Hardware Design of Cryptographic Accelerators

by

Brian Baldwin

Thesis submitted for the degree of

PhD

from the

Department of Electrical Engineering

National University of Ireland

University College, Cork, Ireland

May 7, 2013

Supervisor: Dr. William P. Marnane

“What I cannot create, I do not understand”

- Richard Feynman; on his blackboard at time of death in 1988.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

With the rapid growth of the Internet and digital communications, the volume of sensitive electronic transactions being transferred and stored over and on insecure media has increased dramatically in recent years. The growing demand for cryptographic systems to secure this data, across a multitude of platforms, ranging from large servers to small mobile devices and smart cards, has necessitated research into low cost, flexible and secure solutions. As constraints on architectures such as area, speed and power become key factors in choosing a cryptosystem, methods for speeding up the development and evaluation process are necessary.

This thesis investigates flexible hardware architectures for the main components of a cryptographic system. Dedicated hardware accelerators can provide significant performance improvements when compared to implementations on general purpose processors. Each of the designs proposed are analysed in terms of speed, area, power, energy and efficiency. Field Programmable Gate Arrays (FPGAs) are chosen as the development platform due to their fast development time and reconfigurable nature.

Firstly, a reconfigurable architecture for performing elliptic curve point scalar multiplication on an FPGA is presented. Elliptic curve cryptography is one such method to secure data, offering similar security levels to traditional systems, such as RSA, but with smaller key sizes, translating into lower memory and bandwidth requirements. The architecture is implemented using different underlying algorithms and coordinates for dedicated Double-and-Add algorithms, twisted Edwards algorithms and SPA secure algorithms, and its power consumption and energy on an FPGA

measured. Hardware implementation results for these new algorithms are compared against their software counterparts and the best choices for minimum area-time and area-energy circuits are then identified and examined for larger key and field sizes.

Secondly, implementation methods for another component of a cryptographic system, namely hash functions, developed in the recently concluded SHA-3 hash competition are presented. Various designs from the three rounds of the NIST run competition are implemented on FPGA along with an interface to allow fair comparison of the different hash functions when operating in a standardised and constrained environment. Different methods of implementation for the designs and their subsequent performance is examined in terms of throughput, area and energy costs using various constraint metrics.

Comparing many different implementation methods and algorithms is nontrivial. Another aim of this thesis is the development of generic interfaces used both to reduce implementation and test time and also to enable fair baseline comparisons of different algorithms when operating in a standardised and constrained environment.

Finally, a hardware-software co-design cryptographic architecture is presented. This architecture is capable of supporting multiple types of cryptographic algorithms and is described through an application for performing public key cryptography, namely the Elliptic Curve Digital Signature Algorithm (ECDSA). This architecture makes use of the elliptic curve architecture and the hash functions described previously. These components, along with a random number generator, provide hardware acceleration for a Microblaze based cryptographic system. The trade-off in terms of performance for flexibility is discussed using dedicated software, and hardware-software co-design implementations of the elliptic curve point scalar multiplication block. Results are then presented in terms of the overall cryptographic system.

Associated Publications

- B. Baldwin, R. Moloney, A. Byrne, G. McGuire and W. P. Marnane, **A Hardware Analysis of Twisted Edwards Curves for an Elliptic Curve Cryptosystem**, Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications, -ARC '09, vol. 5453 of LNCS, pp. 355-361, 16th-18th March 2009. Cryptology ePrint Archive, Report 2009/001, 2009.

- B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan and W. P. Marnane, **FPGA Implementations of SHA-3 Candidates:CubeHash, Groestl, Lane, Shabal and Spectral Hash**, Euromicro Symposium on Digital Systems Design -DSD '09, pp. 783-790, 27th-29th August 2009.

- D. V. Bailey, B. Baldwin, L. Batina, D. J. Bernstein, G. Van Damme, G. De Meulenaer, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, N. Mentens, C. Paar, F. Regazzoni, P. Schwabe and L. Uhsadel, **The Certicom Challenges ECC2-X**, Workshop on Special Purpose Hardware for Attacking Cryptographic Systems, -SHARCS '09, September 2009.

- B. Baldwin, W. P. Marnane and R. Granger, **Reconfigurable Hardware Implementation of Arithmetic Modulo Minimal Redundancy Cyclotomic Primes for ECC**, Reconfigurable Computing, FPGAs, International Conference on, -Reconfig '09, pp. 255-260, 9th-11th December 2009.

- B. Baldwin and W. P. Marnane, **An FPGA Technologies Area Examination of the SHA-3 Hash Candidate Implementations**, Cryptology ePrint Archive, Report 2009/603, 2009.

- B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill and W. P. Marnane, **A Hardware Wrapper for the SHA-3 Hash Algorithms**, 21st Irish Signals and Systems Conference, ISSC '10, pp. 1-6, 23rd-24th June 2010. Cryptology ePrint Archive, Report 2010/124, 2010.

- B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill and W. P. Marnane, **FPGA Implementations of the Round Two SHA-3 Candidates**, The Second SHA-3 Candidate Conference, 23rd-24th August 2010.

- B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill and W. P. Marnane, **FPGA Implementations of the Round Two SHA-3 Candidates**, 20th International Conference on Field Programmable Logic and Applications -FPL '10, pp. 400-407, 31 August - 2 September 2010.

- B. Baldwin and W. P. Marnane, **Yet Another SHA-3 Round 3 FPGA Results Paper**, Cryptology ePrint Archive, Report Report 2012/180, 2012.

- B. Baldwin, R.R. Goundar, M. Hamilton and W. P. Marnane, **Co-Z ECC scalar multiplications for hardware, software and hardware-software co-design on embedded systems**, vol.2, no.4, pp. 221-240, Journal of Cryptographic Engineering, 2012.

Acronyms

AEP	Area-Energy Product
AES	Advanced Encryption Standard
AHS	Advanced Hash Standard
ALM	Adaptive Logic Modules
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
API	Application Program Interfaces
ASIC	Application Specific Integrated Circuit
ATHENa	Automated Tool for Hardware EvaluatioN
ATP	Area-Time Product
BRAM	BlockRAM
BSB	Base System Builder
CBC	Cipher-Block Chaining
CBC-MAC	Cipher Block Chaining Message Authentication Code
CLB	Configurable Logic Blocks
CPU	Central Processing Unit
DDR	Double Data Rate
DES	Data Encryption Standard
DL	Discrete Logarithm

DLP	Discrete Logarithm Problem
DPA	Differential Power Analysis
DRBG	Deterministic Random Bit Generators
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
EEA	Extended Euclidean Algorithm
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECP	Elliptic Curve Processor
ECRYPT	European Network of Excellence for Cryptology
EDK	Embedded Design Kit
EOM	End Of Message
FIFO	First In, First Out
FIPS	Federal Information Processing Standard
FPGA	Field-Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
GC&CS	Government Code and Cypher School
GCHQ	Government Communications Headquarters
HAIFA	Hash Iterative Framework
HMAC	Keyed-Hash Message Authentication Code
I/O	Input/Output
IC	Integrated Circuit
IP	Internet Protocol
IEC	International Electrotechnical Commission

IEEE	Institute of Electrical and Electronics Engineers
IF	Integer Factorization
IFP	Integer Factorisation Problem
IP	Intellectual Property
ISO	International Organization for Standardization
IV	Initialisation Vector
LAB	Logic Array Blocks
LBS	List-Based Scheduling
LE	Logic Elements
LMB	Local Memory Bus
LSB	Least Significant Bit
LUT	Look-Up Tables
MAC	Message Authentication Codes
MDS	Maximum Distance Separable
NIST	National Institute of Standards and Technology
NLFSR	Non-Linear Feedback Shift Register
NP-hard	Non-Deterministic Polynomial-Time Hard
NRBG	Non-Deterministic Random Bit Generators
OPB	On-chip Peripheral Bus
PA	Point Addition
PD	Point Doubling
PGP	Pretty Good Privacy
PKC	Public Key Cryptography
PowerPC	Performance Optimization With Enhanced RISC Performance Computing
P-P-R	Post-Place-and-Route
PPR	Project Peripheral Repository
PRNG	Pseudo Random Number Generator

RAM	Random-Access Memory
RBG	random bit generators
RISC	Reduced Instruction Set Computing
RNG	Random Number Generator
ROM	Read-Only Memory
SASEBO	Side-channel Attack Standard Evaluation Board
SCA	Side Channel Attack
SDK	Software Development Kit
SET	Secure Electronic Transaction
SHA	Secure Hash Algorithm
SIS	Signals Intelligence Section
S/MIME	Secure/Multipurpose Internet Mail Extensions
SOP	Sum Of Products
SOC	System On a Chip
SPA	Simple Power Analysis
SPN	Substitution-Permutation Network
SUPERCOP	System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives
TPA	Throughput Per Unit Area
UBI	Unique Block Iteration
VHDL	VHSIC hardware description language
VHSIC	Very-High-Speed Integrated Circuits
XBX	eXternal Benchmarking eXtension
XCL	Xilinx Cache Link
XML	Extensible Markup Language
XPS	Xilinx Platform Studio

Declaration

I hereby state that all of the work undertaken in his thesis is original in content and was carried out by the author. Work carried out by others has been duely acknowledged in the thesis. The work presented has not been accepted in any previous application for a degree.

Signed: _____

Date: _____.

Acknowledgements

First and foremost, this thesis is dedicated to Fiona and Cormac. Thanks for brightening up my days, supporting me throughout, and generally being the most wonderful people I know.

Submission of this Ph.D. thesis would not have been possible without the help and support of a number of people. Both those who helped and supported me to get work and research done and those who provided an occasional sanity check and made the journey both worthwhile and fun (quite often the same people). I cannot thank you all enough for making it an experience worth having.

I would like to thank Liam Marnane for giving me the opportunity to study for a Ph.D. under his supervision, Bob Savage in EMC for encouraging me to take the first step, Gary McGuire and Elva OSullivan and the rest of the people at the Claude Shannon Institute for Discrete Mathematics, Coding, Cryptography and Information Security and Science Foundation Ireland for providing the means and support to perform the work contained within.

Thanks also to the rest of the UCC Cryptography research group, Neil, Andy, Mark, Rob, Maurice, Weibo, who were always willing to take time out to sit down and help work stuff out, proof read, or offer comments and propose solutions.

A lot of thanks also to the rest of the postgrads in the Department of Electrical and Electronic Engineering at UCC for making my time here both enjoyable and rewarding. There are too many to name (but here goes; Dan, Paul, Sean, Dave, Steve, Kieran, James, Eoin, Niamh, Orla, Aiden, Niall, Donagh, Declan, Simon, Brendan, Megan, Andrey, Philip, Sunil and Rehan amongst others). I am truly grateful for all of the good, bad and indifferent times we all shared. Special thanks to Jason Hannon for all the advice, LaTeX or otherwise, down through the years.

Thanks to my fellow postgrads (and postdocs) at CSI, for making me feel welcome and looking after me at all the meetings and events we shared. Richard, John, Cathy, Naomi, Geoff, Danny, Alison, Ezekiel, Manuel, Fernando, Rob, Jens and of course Luis to name but a few.

Thanks also to all the faculty, technical and administrative staff of UCC for their assistance over the years especially Rita, Ralph, Geraldine, Niamh and Mary.

Thanks to the EMC team Clariion guys, Ken, Joe, Rob, Fiachra, Paul Foley, Paul, Noel, Des et al. for continuing to include me in their social events long after I left the place.

Thanks to Karl, Dara, Adam and Tom, for being general sounding boards and proof readers. Always willing to read over stuff, listen to rants, offer suggestions and go for occasional pints whenever the need arose.

Finally, thanks to my parents and sisters for their unwavering support and encouragement down through the years.

1

Introduction

1.1 Motivation

With the rapid growth of the Internet and digital communications, the need for protecting files and other information stored on, and transmitted between computers has become of vital importance. Part of the requirement for trusted computing are cryptographic algorithms, used as network security measures to protect data during transmission. The successful deployment of these electronic systems for cloud computing and virtual private networks (VPN), data communications, mobile commerce, internet banking and public-key infrastructures (PKI) amongst others, is dependent on the effectiveness of the underlying security.

Cryptography, the study and practice of secret or secure data communication in the presence of adversarial third parties, and cryptanalysis, the study of breaking the same secret or secure messages, as subjects have been studied for many centuries. While initially being the preserve of governments, advances in technology and communication in the latter half of the last century has resulted in cryptography becoming a widespread tool in the public domain, through the use of third party security management and corporate data access and storage. Systems such as personal computers, smart cards, wireless mobile phone and smartphone technology, along with network appliances including web servers, firewalls, routers and gateways all depend on cryptography in their day to day general use.

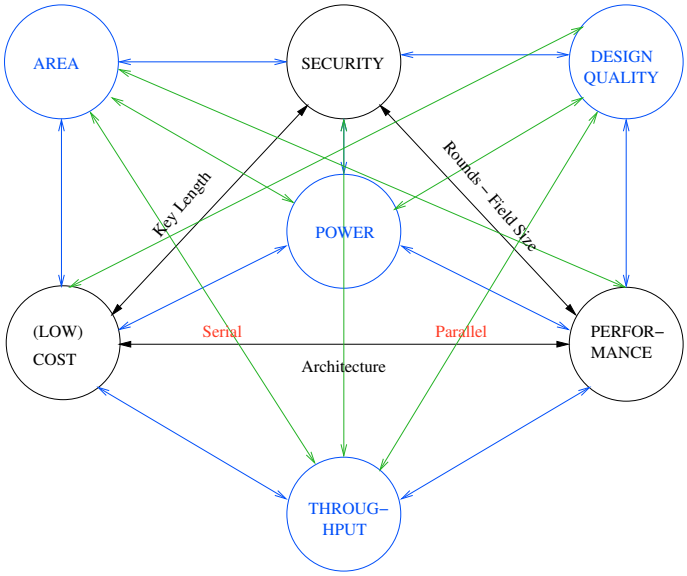


Figure 1.1: Cost-Performance-Security Tradeoff

This rapid growth in communication technology also makes information more easily accessed and available to abuse. Perceived private information can be exposed by eavesdroppers. Data can be fraudently altered for the benefit of adversaries. Systems can be hacked to perform operations other than those intended by their owners. As such, it is vital that these communication systems

be made secure through the use of cryptography.

In addition to these security concerns is the use of these security algorithms in resource constrained environments such as smart cards. Low power, more functionality, high speed in real-time systems and longer battery life all add additional requirements which need to be considered in current and future secure communications applications. Figure 1.1 describes the Cost-Performance-Security Tradeoff modified from [1]. Any design involves tradeoff and the metrics are all interrelated. Therefore it is usually the case that enhancing one metric will comes at a cost of another metric. It is very difficult to optimize all design goals at the same time. For example, using a pipelining technique can be used to provide high throughput and inbuilt security against side-channel attacks but at a cost of area and complexity.

1.2 Thesis Aims

The primary aim of this thesis is to analyse and develop flexible hardware which can be used as components of a cryptographic system, thereby allowing a basic implementation of information security applications such as such as digital signatures, message authentication codes (MACs), and other forms of authentication, such as key exchange, the purpose of which include, confidentiality, authentication, integrity and non-repudiation. Figure 1.2 shows a generic overview of a model for an elliptic curve based cryptosystem. High performance in the top three layers is directly dependent on the efficiency in computation of the bottom three layers. This thesis examines and implements each of these layers, along with the primitives defined in layer four and attempts to find the best algorithms and implementations for efficient use in the top two layers.

Cryptographic algorithms can be implemented in hardware or software. There have been recent increases in data rate speeds, along with an ongoing increase both in the size and the mathematical complexity of security protocols, as cryptanalysis becomes more feasible due to the amount and availability of better and cheaper hardware following Moore's law [2]. While it is relatively easy to implement cryptographic algorithms in software, difficulties arise. Some systems require

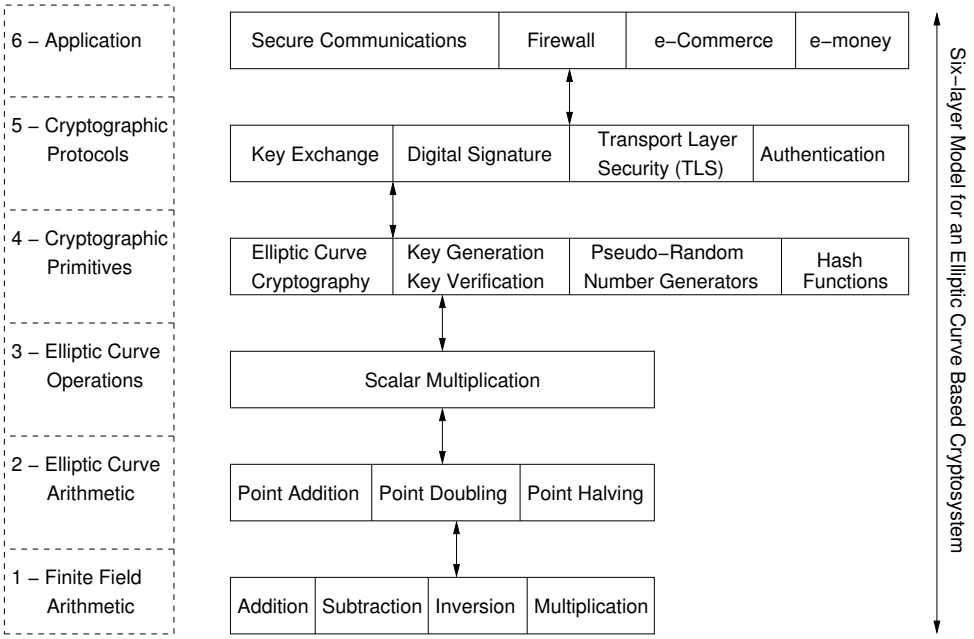


Figure 1.2: Hierarchical Model for Elliptic Curve Based Cryptography

a large number of transactions in a short space of time, i.e. network routers, firewalls and cloud computing, while other devices require resource constrained systems, i.e. smartphones and smartcards.

Dedicated hardware to perform cryptographic operations can help to alleviate both of these needs at the same time. It can achieve high system performance while still maintaining good security at low power. A dedicated circuit will have a lower power consumption than a general purpose processor. In this way, mathematically complex and time consuming calculations can be offloaded to dedicated hardware blocks thus reducing processor usage and potential bottlenecks. Both of these factors combine to offer potential savings that could offset the initial cost of the design of dedicated hardware.

One particular disadvantage of dedicated hardware is in its inherent rigid structure. Custom hardware tends to perform one particular function. In modern systems, cryptographic advances due to faster [3] or more secure algorithms [4,5], or inherent weaknesses found in current systems [6–8], can result in costly (both in development time and in deployment and roll-out) systems being

superceded by the next technology advance after a shelf-life of only a few years. Therefore, flexibility, while being less efficient than dedicated hardware, is also a major requirement for any commercially viable system to allow both new algorithms and protocols to be easily implemented, while also maintaining support for existing and even legacy systems.

One such method to allow this flexibility is through the use of Field Programmable Gate Arrays (FPGA). FPGAs are an attractive choice for implementing cryptographic algorithms as they allow rapid prototyping of designs, and can be re-programmed in-place when adopting security protocol upgrades. Another aim of this thesis is to implement cryptographic designs, (i.e. layer 2 and 3 of Figure 1.2), which can be quickly and easily configured to support various area or speed implementations dependent on the availability of resources and the particular metrics required. This can be achieved through the reconfigurable logic of FPGAs.

Power analysis attack can provide detailed information by observing the power consumption of a hardware device. These attacks are roughly categorized into simple power analysis (SPA) and differential power analysis (DPA). Algorithmically secure countermeasures such as dummy arithmetic operations, unified formulæ and regular structures can be used to reduce the information available to an attacker. Some of these security methods used in elliptic curve cryptography are examined in this thesis.

While security is the most important criterion for a new cryptographic function, performance is arguably almost as important. Analysis of these cryptographic functions and algorithms in the literature tend to be examined and reported on almost exclusively in software. The metrics and computation cost used for software are often quite different to those used in hardware. A third aim of this thesis is to examine various new algorithms used in elliptic curve cryptography and hash functions for use in hardware, (i.e. layer 4 from Figure 1.2). An analysis of these designs for throughput, area, and energy is performed for differing key sizes (the size of the key is directly related to the security of the system; the larger the key size, the greater the security level) and design variants; where different versions of the same base design present higher security or higher speed options. Speed increases due to parallelisation is also examined.

Comparing many different implementation methods and algorithms is nontrivial. Another aim of this thesis is the development of generic interfaces used both to reduce implementation and test time and also to enable fair baseline comparisons of different algorithms when operating in a standardised and constrained environment.

Another benefit to using FPGAs is that it allows the use of microprocessors, embedded in the FPGA architecture, also known as System on Chip (SOC). These SOCs include additional general purpose registers, instruction sets, interfaces to the external world and shift units, along with more advanced features and peripherals to allow a further tailoring of the design to the required specifications in a hardware-software co-design environment. This method allows the computationally intensive components, i.e. the cryptographic algorithms, to be implemented in hardware, while the non computationally intensive components, i.e. control, data processing, padding etc., which are better suited to software are performed by the systems host processor, thereby allowing a full cryptographic system on a single chip. A fourth aim of this thesis is to present a cryptographic processor which uses cryptographic IP cores, comprised of the previously defined cryptographic algorithms, implemented on a Xilinx Virtex-5 FPGA. An example application, namely the Elliptic Curve Digital Signature Algorithm (ECDSA), is presented to demonstrate the functionality of the system, based on layer 5 of Figure 1.2.

1.3 Thesis Outline

This section outlines the structure of the remainder of the thesis. Chapter 2 provides a short introduction to cryptography, and presents a brief history of the subject. The mathematical background necessary for the rest of the thesis is introduced here. More specifically, the mathematics behind *public key cryptography* (PKC), part of the main focus of this work, is described here. This includes *groups, rings, fields, finite fields* and *elliptic curves*. Following this, some example protocols are presented, used both to differentiate between public key cryptography and other types of symmetric key cryptography, and also to show how these public key cryptography schemes are

used. Finally, a brief description of *cryptographic key sizes* is presented and the different security levels provided by them is discussed. The platform tools used to form the underlying technology used in this work are also presented here. *Field Programmable Gate Array* (FPGA) integrated circuits are described along with the technology which underpins them. The *Microblaze* soft-core virtual microprocessor is also described here along with the *Sasebo GII* cryptographic evaluation board and the Xilinx *XUPV5-LX110T* Evaluation Platform. Constraints associated with hardware implementations are examined and the concept of side channel attacks is introduced along with various countermeasures relevant to this work. An outline of trade-offs between area, speed, power and energy is presented. Some *performance metrics* used in the thesis are also presented here.

Chapter 3 examines the performance of various elliptic curve algorithms on a reconfigurable elliptic curve processor. It begins with a description of the classic *Double-and-Add* method for point scalar multiplication, and examines different coordinate systems which can be used. Next a special form of curve is described, the *twisted Edwards* and *extended twisted Edwards*. These curves can perform point scalar multiplication faster than standard curves and can be made unified for additional security. The processor used for the various algorithms and implementations is introduced and the methods to design it are described. The underlying modular arithmetic is presented and different configurations of the processor are investigated. A method for measuring the power dissipation on this elliptic curve processor is described and timing, area and power results are presented. A brief look at simple power analysis (SPA) is next performed and it is shown how the algorithms examined so far are susceptible to this type of simple side channel attack. Following on from this, various SPA secure methods are then examined, namely *dummy arithmetic instructions*, *unified doubling and addition* and *regular algorithms*. The algorithmic cost and area, energy and timing results of these SPA secure algorithms is examined and some further observations are made. The best performing algorithms are then selected for further analysis with larger key and field sizes based on NIST specifications. Additional power, energy area and timing results were acquired and comparisons between the differing security levels were made.

Chapter 4 examines an implementation method for the *SHA-3* hash functions. It begins with a

brief overview of *cryptographic hash functions* and the timeline of the recently concluded SHA-3 competition. It then examines the different types of hash functions accepted for the competition and examines some implementation methods and area-speed trade offs. Some example implementations are also presented. Next, an overview of the algorithm and a description of the implementation methodology for the finalist designs is explored. A standard wrapper, used to interface the designs is also presented here. Results for implementations of the round two of the competitions designs are presented and metrics used for selecting one particular hash function over another are discussed. The updated round three variants of the competition are next presented along with an examination of any changes to the metrics due to these updates. Finally, a comparison against the current state of the art is presented and conclusions are made.

Chapter 5 presents a *cryptographic architecture* that is capable of supporting multiple types of cryptographic algorithms and architectures. This architecture provides support for the underlying field operations performed by Elliptic Curve Cryptography (ECC) along with the curve parameters, algorithm used, number of arithmetic units and key size, thus enabling flexibility in the selection of both the underlying algorithm, the security level and the area-throughput requirements. Additionally, the cryptographic architecture can also support all of the SHA-3 algorithms and their variants. These cryptographic blocks can then be used along with ancillary components as building blocks to build bigger cryptographic protocols. Further examination is performed regarding coordinate conversion between domains and coordinate systems. *Software only* and *hardware-software co-design* implementations are presented and compared against the dedicated hardware implementation. An overview of a commonly used application for performing public key cryptography, namely the *Elliptic Curve Digital Signature Algorithm* (ECDSA), is presented. Another component of ECDSA, the random key generator, is presented and an implementation method using a pseudo random number generator, *Fortuna*, is described. Results for the ECDSA are presented and compared against the current state of the art.

Chapter 6 concludes the thesis. The main contributions of the thesis are summarised and some suggestions for future research directions are outlined.

2

Background

2.1 Introduction

This chapter provides a background to the work presented in this thesis. It is subdivided into four main sections. In Section 2.2, a short introduction to cryptography is presented. Subsequent sections explore the mathematical background and the hardware used for the work. Finally, a short analysis of security and constraints, along with a presentation of some measurement metrics to allow comparison between designs is performed.

An elliptic curve system consists of four main layers, *the finite field layer*, *the elliptic curve point operation layer*, *the scalar multiplication layer* and the *protocol layer*. The mathematical back-

ground of the first layer, *the finite field layer* is presented in Section 2.3.

Section 2.4 presents an overview of elliptic curve cryptography, and provides a mathematical background for the second layer, *the elliptic curve point operation layer*, including the group law for defining points over an elliptic curve. Point representation and an introduction to coordinate systems are also presented here.

Section 2.5 describes the final two layers of an elliptic curve system, *the scalar multiplication layer* and the *protocol layer*. *Symmetric-Key* and *Public-Key* cryptography is described. The *Integer Factorisation Problem* (IFP), the *Discrete Logarithm problem* (DLP) and the *Elliptic Curve Discrete Logarithm problem* (ECDLP) are defined. A short analysis of cryptographic key sizes is also presented. Digital Signature protocols are discussed, making use of the topics presented in previous sections of this chapter. Finally, the security levels of the different types of cryptographic algorithms are compared both in terms of broad design, security level and cryptographic keysizes.

Subsequent sections present an overview of the hardware. Section 2.6 provides an overview of the platform tools used, namely FPGAs. The platform on which the cryptosystems are evaluated and the tools used to do so are introduced. Section 2.7 describes the *Microblaze* soft-core processor and the Xilinx Embedded Design Kit (EDK) used to generate it. The *Sasebo GII* cryptographic evaluation board and the *XUPV5-LX110T* Evaluation Platform are presented in Section 2.8, upon which, most implementation testing was done. Section 2.9 examines the constraints associated with the hardware implementations. Side channel analysis of cryptographic systems and its countermeasures are examined. A brief description of the trade-off between area, speed, power and energy is also presented. In Section 2.10, some *metrics* are developed as a method of comparing the different designs presented in this thesis. Conclusions are presented in Section 2.11.

2.2 Introduction to Cryptography

Cryptography provides the means of securing the communication between two parties across an insecure channel. While there is evidence of the ancient Egyptians and Babylonians scribes delib-

erately transforming ancient writings as a form of showing off their knowledge, the Spartans [9] were the first to make use of cryptography for military purposes through the use of the *skytale*, a staff of wood of fixed dimensions (the key), around which a strip of leather (the ciphertext), is wrapped, and the message is written. Only with an identical staff can the message be decoded. The Greeks wrote the earliest text on secret communication (and indeed the word cryptography itself is an amalgamation the Greek words, secret and writing), *On the Defense of Fortified Places* by Aeneas the Tactician [10]. This text contained details on replacing vowels of plaintext by specific dots and forms of stenography (hidden writing) through the use of punching holes in a document above the letters which spelled out the secret code. As it is unclear whether these systems were ever actually used, the first attested use of a substitution cipher comes from the Romans, namely Julius Caesar and his *Caesar cipher* [11] in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. This is known as a *monoalphabetic substitution* where the same key is always bound to the same plaintext. Cryptology was also extensively used in the Arabic world (the word cipher for example, is arabic), and the fourteen volume *Subh al-a 'sha* [9] included the section *considering the concealment of secret messages within letters*. Frequency analysis of letters is also attributed to them.

Following this period of time, from around the 15^{th} century was a stagnation and even a steady decline of the use of cryptography as past methods were forgotten, and it was not until well into the 16^{th} century that it again began to achieve significance in matters of state. The Renaissance was the next period of significant growth, with a number of people making contributions. One such person was Alberti, inventor of the *polyalphabetic cipher*, where both the plaintext and ciphertext equivalents are changed in relation to each other, through the use of the *Alberti Cipher Disk* [12]. Another, Trithemius, wrote *Stenographia*, and *Polygraphia* thus introducing *polyalphabetic substitution* or the shifting of letters every substitution line [13], i.e. $(a, b, \ldots y, z)$, $(b, c, \ldots y, z, a)$, $(c, d, \ldots z, a, b)$ etc. A third person of note was Belaso, whose book *La cifra del. Sig. Giovan Batista Belaso* proposed the use of a literal, easily memorable and easily changeable key [12]. While these three *polyalphabetic* solutions form the basics of modern day cryptography, they were

at the time rejected as being too time consuming and prone to errors by the cryptographers of the day in favour of the standard (and even at that time insecure) *monoalphabetic* method, and so were forgotten until their re-discovery in the 19^{th} century.

In the 19^{th} century, the advance in technology, such as the telegraph and the computing machine reinvigorated the need for security and ciphers in messages. Previous complaints about errors in messages encrypted using polyalphabetic ciphers no longer applied as these could be quickly fixed and resent by telegraph, whereas before a message could take weeks or months to be hand delivered. This continued to be the case up to, and including World War I, with various wheel, cylinder and disk based ciphers used, similar to the Alberti disk. However, at this point in the 20^{th} century, the use of machinery allowed the more widespread use of printing cipher machines by people with only a basic understanding of their mathematical operation.

The Second World War, arguably the most exciting time for cryptography and cryptanalysis in the history of the world, saw the battle in the Atlantic between the Axis, comprising Germany with the *Enigma machine*, an electro mechanical rotary based cipher machine which implemented a polyalphabetic substitution cipher through repeated changes of the electrical path through a series of replaceable rotors, versus the cryptanalysts of British intelligence's *Government Code and Cypher School* (GC&CS) at Bletchley Park set up to break the enigma codes, codename *Ultra* [14]. In the Pacific, a similar intelligence battle was raging between the United States Army's *Signals Intelligence Section* (SIS) with their cryptanalysis project, codename *Magic*, and the Japanese *Purple* encryption scheme [9]. In both cases the cryptanalysts succeeded and contributed greatly to Allied success in defeating the U-boats in the Battle of the Atlantic in the case of Ultra, and learned in advance of the Japanese attack on Pearl Harbour in the case of Magic. Winston Churchill told Britain's King George VI after World War II: 'It was thanks to Ultra that we won the war'.

After the Second World War, due to the success of cryptography, it was seen as a dangerous weapon and fell within the remit of various governmental intelligence and military organisations. It was not until the 1970s, with the advent of computers and the internet, that need for ordinary people and companies to secure their data became increasingly important. It is here we begin a

description of the work in this thesis, as this was the starting point of modern cryptography. Modern cryptography is heavily based on computer and mathematical theory. Advances in electronics and computers allow for more complex ciphers and cryptanalysis. Modern cryptographic algorithms are designed around *computational hardness* assumptions, making it theoretically possible to break any cryptographic system, but practically infeasible to do so. These schemes are therefore termed computationally secure. Mathematical advances and better computing technology require these solutions to be continually adapted both in mathematical complexity and in increased secret key sizes to maintain adequate levels of security.

2.3 Mathematical Background

In this section a variety of basic algebraic structures that play roles in the generation and analysis of sequences used in communications and cryptography is presented. A mathematical background of the underlying layers are briefly examined, namely, groups, rings, fields, finite fields and elliptic curves.

2.3.1 Groups

Groups are among the most basic building blocks of modern algebra. They are commonly used to model symmetry in structures or sets of transformations. They are also building blocks for more complex algebraic constructions such as rings, fields, vector spaces, and lattices.
A group is a set $(G, *)$ with a distinguished element e (called the identity) and a binary operation on G. This group operation is typically called addition or multiplication, denoted $*$ and satisfies the following laws:

- **Associative law**; The order of the group operation does not matter.
- $a * (b * c) = (a * b) * c, \quad \forall (x, y, z) \in G.$
- **Identity law**; G has an identity (unity) element, 1.

- $a * 1 = 1 * a = a \quad \forall a \in G.$
- **Inverse law**; For each element e of G, there exists an element e^{-1}.
- $e * e^{-1} = e^{-1} * e = 1.$

The group G is said to be commutative or abelian if the composition law is commutative:

$$a * b = b * a, \quad \forall (a, b) \in G \tag{2.1}$$

The order of a group G, denoted $|G|$, is its cardinality as a set. An element $e \in G$ is of finite order if the group contains a finite number of elements. The element e in this case is called a *generator* or *primitive* element if the order of e is equal to the number of elements in the group G,and G is *cyclic*. The order of the element e is the least positive integer, a such that $e^a = e * e * \cdots * e = 1$. Otherwise e is of infinite order.
The order of the above element e also divides the number of elements in G. If G is a group, then a subset $H \subseteq G$ is a subgroup if it is a group with the same operation as G and the same identity as G. i.e., the number of elements of a subgroup divides the number of elements of the group.

2.3.2 Rings

Rings involve algebraic structures with two interrelated operations. For example, addition and multiplication of integers, rational numbers, real numbers, and complex numbers, AND and XOR of Boolean valued functions, and addition and multiplication of $n \times n$ matrices of integers, etc.
A ring $(R, +, .)$ consists of a set, R, along with two binary operations called $(+)$ and $(.)$ (unlike a group which has just one operation), namely addition and multiplication. The ring R is said to be commutative or abelian if it satisfies the following properties for all $(a, b, c) \in R$:

- **Commutativity**; $a.(b.c) = (a.b).c \quad \forall (a, b, c) \in R.$
- R has an **Additive Identity** element, $\boldsymbol{O}$ such that

- $a + \boldsymbol{O} = \boldsymbol{O} + a = a \quad \forall a \in R.$
- **Associativity**; The order of the group operation does not matter.
- $a + (b + c) = (a + b) + c \quad \forall (a, b, c) \in R.$
- R has a **Multiplicitive Identity** element, 1, with $1 \neq 0$ such that
- $a.1 = 1.a = a \quad \forall a \in R.$
- **Distributivity**; . is distributive over $+$.
- $a.(b + c) = (a.b) + (a.c); (a + b).c = (a.c) + (b.c) \quad \forall (a, b, c) \in R.$
- **Inverse**; For each element e of R, there exists an element e^{-1}.
- $e.e^{-1} = e^{-1}.e = 1$, where 1 is called the (invertible) unit element.

2.3.3 Fields

Fields are rings where every nonzero element is a multiplicative inverse unit. A field $(\mathbb{F}, +, .)$ consists of a set, $\mathbb{F}$, along with two binary operations, namely addition and multiplication, along with an additive identity element, $\boldsymbol{O}$, and a multiplicitive identity element, 1. As an algebraic structure, every field is a ring, but not every ring is a field. The main difference being that fields allow for division (though not division by zero), while a ring need not possess multiplicative inverses.

The set must satisfy the following properties:

- **Commutativity**; $(\mathbb{F}, +, .)$ is a commutative ring such that every nonzero element is invertible.
- **Inverse**; all nonzero elements are invertable.

A subset H of a field $\mathbb{F}$ is a *subfield* of $\mathbb{F}$ if H is itself a field with respect to the operations of F. In such a case, $\mathbb{F}$ is said to be an *extension* field of H. if $H \neq \mathbb{F}$ we say that H is a *proper subfield* of $\mathbb{F}$. A field containing no proper subfields is known as a *prime field*.

Due to the fact that $\mathbb{F}$ is a field, all nonzero coefficients have an inverse and standard polynomial division can be performed. Therefore, if $g(x)$ and $h(x) \neq 0$ are polynomials in $\mathbb{F}[x]$, then there exists two polynomials $q(x)$ (the quotient) and $r(x)$ (the remainder) in $\mathbb{F}[x]$ such that:

$$\begin{aligned} g(x) &= q(x)h(x) + r(x), \\ \text{where degree}(r) &< \text{degree}(h). \\ r(x) &= g(x) \bmod h(x), \\ q(x) &= g(x) \text{ div } h(x). \end{aligned} \tag{2.2}$$

2.3.4 Finite Fields

A finite field is a field $\mathbb{F}$ which contains a finite number of elements. Finite fields are also referred to as *Galois fields* after the French mathematician Évariste Galois, who showed that the order of a finite field must be equal to the power of a prime p, and are denoted either $\mathbb{F}_q$ or $G\mathbb{F}(q)$. It can be shown that any finite field contains $q = p^m$ elements, for some prime p and some positive integer m and is isomorphic, i.e. indistinguishable, for $\mathbb{F}_q$. Since finite fields have cyclic groups, they enable algebraic constructions with finite alphabets.

The prime p is known as the characteristic of the field. The characteristic of any field is either 0 or a prime number. In cryptography, the two most studied fields are prime fields, $G(\mathbb{F}_q)$; $q = p$, where p is a prime, and binary extension fields (finite fields of characteristic two), $G\mathbb{F}(2^m)$; $q = 2^m$. The work in this thesis focuses on the prime field.

While binary fields algorithms are considered to have better performance over their prime field counterparts for hardware based implementations, it can be shown that higher level protocol implementations, e.g. digital signature generation and verification using the ECDSA, results in a similar area and power performance[i] between the two fields [15]. For a review of $G\mathbb{F}(2^m)$ and its implementation in hardware, the interested reader is referred to [16].

[i] Binary fields are however still faster and therefore more energy efficient.

2.4 Elliptic Curves

Elliptic curve cryptography (ECC) was established as a form of public key cryptosystem in 1985 independently by Miller [17] and Koblitz [18]. An elliptic curve E defined over a field $\mathbb{F}$ is defined as the set of points (x, y), with coordinates in $\mathbb{F}_q$ in an equation (in this case Weierstraß) of the form:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.3}$$

with $a_1, \ldots, a_4, a_6 \in \mathbb{F}; \delta \neq 0$, where δ is the discriminant (i.e. which shows the nature of the roots) of E, and the coefficients $a_1, \ldots, a_4, a_6$ are the elements of $\mathbb{F}$. The group of points on an elliptic curve contains all of these points (x, y) along with a special point at infinity, $\boldsymbol{O}$, the additive identity of the group. i.e. if H is any extension field of $\mathbb{F}$, then the set of H-rational points on E is:

$$E(H) = \{(x, y) \in H.H : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\boldsymbol{O}\} \tag{2.4}$$

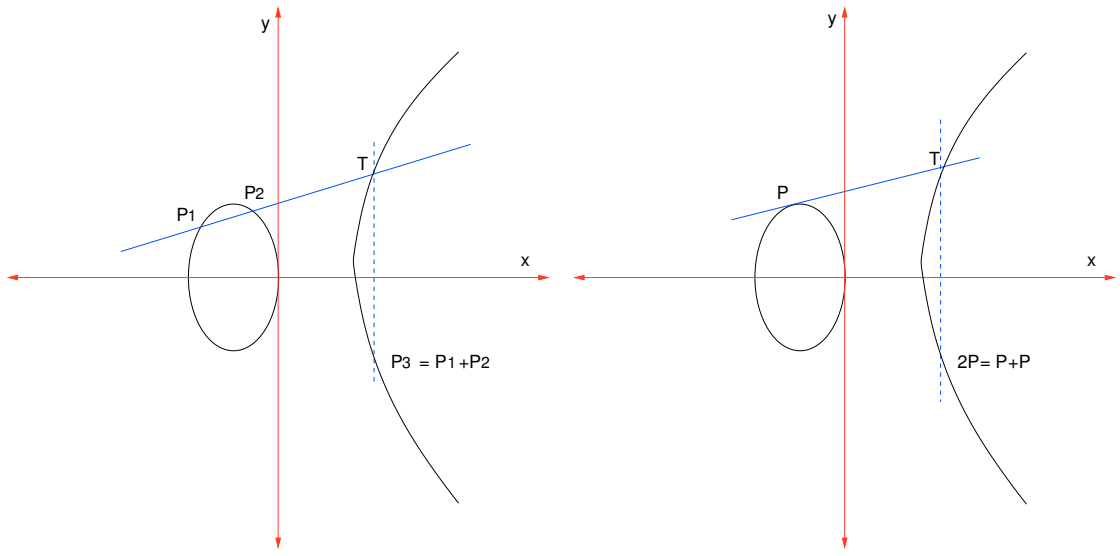
The number of points on the curve is called the order of the curve and is defined as $\#E(\mathbb{F}_q) = q + 1 - t$, where t is the Trace of Frobenius and $|t| \leq 2\sqrt{q}$ (approximated as $\#E(\mathbb{F}_q) \approx q$) [19]. If t is a multiple of the characteristic q, the curve is supersingular. Otherwise, it is non supersingular or ordinary. The basic operations of Elliptic Curve Cryptography are point scalar computations of the form:

$$Q = [k]\, g = \underbrace{g + g + \cdots + g}_{k\ times} \tag{2.5}$$

The order of the group, g, is the smallest integer n such that $[n]g = \boldsymbol{O}$, the identity element. In a cyclic group, the order of the group is the same as the order of the generator g as defined in Section 2.3.1. Repeated addition of points on an elliptic curve is known as point scalar multiplication.

2.4.1 The Group Law

For point addition (PA), the *chord-and-tangent rule* [20] is used for adding two points on the curve E, to give a third point. Together with this addition operation, the set of points over the curve form an abelian group with $\boldsymbol{O}$ as the identity. This is illustrated here in Figure 2.1(a) over a field of real numbers.



(a) Point Addition **(b)** Point Doubling

Figure 2.1: Point Operations on an Elliptic Curve

A line is drawn through $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, two distinct points on the elliptic curve. Where this line intersects the elliptic curve at a third point, T, a reflection about the x-axis is made to give the third point $P_3 = (x_3, y_3)$.

For a large k this can be quite slow. Therefore a second group operation on the elliptic curve is introduced. Point doubling (PD) is a special case of point addition in which the two input points are the same. In this case a tangent is drawn to the point on the curve. This line intersects the curve at exactly one other point, T. Again, a reflection along the x-axis is made to give the point $[2]P$. This is shown in Figure 2.1(b).

2.4.2 Elliptic Curves over Prime Fields

Elliptic Curves over prime fields, $\mathbb{F}_p$, where p is a large prime, consists of the integers $0, 1, \ldots, p-1$ with all arithmetic operations, addition, multiplication, inversion etc. performed modulo p. The Weierstraß equation defined in 2.3 can be simplified down to the *short Weierstraß equation*. If the characteristics of $H > 3$ then:

$$(x, y) \Rightarrow \frac{x - 3a_1^2 - 12a_2}{36}, \qquad \frac{y - 3a_1 x}{216} - \frac{a_1^3 + 4a_1 a_2 - 12a_3}{24}$$

transforms E to the curve

$$y^2 + xy = x^3 + ax^2 + b \tag{2.6}$$

Where $a, b \in H$; $\delta = -16(4a^3 + 27b^2)$.

A point addition, $P_3(x_3, y_3) = P_1(x_1, y_1) + P_2(x_2, y_2)$, can be described algebraically as follows:

$$x_3 = (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2 \quad \text{and} \quad y_3 = (\frac{y_2 - y_1}{x_2 - x_1}).(x_1 - x_3) - y_1 \tag{2.7}$$

A point doubling, $P_3(x_3, y_3) = [2]P_1(x_1, y_1)$, can also be described algebraically as:

$$x_3 = (\frac{3x_1^2 + a}{2y_1})^2 - 2x_1 \quad \text{and} \quad y_3 = (\frac{3x_1^2 + a}{2y_1}).(x_1 - x_3) - y_1 \tag{2.8}$$

where . is equivalent to modulo multiplication, $a * b \bmod p$.

The equations given here for Weierstraß, 2.3 and 2.6 are represented by (x, y), given in affine coordinates [19]. In Chapter 3, the different coordinate systems and their implementations are explored in greater detail.

2.5 Cryptographic Primitives & Protocols

A cryptographic protocol details the steps and components required for a security-related function using cryptographic methods. Schneier [21] defines a protocol as *a series of steps, involving two*

or more parties designed to accomplish a task. Everyone involved in the protocol must know the protocol and all of the steps to follow, and must agree to follow it. The protocol must be unambiguous and must be complete. Most Importantly for cryptographic protocols, *it should not be possible to do more or learn more than what is specified in the protocol.*

2.5.1 Symmetric-Key Cryptography

Symmetric-key cryptography, which would also be in layer 4 of Figure 1.2, involves the use of secret cryptographic keys for the encryption and decryption of data. The study of modern symmetric-key cryptography relates mainly to the study of block ciphers, stream ciphers, message authentication codes (MAC) and their applications. The cryptographic keys for encryption and decryption may be identical or there may be a simple transformation to select between the keys. As such, the keys require a shared secret between two or more parties to ensure privacy. This requirement that both parties have access to the secret key is one of the main drawbacks of symmetric key encryption.

A block cipher is the modern form of a polyalphabetic cipher, in which individual parts of the message would be encrypted differently, first proposed by Alberti in 1467 [12]. Modern block ciphers are based on the concept of an iterated product cipher. Product ciphers, as suggested by Shannon [22], involve combining multiple simple operations, usually through the use of a feistel scheme [23], substitutions and permutations, as a means of improving the security. Block ciphers take a block of plaintext data and a key, carry out encryption over multiple rounds, using different subkeys derived from an original key and output a block of ciphertext data. The strength of a block cipher is entirely dependant on the secrecy of the key value. Block ciphers are not secure cryptosystems themselves (by modern standards, it is unacceptable for the encryption of a single plaintext to always be the same), but may be used in a mode of operation such as Cipher-block chaining (CBC) mode [24] to implement secure encryption through the use of data partitioning, randomisation and padding of variable length messages. The better known block ciphers include DES [25], Triple DES and AES [26].

Stream ciphers, use a key but no plaintext input, and produce a pseudorandom output stream. To encrypt with a stream cipher, the output is combined with the plaintext, as in the one-time pad. RC4 [21] is an example of a well-known stream cipher. A MAC is used to authenticate messages. It uses a key and an arbitrary-length message to be authenticated, and outputs a tag. This tag value is used to verify both a message's data integrity as well as its authenticity, by allowing trusted verifiers to detect any changes to the message content. The verifiers are trusted by also having a copy of the secret key. An example of a MAC is the HMAC [27]. They can also be constructed from block ciphers, such as the CBC-MAC [21].

2.5.2 Public-Key Cryptography

In 1976, Whitfield Diffie and Martin Hellman [28] proposed the notion of public-key cryptography in which two different but mathematically related keys are used, a public key and a private key (although in 1997, it was publicly disclosed that asymmetric key algorithms were developed by Ellis, Cocks, and Williamson at the Government Communications Headquarters (GCHQ) in the UK in 1973, but were kept classified [29]). Deriving the private key from the corresponding public key is equivalent to solving a computational problem that is believed to be intractable. A user, Alice, generates a key pair and releases their public key while maintaining the secrecy of their private key. Any other person, Bob, with access to this public key can use it to encrypt their own data, to allow secure transmission across an unsecure channel, monitored by an eavesdropper, Eve. The message can only be decrypted by the user with the private key, i.e. Alice. Compared to Symmetric-key cryptography, this method allows the ability to transfer keys across unsecure channels while still maintaining the security of the encryption.

The Diffe-Hellman Key Exchange is used as an example to describe the operation of public key cryptography:

- Alice and Bob agree on a group G, with generator g such that for any number n in the group, there is an integer k which satisfies $n = g^k$, i.e. g is primitive mod n
- Alice chooses a random large integer a and sends to Bob: $A = g^a \bmod n$

- Bob chooses a random large integer b and sends to Alice: $B = g^b \bmod n$
- Alice computes: $k^a = B^a \bmod n$
- Bob computes: $k^b = A^b \bmod n$
- Since both k^a and k^b are equal to $g^{ab} \bmod n$, both Alice and Bob arrive at the same value

As it can be seen from above, any Eve, listening in on the channel cannot compute the value (as they only have access to the values n, g, A and B), unless they can compute the discrete logarithm, discussed further in section 2.5.4. An eavesdropper can however spoof their identity, for example in a *man-in-the-middle* attack [21]. In this situation, Eve, intercepts the public key belonging to Alice, k^a and replaces it with her own key, k^e, which is sent on to Bob. Bob, assumes that k^e is Alices public key and replies; thereby establishing a shared key with Eve g^{be}. Eve then replies to Alice to also acquire g^{ae}. It is now trivial for Eve to intercept messages between Alice and Bob, access the contents of the message and then forward it on using either g^{ae} or g^{be}.
A method of protecting against this type of *man-in-the-middle* attack is through the use of digital signatures, discussed further in Section 2.5.6.

2.5.3 The Integer Factorisation Problem (IFP)

In 1978, Ronald Rivest, Adi Shamir, and Len Adleman invented RSA [30], another public-key system.

- Two distinct prime numbers of the same bit length, p and q are chosen at random.
- n is computed, where $n = pq$
- $\varphi(n) = (p-1)(q-1)$ is computed where φ is Eulers totient
- An integer a is chosen such that a and $\varphi(n)$ are coprime
- b is calculated, where $b = a^{-1}\ mod\ \varphi n$

- Set a as the public key and b as the private key

The RSA equation is given as:

$$k^{ab} = k \bmod n, \forall k \tag{2.9}$$

Assuming that a and b are large enough, then the problem of determining the private key b from the public key (n, a) is computationally equivalent to the problem of determining the factors p and q of n. This is otherwise known as the Integer Factorisation Problem (IFP).

2.5.4 The Discrete Logarithm problem (DLP)

The discrete logarithm problem is defined as the problem of determining k such that $[k]g = Q$, given g and Q using Equation 2.5. Described another way, given b mod p and b^n mod p, find n. Computing the exponentiation modulo the group order, (i.e. the encryption of) $[k]g$, can be performed efficiently i.e. in polynomial time. The strength of the DLP comes from the inverse operation; finding the logarithm is deemed intractable, i.e. computable in exponential time, and is considered to be roughly equivalent to the difficulty of the IFP. The Diffe-Hellman key exchange described in Section 2.5.2 is based on this problem.

2.5.5 The Elliptic Curve Discrete Logarithm problem (ECDLP)

The principles of the DLP can be applied to elliptic curves, resulting in the ECDLP.
Given an elliptic curve, E defined over a finite field $\mathbb{F}_q$, a point $P \in E(\mathbb{F}_q)$ of order n, and a point $Q \in E(\mathbb{F}_q)$ determine the integer $n \in [0, \ldots k, \ldots n-1]$ such that $Q = k[P]$

$$Q = [k]\,P = \underbrace{P + P + \cdots + P}_{k\ times} \tag{2.10}$$

The operation $Q = k[P]$, is referred to as a *point scalar multiplication*. Similar to the DLP, computing $Q = k[P]$ can be performed in polynomial time, while the inverse operation is deemed NP-hard (non-deterministic polynomial-time hard) (although no mathematical proof of this exists).

Given a small enough k, a *brute force attack* can be used to calculate the sequence of points of P until k is found, however this is considered intractable once sufficiently large primes are used. There are other attack methods uses to reduce the time taken or the complexity of the ECDLP, however again, these are infeasable when using a large prime. The Pollard Rho [31] and the Pohlig-Hellman [32] are two such attacks.

