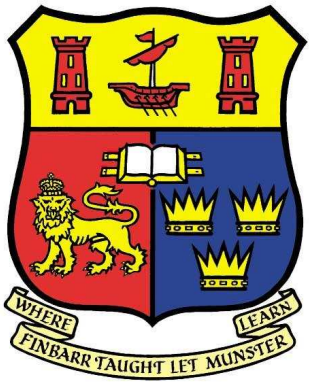
Cryptographic Coprocessors for Embedded Systems



Mark Hamilton

Department of Electrical and Electronic Engineering

National University of Ireland, Cork

Research Supervisor : Dr. William P. Marnane
Head of Department : Prof. Nabeel Riza

A thesis submitted for the degree of

Doctor of Philosophy

January 24, 2014

Abstract

In the field of embedded systems design, coprocessors play an important role as a component to increase performance. Many embedded systems are built around a small *General Purpose Processor* (GPP). If the GPP cannot meet the performance requirements for a certain operation, a coprocessor can be included in the design. The GPP can then offload the computationally intensive operation to the coprocessor; thus increasing the performance of the overall system. A common application of coprocessors is the acceleration of cryptographic algorithms. The work presented in this thesis discusses coprocessor architectures for various cryptographic algorithms that are found in many cryptographic protocols. Their performance is then analysed on a *Field Programmable Gate Array* (FPGA) platform.

Firstly, the acceleration of *Elliptic Curve Cryptography* (ECC) algorithms is investigated through the use of instruction set extension of a GPP. The performance of these algorithms in a full hardware implementation is then investigated, and an architecture for the acceleration the ECC based digital signature algorithm is developed.

Hash functions are also an important component of a cryptographic system. The FPGA implementation of recent hash function designs from the SHA-3 competition are discussed and a fair comparison methodology for hash functions presented.

Many cryptographic protocols involve the generation of random data, for keys or nonces. This requires a *True Random Number Generator* (TRNG) to be present in the system. Various TRNG designs are discussed and a secure implementation, including post-processing and failure detection, is introduced.

Finally, a coprocessor for the acceleration of operations at the protocol level will be discussed, where, a novel aspect of the design is the secure method in which private-key data is handled.

I, Mark Hamilton, certify that this thesis is my own work and I have not obtained a degree in this university or elsewhere on the basis of the work submitted in this thesis.

Mark Hamilton

Acknowledgements

I would like to thank all those who have helped me during the course of completing this thesis, including, but not limited to, my supervisor Liam Marnane for giving me the opportunity to pursue this PhD, and for his guidance over the past few years; Christophe Negre and Emanuel Popovici for taking the time to read this thesis; the staff of the electrical engineering department in UCC; all of the postgraduate students and postdocs that I have worked with over the past few years; Arnaud Tisserand for inviting me to collaborate with him in Lannion; and my family for their support over the last few years.

Contents

Chapter 1

Introduction

1.1 Motivation

The emergence of ubiquitous computing has led to increasing amounts of data being transmitted over a wide range of media; ranging from fiber optic links over distances of hundreds of miles, to wireless transmissions over several centimetres. The transmission of sensitive information is no longer exclusive to large businesses using expensive and computationally powerful equipment. Today, small embedded devices, such as smart cards and mobile phones, also require the ability to transmit data securely. These small devices provide a completely different design challenge than that of large high-speed applications. When implementing cryptographic protocols on embedded devices, a designer must take into consideration the computational power, logic resources available in the device, and power consumption. A trade-off must be made in these areas in order to achieve acceptable performance in terms of computation time, while also minimising the area and power consumption.

Cryptographic protocols can be used to ensure that data is transmitted securely over an unsecured channel. The increase in transmission of financial and other sensitive information across the Internet has led to the definition of many cryptographic standards. Currently, one of the most widely supported cryptographic standards, for use on the Internet, is the *Transport Layer Security* (TLS) protocol. The TLS protocol allows for secure communication over *Virtual Private Networks* (VPNs); is used extensively in securing online financial transactions and also plays an important role in embedded applications, such as wireless sensor networks. However, embedded devices tend to be very constrained in terms of computing power, as many of them are battery powered and do not require a powerful processor to perform their primary task. This

can be problematic as the TLS protocol supports a wide array of encryption and key exchange algorithms, many of which are very computationally intensive as they include large finite field multiplications. These types of computations are not suited to small *General-Purpose Processors* (GPPs) found in embedded devices and can lead to unacceptably poor performance. To alleviate this problem a coprocessor can be used to accelerate cryptographic computations and allow for GPPs to be used for other tasks.

Field Programmable Gate Arrays (FPGAs) are a popular choice for embedded systems as they have a faster time to market than *Application Specific Integrated Circuit* (ASIC) based solutions, and are also more flexible than a design based around a microprocessor *Integrated Circuit* (IC). An FPGA consists of a large array of user programmable *Lookup Tables* (LUTs), memory elements, and routing logic; allowing a designer to implement any logic required for the system, inside the FPGA. This has the advantage that a customised system can be built, without the need to manufacture new hardware components. Another benefit of FPGAs is their reconfigurability, which allows for hardware updates to be made without replacing components in the system. The configurability of FPGAs and the abundance of *Intellectual Property* (IP) cores, that can perform a wide array of tasks, makes it possible to construct a *System on Chip* (SoC) consisting of a microprocessor and also some extra logic, for whatever application is required, all inside the same chip.

The advantage of FPGAs for cryptographic applications is that they are much more suited to processing data in parallel than a GPP. A small processor usually has a datapath in the range of 8 to 32 bits and an *Arithmetic Logic Unit* (ALU) capable of performing operations on data of the same size. This structure works well when processing general data, such as checking message fields in packets of information received over a network. However, an FPGA can be configured in such a way that it can process large data structures in parallel, which allows for the implementation of a cryptographic processor that consists of a datapath of several hundred bits, and also an ALU capable of processing data of the same size. A combination of a GPP and a cryptographic coprocessor can reduce the cost and power consumption of the embedded device, while increasing the performance.

When designing cryptographic applications for an embedded system, security is an important factor. The security needs are very different from that of large systems that are intended for use in a fixed location, such as a web or mail server. In server applications the hardware will usually be physically protected to a much higher degree than in an embedded device. This can include the servers being kept inside a secure room. An embedded device however may have very little, to no physical protection.

This allows the attacker to mount different forms of attacks against the device, such as monitoring the power consumption of the device in order to retrieve secret key information; this form of side channel attack is known as *Simple Power Analysis* (SPA). An attacker might also try to influence how the device generates random numbers, or how the device processes data, possibly compromising the security of cryptographic algorithms running on the device. With the extent of physical access an attacker has, it is critically important to protect the device in some way from these attacks. One of the easiest ways an attacker can recover the secret keys is by attempting some form of software based attack, through which the secret keys would be extracted from the GPPs internal working registers. These sort of attacks can be prevented by segmenting the system into a secure and non-secure areas. The GPP would be placed in the non-secure zone and would not have access to the secret keys, thus removing the option of a software based attack.

1.2 Contribution

In this thesis, a secure architecture for a cryptographic coprocessor will be presented. The goal of the design is to derive an architecture that securely manages the private keys and is resistant against side channel attacks. Firstly, the requirements of a co-processor for the TLS protocol will be analysed; specific operations will be chosen for implementation in the coprocessor, with the aim of improving overall system performance and security. Much of the previous work in the area has focused on very large and high speed designs. In contrast, the architecture presented in this work will focus on embedded applications, where an efficient implementation of all operations is critical. Having identified the components required for the implementation of a TLS coprocessor, a thorough analysis of each of the components will be conducted in order to derive efficient and secure architectures.

Side channel resistance will be built into the system at an architecture and algorithmic level. The use of secure logic styles such as dual rail logic has been avoided in order keep the design portable across a range of platforms. It is not possible to fully protect a device from attack but the goal is to reduce the number of attacks that are feasible and to increase both the computational power and time required to extract the private keys.

The remainder of this thesis is organised as follows: Chapter 2 introduces the background information and theory required for the remaining chapters. Chapter 3 introduces *Elliptic Curve Cryptography* (ECC) theory and its implementation in a

hardware-software co-design setting. The acceleration of ECC algorithms through the use of *Instruction Set Extension* (ISE), and the design of finite field multipliers for this purpose will also be examined. Chapter 4 builds on the results from the previous chapter and explores the design of ECC coprocessors constructed from FPGA resources. Chapter 5 discusses the implementation of hash functions on FPGAs and introduces a fair comparison methodology for different hash function structures. Chapter 6 introduces *True Random Number Generators* (TRNGs) and how they can be implemented on an FPGA platform. Post-processing and failure detection of TRNGs will also be investigated; thus, allowing the construction of a secure TRNG suitable for use as a coprocessor. Chapter 7 details the design of the TLS coprocessor, which incorporates the components discussed in the previous chapters. The final coprocessor is a novel architecture that incorporates secure key management and securely partitions TLS operations between software and hardware.

1.3 Publications

The following papers were published during the course of the research conducted for this thesis:

- Brian Baldwin, Andrew Byrne, Mark Hamilton, Neil Hanley, Robert P. McEvoy, Weibo Pan, and William P. Marnane. FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. In *Euromicro Conference on Digital System Design (DSD)*, pages 783–790, August 2009.

- Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. A Hardware Wrapper for the SHA-3 Hash Algorithms. In *Signals and Systems Conference (ISSC 2010), IET Irish*, pages 1–6. IET, 2010.

- Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. In *International Conference on Field Programmable Logic and Applications (FPL 2010)*, pages 400–407. IEEE, 2010.

- Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. In *The Second SHA-3 Candidate Conference*, August 2010.

- Brian Baldwin, Raveen R. Goundar, Mark Hamilton, and William P. Marnane. Co-*Z* ECC Scalar Multiplications for Hardware, Software and Hardware-Software Co-

Design on Embedded Systems. *Journal of Cryptographic Engineering*, 2(4):221–240, 2012.

- Mark Hamilton and William P. Marnane. FPGA implementation of an Elliptic Curve Processor using the GLV Method. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 249–254, 2009.

- Mark Hamilton, William P. Marnane, and Arnaud Tisserand. A Comparison on FPGA of Modular Multipliers Suitable for Elliptic Curve Cryptography over GF(p) for Specific p values. In *21st International Conference on Field Programmable Logic and Applications (FPL)*, pages 273–276, 2011.

Chapter 2

Cryptography for Embedded Systems

2.1 Introduction

An embedded system is a subsection of a larger system that is designed to perform a specific task. Examples include the *Global Positioning System* (GPS) transceiver in a mobile phone or the traction control system of a car. An embedded system contains at least one processor in its architecture and might also contain several other hardware modules, as shown in Figure 2.1, where μP denotes a microprocessor. The embedded system architecture consists of software running on the μP, which interfaces with the various hardware modules in order to perform its specified task.

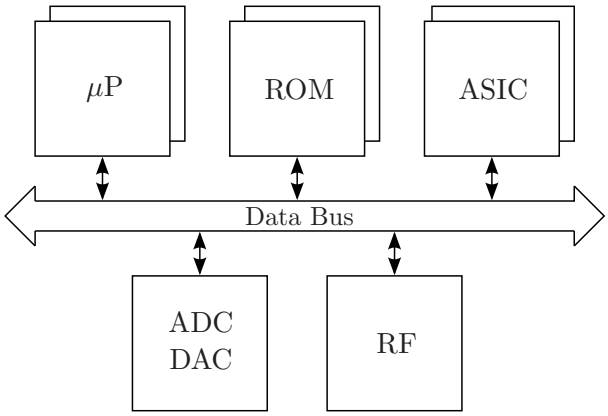


Figure 2.1: Generic architecture of an embedded system.

Embedded devices are usually more constrained in terms of computing power than general purpose devices, such as a desktop computer, as they are optimised to perform one specific task. When adding support for cryptographic protocols, one can therefore not take a generic library of code, compile it for the embedded system, and expect to achieve the required performance. The designer must take into consideration the resources available and optimise the implementation of the cryptographic primitives accordingly. In certain situations, the embedded system may simply not contain the requisite hardware to achieve acceptable performance. The inclusion of a coprocessor can solve this problem.

This chapter introduces the basic cryptographic algorithms and the mathematical concepts that they are based on. It will then be shown how these algorithms are combined into a cryptographic protocol that can be used to secure communications across an unsecured channel. Finally, an outline of the security requirements of embedded systems today and how FPGAs can be used to solve some of the problems that arise when implementing cryptographic algorithms in an embedded environment will be discussed.

2.2 Introduction to Cryptographic Systems

Figure 2.2 shows how the layers of a typical cryptographic system are arranged; the top level layer is where user applications reside. Secure email and mobile commerce are two examples of applications that are highly dependent on the services that the lower layers provide. Many different cryptographic algorithms exist that supply the security protocols with services such as sender authenticity, encryption, non-repudiation, and data integrity. Using a combination of these algorithms, it is possible to transfer data securely over an unsecured channel.

At the lowest layer of the system the core operations are based on finite field arithmetic. The efficient implementation of these operations is very important for overall system performance, as some of these operations need to be executed thousands of times during the processing of an algorithm. A small saving at this level can therefore lead to a large reduction in latency in the upper layers [7].

Cryptographic algorithms generally fall in two main categories, private-key cryptography and public-key cryptography. In private-key cryptography, two entities use an identical shared secret to communicate across an unsecured channel; this form of cryptosystem is discussed in Section 2.5. This approach, however, assumes that both parties have been able to establish a shared secret between them. In practice, this process is not a trivial task. However, the invention of public-key cryptography has provided a

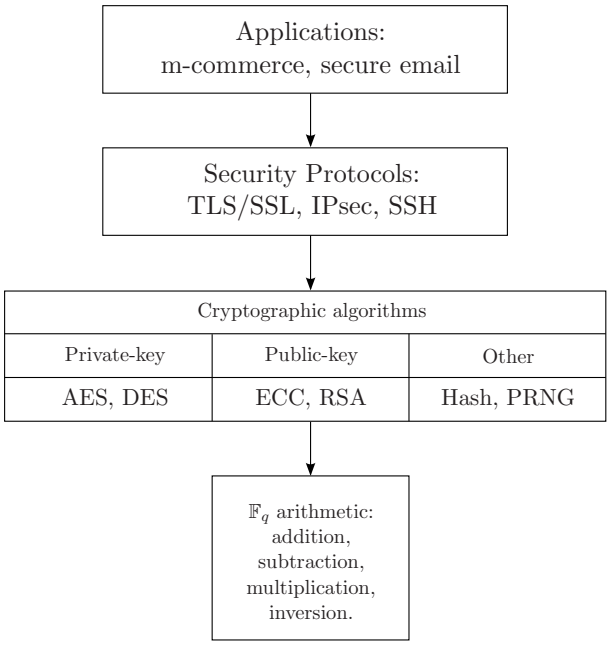


Figure 2.2: Hierarchical model of a cryptographic system.

solution by defining mathematical constructions that allow both keys and encrypted messages to be exchanged securely. Public-key algorithms are generally the most computationally intensive in the system as they require the most arithmetic operations per bit during their execution. Public-key cryptography will be discussed in Section 2.6, but first an introduction to some of the mathematics involved in cryptography will be given.

2.3 Mathematical Background

Many cryptographic primitives are based on the principles of finite field arithmetic; therefore, in this section the construction of finite fields and their associated properties will be discussed. In order to define a finite field, the general concepts of groups, rings, and fields must first be introduced.

2.3.1 Groups

A group is a set of elements G together with an operation "$\cdot$" which when applied to elements of G, the following properties hold:

1. The group operation "$\cdot$" is closed. That is, $a \cdot b = c \in G$ for all $a, b \in G$.

2. The group operation is associative. That is, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in G$.

3. There is an identity element (or neutral element) $\mu \in G$, where $a \cdot \mu = \mu \cdot a = a$ for all $a \in G$.

4. For each $a \in G$ there exists an element $a^{-1} \in G$, where $a \cdot a^{-1} = a^{-1} \cdot a = \mu$. The element a^{-1} is known as the inverse of a.

5. A group G can be referred to as abelian (or commutative) if the property $a \cdot b = b \cdot a$, for all $a, b \in G$, is satisfied.

2.3.2 Rings

A ring $(R, +, \cdot)$ is a set R, together with two binary operations "+" and "$\cdot$", where:

1. R forms an abelian group with respect to "+", where "+" is usually referred to as addition.

2. The associative property holds for the the "$\cdot$" operation. i.e., $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in R$.

3. The distributivity law holds i.e., $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$.

2.3.3 Fields

If we refer to "+" as addition and "$\cdot$" as multiplication, a field $\mathbb{F}$ is a ring where the multiplication is commutative and a multiplicative inverse exists for every nonzero element of $\mathbb{F}$. The properties of a field are therefore:

1. The elements of $\mathbb{F}$ form an additive group with the group operation "+" for which the identity element is 0.

2. All elements of $\mathbb{F}$ except for 0 form a multiplicative group with respect to the group operation "$\cdot$" and corresponding identity element 1.

3. The distributivity law holds for both multiplication, addition, and their combination. i.e., for all $a, b, c \in \mathbb{F}, a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

2.3.4 Finite Fields

Finite fields are of great importance to cryptography, as many cryptographic primitives can be defined using arithmetic over finite fields. In particular ECC, which will be discussed in Chapter 3, and *Rivest-Shamir-Adleman* (RSA) [106] are two examples of algorithms that are derived from this area of mathematics.

A finite field is simply a field $\mathbb{F}$ that has a finite number of elements. It has been shown that a finite field can only exist if the number of elements in the field (also known as the *order* of the field) is the power of a prime p. That is, given a prime p and a positive integer m a finite field has $q = p^m$ elements and is denoted $\mathbb{F}_q$. This result was derived by the mathematician Évariste Galois and hence a finite field is sometimes referred to as a Galois field, with the notation $GF(q)$.

Having introduced some of the mathematical principles used in cryptography, the following sections address how these principles are used in cryptographic algorithms.

2.4 Generating Random Numbers

As will be shown in subsequent sections, many cryptographic algorithms require the generation of random data for keys or nonces. Therefore, in this section a brief introduction into the generation of random data will be presented.

Regardless of how mathematically secure a cryptographic algorithm is, it can still be defeated if its implementation is not secure. This has been shown to be true in the past where cryptosystems were broken due to poorly initialised random data [45, 67]. In this case, an attacker can predict the keys; therefore, breaking the cipher itself is not necessary.

In order for a *Random Number Generator* (RNG) to be secure, random data should be generated in such a way that it's unpredictable, even if an attacker has knowledge of all previously generated data and the physical implementation of the RNG. Ideally, the RNG should produce random data uniformly distributed in the required range and each random bit should be independent. This amounts to the RNG having good statistical properties. In order to achieve this, the RNG must sample some unpredictable source to generate a stream of random bits. The more unpredictable the source, the more entropy will be present in the output bitstream; where entropy is a measure of the uncertainty of a random bit. The quality of the RNG is determined by its ability to extract randomness from the unpredictable source. RNG designs vary depending on the platform of implementation, a detailed discussion on implementing RNGs on FPGAs

will be given in Chapter 6. In the following sections it will be assumed that the keys and nonces have been generated in a secure manner.

2.5 Private-Key Cryptography

A private-key (also known as symmetric-key) cryptosystem consists of two entities, say Alice and Bob, that both have knowledge of a secret key k. If Alice wants to send a message to Bob, Alice encrypts the message using the key k, and some encryption function ϵ, where ϵ is generally a stream cipher or a block cipher. Both stream ciphers and block ciphers are used to encrypt a string of data; stream ciphers encrypt the data one bit at a time, updating the key for each new bit of the plaintext. Block ciphers on the other hand fragment the data in blocks of fixed length and encrypt each block with the same key.

The encryption function ϵ operates on the message and the key to produce the ciphertext c, that is $c = \epsilon_k(m)$. An example of this type of cryptosystem is shown in Figure 2.3, where δ is the corresponding decryption function for ϵ. The *National Institute of Standards and Technology* (NIST) block cipher currently recommended for use as the encryption algorithm is the *Advanced Encryption Standard* (AES) [91]. If the encryption function ϵ is cryptographically strong, it is possible to securely exchange messages without an eavesdropper being able to decipher them. An example of a private-key system is *Wi-Fi Protected Access* (WPA) [2], which uses pre-shared keys.

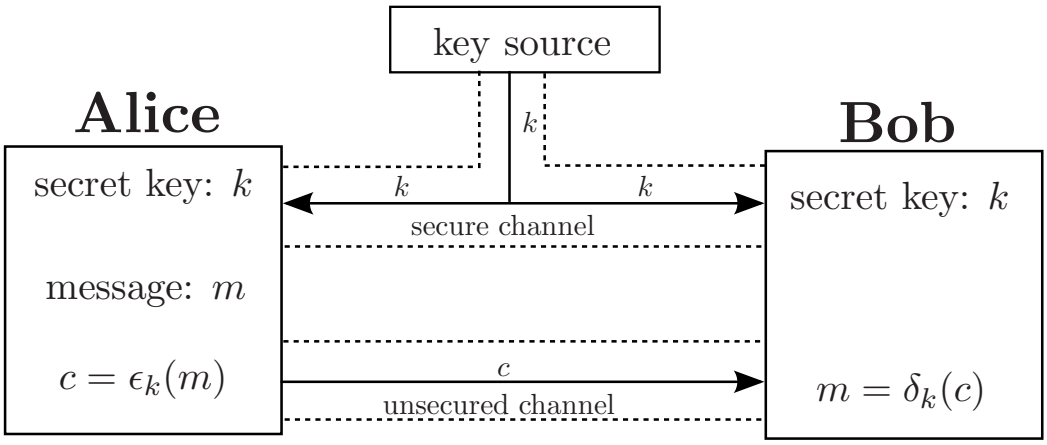


Figure 2.3: Private-key cryptosystem.

Private-key cryptosystems offer high throughput for communications, however, securely distributing the secret key between the two communicating entities is a problem

in certain situations. Over a large network, maintaining the key pairs for each connection is also problematic. Another downside is that private-key does not provide non-repudiation, which is of critical importance for online financial transactions. A solution to this can be found through the use of public-key cryptography for the distribution of keys. Once the secret key has been established, private-key algorithms can then be used for bulk data encryption. This combination of public-key and private-key systems combines the security and easy key distribution properties of public-key algorithms, with the high data throughput that is achievable with private-key algorithms.

2.6 Public-Key Cryptography

Public-key (also known as asymmetric-key) cryptography was introduced by Diffie and Hellman in 1976 in their paper "New Directions in Cryptography" [30]. Public-key cryptography provides a solution to setting up a shared secret key between two entities over an unsecured channel. The two communicating entities each have a key pair (K_{pub}, K_{priv}), where K_{pub} and K_{priv} are the public and private keys of that entity respectively. The keys are generated in such a way that if a message is encrypted using K_{pub}, it can be decrypted using K_{priv}. This allows for an easy key distribution scheme as K_{pub} can be made publicly available and anyone who wants to send an encrypted message only requires knowledge of the recipient's public key. In order for the protocol to be secure the key pair must be calculated in a way that makes it computationally infeasible to compute K_{priv}, given only K_{pub}.

Take as example two entities, Alice and Bob, that are connected by an unsecured channel, as shown in Figure 2.4. If Alice wants to send a message to Bob she first acquires a copy of Bob's public key, B_{pub}. Alice then encrypts the message m, using B_{pub} and some function ξ to obtain a ciphertext c, such that $c = \xi_{B_{pub}}(m)$. Alice then sends the ciphertext to Bob who can decrypt the ciphertext using his private key, B_{priv}, and a decryption function φ.

The main variants of public-key cryptography all stem from [30], they are ElGamal [34], RSA [106], and ECC [63, 81]. RSA uses the integer factorisation problem as its hard problem, whereas the others use the *Discrete Logarithm Problem* (DLP). The DLP is defined as the problem of finding an integer x such that $\alpha^x \equiv \beta \pmod{q}$, where $1 \leq x \leq q-1$, $\alpha, \beta \in G$, and G is a finite cyclic group.

Although public-key cryptography solves the secure key distribution problem, the system can easily be defeated by certain attacks without having to break the underlying mathematical principles. A *Man-in-the-middle* (MIM) attack [114, pages 48–49] can

Figure 2.4: Public-key cryptosystem.

be performed, where an attacker intercepts messages between the two communicating parties and impersonates each party by replacing their public keys with his own. To protect against this type of attack, some form of key authentication must be built into the system; this has lead to the definition of a *Public Key Infrastructure* (PKI).

2.6.1 Public Key Infrastructure

PKI is the term given to the system that incorporates *Certificate Authorities* (CAs), *Registration Authorities* (RAs), and various public-key algorithms to achieve an authenticated distribution of keys. The purpose of the PKI is to bind a user's identity to their public key, this introduces a level of trust into the system. Digital certificates, which will be discussed in Section 2.6.1.2, are the mechanism used to bind a user to their public key. For the PKI to be possible, there must be at some level a trusted source of certificates; this trusted source is known as the root CA. The root CA has the ability to verify other CAs by signing their certificates. This method of trust being passed down to the end user is known as a certificate chain, and is present as it allows many different organisations to issue certificates. This also results in a reduction in the amount of data that an end user would need to store if they were required to possess the public keys of all the CAs.

When a user receives a certificate they must be able to verify the signature of one of the CAs in the chain in order to ensure that the certificate is in fact legitimate. An important process in the distribution of certificates is the ability to digitally sign a piece of data.

2.6.1.1 Digital Signatures

Digital signatures provide a method by which the receiver of a message, that has been digitally signed, can verify the source of the message i.e., data origin authenticity.

Digital signatures are based on pubic-key methods and were introduced in [30]. Many other schemes have been proposed such as ElGamal [34, 106], and the elliptic curve based variant which will be discussed in Chapter 4.

In contrast to how the Diffie-Hellman key exchange works, by using an entity's public key to encrypt a message, digital signatures are produced by generating the signatures based on some operation and the sender's private key. The signature can then be verified through the use of the sender's public key. As an example the *Digital Signature Algorithm* (DSA) [89] is presented here. The sender generates a public key that consists of four parameters (p, q, α, β). The values p and q are primes, and are generated such that $q \mid (p-1)$. The value of α is given by $\alpha = g^{p-1/q}$, where g is chosen at random such that $\alpha \neq 1$ and $g \in [1, p-1]$. The value of β is given by $\beta = \alpha^d \pmod{p}$, where $d \in [1, q-1]$ is the sender's private key. $\mathcal{H}$ is a cryptographically secure hash function, which calculates a fixed length string for the input message m of arbitrary length. This string is effectively a fingerprint of the input message. A detailed description of the design and implementation of hash functions will be given in Chapter 5.

The signature generation process is given in Algorithm 1. In step 1 the value of k should be generated in a secure manner, such as that described in Section 2.4. A hash function $\mathcal{H}$ is used in step 2 to calculate a fixed length string based on the input message m. The hash of the input message, the value k, and the sender's private key d are then combined using some finite field arithmetic which results in a digital signature consisting of two elements r and s.

Algorithm 1 Digital signature generation

Input: private key d, public key (p, q, α, β), message m
Output: Signature (r, s)

1: generate random integer $k \in [1, q-1]$
2: compute $e = \mathcal{H}(m)$
3: compute $r \equiv (\alpha^k \pmod{p}) \pmod{q}$
4: compute $s \equiv (e + dr)k^{-1} \pmod{q}$
5: the signature for message m is then (r, s)

Algorithm 2 shows the steps required to verify the signature of a received message. Using the same hash function as that of Algorithm 1, the receiver calculates the hash of the received message. Steps 3 to 6 of the algorithm then use some finite field arithmetic to generate values based on r, s, $\mathcal{H}(m)$, and the sender's public key. If the signature is correct, the resulting value v should equal that of r from the received signature.

Algorithm 2 Digital signature verification

Input: signature (r, s), senders public key (p, q, α, β), message m
Output: accept or reject signature

1: verify $r, s \in [1, q-1]$
2: compute $e = \mathcal{H}(m)$
3: compute $\lambda = s^{-1} \pmod{q}$
4: compute $u_1 = e\lambda \pmod{q}$
5: compute $u_2 = r\lambda \pmod{q}$
6: compute $v \equiv (\alpha^{u_1}\beta^{u_2} \pmod{p}) \pmod{q}$
7: accept signature if $v \equiv r \pmod{q}$ else reject

This can be shown by observing that in Algorithm 1, the sender computes the value

$$s \equiv (e + dr)k^{-1} \pmod{q}, \tag{2.1}$$

where, $e = \mathcal{H}(m)$. Therefore, the correctness of Algorithm 2 can shown, as

$$\begin{aligned} k &\equiv es^{-1} + drs^{-1} \pmod{q}, \\ &\equiv e\lambda + dr\lambda \pmod{q}. \end{aligned} \tag{2.2}$$

Using the notation from Algorithm 2 this can be rewritten as

$$k \equiv u_1 + du_2 \pmod{q}. \tag{2.3}$$

Raising α to the power of both sides gives

$$\begin{aligned} \alpha^k &\equiv \alpha^{u_1+du_2} \pmod{p}, \\ &\equiv \alpha^{u_1}\alpha^{du_2} \pmod{p}, \\ &\equiv \alpha^{u_1}\beta^{u_2} \pmod{p}. \end{aligned} \tag{2.4}$$

Finally,

$$\begin{aligned} r &\equiv (\alpha^k \pmod{p}) \pmod{q}, \\ &\equiv (\alpha^{u_1}\beta^{u_2} \pmod{p}) \pmod{q}, \\ &\equiv v \pmod{q}. \end{aligned} \tag{2.5}$$

2.6.1.2 Digital Certificates

Digital certificates, standardised in [56], are used to link a specific public key to its user. A *Certificate Authority* (CA) is a registered organisation (trusted third party) that verifies the identities of the owners of public keys and issues certificates accordingly. To create a certificate, the sender uses a digital signature algorithm to sign a combination of the sender's ID and public key. Web browsers contain a list of trusted CAs. When a browser tries to access a website, the website sends the browser a certificate that it has obtained from a CA. If the certificate can be verified i.e., the signature is valid for the website domain name and the accompanying public key, the browser can infer that the website is genuine.

2.7 Cryptographic Protocols

In the previous sections, the underlying mathematical principles used in cryptography were introduced, this was followed by a discussion of the different forms of cryptographic algorithms. This section deals with the use of these cryptographic algorithms as part of a cryptographic protocol.

Cryptographic protocols define how the cryptographic algorithms are combined in order to achieve a specific security goal, such as secure data transmission over a network. They define how data should be encapsulated and give a framework for how the different forms of messages should be exchanged i.e., the exchange of keys or application data. Many cryptographic protocols exist, however, they are all based upon the same underlying cryptographic primitives, such as public-key and private-key cryptography. The *Internet Protocol Security* (IPsec) protocol [96] operates at the IP layer of the TCP/IP model [17, 18] and is used extensively in securing VPNs. *Secure Shell* (SSH) [135] operates at the application layer and allows for secure command-line login and file transfers over a network. Although IPsec and SSH are used for different applications and operate at a different layer of the TCP/IP model, they both use standard public-key and private-key algorithms to achieve their security goals.

One of the most common cryptographic protocols is TLS, which operates at the transport layer of the TCP/IP model. TLS is not application specific and is therefore widely used to establish secure data transmission. For this reason a detailed description of TLS will be given in the following sections. It should be noted however that TLS is just one application of the cryptographic algorithms that will be discussed throughout this thesis. The architectures may be applied to whatever cryptographic application is

required.

2.8 Transport Layer Security

The TLS protocol [29] supports a suite of cryptographic algorithms that can be used in many combinations to achieve the secure transfer of data. The fact that so many cryptographic algorithms are supported, with a varying degree of security levels, means that TLS is suitable for use on a wide range of devices. TLS is based around the private-key and public-key algorithms that were discussed in the previous sections. A detailed description of TLS is given in the following sections, as the processes and algorithms used determine the requirements of a TLS coprocessor.

The TLS protocol operates at the transport layer of the TCP/IP model. As an example, data may be received over a network through the use of the *Transmission Control Protocol* (TCP) [102]. This data is then passed to the TLS record protocol which is responsible for the fragmentation, compression, and encryption of data. The TLS record protocol encapsulates four higher level TLS protocols: the TLS handshake protocol, TLS application data protocol, TLS ChangeCipherSpec Protocol, and the TLS Alert protocol, as shown in Figure 2.5. The TLS handshake protocol is used for setting up a shared secret between two parties and is based around public-key algorithms, such as the Diffie-Hellman key exchange [30]. The TLS record protocol is used to send encrypted data between two communicating parties that have already established a shared secret in some way (possibly through use of the the TLS handshake protocol). The TLS record protocol is based around private-key algorithms, which use block ciphers or stream ciphers to encrypt and decrypt the messages, while digital signatures are used to provide message authenticity and non-repudiation.

The TLS protocol uses different message types to transfer application data or information related to the protocol itself; each message is referred to as a TLS record. Each record has its own header and may be encrypted and protected by a *Message Authentication Code* (MAC). A MAC is a tag that can be appended to a message and allows the receiver to verify the message's integrity and authenticity. A MAC algorithm can be constructed from a block cipher, in which case the algorithm is referred to as a *Cipher-based Message Authentication Code* (CMAC) [32], or more commonly they are constructed from hash functions and referred as a *Hash-based Message Authentication Code* (HMAC) algorithm [68]. The HMAC algorithm generates a MAC based on the hash of the message and the sender's public key; thus, combining the ability of hash functions to provide message integrity, with the ability of public-key algorithms

to provide message authenticity.

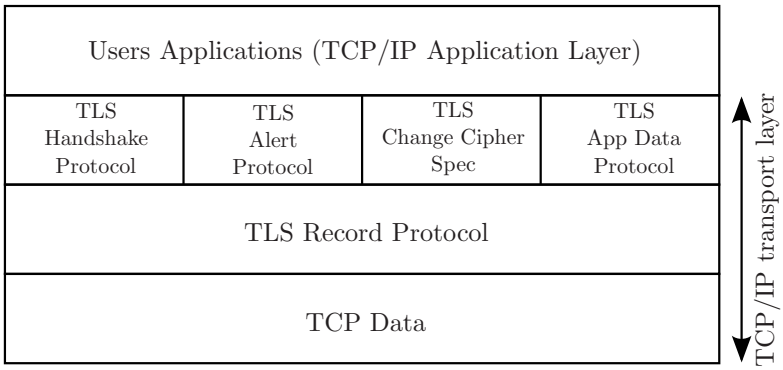


Figure 2.5: Processing of a TLS message fragment.

2.8.1 TLS Record Protocol

The TLS record protocol uses private-key algorithms to encrypt messages sent between the two communicating parties. Once the handshake phase of the TLS protocol has been completed, some method of encrypting messages must be used. Public-key based systems are generally too computationally intensive for this type of application. A more common choice is to use either a stream cipher or block cipher to perform the bulk encryption operation. AES [91] in *Cipher Block Chaining* (CBC) mode [31, 33] is a popular choice for this purpose and is the current NIST standard block cipher. The TLS record protocol uses HMAC to provide message integrity and message authenticity.

The process of encrypting a message begins by segmenting the data into blocks of 2^{14} bytes or less. Each fragment to be encrypted is referred to as a *TLSPlaintext* fragment. At this point there is the option to compress the fragment; a process that is only executed if it was negotiated as part of the agreed cipher suite. A MAC is then calculated for the fragment and appended before encryption takes place. A TLS record header is prepended onto the encrypted *TLSCiphertext*; the data segment is now ready to be transmitted, this process is illustrated in Figure 2.6.

2.8.2 TLS Alert Protocol

The TLS alert protocol allows each communicating party to signal that they have detected a problem with the session, or that they would like to close the session. The alert protocol supports two severity levels of alert messages: a warning message and a fatal error message. A *close notify* message is a type of alert message, with severity of

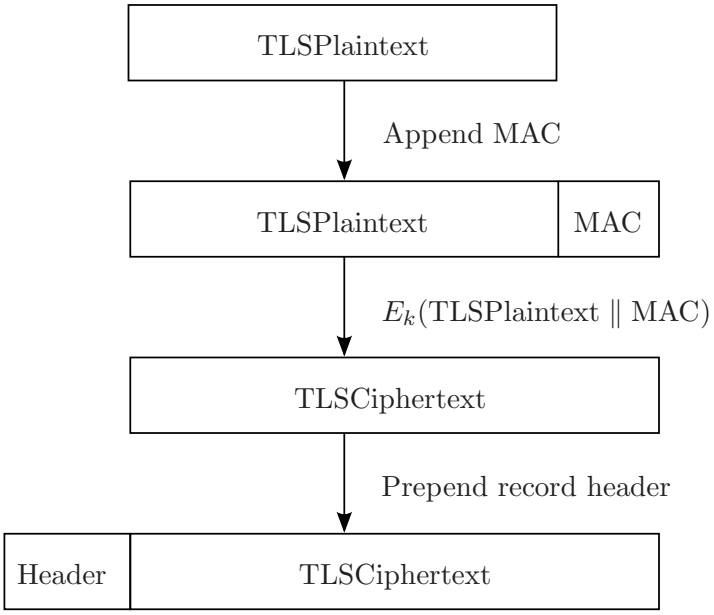


Figure 2.6: TLS protocol layers.

warning, that signifies that the sender is closing the connection. If either party detects an error in the TLS session, an alert message must be sent. If the error is considered fatal, the alert message is sent and the connection is immediately closed.

2.8.3 TLS ChangeCipherSpec Protocol

The TLS ChangeCipherSpec protocol is used to notify the receiving party that the sender will be changing ciphering strategies. During the handshaking phase of setting up a TLS session, the initial messages exchanged are unencrypted. Once a party receives a *ChangeCipherSpec* message it signifies that all of the sender's communications, from this point forward, will be protected by the previously negotiated bulk encryption and MAC algorithms.

2.8.4 TLS Application Data Protocol

The TLS application data protocol is simply the encrypted data carried by the TLS record protocol.

2.8.5 TLS Handshake Protocol

The TLS handshake protocol is used to establish a TLS session between two entities. The protocol supports many different algorithms for performing the negotiation of the shared secret. A typical handshake process, between a client and a server, is shown in Figure 2.7. The exact messages that are exchanged depend on the cipher suite that is negotiated by the first two *Hello* messages. Optional messages that are only sent by some cipher suites are shown with a "∗".

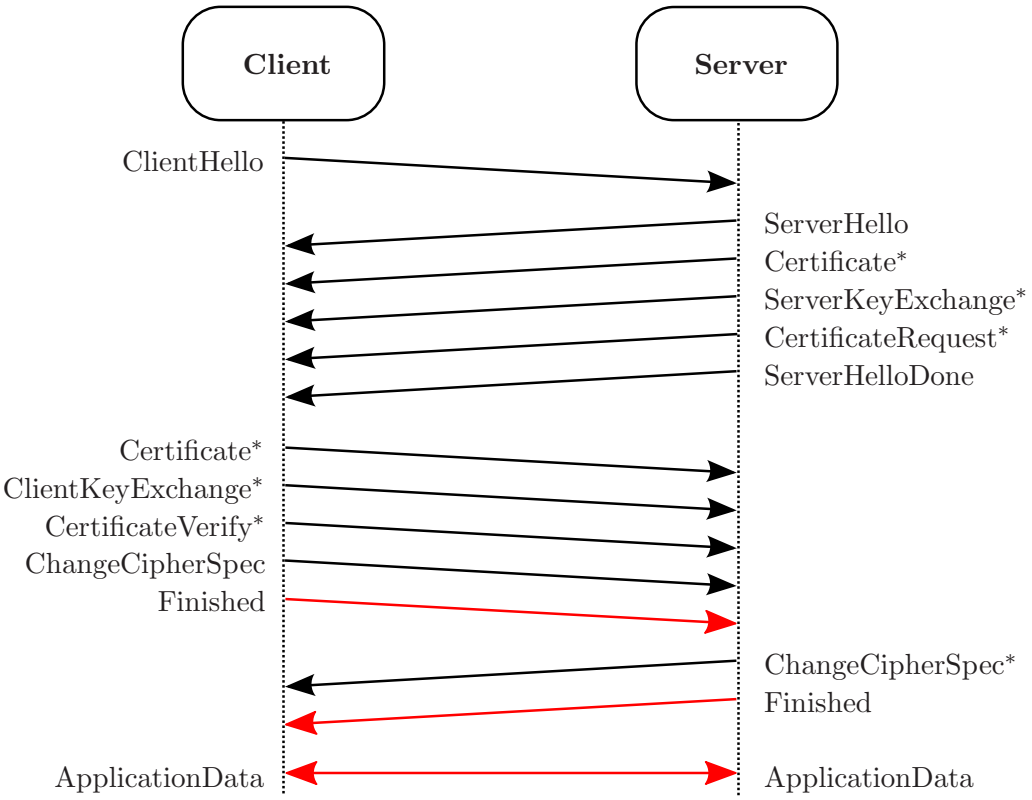


Figure 2.7: TLS handshake protocol.

The *ClientHello* message is sent by the client to notify the server of the cipher suites that it supports. Each cipher suite consists of a key exchange algorithm, a bulk encryption algorithm, a MAC algorithm and an algorithm to be used as the pseudorandom function. The client also generates a random number and sends it as part of the message. The *ServerHello* message is a response from the server that tells the client which security parameters it has accepted; the message also contains a random number generated by the server. These two messages also allow the client and server

to select which version of the TLS protocol is to be used. The descriptions that follow assume that the most recent version of the TLS protocol, TLSv1.2, is used.

The *ServerCertificate* message is sent if the algorithm parameters agreed upon support digital certificates for server side authentication. The certificate is an x.509 encoded data structure that contains the server's public key. Server side authentication is achieved through a trusted CA signing the certificate with the CA's private key.

The *ServerKeyExchange* message is only sent if the cipher suite that has been chosen allows for the negotiation of a shared secret based on a key that is different from the one contained in the server's certificate. This message is usually sent for cipher suites that make use of ephemeral keys. If ephemeral keys are not being used, the server's certificate should have contained all the information required to set up the *premaster secret*, and in this case the *ServerKeyExchange* message would not be sent. The *premaster secret* is used, along with the client and server random numbers, to derive the *master secret* and any keys required for the bulk encryption and HMAC functions.

The optional *CertificateRequest* message, sent by the server, is used if client side authentication is required by the negotiated cipher suite.

The *ServerHelloDone* message indicates that the server has sent all necessary data to the client in order for the client to continue with the key exchange process.

Once the client receives the *ServerHelloDone* message, it calculates its response and sends the data in the *ClientKeyExchange* message. The *ClientKeyExchange* message is always sent and contains the client's ephemeral public key. The client can, at this point, verify the certificate that has been sent by the server and generate the *master secret* based on the server's certificate or the data received in the *ServerKeyExchange* message, if it was sent. Once the server has processed the *ClientKeyExchange* message, both the server and client should have arrived at the same *master secret*.

The client then sends a *ChangeCipherSpec* message; this message notifies the server that all subsequent messages sent by the client will be encrypted with the agreed upon bulk encryption algorithm and authenticated with the HMAC algorithm.

This is then followed by the client's *Finished* message. This is the first encrypted message in the handshake process and contains a MAC and a hash of all previously exchanged handshake messages. Upon receiving the client's *Finished* message, the server can authenticate that the handshake was performed correctly by decrypting the *Finished* message and verifying that the MAC and the hash of handshake messages contained in the message matches the values that it has calculated.

The server then responds with its own *ChangeCipherSpec* and *Finished* messages.

Once the client has processed these messages, the handshake is complete and application data can then be exchanged through the use of the application data protocol.

2.9 Field Programmable Gate Arrays

Throughout this thesis, designs will be implemented and tested on reconfigurable logic devices known as FPGAs. Therefore, an introduction into their structure will be discussed in this section.

Field Programmable Gate Arrays (FPGAs) are a flexible platform that offer the ability to quickly test designs. Modern FPGAs contain not only large amounts of user programmable logic, but also dedicated hardware blocks that offer a high level of performance for specific applications, such as *Digital Signal Processing* (DSP) [134]. Some Xilinx Virtex 5 devices also contain PowerPC processors, Ethernet MACs, and high speed *Input/Output* (I/O) transceivers [131]. FPGAs consist of an array of *Configurable Logic Blocks* (CLBs) and routing logic that can be combined and configured to form complex combinational circuits. The CLBs and routing logic are arranged in columns throughout the FPGA chip. The internal structure of CLBs varies with different vendors and FPGA versions, however, all designs presented in this thesis are generated for members of the Xilinx Virtex 5 family of FPGAs [132]; hence, their structure will be discussed here.

In a Virtex 5 FPGA each CLB consists two slices, as shown in Figure 2.8. Each slice consists of four 6 input LUTs. The number of slices in a Virtex 5 FPGA ranges from 4,800 to 51,840, however, the version used for most of the results in this thesis is the XC5VLX110T, which has 17,280 slices. Each slice also incorporates fast carry chain logic which is designed to improve the performance of arithmetic circuitry. This is achieved by providing a dedicated path between slices for the carry signal, instead of having to route it through the standard routing logic. The carry chain connects slices in two vertically adjacent CLBs, as shown in Figure 2.8.

Each Virtex 5 FPGA also contains between 936 kbit and 18,576 kbit of *Block RAM* (BRAM) which is split into 36 kbit blocks. These blocks, however, can be combined in order to achieve larger memory sizes. BRAM resources are not part of the slice logic on the FPGA; therefore, they are presented as a separate result when used in any design in the following chapters.

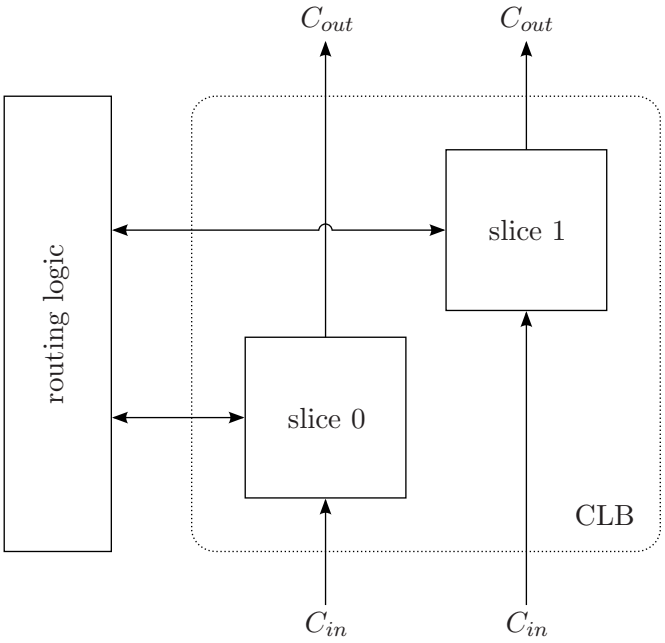


Figure 2.8: Virtex 5 CLB structure.

