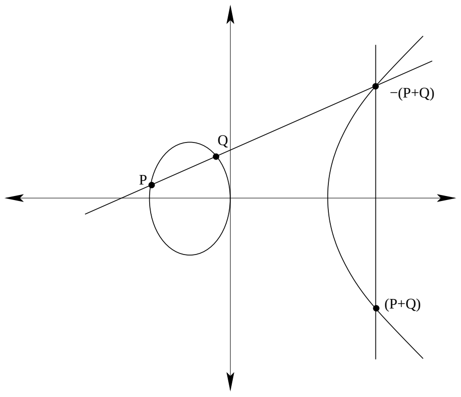

Fig. 3.3: Point Addition

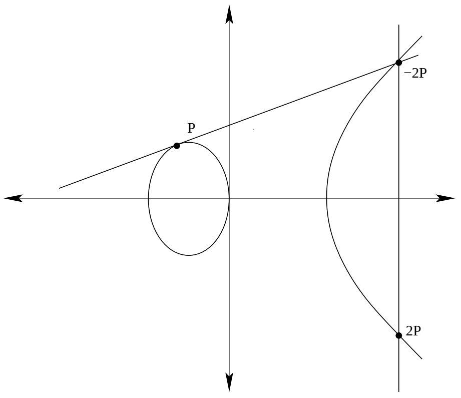

Fig. 3.4: Point Doubling

Point Inversion : Let P be a point on the curve with coordinates (x_1, y_1), then the inverse of P is the point $-P$ with coordinates $(x_1, x_1 + y_1)$. The point $-P$ is obtained by drawing a vertical line through P. The point at which the line intersects the curve is the inverse of P.

Point Addition : Let P and Q be two points on the curve with coordinates (x_1, y_1) and (x_2, y_2). Also, let $P \neq \pm Q$, then adding the two points results in a third point $R = (P + Q)$. The addition is performed by drawing a line through P and Q as shown in Figure 3.3. The point at which the line intersects the curve is $-(P + Q)$. The inverse of this is $R = (P + Q)$. Let the coordinates of R be (x_3, y_3), then the equations for x_3 and y_3 is

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \end{aligned} \tag{3.6}$$

where $\lambda = (y_1 + y_2)/(x_1 + x_2)$. If $P = -Q$, then $P + (-P)$ is $\mathcal{O}$.

Point Doubling : Let P be a point on the curve with coordinates (x_1, y_1) and $P \neq -P$. The double of P is the point $2 \cdot P = (x_3, y_3)$ obtained by drawing a tangent to the curve through P. The inverse of the point at which the tangent intersects the curve is

Algorithm 3.1: Double and Add algorithm for scalar multiplication

```
Input: Basepoint $P = (p_x, p_y)$ and Scalar $k = (k_{m-1}, k_{m-2} \cdots k_0)_2$, where
       $k_{m-1} = 1$
Output: Point on the curve $Q = kP$
1 $Q = P$
2 for $i = m - 2$ to $0$ do
3     $Q = 2 \cdot Q$
4     if $k_i = 1$ then
5         $Q = Q + P$
6     end
7 end
```

Table 3.1: Scalar Multiplication using Double and Add to find $22P$

i	k_i	Operation	Q
3	0	Double only	$2P$
2	1	Double and Add	$5P$
1	1	Double and Add	$11P$
0	0	Double only	**22P**

the double of P (Figure 3.4). The equation for computing $2 \cdot P$ is given as

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a = {x_1}^2 + \frac{b}{{x_1}^2} \\ y_3 &= {x_1}^2 + \lambda x_3 + x_3 \end{aligned} \tag{3.7}$$

where $\lambda = x_1 + (y_1/x_1)$.

The fundamental algorithm for ECC is the *scalar multiplication* (defined in Section 2.1). The basic double and add algorithm to perform scalar multiplication is shown in Algorithm 3.1. The input to the algorithm is a *basepoint* P and a m bit scalar k. The result is the scalar product kP.

As an example of how Algorithm 3.1 works, consider $k = 22$. The binary equivalent of this is $(10110)_2$. Table 3.1 below shows how $22P$ is computed.

Each iteration of i does a doubling on Q if k_i is 0 or a doubling followed by an addition if k_i is 1. The underlying operations in the addition and doubling equations use the finite field arithmetic discussed in the previous section. Both point doubling and point addition have 1 inversion (I) and 2 multiplications (M) each (from Equations 3.6 and 3.7). From this, the entire scalar multiplier for the m bit scalar k will have $m(1I + 2M)$ doublings and $\frac{m}{2}(1I + 2M)$ additions (assuming k has approximately $m/2$ ones on an average). The overall expected running time of the scalar multiplier is therefore obtained as

$$t_a \approx (3M + \frac{3}{2}I)m \tag{3.8}$$

For this expected running time, finite field addition and squaring operations have been neglected as they are simple operations and can be considered to have no overhead to the run time.

3.2.1 Projective Coordinate Representation

The complexity of a finite field inversion is typically eight times that of a finite field multiplier in the same field [44]. Therefore, there is a huge motivation for an alternate point representation which would require lesser inversions. The two point coordinate system (x, y) used in Equations 3.5, 3.6 and 3.7 discussed in the previous section is called *affine representation*. It has been shown that each affine point on the elliptic curve has a one to one correspondence with a unique equivalence class in which each point is represented by three coordinates (X, Y, Z). The three point coordinate system is called the *projective representation* [11]. In the projective representation, inversions are replaced by multiplications. The projective form of the Weierstraß equation can be obtained by replacing x with X/Z^c and y by Y/Z^d. There are several projective coordinates systems proposed. The most commonly used projective coordinate system are the *standard* where $c = 1$ and $d = 1$, the *Jacobian* with $c = 2$ and $d = 3$ and the *López-Dahab (LD) coordinates*[11] which has $c = 1$ and $d = 2$. The LD coordinate system [30] allows point addition using *mixed coordinates*, i.e. one point in affine while the other in projective.

Replacing x by X/Z and y by Y/Z^2 in Equation 3.5 results in the LD projective form of the Weierstraß equation.

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^4 \tag{3.9}$$

Let $P = (X_1, Y_1, Z_1)$ be an LD projective point on the elliptic curve, then the inverse of point P is given by $-P = (X_1, X_1Z_1 + Y_1, Z_1)$. Also, $P + (-P) = \mathcal{O}$, where $\mathcal{O}$ is the point at infinity. In LD projective coordinates $\mathcal{O}$ is represented as $(1, 0, 0)$.

The equation for doubling the point P in LD projective coordinates [30] results in the point $2P = (X_3, Y_3, Z_3)$. This is given by the following equation.

$$\begin{aligned} Z_3 &= X_1^2 \cdot Z_1^2 \\ X_3 &= X_1^4 + b \cdot Z_1^4 \\ Y_3 &= b \cdot Z_1^4 \cdot Z_3 + X_3 \cdot (a \cdot Z_3 + Y_1^2 + b \cdot Z_1^4) \end{aligned} \tag{3.10}$$

The equations for doubling require 5 finite field multiplications and zero inversions.

The equation in LD coordinates for adding the affine point $Q = (x_2, y_2)$ to P, where

$Q \neq \pm P$, is shown in Equation 3.11. The resulting point is $P + Q = (X_3, Y_3, Z_3)$.

$$\begin{aligned}
A &= y_2 \cdot Z_1^2 + Y_1 \\
B &= x_2 \cdot Z_1 + X_1 \\
C &= Z_1 \cdot B \\
D &= B^2 \cdot (C + a \cdot Z_1^2) \\
Z_3 &= C^2 \\
E &= A \cdot C \\
X_3 &= A^2 + D + E \\
F &= X_3 + x_2 \cdot Z_3 \\
G &= (x_2 + y_2) \cdot Z_3^2 \\
Y_3 &= (E + Z_3) \cdot F + G
\end{aligned} \tag{3.11}$$

Point addition in LD coordinates now require 9 finite field multiplications and zero inversions. For an m bit scalar with approximately half the bits one, the running time expected is given by Equation 3.12. One inversion and 2 multiplications are required at the end to convert the result from projective coordinates back into affine.

$$\begin{aligned}
t_{ld} &\approx m(5M + \frac{9M}{2}) + 2M + 1I \\
&= (9.5m + 2)M + 1I
\end{aligned} \tag{3.12}$$

The LD coordinates require several multiplications to be done but have the advantage of requiring just one inversion. To be beneficial, the extra multiplications should have a lower complexity than the inversions removed.

3.3 Conclusion

This chapter presented the necessary mathematical background required for this thesis. The performance of the entire elliptic curve crypto processor depends on the underlying finite field primitives therefore the primitives should be efficiently implemented. The next two chapters discuss implementations of two of the most dominant primitives used in ECC, namely the finite field multiplication and inversion.

CHAPTER 4

Architecting an Efficient Implementation of a Finite Field Multiplier on FPGA Platforms

The finite field multiplier forms the most important component in the elliptic curve crypto processor (ECCP). It occupies the most area on the device and also has the longest latency. The performance of the ECCP is affected most by the multiplier. Finite field multiplication of two elements in the field $GF(2^m)$ is defined as

$$C(x) = A(x) \cdot B(x) \, mod \, P(x) \tag{4.1}$$

where $C(x)$, $A(x)$, and $B(x)$ are in $GF(2^m)$ and $P(x)$ is the irreducible polynomial that generates the field $GF(2^m)$. Implementing the multiplication requires two steps. First, the polynomial product $C'(x) = A(x) \cdot B(x)$ is determined, then the modulo operation is done on $C'(x)$. This chapter deals with polynomial multiplication.

The organization of the chapter is as follows: the next section contains a brief overview of important finite field multipliers in literature. Section 4.2 discusses the Karatsuba algorithm in greater detail. Section 4.3 outlines some of the Karatsuba multiplication variants used for elliptic curves. Section 4.4 presents how a circuit gets mapped to a four input LUT based FPGA. Section 4.5 analyzes how the existing Karatsuba algorithms get mapped on to the FPGA. It also presents the proposed hybrid Karatsuba multiplier which maximizes utilization of FPGA resources. Section 4.6 compares the performance of the hybrid Karatsuba multiplier with existing implementations of the Karatsuba algorithm. The final section has the conclusion.

4.1 Finite Field Multipliers for High Performance Applications

The *school book* method to multiply two polynomials requires m^2 AND gates to generate the partial products. The final product is formed by adding the partial products. Since we deal with binary fields, addition is easily done using XOR gates without any carries being propagated, thus $(m-1)^2$ XOR gates are required to do the additions.

The *Massey-Omura* multiplier operates in normal basis representations of the field elements. With this representation, the structure of the multiplication becomes highly uniform resulting in efficient hardware architecture. The architecture takes a parallel input but the result is produced serially [45].

Another multiplier based on normal basis is the *Sunar-Koç* [46] multiplier. The multiplier requires lesser hardware compared to the Massey-Omura multiplier but has similar timing requirements.

In [47], the *Montgomery multiplier* is adapted to binary finite fields. The multiplication in Equation 4.1 is represented by the following equation

$$C(x) = A(x) \cdot B(x) \cdot R(x)^{-1} mod P(x) \tag{4.2}$$

where, $R(x)$ is of the form x^k and is an element in the field. Also, $gcd(R(x), P(x)) = 1$. The division by $R(x)$ reduces the complexity of the modular operation. For binary finite fields, $R(x)$ has the form 2^k, therefore division by $R(x)$ can be easily accomplished on a computer. This multiplier is best suited for low resource environments where speed of operation is not so important [44].

The *Karatsuba multiplier* [12] uses a divide and conquer approach to multiply $A(x)$ and $B(x)$. The m term polynomials are recursively split into two. With each split the size of the multiplication required reduces by half. This leads to a reduction in the number of AND gates required at the cost of an increase in XOR gates. This also results in the multiplier having a space complexity of $O(m^{log_2 3})$ for polynomial representations of finite fields. A comparison of all available multipliers show that only the Karatsuba multiplier has a complexity which is of sub quadratic order. All other multipliers have a complexity which is quadratic. Besides this, it has been shown in [44] and [48] that the Karatsuba multiplier if designed properly is also the fastest.

For a high performance elliptic curve crypto processor, the finite field multiplier with the smallest delay and the least number of clock cycles is best suited. Karatsuba multiplier if properly designed, attains the above speed requirements and at the same time has a sub-quadratic space complexity. This makes the Karatsuba multiplier the best choice for high performance applications.

4.2 Karatsuba Multiplication

In the Karatsuba multiplier, the m bit multiplicands $A(x)$ and $B(x)$ represented in polynomial basis are split as shown in Equation 4.3. For brevity, the equations that follow represent the polynomials $A_h(x)$, $A_l(x)$, $B_h(x)$, and $B_l(x)$ by A_h, A_l, B_h, and B_l respectively.

$$\begin{aligned} A(x) &= A_h x^{m/2} + A_l \\ B(x) &= B_h x^{m/2} + B_l \end{aligned} \tag{4.3}$$

The multiplication is then done using three $m/2$ bit multiplications as shown in Equation 4.4.

$$\begin{aligned}
C'(x) &= (A_h x^{m/2} + A_l)(B_h x^{m/2} + B_l) \\
&= A_h B_h x^m + (A_h B_l + A_l B_h) x^{m/2} + A_l B_l \\
&= A_h B_h x^m \\
&\quad + ((A_h + A_l)(B_h + B_l) + A_h B_h + A_l B_l) x^{m/2} \\
&\quad + A_l B_l
\end{aligned} \tag{4.4}$$

The Karatsuba multiplier can be applied recursively to each $m/2$ bit multiplication in Equation 4.4. Ideally this multiplier is best suited when m is a power of 2, this allows the multiplicands to be broken down until they reach 2 bits. The final recursion consisting of 2 bit multiplications can be achieved by AND gates. Such a multiplier with m a power of 2 is called the *basic Karatsuba multiplier*.

4.3 Karatsuba Multipliers for Elliptic Curves

The basic recursive Karatsuba multiplier cannot be applied directly to ECC because the binary extension fields used in standards such as [14] have a prime degree. There have been several published works which implement a modified Karatsuba algorithm for use in elliptic curves. There are two main design approaches followed. The first approach is a sequential circuit having less hardware and latency but requiring several clock cycles to produce the result. Generally at every clock cycle the outputs are fed-back into the circuit thus reusing the hardware. The advantage of this approach is that it can be pipelined. Examples of implementations following this approach can be found in[48–51]. The second approach is a combinational circuit having large area and delay but is capable of generating the result in one clock cycle. Examples of this approach can found in [52–55]. Our proposed Karatsuba multiplier follows the second approach, therefore in the remaining part of this section we analyze the combinational circuits for Karatsuba multipliers.

The easiest method to modify the Karatsuba algorithm for elliptic curves is by padding. The *padded Karatsuba multiplier* extends the m bit multiplicands to $2^{\lceil log_2 m \rceil}$ bits by padding the most significant bits with zeroes. This allows the use of the basic recursive Karatsuba algorithm. The obvious drawback of this method is the extra arithmetic introduced due to the padding.

In [53], a *binary Karatsuba multiplier* was proposed to handle multiplications in any field of the form $GF(2^m)$, where $m = 2^k + d$ and k is the largest integer such that $2^k < m$. The binary Karatsuba multiplier splits the m bit multiplicands ($A(x)$ and $B(x)$) into two terms. The lower terms (A_l and B_l) have 2^k bits while the higher terms (A_h and B_h) have d bits. Two 2^k bit multipliers are required to obtain the partial products $A_l B_l$ and $(A_h + A_l)(B_h + B_l)$. For the latter multiplication, the A_h and B_h terms have to be padded with $2^k - d$ bits. $A_h B_h$ product is determined using a d bit binary Karatsuba multiplier.

The *simple Karatsuba multiplier* [55] is the basic recursive Karatsuba multiplier with a small modification. If an m bit multiplication is needed to be done, m being any integer, it is split into two polynomials as in Equation 4.3. The A_l and B_l terms have $\lceil m/2 \rceil$ bits and the A_h and B_h terms have $\lfloor m/2 \rfloor$ bits. The Karatsuba multiplication can then be done with two $\lceil m/2 \rceil$ bit multiplications and one $\lfloor m/2 \rfloor$ bit multiplication. The upper bound for the number of AND gates and XOR gates required for the simple Karatsuba multiplier is the same as that of a $2^{\lceil log_2 m \rceil}$ bit basic recursive Karatsuba multiplier. The maximum number of gates required and the time delay for an m bit simple Karatsuba multiplier is given below.

$$\begin{aligned} \#AND\text{gates} &: 3^{\lceil log_2 m \rceil} \\ \#XOR\text{gates} &: \sum_{r=0}^{\lceil log_2 m \rceil} 3^r \Big(4\lceil m/2^r \rceil - 4\Big) \end{aligned} \tag{4.5}$$

In the *general Karatsuba multiplier* [55], the multiplicands are split into more than two terms. For example an m term multiplier is split into m different terms. The number of gates required is given below.

$$\begin{aligned} \#AND\text{gates } &: m(m+1)/2 \\ \#XOR\text{gates } &: \frac{5}{2}m^2 - \frac{7}{2}m + 1 \end{aligned} \tag{4.6}$$

4.4 Designing for the FPGA Architecture

Maximizing the performance of a hardware design requires the design to be customized for the target architecture. The smallest programmable entity on an FPGA is the look up table (LUT) (Section 2.3.1). A LUT generally has four inputs and can be configured for any logic function having a maximum of four inputs. The LUT can also be used to implement logic functions having less than four inputs, two for example. In this case, only half the LUT is utilized the remaining part is not utilized. Such a LUT having less than four inputs is an *under utilized LUT*. For example, the logic function $y = x_1 + x_2$ under utilizes the LUT as it has only two inputs. *Most compact implementations are obtained when the utilization of each LUT is maximized.* From the above fact it may be derived that the minimum number of LUTs required for a q bit combinational circuit is given by Equation 4.7.

$$\#LUT(q) = \begin{cases} 0 & \text{if } q = 1 \\ 1 & \text{if } 1 < q \leq 4 \\ \lceil q/3 \rceil & \text{if } q > 4 \text{ and } q \bmod 3 = 2 \\ \lfloor q/3 \rfloor & \text{if } q > 4 \text{ and } q \bmod 3 \neq 2 \end{cases} \tag{4.7}$$

The delay of the q bit combinational circuit in terms of LUTs is given by Equation 4.8, where D_{LUT} is the delay of one LUT.

$$DELAY(q) = \lceil log_4(q) \rceil * D_{LUT} \tag{4.8}$$

The percentage of under utilized LUTs in a design is determined using Equation 4.9. Here, LUT_k signifies that k inputs out of 4 are used by the design block realized by the LUT. So, LUT_2 and LUT_3 are under utilized LUTs, while LUT_4 is fully utilized.

$$\%UnderUtilizedLUTs = \frac{LUT_2 + LUT_3}{LUT_2 + LUT_3 + LUT_4} * 100 \tag{4.9}$$

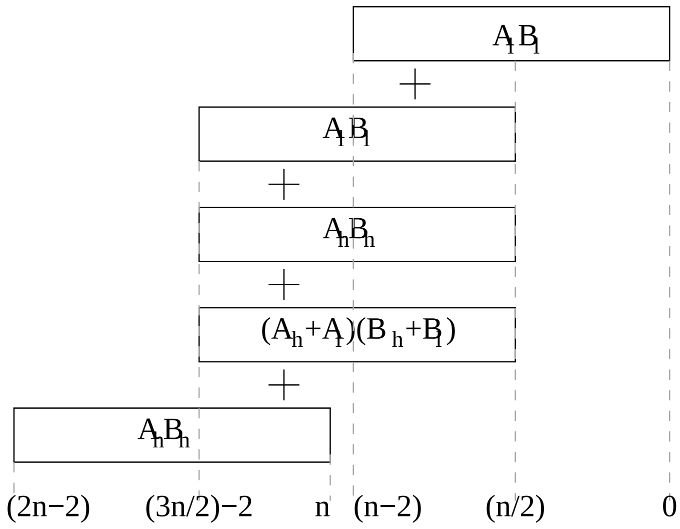

Fig. 4.1: Combining the Partial Products in a Karatsuba Multiplier

4.5 Analyzing Karatsuba Multipliers on FPGA Platforms

In this section we discuss the mapping of various Karatsuba algorithms on an FPGA. We estimate the amount of FPGA resources that is required for the implementations.

Recursive Karatsuba Multiplier : In an m $(= 2^k)$ bit recursive Karatsuba multiplier the basic Karatsuba algorithm of [12] is applied recursively. Each recursion reduces the size of the input by half while tripling the number of multiplications required. At each recursion, except the final, only XOR operations are involved. Let $n = 2^{(log_2 m)-k}$ be the size of the inputs (A and B) for the k^{th} recursion of the m bit multiplier. There are 3^k

such n bit multipliers required. The A and B inputs are split into two: A_h, A_l and B_h, B_l respectively with each term having $n/2$ bits. $n/2$ two input XORs are required for the computation of $A_h + A_l$ and $B_h + B_l$ respectively (Equation 4.4). Each two input XOR requires one LUT on the FPGA, thus in total there are n LUTs required. Combining the partial products as shown in Figure 4.1 is the last step of the recursion. Determining the output bits $n-2$ to $n/2$ and $3n/2-2$ to n requires $3(n/2-1)$ two input XORs each. The output bit $n-1$ requires 2 two input XORs. In all $(3n-4)$ two input $XORs$ are required to add the partial products. The number of LUTs required to combine the partial products is much lower. This is because each LUT implements a four input XOR. Each output bit $n/2$ to $3n/2-2$ requires one LUT, therefore $(n-1)$ LUTs are required for the purpose. In total, $2n-1$ LUTs are required for each recursion on the FPGA. The final recursion has $3^{(log_2 m)-1}$ two bit Karatsuba multipliers. The equation for the two bit Karatsuba multiplier is shown in Equation 4.10.

$$\begin{aligned} C_0 &= A_0 B_0 \\ C_1 &= A_0 B_0 + A_1 B_1 + (A_0 + A_1)(B_0 + B_1) \\ C_2 &= A_1 B_1 \end{aligned} \tag{4.10}$$

This requires three LUTs on the FPGA: one for each of the output bits (C_0, C_1, C_2).