Since the ECDLP appears to be significantly harder than the DLP, the strength per key bit is substantially greater in elliptic curve systems than in conventional discrete logarithm systems. Thus, smaller parameters can be used in ECC than with DL systems but with equivalent levels of security. The advantages that can be gained from smaller parameters include speed (faster computations) and smaller keys and certificates. These advantages are especially important in environments where processing power, storage space, bandwidth, or power consumption is constrained.

2.5.6 Digital Signatures

It was described in Section 2.5.2 how the Diffe-Hellman Key Exchange is susceptible to *man-in-the-middle* attacks. To counter these types of attacks, Diffie and Hellman also proposed the use of a signature scheme [28]. The Digital Signature Algorithm (DSA) was proposed in August 1991, by NIST and was specified in FIPS-186 [33] in May 1994. Its security is based on the intractability of the discrete logarithm problem described in Section 2.5.4. By first signing the message with their private key, prior to encrypting it with the recipients public key, the receiver can verify the authenticity of the message using the senders public key. Essentially, the sender performs the digital equivalent of a handwritten signature.

Johnson [34], defines a digital signature as *a number dependent on some secret known only to the signer (the signers private key), and, additionally, on the contents of the message being signed. Signatures must be verifiable if a dispute arises as to whether an entity signed a document, an unbiased third party should be able to resolve the matter equitably, without requiring access to the signers private key*. Digital signatures should be essentially unforgeable [35], and are used to provide:

- **data Integrity**; The assurance that data has not been altered by unauthorized or unknown means.
- **data origin authentication**; The assurance that the source of data is as claimed.
- **non-repudiation**; The assurance that an entity cannot deny previous actions or commitments.

The digital signature schemes can be classified according to the hard underlying mathematical problem which forms the basis of their security:

- **Integer Factorization** (IF) schemes, which base their security on the intractability of the integer factorization problem. e.g. RSA [30] and Rabin [36] signature schemes.
- **Discrete Logarithm** (DL) schemes, which base their security on the intractability of the (ordinary) discrete logarithm problem in a finite field. e.g. ElGamal [37], and DSA [33] signature schemes.
- **Elliptic Curve** (EC) schemes, which base their security on the intractability of the elliptic curve discrete logarithm problem. e.g. ANSI X9.62 [38], FIPS 186-3 [33], IEEE 1363-2000 [39] and ISO/IEC 15946-2 [40].

The Elliptic Curve Digital Signature Algorithm (ECDSA) along with its implementation are presented in Chapter 5.

2.5.7 Cryptographic Key Sizes

There are many different parameters which need to be chosen with care to ensure that an encryption scheme is resistant to all known attacks. This work only ensures that the size of the prime field and the key size are sufficiently large as to be representative of this security metric. The interested reader is referred to Chapter 4 of [19] for a full list of the other parameters and their generation.
As discussed earlier in this section, in symmetric key cryptography, the size of the key is directly related to the security of the system, whereby, the larger the key size, the greater the security level.

Table 2.1: ECRYPT II Security Level Recommendations (2010)

Level	Protection	Symmetric	Asymmetric	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
1	Attacks in real-time by individuals *Only acceptable for authentication tag size*	32	-	-	-	-	-
2	Very short-term protection against small organizations *Should not be used for confidentiality in new systems*	64	816	128	816	128	128
3	Short-term protection against medium organizations, medium-term protection against small organizations	72	1008	144	1008	144	144
4	Very short-term protection against agencies, long-term protection against small organizations *Smallest general-purpose level, 2-key 3DES restricted to 2^{40} plaintext/ciphertexts, protection from 2009 to 2012*	80	1248	160	1248	160	160
5	Legacy standard level *2-key 3DES restricted to 10^6 plaintext/ciphertexts, protection from 2009 to 2020*	96	1776	192	1776	192	192
6	Medium-term protection *3-key 3DES, protection from 2009 to 2030*	112	2432	224	2432	224	224
7	Long-term protection *Generic application-independent recommendation, protection from 2009 to 2040*	128	3248	256	3248	256	256
8	Foreseeable future *Good protection against quantum computers, unless Shors algorithm applies*	256	15424	512	15424	512	512

For asymmetric key cryptography, the standard benchmark is RSA, and in public-key cryptography, two algorithms are considered to offer equivalent security (in a cryptographic sense), if the work needed to break them is the same using a given resource [41][ii]. As such, Table 2.1 presents the recommendations for the minimum keysize required (in bits) data as defined by the Ecrypt II [42] network of excellence in cryptology. The table gives eight levels of security, with level 5 being the current standard acceptable level of security. The second column presents the protection being offered, while the rest present a symmetric key size from which equivalent asymmetric key sizes are built. Column 3 presents the equivalent security in bits for a private key scheme. Column 4 indicates the bits required in RSA. Column 5 and 6 indicate the bits required in a system based on the DLP in a finite field for both key and field size. Column 7 gives the bits required for systems based on the ECDLP, while column 8 gives the recommended bit sizes for the SHA-2 hash algorithm. It can be immediately seen that the advantage ECC gives in size when compared

[ii]The security strength represents the amount of work (i.e. the number of operations) that is required to break a cryptographic algorithm or system. It is dependent on time complexity, computational resources, memory/data and the possibility that certain specific attacks to that algorithm may provide computational advantages.

to RSA, and how this advantage grows as the security level increases.

Table 2.2: NIST Security Recommendations (2011)

Date	Minimum Strength	Symmetric	Asymmetric	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	(SHA) Hash
2010	80	2TDEA	1024	160	1024	160	224,256,384,512
2011 - 2030	112	3TDEA	2048	224	2048	224	224,256,384,512
$>$ 2030	128	AES-128	3072	256	3072	256	256,384,512
$>>$ 2030	192	AES-192	7680	384	7680	384	384,512
$>>>$ 2030	256	AES-256	15360	512	15360	512	512

For comparison, Table 2.2 gives the NIST recommended minimum key sizes [41]. The first column indicates the timeframe that the level of security is likely to cover. Columns 2 and 3 give the symmetric security level along with the symmetric algorithm required to provide this cover. The subsequent columns again present the equivalent RSA, DLP, ECDLP and SHA-2 security provided. The analysis from both groups is fairly similar with level 5 and 6 from Table 2.1 being comparable to rows 1, 2 and 3 from Table 2.2.

2.6 Hardware Overview

Next, the different hardware used in this thesis is described. A Field Programmable Gate Array (FPGA) is an integrated circuit which is user programmable; as opposed to an Application Specific Integrated Circuit (ASIC), which is customised by the manufacturer for a particular use. FPGAs are an attractive choice for implementing cryptographic algorithms, because of their low cost in prototyping relative to ASICs. FPGAs are flexible when adopting security protocol upgrades, as they can be re-programmed in-place, and FPGAs also allow rapid prototyping of designs. The downsides however are they are larger in area and higher in power usage when compared to ASICs. An FPGA can be described as an array of configurable logic blocks and interconnects, all of which can be programmed by the user to describe combinational and sequential logic circuitry. The components and connections which make up these logic blocks however, vary to a greater or lesser degree between different manufacturers, and even between different families of the same manufacturer.

Xilinx [43] and Altera [44] FPGAs are the two FPGA products used for the implementations of the hash functions. The two basic measurement standards of an FPGA in the case of a Xilinx device are Configurable Logic Blocks (CLB) or Slices, and for an Altera device they are, Logic Array Blocks (LAB), Adaptive Logic Modules (ALMs) or Logic Elements (LEs).
For a more indepth description of the different types of FPGAs, ASICs and microprocessors, the interested reader is invited to examine Chapter 3 of [16], or from the author [45].

2.6.1 Xilinx FPGA

Configurable logic blocks (CLB) are organized in an array and are used to build combinatorial and synchronous logic designs. Each CLB element is tied to a switch matrix to access the general routing matrix. A CLB element comprises a number of similar slices, with fast local feedback within the CLB. These slices are split into columns with independent carry logic chains and common shift chain.
Each slice includes a number of multi-input Look-Up Tables (LUT), carry logic, arithmetic logic gates, wide function multiplexers and storage elements, namely D-type flip-flops. CLBs can be configured to operate as either a logic or a memory element. When operating as a logic element, the LUTs are programmed to operate as combinational logic, with a 1-bit register, or as a variable-tap shift register. As a memory element, each LUT is configured as an $2^n \times$ 1-bit Distributed RAM block, where n is the number of inputs to the LUT. The slices also contains logic that combines function generators to provide multiplexing, sum of products (SOP) chains, shift registers and tri-state buffers used to drive on-chip buses.
In most cases, Xilinx CLBs share a similar make-up. There are distinctions though, as certain families are designed for different specific purposes. For example, the Virtex-II Pro can use each LUT as a $16\times$ 1-bit RAM. Others come with a specific designation; 'L' denoting a low power version for example. The underlying technology is quite similar however. For the Spartan-3 onward, slices are seperated into those which include built-in RAM resources (SLICEM) and those which do not (SLICEL), with two of the four slices in a CLB being SLICEM and two being

SLICEL. This allows each CLB to operate both as logic or memory.

The exception to this is the newer Virtex-5 and Virtex-6 families. They incorporate larger LUTs in their CLB, and as such, can achieve more varied functionality than the previous generation FPGAs (at a greater monetary cost). SLICEM logic for the Virtex-5 and Virtex-6 can be configured as 64-bit Distributed RAM. While the Virtex-5 [46] was the main FPGA technology used in this work, some implementations were also performed and comparisons were made a using the Spartan-3 [47] and as such it is included here. Table 2.3 summarizes the logic resources in one CLB for each of the target FPGA types.

Table 2.3: Area logic resources in one CLB per FPGA Type

FPGA Type	Slices	LUTs 4-input	LUTs 6-input	Storage Elements	Distributed RAM (bits)	Shift Registers (bits)
Spartan 3	4	8	-	8	4x16	4x16
Virtex 5	2	-	8	8	4x16	2x64

All of the CLBs in a given FPGA device are identical and each CLB (or slice equivalent) can be implemented in one of the configurations listed above. For the Distributed RAM and Shift-Registers columns, the values given refer to the number of bits that one slice (or SLICEM) LUT can be configured to store. For example, each Spartan-3 SLICEM LUT can be configured as a 1×16-bit shift register, with 4 SLICEM LUTs per CLB, thereby allowing 4×16-bit shift registers per CLB. These can also be configured in a smaller number of larger shift registers.

2.6.2 Memory and DSP Blocks

Some Xilinx FPGA devices also incorporate large block memory resources (BRAM) and dedicated multiplier or DSP blocks.

The BRAM complements the distributed memory resources that provide shallow RAM structures implemented using the CLBs. Implementing using BRAM (for example, S-boxes), can improve the clock frequency while also reducing the number of slices required. Note the BRAM usage does not show up in the CLB area result and so must be taken into account separately.

For the two devices, the Spartan-3 contains dedicated 18×18-bit, twos complement signed multipliers. The Virtex-5 has an equivalent DSP block (DSP48E). In this case, each DSP48E slice contains a 25×18 multiplier, an adder, and an accumulator. Table 2.4 summarizes the total area per device type used in this work in slice and block memory available. The Spartan-3 device presented here is the XC3S5000 and two different Virtex-5 FPGAs are used; the XC5VLX50, used on the SASEBO-GII cryptographic evaluation board [48], detailed further in Section 2.8, and the XC5VLX110T, used on the Xilinx XUPV5-LX110T Evaluation Platform [49], detailed further in Section 2.8.1.

Table 2.4: Total Area logic resources per FPGA Type

	Slices	BRAM (Blocks)	BRAM (Kb)	Dedicated Multipliers	DSP Blocks	Max User i/o's
xc3s5000	33,280	104	1,872	104	0	633
xc5vlx50	7,200	144	1,728	0	32	560
xc5vlx110t	17,280	444	5,328	0	64	680

2.6.3 FPGA Design

The Xilinx ISE Design Suite 12.3 [50] was used as the toolkit for the designs presented here and VHDL (Very High Speed Integrated Circuit Hardware Description Language) was chosen to implement the designs. The interested reader is directed to [51] for further information on VHDL. The Synthesis tool takes the VHDL code and maps it to the physical components (i.e. the CLBs, Slices and BlockMemory described in Section 2.6.1) of the target FPGA.

An initial *synthesis* determines that the circuit is error free and behaving correctly. Once this is confirmed, *post-place-and-route* (P-P-R) is performed which maps the logic to the target FPGA and returns a more accurate area and timing result. A *bit-file* is then generated which can be downloaded to the FPGA, allowing it to perform the circuitry assigned to it by the VHDL. Further information on architecture, implementation and optimization can be found in [51, 52].

2.7 Microblaze

A microprocessor incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC). Microprocessors available for use in Xilinx FPGAs can be broken down into two broad categories; soft-core and hard-core. Hard-core microprocessors, for example the IBM PowerPC (Performance Optimization With Enhanced RISC Performance Computing) [53], are embedded into the fabric of certain FPGAs, namely some members of the Virtex-II Pro, Virtex-4 and Virtex-5 FPGA families. These hard-core processor blocks cannot be removed from the chip irrespective of whether they are used or not in any particular design. This method is also known as System on a Chip (SOC).

The Microblaze [54], a soft-core virtual microprocessor, is a 32-bit Harvard RISC (Reduced Instruction set Computer) architecture optimised for Xilinx FPGAs. It is designed using predefined blocks, called cores, in the general-purpose memory and logic fabric an FPGA. Using this method of adding cores as necessary, a specific microprocessor can be created containing only the blocks required, thereby only using as much space as is necessary on the FPGA. It was selected for use in the work presented in the thesis over other hard-core microprocessors due to its availability on the Virtex-5 devices used (see Section 2.8). The MicroBlaze is highly configurable, allowing a specific set of features to be applied to a design in addition to the FPGA hardware blocks implemented in VHDL.

2.7.1 Microblaze Architecture & Implementation

The basis of the microblaze architecture is a single issue 3-stage pipeline, thirty-two 32-bit general purpose registers, 32-bit instruction word with three operands and two addressing modes, a 32-bit address bus, an ALU a shift unit and two levels of interrupt. This base design can then be be added to with more advanced features, such as a barrel shifter, divider, multiplier, floating point unit, Fast Simplex Link (FSL) interface etc. to allow a tailoring of the design to the required specifications. Figure 2.2 gives a visual representation of the microblaze system, with the white items comprising the base design and the shaded items showing some optional features.

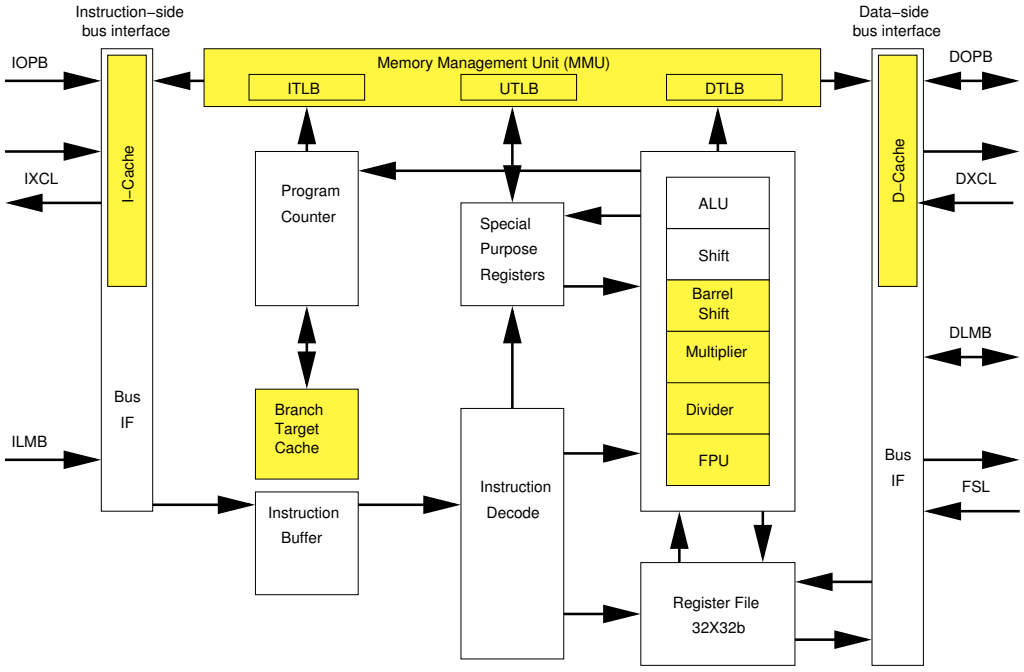


Figure 2.2: Microblaze Processor

The processor has up to three interfaces for memory accesses; Local Memory Bus (LMB) to provides single cycle access to on-chip dual port BRAM, the IBM On-chip Peripheral Bus (OPB) to provide a connection between on-chip and off-chip peripherals and memory, and Xilinx Cache-Link (XCL) for use with specialised external memory controllers. Microblaze also supports up to eight FSL ports, each with one master and one slave FSL interface.

Xilinx Embedded Design Kit (EDK) [55] is a suite of tools and Intellectual Property (IP) that enables a user to design an embedded processor system for implementation. It mainly comprises of two main sections:

- Xilinx Platform Studio (XPS): The development environment used for designing the hardware portion of the embedded processor system.
- The Software Development Kit (SDK): An integrated development environment, complementary to XPS, that is used for C/C++ embedded software application creation and verification. SDK is built on the Eclipse open-source framework

2.7.2 FSL Bus

The Fast Simplex Link (FSL) [56] is a uni-directional point-to-point communication channel bus used to perform fast communication between the Microblaze processor pipeline and user developed custom hardware accelerators (co-processors). This provides a mechanism for unshared and non-arbitrated communication mechanism, thus allowing fast transfer of data words between master and slave implementing the FSL interface. Table 2.5 gives the I/O signals used by the FSL.

Table 2.5: FSL Bus Signals

Signal	IO	Description
FSL_Clk	in	Synchronous clock
FSL_Rst	in	System reset, should always come from FSL bus
FSL_S_Clk	in	Slave asynchronous clock
FSL_S_Read	in	Read signal, requiring next available input to be read
FSL_S_Data	out	Input data. Multiple of 32-bit
FSL_S_Control	out	Control Bit, indicating the input data are control word
FSL_S_Exists	out	Data Exist Bit, indicating data exist on the input FSL bus
FSL_M_Clk	in	Master asynchronous clock
FSL_M_Write	in	Write signal, enabling writing to output FSL bus
FSL_M_Data	in	Output data. Multiple of 32-bit
FSL_M_Control	in	Control Bit, indicating the output data are control word
FSL_M_Full	out	Full Bit, indicating output FSL bus is full

The input and output are read in 32-bit blocks and requires two clock cycles, a read and an acknowledge, per block. FIFO buffers read and release the data and are set prior to runtime.

2.8 Hardware Architecture

The hardware used to test the various algorithms, was implemented on the Xilinx Virtex-5 FPGA and evaluated on the SASEBO-GII cryptographic evaluation board [48], as presented in Figure 2.3. The SASEBO GII evaluation board comprises of:

- Two Xilinx FPGAs:
  - A cryptographic FPGA : XC5VLX50 -FF324 (Virtex-5 series).

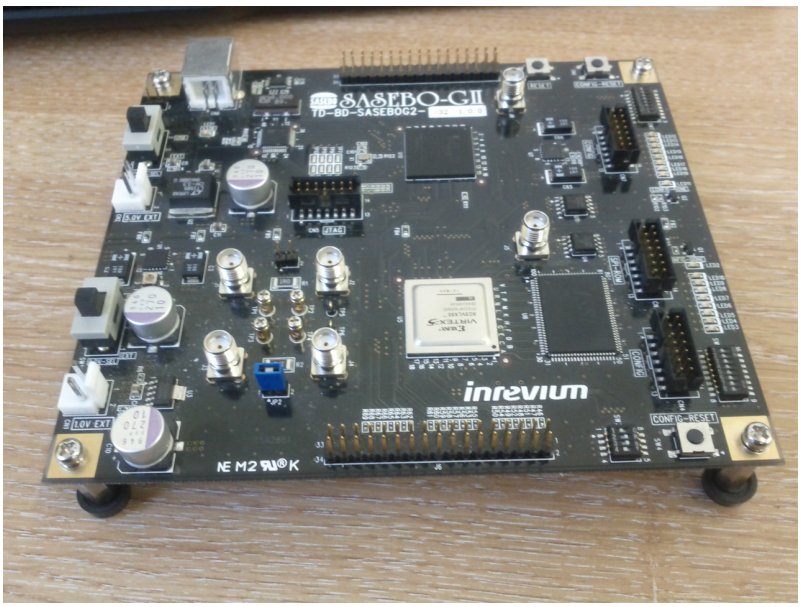


Figure 2.3: Sasebo GII

  - A control FPGA : XC3S400A-4FTG256 (Spartan-3A series).

- An onboard 24Mhz clock. An external clock input is also supported.

- External power source supplying the on-board power regulators and the FPGAs.

2.8.1 Additional Hardware

The Microblaze designed for this work mostly uses a stripped down version, necessitated by size constraints and a lack of external RAM, on the main Virtex-5 XC5VLX50 device used on the SASEBO. However, some testing was also completed for comparison purposes on the Xilinx XUPV5-LX110T Evaluation Platform [49], which uses a Virtex-5 XC5VLX110T, and allows full use of the Microblaze components. The maximum clock frequency of the Microblaze when implemented on this board is 125MHz. The board contains 256MB 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, the heap and stack are placed in 64kB of BRAM internal to the FPGA. The general setup of the system is shown in Figure 2.4. In comparison, the SASEBO

design comprises the base Microblaze, an XPS timer, a clock generator, BRAM and some ancillary components.

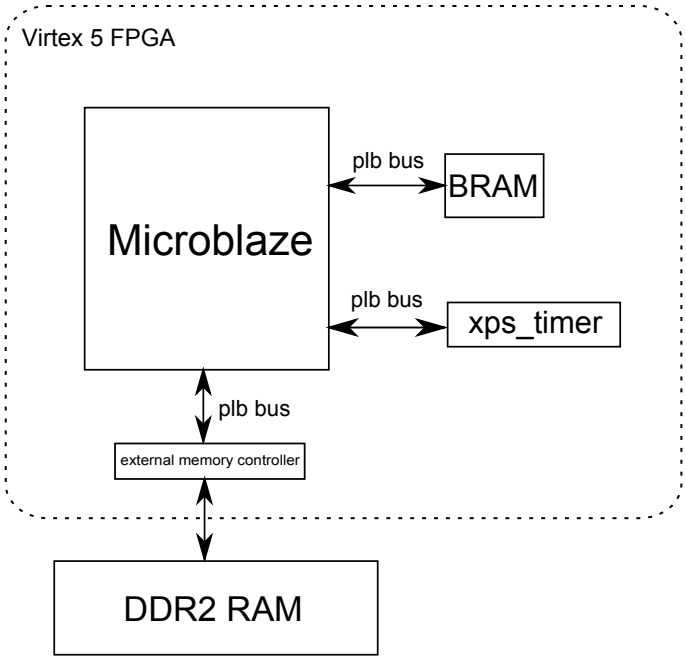


Figure 2.4: Microblaze Design on XUPV5

The description given above is necessarily short and the interested reader is directed to [54, 55, 57] for a complete description of the Microblaze and its components.

2.9 Hardware Constraints

As a final overview in this section, some constraints and trade offs are examined. Whether necessitated by security, the throughput required for a system, or the size limitations associated with a particular device, constraints are required to achieve the best performance for any particular metric at the cost of others. Here, a brief description of side channel attacks along with some design constraints are presented.

2.9.1 Side Channel Attacks

One element to take into account when designing a public key cryptosystem is the security of the system. Implementations can leak sensitive information during the execution of a computation [58], and this may lead to a release of secret information, no matter how mathematically secure the system may be.

A side channel attack (SCA) is any attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms. This method consists of monitoring some side channel information, such as the power consumption emmisions [59], and using the data emitted to deduce, or partially deduce, the inner workings of the system [60] through the leaked information, such as timing information, power consumption or electromagnetic leaks, in some cases making recovery of the key a trivial matter.

Power analysis attacks can be grouped into two categories, simple power analysis (SPA) and differential power analysis (DPA) [59]. Simple power analysis (SPA) involves visually interpreting power traces from an oscilliscope while Differential power analysis (DPA) is a more advanced form of power analysis which statistically analyses data collected from multiple runs of cryptographic operations. As such, in order to carry out such an attack, the attacker needs physical access to the device running the cryptographic algorithm. Smart cards are particularly vulnerable to this type of attack as the attacker normally has unlimited access. There are several countermeasures available to prevent such an attack. Since this work is concerned with elliptic curve implementations we only concern ourselves with the mathematical countermeasures applicable to them, such as the various algorithms presented in Chapter 3. An example of an FPGA under SPA attack is shown in Figure 2.5. The different instructions being processed can clearly be observed on the oscilliscope measuring the power line.

2.9.2 Area, Speed, Power and Energy

Along with the surge in demand for smaller and smaller handheld computerised devices such as smart phones and smart cards, is the demands for longer battery life, more functionality, higher

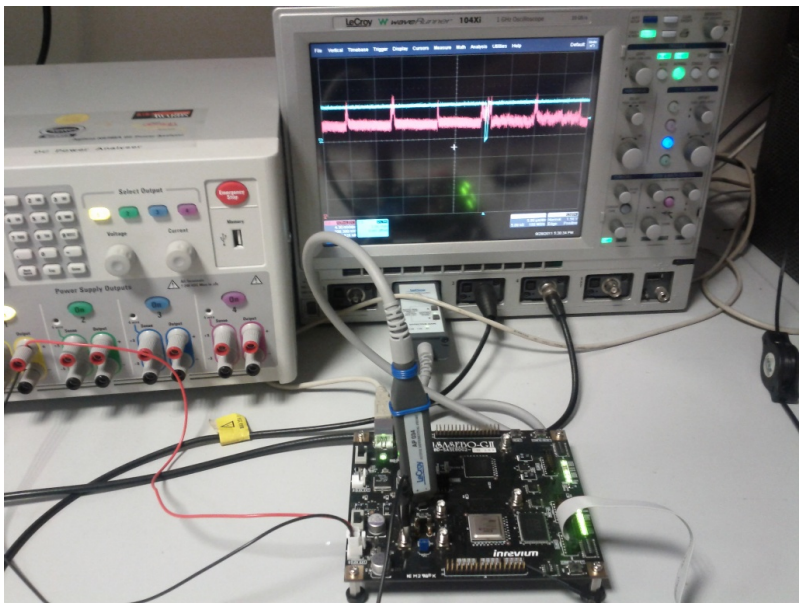


Figure 2.5: SPA analysis using FPGA

speeds and lower power. All of which are critical design constraints, and in many occasions a trade off is needed to select the best ratio of one to another.

The energy consumed by an electronic circuit when performing a computation is equal to the product of the average power and the computation time. Therefore, to minimise the energy a designer must attempt to reduce both the power consumption and the calculation time (speed).

Dedicated hardware to perform elliptic curve operations can help to alleviate both of these needs at the same time. A small dedicated circuit will have a lower power consumption than a large, general purpose processor. Also, by exploiting parallelisation in the design the circuit will also be able to reduce the computation time. However care must be taken, as there comes a point where the addition of extra parallelisation leads to a decrease in the efficiency, and results in a small increase in speed at the cost of large increase of area. Both of these factors combine to offer potential energy savings that could offset the initial cost of the design of dedicated hardware. Through the careful examination of the Area-Time and Area-Energy tradeoff, where Power P and Energy E are defined by:

$$P = IV, \tag{2.11}$$

$$E = P_{Avg}T, \tag{2.12}$$

where V is the voltage, I is the current, P_{Avg} is the average power and T is the time.

Although the Xilinx ISE Design Suite provides its own power estimation tool, XPower Analyzer, to measure the power consumption of FPGA devices, the estimated values derived by the tool are considered to be not particularly accurate and estimation errors have been shown to range from 17.5%-200% [61]. Therefore, for accuracy, all power and energy results are measured from the hardware.

2.10 Performance Metrics

As a method of comparing the different designs presented in this thesis, some metrics were developed. The area-time product (ATP) was calculated to get a representation of any speed decrease relative to the increase in design size. This gives a more accurate representation of the cost in relation to the overall system. The area-energy product (AEP) was calculated using the power to give a representation of the energy increase against the increase in size. The power generally increases with an increase in area, however the energy costs are reduced due to the calculation being performed faster. This shows the best increase in area for a decrease in energy. In both cases, the lower the value, the better the performance.

Another standard metric used to allow a baseline comparison between designs, namely the area-speed trade-off, is the throughput per unit area (TPA) metric. Analysing the throughput per slice of the architectures, determines which designs make the most efficient use of FPGA area.

The throughput is calculated as follows:

$$\text{Throughput} = \frac{\text{\# Bits in a message block} \times \text{Maximum clock frequency}}{\text{\# Clock cycles per message block}} \tag{2.13}$$

Along with these metrics, the area, computation speed, number of clock cycles, power and energy measurements are all used to describe the relative costs and benefits of each of the designs

presented in this thesis.

2.11 Conclusions

This chapter has provided the background information necessary for the rest of the thesis. The concept of cryptography was introduced and a short background of its history was presented.

It was shown that an elliptic curve system consists of four main layers, the finite field layer, the elliptic curve point operation layer, the scalar multiplication layer and the protocol layer. Each of these layers of elliptic curve cryptography were presented and described, along with the mathematical background necessary to understand them.

A description of cryptographic primitives and protocols was given. Symmetric-Key and Public-Key cryptography were described. The Integer Factorisation Problem (IFP), the Discrete Logarithm problem (DLP) and the Elliptic Curve Discrete Logarithm problem (ECDLP) were defined and how they are used in protocols was described. A short analysis of cryptographic key sizes was also presented.

The platform tools used to form the underlying technology were described with a short introduction to FPGAs and the Microblaze soft-core processor along with the Xilinx Embedded Design Kit used to generate it. The Sasebo GII cryptographic evaluation board and the Xilinx XUPV5-LX110T Evaluation Platform were presented, upon which, most implementation testing was done.

Constraints associated with hardware implementations were examined and the concept of side channel attacks was introduced along with various countermeasures relevant to this work. Trade-offs between area, speed, power and energy were also presented and some metrics were defined which allow a baseine comparison between the different implementations presented in the thesis.

3

Elliptic Curve Cryptography

3.1 Introduction

It was seen in Chapter 2 that finite field arithmetic underpins all elliptic curve based cryptosystems. The efficiency of the system will depend in large part on the efficiency with which addition, multiplication, squaring and division/inversion can be performed in the underlying field. An assumption is often made that metrics and operations associated with software easily port over and compare to their hardware equivalents [20, 62]. While for the general case this is true, there are certain aspects specific to hardware, which provides a benefit for using it. Hardware is generally much faster than software, requires no additional processing power on the part of the computer

and generally has more integrity than software-based encryption since it usually operates as a stand alone unit whereas software based solutions tend to share resources. Indeed the inverse is also true and software provides benefits of its own; much simpler to deploy and is relatively easy to extend access to various users.

However, there are some specific differences between implementing any particular algorithm in hardware or in software. For example, on FPGAs, RAM blocks are used to hold the coordinates and results from particular calculations. Remarked in various literatures [63, 64] is the use of as minimal as possible temporary registers for resource constrained systems. This is not so relevant on FPGAs, as through the use of these blocks of RAM (BRAM), the minimum number of BRAM available for any $p_b = 192$, where p_b is the field size in bits, is identical for any number of addresses between 2 and 1024.

Another difference between hardware and software, specifically for prime field ($\mathbb{F}_p$) arithmetic in this case, is the use of squarings. Squarings are relatively straightforward in software, and indeed in binary extension fields, $G\mathbb{F}(2^m)$, where a dedicated squarer can be significantly more efficient in terms of both time and area, and essentially comprises of a two step process of inserting interleaved zeros followed by a reduction [16, 65]. In $\mathbb{F}_p$ a squaring is essentially equivalent to a multiplication, and as such, multiplication architecture tends to be used to perform field squaring simply by setting the two inputs of the multiplier to the same value, rather than implementing a specific squaring architecture, which would result in extra area for only a minor computation time increase.

However, results presented for the current state of the art algorithms tend to be based on the cost in software [20, 66] and distinctions are made between multiplications and squarings. Taking an example from the previous chapter for point addition and point doubling as given in Equation 2.7 and Equation 2.8, it can be seen that the computations involve multiplications (M), squarings (S) inversions (I), additions (add) and subtractions (sub). In the context where p is a large prime, it is often assumed for software that:

- The inversion cost is $\mathsf{I} \approx 10\mathsf{M}$.

- The squaring cost is $\mathsf{S} \approx 0.8\mathsf{M}$.
- The addition cost is equal to the subtraction cost and only takes a few clock cycles, $\mathsf{add} \approx \mathsf{sub}$.

While these costs are architecture reliant in a hardware based system, they are in general equivalent. For the work presented in this thesis, a multiplication takes p_b clock cycles, where p_b is the field size in bits and squarings are seen as equivalent to multiplications. It is also stated numerous times in the literature [63, 64, 67]) that field addition and subtraction costs are generally ignored as being negligable in comparison to multiplications or inversions. This is in general true in a one-to-one conversion, however, as shown in [68], these operations are not insignificant for hardware devices, especially when taken alongside other relatively small but numerous operations. The additions and subtractions presented here take between $2 - 4$ clock cycles.

While these differences between software and hardware would all appear to be minor, the numerous calculations required to be performed over a large prime field can result in some significant differences between results reported in the literature for software based implementations and actual implemented hardware performance.

This chapter examines the performance of various elliptic curve algorithms for prime field ($\mathbb{F}_p$) arithmetic on a reconfigurable elliptic curve processor (ECP). The underlying modular arithmetic is presented and different parallel configurations of the processor are investigated in an attempt to select the best performing algorithms for a hardware based system.

The rest of the chapter is presented as follows. Section 3.2 describes the various forms of dedicated point doubling and point addition algorithms, namely the *Double-and-Add* algorithm, specifically in a number of differing coordinate systems, continuing on from the introduction provided in 2.4.2. Section 3.2.4 describes a new form of special curve, namely the *twisted Edwards*. Recent updates to this type of curve in the form of the *extended twisted Edwards* are also examined.

Section 3.3 then describes the *reconfigurable elliptic curve processor* in detail and presents methods for *modular arithmetic* along with methods for performing the *scheduling* and calculating the *efficiency*. Finally, the algorithmic cost of these algorithms and their cost in area for the elliptic

curve processor are examined for differing numbers of *multipliers in parallel*.

Section 3.4 describes a method and circuitry for examining the *power dissipation* and for calculating the *energy* and *computation time* for the various algorithms.

Section 3.5 then goes on to further describe *power analysis attacks* and how the algorithms examined so far are susceptible to this type of side channel attack. Various *power analysis attack secure methods* are then examined, namely *dummy arithmetic instructions* in Section 3.6, *unified doubling and addition* in Section 3.7 and *regular algorithms* in Section 3.8. The algorithmic cost and area, energy and timing results of these SPA secure algorithms are examined in Section 3.9 and Section 3.10.

Section 3.11 then expands on the top performing algorithms and examines the performance for *larger key and field parameters*, examining the additional cost associated with security for the current and forseeable future based on NIST specifications. Finally conclusions are presented in Section 3.12.

3.2 Dedicated Doubling and Addition

The *Double-and-Add* algorithm is the classic method for evaluating scalar multiplication, $Q = k[p]$. It works on the relation that

$$\begin{aligned} k[p] &= 2([\frac{k}{2}]p); \;\; k = even, \\ k[p] &= (2([\frac{k}{2}]p) + p); \;\; k = odd. \end{aligned} \tag{3.1}$$

Iterating the process yields a scalar multiplication algorithm, left to right scanning scalar k. This algorithm is also known as the *left-to-right binary method* and is presented in Algorithm 1. This *Double-and-Add* algorithm requires $b_k - 1$ point doubling operations where b_k is the bit length of the key, and $w(k)$ point additions, where $w(k)$ is the Hamming weight of the key.

Algorithm 1: Double-and-Add

input : $p \in E(\mathbb{F}_q); k = \sum_{i=0}^{n_k-1} k_i 2^i$
output: $Q = [k]p \in E(\mathbb{F}_q)$

Initialise: $Q = p$;
for $i \leftarrow n_k - 2$ *to* 0 **do**
  $Q = 2Q$ *–point double (PD);*
  if $K_i = 1$ **then**
    $Q = Q + p$ *–point addition (PA)*
  end
end

3.2.1 Affine Coordinate System

Elliptic point representation in two coordinates, (x, y), is called *affine representation*. Simplifying equations 2.7 and 2.8 to equations 3.2 and 3.3 for (PA) and (PD) respectively, it is obvious that the computational cost in affine coordinates for the *point addition* (PA) formula for this type of representation comprises one inversion and three multiplications, 1I, 2M, 1S and 6add while the *point doubling* (PD) formula comprises one inversion and four multiplications, 1I, 2M, 2S and 4add[i].

$$\begin{aligned} \lambda &= (\frac{y_2 - y_1}{x_2 - x_1}) \\ x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1. \end{aligned} \tag{3.2}$$

$$\begin{aligned} \lambda &= (\frac{3x_1^2 + a}{2y_1}) \\ x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1. \end{aligned} \tag{3.3}$$

Clearly, for both PA and PD the field Inversion will be the dominant calculation and as such the most costly operation. To avoid the need to compute these costly inversions, other coordinate systems can be used.

3.2.2 Projective Coordinate System

One such coordinate system is projective coordinates, which uses the set of three points (X, Y, Z). Using this method, inversion can be avoided, albeit at the cost of extra additions. Conversions from

[i] As stated earlier in the introduction M = S in $\mathbb{F}_p$ and as such are interchangeable. Where necessary, they will be referred to specifically as M

and to affine coordinates are made and occur only once, at the beginning and the end of a point scalar multiplication. To convert from affine to projective coordinates, it is sufficient to set the Z coordinate equal to one.

$$(x, y) \mapsto (X, Y, 1) \tag{3.4}$$

The cost of conversion back to affine from projective is two multiplications and an inversion.

$$(\frac{X}{Z}, \frac{Y}{Z}) \mapsto (x, y) \tag{3.5}$$

As such, the projective form of the Weierstraß equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{3.6}$$

defined over a field k is

$$E : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{3.7}$$

For standard projective coordinates, after re-labelling $a_4 = a$, $a_6 = b$, the short Weierstraß equation can be defined [20]:

$$y^2 = x^3 + ax + b \tag{3.8}$$

The projective form of the short Weierstraß equation defined over k is

$$Y^2 = X^3 + aXZ^2 + bZ^3 \tag{3.9}$$

The point at infinity $\boldsymbol{O}$ corresponds to (0:1:0), and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$. The points of the line at infinity do not correspond to any of the affine points.

The equations governing PA and PD in projective coordinates using the *Double-and-Add* method are given in Algorithms 2 and 3 respectively. Each PA requires 12M, 2S and 7add. Each PD requires 8M, 5S and 13add.

Algorithm 2: Point Addition in Projective Coordinates

input : $P(X_1, Y_1, Z_1)$;
$Q(X_2, Y_2, Z_2) \in \mathbb{F}_q$
output: $P + Q(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$A = X_2Z_1 - X_1Z_2, B = Y_2Z_1 - Y_1Z_2,$
$C = B^2Z_1Z_2 - A^3 - 2A^2X_1Z_2,$
$X_3 = AC,$
$Y_3 = B(A^2X_1Z_2 - C) - A^3Y_1Z_2,$
$Z_3 = Z_1Z_2A^3$

Algorithm 3: Point Doubling in Projective Coordinates

input : $P(X_1, Y_1, Z_1) \in \mathbb{F}_q$
output: $[2]P(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$A = 3X_1^2 + a_4Z_1^2, B = Y_1Z_1,$
$C = X_1Y_1B, D = A^2 - 8C,$
$X_3 = -2BD,$
$Y_3 = A(4C - D) - 8Y_1B^2,$
$Z_3 = 8B^3$

The full breakdown for the Double-and-Add algorithms, directly used to generate the instruction set schedule for the elliptic curve processor described in Section 3.3 are described in Appendix A.1.

3.2.3 Jacobian Coordinate System

Another commonly used coordinate system is Jacobian coordinates. The point at infinity $\boldsymbol{O}$ corresponds to $(1 : 1 : 0)$, and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$. Conversion from Jacobian to affine coordinates is trivial and again it is sufficient to set the Z coordinate equal to one.

$$(x, y) \mapsto (X, Y, 1) \tag{3.10}$$

The cost of conversion back to affine from projective is four multiplications and two inversions.

$$(\frac{X}{Z^2}, \frac{Y}{Z^3}) \mapsto (x, y) \tag{3.11}$$

The Jacobian form of the short Weierstraß equation, Equation 3.8, defined over k is

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{3.12}$$

The equations governing PA and PD in Jacobian projective coordinates using the *Double-and-Add* method are given in Algorithms 4 and 5 respectively. Each PA requires 12M, 4S and 7add. Each PD requires 6M, 4S and 13add.

Algorithm 4: Point Addition in Jacobian Coordinates

input : $P(X_1, Y_1, Z_1)$;
$Q(X_2, Y_2, Z_2) \in \mathbb{F}_q$
output: $P + Q(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$A = X_1 Z_2^2, B = X_2 Z_1^2,$
$C = Y_1 Z_2^2, D = Y_2 Z_1^3,$
$E = B - A, F = D - C,$
$X_3 = -E^3 - 2AE^2 + F^2,$
$Y_3 = -CE^3 + F(AE^2 - X_3),$
$Z_3 = Z_1 Z_2 E$

Algorithm 5: Point Doubling in Jacobian Coordinates

input : $P(X_1, Y_1, Z_1) \in \mathbb{F}_q$
output: $[2]P(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$A = 4X_1 Y_1^2, B = 3X_1^2 + a_4 Z_1^4,$
$X_3 = -2A + B^2,$
$Y_3 = -8Y_1^4 + B(A - X_3),$
$Z_3 = 2Y_1 Z_1 E$

It can be seen from the standard projective algorithms, Algorithms 2 and 3, and the Jacobian projective algorithms, Algorithms 4 and 5, that PD in standard projective has more multiplications than Jacobian, but less for PA. Therefore, a key with a low Hamming weight (and therefore less point additions when compared to point doubles), would be better suited to Jacobian coordinates. However, the extra multiplications in the Jacobian PD, would somewhat offset this saving.

3.2.4 Twisted Edwards Curves

In 2007, Edwards [69] introduced an addition law on the curves

$$x^2 + y^2 = c^2(1 + x^2y^2), \quad \forall c \in k \tag{3.13}$$

where k is a field of characteristic not equal to 2. He showed that if k is algebraically closed, then every elliptic curve over k is isomorphic to a curve of this form. If k is finite this is not necessarily true, and in fact holds only in a very limited number of cases.

In [3], Bernstein and Lange generalised this addition law to the curves

$$x^2 + y^2 = 1 + dx^2y^2, \quad \forall d \in k \setminus \{0, 1\} \tag{3.14}$$

More generally, they considered

$$x^2 + y^2 = c^2(1 + dx^2y^2) \tag{3.15}$$

where $c, d \in k$ with $cd(1 - c^4.d) \neq 0$. However, any such curve is isomorphic to one of the form

$$x^2 + y^2 = 1 + d'x^2y^2 \text{ for some } d' \in k \tag{3.16}$$

so it is assumed that $c = 1$. These curves are referred to as Edwards curves.

Bernstein and Lange showed that if k is finite, a large class of elliptic curves over k (all those which have a point of order 4) can be represented in Edwards form. To describe the addition law in more general terms, recall from Section 2.4.1 that for standard operations on the points on any elliptic curve, such as adding, doubling or tripling points, usually, given two points P and Q on an elliptic curve, the point $P + Q$ is directly related to the third point of intersection between the curve and the line that passes through the two points. For Edwards curves this is not the case. The curve expressed in Edwards form has degree 4, so drawing a line provides not 3 but 4 points of

intersection on the curve.

In [70], Bernstein et al. introduced the twisted Edwards curves

$$ax^2 + y^2 = 1 + dx^2y^2 \tag{3.17}$$

where $a, d \in k$ are distinct and non-zero, and showed that every elliptic curve with a representation in Montgomery form [4] is birationally equivalent to a twisted Edwards curve. The projective form of twisted Edwards are used to again avoid inversion.

These curves are not entirely compliant with standard specifications as NIST [71] only gives one elliptic curve over a prime field for each size, i.e., the curves *P-256*, *P-384* and *P-521*. There are as yet no officially recommended curves in Edwards or twisted Edwards form from the standards bodies[ii]. However, these special form curves (also including Montgomery curves [4] and Hessian curves [73, 74]) have performance advantages over general elliptic curves [3] in relation to faster scalar multiplication and faster PA and PD.

Algorithm 6: Point Doubling in twisted Edwards Coordinates

input : $P(X_1, Y_1, Z_1) \in \mathbb{F}_q$
output : $[2]P(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$
Ensure: $(X_1 : Y_1 : Z_1)$ with $Z_1 \neq 0$ satisfy $aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$

$X_3 = 2X_1Y_1(2Z_1^2 - Y_1^2 - aX_1^2)$,
$Y_3 = (Y_1^2 - aX_1^2)(Y_1^2 + aX_1^2)$,
$Z_3 = (Y_1^2 + aX_1^2)(2Z_1^2 - Y_1^2 - aX_1^2)$.

Note that the projective twisted Edwards curve has two singular points, $(1 : 0 : 0)$ and $(0 : 1 : 0)$, and the addition law is not defined at these points. An implementation of Edwards or twisted Edwards curve - based cryptography should not allow either of these points as inputs. For standard projective twisted Edwards, each PA requires 11M and 8add as shown in Algorithm 6, while the PD requires 7M and 7add as shown in Algorithm 7. as given in Appendix A.2.

[ii] The curve parameters used in this work for twisted Edwards are taken from the literature, namely [66, 70, 72]

Algorithm 7: Point Addition in twisted Edwards Coordinates

input : $P(X_1, Y_1, Z_1)$;
$Q(X_2, Y_2, Z_2) \in \mathbb{F}_q$
output : $P + Q(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$
Ensure: $(X_2 : Y_2 : Z_2)$ with $Z_2 \neq 0$ and $T_2 = X_2Y_2/Z_2$ satisfy
$aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$

$X_3 = (X_1Y_2 - Y_1X_2)(X_1Y_1Z_2^2 + X_2Y_2Z_1^2)$,
$Y_3 = (Y_1Y_2 + aX_1X_2)(X_1Y_1Z_2^2 - X_2Y_2Z_1^2)$,
$Z_3 = Z_1Z_2(X_1Y_2 - Y_1X_2)(Y_1Y_2 + aX_1X_2)$.

3.2.5 Extended Twisted Edwards

In [72], Hisil presented homogeneous projective coordinate building on work presented in [70] on twisted Edwards curves. He examined homogeneous projective, inverted coordinate, extended homogeneous projective and mixed coordinate systems. Using this system each point is represented with four coordinates, (X, Y, T, Z) instead of the standard (X, Y, Z). This auxiliary coordinate T, results in a speedup over standard twisted Edwards curves and is defined as $T = XY/Z$. While this additional coordinate system results in an increase to the complexity of the point doubling formula, the cost of the point addition formula decreases and consequently the dedicated addition can be performed much faster. In this work, the extended homogeneous projective coordinates are examined.

Algorithm 8: Point Doubling in Extended twisted Edwards Coordinates

input : $P(X_1, Y_1, T1, Z_1) \in \mathbb{F}_q$
output : $[2]P(X_3 : Y_3 : T_3 : Z_3) \in E(\mathbb{F}_q)$
Ensure: $(X_1 : Y_1 : T_1 : Z_1)$ with $Z_1 \neq 0$ and $T_1 = X_1Y_1/Z_1$ satisfy
$aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$

$X_3 = 2X_1Y_1(2Z_1^2 - Y_1^2 - aX_1^2)$,
$Y_3 = (Y_1^2 - aX_1^2)(Y_1^2 + aX_1^2)$,
$T_3 = 2X_1Y_1(Y_1^2 - aX_1^2)$,
$Z_3 = (Y_1^2 + aX_1^2)(2Z_1^2 - Y_1^2 - aX_1^2)$.

The projective twisted Edwards, requires 9M and 7add for each PA, and requires 8M and 7add for each PD as given in Algorithm 8 and Algorithm 9. The full breakdown for the twisted and

Algorithm 9: Point Addition in Extended twisted Edwards Coordinates

input : $P(X_1, Y_1, T1, Z_1)$;
$Q(X_2, Y_2, T2, Z_2) \in \mathbb{F}_q$
output : $P + Q(X_3, Y_3, T3, Z_3) \in E(\mathbb{F}_q)$
Ensure: $(X_2 : Y_2 : T_2 : Z_2)$ with $Z_2 \neq 0$ and $T_2 = X_2Y_2/Z_2$ satisfy
$aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$

$X_3 = (X_1Y_2 - Y_1X_2)(T_1Z_2 + Z_1T_2)$,
$Y_3 = (Y_1Y_2 + aX_1X_2)(T_1Z_2 - Z_1T_2)$,
$T_3 = (T_1Z_2 + Z_1T_2)(T_1Z_2 + Z_1T_2)$,
$Z_3 = (X_1Y_2 - Y_1X_2)(Y_1Y_2 + aX_1X_2)$.

extended twisted Edwards algorithms used to generate the instruction set schedule are given in Appendix A.2.

3.2.6 Dedicated Algorithm Overview

As a synopsis of each of the *Double-and-Add* algorithms in different coordinate systems covered so far, along with the twisted Edwards algorithms, Table 3.1 gives the calculation cost for PA and PD, where A = Affine, P = Projective, J = Jacobian, tE = twisted Edwards and eE = extended twisted Edwards. The cost is given in multiplications and squarings in columns 2 and 5, and in multiplication only in columns 3 and 6.

Table 3.1: Operation Count for Double-and-Add

Point Addition			Point Double		
Coordinate	M and S	M only	Coordinate	M and S	M only
$A + A \mapsto A$	1I 2M 1S 6add	1I 3M 6add	$[2]A \mapsto A$	1I 2M 2S 4add	1I 4M 4add
$P + P \mapsto P$	12M 2S 7add	14M 7add	$[2]P \mapsto P$	8M 5S 13add	13M 13add
$J + J \mapsto J$	12M 4S 7add	16M 7add	$[2]J \mapsto J$	6M 4S 13add	10M 13add
$tE + tE \mapsto tE$	10M 1S 7add	11M 7add	$[2]tE \mapsto tE$	3M 4S 7add	7M 7add
$eE + eE \mapsto eE$	9M 7add	9M 7add	$[2]eE \mapsto eE$	4M4S 7add	8M 7add

It can be seen from Table 3.1 that the affine coordinates have relatively few multiplications in comparison to the others but both PA and PD involve a costly inversion. The projective *Double-and-Add* has less multiplications for the PA than the Jacobian, but more multiplications for the PD. By contrast, both the twisted Edwards and extended twisted Edwards require less multiplica-

tions than the projective and Jacobian, with the twisted Edwards requiring $2\mathsf{M}$ more for addition and $1\mathsf{M}$ less for doubling than the extended twisted Edwards. It should also be noted that both variants of Edwards require an additional multiplication by a constant, d, for PD. For PA, standard twisted Edwards requires two additional multiplications by constants, while the extended variant requires one. These constants can be set to low values to achieve additional speedup.

There are many more projective coordinate systems, such as LópezDahab, Chudnovsky, and indeed many other mixed coordinate systems. The interested reader is directed to [19] for further information on these curves.

3.3 Elliptic Curve Cryptographic Processor

Next, hardware was developed to implement the various algorithms. The elliptic curve cryptographic processor (ECP), shown in Figure 3.1, is of a similar design to the one used in [75]. It can be modified to perform any number of different algorithms in $\mathbb{F}_q$. It consists of control circuitry, BlockRAM for storage of results and a user defined number of computation units, namely field multipliers, squarers, inversions, adders and subtracters for the operations described in previously, all generated for FPGA using the Xilinx ISE 12.3 design suite.

