A reconfigurable and interoperable hardware architecture for elliptic curve cryptography

por

Miguel Morales-Sandoval

Tesis sometida como requisito parcial para obtener el grado de

DOCTOR EN CIENCIAS EN LA ESPECIALIDAD DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

DICIEMBRE, 2008

Tonantzintla, Puebla

Supervisada por:

Dra. Claudia Feregrino-Uribe, INAOE

Instituto Nacional de Astrofísica, Óptica y Electrónica

Consejo Nacional de Ciencia y Tecnología

A reconfigurable and interoperable hardware architecture for elliptic curve cryptography

a

THESIS

by

MSc. Miguel Morales Sandoval

Presented and defended on December 15, 2008, in the

National Institute for Astrophysics, Optics and Electronics, Puebla

to obtain the degree of

Doctor of Philosophy

in

Computer Science

Coordinación de Ciencias Computacionales

To my family, my son Miguel Alejandro and wife Heidy Marisol, for their endless love.

To my mother, to my sisters and brothers who always have supported me.

In memoriam of my father† José Melitón Morales Ramos

Acknowledgment

To the Consejo Nacional de Ciencia y Tecnología (CONACyT) for financial support through scholarship number 171577. To the Instituto Nacional de Atrofísica, Óptica y Electrónica (INAOE) for the services and support provided to accomplish this research.

To my advisor Dra. Claudia Feregrino-Uribe for her reviews, help, guidance and encouragement in conducting this research. I count myself as being the most fortunate to be able to work under her supervision. To my graduate committee, Dr. René Cumplido Parra, Dr. Miguel Arias Estrada, Dr. Luis Villaseñor Pineda, Dr. Gustavo Rodríguez Gómez and Dr. Paskevas Kitsos, for their valuable input to improve this research. To Prof. Christof Paar for allowing me to visit the COSY group in Ruhr University, in Bochum Germany.

To my wife Heidy Marisol, for her encouragement and inspiration provided during these years to conclude this research. My son Miguel Alejandro has made my life much more cheerful and colorful. To my family for their love and support. To my sisters Lucila and Esperanza, and my brothers Victor and Alfredo. My mother Anastacia for her love, I can never thank her enough.

Many more persons participated in various ways to ensure my research succeeded and I am thankful to them all.

Abstract

Elliptic curve cryptography (ECC) is a kind of Public Key Cryptography founded in the theory of groups. ECC's main advantage is the short length of the keys used compared to the key used by traditional public key cryptosystems like RSA (163bits vs 1024bits) without decreasing the security level. The use of shorter length keys implies less space for key storage, time saving when keys are transmitted and less costly arithmetic computations.

An ECC cryptosystem is defined as the tuple $T = (GF(q), a, b, G, n, h)$, where $GF(q)$ is a finite field, a and b define an elliptic curve on $GF(q)$, G is a generator point of the elliptic curve, n is the order of G, that is, the smaller integer such that $nG = O$ (identity point in the additive group). h is called the co-factor and it is equal to the total number of points in the curve divided by n. Two entities that implement security services like confidentiality, integrity or authentication must agree previously the same set of parameters T in order to interoperate.

ECC-based cryptographic algorithms such as encryption or digital signatures are computationally expensive because several finite field and elliptic curve operation with long number must be carried out. Although a software implementation of ECC could provide interoperability, the resulting processing time will be unacceptable. Proposed work in the literature is for efficient implementation of ECC in hardware, however, most of those works are custom implementations for specific tuples T.

This thesis deals with the interoperability problems of ECC and presents the results of the development of a hardware architecture that allows to adapt dynamically its organization to operate with different parameters T. The development of such architecture is hard due the diversity of parameters T and the complexity of the underlaying algorithms. Although some reported works allow some flexibility in the choice of the ECC parameters, a reconfigurable architecture that provides

interoperability with another implementation is not explored at all. An immediate application of the architecture developed is for IPSec, a security protocol where the cryptographic algorithms and its parameters are negotiated at run time.

The reconfigurable computing paradigm was used in this thesis work. Due a general design methodology for reconfigurable system is not available, this thesis explores and evaluates techniques for developing interoperable ECC hardware architectures. This thesis was developed in three stages: *i*) the first one consisted on the design of a base hardware architecture for evaluating several cryptographic algorithms in order to find the best circuits that produce a compact design without compromising performance; *ii*) the second stage consisted on providing the architecture with reconfigurability capabilities, that enable the architecture to adapt itself to different sets of parameters T at run time; *iii*) finally the third stage consisted on the architecture validation, which is performed by simulating the design and applying test vectors. Validation was also carried out in-circuit.

The main contributions of this thesis are: *i*) a hardware architecture for ECC that allows interoperability; *ii*) a reconfiguration strategy for developing interoperable ECC architectures; and *iii*) an study of finite field arithmetic algorithms performance that allows to establish a trade-off in the architecture.

Resumen

Criptografía de curvas elípticas (ECC) es un tipo de criptografía de llave pública fundada en la teoría de campos finitos y el problema del logaritmo discreto. La principal ventaja de este tipo de criptografía frente a otros tipos como RSA, es el uso de llaves más cortas, con una reducción de hasta 7 veces. El uso de llaves cortas tiene las ventajas de utilizar menos requerimientos de memoria, de realizar operaciones aritméticas con operandos más cortos, de utilizar menos recursos de área si la implementación se realiza en hardware, de requerir menores tiempos de transferencia, entre otras.

Los parámetros de implementación de ECC son de la forma $T = (q, E, G, n, h)$, donde q es un campo finito, E es una curva elíptica definida en q, G es un generador de la curva elíptica, n es el orden de G y h es el co-factor de la curva E. ECC es demandante computacionalmente ya que requiere de varias operaciones aritméticas en campos finitos y curvas elípticas para implementar los esquemas de seguridad, tales como el cifrado y la firma digital. Aunque una implementación de ECC en software podría proveer interoperabilidad, el tiempo de procesamiento es inaceptable. Para acelerar los tiempos de procesamiento de ECC se han propuesto en la literatura varias implementaciones de ECC en hardware, pero optimizadas para un conjunto de parámetros T particulares, incrementando así los problemas de interoperabilidad.

Esta investigación aborda el problema de interoperabilidad en ECC desarrollando una arquitectura hardware que permita una adaptacin dinámica de su estructura para poder operar con diferentes parámetros T y que al mismo tiempo mantenga el alto desempeño de una implementación en hardware. Una implementación ECC interoperable es difícil debido a la diversidad de parámetros T que existen y a la complejidad de los algoritmos subyacentes. Aunque algunos trabajos reportados permiten cierta flexibilidad en la elección de los parámetros,

una arquitectura hardware reconfigurable que provea interoperabilidad con otras implementaciones no se ha explorado aún. Una aplicación inmediata de esta nueva arquitectura es en el protocolo IPSec, donde los algoritmos criptográficos como ECC y sus parámetros de implementación se negocian en tiempo de ejecución.

Para llevar a cabo el diseño, implementación y evaluación de la arquitectura hardware interoperable, se propone la aplicación del cómputo reconfigurable, donde el hardware puede modificarse dinámicamente. Dado que no existe una metodología general de diseño para sistemas reconfigurables, en esta tesis se explora y evalúa una nueva metodología de diseño de sistemas ECC interoperables. El desarrollo de esta investigación se realizó en tres etapas: *i*) la primera etapa consistió en el diseño y evaluación de una arquitectura hardware ECC base para el estudio y evaluación de algoritmos propuestos en la literatura a fin de encontrar los mejores circuitos que lleven a tener mínimos requerimientos de área y altos desempeños; *ii*) la segunda etapa consistió en incorporar a la arquitectura ECC base la capacidad de reconfiguración. Esta capacidad permite que la arquitectura hardware ECC se adapte en tiempo de ejecución a diferentes parámetros T y por tanto que pueda proveer interoperabilidad; *iii*) la tercera etapa consistió en la validación de la arquitectura, la cual se realiza mediante la aplicación de vectores de prueba. La validación se realiza mediante simulación funcional del hardware descrito en VHDL, así como también mediante verificación en el circuito.

Las aportaciones de este trabajo de investigación son: *i*) Una arquitectura hardware para ECC que permite interoperabilidad; *ii*) Una estrategia de reconfiguración para arquitecturas ECC interoperables; y *iii*) Un estudio de los algoritmos aritméticos en campo finito que mejor se desempeñen. Este estudio permite establecer un compromiso área-desempeño en la arquitectura.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces concepts related to this dissertation and states the tracked problem. It presents the motivation for this work and lists the main and specific objectives pursed in this thesis. The next section introduces cryptography and terms related to it. The introductory material presented in this chapter could be extended in [3, 4, 5, 6].

1.1 Information security and cryptography

Security mechanisms to protect sensitive information have been required since ancient times. Digital form of information has made more complicated the way to keep it secure due it is more easy to access and handle. Information security services are provided by cryptography, a discipline of mathematics and, in modern times, of computer science [3]. Cryptography is an interdisciplinary subject that makes extensive use of mathematics, including aspects of information theory, computational complexity, statistics, combinatorics, and especially number theory. Cryptography is used in many applications that touch everyday life; the security of ATM cards and electronic commerce depend on cryptography.

Cryptography provides the information security services of confidentiality, authentication, integrity, and no-repudiation. Confidentiality is provided by private key cryptography (also known as *Symmetric Key Cryptography* or SKC) by the encryption and decryption operations [3]. The four security services can be provided by Public Key Cryptography (PKC), but this kind of cryptography is mainly used to provide the authentication and no-repudiation services by implementing

Table 1.1: Cryptographic algorithms

Kind of cryptography	**Examples**
Hash Functions	MD4-5, SHA-0-1-2, RIPDEM
Symmetric Key Cryptography	DES, AES, 3DES, RC4
Public Key Cryptography	ECC, RSA, DSA, ElGammal

Table 1.2: Key sizes for cryptographic algorithms [1]

Private key size	**Public key size (bits)**		**MIPS**	**Protection**
(bits)	ECC	RSA/DH/DSA	to attack	lifetime
80	160	1024	10^{12}	until 2010
112	224	2048	10^{24}	until 2030
128	256	3072	10^{28}	beyond 2031
192	384	7680	10^{47}	
256	512	15360	10^{66}	

the concept of *digital signatures*. The *hash functions* are cryptographic primitives often used along with public or private key algorithms to provide the integrity service. Examples of hash functions, public key and private key algorithms are given in table 1.1.

SKC and PKC algorithms rely on the use of a key or a pair of keys. A key is a n-bit string that is used to transform data. The size in bits of the key is an important security parameter in the cryptographic algorithms. Table 1.2 shows the key sizes for different SKC and PKC cryptographic algorithms with equivalent security level.

SKC algorithms use the same key to encrypt and decrypt data while PKC uses two different keys, one for encryption (public key) and other for decryption (private key). The use of two different but related keys eliminates the problem in SKC of managing N^2 keys for a network of N nodes. It is well known in the literature that PKC algorithms provide stronger security but they are more complex and slower than the symmetric ones. Symmetric algorithms are faster to encrypt and decrypt but cannot offer the authentication and no-repudiation services, so a combination of SKC and PKC is used in practice; PKC to derive a shared secret key and SKC to provide faster encryption and decryption.

Different to symmetric key cryptography, where encryption and decryption operations are carried out by permutations and transpositions, in public key cryp-

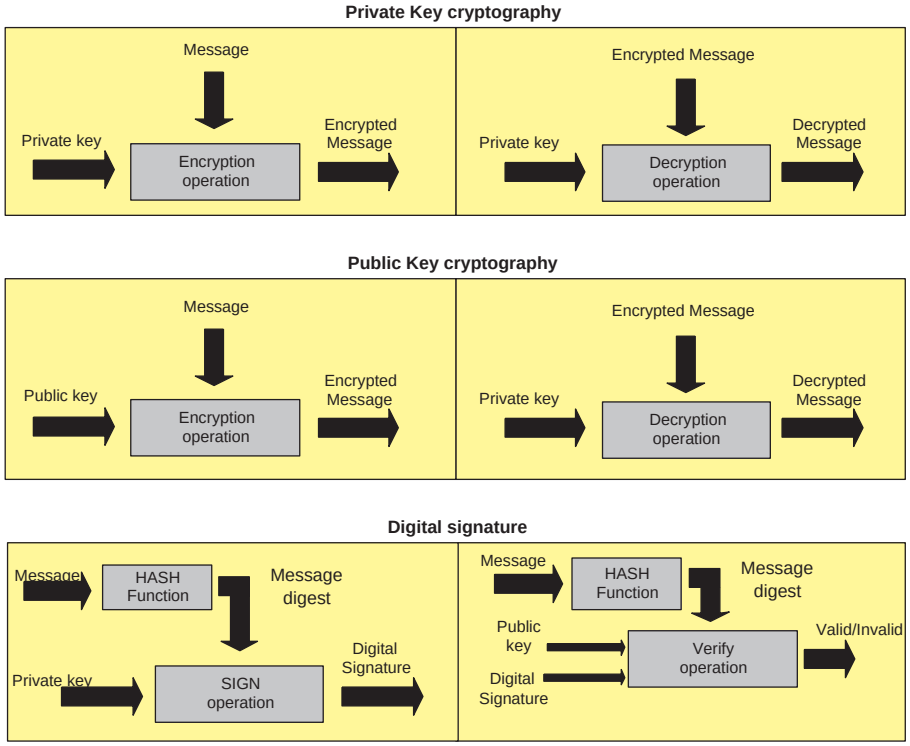


Figure 1.1: Cryptographic operations

tography almost all methods transform data by executing arithmetic operations in mathematical structures like finite fields or elliptic curves. In public key cryptosystems confidentiality is achieved by encrypting data with the public key of the receiver. Then, the receiver, which is the unique posesor of the associated private key, can decrypt data and recover the original information. The integrity, authentication and non-repudiation services are provided by implementing the concept of digital signature. A *digital signature* is the analog to the handwritten signature. An user signs a piece of information using his private key, and any other user can verify the authenticity of the signature, and hence of information, using the signer's public key. Private key, public key and digital signature operations are showed in figure 1.1.

In PKC, private and public keys have a mathematical relation f, but the private key can not be obtained from the public one. In order to recover the private key to decrypt data or to sign documents, a mathematical problem P related to f must be solved. The security of public key cryptosystems depends on the difficulty to solve P. In practice, three problems have been considered to

Table 1.3: Public key cryptosystems and their underlying mathematical problems

Mathematical Problem	**Description**	**Cryptosystems**
Integer factorization	Given a number n, find its prime factors	RSA, Rabin-Williams
Discrete logarithm	Given a prime n, and numbers g and h, find x such that $h = g^x \mod n$	ElGamal, DSA Hellman-Diffie
Elliptic curve discrete logarithm	Given an elliptic curve E and points P and Q on E, find x such that $Q = xP$	ECDSA, EC-Diffie-Hellman

Table 1.4: Complexity of mathematical problems in public key cryptography

Public-key system	**Best known methods for solving mathematical problem**	**Running times**
Integer factorization	Number field sieve: $e^{1.923}(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}$	Sub-exponential
Discrete logarithm	Number field sieve: $e^{1.923}(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}$	Sub-exponential
Elliptic curve discrete logarithm	Pollard-rho algorithm: $\sqrt{n}$	Fully exponential

be difficult to solve and are used for cryptographic applications. Table 1.3 lists these problems and the cryptosystems that rely their security on such problems. Table 1.4 shows the computational complexities for each of these problems.

In table 1.4, n is the size of the keys used. The sub-exponential complexity of the problem on which RSA and other public key methods base their security means that the problem can be considered hard to solve but not as hard as problems that only allow fully exponential solutions, as elliptic curve cryptography. Because of this, ECC can offer a similar security level than other public key cryptosystems but using shorter length keys, which implies less space for key storage, time saving when keys are transmitted and less costly modular computations. These characteristics make ECC the best choice for securing devices with constrained resources, like the mobile ones.

ECC bases its security in the difficulty to solve the *discrete logarithm problem*. In general, the discrete logarithm problem is defined on a finite group $\Gamma = \{S, \diamond\}$. A group is a closed set of elements S under the binary operator $\diamond$ that satisfies axioms (closure, associativity, commutativity, neutral element, inverse) [6]. For a group element $e \in S$ and a number k, e^k denote the element obtained by applying operation $\diamond$, $k-1$ times to e ($e^2 = e \diamond e$, $e^3 = e \diamond e \diamond e$, and so on). Under

Table 1.5: Elliptic curve cryptographic schemes approved

Crypto scheme	**NIST [7]**	**ANSI [8]**	**IEEE [9]**
Key agreement	ECDH, ECMQV	ECDH, ECMQV	ECDH, ECMQV
Digital Signature	ECDSA	ECDSA	ECDSA
Encryption		ECIES	

these conditions, the discrete logarithm problem is defined as follows: given an element $e_1 \in S$ and another element $e_2 \in S$, find an integer k such that $e_1^k = e_2$. The difficulty to solve this problem increases as the number of elements in S gets larger. In Γ, a generator element g in S always exists, that is, each element e in S is equal to g^k for some positive integer k.

In elliptic curve cryptography, the discrete logarithm problem is defined on the group $\Gamma = \{S, \diamond\}$, where S is the set of points of an elliptic curve and the group operation $\diamond$ is the sum of points. In this case, Γ is denominated an additive group. The operation e^k is now interpreted as $e + e + e + ... + e = ke$. Here, e is a point $P = (x, y)$ that satisfies an elliptic curve equation $f(x, y) = 0$ defined on a finite field. The discrete logarithm problem is now defined as: given the points $P, Q \in S$, find the integer d such that $Q = dP$. The operation dP is called *scalar multiplication*. Again, the difficulty to solve this conjectured difficult problem increases as the number of elliptic curve points also increases.

Elliptic curve cryptography provides the security services of confidentiality, authentication, integrity and no-repudiation throughout cryptographic schemes for key agreement, digital signatures and bulk encryption. Some of these schemes are recommended in standards and classified accordingly, as shown in table 1.5.

An elliptic curve cryptosystem is defined as a set of parameters $T = (K, E(K), G, n, h)$, where:

- K is a finite field, either the binary field GF(2^m) or the prime field GF(p).
- $E(K)$ is an elliptic curve on K.
- G is a generator point of the elliptic curve.
- n is the order of G, that is, the smaller integer such that $nG = O$ (identity point in the additive group Γ).
- h is called the co-factor, the total number of points in the curve divided by n.

The public key in ECC schemes is an elliptic curve point $P = (x, y)$ while

the private key is an integer in the range [1, n-1]. More details about the tuple T and each one of its elements are presented in section 2.2.3. ECC schemes for encryption, decryption and digital signature generation and verification are also presented in section 2.2.3.

1.2 Motivation

Future communications systems are expected to enable interaction between an increasingly diverse range of devices, both mobile and fixed [10]. This will allow users to construct their own personal distributed environments using a combination of different communications technologies. Depending on the services availability, the communication configuration could be, for instance, via a cellular system; a wireless *ad hoc* network; a digital broadcast system; or a fixed telephone network. *Interoperability* and *security* are two main requirements in this heterogeneous inter-networked environment. To achieve interoperability, it is necessary that all participants in a communication network have the standardized versions of common security protocols. Often, interoperability is reduced considerably to increase performance and security.

ECC has the important feature of using shorter keys while keeping the same security level compared to traditional public key cryptosystems, like RSA. Being ECC the most efficient public key cryptosystem for constrained devices, it has the inconvenience of presenting interoperability problems. That is, there are several tuples T that can be used to implement ECC, two parties performing cryptographic operations must accord the same tuple to interoperate.

An ECC implementation imposes several challenges, specifically if performance, security and flexibility are compromised. Software ECC implementation offers moderate speed, high power consumption compared to custom hardware, and only limited physical security, especially with respect to key storage. If security algorithms are implemented in hardware, a gain in performance is obtained at cost of flexibility. Dedicated hardware implementations of cryptographic algorithms with low power consumption are expected to outperform the software ones because the available instruction set of a processor does not implement directly the cryptographic functions. Also, hardware implementations of cryptographic algorithms are more secure because they cannot be easily read or modified by an

outside attacker as software implementations. ASIC (Application Specific Integrated Circuit) implementations show lower price per unit, reach high speeds and have low power dissipation. However, the lack of flexibility of ASIC implementations with respect to the algorithms and parameters, leads to higher development costs and switching.

ECC interoperability would be better achieved by software implementations than custom hardware due to the software flexibility to switch among different ECC schemes implementations with several security levels. However, the performance of software implementations is lower. An approach studied in recent years combines the advantages of software (flexibility) and hardware (performance) in a new paradigm of computation named *Reconfigurable Computing* RC [11]. RC is a discipline that covers the computer science and electronic engineering areas. RC involves the use of reconfigurable devices for computing purposes.

Reconfigurable devices are ideal for cryptographic algorithms implementation because of the following criteria [12]:

- **Algorithm Agility**: Switching of cryptographic algorithms during operation of the targeted application. Whereas algorithm agility is costly with traditional hardware, reconfigurable devices can be reprogrammed on the fly.

- **Algorithm Upload**: Devices are upgraded with a new encryption algorithm because of different reasons, for example, algorithm was broken or a new standard was created.

- **Architecture Efficiency**: A hardware architecture can be much more efficient if it is designed for an specific set of parameters, for example the key or the underlying finite field. The more specific an algorithm is implemented the more efficient it can become. Reconfigurable devices allow this type of design and optimization with an specific parameter set. Due to the design of reconfigurable devices, the application can be changed totally or partially.

- **Resource Efficiency**: Since a cryptographic algorithm can offer different security services at different times, the same reconfigurable device can be used to implement the algorithms for different services through runtime reconfiguration.

- **Throughput**: General-purpose processors are not optimized for fast execution especially in the case of public-key algorithms. This is because they do not have instructions for modular arithmetic operations on long operands, which is necessary in that kind of algorithms. Although typically slower than ASIC implementations, implementations on reconfigurable devices have the potential of running substantially faster than software implementations.

New academic proposals suggest the use of security protocols based on both public and private cryptography in order to ensure the security services of authentication, confidentiality and integrity. Strong cryptography like ECC is currently being considered to provide these security services and some protocols including it have been proposed [13, 14]. But at this moment ECC presents interoperability problems. An interoperable security solution based on ECC is difficult due to various implementation choices and the underlying algorithms complexity. Although ECC is considered to be the best choice to provide security services to constrained devices [15], there is much work to be done in order to achieve it.

Although software and hardware implementations of ECC have been reported in the literature (related work is discussed in section 2.3.1), they are optimized for an specific security level, leading to low flexibility and interoperability problems [16]. In the case of software implementations, although some flexibility is achieved, implementations result in low performance. In the case of hardware implementations, optimized versions have been preferred for high performance which has increased the interoperability problems.

1.3 Research question

Elliptic curve cryptography is considered one of the best choice for public key cryptography [15], specially for constrained devices. It uses shorter keys without decreasing the security level which implies less space for storage and better use of the bandwidth if keys are transmitted using a communication network.

An ECC cryptosystem is necessarily associated with a set of parameters like the finite field, elliptic curve, finite field representation, etc. For two parties using ECC to interoperate, they must agree to use the same ECC parameters and to have the implementation of ECC schemes using that parameters.

To improve efficiency and reduce code-size, many implementations are restricted to use an specific parameter set. Typically, a particular set of curves is chosen because of certain algebraic properties that allow for an efficient implementation. For example, some environments choose solely Koblitz curves because they lead to particularly efficient implementations. Given this, there is the potential for widespread interoperability problems among ECC implementations that have chosen disparate curves.

Several standards have published parameters for ECC implementations [2]. These curves represent various security levels included in table of equivalent key strengths (see table 1.2 in section 1.1). With the publication of these recommendations, the implementation parameters of ECC have been converging on these standard curves. In order to improve the chances of interoperability, systems deploying ECC should use these curves and be able to process all of the curves in the list. Tables 1.6 and 1.7 show some of the different recommendations of parameters for different security levels using elliptic curves on the finite fields GF(2^m) and GF(p) respectively. Each one of the recommendations indicates the elliptic curve, security parameters G, n and h (see section 2.2.3).

The elliptic curves for GF(p) are known as odd curves and the ones for GF(2^m) are known as even characteristic curves. In the column *Name* of both tables 1.6 and 1.7, the number indicates the size of the keys in bits (the size in bits of the elements in the finite field), r stands for random curve and k stands for Koblitz curve. Different random and Koblitz curves are specified in the standards and identified by a number (*r1, k1, r2, k2*, etc). The '-' denotes parameters non-conformant with the standard, a C denotes parameters conformant with the standard, and an R denotes parameters explicitly recommended in the standard.

Table 1.6: Tuples T for GF(2^m) recommended in standards [2].

Name	**ANSI X9.62**	**ANSI X9.63**	**IEEE P1363**	**IPSec**	**NIST**	**WAP**
113r1	-	-	C	C	-	R
131r1	-	-	C	C	-	C
163k1	C	R	C	R	R	R
193r1	C	R	C	C	-	C
239k1	C	C	C	C	-	C
283r1	C	R	C	R	R	C
409r1	C	R	C	C	R	C
521r1	C	R	C	C	R	C

Table 1.7: Tuples T for GF(p) recommended in standards [2].

Name	ANSI X9.62	ANSI X9.63	IEEE P1363	IPSec	NIST	WAP
112r1	-	-	C	C	-	R
128r1	-	-	C	C	-	C
160r1	C	C	C	C	-	R
192k1	C	R	C	C	-	C
224k1	C	R	C	C	-	C
256r1	R	R	C	C	R	C
384r1	C	R	C	C	R	C
521r1	C	R	C	C	R	C

Although some reported works allow some flexibility in choosing the ECC parameters, a reconfigurable architecture enabling interoperability with other designs has not been explored at all. Although in some works the design of the arithmetic units is parameterizable in the order field, the architecture needs to be reconfigured out of line for other finite fields orders. It would be desired a real time adaptation of the architecture to different security levels.

Only [17] proposes to manage different elliptic curves without reconfiguring the hardware by implementing wired reduction for three of the NIST curves and implements the technique called *partial reduction* for arbitrary curves. Other works like [18, 19, 20] manage different elliptic curves but need to reconfigure the hardware out of line. Other works like [21] propose a HW/SW partition and use reconfigurable logic only for arithmetic instructions. An example of customized implementation for an specific elliptic curve is [22]. Other efforts to achieve ECC interoperability propose an unified arithmetic unit for both prime and binary fields arithmetic [23, 24].

The research questions on which this thesis deals with are:

1. *How to design an efficient and interoperable hardware architecture for elliptic curve cryptography?.*

2. *Which is the cost of interoperability in terms of area and performance?*

3. *How to achieve architecture reconfigurability to allow the adaptation for different security levels?*

1.4 Thesis objectives

The main interest to propose hardware architectures for elliptic curve cryptography has been the computation, as fast as possible, of the most computational expensive operation in elliptic curve cryptography, the scalar multiplication dP. The approach in this dissertation is different. This PhD project aims to provide a flexible architecture that can adapt to several security levels while achieving high performance. This implies a careful design that performs well in several elliptic curves and finite fields and at the same time, that allows to establish an area/performance trade off.

1.4.1 General objective

The objective of this thesis is to design and implement a reconfigurable hardware architecture that allows interoperability for different security levels and implementation parameters of elliptic curve cryptography while achieving high performance.

This proposal aims to solve the interoperability problem of current elliptic curve cryptography. For an ECC co-processor to be interoperable, it must be flexible to manage different elliptic curves in an specific finite field. In the case of curves defined in GF(p), the co-processor should support any or most of the curves in table 1.7. In the case of curves defined on GF(2^m), the co-processor must support any or most of the curve in table 1.6.

Reconfiguration will be necessary to manage different elliptic curves and finite fields and maximizing the performance and minimizing the resources for each security level. Dynamic adaptation of the architecture to those security levels is desired.

1.4.2 Specific objectives

Specific objectives of this project are:

1. To implement architectures of ECC algorithms with different sets of parameters for the arithmetic and security levels.

2. To select the most promising ECC arithmetic algorithms that may lead to the best area/performance trade-off.

3. To identify key modules in the ECC scheme that can be suitable for reconfiguration.

4. To define a reconfiguration strategy for architecture adaption to different security levels.

The ECC co-processor designed in this work is for elliptic curves in GF(2^m). The arithmetic in GF(2^m) fields is well suited to be implemented in hardware because it is binary arithmetic.

1.5 Thesis outline

This thesis describes proposed research to develop an interoperable hardware architecture for elliptic curve cryptography making use of the reconfigurable computing concept. The overall goal of this research is to design, implement and evaluate a reconfigurable architecture that facilitates the interoperability of elliptic curve cryptography to adapt dynamically to the security levels recommended by various standards and emerging proposal.

This thesis is organized as follows. The mathematical background and the implementation issues of ECC are presented in the next chapter. This chapter also describes related works about implementations of elliptic curve cryptography. Chapter 3 presents the concepts related to reconfigurable computing used in this thesis and the design flow for implementing reconfigurable systems for elliptic curve cryptography. Chapter 4 describes the design and implementation of a hardware ECC co-processor for scalar multiplication dP. The reconfiguration strategy for implementing the reconfigurable system for interoperable ECC is described at the end of chapter 4. Chapter 5 shows and analyzes the results of this research and chapter 6 gives the conclusions of this work and directions for future work.

Chapter 2

Elliptic Curve Cryptography

This chapter presents the mathematical background of elliptic curve cryptography (ECC) and ECC-based cryptographic schemes such as encryption and digital signature. Related work about software and hardware implementations of ECC are also presented and discussed.

2.1 Groups and Finite Fields

Groups and fields are part of abstract algebra, a branch of mathematics. Finite fields are increasingly important in several areas of mathematics, including linear and abstract algebra, number theory and algebraic geometry, as well as in computer science, statistics, information theory, and engineering. Also, many cryptographic algorithms perform arithmetic operations over these fields [3].

A *group* Γ is an algebraic system $\{S, \diamond\}$ consisting of a set S and a binary operation $\diamond$ defined on S that satisfies the following axioms:

1. Closure: $\forall\ x, y \in G, x \diamond y \in G$.
2. Associativity: $\forall\ x, y, z \in G, (x \diamond y) \diamond z = x \diamond (y \diamond z)$.
3. Identity: $\exists\ I \in G$ such as $x \diamond I = x$.
4. Inverse: $\forall\ x \in G$, exist only one $y \in G$ such as $x \diamond y = y \diamond x = I$.

If $\forall\ x, y \in G, x \diamond y = y \diamond x$, Γ is called an *abelian* group.