The total number of LUTs required for the m bit recursive Karatsuba multiplication is given by Equation 4.11.

$$\begin{aligned} \#LUTS_R(m) &= 3 * 3^{log_2 m - 1} + \sum_{k=0}^{log_2 m - 2} 3^k (2 * 2^{log_2 m - k} - 1) \\ &= \sum_{k=0}^{log_2 m - 1} 3^k (2^{log_2 m - k + 1} - 1) \end{aligned} \tag{4.11}$$

The delay of the recursive Karatsuba multiplier in terms of LUTs is given by Equation 4.12. The first $log_2(m) - 1$ recursions have a delay of $2LUTs$. The last recursion has a delay of $1LUT$.

$$\begin{aligned} DELAY_R(m) &= (2(log_2(m) - 1) + 1)D_{LUT} \\ &= (2log_2(m) - 1)D_{LUT} \end{aligned} \tag{4.12}$$

When m is not necessarily a power of 2, the number of recursions of an m bit simple Karatsuba multiplier is equivalent to that of a $2^{\lceil log_2 m \rceil}$ recursive Karatsuba multiplier, therefore Equations 4.11 and 4.12 form the upper bound for the number of LUTs and delay of a simple Karatsuba multiplier [55] (Equations 4.13 and 4.14).

$$\#LUTS_S(m) \leq \#LUTS_R(2^{\lceil log_2 m \rceil}) \tag{4.13}$$

$$DELAY_S(m) \leq DELAY_R(2^{\lceil log_2 m \rceil}) \tag{4.14}$$

General Karatsuba Multiplier : The m bit general Karatsuba algorithm [55] is shown in Algorithm 4.1. Each iteration of i computes two output bits C_i and C_{2m-2-i}. Computing the two output bits require same amount of resources on the FPGA. The lines 6 and 7 in the algorithm is executed once for every even iteration of i and is not executed for odd iterations of i. The term $M_j + M_{i-j} + M_{(j,i-j)}$ is computed with the four inputs A_j, A_{i-j}, B_j and B_{i-j}, therefore on the FPGA, computing the term would require one LUT. For an odd i, C_i would have $\lceil i/2 \rceil$ such LUTs whose outputs have to be added. The number of LUTs required for this is obtained from Equation 4.7. An even value of i would have two additional inputs corresponding to $M_{i/2}$ that have to be added. The number of LUTs required for computing C_i ($0 \leq i \leq m - 1$) is given by Equation 4.15.

$$\#LUT_{c_i} = \begin{cases} 1 & \text{if } i = 0 \\ \lceil i/2 \rceil + \#LUT(\lceil i/2 \rceil) & \text{if } i \text{ is odd} \\ i/2 + \#LUT(i/2 + 2) & \text{if } i \text{ is even} \end{cases} \tag{4.15}$$

Algorithm 4.1: gkmul *(General Karatsuba Multiplier)*

```
   Input: $A, B$ are multiplicands of $m$ bits
   Output: $C$ of length $2m-1$ bits
   /* Define :  $M_x \rightarrow A_x B_x$                                   */
   /* Define :  $M_{(x,y)} \rightarrow (A_x + A_y)(B_x + B_y)$             */
1  begin
2      for $i = 0$ to $m-2$ do
3          $C_i = C_{2m-2-i} = 0$
4          for $j = 0$ to $\lfloor i/2 \rfloor$ do
5              if $i = 2j$ then
6                  $C_i = C_i + M_j$
7                  $C_{2m-2-i} = C_{2m-2-i} + M_{m-1-j}$
8              else
9                  $C_i = C_i + M_j + M_{i-j} + M_{(j,i-j)}$
10                 $C_{2m-2-i} = C_{2m-2-i} + M_{m-1-j}$
11                     $+M_{m-1-i+j} + M_{(m-1-j,m-1-i+j)}$
12             end
13         end
14     end
15     $C_{m-1} = 0$
16     for $j = 0$ to $\lfloor (m-1)/2 \rfloor$ do
17         if $m - 1 = 2j$ then
18             $C_{m-1} = C_{m-1} + M_j$
19         else
20             $C_{m-1} = C_{m-1} + M_j + M_{m-1-j} + M_{(j,m-1-j)}$
21         end
22     end
23 end
```

The total number of LUTs required for the general Karatsuba multiplier is given by Equation 4.16.

$$\#LUTS_G(m) = 2\left(\sum_{i=0}^{m-2} \#LUT_{C_i}\right) + \#LUT_{C_{m-1}} \tag{4.16}$$

When implemented in hardware, all output bits are computed simultaneously. The delay of the general Karatsuba multiplier (Equation 4.17) is equal to the delay of the output bit with the most terms. This is the output bit C_{m-1} (lines 15 to 22 in the Algorithm 4.1). Equation 4.17 is obtained from Equation 4.15 with $i = m-1$. The

Table 4.1: Comparison of LUT Utilization in Multipliers

m	General			Simple		
	Gates	LUTs	LUTs Under Utilized	Gates	LUTs	LUTs Under Utilized
2	7	3	66.6%	7	3	66.6%
4	37	11	45.5%	33	16	68.7%
8	169	53	20.7%	127	63	66.6%
16	721	188	17.0%	441	220	65.0%
29	2437	670	10.7%	1339	669	65.4%
32	2977	799	11.3%	1447	723	63.9%

$\lceil i/2 \rceil$ computations are done with a delay of one LUT (D_{LUT}). Equation 4.8 is used to compute the second term of Equation 4.17.

$$DELAY_G(m) = \begin{cases} D_{LUT} + DELAY(\lceil (m-1)/2 \rceil) & \text{if } m-1 \text{ is odd} \\ D_{LUT} + DELAY((m-1)/2+2) & \text{if } m-1 \text{ is even} \end{cases} \quad (4.17)$$

4.5.1 The Hybrid Karatsuba Multiplier

In this section we present our proposed multiplier called the *hybrid Karatsuba multiplier*. We show how we combine techniques to maximize utilization of LUTs resulting in minimum area.

Table 4.1 compares the general and simple Karatsuba algorithms for gate counts (two input XOR and AND gates), LUTs required on a *Xilinx Virtex 4 FPGA* and the percentage of LUTs under utilized (Equation 4.9).

The simple Karatsuba multiplier alone is not efficient for FPGA platforms as the number of under utilized LUTs is about 65%. For an m bit simple Karatsuba multiplier the two bit multipliers take up approximately a third of the area (for $m = 256$). In a two bit multiplier, two out of three LUTs required are under utilized (In Equation 4.10, C_0

and C_2 result in under utilized LUTs). In addition to this, around half the LUTs used for each recursion is under utilized. The under utilized LUTs results in a bloated area requirement on the FPGA.

The m-term general Karatsuba is more efficient on the FPGA for small values on m (Table 4.1) even though the gate count is significantly higher. This is because a large number of operations can be grouped in fours which fully utilizes the LUT. For small values of m ($m < 29$) the compactness obtained by the fully utilized LUTs is more prominent than the large gate count, resulting in low footprints on the FPGA. For $m \geq 29$, the gate count far exceeds the efficiency obtained by the fully utilized LUTs, resulting in larger footprints with respect to the simple Karatsuba implementation.

Algorithm 4.2: hmul *(Hybrid Karatsuba Multiplier)*

```
   Input: The multiplicands $A, B$ and their length $m$
   Output: $C$ of length $2m - 1$ bits
1  begin
2      if $m < 29$ then
3          return $gkmul(A, B, m)$
4      else
5          $l = \lceil m/2 \rceil$
6          $A^{'} = A_{[m-1 \cdots l]} + A_{[l-1 \cdots 0]}$
7          $B^{'} = B_{[m-1 \cdots l]} + B_{[l-1 \cdots 0]}$
8          $C_{p1} = hmul(A_{[l-1 \cdots 0]}, B_{[l-1 \cdots 0]}, l)$
9          $C_{p2} = hmul(A^{'}, B^{'}, l)$
10         $C_{p3} = hmul(A_{[m-1 \cdots l]}, B_{[m-1 \cdots l]}, m - l)$
11         return $(C_{p3} << 2l) + (C_{p1} + C_{p2} + C_{p3}) << l + C_{p1}$
           ;                                   /* << indicates left shift */
12
13     end
14 end
```

In our proposed hybrid Karatsuba multiplier shown in Algorithm 4.2, the m bit multiplicands are split into two parts when the number of bits is greater than or equal to the threshold 29. The higher term has $\lfloor m/2 \rfloor$ bits while the lower term has $\lceil m/2 \rceil$ bits. If the number of bits of the multiplicand is less than 29 the general Karatsuba algorithm

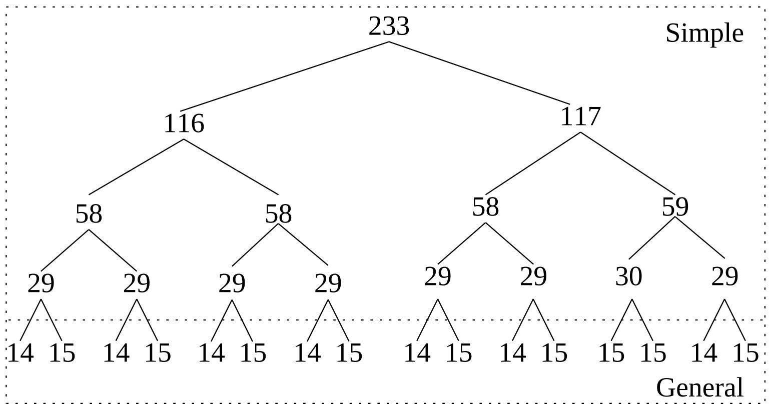

Fig. 4.2: 233 Bit Hybrid Karatsuba Multiplier

is invoked. The general Karatsuba algorithm ensures maximum utilization of the LUTs for the smaller bit multiplications, while the simple Karatsuba algorithm ensures least gate count for the larger bit multiplications. For a 233 bit hybrid Karatsuba multiplier (Figure 4.2), the multiplicands are split into two terms with A_h and B_h of 116 bits and A_l and B_l of 117 bits. The 116 bit multiplication is implemented using three 58 bit multipliers, while the 117 bit multiplier is implemented using two 59 bit multipliers and a 58 bit multiplier. The 58 and 59 bit multiplications are implemented with 29 and 30 bit multipliers, the 29 and 30 bit multiplications are done using 14 and 15 bit general Karatsuba multipliers.

The number of recursions in the hybrid Karatsuba multiplier is given by

$$r = \lceil log_2\Big(\frac{m}{29}\Big)\rceil + 1 \tag{4.18}$$

The i^{th} recursion ($0 < i < r$) of the m bit multiplier has 3^i multiplications. The multipliers in this recursion have bit lengths $\lceil m/2^i \rceil$ and $\lfloor m/2^i \rfloor$. For simplicity we assume the number of gates required for the $\lfloor m/2^i \rfloor$ bit multiplier is equal to that of the $\lceil m/2^i \rceil$ bit multiplier. The total number of AND gates required is the AND gates for the multiplier in the final recursion (i.e. $\lceil m/2^{r-1} \rceil$ bit multiplier) times the number of

$\lceil m/2^{r-1} \rceil$ multipliers present. Using Equation 4.6,

$$\#AND = \frac{3^{r-1}}{2} \lceil \frac{m}{2^{r-1}} \rceil \left(\lceil \frac{m}{2^{r-1}} \rceil + 1 \right) \tag{4.19}$$

The number of XOR gates required for the i^{th} recursion is given by $4\lceil \frac{m}{2^i} \rceil - 4$. The total number of two input $XORs$ is the sum of the $XORs$ required for last recursion, $\#XOR_{g_{r-1}}$, and the $XORs$ required for the other recursions, $\#XOR_{s_i}$. Using Equations 4.5 and 4.6,

$$\begin{aligned} \#XOR &= 3^{r-1} \#XOR_{g_{r-1}} + \sum_{i=1}^{r-2} 3^i \#XOR_{s_i} \\ &= 3^{r-1} \left(10 \lceil \frac{m}{2^r} \rceil^2 - 7 \lceil \frac{m}{2^r} \rceil + 1 \right) + \sum_{i=1}^{r-2} 3^i \left(4 \lceil \frac{m}{2^i} \rceil - 4 \right) \end{aligned} \tag{4.20}$$

The delay of the hybrid Karatsuba multiplier (Equation 4.21) is obtained by subtracting the delay of a $\lceil m/2^{r-1} \rceil$ bit simple Karatsuba multiplier from the delay of an m bit simple Karatsuba multiplier and adding the delay of a $\lceil m/2^{r-1} \rceil$ bit general Karatsuba multiplier.

$$\begin{aligned} DELAY_H(m) &= DELAY_S(m) \\ &\quad - DELAY_S(\lceil m/2^{r-1} \rceil) + DELAY_G(\lceil m/2^{r-1} \rceil) \end{aligned} \tag{4.21}$$

4.6 Performance Evaluation

The graph in Figure 4.3 compares the area time product for the hybrid Karatsuba multiplier with the simple Karatsuba multiplier and the binary Karatsuba multipliers for increasing values of m. The simple and binary Karatsuba multipliers were reimplemented and scaled for different field sizes. The results were obtained by synthesizing

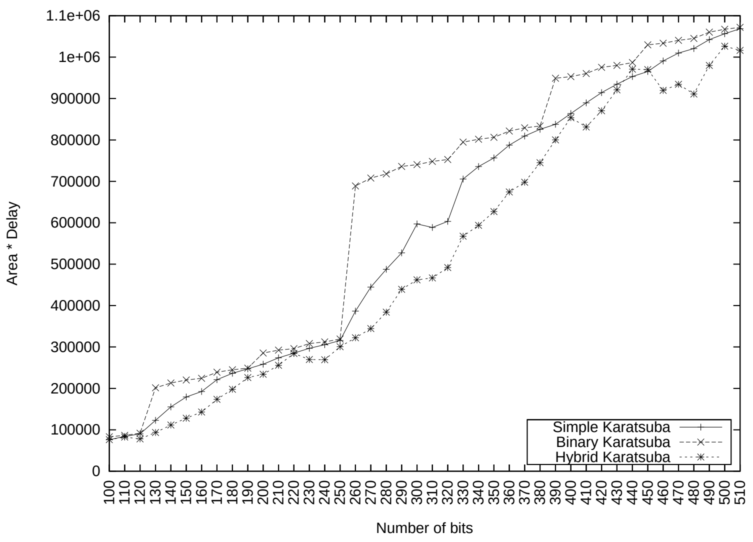

Fig. 4.3: m Bit Multiplication vs Area × Time

Table 4.2: Comparison of the Hybrid Karatsuba Multiplier with Reported FPGA Implementations

Multiplier	Platform	Field	Slices	Delay (ns)	Clock Cycles	Computation Time(ns)	Performance AT (μs)
Grabbe [48]	XC2V6000	240	1660	12.12	54	655	1087
Gathen [50]	XC2V6000	240	1480	12.6	30	378	559
This work	XC4V140	233	10434	16	1	16	154
	XC2VP100	233	12107	19.9	1	19.9	241

using *Xilinx's ISE* for a *Virtex 4* FPGA. The area was determined by the number of LUTs required for the multiplier, and the time in nano seconds includes the I/O pad delay. The graph shows that the area time product for the hybrid Karatsuba multiplier is lesser compared to the other multipliers. The power×delay graph for the multipliers is expected to be similar to the area×delay graph of Figure 4.3.

Table 4.2 compares the hybrid Karatsuba with reported FPGA implementations of Karatsuba variants. The implementations of [48] and [50] are sequential and hence require multiple clock cycles, thus they are not suited for high performance ECC. In order to alleviate this, we proposed a combinational Karatsuba multiplier. However to ensure that the design operates at a high clock frequency, we perform hardware replication. For example, in a 233 bit multiplier, 14 bit and 15 bit general Karatsuba multipliers are replicated, since the general Karatsuba multipliers utilize LUTs efficiently. This gain is reflected in Table 4.2.

4.7 Conclusion

In this chapter we discussed the finite field multiplication unit. We proposed a hybrid technique for implementing the Karatsuba multiplier. Our proposed design results in best area $\times$ time product on an FPGA compared to existing works. The hybrid Karatsuba multiplier forms the most important module for our elliptic curve crypto processor. In the next chapter, we discuss the finite field inversion which would also use the hybrid Karatsuba multiplier.

CHAPTER 5

High Performance Finite Field Inversion for FPGA Platforms

The *inverse* of a non zero element a in the field $GF(2^m)$ is the element $a^{-1} \in GF(2^m)$ such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$. Among all finite field operations, computing the inverse of an element is the most computationally intensive. Yet it forms an integral part of many public key cryptography algorithms including ECC. It is therefore important to have an efficient technique to find the multiplicative inverse.

This chapter is organized as follows : the next section has a brief discussion on various multiplicative inverse algorithms and reasons out why the Itoh-Tsujii algorithm is most suited for elliptic curve cryptography. Section 5.2 describes the Itoh-Tsujii algorithm and some of the reported literature on its implementation. Section 5.3 derives an equation to determine the number of clock cycles required to find the inverse. Section 5.4 proposes a generalized Itoh-Tsujii algorithm and presents a special case of the generalized version called the quad-Itoh Tsujii algorithm, which is efficient for FPGA platforms. This section also builds a controller that implements the quad-Itoh Tsujii algorithm. Section 5.5 has the performance evaluation of the proposed algorithm with the best existing inverse algorithms available. The final section has the conclusion.

5.1 Algorithms for Multiplicative Inverse

The most common algorithms for finding the multiplicative inverse are the extended Euclidean algorithms (EEA) and the Itoh-Tsujii Algorithm (ITA) [13]. Generally, the EEA and its variants, the binary EEA and Montgomery [56] inverse algorithms result in compact hardware implementations, while the ITA is faster. The large area required by the ITA is mainly due to the multiplication unit. All cryptographic applications require to perform finite field multiplications, hence their hardware implementations require a multiplier to be present. This multiplier can be reused by the ITA for inverse computations. In this case the multiplier need not be considered in the area required by the ITA. The resulting ITA without the multiplier is as compact as the EEA making it an ideal choice for multiplicative inverse hardware [44].

The Itoh-Tsujii algorithm was initially proposed to find the multiplicative inverse for normal basis representation of elements in the field $GF(2^m)$[13]. Since then, there have been several works that improved the original algorithm and adapted the algorithm to other basis representations [57–59]. In [57], inversion in polynomial basis representations of field elements was presented. In [58] addition chains were used efficiently to compute the multiplicative inverse in 27 clock cycles for an element represented in polynomial basis in the field $GF(2^{193})$. In [59] a parallel implementation of ITA was proposed to generate the inverse in 20 clock cycles for the same field and basis representation.

5.2 The Itoh-Tsujii Algorithm (ITA)

The Itoh-Tsujii Multiplicative Inverse Algorithm is based on Fermat's little theorem, by which the inverse of an element $a \in GF(2^m)$ is computed using Equation 5.1.

$$a^{-1} = a^{2^m-2} \tag{5.1}$$

The naive technique of implementing a^{-1} requires $(m-2)$ multiplications and $(m-1)$ squarings. Itoh and Tsujii in [13] reduced the number of multiplications required by using addition chains. An *addition chain* [60] for $n \in \mathbb{N}$ is a sequence of integers of the form $U = \begin{pmatrix} u_0 & u_1 & u_2 & \cdots u_r \end{pmatrix}$ satisfying the properties

- $u_0 = 1$
- $u_r = n$
- $u_i = u_j + u_k$, for some $k \leq j < i$

Brauer chains are a special class of addition chains in which $j = i - 1$. An *optimal addition chain* for n is the smallest addition chain for n.