This generic architecture is used in the thesis to perform baseline comparisons between each of the different elliptic curve algorithms and allows the development of a performance based ranking system describing the hierarchy of the different algorithms.

3.3.1 Control

The controller section consists of an finite state machine (FSM), an instruction set stored in ROM, the register for storage and manipulation of the key value and associated logic. The ROM block was generated with Xilinx BlockROM and contains the instructions necessary to run a particular algorithm, namely the address locations in RAM to be accessed and the particular computation units required for each particular control step. Using this method, only minor changes to the FSM

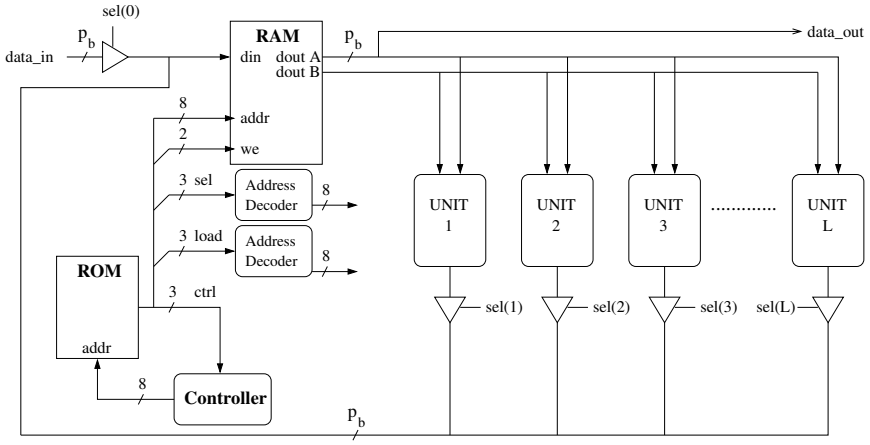


Figure 3.1: Elliptic Curve Processor

and the target algorithm instructions need to be changed to generate the ECP for any particular algorithm; thereby allowing for accurate comparison of the different implementations.

3.3.2 Modular Arithmetic

The adder and subtracter circuitry is generated using single clock carry propagate adders, and is defined as the computation of $S = A + B(\text{mod } p)$. For modular addition, the modular addition operation adds A and B in the first adder and subtracts the modulus p from the sum. To subtract the modulus from the intermediate result, the modulus is bitwise inverted and added to $(A + B)$ with the carry-in set to 1, thus performing a two's complement subtraction. The carry-out of the second adder controls which intermediate result is the correct result. If $(A + B)$ is in the correct range, the result of the first adder is the correct result. Otherwise, the result from the second adder is correct. For modular subtraction, B is bitwise inverted and added to A with the carry in set to 1. If the carry-out of this adder is low, the modulus p is added to give an output in the correct range. The architecture is shown in Figure 3.2, and the circuit performs an addition or subtraction in four clock cycles, comprising two clocks for the additions and two clocks for move operations from and to the RAM.

These arithmetic *units* are connected to the ECP as shown in Figure 3.1. The instruction set stored

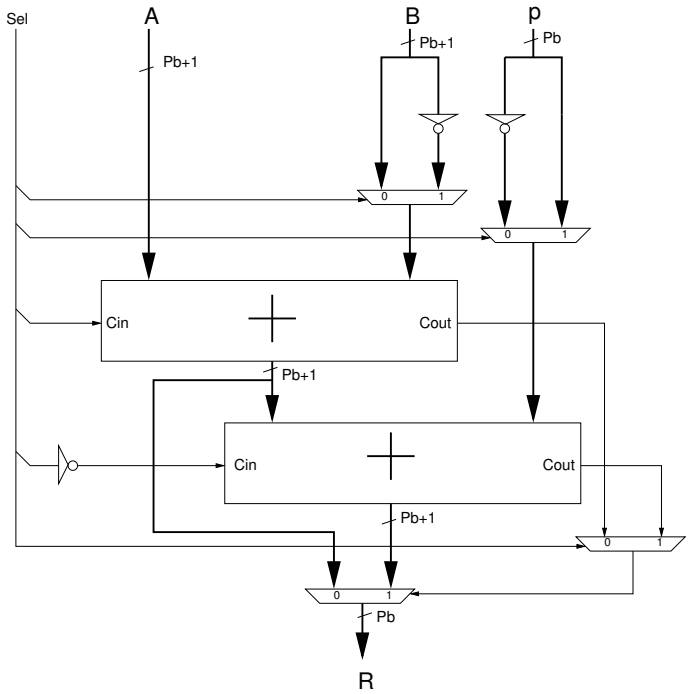


Figure 3.2: Modular Adder-Subtracter

in ROM selects the address locations in the RAM along with any arithmetic units required to perform the calculations for a given step. When the calculations are completed, the result is stored back in RAM. It is clear from the diagram that any number of these units can be added in parallel to the circuit, at a cost of additional area and a larger ROM address bus to allow selection between them.

3.3.3 Modular Multiplication

For modular multiplication, again designed using carry propagate adders and following the process described in the Montgomery multiplication algorithm [76], the binary number can be computed while avoiding the need to perform a division by the modulus.

The result of a Montgomery multiplication is out by a factor of 2^{-p_b+2}, where p_b is the field size in bits. Due to the large number of multiplications required for calculation by the elliptic curve processor, it is more cost effective to initially convert all values to the Montgomery domain. At

Algorithm 10: Montgomery Multiplication

input : $A = \sum_{i=0}^{p_b} a_i 2^i; B = \sum_{i=0}^{p_b} b_i 2^i; M = \sum_{i=0}^{p_b} p_i 2^i$
output: $R = AB2^{-p_b+2} (\text{mod } p)$

Initialise: $R \leftarrow 0; b_{p_b} + 1 \leftarrow 0$
for $i \leftarrow 0$ *to* $p_b + 1$ **do**
  $q_i = R_{i-1} + b_i A (mod\, p)$
  $R_i = (R_{i-1} + q_i M + b_i A)/2$
end

the beginning of a point scalar multiplication, the input points are converted to the Montgomery domain and all subsequent arithmetic operations are carried out in this domain. The final result of the point scalar multiplication is then converted back to the integer domain. The number is Montgomery multiplied by $2^{2p_b+2}(mod\ p)$ for conversion to the Montgomery domain. For conversion from the Montgomery domain, the number is Montgomery multiplied by 1. The Montgomery modular product is defined as:

$$R = Mont(A, B, p) = AB2^{-p_b+2} (\text{ mod } p) \tag{3.18}$$

where $Mont$ is a Montgomery multiplication. The original proposal of Montgomery has a conditional subtraction at the end of the algorithm. In this work, the number of iterations performed is $p_b + 2$ in order to bound the output in the range $[0, 2p - 1]$ for multiplicands up to twice the modulus [77]. This allows it to be used as an input to further multiplications without the need for the conditional subtraction.

The Architecture is shown in Figure 3.3 and Montgomery multiplication is performed according to Algorithm 10. The inputs to the first adder are $B_i A$ and the previous result R_{i-1}. $q_i p$ is added to the sum of the first adder if the LSB of the sum, q_i, is equal to 1. A shift register scans each bit of B for $B_i A$ and the final result is right shift divided by 2. The circuit performs a multiplication in $p_b + 2$ cycles, inclusive of move operations.

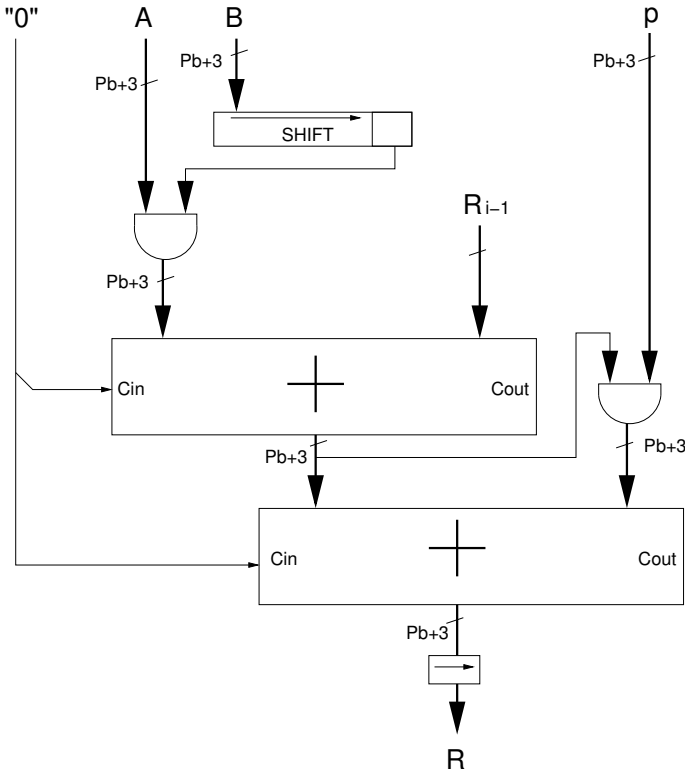


Figure 3.3: Modular Multiplier

3.3.4 Modular Inversion

The Extended Euclidean Algorithm (EEA) forms the basis for computing inversion using the Montgomery inverse [78]. The Montgomery modular inverse of an integer $A \in [1, m-1]$ is given by

$$R = A^{-1}2^k \pmod{M}$$

where k is the bit length of the modulus. The methods presented by Crowe [78] and Kaliski [79] of breaking the algorithm into two distinct phases, as shown in Algorithms 11 and 12 are used in this work.

For Phase 1:

- The While loop performs the EEA.

- For each iteration of the EEA:
  - The $\frac{U}{2}$ and $\frac{V}{2}$ variables are compared, and subtracted dependent on which value is greater.
  - A *switch* variable is used to determine whether the variables RS_a or RS_b should be doubled in each operation.
- Continue until $V = 0$.
- The final steps perform a modular correction to bring the output into the correct range $[1, M-1]$.
  - RS_b is subtracted from RS_a, performing a conditional subtraction of the modulus.
  - M is added to the result if a negation of the result is necessary.
- Following this, the output should be $R = -A^{-1}2^t$, where $R \in [1, 2M-1]$ and t is the number of iterations performed.
- t is dependent on A and M, and is in the range $[k, 2k]$.

For Phase 2:

- The output from phase 1 is doubled $2k - t$ times.
- This gives a final output $R = A^{-1}2^k \ (mod\ M)$.

The total amount of clock cycles for phase 1 is $t + 2$ and the total amount of clocks for phase 1 and 2 together is $2t - k + 2$. Again, similar to Montgomery multiplication, it is more practical to leave the result in the Montgomery domain and convert back to the integer domain at the end of the calculation.

3.3.5 Scheduling and Efficiency

The schedule, defines the time-steps during which operations are to be executed. As such it directly controls the throughput, and therefore the speed of an algorithms calculation. Similarly,

Algorithm 11: Montgomery Inverse (Phase 1)

input : $A \in [1, M-1], M, gcd(A, M) = 1$
output: $R = A^{-1}2^t \ (mod\ M), k \leq t \leq 2k$

$U \Leftarrow M, V \Leftarrow A, RS_a \Leftarrow 1, RS_b \Leftarrow 0, t \Leftarrow 0, switch \Leftarrow 0;$
while $(V > 0)$ **do**
  if $(U_0 = 0)$ **then**
    $U \Leftarrow \frac{U}{2} - 0;$
    if $(switch = 0)$ **then**
      $RS_b \Leftarrow 2(RS_a);$
    else
      $RS_b \Leftarrow 2(RS_b); switch \Leftarrow 1;$
    end
  else if $(V_0 = 0)$ **then**
    $V \Leftarrow \frac{V}{2} - 0;$
    if $(switch = 0)$ **then**
      $RS_b \Leftarrow 2(RS_b);$
    else
      $RS_b \Leftarrow 2(RS_a); switch \Leftarrow 0;$
    end
  else if $(\frac{U}{2} > \frac{V}{2})$ **then**
    $U \Leftarrow \frac{U}{2} - \frac{V}{2};$
    if $(switch = 0)$ **then**
      $RS_b \Leftarrow 2(RS_a);$
    else
      $RS_b \Leftarrow 2(RS_a); RS_a \Leftarrow RS_a + RS_b; switch \Leftarrow 1;$
    end
  else if $(\frac{V}{2} \geq \frac{U}{2})$ **then**
    $V \Leftarrow \frac{V}{2} - \frac{U}{2};$
    if $(switch = 0)$ **then**
      $RS_b \Leftarrow 2(RS_b);$
    else
      $RS_b \Leftarrow 2(RS_a); RS_a \Leftarrow RS_a + RS_b; switch \Leftarrow 0;$
    end
  $t \Leftarrow t + 1$
end
$RS_a \Leftarrow RS_a - RS_b;$
if $(RS_a < 0)$ **then**
  $RS_a \Leftarrow RS_a + M;$
else
  $R \Leftarrow RS_a;$
end

Algorithm 12: Montgomery Inverse (Phase 2)

input : R and t from Phase 1
output: $R = A^{-1}2^k \pmod{M}$

$RS_a \Leftarrow R$;
for $i = 0$ *to* $2k - t$ **do**
  $RS_a \Leftarrow R$;
  if *(*$RS_a \geq M$*)* **then**
    $RS_a \Leftarrow RS_a - M$;
end

by examining the scheduling of multipliers working in parallel, the area-speed tradeoff can be optimised, and the best-fit number of multipliers for each particular algorithm can be selected.

A list-based scheduling (LBS) technique [80] was adopted here. In list scheduling, the number of functional units of each type are constrained and each control step, processed sequentially, attempts to choose the optimum operation to perform, subject to resource constraints. The LBS maintains a list of operations whose predecessors have already been scheduled. At each time step of the algorithm, operations are scheduled to the arithmetic units until the available resources are exhausted. Once an operation, or set of operations have been scheduled at a given time step, the list is updated. The process continues until all operations are scheduled. It can be clearly seen from Table 3.2, that a schedule which allows a number of multipliers to operate in parallel can reduce the number of multiplication stages required, thereby improving the speed of the algorithms execution.

Although the ECP can be configured to run any number of multipliers in parallel, some operations in a particular formula are dependent on the results of other operations, which creates a limit to the amount of parallelism that can be exploited. This leads to redundancy in the design, as a point is reached where the addition of extra multipliers leads to a decrease in the efficiency, and results in a small increase in speed at the cost of a large increase in area. The efficiency is defined as the number of multiplication operations that can be run in parallel at each particular time stage, in relation to the overall number of multipliers available for parallel processing for a time stage. Parallel addition or subtraction results in a saving of only 4 clock cycles, therefore making

Table 3.2: Multiplier Efficiency for Dedicated Doubling and Addition

Algorithm	no. of mults	no. of Mult. Stages	no. of Inv. Stages	Eff. %	no. of Mult. Stages	no. of Inv. Stages	Eff. %
		Point Doubling			Point Addition		
Affine	1	4	1	100	3	1	100
Double	2	4	1	50	3	1	50
and	3	4	1	33	3	1	33
Add	4	4	1	25	3	1	25
Projective	1	13	0	100	14	0	100
Double	2	7	0	92	7	0	100
and	3	5	0	86	6	0	77
Add	4	4	0	81	4	0	87
Jacobian	1	10	0	100	16	0	100
Double	2	6	0	83	8	0	100
and	3	5	0	66	6	0	88
Add	4	5	0	50	5	0	80
Dedicated	1	8	0	100	13	0	100
Twisted	2	4	0	100	7	0	92
Edwards	3	3	0	88	5	0	86
	4	3	0	66	5	0	65
Dedicated	1	9	0	100	10	0	100
Extended	2	5	0	90	5	0	100
Twisted	3	4	0	75	4	0	83
Edwards	4	3	0	75	3	0	83

the parallelisation of additions or subtractions not area-speed cost effective. As such, referring back to Figure 3.1, the arithmetic *units* would normally comprise single addition, subtraction and inversersion blocks and multiple multiplier blocks.

As stated in Section 3.3.1, the minimum number of BRAM required for any $p_b = 192$ is identical for any number of addresses between 2 and 1024. Not being constrained by a finite amount of register/RAM address locations, the scheduler was allowed as many addresses as necessary to suit the best schedule. This was between 16 and 25 for each of the algorithms under investigation here.

It can be seen from Table 3.2, using the Jacobian Double-and-Add algorithm as an example, that when there is only $1M$ operating for all 10 Multiplication stages in PD, the efficiency is at 100%, i.e. always in use. Similarly for PA the 1M is used for all 16 multiplication stages. Two multipliers

in parallel operate at 83%, so the number of Multiplication stages is reduced to 6 for PD. For PA, both multipliers are used 100% of the time, halving the number of stages to 8. For 3M, the efficiency begins to decrease with the number of stages for PD only reducing by one to 5 and for PA only reducing by two to 6. For 4M, there is no reduction for PD and only one multiplication stage is reduced for PA with the efficiencies dropping to 50% and 80% for PD and PA respectively.

It is clear from Table 3.2 that the Double-and-Add in affine coordinates cannot be speeded up any further by the addition of extra multipliers. It is also seen that as multipliers in parallel are added to the Jacobian and projective coordinates, the efficiency of any more than three multipliers in parallel does not result in significant speedup in the designs, and in some cases there is a general reduction in efficiencies for four multipliers.

The efficiency of the parallelisation of dedicated extended twisted Edwards can be seen from the table. With the standard $1M$, the dedicated PD twisted Edwards has one less multiplication stage than the extended twisted Edwards, but three extra multiplication stages for PA. An increase in parallelisations leads to an increased performance in both, resulting in the extended twisted Edwards having, at $3M$, a similar performance to the standard twisted Edwards. However at 4M, it has three multiplication stages for a PD and a PA compared to standard twisted Edwards which while also having three multiplication stages for a PD has five for a PA.

From this point on, the worst performing algorithms deduced from Table 3.2 can be excluded since clearly they will not perform adequately when implemented. These would include all of the parallel affine implementations with performances of 50% or less. The Jacobian for 4M also has relatively poor efficiency for PD but is retained due to adequate performance for PA.

3.3.6 Algorithmic Cost of Field Operations

It was stated in Section 3.2 that field addition and subtraction costs are generally ignored as being negligable in comparison to multiplications or inversions. This is in general true in a one-to-one conversion, i.e. 4 clock cycles for an add versus $p_b + 2$ clock cycles for a multiplication. However,

Table 3.3: Clock cycle count for Dedicated Doubling and Addition

Algorithm	Hardware Multipliers	no. of Mult. clks	no. of Inv. clks	no. of other clks	Total clks	no. of Mult. clks	no. of Inv. clks	no. of other clks	Total clks
		Point Doubling				Point Addition			
Affine	1	776	391	72	1232	582	391	55	1021
Projective	1	2522	0	175	2697	2716	0	160	2879
Double	2	1358	0	157	1515	1358	0	139	1497
and	3	970	0	151	1121	1164	0	136	1300
Add	4	776	0	148	924	776	0	130	906
Jacobian	1	1940	0	148	2088	3104	0	178	3282
Double	2	1164	0	136	1300	1552	0	154	1706
and	3	970	0	133	1103	1164	0	148	1312
Add	4	970	0	133	1103	970	0	145	1152
Dedicated	1	1552	0	106	1658	2522	0	151	2673
Twisted	2	776	0	94	870	1358	0	133	1491
Edwards	3	582	0	91	673	970	0	127	1097
	4	582	0	91	673	970	0	127	1097
Dedicated	1	1746	0	117	1863	1940	0	126	2066
Extended	2	970	0	105	1075	970	0	111	1081
Twisted	3	776	0	102	878	776	0	108	884
Edwards	4	582	0	99	681	582	0	105	687

as shown in [68], these operations are not insignificant for hardware devices, especially when taken alongside other relatively small (but numerous) operations, e.g. move operations. Table 3.3 gives the full cost in clock cycles for each algorithm, subdivided into the cost of the multiplications and the cost of the other operations using the curve parameters secp192r1 [81]. This is also shown graphically in Figure 3.4 for an equal Hamming weight message. The dotted lines signify the M and I calculations only while the solid lines represent the full number of clock cycles required inclusive of additions, subtractions and move operations ($\times 1000$).

While for the single multiplier, adder, subtracter circuit, the ratio of multiplier clocks to other clocks is on average approximately $14:1$, it is also clear that as the number of multiplier clocks reduce due to the scheduling of additional multipliers, the impact of the other clocks attain much more significance. This can be clearly seen from Figure 3.4. For example, in the case of Jacobian for four multipliers, the ratio is approximately $7:1$, thereby giving a big impact on the overall speed of the algorithm. The results here reflect the previous table where the total final clock count for the extended twisted Edwards with $4M$ is approximately 685 for a PA or a PD. The standard twisted Edwards PD has an additional cost of approximately 400 extra clock cycles per point addition operation than the extended twisted Edwards using the auxiliary coordinate T.

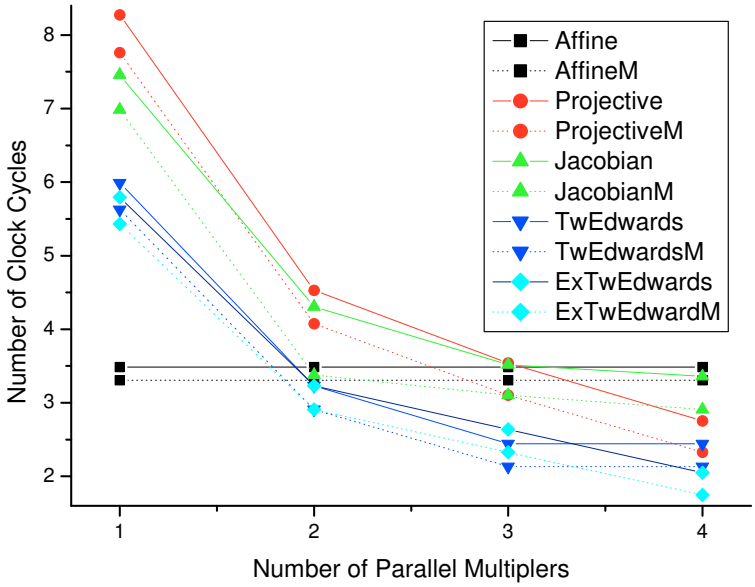


Figure 3.4: Clock Count for Dedicated Doubling and Addition

Again, the data presented in Table 3.3 and Figure 3.4 can be used to exclude a number of designs from further study by examining the performance of the clock cycle count, and predicting how the final set of results should be expected to perform relative to each other. The full examination of the Jacobian clock count shows that the performance increase from 3M to 4M is negligible and so can be excluded. It can also be seen that an increase from 3M to 4M results in no decrease in clock cycles for the standard twisted Edwards and so while it would appear to outperform most of the other algorithms this 4M variant can also be excluded as it underperforms in relation to its 3M counterpart.

3.3.7 Area Results for Dedicated Doubling and Addition

Next, the area of the ECC algorithms was examined for a key and field size of 192 bits. For cases where average results were to be measured, randomly generated keys each of Hamming weight of $k/2$ were used. The number of Occupied Slices is the complete and full measurement of the

FPGA area in Slices. The Slice Registers and Slice LUTs are secondary area measurements used simply to give a deeper breakdown of the area. The main differences in circuitry between the different algorithms is in the control section. As such, there is very minimal divergence in area measurements, mainly due to how the compiler performed the routing (with the only change being in the instruction set). Obviously, the area increases as more multipliers are added. For affine coordinates, only the $1M$ case is examined as the addition of extra multipliers does not result in any speedup as previously shown. The area here is also higher due to the additional inverter circuitry. Similarly, the Jacobian and standard twisted Edwards for 4M have been omitted.

The Block RAM, used to generate the RAM and ROM blocks is measured seperately. In all cases the instruction set BROM requires a single $18K$ block, while the BRAM address block require five $36K$ blocks and a single $18K$ block.

Table 3.4: Dedicated Doubling and Addition Area Results

Mult. Units	Algorithm	Occupied Slices	Slice Reg	Slice LUT	Algorithm	Occupied Slices	Slice Reg	Slice LUT
1	Affine D&A	1470	3605	5416	-	-	-	-
1	Projective	965	2611	3593	Dedicated	955	2610	3600
2	Double	1551	3195	4565	Extended	1470	3195	4577
3	and	1643	3779	5158	Twisted	1672	3779	5175
4	Add	2012	4364	6128	Edwards	1800	4364	6139
1	Jacobian	973	2609	3618	Dedicated	1167	2609	3598
2	Double	1449	3195	4563	Twisted	1470	3195	4577
3	Add	1643	3779	5158	Edwards	1672	3779	5175

3.4 Measuring the Power Dissipation

The aim of this section was to measure the power and energy consumption of the architecture described in Section 3.2 when implementing the various different algorithm and coordinate combinations. The target implementation platform was the SASEBO-GII cryptographic evaluation board[iii] and results were measured using randomly generated keys each of Hamming weight of

[iii] Although the SASEBO-GII using the Xilinx Virtex-5 was used to perform the testing, the results obtained throughout should be equivalent for any different FPGA device.

$k/2$ to measure iterations of each of the algorithms from post initial data load to algorithm complete. The onboard Sasebo 24Mhz clock was used and an external power supply was directly connected for the core voltage of the cryptographic FPGA via an external power meter, namely the Agilent N6705A. Results were taken from both this meter and a Lecroy Waverunner 104Xi oscilliscope, measured across the Sasebo's 1 ohm shunt resistor on the V_{core} line and were generated as follows:

$$\begin{aligned} resistor &= 1; \\ meanResistorVoltageDrop &= mean(trace); \\ calculatedCurrent &= meanResistorVoltageDrop/resistor; \\ resistorPower &= calculatedCurrent^2 \times resistor; \\ meanFpgaVoltage &= suppliedVoltage - meanResistorVoltageDrop; \\ meanFpgaPower &= meanFpgaVoltage \times calculatedCurrent; \\ checkFpgaPower &= meanFpgaPower + resistorPower \\ energy &= sum((time(end) - time(1)) \times meanFpgaPower) \end{aligned} \tag{3.19}$$

In order to obtain a realistic measurement of the power consumption, the circuit must be provided with appropriate inputs. An IO interface between a host computer and the Sasebo board requires additional control circuitry. In order to minimise testing time, a test wrapper was created for the EC processor in place of an IO interface. A diagram of this wrapper is shown in Figure 3.5

The test wrapper, implemented on the Virtex-5 cryptographic FPGA on the Sasebo, consists of a ROM, counter and FSM. The ROM contains the inputs to the processor i.e the input point $P = (x, y)$, the curve parameters and the scalar k. The input point and curve parameters need only be read into the processor once. Then the FSM will load in a scalar k and start the processor. When the processor has completed the calculation it will load in another scalar and repeat the process.

The wrapper FSM continues on a loop, performing point scalar multiplications with each value of k. There are two input pins to the wrapper. The reset pin allows the wrapper and EC processor

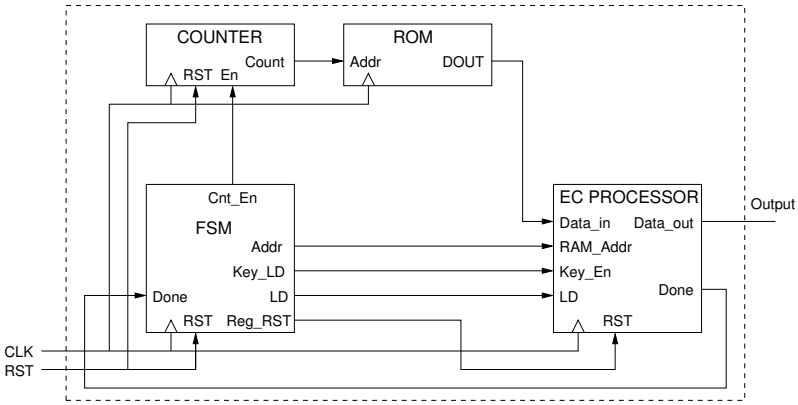


Figure 3.5: Power Wrapper

to be reset. It is connected to a switch on the board. The clock pin allows an external oscillator to clock the design (however, the results presented here are for the onboard Sasebo 24Mhz clock). The start and end trigger signal is routed to an output pin. This is used to trigger the oscilloscope to capture the current waveform as the EC processor performs a calculation. The cost of the wrapper in terms of FPGA resources is only two BRAM and a few hundred slices. This cost is low in comparison to the cost of the EC processor. It is also noted that the wrapper overhead is constant for each of the algorithm and coordinate choices and therefore will not bias any particular set of choices. With the wrapper and the EC processor implemented on the cryptographic FPGA, the current drawn can be measured via a shunt resistor. This allowed the average current to be measured. This is the current of interest when considering energy consumption.

It is noted that the very act of measurement will alter the current drawn by the FPGA due to the voltage drop across the Sasebo's 1 ohm shunt resistor on the V_{core} line which is connected in series. However, experimental measurement found that this voltage drop was very small and that for all the designs tested the voltages supplied to the FPGA remained within the recommended operating levels as specified in [82].

3.4.1 Dedicated Doubling and Addition Power Results

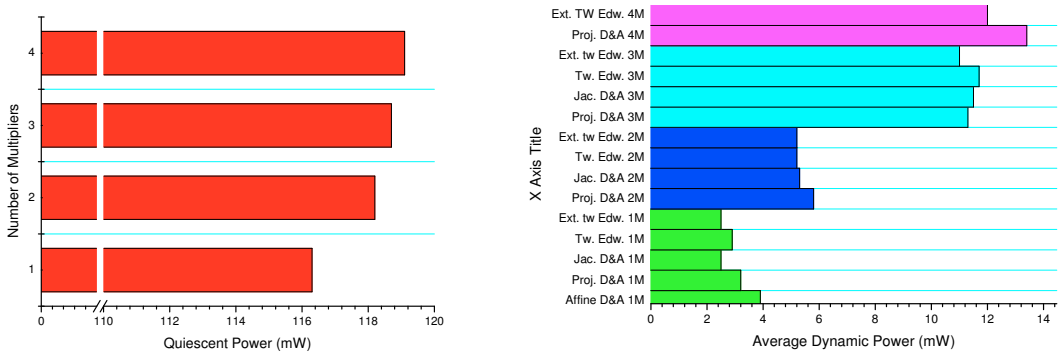
The results generated are given in Table 3.5. Columns 3 gives the reading from the power meter. Both the current and the power supplied use the same values due to the voltage being $1V$ and $P = I \times V$. Subsequent columns give the results calculated using Equation 3.19. Column 4 gives the mean power (μ), while the final two columns present the time taken to perform a full calculation of the algorithm and the energy expended to perform that calculation. In both cases, the lower the value, the better the result, with the best results in each category highlighted in bold.

Table 3.5: FPGA Power and Timing Results for Double-and-Add

Algorithm	Mult. Units	Supplied Current-Power mA - mW	μ FPGA Power mW	Calc. Time mS	Energy mJ
Affine D&A	1	151.9	120.2	**13.9**	**1.7**
Projective	1	150.0	119.5	32.6	3.9
Double	2	157.2	124.0	17.8	2.2
and	3	164.8	130.0	13.9	1.8
Add	4	171.3	132.4	10.8	1.4
Jacobian	1	148.9	118.8	25.4	3.0
Double	2	156.3	123.5	16.9	2.1
Add	3	164.5	130.2	13.8	1.8
Twisted	1	150.7	119.2	23.6	2.8
Edwards	2	157.1	123.4	**12.7**	**1.6**
Dedicated	3	165.6	130.4	**9.6**	**1.2**
Extended	1	148.2	118.8	22.8	2.7
Twisted	2	156.2	123.4	12.8	1.7
Edwards	3	164.6	129.7	10.4	1.3
Dedicated	4	167.5	131.1	**8.0**	**1.1**

Immediately obvious from the table is that the mean power is similar, regardless of the algorithm, for each of the four multiplier units. However the energy differs betweem them due to the computation time difference. Indeed it is also clear from the results that there is a strong correlation between the calculation time and the energy. As such, either metric could be used to categorise the algorithms. It can also be seen from the table that the affine coordinates give quite good timing and energy results when compared to the other coordinate systems for $1\mathsf{M}$ for a key with an

average Hamming weight. However, as it does not scale with parallelisation, it is overtaken in both the calculation time and energy usage by the other coordinate systems by additional parallel multipliers. The Jacobian coordinates initially are faster for 1M with more energy efficiency than the projective coordinates, but the latter makes better use of parallelisation to overtake the former in timing, albeit at a greater cost in energy. The standard twisted Edwards perform best at 2M and 3M, but the extended twisted Edwards give the best performance at 4M and indeed the best performance overall.



(a) Quiescent Power

(b) Dynamic Power

Figure 3.6: Average Power Dissipation

Figures 3.6(a) and 3.6(b) show the average power dissipation for the results, giving the quiescent power and the dynamic power. The quiescent power is the power that the board is drawing when the design is programmed and the clock is connected but reset is held active, thereby preventing any switching from occurring in the circuit. It is a measure of the standby power drawn by the FPGA. The larger the amount of logic resources used, the higher the quiescent power. By contrast, the dynamic power is the power dissipated by logic switching within the FPGA. It is calculated as the difference between the total average power and the quiescent power. The energy results presented in Table 3.5 are calculated using the total power.

Immediately striking from the graphs is the large quiescent power compared to the dynamic power. This is mainy due to the fact that the quiescent power encompasses all of the components on

the Sasebo board, while the dynamic power increase is due only to the operation of the Virtex 5 cryptographic FPGA. Also noticeable is that while the dynamic power is small for a single multiplier, it increases as additional multipliers are added. This is due to the fact that in each architecture, the critical path is through the long carry propagate adders (CPA) as described in Section 3.3.2. These adders are implemented on FPGA using long carry chains in the CLBs organised in columns. The ripple effect of adding two inputs (and a carry) with these adders creates a large amount of switching activity. It is also due in part to the additional switching due to the larger circuit area.

As seen in Figure 3.6(b), the average dynamic power differs between the different algorithms. This is due in part to both the multiplier efficiency as shown in Table 3.2, i.e. how often the multipliers are operational, and the multiplier usage as shown in Table 3.3, i.e. how long the multipliers are operational. Examining the designs individually shows that the affine double and add has a large average power due to the inverter, the projective version has the next highest average dynamic power for 1, 2 and 4M. The reduction at 3M is mostly due to the drop off in efficiency for three multipliers. The Jacobian variant follows a similar pattern to the the projective, with a slightly lower power in most cases. The standard twisted Edwards give the highest average power at 3M due to the relative high efficiency at this setup as seen by the calculation time in Table 3.5. While the extended twisted Edwards efficiency is high throughout, its multiplier usage is also quite low, thereby resulting in the lowest average power throughout.

3.4.2 Area-Time and Area Energy Product

It can be seen from Figure 3.6(b) and Table 3.5 that a high average power does not necessarily result in a high energy result. Indeed, with three multipliers, the twisted Edwards results in the highest average power usage but the lowest energy usage due to its shorter calculation time relative to the other designs. As a method of further comparing the designs examined so far, the metrics from Section 2.10 were used. The area-time product (ATP) was calculated to get a representation of any speed decrease relative to the increase in size. This gives a more accurate representation

of the cost that each increase in multiplier has in relation to the overall system. The area-energy product (AEP) was calculated using the dynamic power to give a representation of the energy increase against the increase in size. The power naturally increases with an increase in area, however the energy costs are reduced due to the calculation being performed faster. This therefore shows the best increase in area for a decrease in energy. In both cases, the lower the value, the better the performance. These are shown graphically in Figure 3.7 and Figure 3.8.

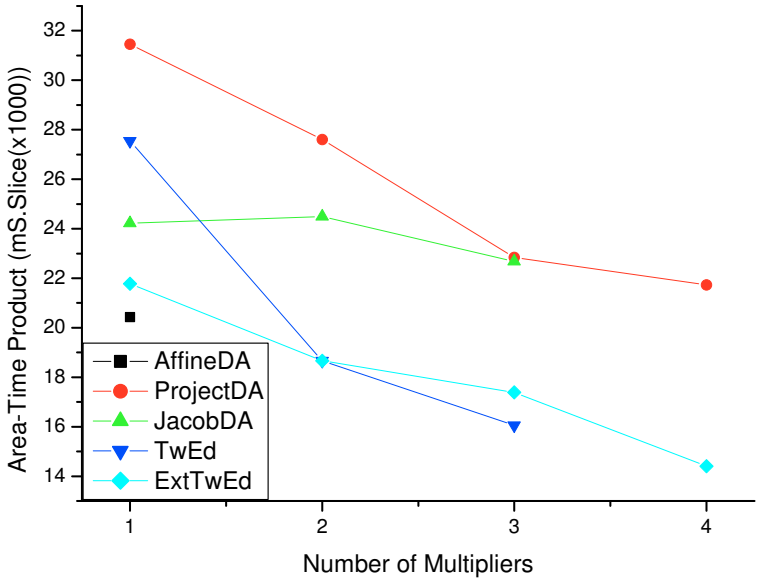


Figure 3.7: Area-Time Product

A number of things can be seen from Figure 3.7. Firstly, the Affine Double-and-Add gives the best ATP for a single multiplier. This outperforms the Projective and Jacobian Double-and-Add algorithms, however it must be noted that this result is determined on the key value, and a different key could cause a large increase in the timing cost for the inversion. In contrast, the two twisted Edwards algorithms outperform the Affine coordinates from 2M onward and subsequently greatly outperform the Projective and Jacobian Double-and-Add algorithms. They have the same result for 2M, while the standard twisted Edwards performing best for 3M and the extended twisted

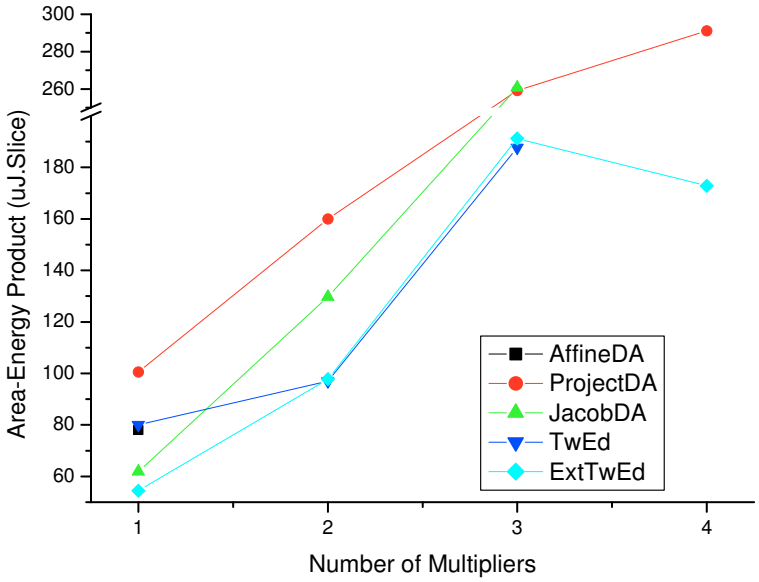


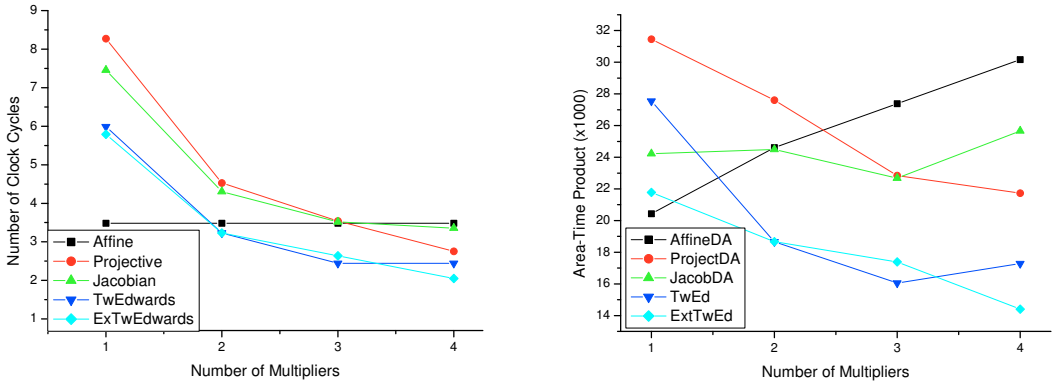
Figure 3.8: Area-Energy Product

Edwards performing best (and best overall) at 4M.

Figure 3.8 gives the AEP. It is seen that there is a general increase in the AEP for each additional multiplier for each of the algorithms. The extended twisted Edwards and the Jacobian perform best with a single multiplier followed closely by the affine coordinates and the standard twisted Edwards. From 2M onwards, both twisted Edwards have much better performance when compared to the Projective and Jacobian forms of the Double-and-Add algorithm and at 4M the extended twisted Edwards actually shows a slight reduction in the AEP compared to its 3M equivalent.

Figure 3.9 provide a comparison between what was expected, with Figure 3.9(a) giving a reduced graph of the clock cycle count presented in Figure 3.4, and Figure 3.9(b) giving the full set of measured results from Figure 3.7. It can be seen in this case that the measured results do somewhat follow the same trend as the expected results, with each algorithm in the same hierarchy from 2M onwards, after allowing for the affine increase in area with no reduction in time. For 1M, the Jacobian performs better than the standard Edwards due to the larger area associated with the

latter.



(a) Expected Results

(b) Measured Results

Figure 3.9: Estimated Dynamic Versus Measured Results

In conclusion, it has been shown that the basic elliptic curve processor design with a single multiplier gives good performance using the affine coordinates. Even though affine gives a larger average power and area due to the inverter circuitry, its computation time is sufficiently fast compared to the other double and add algorithms that it outperforms the others in area-time, and the other Double-and-Add algorithms in area-energy. For multipliers in parallel, the two Edwards special form curves outperform the standard curves, and the speed advantages over general elliptic curves in relation to faster scalar multiplication are clear to see from the area-time and area-energy products. However, all of these algorithms are susceptible to power analysis attacks as is described in Section 3.5.

3.5 Power Analysis Attacks

It was shown in Section 2.9.1 that simple and differential power analysis (SPA and DPA) [59] attacks measure leaked power consumption of a cryptographic device. From this leaked data, an attacker tries to deduce the inner workings of the algorithm, and in so doing, deduce the hidden information. SPA works on the premise that the amount of power drawn from a circuit is dependent

(indirectly) on the operation being performed. As such, various data can be gleaned, such as the Hamming weight, individual bits or bit sequences and memory addressing. Avanzi states in [83] that *observing a single trace of leaked emissions, one can reconstruct the sequence of elementary instructions performed by the processor, and thus infer the sequence of group operations. In an elliptic or hyperelliptic curve implementation with distinguishable group addition and doubling, and with a simple double-and-add scalar multiplication scheme, it is possible to reconstruct the secret scalar just from the observed sequence of distinct group operations*.

Section 3.2 described dedicated doubling and addition formulæ standard algorithms for performing scalar multiplication, whereby the binary algorithm scans the bits and performs a *point doubling* operation for every bit. If a bit is a "1", it also performs a *point addition* operation. These algorithms, while being simple and efficient, are susceptible to power analysis attacks as defined above since the power trace for a point doubling is different from that of a point addition.

To reduce this side channel attack (SCA), some methods of making the observable information independent of the secret scalar are examined. The following sections describe some of these methods, namely dummy arithmetic instructions, unified formulas and regular algorithms and again attempts to find the best performance algorithm when implemented in hardware.

3.6 Dummy Arithmetic Instructions

One such method of ensuring SPA resistance is the *Double-and-Add-Always* [84] algorithm, given in Algorithm 13, where a dummy point operation is completed for every "0" bit, thus making a *regular* succession of *point doublings* followed by *point additions*. This dummy arithmetic operation method is universal and works for all groups, making them homogenous.

However, while securing against SPA, an increase in calculation time and hardware resources is also introduced. In certain situations, such as elliptic curves in affine coordinates, where the PA and PD operations are quite similar, this method is quite attractive as the cost overhead of the dummy operations is minimal. For other coordinate systems where PA and PD differ greatly, it

Algorithm 13: Double-and-Add-Always

input : $p \in E(\mathbb{F}_q); k = \sum_{i=0}^{n_k-1} k_i 2^i$
output: $Q = [k]p \in E(\mathbb{F}_q)$

Initialise: $Q = p$
for $i \leftarrow n_k - 2$ *to* 0 **do**
    $Q[0] = 2Q[0]$
    $Q[1] = Q[0] + p$
    $Q[0] = Q[k_i]$
end

can be a quite expensive protective measure. Also, it has been proven that dummy point operations can be insecure against fault attacks [85,86], i.e. introducing faults into a device in order to produce an erroneous output.

3.7 Unified Doubling and Addition

The unified approach, suitable for elliptic curves, involves using the same set of formulas in the scalar multiplications for both point addition and point multiplication. The main benefit of this type of operation is that there are no dummy operations, thus making them secure against fault attacks. There can however be a reduction in speed, compared to the standard algorithms, due to the nature of the unified formulas. As such, certain representations of the Weierstraß elliptic curve can perform much faster than others, and other non-Weierstraß curves may not profit at all [60].

Along with an increase in calculation speed, an additional advantage of Edwards and twisted Edwards curves is that the addition laws defined on them can be made unified, i.e., a single addition formula can be used to add points and double points, with no exception for the identity. [3]. The unified versions of both the standard twisted Edwards and the extended twisted Edwards are examined, and compared against the standard curves using dummy arithmetic and regular scalar formulæ. Algorithm 14 give the unified formula for standard twisted Edwards, while Algorithm 15 for extended twisted Edwards.

The unified single point operation, processes the same formula for both PA and PD, thereby giving

Algorithm 14: Unified Addition in twisted Edwards Coordinates

input : $P(X_1, Y_1, Z_1)$;
$Q(X_2, Y_2, Z_2) \in \mathbb{F}_q$
output: $P + Q(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$X_3 = Z_1 Z_2 (X_1 Y_2 + X_2 Y_1)(Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2)$,
$Y_3 = Z_1 Z_2 (Y_1 Y_2 - a X_1 X_2)(Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2)$,
$Z_3 = (Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2)(Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2)$.

Algorithm 15: Unified Addition in Extended twisted Edwards Coordinates

input : $P(X_1, Y_1, T1, Z_1)$;
$Q(X_2, Y_2, T2, Z_2) \in \mathbb{F}_q$
output : $P + Q(X_3, Y_3, T3, Z_3) \in E(\mathbb{F}_q)$
Ensure: $Z_2 \neq 0$ and $T_2 = X_2 Y_2 / Z_2$ satisfy $aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$

$X_3 = (X_1 Y_2 + Y_1 X_2)(Z_1 Z_2 - d T_1 T_2)$,
$Y_3 = (Y_1 Y_2 - a X_1 X_2)(Z_1 Z_2 - d T_1 T_2)$,
$T_3 = (X_1 Y_2 + Y_1 X_2)(Y_1 Y_2 - a X_1 X_2)$,
$Z_3 = (Z_1 Z_2 - d T_1 T_2)(Z_1 Z_2 + d T_1 T_2)$,

it the same power trace for either operation, at a cost of 12M and 7add per point operation. The extended unified formula performs better at 9M and 7add. Comparing this against the dedicated twisted Edwards formulæ from Section 3.2.4, as shown in Table 3.6, shows an increase of 2M for the standard twisted Edwards and only an additional multiplication by a constant, d, for the extended case.

Table 3.6: Operation Count for twisted Edwards

	Point Addition		Unified	
Coord.	M and S	M only	M and S	M only
tE	10M 1S 1d 7add	11M 7add	11M 1S 2d 7add	12M 7add
eE	9M 1d 7add	9M 7add	9M 2d 7add	9M 7add

The full breakdown for the Edwards algorithms, directly used to generate the instruction set for the elliptic curve processor are described in Appendix A.2.

3.8 Regular Scalar Multiplication

Using scalar multiplication algorithms which use a fixed sequence of group operations, independent of the scalar, is another method of protecting against SSCA. Two such binary algorithms, which provide efficient regular scalar multiplication without the use of dummy operations are the *Montgomery ladder* [4], given in Algorithm 16 and the *Joye Double-and-Add* [5], given in Algorithm 17.

Algorithm 16: Montgomery Ladder

input : $P \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$
output: $Q = kP$

$R_0 \leftarrow \mathcal{O}$, $R_1 \leftarrow P$
for $i = n - 1$ *down to* 0 **do**
  $b \leftarrow k_i$, $R_{1-b} \leftarrow R_{1-b} + R_b$, $R_b \leftarrow 2R_b$
end
return R_0

Algorithm 17: Joye's Double-Add

input : $P \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$
output: $Q = kP$

$R_0 \leftarrow \mathcal{O}$, $R_1 \leftarrow P$
for $i = 0$ *to* $n - 1$ **do**
  $b \leftarrow k_i$, $R_{1-b} \leftarrow 2R_{1-b} + R_b$
end
return R_0

Each iteration of the Montgomery ladder performs a PA followed by a PD, so to an attacker, the side channel information is viewed as an alternating series of doublings and additions. There is of course a performance penalty associated with this similar to the Double-and-Add-Always along with adding to the storage requirements, as stated earlier, an important constraint in software. Alternatively, the Joye Double-and-Add, while also repeating a pattern of doublings and additions, requires only two point registers, and so is a cheaper memory alternative in software compared to the Montgomery ladder.

In [87], Meloni proposed adding two points while sharing the same Z coordinate, thereby allowing faster addition. The key observation in Meloni's addition is that the computation of $R = P + Q$ yields for free an additional representation for P. This operation is referred to as the ZADD operation.

Goundar et al. [63,88,89] introduced the co-Z operations ZADDU requiring 7M (5M+2S+6add), and *conjugate co-Z addition* and denoted ZADDC (for ZADD conjugate), using the efficient caching technique described in [90, 91]. The total cost for the ZADDC operation is 9M (6M + 3S + 15add).

Furthermore, Hutter et al. [64] presented a co-Z version of x-coordinate only formulæ referred to as *differential addition-and-doubling* and denoted as AddDblCoZ. These formulæ take two input points, $P = (X_1, Z)$ and $Q = (X_2, Z)$ on $E_{\mathcal{H}}$, sharing the same Z-coordinate, and outputs a pair of points, their addition $P' = P + Q = (X_1', Z')$ and doubling $Q' = 2Q = (X_2', Z')$, sharing the same Z-coordinate. The cost of this formula is 15M (11M + 4S + $1M_a$ + $1M_{4b}$ + 14add) as detailed in [64].

They optimise this formula by replacing the multiplication expression X_1X_2 with the equivalent $(X_1^2 + X_2^2 - (X_1 - X_2)^2)/2$ and later multiplied with a factor of 2. The cost of this optimisation is 14M (9M + 5S + $1M_a$ + $1M_{4b}$ + 14add).

They further optimise the formula for cases where the curve parameters a and b are dynamic by initialising three additional coordinates $T_a = aZ^2, T_b = 4bZ^3$ and $T_D = x_DZ$ to finally obtain algorithm 6 in [64]. The given formula reduces the memory by one register and increases the performance by 1M if $M_a = M_b = 1M$. The cost of this is 15M (10M + 5S + 13add).

The full algorithms can be found in Appendix A.3 and the interested reader is referred to [63, 64, 88, 89] for an in-depth explanation on how to efficiently carry out these computations using co-Z arithmetic for elliptic curves.

3.8.1 Co-Z Arithmetic

In the original Montgomery ladder, registers R_0 and R_1 are respectively initialised with point at infinity $\boldsymbol{O}$ and input point P. Since $\boldsymbol{O}$ is the only point with its Z-coordinate equal to 0, assuming that $k_{n-1} = 1$, the loop counter is started at $i = n - 2$ and R_0 is initialised to P and R_1 to $2P$. Next, ensure that the representations of P and $2P$ have the same Z-coordinate. This is achieved through the use of the DBLU which requires $6\mathsf{M}$ ($1Mul + 5Squ$).

Putting together the DBLU algorithm with ZADDU and ZADDC (from Appendix A.3), the implementation depicted in Algorithm 18 is obtained for the Montgomery ladder.