2.9.1 Microblaze Processor

Much of the work presented in this thesis concerns the design of coprocessors. In order to test the performance of these coprocessors in a real world environment, they must be implemented in a system where a GPP is also present. As the target platform for implementations are Xilinx devices, a Microblaze processor [130] is used as the GPP. A Microblaze is soft-core processor designed by Xilinx and is thus easily implemented on Xilinx devices. The Microblaze has a *Reduced Instruction Set Computing* (RISC) architecture and can be implemented using slice logic and BRAM resources on Xilinx FPGAs. The Microblaze can be configured to achieve different levels of performance by incorporating a floating point unit, dedicated multiply instruction, or a *Memory Management Unit* (MMU).

The test system used for many of the designs in the following chapters of this thesis is shown in Figure 2.9. The system is configured to include access to 256 MB of *Double Data Rate* (DDR2) *Random Access Memory* (RAM) through an external memory controller. The Microblaze also has access to a configurable amount of memory internal to the FPGA; this memory is implemented in BRAM. In some designs the system requires access to an Ethernet connection. This is done by utilising one of the embedded Ethernet *Media Access Controller* [133] modules present in the XC5VLX110T FPGA.

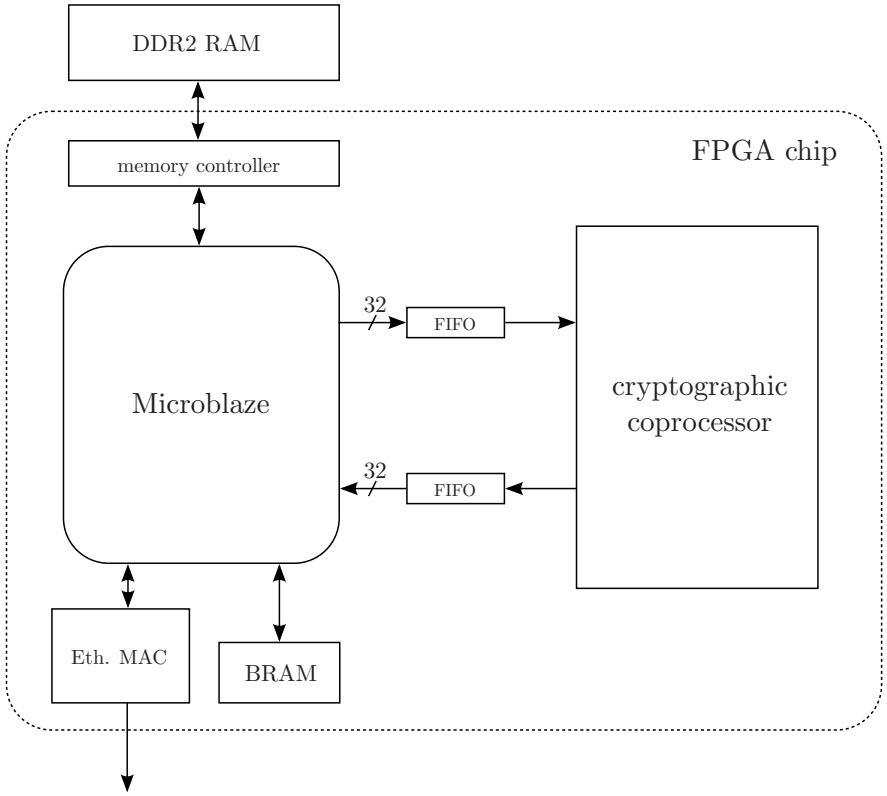


Figure 2.9: Example of an embedded SoC with a GPP and a coprocessor.

Each coprocessor is connected to the Microblaze via a *Fast Simplex Link* (FSL) bus.

2.9.2 FSL Bus

The FSL bus [129] is a high speed interconnect offered by Xilinx that allows the Microblaze processor to communicate with custom logic elsewhere in the FPGA. An FSL bus is unidirectional; hence, two FSL buses are required for two way communication with the coprocessor. The FSL bus consists of a *First In, First Out* (FIFO), of configurable depth, and 32 bits in width. The master side of the FSL bus controls the clock frequency, and writes data into the FIFO; while the slave side reads from the FIFO. An illustration of the FSL bus is shown in Figure 2.10. Data is sent from the master, into the FIFO, 32 bits at a time on the FSL_M_Data signal; the slave can then read from the FIFO. Valid data in the FIFO is indicated by the FSL_S_Exists signal being set high. A control signal, FSL_M_Control, is also present that is used to signal that a specific 32 bit block in the FIFO is a control word. The same setup is used in the

opposite direction, for the coprocessor to send data to the Microblaze.

The clock signal, FSL_Clk, is provided by the Microblaze; hence, the coprocessor must run at the same clock frequency as the Microblaze. The FSL bus used in the design presented in this work was configured to have a depth of 16, and to use LUT RAMs, as opposed to block RAMs. Each 32 bit data block takes 1 clock cycle to be written to the FSL FIFO.

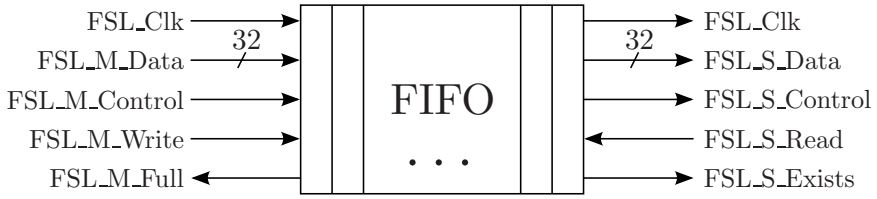


Figure 2.10: FSL bus.

2.10 FPGAs and Cryptography

FPGAs have been a popular choice for implementing cryptographic systems in recent years. This mainly stems from their suitability for implementing large parallel bitwise operations and lower development costs than ASIC based solutions. FPGAs also tend to have lower energy consumption per bit than *Central Processing Unit* (CPU)/*Graphics Processing Unit* (GPU) based designs [117], as they can be configured for a specific task; this is of particular interest to designers of embedded systems. An overview of the area of implementing cryptographic algorithms on FPGAs can be found in [107].

Embedded systems generally contain a small processor and some custom hardware to perform their task. Unfortunately, a small GPP is not the ideal device for implementing cryptographic algorithms. Cryptographic algorithms usually have operands with bitlengths in the range of 128 to 1024 bits, and above, whereas GPPs are designed to work with data of bitlengths in the range of 8 to 64 bits. Embedded systems processors usually fall into the lower end of this range. To execute operations on operands of this size, a GPP will have to decompose the operands until they are sufficiently small so that the processor's ALU can operate on them. This decomposition increases the number of clock cycles required to process the data, which thus increases the overall computation time and can result in a computation time of over a second on small devices [49]. Although this length of time might be acceptable for certain applications, FPGAs offer a solution for reducing the computation time.

FPGAs have the advantage that they can be configured to operate on the data in parallel. This makes them much more efficient than small GPPs for performing cryptographic operations. An added benefit is that an FPGA can be used to implement both the processor and the extra logic for performing cryptographic algorithms, all inside the same chip, as illustrated in Figure 2.9. This type of design is known as an SoC. FPGAs, however, have a limited amount of internal memory and for this reason, memory placed in an external chip is common practice. This can be done by implementing a memory controller inside the FPGA and having an external connection to a DDR RAM or *Static Random Access Memory* (SRAM) module.

This poses a problem when the processor has to perform operations on any secret data stored in the FPGA, as the processor will have to have access the secret keys or secret data in cleartext, in order for it to operate on it. As a result, unless precautions are taken, the secret data will have to be transferred to the main memory, external to the chip. An attacker could simply read this data as it's transferred from the FPGA to the memory module. There are several solutions to this problem:

1. The system can be designed so that all data sent to and from the memory module is encrypted. This solution, however, is not very efficient and will have a big impact on the overall performance of the system. This is caused by the overhead involved in encrypting data before transferring it to and from the memory module, and is inefficient as not all data being processed needs to be kept secure. This solution also increases the logic resources required on the FPGA and does not protect against software based attacks [52, 54, 94].

2. The processor can be implemented such that it has a section of working memory implemented internally to the FPGA and when secure data must be processed, all calculations use this secure internal memory. However, this solution does not protect against the possibility of software based attacks, as the processor still has access to the secret data.

3. The other solution, the one presented in this work, is to design the SoC such that the processor never has to operate on the secret key data. In order to achieve this, all operations requiring the secret key data can be done inside a coprocessor. This has several advantages over the two previous solutions. Firstly, a small GPP is not a very secure platform to implement cryptographic algorithms. If an attacker gains the ability to execute code on the processor, the attacker could simply retrieve the secret keys from memory. With this solution, however, software

based attacks are negated. Secondly, in this design the coprocessor can be used to increase the performance of the system and also add resistance against attackers extracting the secret keys, thus adding protection against hardware based attacks such as *Side Channel Attack* (SCA).

2.11 Side Channel Attacks

A secure cryptosystem relies not only on the theoretical security of cryptographic algorithms, but also on their secure implementation. In Section 2.4, the effect of a weak RNG implementation was discussed, where an adversary does not need to break the cryptographic algorithm if they can predict the secret keys that will be used. A generalisation of this approach can be applied to an entire cryptosystem and is referred to as an SCA. SCAs attempt to infer from physical emissions of the cryptosystem, sensitive information that the cryptosystem is operating on. This can be in the form of variations in the time it takes to perform a specific task [64], electromagnetic emissions [39], or the power consumption of the device [65].

SPA attacks attempt to determine, directly from the power consumption of the device, information about the data that is being processed. This is done by visually inspecting the waveform of the power consumption of the device and inferring from this which operation the device is performing at a given time. If an algorithm is designed in such a way that it executes different operations depending on the value of the secret key, an SPA attack could be used to recover each bit of the key. This can be the case for certain ECC algorithms; preventing this form of attack will be discussed in Chapter 3. *Differential Power Analysis* (DPA) attacks are a more sophisticated form of power analysis attack, where the waveform of the power consumption of the device is recorded over several executions (usually in the order of thousands) of the algorithm under attack. These waveforms are then statistically analysed, and the key recovered. A DPA attack that attempts to recover the secret key by analysing the power consumption of one specific point during the algorithm's execution is referred to as a first order attack. This approach can be extended to higher order attacks, where multiple points in the algorithms execution are simultaneously analysed in order to infer information about the secret key. Where possible, DPA and SPA attacks will be protected against in the designs presented in this thesis. A detailed description of side channel attacks can be found in [73].

2.12 Related Work

Very few hardware implementations that fully support acceleration of the TLS or *Secure Sockets Layer* (SSL) protocols have been published. Much of the currently published research has focused on accelerating specific algorithms supported by the TLS protocol suite, such as ECC arithmetic operations or private-key algorithms [43, 99, 100, 111]; there have also been several IPsec implementations [76, 93]. The following sections will examine the different types of SSL/TLS coprocessor architectures that have appeared in the literature. Coprocessors that accelerate specific algorithms will be discussed in their relevant chapter.

In the previous sections, some of the algorithms used in cryptographic protocols have been introduced. From the description of the TLS protocol, it can be seen that a number of different functions are required in order to implement the cryptographic portions of TLS, these are:

1. An encryption/decryption function, in the form of a stream cipher or block cipher.
2. A public-key processor that supports key exchange and digital signature algorithms.
3. A hash function, including HMAC capabilities.
4. A random number generator.

Therefore, the designs presented in the following sections each have a different structure, but contain some, or all, of the components above.

2.12.1 Isobe et al.

Several high speed designs have been published, such as that presented by Isobe et al. in [59], where the authors present a FPGA/ASIC based design of a SSL/TLS accelerator capable of reaching a throughput of 10 *Gigabits per second* (Gbit/s). The design supports RSA as the public-key algorithm, 128 bit and 256 bit AES and *Rivest Cipher 4* (RC4) [114, pages 397–398] for encryption, and *Message-Digest Algorithm 5* (MD5) [105] and *Secure Hash Algorithm 1* (SHA-1) for hashing operations. The design, shown in Figure 2.11, uses a bus topology for data transfers and consists of three sections: the protocol processing block is used to process TCP and SSL/TLS packets; the cipher processing block contains the cryptographic processors; and the

routing block is used to route data between them. The system has access to a network through an Ethernet MAC.

The authors show that the FPGA/ASIC design outperforms implementations run on a CPU or GPU. The design provides a higher throughput and reduces the power consumption by 80%. The final design occupied 13043 slices and 192 DSP blocks on the FPGA. The paper, however, does not describe in any detail how each hardware block is implemented, or exactly what type of FPGA was used. The authors state that it was an FPGA constructed using a 65 nm process. The lack of information makes it hard to make any comparison with this design, however, the work does show how there can be a dramatic saving in power consumption when custom hardware is used.

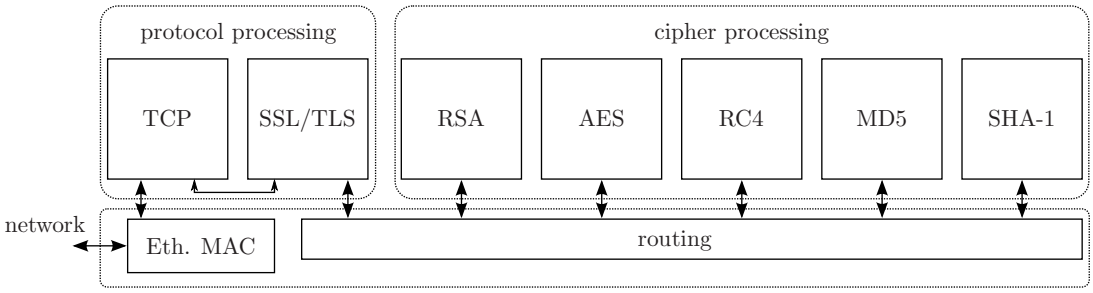


Figure 2.11: Isobe et al. design.

2.12.2 Wang et al.

In [125], the authors discuss a design for implementing a *Network Security Processor* (NSP) which alleviates the problem of bus contention that is present in a bus topology design, such as that shown in Section 2.12.1. Further work on the design can be found in [126, 127]. The processor is designed for use with both the IPsec and SSL protocols. The architecture supports a wide variety of cryptographic algorithms including 256 bit AES, DES/3DES [90], 1024 bit RSA, 256 bit ECC, HMAC, and a random number generator.

The architecture of the design is shown in Figure 2.12. The arithmetic engines are accessed through dedicated read DMA, write DMA and config DMA channels. The authors show that this type of memory access has a performance advantage over that of a standard bus topology, such as that used in [59]. The C*Core310 is a processor similar to an ARM core and is used to parse the SSL data that it receives from the network processor. The C*Core310 then sends a set of tasks, that need to be performed, to the central distributor which schedules the tasks in order to maximise the parallel

processing capabilities of the arithmetic engines.

The ECC unit used is based on the architecture presented in [23] which incorporates two ECC arithmetic engines in order to improve performance when processing two point multiplications in parallel during the signature verification process.

The proposed design is capable of processing 1600 SSL handshakes per second and achieves a throughput of just over 1 Gbit/s when transferring data in the SSL record protocol. The design was implemented on a Xilinx Spartan XC3S5000 FPGA and occupied 14,471 CLBs. The design was operated at a frequency of 150 *Megahertz* (MHz). At the time of writing, this design is currently the most complete SSL NSP that has been published.

2.12.3 Instruction Set Extension

The designs of Isobe et al. and Wang et al. both used a coprocessor style architecture. Another option for increasing the performance of a GPP is *Instruction Set Extension* (ISE). ISE is a method of increasing the performance of certain low level operations in a processor. A standard processor uses specific hardware blocks in its ALU to execute its supported instruction set. These instructions generally include addition, shift, rotate, and floating point multiplication operations. By adding extra hardware blocks into the processor's ALU, it's possible to increase performance for certain cryptographic operations. In [98], the authors include custom instructions to increase the performance of several block ciphers. As extra circuitry for only one specific operation is being added to the ALU, this type of architecture usually occupies a smaller area than that of a coprocessor based design, but has lower performance. This method can also have an impact on the critical path of the GPP, this can lead to reduced performance for all of the instructions that the processor executes, as the frequency at which it's clocked might have to be reduced. In a coprocessor architecture the coprocessor shares the same clock as the GPP, but does not add to its critical path. Further examples can be found in [35, 118]. An analysis of ISE for ECC algorithms will be discussed in Chapter 3, however, the Microblaze does not allow for custom instructions to be added to its ALU; therefore, a coprocessor style approach must be taken, where, the custom instruction is connected to the Microblaze via an FSL bus.

2.12.4 Secure Key Management

An important aspect of designing a secure SoC is securely managing the secret key data. In [40, 41], the authors introduced a method whereby the secret key data could

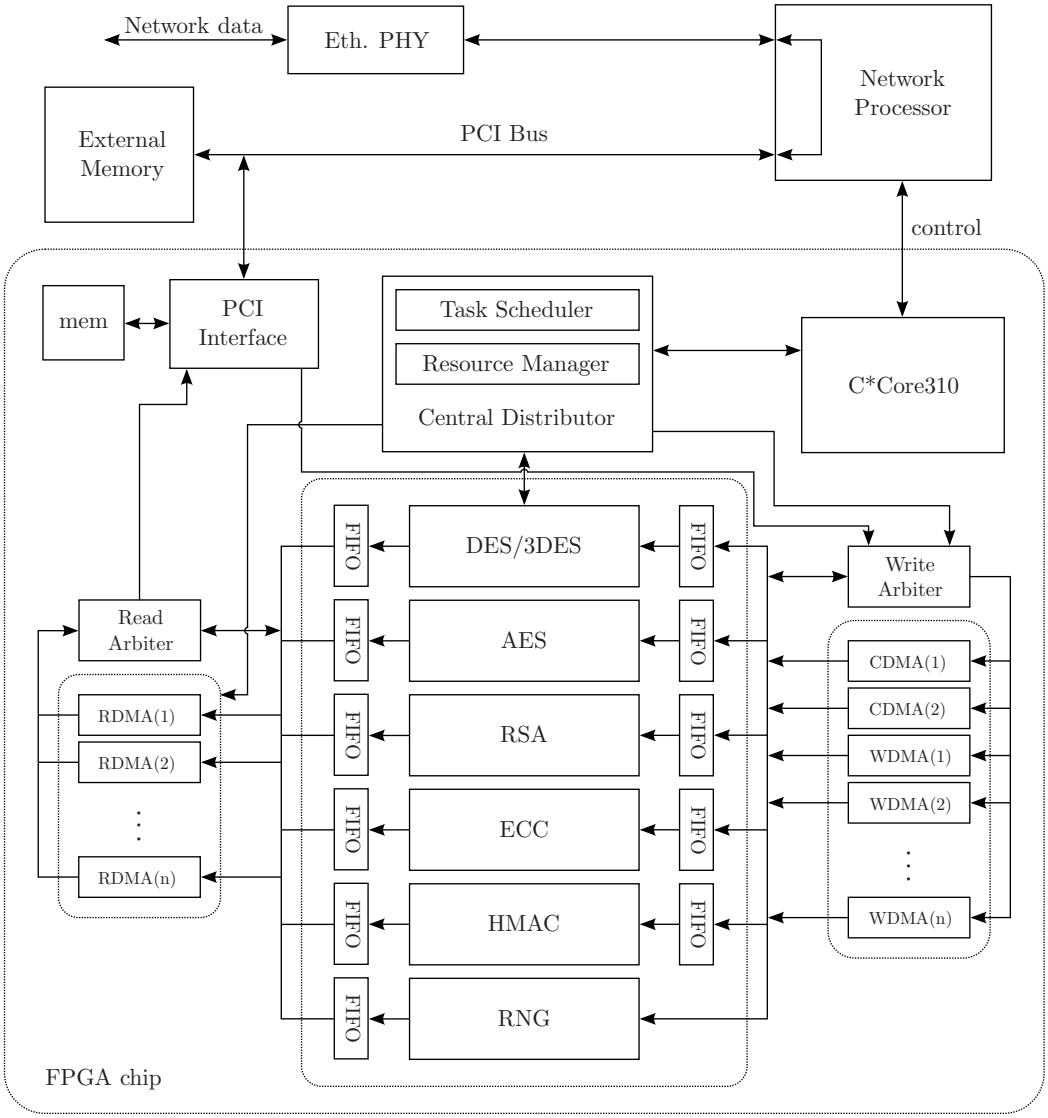


Figure 2.12: Wang et al. design.

be kept separate from the working memory in the GPP. The key data is kept in a physically isolated section of the FPGA, Figure 2.13. When the GPP's ALU is working on data that needs to be decrypted, the ciphertext is sent to the security module where it is decrypted using the secret key. The plaintext data can then be returned to the GPP for processing. The authors state that this form of implementation is suitable for protecting against software based attacks that attempt to recover secret key data from the internal registers or cache memory of the GPP, such as that in [51]. The design was implemented using a Microblaze processor [130] on a Xilinx Virtex 6 FPGA. The security module occupied 604 slices and 432 *kilobytes* (kBs) of BRAM.

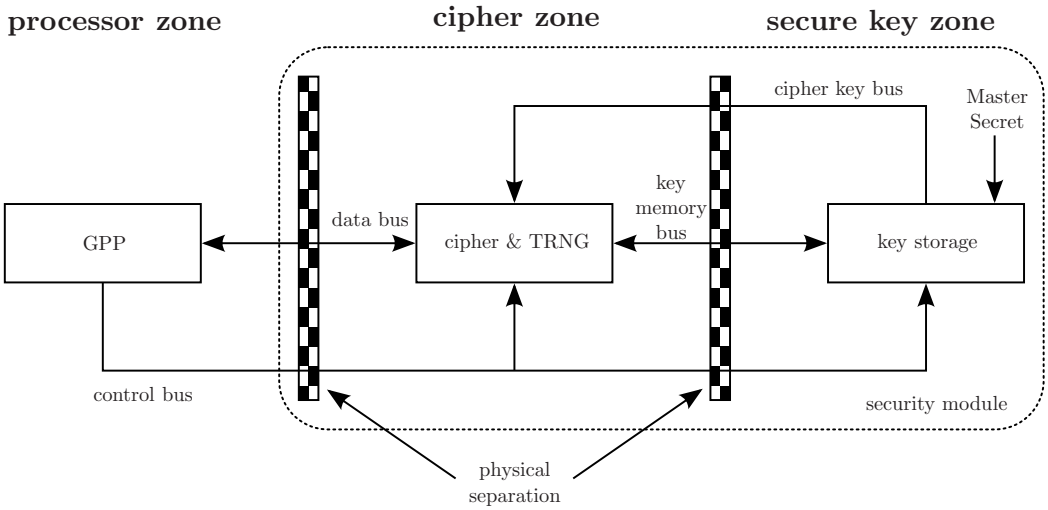


Figure 2.13: Gaspar et al. design.

2.13 Discussion

In this chapter, the basic components of the TLS protocol have been introduced. It has been shown how the various elements of private-key and public-key algorithms are used together to secure digital communications. All of these algorithms determine the requirements of a cryptographic coprocessor, of which several previously published designs have been discussed. These designs, however, do not deal with a secure implementation architecture, instead they tailor their architectures to achieve the best performance. The designs do show the advantages that can be gained by using a coprocessor for accelerating cryptographic functions.

This still leaves an area of work unexplored, which is the implementation of a TLS coprocessor in a secure architecture. The remainder of this thesis discusses architectures

for the various components of a TLS coprocessor. The goal of each architecture is to achieve a secure implementation, with the final coprocessor architecture built around the concept of the Gaspar et al. design, where the secret keys are isolated from the GPP.

In order to design a secure TLS architecture, the underlying components must first be examined. From the description of the TLS protocol presented in this chapter, it can be seen that a coprocessor would require the ability to perform encryption/decryption functions, message hashing operations, a TRNG for the generation of random data, and a processor for elliptic curve arithmetic.

Chapter 3

Hardware-Software Co-Design for Elliptic Curve Cryptography

3.1 Introduction

Since the introduction of ECC by Miller [81] and Koblitz [63], elliptic curve based public-key algorithms have been growing in popularity. This is mainly due to their relatively short key lengths, for the same security level, when compared to RSA [106]. ECC algorithms can be used as a key exchange mechanism and also as the basis of a digital signature scheme. ECC algorithms are therefore a good choice for use in the TLS handshake protocol.

Although ECC algorithms have shorter key lengths than RSA, they are still computationally intensive and usually account for the majority of computation time during the TLS handshake. In an embedded system, it can be advantageous to improve the performance of operations relating to ECC; therefore, reducing the TLS handshake time.

In this chapter, the mathematical principles behind ECC will be introduced, along with various ECC algorithms that have been suggested in order to reduce computation times [24, Section 13.2]. A general embedded system style setup, based around a GPP, will then be discussed and the ECC algorithms examined for performance in a purely software based environment. The impact of ISE on the performance of this system will then be explored; thus, analysing ECC in a hardware-software co-design setting.

3.2 Background to ECC

An elliptic curve defined over the finite field $\mathbb{F}_q$, $q > 3$, is the set of all pairs $(x, y) \in \mathbb{F}_q$ which satisfy the short Weierstraß equation,

$$E : y^2 \equiv x^3 + ax + b \pmod{q}, \tag{3.1}$$

where, $a, b \in \mathbb{F}_q$. The set also contains an identity element, or point at infinity, $\mathcal{O}$. If the condition

$$4a^3 + 27b^2 \not\equiv 0 \pmod{q}, \tag{3.2}$$

is satisfied, it ensures that the curve, defined by Equation 3.1, contains no singularities.

An elliptic curve must be defined over a finite field $\mathbb{F}_q$ for it to be of use in cryptography. The curve must also be non-singular i.e., it must not intersect itself at any point. All points on the elliptic curve, including the point at infinity, should form an abelian group, where $\mathcal{O}$ is the identity element; this set of points is denoted $E(\mathbb{F}_q)$. The number of points in E determines the set of all possible public keys that can be generated on that curve. The number of points on a curve is referred to as the order of E over $\mathbb{F}_q$ and is denoted by $\#E(\mathbb{F}_q)$. At most, a curve defined over $\mathbb{F}_q$ could have $2q + 1$ points. This accounts for every possible pair $P = (x, y)$, its negative, $-P = (x, -y)$, and the point at infinity $\mathcal{O}$. The Hasse interval gives a tighter bound for $\#E(\mathbb{F}_q)$ and is given by,

$$[q + 1 - 2\sqrt{q}, q + 1 + 2\sqrt{q}]. \tag{3.3}$$

The order of all NIST standardised prime field curves, is a large prime [89].

3.2.1 Group operations on Elliptic Curves

Let $E_{\mathcal{A}}$ be an elliptic curve defined over the finite field $\mathbb{F}_q$, where $q \neq 2, 3$ is a large prime and $\mathcal{O}$ is the point at infinity. The group operation on elliptic curves is addition. If two points, P_1 and P_2, on the curve are added together using the group operation, a third point on the curve results. The group operation can be derived from the geometric tangent and chord method, used to create a third point on an elliptic curve, if two points are already known [53, Section 3.1]. The point addition can then be performed using only integer additions, subtractions, multiplications, and divisions, modulo the prime q. The addition or doubling of two affine points on an elliptic curve is defined

as,

$$\begin{aligned}(x_3, y_3) &= (x_1, y_1) + (x_2, y_2),\\ x_3 &= w^2 - x_1 - x_2 \pmod q,\\ y_3 &= w(x_1 - x_3) - y_1 \pmod q,\end{aligned} \tag{3.4}$$

where

$$w = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \pmod q & \text{if } P \neq Q \text{ (point addition)} \\ \frac{3x_1^2 + a}{2y_1} \pmod q & \text{if } P = Q \text{ (point doubling).} \end{cases} \tag{3.5}$$

It can be seen from these formulæ that the computation of a point addition in affine coordinates requires 1 inversion (**I**), 3 multiplications (**M**), and 6 additions (**A**), while a point doubling requires $\{1\mathbf{I}, 4\mathbf{M}, 5\mathbf{A}\}$, assuming a finite field addition is computationally equivalent to a subtraction. In the finite field arithmetic of ECC algorithms, the most computationally intensive operation is modular inversion, followed by modular multiplication, and then modular addition/subtraction. As will be discussed in Section 3.3, ECC algorithms require the evaluation of point additions and doublings several hundred times during their execution. Therefore, other coordinate systems have been suggested that remove the need for the modular inversion in the point addition and doubling operations. These alternative coordinate systems are simply a different way of representing an elliptic curve point; hence, a mapping exists between these alternative coordinate systems and affine points. One example is Jacobian coordinates.

3.2.2 Jacobian Coordinates

Representing points on an elliptic curve as Jacobian projective coordinates removes the need to perform modular inversions during the point addition and doubling operations. The affine points are mapped to their Jacobian projective form, prior to the point addition or doubling, and can then be mapped back afterwards. When representing points on a curve in Jacobian coordinates, the curve equation is given by

$$E_{\mathcal{J}} : Y^2 = X^3 + aXZ^4 + bZ^6. \tag{3.6}$$

A point on the curve is given by $P(X_1, Y_1, Z_1)$ which corresponds to the affine point $P(\frac{X_1}{Z_1^2}, \frac{Y_1}{Z_1^3})$. The point at infinity is $\mathcal{O} = (1, 1, 0)$ and the negative of a point is $-P = (X_1, -Y_1, Z_1)$. Point additions and point doublings are performed as shown in

Algorithms 3 and 4.

Algorithm 3 Jacobian point addition.

Input: $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$, where $P, Q \in E_{\mathcal{J}}$
Output: $R = P + Q = (X_3, Y_3, Z_3)$

1: $A = X_1 Z_2^2$
2: $B = X_2 Z_1^2$
3: $C = Y_1 Z_2^3$
4: $D = Y_2 Z_1^3$
5: $E = B - A$
6: $F = D - C$
7: $X_3 = -E^3 - 2AE^2 + F$
8: $Y_3 = -CE^3 + F(AE^2 - X_3)$
9: $Z_3 = Z_1 Z_2 E$
10: **return** $R = (X_3, Y_3, Z_3)$

Algorithm 4 Jacobian point doubling.

Input: $P = (X_1, Y_1, Z_1)$, where $P \in E_{\mathcal{J}}$
Output: $R = 2P = (X_3, Y_3, Z_3)$

1: $A = 4X_1 Y_1^2$
2: $B = 3X_1^2 + aZ_1^4$
3: $X_3 = -2A + B^2$
4: $Y_3 = -8Y_1^4 + B(A - X_3)$
5: $Z_3 = 2Y_1 Z_1$
6: **return** $R = (X_3, Y_3, Z_3)$

A point addition using Jacobian coordinates requires $\{16\mathbf{M}, 7\mathbf{A}\}$, while the doubling operation requires $\{10\mathbf{M}, 13\mathbf{A}\}$. The point doubling in Jacobian coordinates can be simplified if the value a in the equation of the curve is -3. This is the case for NIST specified curves [89]. In the point doubling operation, the calculation of $B = 3X_1^2 + aZ_1^4$ can be replaced by $B = 3X_1^2 - 3Z_1^4 = 3(X_1 - Z_1^2)(X_1 + Z_1^2)$; thus reducing the computational cost of a point doubling to $\{8\mathbf{M}, 14\mathbf{A}\}$.

Many other projective coordinate systems exist [24, Chapter 13], most of which are designed to improve the performance on a GPP platform, where a software implementation is used. In the next section, some of the more recent proposals, where it was found that optimisations can be made when a point addition involves two points that share the same Z coordinate, will be discussed. The area is referred to as co-Z arithmetic.

3.2.3 Co-Z Arithmetic

In [77], Meloni analysed the operation of adding two different points, $P = (X_1 : Y_1 : Z)$ and $Q = (X_2 : Y_2 : Z)$, on $E_\jmath$ where both points share the same Z-coordinate. It was shown that their sum $P + Q = (X_3 : Y_3 : Z_3)$ can be evaluated faster using Algorithm 5.

Algorithm 5 Co-Z addition (ZADD).

Require: $P = (X_1, Y_1, Z)$ and $Q = (X_2, Y_2, Z)$
Ensure: $R \leftarrow \text{ZADD}(P, Q)$ where $R \leftarrow P + Q = (X_3, Y_3, Z_3)$

1: **function** ZADD(P, Q)
2: $C \leftarrow (X_1 - X_2)^2$
3: $W_1 \leftarrow X_1 C$
4: $W_2 \leftarrow X_2 C$
5: $D \leftarrow (Y_1 - Y_2)^2$
6: $A \leftarrow Y_1(W_1 - W_2)$
7: $X_3 \leftarrow D - W_1 - W_2$
8: $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A$
9: $Z_3 \leftarrow Z(X_1 - X_2)$
10: **return** $R = (X_3, Y_3, Z_3)$
11: **end function**

This operation is referred to as the ZADD operation. The computation of $R = P+Q$ then yields, for free, an equivalent representation for the input point P with its Z-coordinate equal to that of the output point R. This is given by

$$(X_1(X_1 - X_2)^2 : Y_1(X_1 - X_2)^3 : Z_3) = (W_1 : A : Z_3) \sim P. \tag{3.7}$$

The ZADD operation can then be extended to include this update of the Z co-ordinate. The corresponding algorithm, denoted ZADDU is given in Appendix A as Algorithm 24. The algorithm requires $\{7\mathbf{M}, 6\mathbf{A}\}$. This method of point addition, however, cannot be used with standard point scalar multiplication algorithms (point scalar multiplication algorithms will be discussed in Section 3.3).

In [47, 48] Goundar et al. introduced another co-Z operation referred to as *conjugate co-Z addition*, denoted ZADDC. This operation was introduced in order to allow for a more traditional point scalar multiplication algorithm to be used, without any loss of performance. The resulting algorithm is given as Algorithm 25 in Appendix A. The total cost for the ZADDC operation is $\{9\mathbf{M}, 15\mathbf{A}\}$.

Further work on the area of co-Z algorithms was presented by Hutter et al. in [57], where X-coordinate only formulæ for co-Z arithmetic were introduced. These

formulæ are referred to as *differential addition-and-doubling*, denoted AddDblCoZ, and are targeted at memory-constrained environments. The formulæ are defined for the homogeneous coordinate system, where the curve equation is given by

$$E_{\mathcal{H}} : Y^2Z = X^3 + aXZ^2 + bZ^3, \tag{3.8}$$

where, a point on $E_{\mathcal{H}}$, $P(X_1, Y_1, Z_1)$, corresponds to the affine point $P(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1})$. Take two points, $P = (X_1, Z)$ and $Q = (X_2, Z)$ on $E_{\mathcal{H}}$ that share the same Z-coordinate. The doubling and addition formulæ output a pair of points P' and Q' that also share the same Z-coordinate, where $Q' = 2Q = (X_2', Z')$ and $P' = P+Q = (X_1', Z')$ as shown in Algorithm 26. The computational cost of this formula is $\{17\mathbf{M}, 14\mathbf{A}\}$.

Hutter et al. also noticed that an improvement in computation time can be made by replacing the multiplication X_1X_2 with $(X_1^2 + X_2^2 - (X_1 - X_2)^2)/2$, followed by a multiplication by 2 later in the algorithm. The result is shown as Algorithm 27 in Appendix A, and requires $\{16\mathbf{M}, 14\mathbf{A}\}$.

Further optimisations can be made for cases where the curve parameters a and b are dynamic. Three additional coordinates are calculated at the initialisation stage, these are $T_a = aZ^2$, $T_b = 4bZ^3$ and $T_D = x_DZ$. The resulting algorithm is given in Appendix A as Algorithm 28. It can be seen that the performance can be increased by saving $1\mathbf{M}$ if $\mathbf{M}_a = \mathbf{M}_b = 1\mathbf{M}$. The computational cost is then $\{15\mathbf{M}, 13\mathbf{A}\}$.

3.3 Point Scalar Multiplication

In the previous section, several point addition and point doubling formulæ were discussed. The main operation of elliptic curve arithmetic used in cryptography is the multiplication of an integer, times an elliptic curve point. Calculating the product of an integer k, times a point on the curve P, is equivalent to the repeated addition of the point P to itself k times. This operation is known as point scalar multiplication and can be evaluated using the addition and doubling formulæ from the previous section.

$$R = kP = \underbrace{P + P + P... + P}_{k \text{ times}}. \tag{3.9}$$

The double and add algorithm, Algorithm 6, is one example of a point multiplication method. The binary value of the scalar k is right shifted by 1 bit for each iteration of the algorithm. The value of each bit k_i then determines which operations will be performed. If $k_i = 0$, a point doubling is performed; if $k_i = 1$, both a point doubling

and point addition are performed. After l iterations, the algorithm is complete.

Algorithm 6 Double and Add.

Input: $P \in E(\mathbb{F}_q); k = \sum_{i=0}^{l-1} k_i 2^i$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $R = \mathcal{O}$
2: **for** $i = l - 1$ down to 0 **do**
3: $\quad R = 2R;$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad R = R + P$
6: $\quad$ **end if**
7: **end for**
8: **return** R

3.3.1 SPA Resistant Point Scalar Multiplication

The double and add algorithm, Algorithm 6, is the standard point multiplication method, however, it is susceptible to SPA attacks, such as those described in Section 2.11. In the double and add algorithm, the operations performed in the main loop depend on each bit of the key; therefore, the difference in power consumption and length of computation time for each operation would allow an attacker to determine if a specific bit of the key is a 0 or a 1.

An alternative method for performing elliptic curve point multiplications is the Montgomery ladder [83]. A generalised version of the Montgomery ladder, shown in Algorithm 7, has a regular structure and is therefore resistant against SPA attacks.

Algorithm 7 Montgomery ladder.

Input: $R_0 \leftarrow \mathcal{O}; R_1 \leftarrow P; k = \sum_{i=0}^{l-1} k_i 2^i$
Output: $R = kP \in E(\mathbb{F}_q)$

1: **for** $i = l - 1$ down to 0 **do**
2: $\quad b \leftarrow k_i;\ R_{1-b} \leftarrow R_{1-b} + R_b$
3: $\quad R_b \leftarrow 2R_b$
4: **end for**
5: **return** $R = R_0$

The initialisation step of the Montgomery ladder sets $R_0 \leftarrow \mathcal{O}$ and $R_1 \leftarrow P$. The main loop then consists of two operations that are repeated l times, namely

$$R_{1-b} \leftarrow R_{1-b} + R_b, \tag{3.10}$$

$$R_b \leftarrow 2R_b. \tag{3.11}$$

If the condition that $k_{l-1} = 1$ is set, and the initialisation of R_0 and R_1 are replaced by $R_0 \leftarrow P$ and $R_1 \leftarrow 2P$, with both R_0 and R_1 sharing the same Z coordinate. The number of loop iterations can be reduced to $l - 1$ and the co-Z algorithms, ZADDC and ZADDU, can be used in the main loop. Algorithm 8 shows how this is achieved. The DBLU operation, given in Appendix A as Algorithm 35, is used to initialise R_0 and R_1 to the correct values.

Algorithm 8 Montgomery ladder with co-Z addition formulæ.

Input: $P = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_{l-1} = 1$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $(R_1, R_0) \leftarrow \text{DBLU}(P)$
2: **for** $i = l - 2$ down to 0 **do**
3: $\quad b \leftarrow k_i$
4: $\quad (R_{1-b}, R_b) \leftarrow \text{ZADDC}(R_b, R_{1-b})$
5: $\quad (R_b, R_{1-b}) \leftarrow \text{ZADDU}(R_{1-b}, R_b)$
6: **end for**
7: **return** $R = R_0$

An alternative form of the Montgomery ladder is shown in Algorithm 9, where the main loop consists of differential addition-and-doubling formulæ introduced by Hutter et al. in [57] and computations only involve the X and Z coordinates. The initialisation step of Hutter et al.'s algorithm (i.e., Algorithm 3 in [57]) has been replaced by $\text{DBLU}_{\mathcal{H}}$, given in Appendix A. The function *recoverfullcoordinates* recovers the full projective coordinates of the output point $R = kP$, from the X and Z coordinates of $R_0 = (X_1, Z)$ and $R_1 = (X_2, Z)$, on completion of the main loop of the algorithm.

All of the scalar multiplication algorithms that have been presented so far have scanned the bits of the scalar k from left-to-right. An alternative approach is to scan the scalar from right-to-left; this method is used in Joye's version of the double and add algorithm [60], shown in Algorithm 10. The algorithm has a similar structure to that of the Montgomery ladder and always repeats the same pattern of effective operations making it resistant to SPA attacks; a property that the original double and add algorithm does not have.

Algorithm 11 shows how Joye's double and add algorithm is implemented using co-

Algorithm 9 Montgomery ladder with (X,Z)-only co-Z addition formulæ.

Input: $P = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_{l-1} = 1$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $(X_1, X_2, Z) \leftarrow \mathrm{DBLU}_{\mathcal{H}}{}^*(P)$
2: **for** $i = l - 2$ down to 0 **do**
3: $\quad b \leftarrow k_i$
4: $\quad (X_{2-b}, X_{1+b}, Z) \leftarrow \mathrm{AddDblCoZ}(X_{2-b}, X_{1+b}, Z)$
5: **end for**
6: $R \leftarrow \mathrm{recoverfullcoordinates}(X_1, X_2, Z)$
7: **return** R

Algorithm 10 Joye's double-add.

Input: $P \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $R_0 \leftarrow O$; $R_1 \leftarrow P$
2: **for** $i = 0$ to $l - 1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad R_{1-b} \leftarrow 2R_{1-b} + R_b$
5: **end for**
6: **return** $R = R_0$

Z operations [47, 48]. The initialisation step of the algorithm requires a point tripling, which can be evaluated as $3P = P + 2P$ using co-Z arithmetic when $P = (X_1 : Y_1 : 1)$ [71]. The TPLU operation performs a point tripling and is used to initialise the points R_0 and R_1; a description can be found in Appendix A.3.

Algorithm 11 Joye's double-add algorithm with co-Z addition formulæ.

Input: $P = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $b \leftarrow k_1$; $(R_{1-b}, R_b) \leftarrow \mathrm{TPLU}(P)$
2: **for** $i = 2$ to $n - 1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad (R_b, R_{1-b}) \leftarrow \mathrm{ZADDU}(R_{1-b}, R_b)$
5: $\quad (R_{1-b}, R_b) \leftarrow \mathrm{ZADDC}(R_b, R_{1-b})$
6: **end for**
7: **return** $R = R_0$

3.3.1.1 Combined double-add operation

The calculation of $R = 2P + Q$ can be implemented in two steps; a point addition $T = P + Q$, followed by another point addition $R = P + T$. If P and Q have identical Z-coordinates, this operation can be performed through two consecutive applications of the ZADDU function, Algorithm 24, which would require $\{14\mathbf{M}, 12\mathbf{A}\}$. However, due to the structure of Joye's algorithm, these two applications of the ZADDU operation can be combined, resulting in the ZDAU function given in Appendix A. The ZDAU (*co-Z double-add with update*) operation only requires $\{16\mathbf{M}, 27\mathbf{A}\}$.

Joye's double and add algorithm can be rewritten using this new function, Algorithm 12. The cost per bit is then $\{16\mathbf{M}, 27\mathbf{A}\}$, instead of $\{16\mathbf{M}, 21\mathbf{A}\}$. However, in Algorithm 11, five of the multiplications are squaring operations and in Algorithm 12, seven of the multiplications are squarings. On some platforms a squaring operation can be performed faster than a standard multiplication, hence, the introduction of algorithms that attempt to trade multiplications for squarings. However, the implementations presented in this chapter use identical functions for squaring and multiplication.

Algorithm 12 Joye's double-add algorithm with co-Z addition formulæ (II).

Input: $P = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$
Output: $Q = kP \in E(\mathbb{F}_q)$

1: $b \leftarrow k_1$; $(R_{1-b}, R_b) \leftarrow \text{TPLU}(P)$
2: **for** $i = 2$ to $n - 1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad (R_{1-b}, R_b) \leftarrow \text{ZDAU}(R_{1-b}, R_b)$
5: **end for**
6: **return** $R = \text{Jac2aff}(R_0)$

3.3.1.2 (X, Y)-only operations

The co-Z Montgomery ladder can be rewritten so as to operate on only the X and Y coordinates of the input points. The operation ZACAU′[1] is the combination of the ZADDC′ operation followed by ZADDU′, and is given in Appendix A as Algorithm 30. This is used to obtain an (X, Y)-only implementation of the Montgomery ladder, Algorithm 13 [48, 123]. The computational cost of the algorithm is $\{14\mathbf{M}, 39\mathbf{A}\}$. The return function in Algorithm 13 costs $\{1\mathbf{I}, 9\mathbf{M}\}$ and returns the affine coordinates of the output point Q.

[1]The prime symbol ′ is used to denote operations that do not involve the Z-coordinate.

Algorithm 13 Montgomery ladder with (X, Y)-only co-Z addition formulæ.