To understand how the Itoh-Tsujii algorithm works Equation in 5.1 is rewritten as shown below.

$$a^{-1} = (a^{2^{m-1}-1})^2$$

We reuse notations from paper [59]. For $k \in \mathbb{N}$, let

$$\beta_k(a) = a^{2^k-1} \in GF(2^m)$$

then,

$$a^{-1} = [\beta_{m-1}(a)]^2$$

In [59] a recursive sequence (Equation 5.2) is used with an addition chain to compute the multiplicative inverse. $\beta_{k+j}(a) \in GF(2^m)$ can be expressed as shown in Equation 5.2. For simplicity of notation we shall represent $\beta_k(a)$ by β_k.

$$\beta_{k+j}(a) = (\beta_j)^{2^k}\beta_k = (\beta_k)^{2^j}\beta_j \tag{5.2}$$

As an example consider finding the inverse of an element $a \in GF(2^{233})$. This requires computing $\beta_{232}(a) = a^{2^{232}-1}$ and then doing a squaring (i.e. $[\beta_{232}(a)]^2 = a^{-1}$). A Brauer chain for 232 is as shown below.

$$U_1 = (\ 1\ \ 2\ \ 3\ \ 6\ \ 7\ \ 14\ \ 28\ \ 29\ \ 58\ \ 116\ \ 232\ ) \tag{5.3}$$

Table 5.1: Inverse of $a \in GF(2^{233})$ using generic ITA

	$\beta_{\mathbf{u_i}}(\mathbf{a})$	$\beta_{\mathbf{u_j}+\mathbf{u_k}}(\mathbf{a})$	Exponentiation
1	$\beta_1(a)$		a
2	$\beta_2(a)$	$\beta_{1+1}(a)$	$(\beta_1)^{2^1}\beta_1 = a^{2^2-1}$
3	$\beta_3(a)$	$\beta_{2+1}(a)$	$(\beta_2)^{2^1}\beta_1 = a^{2^3-1}$
4	$\beta_6(a)$	$\beta_{3+3}(a)$	$(\beta_3)^{2^3}\beta_3 = a^{2^6-1}$
5	$\beta_7(a)$	$\beta_{6+1}(a)$	$(\beta_6)^{2^1}\beta_1 = a^{2^7-1}$
6	$\beta_{14}(a)$	$\beta_{7+7}(a)$	$(\beta_7)^{2^7}\beta_7 = a^{2^{14}-1}$
7	$\beta_{28}(a)$	$\beta_{14+14}(a)$	$(\beta_{14})^{2^{14}}\beta_{14} = a^{2^{28}-1}$
8	$\beta_{29}(a)$	$\beta_{28+1}(a)$	$(\beta_{28})^{2^1}\beta_1 = a^{2^{29}-1}$
9	$\beta_{58}(a)$	$\beta_{29+29}(a)$	$(\beta_{29})^{2^{29}}\beta_{29} = a^{2^{58}-1}$
10	$\beta_{116}(a)$	$\beta_{58+58}(a)$	$(\beta_{58})^{2^{58}}\beta_{58} = a^{2^{116}-1}$
11	$\beta_{232}(a)$	$\beta_{116+116}(a)$	$(\beta_{116})^{2^{116}}\beta_{116} = a^{2^{232}-1}$

Computing $\beta_{232}(a)$ is done in 10 steps with 231 squarings and 10 multiplications as shown in Table 5.1.

In general if l is the length of the addition chain, finding the inverse of an element in $GF(2^m)$ requires $l - 1$ multiplications and $m - 1$ squarings. The length of the addition chain is related to m by the equation $l \leq \lfloor log_2 m \rfloor$ [60], therefore the number of multiplications required by the ITA is much lesser than that of the naive method.

5.3 Clock Cycles for the ITA

In the ITA for field $GF(2^m)$, the number of squarings required is as high as m. Further from Table 5.1, it may be noted that most of the squarings required is towards the end of the addition chain. The maximum number of squarings at any particular step could be as high as $u_i/2$. Although the circuit for a squarer is relatively simple, the large number of squarings required hampers the performance of the ITA. A straightforward way of implement the squarings would require $u_i/2$ clock cycles at each step. The technique used in [58] and [59] cascades u_s (where u_s is an element in the addition chain) squarers

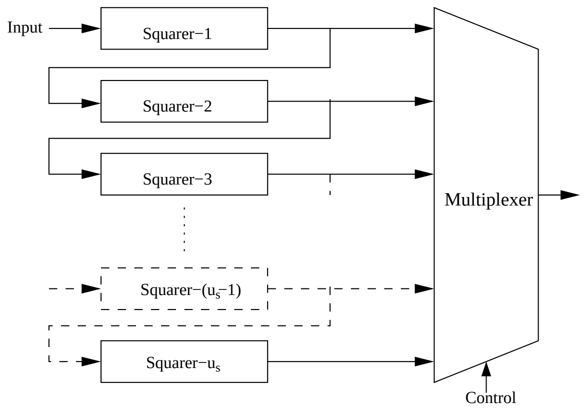

Fig. 5.1: Circuit to Raise the Input to the Power of 2^k

(Figure 5.1) so that the output of one squarer is fed to the input of the next. If the number of squarings required is less than u_s, a multiplexer is used to tap out interim outputs. In this case the output can be obtained in one clock cycle. If the number of squarings required is greater than u_s the output of the squaring block is fed back to get squares which are a multiple of u_s. For example, if u_i ($u_i > u_s$) squarings are needed, the output of the squarer block would be fed back $\lceil u_i/u_s \rceil$ times. This would also require $\lceil u_i/u_s \rceil$ clock cycles.

In addition to the squarings, each step in the ITA has exactly one multiplication requiring one clock cycle. The total number of clock cycles required for this design, assuming a Brauer chain, is given by Equation 5.4. The summation in the equation is the clock cycles for the squarings at each step of the algorithm. The $(l-1)$ term is due to the $(l-1)$ multiplications. The extra clock cycle is for the final squaring.

$$\begin{aligned} \#ClockCycles &= 1 + (l-1) + \sum_{i=2}^{l} \lceil \frac{u_i - u_{i-1}}{u_s} \rceil \\ &= l + \sum_{i=2}^{l} \lceil \frac{u_i - u_{i-1}}{u_s} \rceil \end{aligned} \tag{5.4}$$

In order to reduce the clock cycles a parallel architecture was proposed in [59]. The reduced clock cycles is achieved at the cost of increased hardware. In the remaining part of this section we propose a novel ITA designed for the FPGA architecture. The proposed design, though sequential, requires the same number of clock cycles as the parallel architecture of [59] but has better area$\times$time product.

5.4 Generalizing the Itoh-Tsujii Algorithm

The equation for the square of an element $a \in GF(2^m)$ is given by Equation 5.5, where $p(x)$ is the irreducible polynomial.

$$a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} \bmod p(x) \tag{5.5}$$

This is a linear equation and hence can be represented in the form of a matrix (T) as shown in the equation below.

$$a^2 = T \cdot a$$

The matrix depends on the finite field $GF(2^m)$ and the irreducible polynomial of the field. Exponentiation in the ITA is done with squarer circuits. We extend the ITA so that the exponentiation can be done with any 2^n circuit and not just squarers. Raising a to the power of 2^n is also linear and can be represented in the form of a matrix as shown below.

$$a^{2^n} = T^n(a) = T'a$$

For any $a \in GF(2^m)$ and $k \in \mathbb{N}$, Define,

$$\alpha_k(a) = a^{2^{nk}-1} \tag{5.6}$$

Theorem 5.4.1 *If* $a \in GF(2^m)$ *,* $\alpha_{k_1}(a) = a^{2^{nk_1}-1}$ *and* $\alpha_{k_2}(a) = a^{2^{nk_2}-1}$ *then*

$$\alpha_{k_1+k_2}(a) = (\alpha_{k_1}(a))^{2^{nk_2}} \alpha_{k_2}(a)$$

where k_1*,* k_2*, and* $n \in \mathbb{N}$

Proof

$$\begin{aligned}
\mathit{RHS} &= (\alpha_{k_1}(a))^{2^{nk_2}} \alpha_{k_2}(a) \\
&= (a^{2^{nk_1}-1})^{2^{nk_2}} (a^{2^{nk_2}-1}) \\
&= (a^{2^{n(k_1+k_2)}-2^{nk_2}+2^{nk_2}-1}) \\
&= (a^{2^{n(k_1+k_2)}-1}) \\
&= \alpha_{k_1+k_2}(a) \\
&= \mathit{LHS}
\end{aligned}$$

Theorem 5.4.2 *The inverse of an element* $a \in GF(2^m)$ *is given by*

$$a^{-1} = \begin{cases} \left[\alpha_{\frac{m-1}{n}}(a)\right]^2 & \textit{when } n \mid (m-1) \\ \left[(\alpha_q(a))^{2^r} \beta_r(a)\right]^2 & \textit{when } n \nmid (m-1) \end{cases}$$

where $nq + r = m - 1$ *and* n*,* q*, and* $r \in \mathbb{N}$

Proof *When* $n \mid (m-1)$

$$\begin{aligned}
\left[\alpha_{\frac{m-1}{n}}(a)\right]^2 &= \left[a^{2^{n(\frac{m-1}{n})}-1}\right]^2 \\
&= \left[a^{2^{m-1}-1}\right]^2 \\
&= a^{-1}
\end{aligned}$$

When $n \nmid (m-1)$

$$\begin{aligned}
\left[(\alpha_q(a))^{2^r} \beta_r(a)\right]^2 &= \left[(a^{2^{nq}-1})^{2^r} (a^{2^r-1})\right]^2 \\
&= \left[a^{2^{nq+r}-1}\right]^2 \\
&= \left[a^{2^{m-1}-1}\right]^2 \\
&= a^{-1}
\end{aligned}$$

Table 5.2: Comparison of LUTs Required for a Squarer and Quad Circuit for $GF(2^9)$

Output	Squarer Circuit		Quad Circuit	
bit	$b(x)^2$	#LUTs	$b(x)^4$	#LUTs
0	b_0	0	b_0	0
1	b_5	0	b_7	0
2	$b_1 + b_5$	1	$b_5 + b_7$	1
3	b_6	0	$b_3 + b_7$	1
4	$b_2 + b_6$	1	$b_1 + b_3 + b_5 + b_7$	1
5	b_7	0	b_8	0
6	$b_3 + b_8$	1	$b_6 + b_8$	1
7	b_8	0	$b_4 + b_8$	1
8	$b_4 + b_8$	1	$b_2 + b_4 + b_6 + b_8$	1
Total LUTs		4		6

We note that elliptic curves over the field $GF(2^m)$ used for cryptographic purposes [14] have an odd m, therefore we discuss with respect to such values of m, although the results are valid for all m. In particular, we consider the case when $n = 2$; such that

$$\alpha_k(a) = a^{4^k - 1}$$

To implement this we require *quad* circuits. To show the benefits of using a quad circuit on an FPGA instead of the conventional squarer, consider the equations for a squarer and a quad for an element $b(x) \in GF(2^9)$ (Table 5.2). The irreducible polynomial for the field is $x^9 + x + 1$. In the table, $b_0 \cdots b_8$ are the coefficients of $b(x)$. The #LUTs column shows the number of LUTs required for obtaining the particular output bit.

We would expect the LUTs required by the quad circuit be twice that of the squarer. However this is not the case. The quad circuit's LUT requirement is only 1.5 times that of the squarer. This is because the quad circuit has a lower percentage of *under utilized LUTs* (Equation 4.9). For example, from Table 5.2 we note that output bit 4 requires

Table 5.3: Comparison of Squarer and Quad Circuits on Xilinx Virtex 4 FPGA

Field	Squarer Circuit		Quad Circuit		Size ratio
	$\#LUT_s$	Delay (ns)	$\#LUT_q$	Delay (ns)	$\frac{\#LUT_q}{2(\#LUT_s)}$
$GF(2^{193})$	96	1.48	145	1.48	0.75
$GF(2^{233})$	153	1.48	230	1.48	0.75

three XOR gates in the quad circuit and only one in the squarer. However, both circuits require only 1 LUT. This is also the case with output bit 8. This shows that the quad circuit is better at utilizing FPGA resources compared to the squarer. Moreover, both circuits have the same delay of one LUT. If we generate the fourth power by cascading two squarer circuits (i.e $(b(x)^2)^2$), the resulting circuit would have twice the delay and require 25% more hardware resources than a single quad circuit.

These observations are scalable to larger fields as shown in Table 5.3. The circuits for the finite fields $GF(2^{233})$ and $GF(2^{193})$ use the irreducible polynomials $x^{233} + x^{74} + 1$ and $x^{193} + x^{15} + 1$ respectively. They were synthesized for a *Xilinx Virtex 4* FPGA. The table shows that the area saved even for large fields is about 25%. While the combinational delay of a single squarer is equal to that of the quad.

Based on this observation we propose a *quad-ITA* using quad exponentiation circuits instead of squarers. The procedure for obtaining the inverse for an odd m using the quad-ITA is shown in Algorithm 5.1. The algorithm assumes a Brauer addition chain.

The overhead of the quad-ITA is the need to precompute a^3. Since we do not have a squarer this has to be done by the multiplication block, which is present in the architecture. Using the multiplication unit, cubing is accomplished in two clock cycles without any additional hardware requirements. Similarly, the final squaring can be done in one clock cycle by the multiplier with no additional hardware required.

Consider the example of finding the multiplicative inverse of an element $a \in GF(2^{233})$ using the quad-ITA. From Theorem 5.4.2, setting $n = 2$ and $m = 233$, $a^{-1} = [\alpha_{\frac{232}{2}}(a)]^2 =$

Algorithm 5.1: qitmia *(Quad-ITA)*

```
   Input: The element $a \in GF(2^m)$ and the Brauer chain
          $U = \{1, 2, \cdots, \frac{m-1}{2}, m-1\}$
   Output: The multiplicative inverse $a^{-1}$
1  begin
2      $l$ = length$(U)$
3      $a^2$ = hmul$(a, a)$; /* hmul:  hybrid Karatsuba multiplier */
       ;                    /* proposed in Algorithm 4.2 */
4      $\alpha_{u_1} = a^3 = a^2 \cdot a$
5      foreach $u_i \in U (2 \leq i \leq l-1)$ do
6          $p = u_{i-1}$
7          $q = u_i - u_{i-1}$
8          $\alpha_{u_i}$ = hmul$(\alpha_p^{4^q}, \alpha_q)$
9      end
10     $a^{-1}$ = hmul$(\alpha_{u_{l-1}}, \alpha_{u_{1-1}})$
11 end
```

$[\alpha_{116}(a)]^2$. This requires computation of $\alpha_{116}(a) = a^{2^{2.116}-1} = a^{4^{116}-1}$ and then doing a squaring, $a^{-1} = (\alpha_{116}(a))^2$. We use the same Brauer chain (Equation 5.3) as we did in the previous example. Excluding the precomputation step, computing $\alpha_{116}(a)$ requires 9 steps. The total number of quad operations to compute $\alpha_{116}(a)$ is 115 and the number of multiplications is 9. The precomputation step requires 2 clock cycles and the final squaring takes one clock cycle. In all 12 multiplications are required for the inverse operation. In general for an addition chain for $m-1$ of length l, the quad-ITA requires two additional multiplications compared to the ITA implementation of [59].

$$\#Multiplications : l + 1 \tag{5.7}$$

The number of quad operations required is given by

$$\#QuadPowers : \frac{(m-1)}{2} - 1 \tag{5.8}$$

The number of clock cycles required is given by the Equation 5.9. The summation in the equation is the clock cycles required for the quadblock, while $l+1$ is the clock cycles of the multiplier.

$$\#ClockCycles = (l+1) + \sum_{i=2}^{l-1} \lceil \frac{u_i - u_{i-1}}{u_s} \rceil \tag{5.9}$$

The difference in the clock cycles between the ITA of [59] (Equation 5.4) and the quad-ITA (Equation 5.9) is

$$\lceil \frac{u_l - u_{l-1}}{u_s} - 1 \rceil \tag{5.10}$$

In general for addition chains used in ECC, the value of $u_l - u_{l-1}$ is as large as $(m-1)/2$ and much greater than u_s, therefore the clock cycles saved is significant.

5.4.1 Hardware Architecture

To compare the proposed quad-ITA with other reported inverse implementations we develop a dedicated processor (Figure 5.2) that generates the inverse of the input $a \in GF(2^{233})$. Generating the inverse requires the computation of the steps in Table 5.4 followed by a squaring. The main components of the architecture is a finite field multiplier and a quadblock. The multiplier is an implementation of the hybrid Karatsuba

Table 5.4: Inverse of $a \in GF(2^{233})$ using Quad-ITA

	$\alpha_{\mathbf{u_i}}(\mathbf{a})$	$\alpha_{\mathbf{u_j}+\mathbf{u_k}}(\mathbf{a})$	Exponentiation
1	$\alpha_1(a)$		a^3
2	$\alpha_2(a)$	$\alpha_{1+1}(a)$	$(\alpha_1)^{4^1}\alpha_1 = a^{4^2-1}$
3	$\alpha_3(a)$	$\alpha_{2+1}(a)$	$(\alpha_2)^{4^1}\alpha_1 = a^{4^3-1}$
4	$\alpha_6(a)$	$\alpha_{3+3}(a)$	$(\alpha_3)^{4^3}\alpha_3 = a^{4^6-1}$
5	$\alpha_7(a)$	$\alpha_{6+1}(a)$	$(\alpha_6)^{4^1}\alpha_1 = a^{4^7-1}$
6	$\alpha_{14}(a)$	$\alpha_{7+7}(a)$	$(\alpha_7)^{4^7}\alpha_7 = a^{4^{14}-1}$
7	$\alpha_{28}(a)$	$\alpha_{14+14}(a)$	$(\alpha_{14})^{4^{14}}\alpha_{14} = a^{4^{28}-1}$
8	$\alpha_{29}(a)$	$\alpha_{28+1}(a)$	$(\alpha_{28})^{4^1}\alpha_1 = a^{4^{29}-1}$
9	$\alpha_{58}(a)$	$\alpha_{29+29}(a)$	$(\alpha_{29})^{4^{29}}\alpha_{29} = a^{4^{58}-1}$
10	$\alpha_{116}(a)$	$\alpha_{58+58}(a)$	$(\alpha_{58})^{4^{58}}\alpha_{58} = a^{4^{116}-1}$

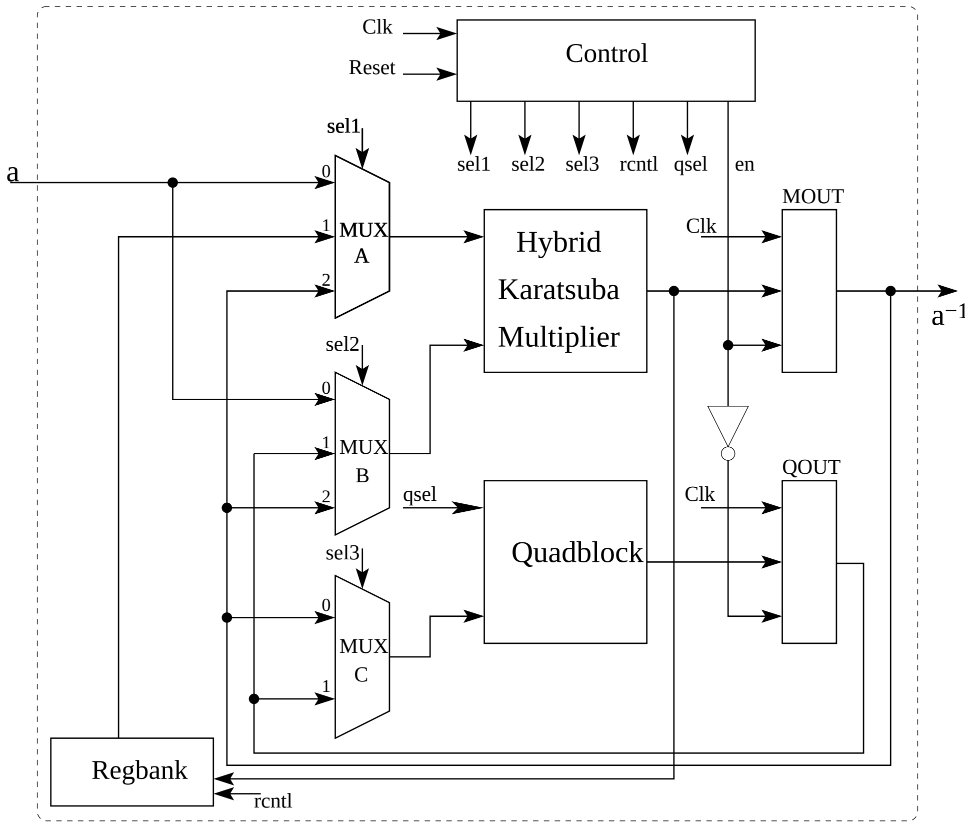

Fig. 5.2: Quad-ITA Architecture for $GF(2^{233})$ with the Addition Chain 5.3

algorithm (Section 4.5.1). The quadblock (Figure 5.3) consists of 14 cascaded circuits, each circuit generating the fourth power of its input. If qin is the input to the quad-

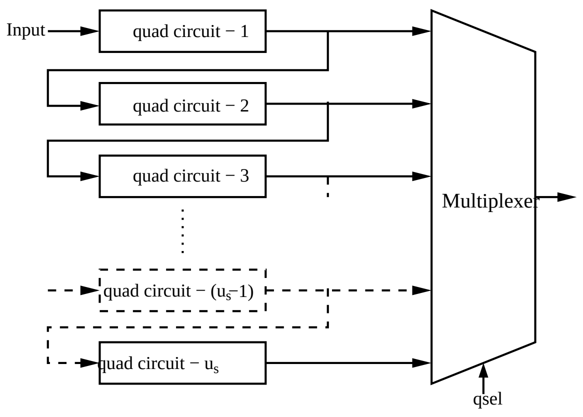

Fig. 5.3: Quadblock Design: Raises the Input to the Power of 4^k

block, the powers of qin generated are qin^4, qin^{4^2}, qin^{4^3} $\cdots$ $qin^{4^{14}}$. A multiplexer in the quadblock, controlled by the select lines $qsel$, determines which of the 14 powers gets passed on to the output. The output of the quadblock can be represented as $qin^{4^{qsel}}$.

Two buffers $MOUT$ and $QOUT$ store the output of the multiplier and the quadblock respectively. At every clock cycle, either the multiplier or the quadblock (but not both) is active (The en signal if 1 enables either the $MOUT$, otherwise the $QOUT$ buffer). A register bank may be used to store results of each step (α_{u_i}) of Algorithm 5.1. A result is stored only if it is required for later computations.