A *finite field* is an algebraic system $\{F, \oplus, \odot\}$ consisting of a set F containing a fixed number of elements and two binary operations, $\oplus$ (*plus*) and $\odot$ (*dot*) on F, satisfying the following axioms:

1. Elements 0 and 1 $\in F$.
2. F is an abelian group respect to operation $\oplus$.
3. F - $\{0\}$ is an abelian group respect to operation $\odot$.
4. $\forall\ x, y, z \in F$, $x \odot (y \oplus z) = (x \odot y) \oplus (x \odot z)$ and $x \oplus (y \odot z) = (x \oplus y) \odot (x \oplus z)$.

The order of a finite field is the number of elements in that field. It has been showed [25] that exists a finite field of order q if and only if q is a prime power. In addition, if q is a prime power, the finite field of order q is unique. A finite field, also known as *Galois Field*, is denoted as F_q or GF(q).

2.1.1 Modular arithmetic

The operation,

$$a \bmod n = z \tag{2.1}$$

means that z is the remainder when a is divided by n, the remainder is an integer in the range $[0, n-1]$. This operation is called modular reduction, and it is used in cryptographic schemes mainly for two reasons: 1) operations like logarithms and square roots module n are hard problems and 2) the space of values is restricted to a fixed group of numbers. In cryptography applications, a, z and n are large integer numbers. Another common notation for equation 2.1 is to say that a and z are equivalent or a is congruent to $z \mod n$, which is written as

$$a \equiv z \pmod{n} \tag{2.2}$$

Modular arithmetic is commutative, associative and distributive. The common integer operations +, *, and - in modular arithmetic are defined as follows:

1. $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
2. $(a - b) \bmod n = ((a \bmod n) - (b \bmod n)) \bmod n$

3. $(a * b) \bmod n = ((a \bmod n) * (b \bmod n)) \bmod n$

4. $a * (b + c) \bmod n = (((a * b) \bmod n) + ((a * c) \bmod n)) \bmod n$

Another important modular operation is the *inversion*. a^{-1} is the inverse mod n of a number a if equivalence in equation 2.3 is true.

$$a * a^{-1} \equiv 1 \pmod{n} \tag{2.3}$$

For a given number a, a^{-1} is the unique solution only if a and n are *relative primes* [5]. If n is a prime number, every number in the range $[1, n-1]$ is relatively prime to n and has exactly one inverse $\pmod{n}$. To calculate a module inverse, two algorithms are commonly used: The Extended Euclidean Algorithm and the Fermat's Little Theorem. These algorithms are described in chapter 4.

2.1.2 Prime and binary finite field

The *prime finite field* has been long used in cryptography. It is denoted as GF(p), where p is a prime number. GF(p) consists of the elements $\{0, 1, 2, \ldots, p-1\}$. The operations $\oplus$ and $\odot$ are performed as the ordinary integer operations sum and multiplication respectively applying reduction $\pmod{p}$. These operations are defined as follows:

1. $\oplus$: $(a + b) \bmod p$, a,$b \in$ GF(p),

2. $\odot$: $(a * b) \bmod p$, a,$b \in$ GF(p),

3. $\forall\ a \neq 0 \in$ GF(p), a^{-1} is the inverse of a if $a * a^{-1} = 1 \pmod{p}$.

The *binary finite field* is denoted as GF(2^m) (also known as *two-characteristic field* or the *Galois field*) and can be viewed as a m-dimension vectors space on $\{0, 1\}$. As a vectorial space, a basis exist in GF(2^m). The set $\{\alpha_0, \alpha_1, ..., \alpha_{m-1}\}$, $\alpha_i \in$ GF(2^m), is called a basis of GF(2^m) if exist $m-1$ elements a_i in $\{0, 1\}$ such that every element $a \in$ GF(2^m) can be expressed as in equation 2.4.

$$a = a_{m-1}\alpha_{m-1} + a_{m-2}\alpha_{m-2} + ... + a_1\alpha_1 + a_0\alpha_0 \tag{2.4}$$

If such basis exist, each element in GF(2^m) can be represented as the binary m-vector $(a_0, a_1, a_2, \ldots, a_{m-1})$. There are different basis for GF(2^m), the most

commonly used are: *polynomial*, *normal* and *dual*. The arithmetic for binary operations $\oplus$ and $\odot$ changes slightly according to the basis employed. When implementing binary field arithmetic, polynomial basis is preferred because of the sum of two elements is a simple XOR operation and the elements in GF(2^m) are binary strings that are well stored in m-bit registers. Details about normal and dual basis can be found in [25] and [26] respectively.

In polynomial basis, each m-vector $a \in \mathrm{GF}(2^m)$, is viewed as a polynomial,

$$a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0 \tag{2.5}$$

The binary field GF(2^m) is generated by an *irreducible polynomial* $F(x)$ of grade m of the form

$$x^m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + ... + f_1x + f_0 \tag{2.6}$$

where $f_i \in \{0, 1\}$. The polynomial is named irreducible because it can not be expressed as the multiplication of two other polynomials (it is as a prime number in integer arithmetic). The polynomial needs to be irreducible, otherwise the math does not work [5].

Let be a, $b \in \mathrm{GF}(2^m)$, $a = (a_{m-1}, a_{m-2}, ..., a_1, a_0)$ $b = (b_{m-1}, b_{m-2}, ..., b_1, b_0)$. The arithmetic operations $\oplus$ and $\odot$ in the finite field GF(2^m) are defined as follows:

1. $\oplus$: $a \oplus b = c$, $c = (c_{m-1}, c_{m-2}, ..., c_1, c_0)$, where $c_i = a_i$ XOR b_i.

2. $\odot$: $a \odot b = c$, $c = (c_{m-1}, c_{m-2}, ..., c_1, c_0)$, where $c(x) = a(x)b(x)$ mod $F(x)$.
   $c(x) = c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + ... + c_1x + c_0$
   $a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0$
   $b(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + ... + b_1x + b_0$

For cryptographic applications, the irreducible polynomial $F(x)$ is a trinomial of the form $x^m + x^i + 1$ or it is pentanomial of the form $x^m + x^i + x^j + x^l + 1$, where i, j and l are positive integers. The use of trinomials or pentanomials leads to efficient software and hardware implementations.

2.2 Elliptic Curve Cryptography (ECC)

Elliptic curves, as geometric algebraic entities have been studied since the second half of the nineteen century, initially, without any cryptographic purpose. In 1985, the application of elliptic curves in public key cryptography was independently proposed by Neals Koblitz [27] and Victor Miller [28].

Koblitz and Miller proposed to use an elliptic curve defined on a finite field, and to define a point addition operation such that the points of the elliptic curve and the point addition operation formed an abelian group. On this group, the discrete logarithm problem, called the elliptic curve discrete logarithm problem (ECLDP), can be defined and so, a cryptosystem could be built on this problem. The main advantage of this elliptic curve cryptosystem is that the ECLDP is more difficult to solve than that defined on the multiplicative group GF(p). The best algorithm known for solving the ECDLP is fully exponential, the Pollar-Rho method [29].

ECC can offer a similar security level than other public key criptosystems using shorter length keys, which implies less space for key storage, time saving when keys are transmitted and modular computations less costly. ECC's security has not been proved; its strength is based on the inability to find attacks.

2.2.1 The elliptic curve group

An elliptic curve over a field K is formed by the point O called point at infinity, and the set of points $P = (x, y) \in K \times K$ satisfying a non-singular Weierstrass equation:

$$E(K) : y^2 + a_1xy + a_3y = x^3 + a_2x_2 + a_4x + a_6\,. \tag{2.7}$$

The elliptic curve $E(K)$ together with the point O form the additive group $\{E(K) \cup O, +\}$, being O the identity element ($P + O = P$). For cryptographic applications, the field K is a finite field. If K is GF(p) the elliptic curve equation is

$$E(GF(p)) : y^2 = x^3 + ax + b\,. \tag{2.8}$$

When the elliptic curve is defined on the binary field GF(2^m), the equation is

$$E(GF(2^m)) : y^2 + xy = x^3 + ax^2 + b\,. \tag{2.9}$$

Since K is finite, $E(K)$ is also a finite set. The group operation $+$ is the sum of points in the elliptic curve and its definition has a geometrical interpretation. In order to satisfy the closure axiom in the group $\{E(K) \cup O, +\}$, three kinds of operation $+$ are defined. The operation ECC-Dbl is defined as the addition of a point P to itself while the operation ECC-Add is the sum of two different points P, Q. The geometric interpretation of point addition in elliptic curve cryptography is shown in figure 2.1. The geometric interpretation of point addition is as follows: Given the points P and Q (figure 2.1 a)), a line is traced from P to Q. Such line will cut the elliptic curve in a unique point R'. The symmetric point R will be the sum of points P and Q. There is a possibility that the line that joins P and Q does not cut the elliptic curve (figure 2.1 b)). In this case, it is considered that the line cuts the elliptic curve *at infinity*. This is the imaginary point O that belongs to the elliptic curve and acts as the neutral element, that is, $P + O = P$ for every point P in the elliptic curve. In the case of the ECC-Dbl operation (figure 2.1 c)) the line traced is the one that is tangent to the point P.

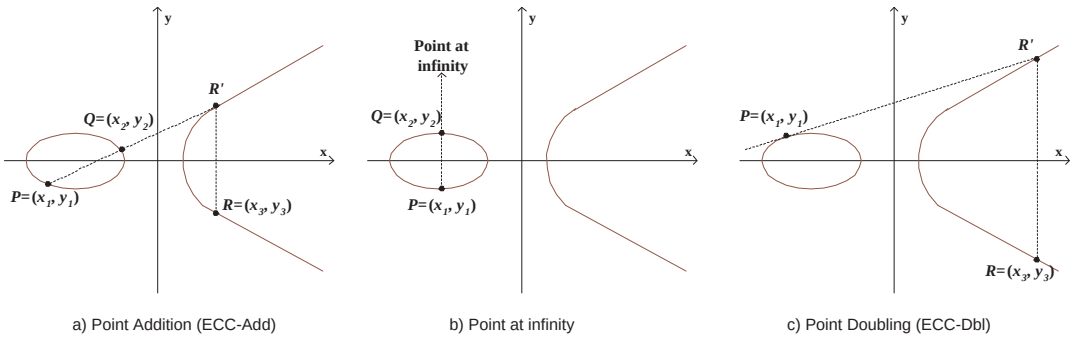


Figure 2.1: Point addition in ECC.

ECC-Add and ECC-Dbl are carried out in different way depending on the coordinate system used to represent the elliptic curve points (see section 2.2.4 for details). Anyway, several finite field operations are always required to implement the $+$ operation in $E(K)$.

2.2.2 The discrete logarithm problem

The discrete logarithm problem DLP is defined on groups, like the multiplicative group GF(p) or the one formed by elliptic curve points. Given the group $\Gamma = \{S, \diamond\}$ and a, $b \in S$, the discrete logarithm of a is z if $a = b^z$. In this case, b^z does not necessary mean exponentiation but the application of operation $\diamond$ to b $z - 1$ times. The elliptic curve discrete logarithm problem ECDLP is the DLP defined on the group formed by the point of the elliptic curve $E(K)$ (being K a finite field). As the group operation is $+$, the ECDLP is defined as: given two points P, $Q \in E(K)$, to find the positive integer d such as $Q = dP$. As commented previously in chapter 1, the best algorithm known to solve the ECDLP is of exponential complexity. This is why ECC is considered more secure than other kind of public key cryptography. On the contrary, knowing the scalar d and the point P, the operation dP is relative easy to compute. The operation dP is called *scalar multiplication*. It is the most time consuming operation in all cryptographic schemes built on elliptic curves. The operation dP is computed as accumulative sum operation of point P with itself. These sum operations are either ECC-Add or ECC-Dbl operations.

There exist curves on which the ECDLP is easy to solve [25], the so-called *supersingular* curves have to be avoided. These curves have an equal number of points that the finite field on which they are defined. The National Institute of Standards and Technology (NIST) has emitted several recommendations and has proposed several elliptic curves over both GF(p) and GF(2^m) for cryptographic applications. Different to other cryptosystems, the security of ECC not only depends on the length of the key but also on other parameters like the elliptic curve being used.

2.2.3 Cryptographic schemes

An elliptic curve cryptosystem consist on a tuple. If the elliptic curve is defined on GF(p), the cryptosystem domain parameters are the six-tuple $T_{GF(p)} = (p, a, b, G, n, h)$, where:

p	is the a big prime number.
a, b	define the elliptic curve E on GF(p).
G	is a generator point of the elliptic curve E(GF(p).
n	is the order of G, that is, the smaller integer such that $nG = O$ (identity point in the additive group Γ).
h	is called the co-factor, the total number of points in the curve divided by n. This value is optional

If the curve is defined on the binary field GF(2^m), the domain parameters consist on a seven-tuple $T_{GF(2^m)} = (m, F(x), a, b, G, n, h)$, where m defines the order of the finite field, $F(x)$ is the irreducible polynomial required in the field arithmetic. All other values have similar definition that in the case of GF(p).

In both cases, the domain parameters specify an elliptic curve, a generator point and the order of this point. How to generate such domain parameters is not concern of this thesis neither their validation. There are proposed methods for this that can be reviewed in [8, 2, 25].

ECDSA is the elliptic curve analogue of DSA for digital signatures. It has been standardized by several international organizations like ISO, IEEE and ANSI. In this thesis, the specification of ECDSA in the ANSI X9.62 document [8] is referred. The ECIES scheme, for public-key bulk encryption is the most promising scheme to be standardized. It has been considered in some drafts and it is currently recommended by the Standards for Efficient Cryptography Group (SECG) [2].

In both ECDSA and ECIES, it is supposed that two entities A and B share either the domain parameters $T_{GF(2^m)}$ or $T_{GF(p)}$. It is also supposed that $\{d_A$, $Q_A\}$ are the private and public keys for entity A and that $\{d_B$,$B\}$ are the ones for entity B. Private keys are elements of the underlying finite field while public keys are points in the elliptic curve.

ECIES scheme

The ECIES scheme employs two algorithms: a symmetric cipher E and a MAC (Message Authentication Code) algorithm. The keys for the symmetrical and the MAC algorithms are generated from a secret shared value between A and B by a key derivation function (KDF). Assume that $k_{Slength}$ is the key's length for the symmetrical algorithm and $k_{MAClength}$ is the one for the MAC algorithm. A sends an encrypted message D to B executing the following steps:

1. Select a random number d from $[1, n-1]$
2. Compute $(x, y) = dQ_B$ and $R = dG$
3. Derive a $(S + M)$-bit key k_{KDF} from x according to [2].
4. Derive a S-bit key k_S from K_{KDF} and encrypt the message. $C = E(m, k_S)$
5. Derive a M-bit key k_M from K_{KDF} and compute the m's MAC value. $V = MAC(m, k_M)$
6. Send (R, C, V) to B

To recover the original message, B does the following:

1. If R is not a valid elliptic curve point, fail and return.
2. Compute $(x', y') = d_B R$
3. Derive a $(S + M)$-bit key k_{KDF} from x' according to [2].
4. Derive a S-bit key k_S from K_{KDF} and decrypt the message C. $m_1 = E(C, k_s)$
5. Derive a M-bit key k_M from K_{KDF} and compute the m_1's MAC value. $V_1 = MAC(m_1, k_M)$
6. Accept message m_1 as valid if and only if $V = V_1$

SEC-1 [2] recommends two MAC algorithms, HMAC-SHA-1-160 and HMAC-SHA-1-80. In the first case the key used is 160-bits long and produces an 80-bit or 160-bit output. The second case is different only in that the key is 80-bits long. Both MAC schemes are described in ANSI X9.71 based on the hash function SHA-1 described in FIPS 180-1 [30]. For symmetrical encryption, SEC recommends two symmetrical algorithms that can be used: 3-Key TDES in CBC [8] mode and XOR encryption. The XOR encryption scheme is the simplest encryption scheme in which encryption consists of XORing the key and the message, and decryption consists of XORing the key and the ciphertext to recover the message. The XOR scheme is commonly used either with truly random keys when it is known as the 'one-time pad', or with pseudorandom keys as a component in the construction of stream ciphers. The XOR encryption scheme uses keys which are of the same length as the message to be encrypted or the ciphertext to be decrypted. 3-key

TDES in CBC mode is designed to provide semantic security in the presence of adversaries launching chosen-plaintext and chosen-ciphertext attacks. The XOR encryption scheme is designed to provide semantic security when used to encrypt a single message in the presence of adversaries capable of launching only passive attacks. The KDF key derivation function only supported at this moment in ECIES is defined in ANSI X9.63.

ECDSA scheme

To sign a message D, entity A does the following:

1. Select a random number d from $[1, n-1]$

2. Compute $R = dG = (x, y)$ and $r = x \bmod n$. If $r = 0$ go to step 1.

3. Compute $s = k^{-1}(H(D) + d_A r) \bmod n$, H is the hash value of the message.

4. The digital signature on message D is the pair (r, s)

Entity B can verify the digital signature (r, s) on D performing the following steps:

1. Verify r and s are integers in $[1, n-1]$, if not, the digital signature is wrong. Finish and reject the message.

2. Compute $w = s^{-1} \bmod n$ and $H(D)$, H is the hash value of the message.

3. Compute $u_1 = H(D)w \bmod n$ and $u_2 = rw \bmod n$

4. Calculate $R' = u_1 G + u_2 Q_A = (x', y')$

5. Compute $v' = x' \bmod n$, accept the digital signature if and only if $v' = r$

The ECDSA scheme requires an elliptic point addition, scalar multiplications, modular arithmetic and the hash value of the message. The hash function recommended in ANSI X9.62 is SHA-1.

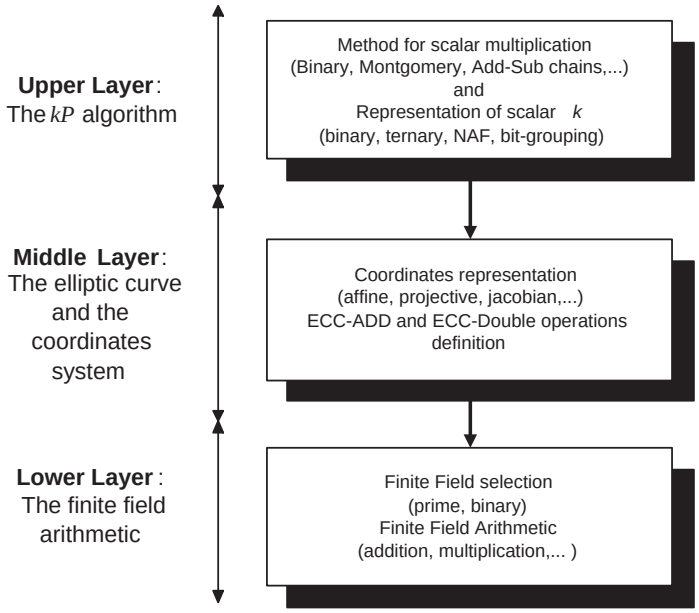


Figure 2.2: Three layers approach for dP implementation

2.2.4 Scalar multiplication dP

By far, the most time consuming operation in ECC cryptographic schemes is the scalar multiplication dP. Efficient hardware/software implementations of this operation have been the main research topic on ECC in recent years (see related work in section 2.3.1). This costly elliptic curve operation is performed according to the three layers shown in figure 2.2. These layers are described in detail in the following sections.

Scalar multiplication top layer

At the top layer of figure 2.2 there are different methods for computing the scalar multiplication, independently of the other layers. An scalar multiplication dP where d is an scalar and P is an elliptic curve point, is the result of adding the point P to itself $d-1$ times. That is,

$$dP = \underbrace{P + P + P + \cdots + P}_{d-1 \ times}$$

This operation is performed by applying a sequence of ECC-Add and ECC-Dbl operations. Some of the reported methods in the literature for dP and their com-

Table 2.1: Scalar multiplication methods

Method	**# ECC-Add (A) and ECC-Dbl (D)**
Binary method (left to rigth)	$(\frac{m}{2})A + mD$
NAF Binary method (rigth to left)	$(\frac{m}{3})A + mD$
Montgomery	$(mA + mD)$
Windowing NAF Binary method (rigth to left)	$(D + (2^{w-2} - 1)A) + (m/(w+1)A + mD)$
Montgomery (Digit)	$((d(2^w - 1)/2^w - 1) + (2^w - 2)A)$
Comb method (Digit)	$((d-1)(2^w - 1)/2^w)A + (d-1)D$
TNAF (Frobenius map)	$(m/3)A$
Window TNAF	$(m/(w+1))A$

plexity expressed in ECC-Add and ECC-Dbl operations, are given in table 2.1 [31].

All reported methods for computing dP parse the scalar d and depending on the bit value, they perform either an ECC-Add or an ECC-Dbl operation. In table 2.1, m is the size in bits of scalar d ($m = \lceil \log_2 d \rceil$). Some methods consider D bits of scalar d at a time and then parse d in $w = \lceil \frac{m}{D} \rceil$ iterations.

The basic technique for scalar multiplication is the *double and add method* (D&A), also known as the *binary method*, which is the additive version of the repeated-square-and-multiply method for exponentiation. It performs an ECC-Dbl operation in each iteration independently of the current bit value of d. A ECC-Add operation is performed only if the current value of d is '1'. On average, d has $\frac{\log_2 d}{2}$ '1's. For the second method in table 2.1, NAF refers to the Non Adjacent Form transformation of scalar d. In such representation, the resulting d has the minimum number of '1's, so the number of additions is reduced but the point subtraction operation is introduced. The NAF version of the binary method is faster than the standard one. The Montgomery method performs both the ECC-Add and the ECC-Dbl operation independently of each bit value of scalar d. This fact makes the Montgomery method resistant to the power analysis attack, which tries to guess the secret key d from analyzing the power consumption of operations ECC-Add or ECC-Dbl. The windowing based methods are commonly implemented in software using pre-computed values and demand more memory for processing than the methods previously commented.

Scalar multiplication middle layer

The middle layer corresponds to the coordinate system in which elliptic points are represented. This layer defines how the group operation, ECC-Add or ECC-Dbl, is performed. The point addition in elliptic curves is not as the traditional point addition in the known coordinate system XY, where $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$.

Due the group is closed under the + group operation, the group law for point addition is such that for any two point in the elliptic curve P, Q, $P + Q$ is also in the elliptic curve. In the literature there are various coordinate systems that can be used which lead to different definitions for point addition.

The most popular coordinate system is the affine one where each elliptic curve point is represented by the pair (x, y) satisfying equation 2.7. In the case of the curve $E(GF(p))$, ECC-Add and ECC-Dbl are defined as follows:

Let a, $b \in$ GF(p) satisfying $4a^3 + 27b^2 \neq 0$. Let $P, Q, R \in E(GF(p)$, $P = (x_1, y_1)$ $Q = (x_2, y_2)$ $R = (x_3, y_3)$.

1. $P + Q = O$, if $P = O$ or $Q = O$

2. *ECC-Add* $(P \neq \pm Q)$.
   $R = P + Q$, where
   $x_3 = \lambda^2 - x_1 - x_2$
   $y_3 = \lambda(x_1 - x_3) - y_1$
   $\lambda = \dfrac{(y_2 - y_1)}{(x_2 - x_1)}$

3. *ECC-Dbl* $(P = Q)$.
   $R = P + Q = 2P$, where
   $x_3 = \lambda^2 - 2x_1$
   $y_3 = \lambda(x_1 - x_3) - y_1$
   $\lambda = \dfrac{3{x_1}^2 + a}{2y_1}$

When the curve $E(GF(2^m))$ is represented in affine coordinates, the + operation is defined as:

Given the points $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $R = (x_3, y_3) \in E(GF(2^m))$, the group law is:

1. $P + Q = O$, if $P = O$ or $Q = O$

2. *ECC-Add* $(P \neq \pm Q)$.
   $R = P + Q$, where
   $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$
   $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$
   $\lambda = \dfrac{(y_2 + y_1)}{(x_2 + x_1)}$

3. *ECC-Dbl* $(P = Q)$.
   $R = P + Q = 2P$, where
   $x_3 = \lambda^2 + \lambda + a$
   $y_3 = x_1^2 + \lambda x_3 + x_3$
   $\lambda = x_1 + \dfrac{y_1}{x_1}$

López and Dahab [32] propose a new definition for the group operation changing from affine to projective coordinates. The projective version of equation 2.9 is given in equation 2.10.

$$E_p(GF(2^m)) : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{2.10}$$

The conversion between affine and projective coordinates is given by

$(x, y) \mapsto (x, y, 1)$ (affine to projective) and
$(X, Y, Z) \mapsto (X/Z, Y/Z^2)$ (projective to affine)

Authors named this kind of coordinates *López-Dahab* coordinates. Supposing the points $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$, $R = (X_3, Y_3, Z_3) \in E_p(F_{2^m})$, the new definition for the group operation is:

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

2. *ECC-Add* ($P \neq \pm Q$).

$$
\begin{aligned}
X_3 &= C^2 + H + G \\
C &= A_1 + A_2 \\
H &= CF \\
G &= D^2(F + aE^2) \\
D &= B_1 + B_2 \\
E &= Z_0 Z_1 \\
F &= DE \\
A_1 &= Y_2 Z_1^2 \\
A_2 &= Y_1 Z_2^2 \\
B_1 &= X_2 Z_1 \\
B_2 &= X_1 Z_2 \\
Z_3 &= F^2 \\
Y_3 &= HI + Z_3 J \\
I &= D^2 B_1 E + X_3 \\
J &= D^2 A_1 + X_3
\end{aligned}
$$

3. *ECC-Dbl* ($P = Q$).

$$
\begin{aligned}
Z_3 &= Z_1^2 X_1^2 \\
X_3 &= X_1^4 + bZ_1^4 \\
Y_3 &= bZ_1^4 Z_3 + X_3(aZ_3 + Y_1^2 + bZ_1^4)
\end{aligned}
$$

This new representation does not require finite field inversion when computing ECC-Add or ECC-Dbl. Inversion is the most time consuming field operation, which is eliminated at the cost of more field multiplication. The same authors propose to use mixed coordinates, a point in affine coordinates ($Z = 1$) and the other in projective. That is, if $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, 1)$, then, the ECC-Add operation can be computed as follows:

ECC-ADD redefinition

$$
\begin{aligned}
X_3 &= A^2 + D + E \\
A &= Y_2 Z_1^2 + Y_1 \\
B &= X_2 Z_1 + X_1 \\
C &= Z_1 B \\
D &= B^2(C + aZ_1^2)
\end{aligned}
$$

$$E = AC$$

$$Y_3 = EF + Z_3 G$$
$$F = X_3 + X_2 Z_3$$
$$G = X_3 + Y_2 Z_3$$

$$Z_3 = C^2$$

The Montgomery method for dP computation was proposed in [33]. In this method, adding and doubling are achieved by using only the x and z coordinates. This method can be implemented using affine or projective coordinates, but the point addition rules change. The original formulas are simplified and less operations are performed although the conversion from projective to affine coordinates is more complex. The Montgomery algorithm in affine coordinates is lised in algorithm 2.1:

Algorithm 2.1 Montgomery dP algorithm in affine coordinates

Input: An integer $d > 0$ in the binary form $(k_{l-1}k_{l-2}\dots k_1 k_0)$ and a point $P = (x, y)$

Output: $Q = dP$

1: **if** $k = 0$ or $x = 0$ **then**
2: output (0, 0) and stop
3: **end if**
4: $x_1 \leftarrow x, x_2 \leftarrow x^2 + \frac{b}{x^2}$
5: **for** $i = l - 2$ downto 0 **do**
6: $t \leftarrow \frac{x_1}{x_1 + x_2}$
7: **if** $k_i = 1$ **then**
8: $x_1 \leftarrow x + t^2 + t$ (ECC-ADD)
9: $x_2 \leftarrow x_2^2 + \frac{b}{x_2^2}$ (ECC-Double)
10: **else**
11: $x_1 \leftarrow x_1^2 + \frac{b}{x_1^2}$ (ECC-Double)
12: $x_2 \leftarrow x + t^2 + t$ (ECC-ADD)
13: **end if**
14: **end for**
15: $r_1 \leftarrow x_1 + x, r_2 \leftarrow x_2 + x$
16: $y_1 \leftarrow \frac{r_1(r_1 r_2 + x^2 + y)}{x} + y$
17: return $Q = (x_1, y_1)$

Algorithm 2.2 Montgomery dP algorithm in projective coordinates

Input: An integer $d > 0$ in the binary form $(k_{l-1}k_{l-2}\dots k_1k_0)$ and a point $P = (x, y)$

Output: $Q = dP$

1: **if** $k = 0$ or $x = 0$ **then**
2: output (0, 0) and stop
3: **end if**
4: $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
5: **for** $i = l - 2$ downto 0 **do**
6: $t \leftarrow x_1/(x_1 + x_2)$
7: **if** $k_i = 1$ **then**
8: $Madd(X_1, Z_1, X_2, Z_2)$, $Mdouble(X_2, Z_2)$
9: **else**
10: $Madd(X_2, Z_2, X_1, Z_1)$, $Mdouble(X_1, Z_1)$
11: **end if**
12: **end for**
13: return $Q = Mxy(X_1, Z_1, X_2, Z_2)$

The Montgomery method using projective coordinates is listed in algorithm 2.2. $Madd$ and $Mdouble$ are the ECC-Add and ECC-Dbl operation using only the x and z coordinates. The Mxy function converts the resulting point back to affine coordinates. These functions are defined as:

$Mdouble$

$$X \leftarrow X^2$$
$$Z \leftarrow Z^2$$
$$T_1 \leftarrow Z^2$$
$$Z \leftarrow ZX$$
$$T_1 \leftarrow T_1 b$$
$$X \leftarrow X^2$$
$$X \leftarrow X + T_1$$

$Madd$

$$T_1 \leftarrow x$$
$$X_1 \leftarrow X_1 x Z_2$$
$$Z_1 \leftarrow Z_1 x X_2$$
$$T_2 \leftarrow X_1 x Z_1$$

$Z_1 \leftarrow Z_1 + X_1$
$Z_1 \leftarrow Z_1^2$
$X_1 \leftarrow Z_1 x T_1$
$X_1 \leftarrow X_1 + T_2$

Mxy

if $Z_1 = 0$ output $(0, 0)$ and stop
if $(Z_2 = 0)$ output $(x, x + y)$ and stop
$T_1 \leftarrow x$
$T_2 \leftarrow y$
$T_3 \leftarrow Z_1 x Z_2$
$Z_1 \leftarrow Z_1 x T_1$
$Z_1 \leftarrow Z_1 + X_1$
$Z_2 \leftarrow Z_2 x T_1$
$X_1 \leftarrow Z_2 x X_1$
$Z_2 \leftarrow Z_2 + X_2$
$Z_2 \leftarrow Z_2 x Z_1$
$T_4 \leftarrow T_1^2$
$T_4 \leftarrow T_4 + T_2$
$T_4 \leftarrow T_4 x T_3$
$T_4 \leftarrow T_4 + Z_2$
$T_3 \leftarrow T_3 x T_1$
$T_3 \leftarrow inverse(T_3)$
$T_4 \leftarrow T_3 x T_4$
$X_2 \leftarrow X_1 x T_3$
$Z_2 \leftarrow X_2 + T_1$
$Z_2 \leftarrow Z_2 x T_4$
$Z_2 \leftarrow Z_2 + T_2$

Table 2.2 shows the total number of field operations required for performing point addition in elliptic curves using each one of the above mentioned coordinate systems. In this table, $\mathcal{M}$, $\mathcal{S}$ and $\mathcal{D}$ stand for multiplication, squaring and division in the finite field GF(2^m). The affine representation requires two field multiplications and one field inversion. In the projective representation inversions are

Table 2.2: Count of finite field arithmetic in Elliptic Curve Cryptography point addition

Name	**Coordinates System**	**ECC-Add**	**ECC-Dbl**	**Conversion Projective $\mapsto$ Affine**
Affine	(x, y)	$1\mathcal{M}+ 1\mathcal{S}+ 1\mathcal{D}$	$1\mathcal{M}+ 1\mathcal{S}+ 1\mathcal{D}$	-
López-Dahab	$(x/z, y/z^2)$	$8\mathcal{M}+ 5\mathcal{S}$	$4\mathcal{M}+ 5\mathcal{S}$	$11\mathcal{M}+ 16\mathcal{3S}$
Jacobian	$(x/z^2, y/z^3)$	$10\mathcal{M}+ 4\mathcal{S}$	$5\mathcal{M}+ 5\mathcal{S}$	$12\mathcal{M}+ 16\mathcal{3S}$

avoided at the cost of more field multiplications and a conversion from projective to affine coordinates. This extra cost is accepted only if the time for computing one field inversion is the equivalent to the time required for computing six or more field multiplications.

