EFFICIENT FPGA IMPLEMENTATION OF ELLIPTIC CURVE SCALAR MULTIPLICATION OVER THE BINARY FIELD

Kuntapong Ruangsantikorakul

Bachelor of Engineering
Computer Engineering

Department of Electronic Engineering
Macquarie University

November, 2016

Supervisor: Dr. Yinan Kong
Co-supervisor: Md. Selim Hossain

ACKNOWLEDGMENTS

I would like to acknowledge and thank my supervisors Dr. Yinan Kong and Selim Hossain for their constant encouragement, enthusiasm, and guidance throughout my studies this past year and especially with this research.

I would also like to acknowledge my mother and father for their endless support just for me to receive a high level of education and take great opportunities that they wish they had themselves.

STATEMENT OF CANDIDATE

I, Kuntapong Ruangsantikorakul, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Electronic Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any academic institution.

Student's Name: Kuntapong Ruangsantikorakul

Student's Signature: Kuntapong Ruangsantikorakul (electronic)

Date: 1st September, 2016

ABSTRACT

Elliptic curve cryptography (ECC) is the more efficient alternative to the widely used Ron Rivest, Adi Shamir and Leonard Adleman (RSA) cryptosystem since 283-bit ECC provides the same security per bit as 3072-bit RSA. This makes it useful for resource constrained portable devices since smaller operands can be used. For practical use of ECC, an efficient hardware implementation must be achieved in terms of area and time. Elliptic curve scalar multiplication (ECSM) is the crucial operation required for ECC processors. The goal of this research is for the implementation of area efficient and high-speed ECSM over a binary field in field-programmable gate array (FPGA) technology for the binary fields $GF(2^{233})$ and $GF(2^{283})$. In order to achieve an efficient implementation of ECSM, the trade-off between area and time for two types of multipliers with various digit sizes was examined so that the most efficient hardware design can be achieved. Traditional digit-serial multipliers have the computational time complexity of $\lceil m/d \rceil$ clock cycles while the modified digit-serial multipliers had a time complexity of $2\lceil \sqrt{m/d} \rceil + 1$ clock cycles for the digit-size d. The comparison of affine coordinates and Jacobian coordinates were also analysed, which revealed that an implementation in affine coordinates is up to 13 times slower than that in Jacobian coordinates. In this research the efficient hardware implementation of ECSM on the Virtex-6 (XC6VLX760-2ff1760) FPGA device in $GF(2^{233})$ takes 16,048 clock cycles, has the computation time of 64.72μs, and occupies 35,615 slices. ECSM in the binary field $GF(2^{283})$ is performed in 13,565 clock cycles in 63.61μs, and occupies 69,908 slices on the same device.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cryptography consists of techniques used to secure information on public communication networks from adversaries. There are many schemes used for encryption, decryption and digital signatures which vary in security and complexity. These schemes are separated into private-key and public-key cryptography. Private-key cryptosystems use identical keys for both the receiver and the sender. Public key cryptosystems use two asymmetric keys where a private key is derived from a public key. Elliptic Curve Cryptography (ECC) is a public-key cryptography approach introduced by Miller [14] and Koblitz [10] in 1985.

ECC is an alternative to the widely used Ron Rivest, Adi Shamir and Leonard Adleman (RSA) cryptosystem [21]. ECC provides high levels of security per bit such that 256-bit ECC provides the same level of security as 3072-bit RSA. This allows significantly smaller key sizes which are favourable for portable devices to protect data with their limited resources. For ECC to be used in practical applications, the implementation of ECC must be efficient in terms of speed, area, and power. The focus of this research is to achieve an area-efficient, high-speed hardware implementation of Elliptic Curve Scalar Multiplication (ECSM) (also known as Elliptic Curve Point Multiplication (ECPM)) over a binary field using Field-Programmable Gate Array (FPGA) technology which can then be used for ECC protocols. ECC Protocols such as the Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) requires the computation of a point on a defined elliptic curve where it is infeasible to derive the private key from this point due to the elliptic curve discrete logarithm problem (ECDLP).

The main advantage of ECC over RSA is that it is computationally more efficient, useful in portable devices, and shorter operands can be used during computation. A summary comparing the Advanced Encryption Standard (AES), ECC, and RSA is shown in table 1.1 [17]. 3072-bit RSA is shown to achieve the same security level as 128-bit AES since there are lots of attacks available, so more bits are required. Although 3072-bit RSA can still be used, it is inefficient for practical use. This is where ECC is useful since the best attack used on ECC is much harder compared to RSA.

An implementation in hardware will produce results that are at least 37 times faster than software implementations of ECC [16]. An implementation in hardware also allows for modulo-2 arithmetic required for binary field operations to be efficiently implemented.