The controller is a state machine designed based on the adder chain and the number of cascaded quad circuits in the quadblock. At every clock cycle, control signals are generated for the multiplexer selection lines, enables to the buffers and access signals to the register bank. As an example, consider the computations of Table 5.4. The corresponding control signals generated by the controller is as shown in Table 5.5. The first step in the computation of a^{-1} is the determination of a^3. This takes two clock cycles. In the first clock, a is fed to both inputs of the multiplier. This is done by controlling the appropriate select lines of the multiplexers. The result, a^2, is used in the following clock along with a to produce a^3. This is stored in the register bank. The second step is the computation of $\alpha_2(a)$. This too requires two clock cycles. The first clock uses a^3 as the input to the quadblock to compute $(\alpha_1)^{4^1}$. In the next clock, this is multiplied with a^3 to produce the required output. In general, computing any step $\alpha_{u_i}(a) = \alpha_{u_j+u_k}(a)$ takes $1 + \lceil \frac{u_j}{14} \rceil$ clock cycles. Of this, $\lceil \frac{u_j}{14} \rceil$ clock cycles are used by the quadblock, while the multiplier requires a single clock cycle. At the end of a step, the result is present in $MOUT$.

Addition Chain Selection Criteria

The length of the addition chain influences the number of clock cycles required to compute the inverse (Equations 5.4 and 5.9), hence proper selection of the addition chain is

Table 5.5: Control Word for $GF(2^{233})$ Quad-ITA for Table 5.4

Step	Clock	sel1	sel2	sel3	qsel	en
$\alpha_1(a)$	1	0	0	$\times$	$\times$	1
	2	0	2	$\times$	$\times$	1
$\alpha_2(a)$	3	$\times$	$\times$	0	1	0
	4	1	1	$\times$	$\times$	1
$\alpha_3(a)$	5	$\times$	$\times$	0	1	0
	6	1	1	$\times$	$\times$	1
$\alpha_6(a)$	7	$\times$	$\times$	0	3	0
	8	2	1	$\times$	$\times$	1
$\alpha_7(a)$	9	$\times$	$\times$	0	1	0
	10	1	1	$\times$	$\times$	1
$\alpha_{14}(a)$	11	$\times$	$\times$	0	7	0
	12	2	1	$\times$	$\times$	1
$\alpha_{28}(a)$	13	$\times$	$\times$	0	14	0
	14	2	1	$\times$	$\times$	1
$\alpha_{29}(a)$	15	$\times$	$\times$	0	1	0
	16	1	1	$\times$	$\times$	1
$\alpha_{58}(a)$	17	$\times$	$\times$	0	14	0
	18	$\times$	$\times$	1	14	0
	19	$\times$	$\times$	1	1	0
	20	2	1	$\times$	$\times$	1
$\alpha_{116}(a)$	21	$\times$	$\times$	0	14	0
	22	$\times$	$\times$	1	14	0
	23	$\times$	$\times$	1	14	0
	24	$\times$	$\times$	1	14	0
	25	$\times$	$\times$	1	2	0
	26	2	1	$\times$	$\times$	1
$FinalSquare$	27	2	2	$\times$	$\times$	1

critical to the design. For a given m, there could be several optimal addition chains. It is required to select one chain from available optimal chains. The amount of memory required by the addition chain can be used as a secondary selection criteria. The memory utilized by an addition chain is the registers required for storage of the results from intermediate steps. The result of step $\alpha_i(a)$ is stored only if it is required to be used in any other step $\alpha_j(a)$ and $j > i + 1$. Consider the addition chain in 5.11.

$$U_2 = \begin{pmatrix} 1 & 2 & 3 & 5 & 6 & 12 & 17 & 29 & 58 & 116 & 232 \end{pmatrix} \tag{5.11}$$

Computing $\alpha_5(a) = \alpha_{2+3}(a)$ requires $\alpha_2(a)$, therefore $\alpha_2(a)$ needs to be stored. Similarly, $\alpha_1(a)$, $\alpha_5(a)$ and $\alpha_{12}(a)$ needs to be stored to compute $\alpha_3(a)$, $\alpha_{17}(a)$ and $\alpha_{29}(a)$ respectively. In all four registers are required. Minimizing the number of registers is important because for cryptographic applications m is generally large therefore each register's size is significant.

Using Brauer chains has the advantage that for every step (except the first) at least one input is read from the output of the previous step. The output of the previous step is stored in $MOUT$ therefore need not be read from any register and no storage is required. The second input to the step would ideally be a doubling. For example, computing $\alpha_{116}(a)$ requires only $\alpha_{58}(a)$. Since $\alpha_{58}(a)$ is the result from the previous step, it is stored in $MOUT$. Therefore computing $\alpha_{116}(a)$ does not require any stored values.

Design of the Quadblock

The number of quad circuits cascaded (u_s) has an influence on the clock cycles, frequency, and area requirements of the quad-ITA. Increasing the number of cascaded blocks would reduce the number of clock cycles (Equation 5.4) required at the cost of an increase in area and delay.

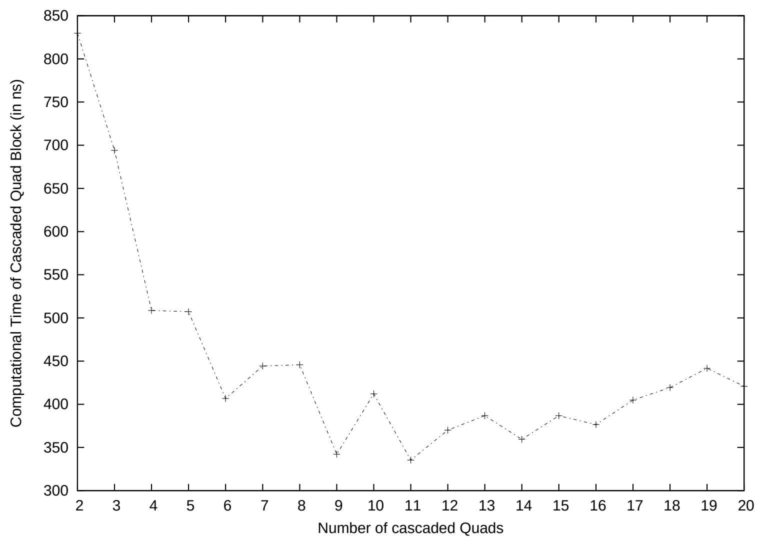

Fig. 5.4: Clock Cycles of Computation Time versus Number of Quads in Quadblock on a Xilinx Virtex 4 FPGA for $GF(2^{233})$

Let a single quad circuit require l_p LUTs and have a combinational delay of t_p. For this analysis we assume that t_p includes the gate delay as well as the path delay. We also assume that the path delay is constant. The values of l_p and t_p depend on the finite field $GF(2^m)$ and the irreducible polynomial. A cascade of u_s quad circuits would require $u_s \cdot l_p$ LUTs and have a delay of $u_s \cdot t_p$.

In order that the quadblock not alter the frequency of operation, u_s should be selected such that $u_s \cdot t_p$ is less than the maximum combinational delay of the entire design. In the quad-ITA hardware, the maximum delay is from the Karatsuba multiplier, therefore we select u_s such that the delay of the quadblock is less than the delay of the multiplier.

$$u_s \cdot t_p \leq \text{Delay of multiplier}$$

However, reducing u_s would increase the clock cycles required. Therefore we select u_s so that the quadblock delay is close to the multiplier delay.

The graph in Figure 5.4 plots the computation delay (clock period in nanoseconds $\times$ the clock cycles) required versus the number of quads in the quad-ITA for the field

$GF(2^{233})$. For small values of u_s, the delay is mainly decided by the multiplier, while the clock cycles required is large. For large number of cascades, the delay of the quad-block exceeds that of the multiplier, therefore the delay of the circuit is now decided by the quadblock. Lowest computation time is obtained with around 11 cascaded quads. For this, the delay of the quadblock is slightly lower than the multiplier. Therefore, the critical delay is the path through the multiplier, while the clock cycles required is around 30. Therefore for the quad-ITA in a field $GF(2^{233})$, 11 cascaded quads result in least computation time. However, in order to make the clock cycles required to compute the finite field inverse in $GF(2^{233})$ equal to the parallel implementation of [59], 14 cascaded quads are used even though this causes a marginal increase in the computation time (which is still quite lesser than the parallel implementation at $0.55\mu sec$).

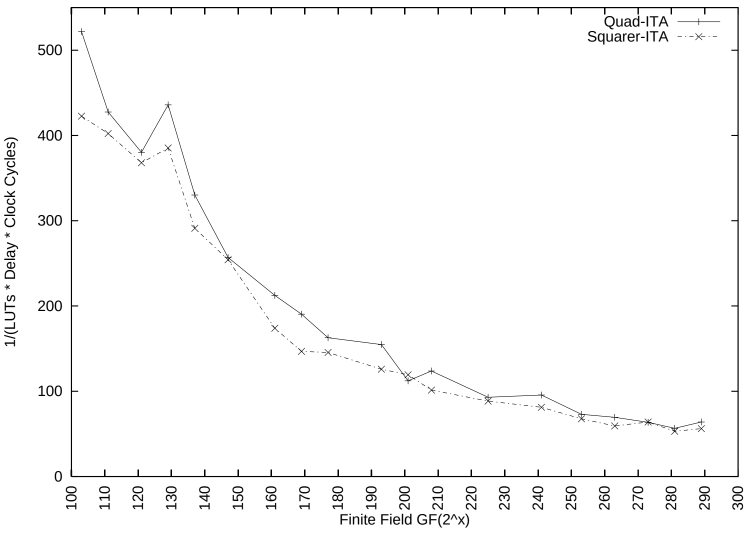

Fig. 5.5: Performance of Quad-ITA vs Squarer-ITA Implementation for Different Fields on a Xilinx Virtex 4 FPGA

5.5 Experimental Results

In this section we compare our work with reported finite field inverse results. We also test our design for scalability over several fields.

The graph in Figure 5.5 shows the scalability of the quad-ITA and compares it with a squarer-ITA. The design of the squarer-ITA is similar to that of the quad-ITA (Figure 5.2) except for the quadblock. The quad circuits in the quadblock is replaced by squarer circuits. Both the quadblock and squarer block have the same number of cascaded circuits. The platform used for generating the graph is a *Xilinx Virtex 4 FPGA*. The X axis has increasing field sizes (see the Appendix for list of finite fields), and the Y axis has the performance metric shown below.

$$Performance = \frac{frequency}{Slices \times ClockCycles} \tag{5.12}$$

The $slices$ is the number of slices required on the FPGA as reported by Xilinx's ISE synthesis tool. The graph shows that the quad-ITA has better performance compared to the squarer-ITA for most fields.

Table 5.6 compares the quad-ITA with the best reported ITA and Montgomery inverse algorithms available. The FPGA used in all designs is the *Xilinx Virtex E*. The quad-ITA has the best computation time and performance compared to the other implementations. It may be noted that the larger area compared to [58] and [59] of the quad-ITA is because it uses distributed RAM [61] for registers, while [58] and [59] use block RAM [39]. The distributed RAM requires additional CLB resources while block RAM does not.

Table 5.6: Comparison for Inversion on Xilinx Virtex E

Implementation	Algorithm	Platform	Field	Slices	Frequency (MHz) (f)	Clock Cycle (c)	Computation Time (c/f)	Performance (Equation 5.12)
Dormale [62]	Montgomery	XCV2000E	160	890	50	-	$9.71\mu sec$	115.7
		XCV2000E	256	1390	41	-	$18.7\mu sec$	38.4
Crowe [63]	Montgomery	XCV2000E	160	1094	51	-	$6.28\mu sec$	145.5
		XCV2000E	256	1722	39	-	$13.17\mu sec$	44.1
Henriquez [58]	ITA	XCV3200E	193	10065	21.2	27	$1.33\mu sec$	78
Henriquez [59]	Parallel ITA	XCV3200E	193	11081	21.2	20	$0.94\mu sec$	95.7
This work	quad-ITA	XCV3200E	193	11911	36.2	20	$0.55\mu sec$	152.1

5.6 Conclusion

This chapter discussed the finite field inverter required for the elliptic curve crypto processor. The Itoh-Tsujii algorithm was used for the inversion. A generalized version of the ITA was proposed that improves the utilization of FPGA resources. With this method, we show that raising an element by a power of 4 (quad operation) on an FPGA is more compact and faster than using squarers. Thus the quad operation forms the core of an improved ITA algorithm called the quad-ITA. The quad-ITA takes least number of clock cycles, has lesser computational time and has better performance compared to the best reported inversion algorithms. The quad-ITA is used for the final inversion required in the elliptic curve crypto processor. This is discussed in the next chapter.

CHAPTER 6

Constructing the Elliptic Curve Crypto Processor

This chapter presents the construction of an *elliptic curve crypto processor* (ECCP) for the NIST specified curve [14] given in Equation 6.1 over the binary finite field $GF(2^{233})$.

$$y^2 + xy = x^3 + ax^2 + b \tag{6.1}$$

The processor implements the double and add scalar multiplication algorithm described in Algorithm 3.1. The processor (Figure 6.1), is capable of doing the elliptic curve operations of point addition and point doubling. Point doubling is done at every iteration of the loop in Algorithm 3.1, while point addition is done for every bit set to one in the binary expansion of the scalar input k. The output produced as a result of the scalar

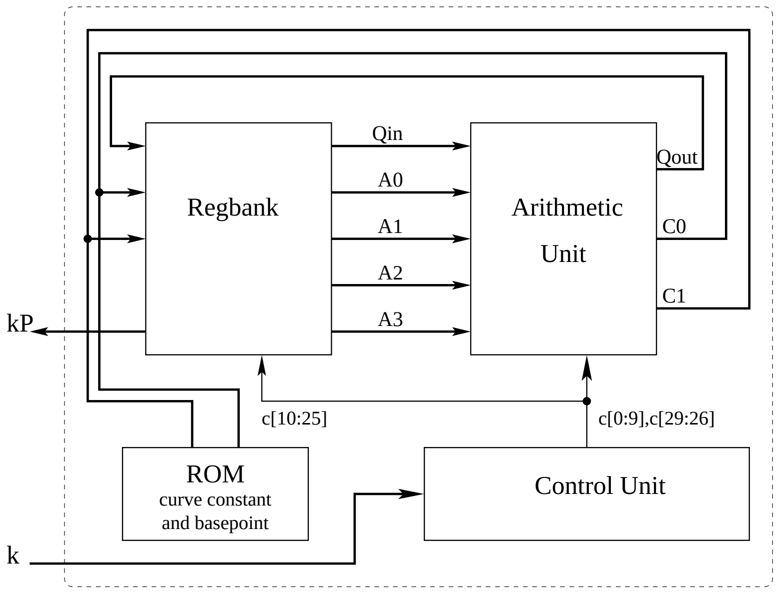

Fig. 6.1: Block Diagram of the Elliptic Curve Crypto Processor

multiplication is the product kP. Here, P is the basepoint of the curve and is stored in the ROM in its affine form. At every clock cycle, the register bank (*regbank*) containing dual ported registers feed the *arithmetic unit* (AU) through five buses ($A0$, $A1$, $A2$, $A3$ and Qin). At the end of the clock cycle, results of the computation are stored in registers through buses $C0$, $C1$ and $Qout$. There can be at most two results produced at every clock. Control signals ($c[0] \cdots c[32]$) generated every clock cycle depending on the elliptic curve operation control the data flow and the computation done. Details about the processor, the flow of data on the buses, the computations done etc. are elaborated in following sections.

The scalar multiplication implemented in the processor of Figure 6.1 is done using the *López-Dahab (LD) projective coordinate* system. The LD coordinate form of the elliptic curve over binary finite fields is

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^4 \tag{6.2}$$

In the ECCP, a is taken as 1, while b is stored in the ROM along with the basepoint P. Equations for point doubling and point addition in LD coordinates are shown in Equations 3.10 and 3.11 respectively.

During the initialization phase the curve constant b and the basepoint P are loaded from the ROM into the registers after which there are two computational phases. The first phase multiplies the scalar k to the basepoint P. The result produced by this phase is in projective coordinates. The second phase of the computation converts the projective point result of the first phase into the affine point kP. The second phase mainly involves an inverse computation. The inverse is computed using the *quad Itoh-Tsujii inverse algorithm* proposed in Algorithm 5.1.

The next section describes in detail the ECCP. Section 6.2 describes the implementation of the elliptic curve operations in the processor. Section 6.3 presents the finite state machine that implements Algorithm 3.1. Section 6.4 has the performance results,

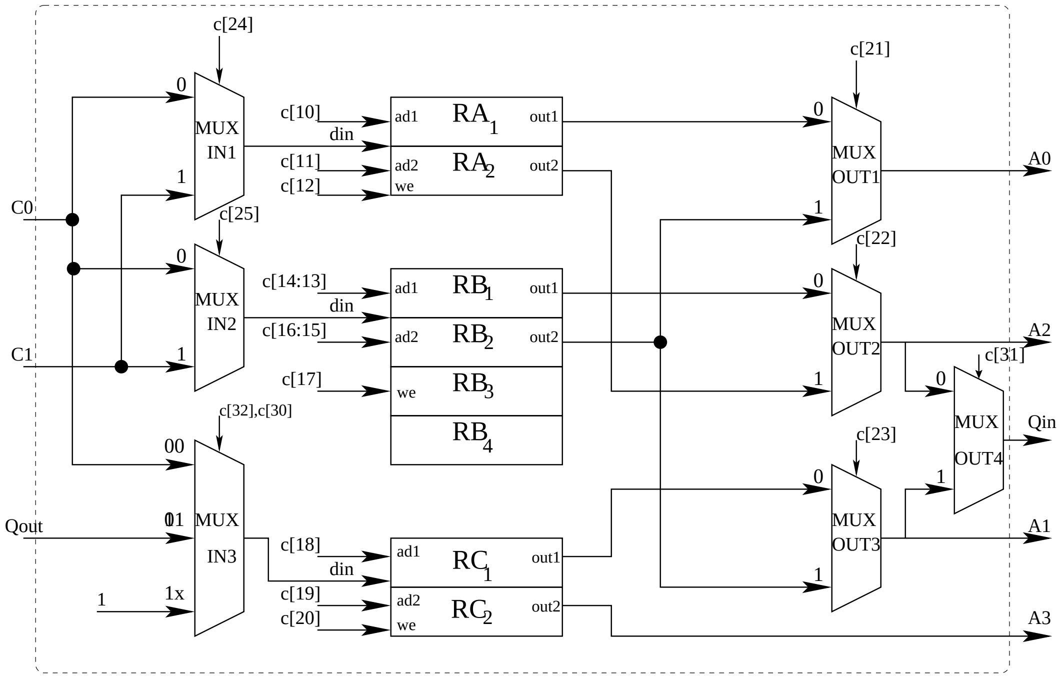

Fig. 6.2: Register File for Elliptic Curve Crypto Processor

while the final section has the conclusion.

6.1 The Elliptic Curve Cryptoprocessor

This section describes in detail the register file, arithmetic unit and the control unit of the elliptic curve crypto processor.

6.1.1 Register Bank

The heart of the register file (Figure 6.2) are eight registers each of size 233 bits. The registers are used to store the results of the computations done at every clock cycle. The registers are dual ported and arranged in three banks, RA, RB, and RC. The dual ported RAM allows asynchronous reads on the lines *out1* and *out2* corresponding to the

Table 6.1: Utility of Registers in the Register Bank

Register	Description
RA_1	1. During initialization it is loaded with P_x.
	2. Stores the x coordinate of the result.
	3. Also used for temporary storage.
RA_2	Stores P_x.
RB_1	1. During initialization it is loaded with P_y.
	2. Stores the y coordinate of the result.
	3. Also used for temporary storage.
RB_2	Stores P_y.
RB_3	Used for temporary storage.
RB_4	Stores the curve constant b.
RC_1	1. During initialization it is set to 1.
	2. Store z coordinate of the projective result.
	3. Also used for temporary storage.
RC_2	Used for temporary storage.

address on the address lines *ad1* and *ad2* respectively. A synchronous write of the data on *din* is done to the location addressed by *ad1*. The *we* signal enables the write. On the FPGA, the registers are implemented as distributed RAM[61]. At every clock cycle, the register file is capable of delivering five operands (on buses $A0$, $A1$, $A2$, $A3$ and Qin) to the arithmetic unit and able to store three results (from buses $C0$, $C1$, and $Qout$). The inputs to the register file is either the arithmetic unit outputs, the curve constant (b of Equation 6.2), or the basepoint $P = (P_x, P_y)$.

Multiplexers $MUXIN1$, $MUXIN2$, and $MUXIN3$ determine which of the three inputs gets stored into the register banks. Further, bits in the control word select a register, or enable or disable a write operation to a particular register bank. Multiplexers $MUXOUT1$, $MUXOUT2$, $MUXOUT3$, and $MUXOUT4$ determine which output of a register bank get driven on the output buses. Table 6.1 shows how the each register in the bank is utilized.