Scalar multiplication lower layer

Finally, in the lowest layer is the finite field arithmetic. The efficient implementation of these arithmetic operations impacts the overall performance of the scalar multiplication. Binary fields $GF(2^m)$ are better preferred for hardware implementations because some arithmetic operations are easier to compute, like addition which is an XOR operation. Another advantage of using binary fields for hardware implementations of finite field arithmetic is that the elements in GF(2^m) are binary strings that are well represented in m-bit registers.

The prime field $GF(p)$ is preferred for software implementations because of elements can be organized in machine words and arithmetic instructions of general purpose processors like integer multiplication and division can be used for implementing modular arithmetic.

Four finite field operations are required in elliptic curve point addition, these are: addition $\mathcal{A}$, multiplication $\mathcal{M}$, inversion $\mathcal{I}$ (can be substituted by direct division $\mathcal{D}$) and squaring $\mathcal{S}$. Field inversion has been the most time consuming finite field operation.

For GF(2^m), finite field operations implementation depends on a basis, which can be polynomial, normal or dual. In polynomial basis, the elements of GF(2^m) are viewed as $m-1$ grade polynomials $A(x)$ with coefficients in $GF(2) = \{0, 1\}$. A basis of GF(2^m) is one of the form $\{1, t, t_1, t_2, \cdots, t_{m-1}\}$, where t is an square of an irreducible m grade polynomial $F(x)$ (cannot be factored as two polynomials). Arithmetic in GF(2^m) with polynomial basis is arithmetic of polynomials modulo

$F(x)$.

There is always a trade-off using different basis for both software and hardware implementations. Squaring is easier in normal basis but inversion is slower than inversion in polynomial basis [21].

Field multiplication can be implemented either in parallel (hardware) or serially (hardware and software). In the former, field multiplication is performed in one clock cycle but implies more hardware to be implemented. Serial multipliers requires smaller area but generates the result in several clocks cycles. In recent works, a combination of these two approaches is used: the digit-serial multiplier, which allows a trade-off speed/area analysis when implemented in hardware. For $GF(p)$, the arithmetic is implemented as the arithmetic of integers *modulo* p. Gura [17] states that the Karatsuba algorithm, for binary field multiplication used in practical implementations, cause some irregularities that increase the delay in the architecture path. Comments like these must be taken into consideration if efficient finite field arithmetic and ECC implementations have to be achieved.

Orlando and Paar [18] state that multipliers for normal basis finite fields are prohibitively expensive in terms of area when that order is high (400, 500). Optimal normal basis (ONB) shows some improvements in efficiency but there are few fields for which there exist this kind of basis, so this kind of basis is not suitable for an iteroperable solution.

Two methods for inversion are often used: the Fermat Theorem and some variant of the Extended Euclidean Algorithm (EEA). The first one computes the inversion operation by exponentiation. EEA is faster but costly to implement.

A survey of methods to compute the finite field arithmetic for both GF(2^m) and GF(p) is found in [25] and [31].

2.3 ECC implementations

This section surveys previous work related to implementation of elliptic curve cryptography either in hardware or software. The main objective in the reported works of ECC implementation has been the efficient implementation of the dP operation, specially the implementation of dP as fast as possible. The strategies adopted for reducing the dP computation time have been in general the following:

1. Reducing the number of ECC-Add and ECC-Dbl operations using special

Table 2.3: ECC implementation on general purpose processors

Ref.	ECC scheme	Operation	Time (ms))	Processor
[34]	ECDSA-p160	Sign Verify	46 94	ARM7TDMI
[35]	ECDSA-r191	Sign	650	16-bit RISC processor 5MHz
[36]	ECDSA-p256	Sign Verify	7 18	ARMSA1110 206 MHz
[37]	- - - - -	dP 163 dP 233 dP 283 dP 409 dP 571	3.6 6.4 9.7 19.8 44.9	Intel 1 GHz

representations of the scalar d, as the NAF (Non Adjacent Form) representation.

2. Processing more than one bit (digit) of d at time (the window-based methods).

3. Parallelizing the execution of ECC-Add and ECC-Dbl operations, as in the left to right version of the binary method or using the Montgomery one.

4. Improving the computation time of finite field operations, either parallelizing the field operations or using more than one unit for each field operation.

Representative implementations of ECC in software and hardware are discussed in the next sections.

2.3.1 ECC in software

Research on efficient ECC software implementations has been carried out because of elliptic curve arithmetic is not supported by conventional microprocessors. Table 2.3 shows representative implementations of ECC in software.

These works show the complexity of elliptic curve cryptography implemented in RISC (Reduced Instruction Set Complex) constrained devices and workstation processors like Intel or Sparc. Aydos [34] implemented the digital signature scheme ECDSA on the prime finite field GF(p) with p of 160 bits while Krasner

[35] used the prime finite field with p of 256 bits. Grobschald [36] reported the ECDSA implementation for the binary finite field GF(2^{191}). In ECDSA, the most time consuming operation for signing is one scalar multiplication dP while the time for the verification operation is almost the same for computing two scalar multiplications. This implies that the time for dP in [34] is around 46 ms and the one in [36] is 7 ms. The time for dP in [35] is higher, 650 ms. Weimerskirch [37] reported better implementation results for computing one scalar multiplication dP in GF(2^m) for the security levels 163, 233, 283, 409 and 591 recommended by NIST [7]. The code was executed on three different processors: SPARC 32-bit 900 MHz, SPARC 64-bit 900 MHz and Intel 1 GHz. The windowed NAF algorithm for point multiplication and López-Dahab projective coordinates were used. The arithmetic algorithms for field multiplication were Karatsuba and Comb. The best results were obtained for the Intel platform using the Comb method for field multiplication, the results are shown in table 2.3 only for comparison purposes against the results obtained using RISC processors.

ECC implementations in general purpose processors and embedded systems can meet some security requirements in some applications but in other, hardware implementations are needed due to the requirements of throughput, power consumption, area constrains and physical security. ECC implementations in hardware have been another research subject. Representative hardware implementations of ECC are presented and discussed in the next section.

2.3.2 ECC in hardware

ECC hardware implementations are aimed to optimize each stage of scalar multiplication dP in figure 2.2:

1. **Field-Stage optimizations**. Choose fields with fast multiplication and inversion.

2. **Coordinates and Scalar multiplication-Stage optimizations**. Reduce the number of field inversions (projective coordinates). Reduce the number of point additions (windowing). Replace point doubles (endomorphism methods).

Works reported in the literature have used reconfigurable devices, FPGAs, to implement ECC algorithms. Computing the scalar multiplication dP as fast as

possible is the main objective of these works. Hardware architectures for dP reported in the literature can be divided into processor or co-processor approaches. In the former, there exist a number of specialized instructions the processor decodes and executes, most of them are for elliptic curve and finite field arithmetic. In the latter, there are no such instructions because the algorithms are implemented directly on specialized hardware. In general, both kinds of implementations are based in a regular structure. They often include:

- A **processing unit** that performs arithmetic operations. The main arithmetic modules are finite field multiplication, squaring and inversion. The organization of this module varies depending on the implemented arithmetic algorithms and the coordinates being used. An efficient design of this module impacts the efficiency of dP computation.
- A **control unit** that commands the data flow and the execution of the dP algorithm. This module has been implemented as a finite state machine or by using microcode.
- A **storage unit** for intermediate results and input operands or output results. Often, this is a register file where each register has the same width as the size in bits of the finite field elements (greater than 113 bits).
- An **interface unit** that provide the interface between the ECC processor or co-processor with a host processor.

Many considerations must be taken into account when these blocks are implemented in hardware, mainly for applications where an area/performance trade off is important. While a custom implementation can perform the operation dP faster, it is difficult to change to another algorithm parameters, which is desirable for a flexible solution.

A survey of reported processors and co-processors is shown in tables 2.4 and 2.5. Both tables show the selection at each layer of dP and the timing to perform this operation in *ms*.

From these results, some questions arise, for example:

How much the dP method affects the timing and area of the architecture?
How much the coordinated affect the organization of the arithmetic units?

Table 2.4: Approaches taken in ECC co-processors in GF(2^m)

Ref.	m	dP **method**	**Coordinates**	**Basis**	**Multiplier**	**Time** ms
[38]	270 191 155	Binary	Projective	ONB	3 Massey-Omura 5 Massey-Omura 7 Massey-Omura	6.8 2.3 1.2
[20]	191	Binary	Jacobian	Polynomial	4 LFSR	3.7
[39]	113	Montgomery	Projective	Polynomial	Karatsuba	10.9
[19]	151 176 191 239	Binary	Affine	Polynomial	ABC coprocessor	5.1 6.9 8.2 12.8
[22]	163	Binary Binary NAF Binary τ-adic NAF	López-Dahab	Polynomial	Digit-Serial digit=41	0.26 0.23 0.07
[40]	191	Montgomery	López-Dahab	Polynomial	Karatsuba	0.05
[21]	113	Montgomery	López-Dahab	ONB	2 Bit-serial	0.27
[41]	113 155 281	Binary	Affine	ONB	Bit-Serial	3.7 6.8 14.4

Table 2.5: Approaches taken in ECC processors in GF(2^m)

Ref.	m	dP **method**	**Coordinates**	**Basis**	**Multiplier**	**Time** ms
[18]	167	Binary Montgomery	Jacobians López-Dahab	Polynomial	Digit-Serial (digit = 4) Digit-Serial (digit = 8) Digit-Serial (digit = 16) Digit-Serial (digit = 4) Digit-Serial (digit = 8) Digit-Serial (digit = 16)	0.96 0.61 0.36 0.55 0.35 0.21
[17]	k163 k193 k233 163 193 233	Montgomery	López-Dahab	Polynomial	Digit-Serial digit = 64	0.14 0.18 0.22 1.5 1.83 2.21
[23]	160	Addition-Sub chain NAF	Jacobians	Polynomial	64-*bit* dual Montgomery	0.19
[42]	160	Binary	López-Dahab	Polynomial	Systolic Montgomery	3.810

How the arithmetic unit organization will affect the performance of the computation?

Although it is well known that the arithmetic unit has a big impact in the timing and area of the dP core, it is not clear if the architecture is fast because of the parallelism in the multipliers, the number of multipliers, or the kind of multipliers. A flexible architecture that allows to change the implementation parameters and analyze the impact in performance and computational cost is necessary and this thesis aims to provide such architecture.

Table 2.6 shows the diversity of technology used to implement elliptic curve cryptography in hardware. It shows the differences in area resources and timing achieved for different selections of the ECC parameters. From the results shown in this table it is not clear why such different results are achieved:

Are they due to the arithmetic modules or to the ALU organization?, if they were, *How much simpler arithmetic algorithms will impact the overall performance?.*

Table 2.7 shows the different approaches to implement the three layers of dP computation. The control unit often implements the dP method using a finite state machine (FSM), microcode, or it is implemented as a set of software instructions executed by the arithmetic unit. The storage unit has been implemented using the memory blocks of the FPGA or as a file register of width m, using the logic of the FPGA. The arithmetic unit has been implemented using several algorithms for multiplication and different number of such multipliers. It is not clear how these choices will impact the area resources of the ECC co-processor and what are the advantages of using one of the reported multipliers: Karatsuba, LFSR (Linear Feedback Shift Register), Massey Omura or digit-serial. The same is for the inversion and squaring algorithms. The last column of table 2.7 indicates if the ECC hardware module provides (A) or not (NA) a host interface for interacting with a master module that commands the execution of dP. The information missing for any of the columns of table 2.7 is represented as ('-').

Although some reported works allow a kind of flexibility in the ECC parameters, a reconfigurable architecture enabling interoperability with other designs for mobile wireless devices is not explored at all. Also, although in some works the

Table 2.6: Devices used, area consumption and execution time in ECC implementations in GF(2^m)

Ref.	m	**Device**	**Area**	**Freq.**	**Time** (ms)
Co-processors					
[38]	155	XC4085XLA	1976 slices	37 MHz	1.2
	191		2164 slices	36 MHz	2.3
	270		2572 slices	34 MHz	6.8
[20]	191	XCV1000	-	50 MHz	3.72
[39]	113	AT94K40 Amtel	38.4 Kgates	12 MHz	10.9
[19]	<255	XCV2000E	4048 slices	40MHz	5.1
[22]	k163	XCV2000E	5,008 slices	66 MHz	0.07
[40]	191	VirtexE 3200	18314 slices	9.9 MHz	0.05
[21]	113	XC2V6000	6961 Slices	56 MHz	0.27
[41]	113	XCV300-4	1290 slices	45 MHz	3.7
	155		1567 slices	36 MHz	6.8
	281		2622 slices	33 MHz	14.4
Processors					
[18]	167	XCV400E-8	1501 slices	76.7 MHz	0.55
[17]	<255	XCV2000E	-	66.4 MHz	-
[23]	160	ASIC	118 Kgates	510 MHz	0.19
[42]	160	XCV800	138 - 150 Kgates	47 MHz	3.81

Table 2.7: Hardware organization in reported ECC implementations in GF(2^m)

Ref.	**Control Unit**	**Storage Unit**	**Arithmetic unit**	**Host Interface**
Co-processors				
[38]	Binary FSM	16 m-*bit* registers	7 Massey-Omura Mult.	NA
[20]	Binary FSM	Dual-port RAM	4 LFSR Mult. 2 Squarers	A
[39]	Montgomery Sw		5 23-*bit* Karatsuba Mult.	A
[19]	Binary (Sw)	RAMs in FPGA	ABC processor	A
[22]	Binary FSM	11 m-*bit* registers	Digit-serial Mult.	A
[40]	Montgomery FSM	16 m-*bit* registers	2 Karatsuba Mult.	NA
[21]	Montgomery Sw	4 registers	4 ONB Mult.	A
[41]	Binary Microcode	16 m-*bit* registers	ONB Mult.	A
Processors				
[18]	Binary, Mont. Sw	File register	Digit-Serial Mult.	A
[17]	Montgomery Sw	10 256-*bit* registers	Digit-Serial Mult.	A
[23]	-	-	Dual field Mult.	-
[42]	Binary FSM	-	Systolic Montgomery Mult.	-

design of the arithmetic units is parameterizable in the order field, the architecture needs to be reconfigured out of line for other finite fields orders. It would be desired a real time adaptation of the hardware architecture to different security levels.

In the proposed ECC hardware cores, the main objective is to perform dP as fast as possible. Parallelization at the three layers of dP hierarchy have been performed. Different arithmetic modules and datapath organizations have been used. For control, some works have used finite state machines or micro-programed modules.

Only [17] proposes to manage different elliptic curves without reconfiguring the hardware by implementing wired reduction for three of the NIST curves and implements the technique called *partial reduction* for arbitrary curves. Other works like [18], [19] and [20] manage different elliptic curves but need to reconfigure the hardware out of line. Other works like [21] propose a HW/SW partition and use reconfigurable logic only for arithmetic instructions. An example of customized implementation for an specific elliptic curve is [22]. Other efforts to achieve ECC interoperability propose an unified arithmetic unit for both prime and binary fields arithmetic [23, 24].

Also, from tables 2.6 and 2.7 it can be seen that different ways of implementing ECC leads to different performances. When there are no problems of area we can choose the best algorithms considered in the literature in each stage of figure 2.2.4 and implement them. But it is not always required, for example when implementing ECC for constrained devices. In order to achieve interoperability of ECC it is necessary a solution that gives support to different finite fields and elliptic curves.

2.3.3 ECC implementations and side channel attacks

Side channel attacks were introduced by Paul C. Kocher in 1996 [43]. A side channel attack (SCA) is any attack based on information gained from the physical implementation of a cryptosystem, rather than theoretical weaknesses in the algorithms [44]. That is, SCA attacks look at the way cryptographic algorithms are implemented, rather than looking at the algorithm itself.

In SCA, extra source information such as timing, power consumption, electromagnetic leaks or even sound can be exploited to break the system. The

assumption of *timing attacks* is that the duration of the execution of an algorithm depends on the secret key, and analyzing these durations provides some information about the secret key. Timings only require a simple chronometer as sensor, therefore, this category of attacks has a very wide application range. Principles of side channel attacks are based on the observation of the *power consumption*. There are two different classes of power consumption-based attacks: *simple power analysis* or SPA, where the attacker analyzes one single power trace for revealing the secret key, and *differential power analysis* or DPA, where a statistical tool allows to extract the smallest details in power traces from intermediate values collected from multiple cryptographic operations. Sometimes the dependence on the secret key induces tiny differences in the power trace, and these differences are embedded in the noise. To extract such information, attackers may use the averaging technique, where the noise level is decreased by averaging the power traces with different inputs to the cryptosystem. The principle of DPA is to guess the value of some bit of the secret key and verify the validity of the assumption with the collected power traces.

Timing and simple power analysis attacks are the most common for any hardware implementation of the dP operation. Timing attacks on a implementation of the algorithms for computing dP is possible due the time to perform this operation depends on the bit values of d. If the scalar d has $\frac{\lceil \log d \rceil}{2}$ '1's on average, then the binary algorithm performs ($\lceil \log d \rceil \cdot$ECC-Dbl$+\frac{\lceil \log d \rceil}{2} \cdot$ECC-Add) operations. The formulas for ECC-Dbl and ECC-Add on a Weierstrass elliptic curve are in essence different. Therefore, a simple power analysis will produce different power traces that may reveal the value of scalar d in the binary method from the distinction between the two operations. Also this distinction can allow timing attacks.

One of the proposed countermeasures to thwart SCA attacks on ECC hardware implementations consists in ultimately having an algorithm that behaves consistently and regularly whatever the processed data, for example the Montgomery algorithm for dP (see algorithms 2.1 and 2.2 in section 2.2.4 of this chapter). The algorithms for ECC-Add and ECC-Dbl can be transformed into a regular algorithm [45, 46] in a such way that doubling and addition be indistinguishable. This last approach is used in this work for developing an interoperable hardware architecture for ECC resistant to side channel attacks.

2.4 Summary

This chapter presented the mathematic background of elliptic curve cryptography and its implementation issues. It presented and discussed the cryptographic schemes and reported works in the literature about software and hardware implementation of these cryptographic schemes and the scalar multiplication dP. This chapter showed the importance of having not only an efficient but also a resistant ECC hardware architecture to side channel attacks.

The next chapter gives an introduction to reconfigurable computing and presents the design methodology used in this thesis for the development and implementation of the reconfigurable system for interoperable elliptic curve cryptography.

Chapter 3

Reconfigurable computing and design methodology

This chapter presents the background on reconfigurable computing and the design flow for developing reconfigurable systems. It also describes the design methodology to design and implement the reconfigurable and interoperable hardware architecture discussed in this thesis.

3.1 Reconfigurable computing

There are two primary methods in conventional computing for the execution of algorithms [47]. The first one is to use hardwired technology, either an Application Specific Integrated Circuit (*ASIC*) or a group of individual components forming a board-level solution, to perform the operations in hardware. ASICs are designed specifically to perform a given computation, and thus they are very fast and efficient when executing the exact computation for which they were designed.

However, the circuit cannot be altered after fabrication. This forces a redesign and re-fabrication of the chip if any part of its circuit requires modification. This is an expensive process, especially when one considers the difficulties of replacing ASICs in a large number of deployed systems. Board-level circuits are also somewhat inflexible, frequently requiring a board redesign and replacement in the event of changes to the application.

The second method is to use software-programmed microprocessors, a far more flexible solution. Processors execute a set of instructions to perform a computa-

tion. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance can suffer, if not in clock speed then in work rate, and is far below that of an ASIC. The processor must read each instruction from memory, decode its meaning, and only then execute it. This results in a high execution overhead for each individual operation. Additionally, the set of instructions that may be used by a program is determined at the processor fabrication time. Any other operations that are to be implemented must be built out of existing instructions.

Reconfigurable devices are intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware.

Reconfigurable computing is a computing paradigm combining some of the flexibility of software with the high performance of hardware by processing with very flexible high speed computing fabrics like FPGAs. The main differences when compared to using ordinary microprocessors are:

1. The use of spatial computation instead of temporal computation.

2. The ability to make substantial changes to the datapath itself in addition to the control flow.

On the other hand, the main difference with custom hardware (ASICs) is the possibility to adapt the hardware during runtime by "loading" a new circuit on the reconfigurable fabric. So, reconfigurable hardware in reconfigurable computing acts as a "general" hardware accelerator, implementing a variety of different computations within or across applications. Compute intensive sections of applications can be swapped into the hardware when needed, and later swapped out to make room for other computations.

Reconfigurable devices have been used to develop *reconfigurable systems*, which are based in the combination of microprocessors and reconfigurable logic. Custom hardware specifically handle compute-intensive highly-parallel sections of application code. The processor controls the hardware, and executes the parts of applications not well-suited to hardware. The coupling methods are best differentiated by how and how often the RH and host processors(s) interact. Reconfigurable systems are classified in three types, these are [48]:

1. **Attached Processors**. In this type, the reconfigurable logic is connected to the I/O bus or to the main memory of the microprocessor and it does not extend the instruction set of the microprocessor. Examples of this type of systems are Splash, Splash2 [49], DECPeRLE-1 [50], and PRISM-I [51].

2. **Co-processors**. In this type, the reconfigurable logic is part of the microprocessors and it is located near to it. Examples of these systems are HARP [52], Garp [53], Spyder [54] and RENCO [54].

3. **Reconfigurable Functional Unities (RFUs)**. In these types of systems, the reconfigurable logic is "inside the microprocessor" and the microprocessor treats the reconfigurable logic as one of the standard units in the datapath. The instruction decoder addresses instructions to the reconfigurable logic. Examples of these systems are Nano [55], DISC (Dynamic Instruction Set Computer) [56], MorphoSys [57], OneChip [58], Chimaera [59] and Proteus [60].

3.2 Reconfigurable devices

A field-programmable gate array, or FPGA, is a semiconductor device containing programmable logic components called *logic blocks*, and *programmable interconnects* [61]. Logic blocks can be programmed to perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions such as decoders or mathematical functions. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. A hierarchy of programmable interconnects allows logic blocks to be interconnected as required by the system designer, somewhat like a one-chip programmable breadboard. Figure 3.1 shows the general FPGA layout and the architecture of a single logic block. As shown in figure 3.1, the logic block consists of a 4-input function generator that allows to implement any 4-input combinatorial boolean function. The result of this generator can be stored in a one bit register or it can be delivered to another logic block for further processing. The function generator or LUT (Look Up Table) can also be used to implement a 16x1 memory block.

Logic blocks and interconnects can be programmed by the designer, after the

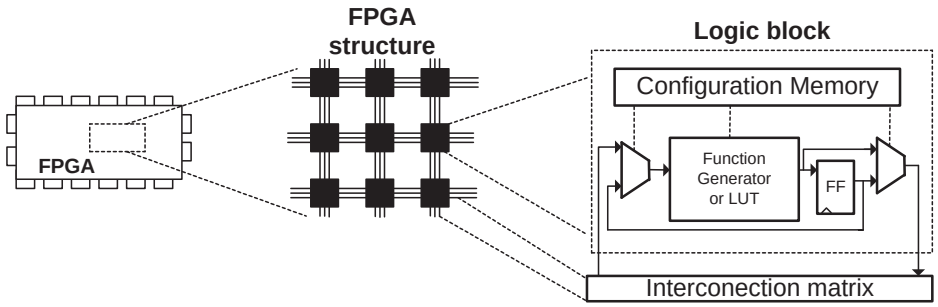


Figure 3.1: FPGA internal structure

FPGA is manufactured, to implement any logical function, hence the name "field-programmable". FPGAs are usually slower than their application-specific integrated circuit (ASIC) counterparts, they cannot handle a complex design and draw more power (for any given semiconductor process). But their advantages include a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs.

Through the years, FPGAs features have been improved and their density has grow. Current FPGAs have embedded processors, giga-bit serial transceivers, clock managers, analog-to-digital converters, dedicated digital signal processing blocks, Ethernet controllers, substantial memory capacity, and other dedicated functional blocks beyond the basic arrays of simple logic elements they started out with in the mid 1980s. The current high density of FPGAs allows to implement complete systems (*System-on-Chip* or SoC) on them.

In addition, the capacity of reconfiguration of FPGAs has increased. The best advantage and the opportunities to design using these devices resides in the way the reconfiguration is performed. The FPGA reconfiguration is based on the SRAM (Static Random Access Memory) technology. The configuration of the device is guided by data stored in the configuration memory. This content determines the interconnection among the configurable blocks and the function these blocks perform. Usually, the configuration memory stores just one configuration (one-context) but some devices can store more than one (multi-context). SRAM memory is volatile so the FPGA must be configured normally by an external memory non volatile each time the FPGA is powered up.

FPGAs as reconfigurable hardware, provide a flexible medium to implement hardware circuits [61, 62]. To implement a needed circuit in reconfigurable hard-

ware, a CAD flow transforms its descriptions into an reconfigurable hardware configuration. First, the circuit is *synthesized*, converting the circuit schematic or hardware design language (HDL) description into a structural circuit netlist. Then a technology *mapper* further decomposes that netlist into components matching the capabilities of the basic blocks (LUTs, ALUs, etc.) in the reconfigurable hardware. Next, the *placer* determines which netlist components should be assigned to which physical hardware blocks, and a *router* decides how to best use the routing resources to connect those blocks to form the needed circuit. Finally, the CAD flow determines the specific binary values to load into the configuration memory for the determined implementation.

The *bitstream* or configuration data are loaded into the FPGA SRAM memory through special configuration pins. These configuration pins serve as the interface for a number of different configuration modes [61, 62]:

- **Master-serial**. The bitstream is loaded serially and the clock signal for loading is generated by the FPGA itself.
- **Slave-serial**. The bitstream is loaded serially but the clock signal for loading is generated by an external device.
- **Master SelectMAP (parallel)**. The bitstream is loaded in parallel (8 bits) and the clock signal for loading is generated by the FPGA itself.
- **Slave SelectMAP (parallel)**. The bitstream is loaded in parallel (8 bits) but the clock signal for loading is generated by an external device.

FPGA technology allows several ways to change the configuration of the logic blocks and interconnections. These are named as:

- **Total or static reconfiguration**. Every element in the FPGA is reconfigured.
- **Partial or dynamic reconfiguration**. Specific parts of the FPGA are reconfigured while the rest of the device keeps its configuration.
- **Self reconfiguration**: Extends the concept of dynamic reconfiguration. The FPGA uses part of itself to control the reconfiguration of other parts of it. Both the dynamic reconfiguration and the self reconfiguration need an

external mechanism for configuring the FPGA when this is powered up by first time or when the device is restarted.

FPGAs have become mainstream already years ago in all kinds of embedded systems. FPGAs are rapidly moving into practically every application area, such as automotive, aerospace, defense, medical, chemistry, molecular biology, physics, astrophysics, high performance computing, supercomputing, and many other areas.

3.3 Design methodology

Implementing ECC in hardware is a complex task. Complexity arises from the almost endless number of possibilities how elliptic-curve operations can be calculated in hardware. A straight-forward approach will not be able to produce hardware that optimizes and balances silicon area, performance, and power consumption as excellent as a structured approach is able to. As it was shown in chapter 2, there are many options for implementing elliptic-curve operations. Beginning with a multitude of different algorithms for implementing the scalar multiplication and ending up at multiple possible representations of finite-field elements. Every option will have an impact on the desired design goal. It is the task of a considered hardware design-methodology to evaluate different options and to compare them to find out which option (or even more complicated which combination of different options) is the best for implementing a fully-functional ECC hardware at lowest cost in area or at the highest performance.

Neil Weste [63] describe four techniques that should be applied during the development of any digital circuit: hierarchy, regularity, modularity, and locality.

- By enforcing **hierarchy**, it is possible to bring in abstraction into the design. Abstraction is necessary to handle complexity by hiding distracting details. Hierarchy is obtained by subdividing hardware modules into a set of smaller submodules, which are more comprehensible than larger modules. Hierarchy helps to lower the complexity of (sub-) modules and improves their reusability.

- **Regularity** means that the hierarchical decomposition of a large system should result in not only simple, but also similar blocks, as much as possible.

Regularity usually reduces the number of different modules that need to be designed and verified, at all levels of abstraction.

- **Modularity** demands well-defined interfaces for sub-modules. Well defined interfaces facilitate assembling larger modules from submodule instances.

- **Locality** is a design strategy that hides details of modules. Internal construction details should be hidden inside a cell to abstract its functionality and other characteristics.

In order to develop a complex digital system that offers a well-balanced mixture of the quality aspects mentioned before, it is necessary to apply a structured design methodology that is capable to detect potential flaws and weaknesses as early as possible to shorten the design time.

A top-down approach that subdivides the problem of developing efficient digital circuits into several layers of abstraction is recommended [64]. The highest level of abstraction will define the intended functionality and some boundary conditions under which the circuit should work. The lowest level will represent the physical implementation of the circuit in silicon. A top-down design methodology creates hardware by defining the highest level of abstraction first and refines it using some intermediate abstraction levels until the physical implementation is obtained. The top-down methodology with its different abstraction levels keeps the complexity of each abstraction level within limits. Abstraction hides details of lower levels. Levels of abstraction in digital circuit design comprise the system level, the algorithmic level, the architectural level, the register-transfer level, and the circuit level.