Algorithm 18: Montgomery ladder with Xo-Z Addition Formulæ

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

$(R_1, R_0) \leftarrow \mathrm{DBLU}(P)$
for $i = n - 2$ *down to* 0 **do**
  $b \leftarrow k_i$, $(R_{1-b}, R_b) \leftarrow \mathrm{ZADDC}(R_b, R_{1-b})$
  $(R_b, R_{1-b}) \leftarrow \mathrm{ZADDU}(R_{1-b}, R_b)$
end
return R_0

In Algorithm 19, the main loop of the Montgomery ladder is replaced by the differential addition-and-doubling formulæ of Hutter et al. Their algorithm (i.e. algorithm 3 in [64]) is modified by substituting the initialisation step for (X, Z)-only coordinate of a new efficient formula referred to as *doubling addition with update in homogeneous coordinates*, and denoted $\mathrm{DBLU}_{\mathcal{H}}$, which requires $9\mathsf{M}$ ($4\mathsf{M} + 5\mathsf{S}$). The notation $\mathrm{DBLU}_{\mathcal{H}}{}^*$ refers to X and Z coordinates only (excludes Y coordinate) and the cost of which is $8\mathsf{M}$ ($3\mathsf{M} + 5\mathsf{S}$).

The function *recoverfullcoordinates* recovers 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, details of which appear in Appendix A.3.

Algorithm 19: Montgomery Ladder with (X,Z)-Only Co-Z Addition Formulæ

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

$(X_1, X_2, Z) \leftarrow \mathrm{DBLU}_{\mathcal{H}}{}^*(P)$
for $i = n - 2$ *down to* 0 **do**
  $b \leftarrow k_i$, $(X_{2-b}, X_{1+b}, Z) \leftarrow ADC(X_{2-b}, X_{1+b}, Z)$
end
$Q \leftarrow$recoverfullcoordinates(X_1, X_2, Z)
return (Q)

3.8.2 Combined Double-Add Operation

A point doubling-addition is the evaluation of $R = [2]P+Q$. This can be done in two steps as $T \leftarrow P+Q$ followed by $R \leftarrow P+T$. If P and Q have the same Z-coordinate, this requires $14\mathsf{M}$ ($10\mathsf{M}+4\mathsf{S}$) by two consecutive applications of the ZADDU function.The resulting algorithm ZDAU (*co-Z double-add with update*) operation only requires $9\mathsf{M} + 7\mathsf{S}$ and is detailed in Appendix A.3. The triple of $P = (X_1 : Y_1 : 1)$ can be evaluated as $[3]P = P + [2]P$ using co-Z arithmetic [92].
The combined ZDAU operation immediately gives rise to an alternative implementation of Joye's double-add algorithm. The resulting algorithm is presented in Algorithm 21 Compared to the first implementation (Alg. 20), from a software perspective, the cost per bit now amounts to $9\mathsf{M} + 7\mathsf{S}$ instead of $11\mathsf{M} + 5\mathsf{S}$, an obvious speedup using the $\mathsf{S} = 0.8\mathsf{M}$ metric. In hardware however, both require $16\mathsf{M}$, and so the scheduling of the algorithm will be the deciding factor for which algorithm is the faster.

3.8.3 (X,Y)-only operations

The co-Z Montgomery ladder can be rewritten so as only to process X- and Y-coordinates. Operation ZACAU$'$[iv] is defined as the combination of operation ZADDC$'$ followed by operation ZACAU$'$. This is used to obtain an (X, Y)-only implementation of the Montgomery ladder, Algorithm 22. Using this formula, the cost per bit amounts to $14\mathsf{M}$ ($8\mathsf{M} + 6\mathsf{S}$).
(X, Y)-only co-Z operations can also be used with left-to-right signed-digit algorithms. Specifi-

[iv] The prime symbol $'$ is used to denote operations that do not involve the Z-coordinate.

Algorithm 20: Joye's Double-Add Algorithm with Co-Z Addition formulæ

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

$b \leftarrow k_1$,
$(R_{1-b}, R_b) \leftarrow \mathrm{TPLU}(P)$
for $i = 2$ *to* $n-1$ **do**
  $b \leftarrow k_i$,
  $(R_b, R_{1-b}) \leftarrow \mathrm{ZADDU}(R_{1-b}, R_b)$
  $(R_{1-b}, R_b) \leftarrow \mathrm{ZADDC}(R_b, R_{1-b})$
end
return R_0

Algorithm 21: Joye's Double-Add Algorithm with Co-Z Addition Formulæ (II)

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

$b \leftarrow k_1$,
$(R_{1-b}, R_b) \leftarrow \mathrm{TPLU}(P)$
for $i = 2$ *to* $n-1$ **do**
  $b \leftarrow k_i$,
  $(R_{1-b}, R_b) \leftarrow \mathrm{ZDAU}(R_{1-b}, R_b)$
end
return R_0

cally, a ZADDU' followed by a ZADDC' can be performd to obtain an (X, Y)-only double-add operation with a co-Z update: ZDAU'. The total cost of this operation is $14\mathsf{M}$ ($8\mathsf{M} + 6\mathsf{S}$). The complete algorithm is detailed in Algorithm 23 [67, 89], or from the author [93].

Table 3.7 gives a summary of scalar multiplication algorithms which are implemented on co-Z addition formulæ. Not taken into account in the designs presented here, and indeed for the Edwards curves, are cases where speed-up is acquired by setting certain curve parameters to constants, such as those described in [66, 94, 95]. While the existence of these are acknowledged, the aim of this work is to determine the underlying performance of the general case. As such, these optimised cases are not examined here. For further reading on the above three methods of mathematically securing algorithms against SPA attacks, the interested reader is directed to the 2005 paper by Avanzi [83] and the Lange chapter on mathematical countermeasures of [20].

Algorithm 22: Montgomery Ladder with (X, Y)-Only Co-Z Addition Formulæ

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

$(R_1, R_0) \leftarrow \mathrm{DBLU}'(P)$
$C \leftarrow (X(R_0) - X(R_1))^2$
for $i = n - 2$ *down to* 1 **do**
  $b \leftarrow k_i$, $(R_b, R_{1-b}, C) \leftarrow \mathrm{ZACAU}'(R_b, R_{1-b}, C)$
end
$b \leftarrow k_0$
$(R_{1-b}, R_b) \leftarrow \mathrm{ZADDC}'(R_b, R_{1-b})$, $(x_P, y_P) \leftarrow P$
$Z \leftarrow x_P Y(R_b)(X(R_0) - X(R_1))$, $\lambda \leftarrow y_P X(R_b)$
$(R_b, R_{1-b}) \leftarrow \mathrm{ZADDU}'(R_{1-b}, R_b)$
return$(\frac{\lambda}{Z})^2 X(R_0), (\frac{\lambda}{Z})^3 Y(R_0))$

Algorithm 23: 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 $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}_{\geq 3}$ with $k_0 = k_{n-1} = 1$
output: $Q = kP$

$(R_0, R_1) \leftarrow \mathrm{TPLU}'(P)$ **for** $i = n - 2$ *down to* 1 **do**
  $b \leftarrow k_i \oplus k_{i+1}$, $R_1 \leftarrow (-1)^b R_1$, $(R_0, R_1) \leftarrow \mathrm{ZDAU}'(R_0, R_1)$
end
$R_1 \leftarrow (-1)^{1+k_1} R_1$
$(x_P, y_P) \leftarrow P$
$\lambda \leftarrow \frac{y_P X(R_1)}{x_P Y(R_1)}$
return $(\lambda^2 X(R_0), \lambda^3 Y(R_0))$

Table 3.7: Operation Usage for Various Co-Z Addition Formulæ

Algorithm	Main op.	Other op.
Left-to-right algorithms:		
Mont. ladder co-Z addition (Alg. 18)	ZADDC, ZADDU	DBLU
Mont. ladder (X, Z)-only (Alg. 19)	AddDblCoZ	$\mathrm{DBLU}_{\mathcal{H}}{}^*$, recoverfullcoordinates
Mont. ladder (X, Y)-only (Alg. 22)	ZACAU′	DBLU′, ZADDC′, ZADDU′
Signed-digit (X, Y)-only (Alg. 23)	ZDAU′	TPLU′
Right-to-Left algorithms:		
Joye's double-add co-Z I (Alg. 20)	ZADDU, ZADDC	TPLU
Joye's double-add co-Z II (Alg. 21)	ZDAU	TPLU

3.9 Algorithmic Cost of SPA Secure Algorithms

Table 3.8 gives the efficiency costs for the SPA secure algorithms based on their response to parallelisation. This response, based on the count of the clock cycles, allows a general comparison of how each of the algorithms are expected to perform independent of the technology used. The table shows that the Double-and-Always-Add algorithm responds well to parallelisation. It is clear from the table that while the Joye II algorithm would appear to have a better computation time in software due to the ratio of S to M, in a hardware based system with an equal number of M, the Joye I algorithm appears to make more efficient use of parallelisation. In the case of the Joye I and the Montgomery Ladder co-Z algorithms, neither the ZADDC or the ZADDU algorithm benefits greatly from parallelisation. Similarly the algorithms for Joye II and Signed Digit also do not benefit much. The Montgomery Ladder XZ algorithms perform better with increased parallelisation. Contrasting against the Edwards algorithms; the unified twisted Edwards, the standard algorithm does not perform so well, while the extended twisted Edwards algorithm makes best use of parallelisation of all the algorithms with 91% multiplier usage for 2M to 4M.

The clock count associated with this is given in Table 3.9. Again, it is seen that the algorithms with the lowest clock counts, such as the AddDblCoZ and extended twisted Edwards algorithms will give the best time performance. It is also seen that the algorithms associated with the Joye I and the Montgomery Ladder co-Z, namely the ZADDC and ZADDU, fail to give any speed-up from 3M to 4M. As such, the 4M case can be omitted in both cases. Similarly the difference in clock cycles between the ML (X,Z)1 and ML (X,Z)2 are minimal for the AddDblCoZ algorithm for 3M and 4M. As such, both of these algorithms are merged going forward for the 3M and 4M cases, with the best results given.

For the ML (X,Z)3, there is no reduction in clock cycles from 3M to 4M for the AddDblCoZ3, however, there is a speed-up for the second part of the algorithm, the $recoverfullcoordinates2$, so this algorithm is kept.

A graphical representation similar to that of Figure 3.4 is again presented. Figure 3.10 gives a *ballpark* estimate of the expected hierarchy of the SPA secure algorithms based on their clock

Table 3.8: Multiplier Efficiency for SPA Secure Algorithms

Algorithm	no. of Mult.	no. of Mult Stages	Eff. %	no. of Mult Stages	Eff. %
		PD-PA_{dummy}		PD-PA	
Double	1	26	100	26	100
and	2	13	100	13	100
Always	3	9	96	9	96
Add	4	8	81	8	81
		Standard		Extended	
Unified	1	14	100	11	100
Twisted	2	9	77	6	91
Edwards	3	7	66	4	91
	4	6	58	3	91
		ZADDU (Alg. 38)		ZADDC (Alg. 38)	
ML co-Z	1	7	100	9	100
(Alg. 18)	2	4	87	5	90
Joye I	3	3	77	3	100
(Alg. 20)	4	3	58	3	75
		ZDAU (Alg. 43)		ZDAU′	
Joye II	1	15	100	14	100
(Alg. 21)	2	9	88	8	87
SD (X,Y)	3	7	76	6	77
(Alg. 23)	4	6	66	5	70
		AddDblCoZ1 (Alg. 40)		AddDblCoZ2 (Alg. 41)	
ML (X,Z)	1	17	100	16	100
(Alg. 19)	2	9	94	8	100
	3	6	94	6	88
	4	5	85	5	80
		AddDblCoZ3 (Alg. 42)		ZACAU′ (Alg. 44)	
ML (X,Z)	1	15	100	14	100
(Alg. 19)	2	8	93	7	100
ML (X,Y)	3	5	100	5	93
(Alg. 22)	4	5	75	4	87

cycle count. From this it would seem that the unified extended twisted Edwards should perform strongly compared to the rest and the Double and Always Add should perform worst.

3.10 Area and Power Results for SPA Secure Algorithms

The area results for the dummy algorithm, regular algorithms and unified algorithms are omitted due to them being very similar to those given for the dedicated case in Table 3.4. In most cases the M requires between 900 and 1200 occupied slices and increases by approximately 200 slices per additional multiplier. The instruction set BROM requires a single $18K$ block, while the BRAM address block requires five $36K$ blocks and a single $18K$ block. In the case of the Double and Always Add, an additional $36K$ block is required.

Table 3.9: Clock Cycle Count per SPA Secure Algorithm

Hardware Multipliers	No. of Mult. clks	No. of other clks	Total clks	No. of Mult. clks	No. of other clks	Total clks	No. of Mult. clks	No. of other clks	Total clks
	PD-PA_{dummy}			Standard tw Ed			Extended tw Ed		
1M	5044	320	5364	2716	148	2864	2134	135	2269
2M	2522	281	2803	1746	133	1879	1164	120	1284
3M	1746	269	2015	1358	127	1485	776	114	890
4M	1552	266	1818	1164	124	1288	582	111	693
	ZADDU			ZADDC			ZDAU		
1M	1358	103	1461	1746	141	1887	3104	264	3368
2M	776	94	870	970	129	1099	1746	243	1989
3M	582	91	673	582	123	705	1358	237	1595
4M	582	91	673	582	123	705	1164	235	1398
	AddDblCoZ1			AddDblCoZ2			recoverfullcoordinates1		
1M	3298	215	3513	3104	206	3310	2328	146	2474
2M	1746	191	1937	1552	182	1734	1164	128	1292
3M	1164	182	1346	1164	176	1340	970	125	1095
4M	970	179	1149	970	176	1143	776	122	898
	AddDblCoZ3			recoverfullcoordinates2			ZACAU′		
1M	2910	197	3107	2522	155	2677	2716	292	3008
2M	1552	176	1728	1358	137	1495	1358	271	1629
3M	970	167	1137	970	131	1101	970	265	1235
4M	970	167	1137	776	128	904	776	262	1038

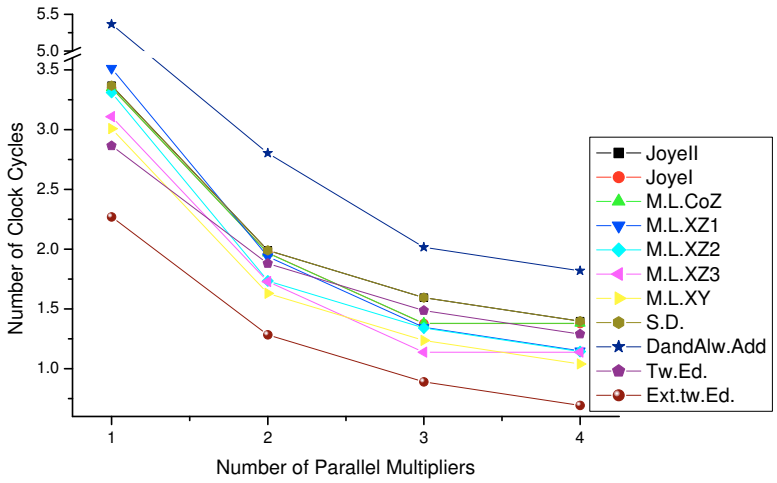


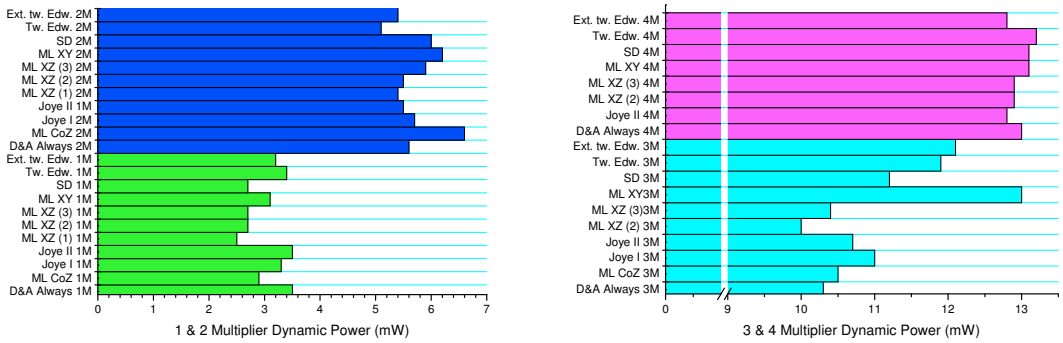
Figure 3.10: Clock Count for SPA Secure Doubling and Addition

Table 3.10 gives the timing, power and energy results. The results in bold again represent the lowest and best values for each set of parallelisation case. The Double-and-Always-Add, while being the slowest at $1M$ performs quite well for parallelisation and the speedup performance is quite acceptable at 4M. The Signed Digit algorithm performs best with the lowest computation time and energy for the standard 1M case. The Montgomery ladder (XY) gives the best results for the 2M case, while the Montgomery ladder (XZ) for three multipliers in parallel. For four multipliers, the extended twisted Edwards and the Montgomery ladder (XY) have the same energy usage, but the extended twisted Edwards has a faster computation time.

Table 3.10: SPA Secure Power and Timing Results

No of. Mult.	Alg.	Suppl. I-P mA - mW	μ Power mW	Calc. Time mS	Energy mJ	No of. Mult.	Alg.	Suppl. I-P mA - mW	μ Power mW	Calc. Time mS	Energy mJ
1	Double	150.6	119.8	42.3	5.1	1	Twisted	148.8	119.7	33.8	4.1
2	and	157.3	123.8	22.0	2.7	2	Edwards	154.7	123.3	22.2	2.7
3	Always	163.9	129.0	15.8	2.0	3	Unified	165.1	130.6	17.5	2.3
4	Add	169.1	132.1	14.2	1.9	4		168.5	156.6	15.2	2.0
1	Extended	148.7	119.5	26.8	3.2	1	Montgomery	148.9	119.2	26.6	3.2
2	Twisted	154.5	123.6	15.1	1.9	2	Ladder Co-Z	158.1	124.8	15.6	2.0
3	Edwards	165.3	130.8	10.5	1.4	3		163.5	129.1	10.9	1.4
4	Unified	169.1	131.9	**8.1**	**1.1**	1	Joye's	150.1	119.6	26.7	3.2
1	Joye's	151.3	119.8	26.8	3.2	2	Double-Add I	157.3	123.9	15.7	1.9
2	Double-Add	157.7	123.7	15.8	2.0	3		164.2	129.7	11.0	1.4
3	with Co-Z	163.9	129.4	12.7	1.6	1	Montgomery	147.3	119.0	26.5	3.2
4	Addition II	169.3	131.9	11.1	1.5	2	Ladder(X,Z)1	155.3	123.7	13.9	1.7
1	Montgomery	147.5	119.0	24.9	3.0	1	Montgomery	147.4	118.8	28.1	3.3
2	Ladder	155.6	124.1	13.9	1.7	2	Ladder	155.1	123.6	15.5	1.9
3	(X,Z) 2	163.1	129.1	**9.1**	**1.2**	3	(X,Z) 3	162.3	127.7	10.8	1.4
4		169.2	132.0	9.1	1.2	4		169.1	132.0	9.1	1.2
1	Montgomery	148.7	119.4	24.1	2.9	1	Signed	147.3	119.0	**23.6**	**2.8**
2	Ladder	156.0	124.4	**13.0**	**1.6**	2	Digit	155.8	124.2	14.2	1.8
3	(X,Y)	167.4	131.7	9.9	1.3	3		164.6	129.9	11.0	1.4
4		169.8	132.2	8.3	**1.1**	4		170.2	132.2	9.5	1.3

Figure 3.11(a) presents the dynamic power results for 1M and 3.11(b) for 2M. The quiescent power remains the same as that shown in Figure 3.6(a). Similar to the non SPA secure algorithms, the average power increases as more multipliers are added. While the Joye algorithms start off with a relatively high dynamic power, this reduces compared to the other algorithms for subsequent parallelisation. The Montgomery Ladder (XZ) algorithms in comparison have lower average power throughout. The Double and Add Always and the Edwards algorithms remain relatively high throughout.



(a) 1 & 2 Multipliers

(b) 3 & 4 Multipliers

Figure 3.11: Average Dynamic Power

Figures 3.12 and 3.13 give the Area-Time product and Area-Energy product. It can be seen from the Area-Time product graph that the Signed Digit algorithm performs best for 1M. The Montgomery ladder (XY) gives the top performance for 2M. The third iteration of the Montgomery ladder (XZ) is lowest for 3M, with the extended twisted Edwards slightly outperforming the Montgomery ladder (XY) at 4M. Most of the rest show similar time performance metrics, with the Double-and-Always-Add and the standard unified Edwards both being outliers on the graph. In this case the Area-Energy product shows similar results, except for the fact that The second iteration of the Montgomery ladder (XZ) is lowest for 3M and the Montgomery ladder (XY) and extended twisted Edwards both show a reduction in the AEP for 4M from 3M.

Figure 3.14 shows again a comparison between the expected and measured results, in this case for the SPA resistant algorithms. It is seen that the different sets of results deviate somewhat more than those presented in Figure 3.9. For 1M it is clear that both of the unified twisted Edwards algorithms are expected to perform better than they actually do. Examining the measured results shows that the standard unified Edwards gives equivalent performance levels to that of the Double and Always Add. However, excluding the Edwards algorithms results in the best and worst performers for each M stage being equivalent between the expected and measured graphs, and the hierarchy of the rest of the algorithms roughly conform between the two. Based on the metrics and results presented

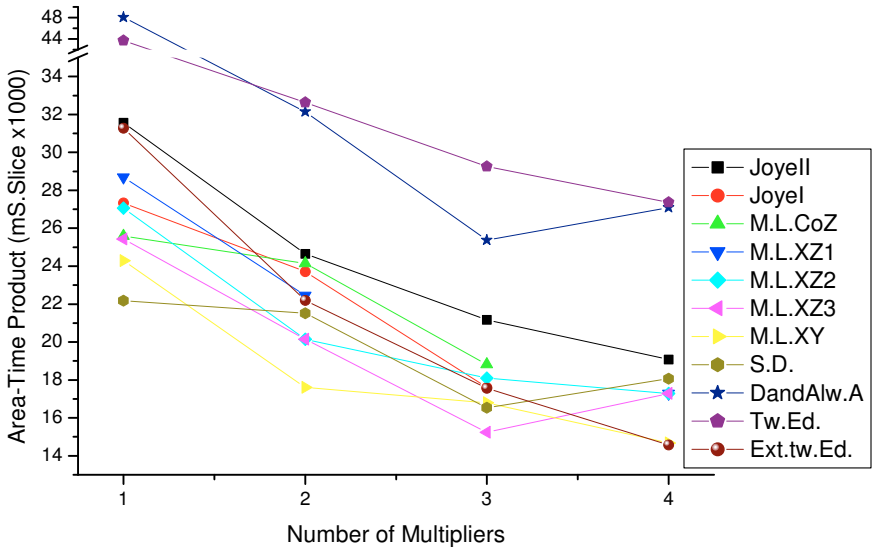


Figure 3.12: SPA Resistant Area-Time Product

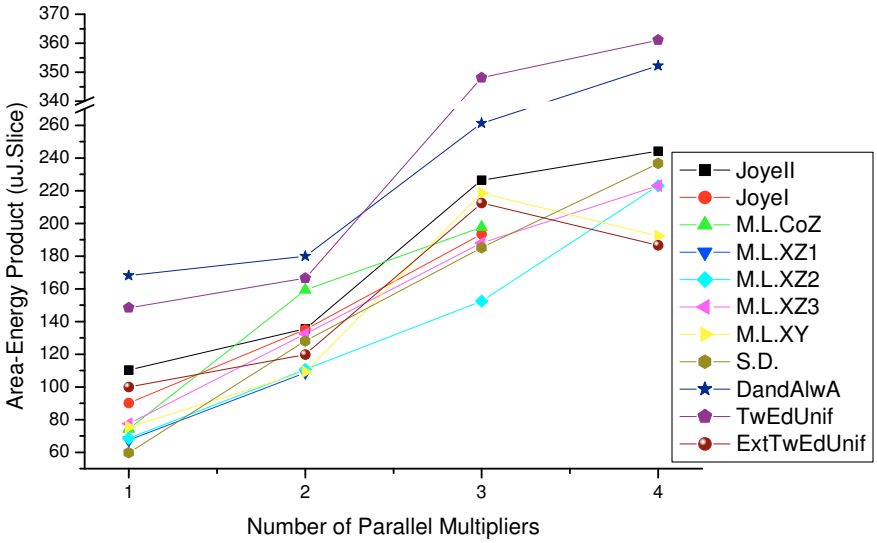
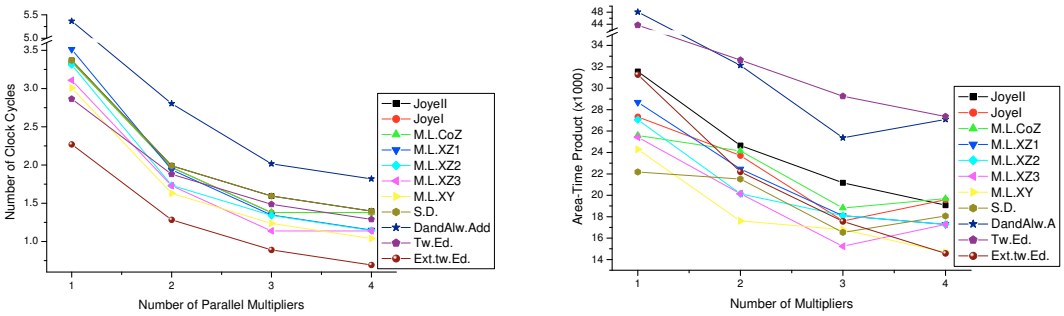


Figure 3.13: SPA Resistant Area-Energy Product

here, it has been shown that it should be possible, as long as care is taken, to estimate their relative performance based on their expected performance. This can be used to reduce test times through an analysis of the expected performance in the case of future algorithms.



(a) Expected Results **(b)** Measured Results

Figure 3.14: SPA Reistant Estimated Versus Measured Results

3.10.1 Comparing Dedicated Addition & SPA Secure Algorithms

A recap is made here and the relative performances of the dedicated addition algorithms are compared against the SPA secure algorithms. Recalling Table 3.5, along with Figure 3.6(a) and Figure 3.6(b) it is seen that in most cases, there is only a slight increase in the iteration time between the different sets of algorithms, which naturally results in only a slight increase in the energy costs. Indeed the best performing dedicated addition algorithm, the extended twisted Edwards at with an ATP at $4\mathrm{M}$ of 14.7 and an AEP at $1\mathrm{M}$ of 54.4 is roughly equivalent to the Montgomery Ladder (XY) and the extended twisted Edwards, both with an ATP at $4\mathrm{M}$ of 14.8 and the Signed Digit at $1\mathrm{M}$ with an AEP of 59.8. As long as care is taken when choosing the correct algorithm, it can be seen that some particular SPA secure algorithms can perform as well as, if not better, than some of the non SPA secure variants through the correct use of scheduling and parallelisation.

3.11 Larger Key and Field Sizes

It was described earlier that the hardware can be configured by the user for any characteristic p, and field extension m, as well as the respective memory sizes. In this work so far, a field size (p_b) and a key size (k) of 192, defined by ECRYPT II [42] as the current standard (protection from 2009 to 2020), are used throughout for each of the target algorithms using the curve parameters secp192r1 [81].

Table 3.11: 256 & 521 Area-Time, Area-Energy Product

Alg.	Key & Field	Area (Slices)	Time (mS)	ATP (Slice.mS)	Energy (mJ)	AEP (Slice.mJ)
SD (X,Y) - $1M$	256	1641	41.0	67281	6.9	11388
(Alg. 23)	521	3310	161.2	533572	30.0	99452
ML (X,Y) - $2M$	256	1828	22.1	40398.8	3.9	7147
(Alg. 22)	521	3985	83.8	333943	16.3	65226
ML (X,Z) - $3M$	256	2049	15.5	31759.5	2.7	5696
(Alg. 19)(Alg.42&46)	521	4032	59.3	239097.6	11.9	48182
ML (X,Y) - $4M$	256	2284	13.8	31519.2	2.5	5755
(Alg. 22)	521	4893	50.2	245628.6	10.3	50593

Next, the best results for each of the SPA secure parallel cases [v] are taken and implemented for a field size and a key size of 256 (protection from 2009 to 2040; defined by NIST [41] as equivalent security to AES-128 [26]) using curve parameters secp256r1 [81], and a field size and a key size of 521 (protection for the foreseeable future) using curve parameters secp521r1 [81] to examine larger more secure implementations and to allow an analysis of scaling.

The results are presented in Table 3.11 and the AT product for all three cases is graphed in Figure 3.15. Evident from Table 3.11 compared with Table 3.4 is the fact that the area is essentially doubled in each case (additionally, the BRAM stays the same for the 256 case but increases to sixteen $36K$ blocks for the 521). The time and the energy required to process each algorithm is also approximately $\times 2$ between 192 and 256, but increases to $\times 4$ between 256 and 521. Table 3.11 and Figure 3.15 also show how each of the algorithms roughly scale equivalently with the larger key and field sizes. While there is no significant change in order between the algorithms for the

[v] The unified extended twisted Edwards, while having equivalent results with the Montgomery ladder (XY) in the 4M case were excluded on the grounds that there is as yet no recommended curves in Edwards or twisted Edwards form

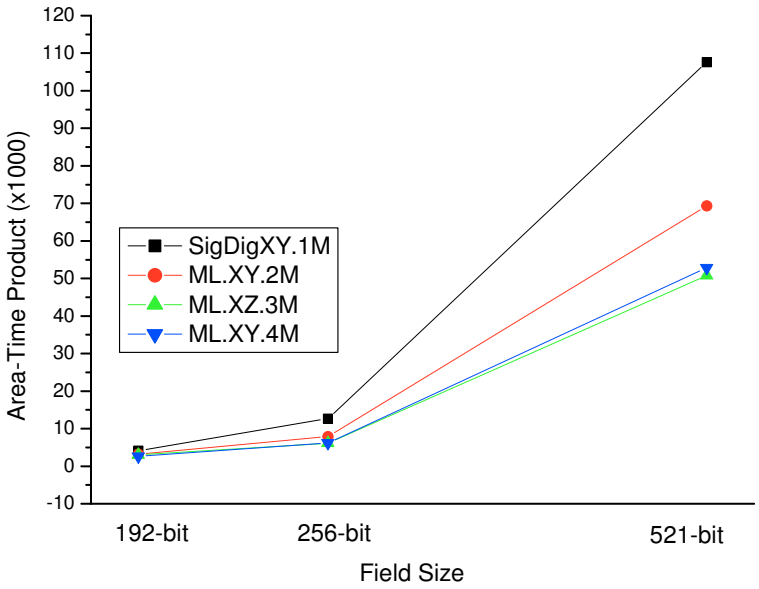


Figure 3.15: Area-Time Product: 192, 256 & 521

different key and field sizes, as the computations become larger it is clear that the differences in ATP between the algorithms also increases.

3.12 Conclusions

In this chapter a number of algorithms and parallel hardware implementations for them were presented. Initially dedicated point doubling and point addition algorithms were examined for different coordinate systems for the Double-and-Add algorithm. A reconfigurable elliptic curve processor was then presented and methods for performing modular addition and multiplication were described. Next, the aforementioned algorithms were implemented on this processor and observations were made. It was shown that the affine coordinate system performs best for a circuit containing a single multiplier, however, this did not scale well with additional multipliers and was overtaken at 3M in computation time and energy usage by its projective and Jacobian equivalents.

Performing best of these dedicated point doubling and point addition algorithms were curves of a special form, the twisted Edwards and the extended form of the twisted Edwards.

A brief look at power analysis attacks was then performed and how the algorithms examined so far are susceptible to this type of side channel attack. Following on from this, various SPA secure methods were then examined, namely dummy arithmetic instructions, unified doubling and addition and regular algorithms. The algorithmic cost and area, energy and timing results of these SPA secure algorithms were examined and some further observations were made. A comparison of the expected and measured results was performed and it was shown that the heirarchy of results in the expected case can be used to estimate how the measured results should roughly perform relative to each other and can be used to alleviate test time. It was also shown that through careful selection of the algorithm, along with through good use of parallelisation and scheduling, that some SPA secure algorithms can be made to perform as well as, or even better than their non SPA secure equivalents.

The best performing algorithms were then selected for further analysis with larger key and field sizes based on NIST specifications. Additional power, energy area and timing results were acquired and comparisons between the differing security levels were made.

4

Hash Functions

4.1 Introduction

Cryptographic hash functions are another type of cryptographic primitive from layer 4 of the hierarchical model shown in Figure 1.2. This chapter examines an implementation method for the SHA-3 hash functions [96]. Different implementation options are examined, a wrapper used to interface to the designs is described, and area, throughput and power results are presented.

A hash function $\mathcal{H}$ maps a message x of variable length to a string of fixed length. The process of applying $\mathcal{H}$ to x is called 'hashing', and the output $\mathcal{H}(x)$ is called the 'message hash' or 'message digest'. Cryptographic hash functions are hash functions that possess the following specific

properties [23]:

- **Pre-image Resistance**. This requirement means that for a given hash value y, it should be computationally infeasible for an adversary to find an input x such that $\mathcal{H}(x) = y$. Pre-image resistance is also known as 'one-wayness'.

- **Second Pre-image Resistance**. This implies that for a given input x_1, it should be computationally infeasible to find another input x_2, such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. This property is also known as 'weak collision resistance'.

- **(Strong) Collision Resistance**. It should be computationally infeasible for an adversary to find any two distinct inputs x_1 and x_2, such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$.

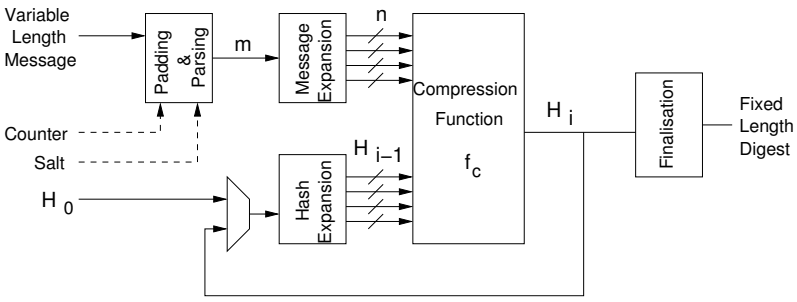


Figure 4.1: Generic Hash Function Internals

Figure 4.1 shows an overview of a generic hash function. A hash algorithm essentially consists of three stages: message padding and parsing; expansion; and compression. The binary message to be processed is appended and padded until it reaches the required bit length. The counter and salt are additional inputs, used in some hash functions to further obfuscate the input; for example, in HAIFA (Hash Iterative Framework) [97] based designs, a counter is fed in with the message, whereas, for Merkle-Damgård [98] [99], it is not.

The resultant padded message is parsed into m-bit blocks. These message blocks are passed individually to the message expansion stage where various rotations and shifting of the data blocks takes place.

The n-bit blocks from the message expansion stage are then passed to the hash compression function, or the hash core, f_C. The output of the compression function is a p-bit intermediate hash value H_i. A number of iterations, or rounds, of the compression function are then performed with the round data, H_i, further rotated and shifted using the hash expansion stage before being re-input to the compression function, f_C. The hash compression algorithm then repeats and begins processing another block from the message padding stage. After all m-bit data blocks have been processed, the output is formed by a concatenation of the final hash values up to the digest size required.

The rest of the chapter is structured as follows. Section 4.2 begins with a brief overview of *cryptographic hash functions* and the *SHA-3 competition*.

Section 4.3 briefly described the *characteristics* of the hash functions and examines different *implementation options* using example designs from the SHA-3 competition. Following this is a brief description of the round two and subsequent round three hash designs and their implementation methodologies in Section 4.4.

Section 4.5 presents the hash *wrapper* used to interface to the designs presented here. The *interface, communications* and *padding* protocols for the *round two* implementations used in this work are then presented.

Section 4.6 presents *area and throughput results* for each round two design, and gives a brief synopsys of how the best results selected here compare with those selected by NIST for the third round of the competition.

Section 4.7 gives an overview of the *round three* designs and begins by comparing the differences between the round two and round three variants. This section also contains updated *area, timing, power* and *energy results* for each of the designs.

Section 4.8 provides a *comparison* of the work presented here against the current state of the art. Finally, Section 4.9 concludes this chapter.

4.2 Background to the SHA-3 Hash Functions

The family of Secure Hash Algorithms (SHA) [100] began in 1993 when the National Institute of Standards and Technology (NIST) published the Secure Hash Standard. This version, known as SHA-0, was withdrawn by the National Security Agency (NSA) and replaced in 1995 by the SHA-1 algorithm after it was found to be insecure [6–8]. Both algorithms produce message hashes of length 160 bits. In 2002, NIST published three new hash functions with longer hash lengths: SHA-256, SHA-384 and SHA-512. In 2004, SHA-224 was added to the standard, and these four algorithms form the SHA-2 family of hash functions [101]. Although the standard still includes the SHA-1 hash function, its security has been compromised through cryptanalytic attacks. The trend in the cryptographic community was to move away from using older hash functions like SHA-1, towards newer functions like those in the SHA-2 family [102]. However, when insecurities were found following attacks against reduced versions of the SHA-2 family [103], the National Institute of Standards and Technology (NIST) started a competition for a new hash algorithm [96], namely SHA-3, similar to the former AES effort [26], with the intention of developing a more secure family of hash functions.

The contest initially received 64 submissions from designers worldwide. 51 of these designs progressing through to round one of the contest which began on November 1^{st} 2008. Approximately a year was given for each round of the competition, with round one being used to examine the security of the applicants. Round two candidates were announced on July 24^{th} of 2009 and the number of competing designs were reduced to 14 [104].

For round two, the NIST competition specifications [96], $6.C$, *Round 2 Technical Evaluation* gave the criteria for hardware and software testing; *"Round 2 testing by NIST will be performed on the required message digest sizes"* and *"the calculation of the time required to compute message digests for various length messages"*.

On December 9^{th} 2010, round two was completed and the field was reduced further to 5 competing designs, BLAKE, Grøstl, JH, Keccak and Skein for round three [105]. Finally, on October 2^{nd} 2012, Keccak was selected as the winner of the SHA-3 hash function competition [106].

4.3 Implementating SHA-3 Hash Functions

The initial 64 submissions recieved by NIST were accepted solely on being complete and proper submissions. Other factors, such as security, cost, and algorithm and implementation characteristics of the candidates did not enter the review process prior to the first round, nor did cryptanalysis or performance data of a submission impact the acceptance of the first-round candidates. However, NIST stated in the status report at the end of the first round [104], after selecting 14 of these designs for round two, that there were *many candidate algorithms with new and interesting designs, and with unique features that are not present in the SHA-2 family of hash algorithms* and felt that *the diversity of designs will provide an opportunity for cryptographers and cryptanalysts to expand the scope of ideas in their field, and it will also be less likely that a single type of attack will eliminate the bulk of the candidates remaining in the competition*.

From an implementation point of view, it was clear that this diversity of designs would require significantly different design methodologies. Each hash function would need to be tailored differently to achieve its best results. While some of the round two designs contained similar features (and indeed shared some basic components), no two designs could be implemented using the same methodology. Table 4.1 gives the constructions of the round two SHA-3 hash functions and their variants as well as the various inputs and state sizes in bits. The *Structure* loosely defines the hash function overview. The *Type* describes the design of the hash functions. The *Counter*, *Message* and *Salt* all form the inputs to the hash functions, while the *State* describes the internal size of each of the hash functions. As the $\{224\}$ variant is almost identical to the $\{256\}$ and similarly, the $\{384\}$ to the $\{512\}$ these values are omitted. The only notable differences being Keccak, where the message size increases to 1152-bits for $\{224\}$ and 832-bits for $\{384\}$, and Luffa, where the state size decreases to 1024-bits for $\{384\}$. Also, each of the hash functions and their variants have an initial vector (IV) as part of the input, but for the SHA-3 competition each of these IV's are fixed to a specific value throughout, and as such they are not considered as part of the input but rather as stored constants within each hash function itself.

Due to the large number of differing designs, a standard metric was needed to allow a baseline

Table 4.1: Hash Function Internals

			224/256				384/512			
Design	Structure	Type	Counter	Message	Salt	State	Counter	Message	Salt	State
SHA-2	Merkle-Damgård	Add-XOR-Rotate	64	512	0	512	128	1024	0	1024
Blake	HAIFA	Add-XOR-Rotate	64	512	128	512	128	1024	256	1024
BMW	Iterative	Add-XOR-Rotate	64	512	-	2048	64	1024	-	4096
Cubehash	Iterative	Add-XOR-Rotate	-	256	-	1024	-	256	-	1024
Echo	HAIFA	AES based	64	1536	128	2048	64	1536	128	2048
Fugue	Iterative	AES based	64	32	-	96	64	32	-	1148
Grøstl	Iterative	AES based	64	512	-	512	64	1024	-	1024
Hamsi	Conc-Permute	Serpent based	64	32	-	512	64	64	-	1024
JH	Iterative	Block Cipher based	128	512	-	1024	128	512	-	1024
Keccak	Sponge	Add-XOR-Rotate	-	1088	-	1600	-	576	-	1600
Luffa	Sponge	S-box based	-	256	-	768	-	256	-	1280
Shabal	Iterative	Add-XOR-Rotate	-	512	-	1408	-	512	-	1408
SHAvite-3	HAIFA	AES based	64	512	256	256	128	1024	512	512
SIMD	Iterative	Block Cipher based	64	512	-	512	64	1024	-	1024
Skein	UBI	Add-XOR-Rotate	96	512	-	512	96	512	-	512

comparison between them. The goal decided upon was to explore area-speed trade-offs in the implementations of the hash functions, and to compare the efficiency of the designs by examining the throughput per unit area (TPA) metric as described in Section 2.10. High-throughput hash function implementations are beneficial in network server applications, for example. Analysing the throughput per slice of the architectures, determines which hash function implementations make the most efficient use of FPGA area. The throughput is calculated as follows:

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

Within each of these hash architectures, various area-speed trade-offs were investigated through an exploration of the design space in an attempt to find the best throughput for the least amount of area used. Selecting implementation and performance metrics is nontrivial. It was described in Section 2.6 how resources in FPGA differ not only between different vendors, but also between different families of the same vendor. Results for a particular hash function architecture on different FPGA platforms cannot be directly compared, since any two platforms have different underlying technologies. As such, dedicated resources specific to a particular FPGA family or vendor were not used in the analysis. All implementions were performed using only slice logic, i.e. distributed memory blocks used instead of BRAM blocks etc. Using this methodology, diffi-

cult comparison[i] is avoided, and a single value can be allocated to a result (e.g. area in slices). A wrapper interface was designed to enable fair comparison of the different hash functions when operating in a standardised and constrained environment. Starting from a basic iterative architecture various implementation methods were examined, some examples of which are:

- **Loop Unrolling**. This involves running multiple rounds of a loop in the same clock cycle. The number of loops to be completed will decrease with each unroll, however the area will increase and the critical path will get longer, causing a reduction in the maximum frequency and thus reducing the clock speed.

- **Parallelised Design**. In some designs, e.g. Grøstl, the permutations are identical except for the execution of the AddRoundConstant step, where different round constants are used. Therefore, when implementing f_C, one design choice is to compute Q in parallel by replicating the hardware for P.

- **Interleaved Design**. This approach is the opposite of above, and re-uses the hardware for P to compute Q, resulting in extra clock cycles but a lower area.

Two examples of this examination method are presented in Sections 4.3.1 and 4.3.2 for the 256-bit versions of Cubehash and Shabal as updated versions of those presented in [107].

4.3.1 CubeHash

CubeHash was submitted by Bernstein to the SHA-3 contest [108]. FPGA implementations of the CubeHash compression function, f_C, were designed. The rotation and swapping operations are implemented in hardware by simply re-labelling the relevant signals. Since the state comprises 1024 bits, the same architecture can be used to produce message digests with any of the lengths required for SHA-3. Therefore, a CubeHash8/32-256 implementation will have the same throughput and throughput per slice performance as a CubeHash8/32-512 implementation.

[i] Such as comparing BRAM and CLB area results as described in Section 2.6.2.

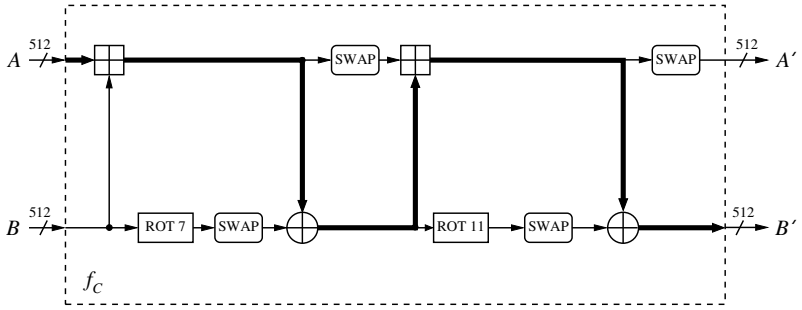


Figure 4.2: Cubehash Compression Function

The critical path through the compression function, consists of two modulo 2^{32} additions and two XOR operations, as indicated by the heavy lines in Figure 4.2. The compression function is used $r = 8$ times for each message block M_i (i.e. for each message byte in this case, since $b = 1$). CubeHash architectures were investigated where f_C is unrolled by various degrees. The lowest area design iteratively uses a single f_C unit and takes 8 clock cycles to process a single message block, and the highest area design uses a chain of four f_C units in series to process a single message block in two clock cycles. The results are shown in Table 4.2. Note that the figures quoted for each design include the initial XOR of the message block with the state, and also include the area of the output register that stores the result of the last f_C calculation in the chain.

Table 4.2: CubeHash Implementation Results

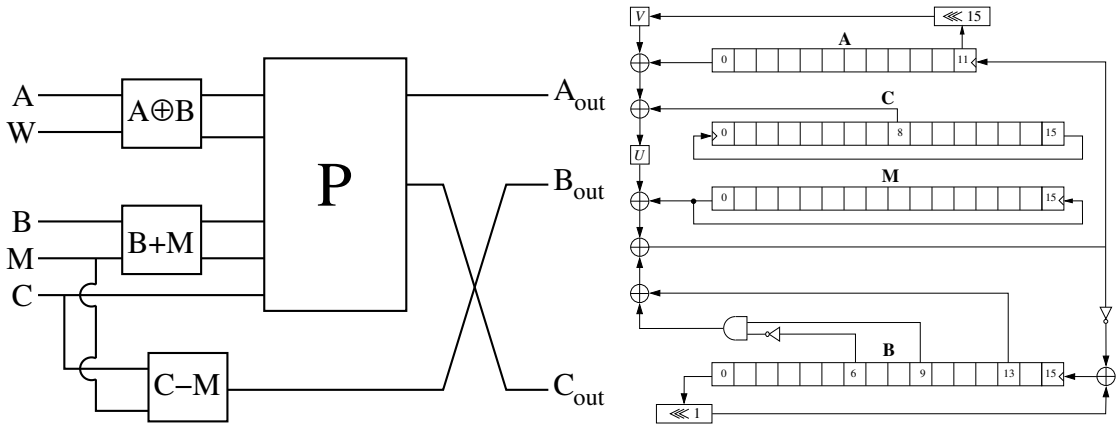
	Architecture	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Mbps)	TP-Area (Mbps/slice)
Virtex-5	Iterative	1025	166.66	17	**2509.7**	**2.45**
	2×-unrolled	1440	55.14	13	1085.8	0.75
	4×-unrolled	Congested Design				

As expected, the critical path of each design increases with the degree of unrolling. However each increase in the critical path is greater than the corresponding benefit of a decrease in the number of clock cycles, so an overall increase in throughput (TP) is not obtained. The iterative design provides the best TP-Area result. The longer critical paths in the unrolled designs are caused by routing delays between the long chains of adders on the FPGA. Indeed a $4\times$ unrolled design

results in a congested design, where the design exceeds the available resources or the compiler fails to optimise the design during place-and-route. A comparison with other Cubehash designs shows that other groups, such as Matsuo *et al.* [109,110] and Homsirikamol *et al.* [111] also chose to present iterative design implementations.

4.3.2 Shabal

Shabal was submitted by the Saphir research project to the SHA-3 contest [112]. Shabal, Figure 4.3(a), uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits, as shown in Figure 4.3(b).



(a) Hash **(b)** Compression Function (P)

Figure 4.3: Shabal

The Shabal compression function is based on a Non-Linear Feedback Shift Register (NLFSR) construction. The precomputed initialisation vector (IV) was used to remove the configuration stage and thus remove the initial two message block from the latency. When designing Shabal, the XOR, addition and subtraction operations were all implemented in parallel. In the permutation $\mathcal{P}$, the rotation operations were implemented through simple wiring. In order to realise the central part of the permutation, a shift-register based approach was adopted, where the state words are shifted along chains of 32-bit registers. The multiplication operations U and V form the non-linear part of the NLFSR; these were implemented using the shift-then-add method. Once the shift registers

have been loaded with the appropriate initial values, the central permutation result is calculated after 48 clock cycles. The final part of the permutation $\mathcal{P}$ adds words from the A and C states.

For these modulo 2^{32} additions, two design choices were investigated. In the first design, the NLFSR is used together with a single adder to compute $A[0] \boxplus C[3]$, and the result is fed back to $A[15]$. Note that the direction in which the C word is shifted must be reversed. This design takes a further 36 clock cycles to produce the final result. The second design for this stage of $\mathcal{P}$ expands the addition into 12×3 series additions, e.g. $A[0] \leftarrow A[0] \boxplus C[3] \boxplus C[15] \boxplus C[11]$. Using this approach, the final result is computed without requiring any extra clock cycles, but at the expense of area for 35 additional adders.

Table 4.3: Shabal Implementation Results

	Final Additions in $\mathcal{P}$	Area (slices)	Max. Freq. (MHz)	#Cycles	TP (Mbps)	TP-Area (Mbps/slice)
Virtex-5	Series	2119	222.22	86	1322.9	**0.624**
	Parallel	2512	143.47	50	**1469.1**	0.584

The area, timing and throughput results for Shabal are shown in Table 4.3. In the lower-area implementations, the critical path is within the NLFSR construction, from register $A[11]$ to register $B[15]$. The higher-area implementations have a longer critical path, due to the three series additions used to compute the final result. However, these higher-area implementations still attain better throughputs than the lower-area implementations, due to the lower number of clock cycles required. Both designs have similar throughput per slice metrics, with the lower-area implementation more efficient on the Virtex-5 platform (it is also noted here that the higher-area implementation was found to be more efficient on a Spartan-3 platform). While some designers implemented similar methods to the ones presented here [110, 111], others such as Julien Francq and Céline Thuillet [113] unfolded Shabal a number of times and found their best result at an unfolding factor of 3. So while the work presented here found the best results to be for the low area approach, it is seen that further examination of the design space can result in improvements to the throughput and throughput-area.

4.4 SHA-3 Round Two Implementations

Next follows a brief overview of some of the SHA-3 hash functions. An overview of the algorithm is first presented followed by a description of the architecture used to implement the design. While the work presented here [114, 115] discusses results for all four hash variants (224, 256, 384, 512) of all of the round two hash designs, attention in this section is focused on the five designs which were selected for round three of the competition; Blake, Grøstl, JH, Keccak and Skein. Here follows a brief description of the make-up and implementation of those designs emphasized for the second round of the competition.

4.4.1 BLAKE

Algorithm: The BLAKE hash function was developed by Aumasson *et al.* [116]. It uses the HAIFA iteration mode. A large inner state is initialised using a counter and a salt along with the input message. The data block is then injectively updated by message-dependent rounds, and finally compressed to return the next chain value. The inner state of the compression function is represented as a 4×4 matrix of words. A round comprises updating this matrix; first, all four columns are updated independently, followed by four disjoint diagonals. In the update of each column or diagonal, two message words are input according to a round-dependent permutation. Each round is parametrized by predefined constants. After the sequence of rounds, the state is reduced to half its length with feedforward of the initial data block and the salt. It is based on LAKE [117] and ChaCha [118] and uses a wide pipe construction where the size of the internal state is significantly larger than the size of the output.