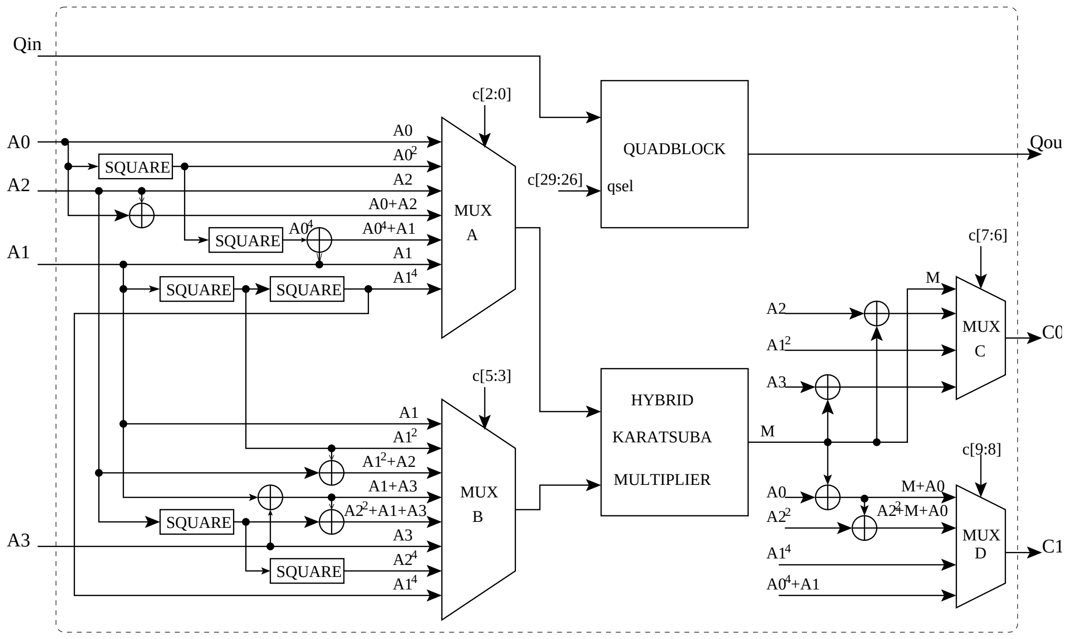

Fig. 6.3: Finite Field Arithmetic Unit

6.1.2 Finite Field Arithmetic Unit

The arithmetic unit (Figure 6.3) is built using finite field arithmetic circuits and organized for efficient implementation of point addition (Equation 3.11) and point doubling (Equation 3.10) in LD coordinates. The AU has 5 inputs ($A0$ to $A3$ and Qin) and 3 outputs ($C0$, $C1$, and $Qout$). The main components of the AU is a quadblock and a multiplier. The multiplier is based on the *hybrid Karatsuba algorithm* (Section 4.5.1). It is used in both phases (during the scalar multiplication phase and conversion to affine coordinate phase) of the computation. The *quadblock* is designed according to Figure 5.3. Here, the quadblock consists of 14 cascaded quad circuits and is capable of generating the output $Qout = Qin^{4^{c[29]\cdots c[26]}}$. The quadblock is used only for inversion which is done during the final phase of the computation. The AU has several adders and squarer circuits. These circuits are small compared to the multiplier and the quadblock and therefore contribute marginally to the overall area and latency of the processor.

6.1.3 Control Unit

At every clock cycle the control unit produces a control word. Control words are produced in a sequence depending on the type of elliptic curve operation being done. The control word signals control the flow of data and also decide the operations performed on the data. There are 33 control signals ($c[0]$ to $c[32]$) that are generated by the control unit. The signals $c[0]$ to $c[9]$ control the inputs to the finite field multiplier and the outputs $C0$ and $C1$ of the AU. The control lines $c[26]$ to $c[29]$ are used for the select lines to the multiplexer in the quadblock (Figure 5.3). The remaining control bits are used in the register file to read and write data to the registers. Section 6.3 has the detailed list of all control words generated.

6.2 Point Arithmetic on the ECCP

This section presents the implementation of LD point addition and doubling equations on the ECCP.

6.2.1 Point Doubling

The equation for doubling the point P in LD projective coordinates was shown in Equation 3.10 and is repeated here (Equation 6.3). [30]. The input required for doubling is the point $P = (X_1, Y_1, Z_1)$ and the output is its double $2P = (X_3, Y_3, Z_3)$. The equation shows that four multiplications are required (assuming $a = 1$). The ECCP has just one multiplier, which is capable of doing one multiplication per clock cycle. Hence, the

ECCP would require at least four clock cycles for computing the double.

$$\begin{aligned} Z_3 &= X_1^2 \cdot Z_1^2 \\ X_3 &= X_1^4 + b \cdot Z_1^4 \\ Y_3 &= b \cdot Z_1^4 \cdot Z_3 + X_3 \cdot (a \cdot Z_3 + Y_1^2 + b \cdot Z_1^4) \end{aligned} \tag{6.3}$$

This doubling operation is mapped to the elliptic curve hardware using Algorithm 6.1.

Algorithm 6.1: Hardware Implementation of Doubling on ECCP

```
**Input**: LD Point P=($X_1, Y_1, Z_1$) present in registers ($RA_1, RB_1, RC_1$)
        respectively. The curve constant $b$ is present in register $RB_4$
**Output**: LD Point 2P=($X_3, Y_3, Z_3$) present in registers ($RA_1, RB_1, RC_1$)
        respectively.
1 $RB_3 = RB_4 \cdot RC_1^4$
2 $RC_1 = RA_1^2 \cdot RC_1^2$
3 $RA_1 = RA_1^4 + RB_3$
4 $RB_1 = RB_3 \cdot RC_1 + RA_1 \cdot (RC_1 + RB_1^2 + RB_3)$
```

Table 6.2: Parallel LD Point Doubling on the ECCP

Clock	Operation 1 ($C0$)	Operation 2($C1$)
1	$RC_1 = RA_1^2 \cdot RC_1^2$	$RB_3 = RC_1^4$
2	$RB_3 = RB_3 \cdot RB_4$	
3	$RC_2 = (RA_1^4 + RB_3) \cdot (RC_1 + RB_1^2 + RB_3)$	$RA_1 = (RA_1^4 + RB_3)$
4	$RB_1 = RB_3 \cdot RC_1 + RC_2$	

Table 6.3: Inputs and Outputs of the Register File for Point Doubling

Clock	$A0$	$A1$	$A2$	$A3$	$C0$	$C1$
1	RA_1	RC_1	-	-	RC_1	RB_3
2	-	RB_4	RB_3	-	RB_3	
3	RA_1	RB_3	RB_1	RC_1	RC_2	RA_1
4	RB_3	RC_1	-	RC_2	RB_1	-

On the ECCP, the LD doubling algorithm can be parallelized to complete in four clock cycles as shown in Table 6.2 [64]. The parallelization is based on the fact that the multiplier is several times more complex than the squarer and adder circuits used. So, in every clock cycle the multiplier is used and it produces one of the outputs of the AU. The other AU output is produced by additions or squaring operations alone.

Table 6.3 shows the data held on the buses at every clock cycle. It also shows where the results are stored. For example, in clock cycle 1, the contents of the registers RA_1 and RC_1 are placed on the bus $A0$ and $A1$ respectively. Control lines in $MUXA$ and $MUXB$ of the AU are set such that $A0^2$ and $A1$ are fed to the multiplier. The output multiplexers $MUXC$ and $MUXD$ are set such that M and $A1^4$ are sent on the buses $C0$ and $C1$. These are stored in registers RC_1 and RB_3 respectively. Effectively, the computation done by the AU are $RC_1 = RA_1^2 \cdot RC_1^2$ and $RB_3 = RC_1^4$. Similarly the subsequent operations required for doubling as stated in 6.2 are performed.

$$
\begin{aligned}
A &= y_2 \cdot Z_1^2 + Y_1 \\
B &= x_2 \cdot Z_1 + X_1 \\
C &= Z_1 \cdot B \\
D &= B^2 \cdot (C + a \cdot Z_1^2) \\
Z_3 &= C^2 \\
E &= A \cdot C \\
X_3 &= A^2 + D + E \\
F &= X_3 + x_2 \cdot Z_3 \\
G &= (x_2 + y_2) \cdot Z_3^2 \\
Y_3 &= (E + Z_3) \cdot F + G
\end{aligned}
\tag{6.4}
$$

6.2.2 Point Addition

The equation for adding an affine point to a point in LD projective coordinates was shown in Equation 3.11 and repeated here in Equation 6.4. The equation adds two points $P = (X_1, Y_1, Z_1)$ and $Q = (x_2, y_2)$ where $Q \neq \pm P$. The resulting point is $P + Q = (X_3, Y_3, Z_3)$.

Algorithm 6.2: Hardware Implementation of Addition on ECCP

```
Input: LD Point P=($X_1, Y_1, Z_1$) present in registers ($RA_1, RB_1, RC_1$)
       respectively and Affine Point Q=($x_2, y_2$) present in registers ($RA_2, RB_2$)
       respectively
Output: LD Point P+Q=($X_3, Y_3, Z_3$) present in registers ($RA_1, RB_1, RC_1$)
        respectively
1  $RB_1 = RB_2 \cdot RC_1^2 + RB_1$ ;                 /* $A$ */
2  $RA_1 = RA_2 \cdot RC_1 + RA_1$ ;                   /* $B$ */
3  $RB_3 = RC_1 \cdot RA_1$ ;                          /* $C$ */
4  $RA_1 = RA_1^2 \cdot (RB_3 + RC_1^2)$ ;             /* $D$ */
5  $RC_1 = RB_3^2$ ;                                   /* $Z_3$ */
6  $RC_2 = RB_1 \cdot RB_3$ ;                          /* $E$ */
7  $RA_1 = RB_1^2 + RA_1 + RC_2$ ;                     /* $X_3$ */
8  $RB_3 = RA_1 + RA_2 \cdot RC_1^2$ ;                 /* $F$ */
9  $RB_1 = (RA_2 + RB_2) \cdot RC_1^2$ ;               /* $G$ */
10 $RB_1 = (RC_2 + RC_1) \cdot RB_3 + RB_1$ ;          /* $Y_3$ */
```

Table 6.4: Parallel LD Point Addition on the ECCP

Clock	Operation 1 (C_0)	Operation 2(C_1)
1	$RB_1 = RB_2 \cdot RC_1^2 + RB_1$	-
2	$RA_1 = RA_2 \cdot RC_1 + RA_1$	-
3	$RB_3 = RC_1 \cdot RA_1$	-
4	$RA_1 = RA_1^2 \cdot (RB_3 + RC_1^2)$	-
5	$RC_2 = RB_1 \cdot RB_3$	$RA_1 = RB_1^2 + RA_1 + RB_1 \cdot RB_3$
6	$RC_1 = RB_3^2$	$RB_3 = RA_1 + RA_2 \cdot RB_3^2$
7	$RB_1 = (RA_2 + RB_2) \cdot RC_1^2$	-
8	$RB_1 = (RC_2 + RC_1) \cdot RB_3 + RB_1$	-

The addition operation is mapped to the elliptic curve hardware using Algorithm 6.2. Note, a is taken as 1. On the ECCP the operations in Algorithm 6.2 are scheduled

Table 6.5: Inputs and Outputs of the Register Bank for Point Addition

Clock	$A0$	$A1$	$A2$	$A3$	$C0$	$C1$
1	RB_2	RC_1	RB_1	-	RB_1	-
2	RA_1	RC_1	RA_2	-	RA_1	-
3	RA_1	-	-	RC_1	RB_3	-
4	RA_1	RC_1	RB_3	-	RA_1	-
5	RA_1	RB_3	RB_1	-	RC_2	RA_1
6	RA_1	RB_3	RA_2	-	RC_1	RB_3
7	RB_2	RC_1	RA_2	-	RB_1	-
8	RB_3	RC_1	RB_1	RC_2	RB_1	-

efficiently to complete in eight clock cycles [64]. The scheduled operations for point addition is shown in Table 6.4, and the inputs and outputs of the registers at each clock cycle is shown in Table 6.5.

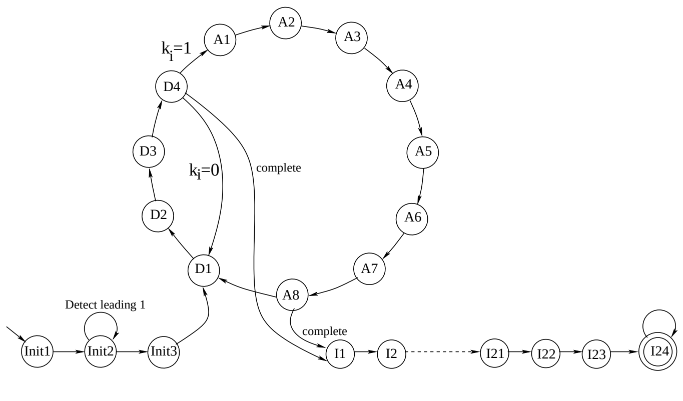

Fig. 6.4: The ECCP Finite State Machine

Table 6.6: Inputs and Outputs of Regbank for Every State

State	Regbank Outputs					Regbank Inputs
	$A0$	$A1$	$A2$	$A3$	Qin	
Init1	-	-	-	-	-	$\mathbf{C0}$:$RA_1 = P_x$; $\mathbf{C1}$:$RB_1 = P_y$; $RC_1 = 1$
Init2	-	-	-	-	-	$\mathbf{C0}$:$RA_2 = P_x$; $\mathbf{C1}$:$RB_2 = P_y$
Init3	-	-	-	-	-	$\mathbf{C1}$:$RB_4 = b$
D1	RA_1	RC_1	-	-	-	$\mathbf{C0}$:$RC_1 = RA_1^2 \cdot RC_1^2$; $\mathbf{C1}$:$RB_3 = RC_1^4$
D2	-	RB_4	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RB_3 \cdot RB_4$
D3	RA_1	RB_3	RB_1	RC_1	-	$\mathbf{C0}$:$RC_2 = (RA_1^4 + RB_3) \cdot (RC_1 + RB_1^2 + RB_3)$;
						$\mathbf{C1}$:$RA_1 = (RA_1^4 + RB_3)$
D4	RB_3	RC_1	-	RC_2	-	$\mathbf{C0}$:$RB_1 = RB_3 \cdot RC_1 + RC_2$
A1	RB_2	RC_1	RB_1	-	-	$\mathbf{C0}$:$RB_1 = RB_2 \cdot RC_1^2 + RB_1$
A2	RA_1	RC_1	RA_2	-	-	$\mathbf{C0}$:$RA_1 = RA_2 \cdot RC_1 + RA_1$
A3	RA_1	-	-	RC_1	-	$\mathbf{C0}$:$RB_3 = RC_1 \cdot RA_1$
A4	RA_1	RC_1	RB_3	-	-	$\mathbf{C0}$:$RA_1 = RA_1^2 \cdot (RB_3 + RC_1^2)$
A5	RA_1	RB_3	RB_1	-	-	$\mathbf{C0}$:$RC_2 = RB_1 \cdot RB_3$
						$\mathbf{C1}$:$RA_1 = RB_1^2 + RA_1 + RB_1 \cdot RB_3$
A6	RA_1	RB_3	RA_2	-	-	$\mathbf{C0}$:$RC_1 = RB_3^2$; $\mathbf{C1}$:$RB_3 = RA_1 + RA_2 \cdot RB_3^2$
A7	RB_2	RC_1	RA_2	-	-	$\mathbf{C0}$:$RB_1 = (RA_2 + RB_2) \cdot RC_1^2$
A8	RB_3	RC_1	RB_1	RC_2	-	$\mathbf{C0}$:$RB_1 = (RC_2 + RC_1) \cdot RB_3 + RB_1$
I1	-	RC_1	-	-	-	$\mathbf{C0}$:$RC_1 = RC_1^2 \cdot RC_1$
I2	-	RC_1	-	-	-	$\mathbf{C0}$:$RB_3 = RC_1^4 \cdot RC_1$
I3	-	RC_1	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RB_3^4 \cdot RC_1$
I4	-	-	-	-	RB_3	$\mathbf{Qout}$:$RC_2 = RB_3^3$
I5	-	RC_2	RB_3	-	-	$\mathbf{CO}$:$RB_3 = RC_2 \cdot RB_3$
I6	-	RC_1	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RB_3^4 \cdot RC_1$
I7	-	-	-	-	RB_3	$\mathbf{Qout}$:$RC_2 = RB_3^7$
I8	-	RC_2	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RC_2 \cdot RB_3$
I9	-	-	-	-	RB_3	$\mathbf{Qout}$:$RC_2 = RB_3^{14}$
I10	-	RC_2	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RC_2 \cdot RB_3$
I11	-	RC_1	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RB_3^4 \cdot RC_1$
I12	-	-	-	-	RB_3	$\mathbf{Qout}$:$RC_2 = RB_3^{14}$
I13	-	-	-	-	RC_2	$\mathbf{Qout}$:$RC_2 = RC_2^{14}$
I14	-	RC_2	RB_3	-	-	$\mathbf{CO}$:$RB_3 = RC_2^4 \cdot RB_3$
I15	-	-	-	-	RB_3	$\mathbf{Qout}$:$RC_2 = RB_3^{14}$
I16	-	-	-	-	RC_2	$\mathbf{Qout}$:$RC_2 = RC_2^{14}$
I17	-	-	-	-	RC_2	$\mathbf{Qout}$:$RC_2 = RC_2^{14}$
I18	-	-	-	-	RC_2	$\mathbf{Qout}$:$RC_2 = RC_2^{14}$
I19	-	-	-	-	RC_2	$\mathbf{Qout}$:$RC_2 = RC_2^2$
I20	-	RC_2	RB_3	-	-	$\mathbf{C0}$:$RB_3 = RC_2 \cdot RB_3$
I21	-	RB_3	-	-	-	$\mathbf{C0}$:$RC_1 = RB_3^2$
I22	RA_1	RC_1	-	-	-	$\mathbf{C0}$:$RA_1 = RA_1 \cdot RC_1$
I23	RB_1	RC_1	-	-	-	$\mathbf{C0}$:$RB_1 = RB_1 \cdot RC_1^2$

Table 6.7: Control Words for ECCP

State	Quadblock	Regfile MUXIN	Regfile MUXOUT	Regbank signals	AU Mux C and D	AU Mux A and B
	$c_{29} \cdots c_{26}$	$c_{32}c_{30}c_{25}c_{24}$	$c_{31}c_{23}c_{22}c_{21}$	$c_{20} \cdots c_{10}$	$c_9 \cdots c_6$	$c_5 \cdots c_0$
Init1	x x x x	1 0 1 0	0 0 x x	1 x 0 1 x x 0 0 1 x 0	0 0 0 0	0 0 0 0 0 0
Init2	x x x x	1 0 1 0	0 0 x x	0 x x 1 x x 0 1 1 x 1	x x x x	x x x x x x
Init3	x x x x	1 x x x	x x x x	0 x x 1 x x 1 1 0 x x	x x x x	x x x x x x
D1	x x x x	0 0 1 x	0 0 x 0	1 x 0 1 x x 1 0 0 x 0	1 0 0 0	0 0 1 0 0 1
D2	x x x x	0 0 0 x	x 1 0 x	0 x x 1 1 1 1 0 0 x x	x x 0 0	0 0 0 0 1 0
D3	x x x x	0 0 x 1	0 1 0 0	1 0 1 0 1 0 0 0 1 x 0	1 1 0 0	1 0 0 1 0 0
D4	x x x x	0 0 0 x	0 0 x 1	0 1 0 1 1 0 0 0 0 x x	x x 1 1	0 0 0 0 0 0
A1	x x x x	0 0 0 x	0 0 0 1	0 x 0 1 0 1 0 0 0 x x	x x 0 1	0 0 1 0 0 0
A2	x x x x	0 0 x 1	0 0 1 0	0 x 0 0 x x 0 0 1 1 0	0 0 x x	0 0 0 0 1 0
A3	x x x x	0 0 x x	0 0 x 0	0 0 x 1 x x 1 0 0 x 0	x x 0 0	1 0 1 0 0 0
A4	x x x x	0 0 x 0	0 0 0 0	0 1 0 0 x x 1 0 1 x 0	x x 0 0	0 1 0 0 0 1
A5	x x x x	0 0 x 1	0 1 0 0	1 x 1 0 1 0 0 0 1 x 0	0 1 0 0	0 0 0 0 1 0
A6	x x x x	0 0 1 x	0 1 1 0	1 x 0 1 1 0 1 0 0 1 0	0 0 1 0	0 0 1 0 1 0
A7	x x x x	0 0 0 x	0 0 1 1	0 x 0 1 0 1 0 0 0 1 x	x x 0 0	0 0 1 0 1 1
A8	x x x x	0 0 0 x	0 0 0 1	0 1 0 1 1 0 0 0 0 x x	x x 0 1	0 1 1 0 0 0
I1	x x x x	0 0 x x	0 0 x x	1 x 0 x x x x x 0 x x	x x 0 0	0 0 1 1 0 1
I2	x x x x	0 0 0 x	0 0 0 x	0 x 0 1 x x 1 0 0 x x	x x 0 0	0 0 0 1 1 0
I3	x x x x	0 0 0 x	0 0 0 x	x x 0 1 x x 1 0 0 x x	x x 0 0	1 1 0 1 0 1
I4	0 0 1 1	0 1 x x	0 0 0 x	1 x 1 0 x x 1 0 0 x x	x x x x	x x x x x x
I5	x x x x	0 0 0 x	0 0 0 x	0 x 1 1 x x 1 0 0 x x	x x 0 0	0 0 0 0 1 0
I6	x x x x	0 0 0 x	0 0 0 x	0 x 0 1 x x 1 0 0 x x	x x 0 0	1 1 0 1 0 1
I7	0 1 1 1	0 1 x x	0 0 0 x	1 x 1 0 x x 1 0 0 x x	x x x x	x x x x x x
I8	x x x x	0 0 0 x	0 0 x x	0 x 1 1 x x 1 0 0 x x	x x 0 0	0 0 0 0 1 0
I9	1 1 1 0	0 1 x x	0 0 0 x	1 x 1 0 x x 1 0 0 x x	x x x x	x x x x x x
I10	x x x x	0 0 0 x	0 0 x x	0 x 1 1 x x 1 0 0 x x	x x 0 0	0 0 0 0 1 0
I11	x x x x	0 0 0 x	0 0 0 x	0 x 0 1 x x 1 0 0 x x	x x 0 0	1 1 0 1 0 1
I12	1 1 1 0	0 1 x x	0 0 0 x	1 x 1 0 x x 1 0 0 x x	x x x x	x x x x x x
I13	1 1 1 0	0 1 x x	1 0 0 x	1 x 1 0 x x x x 0 x x	x x x x	x x x x x x
I14	x x x x	0 0 0 x	0 0 0 x	0 x 1 1 x x 1 0 0 x x	x x 0 0	1 1 1 0 1 0
I15	1 1 1 0	0 1 x x	0 0 0 x	1 x 1 0 x x 1 0 0 x x	x x x x	x x x x x x
I16	1 1 1 0	0 1 x x	1 0 0 x	1 x 1 0 x x x x 0 x x	x x x x	x x x x x x
I17	1 1 1 0	0 1 x x	1 0 0 x	1 x 1 0 x x x x 0 x x	x x x x	x x x x x x
I18	1 1 1 0	0 1 x x	1 0 0 x	1 x 1 0 x x x x 0 x x	x x x x	x x x x x x
I19	0 0 1 0	0 1 x x	1 0 0 x	1 x 1 0 x x x x 0 x x	x x x x	x x x x x x
I20	x x x x	0 0 0 x	0 0 0 x	0 x 1 1 x x 1 0 0 x x	x x 0 0	0 0 0 0 1 0
I21	x x x x	0 0 0 x	0 1 x x	1 x 0 0 1 0 x x 0 x x	x x 1 0	x x x x x x
I22	x x x x	0 0 x 0	0 0 x 0	0 x 0 0 x x x x 1 x 0	x x 0 0	0 0 0 0 0 0
I23	x x x x	0 0 0 x	0 0 x 1	0 x 0 1 0 0 x x 0 x x	x x 0 0	0 0 1 0 0 0
I24	x x x x	0 0 0 x	0 0 0 0	0 x x 0 x x 0 0 0 x 0	x x x x	x x x x x x

6.3 The Finite State Machine (FSM)

The three phases of computation done by the ECCP, namely the initialization, scalar multiplication and projective to affine conversion phase are implemented using the FSM shown in Figure 6.4. The first three states of the FSM do the initialization. In these states the curve constant and basepoint coordinates are loaded from ROM into the registers (Table 6.6). These states also detect the leading MSB in the scalar key k. After initialization, the scalar multiplication is done. This consists of 4 states for doubling and 8 for the point addition. The states that do the doubling are $D1 \cdots D4$. In state $D4$, a decision is made depending on the key bit k_i (i is a loop counter initially set to the position of the leading one in the key, and k_i is the i^{th} bit of the key k). If $k_i = 1$ then a point addition is done and state $A1$ is entered. If $k_i = 0$, the addition is not done and the next key bit (corresponding to $i - 1$) is considered. If $k_i = 0$ and there are no more key bits to be considered then the *complete* signal is issued and it marks the end of the scalar multiplication phase. The states that do the addition are $A1 \cdots A8$. At the end of the addition (state $A8$) state $D1$ is entered and the key bit k_{i-1} is considered. If there are no more key bits remaining the complete signal is asserted. Table 6.7 shows the control words generated at every state.