The usage of a top-down approach to design a complex digital system will automatically subdivide the overall problem into a number of smaller problems. The decomposed problems are smaller and their solution will add up to solve the overall problem. The smaller problems can be classified into several layers. The design flow discussed in the next sections was used to develop the reconfigurable system that allows interoperability for elliptic curve cryptography. It applies a top-down methodology, which obeys the design principles discussed previously. The design flow is divided in three stages. In the first stage, the design flow models an elliptic curve co-processor on various abstraction levels and obtains a physical realization of it by refining the abstraction levels down to the circuit level and

the physical level. This design flow emphasizes early design evaluation on high abstraction levels to yield highly optimized circuits and to prevent re-iterating design stages. Basic elements of the design flow are a C-based high-level model for evaluating different algorithms, a cycle-accurate VHDL model for simulation and synthesis, and backend (background) methods for mapping the circuit on FPGA technology. Continuous verification plays an important role to ensure correct functionality and conformance to defined constraints. In the second stage, the design flow implements a dynamically reconfigurable system that provides interoperability for elliptic curve cryptography at the ECC arithmetic level. The third stage comprises the verification and validations of the proposed system by applying testbeds.

During the whole design flow several high level tools are used. Scripts are well suited for the integration of these different tools and for the connection of various CAD programs. They help to automate the design flow.

3.3.1 Design flow for ECC hardware architectures

Figure 3.2 shows the block diagram for the development of the ECC co-processor in the first stage of the desing flow. The high-level model (software model) of a circuit helps to understand the circuits functionality and the algorithms that are required to implement the functionality. After exploring these high-level models, the next step in the hardware design flow is to find a hardware architecture that implements the algorithms efficiently. There are no sophisticated tools that support this task, which is carried out relying on the experience and the creative ideas of the hardware designer. More than one hardware architecture will result and a selection of the best one will be done. Criteria for this selection are: *i*) the area resources occupied, expressed as the number of logic gates or elemental elements in the targeted device; *ii*) the performance, which consist in counting the number of clock cycles that are necessary to complete a computation and the delay of the critical path, which determines the maximum clock frequency of the hardware architecture, and *iii*) the power consumption.

Hardware description

The hardware architecture of the elliptic curve crypto co-processor addressed in this thesis is described in the hardware description language VHDL [65] and

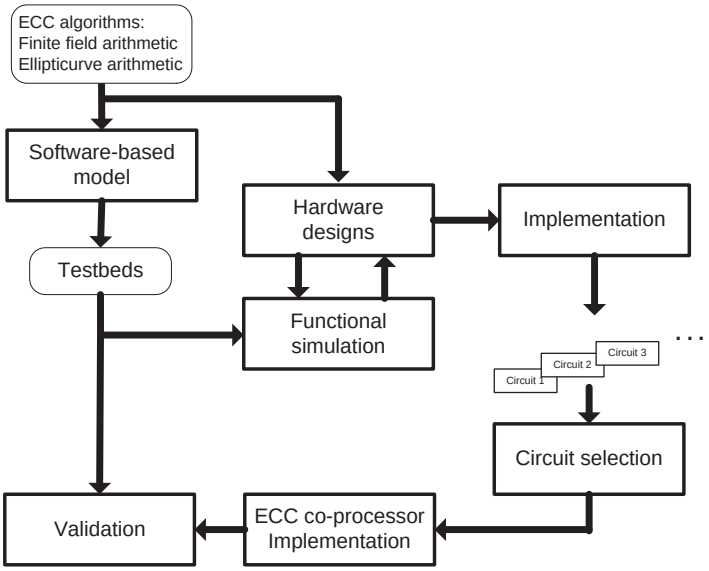


Figure 3.2: Flow for developing the ECC co-processor

physically implemented on FPGA technology according to the Xilinx flow shown in figure 3.3.

Design entry begins with a design concept, expressed as a drawing or functional description. The design is created using a schematic editor, a Hardware Description Language (HDL) for text-based entry, or both. From the original design, a netlist is created, then synthesized and translated into a Native Generic Object (NGO) file. This file is fed into a program called NGDBuild, which produces a logical Native Generic Database (NGD) file. Figure 3.4 shows the design entry and synthesis process. The design may be constrained within certain timing or placement parameters. Mapping, block placement, and timing specifications may be specified. Constraints can be given by hand or using a Constraints Editor, Floorplanner, or FPGA Editor. Block placement can be constrained to a specific location in the FPGA, to one of multiple locations, or to a location range. Locations can be specified in a User Constraint File (UCF). Poor block placement can adversely affect both the placement and the routing of a design. Typically, only I/O blocks require placement to meet external pin requirements.

The VHDL description does not contain any specific target-technology so the synthesis can map the VHDL description on FPGAs as well as on other technologies as the standard-cell one. The VHDL code can be also optimized to yield the best synthesis results. The VHDL model is hierarchically decomposed into

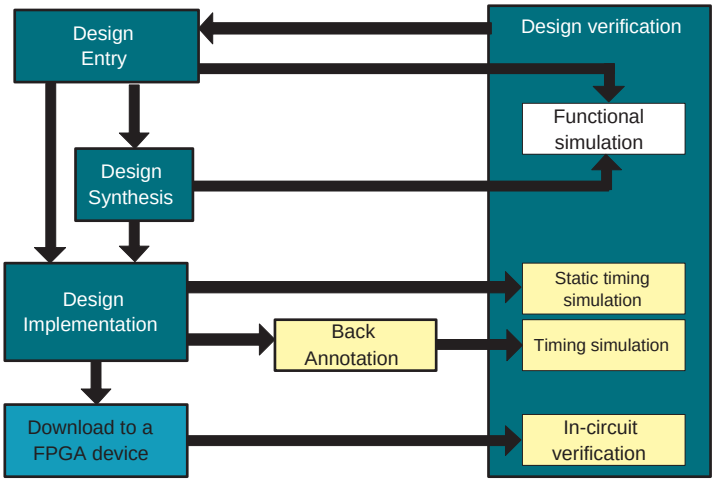


Figure 3.3: Design flow for FPGA-based digital circuits.

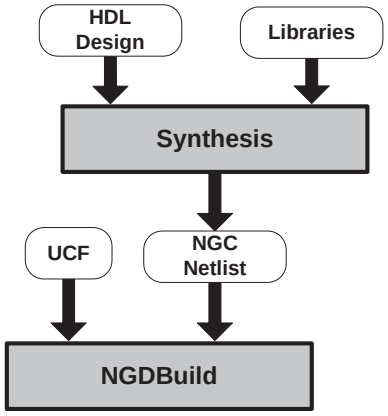


Figure 3.4: Design entry and synthesis process.

modules for facilitating functional verification and improving reusability of components. Modeling hierarchical (or regular) structures in HDLs is called *structural description*. The structural description style for modeling hardware can have advantages for backend tasks like floorplanning and placement. By composing a circuit as a set of relatively small modules and sub-modules, floorplanning activities, which allocate regions of the silicon area for certain circuit parts, are facilitated. The same applies for FPGAs, where configurable blocks are reserved for certain circuit functionality. Floorplanning and placement is especially of interest for regular structures like datapaths. Placing neighboring modules on silicon will shorten the interconnect. In modern process technologies, shortened interconnect improves the circuits performance and lowers power consumption as much as optimizing combinational logic. This consideration is also true for FPGAs, where interconnect contributes predominantly to delay and power consumption. Bit-slice architectures for datapaths are also of interest for writing parameterizable models.

Parameterization allows hardware models to be configured for different hardware sizes. Parameters can determine for instance the width of a datapath. Parameterization improves the reusability of hardware models because the same model can be easily adapted for different purposes. Parameterization of VHDL models uses the *generic* statement, which extends the interface of modules. VHDL strictly separates interfaces from functionality: *Entities* define interfaces of modules, and *architectures* define the functionality of that module.

Simulation

Before mapping the VHDL model onto the target technology by means of synthesis, simulation has to assure the correctness of the model's functionality. Simulation is a method of dynamic circuit verification, which excites the circuit or models of it by applying test patterns and by analyzing the circuits response. There are numerous simulators for VHDL, one of them is Active-VHDL, which was used to verify the ECC VHDL models described in this thesis.

There are several options for verifying a VHDL model by simulation. The options emerge from different possibilities for applying test patterns and for checking the response of the circuit. Prevalent methods are using testbeds or controlling the simulator by scripts. Testbeds, which are sometimes called testbench, are

written in the same HDL as the tested hardware, which is called unit-under-test (UUT). The testbed instantiates the UUT and stimulates its inputs. Input stimuli are generated by assigning outputs of the UUT in each clock cycle accordingly. The testbed may also check the responses of the UUT by monitoring the UUTs outputs and by comparing the values with expected values. In case the testbed includes no verification, the designer has to analyze the simulator output manually. An advantage of testbeds is that they use the same language for verification as for modeling.

Synthesis

The synthesizer translates the HDL code written on register-transfer level into a *netlist* which comprises only gates of the target library. The synthesis process has two stages: Firstly, a technology independent synthesis step produces a description on logic level; a second step maps this description onto the target technology. The mapping step optimizes its output by considering cell characteristics stored in the synthesis library. Such synthesizers are able to adhere to performance constraints and limitations regarding power consumption. In particular, synthesizers perform a static timing analysis to extract the critical path. This ensures that the maximum clock frequency is high enough. Advanced synthesizers are even able to apply measures for lowering the power consumption. For instance, they can insert clock-gating techniques to reduce signal activity that consumes power unnecessarily. In this thesis, the synthesizer included in the framework ISE from Xilinx was used. The synthesis task can also be controlled by scripts. Simple scripts are sufficient to turn VHDL code into standard-cell circuits.

After synthesizing the design the next step is the design implementation. Design implementation begins with the mapping (MAP) or fitting of a logical design file to a specific device and is complete when the physical design is successfully routed and a bitstream is generated. Constraints can be altered during implementation just as during the Design Entry step. The overall view of the design implementation process for FPGAs is shown in figure 3.5.

The input to MAP is an NGD file, which contains a logical description of the design in terms of both the hierarchical components used to develop the design and the lower-level Xilinx primitives, and any number of NMC (hard placed-and-routed macro) files, each of them contains the definition of a physical macro.

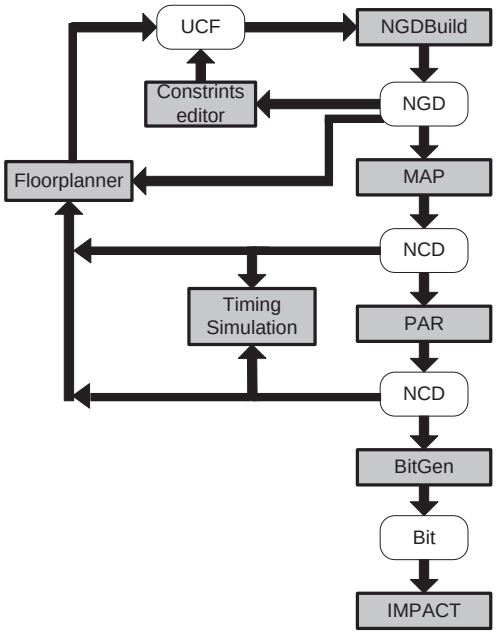


Figure 3.5: Design implementation

MAP then maps the logic to the components (logic cells, I/O cells, and other components) in the target FPGA. The output design is an Native Circuit Description (NCD) file, which is a physical representation of the design mapped to the components in the Xilinx FPGA.

Place-and-Route (PAR)

Floorplanning has to precede placement. Floorplanning is an early activity that locates basic blocks of a circuit on the chip area. This determines the global routing concept and the shape and the size of standard-cell blocks. Moreover, the desired position of input and output terminals gets predefined. All this information has to constrain the placement process. PAR takes an NCD file as input, places and routes the design, and outputs an NCD file, which is used by the bitstream generator, BitGen. Placement arranges standard cells in rows such that the routing effort is minimized. After placing the cells, a balanced and symmetric clock tree has to be inserted. This job is automated largely. It suffices to define constraints for delay, transitions times, and most importantly clock skew. The placement tools synthesize a clock tree and integrate it into the initial placement. The place and route utility included in the framework ISE from Xilinx was used

as placer and router in this thesis. Routing is the remaining task to complete the layout generation of standard-cell circuits. This process performs precisely the interconnections planned by the placement process. After placing and routing the design, the configuration file (bitstream) for the FPGA is generated and downloaded using for example, the IMPACT program provided by Xilinx.

BitGen produces a bitstream for Xilinx device configuration. BitGen takes a fully routed NCD file as its input and produces a configuration bitstream, a binary file with a .bit extension. The BIT file contains all of the configuration information from the NCD file defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device.

Backend Verification

Once a layout of the circuit is obtained, it has to be checked intensively to ensure manufacturability. One may assume that the layout of standard-cell circuits, which was generated by automated tools, is manufacturable. This is not true in general. Thus, full-custom circuits as well as semi-custom circuits need intensive verification of their layout data. Assuring that the layout data are correct gives confidence that manufactured chips will have the desired functionality and meet the specified constraints. Extensive backend verification pays off because detecting faults on produced chips is much more complicated, causes higher costs, and is more time consuming.

3.3.2 Design flow for reconfigurable ECC hardware architectures

Partial reconfiguration involves defining distinct portions of an FPGA design to be reconfigured while the rest of the device remains in active operation. These portions are referred to as reconfigurable modules. A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed. Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using a special *bus macro*. The implementation must be designed so that the static portions of the design do not rely on the state of the module under reconfiguration while reconfiguration is

taking place. The implementation should ensure proper operation of the design during the reconfiguration process. Explicit handshaking (e.g., module ready/not-ready) logic may be required. The state of the storage elements inside the reconfigurable module are preserved during and after the reconfiguration process. It is not possible to utilize the FPGA devices global set/reset (GSR) logic to independently initialize the state of the reconfigurable module. If set/reset initialization is required for the reconfigurable module, user-defined set/reset signals should be defined in the source HDL.

In order to implement partial reconfiguration on an FPGA, the FPGA must inherently support the dynamic reconfiguration of only portions of it, while leaving the other portions unaffected. Then a set of software development tools are needed that support the development of applications restricted to boundaries that comply with the hardware architecture of the FPGA. Finally, some form of basic controller must be available to dynamically manage the reconfiguration of the FPGA. This could be an embedded general-purpose processor (GPP), a soft core GPP, or an external GPP connected to the FPGA. In this shared resources model, the same embedded GPP that is running the design infrastructure and operating environment is also managing the partial reconfiguration of the FPGAs.

Successful implementation of a design using a partially reconfigurable flow requires following a strict design methodology. A reconfigurable design will consist of partially reconfigurable modules (PRMs) that will be swapped in and out of the FPGA and the static logic, which will remain in place. The general picture of the design flow involves the need to insert bus macros between the PRMs and the rest of the design, the static or fixed logic that remains in place. Bus macros are the channels or ports through which modules communicate and pass data. This allows a fixed communication channel for the static logic regardless of the reconfigurable logic on the other side. A layout of a design with a module that is reconfigurable (shaded) is shown in figure 3.6.

Creating a partial reconfiguration design requires the creation and implementation of the design within a set of specific guidelines. The block diagram of such guidelines is shown in figure 3.7. For more details on modular design, refer to [66]. A general description of the flow for partial reconfiguration is:

1. **Design Entry** - This phase consists on writing and synthesizing HDL code in conformance with partial reconfiguration guidelines. This usually implies

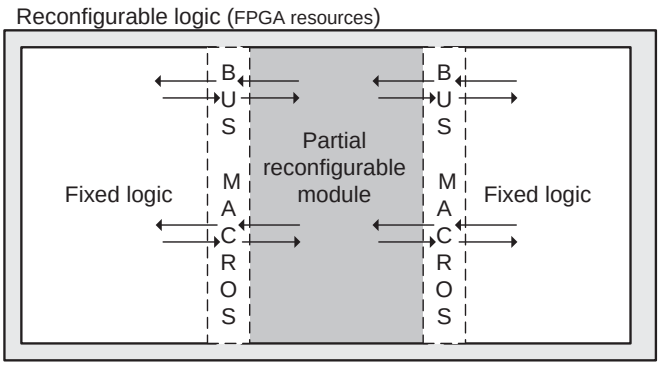


Figure 3.6: Design layout of a reconfigurable fabric with a reconfigurable module

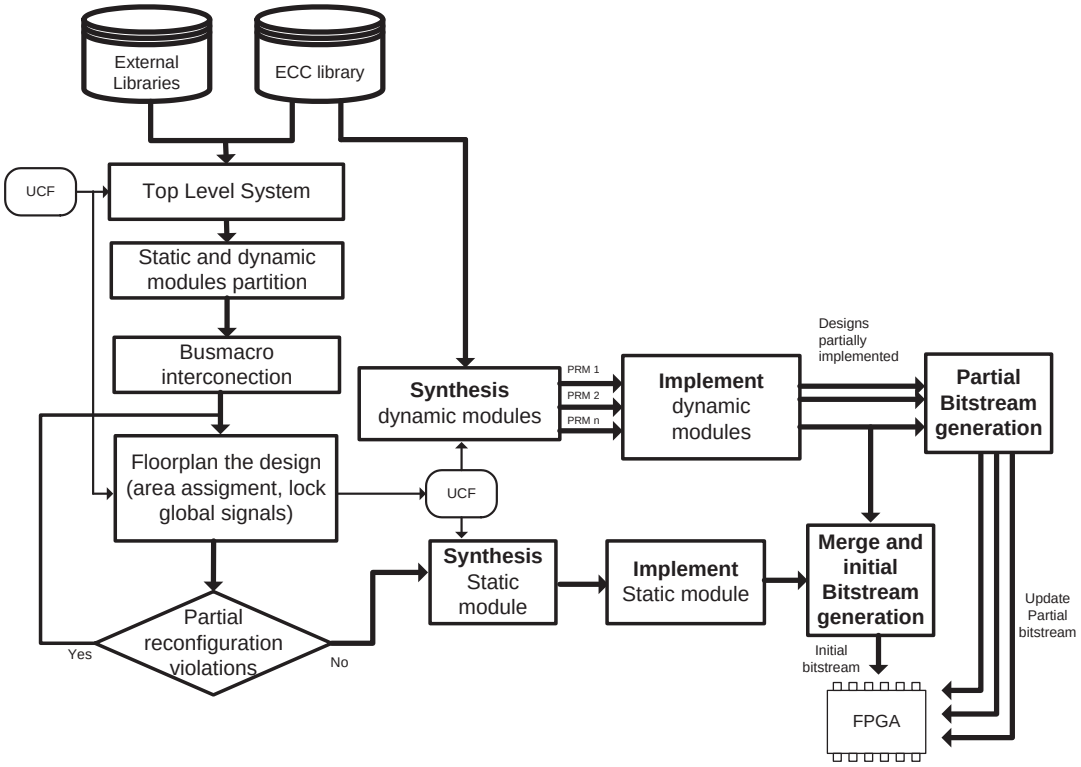


Figure 3.7: Design flow for partial reconfiguration

the partition of the original design in static and reconfigurable parts. Also, this phase includes the insertion of busmacros in the design for communicating the static and the reconfigurable parts.

2. **Initial Budgeting** - This phase consists on designing the floorplan, constraining the logic, and creating timing constraints for the top-level design and each single module, either static or reconfigurable. This phase can be carried out by using CAD tool like Planahead or Floorplaner for example. All the elements in the design must be correctly placed in the FPGA, otherwise the next phases in the flow will fail.

3. **Run Active Implementation (NGDBUILD, MAP, PAR, etc.)** - Each version of the reconfigurable module and each one of the static parts are implemented.

4. **Assembly Phase Implementation**- One of the partial reconfigurable modules is merged with the rest of the fixed logic to get one initial configuration file.

5. **Verify design** - A functional simulation and a static timing analysis is applied.

6. **Visually inspect design** - The FPGA Editor could be used to ensure that no unexpected routing crosses module boundaries. Though the software enforces this rule, it is still important to manually check this result.

7. **Create bitstream for full design** - The initial power-up configuration is created.

8. **Create individual (or partial) bitstreams** - One for each reconfigurable module.

9. **Setup** - Download device with initial power-up configuration.

10. **Reprogramming** - Reprogram reconfigurable modules as needed with individual (or partial) bitstreams.

3.3.3 Verification and Validation

Verification is one aspect of testing a product's fitness for purpose. Validation is the complementary aspect. Often one refers to the overall checking process as V & V [67]. For validation, designers must answer the question: 'Are we trying to make the right thing?', i.e., does the product do what the user really requires?. In the case of verification, designers must answer the question: 'Have we made what we were trying to make?', i.e., does the product conform to the specifications?.

In this thesis, all the developed hardware modules are verified and validated. The desired functionality is verified by simulation. Simulation assigns testdata to a HDL model and tests whether the response of the model is as expected. The most striking argument for using HDLs is the possibility to translate descriptions on register-transfer level by automated tools into structural descriptions on gate-level. Test data from several standards like ANSI X9.63 [8] could be used for verification of implementations. The recommended public ECC parameters from other standards like [2] could also be used for generating more test data using a software model.

Verification is an aspect that is important for all levels of abstraction. Verification ensures correct functionality of the circuit. Besides checking the correct input/output relationship of data, it is also important to ensure that the circuit meets all constraints. Early detection of violations of constraints lowers the design effort for doing a redesign. Verification is an essential part of the design process. It ensures the consistency of circuit models on different layers of abstraction. Consistency can be enforced when models on higher levels of abstraction generate testdata for a more detailed representation. For instance, a high-level software model can be extended to record intermediate results. So, testdata for modules and sub-modules can be generated. Simulation of HDL models and sub-modules can take this testdata as stimulus and can compare results. Automated test scripts for all modules and sub-modules will help to detect undesired side effects of design changes. Hierarchical verification that verifies a module not until all sub-modules have been verified, helps to track down failures quickly. Hierarchical verification can avoid cumbersome debugging of large circuits and circumvent long simulation times. Simulation is the standard method for consistency checking. It is a dynamic verification method. Anyhow, static verification gains ground. Static verification uses formal methods to prove that two representations of a module

on different abstraction levels are identical. Identical means that they both have the same functionality when all defined constraints are considered. Commercial tools for proving that the circuit level and the gate level are identical are for a long time on the market: layout-versus-schematic checker. Formal verification tools that prove that a representation on gate level is identical to a behavioral description are emerging. Formal verification has the advantage that it scales better with the complexity of a circuit than simulation does. Larger circuits have larger netlists and require more excitation data than smaller circuits do. Thus, simulation scales approximately with the square of the circuits size. Formal verification roughly scales linearly. Continuous verification and evaluation should detect flaws of the design as early as possible. But what will happen when no design alternative yields the desired results? For instance, this can happen when all variants of a HDL model describing the register-transfer level fail to implement a hardware architecture such that all constraints are met (e.g. the maximum clock frequency is always too low). In this case, design re-iterations on higher abstraction levels become necessary: One has to search for hardware architectures that have the desired functionality and meet the timing constraints on the target technology. Presumably, exploiting concurrency on the architectural level will solve the problem. In general, design re-iterations on higher abstraction levels might be necessary when no design alternative yields the desired properties. Design re-iterations are laborious and thus early and extensive evaluation of higher abstraction levels will help to circumvent them. Emphasized evaluation will help to approximate the design of a digital circuit to the ideal of the waterfall model where each abstraction level is transformed only once into the next lower layer.

3.4 Summary

This chapter introduced concepts about reconfigurable computing, the paradigm used for implementing a reconfigurable system that allows interoperability for elliptic curve cryptography. It presented the structure of reconfigurable devices and the design flow for developing the ECC reconfigurable system. The reconfigurable system proposed in this thesis is developed in three stages. The first one is concerned with the development and evaluation of an elliptic curve co-processor for executing the most time consuming operation in ECC, the scalar multiplication.

In the second stage deals with the design and implementation of a reconfigurable system that uses the ECC co-processor of stage one as a reconfigurable module that adapts at run time the parameters such as the irreducible polynomial $F(x)$, the order of the finite field GF(2^m) and the elliptic curve E, for supporting a desired security level. The third stage involves the validation of the system by applying testbeds in the simulation phase and performing in-circuit verification.

The next chapter describes the development of the first stage of the flow proposed that consists on the design and implementation of a GF(2^m) ECC co-processor. The design is carried out taking into account the design principles and the flow described in this chapter.

Chapter 4

ECC co-processor design

This chapter presents the design and implementation of an ECC co-processor. The design is based on the three layers of the operation dP presented in chapter 2. The hardware architecture for each one of these layers is discussed and the integration of them in a single hardware architecture for dP computation is presented.

This chapter also describes the design of a reconfigurable system based on a HW/SW approach that allows interoperability for elliptic curve cryptography. The main modules in the system are a microprocessor that commands the ECC co-processor for the computation of scalar multiplication dP supporting several tuples T for binary fields GF(2^m). The design of the reconfigurable system is implemented using the flow described in chapter 3.

4.1 Requirements

A software or hardware implementation of the scalar multiplication implies the choice of the algorithms to perform finite field arithmetic, to select the coordinate system to represent the elliptic curve points and to select the algorithm to compute dP. It is worth to mention that the algorithm choices in each stage for computing dP does not affect the interoperability of the ECC implementation.

As it was reviewed in the previous chapters, there are several choices for ECC implementation and there are several combinations of algorithms at each stage of dP in figure 2.2.4 that have already been explored. Most of the works reported in the literature argue that the Montgomery method [68] is the best choice for computing dP while López-Dahab coordinates [32] are the best way to represent

the elliptic curve points. Arithmetic in GF(2^m) has been considered the best choice for hardware implementation of ECC. This is because the arithmetic with polynomials in GF(2^m) is like binary arithmetic well suited to be implemented in hardware.

The Montgomery method for dP provides more physical security being more resistant to side channel attacks [44]. However, the Montgomery method requires more ECC-Add operations than the binary method (aprox. $\lceil \log d \rceil / 2$) and if projective coordinates are used, more finite field operations are required, as it was shown in table 2.2. Traditionally, projective coordinates have been used for faster dP implementations instead of affine. This argument is based on the fact inversions used in affine representation are very time consuming operations. In projective representation these field inversions are eliminated from the elliptic curve point addition but more field multiplications are introduced. There is a benefit in terms of time when using projective instead of affine coordinates only if field inversion has a computational cost of six or more filed multiplications. The known methods for computing field inversions are based on the Extended Euclidean Method and require around $2m$ clock cycles for computing a single field inversion. Finite field multiplication in binary field has a cost of m clock cycles if a serial multiplier is used or $\lceil m/d \rceil$ cycles is a digit-serial multiplier is used. Digit-serial multipliers implementations have increased area consumption if compared to serial implementations. If implementations with small area requirements are pursued and serial multipliers are selected, the use of projective coordinates is not recommended because the latency for field operation will increase about twice respect to the original affine representation. In addition, affine coordinates are better preferred because they require less operations and also less intermediate registers during the computations, see section 2.2.4.

The hardware for ECC can be implemented following a microprocessor or a co-processor approach. The data flow in the processor approach implies more clock cycles because the data processing must follows the Fetch-Decode-Execute-Store cycle. In addition, a bank register is necessary to store temporary data. In the co-processor approach the data flow requires less registers for intermediate results and the arithmetic modules can be connected directly, that is, the output of a module is the direct input of another one. However this last approach implies the use of buses to connect every module because the hardware implementation is very similar to the data flow of the algorithm being implemented. On one hand the

processor approach involves more clock cycles and imposes a more complicated control unit but it is simpler in the hardware organization of the main modules, like the register bank and arithmetic units. On the other hand, the co-processor approach requires less clock cycles because data are used immediately as they are available from previous modules, reducing the number of registers for storage but increasing the number of buses in the system. In order to achieve a compact design and to take advantage of the availability of data in the computations, the implementation of ECC point addition in this work is based on the co-processor approach.

Based on the above statements, this chapter discusses the design and implementation of an ECC co-processor for elliptic curves defined over GF(2^m) that provides interoperability for different tuples $T_{GF(2^m)}$. The parameters from $T_{GF(2^m)}$ that are intrinsically related to the implementation are: 1) the elliptic curve $E(GF(2^m))$, expressed by the coefficients a and b in equation 2.9; 2) the irreducible polynomial $F(x)$ that generates GF(2^m); and 3) the size of GF(2^m), that is, m. The ECC co-processor discussed in this thesis uses affine representation for the elliptic curve points and implements the binary method for dP.

According to the methodology presented in chapter 3, the first stage of this thesis is the development of hardware architectures for ECC. The best one will be selected and integrated in a reconfigurable system for providing interoperability in elliptic curve cryptography. The co-processor for the elliptic curve scalar multiplication dP discussed in this thesis is organized in three main components: arithmetic unit, control unit and memory.

4.2 Hardware for the lower dP layer: GF(2^m) arithmetic

The ECC co-processor is built on dedicated GF(2^m) arithmetic units so its performance and area requirements are a direct function of those arithmetic units. Multiplication, division, squaring and addition are finite field operations required for implementing the group operation in the elliptic curve. The next sections describe the hardware architectures for the field operations in GF(2^m).

4.2.1 GF(2^m) Multiplication

Multiplication in GF(2^m) in polynomial basis is the operation $A(x) * B(x)$ mod $F(x)$, that can be computed using a variety of proposed algorithms in the literature. Serial or bit-serial algorithms consider each individual bit of the operand $B(x)$ which implies a latency for multiplication of m clock cycles. The serial implementation can be improved gradually if instead of considering just one bit of the polynomial $B(x)$, a group of bits is considered at each step of the field multiplication algorithm. The number of bits considered is called the digit D. Digit-serial multipliers consider a group of D bits of operand $B(x)$ at time and perform the multiplication in $s = \lceil m/D \rceil$ cycles, but introduces complexity in each step of the multiplication. Varying the size of the digit allows to explore the cost in area and performance improvements from a serial implementation up to a parallel multiplication architecture. At each iteration, the operand $A(x)$ is multiplied by a group s_i of D bits of operand $B(x)$ and the result is reduced modulo $F(x)$. The result is added accumulatively to the result of the next iteration, considering the following D bits of $B(x)$ until all $B(x)$' bits are processed. The reduction in the operation latency comes with an increment in the complexity at each step of the multiplication.

Serial GF(2^m) multiplication

The serial algorithm for GF(2^m) multiplication [31] is shown in algorithm 4.1.