Input: $P = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_{l-1} = 1$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $(R_1, R_0) \leftarrow \mathrm{DBLU}'(P)$
2: $C \leftarrow (\mathrm{X}(R_0) - \mathrm{X}(R_1))^2$
3: **for** $i = l - 2$ down to 1 **do**
4: $\quad b \leftarrow k_i$
5: $\quad (R_b, R_{1-b}, C) \leftarrow \mathrm{ZACAU}'(R_b, R_{1-b}, C)$
6: **end for**
7: $b \leftarrow k_0$; $(R_{1-b}, R_b) \leftarrow \mathrm{ZADDC}'(R_b, R_{1-b})$
8: $(x_P, y_P) \leftarrow P$
9: $Z \leftarrow x_P \,\mathrm{Y}(R_b)(\mathrm{X}(R_0) - \mathrm{X}(R_1))$; $\lambda \leftarrow y_P \,\mathrm{X}(R_b)$
10: $(R_b, R_{1-b}) \leftarrow \mathrm{ZADDU}'(R_{1-b}, R_b)$
11: **return** $R = \left(\left(\frac{\lambda}{Z}\right)^2 \mathrm{X}(R_0), \left(\frac{\lambda}{Z}\right)^3 \mathrm{Y}(R_0)\right)$

A left-to-right (X, Y)-only co-Z algorithm can be implemented by performing a ZADDU$'$ followed by a ZADDC$'$ to obtain an (X, Y)-only double-add operation with a co-Z update ZDAU$'$. The total cost of this operation is hence $\{14\mathbf{M}, 28\mathbf{A}\}$, while the cost for the final conversion is $\{1\mathbf{I}, 7\mathbf{M}\}$. The complete algorithm is shown in Algorithm 14 [48, 104].

Algorithm 14 Left-to-right signed-digit algorithm with (X, Y)-only co-Z addition formulæ.

Input: $P = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{l-1}, \ldots, k_0)_2 \in \mathbb{N}_{\geq 3}$ with $k_0 = k_{l-1} = 1$
Output: $R = kP \in E(\mathbb{F}_q)$

1: $(R_0, R_1) \leftarrow \mathrm{TPLU}'(P)$
2: **for** $i = l - 2$ down to 1 **do**
3: $\quad b \leftarrow k_i \oplus k_{i+1}$
4: $\quad R_1 \leftarrow (-1)^b R_1$
5: $\quad (R_0, R_1) \leftarrow \mathrm{ZDAU}'(R_0, R_1)$
6: **end for**
7: $R_1 \leftarrow (-1)^{1+k_1} R_1$
8: $(x_P, y_P) \leftarrow P$; $\lambda \leftarrow \frac{y_P \,\mathrm{X}(R_1)}{x_P \,\mathrm{Y}(R_1)}$
9: **return** $R = \left(\lambda^2 \,\mathrm{X}(R_0), \lambda^3 \,\mathrm{Y}(R_0)\right)$

Table 3.1 gives a summary of the scalar multiplication algorithms which have been presented, along with their corresponding co-Z functions. Each of these algorithms have been introduced in order to change the amount of memory or the number of multiplications required during the processing of the point scalar multiplication, at the

expense of extra precomputations or finite field additions. It would be expected that a reduction in the multiplications in the main loop of the point scalar multiplication algorithm would reduce the overall computation time. In order to investigate this, the performance of these algorithms will be compared in the following sections.

Algorithm	Main op.	Other op.
Left-to-right algorithms:		
Double and Add (Alg. 6)	ADD, DBL	
Montgomery ladder with co-Z addition (Alg. 8)	ZADDC, ZADDU	DBLU
Montgomery ladder with (X, Z)-only co-Z (Alg. 9)	AddDblCoZ	$\text{DBLU}_{\mathfrak{H}}{}^{*}$, recoverfullcoordinates
Montgomery ladder with (X, Y)-only co-Z (Alg. 13)	ZACAU′	DBLU′, ZADDC′, ZADDU′
Left-to-right signed-digit algorithm with (X, Y)-only co-Z (Alg. 14)	ZDAU′	TPLU′
Right-to-Left algorithms:		
Joye's double-add algorithm with co-Z I (Alg. 11)	ZADDU, ZADDC	TPLU
Joye's double-add algorithm with co-Z II (Alg. 12)	ZDAU	TPLU

Table 3.1: Operation usage for various co-Z addition formulæ.

3.4 Montgomery Multiplication

The point addition and doubling operations from the previous sections all require finite field multiplications. It is possible to use standard multiplication and modular reduction algorithms, however, in 1985, Montgomery introduced an algorithm to efficiently compute the modular product of two numbers [82]. The algorithm is very efficient when used for modular exponentiation as is does not require computationally intensive trial division operations by the modulus, q. Instead, the algorithm consists mainly of addition and shift operations. The Montgomery multiplier is one of the most widely used multiplier types in ECC processors. This is due to its relatively short critical path and the fact that it does not require a specific form of modulus.

Given a modulus q, of length l, let $\rho = 2^l$. The Montgomery algorithm requires that ρ and q be relatively prime i.e., $\gcd(\rho, q) = 1$. The Montgomery algorithm achieves an efficient multiplication through modifying the representation of the integers on which it operates. The inputs to the Montgomery algorithm must first be converted to their corresponding Montgomery representation. Given an integer $A < q$, its Montgomery representation is defined by

$$A' = A \cdot \rho \pmod{q}. \tag{3.12}$$

Given two integers in Montgomery representation the Montgomery algorithm computes

their product

$$R' = A' \cdot B' \cdot \rho^{-1} \pmod{q}, \tag{3.13}$$

where, $\rho \cdot \rho^{-1} = 1 \pmod{q}$. The result R' can be converted back to standard domain representation by Montgomery multiplying it by 1. Which can be shown to be true as

$$R = R' \cdot \rho^{-1} = R' \cdot 1 \cdot \rho^{-1} \pmod{q}. \tag{3.14}$$

In order to specify an algorithm for the computation of the Montgomery product, as in Equation 3.13, an additional quantity of $\hat{q}$ is required. $\hat{q}$ can be calculated by using the extended Euclidean algorithm [70, page 22] to solve the relation

$$\rho \cdot \rho^{-1} - q \cdot \hat{q} = 1. \tag{3.15}$$

The result will yield the values of both ρ^{-1} and $\hat{q}$.

The algorithm for computing the Montgomery product is shown in Algorithm 15.

Algorithm 15 Montgomery algorithm.

Input: $A', B', \rho, q, \hat{q}$
Output: $R' = A' \cdot B' \cdot \rho^{-1} \pmod{q}$

1: $t = A' \cdot B'$
2: $s = t \cdot \hat{q} \pmod{\rho}$
3: $u = (t + s \cdot q)/\rho$
4: **if** $u \geq q$ **then**
5: **return** $(R' = u - q)$
6: **else**
7: **return** $(R' = u)$
8: **end if**

The algorithm includes a multiplication in step 1, while steps 2 to 8 perform the modular reduction. The efficiency of the algorithm comes from the ability to specify the value of ρ. As $\rho = 2^l$, if $l = 32$, steps 2 to 8 of the algorithm can be efficiently computed on a GPP with a 32 bit datapath. Similarly, if the algorithm was being implemented on a 64 bit platform, a value of $l = 64$ would be chosen.

The main operation performed in ECC algorithms is the computation of $Q = kP$. Although a Montgomery multiplication is efficient to compute, it requires that the operands be converted to the Montgomery domain before Algorithm 15 can be used. Performing the conversion process for every multiplication would be less efficient than using a standard multiplication algorithm. However, an alternative approach is to

convert all values required for the calculation of $Q = kP$ into the Montgomery domain, perform the point scalar multiplication using values in the Montgomery domain, and then convert back to the standard domain at the end of the calculation.

3.5 Instruction Set Extension for ECC

In the previous sections, the algorithms and mathematical principles of ECC were introduced. On an FPGA platform there are many different options for implementing ECC algorithms. As SoC designs are generally implemented around a GPP [58], the performance of the different ECC algorithms will first be analysed on a GPP constructed from FPGA resources, such as that discussed in Section 2.9.1.

The GPP that will be used is the Microblaze soft-core processor [130] which is based on a 32 bit RISC architecture and can be implemented on any of the Xilinx FPGA families. The Microblaze can be configured so that it is capable of running a version of the Linux kernel, this makes communication with the FPGA a simple task due to the built in networking capabilities of the Linux kernel.

A design for performing elliptic curve arithmetic operations was implemented on an XUPV5-LX110T development board. The maximum clock frequency of the Microblaze when implemented on the board is 125 MHz. The board contains 256 *Megabytes* (MBs) of DDR2 RAM that the Microblaze can access through an external memory controller. The implemented design uses the DDR2 RAM for storing some of the code sections, while the heap and stack are placed in 64 kB of BRAM internal to the FPGA.

3.5.1 Software

The FPGA was configured as shown in Figure 3.1. The GNU GMP library [50] was compiled for the Microblaze and used to implement all of the algorithms. All multiplications were performed using the Montgomery multiplication method, Algorithm 15, with $\rho = 2^{32}$. Table 3.2 shows the results for the three different field sizes with bitlengths of 192, 256, and 521, with the Microblaze clocked at 125 MHz. The curves used are the prime field curves defined by NIST in [89]. The conversion of elliptic curve points, to and from affine coordinates, are included in all the results, along with any precomputations required by the co-Z algorithms.

The results are quite slow for the Microblaze in this setup. This is to be expected as the Microblaze is not optimised in any way to perform large finite field multiplications. All of these algorithms are designed with software implementations in mind and the

results reflect this, with the best result in each field size being the left-to-right signed-digit algorithm with (X, Y)-only co-Z (Algorithm 14) algorithm. All algorithms, apart from the D&A, have a regular structure and therefore the timing results do not change for different values of the scalar k. The timing results for the D&A algorithm, however, are dependant on the Hamming weight of k; with a higher Hamming weight resulting in more point addition operations being executed in Algorithm 6. Therefore, a value of k that has a Hamming weight of $l/2$, where l is the length of k in bits, was used to generate the D&A results. The result is then the average length of time a point scalar multiplication with the D&A algorithm would take. The maximum number of multiplications and additions for the D&A algorithm is $\{24\mathbf{M}, 21\mathbf{A}\}$. This occurs when both a point addition and point doubling are executed in the main loop. The minimum number of multiplications and additions is $\{8\mathbf{M}, 14\mathbf{A}\}$, which occurs when only a point doubling is executed. In order to compare the D&A algorithm with all the other algorithms that have a constant execution time, the computational complexity of the D&A algorithm is taken as the average of these two values, $\{16\mathbf{M}, 18\mathbf{A}\}$.

On average, for the co-Z algorithms, the precomputations and conversion to and from affine coordinates takes about 9 ms in the 192 bit case and can be considered negligible in this software implementation.

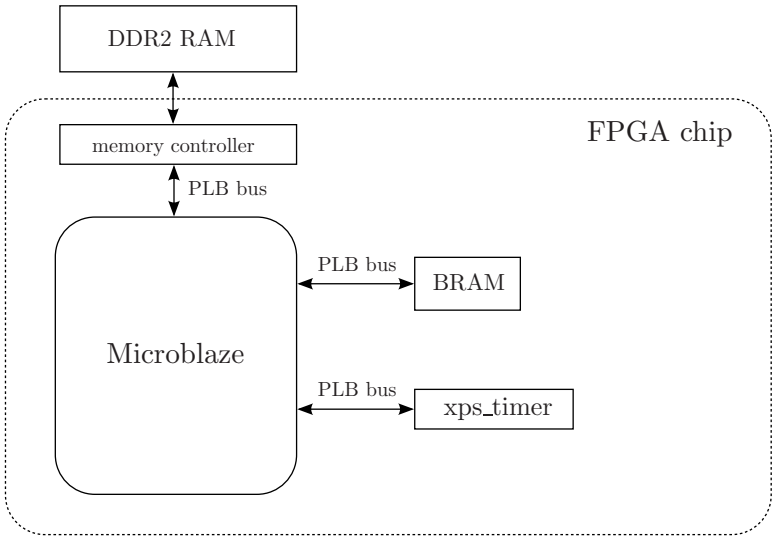


Figure 3.1: Microblaze system setup.

Algorithm	Main loop	192 bit	256 bit	521 bit
		Time (ms)	Time (ms)	Time (ms)
D&A (Alg. 6)	$\{16\mathbf{M}, 18\mathbf{A}\}$	1060	1976	20330
ML co-Z (Alg. 8)	$\{16\mathbf{M}, 21\mathbf{A}\}$	1083	2033	21220
Joye I (Alg. 11)	$\{16\mathbf{M}, 21\mathbf{A}\}$	1085	2036	21286
Joye II (Alg. 12)	$\{16\mathbf{M}, 27\mathbf{A}\}$	1058	1981	20157
ML (X,Z) (Alg. 9) (Alg. 26 & 33)	$\{17\mathbf{M}, 14\mathbf{A}\}$	1134	2141	22355
ML (X,Z) (Alg. 9) (Alg. 27 & 33)	$\{16\mathbf{M}, 14\mathbf{A}\}$	1056	1980	20683
ML (X,Z) (Alg. 9) (Alg. 28 & 34)	$\{15\mathbf{M}, 13\mathbf{A}\}$	991	1876	19299
ML (X,Y) (Alg. 13)	$\{14\mathbf{M}, 39\mathbf{A}\}$	935	1753	17726
SD (X,Y) (Alg. 14)	$\{14\mathbf{M}, 28\mathbf{A}\}$	**924**	**1737**	**17668**

Table 3.2: Software implementation results on a Microblaze processor.

3.6 Custom Hardware Acceleration

In the previous section, the baseline results for a software based design were presented. Due to the configurability of FPGAs, they offer many options in terms of increasing the performance of designs. The Microblaze processor supports the addition of custom instructions, through the use of an FSL bus, as described in Section 2.9.2. In this section an analysis of the use of a custom multiply instruction for the Microblaze will be conducted.

In Section 3.2, various ECC algorithms were introduced that remove the need to perform finite field inversions during the main loop of the scalar multiplication algorithm. The main loop then consists of additions, subtractions, and multiplications over $\mathbb{F}_q$; of which the multiplications are the most computationally intensive. When deciding on an instruction to add custom hardware for, it therefore makes sense to offload the multiplication operation as it has the greatest impact on the computation time of the ECC algorithms. This can be seen in Table 3.2, where the algorithm with the least number of multiplications, SD (X,Y) (Alg. 14), has the best performance.

3.6.1 Montgomery Multiplication in Hardware

To implement the Montgomery multiplication in hardware a modified version of the algorithm is used. From [124], we know that when used as part of the computation of $Q = kP$ in ECC algorithms, the conditional subtraction at the end of the Montgomery multiplication algorithm is not required. Algorithm 16 shows how the Montgomery multiplication is performed without a conditional subtraction. The algorithm can be derived from Algorithm 15 by setting $\rho = 2^1$, resulting in a bitserial method which is

more suited to a hardware implementation than Algorithm 15.

Algorithm 16 Montgomery multiplication.

Input: $A' = \sum_{i=0}^{l} a'_i 2^i, B' = \sum_{i=0}^{l} b'_i 2^i, q$
Output: $R' = A' \cdot B' \cdot 2^{-l+2} \pmod{q}$

1: $R' = 0, a'_{l+1} = b'_{l+1} = 0$;
2: **for** $i = 0$ to $l + 1$ **do**
3: $\quad t_i = R'_{i-1} + (b'_i A' \pmod 2)$
4: $\quad R'_i = (R'_{i-1} + t_i q + b'_i A')/2$
5: **end for**

A circuit for performing a Montgomery multiplication is shown in Figure 3.2. The design requires two $l + 2$ bit full adders, which form the critical path of the design.

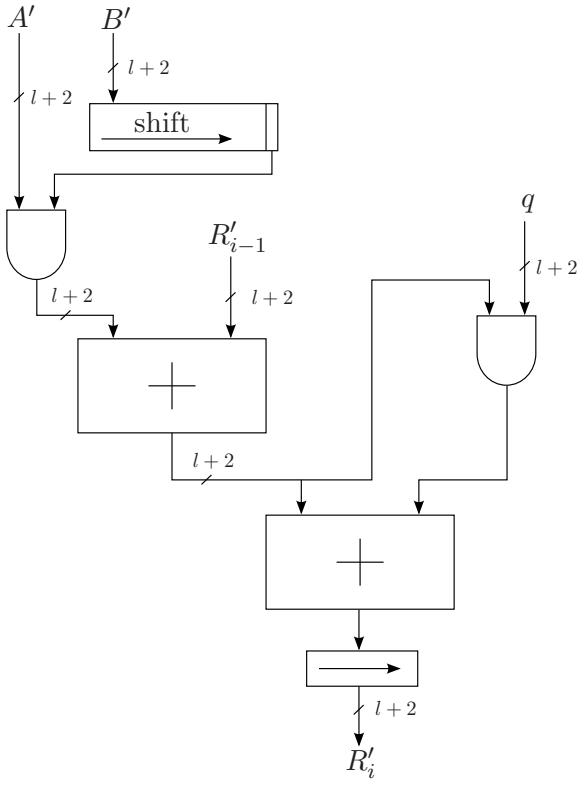


Figure 3.2: Montgomery multiplier.

3.6.2 Instruction Set Extension Results

The Montgomery multiplier was connected via an FSL bus to the Microblaze processor, as shown in Figure 3.3. The clock signal from the multiplier is supplied by the Microb-

laze, through the FSL bus; therefore, the multiplier runs at the same frequency as the Microblaze. In this setup, the multiplier forms the critical path in the design and hence determines the clock frequency for the entire system. In the case of the 192 bit design, the system clock frequency was set to 100 MHz and in the 256 and 521 bit cases, the clock frequency was set to 75 MHz. A pipeline register was added to the multiplier design for the 521 bit implementation in order to reduce its critical path. This doubles the number of clock cycles it takes to perform a multiplication. The FPGA area usage results for each entire system and the multipliers alone are shown in Table 3.3. A timer and debug module were also included in the design in order to measure computation times and for command line output.

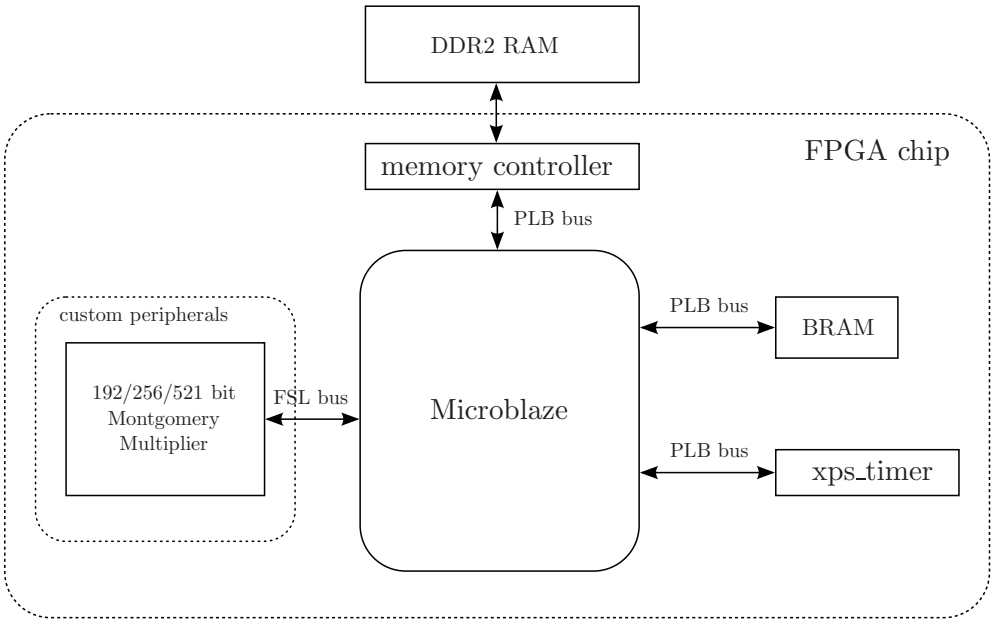


Figure 3.3: Microblaze with hardware multiplier.

Table 3.4 shows the results obtained from the Microblaze with hardware acceleration. The results show the timing for performing a full computation of kP including all conversion to and from affine coordinates, and also any precomputations for each algorithm. The results assume that the point P is unknown and therefore no values that could be precomputed and stored in RAM are used. Comparing Table 3.4 with Table 3.2, it can be seen that the hardware multiplier reduces the computation time of the different algorithms by on average 89-94%. The large reduction in computation time can be attributed to the fact that the hardware multiplier performs both the mul-

Design	Area (Slices)	BRAM	DDR2 RAM	DSP48E	Freq. (MHz)
Microblaze 192 bit	3145	65 × 36 k	256 MB	3	100
Microblaze 256 bit	3454	65 × 36 k	256 MB	3	75
Microblaze 521 bit	3466	65 × 36 k	256 MB	3	75
192 bit mult	334	0	0	0	100
256 bit mult	499	0	0	0	75
521 bit mult	904	0	0	0	75

Table 3.3: Microblaze FPGA resource usage.

tiplication and Montgomery modular reduction, which are more time consuming than modular additions.

In the software implementation results from Table 3.2, the SD(X,Y) (Alg. 14) was fastest. This due to the fact that the SD(X,Y) (Alg. 14) algorithm requires the least number of multiplications. With the addition of the hardware Montgomery multiplier the ML(X,Z) (Alg. 9) (Alg. 27 & 33) and ML(X,Z) (Alg. 9) (Alg. 28 & 34) algorithms are the best performing. These algorithms require one and two extra multiplications, respectively, over the SD(X,Y) (Alg. 14) algorithm, however, the number of additions is reduced. When implemented in software, the multiplication operation is the dominant factor in the computation time and additions account for only a small percentage of the computation time. The inclusion of the hardware multiplier has reduced the computation times to the point where the addition operations also have a noticeable impact on the performance; thus, changing the order of the results.

Algorithm	Main loop	192 bit	256 bit	521 bit
		Time (ms)	Time (ms)	Time (ms)
D&A (Alg. 6)	$\{16\mathbf{M}, 18\mathbf{A}\}$	94	228	1534
ML co-Z (Alg. 8)	$\{16\mathbf{M}, 21\mathbf{A}\}$	113	296	2207
Joye I (Alg. 11)	$\{16\mathbf{M}, 21\mathbf{A}\}$	114	297	2221
Joye II (Alg. 12)	$\{16\mathbf{M}, 27\mathbf{A}\}$	96	252	1686
ML(X,Z) (Alg. 9) (Alg. 26 & 33)	$\{17\mathbf{M}, 14\mathbf{A}\}$	74	203	1176
ML(X,Z) (Alg. 9) (Alg. 27 & 33)	$\{16\mathbf{M}, 14\mathbf{A}\}$	**71**	**166**	1175
ML(X,Z) (Alg. 9) (Alg. 28 & 34)	$\{15\mathbf{M}, 13\mathbf{A}\}$	**71**	185	**1172**
ML(X,Y) (Alg. 13)	$\{14\mathbf{M}, 39\mathbf{A}\}$	97	250	1638
SD(X,Y) (Alg. 14)	$\{14\mathbf{M}, 28\mathbf{A}\}$	87	225	1506

Table 3.4: Microblaze with Montgomery multiplier results.[1]

[1]It should be noted that the times given in this table are average computation times averaged over 1000 executions of each algorithm. As the results are from a software implementation, the computation time of each operation can vary slightly between different executions.

3.7 Optimisations for the $q = 2^n - 1$ case

The curves used in the implementations in the previous section are all defined by NIST in [89], and have moduli of slightly different forms for each field size. The form of modulus is irrelevant when using the Montgomery multiplication algorithm as it uses its own fast method for modular reduction. However, the 521 bit curve uses a modulus where a particularly efficient modular reduction technique can be applied. A prime number of the form $2^n - 1$ is known as a Mersenne prime; with its bitlength equal to n.

In 1992, Hiasat [55] proposed a multiplier architecture that takes advantage of the fast modular reduction that can be performed when working with moduli of the form $2^n \pm 1$. Hiasat's design requires slightly different modular reduction circuitry depending on whether the modulus is of the form $2^n - 1$, 2^n, or $2^n + 1$. The modulus of the form $2^n - 1$ is of most interest for ECC and is therefore discussed below.

If,

$$R = AB, \tag{3.16}$$

and R can be represented as,

$$R = \sum_{i=0}^{2n-1} r_i 2^i, \tag{3.17}$$

the modular reduction of R can be described as,

$$|R|_{2^n-1} = \left| \sum_{i=0}^{n-1} r_i 2^i + 2^n \sum_{i=n}^{2n-1} r_i 2^{i-n} \right|_{2^n-1}, \tag{3.18}$$

where,

$$|2^n|_{2^n-1} = |1|_{2^n-1}. \tag{3.19}$$

This type of multiplier can be implemented through the use of a full width multiplier followed by correction circuitry that performs the modular reduction step. There are many different methods for performing the $n \times n$ bit multiplication which will be discussed in Sections 3.7.1 to 3.7.3. Regardless of how the $n \times n$ bit multiplication is performed, the correction circuitry remains the same and consists of some simple combinational logic; the setup is shown in Figure 3.4. The purpose of the reduction circuitry is to implement the following cases that can result from the full width multiplication of A and B.

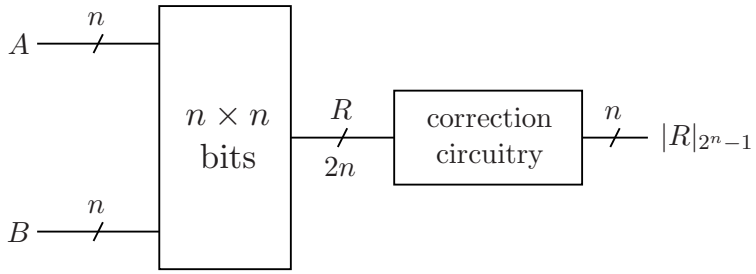


Figure 3.4: Hiasat multiplier.

The most significant and least significant n bits of R are added together, as in Equation 3.18, this is taken as the result if neither of the following two cases occur.

Case 1: If the result of the addition is all 1's, the result is set to all 0's.

Case 2: If the carry bit C_{out} is 1, R is incremented by 1 and is taken as the result.

It can be shown that **Case 1** and **Case 2** cannot occur at the same time [55]. The implementation of case 1 only requires some comparison circuitry, and case 2 requires an n bit adder to increment the value of R. Both of these cases are relatively efficient to implement in hardware and the entire correction step can be performed in a single clock cycle.

It has been shown that the Hiasat modular reduction step above can be performed efficiently, as it consists of simple combinational logic. Next, several architectures for implementing the $n \times n$ bit multiplication will be discussed.

3.7.1 Serial Multiplier

A simple way of calculating the product of two integers of length n, is to implement the multiplication as the sum of n partial products, as shown in Algorithm 17. The *Least Significant Bit* (LSB) of the multiplicand B, is scanned. If the current LSB is a 1, the multiplier A, is added to an accumulator R. If a 0 bit is detected, nothing is added to the accumulator. B is then shifted to the right by 1 bit position. The process continues until every bit of the multiplicand has been scanned. This method is known as the Schoolbook method for multiplication.

This form of multiplier requires a single n bit full adder and performs the multiplication in n clock cycles. Figure 3.5 shows how the output of the adder feeds into a shift register that holds the accumulator value. B is stored in the LSBs of the accumulator and shifted right; thus, B does not need to be stored in a register of its own. After n

Algorithm 17 Schoolbook multiplication algorithm.

Input: $A = \sum_{i=0}^{n-1} a_i 2^i, B = \sum_{i=0}^{n-1} b_i 2^i, n$
Output: $R = A \times B \pmod{2^n - 1}$

1: $R = 0$;
2: **for** $i = 0$ **to** $n - 1$ **do**
3: **if** $b_i = 1$ **then**
4: $R = R + A$
5: **end if**
6: $R \gg 1$
7: **end for**
8: **Reduce**(R)
9: **return** $R = A \times B \pmod{2^n - 1}$

clock cycles B is completely shifted out of the accumulator and only the result of the multiplication, R, remains. R can then be fed into the Hiasat correction circuitry to obtain the result modulo $2^n - 1$, as shown in Figure 3.4. The multiplier requires only FPGA slice logic for its implementation and its critical path is through the n bit adder.

3.7.2 Booth Multiplier

In 1951, Booth [16] introduced a multiplication algorithm which can reduce to $n/2$ the number of partial product additions required, at the cost of some extra hardware. A modified version of Booth's algorithm was introduced in [72]. Both algorithms work by precomputing a set of partial product multiples, such as $\{0, A, 2A, 3A\}$ in the case of the Booth algorithm. Two bits of the multiplicand are then scanned at a time and depending of the value of these two bits, one of the multiples is added to the accumulator.

The modified Booth algorithm [72] improves on the Booth algorithm, allowing for a more efficient hardware implementation of the precomputation circuitry, by modifying the set of precomputed partial product multiples. In the set of multiples $\{0, A, 2A, 3A\}$, $3A$ is the most computationally intensive as it requires the use of an adder, while $2A$ can be computed by a bitwise shift operation which can be efficiently implemented in hardware. The carry chain in the adder required for $3A$, would increase the critical path and the area of the design. The modified Booth algorithm solves this problem by using the set of multiples $\{-2A, -A, 0, A, +2A\}$, where every multiple in this set can be calculated by a bitwise shift, a bitwise inversion, or both. For the modified Booth algorithm, overlapping groups of 3 bits of the multiplicand must be scanned at a time. On the first iteration of the algorithm, the LSB of the 3 bits is taken to be equal to

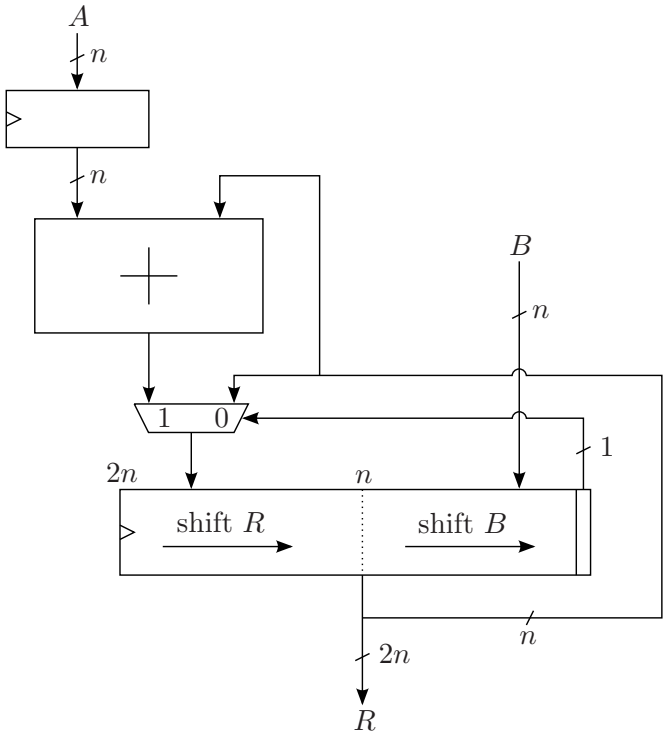


Figure 3.5: Serial multiplier.

0. A detailed description of Booth recoding schemes can be found in [11]. Algorithm 18 defines the modified Booth algorithm and an illustration of an architecture for its implementation is shown in Figure 3.6. The critical path of the multiplier is through the adder and the circuitry used for the precomputations. The architecture of the circuit, shown in Figure 3.6, is similar in structure to that of the serial multiplier from Section 3.7.1 and can also be implemented entirely in slice logic in the FPGA.

The number of iterations of the main loop of Algorithm 18 can be reduced if a larger set of precomputed multiples is used. This, however, would require an adder in the precomputation stage and would therefore increase the critical path and logic resources required for the circuit.

3.7.3 Multiplier with BRAMs and DSP48Es

In the previous sections, multiplier architectures that can be implemented entirely in FPGA slice logic were discussed. However, many FPGAs also contain other resources, such as BRAM and DSP blocks. The registers in the previous designs can then be replaced by BRAM and some of the arithmetic operations implemented using DSP blocks. By incorporating these elements into the design and using different multipli-

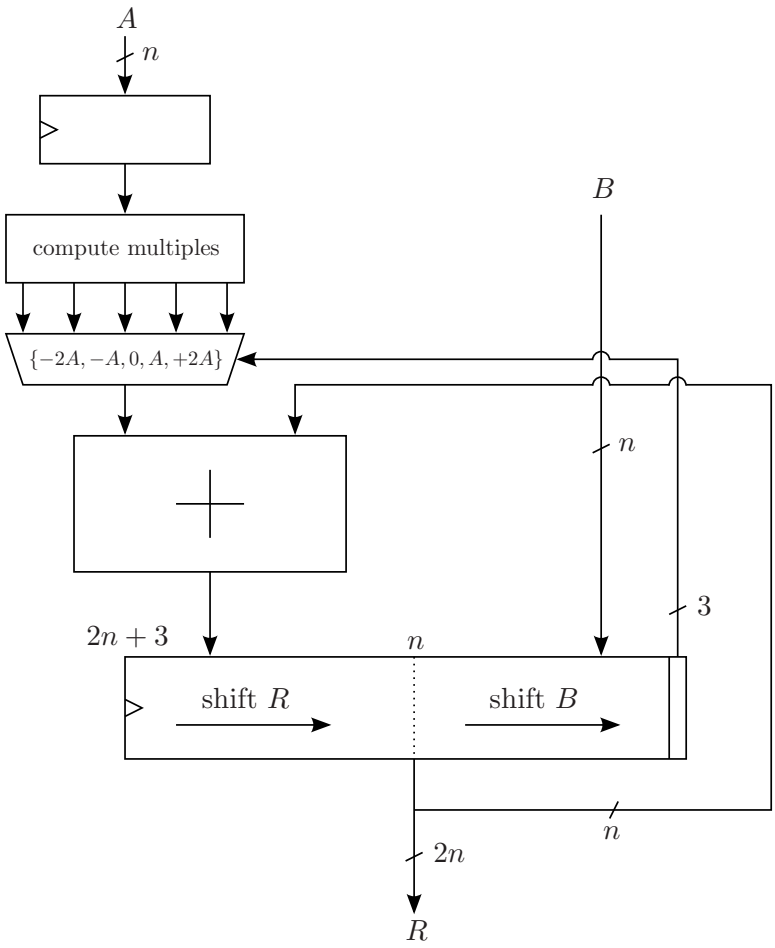


Figure 3.6: Booth2 Multiplier

Algorithm 18 Booth recoded multiplication.

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n} b_i 2^i$ with $b_0 = 0$
Output: $R = A \times B \pmod{2^n - 1}$

Precompute: $-A = \overline{A}$, $2A = A \ll 1$, $-2A = \overline{2A}$, $R = 0$, $ones = \sum_{i=0}^{n}(1)2^i$
for $i = 0$ **to** $n - 1$ **do**
  switch (b_{i+2}, b_{i+1}, b_i)
  case "000": $R = R + 0$
  case "001" or "010": $R = R + A$
  case "011": $R = R + 2A$
  case "100": $R = R - 2A$
  case "101" or "110": $R = R - A$
  case "111": $R = R + ones$
  $R \gg 2$
  $i = i + 2$
  end switch
end for
Reduce(R)
return $R = A \times B \pmod{2^n - 1}$

cation algorithms, alternative multiplier architectures can be developed. The resulting architecture must be designed such that it maximises the usage of the available FPGA resources. The DSP blocks present in Xilinx FPGAs are hardcoded resources and therefore can operate at a higher frequency than if the same operation was implemented in slice logic. They are capable of a full 18×18 bit multiplication; therefore, the multiplication algorithm should use operations of this size. In this section, a multiplier architecture will be introduced that uses BRAM and DSP blocks to reduce the number of clock cycles required to perform a multiplication.

3.7.3.1 Multiplier Architecture

The general structure of the DSP and BRAM based design is shown in Figure 3.7. In comparison to the architectures in the previous sections, the goal of the design is to replace slice logic with hardcoded resources in the FPGA. The design presented in this section uses DSP blocks to implement the arithmetic operations in a finite field multiplication, and BRAM to implement the memory requirements.

The multiplication algorithm will be discussed in Section 3.7.3.2, first an overview of the multiplier design will be given. The architecture is shown in Figure 3.7. The inputs to the multiplier are decomposed until their partial products are small enough to be calculated with a single DSP block. A number of DSP blocks are used to calculate

the partial products which are then stored in dual port BRAM. The adder reads the partial products from BRAM, calculates their sum, and returns the result to BRAM. To decrease the number of clock cycles required to perform a multiplication, the number of DSP blocks used can be increased. The maximum possible operating frequency of the design will be unchanged by the number of DSP blocks used, as they operate in parallel. The design shown in Figure 3.7 consists of four DSP blocks, four BRAM, and one adder. However, if the number of DSP blocks and BRAM is increased, the general setup of the design remains the same.

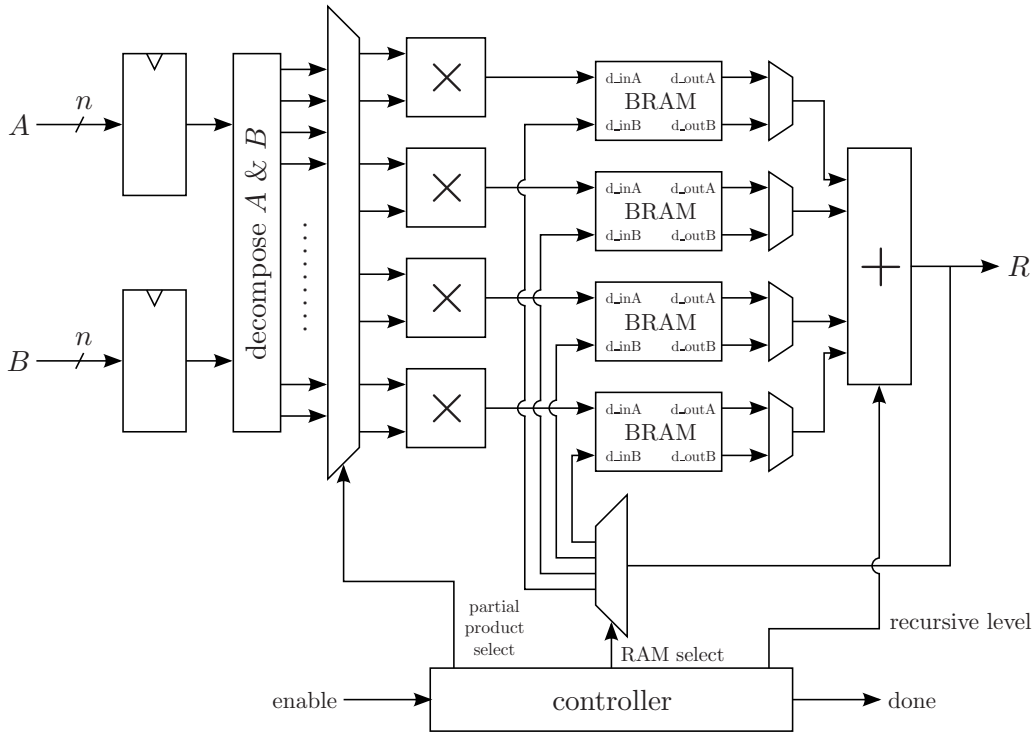


Figure 3.7: DSP48E and BRAM based multiplier.

3.7.3.2 Decomposing the Multiplicands

The standard way to decompose the multiplication $A \times B$, is as follows. Let A and B be represented as a binary string of length t. To split A and B into two equal parts let $n = t/2$, then:

$$
\begin{aligned}
A &= a_1 2^n + a_0, \\
B &= b_1 2^n + b_0, \\
A \times B &= (a_1 2^n + a_0)(b_1 2^n + b_0), \\
&= r_2 2^{2n} + r_1 2^n + r_0,
\end{aligned}
$$

where,

$$
\begin{aligned}
r_2 &= a_1 b_1, \\
r_1 &= a_1 b_0 + a_0 b_1, \\
r_0 &= a_0 b_0.
\end{aligned} \tag{3.20}
$$

This method can be applied recursively to the partial products, a_0b_0, a_0b_1, a_1b_0, a_1b_1, until the calculation of r_0, r_1, and r_2 consist of 18×18 bit, or less, multiplications. For each decomposition there are then four partial products that have to be calculated, a_0b_0, a_0b_1, a_1b_0, and a_1b_1. DSP blocks are used in the design to calculate these partial products. Depending on the bitlength of the multiplication that is being performed, there can be a large number of partial products that need to be calculated. For this reason, some form of memory element must be present in the design in order to store the partial products. Hardcoded BRAM that is present in Xilinx Virtex 5 FPGAs is therefore used for this purpose. After calculating the partial products they must be summed as shown in Figure 3.8.

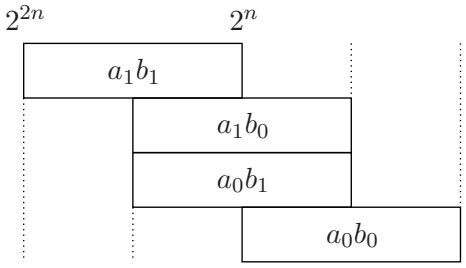


Figure 3.8: Addition of partial products.

The lower half of a_0b_0 is not involved in the addition and can be routed directly to the output of the adder. By using the Karatsuba method for decomposition [97], the number of multiplications required to calculate the partial products can be reduced

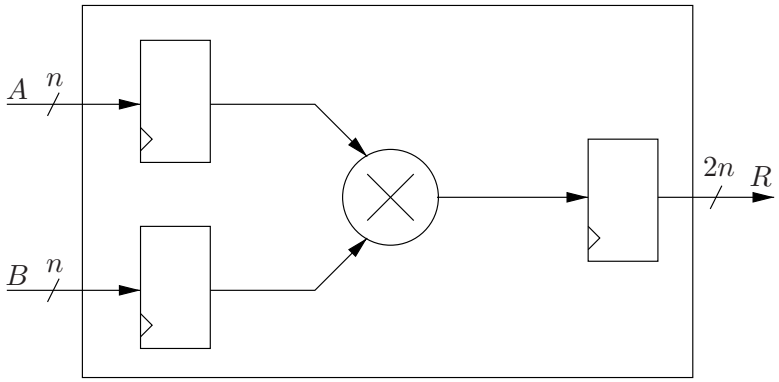


Figure 3.9: Pipelined multiplier in a DSP block.

from four to three. This is done by replacing r_1 in Equation 3.20 with

$$r_1 = (a_1 + a_0)(b_1 + b_0) - r_2 - r_0. \tag{3.21}$$

This method, however, increases the number of additions that need to be performed. Since the critical path in the design is through the adder, the previous method for decomposition, Equation 3.20, is used.

3.7.3.3 DSP Blocks

Each DSP block on a Xilinx Virtex 5 FPGA is capable of performing a full 18×18 bit multiplication, giving a 36 bit result. If the pipelining registers in the DSP blocks are used, Figure 3.9, the maximum possible clock frequency achievable is about 450 MHz. To take advantage of the high clock frequency of these DSP blocks, a large multiplication must be decomposed recursively until the size of the partial product multiplications is 18×18 bits or less. The maximum frequency of the design will not be limited by the DSP blocks but by the critical path of the adder used to sum the partial products. It is possible to pipeline the adder, but for large bit lengths, achieving a clock frequency close to that of the DSP blocks is not possible.

The pipelining registers add a delay of 1 clock cycle on the input and 1 on the output of the DSP block. In the smaller designs these pipelining registers are required, as without them, the DSP blocks would have a longer critical path than the adder circuit in the design. However, as the bit length increases the critical path through the adder becomes larger than the unpipelined DSP blocks; therefore, the pipelining registers are not used and 2 clock cycles per multiplication can be saved. Even though the adder is pipelined, when the bit length reaches several hundred bits, it is not feasible to pipeline

the adder enough so that the critical path is similar to that of the DSP blocks, as doing so would result in an unacceptable number of clock cycles for the addition of the partial products.

3.7.3.4 Block RAM

The FPGAs in the Virtex 5 family contain BRAM which is distributed in columns throughout the chip in 36 *Kilobit* (kbit) blocks. All BRAM used in the multiplier, shown in Figure 3.7, is configured as dual port BRAM. The output ports of the BRAM support three different modes of operation WRITE_FIRST, READ_FIRST and NO_CHANGE. These different modes determine how data is written into memory in BRAM and how data appears on the output port of the BRAM if a read and write are being performed on the same BRAM address, in the same clock cycle. Depending on the bit length of the multiplication being performed, either WRITE_FIRST or READ_FIRST modes are used; NO_CHANGE mode is not used in any of the designs. In WRITE_FIRST mode, the data being written to an address in BRAM also appears on the output port in the same clock cycle. In READ_FIRST mode the data that was previously at the address location appears on the output port and then the new data is written to the address in BRAM. The BRAM mode of operation that is used in the multiplier depends on the complexity of the adder. As the bitlength increases, so does the number of partial products that need to be stored in BRAM. Therefore, WRITE_FIRST mode is used in the smaller multiplier designs, where the adder does not have to write to and read from the same location in BRAM, at the same time. This occurs when only two RAM locations are required in each BRAM and saves a clock cycle at the multiplication stage. READ_FIRST is used for design where the bit length is large and the adder requires the ability to write to, and read from, the same BRAM locations at the same time i.e., when decomposition of the multiplicands has been applied recursively several times.

3.7.3.5 The Adder

An adder is required in the design to sum the partial products. The four inputs to the adder, a_0b_0, a_0b_1, a_1b_0, and a_1b_1, must be bitwise shifted by the correct amount to ensure that they are aligned to give the correct sum on the output, as shown in Figure 3.8. For multiplications several hundred bits in length, the decomposition will have been applied recursively several times until each input to the DSP blocks is at most 18 bits. For the 521 bit case, the multiplicands must be recursively decomposed five times until each multiplication is 17×17 bits. For this reason, the adder will have to

align the partial products differently each time the decomposition is performed. The signal *recursive_level*, shown in Figure 3.7, is used to indicate to the adder how the partial products should be aligned. The output of the adder is fed into a multiplexer which selects which BRAM unit the result is routed to. The results are written into the same address of each different BRAM unit. Spreading the partial products across the multiple BRAM units allows the adder to read the partial products in parallel from across all of the BRAM units, in a read operation.

Due to the fact that the adder is the critical path in the design, it must be pipelined in order to keep the critical path as short as possible, without adding so many extra clock cycles as to make the design slower than a bitserial approach from Sections 3.7.1 or 3.7.2. The adder is implemented in slice logic as the DSP blocks cannot perform additions of a sufficient length without cascading them. A design was tested with DSP blocks but it was found that implementing the adder in slice logic resulted in a shorter critical path.