Architecture: For the implementation of BLAKE the compression function was further subdivided into two identical sections, to allow re-use of the component blocks and thereby reducing the area. This subdivision increases the latency of the hash permutation to complete a round from two to four clock cycles, but reduces the critical path from four adders to two adders thus increasing the maximum frequency of the permutation. For the larger variant which requires 32 clocks

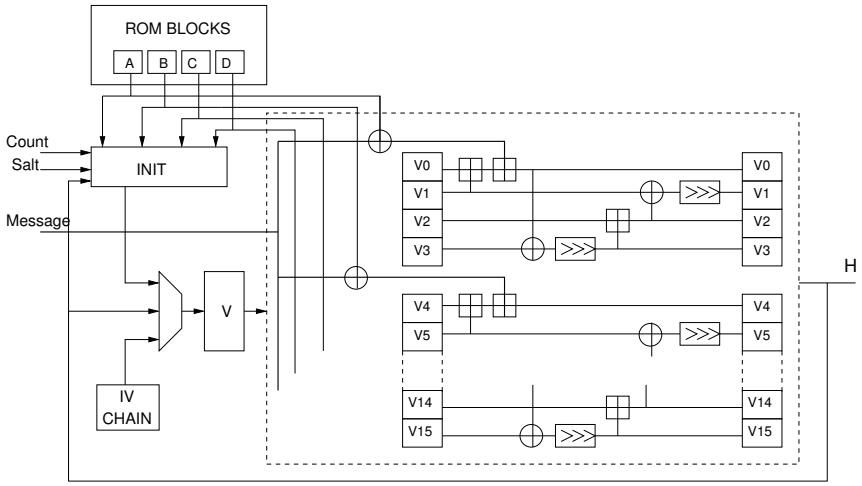


Figure 4.4: Blake Architecture

to load a 1024 bit message, it ensures there is no delay where the hash function needs to wait for loading to complete. Figure 4.4 shows the modified design where V is the compression function. The Adders and XORs are generated using standard operators (using the '+' operator of the IEEE.std logic unsigned package) and the rotation operations were implemented through simple wiring, with multiplexers to select the particular subround rotation. The 16 constants required by the initialisation and round stages are stored in distributed ROM.

4.4.2 Grøstl

Algorithm: Grøstl was submitted by Gauravaram *et al.* [119]. It is an iterated hash function with a compression function built from two fixed, large, distinct permutations using a wide-trail design strategy. Grøstl uses components from the AES [26]; the same S-box is used and the diffusion layers are constructed in a similar manner to those of the AES. Similar to BLAKE, Grøstl is a so-called wide-pipe construction. The initial message is padded and split into n-bit message blocks and each message block is processed sequentially. The compression function maps two inputs of n-bits each to an output of n-bits. The first input is called the chaining input, and the second input is called the message block. The compression function is based on two underlying n-bit permutations P and Q. In the Grøstl permutations, a total of four round transformations

are defined for each permutation; AddRoundConstant, SubBytes, ShiftBytes and MixBytes. The output transformation truncates the final bits to the required digest size.

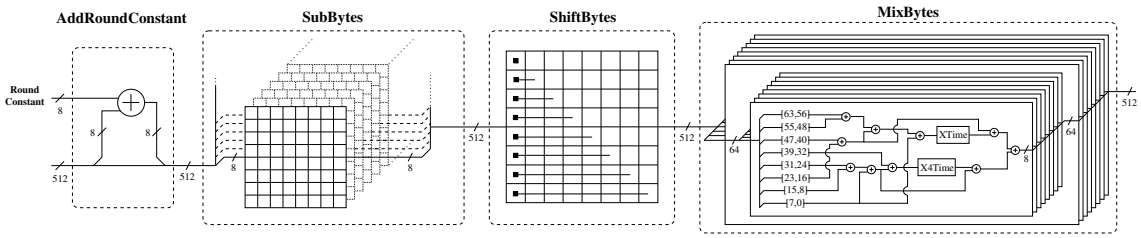


Figure 4.5: Grøstl P/Q Permutation

Architecture: The architecture for the P and Q permutations of Grøstl is illustrated in Figure 4.5. The first stage in each permutation is the AddRoundConstant block which simply performs an XOR on one byte of the ℓ-bit input state. The round constants are stored in distributed memory on the FPGA. The SubBytes stage transforms the state, byte by byte, using the AES S-box generated using distributed ROM. The ShiftBytes transformation was realised in hardware by simply re-labelling the bytes of the state. MixBytes is the final stage of the permutation function, and processes each column of the state matrix separately and in parallel using combinational logic. An output register was used to store the state at the output of the MixBytes transformation.

The compression function f_G for the Grøstl implementation consists of two permutation functions, P and Q. Permutations P and Q are identical except for the execution of the AddRoundConstant step, where different round constants are used. Therefore, the design choice in this case was to compute Q in parallel by replicating the hardware for P. Two XOR arrays are required to complete the compression function for the input to P, and for the final output H_i.

4.4.3 JH

Algorithm: The hash function JH, was developed by Hongjun Wu [120]. Its compression function is constructed from a bijective function (a block cipher with constant key) and the generalized AES design methodology is used to design large block ciphers from small components. The com-

pression function combines a 1024-bit previous hash block (H_{i-1}), a 512-bit message block (M_i) to produce a 1024-bit hash block (H_i). The compression function (CF^{JH}) is applied to each message block, M_i. The bijective function consists of 35 rounds, each consisting of an S-box, linear transformation and permutation, and a single final round consisting of just the S-box. The JH block cipher combines the most important features of AES, namely the substitution-permutation network (SPN) and Maximum Distance Separable (MDS) code, and Serpent [121], namely SPN and bit-slice implementation.

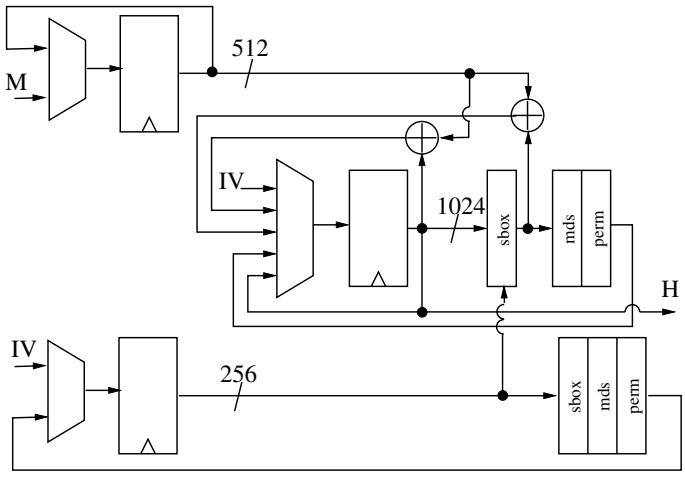


Figure 4.6: JH Architecture

Architecture: JH uses the same design for all four variants and is based on simple components. Two 4-bit S-boxes are used; the selected table depending on the value of a round constant. It can also be viewed as a 5-bit to 4-bit substitution. The linear transformation implements a $(4, 2, 3)$ maximum distance separable (MDS) code over $\mathbb{F}_q(2^4)$, and the permutation shuffles the output according to three distinct smaller permuations. The 256-bit round constants can be generated either in parallel with the data path or pre-computed and stored in memory where they can be re-used. In the design presented, the full 1024-bit data state is operated on at once. Each round completes in one clock cycle. The 256-bit sub-key state is calculated in parallel, as illustrated in Figure 4.6.

The S-box and linear transformation functions are implemented as combinational logic as outlined in the submission documentation, and the grouping and permutation functions are rewiring circuits. In the round constant data path, only the S-box corresponding to select bit '0' is required. Three registers are required for data storage, one each for the round constant, message block and data block respectively.

4.4.4 Keccak

Algorithm: Keccak, submitted by Bertoni *et al.* [122] is a hash function based on the sponge construction [123]. The design philosophy underlying Keccak is the hermetic sponge strategy [124], where, a (cryptographic) sponge function is a class of algorithms with finite internal state that take an input bit stream of any length and produce an output bit stream of any desired length, during an absorbing stage and a squeezing phase. The Keccak sponge function is built from three components; a state memory, S, a function, f, of fixed length that permutes or transforms the state memory and a padding function. The sponge construction state memory is divided into two sections, bitrate, R of size r-bits, and capacity C of size c-bits. After padding of the initial message, the first r-bit block is XORed with R, and S is replaced by $f(S)$. This repeats until all padded blocks are 'absorbed'. The R portion of the state memory provides the output in 'squeezed' out blocks of r-bits during the squeezing phase. The sum $r + c$ determines the width of the keccak-f permutation used in the sponge construction and for the SHA-3 competition is selected by the designers to be 1600. The Keccak hash function uses five functions θ, ρ, π, χ and ι, which are consecutively computed during each round. The functions θ, χ and ι are designed using XOR, AND and NOT operations, while ρ and π provide the permutations.

Architecture: The NIST submissions use the same KECCAK-f permuation for all variants, with different capacity (c), bitrate (r) and diversifier (d) values, where smaller digest sizes have a greater bitrate. The five steps of the permutation consist of addition and multiplication operations in $\mathbb{F}_q(2)$. The full round computes in a single clock cycle, and an extra clock is required for loading in of

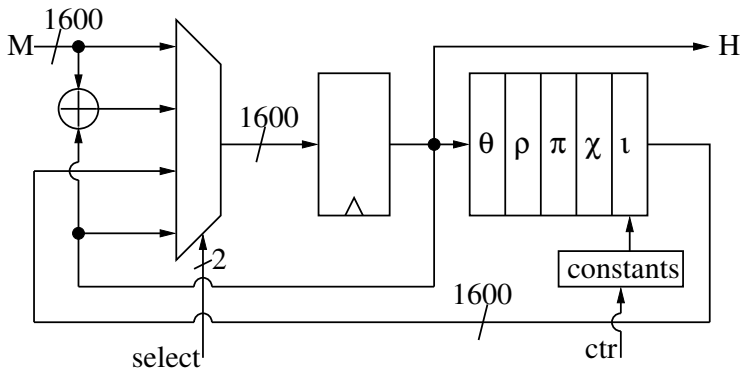
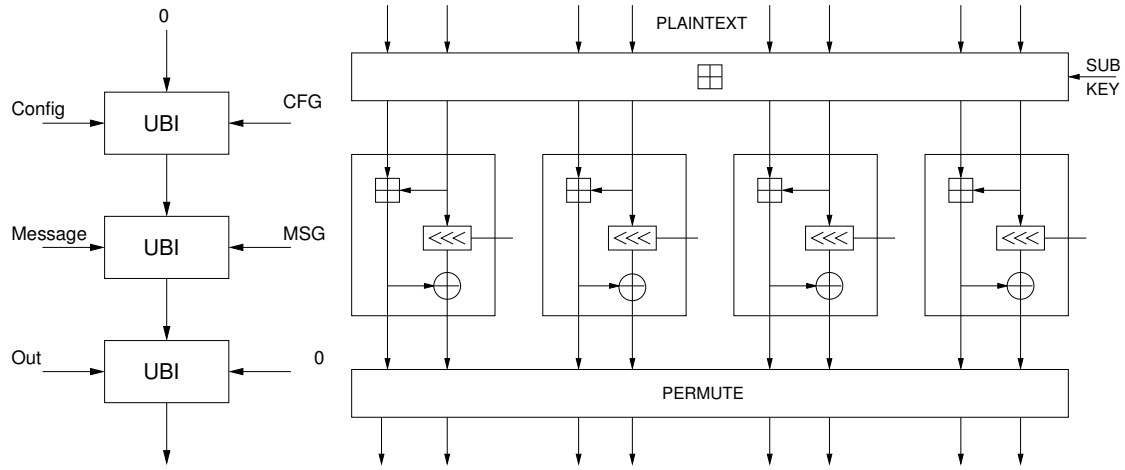


Figure 4.7: Keccak *f(1600)* Architecture

the message. The padded message of length r is loaded in by XOR'ing it with r bits of the state. The 64-bit round constants are defined as the output of a linear feedback shift register and can be pre-computed or generated as required. In the design, presented here in Figure 4.7, they are pre-computed and stored in distributed ROM. Only one register is required and is used to store the state value. The HDL implementation provided in the specification documentation was used as a reference for the permutation steps in the design.

4.4.5 Skein

Algorithm: Skein was submitted by Ferguson *et al.* [125] and Skein-512 is the primary proposal of the Skein family of algorithms. It is based on the Threefish algorithm, a tweakable block cipher [126]. A Unique Block Iteration (UBI) chaining mode takes in the chain value, the message and a 'Tweak' defined by an 128-bit configuration string derived from the message counter and UBI constants as shown in Figure 4.8(a). The Threefish algorithm has 72 rounds consisting of four sets of four MIX functions followed by a permutation of the eight 64-bit words. Each MIX function consists of a single addition, a rotation by a constant, and an XOR. The rotation constants repeat every eight rounds. The key schedule generates the subkeys from the chain and a tweak. A finalisation UBI stage consisting of a null message, a Tweak and the previous chain.



(a) UBI **(b)** Threefish 512 Round

Figure 4.8: Skein Architecture

Architecture: For this design of Skein-512 four rounds of threefish were unrolled, Figure 4.8(b). In this way, a UBI message block of Skein takes 18 clocks for the rounds to complete, plus 5 for preprocessing and data loading. The precomputed IV was used to remove the configuration stage and thus remove the initial message block from the latency. Each subsequent message block and the output block are calculated identically. The tweak, which ensures each message block is different, is generated by the counter in the padding.The Adders and XORs were generated in a generic fashion and the rotation operations were implemented through simple wiring, with multiplexers to select the particular subround rotation.

4.5 Hash Interface

Cryptographic hash functions have many information security applications, such as digital signatures, message authentication codes (MACs), and other forms of authentication, the purpose of which is:

- **Confidentiality**. Privacy.
- **Authentication**. Who created or sent the data and password verification.

- **Integrity**. Confidence that the data has not been altered.

- **Non-repudiation**. The order is final, and it was sent at a certain time or sequence.

The purpose of the Hash function in these applications is to produce a fingerprint. It is usually used as one part of an overall security system. As such, along with a standard metric of TPA to allow a baseline comparison between the different designs, a standard interface was also required to enable fair comparison of the different hash functions when operating in a standardised and constrained environment.

The work next presents a wrapper [127] that can be used to interface between any particular hash function algorithm and the outside world where the padding is included in the wrapper as opposed to the hash function block itself. In this way, 'fully autonomous' designs can be easily and efficiently inserted inside the wrapper thereby allowing fast testing[ii]. It allows re-use of any padding scheme used in multiple hash functions which cuts down on design time. It also alleviates any issues concerning designs which do not take into account bandwidth limitations or extra area or timing due to *external* stages such as *loading*, *initialisation* and *finalisation*.

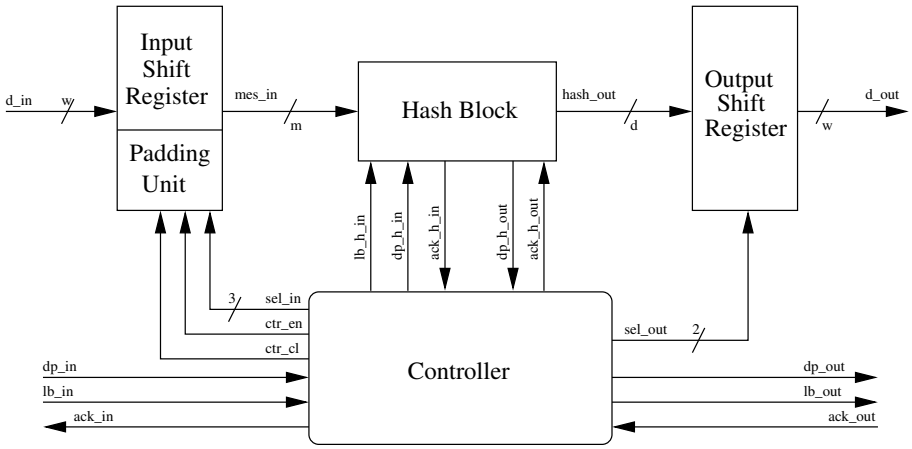


Figure 4.9: Hash Wrapper

[ii] Speed estimate for the algorithms required by NIST were simply defined as; at a minimum, the number of clock cycles required to generate one message digest, and set up the algorithm [96] Section 6.B.

Figure 4.9 shows a block diagram of the interface architecture[iii]. It comprises an input register which includes any padding required, an output parallel load shift-register and control circuitry. 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 such as that used in the FSL bus. The input shift-register reads in and stores these w-bit values to the size required, m, which is the message block size of the hash function under test. If a message ends prior to this register being completely filled, padding is added to the partial message to bring it to the required size. The output shift-register performs a similar task, holding the hashed 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.

Because of the different design methods and message sizes that comprise all the hash functions, as shown previously in Table 4.1, a number of user defined constants are specified at the top level of the implementation code and as such can be easily modified by the user to synthesize the message size and padding scheme necessary to run a particular hash function. These constants are defined as:

- The hash function required.
- The counter size required for message length addition during padding.
- The message digest size required.
- The input message block size of the hash function.

The hash function for a given digest size is then synthesized according to these constants. As stated earlier, each of the hash functions and their variants have an initial vector (IV) as part of the input, but for the SHA-3 competition each of these IV's are fixed to a specific value throughout, and as such they are not considered as part of the input but rather as stored constants within each hash function itself.

[iii] Note that variants on the design shown here were also developed which include extra bus lines from the control block to the hash function block where necessary, i.e. for designs which require counter or salt values to be input as part of the message.

The performance of the implementations on Virtex-5 were based on the following targets: the data bus was defined to be 32-bits wide and the padding explicitly included as part of the hardware interface block. Both decisions were shown to have an impact on the latency, throughput and area of the different hash function implementations. The results presented are post-place and route.

4.5.1 Communications Protocol

Table 4.4 defines the various communication signals between the wrapper and the external world. It can be seen from Figure 4.9 that the communications protocol between the hash function and wrapper, as well as between the wrapper and the outside world is similar to that suggested by Gaj [128]. It differs however in the fact that a user wishing to hash a message does not need to do any preprocessing to their plaintext before sending the message, such as adding message length data, but only needs to set an end of message (EOM) 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.

Table 4.4: Wrapper Interface

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

While there is valid data on the line, a data present *dp_in* flag is set high. To avoid the need to transmit a count of the number of valid message bits in the input block, and thus needing to know

the lengths of message sections prior to transmission, the data present signal is set high when all of the data on the input bus are valid message bits, i.e. each message blocks read in is of size w. When the wrapper reads in the data on *d_in* it acknowledges that the data has been read in by setting *ack_in* high. It is then ready to receive the next block of data. Conversely, when the message digest is ready to be read out on the output bus, *dp_out* is set high by the function wrapper. This data will remain on the output bus until a return acknowledge *ack_out* is received. The system then returns to its initial state in preparation for the next hash message.

Table 4.5 defines the various communication signals between the hash function and the wrapper. These closely mirror the external signal lines, and in most cases perform equivalent functions between the hash block and the interface as the externals do between the interface and the outside world. However, there is no *lb_ out* equivalent as the hashed digest, d, is output to the output shift-register as one complete block.

Table 4.5: Hash Interface

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**

4.5.2 Padding Protocol

There are many different padding schemes utilised by the designers of the hash functions, and in some cases varying padding schemes between the different sizes of the same hash function. Table 4.6 gives a brief outline of the various padding schemes used by the different round two

Table 4.6: Padding Schemes per SHA-3 Type

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
Blake224	1, 0's, until congruent (448 mod 512), 64-bit message length
Blake256	1, 0's, until congruent (447 mod 512), 1, 64-bit message length
Blake384	1, 0's, until congruent (895 mod 1024), 128-bit message length
Blake512	1, 0's, until congruent (894 mod 1024), 1, 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)
Echo224/256	1, 0's until congruent (1392 mod 1536), 16-bit message digest, 128-bit message length
Echo384/512	1, 0's until congruent (880 mod 1024), 16-bit message digest, 128-bit message length
Fugue	0's until a multiple of 32, 64-bit message length
Grøstl224/256	1, 0's until congruent (448 mod 512), 64-bit block counter
Grøstl384/512	1, 0's until congruent (960 mod 1024), 64-bit block counter
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
JH	1, 0's until congruent (384 mod 512), 128-bit message length, min 512-bits added
Keccak224	1, 0's until a multiple of 8, append 8-bit representation of 28, append 8-bit representation of 1152/8, 1, 0's until a multiple of 1152
Keccak256	1, 0's until a multiple of 8, append 8-bit representation of 32, append 8-bit representation of 1088/8, 1, 0's until a multiple of 1088
Keccak384	1, 0's until a multiple of 8, append 8-bit representation of 48, append 8-bit representation of 832/8, 1, 0's until a multiple of 832
Keccak512	1, 0's until a multiple of 8, append 8-bit representation of 64, append 8-bit representation of 576/8, 1, 0's until a multiple of 576
Luffa	1, 0's until a multiple of 256
Shabal	1, 0's until a multiple of 512
SHAvite3-224/256	1, 0's until congruent (432 mod 512), 64-bit message length, 16-bit digest length
SHAvite3-384/512	1, 0's until congruent (880 mod 1024), 128-bit message length, 16-bit digest length
Simd224/256	0's until a multiple of 512, extra block with message length
Simd384/512	0's until a multiple of 1024, extra block with message length
Skein	0's if multiple of 8, else 1, 0s, until a multiple of 512

candidates.

As can be seen from Table 4.6, similarities between most of the different padding schemes allow for generation of a generic block for variants of Merkle-Damgård strengthening [23] padding schemes, as well as paddings types of all-zeros or one-and-trailing-zeros.

Figure 4.10 shows the block diagram of the padding unit used to implement some of the padding schemes. The input word w is read into a multiplexor and combined with the $m - w$ least significant bits of the current register value. If no input value is available on the input bus, the current

register value is held, or if the message is fully read in, the appropriate padding bits are appended. Input blocks consisting of just counter values and/or padding bits are also catered for. For example in the case of Fugue, the register is filled twice more, firstly with all zeros, followed by an m-bit representation of the number of message blocks. In the case of Shabal, the extra message block simply consists of $100\ldots0$ of the size m. Note that not all hash functions require extra message blocks.

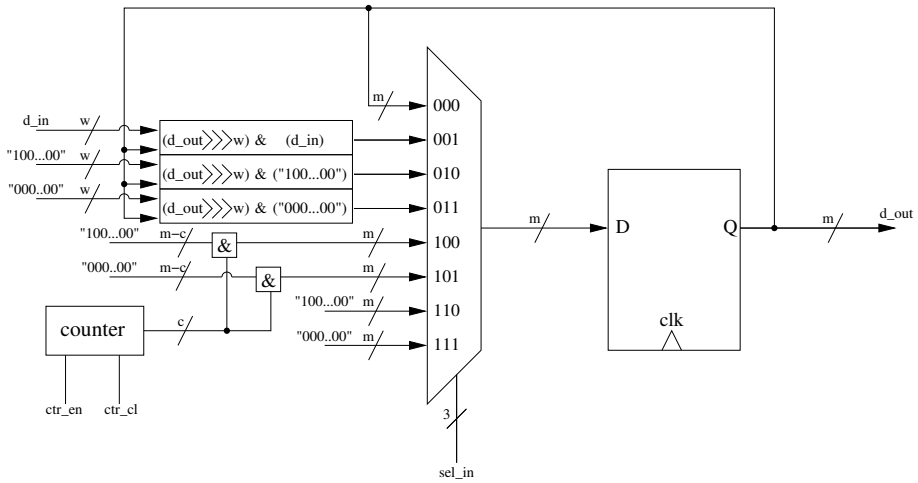


Figure 4.10: Padding Block

4.6 Round Two Results

Next, the round two results are examined. All of the round two algorithms were designed using the wrapper described in Section 4.5 and implemented on the Xilinx Virtex-5 (xc5vlx50-3ff324) FPGA and evaluated on the SASEBO-GII cryptographic evaluation board [48]. A brief definition and presentation of the clock cycle count required to perform a hash is discussed followed by area, frequency and throughput results for each of the round two hash functions and their variants.

Table 4.7 gives the clock count for the various designs. As can be seen from the table, some hash designs require extra time to load in the padding scheme, while others have finalisation stages comprising a number of rounds. For calculating the throughput, as the size of the message

to be hashed increases, these padding and finalisation stages will have less of an impact on the overall calculation time. However for short messages, they have a big impact. **A short message is therefore defined as the time required to process the padding, initialisation, a single message block and finalisation, and a long message as just the time to process the message block**. Note that each hash function operates over the state size given in Table 4.1, and so designs with smaller state sizes will require a larger number of rounds to hash the same amount of data as a design with a large state size. This is also reflected in the throughput.

The larger state sizes however are affected by the loading latency as explained in Section 4.5. Where the time required to hash the message is larger than the time required to load the message this only affects the initial message loading, but in cases where the load latency is longer than the hash latency (denoted * in Table 4.7), there will be a delay as the hash waits for data to load. In this scenario the clock count for the throughput needs to take this additional delay into consideration. Not given here is the output message load time, which in all cases is the hash digest size/output bus size.

The area, frequency and throughput results, as presented in Equation 2.13 given are inclusive of the wrapper with *TP-s* using Table 4.7's clock count for a **short message**, and *TP-l* using the clock count for a **long message** where the padding and finalisation stages will have little impact on the message. SHA-2 is presented as the benchmark to compare the others against. The full area, frequency and throughput results inclusive and exclusive of the wrapper are presented and compared in Appendix B.1 to allow a comparison of the designs exclusive of the padding implemented in hardware or constrained by a fixed size input bus.

The bar graphs, shown in Figure 4.11(a) and Figure 4.11(b) present a graphical representation of throughput/area results for both long and short messages for 256-bit, inclusive of the wrapper. Appendix B.2 presents the hash results for the 512-bit cases along with results for padding implemented in software both inclusive and exclusive of the wrapper.

The performance of the hash functions in each of the scenarios presented in Appendix B.2 for throughput-area, compared to SHA-2 is used as a metric to determine best design. All scenarios

Table 4.7: Hash Function Timing Results

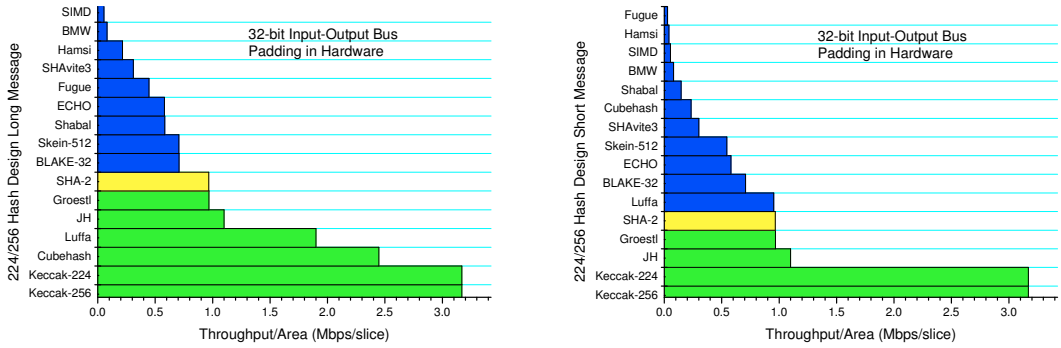
Hash Design	32-bit load #Cycles	Extra Padding	Padding #Cycles	Message Rounds	Round #Cycles	Long Msg #Cycles	Final Rounds	Final #Cycles	Short Msg #Cycles
SHA224/256	16	0	0	64	1	65	0	0	65
SHA384/512	32	0	0	80	1	81	0	0	81
Blake224/256	16	0	0	10	4	40	0	0	40
Blake384/512	32	0	0	14	4	56	0	0	56
BMW224/256*	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
Echo224/256*	48	0	0	8	1	8	1	1	9
Echo384/512*	32	0	0	10	1	10	1	1	11
Fugue224/256	1	2	1	1	7	7	13	91	98
Fugue384	1	2	1	1	10	10	20	180	190
Fugue512	1	2	1	1	13	13	22	264	277
Grøstl224/256*	16	0	0	10	1	10	0	0	10
Grøstl384/512*	32	0	0	14	1	14	0	0	14
Hamsi224/256	1	3	1	3	2	6	6	24	31
Hamsi384/512	2	3	1	6	2	12	12	48	61
JH	16	1	1	35	1	38	0	0	38
Keccak224*	36	0	0	24	1	25	0	0	25
Keccak256*	34	0	0	24	1	25	0	0	25
Keccak384*	26	0	0	24	1	25	0	0	25
Keccak512	18	0	0	24	1	25	0	0	25
Luffa224/256*	16	0	0	8	1	8	1	8	16
Luffa384*	16	0	0	8	1	8	2	16	24
Luffa512*	16	0	0	8	1	8	2	16	24
Shabal	16	0	0	1	50	50	3	150	200
SHAvite3-224/256	16	0	0	12	3	36	1	1	37
SHAvite3-384/512	32	0	0	14	4	56 (70)	1	1	71
Simd224/256	16	1	1	4	8	32(41)	0.5	4	36(45)
Simd384/512	32	1	1	4	8	32(41)	0.5	4	36(45)
Skein	16	0	0	18	22	22	18	22	44

were considered in the analysis to enable a fair and full comparison, allowing equal weighting for both specific and generic bus sizes along with padding implemented directly in hardware or in software. Examining which designs outperform SHA-2 reveal some interesting conclusions. Three designs consistently outperform SHA-2 for all of the categories; Keccak, Grøstl and JH. The next best designs according to these metrics were Echo, Cubehash, Skein, Luffa and Blake. NIST stated in [104] that *cost is of particular concern for constrained platforms*, and Echo is a good deal larger in area than most of the other designs as shown in Table 4.8. Indeed NIST cited this large area as one of the reasons for excluding it from round three [105]. An argument can be made to exclude Luffa on the grounds that, while although performing adequately, it is very similar to Keccak, the other sponge construction, but does not perform as well. Excluding these

Table 4.8: Hash Function Implementation Results

Hash Design	Area (slices)	Max.Freq (MHz)	TP-l (Mbps)	TP-l/Area (Mbps/slice)	TP-s (Mbps)	TP-s/Area (Mbps/slice)
SHA-2-256	1019	125.06	985	0.966	985	0.966
SHA-2-512	1771	100.04	1264	0.713	1264	0.713
BLAKE-32	1653	91.34	1169	0.707	1169	0.707
BLAKE-64	2888	71.04	1299	0.449	1299	0.449
BMW-256*	5584	14.30	457	0.081	457	0.081
BMW-512*	9902	8.98	287	0.028	287	0.028
Cubehash	1025	166.66	2509	2.447	239	0.233
ECHO-256*	8798	161.21	5158	0.58	5158	0.58
ECHO-512*	9130	166.66	8000	0.876	8000	0.876
Fugue-256	2046	200	914	0.446	60	0.029
Fugue-384	2622	200.08	640	0.244	33	0.012
Fugue-512	3137	195.81	481	0.153	22	0.007
Grøstl-256*	2579	78.06	2498	0.968	2498	0.968
Grøstl-512*	4525	113.12	3619	0.799	3619	0.799
Hamsi-256	1664	67.19	358	0.215	69	0.041
Hamsi-512	7364	14.93	79	0.01	15	0.002
JH	1763	144.11	1941	1.1	1941	1.1
Keccak-224*	1971	195.73	6263	3.17	6263	3.17
Keccak-256*	1971	195.73	6263	3.17	6263	3.17
Keccak-384*	1971	195.73	3063	1.55	3063	1.55
Keccak-512	1971	195.73	4509	2.28	4509	2.28
Luffa-256*	2796	166.66	2666	0.953	2666	0.953
Luffa-384*	4233	166.75	2668	0.63	1778	0.42
Luffa-512*	4593	166.66	2666	0.58	1777	0.58
Shabal	2512	143.47	1469	0.584	367	0.146
SHAvite3-256	3776	82.27	1170	0.309	1138	0.301
SHAvite3-512	11443	63.66	931	0.081	918	0.08
SIMD-256	24536	107.2	1338	0.054	1338	0.054
SIMD-512	44673	107.2	2677	0.059	2677	0.059
Skein-512	3027	83.57	1945	0.706	973	0.353

two designs leaves Keccak, Grøstl, JH, Cubehash, Skein and Blake. Five of these six were selected for round three of which Cubehash was disqualified on potential security issues [129–131]. It is also interesting to note that each of these selected designs has a completely different structure or type (as given in Table 4.1) to every other round three selected design.



(a) Long Message Wrapper

(b) Short Message Wrapper

Figure 4.11: 256-bit Wrapper Throughput-Area

4.7 Round Three Analysis

For the round three hardware implementations presented here, minor changes were made to the finalist designs implemented here to bring them more in line with results from other developers. This necessitated some changes to the round two wrapper design. While the fixed size of the bus was unchanged, it was decided to implement the padding in software. This allowed for a comparison with other groups, most of which neglected padding in their calculations. Methods were initially tested whereby the first message block defined the message size and counter, however this led to both an unnecessary extra latency in the load time along with extra logic increasing both the complexity and the area. As such, software was generated in C to generate the padding prior to the message being loaded to the hash function.

4.7.1 Round Three Changes

Of the five designs selected for round three of the competition; Blake, Grøstl, JH, Keccak and Skein, the designers were allowed to *tweak* the round two versions with minor changes.

For Blake, the number of rounds for the 224 and 256-bit digest versions was changed from 10 to 14 and the number of rounds for the 384 and 512-bit digest versions was changed from 14 to

16. The BLAKE naming convention was also changed from BLAKE-28, BLAKE-32, BLAKE-48, and BLAKE-64 to, respectively, BLAKE-224, BLAKE-256, BLAKE-384, and BLAKE-512. These extra rounds result in an increase in timing for the Blake designs and a slight change to the area due to routing.

For round three Grøstl, new Shift Values were selected. In the original Grøstl-224/256, the transformation ShiftBytes was used in both P_{512} and Q_{512}. In the round three version, different ShiftBytes values are used in Q_{512}. An equivalent update is applied to Grøstl-384/512 and Q_{1024}. New Round Constants were also selected. In the original Grøstl-224/256, the round constants $C_{[i]}$ of P_{512} and Q_{512} used in the transformation AddRoundConstant in round i were sparse with only a single byte value different from zero. In the tweaked Grøstl-224/256, additional round constants are added for P_{512} and additional round constants are added for Q_{512} along with an xor by FF. Similar tweaks are applied to Grøstl-384/512 round constants. These updates appear to result in no significant increase in the timing or area.

In the case of JH, the number of rounds is changed from 35.5 to 42 rounds. This tweak results in an increase in the timing due to the additional rounds but a decrease in the area as the additional circuitry required for the final half block is removed.

For Keccak, the padding rule has been shortened and simplified. The new padding rule is the pad10*1 rule. This update only affects the padding, and as such the timing and area results remain the same.

The only change to the Skein hash function is in the key schedule parity constant where the Skein *tweak* constant is changed. This tweak results in no change to timing and area results.

4.7.2 Comparing Different Round Results

The new implementations of the round three designs and their round two counterparts are presented for Post-Place and-Route results in Table 4.9. Both long and short messages produce the same TP and TPA results with the exception of Skein where the throughput reduces to 1117.9 and 1190.2 for round two and round three respectively. The biggest contrast between the tweaked de-

signs is in the 512 variant of Blake where the additional rounds from the tweak result in a reduced TPA. Blake-256 does also have a reduced throughput but a similar TPA to that of Blake-32. Interestingly, round three JH, while having a reduced area at a cost of an increased number of rounds, results in an almost identical TPA as that of the round two version. Round three JH has the lowest area of all of the implementations. All of the other round three designs remain similar to their round two equivalents. Hence, confident analysis of round two holds.

Table 4.9: SHA-3 Round 2 & 3 Results Comparison

Algorithm	Round #	Area (slices)	Max.Freq (MHz)	TP (Mbps)	TP/Area (Mbps/slice)
Blake-256	2	1584	119.2	1525.7	0.963
	3	1568	118.5	1444.5	0.921
Grøstl-256	2	4124	210.3	6732.1	1.632
	3	3137	200.8	6427.2	2.048
JH-256	2	1752	313.5	4224.2	2.411
& 512	3	1426	311.7	3546.6	2.487
Keccak-256	2	1551	244.4	7820.8	5.042
	3	1551	246.0	7872.0	5.075
Skein-256	2	2842	101.2	2355.8	0.828
& 512	3	2718	102.2	2380.4	0.875
Blake-512	2	2757	107.0	1956.5	0.709
	3	2799	102.4	1807.8	0.645
Grøstl-512	2	6675	171.1	5475.5	0.820
	3	6673	167.3	5353.9	0.802
Keccak-512	2	1551	244.1	5624.0	3.620
	3	1553	242.5	5588.1	3.598

4.7.3 SHA-3 Power and Energy

As an additional set of metrics on which to base the selection criteria, the power and energy results were analysed. The updated round three designs were implemented on the SASEBO and a similar method to that in Section 3.4 along with Equation 3.19 were used to get the timing, power and energy measurements. The area results, presented in Table 4.9, give the Post-Place and-Route results of the hash block as a stand-alone entity. Results were taken for both short (S) and long (L) messages, with a short message comprising a message up to 512-bit blocks (or 1088-bit in

Table 4.10: FPGA Power and Timing Results for SHA-3 at 24MHz

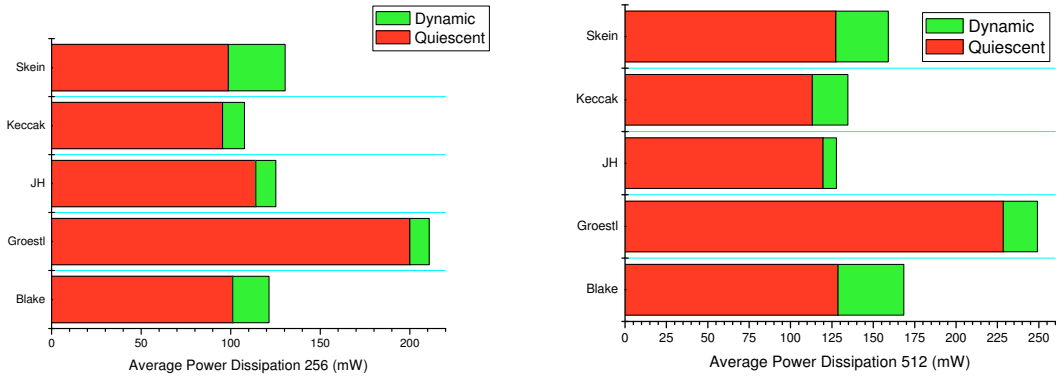
Algorithm 224/256	Supplied Current-Power mA - mW	μ FPGA Power mW	Calc. Time mS	Energy mJ
Blake S	169.3	138.5	4.34	0.601
Blake L	151.4	121.4	92.25	11.20
Grøstl S	263.6	203.5	**2.80**	0.570
Grøstl L	319.9	210.8	**33.78**	7.11
JH S	157.7	128.2	6.78	0.870
JH L	157.4	125.2	77.0	9.64
Keccak S	157.9	128.8	4.32	**0.557**
Keccak L	132.7	107.7	42.92	**4.62**
Skein S	226.1	182.0	3.93	0.715
Skein L	164.0	130.4	59.02	7.57
384/512				
Blake S	204.5	161.9	65	1.057
Blake L	224.6	168.4	70.56	11.88
Grøstl S	440.3	229.3	4.50	1.008
Grøstl L	440.5	249.1	**35.05**	8.725
JH S	158.6	128.6	7.10	0.913
JH L	159.5	127.7	77.50	9.89
Keccak S	159.7	131.7	**3.29**	**0.433**
Keccak L	168.5	134.6	54.76	**7.37**
Skein S	239.6	189.4	4.20	0.806
Skein L	207.1	159.0	59.44	9.45

the case of Keccak), and a long message comprising a randomly selected (from the KAT values (where available)) 11624-bit message. In both cases, the messages tested required padding, and in the latter case a number of rounds is required to process the hash.

Table 4.10 gives the time, power and energy results for the round three designs. It is shown that for a digest size of 256 that Grøstl has the shortest processing time for both long and short messages. It also has the highest mean FPGA power but a low energy usage due to this short processing time. Keccak has the lowest energy for both message types. For the 512-bit designs, Keccak has the lowest energy and shortest processing time for short messages, while Grøstl again gives the shortest processing time for long messages.

The average power dissipation of the processor was measured for a throughput of each of the

designs running at the Sasebo onboard 24MHz clock frequency. The current being drawn by the FPGA was measured for a supply voltage of $1V$. These voltage and current measurements were then used to calculate the total power consumed on both lines. The full set of power measurements and results are presented in Appendix B.3 to allow a full analysis of the energy and iteration timing.



(a) 256-bit Hash

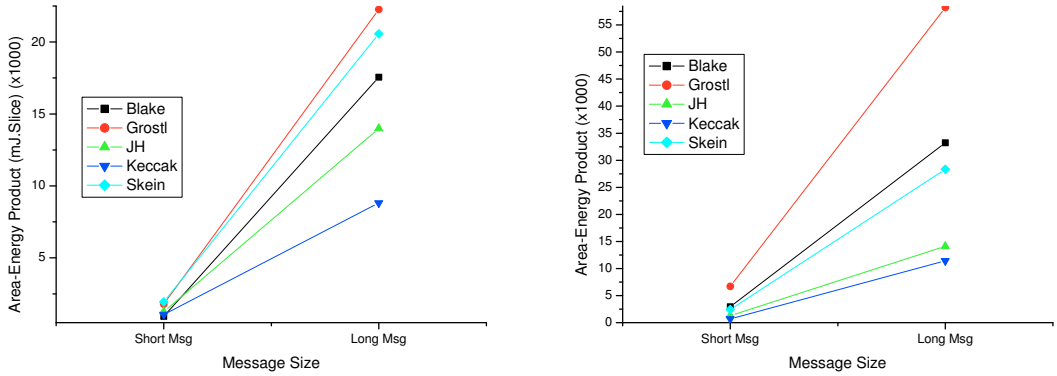
(b) 512-bit Hash

Figure 4.12: Average Power Dissipation at 25MHz

The average power dissipation for the 256-bit designs is shown in Figure 4.12(a) and for the 512-bit designs in Figure 4.12(b). This is further subdivided into the quiescent power and the dynamic power. The quiescent power is the power that the board is drawing when the design is programmed and the clock is connected but reset is held active, thereby preventing any switching from occurring in the circuit. It is a measure of the standby power drawn by the FPGA. The larger the amount of logic resources used, the higher the quiescent power, a case in point being Grøstl where the area is $\times$ 2 or 3 that of the other designs. The 512 designs quiescent power is also on average higher than that of their 256-bit counterparts also due to larger areas. The dynamic power is the power dissipated by logic switching within the FPGA. It is calculated as the the difference between the total average power and the quiescent power. Skein has the highest dynamic power of the designs, probably due to the large 64-bit adder circuitry. Similarly so for the adders in Blake. It is clear from the graphs that the quiescent power is the dominant factor, comprising on average 86% of

the total power for the hash.

Graphing the Area-Energy product for the implementations, based on the average power dissipation, in Figures 4.13(a) and 4.13(b), show that Keccak has the lowest and best Area-Energy product for all digests and message sizes. JH again performs strongly, showing quite similar AEP results to Keccak in the 512-bit case. Blake outperforms Skein in the 256-bit case, however the opposite occurs for the 512-bit. Grøstl, while having good computation time and energy is penalised due to its large area and performs worst for the AEP.



(a) 256-bit Hash **(b)** 512-bit Hash

Figure 4.13: Area-Energy Product

4.8 Comparison with Other Work

NIST stated that computational efficiency of the algorithms in hardware, over a wide range of platforms, was to be addressed during the second round of the contest [96]. The cryptographic research community responded by dedicating a lot of resources in performance evaluation. There are roughly four areas which are of primary interest:

- Software implementations on typical desktop and server hardware
- Software implementations on microcontrollers

- FPGA implementations
- ASIC implementations

The first two categories are comprehensively covered by benchmarking suites based on the work of Bernstein (SUPERCOP [132]) and Wenzel-Benner *et al.* (XBX [133]). The implementations come from many different designers, and all results are publicly available [132, 134].

In the third and fourth categories, various research teams implemented all of the second round candidates, and a web page called the 'SHA-3 Zoo' was set up by the Institute for Applied Information Processing and Communications (IAIK) in Graz to track these hardware implementation results [135]. Each report itself contains a fair comparison between the candidates. However, comparing the results between different reports is difficult, due to technological differences, varying quality of the implementations and the variations between different hardware interfaces.

For FPGA implementations in particular, the significance of inter-report comparisons is weak. Similar to the work presented here, several research groups attempted to improve this situation by defining a standard interface and applying their methodology to their own implementations. Again each methodology only applies to their own results and comparisons with other reports continues to be difficult, due to each groups placing different emphasis on particular metrics, bus sizes and approaches to padding and area usage. Of the different groups which fully evaluated the round two designs for FPGA using a standard interface, each used a number of different design and constraint metrics which they felt most fairly and accurately described a true evaluation. However, each group formulated a different approach as to how this was to be achieved.

For example, Homsirikamol *et al.* [111] implemented the 256 and 512 hash variants using an optimization toolkit called ATHENa [136] using the interface defined in [128]. ATHENa attempts to find the optimal combination of options and seed of the implementation tools. They set the data bus width to 64 bit where possible to ensure that the throughput would not be the limiting factor. For the cases where a larger bus width was necessary, they used two synchronized clocks and an internal serial-in-parallel-out register. They also assume that the padding is completed outside of

the hash cores and thus will not affect the speed of the core hash function. The results presented are post-place and route.

A second group to provide an FPGA evaluation were Kneževiċ *et al.* [137] using the hash function wrapper of Kobayashi *et al.* [109]. They define a 16-bit data bus best suited for their target evaluation board, the SASEBO GII. Similar to Homsirikamol *et al.*, they leave the padding to be completed outside of the hardware module. They compare all 256 bit version candidates. The results presented are measured from the FPGA and include power and energy results.

Another group provided FPGA evaluations (Guo *et al.* [138]) using the wrapper presented in [139]. They compare all 256 bit candidates, use a 16 bit data bus and exclude the padding. The results presented are post-place and route. They also presented estimated power results.

The Third SHA-3 Candidate Conference took place on March 22-23, 2012 and a number of groups again implemented the round three tweaked versions of the remaining contestants. For FPGA based designs, George Mason University [140, 141] presented a comprehensive comparison of all the candidates. They implemented each algorithm using multiple architectures based on the concepts of iteration, folding, unrolling, pipelining, and circuit replication. The major performance metrics used were throughput, area, and throughput to area ratio. Latif *et al.* [142] were another group to implement all of the round three designs and also showed their results in the form of chip area consumption, throughput and throughput per area. Similarly, Bernhard Jungk also evaluated the designs [143] and presented results for lightweight area-efficient designs and used the area and throughput-area metric.

Kaps *et al.* [144], also of George Mason University, also presented implementations on resource-constrained platforms, adhering to an area constraint of 800 slices or 400-600 slices and one Block RAM. This group measured the power consumption of all 256 versions of the implementations on Spartan-3 to evaluate the efficiency of the algorithms.

The UCL crypto group in Louvain also presented resource constrained analysis of the round three designs [145] using a similar approach to the one used in [144]. Area, throughput and efficiency of the 512-bit variant results were obtained for Virtex-6 and Spartan-6 devices.

4.8.1 Comparison of Round Three Results

To allow a snapshot of how the results presented in this thesis stand, a comparison against some the other implementations from Section 4.8 is presented. Throughput and throughput-area was used as the standard metric given by the other implementations. Where this was not completely clear, the throughput metric was examined and the best estimate was used to classify the work. The load time cycle count was also removed as none of the other designers appeared to take it into account (although it is referenced in their respective interfaces). The number of clock cycles was recorded using a Microblaze XPS timer as described in Section 2.8.1. The area was taken for only the hash function without the wrapper interface at Place-and-Route and the clock frequency was taken from the Post-Place-and-Route static timing report inclusive of the wrapper.

Table 4.11 presents area, throughput and throughput-area results for the Virtex-5 implementations[iv].

The short message metric gives a clearer more realistic set of results. While the long message metric just takes into account the round time, the short message metric takes into account initialisation and finalisation stages. For example, Skein and Grøstl both have finalisation rounds of length equal to the round time given for long messages, thus at the very least, halving the throughput for a single block message. JH, likewise requires two message blocks to process a single 512-bit message, thereby doubling the process time and again halving the throughput. Therefore, it is suggested that more weight is associated to the short message metric. However most groups only presented results for the long message metric (probably due to the simplicity of calculation).

Gaj *et al.* [141] presented many results using their ATHENa (Automated Tool for Hardware EvaluatioN) [136] database. The best results as given in [141] are presented here. For a breakdown of the variants and their methodologies the interested reader is directed to [140, 141]. The metrics use by Gaj would closer reflect those of long messages than the time required to process a single message block, although finalisation clock cycles are included. Load times are not included, and

[iv] In an effort to reduce clutter, only the similar Virtex-5 round three implementations were taken of the many results presented by the different groups. Those implemented using other than slice logic, i.e. with BRAM blocks, were also neglected due to the inability to perform fair comparison

short message metrics are not given. Results including padding (as presented in the appendix) are not considered here as other groups did not include them. The clock count here is estimated as a metric of *number of rounds* is used in the documentation.

The results from Latif [142] are given for both 256 and 512-bit digests. Although it is not explicitely stated, it is clear from the paper that the results are taken for one execution of the compression function, i.e. long messages, and load times are not included.

The results from Kaps [144] are only given for SHA-3 variants with 256-bit digest, as they felt that these are the most likely variants to be used in area-constrained designs. The low area can clearly be seen from the area column when compared against the others. Both long and short throughput results are given, with the short messages including the load times, initialisation and finalisation, and the long messages excluding initialisation and finalisation but including the load time. To allow standardisation between the results, the load times were removed (where possible; in certain cases, i.e. JH and Keccak, processing takes place during the *load in* cycle, and in other cases, i.e. Blake, processing takes place during the *load out* cycle) and the results were recalculated in line with the comparison.

The results from Jungk [143] are only given for SHA-3 variants with 256-bit digest. Similar to Kaps, they are lightweight implementations. Although it is not explicitely stated, it is clear from the paper that the results are taken for one execution of the compression function, i.e. long messages and load times are not included.