Algorithm 4.1 Serial multiplication in GF(2^m)

Input: $A(x)$, $B(x) \in$ GF(2^m), $F(x)$ the irreducible polynomial of degree m
Output: $C(x) = A(x) * B(x)$ mod $F(x)$

1: $C(x) \leftarrow 0$
2: **for** i from $m-1$ down to 0 **do**
3: $\quad C(x) \leftarrow C(x)x + A(x)b_i$
4: $\quad C(x) \leftarrow C(x) + c_m F(x)$
5: **end for**
6: return $C(x)$

Hardware implementation of a serial multiplier requires less area resources but performs the operation slowly compared to the digit-serial multiplier, which allows to explore area/performance trade-offs.

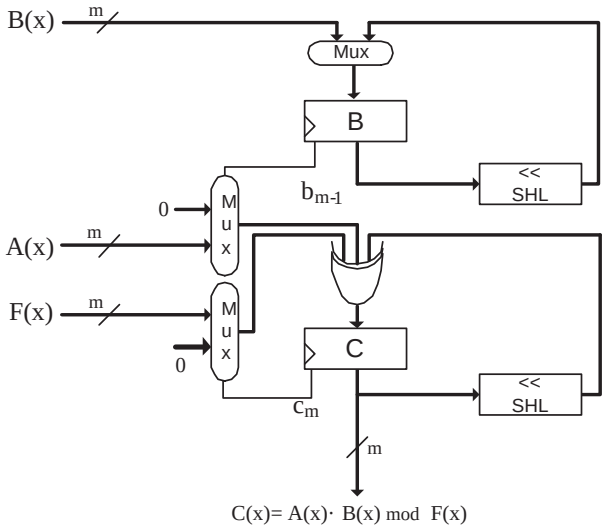


Figure 4.1: Circuit `GF2m_Mul_Serial_1` for GF(2^m) serial multiplication

In this thesis, several designs for the serial GF(2^m) multiplication algorithm were proposed. Each one of these versions is parametrized for different finite fields GF(2^m). Different versions for the multiplier were designed in order to have that implementation that occupied the fewer area resources while keeping a high performance.

The first version of the GF(2^m) serial multiplier is called `GF2m_Mul_Serial_1` and it is based on the diagram of figure 4.1. This architecture scans each bit of operand $B(x)$ and performs the operation in exactly m cycles. The HDL description of this architecture allows to parameterize it for different finite fields by specifying the size m and the irreducible polynomial $P(x)$.

The second version of the GF(2^m) multiplier is based on the serial multiplier proposed by Kumar and Paar [69]. The diagram of such multiplier, called `GF2m_Mul_Serial_2` is shown in figure 4.2. Kumar and Paar implemented a customized GF(2^{163}) serial multiplier. In this work, the original architecture was extended to the the finite fields orders of $m = 113, 131, 233, 287, 409$ and 571, all of them recommended in the standards NIST and SECG.

The circuit `GF2m_Mul_Serial_1` in figure 4.1 was implemented in two versions. The first one uses a finite state machine to implement the control logic. In the second one, the finite state machine is removed and the control is implemented only with combinatorial logic.

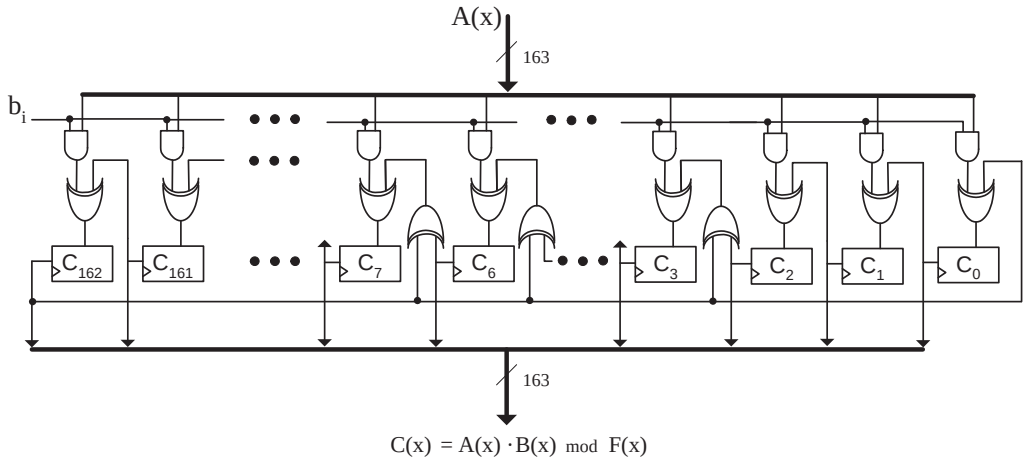


Figure 4.2: Circuit `GF2m_Mul_Serial_2` for GF(2^{163}) serial multiplication

The circuit `GF2m_Mul_Serial_2` was implemented in three different versions. In the first one, the circuit is implemented using a basic cell that models the combinatorial and sequential logic associated to each register in the circuit. For this case, the control logic was implemented with combinatorial logic. In the second version, the control is implemented using a finite state machine instead of combinatorial logic. This modification was done in order to obtain a higher clock frequency and hence a faster circuit. In the third version, the finite state machine was not used and the circuit was HDL designed as a single entity, that is, the basic cell was not used. This was done to compare the synthesis results of the structured version against the non-structured one.

Digit-Serial GF(2^m) multiplication

The digit-serial algorithm for GF(2^m) multiplication is shown in algorithm 4.2.

Algorithm 4.2 Digit-Serial multiplication in GF(2^m)

Input: $A(x)$, $B(x) \in$ GF(2^m), $F(x)$ the irreducible polynomial of degree m
Output: $C(x) = A(x) * B(x) \bmod F(x)$

1: $C(x) \leftarrow B_{s-1}(x)A(x) \bmod F(x)$
2: **for** k from $s-2$ downto 0 **do**
3: $\quad C(x) \leftarrow x^d C(x)$
4: $\quad C(x) \leftarrow B_k(x) * A(x) + C(x) \bmod F(x)$
5: **end for**

In this digit-serial algorithm, being $B(x)$ an element in GF(2^m) using polyno-

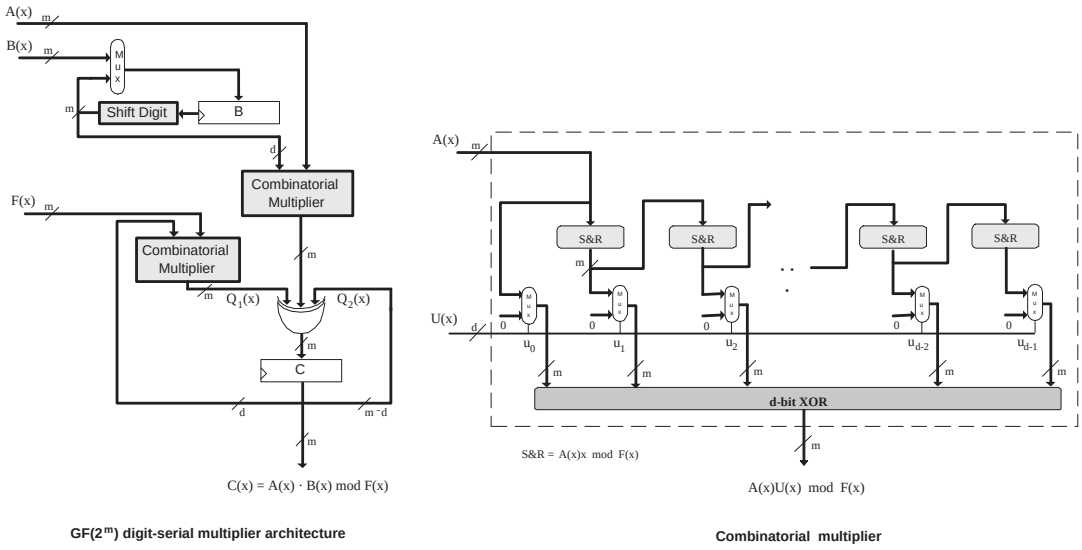


Figure 4.3: Hardware architecture `GF2m_Dserial_Mul_1` for GF(2^m) digit-serial multiplication.

mial basis, this is viewed as the polynomial

$$b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1 x + b_0 \,.$$

For a positive digit number $D < m$, the polynomial $B(x)$ can be grouped so that it can be expressed as

$$B(x) = x^D(s-1)B_{s-1}(x) + x^D(s-2)B_{s-2}(x) + \cdots + x^D B_1(x) + B_0(x)$$

where $s = \lceil m/D \rceil$ and each word $B_i(x)$ is defined as

$$B_i(x) = \begin{cases} \sum_{j=0}^{D-1} b_{iD+j}x^j & 0 \le x \le s-2 \\ \sum_{j=0}^{(m \bmod D)-1} b_{iD+j}x^j & i = s-1 \end{cases}$$

The hardware architecture for the GF(2^m) digit serial multiplier is shown in figure 4.3. This is called `GF2m_Dserial_Mul_1`.

If x^D is factored from the grouped representation of $B(x)$, the resulting expression is $B(x) = x^D(x^D(\cdots(x^D(x^D B_{s-1}(x) + B_{s-2}(x)) + \cdots) + B_1) + B_0$. This last representation of operand $B(x)$ is used in algorithm 1 to compute the field multiplication. That is, $A(x)*B(x) \bmod F(x) = x^D(x^D(\cdots(x^D(x^D B_{s-1}(x)A(x)+$

$B_{s-2}(x)A(x)) + \cdots) + B_1A(x)) + B_0A(x) \bmod F(x)$. At each iteration, the accumulator $C(x)$ is multiplied by x^D and the result is added to the multiplication of $A(x)$ by each word $B_i(x)$ of $B(x)$. The partial result $C(x)$ is reduced modulo $F(x)$.

The execution of the digit-serial algorithm is exemplified as

$$
\begin{aligned}
&C(x) = B_{s-1}(x)A(x) \bmod F(x) && \textit{Initialization}\\
&C(x) = x^D B_{s-1}(x)A(x) \bmod F(x) && \textit{Iteration}1\\
&C(x) = x^D B_{s-1}(x)A(x) + B_{s-2}(x)A(x) \bmod F(x) &&\\
&C(x) = x^D(x^D B_{s-1}(x)A(x) + B_{s-2}(x)A(x)) \bmod F(x) && \textit{Iteration}2\\
&C(x) = x^D(x^D B_{s-1}(x)A(x) + B_{s-2}(x)A(x)) + B_{s-3}(x)A(x) \bmod F(x) &&\\
&\cdots\cdots &&
\end{aligned}
$$

The hardware for the digit-serial algorithm in figure 4.3 is controlled by a finite state machine. In each iteration, a new digit of D bits from $B(x)$ is processed so the operation is performed in D/m cycles. The operations $x^D C(x)$ and $B_i(x)A(x)$ are computed using parallel combinatorial multipliers, that multiplies a $D-1$ grade polynomial with a $m-1$ grade polynomial. Being $U(x)$ a $D-1$ grade polynomial $u_{D-1}x^{D-1} + u_{D-2}x^{D-2} + \cdots + u_1x + u_0$, and $A(x)$ a $m-1$ grade polynomial, the parallel multiplication is

$$
\begin{aligned}
U(x)A(x) \bmod F(x) = {} & u_{D-1}x^{D-1}A(x) \bmod F(x)\\
& + u_{D-2}x^{D-2}A(x) \bmod F(x)\\
& + \cdots\\
& + u_1 xA(x) \bmod F(x)\\
& + u_0 A(x) \bmod F(x)
\end{aligned}
\tag{4.1}
$$

The operation $xA(x) \bmod F(x)$ is a shift to the left operation of $A(x)$ together with a reduction of $F(x)$. Thus, the value $x_iA(x) \bmod F(x)$ is the shifted and reduced version of $x_{i-1}A(x) \bmod F(x)$. So each value $x_iA(x) \bmod F(x)$ can be generated sequentially starting with $x_0A(x)$. Finally, each $x_iA(x) \bmod F(x)$ value is added depending on the bit value of u_i. These operations are executed by the parallel multiplier shown in the right side of figure 4.3. The operation $x^D C(x) \bmod F(x)$ is computed in two steps. Using the polynomial representation

of $C(x)$,

$$\begin{aligned} x^D C(x) \bmod F(x) &= x^D(c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + \cdots + c_{m-D}x^{m-D} \\ &\quad + c_{m-D-1}x^{m-D-1} + \cdots + c_1 x + c_0) \bmod F(x) \\ &= x^D(c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + \cdots + c_{m-D}x^{m-D}) \bmod F(x) \\ &\quad + x^D(c_{m-D-1}x^{m-D-1} + \cdots + c_1 x + c_0) \bmod F(x) \\ &= (c_{m-1}x^{m+D-1} + c_{m-2}x^{m+D-2} + \cdots + c_{m-D}x^m) \bmod F(x) \\ &\quad + (c_{m-D-1}x^{m-1} + \cdots + c_1 x^{D+1} + c_0 x^D) \bmod F(x) \\ &= Q_1(x) \bmod F(x) + Q_2(x) \bmod F(x) \end{aligned} \tag{4.2}$$

$Q_2(x)$ is a $m-1$ grade polynomial, corresponding to the $m-D$ least significant bits of $C(x)$ shifted D positions to the left. $Q_2(x)$ does not need to be reduced.

By factoring x^m from $Q_1(x)$, it is obtained $Q_1(x) = x^m(c_{m-1}x^{D-1}+c_{m-2}x^{D-2}+\cdots+c_{m-D})$. In this case, being $F(x)$ a $m+1$ trinomial or pentanomial of the form $F(x) = x^m+g(x)$, where $g(x)$ is a polynomial with grade $g << m$, the equivalence $x^m = g(x)$ can be used. In this case, $g(x)$ correspond to all bits of $F(x)$ except the m-bit. Thus, $Q_1(x) \bmod F(x) = g(x)(c_{m-1}x^{D-1} + c_{m-2}x^{D-2} + \cdots + c_{m-D})$. That is, the operation is a multiplication of $g(x)$ of grade g, and a polynomial of grade D, corresponding to the most significant D bits of $C(x)$. The resulting polynomial is of grade $g + D$. It must be verified that for any $F(x)$ and digit D, $g + D << m$. The polynomial $g(x)$ is expanded to a $m - 1$ grade polynomial so $Q_1(x) \bmod F(x)$ is computed using the parallel combinatorial multiplier. All these computations are performed by the modules in the architecture for the multipliers, which includes the parallel multipliers, a shift to the left module of d-bits, two registers and a $3m$-input XOR gate.

4.2.2 GF(2^m) Squaring

GF(2^m) squaring is the operation $A^2(x)$ for any $A(x) \in GF(2^m)$. This can be computed using the finite field multiplier or by a special circuit implemented using pure combinatorial logic. In this last approach the squaring operation can be performed in just one clock cycle taking advantage of some GF(2^m) arithmetic properties.

The square of an element a represented by $A(x) = a_{m-1}x^{m-1}+\cdots+a_2x^2+a_1x+a_0$ involves a polynomial multiplication of $A(x)$ with itself and then the reduction modulo $F(x)$. $A(x)A(x) \bmod F(x) = A^2(x) \bmod F(x)$ is given in equation 4.3.

$$A^2(x) = a_{m-1}x^{2m-2} + \cdots + a_2x^4 + a_1x^2 + a_0 \tag{4.3}$$

By factoring x^{m+1}, equation 4.3 can be written as $A^2(x) = (A_h(x)x^{m+1} + A_l(x)) \bmod F(x)$, where
$A_h(x) = a_{m-1}x^{m-3} + \cdots + a_{(m+3)/2)}x^2 + a_{(m+1)/2}$
$A_l(x) = a_{(m-1)/2}x^{m-1} + \cdots + a_1x^2 + a_0$

The degree of $A_l(x)$ is lower than m and reduction is not necessary. The product $A_h(x)x^{m+1}$ may have degree as high as $2m-2$. The irreducible polynomial $F(x)$ usually is a trinomial or pentanomial of the form
$F(x) = x^m + x^d + \cdots + 1$

By multiplying both sides of the field equivalence $x^m = x^d + \cdots + 1$ by x, it is deduced that $x^{m+1} = x^{d+1} + \cdots + x$. So

$$A_h(x)x^{m+1} = A_h(x)(x^{d+1} + \cdots + x)$$

This operation is performed using the architecture for combinatorial multiplication in the digit-serial multiplier. Here, the size of the digit is $d + 2$. The diagram of the squarer is depicted in figure 4.4.

The operation $A^2(x)$ results is a polynomial of degree as high as $2m - 2$. It is a polynomial with interleaved insertion of '0'. Once $A^2(x)$ is computed, a combinatorial multiplier computes $A_h(x)x^{m+1} \bmod F(x)$. The final result is obtained by adding the polynomial $A_l(x)$ to the result of the polynomial multiplication.

Kumar and Paar proposed another implementation of the combinatorial squarer [69]. The implementation is specialized for the finite field GF(2^{163}) and a particular irreducible polynomial. This idea can be extended to other finite fields and irreducible polynomials.

4.2.3 GF(2^m) Inversion

Typically, division in GF(2^m) u/v is implemented as two consecutive operations, the inversion v^{-1} and then the multiplication uv^{-1}. There are well known algo-

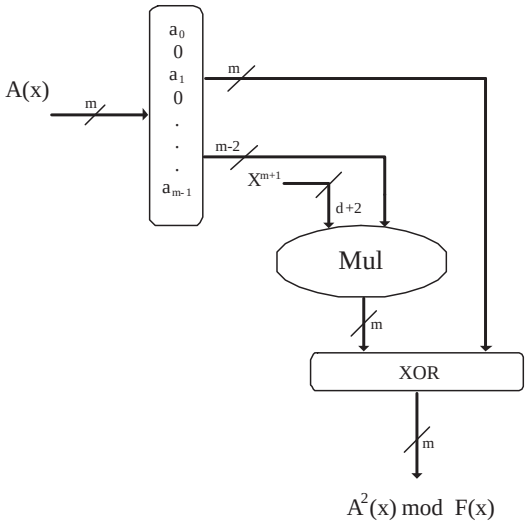


Figure 4.4: Circuit for GF(2^m) squaring

rithms for field inversion, like The Modified Almost Inversion Algorithm MAIA, the Fermat's theorem or the Ito-Tsujii algorithm [31]. The latency of MAIA is $2m$ clock cycles so the whole latency of field division would be at worst $3m$ using a serial multiplication algorithm. The Fermant theorem requires $m-1$ field multiplications and m squarings. Using this theorem, a division will result very expensive even though the squaring was implemented in just one clock cycle. The Ito-Tsujii algorithm requires only $\log m$ iterations to complete an inversion but at the cost of more complex control and more squarings.

The algorithm proposed by S. C. Shantz [70] shown in algorithm 4.3 can perform a direct division $u/v \bmod p$ in at most $2m-2$ clock cycles. That is, this algorithm requires almost the same time to compute a single inversion but saves the additional time for the field multiplication in the operation uv^{-1}.

Algorithm was selected in this thesis for field division and it was implemented in the next different ways:

1. *Version 1.* This architecture is called `GF2m_Div_1` and shown in figure 4.5. In this implementation four cells were modeled for each register in the diagram together with the associated combinatorial logic. This was done because according to the literature [71], such model fits better in the basic cells of the FPGA and the resources are better used. A cell in the FPGA can implement any combinatorial function of four inputs and store the result

Algorithm 4.3 Division algorithm: Division in F_{2^m}

Input: $X_1(x), Y_1(x) \in F_{2^m}$, $X_1(x) \neq 0$ and $F(x)$ the irreducible polynomial of degree m

Output: $U(x) = Y_1(x)/X_1(x) \bmod P(x)$

1: $A(x) \leftarrow X_1(x)$
2: $B(x) \leftarrow F(x)$
3: $U(x) \leftarrow Y_1(x)$
4: $V(x) \leftarrow 0$
5: **while** $A(x) \neq B(x)$ do **do**
6: **if** x divides to $A(x)$ **then**
7: $A(x) \leftarrow A(x)x^{-1}$
8: $U(x) \leftarrow U(x)x^{-1} \bmod F(x)$
9: **else if** x divides to $B(x)$ **then**
10: $B(x) \leftarrow B(x)x^{-1}$
11: $V(x) \leftarrow V(x)x^{-1} \bmod F(x)$
12: **else if** grade of $A(x)$ is greater than grade of $B(x)$ **then**
13: $A(x) \leftarrow (A(x) + B(x))x^{-1}$
14: $U(x) \leftarrow (U(x) + V(x))x^{-1} \bmod F(x)$
15: **else**
16: $B(x) \leftarrow (A(x) + B(x))x^{-1}$
17: $V(x) \leftarrow (U(x) + V(x))x^{-1} \bmod F(x)$
18: **end if**
19: **end while**

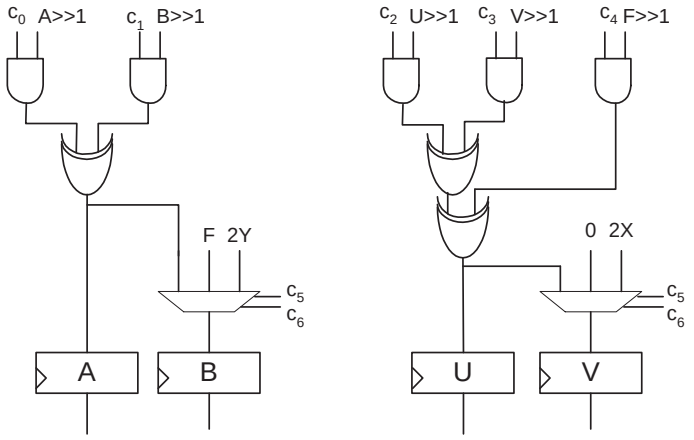


Figure 4.5: Architecture `GF2m_Div_1` for GF(2^m) division

(binary) in a flip-flop.

2. *Version 2*. This architecture is called `GF2m_Div_2`, it is based on the diagram of figure 4.6. In this case the cells for the register A, B, U, V were not modeled. Instead, the circuit was HDL-described as just one entity. In this version and also in version 1, the implementation requires of a m-bit magnitude comparer, which is supposed to be expensive in terms of area and increases the latency of the operation.

3. *Version 3* and *Version 4* of the GF(2^m) divisor are the same that versions 1 and 2 respectively but using a 8-bit comparer instead of the m-bit one. This is based on the modification proposed by Gura [17] to the original Shantz's algorithm.

4.3 Hardware for the middle dP layer: Coordinate system

The data flow of the group law for point addition presented in section 2.2.4 is shown in figure 4.7.

In the co-processor approach, parallel operations are identified from the data flow of ECC point addition to arrange the GF(2^m) arithmetic modules in a such way that the ECC point addition be performed efficiently. Dedicated units for

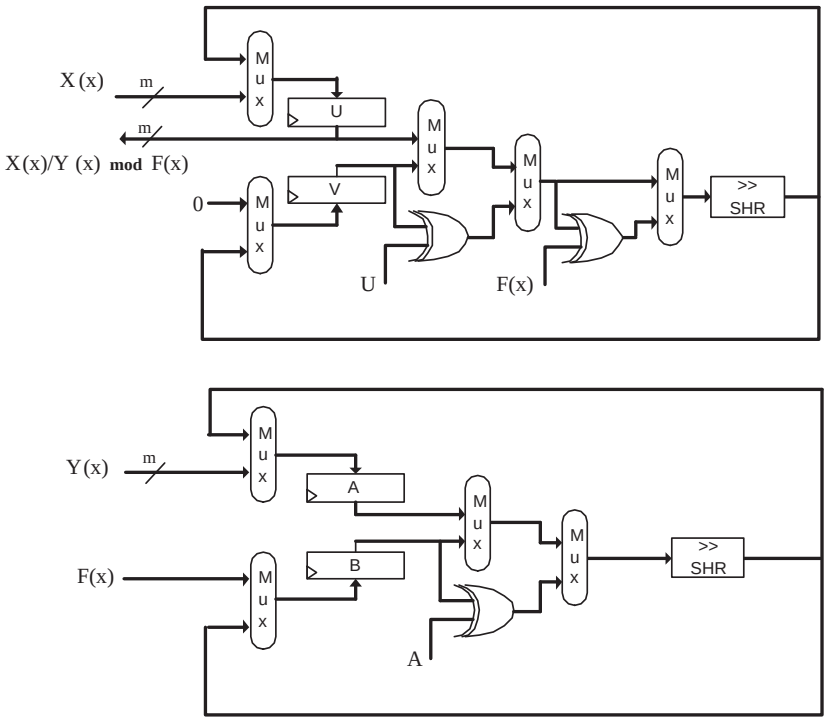


Figure 4.6: Architecture `GF2m_Div_2` for GF(2^m) division

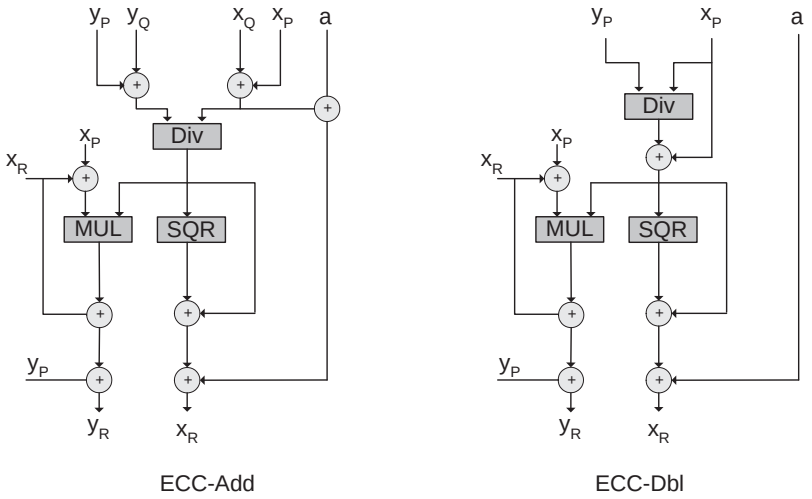


Figure 4.7: Data flow for ECC point addition

ECC-Add and ECC-Dbl operations could be implemented for parallel execution of the operation dP. Regardless of the parallel execution would perform the dP operation faster, this hardware implementation would be vulnerable to side channel attacks. Side channel attacks and their countermeasures were reviewed in the last section of chapter 2.

The traditional method for computing dP is the binary method. It parses every bit value of scalar d and executes at each iteration one ECC-Dbl operation followed by one ECC-Add only if the current bit value of d is '1'. The direct hardware implementation of this dP method is vulnerable to side channel attacks, such as the simple power analysis attack (SPA). In SPA, the attacker measures the power produced by the hardware executing the operation dP and tries to reveal the private key from those traces. An SPA attack for the hardware implementation of the binary method for dP is possible because ECC-Add and ECC-Dbl are different in essence and they will produce different power traces. Due the operations ECC-Add and ECC-Dbl are strongly related to the d's bits, the security of the system could be compromised.

One approach for preventing SPA attacks is to rewrite the addition formulas ECC-Add and ECC-Dbl so that a single formula can be used for both kinds of point sums, indifferently [45]. This approach has been considered in the literature but using projective coordinates [72] or special forms of the elliptic curve [73]. The next section explains a new unified formula for performing both ECC-Add and ECC-Dbl operations. The implementation of this new formulation increases the resistance of a hardware implementation of dP to side channel attacks.

4.3.1 A new affine formula for point addition

Addition and doubling operations are very similar in affine representation. Given the points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, the rules to perform ECC-ADD$(P, Q) = (x_{ADD}, y_{ADD})$ and ECC-DBL$(P) = (x_{DBL}, y_{DBL})$ are:

$$\lambda_1 = \frac{y_Q + y_P}{x_Q + x_P}$$
$$x_{ADD} = \lambda_1^2 + \lambda_1 + x_Q + x_P + a$$
$$y_{ADD} = \lambda_1(x_P + x_{ADD}) + x_{ADD} + y_P$$

$$\lambda_2 = x_P + \frac{y_P}{x_P}$$
$$x_{DBL} = \lambda_2^2 + \lambda_2 + a$$
$$y_{DBL} = x_P^2 + \lambda_2 x_{DBL} + x_{DBL}$$

The last equation can be rewritten in the form

$$y_{DBL} = \lambda_2(x_P + x_{DBL}) + x_{DBL} + y_P.$$

Both ECC-Add and ECC-Dbl operations require to perform one division, one multiplication and one squaring. The ECC-Add operation requires to perform nine additions and the ECC-Dbl requires six. Although both kinds of elliptic curve point addition use almost the same number of operations, the way in which each one is defined is in essence different. This implies a dedicated implementation for each one of these operations. When implemented in hardware, these different modules have different power traces that could be used in side channel attacks.

To unify the operations ECC-Add and ECC-Dbl in affine coordinates aims: *i*) to reduce the hardware used for implementing the addition operation in elliptic curves, used for performing scalar multiplications, and *ii*) to increase the resistance of the dP hardware implementation to side channel attacks. The main idea for the new formula is to unify the ECC-Add and the ECC-Dbl operations by multiplexing data according to the operation being performed. Such multiplexing is implemented by introducing the operation $s_0 \cdot x$, which is the bitwise AND operation of bit s_0 with each bit-value of x.

By introducing the $s_0 \cdot x$ operation in the original formulas for point addition and applying boolean reductions, the new formulas to perform an ECC-Add operation if $s_0 =$ '1' or an ECC-Dbl operation if $s_0 =$ '0', are:

$$\lambda = \frac{s_0 \cdot y_Q + y_P}{s_0 \cdot x_Q + x_P}$$
$$X = (\lambda + s_0 \cdot x_P)^2 + \lambda + s_0 \cdot x_Q + x_P + a$$
$$Y = (\lambda + s_0 \cdot x_P)(x_P + X) + X + y_P$$

The new formula for unified ECC point addition operation requires the following field operations: 10 additions $\mathcal{A}$, 1 multiplication $\mathcal{M}$, 1 division $\mathcal{D}$ and 1 squaring $\mathcal{S}$. That is, the new formula requires one more addition in the case of the

ECC-Add operation and four additions in the case of the ECC-Dbl (see table 2.2 in chapter 2). Field additions in GF(2^m) are trivial operations implemented as XOR operations so this difference has not a serious impact in the timing to compute any of the two elliptic curve point additions. Instead of having two distinct hardware modules for each ECC elliptic curve point addition operation, a single hardware module is provided thus resulting in smaller area requirements.

Figure 4.8 shows the data flow for the point addition module. Since field addition is an XOR operation and squaring can be implemented using combinatorial logic, the whole latency for point addition is the latency of a field division plus the one of a field multiplication. In figure 4.8, the combinatorial operations like AND and XOR are represented as black boxes of two or three inputs (Lut2, Lut3). The black boxes are well mapped to LUTs (Look Up Table), which are elements in FPGAs that implement any boolean function of up to 4-inputs.