3.7.3.6 Controller

A controller is required to schedule all multiplications, additions, and all of the BRAM read/write operations along with all the data routing. The controller consists of a *Finite State Machine* (FSM) which starts and stops various counters. In total, there are six counters used in the design.

BRAM address counter: There are two BRAM address counters used in the design, one for each input port of the BRAMs. Every time an output from a DSP block is written to BRAM, the counter for BRAM input port A increments. Every time a result from the adder is written to BRAM, the counter for port B increments.

BRAM input select counter: This counter is incremented when the output of the adder needs to be written to the next BRAM unit.

Delay counter: Determines how long to wait before the multiplier can move to the next state in the FSM. The delays are caused by pipelining in the adder.

Partial product counter: The output of this counter is connected to a multiplexer that routes the correct parts of the decomposed multiplicands, to the different DSP blocks. When the counter increments, the multiplexer routes a different set of partial products to the DSP blocks.

Recursive level counter: This counter is connected to the adder and through the signal *recursive_level*, it controls how the adder aligns the partial products that are being added, as shown in Figure 3.8.

3.7.3.7 Multiplier Operation

The inputs A and B are decomposed by routing the correct portions of the array of bits, to the DSP blocks. A multiplexer routes the decomposed A and B signals into the DSP blocks, which are then processed. The controller determines which partial products are routed to the DSP block at each clock cycle through the use of the partial product select line, shown in Figure 3.7. For every DSP block in the design, there is a dual port BRAM unit and for every four DSP blocks there is a single adder. The output of the DSP blocks are fed into the "d_inA" port of each BRAM and the value is stored in address location 0. The "d_inB" port of the BRAM is only used by the feedback from the adder stage. For all larger bit lengths READ_FIRST RAM is used. The controller also controls the BRAM enable lines and address signals.

Due to the type of multiplicand decomposition that is used, the number of DSP blocks in the design is always a multiple of 4. This ensures that the adder can sum the four partial products in the clock cycle after they have been calculated by the DSP blocks. If the number of DSP blocks was not a multiple of 4, some DSP blocks would remain unused for periods of time. This would lead to a reduction in the efficiency of the design. The control circuitry would also be more complex; thus, increasing the area of the circuit.

A generator was written in C++ to generate the *VHSIC Hardware Description Language* (VHDL) code for the multiplier. This allows for the generation of multipliers with different bit lengths, with minimal effort. The generator is designed to take as inputs, the bit length of the multiplication, and number of DSP blocks to be used. From these values the generator produces all the VHDL files necessary to implement the multiplier.

3.7.4 Results

Shown in Table 3.5 are post place and route results for a Virtex XC5VLX220-1ff1760 FPGA. The results for a 127 bit version of the multiplier are also included as an ECC implementation using a modulus of the form $2^{127} - 1$ has been suggested by Galbraith et al. in [38]. The best results in each category are highlighted in bold.

In order to properly route the 521 bit Booth circuit, a pipelined design was used.

Multiplier	Bit Length	Area (slices)	Max. Freq. (MHz)	Clk. Cycles	Throughput (Mbit/s)
Montgomery	127	**264**	161	129	159
Serial	127	352	**186**	128	185
Booth	127	353	167	65	327
DSP_BRAM (4DSPs)	127	1139	110	**31**	**451**
Montgomery	521	**957**	54	523	54
Serial	521	1280	50	522	50
Booth	521	1601	**84**	525	**83**
DSP_BRAM (4DSPs)	521	6250	52	**353**	77

Table 3.5: Multiplier performance and power consumption results.

This allowed the design to be routed onto the chip without exceeding the length of the carry chain that run vertically in each column of CLBs in the FPGA. Routing from one column to the next can significantly increase the critical path of the design, as there is no dedicated carry routing from one column to the next. The circuits that make use of BRAM and DSP blocks are capable of performing the multiplications in fewer clock cycles than the other designs. However, the area used by these multipliers is much higher than the circuits that implement the multiplication using slice logic alone. This is due in part to the large multiplexer that is required to route the correct partial products to the DSP blocks, but also the adder that is used to sum the partial products.

In certain applications such as handling large amounts of traffic on a network, high speed encryption is more desirable than a low area design. In this situation making use of the BRAM and DSP resources on the FPGA may be a desirable way of implementing the multiplier. The Booth multiplier, however, appears to be the best alternative to the Montgomery, for implementation of the 521 bit NIST curve, as it has the shortest critical path and the highest throughput.

In order to compare the performance of the Booth multiplier against Montgomery for ECC applications, the Booth multiplier was implemented alongside a Microblaze processor; a similar setup to that from Section 3.6. The clock frequency of the system was set to 75 MHz. The results are shown in Table 3.6.

It can be seen that the use of the Booth multiplier instead of the Montgomery reduces the computation time by 4%–15%, with the relative performance of each algorithm remaining the same. Although the Booth multiplier appears significantly faster from the results in Table 3.5, these do not take into account the latency introduced by the Microblaze while communicating with the multiplier over the FSL bus. Both the latency of the FSL bus and the software overhead introduce delays in the system

Algorithm	Booth	Montgomery
	Time (ms)	Time (ms)
D&A (Alg. 6)	1386	1534
ML co-Z (Alg. 8)	2089	2207
Joye I (Alg. 11)	2094	2221
Joye II (Alg. 12)	1623	1686
ML(X,Z) (Alg. 9) (Alg. 26 &33)	1120	1176
ML(X,Z) (Alg. 9) (Alg. 27 &33)	1019	1175
ML(X,Z) (Alg. 9) (Alg. 28 &34)	**996**	**1172**
ML(X,Y) (Alg. 13)	1510	1638
SD(X,Y) (Alg. 14)	1387	1506

Table 3.6: Microblaze and 521 bit Booth multiplier results.

that increase the multiplication time for the Montgomery multiplier to 94 μs, and the Booth to 81 μs, when they are accessed through a function call in software.

Although the performance of the system can be increased by using the Booth multiplier architecture, this limits the type of modulus to that of $2^n - 1$, 2^n, or $2^n + 1$. The Montgomery multiplier, however, can be used with any form of modulus and would therefore lead to a more flexible system, suitable for a larger number of applications.

3.8 Discussion

In this chapter, the mathematical principles of ECC were introduced. The standard point scalar multiplication method, D&A with Jacobian coordinates, was compared against several more efficient algorithms based on co-Z coordinate addition. From the results presented, it is clear that in a software setting, finite field multiplications are the dominant factor in the computation time for the ECC algorithms discussed in this chapter.

ISE was then explored as a method to speed up the ECC point scalar multiplication by including a custom multiplication circuit in the design. This resulted in an 89%–94% reduction in computation times; thus, proving the effectiveness of ISE in this case. Further optimisations were made for the case where the modulus used for the finite field arithmetic was a Mersenne prime, with the Booth multiplier proving to be optimal in this case.

The performance gains that have been achieved in this chapter are unsurprising as a custom designed circuit will always outperform general arithmetic circuitry, such as is present in GPPs. From the results it can be seen that as further optimisations are

made, the flexibility of the overall design is reduced in order to improve performance. Therefore, a choice must be made as to the requirements of the system and a trade-off made between area, performance, and flexibility. In the next chapter, an ECC processor will be introduced, where the entire point scalar multiplication algorithm can be implemented outside of the GPP. Although ISE has been shown to improve the performance of the system, there is still a software overhead present in the results. An ECC processor can remove some of this software overhead and achieve better performance.

Chapter 4

FPGA Implementation of an ECDSA Coprocessor

4.1 Introduction

In the previous chapter, various ECC algorithms were implemented in a hardware-software co-design setting where only the finite field multiplications were performed in dedicated FPGA logic. In this chapter, the implementation of the entire ECC point scalar multiplication algorithm in FPGA logic will be examined.

This method of implementation has the advantage of increasing the performance of the system, but also lends itself to a more secure architecture, as intermediate values during the point scalar multiplication can be kept internal to the coprocessor. First, a generic structure for an ECC processor will be introduced.

4.2 ECC Processor

In [20], Byrne et al. introduced a configurable processor architecture capable of performing arithmetic over any extension field $\mathbb{F}_{q^m}$. The authors developed software that generates a VHDL description of a processor based on some configuration options that define the number, and type of arithmetic units in the design. The generic structure is shown in Figure 4.1. The arithmetic units can be configured to be adders, subtractors, multipliers, or field inversion units. This structure allows for the rapid comparison of the various ECC algorithms and coordinate systems. The user defines the scalar multiplication and point operation algorithms, which are then processed by the generator software. The algorithms are analysed and a list based scheduling algorithm [115] is

applied in order to maximise the usage of the arithmetic units.

The controller implements the scalar multiplication algorithm as an FSM. The group operations, such as point addition or point doubling, are then performed through the use of microcoded instructions stored in the *Read Only Memory* (ROM) block. The ROM instructions determine which arithmetic unit is used to execute each operation, through the use of the address decoders. The ROM instructions also control the I/O and addressing for the RAM block. The architecture for the multiplier unit is that of the Montgomery multiplier from Section 3.4. The architectures of the inversion and addition/subtraction arithmetic units are discussed in the following sections.

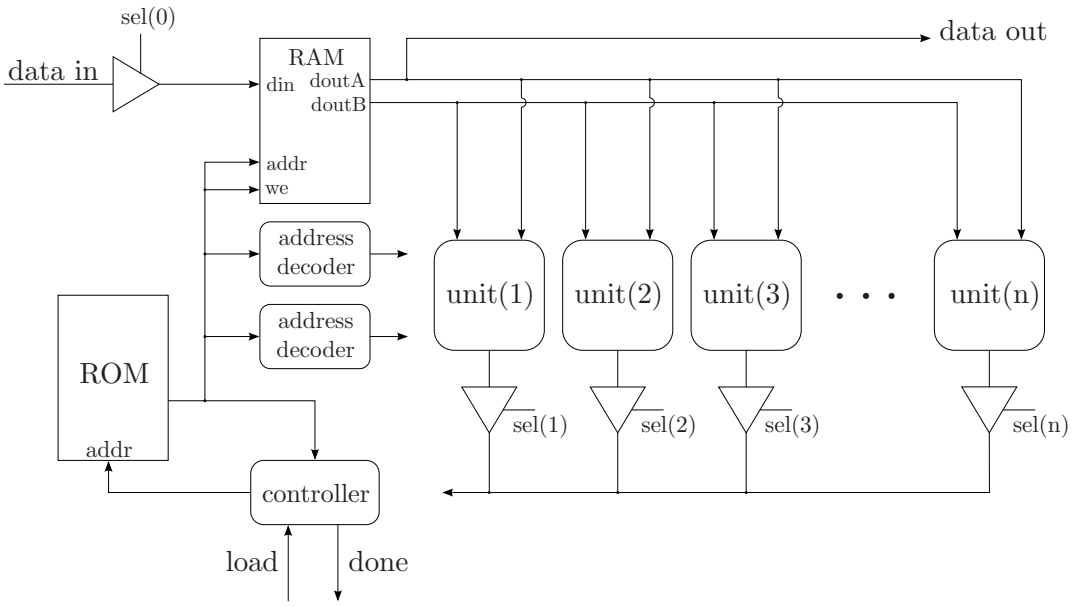


Figure 4.1: Byrne et al. design.

4.2.1 $\mathbb{F}_q$ Addition/Subtraction

The addition of two integers A and B modulo q can be achieved through the use of the circuit shown in Figure 4.2. The circuit consists of two full adders of bit width l, where $A = \sum_{i=0}^{l-1} a_i 2^i$ and $B = \sum_{i=0}^{l-1} b_i 2^i$. In the case of modular addition, the first adder has its carry input set to 0, and adds the two values A and B. The result is fed into the second adder which adds the result to the inverse of the modulus q, with the carry input set to 1; thus, performing a two's complement subtraction. The carry out of the second adder determines which value is taken as the final result, as this indicates whether the output from the first adder was in the correct range or whether the correction provided

by the second adder was required.

The corresponding subtraction, circuit is shown in Figure 4.3. In the case of subtraction, the B input to the first adder is inverted and its carry input is set to 1. This performs a two's complement subtraction which is then corrected by the second adder if the result $(A - B)$, from the first adder, was not in the correct range.

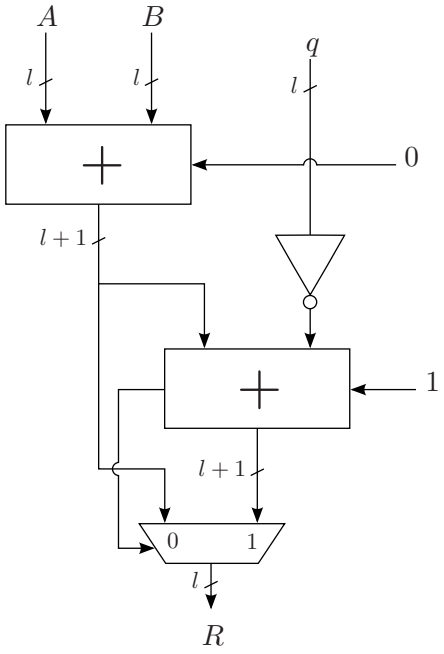


Figure 4.2: $\mathbb{F}_q$ adder.

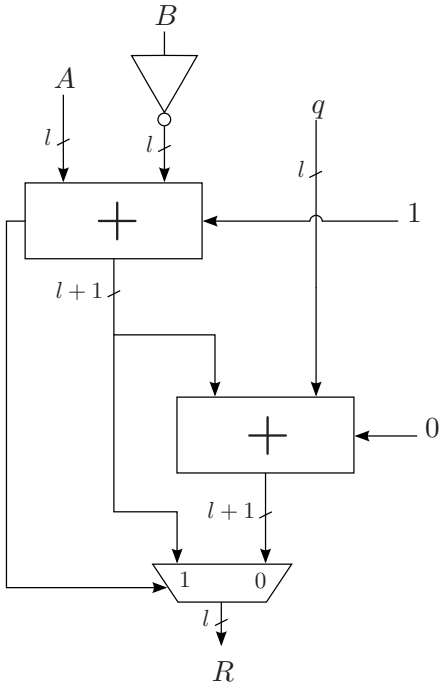


Figure 4.3: $\mathbb{F}_q$ subtractor.

4.2.2 $\mathbb{F}_q$ Inversion

An important and highly computationally intensive operation in finite field arithmetic is modular inversion. The modular inversion of an integer $x \in [1, q-1]$ is defined as the integer z such that $x \times z \pmod{q} = 1$. The Montgomery representation of an integer and its corresponding multiplication algorithm were introduced in Section 3.6.1. Similarly the Montgomery representation of the modular inverse of an integer is given by $x^{-1}2^l \pmod{q}$, where l is the length of q in bits [61].

One possible method to calculate the multiplicative inverse of an integer over $\mathbb{F}_q$ is to use use Fermat's little theorem. Let x be and integer and q be a prime, the theorem states that

$$x^q \equiv x \pmod{q}. \tag{4.1}$$

Equation 4.1 can be rewritten as

$$x^{q-1} \equiv 1 \pmod{q}. \tag{4.2}$$

The inverse of an element is then given by

$$x^{-1} \equiv x^{q-2} \pmod{q}. \tag{4.3}$$

Calculating the inverse using this method requires only multiplications over $\mathbb{F}_q$. As the bit length increases, however, this method becomes very inefficient; therefore, a method based on the extended Euclidean algorithm is preferred for these cases.

The Montgomery modular inverse algorithm, based on the binary extended Euclidean algorithm, requires only additions and subtractions over $\mathbb{F}_q$. The algorithm is split into two phases and is shown in Algorithms 19 and 20 [61]. Algorithm 19 shows the first phase, where the binary extended Euclidean algorithm is performed by steps 2 to 24. This is followed by a modular correction operation in steps 25 and 26. The binary extended Euclidean algorithm factors powers of 2 out of the variables u and v by dividing u, v, or their difference by 2, at each iteration updating the values of r and s. The $flag$ variable keeps track of which value s should take on the next iteration of the while loop. After k iterations, step 24 of the algorithm is reached and the value of r will be $r = -x^{-1}2^k \pmod{2q}$. The final modular reduction step gives $r = x^{-1}2^k \pmod{q}$. Thus, Algorithm 19 is complete in $k + 2$ iterations. The algorithm returns the values of $z = r = x^{-1}2^k \pmod{q}$ and k, which can be used as inputs to phase 2.

Phase 2 of the Montgomery modular inverse is shown in Algorithm 20, where the input is the result from Algorithm 19. Each iteration of the algorithm doubles the input modulo q and gives a final output of $z = x^{-1}2^{2l} \pmod{q}$. The algorithm requires $k - l$ clock cycles to complete.

The architecture used to implemented this algorithm was introduced by Crowe et al. in [25]. Each iteration of the loops in Algorithms 19 and 20 are evaluated in a single clock cycle; therefore, the total number of clock cycles required to calculate the Montgomery modular inverse is $2k - l + 2$. The inverter unit occupies more area than the multiplication or addition units, and also has a longer critical path.

4.3 Comparing Coordinate Performance

When designing an ECC processor for FPGAs, it is possible to implement a circuit that performs a field inversion; hence, using affine coordinates on an FPGA platform can be

Algorithm 19 Montgomery Inversion Algorithm: Phase 1

Input: $x = \sum_{i=0}^{k} x_i 2^i$, q, $gcd(q, x) = 1$
Output: $z = x^{-1}2^k \pmod{q}$, k where $l \leq k < 2l$

1: **Initialise:** $u \leftarrow q$, $v \leftarrow a$, $r \leftarrow 1$, $s \leftarrow 0$
2: **while** $v > 0$ **do**
3: **if** $u_0 = 0$ **then**
4: $u \leftarrow u/2$
5: **if** $flag = 0$ **then** $s \leftarrow 2r$
6: **else** $s \leftarrow 2s$
7: $flag = 1$
8: **else if** $v_0 = 0$ **then**
9: $v \leftarrow u/2$
10: **if** $flag = 0$ **then** $s \leftarrow 2s$
11: **else** $s \leftarrow 2r$, $flag = 0$
12: **else if** $u/2 > v/2$ **then**
13: $u \leftarrow u/2 - v/2$
14: **if** $flag = 0$ **then** $s \leftarrow 2r$
15: **else** $s \leftarrow 2s$
16: $r \leftarrow r + s$, $flag = 1$
17: **else if** $v/2 \geq u/2$ **then**
18: $v \leftarrow u/2 - v/2$
19: **if** $flag = 0$ **then** $s \leftarrow 2s$
20: **else** $s \leftarrow 2r$
21: $r \leftarrow r + s$, $flag = 0$
22: **end if**
23: $k = k + 1$
24: **end while**
25: $r \leftarrow r - s$
26: **if** $r < 0$ **then** $r \leftarrow r + q$
27: **return** $z = r$, k

Algorithm 20 Montgomery Inversion Algorithm: Phase 2

Input: z, k from phase 1
Output: $z = x^{-1}2^{2l} \pmod{q}$

1: **Initialise:** $r \leftarrow z$
2: **for** $i \leftarrow 1$ **to** $2l - k$ **do**
3: $r \leftarrow 2r$
4: **if** $r \geq q$ **then** $r \leftarrow r - q$
5: **end for**
6: **return** $z = r$

done relatively efficiently. However, using other coordinate systems in an FPGA implementation allows for parallel finite field multiplications to be used; thus, a performance increase may be achieved. It is therefore advantageous to analyse the performance of each coordinate system, while varying the number of parallel computations that are used.

The Affine, Jacobian, and ML(X,Y) (Alg. 13) algorithms were implemented using the test platform from Section 4.2. A Montgomery point scalar multiplication algorithm was used for both the affine and Jacobian coordinate systems. The ML(X,Y) (Alg. 13) algorithm was chosen as it has been shown in [5] to have the best performance and lowest energy requirements of the co-Z algorithms when implemented in hardware. A full analysis of implementing co-Z algorithms on an FPGA can be found in [5], where the co-Z algorithms are compared in terms of performance and power consumption. All results are for a field size of 256 bits as this represents a medium level of security that should offer protection for at least the next 30 years [95].

The algorithms were implemented with a varying number of multiplication units, one addition unit, and one subtraction unit. Only the core operations of the algorithms were implemented, as they constitute the vast majority of the computation time; therefore, no inversion unit was required for the Jacobian or co-Z algorithms. In the case of affine coordinates, however, the circuit also required an inversion unit. The implementation results are shown in Table 4.1 and graphed in Figure 4.4. The results for the affine coordinate system in Figure 4.4 are for when 1 multiplication unit is present in the design.

Design	Mult's	slices	slice Reg's	slice LUTs	BRAM	Max. Freq. (MHz)	Time (ms)
Affine Mont. Ladder	1	2472	5008	7071	9	80.16	9.46
Jacobian Mont. Ladder	1	1840	3694	4698	9	83.65	15.68
Jacobian Mont. Ladder	2	2059	4473	5998	10	79.43	8.99
Jacobian Mont. Ladder	3	2307	5250	6776	9	80.27	6.42
Jacobian Mont. Ladder	4	2702	6028	7817	9	80.24	4.77
ML(X,Y) (Alg. 13)	1	1647	3701	4722	10	78.90	12.52
ML(X,Y) (Alg. 13)	2	2131	4479	6019	9	78.92	6.63
ML(X,Y) (Alg. 13)	3	2301	5256	6797	9	79.20	4.92
ML(X,Y) (Alg. 13)	4	2618	6034	7841	9	79.41	4.08

Table 4.1: ECC area results.

The results show that when 1 multiplication unit is present in the design, the affine coordinate system provides the best performance. This is to be expected, as the number of operations required for affine coordinates is much less than for Jacobian or ML(X,Y) (Alg. 13). However, if the number of multipliers is increased, there are no multiplications in the affine algorithms that can be performed in parallel; therefore,

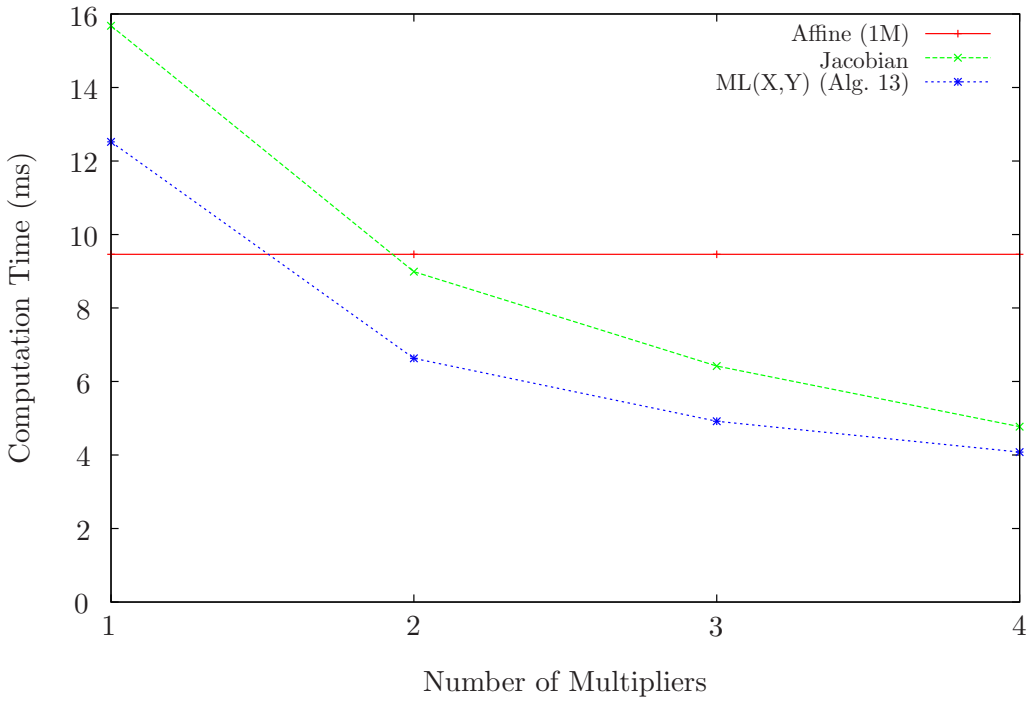


Figure 4.4: Timing results.

affine coordinates cannot benefit from extra multiplication units. Both Jacobian and ML(X,Y) (Alg. 13) algorithms have many multiplications that can be performed in parallel, and as a result, the performance quickly exceeds that of affine coordinates. The ML(X,Y) (Alg. 13) algorithm performs best for every number of multiplication units above 1. The increase in the amount of block RAM required for the ML(X,Y) (Alg. 13) design with 1 multiplier, and the Jacobian design with 2 multipliers, is a result of how the width and number of ROM instructions change for each design. The number of RAM locations required is also a factor. As the number of multipliers increases, the number of RAM locations required also increases. An increase in the number of multipliers also requires the width of ROM instructions to increase, but simultaneously, the number of ROM locations that are required, to store the instruction set, is reduced. As a result, the number of BRAMs required does not increase linearly with the number of multiplication units in the design.

Table 4.2 shows the characteristics of how each algorithm utilises the multiplication units. The Mult. Eff. column indicates the efficiency with which the multipliers are used i.e., 100% efficiency occurs when, at every multiplication stage, all of the multipliers are being used in parallel. A low efficiency indicates that some of the

multipliers are left idle for periods during the algorithm, as certain multiplications cannot be performed in parallel. When a single multiplication unit is present, the efficiency is always 100%. All of the results shown in Table 4.2 are for the core operations of each algorithm. The Affine PA_PD result is for a point addition followed by a point doubling in affine coordinates, and Jacobian PA_PD, the corresponding Jacobian coordinate version. The ZACAU′ (Alg. 30) algorithm is the operation used in the main loop of the ML(X,Y) (Alg. 13) algorithm.

Algorithm	Mult's	Mult. clk's	Mult. Eff.(%)	other clk's	total clk's
Affine PA_PD	1	1806	100	1151	2957
Jacobian PA_PD	1	4902	100	267	5169
Jacobian PA_PD	2	2580	95	240	2820
Jacobian PA_PD	3	1806	90	231	2037
Jacobian PA_PD	4	1290	95	225	1515
ZACAU′ (Alg. 30)	1	3612	100	276	3888
ZACAU′ (Alg. 30)	2	1806	100	255	2061
ZACAU′ (Alg. 30)	3	1290	93	249	1539
ZACAU′ (Alg. 30)	4	1032	87	246	1278

Table 4.2: Clock cycles per sub-algorithm.

Figure 4.5, illustrates how the algorithms perform when both occupied area, and computation time is taken into account. This is done by multiplying the area (slices), with the computation time (ms). This method of comparing the algorithms takes into account the efficiency of the design. In many cases, doubling the area of a circuit does not result in a doubling of performance and this will be reflected in the Area $\times$ Time results. A lower area-time product signifies a more efficient use of FPGA resources. The co-Z coordinates, ZACAU′ (Alg. 30), are again the best performing system in this metric. It can be seen from Figure 4.5 that the addition of a second multiplier causes a sharp increase in performance with respect to the increase of area, however, the addition of a third and fourth multiplier does not result in the same increase in performance, with respect to area. This is caused by a decrease in how efficiently the multipliers are being used, as was shown in Table 4.2.

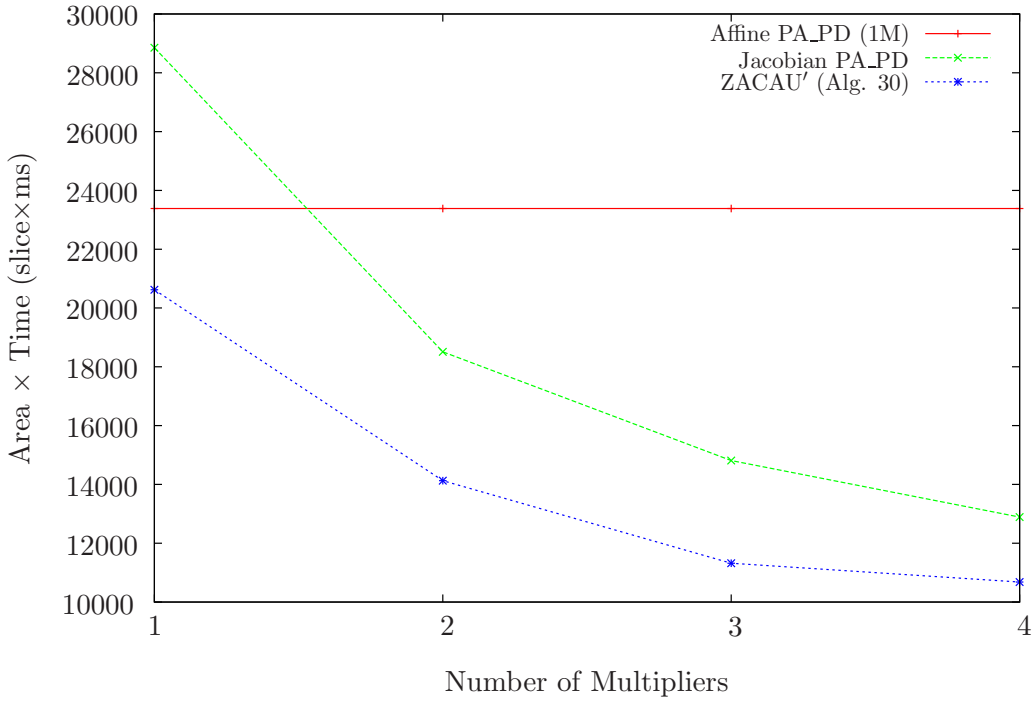


Figure 4.5: Area-time product.

4.4 Applications of Elliptic Curves in TLS

In the previous sections, the implementation of elliptic curve arithmetic on FPGAs was examined. The co-Z algorithms have been identified as the fastest and most efficient. In this section, the use of these algorithms at the protocol level will be discussed. First, the use of ECC in the TLS protocol will be investigated in order to identify the components that will be required to construct a coprocessor for ECC at the protocol level.

In [14], the various elliptic curve based extensions for TLS are defined. The two key exchange algorithms of interest here are; ECDH_ECDSA which uses fixed *Elliptic Curve Diffie-Hellman* (ECDH) keys and digital certificates signed with ECDSA; and ECDHE_ECDSA which uses ephemeral (i.e., short term) ECDH keys and digital certificates signed with *Elliptic Curve Digital Signature Algorithm* (ECDSA). In both cases, the server possesses a certificate with a fixed public key. During the handshake process of ECDHE_ECDSA, the server sends both a certificate signed with ECDSA and an ephemeral public key in the *ServerKeyExchange* message. This *ServerKeyExchange* message is signed with the private key that corresponds to the server's certificate. In the case of ECDH_ECDSA, the server uses only a certificate with a fixed public key

and no *ServerKeyExchange* message is sent.

For both ECDH_ECDSA and ECDHE_ECDSA, the client responds with an ephemeral public key in the *ClientKeyExchange* message. Client authentication is possible if a client also possesses a certificate, however, this is not a common setup.

To use ECDH_ECDSA and ECDHE_ECDSA, both the client and server require the ability to perform scalar multiplication of elliptic curve points and verify ECDSA signatures. If ECDHE_ECDSA is being used, the server also requires the ability to sign the *ServerKeyExchange* message using ECDSA. The ECC processor presented in this chapter, therefore, contains dedicated hardware capable of performing elliptic curve point multiplications for ECDH, signing of a message with ECDSA and verification of ECDSA signatures. The point multiplication methods used for each algorithm were chosen based on the results of the comparison from Section 4.3. An introduction to each protocol is given in the following sections.

4.4.1 Elliptic Curve Diffie-Hellman Key Exchange (ECDH)

The public-key protocol described in Section 2.6 can be used to encrypt messages sent between two entities over an unsecured channel. However, the encryption algorithm used, ξ, is generally more computationally intensive, per bit, than the block cipher or stream cipher type algorithms used in private-key cryptography. Therefore, a combination of private-key and public-key cryptography can be used to achieve secure communications with a higher throughput than that of public-key alone. A public-key algorithm, such as the Diffie-Hellman key exchange [30], can be used to set up a shared secret between two communicating parties. This shared secret can then be used as the key for the encryption algorithm in the private-key protocol.

The security of the Diffie-Hellman key exchange relies on the intractability of the discrete logarithm problem and can be extended to use elliptic curves, where the scalar multiplication of kP provides the intractable problem. First, the two communicating entities, Alice and Bob, must agree upon the domain parameters of the protocol, (q, A, B, G, n); where q is the field size; A and B define the short Weierstraß equation of the curve and $G = (x_g, y_g)$ specifies a point of order n on the curve. Alice and Bob must then each generate a private/public key pair. Alice chooses at random a private key $A_{priv} \in \{2, 3, ..., n-1\}$, and generates her public key such that $A_{pub} = A_{priv}G = (x_A, y_A)$. Bob performs the same process to generate his key pair, $B_{priv} \in \{2, 3, ..., n-1\}$ and $B_{pub} = B_{priv}G = (x_B, y_B)$. Alice and Bob then exchange their public keys. If Alice then computes $S_A = A_{priv}B_{pub}$ and Bob com-

putes $S_B = B_{priv}A_{pub}$, both Alice and Bob should have arrived at the same value, $S_A = S_B = B_{priv}(A_{priv}G) = A_{priv}(B_{priv}G)$, which is a point on the elliptic curve. Alice and Bob can then use the x coordinate of this point to generate the key for a symmetric-key encryption protocol.

4.4.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is the elliptic curve based version of DSA. As from Section 4.4.1, the domain parameters of the curve are given by (q, A, B, G, n).

The algorithms for generating and verifying elliptic curve digital signatures are shown in Algorithms 21 and 22, respectively. The most computationally intensive operation in Algorithm 21 is the calculation of $R = kG$, and in Algorithm 22 it is the computation of $X = u_1G + u_2\theta$. Signature verification is the more computationally intensive of the two algorithms as it requires two elliptic curve point scalar multiplications to be performed, as opposed to only one in the signature generation algorithm.

The security of the signature generation algorithm relies on several assumptions: that the random number k is generated securely; that the hash function used is cryptographically secure; and that the *Elliptic Curve Discrete Logarithm Problem* (ECDLP) is intractable. If any of these assumptions fail, it would be possible for an adversary to forge signatures.

If an entity wishes to sign a message they must first generate their private/public key pair. The individual's private key is chosen at random such that $d \in [1, n-1]$. Their public key can then be generated as $\theta = dG$. The resulting key pair (d, θ) is used in the signature generation and verification algorithms.

The generation of the integer k must be done using a cryptographically secure random number generator, such as that described in Chapter 6. Step 3 of Algorithm 21 can be done using any coordinate system as long as the x coordinate of the resulting point R is converted back to affine coordinates. The hash function used in step 2 of Algorithm 21 must be a cryptographically secure hash function such as those described in [87]. The overall system for implementing Algorithms 21 and 22 is described in Section 4.6.

To show that signature verification works, it must be shown that the x coordinate of the point X in Algorithm 22 equals the signature parameter r. First, from Algorithm 21

$$s = (k^{-1}(e + dr)) \pmod{n}, \tag{4.4}$$

Algorithm 21 Elliptic Curve Digital Signature Generation

Input: private key d, message m, domain parameters (q, A, B, G, n)
Output: Signature (r, s)

1: generate random integer $k \in [1, n-1]$
2: compute $e = \mathcal{H}(m)$
3: compute $R = kG = (x_1, y_1)$
4: set $r = x_1 \pmod{n}$. If $r = 0$ return to step 1
5: compute $k^{-1} \pmod{n}$
6: compute $s = (k^{-1}(e + dr)) \pmod{n}$
7: the signature for message m is then (r, s)

Algorithm 22 Elliptic Curve Digital Signature Verification

Input: signature (r, s), domain parameters (q, A, B, G, n), senders public key θ
Output: accept or reject signature

1: verify $r, s \in [1, n-1]$
2: compute $e = \mathcal{H}(m)$
3: compute $\lambda = s^{-1} \pmod{n}$
4: compute $u_1 = e\lambda \pmod{n}$
5: compute $u_2 = r\lambda \pmod{n}$
6: compute $X = u_1 G + u_2 \theta$
7: if $X = \mathcal{O} \rightarrow$ reject signature
8: else convert x_1 of X to an integer $c \pmod{n}$ accept signature if $c = r$

rearranging gives

$$k = (s^{-1}(e + dr)) \pmod{n}, \tag{4.5}$$
$$= s^{-1}e + s^{-1}dr \pmod{n}, \tag{4.6}$$
$$= \omega e + \omega dr \pmod{n}, \tag{4.7}$$

and

$$u_1 = e\omega \pmod{n}, \tag{4.8}$$
$$u_2 = r\omega \pmod{n}. \tag{4.9}$$

Therefore,

$$k = u_1 + u_2 d \pmod{n}. \tag{4.10}$$

The message signer's public key is given by $Q = dG$; therefore,

$$X = u_1G + u_2Q = u_1G + u_2dG = (u_1 + u_2d)G = kG \pmod{n}. \tag{4.11}$$

It follows that if $x_1 = r$, the signature is correct.

4.4.3 DPA resistant ECDSA

When implementing the signature generation algorithm the designer must take into consideration which operations are susceptible to side channel attack. The use of affine coordinates and the Montgomery ladder is naturally resistant to SPA attacks but is still susceptible to DPA attacks [65]. Using the co-Z formulæ also allows for the use of projective point randomisation which is a countermeasure against DPA attacks. However, since the value of k is random and changes for every signature generated, this part of the ECDSA algorithm is not susceptible to DPA attacks. For DPA attacks to be applicable the key must remain constant for many executions of the algorithm.

One part of the ECDSA key generation that can be susceptible to first order DPA attacks is the computation of $s = (k^{-1}(e + dr)) \pmod{n}$. The message signer's private key d remains constant and an attacker would have knowledge of r; hence, this operation can be attacked [80]; specifically the multiplication $d \times r$. To protect this operation against a DPA attack, the message e and the private key d should be masked in some way. Since k is random and changes with every signature generated, it can be used to mask the operands at the expense of an extra multiplication. First, the value $\phi = k^{-1}d \pmod{n}$ is computed. s can then be computed as $s = (ek^{-1} + \phi r) \pmod{n}$, which results in the same value without d and r being multiplied together directly. This removes any correlation between the power consumption of the device and the value of d.

4.4.4 Simultaneous multiple point multiplication

In line 6 of Algorithm 22, two ECC point scalar multiplications are required. The most basic way of accomplishing this task is to calculate u_1G and u_2Q separately and then add the result. There is, however, a more efficient technique for performing this computation, known as Shamir's trick, see [24] for details. Algorithm 23 shows how the computation is performed. First, the value $(G + Q)$ is pre-computed, the algorithm then proceeds in the same way as the double and add algorithm, except for the fact that both u_1 and u_2 are simultaneously used to determine which operations are performed. Computing u_1G and u_2Q separately would result in at least $2l$ point doublings and up

to $2l$ point additions. Algorithm 23, however, will only require l point doublings and up to $l+1$ point additions. This results in a saving in computation time at the cost of only 1 extra register for storing the point $(G+Q)$.

Algorithm 23 Simultaneous multiple point multiplication

Input: $G, Q \in E(\mathbb{F}_q)$;
$u_1 = \sum_{i=0}^{a-1} k_i 2^i;\ u_2 = \sum_{i=0}^{b-1} k_i 2^i;\ l = max(a, b)$;
Output: $X = u_1 G + u_2 Q \in E(\mathbb{F}_q)$

1: pre-compute $G+Q$
2: $X = \mathcal{O}$
3: **for** $i = l-1$ down to 0 **do**
4: $X = 2X$;
5: **if** $u_{1_i} = 1$ AND $u_{2_i} = 0$ **then**
6: $X = X + G$
7: **else if** $u_{1_i} = 0$ AND $u_{2_i} = 1$ **then**
8: $X = X + Q$
9: **else if** $u_{1_i} = 1$ AND $u_{2_i} = 1$ **then**
10: $X = X + (G+Q)$
11: **end if**
12: **end for**
13: **return** X

4.5 Related Work

In [43], Glas et al. detail a design for accelerating ECC operations in a vehicle-to-vehicle communication setting. The design is capable of performing hashing, ECDSA signature generation and verification, and general ECC operations. The implementation uses an ALU that was introduced in [42], where, the multiplication unit uses the interleaved multiplication method from [15]. The addition and subtraction units are implemented in a similar method to those introduced in Section 4.2.1. The ALU is capable of performing $\mathbb{F}_q$ operations with bitlengths of 224 and 256 bits. The ECC operations are implemented using affine coordinates, where signature generation and verification are done using Algorithms 7 and 23 respectively. The division unit, shown in Figure 4.6, is used for field inversion and registers R1, R2, R3, and R4 are used to store the intermediate results during the processing of an algorithm. The processor is controlled by an FSM. An overview of the design is shown in Figure 4.6.

The design was implemented on a Xilinx XC5VLX110T Virtex-5 FPGA and occupied 14256 LUT/FF pairs. The critical path of the design restricted the clock frequency

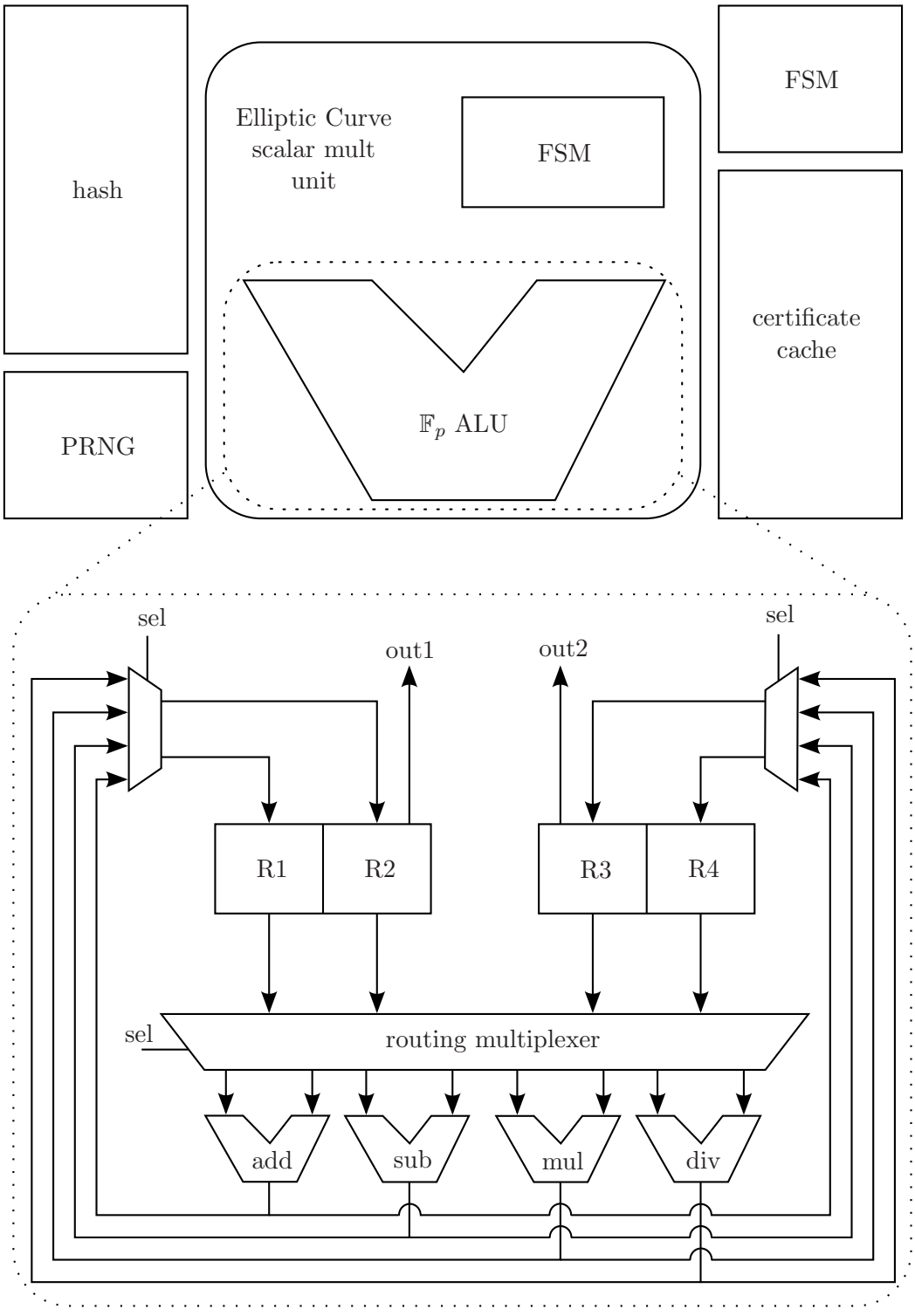


Figure 4.6: Glas et al. design.

to 50 MHz, resulting in an average signature generation time of 7.15 ms and signature verification time of 9.09 ms; a full list of the results can be found in Table 4.3.

The work shows how useful a coprocessor can be in a real world setting, as the design was implemented in a fully working system and showed a performance gain over several microcontroller implementations that it was compared with. The coprocessor did not achieve performance comparable to that of high end processors usually found in desktop computers, such as the Intel Pentium D or Intel Core 2 Duo. However, these types of processors generally have a much higher power consumption than FPGA based designs, and are also targeted at much larger applications than embedded devices.

4.6 ECDSA Processor Architecture

In this section, a design for an ECDSA coprocessor will be introduced. The goal of the design is to be usable as part of a larger coprocessor for acceleration the TLS protocol. When used in the TLS coprocessor, the ECDSA coprocessor will take instructions from a GPP. The GPP will be used to implement the non-computationally intensive operations such as parsing incoming TLS records. It is important when designing the instructions for the coprocessor to offload as many finite field multiplications as possible. The coprocessor is much more efficient at performing finite field multiplications; thus, the correct partitioning of operations between the GPP and the coprocessor will maximise the performance of the overall design. Therefore, all pre-computations are performed by the coprocessor. The GPP should be able to send information retrieved from TLS messages directly to the coprocessor, without having to perform any computationally intensive operations on them.