At the end of the scalar multiplication phase, the result obtained is in projective coordinates and the X, Y, and Z coordinates are stored in the registers RA_1, RB_1, and RC_1 respectively. To convert the projective point to affine, the following equation is used.

$$\begin{aligned} x &= X \cdot Z^{-1} \\ y &= Y \cdot (Z^{-1})^2 \end{aligned} \tag{6.5}$$

The inverse of Z is obtained using the *quad-ITA* discussed in Algorithm 5.1. The addition chain used is the Brauer chain in Equation 5.3. The processor implements the steps given in Table 5.4. Each step in Table 5.4 gets mapped into one or more states from $I1$ to $I21$. The number of clock cycles required to find the inverse is 21. This is lesser than the clock cycles estimated by Equation 5.9. This is because, inverse can be implemented more efficiently in the ECCP by utilizing the squarers present in the AU.

At the end of state $I21$, the inverse of Z is present in the register RC_1. The states $I22$ and $I23$ compute the affine coordinates x and y respectively.

The number of clock cycles required for the ECCP to produce the output is computed as follows. Let the scalar k has length l and hamming weight h, then the clock cycles required to produce the output is given by the following equation.

$$\begin{aligned} \#ClockCycles &= 3 + 12(h-1) + 4(l-h) + 24 \\ &= 15 + 8h + 4l \end{aligned} \tag{6.6}$$

Three clock cycles are added for the initial states, 24 clock cycles are required for the final projective to affine conversion. $12(h-1)$ cycles are required to handle the 1's in k. Note that the MSB of k does not need to be considered. $4(l-h)$ cycles are required for the 0's in k.

6.4 Performance Evaluation

In this section we compare our work with reported $GF(2^m)$ elliptic curve crypto processors implemented on FPGA platforms (Table 6.8). Our ECCP was synthesized using *Xilinx's ISE* for *Virtex 4* and *Virtex E* platforms. Since, the reported works are done on different field sizes. We use the measure $latency/bit$ for evaluation. Here latency is the time required to compute kP. Latency is computed by assuming the scalar k has half the number of bits 1. The only faster implementations are [37] and [1]. However, [37] does not perform the final inverse computation required for converting from LD to affine coordinates. Also, as shown in Table 6.9 our implementation has a better area time product compared to [1], while the latency is almost equal. To compare the two designs we scaled the area of [1] by a factor of $(233/m)^2$, since area of the elliptic curve processors is mostly influenced by the multiplier which has an area of $O(n^2)$. The time is scaled by a factor $(233/m)$, since the time required is linear.

Table 6.8: Comparison of the Proposed $GF(2^m)$ ECCP with FPGA based Published Results

Work	Platform	Field m	Slices	LUTs	Gate Count	Freq (MHz)	Latency (ms)	Latency /bit (ns)
Orlando [29]	XCV400E	163	-	3002	-	76.7	0.21	1288
Bednara [33]	XCV1000	191	-	48300	-	36	0.27	1413
Kerins [32]	XCV2000	239	-	-	74103	30	12.8	53556
Gura [34]	XCV2000E	163	-	19508	-	66.5	0.14	858
Mentens [65]	XCV800	160	-	-	150678	47	3.810	23812
Lutz [35]	XCV2000E	163	-	10017	-	66	0.075	460
Saqib [37]	XCV3200	191	18314	-	-	10	0.056	293
Pu [38]	XC2V1000	193	-	3601	-	115	0.167	865
Ansari [40]	XC2V2000	163	-	8300	-	100	0.042	257
Chelton [1]	XCV2600E	163	15368	26390	238145	91	0.033	202
	XC4V200	163	16209	26364	264197	153.9	0.019	116
This Work	XCV3200E	233	20325	40686	333063	25.31	0.074	317
	XC4V140	233	20917	39303	334709	64.46	0.029	124

Table 6.9: Comparing Area$\times$Time Requirements with [1]

Work	Field	Platform	Slices	Scaled	Latency	Scaled	Area
				Slices	(ms)	Latency (ms)	$\times$Time
	(m)		(S)	$SS = S(\frac{233}{m})^2$	(T)	$TS = T(\frac{233}{m})$	$(SS \times TS)$
Chelton [1]	163	XC4V200	16209	33120	0.019	0.027	894
This Work	233	XC4V140	20917	20917	0.029	0.029	606

6.5 Conclusion

This chapter integrates the previously developed finite field arithmetic blocks to form an arithmetic unit. The AU is used in a elliptic curve crypto processor to compute the scalar product kP for a NIST specified curve. Our ECCP has the best timing per bit compared to most of the reported works. Of all works compared, only two have better timing compared to ours. We showed that our design has more efficient FPGA utilization compared to these works.

CHAPTER 7

Side Channel Analysis of the ECCP

The previous chapter presented the construction of an elliptic curve crypto processor. This chapter discusses issues regarding side channel analysis of the processor. First a side channel attack based on *simple power analysis* (SPA) of the ECCP is demonstrated. Then the architecture of the ECCP is modified to reduce the threat of SPA. We call this new architecture *SPA resistant elliptic curve crypto processor* (SR-ECCP).

This chapter is organized as follows : the next section demonstrates a simple power analysis on the ECCP. Section 7.2 presents the SR-ECCP and shows how the power traces do not reveal the key any more. The final section has the conclusion.

7.1 Simple Power Analysis on the ECCP

The state machine for the scalar multiplication in the ECCP has 12 states (Figure 6.4), 4 states ($D1 \cdots D4$) for doubling and 8 states ($A1 \cdots A8$) for addition. Each iteration in the scalar multiplication handles a bit in the key starting from the most significant one to the least significant bit. If the key bit is zero a doubling is done and no addition is done. If the key bit is one the doubling is followed by an addition. The dissimilarity in the way a 1 and a 0 in the key is handled makes the ECCP vulnerable to side channel attacks as enumerated below.

- The duration of an iteration depends on the key bit. A key bit of 0 leads to a short cycle compared to a key bit of 1. Thus measuring the duration of an iteration will give an attacker knowledge about the key bit.

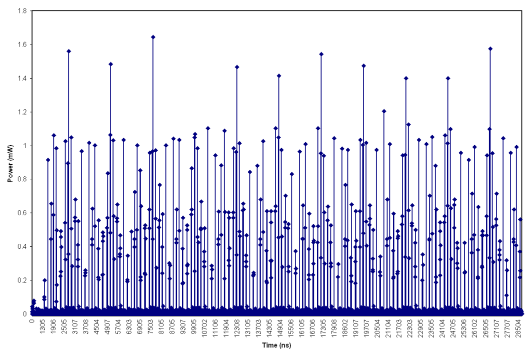

Fig. 7.1: Power Trace for a Key with all 1

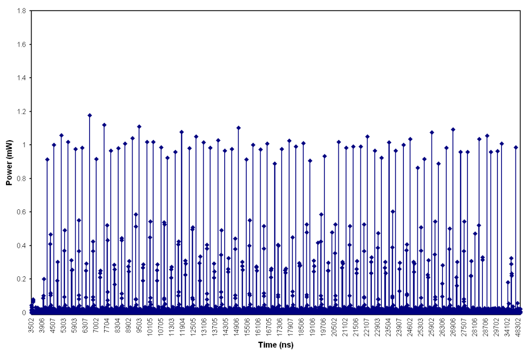

Fig. 7.2: Power Trace for a Key with all 0

- Each state in the FSM has a unique power consumption trace. Monitoring the power consumption trace would reveal if an addition is done thus revealing the key bit.

To demonstrate the attack we used *Xilinx's XPower* [1] tool. Given a *value change dump* (VCD) file generated from a flattened post map or post route netlist, XPower is capable of generating a power trace for a given testbench (details on generating the power trace is given in Appendix C).

Figures 7.1 and 7.2 are partial power traces generated for the key $(FFFFFFFF)_{16}$ and $(80000000)_{16}$ respectively. The graphs plots the power on the Y axis with the time line on the X axis for a *Xilinx Virtex 4 FPGA*. The difference in the graphs is easily noticeable. The spikes in Figure 7.1 occurs in state $A6$. This state is entered only when a point addition is done, which in turn is done only when the key bit is 1. The spikes are not present in Figure 7.2 as the state $A6$ is never entered. Therefore the spikes in the trace can be used to identify ones in the key.

The duration between two spikes in Figure 7.1 is the time taken to do a point doubling and a point addition. This is 12 clock cycles. If there are two spikes with a distance greater than 12 clock cycles, it indicates that one or more zeroes are present in the key. The number of zeroes (n) present can be determined by Equation 7.1. In the

[1]http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

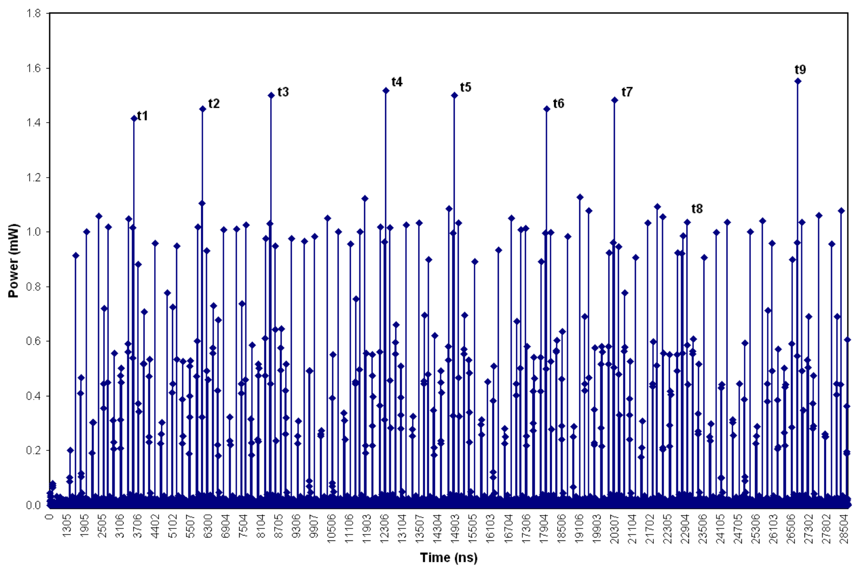

Fig. 7.3: Power Trace when $k = (B9B9)_{16}$

equation t is the duration between the two spikes and T is the time period of the clock.

$$n = \frac{t}{4T} - 3 \tag{7.1}$$

The number of zeroes between the leading one in k and the one due to the first spike can be inferred by the amount of shift in the first spike.

As an example consider the power trace (Figure 7.3) for the ECCP obtained when the key was set to $(B9B9)_{16}$. There are 9 spikes indicating 9 ones in the key (excluding the leading one). Table 7.1 infers the key from the time duration between spikes. The clock has a period $T = 200ns$.

The first spike t_1 is obtained at 3506^{th} ns. If there were no zeros before t_1 the spike should have been present at 2706^{th} ns (this is obtained from the first spike of Figure 7.1). The shift is 800 ns equal to four clock cycles. Therefore a 0 is present before the t_1 spike.

Table 7.1: SPA for the key $B9B9_{16}$

i	$t_i - t_{i-1}$	n	Key Inferred
1	-	-	01
2	2400 ns	0	1
3	2400 ns	0	1
4	4000 ns	2	001
5	2400 ns	0	1
6	3200 ns	0	01
7	2400 ns	0	1
8	2400 ns	0	1
9	4000 ns	2	001

The key obtained from the attack is $(1011100110111001)_2$, and it matches the actual key.

7.2 SPA Resistant ECCP

To harden the ECCP against SPA, the sequence of computations involved when the key bit is 1 and when the key bit is 0 must be indistinguishable. There are several ways to achieve this. The most common technique is by inserting a dummy addition when the key bit is 0[66]. This is shown in Figure 7.4. With this method, a doubling and an addition is always done. The value of the key bit decides if the addition should be considered. This makes the sequence for a key bit of 1 indistinguishable from a 0. The time for an iteration is a constant therefore reducing timing attacks. Similar power traces are seen at every iteration thus reducing threats of power attacks. The following section modifies the ECCP architecture using the dummy addition to make it robust against SPA.

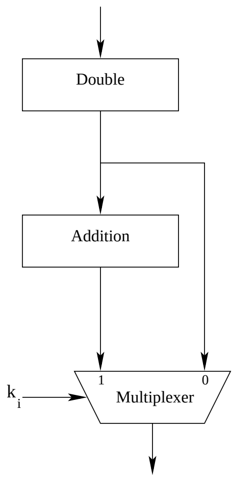

Fig. 7.4: Always Add Method to Prevent SPA

7.2.1 The SR-ECCP

Modifying the ECCP to incorporate 'adding always' requires a change in the FSM and the register file. The new FSM is as shown in Figure 7.5. Irrespective of the key bit all states $D1 \cdots D4$ and $A1 \cdots A8$ are entered in every iteration. If the key bit is 1 the result of state $A8$ is considered as the output of the iteration. If the key bit is 0 the result of $D4$ is taken as the output. After all key bits are processed the $complete$ signal is asserted.

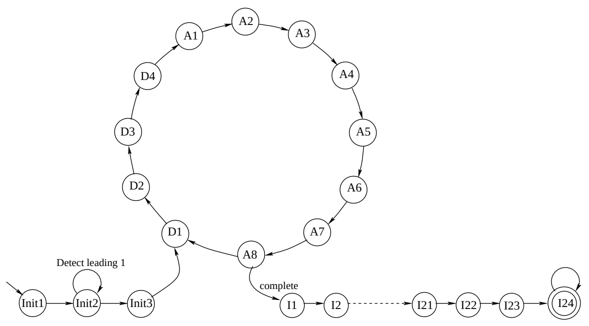

Fig. 7.5: FSM for SR-ECCP

The SR-ECCP also requires a modification in the register file as shown in Figure

7.6. An additional register bank RD containing three registers is introduced. The three registers in the bank RD_1, RD_2 and RD_3 store the coordinates of the computed double. The outputs of the register bank is used in state $A8$ only when the key bit is 0. RD requires an additional input multiplexer $MUXIN4$ to store the doubled result. The size of the output multiplexers $MUXOUT1$, $MUXOUT2$ and $MUXOUT3$ are increased to incorporate $RD's$ outputs.

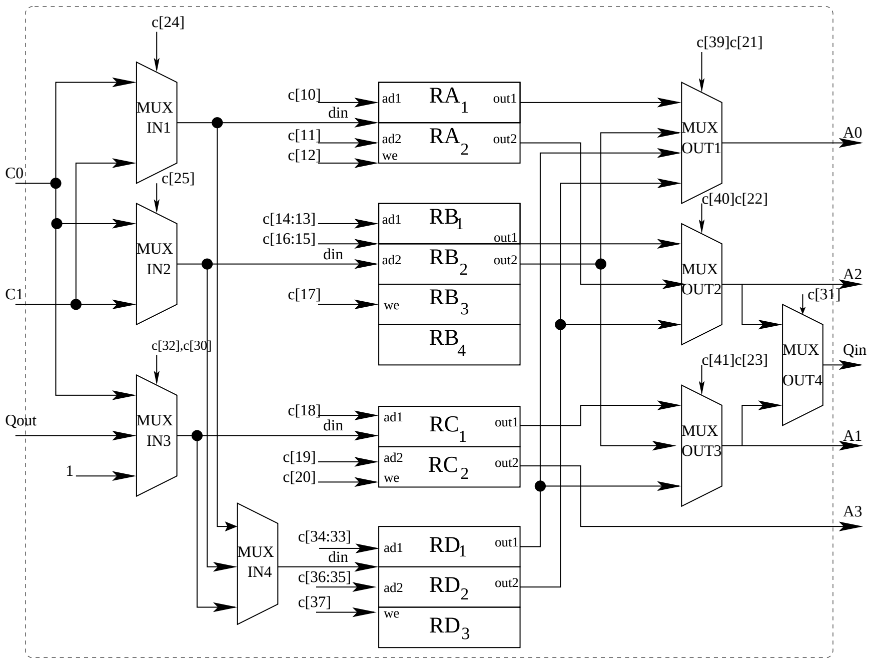

Fig. 7.6: Register File for SR-ECCP

7.2.2 Power Trace of the SR-ECCP

Figure 7.7 has the power trace for the SR-ECCP for the key $(B9B9)_{16}$. This is the same key used in the power trace of Figure 7.3. However, unlike Figure 7.3, Figure 7.7 has no periodic spikes. Thus using a simple power analysis the key cannot be inferred from Figure 7.7.

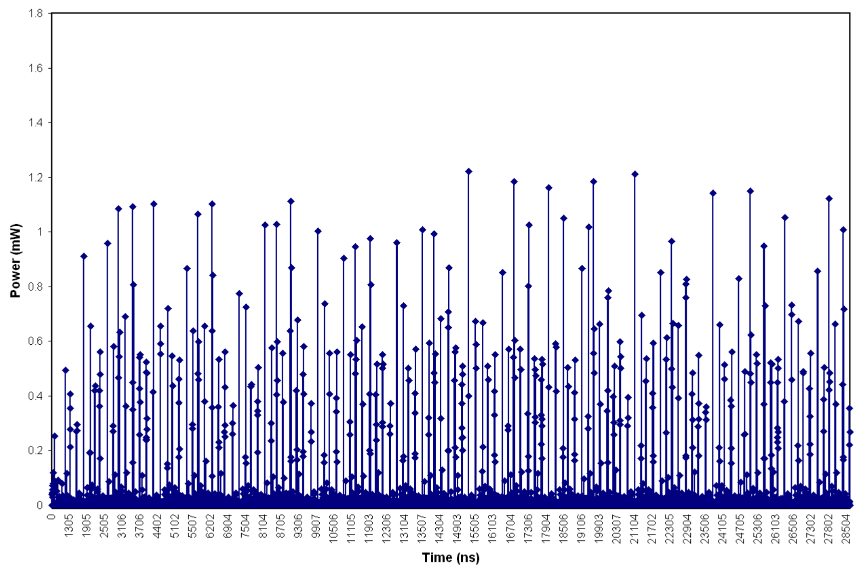

Fig. 7.7: Power Trace when $k = (B9B9)_{16}$

Table 7.2: Performance Evaluation of the SR-ECCP

Processor	Device	Slices	Frequency	Clock Cycles
ECCP	Xilinx Virtex 4 (XC4VFX140)	21852	64.46MHz	1883
SR-ECCP	Xilinx Virtex 4 (XC4VFX140)	23511	56.46MHz	2811

7.2.3 Performance Evaluation

The modification of the ECCP to improve its security comes at a cost of increased area, lower frequency and increased computation time. Table 7.2 shows the overhead of the SR-ECCP compared to the ECCP. The *clock cycles* is the number of clocks required to compute kP, assuming k has 116 zeroes out of 233 and the MSB of k is 1. The clock cycles required for the SR-ECCP is always a constant irrespective of the number of zeroes in k.