4.4 Hardware for the higher dP layer: dP method

As it was reviewed in section 2.2.4, there are several methods to compute the operation dP. Implementing a method for dP is to implement a control unit that commands the necessary control signal of the hardware modules for point addition discussed in the previous section.

Algorithm 4.4 computes dP by executing ECC-Dbl and ECC-Add operations serially. By using the Point Addition module shown in figure 4.8, the algorithm 4.4 can be implemented adding the control unit to orchestrate the data flow as indicated in the algorithm itself. Figure 4.9 shows the block diagram of the resulting dP co-processor.

The control unit of the ECC co-processor is a finite state machine (FSM) that parses the scalar d and sends control signals to the GF(2^m) arithmetic modules and multiplexers. Both, the scalar d and point P are entered to the architecture in groups of 32-bit words. Additionally to scalar multiplication dP, the architecture can perform an ECC-Add operation, which is required in elliptic curve digital signature verification schemes.

The right to left version of algorithm 4.4 listed in algorithm 4.5, can perform ECC-Add and ECC-Dbl independently in each iteration. The implementation of this algorithm will result in a faster computation of dP but at the cost of an

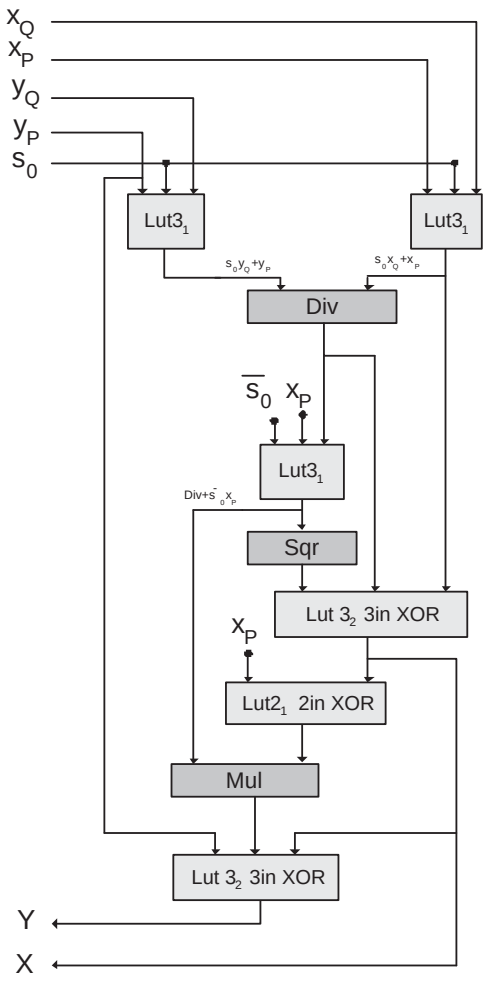


Figure 4.8: Diagram block for the new Point Addition formula

additional Point Addition module.

Even though the implementation of algorithms 4.4 and 4.5, using the Point Addition module, ensures that the power traces of ECC-Add and ECC-Dbl not to be distinguishable, the dP hardware implementation can be still vulnerable to timing attacks.

4.4.1 A co-processor resistant to side channel attacks

The algorithms 4.4 and 4.5 for dP were modified by Coron [46] to be resistant to simple power traces (SPA) and timing attacks (TA). These new algorithm for dP are shown in algorithms 4.6 and 4.7.

Algorithm 4.4 Binary method for scalar multiplication dP. Left to right version, Serial execution of ECC-Dbl and ECC-Add

Input: $P = (x, y)$ $x, y \in \mathrm{GF}(2^m)$, $d = [d_{m-1}, d_{m-2}, ..., d_0]_2$
Output: dP

1: $Q \leftarrow (0, 0)$
2: **for** i from $m - 1$ downto 0 **do**
3: $\quad Q \leftarrow \text{ECC-DBL}(Q)$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow \text{ECC-ADD}(P, Q)$
6: $\quad$ **end if**
7: **end for**
8: return Q

Algorithm 4.5 Binary method for scalar multiplication dP. Right to Left, parallel execution of ECC-Dbl and ECC-Add

Input: $P = (x, y)$ $x, y \in \mathrm{GF}(2^m)$, $d = [d_{m-1}, d_{m-2}, ..., d_0]_2$
Output: dP

1: $Q \leftarrow (0, 0)$
2: $R \leftarrow (0, 0)$
3: **for** i from 0 to $m - 1$ **do**
4: $\quad Q \leftarrow \text{ECC-DBL}(Q)$
5: $\quad$ **if** $k_i = 1$ **then**
6: $\quad\quad Q \leftarrow \text{ECC-ADD}(Q, R)$
7: $\quad$ **end if**
8: $\quad R \leftarrow \text{ECC-DBL}(R)$
9: **end for**
10: return R

Algorithm 4.6 Binary method for scalar multiplication dP. Left to right version, Serial execution of ECC-Dbl and ECC-Add

Input: $P = (x, y)$ $x, y \in \mathrm{GF}(2^m)$, $d = [1, d_{t-2}, ..., d_0]_2$
Output: dP

1: $Q_0 \leftarrow P$
2: **for** i from $t - 2$ downto 0 **do**
3: $\quad Q_0 \leftarrow \text{ECC-DBL}(Q_0)$
4: $\quad Q_1 \leftarrow \text{ECC-ADD}(Q_0, P)$
5: $\quad Q_0 \leftarrow Q_{d_i}$
6: **end for**
7: return Q_0

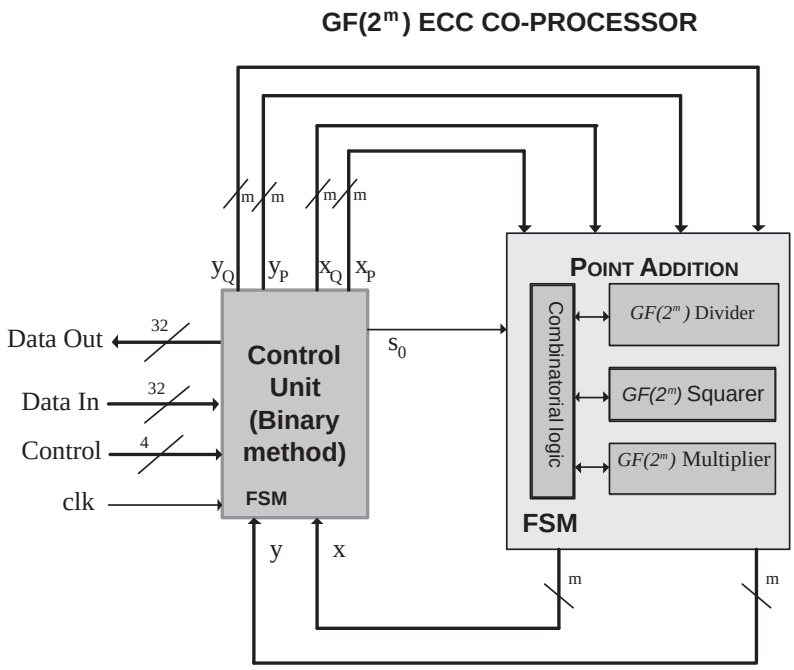


Figure 4.9: Elliptic curve co-processor for dP.

Again, the serial implementation implies less area but higher latency while the parallel implementation implies the opposite, more area but faster computation. Figure 4.10 shows the proposed serial and parallel architectures for algorithms 4.6 and 4.7 respectively, using the implementation of the unified formulas for point doubling and addition showed in figure 4.8.

In the case of the serial architecture, the resulting point from the double and add operations are temporary stored in the register Q_0 and Q_1. After completing both operations, the result is back propagated using a simple multiplexer, whose selector is given by a parser module. At each iteration the control unit shifts the scalar d one bit to the left and emits a pulse from 0 to 1 for s_0. The point addition module starts and ends performing an ECC-ADD operation because the initial value $Q_0 = P$ must be loaded at the beginning using the available circuit. So the implementation of algorithm 4.6 performs (t·ECC-ADD + $(t-1)$ ECC-DBL) operations. Because each point addition is computed in at most $3m$ clock cycles, the whole latency for the dP operation is $(6t-3)m$ clock cycles.

In the case of the parallel architecture, each module for point addition can

Algorithm 4.7 Binary method for scalar multiplication dP. Right to Left, parallel execution of ECC-Dbl and ECC-Add

Input: $P = (x, y)$ $x, y \in \text{GF}(2^m)$, $d = [1, d_{t-2}, ..., t_0]_2$
Output: dP
1: $Q_0 \leftarrow P$
2: $Q_1 \leftarrow 2P$
3: **for** i from $d-2$ downto 0 **do**
4: $\quad Q_2 \leftarrow \text{ECC-DBL}(Q_{d_i})$
5: $\quad Q_1 \leftarrow \text{ECC-ADD}(Q_0, Q_1)$
6: $\quad Q_0 \leftarrow Q_{2-k_i}$
7: $\quad Q_1 \leftarrow Q_{1+k_i}$
8: **end for**
9: return Q_0

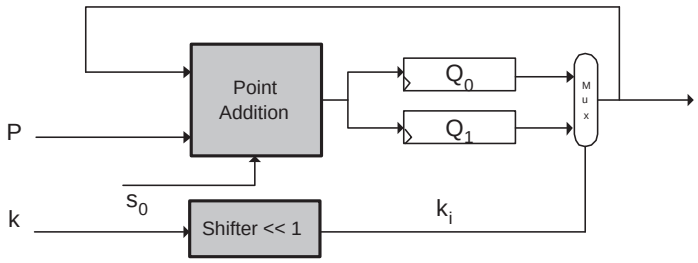
be optimized to perform each elliptic curve point sum. That is, s_0 keeps the same value during the dP operation. The resulting points from the double and add operations are interchanged and back propagated. The selector is given by a parser module. In each iteration the control unit shifts the scalar d one bit to the left. A multiplexer at the input of the doubling module is required to start the computation of dP. Initially, the input to the doubling module is P and the other one for the addition module is 0. So, the initial values are stored as P and $2P$ as required in the algorithm. This implementation of algorithm 4.7 performs (t·ECC-ADD + t ECC-DBL) operations. Because each point addition is computed in at most $3m$ clock cycles, the whole latency for the dP operation is $6tm$ clock cycles.

4.5 The ECC reconfigurable system

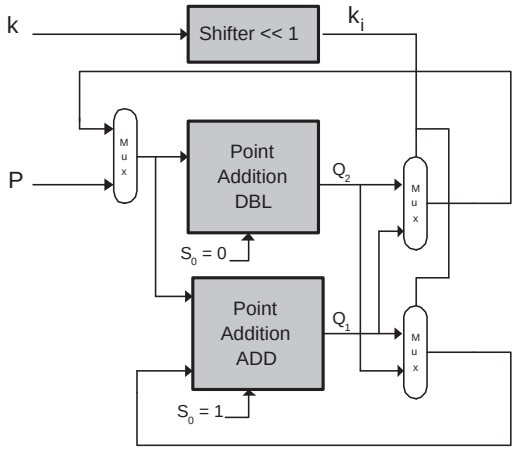
The dP co-processor discussed in the previous sections needs to be modified given an specific tuple T defined on GF(2^m) fields. These parameters are:

1. The **binary field** (given by the value of m). This parameter determines the width of data buses and the value of counters in the control units.

2. The **irreducible polynomial** $F(x)$ defining GF(2^m). This parameter affects the design and structure of each dedicated module for GF(2^m) arithmetic.



a) Algorithm 3 implementation



b) Algorithm 4 implementation

Figure 4.10: Serial a) and parallel b) implementation for the Coron's binary methods for dP.

3. The **elliptic curve** (see equation 2.9 in section 2.2) given by the constants a and b. These values affect the definition of the point addition module.

The approach taken in this thesis is to design generic architectures for each main module in the dP co-processor (arithmetic modules and control), and then to extend those modules for different tuples T. The dP co-processor supporting a specific tuple T is created by selecting the proper modules in each stage of the dP operation according to the tuple T being used. This is shown in figure 4.11.

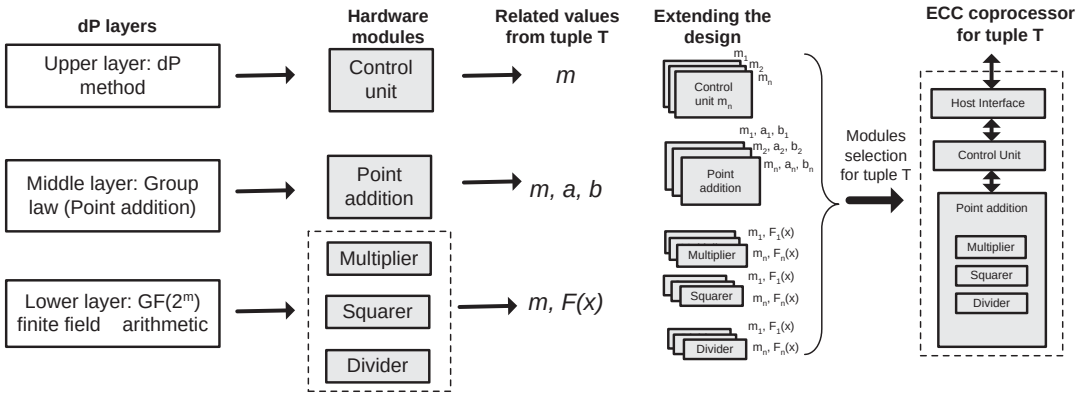


Figure 4.11: Extending the dP processor to support different tuples T

4.6 Proposed reconfigurable system

Figure 4.12 shows the reconfigurable system proposed for interoperable ECC. The target system performs ECC based cryptographic schemes using both software and hardware. The ECC co-processor is configured for specific tuples T given by the software application running on the embedded microprocessor. The reconfigurable system also includes local buses PLB and OPB, an universal asynchronous receiver/transmitter UART module, memory blocks for data and program. The original co-processor shown in figure 4.9 (see chapter 4) was wrapped with the IPIF EDK core [74] to interconnect it to the OPB bus.

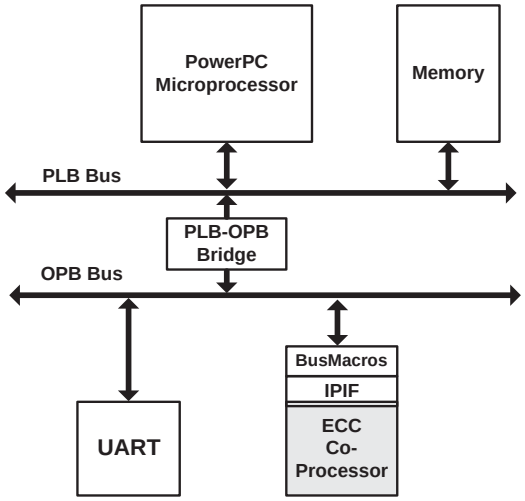


Figure 4.12: Co-processor attached to a microprocessor

The software application enables the ECC co-processor and sends the point

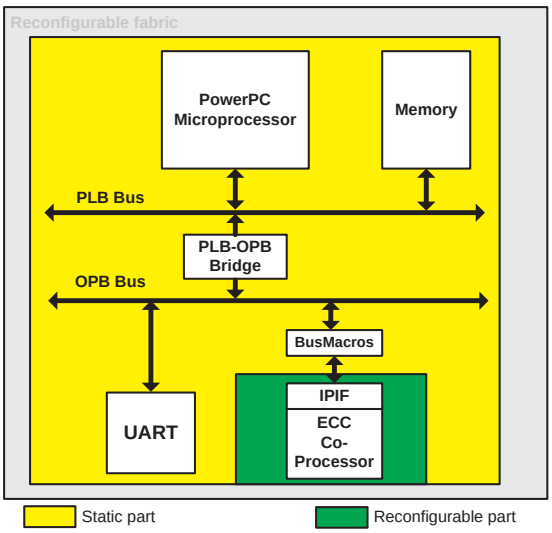


Figure 4.13: Layout of the proposed reconfigurable system

P and scalar d through the PLB bus as a group of 32-bit words. After reading the input parameters, the ECC co-processor starts the computation while the processor waits for the results. By asserting a signal, the co-processor notifies the end of the computation and then the application reads back the results and shows them via the UART module to the user. The system shown in figure 4.12 can be generated using EDK from Xilinx. It is a GUI interface that allows to generate a complete system and implement it in a physical device.

As described in chapter 3, the system shown in figure 4.12 is partitioned in both, static and reconfigurable parts. In this case, the system is composed of only one static and one reconfigurable part. The static part is composed of all the modules in figure 4.12 except the ECC co-processor wrapped by the IPIF module, which is the reconfigurable part. All signals that connect the fixed and reconfigurable part go through busmacros. The layout of the reconfigurable system is shown in figure 4.13

The reconfigurable and static parts, the busmacros, and other global resource like clock of buffers must be correctly placed in the reconfigurable fabric. Previous to the implementation phase, the placement of these components is specified in a constraint file, that is generated manually or using advanced tools that also check for inconsistences in the placement process. PlanAhead is a useful tool that allows to generate this constraint file and also implement the completed reconfigurable flow.

4.7 Summary

This chapter discussed the design of a GF(2^m) ECC co-processor using affine coordinates and the binary method for computing the operation dP. Different co-processors were designed and evaluated for studying the area/performance trade offs given several designs for the GF(2^m) arithmetic modules. The designed co-processor implements a new representation of the formulas for elliptic curve point addition that is more resistant to side channel attacks, such as timing attacks and simple power analysis.

This chapter also presented the design issues of the proposed reconfigurable system. It is based on a microprocessor that commands the execution of the dP operation by the ECC co-processor. The design of the reconfigurable system imposed several issues like the placement of the component of the system in the reconfigurable fabric.

The next chapter discusses the design and implementation of a reconfigurable system that uses ECC co-processor presented in this chapter. Partial dynamic reconfiguration is used for implementing the system that allows interoperability for elliptic curve cryptography.

Chapter 5

Results

This chapter presents the results of this research and is organized in three parts. The first one presents the results of the GF(2^m) arithmetic units, which are the basis of the dP co-processor developed. The second part describes the results of the dP co-processor, using different configurations depending on the area/performance requirements. The third part presents the results of the reconfigurable system for interoperable ECC. It shows the achievements in time and area of the proposed system that allows run-time adaptation of hardware for elliptic curve arithmetic depending on the security level required by the application.

5.1 Target technology for implementation

The results presented in this chapter are the ones obtained by implementing the hardware designs in FPGA technology. The targeted FPGA is the Virtex4 XC4VFX12 from Xilinx [75]. The Virtex-4 device logic unit is the *slice*. Each slice (see figure 5.1) consists of two fixed 4-input LUTs, embedded multiplexers, carry logic, and two registers.

Configurable Logic Blocks (CLBs) in Virtex4 FPGAs are made up of four slices. The function generators are configurable as 4-input look-up tables (LUTs). Two slices in a CLB can have their LUTs configured as 16-bit shift registers, or as 16-bit distributed RAM. In addition, the two storage elements are either edge-triggered D-type flip-flops or level sensitive latches. Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources. The Virtex4 XC4VFX12 FPGA has 5472 slices available. It also includes an

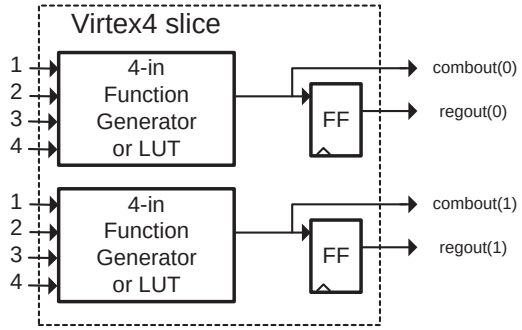


Figure 5.1: Virtex4 slice

embedded IBM PowerPC 405 RISC CPU with a working frequency up to 450 MHz.

5.2 Metrics of performance

The developed hardware architectures are evaluated and measured by using the *area* and *throughput* factors. These metrics provide a measure for comparing different designs. Their definitions are [76]:

1. **Area**: The space occupied by a hardware design is expressed in terms of the logic blocks of the FPGA. The additional resources in FPGAs like BRAMs, multipliers, etc., should be also mentioned if they are used in the design. The lesser the FPGA elements are used the most economical design in terms of the area occupied by it.

2. **Throughput**: Throughput measures timing performances of hardware designs. It is obtained by multiplying the allowed frequency for the design with the number of bits processed per cycle. For cryptographic algorithms, throughput (in $bits/sec$) is defined as:

$$Throughput = \frac{Allowed\ Frequency}{Number\ of\ Bits\ Processed}$$

   The higher the throughput of the design, the better its efficiency.

3. **Throughput/Area**: This is the ratio of the above two metrics and shows how efficient the design is with respect to both area and throughput. The ratio is high in case of high throughput and less space.

5.3 Tools

The results presented in this chapter were obtained using CAD tool from Xilinx. ISE 8.2 was used for synthesizing, mapping, and implementing all the HDL designed modules. The software tools for partial reconfiguration from Xilinx were used. EDK and PlanAhead 8.2 from Xilinx were used for implementing the reconfigurable system and for floorplanning.

5.4 Results of GF(2^m) arithmetic modules

5.4.1 Serial GF(2^m) multiplication

The synthesis results for each one of the five versions for the implementation of algorithm 4.1 are summarized in table 5.1. The main differences of the five versions for the serial multiplier, as they were mentioned in chapter 4, are:

`GF2m_Mul_Serial_1`

- Version 1: Functional description in VHDL.
  The control logic is implemented as a FSM (Finite State Machine).
- Version 2: Functional description in VHDL.
  The control logic is implemented as a combinatorial circuit.

`GF2m_Mul_Serial_2`

- Version 1: Modular description in VHDL.
  The control logic is implemented as a combinatorial circuit.
- Version 2: Modular description in VHDL.
  The control logic is implemented as a FSM (Finite State Machine).
- Version 3: Functional description in VHDL.
  The control logic is implemented as a combinatorial circuit.

All the circuits were implemented using two optimization criteria of the synthesis tool, the *Area* and *Speed* optimization. In the first one, the synthesis tool optimizes the area resources by reusing internal hardware of the circuit at the cost of slower designs. In the speed criteria, the synthesis tool uses area as much as required but tries to reduce the latency of the circuit and increases the clock frequency for getting a faster circuit. The results in table 5.1 show that for

Table 5.1: Synthesis results of the $GF(2^m)$ multiplication algorithm on the Virtex4 FPGA.

Hw Design		113	131	163	
GF2m_Mul_Serial_1	v1	132/183	151/208	186/257	Slices
	v2	130/129	149/148	184/183	Area/Speed
GF2m_Mul_Serial_2	v1	121/123	214/219	265/271	
	v2	187/236	216/272	268/337	
	v3	129/128	149/148	183/182	
GF2m_Mul_Serial_1	v1	211/314	242/310	230/308	Freq.
	v2	328/328	321/321	308/308	(MHz)
GF2m_Mul_Serial_2	v1	249/99	245/297	338/296	Area/Speed
	v2	247/312	242/310	235/308	
	v3	317/363	317/354	317/339	

area optimization, the versions one and two of circuit `GF2m_Mul_Serial_1` and the version three of circuit `GF2m_Mul_Serial_2` uses almost the same amount of area. However, the last one has a higher clock frequency. The fastest of the five multipliers is the version two of circuit `GF2m_Mul_Serial_1` but uses the double of the area used by the multipliers with smaller area. From table 5.1, designers can choose the multiplier that better meets the application requirements in terms of area or speed.

5.4.2 Digit-Serial GF(2^m) multiplication

The results for the digit-serial multiplier presented in chapter 4 are shown in table 5.2. The table shows the are and frequency of the circuit `GF2m_Dserial_Mul_1` for the digit sizes $D = 4, 8, 16, 32$ and the finite field order $m = 163, 233, 239, 409$ and 571. Figure 5.2 shows the timing in microseconds to compute one field multiplication using a specific combination of digit and field order [1].

5.4.3 GF(2^m) squarer

In both implementations the area results and latency were the same. The squarer occupied only 95 slices and 165 LUTs for the finite field $GF(2^{163})$.

[1] Results using a XCV2 FPGA.

Table 5.2: Synthesis results for the GF(2^m) digit serial multiplier on the Virtex4 FPGA.

Digit	m 163	233	277	283	409	571	
4	462/460	647/641	770/760	787/775	1126/1333	1571/1589	
8	749/808	1156/1016	1380/1193	1281/1201	1833/1779	2562/2418	Slices
16	1515/1488	2128/1946	2559/2332	2608/2370	3721/3635	5220/4731	Area/Speed [a]
32	2853/2627	3993/3744	4835/4431	4925/4530	7002/6399	9857/8936	
4	200/365	236/366	172/330	172/330	203/383	172/370	
8	197/308	138/315	114/285	180/277	173/284	188/276	Freq.(MHz)
16	145/246	132/252	131/245	130/248	119/264	127/244	Area/Speed
32	137/218	132/227	116/223	116/220	119/224	115/225	

[a]Optimization criteria in the synthesis process.

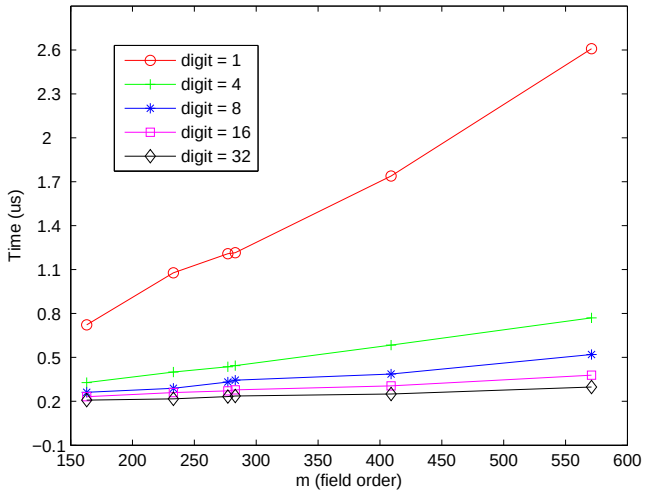


Figure 5.2: Timing *us* for $GF(2^m)$ digit serial multiplier

5.4.4 GF(2^m) division

Table 5.3 shows the area results for the different implementations of the $GF(2^m)$ division algorithm. The results are for the finite fields $m = 113, 131, 163$ optimizing for area or speed.

The second version of both architectures `GF2m_Div_1` and `GF2m_Div_2` for GF(2^m) division resulted faster than the first one due the use of a smaller comparator. Version one of the circuit `GF2m_Div_1` is the smallest when area optimization is applied while version two of the same circuit is the fastest using speed optimization.

Table 5.3: Synthesis results for the $GF(2^m)$ division algorithm on the Virtex4 FPGA.

Hw Design		113	131	163	
GF2m_Div_1	v1	410/611	475/709	589/879	Slices
	v2	627/600	723/693	893/852	Area/Speed
GF2m_Div_2	v1	501/737	581/851	723/1057	
	v2	459/722	529/828	655/1025	
GF2m_Div_1	v1	95/105	87/98	76/86	Freq.
	v2	113/154	99/152	94/146	(MHz)
GF2m_Div_2	v1	76/106	73/98	64/86	Area/Speed
	v2	113/164	101/164	99/160	

5.4.5 Discussion

The results of the different hardware designs of the GF(2^m) arithmetic modules allow to explore an area/performance trade off in order to select the most appropriate modules for building the ECC co-processor. If a compact design is pursed, then the mos appropriate hardware modules are the ones occupying the smallest area. Otherwise, the designs with the highest clock frequency should be selected if a faster architecture is required.

Even though a hardware design is good for a finite field order, it could not be good for another. So, designer should consider all the available designs for each arithmetic algorithm in order to select the most appropriate based on the application requirements.

It can be seen from table 5.3 how the GF(2^m) divider has a greater area consumption and also, lower operational frequencies. This module will affect the area/performance of the whole design of the dP co-processor so a carefully selection of this module should be done. In the case of the multiplier, some designs are cheaper in terms of area and still keep a high clock frequency. Although digit-serial multiplier achieves the field multiplication faster, the area consumption for digits greater that 16 grows quickly. The field multiplication is not the most time consuming field operation in the dP operation. Digit sizes greater than 16 could not offer a considerable advantage in the computation of scalar multiplication thus the extra cost in terms of area for digit sizes greater that 32 bits will not be justified.

5.5 Results of the GF(2^m) dP co-processor

Previous sections discussed the design of hardware for each layer in the dP operation. Different versions for each GF(2^m) arithmetic modules were implemented leading to different area/performance results. This section presents the synthesis results of the ECC co-processor discussed in chapter 4 using the GF(2^m) arithmetic units.

As it was shown, the dP operation can be executed in parallel or serially and even side channel resistance could be provided. The implementation results of the serial, parallel and side channels attacks resistant architecture for dP are presented in this section.

Synthesis results of different hardware modules for dP were obtained in order to select the best one in terms of area and performance. The selected hardware for dP will be used to implement the reconfigurable system that will provide interoperability for ECC. Different ECC co-processors were designed, one for each tuple presented in appendix B. These are standard implementation parameters recommended in [7] and [2]. All the hardware designs of the ECC co-processor were validated by simulation, applying the test vectors shown in appendix B.

5.5.1 Parallel architecture for ECC

The first explored hardware architecture for dP was the parallel implementation of algorithm 4.5.

Each one of the designed ECC co-processors were synthesized for a xc2v4000 FPGA using the area and speed optimization options and enabling/disabling the *keep hierarchy* option. Optimizing for area indicates to the synthesis tool to reuse the hardware as much as possible. In optimization for speed, the clock frequency is optimized without any restriction of area resources. If the option *keep hierarchy* is enabled, the synthesis tool synthesizes each modules in the design to a dedicated area in the FPGA. If this option is not enabled, all the modules are merged in just one big module and synthesized. Keeping the hierarchy is recommended if FPGA reconfigurability is going to be exploited although more area resources are needed and maybe, low performance is obtained compared to the results using the no keep hierarchy option.