The algorithms presented in Section 4.3 contained only the core operations. The coprocessor, however, requires data to be sent to it in the Montgomery domain. For the results of point multiplications to be useful in the TLS protocol, they must be in affine coordinates and standard domain. The computation time for co-Z coordinates was shown to be optimal for the processor architecture shown in Section 4.2. Therefore, the algorithms presented were modified so that all data sent to the coprocessor is in standard domain and affine coordinates. The coprocessor then transforms the values into the Montgomery domain and co-Z coordinates, using a pre-computation phase in the FSM. The scalar multiplication algorithm is then performed and upon completion a post-computation phase is started, where the resultant ECC point is converted back to standard domain and affine coordinates. These pre-computation and post-computation stages are each roughly equivalent to one iteration of the main loop of the scalar mul-

tiplication algorithm. Considering that the main loop is executed 253 times for a field size of 256 bits, the main loop of the algorithm is by far the dominating factor in the performance of the design.

The circuit for the ECDSA processor is shown in Figure 4.7 and is based around the design of Byrne et al. [20]. One modification that has been made is the addition of extra control FSMs. The original architecture presented by Byrne et al. is only capable of performing one type of point scalar multiplication algorithm, as only one FSM and ROM instruction set is present in the design. The addition of extra FSMs allows for a more flexible design, as the RAM and ALUs can be shared between multiple algorithms. When the control unit receives input data, it loads the data into block RAM. The control unit also enables the correct FSM that will perform either a point multiplication, signature generation, or signature verification operation. Each FSM contains a ROM instruction set that defines the sub algorithms that it will perform (i.e., DBLU, ZADDC etc..). The ECDSA processor also contains a number of ALUs, such as a 256 bit adder and subtractor; a multiplication unit; and a field inversion unit, as were discussed in Section 4.2. The FSMs also have the ability to select which modulus is used in each ALU. This is required, as in Algorithms 21 and 22, the ECC scalar multiplications are done mod q, while all other finite field arithmetic is performed mod n. All intermediate values are stored in RAM.

4.7 Implementation Results

Table 4.3 shows the area and timing results for the ECDSA processor, implemented on a Xilinx Virtex 5 XC5VLX110T FPGA. For comparison, the results from [43] are also included.

Design	slices	slice Reg's	slice LUTs	LUT/FF pairs	BRAM	Max. Freq. (MHz)	P_mul (ms)	Sig. Gen. (ms)	Sig. Ver. (ms)
1 mult	3974	9065	10366	13519	14	78.29	12.515	12.599	17.636
2 mult	3905	9845	11305	14092	14	78.14	6.644	6.694	9.166
3 mult	4508	10363	12857	16108	14	74.33	5.218	5.263	8.044
4 mult	5347	11403	13900	18380	13	74.51	4.325	4.364	7.605
Glas et al.	-			14256	0	50	7.14	7.15	9.09

Table 4.3: ECDSA processor implementation results.

Figures 4.8 and 4.9 show how the performance varies with the number of multipliers in the circuit. A sharp increase in performance can be seen after the addition of a second multiplier. The addition of further multiplier units has much less of an impact on the performance of the design as these extra multipliers cannot be fully utilised in parallel.

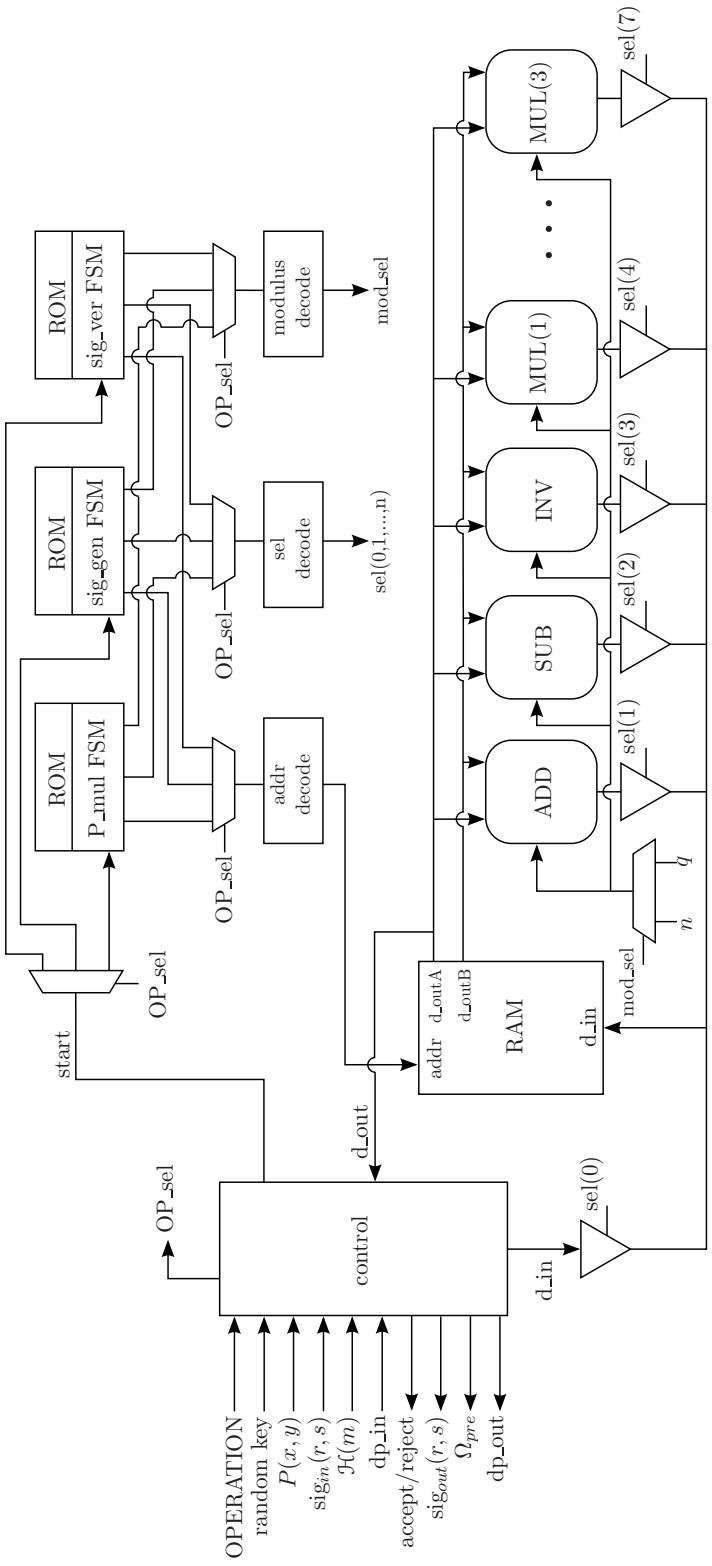


Figure 4.7: ECDSA processor.

As these operations are based around the algorithms shown in Table 4.2, it can be seen that the algorithms from this section follow the same trend of loss in multiplier efficiency.

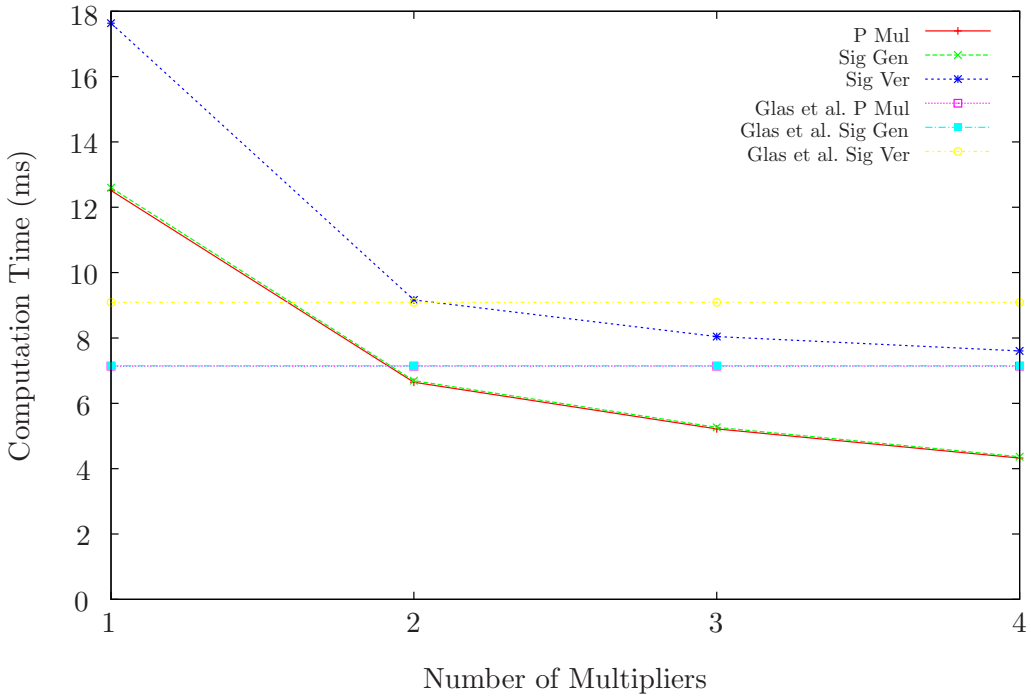


Figure 4.8: ECDSA processor timing results.

It can also be seen from Figures 4.8 and 4.9 that for each multiplier added, there is an increase in area and there comes a point when adding extra multipliers does not improve performance enough to justify the extra area that they consume. This can be seen in Figure 4.9, where the area time product is given. It can clearly be seen that the addition of the second multiplier decreases the area-time product significantly, however, the addition of any further multiplier units has a very small impact on reducing the area-time product. In the case of the signature generation algorithm, having more than three multipliers causes the area-time product to increase. For this reason, it would appear that three multiplier units is the optimal solution. With two or three multipliers in the circuit, the area-time product is lower than that of the Glas et al. design.

One important result is the critical path of the design as the number of multipliers is increased. Due to the fact that the multipliers are in parallel, there is very little impact on the critical path. This is of importance as the coprocessor must run at the same frequency as the GPP it's connected to; thus, the ECDSA processor will generally

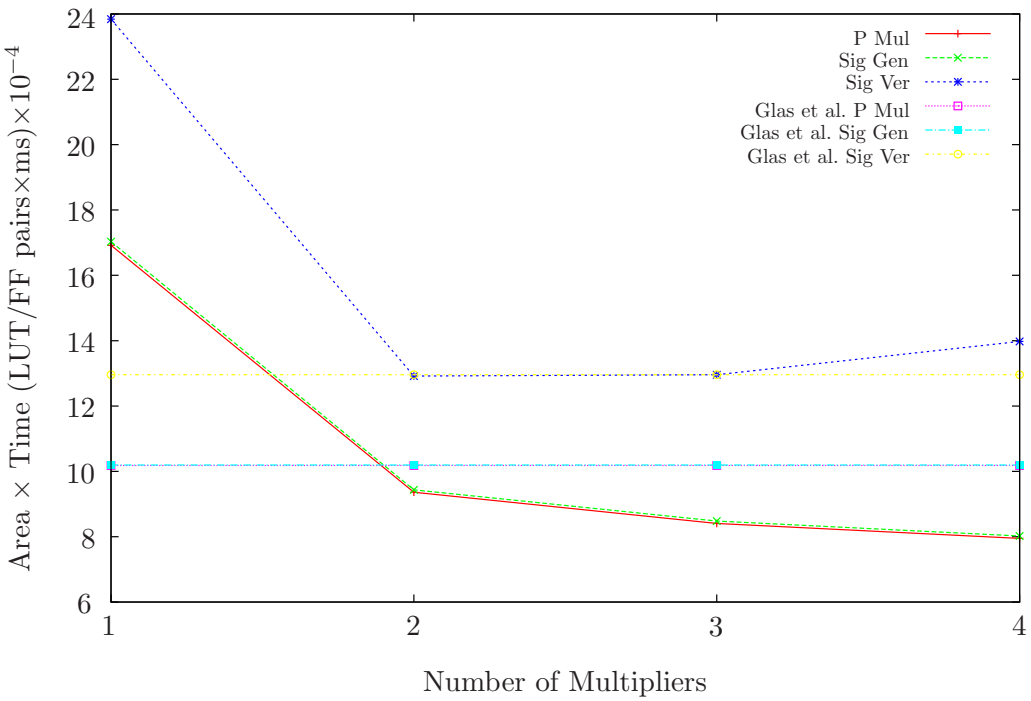


Figure 4.9: ECDSA processor area time product.

be the circuit that constrains the maximum frequency of the overall system.

4.8 Discussion

In this chapter, the design of an ECC processor was introduced. Various ECC algorithms were compared for performance on the architecture. The optimal algorithms, for the architecture presented, were found to be a version of the projective coordinates using co-Z algorithms. As a result, these algorithms were extended for use as part of the ECDSA signature generation and verification algorithms. An architecture for an ECDSA processor was then introduced.

The architecture presented in this chapter can be used to perform the public-key based operations required by the TLS protocol. Public-key algorithms are the most computationally intensive of all the algorithms supported by TLS. For this reason, an efficient implementation is very important, as it will be the limiting factor on the performance of the overall system. It has been shown that through changing the number of multipliers in the circuit, various levels of performance can be achieved. The best solution appears to be the use of three multiplier units, as this provides a good trade-off

between efficient use of resources and performance.

Although public-key algorithms are very computationally intensive, they are not the only operations that a coprocessor would be required to perform. A hash function is required for the signature generation algorithm, key generation functions, and to provide message integrity. All the operations presented so far have been required by the TLS handshake protocol. Once a shared *master secret* has been established, encryption and hashing functions are the core operations used by the TLS record layer to provide secure data transmission. In the next chapter the implementation of hash functions will be discussed.

Chapter 5

Hash Functions and their Applications

5.1 Introduction

Hash functions are cryptographic primitives that are used to generate a fingerprint, known as a message digest, for a string of data. If this data is then transmitted, the receiver should be able to calculate a matching fingerprint; thus indicating that the data has not been altered in transit. Hash functions have many other uses in cryptographic protocols such as the generation of MACs, which provide message authenticity, or as part of the digital signature algorithm discussed in Section 2.6.1.1.

Hash functions are not just implemented as standalone algorithms and are frequently used in the construction other cryptographic functions. Examples include the TLS pseudorandom function described in Section 5.4.2 or as a post-processing mechanism for TRNGs, which will be discussed in Section 6.8.

Many hash function constructions have been presented in literature over the past few years, several of which were submissions to the NIST *Secure Hash Algorithm 3* (SHA-3) competition. With the wide variety of different architectures available, comparing the performance can be a difficult task. In this chapter, the application of these hash functions as part of the TLS protocol will be discussed. A generic hardware wrapper will then be introduced that is believed to allow for a fair comparison of different hash function designs to be made. Finally, several recent hash designs will then be compared on an FPGA platform, using this wrapper.

5.2 Hash Function Design

A cryptographic hash function is an algorithm that maps an arbitrary length string of bits, to a fixed length string of bits. Ideally, the algorithm is a computationally efficient one way function, where computing the hash of a message block requires minimal computational power, while computing the message for a given hash value should be computationally infeasible [78, Chapter 9].

A generic iterative hash function, shown in Figure 5.1, consists of a core operation referred to as a compression function f_c. The message to be hashed, M, is padded and then fragmented into n blocks of length x, where x is the input bitlength required by f_c. Each n bit block of the message is denoted M_i. The compression function iterates on the input message block for a defined number of rounds. The output is then fed back into the input of f_c, along with the next message block M_i. This process continues until the last message block has been processed. The final output of f_c is then taken as the message digest. Depending on the specification of the hash function, the final application of f_c sometimes differs from all previous applications; it is therefore denoted here as f'_c. Several specific subsets of this iterative structure exist such as the Merkle-Damgård construction [27, 79] used by the *256 bit Secure Hash Algorithm* (SHA256), or the *Hash Iterative Framework* (HAIFA) construction [12] which uses a counter and salt value, along with each message block, as inputs to the compression function. Two examples of HAIFA based hash functions are ECHO [8] and SHAvite-3 [13], both of which were knocked out in round 2 of the SHA-3 competition.

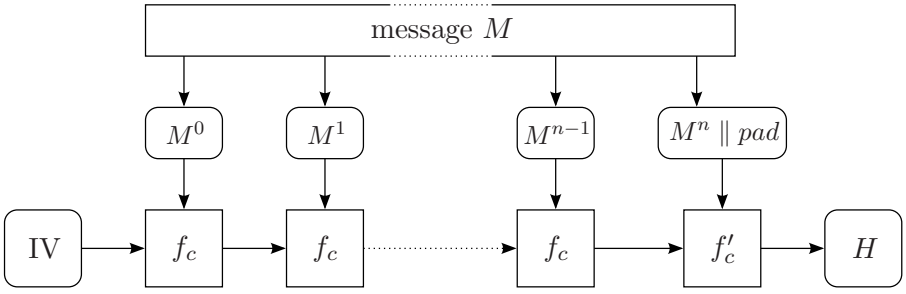


Figure 5.1: Generic hash function architecture.

5.2.1 Implementation Options

Although many hash functions follow the generic structure of Figure 5.1, the complexity of f_c is dependent on the specification of the hash function. Therefore, certain design techniques can be used to alter the critical path and area of f_c in hardware.

- Loop unrolling involves feeding the output of one function into the next without an intermediate register. This decreases the number of clock cycles required by the function, but increases the area and critical path of the design.

- Parallelisation may be used when a function is used repeatedly and the input for each iteration is not dependent on any of the previous results. In this case the hardware can be replicated; thus, reducing the number of clock cycles required, at the expense of extra area.

- Pipelining is used to shorten the critical path of a design, at the expense of extra clock cycles. The resulting design will also require more memory resources.

These different techniques may be applied to the compression function as a whole, or to operations within the compression function.

5.3 Hash Function Usage

One application of hash functions, as part of a cryptographic protocol, is to provide message integrity. The hash function $\mathcal{H}$ is applied to the message m to produce a message digest d_m, such that $\mathcal{H}(m) = d_m$. In the case of TLS, this process takes place prior to the message being passed through an encryption function. The message can then be transmitted, along with d_m. The receiver can apply the same process to m and should arrive at the same result d_m. If any part of the message had been altered during transmission, there is a very high probability that the receiver would not calculate the same message digest d_m.

A cryptographically secure hash function should have the following properties:

- Preimage resistance: It should be computationally infeasible to calculate m, given d_m.

- Second Preimage resistance: given a message m_1, it should be computationally infeasible to find another message m_2, such that $\mathcal{H}(m_1) = \mathcal{H}(m_2)$, where, $m_1 \neq m_2$.

- Collision Resistance: It should be computationally infeasible to find two messages, where, $m_1 \neq m_2$ and $\mathcal{H}(m_1) = \mathcal{H}(m_2)$.

5.4 Hash Functions and TLS

As mentioned previously, hash functions form an important part of cryptographic protocols, by providing a mechanism to ensure message integrity and as building blocks for other functions. In this section, a description will be given as to how hash functions are used to provide different services for the TLS protocol.

During the handshake phase of the TLS protocol, a hash function is required by several different operations. The derivation of the key block is done using a pseudorandom function, based on a hash function. A hash of all of the handshake messages is required as part of the *Finished* messages that are exchanged. A hashing operation is necessary in both the generation and verification of ECDSA signatures. The HMAC function used to generate a MAC for messages, that are to be encrypted, is also constructed from a hash function.

5.4.1 HMAC Function

The HMAC function [68] is a hash function based construction that is used to provide message integrity and message authenticity. The algorithm is used to calculate a MAC and combines the hash of a message with a key. The construction of the HMAC function is given by

$$\mathrm{HMAC}_k(m) = \mathcal{H}(k \oplus opad \parallel \mathcal{H}(k \oplus ipad \parallel m)), \tag{5.1}$$

where,

- $\mathcal{H}$ is a cryptographic hash function.
- k is the key. If k is longer than the input block length of $\mathcal{H}$, then the value $k = \mathcal{H}(k)$ is used.
- $\oplus$ denotes an XOR operation.
- $\parallel$ denotes concatenation.
- *opad* is the hexadecimal value $5c$, repeated until *opad* equals the input block length of $\mathcal{H}$.
- *ipad* is the hexadecimal value 36, repeated until *ipad* equals the input block length of $\mathcal{H}$.

The HMAC function is used in the TLS protocol as part of the pseudorandom function, Section 5.4.2, and is also used to append a MAC to any message, prior to it being encrypted.

5.4.2 TLS Pseudorandom Function

A pseudorandom function is described in Section 5 of [29]. The TLS pseudorandom function is used to generate the 48 byte *master secret* Ω_m, and also any keys required for the HMAC or data encryption algorithm. It is also used in calculating data for the *Finished* messages sent during the TLS handshake protocol.

The pseudorandom function takes as input, a *secret* (usually the shared secret that was set up between the two communicating parties), a random value referred to as a *seed*, and an *American Standard Code for Information Interchange* (ASCII) encoded string of characters denoted *label* e.g., "master secret". The pseudorandom function then uses a HMAC function to generate an arbitrary amount of data.

The pseudorandom function denoted TLS_PRF is defined as follows:
Firstly, a function based on HMAC is defined to expand a secret and a seed into an arbitrary amount of data.

$$\begin{aligned} P_{\mathcal{H}}(secret,\ seed) =& \mathrm{HMAC}(secret,\ A(1) \parallel seed) \parallel \\ & \mathrm{HMAC}(secret,\ A(2) \parallel seed) \parallel \\ & \mathrm{HMAC}(secret,\ A(3) \parallel seed) \parallel \cdots, \end{aligned} \tag{5.2}$$

where $\parallel$ indicates concatenation and

$$\begin{aligned} A(0) &= seed, \\ A(i) &= \mathrm{HMAC}(secret,\ A(i-1)). \end{aligned} \tag{5.3}$$

The pseudorandom function is then given by

$$\mathrm{TLS\_PRF}(secret,\ label,\ seed) = P_{\mathcal{H}}(secret,\ label \parallel seed). \tag{5.4}$$

5.4.3 Key derivation

One of the main uses of the TLS_PRF is to generate the keys for use in the TLS handshake and also the *master secret* Ω_m. To generate Ω_m, the operation defined in Equation 5.5 is used. The ASCII encoded string "master secret" along with the *premaster secret* Ω_{pre}, and a 512 bit block of random data Γ, are operated on by the TLS_PRF function. The 512 bit block of random data contains 256 bits generated by the server and 256 bits generated by the client. The *premaster secret* is the result of whichever key exchange algorithm was used during the TLS handshake process. In the case of ECDH the *premaster secret* is the x coordinate of an elliptic curve point.

$$\Omega_m = \text{TLS\_PRF}(\Omega_{pre}, \text{“master secret”}, \Gamma). \tag{5.5}$$

A key block K_B, is required by the key expansion block of encryption/decryption algorithm. The key block is generated from the *master secret* using Equation 5.6, where "key expansion" is an ASCII encoded string.

$$K_B = \text{TLS\_PRF}(\Omega_m, \text{“key expansion”}, \Gamma). \tag{5.6}$$

Not all of the data in the key block is required by all TLS suites. For the two suites implemented in this work only the AES encryption and decryption keys and two secret values for the HMAC operation are required.

5.4.4 Finished Message Calculation

The verification data D_v, is used to ensure that the TLS handshake protocol has completed successfully. Both the client and the server generate D_v and send it in their respective *finished* messages. If the received data matches the generated data, the TLS handshake has completed successfully. The verification data is generated using Equation 5.7. The finished label L_f, is either the ASCII string "client finished" or "server finished" depending which entity is generating the data. H_{mes} is the hash of all of the previously exchanged handshake messages.

$$D_v = \text{TLS\_PRF}(\Omega_m, L_f, H_{mes}). \tag{5.7}$$

5.5 SHA Algorithms

The *Secure Hash Algorithm* (SHA) family of hash functions is standardised by NIST in the *Secure Hash Standard* (SHS) [88]. Several different algorithms are present, with varying levels of security, ranging from 160 to 512 bit hash lengths. Although the most recent standard was published in 2012, the SHA algorithms date back to 1993 when the original *Secure Hash Algorithm 0* (SHA-0) algorithm was introduced [84]. SHA-0, however, was superseded by the SHA-1 algorithm [85] in 1995, due to several weaknesses in its design, and has been declared broken since the publication of [22] in 1998, where a practical attack against the design was introduced. SHA-1 is still part of the SHS [88], however, its use is no longer recommended for critical applications due to the attack presented in [128]. Mounting such an attack using today's hardware would cost several million euros to calculate a single hash collision, however, this figure will drop significantly over the coming years. It is therefore currently recommended to use the *Secure Hash Algorithm 2* (SHA-2) family of hash functions [88] which were introduced by NIST in 2001. As it is a common variant, the 256 bit version, SHA256, is described in Section 5.5.1; larger versions are based on the same construction.

5.5.1 SHA256

The SHA256 algorithm uses a Merkle-Damgård construction; its compression function is shown in Figure 5.2, where

$$\mathrm{Ch}(E, Z, \Theta) = (E \wedge Z) \oplus (\overline{E} \wedge \Theta), \tag{5.8}$$

$$\mathrm{Maj}(A, B, \Gamma) = (A \wedge B) \oplus (A \wedge \Gamma) \oplus (B \wedge \Gamma), \tag{5.9}$$

$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22), \tag{5.10}$$

$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25). \tag{5.11}$$

and $\wedge$ denotes a bitwise logical AND, $\oplus$ denotes bitwise logical XOR and, $(x \ggg n)$ denotes a bitwise rotation of x to the right by n bits.

The input message is split into n 512-bit message blocks, $M^0, M^1, \ldots M^n$. Each message block M^i is expanded, using the the message schedule [88], into 64 32-bit words, W_t for $0 \leq t \leq 63$, where

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \end{cases} \tag{5.12}$$

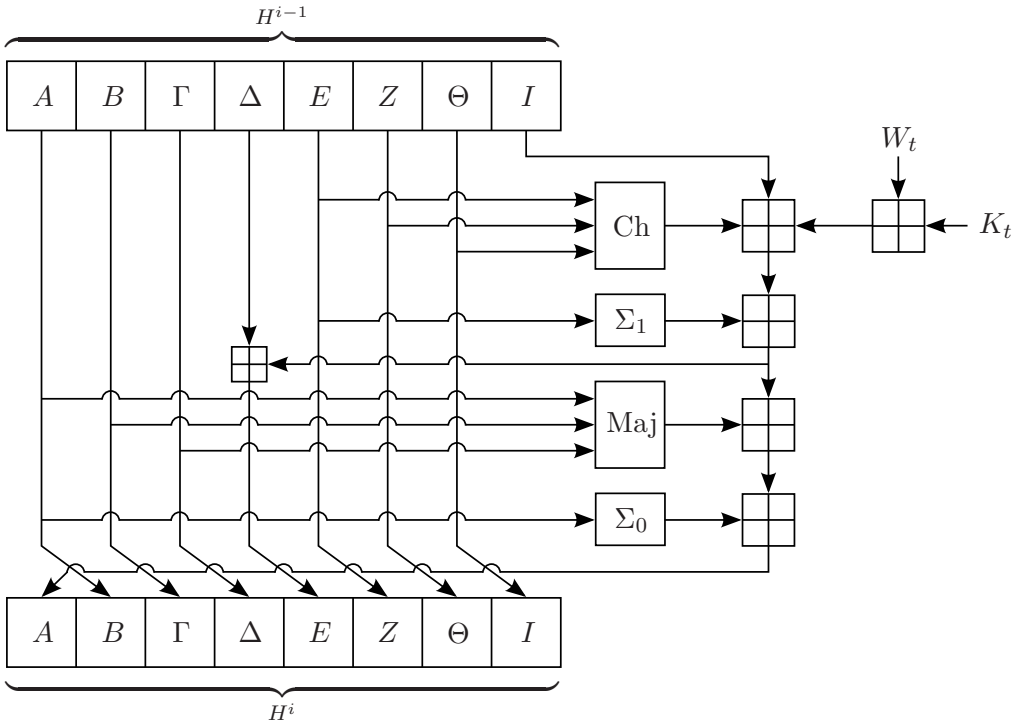


Figure 5.2: SHA256 compression function, f_c.

and

$$\sigma_0(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 22), \tag{5.13}$$

$$\sigma_1(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10). \tag{5.14}$$

Where $(x \gg n)$ denotes a bitwise shift of x to the right by n bits. Each W_t is added modulo 2^{32} to a set of 64 eight bit constants K_t, for $0 \leq t \leq 63$.

The most computationally intensive operations of the compression function are modulo 2^{32} additions, denoted by $\boxplus$ in Figure 5.2. The output hash of each round H^i, is fed back in as the input to the next. This follows the iterative structure from Figure 5.1, where the IV in Figure 5.1 corresponds to K_t for the SHA256 algorithm. A circuit for the algorithm was implemented such that each round of the compression function takes 1 clock cycle.

5.6 SHA-3 Competition

The construction of the SHA-2 algorithm is similar to that of SHA-1, for which significant weaknesses have been discovered. Although no practical attack against SHA-2 has yet been found, there were concerns that, due to its similarity to SHA-1, an exploit might be found in the coming years. Due to these security concerns, NIST decided to hold a competition for a new SHA standard [92]. Designers were asked to submit their designs, which would then analysed by the academic community and by NIST themselves. The competition initially received 64 submissions; 51 of these progressed to round 1 of the competition. Following a year of scrutiny from the academic community, resulting in weaknesses being found in many submissions, 14 designs were chosen to progress to round 2 of the competition, where a year long period was given to more thoroughly analyse these designs for potential weaknesses and also to compare the performance of the different designs on a wide range of platforms.

In the following sections, several of these designs will be discussed, along with a method to compare their performance in a fair manner. A full comparison of all 14 round 2 designs can be found in [1, 4], which are publications resulting from the work presented in this chapter.

5.7 Blue Midnight Wish

Blue Midnight Wish (BMW) [44] was designed by Gligoroski et al. and was knocked out in round 2 of the SHA-3 competition. The algorithm has four variants of differing bitlengths, BMW224, BMW256, BMW384, and BMW512. Both BMW224 and BMW256 are very similar in design; they take an input message block of size 512 bits and operations in the compression function are performed on 32 bit blocks at a time. BMW384 and BMW512 take an input message block of size 1024 bits and operations are performed on 64 bit blocks. The BMW design is an iterative hash function that uses a wide pipe design for the compression function. A wide pipe design has an internal state size larger than the input block size. In the case of BMW, the algorithm has both a double-pipe stage, where the internal state is twice the size of the input, and a quadruple-pipe, where, the internal state size is four times the size of the input. BMW consists of three core functions, whose structure defines the wide pipe design:

- $f_0 : \{0,1\}^{2m} \rightarrow \{0,1\}^m$
- $f_1 : \{0,1\}^{3m} \rightarrow \{0,1\}^m$

- $f_2 : \{0,1\}^{3m} \rightarrow \{0,1\}^m$

where m is the number of bits in the message block $M^{(i)}$ and the double pipe value $H^{(i-1)}$. Each of $M^{(i)}$, $H^{(i)}$, $Q_a^{(i)}$, and $Q_b^{(i)}$ are represented as sixteen 32 or 64 bit blocks depending on which variant of the hash function is being used. All additions and subtractions in the design are performed modulo 2^{32} for BMW224 and BMW256 and modulo 2^{64} for BMW384 and BMW512. An illustration of the algorithm is shown in Figure 5.3; the design follows the same generic structure as was discussed in Section 5.2.

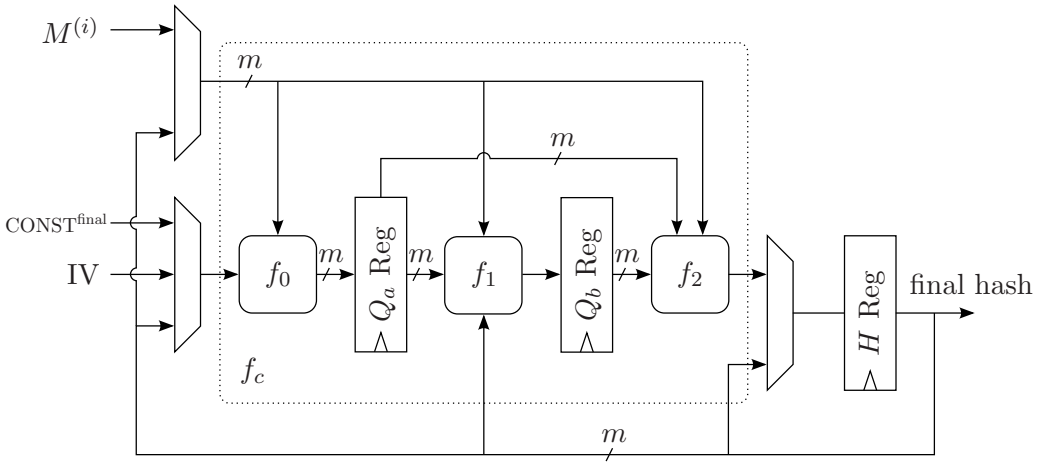


Figure 5.3: BMW hash function.

The three functions f_0, f_1, and f_2 are constructed as follows:

- f_0 takes $M^{(i)}$ and $H^{(i-1)}$ as its inputs and produces the first half of the quadrupled-pipe value $Q_a^{(i)}$. f_0 consists of 80 additions/subtractions. The function also consists of many XORs, bitwise shifts, and rotations.

- f_1 takes $M^{(i)}$, $H^{(i-1)}$, and $Q_a^{(i)}$ and produces the second half of the quadrupled-pipe value $Q_b^{(i)}$. The quadrupled-pipe is then $Q^{(i)} = (Q_a^{(i)}, Q_b^{(i)})$. f_1 is the most complex of the functions performed by BMW. It consists of two sub functions $ExpandRounds_1$ and $ExpandRounds_2$. Together these functions must be performed a total of sixteen times. The specification recommends that $ExpandRounds_1$ be performed twice and $ExpandRounds_2$ be performed fourteen times. This is due to the fact that $ExpandRounds_1$ is a much more complex function than $ExpandRounds_2$.

- The final function f_2 takes $M^{(i)}$, $Q_a^{(i)}$, and $Q_b^{(i)}$ as inputs and produces the new double-pipe value $H^{(i)}$. f_2 consists of XOR, bitwise shift, rotation, and modular addition operations.

The two functions *ExpandRounds*$_1$ and *ExpandRounds*$_2$ allow for modification of the security of the hash function. To increase security, the number of times *ExpandRounds*$_1$ is performed can be increased. Both *ExpandRounds*$_1$ and *ExpandRounds*$_2$ contain sixteen modular addition operations but *ExpandRounds*$_1$ contains much more bitwise shift and rotate operations than *ExpandRounds*$_2$. Both functions use an *AddElement* operation that uses modular additions, subtractions, and rotations to combine a block of the message and of the double-pipe with a predefined set of constants.

A pipelined design was chosen for implementation due to the large amount of modular addition/subtraction operations that need to be performed. A fully unrolled design was tested but its critical path was too long to be efficiently routed on the FPGA. Each of the functions f_0, f_1, and f_2 are separated by a pipeline register. As as a result, one operation of the compression function takes three clock cycles. A diagram of the pipelined design is shown in Figure 5.3.

5.8 Hamsi

Hamsi was designed by Özgül Küçük and was also eliminated in the second round of the SHA-3 competition; its specification is given in [101]. Hamsi allows for message digest sizes of 224, 256, 384, and 512 bits. Hamsi224 and Hamsi256 are very similar in construction, as are Hamsi384 and Hamsi512, differing only in initialisation values and the final truncation. The message input size and state size is 32 and 512 bits for Hamsi224/Hamsi256 and is 64 and 1024 for Hamsi384/Hamsi512 respectively. The design is based on a concatenate-permute-truncate construction and consists of four main operations:

- Message Expansion: The input message is expanded using linear codes, which can be implemented as a set of lookup tables. For Hamsi224/Hamsi256 the 32 bit input message is expanded to 256 bits, for Hamsi384 and Hamsi512 the 64 bit input message is expanded to 512 bits.

- Concatenation: The expanded message is concatenated with an initial value or the output from the previous stage of the hash function. This forms the full state of Hamsi which is either 512 or 1024 bits in length depending on the variant used.

- Non linear permutation P: The non linear permutation P is made up of three sub operations. The first operation XORs the state with a table of predefined constants and also a counter. The second operation is the application of one of Serpent's S-boxes [3]. Each S-box operates on 4 bits of the state; therefore, 128 S-boxes for Hamsi224/Hamsi256 (or 256 for Hamsi384/Hamsi512) can be implemented in parallel. The final operation in P is a diffusion operation L. This operation consists of several bitwise shifts and XORs. For Hamsi224/Hamsi256 P is executed three times during normal hashing of a message and six times when it is the last message block. There are also a change in the round constants used when hashing the last block of a message. For Hamsi384/Hamsi512 P is executed six times during normal hashing and twelve times for the final message block.

- Truncation: The truncation stage reduces the Hamsi state down to the size of the input message. In the case of Hamsi224 and Hamsi384 the state must be further truncated to achieve message digest lengths of 224 and 384 bits, this is only done after the last message block has been processed.

A fully parallel design was used for implementation, as shown in Figure 5.4; where $m = 32$, $n = 256$ and, $s = 512$ for Hamsi224/Hamsi256; and $m = 64$, $n = 512$ and, $s = 1024$ for Hamsi384/Hamsi512. The non-linear permutation P was unrolled three times. Further unrolling resulted in a congested design that could not be routed onto the FPGA. Therefore, it takes one clock cycle for a normal message block to be hashed and two clock cycles for the final message block. The Serpent S-boxes were implemented using distributed ROM

5.9 CubeHash

The CubeHash algorithm, designed by Bernstein [9], is defined by three parameters h, r, and b where $h \in \{8, 16, 24, ..., 512\}$, $r \in \{1, 2, 3, ...\}$, and $b \in \{1, 2, 3, ..., 128\}$. A specific version of CubeHash is then defined as CubeHash$r/b-h$. $10r$ rounds are performed to initialise the state, the first b byte block of data to be hashed is then XORed into the first b bytes of the state and r rounds of the compression function are performed. The CubeHash version used in this implementation, is CubeHash$16/32-512$. CubeHash was eliminated in round 2 of the SHA-3 competition.

A diagram of the CubeHash compression function is shown in Figure 5.5. The compression function takes two 512 bit inputs A and B which are each half of the 1024 bit state. The outputs A' and B' are fed back to the inputs if several rounds of

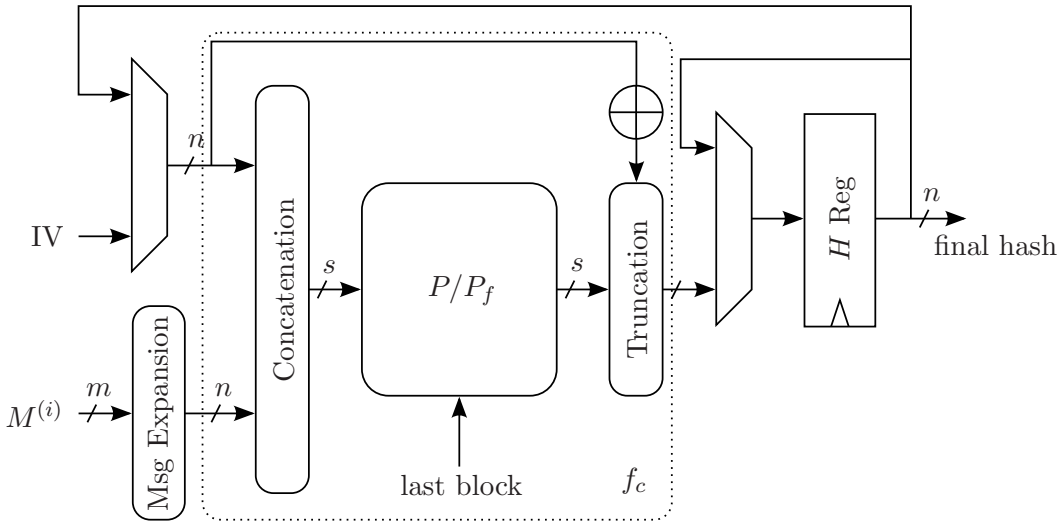


Figure 5.4: Hamsi hash function.

compression are to be performed; this follows the same structure as shown in Figure 5.1. The compression function performs several operations; 2×16 additions modulo 2^{32} ($\boxplus$); 2×16 Boolean XORs ($\oplus$); 2×16 rotation operations, where each word in the B datapath is rotated cyclically by a fixed number of bits; 4×8 swapping operations where certain words in the datapath exchange positions. The finalisation round of CubeHash is performed by iterating the compression function 160 times.

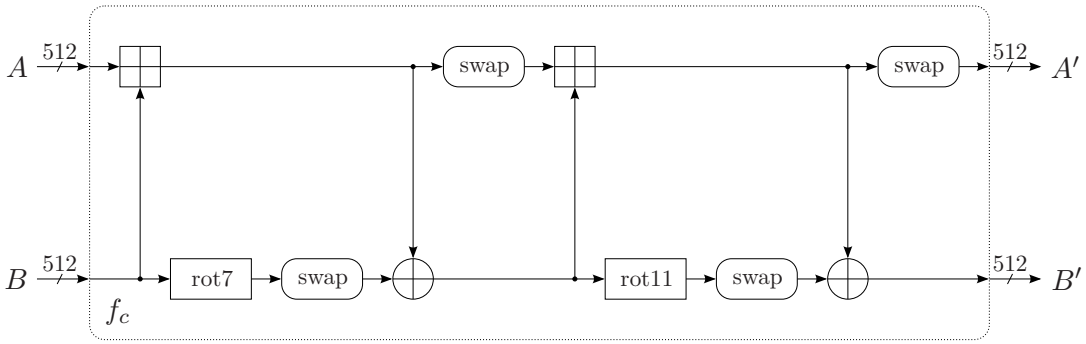


Figure 5.5: CubeHash compression function.

The algorithm was implemented such that one iteration of the compression function takes 1 clock cycle. For CubeHash16/32−512, 16 iterations of the compression function are required to process each message block. An unrolled design was tested; however, no improvement in performance was achieved. Upon reaching a loop unrolled design of length 4 (i.e., 4 concatenated compression functions), the design was no longer routable

on the FPGA as the design was too congested.

5.10 Fair Comparison Methodology

When processing data with a hash function, the data must be padded so that it can be fragmented into blocks of equal size. The size of each of the fragmented blocks is determined by the input message size required by the hash function. There are many different methods used for the padding of input data. The padding schemes used by the hash functions described in the previous sections are given in Table 5.1. For comparison purposes, details of the Keccak hash function [10] are also given, as Keccak was selected as the winning design of the SHA-3 competition.

Design	Padding Scheme
SHA224/256	1, 0's until congruent (448 mod 512), 64-bit message length
SHA384/512	1, 0's until congruent (896 mod 1024), 128-bit message length
BMW224/256	1, 0's until congruent (448 mod 512), 64-bit message length
BMW384/512	1, 0's until congruent (960 mod 1024), 64-bit message length
Cubehash	1, 0's until a multiple of 256 ($256 = 8 * b, b = 32$)
Hamsi224/256	1, 0's until a multiple of 32, 64-bit message length
Hamsi384/512	1, 0's until a multiple of 64, 64-bit message length
Keccak-224	1, 0's until a multiple of 8, append 8-bit rep. of 28, append 8-bit rep. of 1152/8, 1, 0's until a multiple of 1152
Keccak-256	1, 0's until a multiple of 8, append 8-bit rep. of 32, append 8-bit rep. of 1088/8, 1, 0's until a multiple of 1088
Keccak-384	1, 0's until a multiple of 8, append 8-bit rep. of 48, append 8-bit rep. of 832/8, 1, 0's until a multiple of 832
Keccak-512	1, 0's until a multiple of 8, append 8-bit rep. of 64, append 8-bit rep. of 576/8, 1, 0's until a multiple of 576

Table 5.1: Padding schemes.

Due to the different padding schemes and input block sizes used by the submissions to the SHA-3 competition, it was decided that a fair comparison of the designs, in hardware, should include the padding operation and a standard width I/O data bus. The rationale for this decision is based on the fact that the different input block sizes for the hash functions have different impacts on performance. Some of the hash functions require input block sizes up to 1024 bits in length; a data bus of this size is not common practice in most systems. Therefore, comparing only the compression functions does not give a true indication of the relative performance of the designs, as it does not take into account the extra latency introduced by padding and I/O delays. In order to achieve a fair comparison, a generic wrapper for the hash functions was designed, that incorporates a padding unit and a fixed size data bus.

5.10.1 Wrapper Overview

Figure 5.6 shows a block diagram of the wrapper architecture, and its interface with both the hash function and the outside world. The design consists of an input shift reg-

ister with associated padding circuitry that appends the padding data to the incoming message. The input data can be set to any size w, but for a representation of a real world communications system it is set to 32 bits, a standard word size. The input shift register reads and stores the incoming message w bits at a time. The value m is the input message size of the hash function under test. Once the input shift register has stored m bits of message data, it can be passed to the hash function. A shift register is also present at the output which stores the output message digest of size d, while the output bus reads it out w bits at a time. The control circuitry synchronises the shift register operation, padding, and all communication signals.

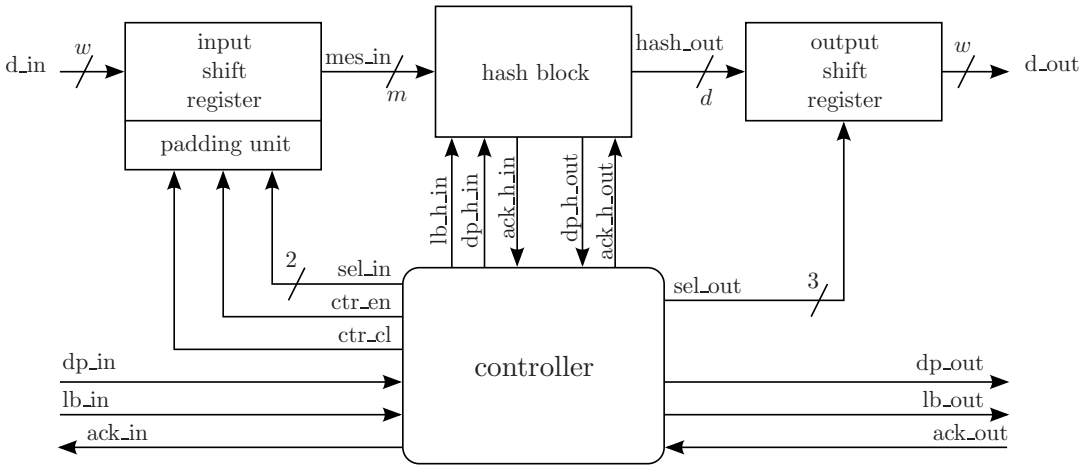


Figure 5.6: Wrapper interface.