7.3 Conclusion

This chapter demonstrated the vulnerability of the ECCP to simple power analysis. Simulations show that power trace of the processor leak the secret key. The vulnerabilities of the ECCP were fixed in the SR-ECCP, which does homogeneous operations irrespective of the key bit. The penalty of the SR-ECCP is a larger area requirement and lower frequency compared to the ECCP.

CHAPTER 8

Conclusions and Future Work

The thesis explores various architectures for the construction of an elliptic curve crypto processor for high performance applications. The most important factor contributing to the performance is the finite field multiplication and finite field inversion. A combinational multiplier is able to obtain the product in one clock cycle at the cost of increased area and delay. In order to ensure that the primitives have a good area delay product, the thesis suggests techniques to reduce the area time product by effectively utilizing the available FPGA resources.

A hybrid Karatsuba multiplier is proposed for finite field multiplication, which has been shown to possess the best area time product compared to reported Karatsuba implementations. The hybrid Karatsuba multiplier is a recursive algorithm which does the initial recursions using the simple Karatsuba multiplier [55], while the final recursion is done using the general Karatsuba multiplier [55]. The general Karatsuba has large gate counts, however it is more compact for small sized multiplications due to the better LUT utilization. The simple Karatsuba multiplier is more efficient for large sized multiplications. After a thorough search, a threshold of 29 was found. Multiplications smaller than 29 bits is done using the general Karatsuba multiplier, while larger multiplications are done with the simple Karatsuba multiplier.

The quad-Itoh Tsujii inversion algorithm proposed to find the multiplicative inverse has the best computation time and area time product compared to works reported in literature. This work first generalizes the Itoh-Tsujii algorithm and then shows that a specific instance of the generalization, which uses quad circuits instead of squarers, is more efficient on FPGAs.

An elliptic curve crypto processor is built using the proposed finite field primitives. Except for [1], the constructed processor has better timing than all reported works. However, the constructed processor has much better area requirements and area time product compared to [1]. These were achieved in spite of the fact that the scalar multiplication implemented was straight forward and no parallelism or pipelining in the architecture was used.

8.1 Future Work

- The focus of this work was on the implementation of efficient elliptic curve primitives for ECC and its impact on the overall performance of the ECCP. Thus a possible future work could be to combine architectural techniques like pipelining and parallelism in the higher level scalar multiplier with techniques proposed in this thesis.
- The toplevel is a simple implementation of the Montgomery multiplication using López-Dahab (LD) projective coordinates. The combination of more sophisticated methods like add and half method, LD method, non adjacent form methods, mixed coordinates etc. with the proposed primitives may be experimented.
- A simple power attack was analyzed and prevented in the side channel resistant version of the elliptic curve crypto processor. A very interesting field of research, would be to study the effect of the more powerful *differential power analysis* (DPA) on the proposed architecture.
- To make the work proposed in this thesis usable in practice, the developed elliptic curve crypto processor may be incorporated in security toolkits such as OpenSSL[1]. This involves the development of a communication interface for com-

[1]http://www.openssl.org

munication with the host processor, operating system device drivers and library modifications.

APPENDIX A

Verification and Testing of the ECCP

A.1 Verification of the ECCP and SR-ECCP

The elliptic curve crypto processor (ECCP) and the side channel resistant version of the ECCP, the SR-ECCP, have to be verified for their correctness. The verification was done for the curve given Equation A.1.

$$y^2 + xy = x^3 + ax^2 + b \tag{A.1}$$

The basepoint and the values of the curve constants used is given in Table A.1. These constants were taken from NIST's digital signature specification [14] for elliptic curves over $GF(2^{233})$.

For a key (k), the scalar product kP is determined by simulation of the ECCP (or the SR-ECCP) with Modelsim or iVerilog. Here, P is the basepoint with coordinates (P_x, P_y). The result thus obtained is verified against the result obtained by running the

Table A.1: Basepoint and Curve Constants used for Verification of the ECCP and the SR-ECCP

Basepoint X (P_x)	233'h0FAC9DFCBAC8313BB2139F1 BB755FEF65BC391F8B36F8F8EB7371FD558B
Basepoint Y (P_y)	233'h1006A08A41903350678E585 28BEBF8A0BEFF867A7CA36716F7E01F81052
Curve constant (b)	233'h066647EDE6C332C7F8C0923 BB58213B333B20E9CE4281FE115F7D8F90AD
Curve constant (a)	1

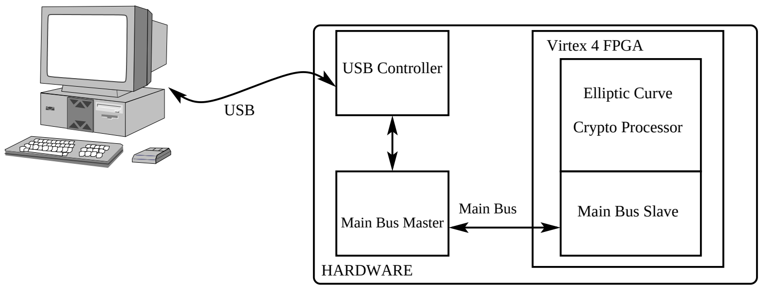

Fig. A.1: Test Platform for the ECCP

elliptic curve software with the same key k. The elliptic curve software was obtained from the book *Implementing Elliptic Curve Cryptography* by Michael Rosing [67].

A Python [1] script was developed which would automatically generate a random key k. This key is used by Rosing's software to determine $Q_1 = kP$. The key is also used in the test vector of the ECCP(or SR-ECCP) to determine $Q_2 = kP$. The python script would then verify if $Q_1 = Q_2$. A large number of scalar multiplications were tested using the above mentioned procedure.

A.2 Testing of the ECCP

The testing of the ECCP was done using the Virtex 4 FPGA board from Dinigroup[2]. The simplified block diagram of the test platform is shown in Figure A.1. USB communication software supplied by the manufacturer was used to communicate between the PC and the hardware. Onboard devices convert the USB protocol into a proprietary *main bus* protocol[3]. This channel is used to configure the FPGAs as well as communicate with the elliptic curve processor. Our implementations resides in the Virtex 4 FPGA (FPGA_A). The *main bus slave* has eight 32 bit input registers ($Rin0 \cdots Rin7$) and sixteen output registers ($Rout0 \cdots Rout15$). It also has a control register contain-

[1]www.python.org
[2]http://www.dinigroup.com/DN8000k10pcie.php
[3]http://www.dinigroup.com/product/common/mainbus_spec.pdf

Table A.2: ECCP System Specifications on the Dini Hardware

Frequency	24MHz
Slices occupied	22526
Size on device	25%
Clock Cycles Required	1883 (average case)
Critical Path	Follows the path from register bank, quadblock MUX C and register bank again.

ing status bits such as *start* (to start the scalar multiplication) and *done* (to indicate completion). To initialize, the 233 bit scalar k (in Algorithm 3.1) is loaded into the registers $Rin0$ to $Rin7$. On completion the result Q_x and Q_y can be read from $Rout0$ to $Rout15$. The results from testing on the hardware was as expected. Table A.2 shows the specifications of the system when used with the Dini card.

APPENDIX B

Finite Fields used for Performance Evaluation of ITA

The graph in Figure 5.5 was plotted after synthesizing the quad-ITA and the squarer-ITA for several finite fields. The following table contains the addition chains, irreducible polynomials and number of cascaded quad circuits in the quadblock for each implementation of the (quad-)ITA.

Finite Field	**Addition Chain**	**Irreducible Polynomial**	u_s
$GF(2^{103})$	(1 2 3 6 12 24 25 50 51 102)	$x^{103}+x^9+1=0$	12
$GF(2^{111})$	(1 2 3 6 12 13 26 27 54 55 110)	$x^{111}+x^{10}+1=0$	13
$GF(2^{121})$	(1 2 3 6 7 14 15 30 60 120)	$x^{121}+x^{18}+1=0$	14
$GF(2^{129})$	(1 2 4 8 16 32 64 128)	$x^{129}+x^5+1=0$	16
$GF(2^{147})$	(1 2 4 8 9 18 36 72 73 146)	$x^{147}+x^{14}+1=0$	18
$GF(2^{161})$	(1 2 4 5 10 20 40 80 160)	$x^{161}+x^{18}+1=0$	10
$GF(2^{169})$	(1 2 4 5 10 20 21 42 84 168)	$x^{169}+x^{34}+1=0$	10
$GF(2^{177})$	(1 2 4 5 10 11 22 44 88 176)	$x^{177}+x^8+1=0$	11
$GF(2^{193})$	(1 2 3 6 12 24 48 96 192)	$x^{193}+x^{15}+1=0$	12
$GF(2^{201})$	(1 2 3 6 12 24 25 50 100 200)	$x^{201}+x^{14}+1=0$	12
$GF(2^{209})$	(1 2 3 6 12 13 26 52 104 208)	$x^{209}+x^6+1=0$	13
$GF(2^{225})$	(1 2 3 6 7 14 28 56 112 224)	$x^{225}+x^{32}+1=0$	14
$GF(2^{233})$	(1 2 3 6 7 14 28 29 58 116 232)	$x^{233}+x^{74}+1=0$	14
$GF(2^{241})$	(1 2 3 6 7 14 15 30 60 120 240)	$x^{241}+x^{70}+1=0$	15
$GF(2^{253})$	(1 2 3 6 7 14 15 30 31 62 63 126 252)	$x^{253}+x^{46}+1=0$	15
$GF(2^{273})$	(1 2 4 8 16 17 34 68 136 272)	$x^{273}+x^{23}+1=0$	17
$GF(2^{281})$	(1 2 4 8 16 17 34 35 70 140 280)	$x^{281}+x^{93}+1=0$	17
$GF(2^{289})$	(1 2 4 8 9 18 36 72 144 288)	$x^{289}+x^{21}+1=0$	18

APPENDIX C

Using XPower to Obtain Power Traces of a Device

There are two forms of power dissipation for a device; static and dynamic power. *Static power* is the amount of power dissipated by the device when no clock is running. During this phase no signals toggle, hence the power consumed is the minimum power required to maintain the state of the logic cell. *Dynamic power* is the amount of power dissipated by the device when the clock is running. The dynamic power is considerably higher than the static power consumed by the device, and it is generally caused when one or more of the inputs toggle. Analysis of the instantaneous dynamic power of the device is used in side channel attacks.

Obtaining power traces of a device require equipments such as storage oscilloscopes and power analyzers. However these equipments are expensive and therefore not easy to procure. Most importantly, through this flow we can cross check the side channel vulnerability using simulation without being hampered by noise picked up during an actual measurement. We therefore use Xilinx's XPower tool to analyze the power consumption of a design after it has been placed and routed.

C.1 XPower

The XPower tool estimates the power consumption for a variety of Xilinx FPGA architectures. The estimation is based on the device and the number of transitions (activity rate) of the device.

The following procedure is used to estimate the power consumed by a device using Xilinx's ISE and XPower.

- The developed verilog code is synthesized using the Xilinx ISE tool. The result of synthesis is a *.ngd* file. This file is a netlist of primitive gates which could be implemented on several of the Xilinx FPGAs.

- The next step is to map the primitives onto the resources available on the specific FPGA platform. This is done by the Xilinx map tool. The output of the tool is an *.ncd* file.

- The *.ncd* file is then passed to the place and route tool, where specific locations on the FPGA are assigned. This tool tries to incorporate all the timing constraints specified in the constraints file. The output of the place and route tool is an updated *.ncd* file.

- In ISE, a flattened verilog netlist can be generated after the mapping or the place and route. This verilog netlist after the mapping can be created by clicking the *generate post-map simulation model*. This would create a verilog netlist called *topmodule_map.v*. Also a *.sdf* file is created containing timing information of the device.

- Now the flattened verilog file and the *sdf* along with a testbench can be simulated in Modelsim. A value change dump file containing all the signal transitions can be generated from the simulation. This requires the following lines to be present in the test bench.

```
initial begin
$dumpfile ("dump.vcd");    /* File to place signal activity report */
$dumpvars;                 /* Dump all signals in the design */
$dumpon;                   /* Turn on dump */
#100000 $dumpoff;          /* Turn off dump */
end
```

  These lines will result in a file called *dump.vcd* to be generated during simulation. The VCD file contains the activity on each signal in the design.

- The constraints file (*.pcf*), the *.vcd* file and the *.ncd* file are used as inputs to XPower. XPower can be run from command line as shown below.
  xpwr topmodule_map.ncd topmodule.pcf -s dump.vcd
  The result produced by xpwr is present in a text file called *topmodule.txt*. The topmodule.txt file contains the instantaneous power consumption for the given test vector.
- This text file is plotted on a graph to obtain the power trace.

If the *.sdf* file generated by ISE is used in XPower, then the power measurement would include the power consumed due to glitches. If the post place and route verilog netlist was used instead of the mapped netlist then more accurate power measurement is possible.

APPENDIX D

Elliptic Curve Arithmetic

This appendix derives the elliptic curve equations for points in affine coordinates and López-Dahab projective coordinates.

Consider the elliptic curve E over the field $GF(2^m)$. This is given by

$$y^2 + xy = x^3 + ax^2 + b \tag{D.1}$$

where a, $b \in GF(2^m)$.

Equation D.1 can be rewritten as

$$F(x,y) : y^2 + x^3 + xy + ax^2 + b = 0 \tag{D.2}$$

The partial derivatives of this equation are

$$\begin{aligned} \frac{dF}{dy} &= x \\ \frac{dF}{dx} &= x^2 + y \end{aligned} \tag{D.3}$$

If we consider the curve given in Equation D.1, with $b = 0$, then the point $(0,0)$ lies on the curve. At this point $dF/dy = dF/dx = 0$. This forms a *singular point* and cannot be included in the elliptic curve group, therefore an additional condition of $b \neq 0$ is required on the elliptic curve of Equation D.1. This condition ensures that the curve is *non singular*.

D.1 Equations for Arithmetic in Affine Coordinates

D.1.1 Point Inversion

Let $P = (x_1, y_1)$ be a point on the elliptic curve of Equation D.1. To find the inverse of point P, a vertical line is drawn passing through P. The equation of this line is $x = x_1$. The point at which this line intersects the curve is the inverse $-P$. The coordinates of $-P$ is (x_1, y_1'). To find y_1', the point of intersection between the line and the curve must be found. Equation D.2 is represented in terms of its roots p and q as shown below.

$$(y - A)(y - B) = y^2 - (p + q)y + pq \tag{D.4}$$

The coefficients of y is the sum of the roots. Equating the coefficients of y in Equations D.2 and D.4.

$$p + q = x_1$$

One of the roots is $q = y_1$, therefore the other root p is given by

$$p = x_1 + y_1$$

This is the y coordinate of the inverse. The inverse of the point P is therefore given by $(x_1, x_1 + y_1)$.

D.1.2 Point Addition

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points on the elliptic curve. To add the two points, a line (l) is drawn through P and Q. If $P \neq \pm Q$, the line intersects the curve of Equation D.1 at the point $-R = (x_3, y_3')$. The inverse of the point $-R$ is $R = (P + Q)$ having coordinates (x_3, y_3).

The slope of the line l passing through P and Q is given by

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

Equation of the line l is

$$\begin{aligned} y - y_1 &= \lambda(x - x_1) \\ y &= \lambda(x - x_1) + y_1 \end{aligned} \tag{D.5}$$

Substituting y from D.5 in the elliptic curve equation D.1 we get,

$$(\lambda(x - x_1) + y_1)^2 + x(\lambda(x - x_1) + y_1) = x^3 + ax^2 + b$$

This can be rewritten as

$$x^3 + (\lambda^2 + \lambda + a)x^2 + \cdots = 0 \tag{D.6}$$

Equation D.6 is a cubic equation having three roots. Let the roots be p, q and r. These roots represent the x coordinates of the points on the line that intersect the curve (the point P, Q and $-R$). Equation D.6 can be also represented in terms of its roots as

$$\begin{aligned} (x - p)(x - q)(x - r) = 0 \\ x^3 - (p + q + r)x^2 \cdots = 0 \end{aligned} \tag{D.7}$$

Equating the x^2 coefficients of Equations D.7 and D.6 we get,

$$p + q + r = \lambda^2 + \lambda + a \tag{D.8}$$

Since $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ lie on the line l, therefore two roots of Equation D.6 are x_1 and x_2. Substituting $p = x_1$ and $q = x_2$ in Equation D.8 we get the third root, this is the x coordinate of the third point on the line which intersects the curve(ie.

$-R$). This point is denoted by x_3, and it also represents the x coordinate of R.

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \tag{D.9}$$

The y coordinate of $-R$ can be obtained by substituting $x = x_3$ in Equation D.5. This point is denoted as y_3'.

$$y_3' = \lambda(x_3 + x_1) + y_1 \tag{D.10}$$

Reflecting this point about the x axis is done by substituting $y_3' = x_3 + y_3$. This gives the y coordinate of R, denoted by y_3.

$$y_3 = \lambda(x_3 + x_1) + y_1 + x_3 \tag{D.11}$$

Since we are working with binary finite fields, subtraction is the same as addition. Therefore,

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= \lambda(x_3 + x_1) + y_1 + x_3 \\ \lambda &= \frac{y_2 + y_1}{x_2 + x_1} \end{aligned} \tag{D.12}$$

D.1.3 Point Doubling

Let $P = (x_1, y_1)$ be a point on the elliptic curve. The double of P, ie. $2P$, is found by drawing a tangent t through P. This tangent intersects the curve at the point $-2P = (x_3, y_3')$. Taking the reflection of the point $-2P$ about the X axis gives $2P = (x_3, y_3)$.

First, let us look at the tangent t through P. The slope of the tangent t is obtained by implicit differentiation of Equation D.1.

$$2y\frac{dy}{dx} + x\frac{dy}{dx} + y = 3x^2 + 2ax$$

Since we are using modular 2 arithmetic,

$$x\frac{dy}{dx} + y = x^2$$

The slope dy/dx of the line t passing through the point P is given by

$$\lambda = \frac{{x_1}^2 + y_1}{x_1} \tag{D.13}$$

The equation of the line t can be represented by the following.

$$y + y_1 = \lambda(x + x_1) \tag{D.14}$$

This gives,

$$\begin{aligned} y &= \lambda(x + x_1) + y_1 \\ y &= \lambda x + c \;\text{ for some constant } c \end{aligned}$$

To find x_3 (the x coordinate of $-2P$), substitute for y in Equation D.1.

$$(\lambda x + c)^2 + x(\lambda x + c) = x^3 + ax + b$$

This equation can be rewritten as

$$0 = x^3 + (\lambda^2 + \lambda + a)x + \cdots \tag{D.15}$$

This equation is cubic and has three roots. Of these three roots, two roots must be equal since the line intersects the curve at exactly two points. The two equal roots are represented by p. The sum of the three roots is $(\lambda^2 + \lambda + a)$, similar to Equation D.7.

Therefore,

$$\begin{aligned} p+p+r &= \lambda^2+\lambda+a \\ r &= \lambda^2+\lambda+a \end{aligned}$$

The dissimilar root is r. This root corresponds to the x coordinate of $-2P$ ie. x_3. Therefore,

$$x_3 = \lambda^2+\lambda+a$$

To find the y coordinate of $-2P$, ie. y_3', substitute x_3 in Equation D.14. This gives,

$$\begin{aligned} y_3' &= \lambda x_3+\lambda x_1+y_1 \\ y_3' &= \lambda x_3+{x_1}^2 \end{aligned}$$

To find y_3, the y coordinate of $2P$, the point y_3' is reflected on the x axis. From the point inverse equation

$$y_3 = \lambda x_3+{x_1}^2+x_3$$

To summarize, the coordinates of the double are given by Equation D.16

$$\begin{aligned} x_3 &= \lambda^2+\lambda+a \\ y_3 &= {x_1}^2+\lambda x_3+x_3 \\ \lambda &= x_1+\frac{y_1}{x_1} \end{aligned} \tag{D.16}$$

D.2 Equations for Arithmetic in LD Projective Coordinates

D.2.1 Point Inversion

Inverting a point $P = (x_1, y_1)$ on the elliptic curve results in the point $-P = (x_3, y_3) = (x_1, x_1 + y_1)$. Converting x_1 to X_1/Z_1, x_3 to X_3/Z_3 and y_1 to $Y_1/{Z_1}^2$, y_3 to $Y_3/{Z_3}^2$ Then $\frac{X_3}{Z_3} = \frac{X_1}{Z_1}$, therefore $X_3 = X_1$ and $Z_3 = Z_1$. Also,

$$\begin{aligned}\frac{Y_3}{{Z_3}^2} &= \frac{X_1}{Z_1} + \frac{Y_1}{{Z_1}^2} \\ &= \frac{X_1 Z_1 + Y_1}{{Z_1}^2}\end{aligned}$$

Therefore, $-P = (X_3, Y_3, Z_3)$ in projective coordinates is $(X_1, X_1 Z_1 + Y_1, Z_1)$.