Cryptosystem	Equivalent Security Size				
AES (symmetric key)	80	112	128	192	256
ECC (public key)	163	233	283	409	7680
RSA (public key)	1024	2048	3072	7680	15360

Table 1.1: Comparison of security levels per bit in cryptosystems.

The chosen technology for the hardware implementation is FPGA which provides high flexibility for hardware design and development. FPGA is reprogrammable by using VHDL (Very High Speed Integrated Circuit Hardware Description Language) to describe digital signals and circuits. FPGA brings a cost and time effective design approach since multiple designs can be updated on the same device which does not need to be fabricated for each new implementation. The use of Xilinx products for hardware development such as Xilinx Integrated Software Environment (ISE) allows project management of VHDL code, generation of synthesis reports, generation of programming files and the use of Xilinx ISim for waveform simulations.

1.1 Project Scope

The primary goal of the project is to achieve an efficient hardware implementation of ECC over the binary field using FPGA. The scope of the project will only consist of the implementation of ECC up to the point of elliptic curve scalar multiplication. This means that the implementation of actual ECC protocols such as ECDH and ECDSA will be considered out-of-scope. The project will be completed with the following specifications:

- All implementations for FPGA will be done using VHDL.
- The implementation of Galois Field operations addition, squaring, multiplication, inversion and reduction will be over the binary fields $GF(2^{233})$ and $GF(2^{283})$.
- The implementation of elliptic curve group operations elliptic curve point doubling (ECPD) and elliptic curve point addition (ECPA) will be done using the National Institute of Standards and Technology (NIST) recommended non-supersingular elliptic curves and associated parameters.
- The implementation of ECSM will be achieved using the NIST recommended curves and parameters.

The aim of this project is also to explore alternative methods, algorithms and hardware architectures which will improve the efficiency of the computation of ECSM. Some implementations will improve the speed of the system while others will produce a better result in terms of area. Through trying different solutions, the trade-off between area and time can be analysed and the optimal solution can be proposed for ECSM over the binary fields $GF(2^{233})$ and $GF(2^{283})$.

Chapter 2 presents the background knowledge that is required for ECC to be used in the later chapters. The relevant theories, algorithms and hardware design is introduced for the foundations of ECC. Relevant work by other authors and the performances of their implementation is then noted which is later used as a comparison for the work of this project.

The experimental procedures used for the design, implementation and verification of the hardware architecture used for the ECSM processor is shown in chapter 3. This chapter describes the resources used and the typical design flow that was used for the implementation of each module used in this project.

In Chapter 4, the hardware design and development for Galois field arithmetic is proposed. The algorithms chosen are presented here which were then used to construct binary field addition, squaring, multiplication and inversion.

Chapter 5 then presents the hardware design and implementation for elliptic curve arithmetic using the hardware for Galois field arithmetic that was implemented in Chapter 4. In this chapter, ECPA and ECPD were constructed for affine coordinates and Jacobian coordinates. ECPA and ECPD were then used to implement the hardware which performs ECSM.

Chapter 6 reveals the optimisation techniques explored to improve the area and time efficiency of the overall ECSM hardware architecture. These include the design and implementation of modified multipliers, concurrent hardware methods, and the optimisation of ECPA and ECPD.

Chapter 7 then summarises the results of the hardware implementations proposed in chapter 4, 5, and 6. The synthesis reports generated from the VHDL design are analysed for advantages and disadvantages of the different methods used at each level of the ECSM architecture. The trade-off between area and time is then discussed where the most efficient design of ECSM in FPGA for $GF(2^{233})$ and $GF(2^{283})$ is proposed.

Chapter 2

Background and Related Work

2.1 Elliptic Curve Cryptography

A detailed overview of ECC was presented by Hankerson et al. [7] where they present core concepts, algorithms, and explores various methods of implementation. The ECC scheme can be broken down into a hierarchy of operations as shown in figure 2.1. Each level of this hierarchy depends on the efficiency of the underlying level of operations. Because of this, it is highly important that the Galois Field operations are implemented as efficiently as possible in a bottom up approach to achieve a high performance ECSM hardware architecture. There are different techniques that can be used to improve the efficiency of operations at each level of operation that will be discussed in the following subsections.

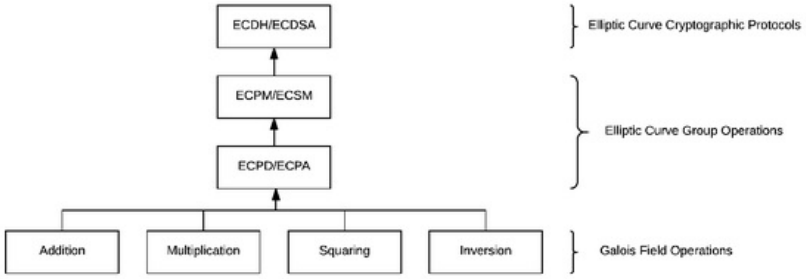


Figure 2.1: Hierarchy of operations for elliptic curve cryptography.