5.10.2 Communications Protocol

In order to communicate with the different hash designs, a generic communications protocol between the wrapper and the external world, and between the wrapper and the hash function, had to be specified. Table 5.2 defines these communication signals, which correspond to the signals shown in Figure 5.6. The approach taken is similar to that suggested by Gaj [37]. It differs, however, in the fact that a user wishing to hash a message does not need to do any pre-processing to their plaintext before sending the message, such as adding message length data, but only needs to set an end of message signal high, in this case defined as a last block *(lb_in)* signal either simultaneously with the last block of the message or at any time after transmission of the last message block.

The *dp_in* signal is used to indicate that there is valid data present on the input

bus *d_in*. Once the current w bit message block has been read in, the *ack_in* signal is set high to indicate that the next message block can be placed on the input bus. A similar structure is present on the output side of the wrapper, where the *d_out* data bus holds the next unread w bit block of the message digest. Valid data is present on *d_out* while the *dp_out* signal is set high. The output data is updated when a return acknowledge *(ack_out)* is received.

Signal	**IO**	**Description**
clk	in	Global clock
rst	in	Global reset, **Active HIGH**. Initialises the circuitry to begin hashing a new message
d_in	in	The input bus
dp_in	in	Data present on the **input bus**
ack_in	out	Data present on the **input bus** has been read
lb_in	in	Data present on the **input bus** is the last block of the message to be hashed
d_out	out	The output bus
dp_out	out	Data present on the **output bus**
ack_out	in	Data present on the **output bus** has been read
lb_out	out	Data present on the **output bus** is the last block of the hashed message

Table 5.2: Wrapper interface communication signals.

Table 5.3 defines the communication signals between the hash function and the wrapper. The function of each of these signals is similar to that of the ones presented in Table 5.2. A *lb_out* is not required in this case as the entire message digest is transferred from the hash function to the output shift register in a single clock cycle.

5.10.3 Padding Protocol

Many similarities are present between the different padding schemes of the hash functions, see Table 5.1. Therefore; a generic circuit structure can be developed to implement the different variants of Merkle-Damgård strengthening [78] padding schemes, as well as padding types of all-zeros or one-and-trailing-zeros.

Figure 5.7 shows the block diagram of the selection process for some of the different padding schemes. Data is fed in on the *d_in* bus, w bits at a time and saved in a register. The data in the register is shifted w bits every subsequent input until the register is full. When a total of m bits have been read in, the data can be transferred to the hash function. If the message ends prior to filling of the register, the relevant

Signal	IO	Description
clk	in	Global clock
rst	in	Global reset **Active HIGH**
mes_in	in	Data-in bus
dp_h_in	in	Valid data on **Data-in bus** Set when **buffer-in** shift-register is full
ack_h_in	out	Data present on **Data-in bus** has been read
lb_h_in	in	Last block is present on **Data-in bus** Inclusive of padding where required
hash_out	out	Data-out bus
ack_h_out	in	Data present on **Data-out bus** has been read.
dp_h_out	out	Message digest is present on **Data-out bus**

Table 5.3: Hash interface communication signals.

padding scheme is selected via multiplexer.

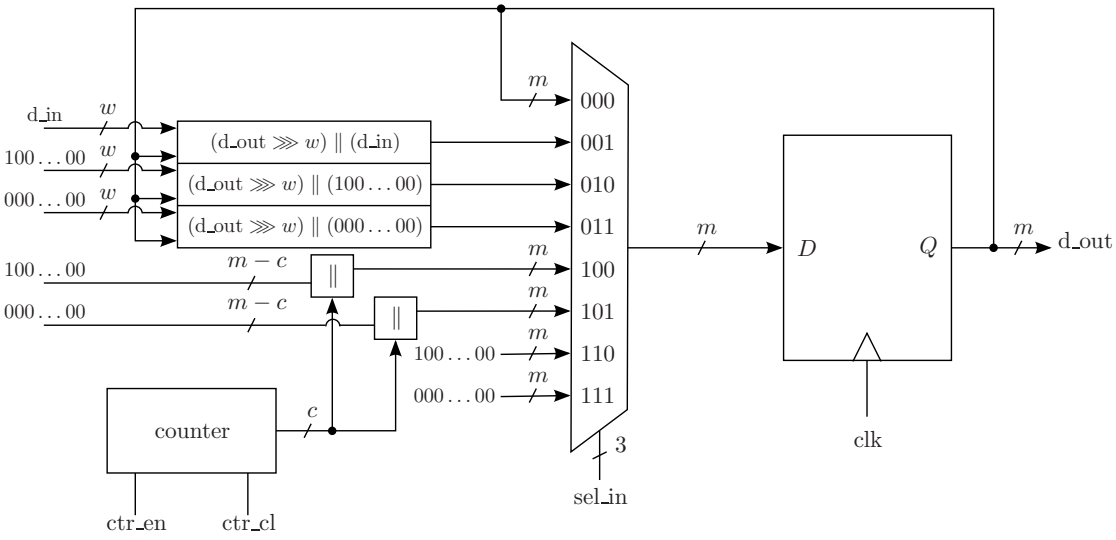


Figure 5.7: Padding block.

5.11 Implementation Results

Table 5.4 gives the clock cycle count for the 3 hash function designs and their variants. As can be seen from the table, the time required to load the input message block is different for each design. This is due to the difference in input block size and also the different padding schemes. Finalisation rounds are also defined differently for each hash function; CubeHash in particular has a very thorough finalisation process. These

differences, however, have less of an impact on the overall calculation time as the message size increases. Two different results were therefore used to compare the designs. A short message is defined as the time required to process the padding, a message block, and perform the finalisation. A long message is defined as just the time to process the message block. The use of these two metrics can be justified by examining the different uses of hash functions in the TLS protocol, such as those discussed in Section 5.4. Many of these functions require a hash operation to be applied to messages of arbitrary length, in this case the long message results are most applicable. However, by examining the construction of the HMAC function, which is used throughout the TLS key derivation process, it can be seen that the outer hashing operation from Equation 5.1 is always of fixed length. In the case of SHA256, this corresponds to applying the hash operation to 2 message blocks; therefore, a hash function that performs poorly for short messages will have a larger impact on this specific operation, than hash functions that perform well.

It should be noted that in the case of BMW, the time required to hash a message is actually shorter than the time required to load a message. This results in the hash function remaining idle for a period of time as the message block is loaded. Calculating the throughput for this case must therefore take into consideration this delay.

Hash Design	32-bit load #Cycles	Extra Padding	Padding #Cycles	Message Rounds	Round #Cycles	Long Msg #Cycles	Final Rounds	Final #Cycles	Short Msg #Cycles
SHA224/256	16	0	0	64	1	65	0	0	65
SHA384/512	32	0	0	80	1	81	0	0	81
BMW224/256	16	0	0	1	4	4	1	3	7
BMW384/512	32	0	0	1	4	4	1	3	7
Cubehash	8	0	0	16	17	17	160	161	178
Hamsi224/256	1	3	1	3	2	6	6	24	31
Hamsi384/512	2	3	1	6	2	12	12	48	61
Keccak-224	36	0	0	24	1	25	0	0	25
Keccak-256	34	0	0	24	1	25	0	0	25
Keccak-384	26	0	0	24	1	25	0	0	25
Keccak-512	18	0	0	24	1	25	0	0	25

Table 5.4: Hash function clock cycle count.

The performance of the different designs are compared in terms of area consumption and throughput. The throughput is given by

$$\text{Throughput} = \frac{\#\text{ Bits in a message block} \times \text{Maximum clock frequency}}{\#\text{ Clock cycles per message block}} \tag{5.15}$$

The throughput per area (TP/Area) is another metric used in the design comparisons. It indicates how efficiently a design makes use of the logic resources that it

consumes. A higher TP/Area is desirable as it shows that the design is processing more bits per slice. The performance of each of the designs is shown in Table 5.5. All designs are implemented such that they only occupy slice and routing logic on the FPGA. Memory components are implemented as distributed RAM and the use of vendor specific FPGA resources such as DSP blocks were avoided. All results shown in Table 5.5 are post place and route for a Xilinx XC5VLX330T-2-ff1173 FPGA. The *-w* designation, defines the results inclusive of the wrapper, while *-nw* gives the hash function as a stand alone entity. The *TP-s* and *TP-l* results are for long and short message clock cycle counts, as shown in Table 5.4, and are inclusive of the latency introduced by the wrapper.

Hash Design	Area-w (slices)	Max. Freq-w (MHz)	Area-nw (slices)	Max. Freq-nw (MHz)	TP-l (Mbit/s)	TP-l/Area ((Mbit/s)/slice)	TP-s (Mbit/s)	TP-s/Area ((Mbit/s)/slice)
SHA256	1,019	125.063	656	125.125	985	0.966	985	0.966
SHA512	1,771	100.04	1,213	110.096	1264	0.713	1264	0.713
BMW-256	5,584	14.306	4,997	14.016	457	0.081	457	0.081
BMW-512	9,902	8.985	9,810	10.004	287	0.028	287	0.028
Cubehash	1,025	166.667	695	166.833	2509	2.447	239	0.233
Hamsi-256	1,664	67.195	1,518	72.411	358	0.215	69	0.041
Hamsi-512	7,364	14.931	6,229	16.51	79	0.01	15	0.002
Keccak-224	1,971	195.733	1,117	189	5915	3	5915	3
Keccak-256	1,971	195.733	1,117	189	6263	3.17	6263	3.17
Keccak-384	1,971	195.733	1,117	189	8190	4.15	8190	4.15
Keccak-512	1,971	195.733	1,117	189	8518	4.32	8518	4.32

Table 5.5: Hash function implementation results.

A clear difference can be seen between the performance of the CubeHash design for short messages versus long messages, with a significant increase in throughput and throughput per area for the long message results. This is due to the number of finalisation rounds required by the CubeHash specification. The 160 finalisation rounds of the CubeHash design account for the majority of the processing time for short messages. In contrast, the SHA256 design does not require a modified finalisation process; therefore, no change is present in the performance for short versus long messages. The Keccak design has a significantly higher throughput per area than the other designs, which was a contributing factor in Keccak winning the SHA-3 competition.

5.12 Discussion

In this chapter, a thorough analysis of hash function designs has been presented. First, the application of hash functions in the TLS protocol was discussed. A generic hash function wrapper was then introduced that allows for the fair comparison of different hash function designs by including the latency of data I/O and the application of

padding in the results. Several hash function designs were then analysed for area and performance using the wrapper.

In the previous chapters, various cryptographic functions have been discussed. So far all have been deterministic functions that can be analysed mathematically to ensure that they provide a specific level of security. All of these functions, however, also rely on secret keys and nonces that must be generated at random in some secure manner. The next chapter deals with this problem by discussing the design of TRNGs.

Chapter 6

True Random Number Generators

6.1 Introduction

A *Random Number Generator* (RNG) is required in most cryptographic systems to provide an unpredictable source of data. This data source is generally used to generate initialisation vectors or cryptographic keys. It is imperative that the output of the RNG is unpredictable, as all subsequent operations are usually related, in some way, to the initial random data generated by the RNG. A high quality RNG will output an unbiased stream of bits that possess good statistical properties.

RNGs fall into two main categories: *Pseudorandom Number Generators* (PRNGs) and *True Random Number Generators* (TRNGs). A PRNG is a mathematically defined algorithm that outputs a stream of bits that have excellent statistical properties. As a PRNG is deterministic, it does not generate any entropy by itself and requires a random seed as an input. The output bitstream will then only contain as much entropy as was present in the seed. It is possible that the entropy per bit can be increased, if the PRNG compresses the seed in some way, however, this would reduce the throughput of the design.

In contrast, a TRNG is non-deterministic and extracts entropy from some unpredictable physical source; with the goal being to extract as much entropy as possible. The TRNG must, in some way, convert this physical source of randomness into a stream of 1's and 0's. A well designed TRNG will produce a stream of bits that have good statistical properties, similar to that of a PRNG, however, TRNGs generally have a much lower bit rate than PRNGs. Therefore, if a high bitrate is required, a PRNG is

used to generate the random bitstream and the TRNG is used to refresh the seed at regular intervals.

In this chapter, several different TRNGs will be compared in order to analyse their performance on an FPGA platform. Post-processing techniques and testing of the statistical properties of TRNGs will also be discussed. These specific designs were chosen as they are, at the time of writing, the most recent designs published and have displayed the ability to produce a raw stream of bits with good statistical properties, in previous implementations. A detailed list of TRNG designs, can also be found in Chapter 4 of [21].

6.2 Implementation of TRNGs

TRNGs are one of the most critical components of a cryptographic system to implement, as they are used to produce secret keys and initialisation vectors for other cryptographic algorithms. A TRNG should be able to provide a high entropy source of random data, at an acceptable bitrate. For this to be possible, the circuit must be able to sample some unpredictable physical source, at often enough intervals, without there being any correlation between the resulting bits. Sources of entropy on an FPGA are usually in the form of phase or frequency jitter in ring oscillators, or some type of metastable circuit. The underlying physical mechanisms that provide these sources of randomness are generally attributed to the existence of thermal and shot noise in semiconductor devices [21, Section 4.2]. Semiconductor devices are designed to minimise these sources of noise, which therefore makes the implementation of TRNGs using standard logic components a difficult task.

To provide security against an attacker, a TRNG should be implemented within the FPGA, as opposed to being an external component. If the TRNG, which is effectively a noise source, is implemented off chip, an attacker may be able to inject some bias into the noise source; thus, causing the TRNG to fail as a random source. An attacker would also be able to intercept all data that is transmitted between the external RNG and the FPGA, completely compromising the secrecy of the data. When implementing a TRNG inside an FPGA, only digital components are available; this poses a more difficult design challenge when compared to ASICs. Consequently, a large amount of research has been conducted in this area [66, 112, 116, 119, 120].

Although it's preferable that the TRNG produces a stream of bits with good statistical properties, this is not usually the case. FPGA vendors try to minimise the amount of unpredictable behaviour in their ICs; therefore, designing an unpredictable

noise source can be difficult. For this reason, the bitstream generated by most TRNGs contains a certain amount of bias. In order to correct this, some form of post-processing can be applied to the bitstream in order to improve its statistical properties. A common post-processing technique is to use a cryptographic hash function [36, pages 161–182] or PRNG to process the output of the TRNG. A hash function provides a very robust method of post-processing as it has strong cryptographic properties that prevent an adversary from predicting output sequences, see Section 5.3 and [6]. It is also advantageous to monitor the statistical properties of the raw stream of bits that the TRNG produces (i.e., before any post-processing is applied), as TRNGs can at times fail to produce sufficiently random data and it is necessary to be able to detect these events in order to maintain the security of the overall system.

6.2.1 Analysing the Quality of TRNG Output Data

When designing a TRNG, the quality of the output bitstream must be analysed to ensure that the TRNG is working correctly and that the data it produces is sufficiently random. In order to test the quality of a bitstream, several offline test suites have been developed [62, 86, 108, 113]. These test suites can be used, during the design phase, to process a file of binary data taken from the TRNG and will return a pass/fail result as to whether the data has good statistical properties and appears random. This process of testing the statistical properties of the TRNG allows for any weaknesses in the underlying architecture to be found, such as any bias in the output bitstream.

One of the most thorough test suites is the diehard battery of statistical tests [74]. It cannot guarantee that a TRNG is producing truly random random data, however, it does indicate what quality of output the circuit is producing. There are 16 diehard tests in total:

1. Birthday spacings test: Based on the birthday paradox, this test uses the input data as birthdays in a year of 2^{24} days. The spacings between birthdays form a list and the occurrence of each value in the list should follow a certain distribution.

2. Overlapping permutations test: The ordering of each set of 5 consecutive 32 bit numbers from input data are tested. Each set can be in one of 120 orders, which are then counted and should follow a known distribution.

3. Binary rank test for 31×31 matrices: The leftmost bits of random integers in the test data are used to generate 31×31 matrices over the field $\mathbb{F}_2$. The rank of

each of the generated matrices is calculated and a chi-squared test is applied to the set of results.

4. Binary rank test for 32×32 matrices: Same as test 3, except 32×32 matrices are used.

5. Binary rank test for 6×8 matrices: Same as test 3, except 6×8 matrices are used.

6. Bitstream test on 20 bit words: The input file is divided into overlapping 20 bit words ($b_0, b_1, b_2, \ldots b_{20}$, $b_1, b_2, b_3, \ldots b_{21}$ etc..). Each word can therefore be one of 2^{20} possible values. The number of missing possibilities are counted and should follow a certain distribution.

7. In a similar method to test 6, the OPSO, OQSO, and DNA tests convert the input data into words and count the number of missing words in the generated sequence. Each test uses a different word length or alphabet.

   - *Overlapping-Pairs-Sparse-Occupancy* (OPSO)
   - *Overlapping-Quadruples-Sparse-Occupancy* (OQSO)
   - DNA test

8. Count the 1's in a stream of bytes: The input test data is broken into a stream of 8 bit bytes. The number of 1's in each byte are counted and each count converted to a letter; resulting in a sequence of letters based on the input data. This sequence is broken up into overlapping five letter words. The frequency of each possible five letter word is counted and should follow a known distribution.

9. Count the 1's in specific bytes: Follows the same principle as test 8, however, only one randomly chosen byte in every four of the input bytes is used to generate a letter.

10. Parking lot test: Attempts are made to place circles of radius 1 in a square of side 100 such that they do not overlap. The test data provides the coordinates of each attempt. The number of attempts versus the number of successfully placed circles should follow a known distribution.

11. Minimum distance test: The test data is used to generate the coordinates of 8000 points in a square of side 10000. The square of the minimum distance between points should follow a known distribution.

12. 3D spheres test: Each 32 bit value from the test data is converted to floating point values on the range $[0, 1000)$. Every three values form a triplet (x_n, y_n, z_n) which are taken as points in a cube of edge 1000. The radius of each sphere is taken to be the minimum distance such that it does not overlap with any other sphere in the cube. The resulting radii should follow a known distribution.

13. Squeeze test: Each 32 bit value from the test data is converted to floating point values on the range $[0, 1)$. These floating point values are multiplied by $k = 2^{31}$ and the number of multiplications required to reduce k to 1 are counted. A chi-squared test is then applied to the result.

14. Overlapping sums test: Each 32 bit value from the test data is converted to floating point values on the range $[0, 1)$ and denoted $X_0, X_1, X_2, \ldots X_n$. The overlapping sums $S_0 = X_0 + X_1 + \ldots X_{100}$, $S_1 = X_1 + X_2 + \ldots X_{101}$, ..., $S_n = X_{n-100} + X_{n-99} + \ldots X_n$ are calculated and a chi-squared test applied to the result.

15. Runs test: The sequence of test data is converted to floating point values on the range $[0, 1)$. The ascending and descending runs in the sequence are counted and should follow a known distribution.

16. Craps test: 200,000 games of craps are played, where each 32 bit value of the test data is used as the result of the throw of a die. The number of wins and throws necessary to end each game are counted. A chi-squared test is applied to the result.

For a full description of how these tests are implemented, see [74].

6.3 Vasyltsov et al.

In [122], the authors introduced a new form of ring oscillator that can be forced into a metastable state. The authors refer to this circuit, shown in Figure 6.1, as a metastable ring oscillator. The circuit can be constructed on an FPGA as it consists of digital components only. The design consists of an odd number of inverters, connected in series, through a set of multiplexers. The multiplexers are used to change the configuration of the circuit and are controlled by the clock control signal. The multiplexers can configure the circuit so that the inverters are isolated from each other, or are connected in series; effectively making the circuit a ring oscillator.

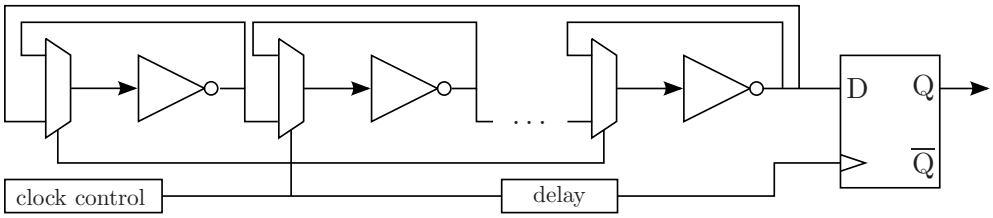


Figure 6.1: Vasyltsov et al. design.

The design works by disconnecting all the inverters from each other; thus, turning them into a series of individual noise sources. At this point the inverters are in a metastable state. The entropy in the circuit is obtained from the thermal noise that affects the initial oscillations of the inverters, when they are reconnected in series. Figure 6.2 shows how the circuit behaves just as the inverters are connected back in series. Before the clock control pulse, the circuit is in a metastable state. When the clock control signal transitions to a positive voltage level, the multiplexers connect all of the inverters back in series. At this point the circuit is effectively a ring oscillator, however, due to the random starting conditions of the circuit, the initial waveform is chaotic. This initial chaotic behaviour is what the authors say contains the entropy. Figure 6.3 shows the operation of the circuit as it transitions to and from the metastable state, over three sampling periods. The random bit is sampled a few hundred nanoseconds after the circuit is brought out of the metastable state. Sampling cannot take place before this, as the output voltage may not achieve a suitable level to be sampled by a flip-flop. This can be seen in Figure 6.2 where the TRNG output voltage takes approximately 200 ns, after the control signal transitions from logic 0 to logic 1, to reach a stable level.

The design was implemented on a Xilinx Virtex 5 XCVLX110T FPGA and occupied 10 slice LUTs. Table 6.1 shows the results of the diehard tests when applied to random data taken from the circuit. Four different sampling frequencies are given in order to analyse the effect of holding the circuit in a reset state for longer periods of time. A longer time between subsequent samples should result in less correlation between samples. The authors state that they achieved a throughput of 2.5 *Megabits per second* (Mbit/s) in their FPGA implementation. However, it can be seen from the results that the circuit did not operate effectively at 2.5 Mbit/s and reducing the sampling rate below 1 Mbit/s was required to improved the randomness properties of the output bitstream.

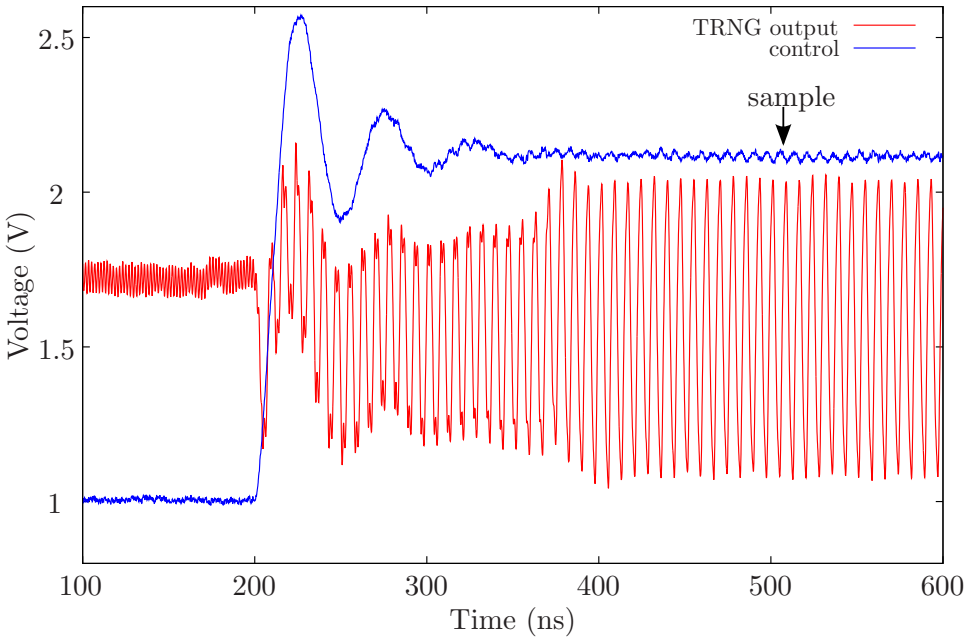


Figure 6.2: Output of Vasyltsov et al. design as clock control pulse is applied.[1]

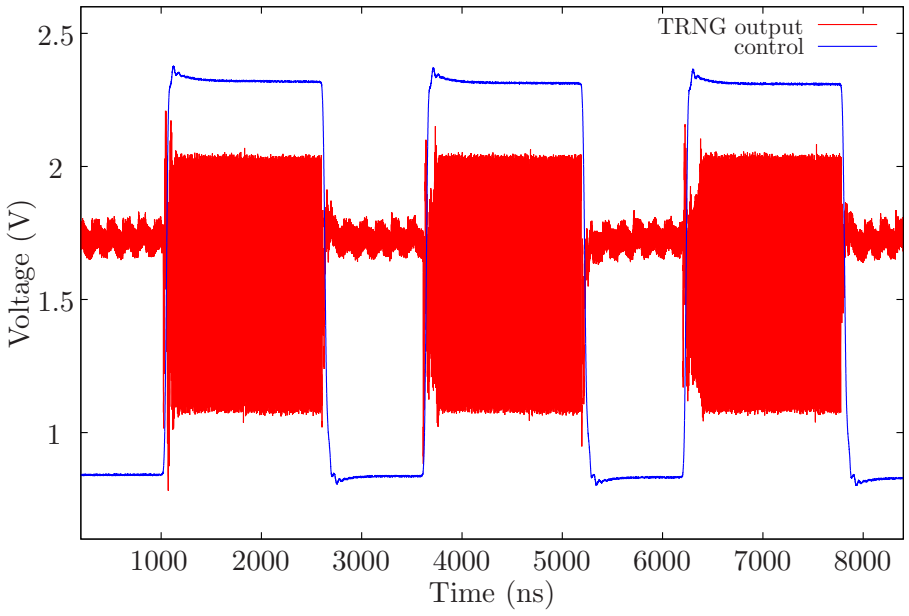


Figure 6.3: Operation of Vasyltsov et al. design over three sampling periods.[2]

[1]The voltage of the control signal has been scaled for clarity.
[2]The voltage of the control signal has been scaled for clarity.

Speed (Mbit/s)	Diehard Test No.															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0.4	**P**	**P**	**P**	**P**	**P**	F	F	F	F	**P**	**P**	**P**	F	**P**	**P**	**P**
0.625	**P**	**P**	**P**	**P**	**P**	F	F	F	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**
1	**P**	**P**	**P**	**P**	F	F	F	F	F	F	**P**	**P**	F	**P**	**P**	F
2.5	F	F	**P**	**P**	F	F	F	F	F	F	F	F	F	F	**P**	F

Table 6.1: Diehard test results for Vasyltsov et al. design, implemented at several sampling speeds.

6.4 Varchola and Drutarovský

A very low area TRNG design was presented by Varchola and Drutarovský in [121]. The circuit, referred to as a *Transition Effect Ring Oscillator* (TERO), can be implemented in a single CLB of a Xilinx Spartan 3E FPGA, as it requires only 9 logic functions. The design, shown in Figure 6.4, was implemented on a Xilinx Virtex 5 FPGA. Although the CLB structure of the Virtex 5 differs from that of the Spartan 3E, the circuit can still be implemented in a single CLB. In order to implement the XOR gates in the circuit, the authors suggest that the dedicated XOR gates, present in the carry chain of a CLB, should be used. The authors state that this can result in a more stable operation of the circuit. To do this the design must be written using Xilinx library primitives, and the routing specified by the designer.

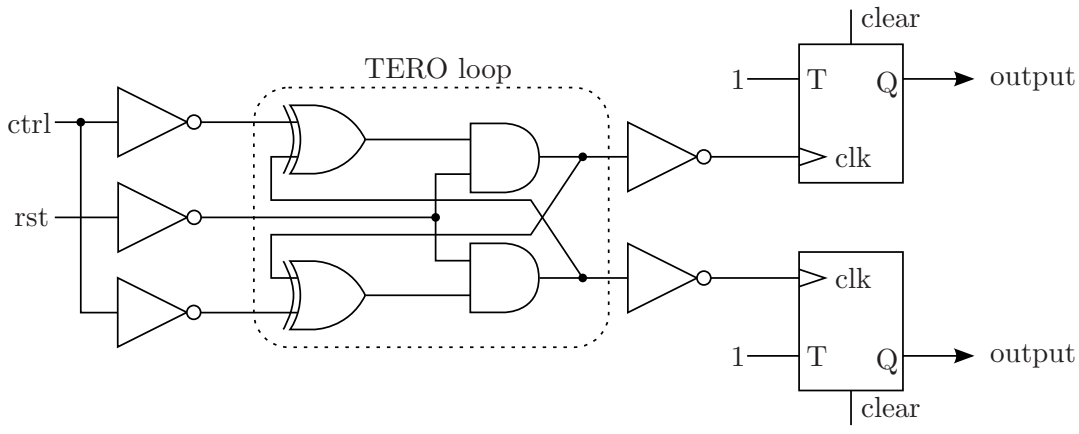


Figure 6.4: Varchola and Drutarovský design.

The design works on the principle that the TERO loop portion of the circuit, shown in Figure 6.4, takes an unpredictable number of oscillations to return to a steady state

condition after it is disturbed by toggling the inputs of the two XOR gates. This operation can be seen in Figure 6.5, where, at each edge of the *ctrl* signal, the TERO output waveform is excited into a metastable state. Two control pulses are shown in Figure 6.5, and it can clearly be seen that after the rising edge of each control pulse, the TERO oscillates for a different length of time (Δ_1 and Δ_2). The number of oscillations is then measured using a T-type flip-flop; which is effectively a 1 bit counter. The period of the control signal is 4000 ns in length, and 1 random bit is sampled 200 ns before the falling edge of the control signal, shown as s_1 and s_2 in Figure 6.5. Therefore, a throughput of 250 *Kilobits per second* (kbit/s) can be achieved. The reset signal is applied to the circuit as the random bit is being sampled. The AND gates in the circuit allow the TERO loop to be returned to the same initial conditions before each control signal edge. This is done by setting the reset signal high.

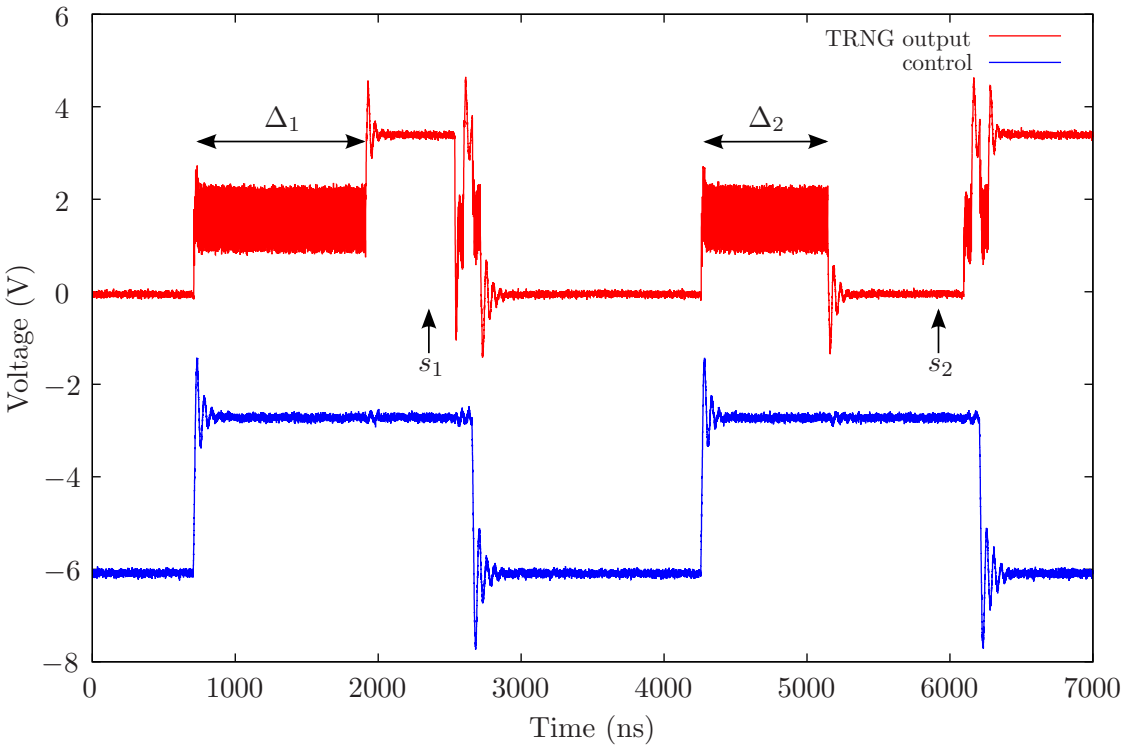


Figure 6.5: TERO output waveform.[1]

Table 6.2 gives the results of the diehard tests when applied to random data produced by the circuit. Three sampling frequencies are given, where, a reduction in sampling frequency is a result of holding the circuit in a reset state for a longer period

[1]The control signal has been shifted by -6 volts.

of time between each sample. As the output oscillations resolve to a stable level after several hundred nanoseconds, increasing the time between the circuit being activated and taking the sample would not affect the randomness of the data produced. However, holding the circuit in a reset state for a longer time between samples may reduce the chance of subsequent samples being correlated. However, from the obtained results it does not appear that increasing the time that the circuit is held in reset has any impact on the randomness properties of the output data as each set of data performs similarly in the diehard tests.

Speed (kbit/s)	Diehard Test No.															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
185	**P**	**P**	**P**	**P**	F	F	F	F	F	**P**	F	**P**	F	**P**	**P**	**P**
240	**P**	**P**	**P**	**P**	**P**	F	F	F	F	F	**P**	**P**	**P**	**P**	**P**	**P**
250	**P**	**P**	**P**	**P**	F	F	F	F	F	F	**P**	**P**	F	**P**	**P**	F

Table 6.2: Diehard test results for Varchola and Drutarovský design, implemented at several sampling speeds.

6.5 Dichtl and Golić

The FIGARO random number generator is based on Galois and Fibonacci ring oscillators and was introduced by Golić in [46]. A thorough analysis was carried out by Dichtl and Golić in [28]. A ring oscillator is an odd number of inverters connected together in series where the output of the final inverter is connected to the input of the first; thus, creating a combinatorial loop. The circuit will then oscillate with a period defined by the sum of the delays of the inverters and wiring delays. The output of the ring oscillator will contain some phase jitter, which has been used in some TRNG designs [116], however, a large number of ring oscillators are required to provide enough entropy. Very tight timing constraints are also required in order to ensure that only the phase jitter is sampled. Galois and Fibonacci ring oscillators are a combination of *Linear Feedback Shift Registers* (LFSRs) and ring oscillators. Some feedback paths and XOR logic are introduced into the circuit as shown in Figures 6.6 and 6.7. As a result, the circuit no longer outputs a waveform of defined period.

The circuits are believed to combine the true randomness properties of ring oscillators with the pseudorandomness properties of LFSRs. A useful way to represent the feedback paths of a *Fibonacci Ring Oscillator* (FIRO) or a *Galois Ring Oscillator* (GARO) is with a feedback polynomial, such as $f(x) = \sum_{i=0}^{r} f_i x^i$, $f_0 = f_r = 1$,

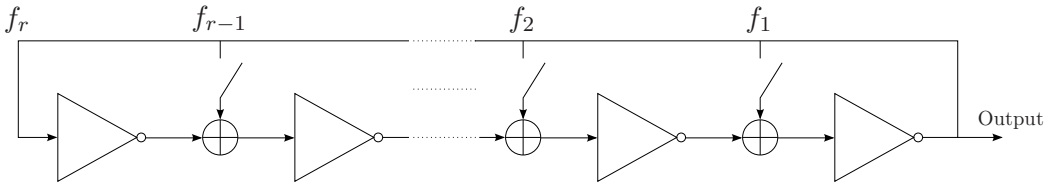


Figure 6.6: Galois ring oscillator.

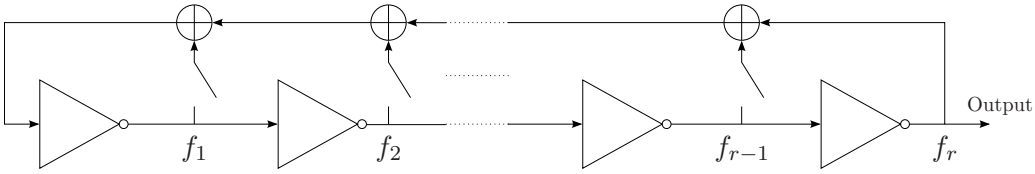


Figure 6.7: Fibonacci ring oscillator.

where each $f_i = 1$ define that a feedback path is present at that point. In order to ensure that the outputs of the circuits cannot get stuck in a fixed state, the feedback polynomials should satisfy the following properties. The feedback polynomial for both circuits should have the form $f(x) = (1 + x)g(x)$, where $g(x)$ is an irreducible polynomial [70, page 82]. An additional property for the correct implementation of the the FIRO is that $g(1) = 1$ and for a GARO the number of inverters should be odd (i.e., r should be odd). The output of a FIRO or a GARO should be a high entropy white noise signal. The output can be sampled by a D-type flip flop.

In order to reduce power consumption and also to remove the possibility of any statistical dependence between successively sampled bits, the GARO or FIRO can be operated in restart mode. This is done by replacing one of the inverters with an AND gate. The output of the AND gate can then be forced to zero, in order to shut down the circuit. In restart mode the circuit is run for a short period of time after a restart. After enough time has passed the output of the circuit is sampled, the circuit is then shut down until another random bit is required.

In order to improve the randomness of the TRNG, the outputs of a FIRO and a GARO can be combined with an XOR gate. This type of design is known as a FIGARO. Combining the outputs of the circuits should improve the overall randomness of the bitstream, as the sampling of an additional independent noise source should increase the amount of entropy that is being sampled. This design is shown in Figure 6.8.

The circuit was implemented on a Xilinx Virtex 5 XCVLX110T FPGA and the output analysed with an oscilloscope. Figure 6.9 shows how the output of a GARO circuit behaves just after the circuit is started. The output voltage quickly assumes a

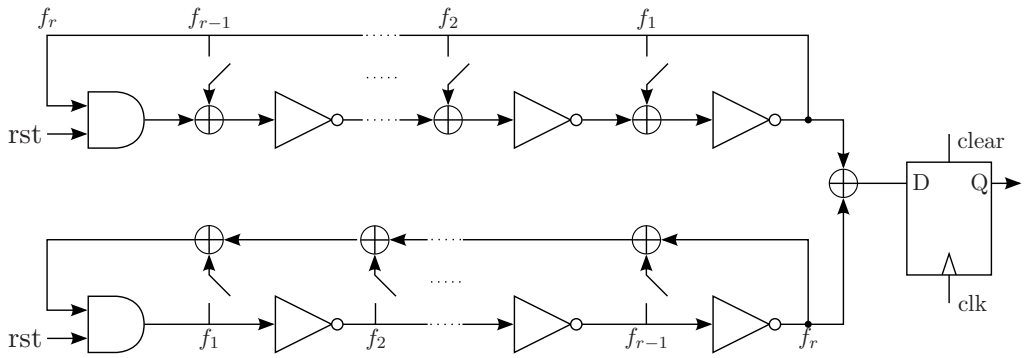


Figure 6.8: FIGARO TRNG.

chaotic behaviour. This waveform was achieved by placing the output of the GARO circuit in a slice as close to the output pin of the FPGA as possible. This minimises the interference that would be present in the signal if it had to pass through a long routing path, inside the FPGA. Internal capacitances would also have distorted the output waveform. The arrow indicates the time at which the output is sampled by the flip-flop. The output of the FIRO circuit operates in the same manner.

In Figure 6.10, the sampling of 8 consecutive bits from a FIGARO circuit can be seen. The circuit is operating in restart mode and a single bit is sampled 160 ns after every falling edge of the *control* signal. This gives the design a throughput of 6.25 Mbit/s, which is the highest of all designs presented in this work. The design used a FIRO of length 29 and a GARO of length 31. The feedback polynomial of the FIRO circuit is given by

$$1+x^1+x^2+x^5+x^6+x^8+x^9+x^{10}+x^{13}+x^{15}+x^{18}+x^{21}+x^{24}+x^{25}+x^{28}+x^{29}, \quad (6.1)$$

while, the GARO feedback polynomial used was

$$1+x^1+x^3+x^4+x^5+x^6+x^8+x^9+x^{10}+x^{13}+x^{15}+x^{16}+x^{17}+x^{20}+x^{21}+x^{23}+x^{27}+x^{31}. \quad (6.2)$$

Table 6.3 shows the results of the diehard tests when applied to a random bitstream of the output data of the FIGARO circuit. Sampling speeds of 6.25, 9, and 12.5 Mbit/s were used to generate three sets of results. As can be seen from Table 6.3, increasing the sampling frequencies beyond 6.25 Mbit/s has a significant impact on the statistical properties of the output bitstream. This is most likely a result of the circuit not having

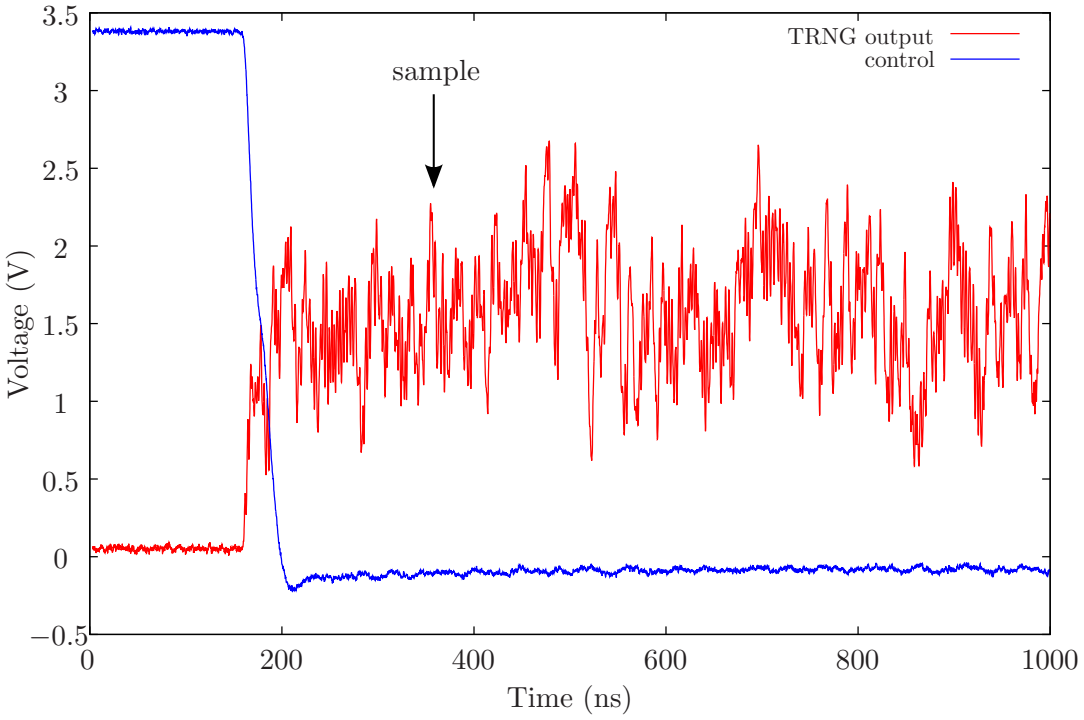


Figure 6.9: Output of GARO circuit just after restart.

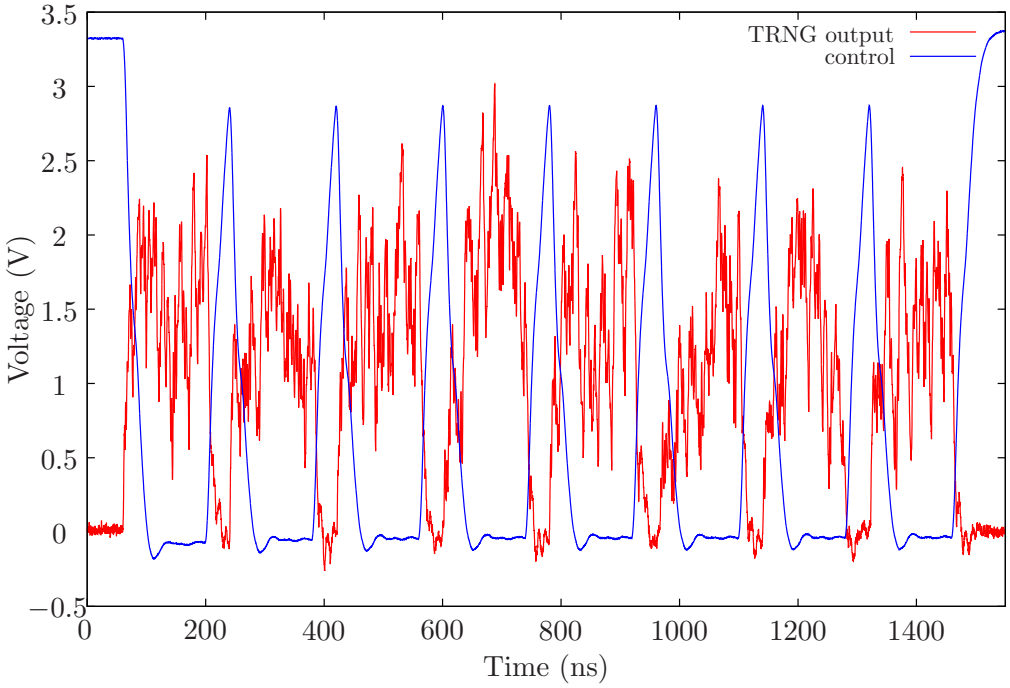


Figure 6.10: Eight sampling periods of the FIGARO circuit.

enough time in the active state to diverge from the reset state output voltage.