D.2.2 Point Addition

In Equation D.12, change x_1 to X_1/Z_1, x_3 to X_3/Z_3 and y_1 to $Y_1/{Z_1}^2$, y_3 to $Y_3/{Z_3}^2$. Then the slope λ becomes

$$\begin{aligned}\lambda &= \frac{y_2 + (Y_1/{Z_1}^2)}{x_2 + (X_1/Z_1)} \\ &= \frac{y_2 {Z_1}^2 + y_1}{Z_1(x_2 Z_1 + X_1)}\end{aligned}$$

Let $\mathbf{A} = y_2 {Z_1}^2 + Y_1$, $\mathbf{B} = x_2 Z_1 + X_1$ and $\mathbf{C} = Z_1 B$. Then,

$$\lambda = \frac{A}{Z_1 \cdot B}$$

Consider equation for x_3 in Equation D.12.

$$\begin{aligned} x_3 = \frac{X_3}{Z_3} &= \left(\frac{A}{BZ_1}\right)^2 + \left(\frac{A}{BZ_1}\right) + \frac{X_1}{Z_1} + x_2 + a \\ &= \frac{A^2 + ABZ_1 + B^2X_1Z_1 + B^2x_2Z_1^2 + aB^2{Z_1}^2}{(BZ_1)^2} \end{aligned}$$

Therefore,

$$Z_3 = (BZ_1)^2 = C^2 \tag{D.17}$$

and,

$$\begin{aligned} X_3 &= A^2 + AC + B^2X_1Z_1 + B^2x_2{Z_1}^2 + aB^2{Z_1}^2 \\ &= A^2 + AC + B^2(Z_1(X_1 + x_2Z_1) + a{Z_1}^2) \\ &= A^2 + AC + B^2(Z_1B + a{Z_1}^2) \end{aligned}$$

Let, $\mathbf{E} = AC$ and $\mathbf{D} = B^2(Z_1B + a{Z_1}^2)$, then

$$X_3 = A^2 + E + D \tag{D.18}$$

Consider the equation for y_3 in Equation D.12.

$$\begin{aligned} y_3 = \frac{Y_3}{{Z_3}^2} &= \frac{A}{Z_1B}\left(\frac{X_1}{Z_1} + \frac{X_3}{Z_3}\right) + \left(\frac{X_3}{Z_3}\right) + \left(\frac{Y_1}{{Z_1}^2}\right) \\ &= \frac{AB^3X_1{Z_1}^2 + ABX_3Z_1 + X_3Z_3 + B^4Y_1{Z_1}^2}{{Z_3}^2} \\ Y_3 &= AB^3X_1{Z_1}^2 + ABX_3Z_1 + X_3Z_3 + B^4Y_1{Z_1}^2 \end{aligned}$$

Substituting $X_1 = B + x_2Z_1$ and $E = ABZ_1$ we get

$$\begin{aligned} Y_3 &= (B + x_2Z_1)AB^3{Z_1}^2 + EX_3 + X_3Z_3 + B^4Y_1{Z_1}^2 \\ &= (AB^4{Z_1}^2 + Ex_2Z_3) + EX_3 + X_3Z_3 + B^4Y_1{Z_1}^2 \\ &= (y_2{Z_1}^2 + Y_1)B^4{Z_1}^2 + Ex_2Z_3 + EX_3 + X_3Z_3 + B^4Y_1{Z_1}^2 \\ &= y_2{Z_3}^2 + Ex_2Z_3 + EX_3 + X_3Z_3 \end{aligned}$$

Let **F** $= X_3 + x_2Z_3$ and **G** $= (x_2 + y_2){Z_3}^2$.

$$\begin{aligned} Y_3 &= (G + x_2{Z_3}^2) + Ex_2Z_3 + EX_3 + X_3Z_3 \\ Y_3 &= G + F(E + Z_3) \end{aligned} \tag{D.19}$$

D.2.3 Point Doubling

The x_3 equation in D.16 can be rewritten as follow.

$$\begin{aligned} x_3 &= \left(x_1 + \frac{y_1}{x_1}\right)^2 + \left(x_1 + \frac{y_1}{x_1}\right) + a \\ &= \frac{x_1^4 + y_1^2 + x_1^3 + x_1y_1 + ax_1^2}{x_1^2} \end{aligned} \tag{D.20}$$

From Equation D.1

$$b = x_1^3 + y_1^2 + x_1y_1 + ax_1^2$$

Substituting in Equation D.20

$$x_3 = x_1^2 + \frac{b}{x_1^2} \tag{D.21}$$

Convert x_1 to X_1/Z_1 and x_3 to X_3/Z_3.

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

Therefore,

$$X_3 = X_1^4 + bZ_1^4$$
$$Z_3 = X_1^2 Z_1^2$$

The y_3 equation in D.16 can be represented by the following.

$$\begin{aligned} y_3 &= x_1^2 + \Big(x_1 + \frac{y_1}{x_1}\Big)x_3 + x_3 \\ &= (x_1^2 + x_3) + \Big(\frac{x_1^3 + x_1 y_1}{x_1^2}\Big)x_3 \end{aligned}$$

From Equations for D.21 and D.1,

$$y_3 = \frac{b}{x_1^2} + \Big(\frac{y_1^2 + ax_1^2 + b}{x_1^2}\Big)x_3$$

Converting this equation to projective coordinates by changing y_3 to Y_3/Z_3^2, and y_1 to Y_1/Z_1^2.

$$\begin{aligned} \frac{Y_3}{Z_3^2} &= \frac{bZ_1^2}{X_1^2} + \Big(\frac{Y_1^2}{X_1^2 Z_1^2} + a + b\frac{Z_1^2}{X_1^2}\Big)\frac{X_3}{Z_3} \\ \frac{Y_3}{Z_3^2} &= \frac{bZ_1^4 Z_3 + (Y_1^2 + aX_1^2 Z_1^2 + bZ_1^4)X_3}{Z_3^2} \end{aligned}$$

Therefore

$$Y_3 = bZ_1^4 Z_3 + (Y_1^2 + aX_1^2 Z_1^2 + bZ_1^4)X_3$$

APPENDIX E

Gates Requirements for the Simple Karatsuba Multiplier

This appendix determines the estimates of AND and XOR gates for the simple Karatsuba multiplier.

E.1 Gate Requirements for the Basic Karatsuba Multiplier

E.1.1 AND Gate Estimate

For an $m = 2^k$ bit basic Karatsuba multiplier, the first recursion splits the m bit multiplicands into $m/2$ bits. Three $m/2 = 2^{k-1}$ bit multipliers are then required. The second recursion has nine $m/4 = 2^{k-2}$ bit multipliers. The i^{th} recursion has 3^i multipliers with each multiplier being $m/2^i = 2^{k-i}$ bits in length. There are $k = log_2 m$ such recursions. The final recursion containing two bit multiplications has $3^{log_2 m}$ multipliers. In the final recursion each multiplication is done using a single AND gates. Therefore,

$$\#AND \text{ gates} : 3^{log_2 m} \tag{E.1}$$

E.1.2 XOR Gate Estimate

Let A and B be the two $m = 2^k$ bit multiplicands. In the first recursion, the multiplicands are split into two halves. Let the higher bits be A_h and B_h and the lower bits

Table E.1: Combining the Partial Products

$4n-4$ to $3n-1$	$3n-2$ to $2n$	$2n-1$	$2n-2$ to n	$n-1$ to 0
-	-	-	M_l	M_l
-	M_l	M_l	M_l	-
-	M_h	M_h	M_h	-
-	M_{hl}	M_{hl}	M_{hl}	-
M_h	M_h	M_h	-	-

be A_l and B_l. The three $m/2$ bit multiplications that are performed are $M_h = A_hB_h$, $M_l = A_lB_l$ and $M_{hl} = (A_h + A_l)(B_h + B_l)$. Let $n = m/2$. Forming the terms $A_h + A_l$ requires n XOR gates. Similarly the terms $B_h + B_l$ requires n XOR gates. In all, $2n$ XORs are required. After the three multiplications are completed, the partial products are added as shown in the Table E.1. The columns in the table show the output bits of the multiplier and partial products that need to be combined to form the output bit. Combining the terms $(2n-2)$ to n requires $3(n-1)$ XOR gates. Similarly the terms from $(3n-2)$ to $2n$ require $3(n-1)$ XOR gates. Combining the terms $(2n-1)$ requires 2 XOR gates. Thus, the total number of XOR gates required for combining the partial products is $6n-4$, and the number of XOR gates required is $6n-4+2n = 4m-4$. Since $m/2^r$ is the length of the multiplier in the r^{th} recursion, the number of XOR gates required in the r^{th} recursion is $4(m/2^r) - 4$. Adding up the XOR gates required for all the recursions gives the XOR gate estimate (Equation E.2.

$$\#XOR \text{ gates : } \sum_{r=0}^{log_2 m} 3^r \Big(4m/2^r - 4\Big) \tag{E.2}$$

E.2 Gate Requirements for the Simple Karatsuba Multiplier

The simple Karatsuba is basically the basic Karatsuba multiplier with a small modification to handle bit lengths of the form $m \neq 2^k$. The number of XOR and AND gates for the basic Karatsuba multiplier form the upper bound for the number of gates required by the simple Karatsuba multiplier. Therefore,

$$\begin{aligned} \#AND\,\text{gates} &: 3^{\lceil log_2 m \rceil} \\ \#XOR\,\text{gates} &: \sum_{r=0}^{\lceil log_2 m \rceil} 3^r \Big(4\lceil m/2^r \rceil - 4\Big) \end{aligned} \tag{E.3}$$

REFERENCES

[1] W. N. Chelton and M. Benaissa, “Fast Elliptic Curve Cryptography on FPGA,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 198–205, Feb. 2008.

[2] RSA Laboratories, “RSA Cryptograhy Standard,” 2002.

[3] Paul C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” in *CRYPTO ’96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, London, UK, 1996, pp. 104–113, Springer-Verlag.

[4] Paul Kocher, Joshua Jaffe, and Benjamin Jun, “Differential Power Analysis,” *Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[5] Mitsuru Matsui and Junko Nakajima, “On the Power of Bitslice Implementation on Intel Core2 Processor,” in *CHES*, 2007, pp. 121–134.

[6] Thomas Wollinger, Jan Pelzl, Volker Wittelsberger, Christof Paar, Gökay Saldamli, and Çetin K. Koç, “Elliptic and Hyperelliptic Curves on Embedded μP,” *Trans. on Embedded Computing Sys.*, vol. 3, no. 3, pp. 509–533, 2004.

[7] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi, “Bitslice Implementation of AES,” in *CANS*, 2006, pp. 203–212.

[8] Robert Konighofer, “A Fast and Cache-Timing Resistant Implementation of the AES,” in *Topics in Cryptology CT-RSA 2008*. 2008, pp. 187–202, Springer Berlin / Heidelberg.

[9] Lawrence C. Washington, *Elliptic Curves: Number Theory and Cryptography*, CRC Press, Inc., Boca Raton, FL, USA, 2003.

[10] Victor Miller, “Uses of Elliptic Curves in Cryptography,” *Advances in Cryptology, Crypto’85*, vol. 218, pp. 417–426, 1986.

[11] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 2001.

[12] Anatoly A. Karatsuba and Y. Ofman, “Multiplication of Multidigit Numbers on Automata,” *Soviet Physics Doklady*, vol. 7, pp. 595–596, 1963.

[13] Toshiya Itoh and Shigeo Tsujii, "A Fast Algorithm For Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases," *Inf. Comput.*, vol. 78, no. 3, pp. 171–177, 1988.

[14] U.S. Department of Commerce,National Institute of Standards and Technology, "Digital signature standard (DSS)," 2000.

[15] Xilinx, *Virtex-4 User Guide*, 2007.

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

[17] Whitfield Diffie and Martin E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976.

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

[19] Neal Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[20] IEEE Computer Society, "IEEE Standard Specifications for Public-key Cryptography," 2000.

[21] American National Standards Institute, "Public Key Cryptography for the Financial Service Industry : The Elliptic Curve Digital Signature Algorithm (ECDSA)," 1998.

[22] N Mazzocca A. Cilardo, L Coppolino and L Romano, "Elliptic Curve Cryptography Engineering," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 395–406, Feb 2006.

[23] Johannes Wolkerstorfer, *Hardware Aspects of Elliptic Curve Cryptography*, Ph.D. thesis, Institute for Applied Information Processing and Communications, Graz University of Technology, 2004.

[24] Thomas Wollinger, Jorge Guajardo, and Christof Paar, "Security on FPGAs: State-of-the-art Implementations and Attacks," *Trans. on Embedded Computing Sys.*, vol. 3, no. 3, pp. 534–574, 2004.

[25] Deming Chen, Jason Cong, and Peichen Pan, "FPGA Design Automation: A Survey," *Found. Trends Electron. Des. Autom.*, vol. 1, no. 3, pp. 139–169, 2006.

[26] Takashi Horiyama, Masaki Nakanishi, Hirotsugu Kajihara, and Shinji Kimura, "Folding of Logic Functions and its Application to Look Up Table Compaction," *ICCAD*, vol. 00, pp. 694–697, 2002.

[27] Michael Hutton, Jay Schleicher, David M. Lewis, Bruce Pedersen, Richard Yuan, Sinan Kaptanoglu, Gregg Baeckler, Boris Ratchev, Ketan Padalia, Mark Bourgeault, Andy Lee, Henry Kim, and Rahul Saini, "Improving FPGA Performance and Area Using an Adaptive Logic Module," in *FPL*, 2004, pp. 135–144.

[28] Eli Biham and Adi Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, London, UK, 1997, pp. 513–525, Springer-Verlag.

[29] Gerardo Orlando and Christof Paar, "A High Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$," in *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, London, UK, 2000, pp. 41–56, Springer-Verlag.

[30] Julio López and Ricardo Dahab, "Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$," in *SAC '98: Proceedings of the Selected Areas in Cryptography*, London, UK, 1999, pp. 201–212, Springer-Verlag.

[31] Leilei Song and Keshab K. Parhi, "Low-Energy Digit-Serial/Parallel Finite Field Multipliers," *J. VLSI Signal Process. Syst.*, vol. 19, no. 2, pp. 149–166, 1998.

[32] Tim Kerins, Emanuel Popovici, William P. Marnane, and Patrick Fitzpatrick, "Fully Parameterizable Elliptic Curve Cryptography Processor over $GF(2)$," in *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, London, UK, 2002, pp. 750–759, Springer-Verlag.

[33] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, 2002, pp. 157–164.

[34] Nils Gura, Sheueling Chang Shantz, Hans Eberle, Sumit Gupta, Vipul Gupta, Daniel Finchelstein, Edouard Goupy, and Douglas Stebila, "An End-to-End Systems Approach to Elliptic Curve Cryptography," in *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, London, UK, 2003, pp. 349–365, Springer-Verlag.

[35] Jonathan Lutz and Anwarul Hasan, "High Performance FPGA based Elliptic Curve Cryptographic Co-Processor," in *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, Washington, DC, USA, 2004, p. 486, IEEE Computer Society.

[36] Jerome A. Solinas, "Efficient Arithmetic on Koblitz Curves," *Des. Codes Cryptography*, vol. 19, no. 2-3, pp. 195–249, 2000.

[37] N. A. Saqib, F. Rodríiguez-Henríquez, and A. Diaz-Perez, "A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication Over $GF(2^m)$," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*, Apr. 2004.

[38] Qiong Pu and Jianhua Huang, "A Microcoded Elliptic Curve Processor for $GF(2^m)$ Using FPGA Technology," in *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, June 2006, vol. 4, pp. 2771–2775.

[39] Xilinx, "Using Block RAM in Spartan-3 Generation FPGAs," Application Note, XAPP-463, 2005.

[40] Bijan Ansari and M. Anwar Hasan, "High Performance Architecture of Elliptic Curve Scalar Multiplication," Tech. Rep., Department of Electrical and Computer Engineering, University of Waterloo, 2006.

[41] John B. Fraleigh, *First Course in Abstract Algebra*, Addison-Wesley, Boston, MA, USA, 2002.

[42] William Stallings, *Cryptography and Network Security (4th Edition)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[43] Christof Paar, *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*, Ph.D. thesis, Institute for Experimental Mathematics, Universität Essen, Germany, June 1994.

[44] Francisco Rodríguez-Henríquez, N. A. Saqib, A. Díaz-Pèrez, and Çetin Kaya Ķoc, *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[45] Gregory C. Ahlquist, Brent E. Nelson, and Michael Rice, "Optimal Finite Field Multipliers for FPGAs," in *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, London, UK, 1999, pp. 51–60, Springer-Verlag.

[46] Ç. K. Koç and B. Sunar, "An Efficient Optimal Normal Basis Type II Multiplier," *IEEE Trans. Comput.*, vol. 50, no. 1, pp. 83–87, 2001.

[47] Çetin K. Koç and Tolga Acar, "Montgomery Multiplication in $GF(2^k)$," *DES Codes Cryptography*, vol. 14, no. 1, pp. 57–69, 1998.

[48] C. Grabbe, M. Bednara, J. Shokrollahi, J. Teich, and J. von zur Gathen, "FPGA Designs of Parallel High Performance $GF(2^{233})$ Multipliers," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS-03)*, Bangkok, Thailand, May 2003, vol. II, pp. 268–271.

[49] Zoya Dyka and Peter Langendoerfer, “Area Efficient Hardware Implementation of Elliptic Curve Cryptography by Iteratively Applying Karatsuba’s Method,” in *DATE ’05: Proceedings of the conference on Design, Automation and Test in Europe*, Washington, DC, USA, 2005, pp. 70–75, IEEE Computer Society.

[50] Joachim von zur Gathen and Jamshid Shokrollahi, “Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F_2,” in *Selected Areas in Cryptography*, 2005, pp. 359–369.

[51] Steffen Peter and Peter Langendörfer, “An efficient polynomial multiplier in $GF(2^m)$ and its application to ECC designs,” in *DATE ’07: Proceedings of the conference on Design, automation and test in Europe*, San Jose, CA, USA, 2007, pp. 1253–1258, EDA Consortium.

[52] Christof Paar, “A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields,” *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 856–861, 1996.

[53] Francisco Rodríguez-Henríquez and Çetin Kaya Koç, “On Fully Parallel Karatsuba Multipliers for $GF(2^m)$,” in *Proc. of the International Conference on Computer Science and Technology (CST)*, 2003, pp. 405–410.

[54] Peter L. Montgomery, “Five, Six, and Seven-Term Karatsuba-Like Formulae,” *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 362–369, 2005.

[55] André Weimerskirch and Christof Paar, “Generalizations of the Karatsuba Algorithm for Efficient Implementations,” Cryptology ePrint Archive, Report 2006/224, 2006.

[56] Burton S. Kaliski, “The Montgomery Inverse and its Applications,” *IEEE Transactions on Computers*, vol. 44, no. 8, pp. 1064–1065, 1995.

[57] Jorge Guajardo and Christof Paar, “Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes,” *Des. Codes Cryptography*, vol. 25, no. 2, pp. 207–216, 2002.

[58] Francisco Rodríguez-Henríquez, Nazar A. Saqib, and Nareli Cruz-Cortés, “A Fast Implementation of Multiplicative Inversion Over $GF(2^m)$,” in *ITCC ’05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’05) - Volume I*, Washington, DC, USA, 2005, pp. 574–579, IEEE Computer Society.

[59] Francisco Rodríguez-Henríquez, Guillermo Morales-Luna, Nazar A. Saqib, and Nareli Cruz-Cortés, “Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for a Special Class of Trinomials,” *Des. Codes Cryptography*, vol. 45, no. 1, pp. 19–37, 2007.

[60] Donald E. Knuth, *The Art of Computer Programming Volumes 1-3 Boxed Set*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[61] Xilinx, "Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs," Application Note, XAPP-464, 2005.

[62] Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater, "An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA," in *International Conference on Field-Programmable Technology - FPT 2004*, O. Diessel and J.A. Williams, Eds., 2004, pp. 441–444.

[63] F. Crowe, A. Daly, and W. Marnane, "Optimised Montgomery Domain Inversion on FPGA," in *Circuit Theory and Design, 2005. Proceedings of the 2005 European Conference on*, Aug./Sept. 2005, vol. 1.

[64] Sabel Mercurio Henríquez Rodríguez and Francisco Rodríguez-Henríquez, "An FPGA Arithmetic Logic Unit for Computing Scalar Multiplication using the Half-and-Add Method," in *ReConFig 2005: International Conference on Reconfigurable Computing and FPGAs*, Washington, DC, USA, 2005, IEEE Computer Society.

[65] Nele Mentens, Siddika Berna Ors, and Bart Preneel, "An FPGA Implementation of an Elliptic Curve Processor $GF(2^m)$," in *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, New York, NY, USA, 2004, pp. 454–457, ACM.

[66] Jean-Sébastien Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," in *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, London, UK, 1999, pp. 292–302, Springer-Verlag.

[67] Michael Rosing, *Implementing Elliptic Curve Cryptography*, Manning Publications Co, Sound View Ct. 3B Greenwich, CT 06830, 1998.

PUBLICATIONS AND AWARDS BASED ON THESIS

Publications

1. Chester Rebeiro and Debdeep Mukhopadhyay, "Hybrid Masked Karatsuba Multiplier for $GF(2^{233})$" in *Proceedings of the* 11^{th} *IEEE VLSI Design and Test Symposium*, Kolkata, August 2007, pp 379-387, VLSI Society of India.

2. Chester Rebeiro and Debdeep Mukhopadhyay, "Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier" in *Proceedings of the* 21^{st} *International Conference on VLSI Design*, Hyderabad, January 2008, pp 706–711, IEEE Computer Society.

3. Chester Rebeiro and Debdeep Mukhopadhyay, "High Performance Elliptic Curve Crypto Processor for FPGA Platforms" in *Proceedings of the* 12^{th} *IEEE VLSI Design and Test Symposium*, Bangalore, July 2008, pp. 107–117, VLSI Society of India.

4. Chester Rebeiro and Debdeep Mukhopadhyay, "High Speed Compact Elliptic Curve Cryptoprocessor for FPGA Platforms" in *INDOCRYPT 2008 :* 9^{th} *International Conference of Cryptology in India*, Kharagpur, December 2008, pp. 376–388, Springer-Verlag.

Awards

1. Chester Rebeiro and Debdeep Mukhopadhyay won the second prize at the design contest conducted by the 22^{nd} International Conference on VLSI Design, New Delhi, January 2009. The entry was titled "High Performance Galois Field Elliptic Curve Cryptographic Processor for FPGA Platforms".