Two things are immediately clear from Table 4.11. Firstly that the designs presented in this thesis are not as optimised for throughput or area when compared to the others (Throughput/Area in the case of Gaj and Latif, minimal area in the case of Jungk and Kaps), and as such, the performance metrics show this. The other thing that is clear is that the work presented in this thesis attempts to cover all possible implementation versions, whereas the other implementations, due to their nature of being specifically tailored, are only examined and presented for their percieved optimal operating performance (i.e. 256-bit for area constrained, long message variant to present throughput). Therefore, while the results presented here are not the most area or throughput efficient, they do

Table 4.11: Comparison of SHA-3 Round 3 Implementations

	256					512				
	Clk Count #Cycles	Area (slices)	Max.Freq (MHz)	TP (Mbps)	TPA (Mbps/slice)	Clk Count #Cycles	Area (slices)	Max.Freq (MHz)	TP (Mbps)	TPA (Mbps/slice)
Blake S	42	1118	118.5	1444.5	1.292	58	1718	102.4	1807.8	1.05
Kaps [144]	277	271	253.8	469.11	1.731	-	-	-	-	-
Blake L	42	1118	118.5	1444.5	1.292	58	1718	102.4	1807.8	1.05
Gaj [141]	14	3495	210.04	7547	2.16	272	386	148.81	560	1.45
Latif [142]	28	1382	125.55	2290	1.66	32	2582	100.02	3210	1.24
Jungk [143]	115	374	163	725	1.94	-	-	-	-	-
Kaps [144]	258	271	253.8	503.66	1.858	-	-	-	-	-
Grøstl S	10	2391	200.85	10283.5	4.3	14	4845	167.311	12237.6	2.524
Kaps [144]	720	313	317.4	225.7	0.721	-	-	-	-	-
Grøstl L	10	2391	200.85	10283.5	4.3	14	4845	167.311	12237.6	2.524
Gaj [141]	10	2971	243.724	12479	4.2	58	2336	272.777	4816	2.06
Latif [142]	10	1419	121.03	6200	4.37	14	2523	101.22	7400	2.94
Jungk [143]	160	368	305	975	2.64	-	-	-	-	-
Kaps [144]	357	313	317.4	455.2	1.45	-	-	-	-	-
JH S	45	1291	311.72	3546.6	2.74	45	1291	311.72	3546.6	2.74
Kaps [144]	1618	183	250.6	79.29	0.433	-	-	-	-	-
JH L	43	1291	311.72	3711.6	2.87	43	1291	311.72	3711.64	2.875
Gaj [141]	43	982	416.146	4955	5.05	43	992	393.546	4686	4.72
Latif [142]	42	865	287.44	3500	4.05	42	888	292.48	3570	4.02
Jungk [143]	168	377	278	847	2.24	-	-	-	-	-
Kaps [144]	800	183	250.6	160.38	0.88	-	-	-	-	-
Keccak S	25	1358	246	10705.9	7.883	25	1361	242.54	5588.1	4.10
Kaps [144]	2395	275	259.7	117.97	0.429	-	-	-	-	-
Keccak L	24	1358	246	11152	8.212	24	1361	242.54	5820.96	4.27
Gaj [141]	24	1369	294.204	13337	9.74	24	1320	317.158	7612	5.77
Latif [142]	24	1333	275.56	12490	9.37	24	1197	263.16	6320	5.28
Jungk [143]	200	379	159	864	2.29	-	-	-	-	-
Kaps [144]	2392	275	259.7	118.1	0.43	-	-	-	-	-
Skein S	46	1786	102.283	1138.45	0.637	46	1786	102.283	1138.45	0.637
Kaps [144]	4461	246	176.6	19.39	0.078	-	-	-	-	-
Skein L	19	1786	102.283	2756.25	1.543	19	1786	102.283	2756.25	1.543
Gaj [141]	19	1858	198.098	5338	2.87	19	2026	199.561	5378	2.65
Latif [142]	19	1492	113.78	3070	2.05	19	1544	113.6	3060	1.95
Jungk [143]	584	519	299	262	0.5	-	-	-	-	-
Kaps [144]	2366	246	176.6	38.21	0.155	-	-	-	-	-

tend in a lot of cases to give similar enough TPA results to those of the Throughput/Area compared designs, and in all cases except Blake give a better TPA than the minimal area compared designs. Similarly, Table 4.12 presents the power and energy comparison results. The only other groups to present power results were Kaps *et al.* [144] for round 3 and Kneževiċ *et al.* [137] and Guo *et al.* [138] for round two. In all of the cases, power and energy results were only examined for the 256-bit variant (additionally Guo only presents power and not energy results). The results presented here can be most directly compared against the round two results of Kneževiċ, as while a different method was used to calculate the energy, similar hardware, i.e. the Sasebo, was used. The round three results from Kaps cannot be so directly compared against the results presented in this

work, as both a different chip (Spartan-3)[v] and design methodology (constrained implementations) were used. Also, only a metric for measuring the power and energy for long messages is used. However as the only other set of round three power and energy results available, a presentation and comparison are attempted.

Table 4.12: Power Comparison of SHA-3 Implementations

	Area (slices)	Power mW	Energy nJ/bit	AEP nJ/bit.slice	Area (slices)	Power mW	Energy nJ/bit	AEP nJ/bit.slice
Blake S	1118	138.5	1.173	1311.4	1718	161.9	1.280	2199.0
Kneževiċ [137]	1660	270	0.98	1626.8	-	-	-	-
Blake L	1118	121.4	0.964	1077.7	1718	168.4	1.020	1752.3
Kneževiċ [137]	1660	270	0.49	813.4	-	-	-	-
Kaps [144]	631	42.9	0.486	306.6	-	-	-	-
Guo [138]	1612	139.8	-	-	-	-	-	-
Grøstl S	2391	203.5	1.110	2654.0	4845	229.3	0.984	4767.4
Kneževiċ [137]	2616	310	1	2616	-	-	-	-
Grøstl L	2391	210.8	0.611	1460.9	4845	249.1	0.750	3633.7
Kneževiċ [137]	2616	310	0.25	645	-	-	-	-
Kaps [144]	766	39.8	0.605	463.43	-	-	-	-
Guo [138]	3308	364.36	-	-	-	-	-	-
JH S	1291	128.2	1.699	2193.4	1291	128.6	1.780	2297.9
Kneževiċ [137]	2661	250	1.6	4257.6	-	-	-	-
JH L	1291	125.2	0.829	1070.2	1291	127.7	0.850	1097.3
Kneževiċ [137]	2661	250	0.8	2128.8	-	-	-	-
Kaps [144]	558	43.8	1.369	763.9	-	-	-	-
Guo [138]	2406	225.93	-	-	-	-	-	-
Keccak S	1358	128.8	0.511	693.9	1361	131.7	0.751	1022.1
Kneževiċ [137]	1433	290	1.16	1662.2	-	-	-	-
Keccak L	1358	107.7	0.397	539.1	1361	134.6	0.634	862.8
Kneževiċ [137]	1433	290	0.29	415.5	-	-	-	-
Kaps [144]	766	27.7	1.22	934.5	-	-	-	-
Guo [138]	1556	147.99	-	-	-	-	-	-
Skein S	1786	182.0	1.390	2482.5	1786	189.4	1.574	2811.1
Kneževiċ [137]	1370	300	1.86	2548.2	-	-	-	-
Skein L	1786	130.4	0.651	1162.6	1786	159.0	0.812	1450.2
Kneževiċ [137]	1370	300	0.47	643.9	-	-	-	-
Kaps [144]	766	36	3.372	2616.6	-	-	-	-
Guo [138]	788	88.65	-	-	-	-	-	-

A general rule of thumb to get the timing or energy results per bit is to divide the result by the

[v] The Kaps area and throughput results presented earlier in this section are for Virtex-5 similar to this work. However power measurements were only taken for the Spartan-3. Again, the logic only version is chosen for comparison.

message size. Table 4.12 presents the energy per bit results calculated from Table 4.10. The Area-Energy Product is then recalculated and compared against the other implementations.

The results show that Kneževič [137] and Guo [138] have a higher power result in the majority of cases but Kneževič has similar energy per bit results. This difference in results, while implementation dependent, decreases for the short messages. This is due to the fact that the power results presented in this work, for both long and short messages are inclusive of *loading*, *initialisation*, *hashing* and *finalisation* stages, while those of Kneževič are taken just for *hashing* in the case of long messages, and are exclusive of *loading* for short messages.

So while the results are more similar for short than for long messages, one would expect the results presented in this work to result in lower energy for a similar throughput as the average power is lower if all stages were included. For example, using the FSL bus requires 16 *load in* 32-bit message blocks (18 in the case of Blake and 19 in the case of Skein (i.e. 512-bit input message (16×32) plus 96-bit counter (3×32))) each of two clock cycles. *load out* in all cases requires 8×2 for 256 and 16×2 for 512. When this is added to the clock count to process a message, the overhead results in a large additional cost (approximately $\times 2$ extra, e.g. Skein requires 46 clock cycles to hash a 224/256-bit message and 54 clock cycles to load the message) to the iteration time and therefore also to the energy calculation.

The round three results from Kaps et al. [144], obviously show the lowest area and energy per bit results, as the implementations are for area constrained devices with low power, low area and low throughput. Only the long message metric was calculated, however, and the AEP results show a similar enough pattern as that of this work and Kneževič, with Blake, Grøstl and Keccak performing strongly.

4.9 Conclusions

In this chapter a method for examining the performance of hardware implementations of hash functions was presented, specifically those selected for round two and round three of the SHA-3

hash competition. While some of the round two designs contained similar features, no two designs could be implemented using the same methodology. Starting from a basic iterative architecture various implementation methods were examined and an optimal solution was explored for each of the hash functions.

A brief examination of the various platforms used along with an overview of the different FPGA implementations for the round two designs was presented. This overview included the interface, communications and padding protocols for the round two implementations used in this work. It was shown how the interface was important not only to enable fast testing, but also to allow a uniform and constrained environment in which all hash functions could be tested equally.

A brief description of the hash designs and their implementation methodologies was then provided and a presentation of the results was presented. These results were then compared against the selection process used by NIST to select the finalist designs for round three of the competition. It was seen that the three best performing hash functions presented here, Keccak, Grøstl and JH, were also selected by NIST as finalists. Two other strong performers Skein and Blake were also selected for round three by NIST.

An overview was then provided of the updated designs for round three, and area, throughput, area-throughput, timing, power and energy results of these implementations were presented. Keccak continued to perform the best for the metrics chosen and indeed was subsequently pronounced the overall winner of the SHA-3 competition by NIST.

Finally a comparison of the current state of the art and how it compares against the work presented here was provided. Interestingly, it was seen that while Keccak performed best using throughput and throughput-area metrics, its performance was not so good compared to the other designs when implemented for area constrained systems.

5

Cryptographic Processor

5.1 Introduction

The Internet, while widely used, is not very well understood from a security point of view for the average user. Complex software hides many security flaws and adversaries can actively exploit these flaws to access supposedly secure data. Due to this complex nature, networking and internet protocols have been developed along with frameworks, which allow both standardisation and transparancy between the different layers of a network. This allows communicating entities to use whichever algorithms provide security appropriate for any given communication. The Internet protocol suite [146] is the set of communications protocols used for the Internet and similar net-

works. It is commonly referred to as TCP/IP due to its two main protocols; Transmission Control Protocol (TCP) and Internet Protocol (IP). It consists of four abstraction layers, each with its own protocols [147, 148]:

- The **Link layer**. Contains communication technologies for a local network.
- The **Internet layer (IP)**, or network layer. Connects local networks
- The **Transport layer (TCP)**. Handles host-to-host communication.
- The **Application layer**. Contains all protocols for specific data communications services on a process-to-process level.

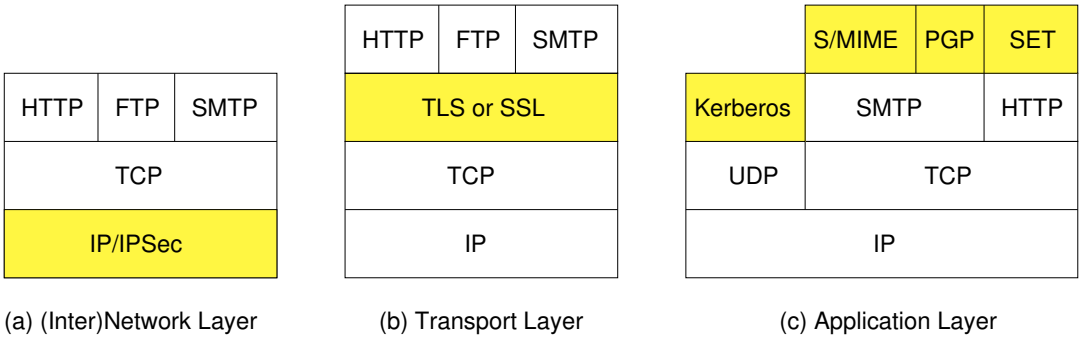


Figure 5.1: Security in the TCP/IP stack

Figure 5.1 highlights some of the various security measures in the different layers of TCP/IP. SSL and TLS are two cryptographic protocols used to encrypt network connections at the Application Layer for the Transport Layer, using asymmetric cryptography for key exchange, symmetric encryption for confidentiality, and message authentication codes for message integrity. SSL was developed by Netscape and was made publicly available in 1994 [149] and after various updates, formed the basis of TLS [150, 151] (i.e. the first version of TLS can be viewed as SSL v3.1).
The Secure/Multipurpose Internet Mail Extensions (S/MIME) [152], Pretty Good Privacy (PGP) [153] and Secure Electronic Transaction (SET) protocols can be used to send messages confidentially by combining symmetric-key encryption and public-key encryption. Message authentication and

integrity checking is performed using digital signatures. The most important part of the protocols from a security point of view is the secure key agreement protocol containing entity authentication and key distribution [154–156]. NIST state that *each entity in an authentication exchange must use an approved digital signature algorithm to generate and/or verify digital signatures* and *each entity acting as a claimant must be bound to a public/private key pair* [157].

The goal of this chapter is to present a *cryptographic architecture* that is capable of supporting multiple types of cryptographic algorithms and architectures. For a more general understanding of the cryptosystem, an overview of a commonly used application for performing public key cryptography, namely the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [158], is presented. Using this algorithm, sensitive documents are made secure by encrypting and signing in real time using the proposed cryptographic hardware. The background theory behind the ECDSA is presented in Section 5.2, while Section 5.3 presents the mathematical background.

Section 5.4 presents another component of ECDSA, namely the *random key generation*. Various forms of entropy are proposed and a pseudo random number generator, *Fortuna*, is presented, along with an implementation method.

The architecture also necessarily needs to support the underlying field operations performed by ECC. In addition to this, the curve parameters, algorithm used, number of arithmetic units and key size all are user definable prior to runtime, thus enabling flexibility in the selection of both the underlying algorithm, the security level and the area-throughput requirements. Additionally, the cryptographic architecture can also support all of the SHA-3 algorithms and their variants. These cryptographic blocks can then be used along with ancillary components as building blocks for cryptographic protocols, e.g. authentication, key exchange, and other cryptographic applications as shown in layer 5 of the model in Figure 1.2. Section 5.5 describes a cryptographic processor based on *Microblaze*. The insertion of the hash functions described in Chapter 4 and the elliptic curve processor described in Chapter 3 as *hardware accelerators* or *SOC* into the Microblaze are also described here, followed by a brief description of the additional coordinate conversions required for converting between domains.

Section 5.6 investigates the *performance* of the various *ECC* algorithms presented in Chapter 3 in *software* and in a *hardware-software co-design* implementation. This allows a comparison between software only, hardware-software co-design and the dedicated hardware results presented in Chapter 3.
Section 5.7 presents an implementation of the ECDSA using the previously described blocks and results are given. Comparisons to the current state of the art are presented in Section 5.8. Finally, Section 5.9 concludes the chapter.

5.2 Background to Signature Algorithms

In this section an example application is described to show the workings of the cryptographic processor, namely the Elliptic Curve Digital Signature Algorithm. It was described in Section 2.5.6 that digital signature schemes provide the digital equivalent of handwritten signatures. By first signing the message with their private key, prior to encrypting it with the recipients public key, the receiver can verify the authenticity of the message using the senders public key. Essentially, the sender performs the digital equivalent of a handwritten signature.
For digital signing:

- Alice computes the signature $s = S_A m$ for a message $m \in M$, where M is the set of messages, S is the set of signatures and S_A is the transformation from M to S
- Alice sends the pair (m, s)

For digital verification:

- Bob calculates the verification function V_A, based on Alices public key
- Bob receives the pair (m, s) and performs the calculation $r = V_A(m, s)$
- Bob accepts the signature and message as being from alice if r is true, and rejects if false

As well as providing security against adversaries, digital signatures are also used to provide data origin authentication and timestamping, data integrity and non-repudiation. Alice cannot deny

that she sent a message and Bob cannot change the contents of that message after receiving and subsequently verify that it originated from Alice if it includes a valid signature.

5.2.1 The Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve version of the DSA. It was first proposed by Vanstone [158] in 1992 in response to the NIST request for public comments on the first proposal for DSS. As its security is based on the computational intractability of the ECDLP, as presented in Section 2.5.5, it is considered significantly more secure than the DSA for an equivalent security level. The ECDSA is the most widely standardised elliptic curve based signature scheme, used in ANSI X9.62 [38], FIPS 186-3 [71], IEEE 1363-2000 [39], ISO/IEC 15946-2 [40]. The principles of the DSA can be applied to the Elliptic Curve Digital Signature Algorithm (ECDSA). Again, it includes a signature generation process to generate a digital signature on the data and a signature verification process to verify the authenticity of the signature, as shown in Figure 5.2. The private key is used in the signature process, and the public key is used in the verification process, thereby stopping fraudulent signatures.

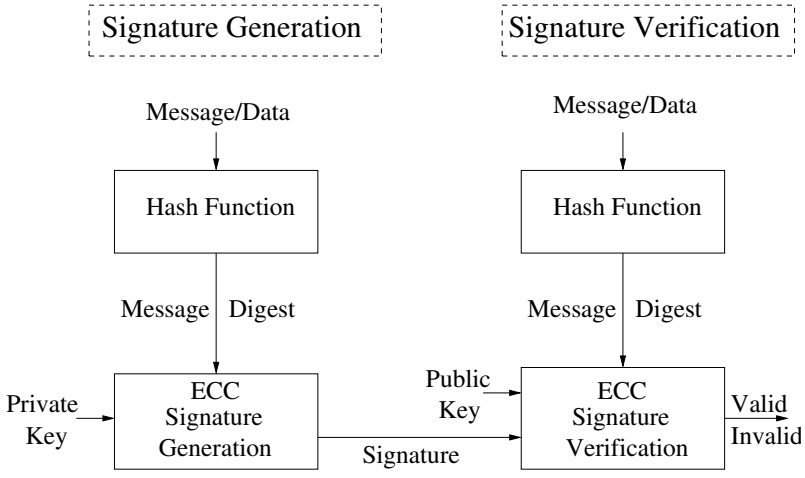
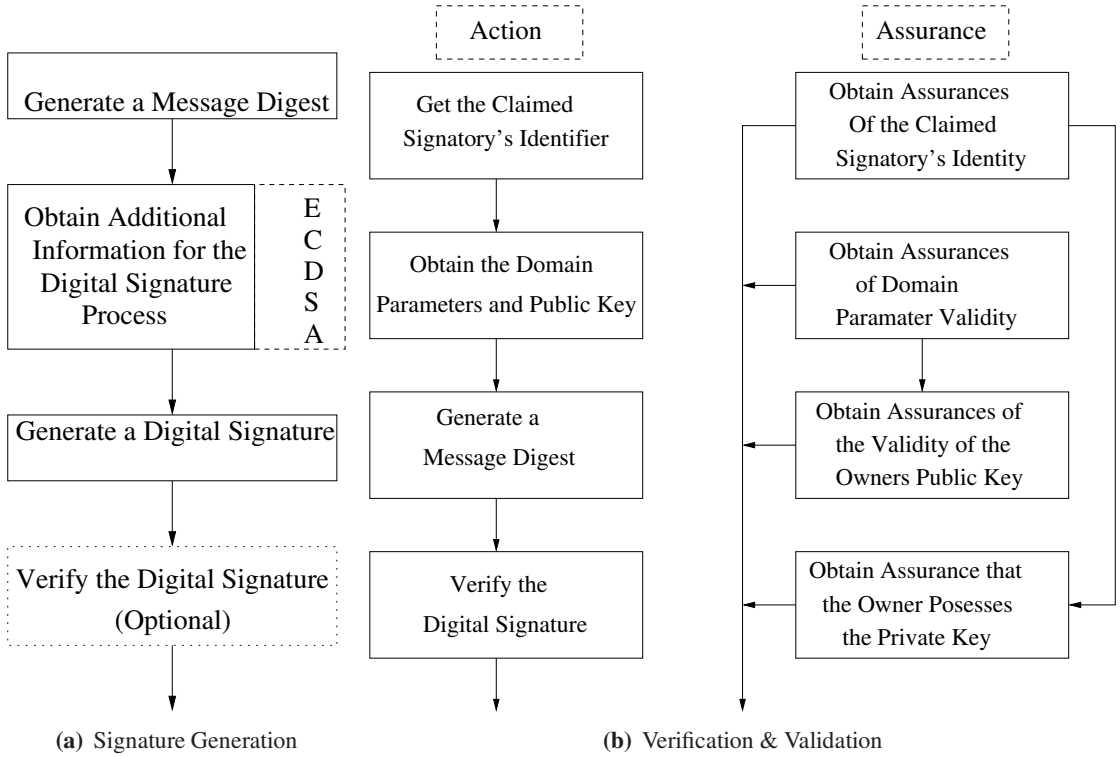


Figure 5.2: Digital Signature Process

Figures 5.3(a) and 5.3(b) give an indepth description of the signature generation and signature verfication blocks from Figure 5.2.



(a) Signature Generation **(b)** Verification & Validation

Figure 5.3: Elliptic Curve Digital Signature Algorithm

5.2.2 ECDSA Domain Parameters

The domain parameters for ECDSA consist of a suitably chosen elliptic Curve E defined over a finite field $\mathbb{F}_q$ of characteristic p, and a base point $P \in E(\mathbb{F}_q)$. The domain parameters are comprised of:

- The order of the underlying field is either $q = p$, an odd prime, or $q = 2^m$.
- Having selected an underlying field $\mathbb{F}_q$, a sufficiently large prime n ($n > 2^{160}$) is selected.
- An indication FR (field representation), of the representation used for the elements of $\mathbb{F}_q$
- A cryptographic hash function H of bitlength $\leq n$
- Two field elements a and b which define the equation of the elliptic curve E over $\mathbb{F}_q$

5.3 Implementing ECDSA

Before implementing ECDSA, several basic choices need to be considered:

- Type of underlying finite field $\mathbb{F}_q$
- Field representation (e.g. polynomial or normal basis)
- Type of elliptic curve E over $\mathbb{F}_q$
- Cryptographic hash function H

Algorithm 24: ECDSA Signature Generation

input : Domain Parameters $D = (q, FR, S, a, b, P, n, h)$, private key d, message m
output: Signature (r, s)

Select $k \in r[1, n-1]$
$kP = (x_1, y_1)$
$x_1 \mapsto \bar{x_1}$ (conversion to integer)
$r = \bar{x_1} \bmod n$
if $r = 0$ then restart
$e = H(m)$
$s = k^{-1}(e + dr) \bmod n$
if $s = 0$ then restart
return (r, s)

Algorithm 24 gives the ECDSA signature generation, Breaking it down into its component parts gives:

5.3.1 Key Pair Generation

The private and public key required are generated as follows:

$$\begin{aligned} k &\in r[1, n-1] \\ Q &= kP \end{aligned} \tag{5.1}$$

where, k, the signer Alices A private key, is a randomly selected integer from $[1, n-1]$ and Q is Alices public key. This computation requires one random integer generation and one elliptic curve

point multiplication operation.

5.3.2 Signature Generation

To generate the signature for a message, m, Alice computes:

$$\begin{aligned} k &\in r[1, n-1] \\ R &= [kP]_x \bmod n \\ e &= H(m) \\ s &= k^{-1}(e + dr) \bmod n \end{aligned} \tag{5.2}$$

The signature for the message m is (r, s). If s is equal to zero then Alice must restart the process. $[kP]_x$ defines the x-coordinate of the result point of kP. Alices private key k is used as part of the signature generation. As such, other entities cannot produce the same signature as they have a different value for k. Message signing requires one random integer generation, one hashing operation and elliptic curve computation operations, namely inversion, addition and two modular multiplications.

5.3.3 Signature Verification

Algorithm 25 gives the ECDSA signature verification, where q is the field size, FR is an indication of the basis used, a and b are two field elements that define the equation of the curve and H is a cryptographic hash function of bitlength $\leq n$.

Algorithm 25: ECDSA Signature Verification

input : Domain Parameters $D = (q, FR, S, a, b, P, n, h)$, public key Q, message m, signature (r, s)
output: Accept/Reject

Verify $(r, s) \in R[1, n-1]$, Reject if false;
$e = H(m)$;
$w = s^{-1} \bmod n$;
$u_1 = ew \bmod n$;
$u_2 = rw \bmod n$;
$X = u_1 P + u_2 Q$;
Reject if $X = \infty$;
$x_1 \mapsto \bar{x_1}$;
$v = \bar{x_1} \bmod n$;
if $v = r$ then Accept;
else Reject

The verifier Bob B, verifies Alices signature (r, s) for the message m using Alices public key Q:

$$\begin{aligned} e &= H(m) \\ w &= s^{-1} \bmod n \\ u_1 &= ew \bmod n \\ u_2 &= rw \bmod n \\ X &= [u_1 P + u_2 Q]_x \end{aligned} \tag{5.3}$$

If $X = r$ then Bob accepts the signature. Otherwise it is rejected. The verification stage requires one hashing operation and elliptic curve computation operations, namely, two point multiplications, point addition, modular inversion and two modular multiplications.

5.3.4 ECDSA Implementation Options

It is clear from above, that the main requirements for implementing ECDSA are elliptic curve cryptography, hash functions and random number generators. In this thesis, the hardware used to implement the designs is presented in Chapter 2, the different types of elliptic curves to be used

were presented in Chapter 3, the different types of hash functions were presented in Chapter 4, a field size (p_b) and a key size (k) of 192 are used throughout for each of the target algorithms, and the curve parameter selected for use is secp192r1 [81].

A number of design options were examined initially. The elliptic curve processor was implemented entirely in software using the Microblaze soft-core processor [54] and also in software with a hardware multiplier. These implementations are described fully in Section 5.6. These implementations were then compared against the previous dedicated hardware results from Chapter 3. In this way, an examination is made of the performance levels associated with each design strategy.

It was decided the hash functions were best implemented in dedicated hardware due to the large file size associated with holding and storing large amounts of data while a message is being hashed. Additionally, a method for random number generation was examined, allowing re-use of the previously implemented components. This is examined further in Section 5.4.

5.4 Random Key Generation

An important process in signature verification is the generation of random keys. In [159], NIST state that *A new nonzero secret random number k shall be generated prior to the generation of each digital signature for use during the signature generation process.* To generate the key material, random data must be acquired. There are two fundamentally different strategies for generating random bits [160]. One strategy is to produce bits non-deterministically, where every bit of output is based on a physical process that is unpredictable; this class of random bit generators (RBG) is commonly known as non-deterministic random bit generators (NRBGs), i.e. True Random Number Generators. The other strategy is to compute bits deterministically using an algorithm; this class of RBGs is known as Deterministic Random Bit Generators (DRBGs) i.e. Pseudorandom Number (or Bit) Generators.

NRBG data is not considered truly random in a cryptographic sense. Although the literature

presents an indepth analysis of pseudorandom generators [21, 161], the generators are analysed for statistical randomness only, and of little security benefit if an adversary knows the algorithm used to generate the random data.

A DRBG includes a source of entropy input. A DRBG mechanism uses an algorithm (i.e. a DRBG algorithm) that produces a sequence of bits from an initial value that is determined by a seed that is determined from the entropy input. Once the seed is provided and the initial value is determined, the DRBG is said to be instantiated and may be used to produce output. Because of the deterministic nature of the process, a DRBG is said to produce pseudorandom bits, rather than random bits. The seed used to instantiate the DRBG must contain sufficient entropy to provide an assurance of randomness. If the seed is kept secret, and the algorithm is well designed, the bits output by the DRBG will be unpredictable, up to the instantiated security strength of the DRBG.

5.4.1 Entropy

While the task of generating random or pseudorandom numbers from a seed is fairly simple, acquiring the random seed and its secrecy is inherently more difficult [162]. The measure of randomness in a message is called entropy, defined by Shannon in terms of a cryptosystem in [22, 163], and is defined as a measure of the disorder, randomness or variability in a closed system. The most common definition of entropy for a variable X is:

$$H(X) := -\sum P(X = x) log_2 P(X = x) \tag{5.4}$$

where $P(X = x)$ is the probability that the variable X takes on the value of x.

There have been many suggested sources of entropy. A *hardware random number generator* is an apparatus that generates random numbers from a physical process, rather than an algorithm. This typically consists of a transducer to convert some aspect of the physical phenomena to an electrical signal, an amplifier to increase the amplitude of the random fluctuations to a macroscopic level, and an analog to digital converter to convert the output into a digital bit. The physical process

can be derived from many sources, such as keyboard and mouse access timing, sampling from the computers microphone or even air turbulence inside the hard drive [164].
However, each source of entropy presented in the literature results in an equal amount of discussion and debate about the amount of actual randomness or possible insecurity associated with any potential source [165] and tests have been developed to perform analysis of these [166].

5.4.2 Fortuna

Fortuna is a cryptographically secure DRBG devised by Niels Ferguson and Bruce Schneier [167]. Its design leaves some choices open to the implementer. It comprises four main parts:

- The **Accumulator**; collects 32 pools of random entropy data from various sources and uses it to reseed the generator when enough new randomness has arrived.
- The **seed file manager**; maintains and stores a copy of the state of the pools to enable immediate generation of random numbers at bootup.
- The **source manager**; manages the pools of source data.
- The **Generator**; produces pseudorandom data based on the above.

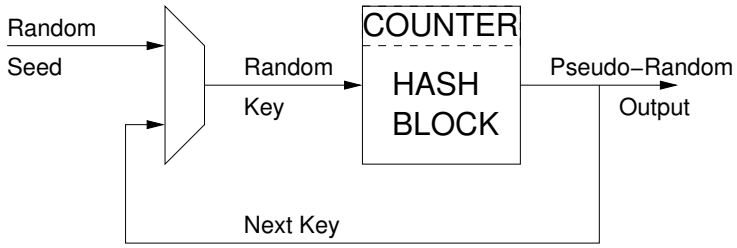


Figure 5.4: Fortuna Generator

The Generator, shown in Figure 5.4, converts fixed-size states into arbitrarily long outputs. The designers specify that any modern block cipher can be used, and indeed, as members of the same team that developed the Skein hash function, present methods whereby Skein can be used as

the generator block in a PRNG [125]. This method can also be modified for other similar hash functions containing a counter and an input, allowing them produce random data at the same speed that they hash data. The key is changed periodically and is also changed after every data request, so that a key compromise doesn't endanger old outputs. Generation is shown in Algorithm 26.

Algorithm 26: Generate Blocks

input : G Generator state; modified by this function.
k Number of random blocks of data to generate.
output: r Pseudorandom. string of $16k$ bytes

assert : $C \neq 0$
Start with the empty string;
$r \leftarrow \epsilon$;
Append the necessary blocks;
for $i = 1, \ldots, k$ **do**
  $r \leftarrow r \| H(k, C)$;
  $C \leftarrow C + 1$;
end
return r;

There are 32 pools, $P_0, \ldots, P_{31}$. Each entropy source distributes its entropy evenly over the pools in a cyclic manner. The generator is reseeded every time pool P_0 is large enough. One or more pools are included in the reseed dependent on the reseed number. After a pool is used, it is reset to an empty string. The seed file stores enough state to enable the computer to start generating random numbers as soon as boot has occured.

5.4.3 Random Number Generator Block

The hardware implementation of the RNG block is a slightly modified (dependent on the hash function used) version of Fortuna. For the SHA-3 hash functions, BLAKE offers a dedicated interface for randomized hashing [116]. The advantage of randomised hashing [168] is that it relaxes the security requirements of the hash function as the input message to the hash function can be randomised using a fresh random value for every signature, in order to free the security of digital signatures from relying on the collision resistance of a hash function. Grøstl also has a

randomized hashing mode [119]. As stated earlier, Skein can be used as a PRNG and computes the output function using the state as the chaining input [125]. JH and Keccak, while not explicitely stated in the literature, should also be able to provide adequate security for the RNG block.

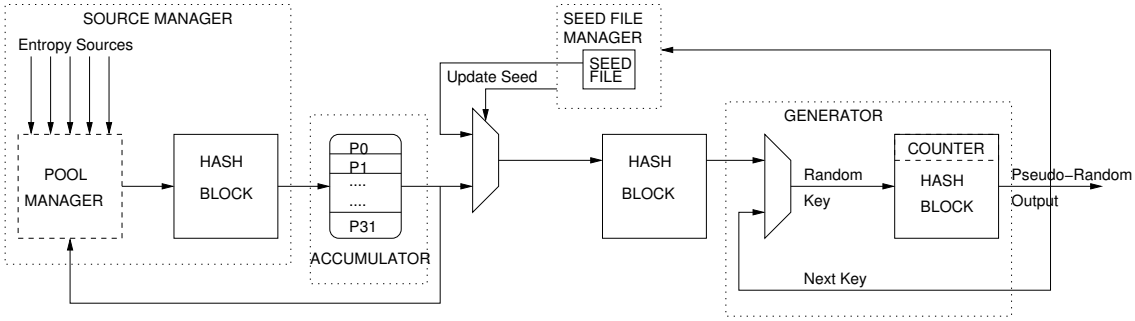


Figure 5.5: Fortuna Flow Diagram

A description of the implementation of the Fortuna algorithm is given in Figure 5.5. The software sections were written and compiled using Xilinx SDK and used the in-built Eclipse software package. The *source manager* controls the initialisation, reseed, data generation and start and stop functionality. The *pool manager* controls the management of the entropy pools. The *seed file manager* updates the seed and writes a new seed to the seed file prior to exiting.

A number of modifications were made to the Fortuna algorithm as set out in the literature. Ferguson recommends [167] that the entropy sources be of random sizes and the data hashed while added to the pools. As this would necessitate 32 running hash calculations for each of the pools of entropy, and result in a substantial use of resources an alternative method was used. A similar method to that used and shown to be statistically random in [169] was recreated here where each pool size was kept to 256 bits and the random data was set to input in 1-byte blocks. The data is appended to each pool producing a 264-bit string. This string is then hashed and returned to the pool.

The key is changed periodically and no more than 1Mb of data is generated without a key change. The generator will then change the key or check for a reseed. The key is also changed after every data request as shown in Algorithm 27. The output is generated by a call to GenerateBlocks.

Algorithm 27: Generate Random Data

input : G Generator state; modified by this function.
n Number of random bytes of data to generate.
output: r Pseudorandom. string of n bytes
Limit the output length to reduce the statistical deviation from perfectly random outputs. Ensure the length is not negative
assert : $0 \leq n \leq 2^{20}$
Compute the output;
$r \leftarrow$ first-n-bytes GenerateBlocks$(G, [\frac{n}{16}])$;
Switch to a new key to avoid later compromise of the output;
$K \leftarrow$ GenerateBlocks$(G, 2)$

Two more blocks are then generated to get the new key, to be used the next time it is required to generate data. Once the old key K has been erased, there is no way to recompute the result r. The entropy, input in 1-byte blocks results in $\frac{256}{8} = 32$ events having to be added to pool P_0 before it can be used in a reseed.

There are five main operations used to write to the hardware hash block; (i) add random data from entropy sources, (ii) write from a seed file, (iii) write from pools (iv) write key and (v) write next key. The seed file of Fortuna ensures that random number generation can begin immediately at startup, and each block of generation is as fast as the underlying hash function. Through re-use of the hash function, the area and complexity of the design is further reduced, resulting in a cost of two or three iterations of the hash function per output block of pseudo-random data. There is a bottleneck due to the 32-bit FSL bus, requiring 16 clock cycles to load the hash block for the seed and generator blocks and 18 clock cycles to load the entropy hash.

The different sources of entropy are considered outside of the scope of the work presented here and it is sufficient to say that external dedicated hardware or software would be used for the collection and safeguarding of these pools prior to their input to the generator block[i]. It must be noted however that using this method of not gathering the entropy internally in the FPGA could lead to insecurity if an eavesdropper can get access to the data line between the computer and the FPGA when the entropy is being input. Another possible source of attack is if an attacker can gain access

[i]For examples of methods of performing this on FPGA see e.g. [170–173].

to the CPU usage of the source manager process and detect the function being performed, then this theoretically could be used to estimate the fixed amount of data being added to the pools and hence the pool size. The attacker could then potentially deduce when the generator is (or more importantly, is not) reseeded, thereby deducing the state of the generator by monitoring the CPU.

5.5 Crypto Processor Using Microblaze

The Microblaze soft-core processor [54] is based on a 32-bit RISC architecture and can be implemented on any of the Xilinx FPGA families. Implementing a processor on the FPGA increases the flexibility of the overall design and makes communication with the FPGA a simple operation. For some applications the Microblaze processor may not provide high enough performance, such as computationally intensive parts of ECC. In this case operations that are slow to perform in software may be performed much faster in dedicated hardware on the FPGA.

The Microblaze, described in Section 2.7 is used as shown in Figure 5.6 to build the Cryptographic processor. The design comprises a *Random Number Generator* (RNG) block, a *Hash* block, an *Elliptic Curve Processor* (ECP) block, an *XPS Timer* and ancillery components. There is also a *Project Peripheral Repository* (PPR) where unused components are stored to allow easy switching between specified components for a given implementation. It can also be seen how additional blocks can be easily and quickly added to the repository, and thus to the design. For example, an AES block would allow SSL/TLS to be implemented.

5.5.1 Hash Block

For the hardware implementations presented here, hash function based on the round three designs as given in Chapter 4 were implemented for the Microblaze (see Section 2.7.1). The hash block comprises a user selected hash function. The hash designs used here are the 256-bit designs, but no changes are required to the overall Microblaze design to allow the 512-bit designs be used. Each Hash design is stored locally in the project peripheral repository, and the IP is added prior to

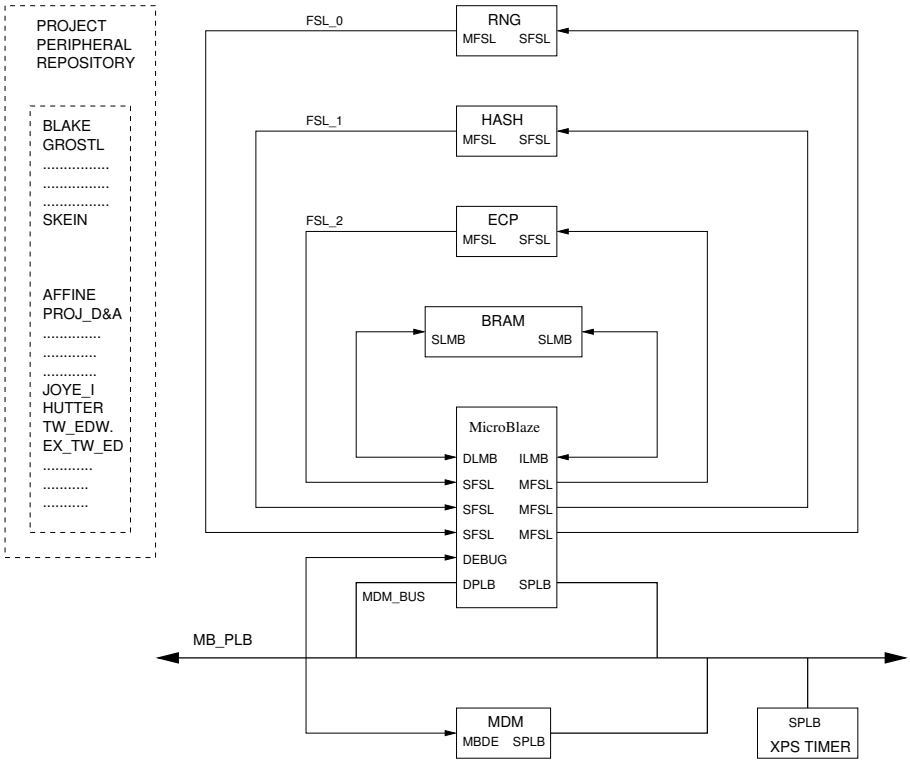


Figure 5.6: Cryptographic Processor using Microblaze

runtime.

This necessitated some changes to the round two wrapper design. Firstly, due to the fixed size of the FSL bus FIFO, it was easier to implement the padding in software. Methods were initially tested whereby the first message block defined the message size and counter, however this led to both an unnecessary extra latency in the load time along with extra logic increasing both the complexity and the area. As such, software was generated in C to generate the padding prior to the message being loaded to the hash function. The connections between the Microblaze and the wrapper from Section 4.5.1 are shown in Table 5.1.

It was described in Section 2.7.2 that the input and output are read in 32-bit blocks on the FSL bus and require two clock cycles[ii], a read and an acknowledge, per block. Therefore each input

[ii] In actuality, only the first block requires two clock cycles with subsequent blocks only requiring one. However, in order to avoid undue complication when reading in messages of different sizes, the two clocks per message block standard was used

Table 5.1: Wrapper Interface for Hash and ECC

FSL	IO	hash	ECC
FSL_CLK	in	clk	clk
FSL_RST	in	rst	rst
FSL_S_DATA	in	d_in	d_in
FSL_S_EXISTS	in	dp_in	load
FSL_M_FULL	in	ack_out	RAM_addr_en
FSL_S_CONTROL	in	lb_in	key_en
FSL_M_DATA	out	d_out	d_out
FSL_M_WRITE	out	dp_out	done
FSL_S_READ	out	ack_in	-
FSL_M_CONTROL	out	lb_out	-

message requires $(MES/32) \times 2$ clock cycles to load, where MES is the input message size. Also communication issues occured when loading of a new message and processing of the current message occured simultaneously. In order to work around this, the load stage and the processing stage were completely separated out in respect to the round two wrapper. An example description of the timing is given in Figure 5.7.

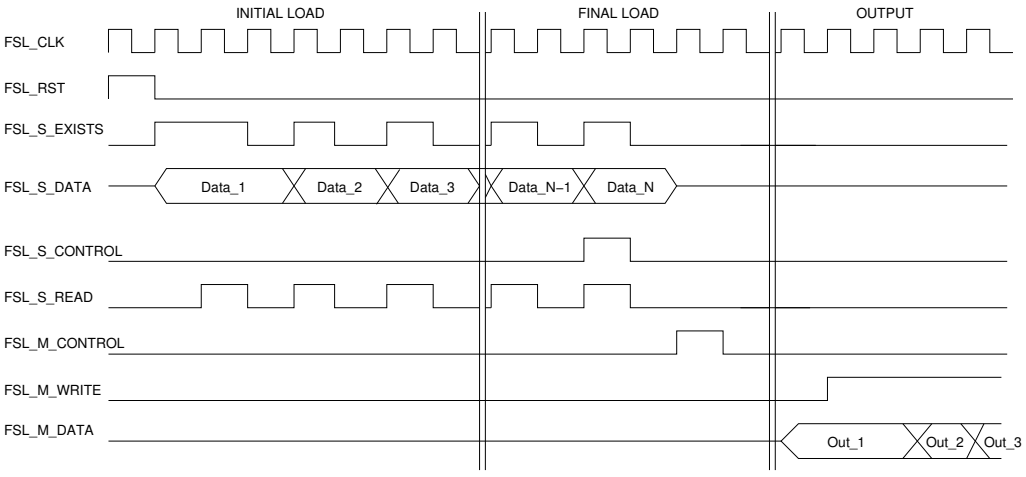


Figure 5.7: Timing Diagram for Microblaze I/O

5.5.2 Elliptic Curve Processor Block

Similar to the hash block, the ECP block uses the designs from Chapter 3 again stored locally in the project peripheral repository. As the hardware implementation of binary field operations result in carry-free logic, these fields are better in terms of speed and area for hardware. However, since ECDSA contains modular operations over $\mathbb{F}_q$ along with elliptic curve point operations, the addition, multiplier and inversion blocks defined in Sections 3.3 can be reused.

A number of points need to be addressed here as regards the designs as presented in Chapter 3. The designs themselves, in many cases require precomputations to get them to the required state. For the ECDSA, the designs necessarilly begin in the affine domain and need to be converted, e.g. as described in Section 3.2.2 and Section 3.2.3. Similarly, for the Montgomery multiplier, the values need firstly to be converted to the montgomery domain and afterward converted back as described in Section 3.3.3. This adds additional delays to the design. In most cases in the literature, these precomputations are neglected from the timing results.

The IP blocks described in Section 2.7.1 were then implemented for the ECC circuitry previously described. This involves simply defining a wrapper, also described in Table 5.1, to allow compatibility and some control logic and has a net result of a small increase in timing due to the coordinate loading in 32-bit blocks, i.e. six $\times 2$ extra clock cycles per coordinate, resulting in a mostly negligible increase when compared against the overall timing in respect to the timing results given in Table 3.9, and a small increase in area due to the control logic.

5.5.3 Coordinate Conversion

As stated above, there is a minor conversion cost associated with converting the various algorithms described in Chapter 3. Table 5.2 presents the different conversions required for each of the regular scalar multiplication algorithms presented in Section 3.8.

Twisted Edwards curves are a bit more involved and require an additional conversion from Weierstraß to twisted Edwards form along with the conversion from Jacobian projective to affine and vica versa. To convert from Weierstraß to twisted Edwards form, let E_W be a projective Weierstraß-

Table 5.2: Conversion for co-Z addition formulæ

Algorithm	Conversion	Cost
Left-to-right algorithms:		
Mont. ladder co-Z addition (Alg. 18)	Jacobian - affine	$1\mathsf{I} + 4\mathsf{M}$
Mont. ladder (X, Z)-only (Alg. 19)	homogenous - affine	$1\mathsf{I} + 2\mathsf{M}$
Mont. ladder (X, Y)-only (Alg. 22)	$\left(\left(\frac{\lambda}{Z}\right)^2 \mathrm{X}(\boldsymbol{R_0}), \left(\frac{\lambda}{Z}\right)^3 \mathrm{Y}(\boldsymbol{R_0})\right)$	$1\mathsf{I} + 9\mathsf{M}$
Signed-digit (X, Y)-only (Alg. 23)	$\left(\lambda^2 \mathrm{X}(\boldsymbol{R_0}), \lambda^3 \mathrm{Y}(\boldsymbol{R_0})\right)$ where $\lambda \leftarrow \frac{y_P \mathrm{X}(\boldsymbol{R_1})}{x_P \mathrm{Y}(\boldsymbol{R_1})}$	$1\mathsf{I} + 7\mathsf{M}$
Right-to-Left algorithms:		
Joye's double-add co-Z I (Alg. 20)	Jacobian - affine	$1\mathsf{I} + 4\mathsf{M}$
Joye's double-add co-Z II (Alg. 21)	Jacobian - affine	$1\mathsf{I} + 4\mathsf{M}$

form elliptic curve defined over $\mathbb{F}_q$.

$$E_W : \; Y^2Z = X^3 + AXZ^2 + BZ^3$$

where $A, B \in \mathbb{F}_q,\ 4A^3 + 27B^2 \neq 0$. E_W is birationally equivalent to the projective twisted Edwards curve

$$E_E : \; (3\alpha + 2t)X^2Z^2 + Y^2Z^2 = Z^4 + (3\alpha - 2t)X^2Y^2$$

where $\alpha \in \mathbb{F}_q$ satisfies $\alpha^3 + A\alpha + B = 0$ and $t \in \mathbb{F}_q$ satisfies $t^2 = 3\alpha^2 + A$, provided such α and t exist. This is precisely the condition that E_W can be represented in Montgomery form [174].

The transformation from E_W to E_E is given by

$$\begin{aligned} X_E &= (X_W - \alpha Z_W)\,(X_W + (t - \alpha)Z_W) \\ Y_E &= Y_W\,(X_W - (t + \alpha)Z_W) \\ Z_E &= Y_W\,(X_W + (t - \alpha)Z_W) \end{aligned} \tag{5.5}$$

$$\begin{aligned} (0 : 1 : 0)_W &\mapsto (0 : 1 : 1)_E \\ (\alpha : 0 : 1)_W &\mapsto (0 : -1 : 1)_E. \end{aligned} \tag{5.6}$$

The transformation from E_E to E_W is given by:

$$\begin{aligned} X_W &= X_E\left((t+\alpha)Z_E + (t-\alpha)Y_E\right) \\ Y_W &= t(Z_E^2 + Y_E Z_E) \\ Z_W &= X_E(Z_E - Y_E) \end{aligned} \tag{5.7}$$

$$\begin{aligned} (0:1:1)_E &\mapsto (0:1:0)_W \\ (0:-1:1)_E &\mapsto (\alpha:0:1)_W. \end{aligned} \tag{5.8}$$

To convert back from twisted Edwards curve to Weierstraß form is more straightforward. Let E_E be the projective twisted Edwards curve over $\mathbb{F}_q$ with coefficients a and d, where a and d are distinct and non-zero.

$$E_E : aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$$

E_E is birationally equivalent to the projective Weierstraß-form elliptic curve

$$E_W : Y^2Z = X^3 + AXY^2 + BZ^3,$$

where

$$A = -\frac{(a^2 + 14ad + d^2)}{48},\ B = -\frac{(a^3 - 33a^2d - 33ad^2 + d^3)}{864},$$

under the transformation from E_E to E_W:

$$\begin{aligned} X_W &= (5a-d)X_EZ_E + (a-5d)X_EY_E \\ Y_W &= 3(a-d)(Z_E^2 + Y_EZ_E) \\ Z_W &= 12X_E(Z_E - Y_E) \end{aligned} \tag{5.9}$$

and

$$(0:-1:1)_E \mapsto (a+d:0:6)_W. \tag{5.10}$$

The transformation from E_W to E_E is given by:

$$\begin{aligned} X_E &= (6X_W - (a+d)Z_W)(12X_W + (a-5d)Z_W) \\ Y_E &= 6Y_W(12X_W + (d-5a)Z_W) \\ Z_E &= 6Y_W(12X_W + (a-5d)Z_W) \end{aligned} \tag{5.11}$$

and

$$\begin{aligned} (0:1:0)_W &\mapsto (0:1:1)_E \\ (a+d:0:6)_W &\mapsto (0:-1:1)_E. \end{aligned} \tag{5.12}$$

It should be noted here that while all Edwards and twisted Edwards curves map to Weierstraß curves, the inverse is not true, i.e. determining whether α exists is equivalent to determining whether the curve has a twisted Edwards form. Therefore choosing curve parameters randomly can be time costly [175]. Otherwise, if starting with a Weierstraß curve with a point of order 4, the transformation from Bernstein and Lange [3] can be used. The formula for converting from Weierstraß to Twisted Edwards involves 15M and 10add using constants $i = 1/6$, $j = 1/4$ and $k = -1$, while for converting from Twisted Edwards to Weierstraß is 25M and 42add using constants $j = 1/48$ and $k = 1/864$.

5.6 Implementing ECC in Software and Co-Design

In this section the performance of the various ECC algorithms presented in Section 3.8 are investigated both in a software and in a hardware-software co-design implementation. These implementations are then compared against the previous dedicated hardware results from Chapter 3. In this way, an examination is made of the performance levels associated with each design strategy.

A design for performing elliptic curve arithmetic operations was implemented on a XUPV5-LX110T development board [49]. The maximum clock frequency of the Microblaze when implemented on the board is 125MHz. The board contains 256MB 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, the heap and stack are placed in 64kB of BRAM internal to the FPGA. This design was presented in [93].

5.6.1 Dedicated Software Results

Results are first presented for a purely software implementation where the Microblaze does not make use of the 192, 256 or 521-bit hardware multiplier. The GNU GMP library was compiled for the Microblaze and used to implement all of the algorithms. Table 5.3 shows the timing results for the three different field sizes with bit-lengths of 192, 256 and 521, with the Microblaze clocked at 125MHz. The results include all precomputations and conversion to and from affine coordinates. 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. This software implementation does not perform any computations in parallel and it can be seen that the relative speed of each algorithm follows the same trend as the hardware results in Section 3.10 when only 1 multiplier was used. 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 (Alg. 23) algorithm.

Table 5.3: Software Results

	192-bit	256-bit	521-bit
Alg.	Time (mS)	Time (mS)	Time (mS)
D&A (Alg. 3)	1060	1976	20330
ML co-Z (Alg. 18)	1083	2033	21220
Joye I (Alg. 20)	1085	2036	21286
Joye II (Alg. 21)	1058	1981	20157
ML (X,Z) (Alg. 19) (Alg. 40 &45)	1134	2141	22355
ML (X,Z) (Alg. 19) (Alg. 41 &45)	1056	1980	20683
ML (X,Z) (Alg. 19) (Alg. 42 &46)	991	1876	19299
ML (X,Y) (Alg. 22)	935	1753	17726
SD (X,Y) (Alg. 23)	**924**	**1737**	**17668**

As can be seen from the results in Table 5.3, performing large multiplications on the Microblaze takes a considerable number of clock cycles. This leads to a very slow computation time

for an Elliptic Curve point multiplication. This is particularly evident in the 521-bit case where computation times are as long as 22 seconds.

5.6.2 Instruction Set Extensions

To improve the performance of the dedicated software system, a dedicated multiplier was implemented on the FPGA. The multiplier used for this design is again the Montgomery multiplier [76] that was used in the EC processor. Adding a multiplier for hardware acceleration instead of an EC processor allows the Microblaze to use the hardware acceleration block to increase performance on any ECC algorithm that use 192, 256 or 521-bit Montgomery multiplications. This type of design is suitable when flexibility and low area are required more than absolute performance. Using the EC processor described in Section 3.3 to implement the algorithms achieves good performance, but the design uses a large amount of the available area on the FPGA. If a more flexible design is required then a trade-off is made where only the large finite field multiplications are implemented in hardware. This reduces the performance but also decreases the number of slices that are used for accelerating elliptic curve operations.