Speed (Mbit/s)	Diehard Test No.															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6.25	**P**	**P**	**P**	**P**	**P**	**P**	**P**	F	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**
9	**P**	**P**	**P**	**P**	**P**	F	F	F	F	**P**	**P**	**P**	F	**P**	**P**	F
12.5	**P**	**P**	**P**	**P**	**P**	F	F	F	F	**P**	**P**	**P**	F	**P**	**P**	**P**

Table 6.3: Diehard test results for Dichtl and Golić design, implemented at several sampling speeds.

6.6 Comparing the Results

The designs discussed in the previous sections were all implemented on the same Virtex 5 XC5VLX110T FPGA and their performance and statistical properties analysed at different sampling speeds. Table 6.4 shows the results for each of the TRNG designs at their best performing sampling rate. A 14 MB stream of bits was taken from each design and passed through the diehard battery of statistical tests [74].

Design	Speed (Mbit/s)	Diehard Test No.															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Vasyltsov et al.	0.4	**P**	**P**	**P**	**P**	**P**	F	F	F	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**
Varchola and Drutarovský	0.24	**P**	**P**	**P**	**P**	**P**	F	F	F	F	F	**P**	**P**	**P**	**P**	**P**	**P**
Dichtl and Golić	6.25	**P**	**P**	**P**	**P**	**P**	**P**	**P**	F	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**

Table 6.4: Diehard test results.

The Varchola and Drutarovský design performed worst; failing five of the diehard tests while also having a very low throughput of only 250 kbit/s. For this reason, it was found to be unsuitable for use as the TRNG module that will be present in the final coprocessor design. The design of Vasyltsov et al. passed all but three of the tests, and with post-processing would likely provide a good source of random data. The design of Dichtl and Golić, however, was clearly the best performing design; having the highest throughput of the designs that were tested and only failing the *count the 1's in a stream of bytes* test. This is possibly due to the fact that the design relies much less on having balanced routing paths, than the other two designs. The Dichtl and Golić design is also the only one to incorporate an element of pseudorandomness in its output.

Portability is also as factor in choosing the correct design, as the designs of Vasyltsov et al. and Varchola and Drutarovský required the circuit to be hand routed, while, the design of Dichtl and Golić can be routed by the FPGA tools. For these reasons, the

design of Dichtl and Golić was chosen to be used in the design of the coprocessor that will be presented in Chapter 7.

6.7 TRNG Failure Detection

When designing a random number generator it's possible to analyse the data produced using a large battery of statistical tests; this may not be the case during the standard operation of the TRNG. If the TRNG fails during operation, the entire system may become vulnerable to attack. To prevent this, some statistical tests can be used for on the fly evaluation of the random bitstream that is produced by the TRNG. If a failure in the randomness is detected, an alarm can be set so that no more data from the TRNG is used throughout the system. The reader is referred to [110] and [109] for a more detailed explanation of on-the-fly testing of TRNGs for randomness.

The statistical tests implemented here were presented in [109] and operate on 20000 bits of data at a time. The designers used four of the statistical test from the FIPS140-2 standard [86]. The tests take the output bitstream of the TRNG prior to the post-processing stage, this is done as the post-processing would mask any drop in randomness of the TRNG. The designers chose four tests that could be implemented efficiently on an FPGA; implementing the entire battery of tests would consume too much area. The four tests implemented are:

1. Frequency test (Monobit test): This test verifies that there is a uniform distribution of 1's and 0's. The number of 1's in the 2×10^4 bitstream are counted and should lie in the range $[9726, 10274]$.

2. Poker test: For this test the 2×10^4 bit stream is split into four non-overlapping segments each of length 5×10^3 bits. The number of occurrences of each of the 16 possible 4 bit values are counted and stored. The number of occurrences of each 4 bit value is denoted f_i, where $0 \leq i \leq 15$, and the following evaluated:

$$\chi^2_{15} = \left(\frac{16}{5000}\right) \times \left(\sum_{i=0}^{15} f_i^2\right) - 5000. \tag{6.3}$$

   The result follows a Chi-squared distribution and the test is passed if $2.16 \leq \chi^2_{15} \leq 46.17$.

3. Runs test: The number of runs (consecutive bits of all 1's or all 0's) are counted and stored for run lengths of 1 to 6. For the test to pass the number of runs of

each length should not exceed the intervals shown in Table 6.5.

i	Required Interval
1	2315–2685
2	1114–1386
3	527–723
4	240–384
5	103–209
≥ 6	103–209

Table 6.5: Runs test interval requirements.

4. Long run test: This test verifies that there are no runs of length 26 or more in the bitstream.

6.7.1 FPGA Implementation

The frequency test requires very little logic and can be implemented using a 15 bit wide counter and some logic for comparing the results. As the TRNG generates each random bit, it is tested and a counter incremented depending on its value. Once all 20000 bits have passed through the test, the final result held in the counter is checked to see if it is in the correct range. Post map results from the Xilinx ISE toolchain show that the test occupies 12 slices on a Virtex-5 FPGA.

The poker test is the most computationally intensive of the four tests as it requires both multiplication and addition operations. The multiplication operation can be implemented by using a DSP block in the FPGA; thus, reducing the slice logic requirements of the test. Post-map results indicate that the test occupies 117 slices and one DSP block on a Virtex-5 FPGA.

The runs test can be implemented by using a counter to store the number of runs for each of the run lengths of 1 to 6. As a counter is needed for each run length and also some comparison logic to evaluate a pass or fail result, the test consumes significantly more area than the frequency or long run tests. The post map results show that the test consumes 81 slices on a Virtex-5 FPGA.

The long run test consumes the least amount of area of the four tests. The test only requires a 5 bit counter and some comparison logic to ensure that there are no runs of length 26 or more in the random data. The test occupies 9 slices on a Virtex-5 FPGA.

If any of these tests fail during the operation of the TRNG, all previously generated random data is discarded and the TRNG is restarted.

6.8 Post-processing of TRNGs

As a TRNG circuit produces an output based on unpredictable physical sources of noise in the semiconductor fabric, a change in the operating conditions, such as temperature fluctuations or the presence of an external noise source, can cause a drop in the entropy that is present in the output of the TRNG. Post-processing of the bitstream can be used to mask these imperfections and maintain an output that has good statistical properties. Even when operating correctly, the TRNG may produce a bitstream that contains some bias i.e., the TRNG has a tendency to produce more 1's than 0's, or 0's than 1's. This bias can be removed by compressing the bitstream; therefore, increasing the entropy per bit.

Post-processing is usually implemented by seeding a PRNG, such as an LFSR with a long period, or any PRNG that is cryptographically secure [69]. A cryptographic hash function can also be used to post-process the TRNG output and should have excellent statistical properties. The downside to using a cryptographic hash function is that the area consumption is significantly more than LFSR based designs, due to the increased amount of logic required. However, this is a more robust method of post-processing and was therefore used for the TRNG design presented in this thesis. The output of the TRNG was passed through the CubeHash function that was described in Section 5.9. CubeHash was chosen as it was shown to have a low area consumption and a high throughput for long messages. As the hash function, for the purpose of post-processing the TRNG output, is required to operate on data blocks of 20000 bits, CubeHash is a better choice than a SHA256 based design. Although both have a similar area consumption, CubeHash has a higher throughput for long message lengths.

6.9 Secure Architecture Implementation Results

The overall design of the TRNG module is shown in Figure 6.11. The design includes the FIGARO TRNG which, when enabled, produces a stream of bits at 6.25 Mbit/s. This data is passed to the post-processing block through a *Serial-in, parallel-out* (SIPO) shift register, denoted s_reg. At the same time, the data is processed by the statistical tests which produce a pass/fail result for every 20000 bits that are processed. The controller block is responsible for managing the generation of random data and only releasing that data when it has passed the statistical tests. On start-up there is an initial latency in the output of data from this block. The statistical tests must wait for 20000 bits to be generated by the TRNG before any of the post-processed random

data can be released by the controller.

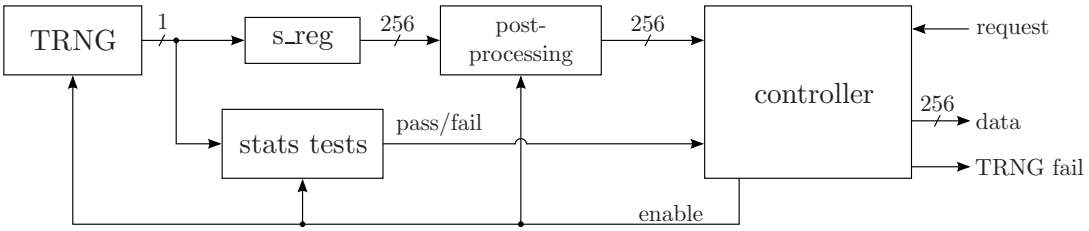


Figure 6.11: Complete TRNG design.

A breakdown of the area used by each part of the design is given in Table 6.6. It can be seen from the results that the vast majority of area is taken up by the CubeHash post-processing block. This is due to the fact that the internal state of the CubeHash block is 1024 bits and the operations performed in the compression function require more logic resources than the statistical tests, or FIGARO design. The FIGARO circuit consumes very little area as it only consists of effectively two ring oscillators, an XOR gate, and a flip-flop. The statistical tests can also be implemented very efficiently on an FPGA. It should be noted that the poker test uses a DSP block to implement the multiplication required by the test; all other tests use slice logic only. A DSP block is a hardcoded resource in the FPGA and is therefore not included in the slice count. The "complete design" result includes every component present in Figure 6.11.

Design	Area (slices)	slice LUTs	slice Reg's
complete design	734	2487	1337
stats tests	231	655	380
CubeHash	638	2325	1035

Table 6.6: TRNG implementation results.

6.10 Discussion

In this chapter, several TRNG designs were implemented on an FPGA and compared. Although some of the designs performed well by passing most of the statistical tests, it is clear that generating high quality random data from digital components alone is not a trivial task. When implementing a TRNG based solely on digital components, the resulting design requires a thorough statistical analysis, as the behaviour of the design will vary across different implementation platforms.

The final design, presented in Section 6.9 is far more robust than a TRNG alone.

The addition of statistical tests prevents against a complete failure of the TRNG, and ensures that only correctly produced random data is forwarded on to the rest of the system for use in cryptographic protocols. The application of post-processing also improves the security of the design, as any statistical bias in the raw binary stream from the TRNG will be removed.

In the previous chapters, many different cryptographic components have been discussed at the algorithmic level. In the next chapter, these components will be combined in order to analyse the design of a coprocessor at the protocol level.

Chapter 7

Coprocessor Design For the Protocol Level

7.1 Introduction

In Chapter 2, the TLS protocol and the necessary components to implement a TLS coprocessor were discussed. In the chapters that followed, coprocessor designs for ECC, hash functions, and random number generation were examined. The best candidate for a secure implementation was identified for each case. The designs presented in the previous chapters are usable as a standalone coprocessor, however, by combining them it is possible to create a full TLS coprocessor.

Many of the operations required by the TLS protocol, such as parsing incoming messages or verifying certain fields of digital certificates, can implemented in software. These types of operations would not benefit from hardware acceleration, as the time taken to transfer data to the coprocessor would be comparable to the time taken for the processor to access its own memory. However, performance can be increased by performing the computationally intensive operations, such as large multiplications, block ciphers, and hash functions in a coprocessor. With this approach, it is also possible to make the device more resistant to leaking the private keys of the user, as, if implemented correctly, storing the keys in hardware eliminates the possibility of an attacker mounting a software based attack. If the coprocessor is designed with security against hardware based attacks in mind, it can also be made more resistant to certain side channel attacks, in particular SPA and to a lesser extent DPA [65].

Secure key management is also of utmost importance for a cryptographic system. In Section 2.12, several TLS coprocessor designs from the literature were discussed,

however, their focus was on achieving high throughput designs and therefore a coprocessor at the protocol level that addresses the issue of secure key management has yet to be published. In this chapter, this problem will be addressed and a coprocessor for the acceleration of the TLS protocol will be introduced. The architecture will focus on the secure management and generation of secret key data.

7.2 Designing a Secure Coprocessor

Many embedded systems have different requirements, in terms of physical security of the device, than that of a personal computer or other large system that operates in a fixed location. Embedded devices, such as smartcards and keyfobs, are not usually kept in secure locations; therefore, an attacker has physical access to them. This gives an attacker more options in terms of the techniques that can be used to retrieve secret information from the device. Power consumption, electromagnetic radiation, and the time it takes the device to perform different operations can all be used to infer what data the device is processing at a given time.

The most valuable data in a cryptographic system, to an attacker, are the secret keys stored in the device. In the case of the TLS protocol, these secret keys are used to authenticate the device during the handshake protocol and to encrypt data in the record protocol. With access to these keys, an attacker can encrypt and decrypt data, forge digital signatures, or initiate connections on the network as if they were a legitimate user. The extent to which the system is compromised is then determined by the lifetime of the secret keys. It is feasible in many systems to have the private keys refreshed regularly; although this does degrade the performance, as a TLS handshake would have to be performed for each refresh. The system is then only compromised for the amount of time that the current keys are active, however, the attacker may then simply attempt the attack again. It is therefore advantageous to design a system that makes it difficult for an attacker to retrieve the secret keys in a timeframe shorter than that in which they are refreshed.

If a GPP is used to perform cryptographic functions in an embedded device, the private keys are usually stored in ROM or RAM. However, for the GPP to operate on these keys it must transfer them to its working memory. In many applications this working memory is a mixture of a small amount of registers internal to the processor and a larger memory bank located externally to the GPP device in the form of an SRAM or *Double Data Rate* (DDR) RAM module. This is a common setup in FPGA based systems where the internal memory of the FPGA is limited and mainly designed

to serve as small block RAM elements for custom logic functions. If memory internal to the FPGA is used to store the keys, they are vulnerable to software based attacks. If external memory is used, an attacker can simply read off the secret keys as they are transferred over an external bus. There are two solutions to this problem; all data sent over the bus to the external memory is encrypted or the keys are kept internal to the GPP. Encrypting all the data that is sent to external memory has an impact on performance for all operations that the device performs. This method also fails if an attacker develops the ability to run his own code on the device as it would be a simple task to recover the secret keys. A more secure solution is to isolate the keys from the GPP in a secure section of the chip, such as that proposed by Gaspar et al. in [40]. In this setup, if an attacker gains the ability to run code on the GPP, they will still not have access to the private keys. This limits the attacker's options of retrieving the private keys to performing side channel attacks or physically tampering with the device.

In order to provide a methodology for designing a secure coprocessor, several assumptions as to the level of access an attacker has to the device must be made.

1. An attacker has the ability to execute their own software on the GPP.

2. An attacker has physical access to the device itself. This would include how the device is powered and also any external connections to the device.

3. An attacker has full control over what data can be sent to the cryptographic coprocessor.

To fully protect a device against an attacker is nearly an impossible task. The cost of manufacturing such a device would also be prohibitively expensive. Making a device sufficiently secure involves ensuring that the financial cost and computing power required to retrieve the private keys makes it infeasible and also economically unprofitable to an attacker.

The design presented in this thesis attempts to provide security against an attacker in several ways:

1. The private keys and nonces should be isolated from the GPP. Under no circumstances should an attacker be able to retrieve the keys over the interface between the GPP and coprocessor.

2. At an algorithmic level the device should be protected against simple power analysis and differential power analysis attacks.

3. A random number generator should be present, internally to the chip, that produces high entropy random numbers and is also protected from being influenced by conditions external to the device by being able to detect a failure in the randomness of the data that it's producing.

7.3 Requirements of a TLS Coprocessor

The requirements of a coprocessor are determined by how the various TLS functions are partitioned between software running on the GPP and custom hardware in the coprocessor. Many TLS functions are suited to software implementation and would not reduce the security of the overall design if they were implemented by the GPP. The goal of the design presented in this work is to increase both the performance and security of the system through the use of a coprocessor; therefore, the coprocessor contains hardware modules that can be used to perform computationally intensive operations and any operation involving the private keys. The specific cryptographic operations required by the TLS protocol are discussed below; these operations have a direct impact on the structure of the final TLS coprocessor.

7.3.1 Public-key algorithms

The public-key algorithms used by the TLS handshake protocol were introduced in Chapter 3 and 4. The two key exchange algorithms that will be used in this implementation are:

- ECDH_ECDSA, which uses fixed ECDH keys and digital certificates signed with ECDSA. The fixed keys are present in the digital certificates.

- ECDHE_ECDSA, which uses ephemeral ECDH keys and digital certificates signed with ECDSA.

7.3.2 Private-key Algorithms

The TLS record protocol requires the ability to encrypt/decrypt messages exchanged between the server and the client. The design presented in this work uses AES in CBC mode as the encryption function. An AES key expansion operation generates the roundkeys from the shared secret established using ECDH.

7.3.3 Hashing Operations

Both the TLS handshake and TLS record protocol use hash functions in order to provide message integrity and authenticity during the transmission of messages. TLS makes use of both the standard hashing operation and also the HMAC operation defined in [68]. The work presented in this thesis uses the SHA256 algorithm for all hashing operations. The HMAC function generates a MAC that is appended to messages just before they are encrypted. The entity receiving the message can use the same operation just after a message has been decrypted, where a matching MAC indicates that the message has not been altered in transit and also infers the authenticity of the sender.

7.3.4 Operations Involving Private Keys

The *Finished* messages of the TLS handshake protocol include data that can be used to verify that the handshake process has completed successfully. The operation used to generate this *verification data* is based on the TLS pseudorandom function and involves the *master secret*. For this reason, a dedicated hardware module is incorporated in the coprocessor that is capable of calculating the *verification data*.

The secure management of the private keys is of paramount importance to the security of the overall system. A hardware block is incorporated into the coprocessor which is used to generate all of the secret key data required by the TLS protocol. The HMAC function requires two secret values, *mac_enc* and *mac_dec*. The *mac_enc* value is used for calculating the MAC for outgoing messages that are to be encrypted, while *mac_dec* is used for messages that have been decrypted. The AES function also requires two keys, one for encryption and another for decryption. They are denoted *aes_enc_key* and *aes_dec_key* respectively. The *master secret* is also generated by the key management module and is denoted Ω_m.

7.4 Encryption for TLS

The TLS protocol supports a wide range of algorithms for bulk data encryption. However, the obvious choice is to use AES, as it is the NIST standard block cipher and its cryptographic strength has been analysed for many years. AES [91] was introduced in 2001 as a replacement for the *Data Encryption Standard* (DES) [90], as DES's short key length of 56 bits was no longer considered to be secure.

AES is a symmetric-key block cipher based on the Rijndael algorithm [26]. AES is effectively identical to the Rijndael algorithm except the allowable key sizes and block

lengths have been fixed to 128, 192, and 256 bits. The 128 bit version of the algorithm will be discussed for the remainder of this chapter.

The input message to be encrypted, known as the plaintext P, is split into n 128 bit blocks, where $P = \sum_{x=1}^{n} P_x$. Padding is applied if the message length is not divisible by 128. Each plaintext block P_x is then fed into the AES encryption function E_k, along with a key of the same size. Each 128 bit plaintext block P_x produces a corresponding ciphertext block C_x, where $C_x = E_k(P_x)$ and the full ciphertext is then $C = \sum_{x=1}^{n} C_x$, this type of operation is known as *Electronic Code Book* (ECB) mode. ECB mode is not suitable for use in the TLS record protocol as it would be susceptible to attacks [114, page 189]. An alternative mode of operation, known as CBC mode is discussed in Section 7.4.1.

The AES encryption and decryption algorithms are shown in Figures 7.1 and 7.2. The AES encryption algorithm consists of four operations, SubBytes, ShiftRows, MixColumns, and AddRoundKey. One pass through these four operations is known as a *round*. The output of the AddRoundKey operation is then fed back into the SubBytes operation. In the case of 128 bit AES, nine standard rounds are performed, followed by a final round with the MixColumns operation omitted; the round number is denoted N_r. The roundkey values $R_k[0], R_k[1] \ldots R_k[N_r]$, are derived from the key through the use of the AES key expansion, see [91].

The AES decryption algorithm is simply the inverse of the encryption algorithm. Each of the operations SubBytes, ShiftRows, and MixColumns are invertable; their inverses denoted InvSubBytes, InvShiftRows, and InvMixColumns. The AddRoundKey operation is simply a bitwise XOR operation and is therefore its own inverse.

The AES *state* is 128 bits in length and can be represented as a 4×4 matrix consisting of 16 bytes S_{ij}, where i is the row and j is the column of the matrix.

SubBytes: The SubBytes operation uses an invertable substitution box, S-box, derived from the multiplicative inverse over GF(2^8) where the value 00_{16} is mapped to itself. For all other values, given an 8 bit byte represented by $X = \sum_{i=0}^{7} x_i$ and $C = 01100011_2 = \sum_{i=0}^{7} c_i$, the following transformation, Equation 7.1, is applied to each byte $S_{ij} = X$ of the state matrix.

$$x'_i = x_i \oplus x_{(i+4) \bmod 8} \oplus x_{(i+5) \bmod 8} \oplus x_{(i+6) \bmod 8} \oplus x_{(i+7) \bmod 8} \oplus c_i \qquad (7.1)$$

ShiftRows: Each row of the *state* is cyclically shifted left by i positions, where i is the row index of the *state* matrix.

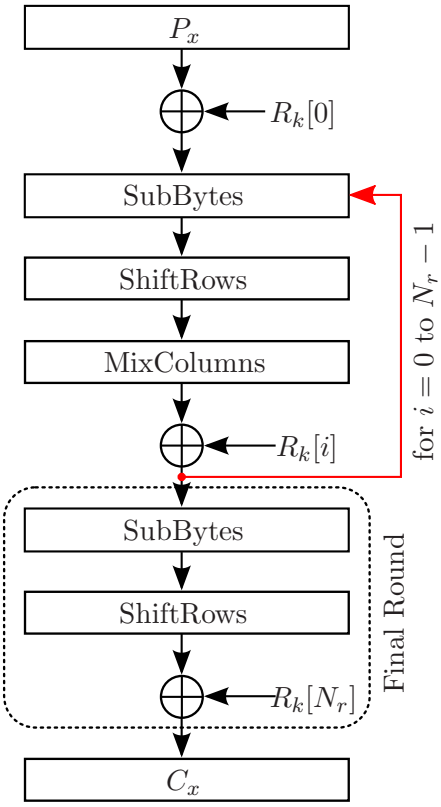


Figure 7.1: AES encryption.

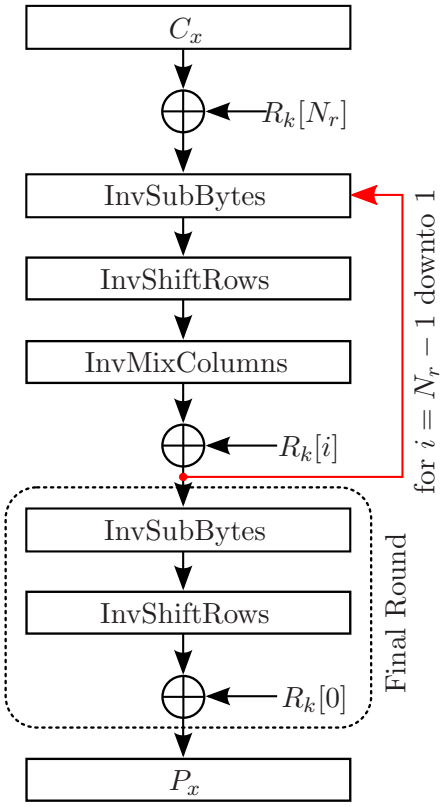


Figure 7.2: AES decryption.

MixColumns: Each column of the *state* is treated as a polynomial over GF(2^8) and multiplied modulo $x^4 + 1$ with the polynomial $a(x) = 03_{16}x^3 + 01_{16}x^2 + 01_{16}x^1 + 02_{16}$

AddRoundKey: A round key, generated by the AES key schedule, is XOR'd with the output of the MixColumns step for rounds 1 to 9, and with the output of the ShiftRows step for the final round.

7.4.1 Cipher Block Chaining

The AES module in the coprocessor implements the AES key expansion and performs AES encryption and decryption functions in CBC mode. In CBC mode, the ciphertext output from one 128 bit block is fed into the next iteration of the algorithm. Figures 7.3 and 7.4 show the AES algorithm operating in CBC mode, where $P_1, P_2 \ldots P_n$ is the plaintext message to be encrypted, split into 128 bit blocks. $C_1, C_2 \ldots C_n$ is the output ciphertext, split into 128 bit blocks. The initialisation vector is denoted IV.

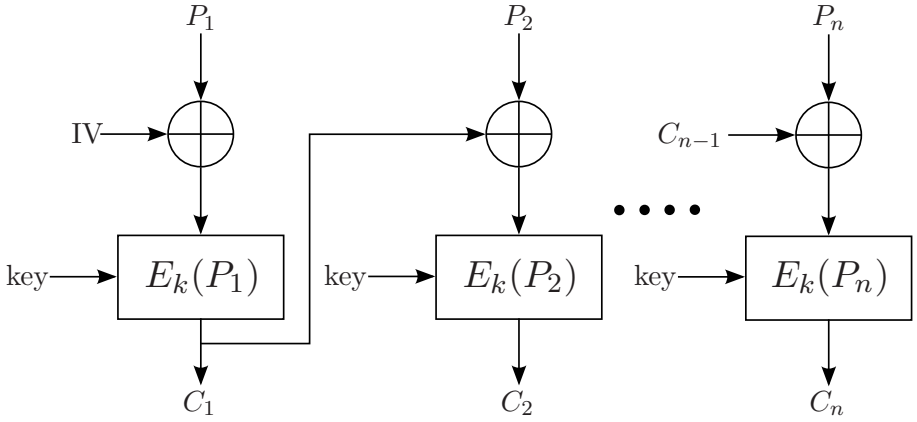


Figure 7.3: AES encryption in CBC mode.

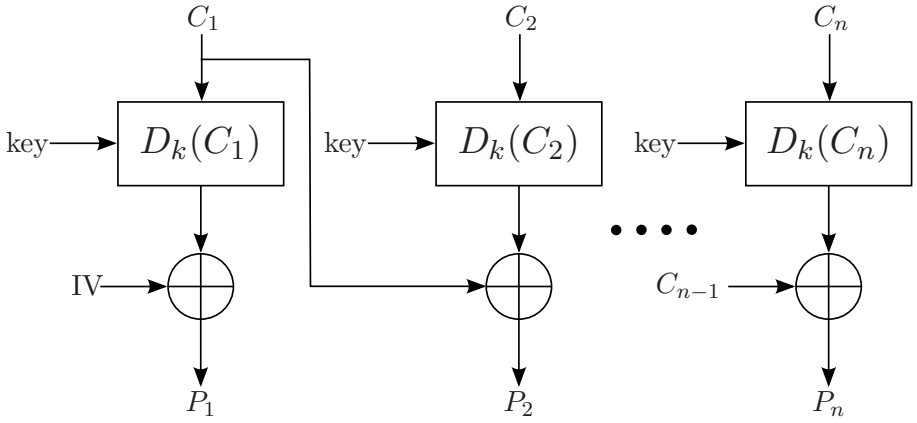


Figure 7.4: AES decryption in CBC mode.

7.4.2 AES Implementation

A detailed analysis of hardware architectures for AES is given in [136]. For each iteration of the AES algorithm, the encryption key must be expanded so that it can be combined with the state during each round. Two separate key expansion blocks are used in the design: one generates the keys for message encryption, the other generates the keys for message decryption. The AES core was implemented such that each round takes two clock cycles to complete. The second clock cycle is required as the SubBytes operation is implemented as a lookup table in BRAM. Table 7.1 shows the performance results when implemented in software versus hardware. The simulation results indicate what would be an upper limit on the performance of the AES module. The simulation results do not take into account the padding that is done in software and hence there is a significant decrease in performance between the simulation results and hardware

implementation.

Msg. Size (bits)	sim	Microblaze SW	Microblaze SW + HW
512	1563	1.286	5.590
100 K	1563	1.284	5.718

Table 7.1: AES performance results in Mbit/s.

Table 7.2 shows the area consumed by the final design. The AES core result includes both the encryption and decryption functions for AES.

Component	Area (slices)	BRAM
AES core	1525	9
Key Expansion	660	1

Table 7.2: AES area results.

7.5 SHA256 Implementation for TLS

Hash functions were discussed in Chapter 5 and are a key component of the TLS protocol. A SHA256 module is therefore included in the coprocessor design to provide all hash related operations.

In order to test its performance, the SHA256 core was implemented alongside a Microblaze processor on a Virtex XC5VLX110T FPGA, similar to the structure shown in Figure 2.9 in Chapter 2. The usual method for implementing a hash function is to process each data block as it becomes available, the state of the hash function is then stored. This removes the need to have to store all previous handshake messages. Firstly, this would increase the amount of data sent to the hash function, as the state of the hash function would have to be loaded into hardware with each message processed. Secondly, allowing the state of the hash function to be loaded for each message would reduce the security of the implementation, by allowing an adversary to view intermediate values of the hash function state. Therefore, new drivers were written that store the handshake messages, apply padding, and then, only when the hash is required are the messages processed.

The compression function of the SHA256 algorithm is implemented such that one round takes a single clock cycle and 64 rounds are required in total for the processing of one 512 bit message block. The SHA256 core occupies 2112 slices. The performance results are shown in Table 7.3; the results assume that the circuit is clocked at 75 MHz.

sim	Microblaze SW	Microblaze SW + HW
383	0.98	11.65

Table 7.3: SHA256 performance results in Mbit/s.

7.6 Design Overview

Figure 7.5 shows the construction of the coprocessor. The design is broken up into three sections: FSL bus, control logic, and hardware modules. Two FSL bus connections are present, one for receiving data, and one for transmitting data. The control logic is used to decode the incoming instructions and forward data to the relevant hardware modules for processing. The majority of the coprocessor contains the hardware modules that were discussed in the previous chapters. All private data is stored in the key manager and each cryptographic module that requires secret key data has a connection to the key manager. Any data sent by the GPP is sent through the input registers and then forwarded to the correct cryptographic module. The routing of data, internally in the coprocessor, is achieved by configuring several routing multiplexers depending on the instruction that the coprocessor is currently processing.

Each cryptographic module also has their own FSM that handles the communication with the control logic FSM. The ECDSA processor controller is used to load the correct values into the BRAM of the ECDSA processor and also check the result of the signature verification algorithm.

In order to minimise the impact on the area of the design, a single SHA256 core is shared between the key manager, HMAC module, and the finished message calculation module. The hash function core can be shared among many functions. This is done through the use of routing multiplexers, external to the SHA256 core.

7.7 Hardware/Software Partition

A key element in designing a coprocessor is dividing the workload between the GPP and the coprocessor. The coprocessor is mainly required to increase the performance of the design. If the entire protocol was implemented in hardware it would achieve the highest performance. However, the area consumption of such a design would usually be unacceptable. On an FPGA platform, the designer is constrained by the number of slices and hardware blocks present in the FPGA; hence, it is usually best practice to implement very computationally intensive parts of the protocol in FPGA logic. The strength of a GPP is that it can be configured to implement very broad set of tasks,

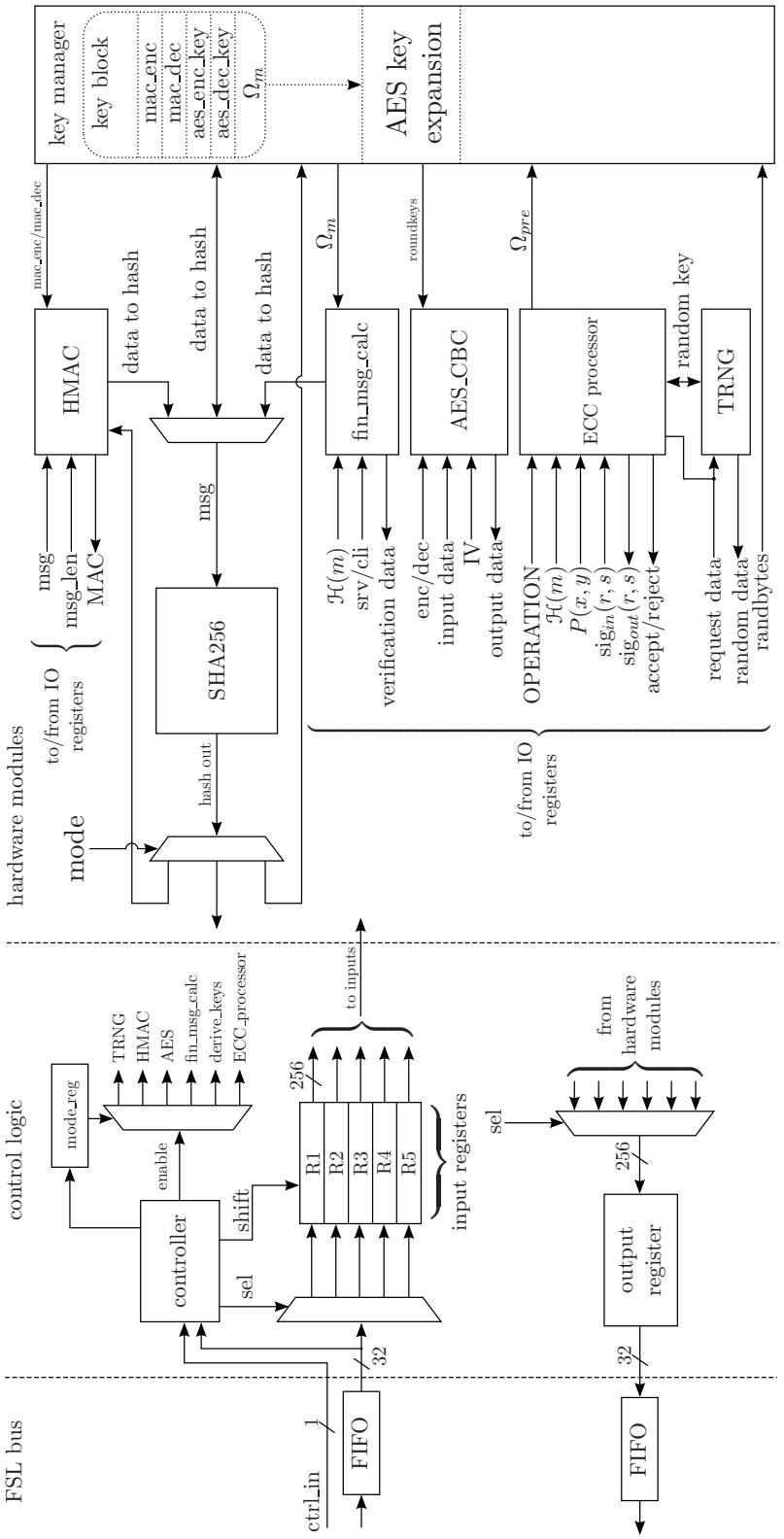


Figure 7.5: Coprocessor architecture.

by increasing the code size, rather than the amount of logic that the design consumes. Decision making, such as checking message fields, are a prime example of a task that is suited to a GPP. These types of tasks also do not involve secret key information.

Many open-source libraries that support TLS are freely available on the Internet; OpenSSL [103] and GNUTLS [75] are two popular example. The PolarSSL library [19] was chosen for use in this design as it offers an easy platform in which to add support for a coprocessor. The PolarSSL library does not support elliptic curve routines and, therefore, modifications were made to add support for elliptic curve arithmetic and also the ability to parse certificates based on elliptic curve algorithms. The design has been implemented such that a comparison can be made between the code running entirely in software, or offloading different cryptographic tasks to the coprocessor, to a varying degree. However, all tasks must be offloaded to the coprocessor for the design to conform to the secure implementation methodology introduced in Section 7.2.

7.8 Coprocessor Operation

The coprocessor accepts certain instructions from the connected microprocessor. The instructions are designed to fulfil the cryptographic requirements of the TLS protocol that were discussed in Section 7.3. The instructions are:

- Reset the coprocessor
- Sign a message with ECDSA
- Verify the ECDSA signature of a message
- SHA256
- SHA256 HMAC
- Encrypt a message using AES
- Decrypt a message using AES
- Generate random data
- Generate the key block
- Calculate the finished message verification data

Instructions are sent over the FSL bus with the control signal set high to indicate that the accompanying 32 bit block of data is an instruction to be decoded. Upon receiving an instruction message, the processor control unit decodes the message and sets the mode register for the current instruction. The mode register determines, to which core the data will be sent. The control unit then reads in all of the relevant data and forwards it from the input registers, (R1, R2 ..., R5), to the correct module. The control unit then waits for the data to be processed and then returns the data to the Microblaze via the FSL bus. The control unit then sets the mode register back to *general* mode, which indicates that the coprocessor is ready to process the next instruction.

A detailed description of how each instruction is executed, is given below.

- Reset the coprocessor: Upon receiving this instruction, the controller sends a reset signal to all arithmetic modules. This signal clears all data inside the coprocessor, including any keys for the current TLS session. The private key corresponding to the coprocessor certificate is hardcoded in the processor and can only cleared by reprogramming the FPGA.

- Sign a message with ECDSA: For this operation to be performed, all keys must have already been setup in the coprocessor. The processor receives the instruction followed by the 256 bit hash of the message it should sign. Once the message has been signed, the coprocessor returns the signature to the Microblaze via the FSL bus.

- Verify the signature of a message: This instruction does not require any secret key information and receives all necessary data via the FSL bus. The coprocessor receives the instruction followed by the signer's public key and the signed message. Upon completion, the coprocessor returns a value indicating whether the signature was correct or incorrect.

- Generate the premaster secret: This operation implements the final stage of the ECDH algorithm, where, an elliptic curve point multiplication is used to generate the *premaster secret* Ω_{pre}. The x coordinate of the resultant elliptic curve point is used as the *premaster secret*. Ω_{pre} is then transferred to the key manager where it's stored until the generate key block instruction is received.

- SHA256 & SHA256 HMAC: The *mac_enc* and *mac_dec* values must have been generated by the key manager prior to receiving an instruction to perform a

HMAC operation. No *Initialisation Vectors* (IVs) are required for a normal SHA256 hashing operation. The coprocessor receives the instruction followed by the padded message to operate on.

- AES encrypt/decrypt: The AES roundkeys must have already been generated by the coprocessor for it to be possible to execute this instruction. The coprocessor receives the instruction followed by the padded message to be encrypted or decrypted.

- Generate Random Data: This instruction generates 512 bits of random data using the TRNG. The data is post-processed and checked for randomness before it's returned. If the randomness tests fail, an error code is returned to the Microblaze.

- Generate key block: This instruction takes as input, the random bytes received from the server or client (i.e., *randbytes*). Ω_{pre} must also have been computed prior to receiving this instruction. No output is returned from the coprocessor for this instruction. All keys and data remain internal to the coprocessor.

7.9 Test Platform

The design was implemented on a Xilinx XUPv5 evaluation board which consists of a Xilinx Virtex XC5VLX110T FPGA. The board also contains 256 MB of DDR RAM. The bitstream is loaded onto the FPGA through the use of a compact flash card on the board. The design uses a Microblaze processor as the GPP for testing purposes, as shown in Figure 2.9. The coprocessor is a generic structure and could be implemented alongside any GPP. The Microblaze has access to the DDR RAM through the use of a memory controller. Network communications for the Microblaze are done through one of the hard Ethernet MACs, present inside the FPGA. External to the FPGA there is also an Ethernet PHY, which is connected to an Ethernet cable. The evaluation board was connected to a PC using an Ethernet cable and the TLS handshake process was performed between the PC and evaluation board using this link.

7.9.1 Microblaze Configuration

Version 8.30 of the Microblaze was configured for use with a version of the Xilinx maintained Linux kernel. For this to be possible, the Microblaze must be configured with a memory management unit. The Microblaze was also configured to use a full

32×32 bit multiplier, which is capable of generating a full 64 bit result. The instruction cache and data cache were both configured to be 16 kB in size.

7.10 Implementation Results

The post map area results for the system are shown in Table 7.4, where CP denotes the coprocessor circuit and μB is the Microblaze. The top four results show the area consumption with a varying number of multiplier units used in the ECDSA processor. These results are post place and route, while the bottom of the table shows post map results. Post map results were used, as a breakdown of the area occupied by submodules in the design cannot be generated for post place and route results. Some of the modules share resources; hence, modules such as HMAC have a smaller area than SHA256. The HMAC module uses the SHA256 module for hashing operations. The HMAC module is therefore mostly control logic and hence has a very small area. The design also uses 5 DSP48E blocks, of which 4 are used by the Microblaze's multiplier circuit, and 1 is used to implement the poker test in the post processing circuit of the TRNG. The Xilinx Virtex XC5VLX110T FPGA used to generate the results, in total, contains 17280 slices, 148 36 kB BRAM blocks, and 64 DSP48E blocks.

Design	Area (slices)	slice LUTs	slice Reg's	BRAM	DSP48E
μB + CP 1M	13526	36124	27352	76	5
μB + CP 2M	13797	37148	28132	76	5
μB + CP 3M	14203	38722	28651	76	5
μB + CP 4M	14403	40011	29689	76	5
CP with 3 multipliers (post map)					
full CP	10088	28508	18273	23	1
ECDSA Proc.	4624	12117	8234	13	0
AES	1525	2366	270	9	0
AES key exp	664	1157	1416	1	0
SHA256	2112	5165	1033	0	0
HMAC	73	20	267	0	0
Key manager	448	50	1682	0	0
fin msg calc	321	288	619	0	0
TRNG	1170	3217	1700	0	1

Table 7.4: Coprocessor area results.

The timing results for an average handshake are shown in Table 7.5, where the coprocessor's ECDSA unit contains 3 multipliers. The GPP result is the average time taken to complete a TLS handshake when the Microblaze has no access to the copro-

cessor. These timing results are also dependent on the performance of the PC that takes part in the handshake process. It can be seen, however, that there is a huge performance increase when the coprocessor is used to offload the computationally intensive operations. The poor performance of the GPP only design is due to the fact that the Microblaze is not a very powerful processor and is not designed for performing the large amounts of finite field arithmetic found in the ECC algorithms.

Cipher Suite	μB (ms)	μB + CP (ms)
ECDH_ECDSA	15400	132
ECDHE_ECDSA	11300	152

Table 7.5: Average TLS handshake time.

In Section 2.12.2 an SSL security processor by Wang et al. was discussed. The design is a high speed architecture and is capable of performing a full SSL handshake in 0.61 ms. The coprocessor presented in this chapter is not capable of performing SSL/TLS handshakes as quickly as the design by Wang et al., however, the two architectures cannot be compared directly. The design by Wang et al. consumes much more area and is a highly parallel architecture. In contrast, the goal of the design presented in this chapter was to derive an architecture that manages the keys securely. The resulting architecture is therefore less efficient in terms of the resources it uses as not all of the logic is used to increase performance. However, the architecture presented in this chapter offers a much more secure implementation. This comes as a result of including protection of the secret key data, performing ECC related operations in a side channel attack resistant way, and including a TRNG in the design which incorporates failure detection and post-processing of random data.

7.11 Conclusions

In this chapter, a secure coprocessor architecture has been given. The performance of the system was analysed in a real world environment and the results show a huge performance increase can be achieved by using a coprocessor in an embedded device. However, the main result is the architecture that increases security against side channel and software based attacks. A fully software based design would have significant weaknesses when subjected to software based attacks. A specific partitioning of functions between hardware and software, in order that the GPP does not require access to the private keys, has been described. The structure presented is also portable across multiple platforms.

It has been shown that with the help of a coprocessor, a GPP can achieve a significant performance increase. This, however, is to be expected as the Microblaze used in this design is not optimised to provide high performance in this setting.

Chapter 8

Conclusions and Future Work

8.1 Contribution to the Field

The overall theme of the work presented in this thesis has been on the design of cryptographic coprocessors for use in embedded systems. In this section, a summary of the main contributions of this thesis to the area will be discussed. In Chapter 2, a background to coprocessor design at the protocol level was introduced. Relevant mathematical principles, algorithms, and protocols were discussed with a focus on their implementation on an FPGA platform. An analysis of currently published work on the area of cryptographic coprocessor design was conducted and it was found that a secure architecture has yet to appear in the literature. Therefore, the remaining chapters of this thesis investigated the implementation of different cryptographic components required to implement a secure coprocessor at the protocol level.

In Chapter 3, an introduction to ECC was given. Various algorithms were presented and compared in a hardware-software co-design setting. From the results obtained, it was confirmed that finite field multiplication was the dominant factor in determining the overall computation time in a software implementation. Following this result, various multiplier architectures were presented in order to analyse the effect of ISE on the performance of the system. It was shown that by using ISE, the dominance of the finite field multiplications can be reduced. While most ECC algorithms presented in the literature are designed with the goal of minimising the number of multiplication operations at the expense of extra additions and subtractions, the results from Chapter 3 shows that this does not always result in the fastest algorithm across all platforms; as addition and subtractions can become on influencing factor in performance.

Chapter 4 extends the work of Chapter 3 by analysing the implementation of ECC

algorithms using an ECC processor built from FPGA logic. As the processor was custom designed for the implementation of ECC algorithms, it was expected that significant performance over a software based system could be achieved. Once again, an analysis of the various ECC algorithms was performed. The main contribution of Chapter 4, however, is the introduction of a processor capable of performing the ECDSA algorithm. This architecture was developed from the existing work presented by Byrne et al.. By modifying the architecture of Byrne et al., a more flexible design was achieved; capable of performing more than one ECC algorithm per processor.

A common operation required by cryptographic protocols is the hashing of data; hence, in Chapter 5, an analysis of hash function architectures was given. A comparison of some of the designs from the recent SHA-3 competition was performed, on an FPGA platform. A fair comparison methodology was introduced in order to accurately and fairly compare the performance of hash functions of differing architectures. A generic wrapper structure was developed that incorporated padding. Using this structure a fairer comparison can be made between the hash functions, as the impact of their different padding schemes can be taken into account.

A key component of many cryptographic systems is a circuit capable of generating high quality random data; therefore, in Chapter 6, various TRNG designs were analysed. Some of the more recent designs published in the literature were implemented on a Virtex 5 FPGA and their output statistically tested in order to ensure that the circuit generates statistically high quality random data. The best performing TRNG design on the Virtex 5 platform was identified and the method of failure detection introduced by Santoro et al. incorporated into the TRNG design. Using the results from Chapter 5 the CubeHash hash function was chosen for use as a post-processing mechanism for the TRNG, due to its low area consumption and good throughput for long message sizes, when compared to SHA256. The TRNG design achieved allows for high quality random data to be produced in a secure manner.