Area results of dP architectures are shown in figure 5.3. The ECC co-processor

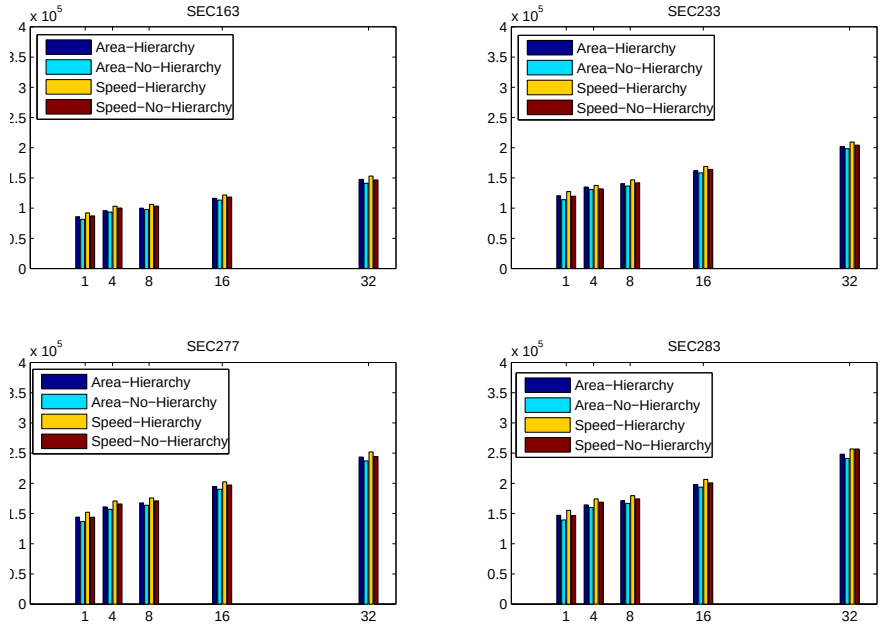


Figure 5.3: Area resources of the parallel implementation of dP for different security levels and parallelism grade in the field multiplier.

uses the following GF(2^m) arithmetic modules:

- `GF2m_Mul_1` and `GF2m_Dserial_Mul_1` for multiplication.
- `GF2m_Div_2` for division.
- `GF2m_Sqr_1` for squaring.

The figure shows in the x-axis the different digit sizes (digit = 1 in the case of the serial multiplier) for the multiplier. The number of logic gates used by the architecture for each security level is shown in the y-axis. Keeping the hierarchical structure of the HDL design and optimizing for speed leads to use bigger amount of gates compared with the other synthesis options. In all cases the increase in area is linear respect to the security level.

The timing achieved by dP architecture using different finite field multipliers and security levels is shown in figure 5.4. Different curves correspond to the different synthesis options previously mentioned. The results were obtained by simulating the architecture and counting the cycles per clock spent. Then, the real

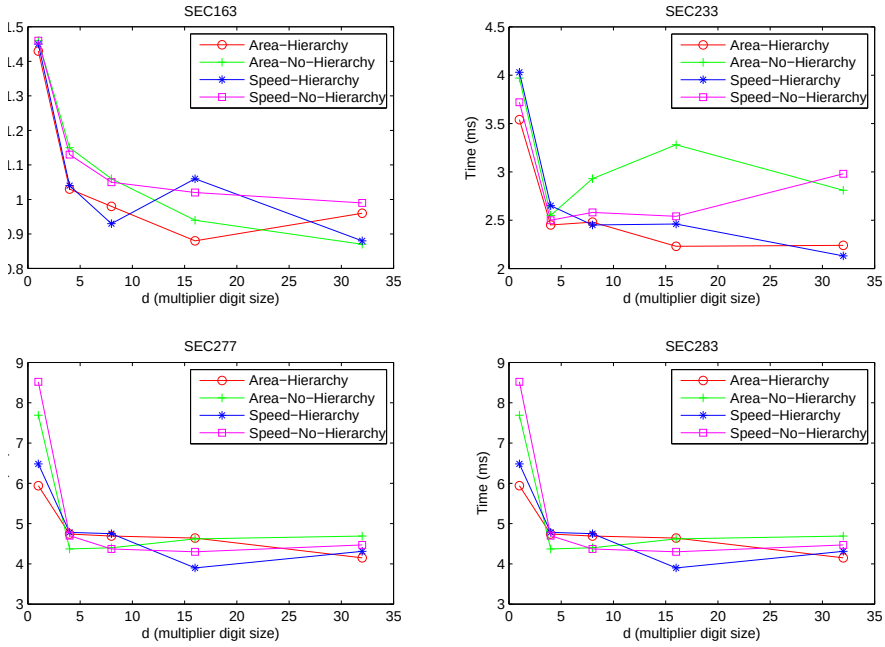


Figure 5.4: Timing to compute dP using the parallel architecture for different security levels and parallelism grade in the field multiplier.

time was computed by multiplying the clock cycles with the delay of the clock cycle given by the synthesis tool. From the figures it is concluded that the architecture using the serial multiplier requires the minimum area but is the worst performer. Using a digit multiplier with $d = 4, 8, 16$ results in a similar performance but almost the half of time respect the serial multiplier. A digit-serial multiplier with $d = 32$ performs well in some cases but the area requirements increase approximately 40% respect to the architecture that uses the serial multiplier.

Figures 5.5 and 5.6 show the different area resources and performance of the dP architecture for different security levels. These are results from synthesis optimized by area and keeping the hierarchical structure of the design. These results remark the importance of using efficiently the available silicon to obtain the best performer implementations.

5.5.2 Serial architecture for ECC

A new architecture for dP was derived from the parallel one. Instead of having a dedicated unit for ECC-Add and ECC-Dbl and performing both operations in

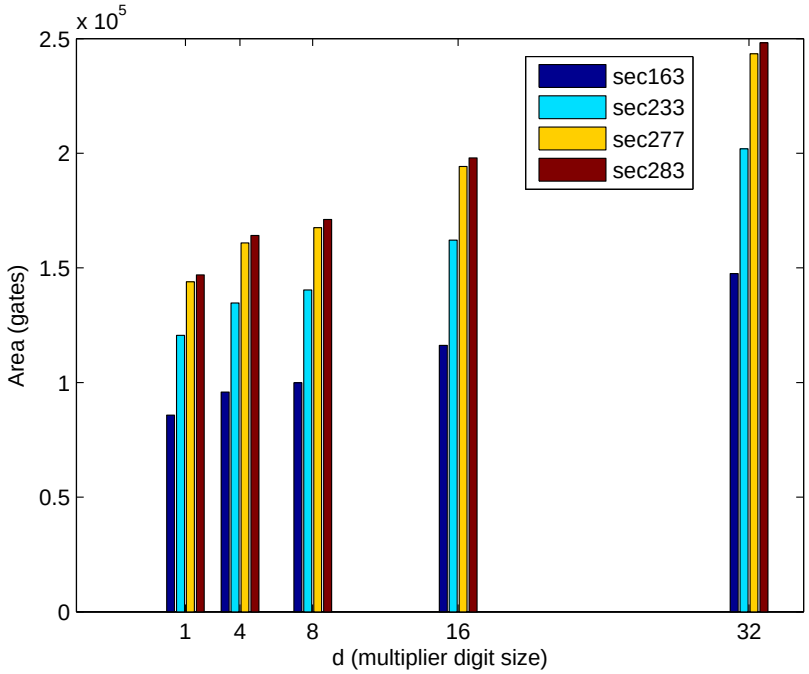


Figure 5.5: Architecture 1 area resources for different security levels

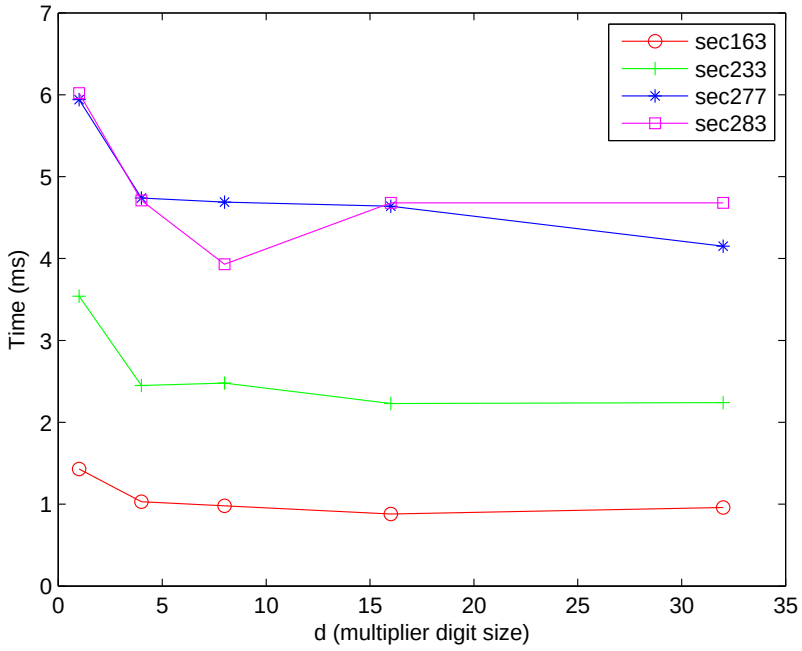


Figure 5.6: Timing to compute dP using architecture 1 and different parallelism grade in the field multiplier

parallel, it was explored the area usage and performance of an architecture that implements the serial algorithm for dP listed in algorithm 4.1. In this way, a hardware module to support both kinds of elliptic point addition was developed. The difference with the parallel architecture is that the control was changed by adding the required states to the finite state machine and adding more control signals. Although the affine coordinates remain, the organization of the module for point addition was slightly modified but the arithmetic modules were not.

The serial architecture was validated and synthesized for all the cryptosystems as in the parallel one. Figures 5.7 and 5.8 show the area and time results for this new hardware architecture for dP. The results were obtained from the synthesis tool optimizing by area and enabling the keep hierarchy option.

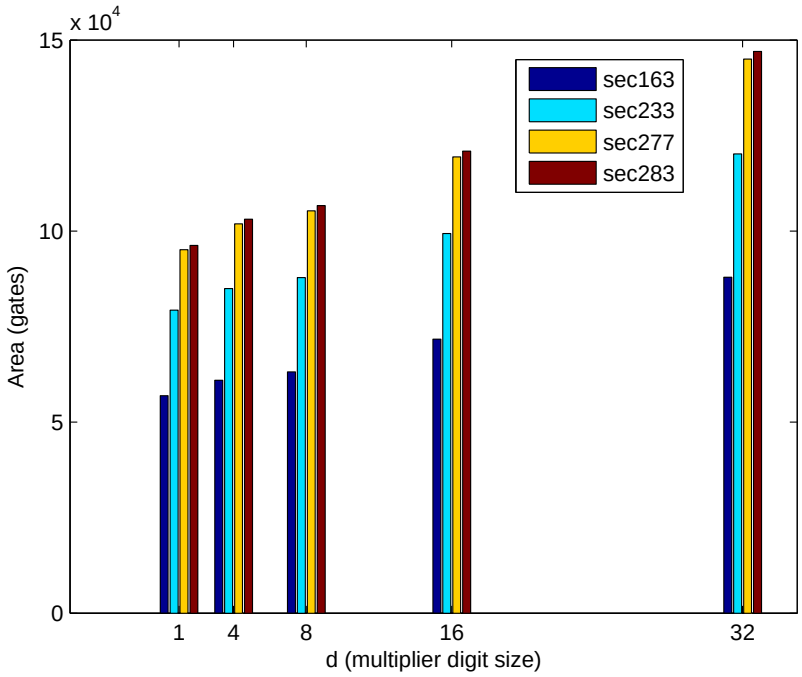


Figure 5.7: Area resources (logic gates) used by the ECC serial architecture for different security levels.

Comparing both the serial and the parallel ECC architectures for each security level, the serial dP implementation saves 40% of area resources ocupied by the first architecture. However, the time to compute the scalar multiplication increases in 40% - 30%. For illustration purposes, figure 5.9 compares the area resources for both implementations for the cryptosystem in the field $m = 163$ and different

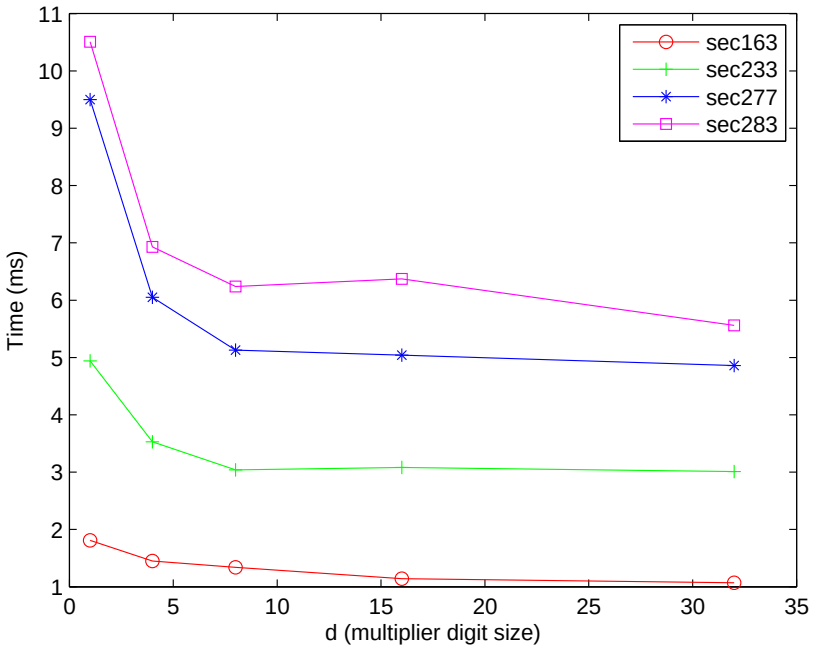


Figure 5.8: Timing (ms) to compute dP by the ECC serial architecture for different security levels.

finite field multipliers.

5.5.3 An ECC hardware architecture resistant to Side Channel Attacks

This ECC co-processor is the one resulting of the implementation of the unified point addition formula presented in section 4.4.1 and the implementation of the serial algorithm 4.6 for dP.

For this case, the ECC co-processor was tested using each version of the arithmetic modules presented in section 4.2 in order to select the best in terms of area or latency. For the implementation of this new ECC co-processor the digit-serial multiplier was not considered. The serial multiplier was only considered in order to have a more compact architecture of the ECC co-processor. With four implementations of the divider and five for the serial multiplier, 4x5 = 20 different combination of arithmetic modules should be tested. However, due the circuits one and two for the multiplier occupy almost the same amount of area, only the circuit two was considered. The same is for the version one of the divider

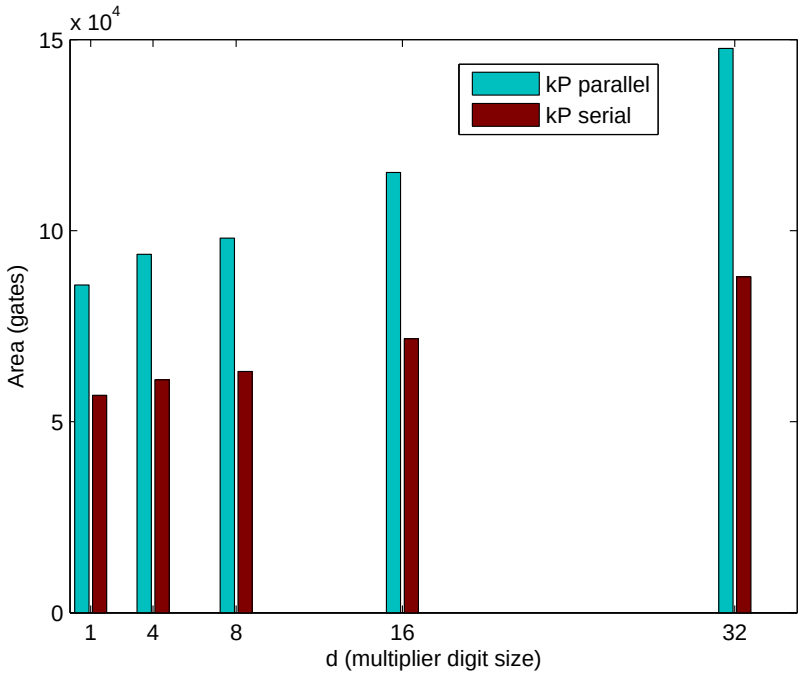


Figure 5.9: Comparison of area resources for the parallel and serial implementation of dP algorithm.

`GF2m_Div_1` and `GF2m_Div_2`, which have the same operational frequency. So, version one of the circuit `GF2m_Div_2` was discarded and the other three versions of the divider were considered for designing the ECC co-processor. The results shown in tables 5.4 and 5.5 are for the twelve $GF(2^{163})$ co-processors synthesized and optimized by area and speed respectively.

According to table 5.4, the co-processor that occupies less area is the one that uses the version one of circuit `GF2m_Mul_2` for multiplication and the version four of the division algorithm. However, in table 5.4, the co-processor that occupies less area and runs faster is the one that uses the version three of circuit `GF2m_Mul_2` for multiplication, and the version four of the divider. According to table 5.5, the fastest co-processors are the ones that use the circuit `GF2m_Div_2` but requires more are compared to the co-processors using the circuit `GF2m_Mul_1`

These results shown the importance of trying with different GF(2^m) arithmetic algorithms. Theoretically, the version three of circuit two for multiplication required minimum area but when it was integrated together the other modules, it was not the best performer.

Table 5.4: Synthesis results for three implementations of the dP co-processor on the Virtex4 FPGA (Optimized by area).

Multiplier: Circuit , version 2				
Divider	V1	V2	V3	V4
Slices	2780	2779	3091	2713
Slice FF	2520	2518	2536	2534
4-in LUTs	4977	4970	5497	4836
Freq. (MHz)	69	64	88	88
Multiplier: circuit 2, version 1				
Divider	V1	V2	V3	V4
Slices	2769	2768	3080	2702
Slice FF	2683	2681	2699	2697
4-in LUTs	4975	4968	5495	4835
Freq. (MHz)	69	64	88	88
Multiplier: circuit 2, version 3				
Divider	V1	V2	V3	V4
Slices	2778	2777	3089	2711
Slice FF	2520	2518	2536	2534
4-in LUTs	4974	4967	5494	4833
Freq. (MHz)	69	64	88	88

Table 5.5: Synthesis results for three implementations of the *dP* co-processor on the Virtex4 FPGA (Optimized by speed).

Multiplier: circuit 1, version 2				
Divider	V1	V2	V3	V4
Slices	3045	3040	3613	3529
Slice FF	2524	2521	2820	2826
4-in LUTs	5707	5702	6834	6752
Freq. (MHz)	81	81	136	150
Multiplier: circuit 2, version 1				
Divider	V1	V2	V3	V4
Slices	3051	3038	3080	3557
Slice FF	2687	2684	2699	2922
4-in LUTs	5718	5699	5495	6733
Freq. (MHz)	81	81	88	138
Multiplier: circuit 2, version 3				
Divider	V1	V2	V3	V4
Slices	3053	3038	3637	3464
Slice FF	2524	2521	2819	2758
4-in LUTs	5721	5699	6880	6568
Freq. (MHz)	81	81	136	138

This last version of the ECC co-processor was the one that used low area while keeping a high throughput. This ECC co-processor will be used to implement the reconfigurable system that will allow interoperability for elliptic curve cryptography.

5.6 ECC reconfigurable system results

The reconfigurable design shown in figure 4.12 was implemented using the ISE tools for partial reconfiguration [77] and scripts. This system was first defined using the EDK 8.2 tools. The system includes the PowerPC microprocessor, local buses PLB and OPB for data communication, an universal asynchronous receiver/transmitter UART module for the I/O user interface, memory blocks for data and program, and the ECC co-processor. A step by step guideline to implement the reconfigurable system is presented in appendix A.

For the reconfigurable module, the best co-processor optimized by area was used. Unfortunately, that module does not function properly. That is, the co-processor uses the clock signal of the ML403 board, that is the same clock signal that the microprocessor uses. The clock signal in the ML403 is 100 MHz but the most compact dP co-processor optimized by area reaches a maximum speed of 88 MHz. This caused some problems for the correct operation of the co-processor. So, the best performer co-processor optimized by speed was used. The co-processor that fitted in the maximum available area in the FPGA was the one for the finite fields GF(2^{113}) and GF(2^{131}).

For the implementation, the FPGA was divided in two columns. The resources of the left column were assigned to the fixed part of the system. In this part there is a hard core of the PowerPC 405 processor. The resources of the right column were assigned for the reconfigurable part, that in this case correspond to the dP co-processor. The available area for the reconfigurable part was 2,760 slices. The dP co-processor for the 113-bit security fits well in this area but the one for 131-bit security did not. Then, the co-processor was synthesized using the area optimization criteria. Results for each part of the system, fixed and reconfigurable are shown in table 5.6.

The time to perform the scalar multiplication is given in table 5.7. The co-processor uses the same clock frequency of the bus system, which is the same

Table 5.6: Area results for the reconfigurable system.

Hw Resources	**Fixed part**	**Reconfigurable part**	
		Security level (bits)	
		113	131
Flip-Flops	942	1,966	2,233
LUTs	891	4,535	4,053
Slices	1027	2,748	2,336
Dual Port RAMs	216	0	0
Shift registers	64	0	0
RAMB16s	16	0	0
Gate count	1,079,972	44,411	43,751

Table 5.7: Time results for the reconfigurable dP co-processor.

Sec. level	**Cycles/**dP	**Time** (ms)
113	51,730	0.52 ms
131	68,887	0.69 ms
163	107,043	1.07 ms

that the microprocessor clock, 100 MHz. All results were validated by comparing them against a software implementation that is a slight modification of the code available in [78].

5.7 Comparison with related work

A reconfigurable system for providing interoperability of elliptic curve cryptography had not been considered in the literature. So it is difficult to provide a comparison against related work. However, the dP co-processor can be compared because most of the related work is concerned with hardware implementations of dP.

In order to provide a fair comparison, the results achieved in this work are compared against the ones reported in the literature under the same conditions, that is, against related works using binary fields, affine coordinates and the binary method for the dP computation. The time and area results comparison against these works is shown in table 5.8. The ECC co-processor was synthesized using

the same FPGA used in related work for a fair comparison. Although other parameterizable implementations reported in the literature have used different arithmetic algorithms in order to achieve the dP operation faster, in this work it was found that reduced area designs are desired due the problems experienced when the reconfigurability of the co-processor was implemented. The proposed co-processor uses less resources than Kerins et al., [19] and computes dP faster. The co-processor discussed in this paper is also faster than Leong and Leung [41] at the expense of higher area resources.

Table 5.8: Comparison results.

Ref.	m	**Time** *(ms)*	**Device**	**FPGA Slices**
[19]	151	5.1	XCV2000E	4048
	176	6.9		
	191	8.2		
	239	12.8		
[41]	113	3.7	XCV300-4	1290
	155	6.8		1567
	281	14.4		2622
[72]	179	2.47	XCV800	10,626
[39]	113	10.9	AT94K40	-
[17]	163	0.14	XCV2000E	19,000
[42]	160	3.81	XCV800	-
[79]	163	0.49	V2Pro	4,749
[22]	163	0.07	XCV2000E	5,008
[40]	191	0.05	VirtexE 3200	18, 314
[21]	113	0.27	XC2V6000	6,961
This work	113	0.84	XCV2000E	2449
	131	1.25		2582
	163	2.09		3324
	113	1.05	XCV300-4	2515
	131	1.58		2516
	113	0.52	Virtex4	2405
	131	0.69		2871
	163	1.07		3528

Table 5.8 also shows the comparison results of the proposed dP co-processor against other works that have used projective coordinates, like [39] (10.9 ms for $m = 113$), [42] (3.8 ms for $m = 160$) or [72] (2.47 ms for $m = 179$). The

use of projective coordinates supposes a better performance because inversions are avoided in each point addition operation at the cost of more multiplications. Other works using projective coordinates perform dP faster than the co-processor presented in this article but they require higher area resources. For example, [79] uses 4,749 slices from a Virtex2 Pro and performs the dP in the field $GF(2^{163})$ in 0.49 ms. In [17], the area required is 19,000 slices from a XCV2000E FPGA while the dP operation in the field $GF(2^{163})$ is computed in 0.14 ms. The area used in [17] is six times bigger that the area used by the co-processor proposed in this thesis, and the one used in [72] is three times bigger. In [21], the operation dP is performed in a half of the time achieved in this work but the area required is almost three times bigger.

The results of the co-processor discussed in this thesis are not only competitive or better that the reported in the literature but also provide the capability of adapting at run time the hardware to support the costly arithmetic in elliptic curve cryptography.

5.8 Summary

This chapter presented and discussed the results of this research. These results show that it is possible to have an interoperable elliptic curve cryptographic co-processor that enables interoperability for the ECC cryptographic schemes using different implementation parameters. Although other parameterizable implementations (non interoperable) reported in the literature have used different arithmetic algorithms in order to achieve the dP operation faster, we found that reduced area designs are desired due the problems experienced when implemented the reconfigurability of the co-processor. The area resources for our co-processor and its performance are competitive among the reported works using the same finite field, coordinates and the method for the dP computation. Comparison results against those works were presented.

Chapter 6

Conclusions and directions

This thesis presented the design and implementation of a reconfigurable system for providing interoperability of elliptic curve cryptography. The proposed system is well suited for IPSec or cryptographic schemes executed on the server side where the agreement of the implementation parameters is at runtime. Dynamic reconfiguration of the ECC co-processor allows to support different security levels while retaining high performance. A new co-processor for ECC using affine representation was designed and implemented in FPGA technology. This co-processor is well performer as the ones using projective representation while the architecture design is simpler and in a modular way, so the system can be updated with better performer modules for finite field arithmetic, which still determine the whole latency for the dP operation.

The unified formula for ECC point addition makes the elliptic curve point addition operations indistinguisible, which makes the dP hardware implementation more resistant to side channel attacks.

This is the first reported partially reconfigurable solution that is well suited to provide dynamic adaptation to different tuples T and hence to provide interoperability in elliptic curve cryptography. Further work would be done to have extremely light-weight (low area) ECC in hardware. Low-power ECC is necessary mainly if the co-processor is used in constrained devices.

6.1 Summary of contributions

The contributions of this work are:

1. A reconfigurable hardware architecture for ECC that allows interoperability.

2. A reconfiguration strategy for interoperable ECC architectures.

3. An study of the best performer ECC arithmetic algorithms and implementation parameters. This study will allow to establish an area/performance trade off in the architecture.

4. A new formulation for elliptic curve point addition using affine representation that increases the physical security of a hardware implementation of dP using this new formulation.

The proposed architecture will operate in an environment where a secure communication link is required. The entities involved in a communication will be capable enough to authenticate each other and establishing the secure communication link (confidentiality and integrity) using the reconfigurable solution proposed. Also, they will be able to manage different security levels depending on the available computational resources.

6.2 Future work

Further work could be done considering more implementation issues for elliptic curve cryptography. The ECC co-processor discussed in this thesis is for binary field of the form GF(2^m). Further work could consider the use of prime fields GF(p) and unified hardware architectures for finite field arithmetic. That is, to use a single arithmetic unit that perform arithmetic for both fields. This will enable the use of more curves than only consider the ones for binary fields. Also, it is interesting to explore the design of a hardware module for scalar multiplication that consider not only polynomial but also normal basis. The squaring operation is binary fields that use normal basis is a simple shift operation. Some area reduction could be achieved and the design of the field multiplier and inverter considering both kinds of basis could be explored.

By the side of the implementation, a softcore processor could be used instead of a hardcore processor as it was done in this work for validation purposes. This will allow to have an open design independent of a the technology.

Publications

The results derived from this research work were published and presented in different forums. The next are articles that reports the results of this dissertation.

1. M. Morales-Sandoval, C. Feregrino-Uribe, René Cumplido, and I. Algredo-Badillo. An area/performance trade-off analysis of a GF(2^m) multiplier architecture for elliptic curve cryptography. *In Computers and Electrical Engineering, Elsevier*, doi:10.1016/j.compeleceng.2008.05.008, 2008.

2. M. Morales-Sandoval, C. Feregrino-Uribe, R. Cumplido, and I. Algredo-Badillo. A run time $GF(2^m)$ reconfigurable co-processor for elliptic curve scalar multiplication. In *16th International Conference on Computing (CIC07).* IEEE Computer Society. 2007.

3. R. Duraisamy, Z. Salcic, M. Adriano Strangio, and M. Morales-Sandoval. Supporting symmetric 128-bit aes in networked embedded systems: An elliptic curve key establishment protocol-on-chip. *EURASIP Journal on Embedded Systems*, 2007:Article ID 65751, 9 pages, 2007. doi:10.1155/2007/65751.

4. M. Morales-Sandoval and C. Feregrino-Uribe. $GF(2^m)$ arithmetic modules for elliptic curve cryptography. In *3rd International Conference on ReConFigurable Computing and FPGAs (ReConFig06).*, pages 176–183. IEEE Computer Society, September 2006.

5. R. Duraisamy, Z. Salcic, M. Morales-Sandoval, and C. Feregrino-Uribe. A fast elliptic curve based key agreement protocol-on-chip (PoC) for securing networked embedded systems. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*, pages 154–161. IEEE Computer Society, August 2006.

Appendix A

Guidelines for partial reconfiguration of a $GF(2^m)$ ECC co-processor

This appendix describes the steps to implement partial reconfiguration of an elliptic curve crypto-processor using the tools ISE, EDK and PlanAhead.

A.1 The base design

The design of the system that includes both the fix and reconfigurable modules must be HDL-designed. The EDK tool allows to define this system using a wizard. The system shown in figure 4.12 in section 4.6 was targeted to the prototyping board ML403 [80] which includes a FPGA Virtex xc4vfx12-10ff668.

A guide to develop the base design using EDK is provided by Xilinx [81]. Throughout the EDK wizard, the base design is defined using the minimum components. The size of the memory must be as big as the size of the code to be executed by the microprocessor. Once all the components, sizes for memory and applications have been selected, the HDL-design will be generated and the EDK Platform Studio will launch.

The module corresponding to the reconfigurable one will be integrated later using the EDK wizard for peripherals. This wizard asks for the peripheral name, how it will be interconnected, what signals will be used as interface with the local bus, etc. For our case, the OPB bus interconnected by the IPIF module

was selected. The communication of our module with the local microprocessor is by registers mapped to memory. For detailed steps to use the EDK wizard for peripherals the reader can refer to [81].

To add the reconfigurable module to the design, use the view `System Assembly` and the tab `IP Catalog` from the window `Project Information Area` of the EDK Platform Studio. Expand the option `Project Repository` to locate the peripheral to be added to the base system. Drag and drop the peripheral to the view `System Assembly`. The added peripheral must be connected to the local bus, in this case, to the OPB bus. Finally, the peripheral clock signal is connected by connecting the signal `OPB_CLK` with the signal `sys_clk_s`. Then, a logical address for the new peripheral must be generated.

An application program is generated by default to test the peripheral. This application writes and reads to and from the registers mapped to memory to interact with the peripheral. To add these files and test the new added peripheral do the following:

1. Select the tap `Applications` from the window `Project Information Area` and expand the option `Sources`.

2. Remove all `.c` files and add the ones from the directory `\edk_design\drivers\peripheral_name_v1_00_a\src`

Then the application is built and the bitstream to configurate the FPGA is generated.

Before downloading the bitstream to the FPGA and to see the application running on the microprocesor, open a windows terminal with the following configuration: 9600 bps, 8 bits for data, no bits for parity and one bit for stop. The application IMPACT can be used to download the bitstream.