The multiplier is connected to the Microblaze through a Fast Simplex Link (FSL). The FSL bus supplies the clock to the multiplier, therefore the multiplier runs at the same frequency as the Microblaze which is 100 MHz. The maximum frequency that the multiplier could run at on a Virtex 5 FPGA is 112 MHz.

The FPGA area usage results for the entire system and the multiplier alone are shown in Table 5.4. The multiplier is connected to the Microblaze through a Fast Simplex Link (FSL). The FSL bus is 32-bits wide and allows for very fast data transfer to and from the Microblaze through the use of FIFOs. The FSL bus supplies the clock to the multiplier, therefore the multiplier runs at the same frequency as the Microblaze, which is 100 MHz in the case of the 192-bit implementation and 75 MHz for both the 256-bit and 521-bit implementations. These clock frequencies are determined by the critical path of each multiplier. 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

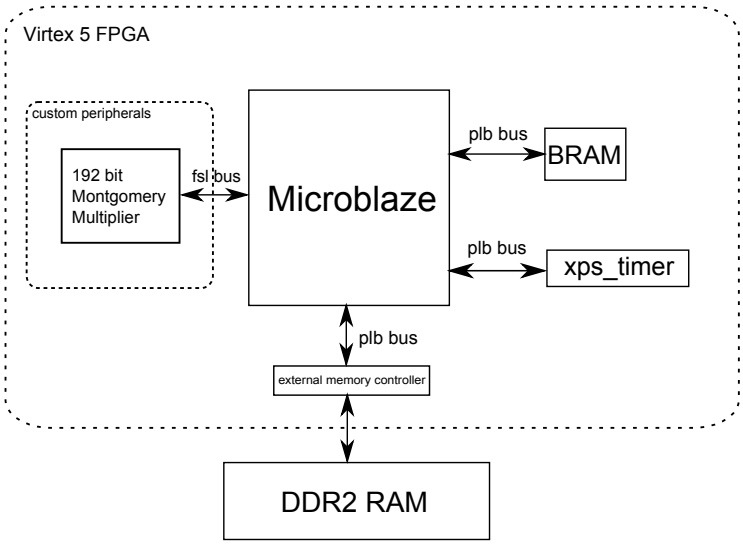


Figure 5.8: Microblaze with Hardware Multiplier

it takes to perform a multiplication. A diagram of the Microblaze with the multiplier connected is shown in Figure 5.8. The FPGA area usage results for each entire system and the multipliers alone are shown in Table 5.4. A timer was also included in the design in order to measure computation times.

Table 5.4: Microblaze FPGA usage

Design	Area (Slices)	BRAM	DDR2 RAM	DSP48E	Freq (MHz)
Microblaze 192-bit	3145	65×36k	256MB	3	100
Microblaze 256-bit	3454	65×36k	256MB	3	75
Microblaze 521-bit	3466	65×36k	256MB	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 5.5 shows the results obtained from the Microblaze with hardware acceleration. It can be seen from comparing Table 5.5 with Table 5.3 that the hardware multiplier reduces the computation time by on average 89-94% for the algorithms. The large reduction in computation time is due to the fact that the hardware multiplier performs both the multiplication and Montgomery modular reduction which are more time consuming than modular additions. The results show the

timing for performing a full computation of kP including all conversion to and from affine co-ordinates 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. In both cases the Montgomery ladder with (X, Z)-only co-Z, using the (Alg. 42 & 46) and also the (Alg. 41 & 45) variants, are the fastest algorithms and both have very similar computation times. On average the precomputations and conversion to and from affine coordinates take about 9ms in the 192-bit case. The precomputations are negligible in the software implementation. If the EC processor from Chapter 3 was used as a hardware acceleration block for the Microblaze, the precomputations alone would take longer than the actual calculation of kP.

The power consumption could not be physically measured as the design uses external DDR2 memory and the board does not allow access to the $V_{cc_{int}}$ pin of the FPGA, however, the power consumption of this design should be higher than the EC processor used in Chapter 3 due to the higher number of slices occupied.

Table 5.5: Microblaze Results

	192-bit	256-bit	521-bit
Alg.	Time (mS)	Time (mS)	Time (mS)
D&A (Alg. 3)	94	228	785
ML co-Z (Alg. 18)	113	296	1053
Joye I (Alg. 20)	114	297	1094
Joye II (Alg. 21)	96	252	956
ML(X,Z) (Alg. 19) (Alg. 40 &45)	74	203	731
ML(X,Z) (Alg. 19) (Alg. 41 &45)	**71**	**166**	692
ML(X,Z) (Alg. 19) (Alg. 42 &46)	**71**	185	**681**
ML(X,Y) (Alg. 22)	97	250	928
SD(X,Y) (Alg. 23)	87	225	845

The results show, as expected, that the more computations that are performed by dedicated FPGA slice logic, the faster the design. Using a standard Microblaze setup to perform all of the calculations is slow but may be of use in non time critical applications. For a very small increase in area usage, the performance of the Microblaze system can be increased by a factor of $\times 15$.

For the highest performance the dedicated EC processor described in Section 3.3 is the best choice

as it is designed solely for elliptic curve computations and hence does not suffer from some of the overhead of a generic processor design. As such, the hardware version explored in Section 3.3 is the one selected for use in the ECDSA.

5.7 ECDSA Design

A design for ECDSA was implemented on Microblaze. As shown in Section 5.3, the ECDSA requires a PRNG, a hash block and an elliptic curve block for its operations. A design was selected based on the information available in Tables 3.4, 3.10, 4.9 and 4.10. This design used the Montgomery ladder (XY), Alg. 22, using the EC processor described in Section 3.3, the round three version of the Grøstl hash function described in Section 4.7.1, the random number generator from Section 5.4.3 and the ancillary components described in Section 2.8.1.

For the ECC block, when using four multipliers, the extended twisted Edwards and the Montgomery ladder (XY) have the same energy usage, but the extended twisted Edwards has a faster computation time. However, the additional conversion to and from the Weierstraß to Edwards form, as shown in Section 5.5.3 results in an additional time cost. The 192-bit design was chosen for implementation here.

For the hash block, Grøstl has the shortest iteration time for long and short messages. Its in-built use of a counter along with its randomized hashing mode as described in Section 5.4.3 allows it to be re-used for the PRNG block.

In addition, the SASEBO design comprises the base Microblaze, an XPS timer, a clock generator, BRAM and some ancillary components. This selection, gives a relatively fast design, albeit at a cost of additional power and area. However all of these components can be easily swapped out and replaced with any of the other designs presented in this thesis, all stored locally in the project peripheral repository.

The most computationally intensive operation for signature generation is kP as described in Algorithm 24. For signature verification, shown in Algorithm 25, it is $X = u_1P + u_2Q$. Signature

verification is more computationally intensive due to the need to perform two elliptic curve scalar operations, compared to only one for signature verification.

An entity wishing to sign a message must first generate a private/public key pair. The private key is selected at random such that $k \in r[1, n-1]$. The public key, $Q = kP$, is then generated as described in Equation 5.1. The generation of the integer k is done using the random number generator. The computation of $r = [kP]_x \bmod n$ described in Equation 5.2 can be completed in any of the coordinate systems described in Chapter 3. However the obtained point, r, needs to be converted back to affine coordinates. The computation of $e = H(m)$ can be completed using any of the hash functions presented in Chapter 4.

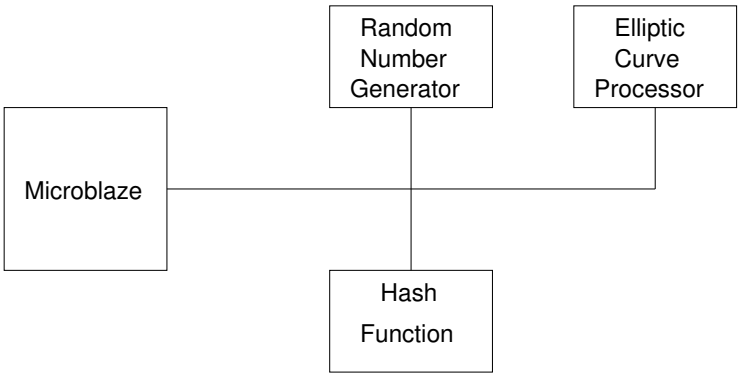


Figure 5.9: Microblaze Signature Platform

Figure 5.9 gives the general layout of the platform for generating and verifying signatures. The Microblaze processor is implemented to allow an easy method of communication between the hash, ECC and RNG blocks. In addition to this, the microblaze is also used to implement padding in the hash block and controls the entropy and managenent sections of the RNG block.

When the ECDSA processor receives a command to generate a signature, the Microblaze sends an instruction to the hash block to hash the message and an instruction to the RNG requesting an 192-bit random integer, k. Both of these values are sent to the ECP block, where the signature is generated. Along with the hashed message, e, and the random integer, k, the BlockRAM is loaded with the generator point, P, and the private key, k. The (r, s) values are returned via the FSL

bus to the Microblaze, which controls the loading of data. Similarly, for verification, the values are again loaded along with the signature pair (r, s) to the BRAM. The operations to compute $r = [kP]$ are performed mod q. All other operations are performed mod n.

5.7.1 ECDSA Results

The area results for the specified design is shown in Table 5.6. It is seen that the system takes up approximately half of the FPGA area. Additional multipliers in the ECP design do not add a large amount of area to the overall circuit. The Grøstl hash function is however the largest user of area of the hash functions implemented in this thesis. As such, this implementation method should result in approximately the largest area usage of any of the combination of cryptography blocks, assuming that the hash block is also used for the PRNG. Obviously a user using different hash functions for the PRNG and the hash block, or indeed a block cipher in CBC mode for the PRNG, will result in an additional area cost.

Table 5.6: ECDSA Total Area Usage

Component	FF used	LUT used	BRAM
System	11846	24103	6
ECP	5373	7919	10
Hash	2594	11632	-
Other	2173	2632	-
Total	21986	46286	16

The critical path of the design is through the inverter in the ECP block. Therefore the maximum operating frequency is 91 MHz. This is the standard operating frequency throughout any combination of blocks used, as the inverter is a standard component used in all of the elliptic curve processor designs for conversion to and from various domains. Table 5.7 gives the timing results for each subsequent section.

Table 5.7: ECDSA Timing using Grøstl & ML (XY)

Component	Calc. Time (mS)
PRNG	5.56
Hash	2.83
ECP (Gen.)	9.24
ECP (Ver.)	18.44

5.8 ECDSA Comparison

This section presents a performance comparison with other implementations of ECDSA. While there are a lot of software implementations, the amount of results available for hardware is again quite lacking. For example, the System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives (SUPERCOP) [132], an ECRYPT initiative for Benchmarking of Cryptographic Systems measures the performance of hash functions, secret-key stream ciphers, public-key encryption systems, public-key signature systems, and public-key secret-sharing systems, of which eBATS (ECRYPT Benchmarking of Asymmetric Systems) [176] is used to identify the most efficient public-key systems. These software implementations, while faster than hardware, require a lot of computation power and expensive dedicated CPUs. Lenstra and Verheul [177] presented analysis which concluded that for curves over $\mathbb{F}_p$, software was over 2000 times more expensive than hardware.

There are also a small number of microcontroller implementations [178, 179]. In general, these suffer from the same timing constraints as were shown in Section 5.6.2, and result in signature generation and verification times that range in the hundreds to thousands of milliseconds.

For the FPGA and ASIC implementations, in most cases, the authors neglect the pre and post-processing steps along with the hashing and simply give the core functionality, i.e. the scalar multiplications for ECC. While these results can somewhat be compared to the results presented in Chapter 3, in Table 3.5 and Table 3.10, direct comparison of the entire system performance is difficult. Glas et al. [180] present a full ECDSA system from which comparisons can be made.

Table 5.8 presents a comparison of the results. It can be seen that the design presented here per-

Table 5.8: Comparison of ECDSA and Core Functionality for FPGA

Imp.	Field Size	Hardware	Area	Max Freq. MHz	Gen. Time mS	Ver. Time mS
Full ECDSA						
Grøstl & ML (XY)	192	Virtex-5	21986 LUT (16 BRAM)	91	12.07	21.27
Glas et al. [180]	256	Virtex-5	14256 LUT	20	7.15	9.09
Core Functionality						
McIvor et. al [181]	256	Virtex-2 Pro	15755 CLB (256 Mul)	39.5	3.84	-
Orlando and Paar [182]	192	Virtex-E	11416 LUT (35 BRAM)	40	3.0	-
Sakiyama et al. [183]	256	Spartan 3S-5000	27597 LUT	40	17.7	-
Örs et al. [184]	160	Virtex-1000E	5614 slices (est.)	91.31 (est.)	42.52	-
Vliegen et al. [185]	256	Virtex-2 Pro	2085 slices	68.17	15.76	-

forms adequately when compared against the current state of the art. For the signature generation, the design maintains this performance level while also being completely reconfigurable. The field operations, the curve parameters, algorithm used, number of arithmetic units and key size all are user definable prior to runtime. A dedicated ECDSA processor could be more easily optimised for increased speed performance, but would lose this reconfigurability. The results presented here also greatly outperform the microcontroller implementations [178, 179]. The signature verification results presented by Glas et al. [180] also outperform the results presented here. This is done through a method known as Shamir's trick [37]. Instead of computing the two scalar multiplications independently in sequence, it is faster to compute them together. In the design presented in this thesis, the instruction set ROM for the EC processor is generated prior to runtime and so this method cannot be accomplished without the use of additional instruction set ROM blocks and control circuitry to select between signature generation and verification instructions.

It is clear that the reconfigurable nature of the processor presented here results in reduced speed when compared to the dedicated circuitry presented by the comparison against other ECDSA designs. However, its benefits are from its ability to implement many additional protocols, both as presented here and through the addition of further hardware blocks such as AES, resulting in a far more accomplished design, albeit with a greater speed cost associated, than those it is compared against.

5.9 Conclusions

In this chapter a cryptographic architecture that is capable of supporting multiple types of cryptographic algorithms and architectures was presented. This architecture can support the underlying field operations performed by ECC, the curve parameters, algorithm used, number of arithmetic units and key size, thus enabling flexibility in the selection of both the underlying algorithm, the security level and the area-throughput requirements. Additionally, the cryptographic architecture can also support all of the SHA-3 algorithms and their variants. These cryptographic blocks can then be used along with ancillary components as building blocks to build cryptographic protocols along with other cryptographic algorithms, e.g. some types of random number generators, and other cryptographic applications.

An implementation of a cryptographic protocol, the Elliptic Curve Digital Signature Algorithm, was presented to allow a more general understanding of the cryptosystem described. Using this algorithm, sensitive documents are made secure by encrypting and signing in real time using the proposed cryptographic hardware. The background theory and mathematical background behind the ECDSA was also presented.

A Microblaze processor used to build the cryptographic processor was also presented. The insertion of the hash functions described in Chapter 4 and the elliptic curve processor described in Chapter 3 into the Microblaze are also described, along with a brief description of the additional coordinate conversions required for converting between domains.

The performance of the various algorithms presented in Chapter 3 in software and in a hardware-software co-design implementation was investigated. This allowed a comparison between software only, hardware-software codesign and the dedicated hardware results presented in Chapter 3.

Another part of the ECDSA, random key generation was described. Various forms of entropy were examined and a pseudo random number generator, Fortuna, was presented, along with an implementation method.

Finally, an implementation of the ECDSA using the previously described blocks was presented. Results were given and comparisons to the current state of the art were presented.

6

Conclusions

6.1 Contributions of this Thesis

In this section the main contributions of this thesis are summarised. In Chapter 3 a reconfigurable architecture for a cryptographic processor was introduced. This processor was used to examine the performance of various algorithms over prime fields, $\mathbb{F}_p$, for elliptic curves and a differing number of arithmetic units supported by the processor were evaluated. Different dedicated and doubling algorithms were examined along with a relatively new special form of curve, the twisted Edwards. It was shown how parallel versions of the twisted Edwards and extended twisted Edwards far outperform the various implementations of the Double-and-Add algorithm in different

coordinate systems. This work was presented at the 5th International Workshop on Reconfigurable Computing, ARC, in 2009 [186] and provided the first implementation results for Edwards curves in hardware.

SPA secure algorithms were also examined and metrics were presented for selecting the fastest algorithms for the lowest area and energy cost. Three different methods of SPA secure algorithms were presented; dummy arithmetic instructions, unified doubling and addition and regular scalar multiplication algorithms. Point sharing was introduced as a means of speeding up these regular algorithms. With four multipliers operating in parallel, the best performance was achieved with the extended twisted Edwards and the Montgomery ladder (XY) giving the same energy usage, but the extended twisted Edwards having a faster computation time. This portion of the work was published in the Journal of Cryptographic Engineering, JCEN, in 2012 [93]. This work presented the first hardware results for the various Co-Z, combined double-add and (X,Y)-only algorithms along with providing an overall comparison between them.

Additional work based on the elliptic curve processor itself was presented at the Workshop on Special Purpose Hardware for Attacking Cryptographic Systems, SHARCS, in 2009 [187], and at the International Conference on Reconfigurable Computing and FPGAs, ReConFig, in 2009 [188].

In Chapter 4 methods for implementing and evaluating the hash functions from the NIST run SHA-3 competition were presented. The contest initially received sixty four submissions from designers worldwide. Fifty one of these designs progressing through to round one of the contest which began in November 2008. Core functionality results were presented for five of these designs during the first round of the competition. Three of the designs implemented were selected for round two of the competition. These results were presented at the Euromicro Symposium on Digital Systems Design, DSD, in 2009 [107].

An interface was presented to allow a fair and consistent evaluation of each of the designs. This wrapper was presented at the IET Irish Signals and Systems Conference, ISSC, in 2010 [127]. The first published full evaluation was carried out for each of the fourteen designs selected for round two of the competition. This was presented at the International Conference on Field Pro-

grammable Logic and Applications, FPL, in 2010 [115], and also selected by NIST to be presented at the Second SHA-3 Candidate Conference [114]. Three of the round two designs implemented consistently outperformed SHA-2 for all of the metrics selected; Keccak, Grøstl and JH. All of which were selected by NIST for the final round of the competition. All five round three designs selected by NIST were in the top eight implementations based on the work presented here.

Updated area and power results for the remaining five designs selected for the third and final round of the competition were presented on the Cryptology ePrint Archive [189]. The eventual finalist selected, Keccak, was consistently the top performing hardware design implemented in this work. Additional work in this area based on an overview of the hardware used by various teams to implement the hash functions was also presented on ePrint [45].

In Chapter 5 a cryptographic architecture that is capable of supporting multiple types of cryptographic algorithms and architectures is presented. This architecture supports the underlying field operations performed by ECC. In addition to this, the curve parameters, algorithm used, number of arithmetic units and key size all are user definable prior to runtime, thus enabling flexibility in the selection of both the underlying algorithm, the security level and the area-throughput requirements. Additionally, the cryptographic architecture can also support all of the SHA-3 algorithms and their variants. These cryptographic blocks can then be used along with ancillary components as building blocks to build bigger cryptographic algorithms and applications.

Implementations are presented for performing ECC scalar multiplication in dedicated microcontroller software, hardware and hardware-software co-design and comparisons were made. This work was also selected for publication in the Journal of Cryptographic Engineering, JCEN, in 2012 [93]. The Elliptic Curve Digital Signature Algorithm was presented and novel implementations consisting of reconfigurable blocks implemented using the previously defined elliptic curve algorithms and sha-3 hash algorithms were presented. A pseudo random number generator, Fortuna, was also designed and implemented using the aforementioned blocks.

6.2 Future Research Directions

An immediate natural addition to the work presented in this thesis is to implement the top layer from Figure 1.2, the application layer, and furthermore, to connect the circuitry directly to a network point and perform testing and analysis in a real local area network environment. A number of additional updates would also benefit from being implemented. These are broken down by chapter.

The focus of Chapter 3 was to examine the performance of various algorithms used to perform point scalar multiplication. Not taken into account in the designs presented in this thesis are cases where speed-up is acquired by setting certain curve parameters to constants, such as those described in [66, 94, 95]. While the existence of these curve parameters was acknowledged, the aim of the work presented here was to determine the underlying performance of the general case. As such, these optimised cases were not examined. Interesting future work would involve implementing these special cases in hardware to deduce any increase in speed or decrease in energy associated with them. Similarly, using Mersenne primes could also help to increase the speed of the designs [23].

Chapter 4 presents methods for implementing and evaluating the hash functions from the NIST run SHA-3 competition. The competition has recently ended with Keccak being announced the ultimate winner. With one defined standard now selected, concentrated work can now begin to optimise Keccak. However, NIST state [106] that *all five finalists would have made acceptable choices for SHA-3*. As such, while not being the SHA-3 standard, the other four designs, Blake, Grøstl, JH and Skein, are still acceptable for use as hash functions. While a lot of analysis has been completed on the hash functions in various modes of operation, i.e. low power, high throughput, etc., there is still more tweaking and different modes of operation that can be examined and implemented to improve performance for all of the round three designs.

Chapter 5 presents a cryptographic architecture capable of supporting multiple types of cryptographic algorithms and architectures. On speed-up of the cryptographic architecture, implementations using additional instruction set ROM blocks to differentiate between signature generation

and verification could be further examined to allow optimisations such as Shamir's trick [37] which allows two simultaneous point multiplications be performed, to be implemented at a cost of additional resources. Other methods which use algorithms to increase the speed and security and reduce complexity using combinations of mixed coordinates, non adjacent form (NAF) and variable length sliding window could also be examined for hardware implementations, such as the Windowing method [190], Addition Chains [191] and the Width-W NAF method [192] amongst others.

On the security of the cryptographic architecture, some further work can also be done to improve the security of the device. The security of the signature generation algorithm depends on several assumptions. From the algorithm side, that; the random number, k, is securely generated; the hash function used is cryptographically secure; and the elliptic curve discrete logarithm problem is intractable. If any of these assumptions fail then it is possible for an adversary to forge signatures. For the random number generator, the sources of entropy can be examined in further detail and best fit solutions found. In the case of the hash functions, use of any of the SHA-3 implementations is deemed by NIST to be sufficient [106]. For the ECDLP, the use of larger key and field sizes should maintain the intractability as shown in Table 2.1 and Table 2.2. As such, future work should also involve implementing the larger ECC variants presented in Section 3.11 for the cryptographic architecture. While no implementation is completely secure, this, along with the use of SPA secure algorithms should ensure adequate protection. Indeed, the co-Z formulæ also allow for the use of projective point randomization 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 sufficiently protected against DPA attacks. For DPA attacks to be applicable the key must remain constant for many executions of the algorithm. The main part of the ECDSA susceptible to DPA is in the generation of the signature, specifically the computation of $s = k^{-1}(e + dr) \bmod n$. Since the signers private key d remains constant, and r can be obtained, this operation is open to attack [193]. Masking the message e and the private key d offers sufficient protection and this can be done at a cost of an extra multiplication using k.

From a hardware security point of view there are also many sources of attack. Since the PRNG is partially implemented off chip, an attacker may be able to inject some bias into the entropy source. One alternative to the method described here is to implement the entire PRNG on-chip [170–173]. External RAM in the Microblaze also presents another location for attack. Additionally, while the work presented here concentrated on side channel protection at the algorithm level, there are many other methods for protecting against side channel attacks, such as smoothing the power trace at a circuit level using dual-rail logic, or through the use of controlled place and route [194–196]. All of these methods are included in future work as a means of further protecting the circuitry from potential adversaries. A fully secure future device should ensure that no keys should be allowed to leave the ECDSA processor. Any sensitive information transferred off-chip to the external RAM could be easily intercepted. The Microblaze processor should also not have any access to private key information. In this way, for security partitioning, only the hash function block and the Microblaze should be unsecured against all forms of attack.

Finally, it was shown that additional designs and cryptographic algorithms can easily and quickly be added through the use of the project peripheral repository. As such, additional types of cryptographic primitives and protocols not implemented in this thesis, i.e. AES [26], DES [25], RSA [30], could be added to the overall architecture, thereby allowing further complexity in applications using this architecture.

Additionally, from a design point of view, due to the large development time associated with the large amount of designs implemented in the thesis, further automated testing would result in a reduced test time for the algorithms presented. Furthermore, the work presented throughout this thesis shows that the expected results from any particular set of implementations transpired to fairly accurately mirror the measured results. As such, it was shown that the methods used to generate these expected results are sufficiently accurate as to negate the need for full further testing, and thereby further alleviate test times in future cases.

A

Appendix - Elliptic Curve Cryptography

A.1 Double-and-Add Algorithms

The following section derives the Double-and-Add algorithms described in Chapter 3 in full. The equations given here were directly used to generate the instruction set necessary for the ECC processor.

Algorithm 28: Point Doubling in Affine Coordinates

input : $P(x_1, y_1) \in \mathbb{F}_q$
output: $[2]P(x_3, y_3) \in E(\mathbb{F}_q)$

$x_2^2 = x_2.x_2, 2x_2^2 = x_2^2 + x_2^2,;$
$A = 3x_2^2 = 2x_2 + x_2, B = A + a,;$
$2y_2 = y_2 + y_2, inv2y_2, \lambda = B/2y_2,;$
$\lambda^2 = \lambda.\lambda, 2x_2 = x_2 + x_2, x_3 = \lambda^2 - 2x_2,;$
$C = x_2 - x_3, D = \lambda.C, y3 = D - y_2$

Algorithm 29: Point Addition in Affine Coordinates

input : $P(x_1, y_1);$
$Q(x_2, y_2) \in \mathbb{F}_q$
output: $P + Q(x_3, y_3) \in E(\mathbb{F}_q)$

$A = y_2 - y_1, B = x_2 - x_1, invB,;$
$\lambda = A/B, \lambda^2 = \lambda.\lambda, C = \lambda^2 - x1,;$
$x3 = C - x_2, D = x_1 - x_3, E = D.\lambda, y3 = E - y_1$

Algorithm 30: Point Doubling in Projective Coordinates

input : $P(X_1, Y_1, Z_1) \in \mathbb{F}_q$
output: $[2]P(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$z_2^2 = z_2.z_2, a.z_2^2, x_2^2 = x_2.x_2;$
$2x_2^2 = x_2^2 + x_2^2, 3x_2^2 = x_2^2 + 2x_2^2, A = (a.z_2)^2 + (3x_2)^2;$
$y_2.z_2, x_2.y_2, B = x_2y_2.y_2z_2,;$
$A^2 = A.A, 2B = B + B, 4B = 2B + 2B,;$
$8B = 4B + 4B, C = A^2 - 8B, y_2z_2.C,;$
$X_3 = (y_2z_2.C).(y_2z_2.C), 4B - C, D = A(4B - C),;$
$y_2^2 = y_2.y_2, (y_2.z_2)^2, 2(y_2.z_2)^2,;$
$4(y_2.z_2)^2, 8(y_2.z_2)^2, E = y_2^2.8(y_2z_2)^2,;$
$Y_3 = D - E, Z_3 = 8(y_2.z_2)^3,$

Algorithm 31: Point Addition in Projective Coordinates

input : $P(X_1, Y_1, Z_1)$;
$Q(X_2, Y_2, Z_2) \in \mathbb{F}_q$
output: $P + Q(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$y_2.z_1, y_1.z_2, x_2.z_1, x_1.z_2$,;
$A = (y_2 z_1).(y_1 z_2), B = (x_2 z_1).(x_1 z_2), A^2 = A.A$,;
$B^2 = B.B, B^3 = 2B.B, z_1.z_2$,;
$A^2.(z_1 z_2), C = A^2(z_1 z_2) - B^3, B^2.(x_1 z_2)$,;
$D = 2(B^2.(x_1 z_2)), C - D, X_3 = B.(C - D)$,;
$E = (B^2.(x_1 z_2)) - (C - D), A.E, F = B^3.(y_1 z_2)$,;
$Y_3 = AE - F, Z_3 = B^3.(z_1 z_2)$,;

A.2 Edwards Curves

For standard projective twisted Edwards, each PA requires 11M and 7add, while the PD requires 7M and 7add as given in Algorithm 32 and Algorithm 33.

Algorithm 32: Point Addition for twisted Edwards

input : $P(X_1, Y_1, Z_1)$;
$Q(X_2, Y_2, Z_2) \in \mathbb{F}_q$
output: $P + Q(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$A = Z_1Z_2, B = A^2, C = X_1X_2, D = Y_1Y_2,$
$E = dCD, F = B - E, G = B + E,$
$X_3 = AF((X_1 + Y_1)(X_2 + Y_2) - C - D, Y_3 = AG(D - aC), Z_3 = FG$

Algorithm 33: Point Doubling for twisted Edwards

input : $P(X_1, Y_1, Z_1) \in \mathbb{F}_q$
output: $[2]P(X_3, Y_3, Z_3) \in E(\mathbb{F}_q)$

$B = (X_1 + Y_1)^2, C = X_1^2, C = X_1X_2, D = Y_1Y_2, E = aC,$
$F = E + D, H = Z_1^2, J = F - 2H,$
$X_3 = (B - C - D)J, Y_3 = F(E - D), Z_3 = FJ$

The extended projective twisted Edwards, requires 9M and 7add for each PA, and requires 7M and 7add for each PD as given in Algorithm 34 and Algorithm 35.

Algorithm 34: Point Addition for Extended twisted Edwards

input : $P(X_1, Y_1, T_1, Z_1)$;
$Q(X_2, Y_2, T_2, Z_2) \in \mathbb{F}_q$
output: $P + Q(X_3, Y_3, T_3, Z_3) \in E(\mathbb{F}_q)$

$A = X_1.X_2, B = Y_1.Y_2, C = Z_1.T_2, D = T_1.Z_2,$
$E = D + C, F = (X_1 - Y_1).(X_2 + Y_2) + B - A, G = B + aA, H = D - C,$
$X_3 = E.F, Y_3 = G.H, T_3 = E.H, Z_3 = F.G$

Algorithm 36 give the unified formula for standard twisted Edwards, while Algorithm 37 for extended twisted Edwards. The unified single point operation, processes the same formula for both PA and PD, thereby giving it the same power trace for either operation, at a cost of 12M and 7add per point operation. The extended unified formula performs better at 9M and 7add.

Algorithm 35: Point Doubling for Extended twisted Edwards

input : $P(X_1, Y_1, T_1, Z_1) \in \mathbb{F}_q$
output: $[2]P(X_3, Y_3, T_3, Z_3) \in E(\mathbb{F}_q)$

$A = X_1^2, B = Y_1^2, C = 2Z_1^2$
$D = aA, E = B + A, F = B - D, G = C - E,$
$H = (X_1 + Y_1)^2 - A - B,$
$X_3 = G.H, Y_3 = E.F, T_3 = F.H, Z_3 = E.G$

Algorithm 36: Unified twisted Edwards Point Operation

input : $P = (X_1 : Y_1 : Z_1), Q = (X_2 : Y_2 : Z_2)$
output: $P + Q = (X_3 : Y_3 : Z_3)$

$A = Z_1 Z_2, B = A^2, C_1 = aX_1X_2, C_2 = X_1Y_2$
$D_1 = Y_1Y_2, D_2 = X_2Y_1, E = dC_2D_2, F = B - E, G = B + E$
$X_3 = AF(C_2 + D_2), Y_3 = AG(D_1 - C_1), Z_3 = FG$

Algorithm 37: Extended Unified twisted Edwards Point Operation

input : $P = (X_1 : Y_1 : Z_1), Q = (X_2 : Y_2 : Z_2)$
output: $P + Q = (X_3 : Y_3 : Z_3)$

$A = (X_1.X_2), B = (Y_1.Y_2), C = dT_1.T_2$
$D = Z_1.Z_2, E = (X_1 + Y_1).(X_2 + Y_2) - A - B,$
$F = D - C, G = D + C, H = B - aA$
$X_3 = E.F, Y_3 = G.H, T_3 = E.H, Z_3 = F.G$

A.3 Co-Z Algorithms

In this section we present some of the Co-Z operations defined in Chapter 3, presented as Alg 38–Alg 44.

Algorithm 38: Co-Z Addition with Update (ZADDU)

Require : $P = (X_1 : Y_1 : Z)$ and $Q = (X_2 : Y_2 : Z)$
Ensure : $(R, P) \leftarrow \text{ZADDU}(P, Q)$ 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 $\lambda \neq 0$

Function: ZADDU (P, Q)

$C \leftarrow (X_1 - X_2)^2$, $W_1 \leftarrow X_1 C$; $W_2 \leftarrow X_2 C$
$D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$, $X_3 \leftarrow D - W_1 - W_2$
$Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$, $Z_3 \leftarrow Z(X_1 - X_2)$
$X_1 \leftarrow W_1$; $Y_1 \leftarrow A_1$; $Z_1 \leftarrow Z_3$
$R = (X_3 : Y_3 : Z_3)$, $P = (X_1 : Y_1 : Z_1)$

Algorithm 39: Conjugate Co-Z Addition (ZADDC)

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

Algorithm 40: Out-of-Place Differential Addition-and-Doubling 1 (AddDblCoZ1)

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

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

Algorithm 41: Out-of-Place Differential Addition-and-Doubling 2 (AddDblCoZ2)

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

Function: AddDblCoZ2(X_1, X_2, Z)
$U = (X_1 - X_2)^2$,$V = 4X_2(X_2^2 + aZ^2) + 4bZ^3$
$X_1' = V[(X_1 + X_2)(X_1^2 + X_2^2 - U + 2aZ^2) + 4bZ^3 - x_DZU]$
$X_2' = U\left[(X_2^2 - aZ^2)^2 - 8bZ^3X_2\right]$, $Z' = UVZ$

Algorithm 42: Out-of-Place Differential Addition-and-Doubling 3 (AddDblCoZ3)

Require : $X_1, X_2, T_D = x_DZ, T_a = aZ^2, T_b = 4bZ^3$
Ensure : $X_1', X_2', T_D', T_a', T_b'$

Function: AddDblCoZ3(X_1, X_2, Z)
$U = (X_1 - X_2)^2$, $V = 4X_2(X_2^2 + T_a) + T_b$
$W = UV$, $T_D' = T_DW$, $T_a' = T_aW^2$
$T_b' = T_bW^3$, $X_1' = V\left[(X_1 + X_2)(X_1^2 + X_2^2 - U + 2T_a) + T_b\right] - T_D'$
$X_2' = U\left[(X_2^2 - T_a)^2 - 2X_2T_b\right]$

Algorithm 43: Co-Z Doubling-Addition with Update (ZDAU)

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

Function: ZDAU(P, Q)
$C' \leftarrow (X_1 - X_2)^2$; $W_1' \leftarrow X_1C'$; $W_2' \leftarrow X_2C'$
$D' \leftarrow (Y_1 - Y_2)^2$; $A_1' \leftarrow Y_1(W_1' - W_2')$, $\hat{X}_3' \leftarrow D' - W_1' - W_2'$
$C \leftarrow (\hat{X}_3' - W_1')^2$, $Y_3' \leftarrow [(Y_1 - Y_2) + (W_1' - \hat{X}_3')]^2 - D' - C - 2A_1'$
$W_1 \leftarrow 4\hat{X}_3'C$; $W_2 \leftarrow 4W_1'C$, $D \leftarrow (Y_3' - 2A_1')^2$; $A_1 \leftarrow Y_3'(W_1 - W_2)$
$X_3 \leftarrow D - W_1 - W_2$, $Y_3 \leftarrow (Y_3' - 2A_1')(W_1 - X_3) - A_1$
$Z_3 \leftarrow Z\big((X_1 - X_2 + \hat{X}_3' - W_1')^2 - C' - C\big)$ $\overline{D} \leftarrow (Y_3' + 2A_1')^2$
$X_2 \leftarrow \overline{D} - W_1 - W_2$, $Y_2 \leftarrow (Y_3' + 2A_1')(W_1 - X_2) - A_1$, $Z_2 \leftarrow Z_3$
$R = (X_3 : Y_3; Z_3)$, $Q = (X_2 : Y_2 : Z_2)$

Algorithm 44: (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 $C = (X_1 - X_2)^2$

Ensure : $(R', S', C) \leftarrow \mathrm{ZACAU}'(P', Q', C)$ 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: $\mathrm{ZACAU}'(P', Q', C)$

$W_1 \leftarrow X_1C$; $W_2 \leftarrow X_2C$, $D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$
$X_1' \leftarrow D - W_1 - W_2$; $Y_1' \leftarrow (Y_1 - Y_2)(W_1 - X_1') - A_1$, $\overline{D} \leftarrow (Y_1 + Y_2)^2$
$X_2' \leftarrow \overline{D} - W_1 - W_2$, $Y_2' \leftarrow (Y_1 + Y_2)(W_1 - X_2') - A_1$, $C' \leftarrow (X_1' - X_2')^2$
$X_4 \leftarrow X_1'C'$; $W_2' \leftarrow X_2'C'$, $D' \leftarrow (Y_1' - Y_2')^2$; $Y_4 \leftarrow Y_1'(X_4 - W_2')$
$X_3 \leftarrow D' - X_4 - W_2'$, $Y_3 \leftarrow (Y_1' - Y_2')(X_4 - X_3) - Y_4$, $C \leftarrow (X_3 - X_4)^2$;
$Y_3 \leftarrow (Y_1' - Y_2' + X_4 - X_3)^2 - D' - C - 2Y_4$, $X_3 \leftarrow 4X_3$; $Y_3 \leftarrow 4Y_3$; $X_4 \leftarrow 4X_4$
$Y_4 \leftarrow 8Y_4$; $C \leftarrow 16C$
$R' = (X_3 : Y_3)$, $S' = (X_4 : Y_4)$, C

A.4 Point Doubling Formulæ with Update in Homogeneous Coordinates.

A double of point $P = (X_1 : Y_1 : Z_1)$ on $E_{\mathcal{H}}$, denoted $\mathrm{DBL}_{\mathcal{H}}$, is computed as $2P = (X_3 : Y_3 : Z_3)$ with

$$X_3 = 2BD, Y_3 = A(4C - D) - 8(Y_1B)^2, Z_3 = 8B^3$$

where $A = aZ_1^2 + 3X_1^2$, $B = Y_1Z_1$, $C = X_1(Y_1B)$, and $D = A^2 - 8C$. The cost of it is $6\mathsf{M} + 5\mathsf{S} + 1\mathsf{c}$. $\mathrm{DBL}_{\mathcal{H}}$ was optimised by trading one multiplication with one squaring which results in a cost of $5\mathsf{M} + 6\mathsf{S} + 1\mathsf{c}$ and is given as

$$X_3 = 4BD, Y_3 = A(4C - D) - 64(Y_1B)^2, Z_3 = 64B^3$$

where $A = 2(aZ_1^2 + 3X_1^2)$, $B = Y_1Z_1$, $C = 2[(X_1 + Y_1B)^2 - X_1^2 - (Y_1B)^2]$, and $D = A^2 - 8C$. If $Z_1 = 1$, the cost drops to $3\mathsf{M} + 5\mathsf{S}$, with

$$X_3 = 4Y_1D, Y_3 = A(4C - D) - 64B, Z_3 = 64Y_1\alpha$$

where $A = 2(a+3X_1^2)$, $\alpha = Y_1^2$, $B = \alpha^2$, $C = 2[(X_1+\alpha)^2 - X_1^2 - B]$, and $D = A^2 - 8C$. Notice that together with $2P$ a representation of a point P is obtained having the same Z coordinate at a cost of only one multiplication.

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

Let $(\tilde{P}, 2P) \leftarrow \mathrm{DBLU}_{\mathcal{H}}(P)$ denote the corresponding operation, where $\tilde{P} \sim P$ and $\mathrm{Z}(\tilde{P}) = \mathrm{Z}(2P)$. The cost of $\mathrm{DBLU}_{\mathcal{H}}$ operation (doubling with update) is $4\mathsf{M} + 5\mathsf{S}$.

Furthermore, for implementation purpose of Algorithm 19, an (X, Z)-only point doubling is defined with an update in homogeneous coordinate, denoted as $\mathrm{DBLU}_{\mathcal{H}}{}^*$ as $\mathrm{DBLU}_{\mathcal{H}}{}^*(P) \leftarrow$

$(X(\tilde{P}) : X(2P) : Z(2P)) = (X_1 \cdot 64Y_1\alpha : 4Y_1D : 64Y_1\alpha)$. The cost of $\mathrm{DBLU}_{\mathcal{H}^*}$ operation is $3\mathsf{M} + 5\mathsf{S}$.

A.5 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 $R_1, Z = (X_2, Z)$ at the end of the Montgomery ladder is described in Alg. 45.

Algorithm 45: Out-of-Place $(X : Y : Z)$-Recovery 1

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

Function: $recoverfullcoordinates1(X_1, X_2, Z)$
$A = Z^2$, $B = ZA$, $C = x_D Z$, $D = 4y_D X_1$,$X_1 = DX_1A$
$X_2 = 2\left[(CX_1 + aA)(C + X_1) - X_2(C - X_1)^2\right] + 4bB$, $Z = DB$
$R = (X : Y : Z) = (X_1 : X_2 : Z)$

Note that $D = (x_D, y_D)$ represents the invariant, input point P, of the Montgomery ladder in affine coordinates. The cost of this formula is $8\mathsf{M} + 2\mathsf{S} + 1M_a + 1M_{4b} + 8\mathsf{add}$ detailed in algorithm 7 of [64].

The full coordinates recovery formula given by Algorithm 19 is evaluated in Alg 46.

Algorithm 46: Out-of-Place $(X : Y : Z)$-Recovery 2

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

Function: $recoverfullcoordinates2(X_1, X_2, Z)$
$X_1 = 4y_D x_D T_D^2 X_1$
$X_2 = X_D^3[T_b + 2(T_D X_1 + T_a)(X_1 + T_D) - 2X_2(X_1 - T_D)^2]$
$Z = 4y_D T_D^3$
$R = (X : Y : Z) = (X_1 : X_2 : Z)$

The cost of which is $10\mathsf{M} + 3\mathsf{S} + 8add$ as detailed in algorithm 8 of [64].

A.6 Point Doubling and Tripling with Co-Z Update

Algorithms 18, 20, 21, 22 and 23 require a point doubling or a point tripling operation for their initialisation. We describe here how this can be implemented.

Initial Point Doubling: The double of a point is computed using the DBLU operation below.

$$\begin{cases} \mathrm{X}(2P) = M^2 - 2S, \\ \mathrm{Y}(2P) = M(S - \mathrm{X}(2P)) - 8L, \\ \mathrm{Z}(2P) = 2Y_1 \end{cases} \tag{A.1}$$

with $M = 3B + a$, $S = 2((X_1 + E)^2 - B - L)$, $L = E^2$, $B = {X_1}^2$, and $E = {Y_1}^2$. Since $Z(2P) = 2Y_1$, it follows that

$$(S : 8L : Z(2P)) \sim P \quad \text{with } S = 4X_1{Y_1}^2 \text{ and } L = {Y_1}^4$$

is an equivalent representation for point P. Updating point P such that its Z-coordinate is equal to that of $2P$ comes thus for free [87]. Let $(2P, \tilde{P}) \leftarrow \mathrm{DBLU}(P)$ denote the corresponding operation, where $\tilde{P} \sim P$ and $\mathrm{Z}(\tilde{P}) = \mathrm{Z}(2P)$. The cost of DBLU operation (doubling with update) is $1\mathsf{M} + 5\mathsf{S}$.

Initial Point Tripling: The triple of $P = (X_1 : Y_1 : 1)$ can be evaluated as $3P = P + 2P$ using co-Z arithmetic [92]. From $(2P, \tilde{P}) \leftarrow \mathrm{DBLU}(P)$, this can be obtained as $\mathrm{ZADDU}(\tilde{P}, 2P)$ with $5\mathsf{M} + 2\mathsf{S}$ and no additional cost to update P for its Z-coordinate becoming equal to that of $3P$. The corresponding operation, tripling with update, is denoted $\mathrm{TPLU}(P)$ and its total cost is of $6\mathsf{M} + 7\mathsf{S}$.

A.7 Full Power, Energy and Timing Results

Presented here are the full set of results for the dedicated and SPA secure algorithms presented in Section 3.4.1 and Section 3.10, using Equation 3.19.

Table A.1 presents the full results for the dedicated double and add algorithms. Columns 3 and 4 give the supply data from the power meter, while subsequent columns give the results either obtained from the oscilliscope or calculated using equation 3.19. We draw attention to the final two columns, the former being the time taken to perform a full calculation of the algorithm and the latter being the energy expended to perform that calculation. In both cases, the lower the value, the better the result.

Table A.1: FPGA Power and Timing Results for Double-and-Add

		Power Meter		Oscilliscope					
Algorithm	No of. Mult.	Suppl. Current mA	Suppl. Power mW	μ Resistor Voltage Drop mV	Resistor Power mW	μ FPGA Voltage mV	μ FPGA Power mW	Calc. Time mS	Energy mJ
Affine D&A	1	151.9	151.9	139.8	19.5	860.2	120.2	13.9	1.7
Projective	1	150.0	150.0	138.8	19.3	861.2	119.5	32.6	3.9
Double	2	157.2	157.2	145.0	21.0	855.0	124.0	17.8	2.2
and	3	164.8	164.8	153.6	23.6	846.4	130.0	13.9	1.8
Add	4	171.3	171.3	157.1	24.7	842.9	132.4	10.8	1.4
Jacobian	1	148.9	148.9	137.8	19.0	862.2	118.8	25.4	3.0
Double	2	156.3	156.3	144.4	20.8	855.6	123.5	16.9	2.1
and	3	164.5	164.8	153.8	23.7	846.2	130.2	13.8	1.8
Add	4	169.6	169.6	156.7	24.5	843.3	132.1	13.0	1.7
Twisted	1	150.7	150.7	138.3	19.1	861.7	119.2	23.6	2.8
Edwards	2	157.1	157.1	144.4	20.8	855.5	123.4	12.7	1.6
Dedicated	3	165.6	165.6	153.8	23.8	845.9	130.4	9.6	1.2
	4	168.9	168.9	156.7	24.1	843.8	131.8	9.6	1.3
Extended	1	148.2	148.2	138.3	19.0	862.2	118.8	22.8	2.7
Twisted	2	156.2	156.2	144.3	20.8	855.8	123.4	12.8	1.7
Edwards	3	164.6	164.6	154.1	23.5	846.8	129.7	10.4	1.3
Dedicated	4	167.5	167.5	156.2	24.1	844.9	131.1	8.0	1.1

It can be seen from the table that the affine coordinates give quite good timing and energy results when compared to the Projective and Jacobian coordinates for 1M for a key with an average hamming weight. However, as it does not scale with parallelisation, it is overtaken in both the calculation time and energy usage by the other coordinate systems by 4M. The Jacobian coordinates

start off quicker with more energy efficiency than the Projective coordinates, but the latter makes better use of parallelisation to overtake the former in timing, albeit at a greater cost in energy. Table A.2 gives the timing, power and energy results. The Double-and-Always-Add, while being the slowest at $1M$ performs quite well for parallelisation and the speedup performance is quite acceptable at $4\mathsf{M}$. There is very little difference in the Joye and Hutter algorithms until $4M$ where the Hutter algorithms perform better. The standard unified twisted Edwards does not have a very good speed or energy performance in comparison, however, the extended unified algorithm performs best of the algorithms under examination at $4\mathsf{M}$.

Table A.2: SPA Secure Power and Timing Results

		Power Meter		Oscilliscope					
Algorithm	No of. Mult.	Suppl. Current mA	Suppl. Power mW	μ Resistor Voltage Drop mV	Resistor Power mW	μ FPGA Voltage mV	μ FPGA Power mW	Calc. Time mS	Energy mJ
Double	1	150.6	150.6	139.2	19.4	860.8	119.8	42.3	5.1
and	2	157.3	157.3	144.8	21.0	855.2	123.8	22.0	2.7
Always	3	163.9	163.9	152.2	23.2	847.8	129.0	15.8	2.0
Add	4	169.1	169.1	156.7	24.6	843.3	132.1	14.2	1.9
Montgomery	1	148.9	148.9	138.4	19.2	861.6	119.2	26.6	3.2
Ladder	2	158.1	158.1	146.2	21.4	853.8	124.8	15.6	2.0
with Co-Z	3	163.5	163.5	152.2	23.2	847.8	129.1	10.9	1.4
Addition	4	168.6	168.6	155.7	24.2	844.3	131.5	10.9	1.4
Joye's	1	150.1	150.1	138.9	19.3	861.1	119.6	26.7	3.2
Double-Add	2	157.3	157.3	144.8	21.0	855.2	123.9	15.7	1.9
with Co-Z	3	164.2	164.2	153.2	23.5	846.8	129.7	11.0	1.4
Addition I	4	169.2	169.2	156.4	24.5	843.6	131.9	11.0	1.4
Joye's	1	151.3	151.3	139.2	19.4	860.8	119.8	26.8	3.2
Double-Add	2	157.7	157.7	144.6	20.9	855.4	123.7	15.8	2.0
with Co-Z	3	163.9	163.9	152.7	23.3	847.3	129.4	12.7	1.6
Addition II	4	169.3	169.3	156.3	24.4	843.7	131.9	11.1	1.5
Montgomery	1	147.4	147.4	137.8	19.0	862.2	118.8	28.1	3.3
Ladder	2	155.1	155.1	144.5	20.9	855.5	123.6	15.5	1.9
(X,Z)	3	162.3	162.3	150.2	22.6	849.8	127.7	10.8	1.4
(Alg.40&45)	4	169.1	169.1	156.5	24.5	843.5	132.0	9.1	1.2
Montgomery	1	147.3	147.3	138.0	19.0	862.0	119.0	26.5	3.2
Ladder	2	155.3	155.3	144.6	20.9	855.4	123.7	13.9	1.7
(X,Z)	3	162.3	162.3	151.7	23.0	848.3	128.7	10.8	1.4
(Alg.41&45)	4	169.0	169.0	156.4	24.5	843.6	132.0	9.1	1.2
Montgomery	1	147.5	147.5	138.1	19.0	861.9	119.0	24.9	3.0
Ladder	2	155.6	155.6	145.2	20.9	854.8	124.1	13.9	1.7
(X,Z)	3	163.1	163.1	152.3	23.0	847.7	129.1	9.1	1.2
(Alg.42&46)	4	169.2	169.2	156.5	24.5	843.5	132.0	9.1	1.2
Montgomery	1	148.7	148.7	138.6	19.2	861.4	119.4	24.1	2.9
Ladder	2	156.0	156.0	145.6	21.2	854.4	124.4	13.0	1.6
(X,Y)	3	167.4	167.4	156.1	24.4	843.9	131.7	9.9	1.3
	4	169.8	169.8	156.8	24.6	843.2	132.2	8.3	1.1
Signed	1	147.3	147.3	138.1	19.1	861.9	119.0	23.6	2.8
Digit	2	155.8	155.8	145.3	21.1	854.7	124.2	14.2	1.8
	3	164.6	164.6	153.4	23.5	846.6	129.9	11.0	1.4
	4	170.2	170.2	156.8	24.6	843.2	132.2	9.5	1.3
Twisted	1	148.8	148.8	139.0	19.3	861.0	119.7	33.8	4.1
Edwards	2	154.7	154.7	144.0	20.7	856.0	123.3	22.2	2.7
Unified	3	165.1	165.1	154.5	23.9	845.5	130.6	17.5	2.3
	4	168.5	168.5	156.5	24.5	843.5	156.6	15.2	2.0
Extended	1	148.7	148.7	138.7	19.2	861.3	119.5	26.8	3.2
Twisted	2	154.5	154.5	144.5	20.9	855.5	123.6	15.1	1.9
Edwards	3	165.3	165.3	154.8	24.0	845.2	130.8	10.5	1.4
Unified	4	169.1	169.1	156.4	24.5	843.6	131.9	8.1	1.1

B

Appendix - Hash Functions

B.1 Round Two Hash Function Implementation Results

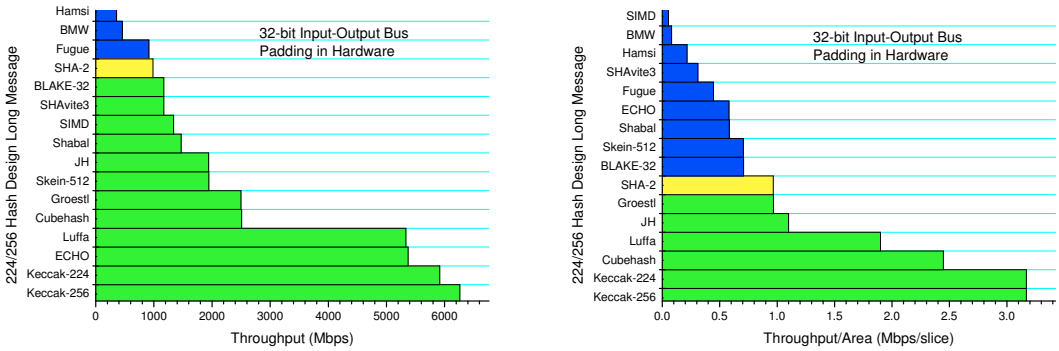
The area results for the implemented hash functions on the Xilinx Virtex-5 are presented in Table B.1. This table should be used as an addendum to Table 4.8 The *-w* designation defines the results inclusive of the wrapper, while $-nw$ gives the hash function as a stand alone entity.