The final chapter of work presented in this thesis, Chapter 7, defined a secure architecture for a coprocessor, suitable for the acceleration of the TLS protocol. The TLS protocol was discussed and the functions requiring acceleration by a coprocessor were identified. These functions were chosen with the goal of offloading all operations, involving secret key information, to the coprocessor. This architecture leads to a design where software based attacks are protected against; as the GPP is never operating on sensitive data. The processor was implemented in different configurations and a final design chosen in order to maximise the parallel multiplication capabilities of the architecture. Finally, the design was implemented in a real world environment and

the performance evaluated. In comparison to other designs that have been published previously, this work has presented the most detailed analysis of a TLS coprocessor to date.

8.2 Future Work

Although a fully working design was presented in this thesis, there are areas in which further research might yield improvements to the architecture. One of the main drawbacks of the design is that the coprocessor is only capable of handling one TLS session at a time. This was done in order to simplify the interface with the coprocessor and allow for a more secure design. Additional instructions would be required in order to allow the GPP to specify a TLS session ID for each block of data that is sent to the coprocessor; thus, requiring the coprocessor to store key data for multiple sessions. As all of the keys are stored in registers in the coprocessor, memory space is limited. A possible solution is to have an external memory component, where all data transfers to and from the memory are encrypted. This was ruled out as a possibility for use with the GPP, as there would have been a large impact on the performance of the GPP. However, the same is not true if implemented as an external memory bank for the coprocessor, as the coprocessor only requires the key data to be transferred to the external memory location. All intermediate results are still kept internal to the coprocessor; therefore, the impact on performance is only on the initial key transfer.

Another area of possible research is the definition of how the coprocessor should respond to denial of service attacks. These form of attacks are currently possible if an attacker can cause the TRNG to fail to produce random data. If this happens, the output of the TRNG will never pass the statistical tests; therefore, random data will never be released from the TRNG post-processing block. Further investigation of this area would involve the definition of how the coprocessor should respond to such an attack in order to minimise its impact on the system, without any loss in security.

In terms of the ECC processor module of the coprocessor, there many areas of research that could be carried out. The design presented in this thesis, worked with NIST standardised curves, however, many different implementations have been proposed that use special forms of curves and moduli. In [38], the authors present a construction based on a Mersenne prime where $p = 2^{127} - 1$, over the extension field $\mathbb{F}_{p^2}$, and make use of the *Gallant, Lambert and Vanstone* (GLV) method to speed up the point scalar multiplication. The design could then make use of one of the multipliers discussed in Chapter 3 and may lead to a more efficient design.

Appendix A

Co-Z Algorithms

In Chapter 3, co-Z algorithms were introduced. In this section, the remaining co-Z operations are presented.

Algorithm 24 Co-Z addition with update (ZADDU)

Require: $P = (X_1, Y_1, Z)$ and $Q = (X_2, Y_2, Z)$
Output: $(R, P) \leftarrow \text{ZADDU}(P, Q)$ where $R \leftarrow P + Q = (X_3, Y_3, Z_3)$ and $P \leftarrow (\lambda^2 X_1, \lambda^3 Y_1, Z_3)$ with $Z_3 = \lambda Z$ for some $\lambda \neq 0$

1: **function** ZADDU(P, Q)
2: $C \leftarrow (X_1 - X_2)^2$
3: $W_1 \leftarrow X_1 C$
4: $W_2 \leftarrow X_2 C$
5: $D \leftarrow (Y_1 - Y_2)^2$
6: $A_1 \leftarrow Y_1(W_1 - W_2)$
7: $X_3 \leftarrow D - W_1 - W_2$
8: $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$
9: $Z_3 \leftarrow Z(X_1 - X_2)$
10: $X_1 \leftarrow W_1$
11: $Y_1 \leftarrow A_1$; $Z_1 \leftarrow Z_3$
12: **return** $(R = (X_3, Y_3, Z_3), P = (X_1, Y_1, Z_1))$
13: **end function**

Algorithm 25 Conjugate co-Z addition (ZADDC)

Require: $P = (X_1, Y_1, Z)$ and $Q = (X_2, Y_2, Z)$
Output: $(R, S) \leftarrow \text{ZADDC}(P, Q)$ where $R \leftarrow P + Q = (X_3, Y_3, Z_3)$ and $S \leftarrow P - Q = (\overline{X_3}, \overline{Y_3}, Z_3)$

function ZADDC(P, Q)
  $C \leftarrow (X_1 - X_2)^2$
  $W_1 \leftarrow X_1 C$; $W_2 \leftarrow X_2 C$
  $D \leftarrow (Y_1 - Y_2)^2$
  $A_1 \leftarrow Y_1(W_1 - W_2)$
  $X_3 \leftarrow D - W_1 - W_2$
  $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$
  $Z_3 \leftarrow Z(X_1 - X_2)$
  $\overline{D} \leftarrow (Y_1 + Y_2)^2$
  $\overline{X_3} \leftarrow \overline{D} - W_1 - W_2$
  $\overline{Y_3} \leftarrow (Y_1 + Y_2)(W_1 - \overline{X_3}) - A_1$
  return $R = (X_3, Y_3, Z_3)$, $S = (\overline{X_3}, \overline{Y_3}, Z_3)$
end function

Algorithm 26 Out-of-place differential addition-and-doubling 1 (AddDblCoZ1)

Require: $X_1, X_2, Z, x_D, a, 4b$
Output: X_1', X_2', Z'

function AddDblCoZ1(X_1, X_2, Z)
  $U = (X_1 - X_2)^2$
  $V = 4X_2(X_2^2 + aZ^2) + 4bZ^3$
  $X_1' = V\left[2(X_1 + X_2)(X_1X_2 + aZ^2) + 4bZ^3 - x_D ZU\right]$
  $X_2' = U\left[(X_2^2 - aZ^2)^2 - 8bZ^3X_2\right]$
  $Z' = UVZ$
end function

Algorithm 27 Out-of-place differential addition-and-doubling 2 (AddDblCoZ2)

Require: $X_1, X_2, Z, x_D, a, 4b$
Output: X_1', X_2', Z'

function AddDblCoZ2(X_1, X_2, Z)
  $U = (X_1 - X_2)^2$
  $V = 4X_2(X_2^2 + aZ^2) + 4bZ^3$
  $X_1' = V[(X_1 + X_2)(X_1^2 + X_2^2 - U + 2aZ^2) + 4bZ^3 - x_D ZU]$
  $X_2' = U\left[(X_2^2 - aZ^2)^2 - 8bZ^3X_2\right]$
  $Z' = UVZ$
end function

Algorithm 28 Out-of-place differential addition-and-doubling 3 (AddDblCoZ3)

Require: $X_1, X_2, T_D = x_D Z, T_a = aZ^2, T_b = 4bZ^3$
Output: $X_1', X_2', T_D', T_a', T_b'$

function AddDblCoZ3(X_1, X_2, Z)
  $U = (X_1 - X_2)^2$
  $V = 4X_2(X_2^2 + T_a) + T_b$
  $W = UV$
  $T_D' = T_D W$
  $T_a' = T_a W^2$
  $T_b' = T_b W^3$
  $X_1' = V\left[(X_1 + X_2)(X_1^2 + X_2^2 - U + 2T_a) + T_b\right] - T_D'$
  $X_2' = U\left[(X_2^2 - T_a)^2 - 2X_2 T_b\right]$
end function

Algorithm 29 Co-Z doubling-addition with update (ZDAU)

Require: $P = (X_1, Y_1, Z)$ and $Q = (X_2, Y_2, Z)$
Output: $(R, Q) \leftarrow$ ZDAU(P, Q) where $R \leftarrow 2P + Q = (X_3, Y_3, Z_3)$ and $Q \leftarrow (\lambda^2 X_2, \lambda^3 Y_2, Z_3)$ with $Z_3 = \lambda Z$ for some $\lambda \neq 0$

function ZDAU(P, Q)
  $C' \leftarrow (X_1 - X_2)^2$
  $W_1' \leftarrow X_1 C'$
  $W_2' \leftarrow X_2 C'$
  $D' \leftarrow (Y_1 - Y_2)^2$
  $A_1' \leftarrow Y_1(W_1' - W_2')$
  $\hat{X}_3' \leftarrow D' - W_1' - W_2'$
  $C \leftarrow (\hat{X}_3' - W_1')^2$
  $Y_3' \leftarrow [(Y_1 - Y_2) + (W_1' - \hat{X}_3')]^2 - D' - C - 2A_1'$
  $W_1 \leftarrow 4\hat{X}_3' C$
  $W_2 \leftarrow 4W_1' C$
  $D \leftarrow (Y_3' - 2A_1')^2$
  $A_1 \leftarrow Y_3'(W_1 - W_2)$
  $X_3 \leftarrow D - W_1 - W_2$
  $Y_3 \leftarrow (Y_3' - 2A_1')(W_1 - X_3) - A_1$
  $Z_3 \leftarrow Z\big((X_1 - X_2 + \hat{X}_3' - W_1')^2 - C' - C\big)$
  $\overline{D} \leftarrow (Y_3' + 2A_1')^2$
  $X_2 \leftarrow \overline{D} - W_1 - W_2$
  $Y_2 \leftarrow (Y_3' + 2A_1')(W_1 - X_2) - A_1$
  $Z_2 \leftarrow Z_3$
  return $(R = (X_3, Y_3, Z_3),\ Q = (X_2, Y_2, Z_2))$
end function

Algorithm 30 (X, Y)-only co-Z conjugate-addition–addition with update (ZACAU′)

Require: $P' = (X_1, Y_1)$ and $Q' = (X_2, Y_2)$ for some $P = (X_1, Y_1, Z)$ and $Q = (X_2, Y_2, Z)$, and $C = (X_1 - X_2)^2$

Output: $(R', S', C) \leftarrow \text{ZACAU}'(P', Q', C)$ where $R' \leftarrow (X_3, Y_3)$ and $S' \leftarrow (X_4, Y_4)$ for some $R = 2P = (X_3, Y_3, Z_3)$ and $S = P + Q = (X_4, Y_4, Z_4)$ such that $Z_3 = Z_4$, and $C \leftarrow (X_3 - X_4)^2$

function ZACAU′(P', Q', C)
  $W_1 \leftarrow X_1 C$
  $W_2 \leftarrow X_2 C$
  $D \leftarrow (Y_1 - Y_2)^2$
  $A_1 \leftarrow Y_1(W_1 - W_2)$
  $X_1' \leftarrow D - W_1 - W_2$
  $Y_1' \leftarrow (Y_1 - Y_2)(W_1 - X_1') - A_1$
  $\overline{D} \leftarrow (Y_1 + Y_2)^2$
  $X_2' \leftarrow \overline{D} - W_1 - W_2$
  $Y_2' \leftarrow (Y_1 + Y_2)(W_1 - X_2') - A_1$
  $C' \leftarrow (X_1' - X_2')^2$
  $X_4 \leftarrow X_1' C'$
  $W_2' \leftarrow X_2' C'$
  $D' \leftarrow (Y_1' - Y_2')^2$
  $Y_4 \leftarrow Y_1'(X_4 - W_2')$
  $X_3 \leftarrow D' - X_4 - W_2'$
  $C \leftarrow (X_3 - X_4)^2$
  $Y_3 \leftarrow (Y_1' - Y_2' + X_4 - X_3)^2 - D' - C - 2Y_4$
  $X_3 \leftarrow 4X_3$
  $Y_3 \leftarrow 4Y_3$
  $X_4 \leftarrow 4X_4$
  $Y_4 \leftarrow 8Y_4$
  $C \leftarrow 16C$
  return $(R' = (X_3, Y_3),\ S' = (X_4, Y_4),\ C)$
end function

A.1 Point Doubling Formulæ with Update in Homogeneous Coordinates.

A double of point $P = (X_1, Y_1, Z_1)$ on $E_{\mathcal{H}}$, denoted $\text{DBL}_{\mathcal{H}}$, is computed as $2P = (X_3, Y_3, Z_3)$ with the $\text{DBL}_{\mathcal{H}}$ operation. The cost of the operation is $\{5\mathbf{M}, 6\mathbf{S}, 1\mathbf{C}\}$. Where, $\mathbf{C}$ denotes multiplication by a constant.

Algorithm 31 $\text{DBL}_{\mathcal{H}}$ Operation

Input: $P = (X_1, Y_1, Z_1)$, $a = A$ paramter of elliptic curve $E_{\mathcal{H}}$.
Output: $(R) \leftarrow \text{DBL}_{\mathcal{H}}(P)$ where $R = 2P = (X_3, Y_3, Z_3)$.

1: **function** $\text{DBL}_{\mathcal{H}}(P)$
2: $A = 2(aZ_1^2 + 3X_1^2)$
3: $B = Y_1 Z_1$
4: $C = 2[(X_1 + Y_1 B)^2 - X_1^2 - (Y_1 B)^2]$
5: $D = A^2 - 8C$
6: $X_3 = 4BD$
7: $Y_3 = A(4C - D) - 64(Y_1 B)^2$
8: $Z_3 = 64B^3$
9: **return** $(R = (X_3, Y_3, Z_3))$
10: **end function**

If $Z_1 = 1$, the cost drops to $\{3\mathbf{M}, 5\mathbf{S}\}$, with

Algorithm 32 $\text{DBL}_{\mathcal{H}}$ Operation (with $Z_1 = 1$)

Input: $P = (X_1, Y_1, Z_1)$, $a = A$ paramter of elliptic curve $E_{\mathcal{H}}$.
Output: $(R) \leftarrow \text{DBL}_{\mathcal{H}}(P)$ where $R = 2P = (X_3, Y_3, Z_3)$.

1: **function** $\text{DBL}_{\mathcal{H}}(P)$
2: $A = 2(a + 3X_1^2)$
3: $\alpha = Y_1^2$
4: $B = \alpha^2$
5: $C = 2[(X_1 + \alpha)^2 - X_1^2 - B]$
6: $D = A^2 - 8C$
7: $X_3 = 4Y_1 D$
8: $Y_3 = A(4C - D) - 64B$
9: $Z_3 = 64Y_1 \alpha$
10: **return** $(R = (X_3, Y_3, Z_3))$
11: **end function**

As with other co-Z algorithms, an updated representation of the input P, such that it has an equivalent Z coordinate to the output point R, is required. This can be evaluated, in this case, at the cost of one extra multiplication.

$$\tilde{P} = (64Y_1\alpha \cdot X_1, 64B, 64Y_1\alpha) \sim (X_1, Y_1, Z_1) = P.$$

Let $(\tilde{P}, 2P) \leftarrow \text{DBLU}_{\mathcal{H}}(P)$ denote the corresponding operation, where $\tilde{P}$ and P share the same Z coordinate. The cost of $\text{DBLU}_{\mathcal{H}}$ operation (doubling with update) is $\{4\mathbf{M}, 5\mathbf{S}\}$.

Modifications must be made to the $\text{DBLU}_{\mathcal{H}}$ algorithm in order to support the (X, Z)-only operations required by Algorithm 9. This algorithm is denoted $\text{DBLU}_{\mathcal{H}}{}^*$ and is given by $\text{DBLU}_{\mathcal{H}}{}^*(P) \leftarrow (X(\tilde{P}) : X(2P) : Z(2P)) = (X_1 \cdot 64Y_1\alpha : 4Y_1D : 64Y_1\alpha)$. With a cost of $\{3\mathbf{M}, 5\mathbf{S}\}$.

A.2 Full Coordinate Recovery

The formula for the recovery of the full projective coordinates of the output point $Q = kP$, from the X-coordinates $R_0 = (X_1, Z)$ and $R_1 = (X_2, Z)$ at the end of the Montgomery ladder is given by Algorithm 33, costing $\{8\mathbf{M}, 2\mathbf{S}, 1M_a, 1M_{4b}, 8\mathbf{A}\}$.

Algorithm 33 Out of place (X, Y, Z)-recovery 1

Require: $X_1, X_2, T_D = x_D Z, T_a = aZ^2, T_b = 4bZ^3$
Output: $R = (X, Y, Z) = (X_1, X_2, Z)$

1: **function** $recoverfullcoordinates1(X_1, X_2, Z)$
2: $A = Z^2$
3: $B = ZA$
4: $C = x_D Z$
5: $D = 4y_D X_1$
6: $X_1 = DX_1A$
7: $X_2 = 2\left[(CX_1 + aA)(C + X_1) - X_2(C - X_1)^2\right] + 4bB$
8: $Z = DB$
9: **return** $R = (X, Y, Z) = (X_1, X_2, Z)$
10: **end function**

Where $D = (x_D, y_D)$ represents the invariant, input point P, of the Montgomery ladder in affine coordinates.

An alternative full coordinates recovery formula required by Algorithm 9 is given in Algorithm 34, costing $\{10\mathbf{M}, 3\mathbf{S}, 8\mathbf{A}\}$.

Algorithm 34 Out of place (X, Y, Z)-recovery 2

Require: X_1, X_2, T_D, T_a, T_b
Output: $R = (X, Y, Z) = (X_1, X_2, Z)$

1: **function** $recoverfullcoordinates2(X_1, X_2, Z)$
2: $X_1 = 4y_D x_D T_D^2 X_1$
3: $X_2 = X_D^3[T_b + 2(T_D X_1 + T_a)(X_1 + T_D) - 2X_2(X_1 - T_D)^2]$
4: $Z = 4y_D T_D^3$
5: **return** $R = (X, Y, Z) = (X_1, X_2, Z)$
6: **end function**

A.3 Point doubling and tripling with co-Z update

The initialisation step of Algorithms 8, 11, 12, 13 and 14 require a point doubling or a point tripling operation. These operations are referred to as DBLU and TPLU respectivly, and are discussed below.

Initial Point Doubling: The double of a point is computed using the DBLU operation, where

Algorithm 35 DBLU Operation

Input: $P' = (X_1, Y_1)$ for some $P = (X_1, Y_1, Z_1)$, $\alpha = A$ paramter of elliptic curve
Output: $(R', S') \leftarrow \text{DBLU}(P')$ where $R' = (X_2, Y_2)$, $S' = (X_3, Y_3)$ for some $R = 2P = (X_2, Y_2, Z_2)$ and $S = (X_3, Y_3, Z_3)$ where $Z_2 = Z_3$

1: **function** DBLU(P')
2: $B = X_1^2$
3: $E = Y_1^2$
4: $L = E^2$
5: $S = 2((X_1 + E)^2 - B - L)$
6: $M = 3B + \alpha$
7: $X_2 = M^2 - 2S$
8: $Y_2 = M(S - X_2) - 8L$
9: $X_3 = 4X_1 Y_1^2$
10: $Y_3 = Y_1^4$
11: **return** $(R' = (X_2, Y_2), S' = (X_3, Y_3))$
12: **end function**

The DBLU operation is then applied such that $(2P, \tilde{P}) \leftarrow \text{DBLU}(P)$, where $2P$ and $\tilde{P}$ have equivalent Z coordinates. This operation can therefore be used to calculate the double of the input point P and the S' output is then used to update the input point P such that all points used in the scalar multiplication algorithm have equivalent Z

coordinates. The cost of DBLU operation (doubling with update) is $\{1\mathbf{M}, 5\mathbf{S}\}$.

Initial Point Tripling: The triple of the point $P = (X_1, Y_1, 1)$ can be evaluated as $3P = P + 2P$ using co-Z arithmetic [71]. Using the DBLU operations, such that, $(2P, \tilde{P}) \leftarrow \text{DBLU}(P)$, followed by applying the ZADDU operation as $\text{ZADDU}(\tilde{P}, 2P)$, the value $3P$ results along with the input P having Z coordinate updated to be equal to that of $3P$. This operation is referred to as the TPLU operation and requires $\{6\mathbf{M}, 7\mathbf{S}\}$.

List of Abbreviations

3DES Triple DES

ADC Analog-to-Digital Converter

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

ASCII American Standard Code for Information Interchange

ASIC Application Specific Integrated Circuit

BMW Blue Midnight Wish

BRAM Block RAM

CA Certificate Authority

CBC Cipher Block Chaining

CLB Configurable Logic Block

CMAC Cipher-based Message Authentication Code

CPU Central Processing Unit

DAC Digital-to-Analog Converter

DDR Double Data Rate

DDR2 Double Data Rate

DES Data Encryption Standard

DLP Discrete Logarithm Problem

DMA Direct Memory Access

DPA Differential Power Analysis

DSA Digital Signature Algorithm

DSP Digital Signal Processing

ECB Electronic Code Book

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

ECDLP Elliptic Curve Discrete Logarithm Problem

ECDSA Elliptic Curve Digital Signature Algorithm

FF Flip Flop

FIFO First In, First Out

FIRO Fibonacci Ring Oscillator

FPGA Field Programmable Gate Array

FSL Fast Simplex Link

FSM Finite State Machine

GARO Galois Ring Oscillator

Gbit/s Gigabits per second

GLV Gallant, Lambert and Vanstone

GPP General-Purpose Processor

GPS Global Positioning System

GPU Graphics Processing Unit

HAIFA Hash Iterative Framework

HMAC Hash-based Message Authentication Code

IC Integrated Circuit

I/O	Input/Output
IP	Intellectual Property
IPsec	Internet Protocol Security
ISE	Instruction Set Extension
IV	Initialisation Vector
kB	kilobyte
kbit	Kilobit
kbit/s	Kilobits per second
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
LUT	Lookup Table
MAC	Message Authentication Code
MB	Megabyte
Mbit/s	Megabits per second
MD5	Message-Digest Algorithm 5
MHz	Megahertz
MIM	Man-in-the-middle
MMU	Memory Management Unit
ms	millisecond
NIST	National Institute of Standards and Technology
nm	nanometer
ns	nanosecond
NSP	Network Security Processor
OPSO	Overlapping-Pairs-Sparse-Occupancy

OQSO Overlapping-Quadruples-Sparse-Occupancy

PCI Peripheral Component Interconnect

PKI Public Key Infrastructure

PRNG Pseudorandom Number Generator

RA Registration Authority

RAM Random Access Memory

RC4 Rivest Cipher 4

RF Radio Frequency

RISC Reduced Instruction Set Computing

RNG Random Number Generator

ROM Read Only Memory

RSA Rivest-Shamir-Adleman

SCA Side Channel Attack

SHA Secure Hash Algorithm

SHA-0 Secure Hash Algorithm 0

SHA-1 Secure Hash Algorithm 1

SHA-2 Secure Hash Algorithm 2

SHA-3 Secure Hash Algorithm 3

SHA256 256 bit Secure Hash Algorithm

SHS Secure Hash Standard

SIPO Serial-in, parallel-out

SoC System on Chip

SPA Simple Power Analysis

SRAM Static Random Access Memory

SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TERO	Transition Effect Ring Oscillator
TLS	Transport Layer Security
TRNG	True Random Number Generator
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VPN	Virtual Private Network
WPA	Wi-Fi Protected Access
XOR	exclusive OR

References

[1] FPGA Implementations of the Round Two SHA-3 Candidates. In *The Second SHA-3 Candidate Conference*, August 2010.

[2] IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks – Specific Requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, March 2012.

[3] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard, 1998.

[4] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. In *International Conference on Field Programmable Logic and Applications (FPL 2010)*, pages 400–407. IEEE, 2010.

[5] Brian Baldwin, Raveen R. Goundar, Mark Hamilton, and William P. Marnane. Co-Z ECC Scalar Multiplications for Hardware, Software and Hardware-Software Co-Design on Embedded Systems. *Journal of Cryptographic Engineering*, 2(4):221–240, 2012. ISSN 2190-8508. doi: 10.1007/s13389-012-0042-2.

[6] Elaine Barker and John Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST Special Publication 800-90A, January 2012.

[7] Sandro Bartolini, Irina Branovic, Roberto Giorgi, and Enrico Martinelli. Effects of Instruction-Set Extensions on an Embedded Processor: A Case Study on Elliptic Curve Cryptography over $GF(2^m)$. *IEEE Transactions on Computers*, 57(5):672–685, 2008. ISSN 0018-9340. doi: 10.1109/TC.2007.70832.

[8] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrinand, Matt Robshaw, and Yannick Seurin. SHA-3 proposal: ECHO. Submission to NIST, 2008.

[9] Daniel J. Bernstein. CubeHash specification (2.B.1). Submission to NIST, 2008.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. KECCAK specifications, September 2009.

[11] Gary W. Bewick. *Fast Multiplication: Algorithms and Implementation*. PhD thesis, Electrical Engineering, Stanford University, 1994.

[12] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007.

[13] Eli Biham and Orr Dunkelman. The SHAvite-3 Hash Function. Submission to NIST (updated), 2009.

[14] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, May 2006. URL `http://www.ietf.org/rfc/rfc4492.txt`. Updated by RFC 5246.

[15] George R. Blakely. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Transactions on Computers*, 32(5):497–500, 1983. ISSN 0018-9340. doi: 10.1109/TC.1983.1676262.

[16] Andrew D. Booth. A Signed Binary Multiplication Technique. *Quarterly Journal of Mechanics and Applied Mathematics,*, 4(2):236–240, 1951. doi: 10.1093/qjmam/4.2.236.

[17] R. T. Braden. RFC 1122: Requirements for Internet hosts — Communication Layers, October 1989. URL `www.ietf.org/rfc/rfc1122.txt`.

[18] R. T. Braden. RFC 1123: Requirements for Internet Hosts — Application and Support, October 1989. URL `www.ietf.org/rfc/rfc1123.txt`.

[19] Offspark B.V. PolarSSL (1.1.7), June 2013. URL `https://polarssl.org/`.

[20] Andrew Byrne, Emanuel Popovici, and William P. Marnane. Versatile Processor for GF(p^m) Arithmetic for Use in Cryptographic Applications. In *IET Computers & Digital Techniques*, volume 2, pages 253–264, 2008.

[21] Çetin Kaya Koç, editor. *Cryptographic Engineering*. Signals & Communication. Springer, 2009.

[22] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *Advances in Cryptology (CRYPTO)*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64892-5. doi: 10.1007/BFb0055720.

[23] Gang Chen, Guoqiang Bai, and Hongyi Chen. A High-Performance Elliptic Curve Cryptographic Processor for General Curves Over GF(p) Based on a Systolic Arithmetic Unit. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(5): 412–416, 2007. ISSN 1549-7747. doi: 10.1109/TCSII.2006.889459.

[24] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, 2006.

[25] Francis Crowe, Alan Daly, and William Marnane. Optimised Montgomery Domain Inversion on FPGA. In *Proceedings of the 2005 European Conference on Circuit Theory and Design*, volume 1, pages 277–280, 2005. doi: 10.1109/ECCTD.2005.1522964.

[26] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2.

[27] Ivan Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology (CRYPTO)*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1989. ISBN 3-540-97317-6.

[28] Markus Dichtl and Jovan Dj. Golić. High-Speed True Random Number Generation with Logic Gates Only. In *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, CHES, pages 45–62. Springer, 2007. ISBN 978-3-540-74734-5.

[29] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. URL `http://www.ietf.org/rfc/rfc5246.txt`.

[30] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[31] Morris Dworkin. Recommendation for Block Cipher Modes of Operation : Methods and Techniques. NIST Special Publication 800-38A, December 2001. URL `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

[32] Morris Dworkin. Recommendation for Block Cipher Modes of Operation : The CMAC Mode for Authentication. NIST Special Publication 800-38B, May 2005. URL `http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf`.

[33] William F. Ehrsam, Carl H. W. Meyer, John L. Smith, and Walter L. Tuchman. Message Verification and Transmission Error Detection by Block Chaining. Patent US4074066 A, February 1978.

[34] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. ISSN 0018-9448. doi: 10.1109/TIT.1985.1057074.

[35] Yadollah Eslami, Ali Sheikholeslami, P. Glenn Gulak, Shoichi Masui, and Kenji Mukaida. An Area-Efficient Universal Cryptography Processor for Smart Cards. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(1):43–56, 2006. ISSN 1063-8210. doi: 10.1109/TVLSI.2005.863188.

[36] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., 2003.

[37] Kris Gaj. Hardware Interface of a Secure Hash Algorithm (SHA). Functional Specification, October 2009.

[38] Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. In *EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 518–535, 2009.

[39] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In C. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer-Verlag, 2001. ISBN 3-540-42521-7.

[40] Lubos Gaspar, Viktor Fischer, Lilian Bossuet, and Milos Drutarovský. Cryptographic Extension for Soft General-Purpose Processors with Secure Key Management. In *Proceedings of International Conference on Field Programmable Logic and*

Applications (FPL 2011), pages 500 – 505, Chania, Crete, Grèce, October 2011. doi: 10.1109/FPL.2011.99. URL `http://hal-ujm.ccsd.cnrs.fr/ujm-00664312`.

[41] Lubos Gaspar, Viktor Fischer, Lilian Bossuet, and Robert Fouquet. Secure Extension of FPGA General Purpose Processors for Symmetric Key Cryptography with Partial Reconfiguration Capabilities. *ACM Transactions on Reconfigurable Technology and Systems*, 5(3), October 2012. ISSN 1936-7406. doi: 10.1145/2362374.2362380.

[42] Santosh Ghosh, Monjur Alam, Indranil Sen Gupta, and Dipanwita Roy Chowdhury. A Robust $GF(p)$ Parallel Arithmetic Unit for Public Key Cryptography. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, (DSD 2007)*, pages 109–115, 2007. doi: 10.1109/DSD.2007.4341457.

[43] Benjamin Glas, Oliver Sander, Vitali Stuckert, Klaus D. Müller-Glaser, and Jürgen Becker. Prime Field ECDSA Signature Processing for Reconfigurable Embedded Systems. *International Journal of Reconfigurable Computing*, 2011, January 2011. ISSN 1687-7195. doi: 10.1155/2011/836460.

[44] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic Hash Function BLUE MIDNIGHT WISH. Submission to NIST (Round 2), 2009.

[45] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. Dr. Dobb's Journal, January 1996. URL `http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html`.

[46] Jovan Dj. Golić. New Methods for Digital Generation and Postprocessing of Random Data. *IEEE Transactions on Computers*, 55(10):1217–1229, October 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.164.

[47] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves. In Stefan Mangard and Franois-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems (CHES 2010)*, volume 6225 of *Lecture Notes in Computer Science*, pages 65–79. Springer-Verlag, 2010. ISBN 978-3-642-15030-2. doi: 10.1007/978-3-642-15031-9_5.

[48] Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Vanelli. Scalar Multiplication on Weierstraß Elliptic Curves from Co-Z Arithmetic. *Journal of Cryptographic Engineering*, 1(2):161–176, 2011.

[49] Conrado P. L. Gouvêa, Leonardo B. Oliveira, and Julio López. Efficient Software Implementation of Public-key Cryptography on Sensor Networks Using the MSP430X Microcontroller. *Journal of Cryptographic Engineering*, 2(1):19–29, 2012. ISSN 2190-8508. doi: 10.1007/s13389-012-0029-z.

[50] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.1.2 edition, February 2013. `http://gmplib.org/`.

[51] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[52] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60, August 2008.

[53] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 038795273X.

[54] Christopher Hargreaves and Howard Chivers. Recovery of Encryption Keys from Memory Using a Linear Scan. In *Third International Conference on Availability, Reliability and Security (ARES 08)*, pages 1369–1376, 2008. doi: 10.1109/ARES.2008.109.

[55] Ahmad Hiasat. New Memoryless, mod $(2^n \pm 1)$ Residue Multiplier. In *IEEE Electronics Letters*, volume 28, pages 314–315, January 1992.

[56] R. Housley, W. Ford, W. Polk, and D. Solo. RFC 2459: Internet X.509 public key infrastructure certificate and CRL profile, 1999. URL `www.ietf.org/rfc/rfc2459.txt`. Status: PROPOSED STANDARD.

[57] Michael Hutter, Marc Joye, and Yannick Sierra. Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. In A. Nitaj and D. Pointcheval, editors, *AFRICACRYPT*, volume 6737, pages 170–187. Springer-Verlag, 2011.

[58] Elliptic Technologies Inc. Crypto Offload Options. Technical report, May 2008. URL `http://elliptictech.com/images/stories/whitepapers/Crypto_Acceleration_Options_81030.pdf`.

[59] Takashi Isobe, Satoshi Tsutsumi, Koichiro Seto, Kenji Aoshima, and Kazutoshi Kariya. 10Gbps Implementation of TLS/SSL Accelerator on FPGA. In *18th International Workshop on Quality of Service (IWQoS 2010)*, pages 1–6, 2010. doi: 10.1109/IWQoS.2010.5542723.

[60] Marc Joye. Highly Regular right-to-left Algorithms for Scalar Multiplication. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems (CHES 2007)*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2007.

[61] Burton S. Kaliski. The Montgomery Inverse and Its Applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995. ISSN 0018-9340. doi: 10.1109/12.403725.

[62] Wolfgang Killmann and Werner Schindler. Functionality Classes and Evaluation Methodology for True (physical) Random Number Generators – Version 3.1. Bundesamt für Sicherheit in der Informationstechnik (BSI), September 2001.

[63] Neal Koblitz. Elliptic Curve Cryptosystem. *Mathematics of Computation*, 48(177): 203–209, 1987.

[64] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology (CRYPTO 96)*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[65] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 99)*, pages 388–397, London, UK, 1999. Springer-Verlag. ISBN 3-540-66347-9.

[66] Paul Kohlbrenner and Kris Gaj. An Embedded True Random Number Generator for FPGAs. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays FPGA 04*, pages 71–78, New York, NY, USA, 2004. ACM. ISBN 1-58113-829-6. doi: 10.1145/968280.968292.

[67] Gerhard Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A Practical Attack on the MIFARE Classic. In *Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS*

08), LNCS, pages 267–282. Springer-Verlag, 2008. ISBN 978-3-540-85892-8. doi: 10.1007/978-3-540-85893-5_20.

[68] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. URL `http://www.ietf.org/rfc/rfc2104.txt`. Updated by RFC 6151.

[69] Siew-Hwee Kwok, Yen-Ling Ee, Guanhan Chew, Kanghong Zheng, Khoongming Khoo, and Chik-How Tan. A Comparison of Post-Processing Techniques for Biased Random Number Generators. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, volume 6633 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 2011. ISBN 978-3-642-21039-6. doi: 10.1007/978-3-642-21040-2_12.

[70] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, New York, USA, 1986. ISBN 0-521-30706-6.

[71] Patrick Longa and Ali Miri. New Composite Operations and Precomputation for Elliptic Curve Cryptosystems Over Prime Fields. In R. Cramer, editor, *Public Key Cryptography (PKC 2008)*, volume 4939 of *Lecture Notes in Computer Science*, pages 229–247. Springer-Verlag, 2008.

[72] O.L. Macsorley. High-Speed Arithmetic in Binary Computers. In *Proceedings of the IRE*, volume 49, pages 67–91, January 1961. doi: 10.1109/JRPROC.1961.287779.

[73] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.

[74] George Marsaglia. Diehard battery of tests of randomness, 1995. URL `http://www.stat.fsu.edu/pub/diehard/`.

[75] Nikos Mavrogiannopoulos and Simon Josefsson. The GnuTLS Transport Layer Security Library (3.2.3), February 2013. URL `http://www.gnutls.org/`.

[76] Máire McLoone and John V. McCanny. A Single-Chip IPsec Cryptographic Processor. In *IEEE Workshop on Signal Processing Systems (SIPS 2002)*, pages 133–138, 2002. doi: 10.1109/SIPS.2002.1049698.

[77] Nicolas Meloni. New Point Addition Formulæ for ECC Applications. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lecture*

Notes in Computer Science, pages 189–201. Springer-Verlag, 2007. ISBN 978-3-540-73073-6. doi: 10.1007/978-3-540-73074-3_15.

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

[79] Ralph C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO 89*, volume 435 of *LNCS*, pages 428–446. Springer-Verlag, 1989. ISBN 0-387-97317-6.

[80] Thomas S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965 of *Lecture Notes in Computer Science*, pages 71–77. Springer-Verlag, 2000.

[81] Victor S. Miller. Use of Elliptic Curves in Cryptography. In *Advances in cryptology – CRYPTO 85*, LNCS, pages 417–426. Springer-Verlag, 1986. ISBN 0-387-16463-4.

[82] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[83] Peter L. Montgomery. Speeding Up the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[84] *FIPS PUB 180 Secure Hash Standard*. National Institute of Standards and Technology, May 1993.

[85] *FIPS PUB 180-1 Secure Hash Standard*. National Institute of Standards and Technology, 1995.

[86] *Security Requirements for Cryptographic Modules (FIPS–140-2)*. National Institute of Standards and Technology, 2002.

[87] *Secure Hash Standard (FIPS 180-2)*. National Institute of Standards and Technology, August 2002.

[88] *Secure Hash Standard (SHS), (FIPS 180-4)*. National Institute of Standards and Technology, 2012.

[89] *Digital Signature Standard (DSS), (FIPS 186-4)*. National Institute of Standards and Technology (NIST), July 2013.

[90] NIST. *Data Encryption Standard (DES) (FIPS–46-3).* National Institute of Standards and Technology, 1999.

[91] NIST. *Advanced Encryption Standard (AES) (FIPS–197).* National Institute of Standards and Technology, 2001.

[92] NIST. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 72(212):66212–66220, November 2007.

[93] Yun Niu, Liji Wu, Li Wang, Xiangmin Zhang, and Jun Xu. A Configurable IPSec Processor for High Performance In-Line Security Network Processor. In *7th International Conference on Computational Intelligence and Security (CIS 2011)*, pages 674–678, 2011. doi: 10.1109/CIS.2011.154.

[94] Albert Eugene Novark. *Hardening Software Against Memory Errors and Attacks*. PhD thesis, University of Massachusetts - Amherst, 2011. URL `http://scholarworks.umass.edu/open_access_dissertations/346`.

[95] European Network of Excellence in Cryptology II. ECRYPT II Yearly Report on Algorithms and Keysizes, September 2012.

[96] Network Working Group of the IETF. RFC4301 - Security Architecture for the Internet Protocol, December 2005.

[97] A. Karatsuba Yu. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

[98] Sean O'Melia and Adam J. Elbirt. Enhancing the Performance of Symmetric-Key Cryptography via Instruction Set Extensions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(11):1505–1518, 2010. ISSN 1063-8210. doi: 10.1109/TVLSI.2009.2025171.

[99] Gerardo Orlando and Christof Paar. A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware. In *Cryptographic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *LNCS*, pages 348–363. Springer-Verlag, May 2001. doi: 10.1007/3-540-44709-1_29.

[100] Siddika Berna Örs, Lejla Batina, and Bart Preneel. Hardware Implementation of Elliptic Curve Processor over GF(p). In *International Journal of Embedded Systems*, pages 433–443, 2003.

[101] Özgül Küçük. The hash function Hamsi. Submission to NIST, 2009.

[102] Jon Postel. RFC 793: Transmission Control Protocol, September 1981. URL www.ietf.org/rfc/rfc793.txt.

[103] The OpenSSL Project. Open Source Toolkit for SSL/TLS – OpenSSL (1.0.1e), February 2013. URL http://www.openssl.org/.

[104] Matthieu Rivain. Fast and Regular Algorithms for Scalar Multiplication Over Elliptic Curves. Cryptology ePrint Archive, Report 2011/338, 2011. http://eprint.iacr.org/.

[105] Ronald L. Rivest. The MD5 Message-Digest Algorithm (RFC 1321). http://www.ietf.org/rfc/rfc1321.txt?number=1321.

[106] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[107] Francisco Rodríguez-Henríquez, N. A. Saqib, Arturo Díaz Pérez, and Çetin Kaya Koç. *Cryptographic Algorithms on Reconfigurable Hardware*. Signals and Communication Tecnology. Springer, 2007.

[108] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22, April 2010. URL http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf. Revised by Lawrence E Bassham III.

[109] Renaud Santoro, Olivier Sentieys, and Sebastien Roy. On-the-Fly Evaluation of FPGA-Based True Random Number Generator. In *Proceedings of the 2009 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 09)*, pages 55–60. IEEE Computer Society, 2009. ISBN 978-0-7695-3684-2. doi: 10.1109/ISVLSI.2009.33.

[110] Renaud Santoro, Arnaud Tisserand, Olivier Sentieys, and Sébastien Roy. Arithmetic Operators for On-the-Fly Evaluation of TRNGs. In *Advanced Signal Processing Algorithms, Architectures and Implementations XVIII*, volume 7444, pages 1–12. SPIE, August 2009. doi: 10.1117/12.826336.

[111] Akashi Satoh and Kohji Takano. A Scalable Dual-Field Elliptic Curve Cryptographic Processor. *IEEE Transactions on Computers*, 52(4):449–460, 2003. ISSN 0018-9340. doi: 10.1109/TC.2003.1190586.

[112] Dries Schellekens, Bart Preneel, and Ingrid Verbauwhede. FPGA Vendor Agnostic True Random Number Generator. In *International Conference on Field Programmable Logic and Applications (FPL 06)*, pages 1–6, 2006. doi: 10.1109/FPL.2006.311206.

[113] Werner Schindler and Wolfgang Killmann. Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES, pages 431–449. Springer-Verlag, 2003. ISBN 3-540-00409-2.

[114] Bruce Schneier. *Applied Cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, USA, 1995. ISBN 0-471-11709-9.

[115] Azeddien Sllame and Vladimir Drabek. An Efficient List-Based Scheduling Algorithm for High-Level Synthesis. In *Euromicro Symposium on Digital System Design (EUROMICRO 02)*, pages 316–323, 2002. doi: 10.1109/DSD.2002.1115384.

[116] Berk Sunar, William J. Martin, and Douglas R. Stinson. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *IEEE Transactions on Computers*, 58(1):109–119, 2007. ISSN 0018-9340. doi: 10.1109/TC.2007.250627.

[117] David Barrie Thomas, Lee Howes, and Wayne Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 09)*, pages 63–72. ACM, 2009. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508139.

[118] Stefan Tillich and Johann Groschdl. Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography. In Osvaldo Gervasi, MarinaL. Gavrilova, Vipin Kumar, Antonio Lagan, HeowPueh Lee, Youngsong Mun, David Taniar, and ChihJengKenneth Tan, editors, *Computational Science and Its Applications (ICCSA 05)*, volume 3481 of *Lecture Notes in Computer Science*, pages 665–675. Springer-Verlag, 2005. ISBN 978-3-540-25861-2. doi: 10.1007/11424826_70.

[119] K. H. Tsoi, K. H. Leung, and P. H. W. Leong. Compact FPGA-Based True and Pseudo Random Number Generators. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 03)*, pages 51–61. IEEE Computer Society, 2003. ISBN 0-7695-1979-2.

[120] Boyan Valtchanov, Viktor. Fischer, Alain Aubert, and Florent Bernard. Characterization of Randomness Sources in Ring Oscillator-Based True Random Number Generators in FPGAs. In *IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 10)*, pages 48–53, 2010. doi: 10.1109/DDECS.2010.5491819.

[121] Michal Varchola and Miloš Drutarovský. New High Entropy Element for FPGA Based True Random Number Generators. In Stefan Mangard and Franois-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems (CHES 2010)*, volume 6225 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, 2010. ISBN 978-3-642-15030-2. doi: 10.1007/978-3-642-15031-9_24.

[122] Ihor Vasyltsov, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. Fast Digital TRNG Based on Metastable Ring Oscillator. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems (CHES 2008)*, volume 5154 of *Lecture Notes in Computer Science*, pages 164–180. Springer-Verlag, 2008. ISBN 978-3-540-85052-6. doi: 10.1007/978-3-540-85053-3_11.

[123] Alexandre Venelli and François Dassance. Faster Side-Channel Resistant Elliptic Curve Scalar Multiplication. *Contemporary Mathematics*, 521:29–40, 2010.

[124] C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronic Letters*, 35(21):1831–1832, October 1999.

[125] Haixin Wang, Guoqiang Bai, and Hongyi Chen. Zodiac: System Architecture Implementation for a High-Performance Network Security Processor. In *International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2008)*, pages 91–96, 2008. doi: 10.1109/ASAP.2008.4580160.

[126] Haixin Wang, Guoqiang Bai, and Hongyi Chen. A Gbps IPSec SSL Security Processor Design and Implementation in an FPGA Prototyping Platform. *Journal of Signal Processing Systems*, 58(3):311–324, March 2010. ISSN 1939-8018. doi: 10.1007/s11265-009-0371-2.

[127] Haixin Wang, Guoqiang Bai, and Hongyi Chen. Design and Implementation of a High Performance Network Security Processor. *International Journal of Electronics*, 97(3):309–325, 2010. doi: 10.1080/00207210903289383.

[128] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer-Verlag, August 2005.

[129] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11e)*. Xilinx, ds449 edition, October 2011. URL `http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20/v2_11_e/fsl_v20.pdf`.

[130] Xilinx. *MicroBlaze Processor Reference Guide*. Xilinx, December 2012. URL `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf`.

[131] Inc. Xilinx. Virtex-5 Family Overview. Technical Report DS100 (v5.0), February 2009. URL `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`.

[132] Inc. Xilinx. Virtex-5 FPGA User Guide. Technical Report UG190 (v5.3), May 2010. URL `http://www.xilinx.com/support/documentation/user_guides/ug190.pdf`.

[133] Inc. Xilinx. Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC User Guide. Technical Report UG194 (v1.10), February 2011. URL `http://www.xilinx.com/support/documentation/user_guides/ug194.pdf`.

[134] Inc. Xilinx. Virtex-5 FPGA XtremeDSP Design Considerations. Technical Report UG193 (v3.5), January 2012. URL `http://www.xilinx.com/support/documentation/user_guides/ug193.pdf`.

[135] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), January 2006. URL `http://www.ietf.org/rfc/rfc4254.txt`.

[136] Xinmiao Zhang and K.K. Parhi. High-speed VLSI Architectures for the AES Algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(9):957–967, 2004. ISSN 1063-8210. doi: 10.1109/TVLSI.2004.832943.