A.2 Modifying the base design

To have our peripheral interacting with the local microprocessor, the file `user_logic.vhd` in the directory `\edk_design\pcores\periferico_name\hdl\vhdl` is modified.

This file defines the peripheral connected to the IPIF module that is the one that communicates the peripheral with the bus of the microprocessor.

The original `user_logic.vhd` file reads data coming from the microprocessor and writes the same data back to the microprocessor. This file can be modified to create an instance of the reconfigurable module and use the input/output registers as input and output signals for this instance. So, by writing or reading values from the registers we interact with the reconfigurable module.

We have to add all the associated vhdl files of the reconfigurable module to be considered in the synthesis process. We do this by including the full path to the vhdl files at the end of the `.pao` file located in the directory
`\edk_design\pcores\periferic_name_v1_00_a\data`

The software application is modified to write and read to and from the registers that are now used as interface signals for our reconfigurable module. The new bitstream is updated and downloaded to the FPGA to have our peripheral module working.

A.3 Different versions of the peripheral: partial reconfiguration

A partially reconfigurable system can be designed if the user peripheral is going to have different versions of implementation but keeps the same interface. To generate a partially reconfigurable system it is recommended to have a well structured directory tree to process all the related files to the design. This is because the synthesis process and the generation of the configuration files will be performed manually (using batch files). The recommended directories are:

- `Reconfig_Design`: This is the project directory.
- `Base`: It will contain all the files associated to the fixed part of the system.
- `BITS`: It will contain the full and partial configuration bitstreams.
- `Data`: It will contain the constraint files and the busmacro files.
- `Files_base`: It will contain all the `.ngc` files and the file `system.bmm` located in the directory `\edk_desig\implementation`. The file `.ngc` for the reconfigurable module will not be considered.
- `Final`: In this directory the `.bit` file will be merge together the `.elf` file.

- `Merges_Modules`: It will contain the static and partial bitstreams.
- `Merges_full`: In this directory will be merged the static and partial bitstreams.
- `PlanAhead`: In this directory will be created the PlanAhead project to generate the constraint file for the design.
- `PRM`: It will contain the `.ngc` files for each one of the versions of the reconfigurable module.
- `Top`: It contains the `system.ngc` that is the synthesized system that includes the fixed, reconfigurable module and busmacros.

A.4 `.ngc` files generation

The original `system.vhd` file that defines the complete system including the fixed part, the reconfigurable part and the busmacros is modified to include the following:

1. A new source for the clock signal. To do this, the instruction `dcm_clk_s <= sys_clk_pin;` in the file `system.vhd` should be commented. `dcm_clk_s` is the global clock signal for DCM module that produces the clock signal `sys_clk_s`. This last signal is the clock signal for all modules in the base design. The clock signal for DCM must now pass throw a buffer.

```
ibufg_0 : IBUFG
    port map (
      I => sys_clk_pin,
      O => dcm_clk_s
    );
```

   Then, the global clock signal `sys_clk_s` is taken from one of the outputs of the DCM module (`dcm_out_clk`) and passed throw a buffer.

```
bufg_clk : BUFG
    port map (
```

```
      I => dcm_out_clk,
      O => sys_clk_s
    );
```

2. The bus macros are instantiated and added to the original `system.vhd` file. All the necessary signals for interconnection must be declared and instantiated.

3. The `system.vhd` file with all the changes is synthesized to generate the `system.ngc` file. The synthesis process is performed out of the EDK environment. To do this, create a `.bat` file in the directory `\edk_design\synthesis` with the command:
   `>xst -ifn system_xst_scr`
   The resulting `.ngc` file must be placed in the directory `\regonfig_design\top`.

4. Now, the constraint file is generated. This is done using the PlanAhead tool to define the area in the FPGA for the fixed and reconfigurable parts. The buffers and busmacros must be specified to assign them specific locations in the FPGA. Input files to the PlanAhead tool are the `.ngc` file generated previously and the `.ucf` file generated by EDK and located in the directory `\edk_design\data`.

   After verifying that the placement of the components in the design is correct, all the design is exported to the directory `PlanAhead\export` directory. The new generated `system.ucf` file is copied to the directory `\reconfig_desig\data`.

   The file `system.bmm` and all the `.ngc` are copied from the directory `\edk_design\implementation` to the `\reconfig_design\files_base` directory.

   The `.nmc` busmacro files are also copied to the directory `\reconfig_design\data`.

5. All the versions for the reconfigurable module are generated. Batch files can be used to do this task.

6. Finally, the design flow for partial reconfiguration is performed. The batch files that implement this design flow are listed next:

   Command line instructions to generate the top design.

```
@echo Cleaning files ..
del/q Top\Initial\*

@rem Initial
@echo Step 1 - Build top level context
cd Top\Initial
copy ..\system.ngc
copy ..\..\Data\system.ucf
copy ..\..\Data\*.nmc

REM Translate the top level design
ngdbuild -p xc4vfx12-10ff668 -modular initial system.ngc
pause
```

Command line instructions to implement the fixed part of the design.

```
del/q base\*
del/q Merges\*
pause
@rem Build the static modules
@echo Step 2 - Build the static modules
cd Base
copy ..\files_base\*.ngc
copy ..\Data\system_full.ucf system.ucf
copy ..\Data\*.nmc
copy ..\files_base\system.bmm
copy ..\files_base\system.bmm ..\Merges

ngdbuild -p xc4vfx12-10ff668 -bm system.bmm -modular
   initial ..\Top\Initial\system.ngo
map -timing system.ngd
par -w system.ncd system_base_routed.ncd
```

Now, each version of the reconfigurable module is generated.

```
```

```
@rem Build partial modules
@echo Step 3 - Build the partial modules

del/q PRM\PRM_113\*
del/q PRM\PRM_131\*
del/q PRM\PRM_163\*
pause

cd PRM\PRM_113
copy ..\..\Synt_PRM\PRM_113\ecc_core_0_wrapper.ngc
copy ..\..\Data\system.ucf system.ucf
copy ..\..\Data\*.nmc
copy ..\..\Base\static.used arcs.exclude
ngdbuild -p xc4vfx12-10ff668 -modular module -active
   ecc_core_0_wrapper ..\..\Top\Initial\system.ngo
map system.ngd
par -w system.ncd system_routed.ncd

cd ..\..
cd PRM\PRM_131
copy ..\..\Synt_PRM\PRM_131\ecc_core_0_wrapper.ngc
copy ..\..\Data\system.ucf system.ucf
copy ..\..\Data\*.nmc
copy ..\..\Base\static.used arcs.exclude
ngdbuild -p xc4vfx12-10ff668 -modular module -active
   ecc_core_0_wrapper ..\..\Top\Initial\system.ngo
map system.ngd
par -w system.ncd system_routed.ncd
cd ..\..
```

Now, the bitstreams of the fixed and reconfigurable parts are generated.

```
@rem Assemble full design
@echo Step 4 - Merge and generate bitstreams
del/q Bits\*.bit
del/q Bits\*.bmm
del/q Merges_Modules\*
```

```
pause
cd Merges_Modules
copy ..\files_base\system.bmm
copy ..\Base\system_base_routed.ncd static.ncd
copy ..\PRM\PRM_113\system_routed.ncd SEC_113.ncd
copy ..\PRM\PRM_131\system_routed.ncd SEC_131.ncd
pause
PR_verifydesign.bat static.ncd SEC_113.ncd SEC_131.ncd
```

The fixed and reconfigurable parts are merged to generate a single configuration file.

```
del/q Merges_Full\*

cd Merges_Full
copy ..\files_base\system.bmm
copy ..\Base\system_base_routed.ncd static.ncd
copy ..\PRM\PRM_113\system_routed.ncd SEC_113.ncd
copy ..\PRM\PRM_131\system_routed.ncd SEC_131.ncd
pause
PR_assemble static.ncd SEC_113.ncd

@Los archivos bit generados se copian a la carpeta \
   reconfig_design\bits.

cd Merges_Modules
copy SEC_113_partial.bit ..\Bits
copy SEC_131_partial.bit ..\Bits
copy PRtmpdir\SEC_113_full.bit ..\Bits
copy PRtmpdir\SEC_131_full.bit ..\Bits
copy PRtmpdir\static.bit ..\Bits
copy system_bd.bmm ..\Bits
cd ..
cd Merges\_Modules
copy static\_full.bit ..\Bits
```

The .elf executable C program from the EDK design is added to the full bitstream using the bitgen program. The following command line instructions are executed.

```
cd Final
REM del/q Final/*
REM copy .elf and .mhs and todo el directorio pcores

copy ../Bits/system_bd.bmm
copy ../Bits/static_full.bit
pause
bitinit system.mhs -bm system_bd.bmm -bt static_full.bit -
   o download.bit -pe ppc405_0 executable.elf
pause
```

The result is the download.bit file that is downloaded to the FPGA as the initial configuration file. After, the Partial reconfiguration version of IMPACT is used to download the partial bitstreams.

Appendix B

GF(2^m) ECC co-processor test vectors

This appendix presents the test vectors used to verify the correct operation of the arithmetic units and the ECC co-processor developed in this thesis. These test vectors were generated from a software application that is a slight modification of the code available in [78]. The elliptic curve domain parameters over GF(2^m) are specified by the tuple $T = (m, f(x), a, b, G, n, h)$. m is the order of the finite field and determines the security level. $f(x)$ generates the finite field GF(2^m) and a and b defines the elliptic curve $E : y^2 + xy = x^3 + ax^2 + b$ over GF(2^m). G is the generator of E and n and h are the order of G and the co-factor of E respectively. Details of these parameters were presented in section 2.2.3.

B.1 Test vectors for finite field arithmetic

GF(2^{163}) multiplication and division

A: 6 3f497f91 531bca6e 2f2f677b 4c3a11d2 2c3b0a08

B: 5 284f6600 ca2b553 67eb4631 75184da9 f2bed412

A/B: 2 00bdc727 d7b7a196 88bf3fe 9177fdbe 4f500c2c

$A * B$: 6 a6532ac9 e16d6657 a3aaa7b5 a6287ba2 6a11a2f4

B.2 Test vectors for scalar multiplication dP

B.2.1 Test vectors for $m = 113$

Tuple T SEC113r1 recommended by [2] and [9].

$f(x) = x^{113} + x^9 + 1$
a = 003088 250CA6E7 C7FE649C E85820F7
b = 00E8BE E4D3E226 0744188B E0E9C723
$G = (x, y)$
x = 009D73 616F35F4 AB1407D7 3562C10F
y = 00A528 30277958 EE84D131 5ED31886
n = 010000 00000000 00D9CCEC 8A39E56F
$h = 2$

The scalar multiplication $(u, v) = dG$ is

d = 17876 FC01A2EA 920FF4E4 789CF04B
u = 03650 2761D847 1F2FDE59 3713DFEF
v = 1ECB1 4E64590E 95EB1424 6703532F

B.2.2 Test vectors for $m = 131$

Tuple T SEC131r1 recommended by [2] and [9].

$f(x) = x^{131} + x^8 + x^3 + x^2 + 1$
a = 07 A11B09A7 6B562144 418FF3FF 8C2570B8
b = 02 17C05610 884B63B9 C6C72916 78F9D341
$G = (x, y)$
x = 00 81BAF91F DF9833C4 0F9C1813 43638399
y = 07 8C6E7EA3 8C001F73 C8134B1B 4EF9E150
n = 04 00000000 00000002 3123953A 9464B54D
$h = 2$

The scalar multiplication $(u, v) = dG$ is

d = 6 FC01A2EA 920FF4E4 789CF04B E39E9302
u = 7 E1FC59FF 9584DC1A 821CD518 3AFD0CC4
v = 5 75F2BC49 31BBE7CC 433E7037 0A9FBDD9

B.2.3 Test vectors for $m = 163$

Tuple T SEC163r1 recommended by [2], [9] and [8].

$f(x) = x^{163} + x^8 + x^7 + x^3 + 1$
a = 7 B6882CAA EFA84F95 54FF8428 BD88E246 D2782AE2
b = 7 13612DCD DCB40AAB 946BDA29 CA91F73A F958AFD9
G = (x, y)
x = 3 69979697 AB438977 89566789 567F787A 7876A654
y = 4 035EDB42 EFAFB298 9D51FEFC E3C80988 F41FF883
n = 3 FFFFFFFF FFFFFFFF FFFF48AA B689C29C A710279B
h = 2

The scalar multiplication $(u, v) = dG$ is

d = 1 33E3CAE7 2CD0F448 B2954810 FB75B5E3 D8F43D07
u = 0 70326580 B9D897AB 325BCD03 289C8E4F 99BF0598
v = 1 7F250719 80A6052C 67E2EBAA 62606AB1 DFB3312E

B.2.4 Test vectors for $m = 233$

Tuple T SEC233r1 recommended by [2], [9] and [8].

$f(x) = x^{233} + x^{74} + 1$
a = 01
b = 066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD
G = (x, y)
x = 0FA C9DFCBAC 8313BB21 39F1BB75 5FEF65BC 391F8B36 F8F8EB73 71FD558B
y = 100 6A08A419 03350678 E58528BE BF8A0BEF F867A7CA 36716F7E 01F81052
n = 080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF
h = 4

The scalar multiplication $(u, v) = dG$ is

d = 76 FC01A2EA 920FF4E4 789CF04B
E39E9302 1D486C8 CF0EC27F AB882021
u = F3 EE481F9D E6668307 8925D697
8AAE5768 636A6B7E 1FB4163D 219ADA5A
v = 59 90AE743E 91F662F3 BB1D28E1
8A47720D EE01810B 4F10A05F B5CBCE7B

B.2.5 Test vectors for $m = 277$

Tuple T 277-bit recommended by [82].

$f(x)$ = $x^{277} + x^{12} + x^6 + x^3 + 1$
a = 185304 4e52ac19 59e666eb 97684079 46267563
89C3084E 1C0E8EE5 8B5ADE55 B0E94F06
b = 12709B 9501DBD0 C98DC5E7 E17AF396 B445303D
FDBDEA0A AE05840A 8204625E 0B9157B9
G = (x, y)
x = 18094 9B3BBF7F 5168DA76 47F9BBAE 716F02F6
174EC79D EA5AC9AE C5FF48E4 D696323B
y = 1CB7 297D4520 04A0F2C3 4F33E5A6 90122103
B5F78BE5 B838AA97 848CCFED D1F60618
n = 0FFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFB82 4C3073AC E595EDF7 ACEEC635 A1F5740B
h = 2

The scalar multiplication $(u, v) = dG$ is

d = 1B7876 FC01A2EA 920FF4E4 789CF04B E39E9302
1D486C8 CF0EC27F AB882021 AF8E5BEA
u = 38425 E18D964F 953C9404 7D3C4052 35C1BC6D
B0A516D3 EEF6409A 6C784F81 8F4ACCD8
v = 8866 7913681D 12EF75FE 892CE074 FC5F4166
96703641 B59B00D3 704679AC 77A5C414

B.2.6 Test vectors for $m = 283$

Tuple T SEC283r1 recommended by [2], [9], [8] and [7].

$$
\begin{aligned}
f(x) &= x^{283} + x^{12} + x^7 + x^5 + 1 \\
a &= 01 \\
b &= 27B680A\ C8B8596D\ A5A4AF8A\ 19A0303F\ CA97FD76 \\
&\quad 45309FA2\ A581485A\ F6263E31\ 3B79A2F5 \\
G &= (x, y) \\
x &= 5F93925\ 8DB7DD90\ E1934F8C\ 70B0DFEC\ 2EED25B8 \\
&\quad 557EAC9C\ 80E2E198\ F8CDBECD\ 86B12053 \\
y &= 03676854\ FE24141C\ B98FE6D4\ B20D02B4\ 516FF702 \\
&\quad 350EDDB0\ 826779C8\ 13F0DF45\ BE8112F4 \\
n &= 3FFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF \\
&\quad FFFFEF90\ 399660FC\ 938A9016\ 5B042A7C\ EFADB307 \\
h &= 02
\end{aligned}
$$

The scalar multiplication $(u, v) = dG$ is

$$
\begin{aligned}
d &= 45B7876\ FC01A2EA\ 920FF4E4\ 789CF04B\ E39E9302 \\
&\quad 1D486C8\ CF0EC27F\ AB882021\ AF8E5BEA \\
u &= 0038294\ A063535C\ 2CC0758B\ 111D0026\ F67A5918 \\
&\quad 2E71FF1C\ A2675EDC\ C4DA7538\ 708B55A2 \\
v &= 140389B\ F2FFBE5C\ DB3B84A\ 9BE08A3E\ 6933A6CA \\
&\quad 64CC084B\ 1E8A54D8\ DCF09C9B\ 8E87D8BE
\end{aligned}
$$

Bibliography

[1] B. Kaliski, TWIRL and RSA Key Size, RSA Laboratories Technical Note, May, 2003, http://www.rsasecurity.com/rsalabs/technotes/twirl.html.

[2] SEC 1, Elliptic Curve Cryptography: Standards for Efficient Cryptography Group, 2000, http://www.secg.org.

[3] W. Stallings, *Cryptography and Network Security*, Prentice Hall, NJ, 2003.

[4] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[5] B. Schneier, *Applied Cryptography*, John Wiley & Sons, NY, 1996.

[6] N. Ferguson and B. Schneier, *Practical Cryptography*, Wiley, Indianapolis, Indiana, 2003.

[7] NIST, Recommended Elliptic Curves for Federal Government Use, 1999, http://csrc.nist.gov/csrc/fedstandards.html.

[8] American Bankers Association, ANSI X9.62-1998: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1998.

[9] IEEE P1363 Committee, Standards Specification for Public Key Cryptography, 1998, http://grouper.ieee.org/groups/1363/.

[10] S. Schwiderski-Grosche *et al.*, Security Challenges in the Personal Distributed Environment, in *60th Vehicular Technology Conference, VTC Fall '04*, volume 5, pp. 3267–3270, IEEE, 2004.

[11] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, Reconfigurable computing: architectures and design methods, *IEE Proceedings Computers and Digital Techniques* **152**, 193 (2005).

[12] T. Wollinger and C. Paar, How Secure are FPGAs in Cryptographic Applications?, Cryptology ePrint Archive, Report 2003/119, 2003, http://eprint.iacr.org/.

[13] B. Kasim and L. Ertaul, GSM Security., in *ICWN'05: International Conference on Wireless Networks*, pp. 555–561, CSREA Press, 2005.

[14] G. Gaubatz, J.-P. Kaps, E. Ozturk, and B. Sunar, State of the Art in Ultra-Low Power Public Key Cryptography for Wireless Sensor Networks, in *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, pp. 146–150, Washington, DC, USA, 2005, IEEE Computer Society.

[15] K. Lauter, The advantages of Elliptic Curve Cryptography for Wireless Security, *IEEE Wireless Communications* , 62 (2004).

[16] R. Zuccherato, Using a PKI based upon Elliptic Curve Cryptography, Entrust white paper, 2003, http://www.entrust.com/resources.

[17] N. Gura, S. C. Shantz, H. Eberle, S. Gupta, and V. Gupta, An End to End Systems Approach to Elliptic Curve Cryptography, in *Proc. of CHES 2002*, volume 2523, pp. 349–365, Springer, 2002.

[18] G. Orlando and C. Paar, A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m), in *Proc. of the Second International Workshop on Cryptographic Hardware and Embedded Systems, CHES'2000*, volume 1965 of *Lecture Notes in Computer Science*, pp. 41–56, Worcester, MA, 2000, Springer.

[19] T. Kerins, E. Popovici, W. Marnane, and P. Fitzpatrick, Fully Parameterizable Elliptic Curve Cryptography Processor over GF(2^m), in

Proc. of 12th International Conference on Field Programmable Logic and Application, FPL'2002, volume 2438 of *Lecture Notes in Computer Science*, pp. 750–759, Montpellier, France, 2002, Springer.

[20] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, Reconfigurable Implementation of Elliptic Curve Crypto Algorithms, in *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pp. 284–291, Washington, DC, USA, 2002, IEEE Computer Society.

[21] R. Cheung, N. Telle, W. Luk, and P. Cheung, Customizable Elliptic Curve Cryptosystems, *IEEE Trans. on VLSI Systems* **13**, 1048 (2005).

[22] J. Lutz and A. Hasan, High Performance FPGA based Elliptic Curve Cryptographic Co-Processor, in *ITCC'04: International Conference on Information Technology: Coding and Computing*, volume 2, pp. 486–492, IEEE Society Press, 2004.

[23] A. Satoh and K. Takano, A Scalable Dual-Field Elliptic Curve Cryptographic Processor, *Transactions on Computers* **52**, 449 (2003).

[24] H. Eberle *et al.*, A Public-Key Cryptographic Processor for RSA and ECC, in *ASAP'04: 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 98–110, 2004.

[25] R. Dahab and J. López, An Overview of Elliptic Curve Cryptography, Technical Report IC-00-10, State University of Campinas, Brazil, 2000.

[26] S. T. J. Fenn, M. Benaissa, and D. Taylor, GF(2m) Multiplication and Division Over the Dual Basis, *IEEE Trans. Comput.* **45**, 319 (1996).

[27] N. Koblitz, Elliptic Curve Cryptosystems, *Mathematics of Computation* **48**, 203 (1987).

[28] V. Miller, Use of Elliptic Curves in Cryptography, in *Proc. of Advances in Cryptology, CRYPTO'85*, pp. 417–426, Santa Barbara, CA, 1985.

[29] J. Pollar, Monte Carlo Methods for Index Computation mod p, *Mathematics of Computation* **32**, 918 (1978).

[30] NIST, FIPS 180-2: Secure Hash Standard (SHS), 2002, http://csrc.nist.gov/publications/fips/.

[31] D. Hankerson, L. López, and A. Menezes, Software Implementation of Elliptic Curve Cryptography Over Binary Fields, in *Proc. of the Second International Workshop on Cryptographic Hardware and Embedded Systems, CHES'2000*, volume 1965 of *Lecture Notes in Computer Science*, pp. 1–24, Worcester, MA, August 2000, Springer.

[32] J. López and R. Dahab, Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n, in *Proc. of Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pp. 201–212, Springer, 1998.

[33] J. López and R. Dahab, Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation, in *Proc. of the First International Workshop on Cryptographic Hardware and Embedded Systems, CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pp. 316–327, Berlin, 1999, Springer.

[34] M. Aydos, T. Yantk, and C. Koc, A high-speed ECC-based wireless authentication on an ARM microprocessor, in *16th Annual Computer Security Applications Conference (ACSAC'00)*, pp. 401–410, Los Alamitos, CA, USA, 2000, IEEE Computer Society.

[35] J. Grobschadl and G.-A. Kamendje, Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields GF(2^m), in *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468, IEEE Computer Society Press, 2003.

[36] J. Krasner, Using Elliptic Curve Cryptography (ECC) for Enhanced Embedded Security, Certicom white paper, 2004, http:///www.techonline.com/community/ed_resource/tech_paper/37933.

[37] A. Weimerskirch, D. Stebila, and S. C. Shantz, Generic GF(2^m) Arithmetic in Software and its Application to ECC, in *Proc. of 8th Australasian Conference on Information Security and Privacy (ACISP 2003)*,

volume 2727 of *Lecture Notes in Computer Science*, pp. 79–92, Wollongong, Australia, July 2003, Springer.

[38] M. Ernest, S. Klupsch, O. Hauck, and S. Huss, Rapid Prototyping for Hardware Accelerated Elliptic Curve Public Key Cryptosystems, in *Proc. of 12th IEEE Workshop on Rapid System Prototyping, RSP'2001*, pp. 24–31, Monterey, CA, 2001.

[39] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blumel, A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n), in *Proc. of the 4th International Workshop on Cryptographic Hardware and Embedded Systems - CHES'2002*, volume 2523 of *Lecture Notes in Computer Science*, pp. 381–399, Redwood Shores, CA, 2002, Springer.

[40] N. Saquib, F. Rodriguez, and A. Diaz, A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over GF(2^n), in *Proc. of 11th Reconfigurable Architectures Workshop, RAW'04*, pp. 26–27, Sta. Fe, USA, 2004.

[41] P. Leong and K. Leung, A Microcoded Elliptic Curve Processor Using FPGA Technology, *IEEE Trans. on VLSI Systems* **10**, 550 (2002).

[42] N. Mentens, S. Berna, and B. Preneel, An FPGA Implementation of an Elliptic Curve Processor GF(2^m), in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp. 454–457, Boston, MA, 2004.

[43] P. C. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, *Lecture Notes in Computer Science* **1109**, 104 (1996).

[44] Wikipedia, Side channel attack — Wikipedia, The Free Encyclopedia, 2007, [Online; accessed 27-December-2007].

[45] B. Chevallier-Mames, M. Ciet, and M. Joye, Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity, *IEEE Transactions on Computers* **53**, 760 (2004).

[46] J.-S. Coron, Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems, in *Cryptographic Hardware and Embedded Systems*, number Generators, pp. 292–302, 1999.

[47] K. Compton, Reconfigurable Computing: A Survey of Systems and Software, *ACM Computing Surveys* **34**, 171 (2002).

[48] F. Barat and R. Lauwereins, Reconfigurable Instruction Set Processors: A Survey, in *Proc. of the 11th International Workshop on Rapid System Prototyping (RSP'00)*, pp. 168–173, 2000.

[49] D. A. Buel, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, Wiley-IEEE Computer Society Press, 1996.

[50] R. A. Keaney, C. H. Lee, D. J. Skellern, J. Vuillemin, and M. Shand, Implementation of Long Constraint Length Viterbi Decoders using Programmable Active Memories, in *11th Australian Microelectronics Conference*, pp. 52–57, 1993.

[51] P. M. Athanas and H. F. Silverman, Processor reconfiguration through instruction-set metamorphosis, *Computer* **26**, 11 (1993).

[52] A. Lawrence, A. Kay, W. Luk, T. Nomura, and I. Page, Using Reconfigurable Hardware to Speed up Product Development and Performance, in *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, pp. 111–118, London, UK, 1995, Springer-Verlag.

[53] J. R. Hauser and J. Wawrzynek, Garp: a MIPS processor with a reconfigurable coprocessor, in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, p. 12, Washington, DC, USA, 1997, IEEE Computer Society.

[54] E. Sanchez, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, A. Perez-Uribe, and M. Sipper, Static and Dynamic Configurable Systems, *IEEE Trans. Comput.* **48**, 556 (1999).

[55] M. Wirthlin, B. Hutchings, and K. Gilson, The Nano Processor: a low resource reconfigurable processor, in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 23–30, 1994.

[56] M. J. Wirthlin and B. L. Hutchings, DISC: the dynamic instruction set computer, in *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607*, edited

by J. Schewel, pp. 92–103, Bellingham, WA, 1995, SPIE – The International Society for Optical Engineering.

[57] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, Design and Implementation of the MorphoSys Reconfigurable ComputingProcessor, *J. VLSI Signal Process. Syst.* **24**, 147 (2000).

[58] R. Wittig and P. Chow, OneChip: An FPGA Processor with Reconfigurable Logic, in *IEEE Symposium on FPGAs for Custom Computing Machines*, edited by K. L. Pocek and J. Arnold, pp. 126–135, Los Alamitos, CA, 1996, IEEE Computer Society Press.

[59] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, The chimaera reconfigurable functional unit, *IEEE Trans. Very Large Scale Integr. Syst.* **12**, 206 (2004).

[60] M. Dales, Initial Analysis of the Proteus Architecture, in *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pp. 623–627, London, UK, 2001, Springer-Verlag.

[61] B. Zeidman, *Designing with FPGAs and CPLDs*, CMP Books, 2002.

[62] W. Wolf, *FPGA-Based System Design*, Prentice Hall, 2002.

[63] N. H. E.Weste and K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, Addison-Wesley, 1993.

[64] J. Wolkerstorfer, *Hardware Aspects of Elliptic Curve Cryptography*, PhD thesis, Graz University of Technology, 2004.

[65] IEEE 1076-2002, IEEE Standard VHDL Language Reference Manual, 2002.

[66] Xilinx Inc., Two Flows for Partial Reconfiguration: Module Based or Difference Based, Application Note 290, XAPP290., 2004, www.xilinx.com.

[67] Wikipedia, Formal verification — Wikipedia, The Free Encyclopedia, 2008, [Online; accessed 28-July-2008].

[68] J. López and R. Dahab, Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation, in *Proc. of the First International Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, pp. 316–327, Springer, 1999.

[69] S. Kumar and C. Paar, Low-Cost Elliptic Curve Digital Signature Coprocessor for Smart Cards, in *Workshop on RFID Security 2006*, Graz, Austria, 2006.

[70] S. C. Shantz, From Euclid's GCD to Montgomery Multiplication to the Great Divide, Technical Report TR-2001-95, Sun Microsystems Laboratories, 2001.

[71] G. M. de Dormale and J.-J. Quisquater, Iterative Modular Division over $GF(2^m)$: Novel Algorithm and Implementations on FPGA, in *International Workshop on Applied Reconfigurable Computing (ARC2006)*, edited by K. Bertels, J. Cardoso, and S. Vassiliadis, pp. 370–382, Springer, 2006.

[72] L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede, Balanced point operations for side-channel protection of elliptic curve cryptography, in *IEE Proceedings of Information Security*, volume 152, pp. 57–65, 2005.

[73] T. Izu and T. Takagi, A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks, in *PKC '02: Proceedings of the 5th International Workshop on Practice and Theory in Public Key Cryptosystems*, pp. 280–296, London, UK, 2002, Springer-Verlag.

[74] Xilinx Inc., OPB IPIF (v3.01a), Data Sheet, ds414., 2004, www.xilinx.com.

[75] Xilinx Inc., Virtex4 Family Overview, Data Sheet, DS112 (v2.0)., 2007, www.xilinx.com.

[76] N. A. Saqib, *Efficient Implementation of Cryptographic Algorithms on Reconfigurable Hardware Devices*, PhD thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2004.

[77] Xilinx Inc., Early Access Partial Reconfiguration., User Guide, UG208., 2006, www.xilinx.com.

[78] M. Rosing, *Implementing Elliptic Curve Cryptography*, Manning Publications, Greenwich, CT, USA, 1999.

[79] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede, Superscalar Coprocessor for High-speed Curve-based Cryptography, in *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249, pp. 415–429, Springer-Verlag, 2006.

[80] Xilinx Inc., ML401/ML402/ML403 Evaluation Platform, User Guide, UG080., 2006, www.xilinx.com.

[81] Xilinx Inc., EDK 8.2 PowerPC Tutorial in Virtex-4, , WT001 (v4.0)., 2006, www.xilinx.com.

[82] P. Panjwani and Y. Poeluev, Additional ECC Groups For IKE, 1999, IPSecWorking Group, INTERNET-DRAFT.