Table B.1: Full Hash Round Two Area & Frequency Results

Hash Design	Area-w (slices)	Max.Freq-w (MHz)	Area-nw (slices)	Max.Freq-nw (MHz)
SHA-2-256	1019	125.06	656	125.125
SHA-2-512	1771	100.04	1213	110.09
BLAKE-32	1653	91.34	1118	118.06
BLAKE-64	2888	71.04	1718	90.90
BMW-256*	5584	14.30	4997	14.01
BMW-512*	9902	8.98	9810	10
Cubehash	1025	166.66	695	166.83
ECHO-256*	8798	161.21	7372	198.92
ECHO-512*	9130	166.66	8633	166.69
Fugue-256	2046	200	1689	200.04
Fugue-384	2622	200.08	2380	200.08
Fugue-512	3137	195.81	2596	200.16
Grøstl-256*	2579	78.06	2391	101.31
Grøstl-512*	4525	113.12	4845	123.39
Hamsi-256	1664	67.19	1518	72.41
Hamsi-512	7364	14.93	6229	16.51
JH	1763	144.11	1291	250.12
Keccak-224*	1971	195.73	1117	189
Keccak-256*	1971	195.73	1117	189
Keccak-384*	1971	195.73	1117	189
Keccak-512	1971	195.73	1117	189
Luffa-256*	2796	166.66	2221	166.66
Luffa-384*	4233	166.75	3740	166.75
Luffa-512*	4593	166.66	3700	166.75
Shabal	2512	143.47	1583	148.03
SHAvite3-256	3776	82.27	3125	109.17
SHAvite3-512	11443	63.66	9775	59.4
SIMD-256	24536	107.2	22704	107.2
SIMD-512	44673	107.2	43729	107.2
Skein-512	3027	83.57	1938	83.65

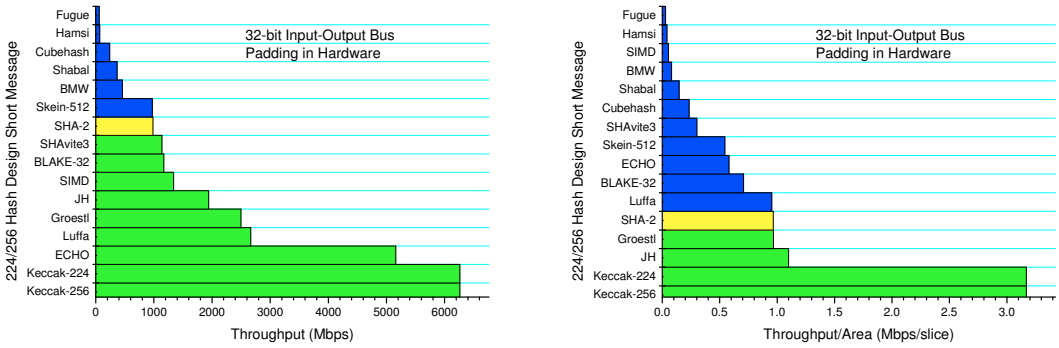
B.2 Round Two Hash Function Results

Here follows the additional Round two hash function results in graphical form as presented in Section 4.6. the results for a 32-bit bus with padding in hardware are presented first, followed by the same 32-bit bus with the padding in software. Finally an ideal bus, i.e. equal to the input size required is presented. The performance of the hash functions in each of the scenarios for throughput and throughput-area, compared to SHA-2 is used as a metric to determine best design.



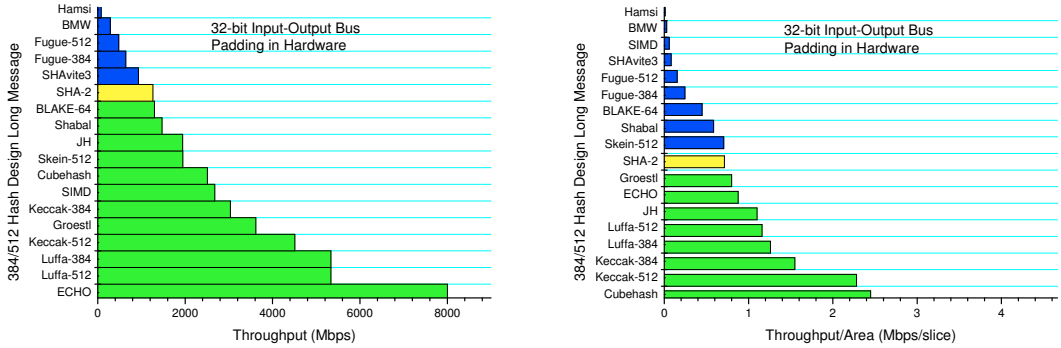
(a) Throughput **(b)** Throughput-Area

Figure B.1: 256-bit Long 32-bit Bus Padding Hardware



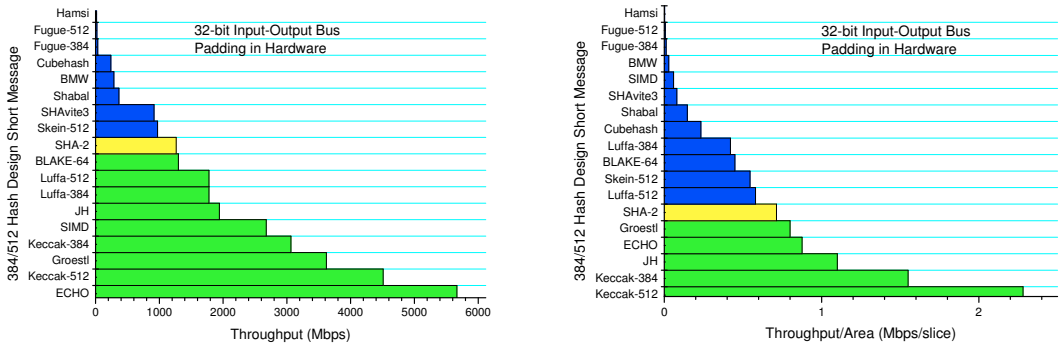
(a) Throughput **(b)** Throughput-Area

Figure B.2: 256-bit Short 32-bit Bus Padding Hardware



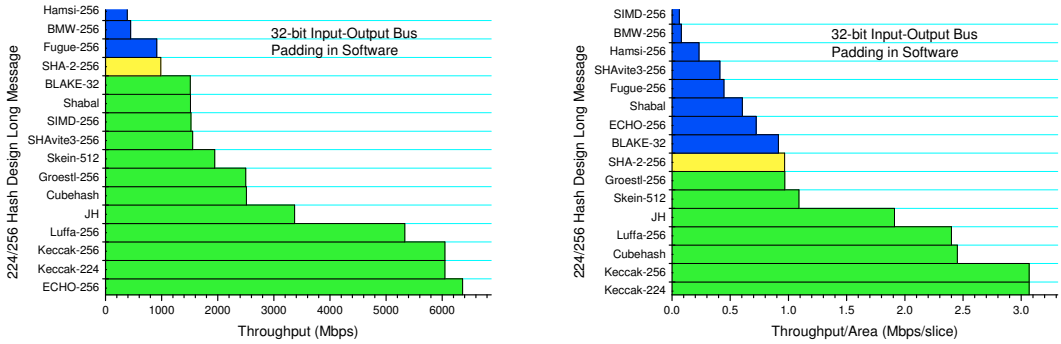
(a) Throughput **(b)** Throughput-Area

Figure B.3: 512-bit Long 32-bit Bus Padding Hardware



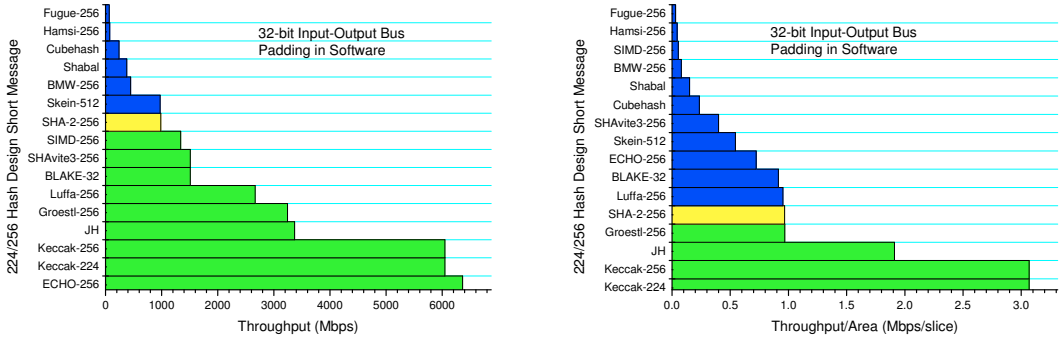
(a) Throughput **(b)** Throughput-Area

Figure B.4: 512-bit Short 32-bit Bus Padding Hardware



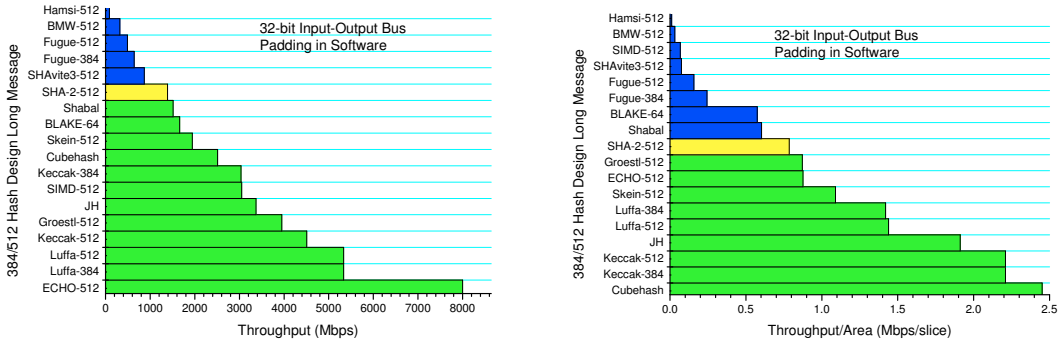
(a) Throughput **(b)** Throughput-Area

Figure B.5: 256-bit Long 32-bit Bus Padding Software



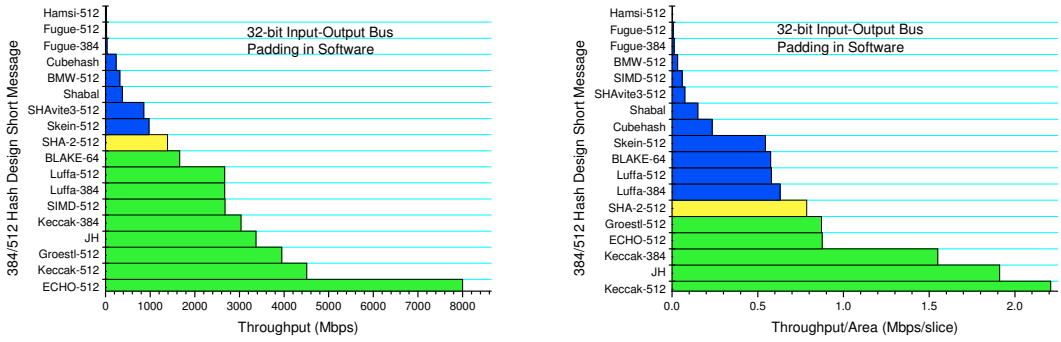
(a) Throughput **(b)** Throughput-Area

Figure B.6: 256-bit Short 32-bit Bus Padding Software



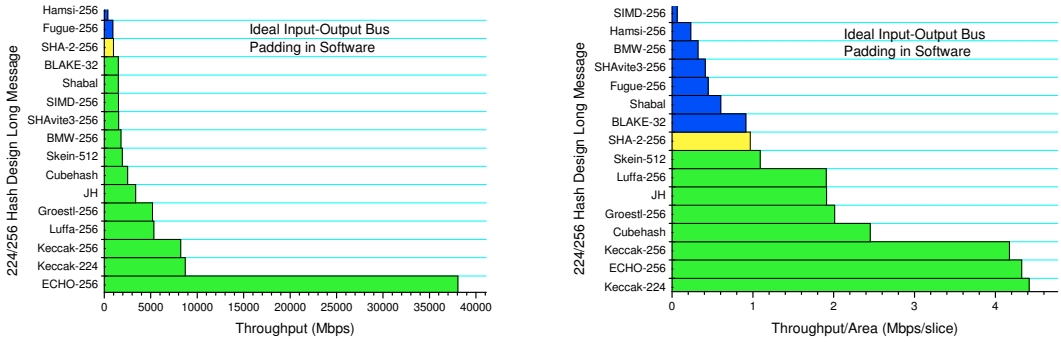
(a) Throughput **(b)** Throughput-Area

Figure B.7: 512-bit Long 32-bit Bus Padding Software



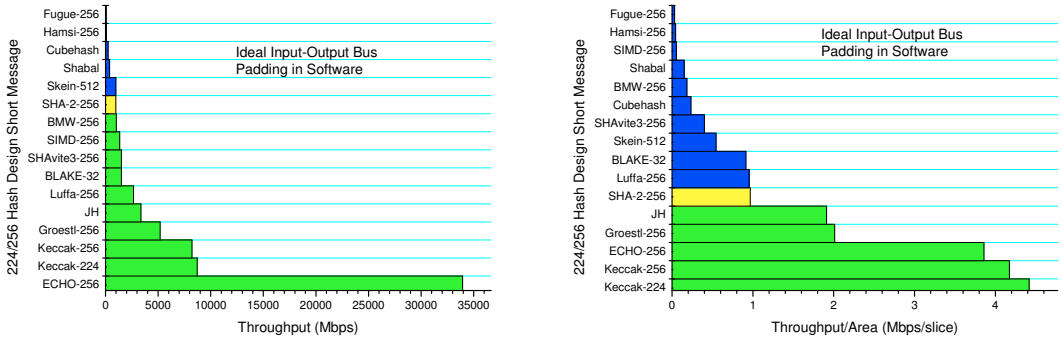
(a) Throughput **(b)** Throughput-Area

Figure B.8: 512-bit Short 32-bit Bus Padding Software



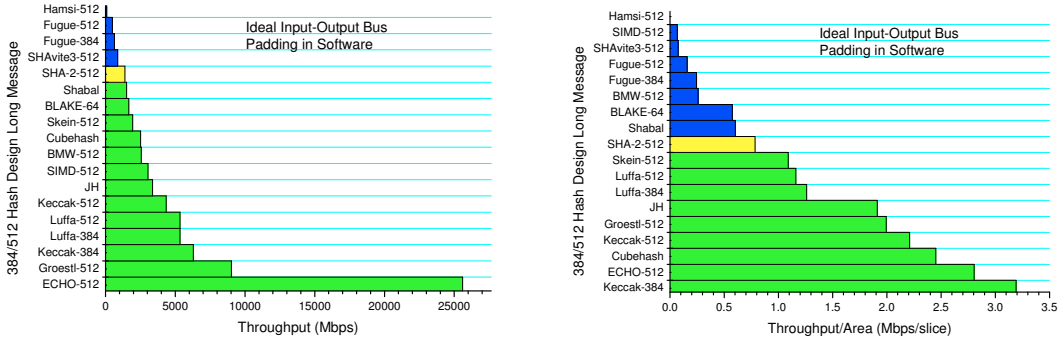
(a) Throughput **(b)** Throughput-Area

Figure B.9: 256-bit Long Ideal-Bus Padding Software



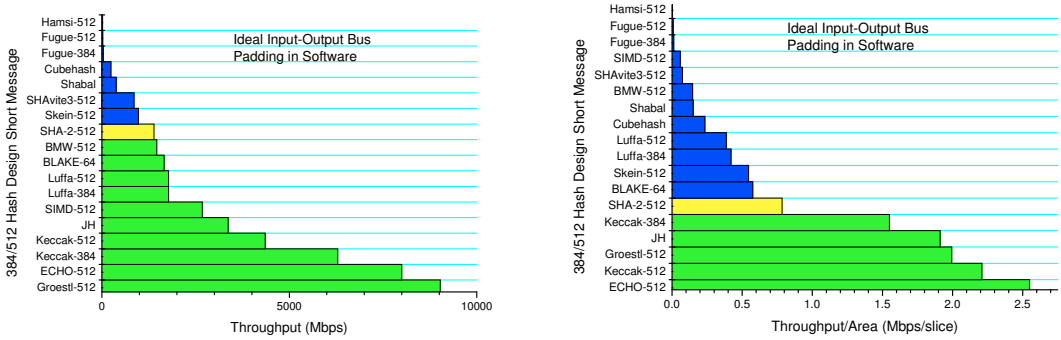
(a) Throughput **(b)** Throughput-Area

Figure B.10: 256-bit Short Ideal-Bus Padding Software



(a) Throughput **(b)** Throughput-Area

Figure B.11: 512-bit Long Ideal Bus Padding Software



(a) Throughput **(b)** Throughput-Area

Figure B.12: 512-bit Short Ideal Bus Padding Software

B.3 Round Three FPGA Power and Timing Results

Here the full FPGA Power and Timing Results for the round three SHA-3 designs are presented in Table B.2. This table is meant as an addendum to Table 4.10

Table B.2: Full FPGA Power and Timing Results for SHA-3 at 24MHz

	Power Meter		Oscilliscope					
Algorithm	Supplied Current	Supplied Power	Mean Resistor Voltage Drop	Resistor Power	Mean FPGA Voltage	Mean FPGA Power	Iteration Time	Energy
224/256	mA	mW	mV	mW	mV	mW	μS	μJ
Blake S	169.3	169.4	166.1	27.6	833.9	138.5	4.34	0.601
Blake L	151.4	151.5	141.4	20.0	858.6	121.4	92.25	11.20
Grøstl S	263.6	263.8	284.4	80.9	715.6	203.5	**2.80**	0.570
Grøstl L	319.9	321.0	302.0	91.2	698.0	210.8	**33.78**	7.11
JH S	157.7	157.7	150.9	22.8	849.1	128.2	6.78	0.870
JH L	157.4	157.3	146.7	21.5	853.3	125.2	77.0	9.64
Keccak S	157.9	158.0	151.9	23.1	848.1	128.8	4.32	**0.557**
Keccak L	132.7	132.8	122.7	15.1	877.3	107.7	42.92	**4.62**
Skein S	226.1	226.2	239.3	57.3	760.7	182.0	3.93	0.715
Skein L	164.0	164.1	154.2	23.8	845.8	130.4	59.02	7.57
384/512								
Blake S	204.5	204.6	203.1	41.3	796.9	161.9	65	1.057
Blake L	224.6	224.8	214.3	45.9	785.7	168.4	70.56	11.88
Grøstl S	440.3	440.0	525.3	27.59	474.7	229.3	4.50	1.008
Grøstl L	440.5	440.5	530.5	28.14	469.5	249.1	**35.05**	8.725
JH S	158.6	158.7	151.6	23.0	848.4	128.6	7.10	0.913
JH L	159.5	159.6	150.2	22.6	849.8	127.7	77.50	9.89
Keccak S	159.7	159.8	156.0	24.3	844.0	131.7	**3.29**	**0.433**
Keccak L	168.5	168.6	160.3	25.7	839.7	134.6	54.76	**7.37**
Skein S	239.6	239.7	253.8	64.4	746.2	189.4	4.20	0.806
Skein L	207.1	207.2	198.4	39.3	801.6	159.0	59.44	9.45

Bibliography

[1] A. Poschmann, "Lightweight cryptography cryptographic engineering for a pervasive world," Ph.D. dissertation, Faculty of Electrical Engineering and Information Technology Ruhr-University Bochum, Germany, 2009.

[2] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[3] D. J. Bernstein and T. Lange, "Faster addition and doubling on elliptic curves," in *Advances in Cryptology - ASIACRYPT '07*, vol. 4833 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2007, pp. 29–50.

[4] P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[5] M. Joye, "Highly Regular Right-to-Left Algorithms for Scalar Multiplication," in *Cryptographic Hardware and Embedded Systems - CHES '07*, vol. 4727 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2007, pp. 135–147.

[6] F. Chabaud and A. Joux, "Differential Collisions in SHA-0," in *Advances in Cryptology - CRYPTO '98*, vol. 1462 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1998, pp. 56–71.

[7] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *Advances in Cryptology - CRYPTO '05*, vol. 3621 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2005, pp. 17–36.

[8] C. D. Canniere and C. Rechberger, "Finding SHA-1 Characteristics: General Results and Applications," in *Advances in Cryptology ASIACRYPT '06*, vol. 4284 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2006, pp. 1–20.

[9] D. Kahn, *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*, 2nd ed. Simon & Schuster Inc., 1997.

[10] A. Tacticus, *How to Survive Under Siege*, 2nd ed., D. Whitehead, Ed. Bristol Classical Press, 2002.

[11] G. S. Tranquillus, *De vita Caesarum - The Lives of the Twelve Caesars*, J. C. Rolfe, Ed. Bill Thayer, 1913.

[12] L. B. Alberti, *A Treatise on Ciphers*, A. Zaccagnini, Ed. Galimberti, Torino, 1997.

[13] J. A. Reeds, "Solved: The Ciphers in Book III of Trithemius's Steganographia," *Cryptologia*, vol. 22, pp. 291–319, 1998.

[14] S. Singh, *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*, 1st ed. New York, NY, USA: Doubleday, 1999.

[15] E. Wenger and M. Hutter, "Exploring the Design Space of Prime Field vs. Binary Field ECC-Hardware Implementations," in *Information Security Technology for Applications - ISTA '12*, vol. 7161 of Lecture Notes in Computer Science (LNCS). Springer-V, 2012, pp. 256–271.

[16] F. Rodríguez-Henríquez, N. A. Saqib, A. D. Pérez, and C. K. Koc, *Cryptographic Algorithms on Reconfigurable Hardware*, ser. Signals and Communication Technology. Springer, 2007.

[17] V. S. Miller, "Uses of Elliptic Curves in Cryptography," in *Advances in Cryptology - CRYPTO '85*, vol. 218 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1985, pp. 417–426.

[18] N. Koblitz, "Elliptic Curve Cryptosystems," in *Mathematics of Computation*, vol. 48, no. 177, 1987, pp. 203–209.

[19] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[20] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, and K. N. F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, ser. Discrete Mathematics and Its Applications, K. H. Rosen, Ed. Chapman & Hall/CRC, 2006.

[21] B. Schneier, *Applied Cryptography*, 2nd ed. John Wiley & Sons, Inc., 1996.

[22] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[24] W. F. Ehrsam, C. H. W. Meyer, J. L. Smith, and W. L. Tuchman, "Message verification and transmission error detection by block chaining," U.S. Patent 4 074 066, April, 1978.

[25] NIST, *Data Encryption Standard (DES) (FIPS–46-3)*, National Institute of Standards and Technology, 1999.

[26] ——, *Advanced Encryption Standard (AES) (FIPS–197)*, National Institute of Standards and Technology, 2001.

[27] ——, *The Keyed-Hash Message Authentication Code (HMAC)(FIPS PUB–198-1)*, National Institute of Standards and Technology, 2008.

[28] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[29] J. H. Ellis, "The History of Non-Secret Encryption," *Cryptologia*, vol. 23, no. 3, pp. 267–273, 1999.

[30] R. Rivest, A. Shamir, and L. M. Adleman, "Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[31] J. Pollard, "Monte Carlo Methods for Index Computation mod p," *Mathematics of Computation*, vol. 32, pp. 918–924, 1978.

[32] S. Pohlig and M. E. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, pp. 106–110, 1978.

[33] NIST, *Digital Signature Standard (DSS) (FIPS–186)*, National Institute of Standards and Technology, 1994.

[34] D. Johnson, A. Menezes, and S. A. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[35] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *Siam Journal on Computing*, vol. 17, no. 2, pp. 281–308, April 1988.

[36] M. O. Rabin, "Digitalized signatures and public-key functions as intractable as factorization," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1979.

[37] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[38] ANSI, *Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA) (ANSI X9.62:2005)*, American National Standards Institute, 2005.

[39] IEEE, *Standard Specifications For Public-Key Cryptography (IEEE 1363-2000)*, Institute of Electrical and Electronics Engineers, 1999.

[40] ISO/IEC, *Information Technology – Security Techniques – Cryptographic Techniques based on Elliptic Curves (15946) – Part 5: Elliptic Curve Generation*, 5th ed., International Organization for Standardization and International Electrotechnical Commission, 2009.

[41] NIST, *Recommendation for Key Management-Part 1(Special Publication 800-57)*, 3rd ed., National Institute of Standards and Technology, 2007.

[42] ECRYPT, *ECRYPT II Yearly Report on Algorithms and Keysizes (ICT-2007-216676)*, 1st ed., European Network of Excellence in Cryptology II, 2010.

[43] Xilinx, "Support documentation," http://www.xilinx.com/support/documentation/.

[44] Altera, "Support documentation," http://www.altera.com/literature/lit-index.html.

[45] B. Baldwin and W. P. Marnane, "An FPGA Technologies Area Examination of the SHA-3 Hash Candidate Implementations," Cryptology ePrint Archive, Report 2009/603, 2009.

[46] Xilinx, *Virtex-5 Family Overview (DS 100 (V5))*, February 2009.

[47] ——, *Spartan-3 FPGA family: Complete data sheet*, 2008.

[48] AIST and RCIS, *Sidechannel Attack Standard Evaluation Board (SASEBO).*, National Institute of Advanced Industrial Science and Technology , Research Center for Information Security.

[49] Xilinx, *ML505/ML506/ML507 Evaluation ML507 Evaluation Platform (UG347 (v3.1.2))*, May 2011.

[50] ——, *ISE Design Suite UG631 (v 12.3)*, 2010.

[51] P. J. Ashenden, *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1995.

[52] S. Kilts, *Advanced FPGA Design*, IEEE, Ed. John Wiley & Sons, Inc., 2006.

[53] Xilinx, *Embedded Processor Block in Virtex-5 FPGAs Reference Guide (UG200 (v1.8))*, 2010.

[54] ——, *MicroBlaze Processor Reference Guide: Embedded Development Kit (EDK 12.3)(UG081 (v11.2))*, 2010.

[55] ——, *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design (UG683 EDK (v.12.2))*, 2010.

[56] ——, *Fast Simplex Link (FSL) Bus (DS449 (v2.11b))*, 2010.

[57] R. Jesman, F. M. Vallina, and J. Saniie, *MicroBlaze Tutorial: Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools*, Embedded Computing and Signal Processing Laboratory - Illinois Institute of Technology, Illinois, USA.

[58] Ç. K. Koç, T. Acar, and B. S. K. Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.

[59] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology - CRYPTO '99*, M. J. Wiener, Ed., vol. 1666 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1999, pp. 388–397.

[60] E. Brier and M. Joye, "Weierstraß Elliptic Curve and Side-Channel Attacks," in *Public Key Cryptography - PKC '02*, D. Naccache and P. Paillier, Eds., vol. 2274 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2002, pp. 335–345.

[61] D. Meidanis, K. Georgopoulos, and I. Papaefstathiou, "FPGA power consumption measurements and estimations under different implementation parameters," in *Field-Programmable Technology - FPT '11*. IEEE, 2011, pp. 1–6.

[62] S. Atay, A. Koltuksuz, H. Hisil, and S. Eren, "Computational Cost Analysis of Elliptic Curve Arithmetic," in *Hybrid Information Technology - ICHIT '06.*, November 2006, pp. 578–582.

[63] R. R. Goundar, M. Joye, and A. Miyaji, "Co-Z addition formulae and Binary Ladders on Elliptic Curves," in *Cryptographic Hardware and Embedded Systems - CHES '10*, vol. 6225 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2010, pp. 65–79.

[64] M. Hutter, M. Joye, and Y. Sierra, "Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-*Z* Coordinate Representation," in *Progress in Cryptology - AFRICACRYPT '11*, vol. 6737 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2011, pp. 170–187.

[65] H. Wu, "Bit-parallel finite field multiplier and squarer using polynomial basis," *IEEE Transactions on Computers*, vol. 51, pp. 750–758, July 2002.

[66] D. J. Bernstein, "Explicit-formulas database," http://hyperelliptic.org/EFD.

[67] M. Rivain, "Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves," Cryptology ePrint Archive, Report 2011/338, 2011.

[68] C. Giraud and V. Verneuil, "Atomicity Improvement for Elliptic Curve Scalar Multiplication," *Computing Research Repository*, vol. abs/1002.4, pp. 80–101, 2010.

[69] H. M. Edwards, "A normal form for elliptic curves," *Bulletin of the American Mathematical Society*, vol. 44, no. 3, pp. 393–422, 2007.

[70] D. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted Edwards curves," in *Progress in Cryptology - AFRICACRYPT '08*, vol. 5023 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2008, pp. 389–405.

[71] NIST, *Digital Signature Standard (DSS) (FIPS–186-3)*, National Institute of Standards and Technology, 2009.

[72] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson, "Twisted Edwards Curves Revisited," in *Advances in Cryptology - ASIACRYPT '08*, vol. 5350 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2008, pp. 326–343.

[73] N. Smart, "The Hessian Form of an Elliptic Curve," in *Cryptographic Hardware and Embedded Systems – CHES '01*, vol. 2162 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2001, pp. 118–125.

[74] M. Joye and J. Quisquater, "Hessian Elliptic Curves and Side-Channel Attacks," in *Cryptographic Hardware and Embedded Systems - CHES '01*, vol. 2162 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2001, pp. 402–410.

[75] A. Byrne, N. Meloni, F. Crowe, W. Marnane, A. Tisserand, and E. Popovici, "SPA resistant Elliptic Curve Cryptosystem using Addition Chains," *International Journal of High Performance Systems Architecture*, vol. 1, no. 2, pp. 133–142, 2007.

[76] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[77] C. D. Walter, "Montgomery Exponentiation Needs no Final Subtractions," in *Electronics Letters*, vol. 35, no. 21, October 1999, pp. 1831–1832.

[78] F. Crowe, A. Daly, and W. Marnane, "Optimised Montgomery Domain Inversion on FPGA," in *European Conference on Circuit Theory and Design - ECCTD '05*, vol. 1, 2005, pp. 277–280.

[79] B. S. Kaliski, "The Montgomery Inverse and Its Applications," *IEEE Transactions on Computers*, vol. 44, no. 8, pp. 1064–1065, 1995.

[80] A. M. Slla and V. Drabek, "An Efficient List-Based Scheduling Algorithm for High-Level Synthesis," in *Euromicro Symposium on Digital Systems Design - DSD '09*. IEEE Computer Society, 2002, pp. 316–323.

[81] Certicom, *Recommended Elliptic Curve Domain Parameters SEC-2*, 1st ed., Standards for Efficient Cryptography Group (SECG), September 2000.

[82] Xilinx, *Virtex-5 Data Sheet: DC and Switching Characteristics*, ds202 (v3.6) ed., November 2007.

[83] R. M. Avanzi, "Side Channel Attacks on Implementations of Curve-Based Cryptographic Primitives," Cryptology ePrint Archive, Report 2005/017, 2005.

[84] J. S. Coron, "Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems," in *Cryptographic Hardware and Embedded Systems - CHES '99*, vol. 1717 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1999, pp. 292–302.

[85] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[86] C. Aumüller, P. Bier, P. Hofreiter, W. Fischer, and J.-P. Seifert, "Fault attacks on RSA with CRT: Concrete Results and Practical Countermeasures," in *Cryptographic Hardware and Embedded Systems - CHES '02*, vol. 2523 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2002, pp. 260–275.

[87] N. Meloni, "New point addition formulæ for ECC applications," in *Workshop on Arithmetic of Finite Fields - WAIFI '07*, C. Carlet and B. Sunar, Eds., vol. 4547 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2007, pp. 189–201.

[88] R. R. Goundar, M. Joye, and A. Miyaji, "Co-Z addition formulæ and binary ladders on elliptic curves," Cryptology ePrint Archive, Report 2010/309, 2010.

[89] R. R. Goundar, M. Joye, A. Miyaji, M. Rivain, and A. Vanelli, "A Scalar multiplication on Weierstraß elliptic curves from Co-Z arithmetic," *Journal of Cryptographic Engineering*, vol. 1, no. 2, pp. 161–176, 2011.

[90] S. Galbraith, X. Lin, and M. Scott, "A faster way to do ECC," Workshop on Elliptic Curve Cryptography - ECC '08, September 2008.

[91] P. Longa and C. H. Gebotys, "Novel precomputation schemes for elliptic curve cryptosystems," in *Applied Cryptography and Network Security -ACNS '09*, M. Abdalla *et al.*, Eds., vol. 5536 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2009, pp. 71–88.

[92] P. Longa and A. Miri, "New composite operations and precomputation for elliptic curve cryptosystems over prime fields," in *Public Key Cryptography - PKC '08*, R. Cramer, Ed., vol. 4939 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2008, pp. 229–247.

[93] B. Baldwin, R. R. Goundar, M. Hamilton, and W. Marnane, "Co-Z ECC Scalar Multiplications for Hardware, Software and Hardware-Software Co-Design on Embedded Systems," *Journal of Cryptographic Engineering*, vol. 2, no. 4, pp. 221–240, 2012.

[94] T. Izu, B. Möller, and T. Takagi, "Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks," in *Progress in Cryptology - INDOCRYPT '02*, A. Menezes and P. Sarkar, Eds., vol. 2551 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2002, pp. 296–313.

[95] H. Cohen, A. Miyaji, and T. Ono, "Efficient Elliptic Curve Exponentiation Using Mixed Coordinates," in *Advances in Cryptology - ASIACRYPT '98*, K. Ohta and D. Pei, Eds., vol. 1514 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1998, pp. 51–65.

[96] NIST, "National Institute of Standards and Technology. [docket no.: 070911510751201] Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA3) Family," *Federal Register*, vol. 72, pp. 62 212–62 220, November 2007.

[97] E. Biham and O. Dunkelman, "A Framework for Iterative Hash Functions - HAIFA," Cryptology ePrint Archive, Report 2007/278, 2007.

[98] R. C. Merkle, "One way hash functions and DES," in *Advances in Cryptology - CRYPTO '89*, G. Brassard, Ed., vol. 435 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1989, pp. 428–446.

[99] I. Damgård, "A Design Principle for Hash Functions," in *Advances in Cryptology - CRYPTO '89*, G. Brassard, Ed., vol. 435 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1989, pp. 416–427.

[100] NIST, *Secure Hash Standard (FIPS–180-4)*, National Institute of Standards and Technology, March 2012.

[101] ——, *Secure Hash Standard (FIPS–180-1)*, National Institute of Standards and Technology, April 1995.

[102] A. K. Lenstra, "Further Progress in Hashing Cryptanalysis (white paper)," http://cm.bell-labs.com/who/akl/hash.pdf, February 2005.

[103] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang, "Preimages for Step-Reduced SHA-2," in *Advances in Cryptology - ASIACRYPT '09*, vol. 5912 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2009, pp. 578–597.

[104] NIST, "National Institute of Standards and Technology. Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition," *NIST Interagency Report*, vol. 7620, September 2009.

[105] ——, "National Institute of Standards and Technology. Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition," *NIST Interagency Report*, vol. 7764, February 2011.

[106] ——, "Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition," *NIST Interagency Report*, vol. 7896, November 2012.

[107] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan, and W. P. Marnane, "FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash," in *Euromicro Symposium on Digital Systems Design - DSD '09*. IEEE Computer Society, 2009, pp. 783–790.

[108] D. J. Bernstein, "CubeHash specification (2.B.1)," Submission to NIST, 2008.

[109] K. Kobayashi, J. Ikegami, S. Matsuo, K. Sakiyama, and K. Ohta, "Evaluation of Hardware Performance for the SHA-3 candidates using SASEBO-GII," Cryptology ePrint Archive, Report 2010/010, 2010.

[110] S. Matsuo, M. Knežević, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyamay, and K. Ohtay, "How Can We Conduct "Fair and Consistent" Hardware Evaluation for SHA-3 Candidate?" in *The Second SHA-3 Candidate Conference*, 2010.

[111] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," Cryptology ePrint Archive, Report 2010/445, 2010.

[112] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau, "Shabal, a submission to NIST's cryptographic hash algorithm competition," Submission to NIST, 2008.

[113] J. Francq and C. Thuillet, "Unfolding Method for Shabal on Virtex-5 FPGAs: Concrete Results," Cryptology ePrint Archive, Report 2010/406, 2010.

[114] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "Fpga implementations of the round two sha-3 candidates," The Second SHA-3 Candidate Conference, August 2010.

[115] ——, "FPGA Implementations of the Round Two SHA-3 Candidates," in *Field Programmable Logic and Applications - FPL '10*. IEEE Computer Society, 2010, pp. 400–407.

[116] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "SHA-3 proposal BLAKE," Submission to NIST, 2008.

[117] J. Li and R. Karri, "Compact hardware architectures for BLAKE and LAKE hash functions," in *IEEE International Symposium on Circuits and Systems - ISCAS '10*. IEEE Computer Society, June 2010, pp. 2107–2110.

[118] D. J. Bernstein, "ChaCha, a variant of Salsa20," http://cr.yp.to/chacha/chacha-20080120.pdf, 2008.

[119] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate," Submission to NIST, 2008.

[120] H. Wu, "The hash function JH," Submission to NIST, 2008.

[121] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Flexible Block Cipher With Maximum Assurance," in *The First Advanced Encryption Standard Candidate Conference*, 1998.

[122] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Keccak specifications," Submission to NIST, September 2009, version 2.

[123] ——, "On the Indifferentiability of the Sponge Construction," in *Advances in Cryptology - EUROCRYPT '08*, vol. 4965 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2008, pp. 181–197.

[124] ——, "Cryptographic Sponge Functions," sponge.noekeon.org/, January 2011.

[125] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein hash function family," Submission to NIST, 2009.

[126] M. Liskov, R. L. Rivest, and D. Wagner, "Tweakable Block Ciphers," in *Advances in Cryptology - CRYPTO '02*, vol. 2442 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2002, pp. 31–46.

[127] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "A Hardware Wrapper for the SHA-3 Hash Algorithms," in *IET Irish Signals and Systems Conference - ISSC '10*. Institution of Engineering and Technology - IET, June 2010, pp. 1–6.

[128] K. Gaj, "Hardware Interface of a Secure Hash Algorithm (SHA)," cryptography.gmu.edu/athena/interfaces/SHA_interface.pdf, October 2009.

[129] J.-P. Aumasson, E. Brier, W. Meier, M. Naya-Plasencia, and T. Peyrin, "Inside the Hypercube," Cryptology ePrint Archive, Report 2008/486, 2008.

[130] N. Ferguson, S. Lucks, and K. A. McKay, "Symmetric States and their Structure: Improved Analysis of CubeHash," Cryptology ePrint Archive, Report 2010/273, 2010.

[131] G. Leurent, "Quantum Preimage and Collision Attacks on CubeHash," Cryptology ePrint Archive, Report 2010/506, 2010.

[132] ECRYPT. eBACS: SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives). http://bench.cr.yp.to/supercop.html.

[133] C. Wenzel-Benner and J. Gräf, "XBX: eXternal benchmarking eXtension for the SUPERCOP crypto benchmarking framework," in *Cryptographic Hardware and Embedded Systems - CHES '10*, vol. 6225 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2010, pp. 294–305.

[134] D. Otte, "AVR Crypto Lib Website," http://das-labor.org/wiki/AVR-Crypto-Lib/en.

[135] IAIK, "SHA-3 Zoo," http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[136] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs," in *Field Programmable Logic and Applications - FPL '10*. IEEE Computer Society, 2010, pp. 414–421.

[137] M. Kneževič, K. Kobayashiy, J. Ikegamiy, S. Matsuoz, A. Satoh, Ü. Kocobaş, J. Fan, T. Katashita, T. Sugawarax, K. Sakiyamay, I. Verbauwhede, K. Ohtay, N. Hommax, and T. Aokix, "Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates," *IEEE Transactions on Very Large Scale Integration Systems - TVLSI*, vol. 20, no. 5, pp. 827–840, May 2011.

[138] X. Guo, S. Huang, L. Nazhandali, and P. Schaumont, "On The Impact of Target Technology in SHA-3 Hardware Benchmark Rankings," Cryptology ePrint Archive, Report 2010/536, 2010.

[139] Z. Chen, S. Morozov, and P. Schaumont, "A hardware interface for hashing algorithms," Cryptology ePrint Archive, Report 2008/529, 2008.

[140] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs," ECRYPT II Hash Workshop 2011, May 2011.

[141] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive evaluation of high-speed and medium-speed implementations of five sha-3 finalists using xilinx and altera fpgas," in *The Third SHA-3 Candidate Conference*, 2012.

[142] K. Latif, M. M. Rao, A. Aziz, and A. Mahboob, "Efficient hardware implementations and hardware performance evaluation of sha-3 finalists," in *The Third SHA-3 Candidate Conference*, 2012.

[143] B. Jungk, "Evaluation of compact fpga implementations for all sha-3 finalists," in *The Third SHA-3 Candidate Conference*, 2012.

[144] J.-P. Kaps, P. Yalla, K. K. Surapathi, B. Habib, S. Vadlamudi, and S. Gurung, "Lightweight implementations of sha-3 finalists on fpgas," in *The Third SHA-3 Candidate Conference*, 2012.

[145] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. De Dormale, and F.-X. Standaert, "Compact FPGA Implementations of the Five SHA-3 Finalists," in *Smart Card Research and Advanced Applications*, vol. 7079 of Lecture Notes in Computer Science (LNCS). Springer, 2011, pp. 217–233.

[146] IETF, *IP Encapsulating Security Payload (ESP)*, rfc 2406 ed., Internet Engineering Task Force, 1998.

[147] ——, *Requirements for Internet Hosts – Communication Layers*, rfc 1122 ed., Internet Engineering Task Force, 1989.

[148] ——, *Requirements for Internet Hosts – Application and Support*, rfc 1123 ed., Internet Engineering Task Force, 1989.

[149] K. E. B. Hickman, *The SSL Protocol version 2.0*, internet draft ed., 1994.

[150] A. O. Freier, P. Karlton, and P. Kocher, *The SSL Protocol version 3.0*, internet draft ed., 1996.

[151] T. Dierks and C. Allen, *The TLS Protocol version 1.0*, rfc 2246 ed., January 1999.

[152] IETF, *Multipurpose Internet Mail Extensions (MIME). Part One: Format of Internet Message Bodies*, rfc 2045 ed., Internet Engineering Task Force, 1996.

[153] ——, *OpenPGP Message Format*, rfc 4880 ed., Internet Engineering Task Force, 2007.

[154] M. Bellare and P. Rogaway, "Entity Authentication and Key Distribution," in *Advances in Cryptology - Crypto '93*, vol. 773 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1993, pp. 239–249.

[155] S. Blake-Wilson and A. J. Menezes, "Entity Authentication and Authenticated Key Transport Protocols Employing Asymmetric Techniques," in *Security Protocols Workshop - IWSP '97*, vol. 1361 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1997, pp. 137–158.

[156] P. Morrissey, N. P. Smart, and B. Warinschi, "A Modular Security Analysis of the TLS Handshake Protocol," in *Advances in Cryptology - ASIACRYPT '08*, vol. 5350 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2008, pp. 55–73.

[157] NIST, *Entity Authentication Using Public Key Cryptography (FIPS –196)*, National Institute of Standards and Technology, 1997.

[158] S. A. Vanstone, "Responses to NISTs Proposal," in *Communications of the ACM*, C. by John Anderson, Ed., no. 35, July 1992, pp. 50–52.

[159] NIST, *Suite B Implementers Guide to FIPS 186-3 (ECDSA)*, National Institute of Standards and Technology, February 2010.

[160] E. Barker and J. Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, special publication 800-90a ed., National Institute of Standards and Technology, January 2012.

[161] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison–Wesley, 2001, vol. 2, Seminumerical Algorithms.

[162] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators," in *Fast Software Encryption - FSE '98*, Vaudenay, Serge, Ed., vol. 1372 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1998, pp. 168–188.

[163] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, January 2001.

[164] D. Davis, R. Ihaka, and P. Fenstermacher, "Cryptographic Randomness from Air Turbulence in Disk Drives," in *Advances in Cryptology - CRYPTO '94*, vol. 839 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1994, pp. 114–120.

[165] D. Eastlake, J. Schiller, and S. Crocker, *Randomness Requirements for Security (RFC 4086)*, IETF Network Working Group, 2005.

[166] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, special publication 800-22 revision 1a ed., National Institute of Standards and Technology, April 2010.

[167] N. Ferguson and B. Schneier, *Practical Cryptography*, C. A. Long, Ed. Wiley, 2003.

[168] S. Halevi and H. Krawczyk, "Strengthening Digital Signatures via Randomized Hashing," in *Advances in Cryptology CRYPTO '06*, C. Dwork, Ed., vol. 4117 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2006, pp. 41–59.

[169] R. McEvoy, J. Curran, P. Cotter, and C. Murphy, "Fortuna: Cryptographically Secure Pseudo-Random Number Generation In Software And Hardware," in *Irish Signals and Systems Conference - ISSC '06*. Institution of Engineering and Technology - IET, June 2006, pp. 457–462.

[170] J. Golic, "New Methods for Digital Generation and Postprocessing of Random Data," *IEEE Transactions on Computers*, vol. 55, no. 10, pp. 1217–1229, October 2006.

[171] M. Dichtl and J. D. Golić, "High-Speed True Random Number Generation with Logic Gates Only," in *Cryptographic Hardware and Embedded Systems - CHES '07*, vol. 4727 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2007, pp. 45–62.

[172] X. Xin, J. Kaps, and K. Gaj, "A Configurable Ring-Oscillator-Based PUF for Xilinx FPGAs," in *Euromicro Conference on Digital System Design DSD '11*. IEEE Computer Society, August 2011, pp. 651–657.

[173] Ç. K. Koç, Ed., *Cryptographic Engineering*, ser. Signals & Communication. Springer, 2009.

[174] K. Okeya, H. Kurumatani, and K. Sakurai, "Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications," in *Public Key Cryptography*, vol. 1751 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2000, pp. 238–257.

[175] R. Moloney, G. McGuire, and M. Markowitz, "Elliptic Curves in Montgomery Form with B=1 and Their Low Order Torsion," Cryptology ePrint Archive, Report 2009/213, 2009.

[176] ECRYPT. ebats (ecrypt benchmarking of asymmetric systems). http://bench.cr.yp.to/ebats.html.

[177] A. K. Lenstra and E. R. Verheul, "Selecting Cryptographic Key Sizes," *Journal of Cryptology*, vol. 14, pp. 255–293, 1999.

[178] M. Drutarovsky and M. Varchola, "Cryptographic System on a Chip based on Actel ARM7 Soft-Core with Embedded True Random Number Generator," in *Workshop on Design and Diagnostics of Electronic Circuits and Systems - DDECS '08*. IEEE Computer Society, 2008, pp. 1–6.

[179] D. Hankerson, "Implementing Elliptic Curve Cryptography (a narrow survey)," Institute of Computing UNICAMP Campinas, Brazil, April 2005.

[180] B. Glas, O. Sander, V. Stuckert, K. D. Müller-Glaser, and J. Becker, "Prime field ECDSA signature processing for reconfigurable embedded systems," *International Journal on Reconfigurable Computing*, vol. 5, pp. 1–12, January 2011.

[181] C. J. McIvor, M. Mcloone, and J. V. McCanny, "Hardware Elliptic Curve Cryptographic Processor Over GF(p)," *IEEE Transactions on Circuits and Systems*, vol. 53, no. 9, pp. 1946–1957, 2006.

[182] G. Orlando and C. Paar, "A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware," in *Cryptographic Hardware and Embedded Systems -CHES '01*, vol. 2162 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2001, pp. 356–371.

[183] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede, "Multicore Curve-Based Cryptoprocessor with Reconfigurable Modular Arithmetic Logic Units over GF(2^n)," *IEEE Transactions on Computers*, vol. 56, no. 9, pp. 1269–1282, September 2007.

[184] S. B. Örs, L. Batina, and B. Preneel, "Hardware implementation of an Elliptic Curve Processor over GF(p)," *International Journal of Embedded Systems*, vol. 3, no. 4, pp. 433–443, 2003.

[185] J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi, and I. Verbauwhede, "A compact FPGA-based architecture for elliptic curve cryptography over prime fields," in *Application-specific Systems Architectures and Processors - ASAP '10*, July 2010, pp. 313–316.

[186] B. Baldwin, R. Moloney, A. Byrne, G. McGuire, and W. P. Marnane, "A Hardware Analysis of Twisted Edwards Curves for an Elliptic Curve Cryptosystem," in *Applied Reconfigurable Computing - ARC '09*, vol. 5453 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2009, pp. 355–361.

[187] D. V. Bailey, B. Baldwin, L. Batina, D. J. Bernstein, G. V. Damme, G. D. Meulenaer, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, N. Mentens, C. Paar, F. Regazzoni, P. Schwabe, and L. Uhsadel, "The Certicom Challenges ECC2-X," in *Workshop on Special Purpose Hardware for Attacking Cryptographic Systems - SHARCS '09*, September 2009.

[188] B. Baldwin, W. Marnane, and R. Granger, "Reconfigurable Hardware Implementation of Arithmetic Modulo Minimal Redundancy Cyclotomic Primes for ECC," in *International Conference on Reconfigurable Computing and FPGAs - ReConFig '09*. IEEE Computer Society, December 2009, pp. 255–260.

[189] B. Baldwin and W. P. Marnane, "Yet Another SHA-3 Round 3 FPGA Results Paper," Cryptology ePrint Archive, Report 2012/180, 2012.

[190] B. Möller, "Fractional windows revisited: improved signed-digit representations for efficient exponentiation," in *Information Security and Cryptology - ICISC '04*, vol. 3506 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2004, pp. 137–153.

[191] A. Byrne, N. Meloni, F. Crowe, W. Marnane, and A. T. andE.M. Popovici, "Spa resistant elliptic curve cryptosystem using addition chains," *International Journal of High Performance Systems Architecture*, vol. 1, no. 2, pp. 133–142, 2007.

[192] K. Okeya and T. Takagi, "The width-W NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks," in *Topics in Cryptology - CT-RSA '03*, vol. 2612 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2003, pp. 328–343.

[193] T. S. Messerges, "Using second-order power analysis to attack DPA resistant software," in *Cryptographic Hardware and Embedded Systems - CHES '00*, Ç. K. Koç and C. Paar, Eds., vol. 1965 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2000, pp. 71–77.

[194] P. Yu and P. Schaumont, "Secure FPGA circuits using controlled placement and routing," in *Hardware/Software Codesign and System Synthesis - CODES+ISSS*. ACM, 2007, pp. 45–50.

[195] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks — Revealing the Secrets of Smart Cards*. Springer US, 2007.

[196] K. Tiri and I. Verbauwhede, “A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation,” in *Design, automation and test in Europe - DATE '04*, vol. 1. IEEE Computer Society, February 2004, pp. 246–251.