2.2 Galois Field Arithmetic

Galois Field Arithmetic (GFA) also known as finite field arithmetic (FFA) is crucial in the implementation of ECC. There are three types of finite fields as shown in [7], including prime fields, binary fields, and optimal extension fields. Modulo-2 arithmetic is very efficient and easily implemented in hardware and so a focus on the binary field will be used for this research. A binary field has an order of 2^m with a maximum degree of $m-1$ and can be denoted as $GF(2^m)$. A polynomial basis is used to represent elements in the field such that x^3+x^2+1 is represented in $GF(2^4)$ as 1101_2. $f(x)$ is used to denote the reduction polynomial with a degree of at most m for a chosen binary field. Any arithmetic using a binary field is performed modulo the corresponding reduction polynomial of the binary field used. The following subsections will introduce addition, squaring, multiplication and inversion over $GF(2^m)$.

2.2.1 Addition over the Binary Field

Addition is the simplest operation over a binary field which can be achieved in one clock cycle. Adding two elements in $GF(2^m)$ is done using bit-wise exclusive-or (XOR). Using bit-wise XOR is the best method of addition over a binary field and is used for all ECC implementations.

2.2.2 Squaring over the Binary Field

Squaring is the multiplication of two identical binary polynomials. Squaring in $GF(2^m)$ is performed in m clock cycles [8]. Squaring can be achieved by inserting a '0' bit between each bit of a binary polynomial followed by reduction of the result. The result of squaring must be reduced if it lies outside of $GF(2^m)$. A squaring method is often implemented since it can be more efficient than the multiplication of two identical numbers. This consideration will depend on the efficiency of multiplication implemented, since if multiplication becomes more efficient that squaring, then the squaring method can be replaced.

Reduction

Reduction may also need to be performed on the result of squaring and multiplication to ensure that the result exists within the binary field chosen. Reduction uses the reduction polynomial $f(x)$ of the corresponding binary field so that a binary polynomial $a(x)$ is reduced to $r(x)$, where $r(x) = a(z)\ mod\ f(x)$. Reduction can be done one bit at a time through modular reduction, or by fast reduction algorithms are also shown by Hankerson for the NIST recommended binary fields for $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$ [7].

2.2.3 Multiplication over the Binary Field

Multiplication is the second most expensive operation over a binary field. Similar to squaring, multiplication can also be done through normal polynomial multiplication and addition of partial products followed by a reduction of the final result. However, this is inefficient and more popular algorithms use interleaved modular reduction so a specific reduction method is not required. The common methods to perform multiplication include multiplication with an interleaved modular reduction algorithm [8], bit-serial multiplication, radix-multipliers, digit-serial multiplication, and digit-parrallel multiplication [7] [11]. Pan have also proposed a more efficient version of the digit-serial multiplication algorithm in [11].

2.2.4 Inversion over the Binary Field

Inversion is the most expensive operation of binary field arithmetic for ECC. The inverse of a non-zero element $a \in GF(2^m)$ is $g \in GF(2^m)$, where $ag = 1 \ mod \ f(x)$ and $g = a^{-1}$. Inversion takes approximately double the time for multiplication in the same binary field as shown in table 2.1, [8] and [6]. Popular algorithms for inversion used for ECC implementation include the extended Euclidean algorithm and the almost inverse algorithm. Another algorithm for inversion is the Itoh-Tsuji algorithm based on Fermat's little theorem [9], where a high-speed implementation was achieved in [18]. A systolic array implementation of a modified Euclid's algorithm for inversion is also proposed in [5].

2.3 Elliptic Curve Scalar Multiplication

2.3.1 Elliptic Curve Group Operations

Elliptic Curve Cryptography is the use of elliptic curves over finite fields for key exchange, encryption and digital signatures. ECC is based on the elliptic curve discrete logarithm problem (ECDLP) where it is infeasible to derive a private key, k from a public key $Q = kP$, where k is an integer value and P is a point on the elliptic curve. There are multiple elliptic curves that can be used. However, the elliptic curves as recommended by the National Institute of Standards and Technology (NIST) [15] use a non-supersingular elliptic curve for ECC over binary fields as shown by equation 2.1. ECSM is used to compute multiples of points on an elliptic curve by the use of repeated addition and doubling of points on an elliptic curve. Elliptic curve point doubling and point addition are illustrated as shown by the graphs in figure 2.2. There are multiple algorithms that can be used for elliptic curve scalar multiplication. The simplest is the right-to-left double-and-add method. Other methods that have been implemented include the binary/window non-adjacent form (NAF) method and the Montgomery Ladder method [2, 7, 16].

The points on an elliptic curve can be represented using different coordinate systems such as affine coordinates and Jacobian projective coordinates which will be discussed further in section 2.3.2 and 2.3.3. The choice of coordinate system will determine how

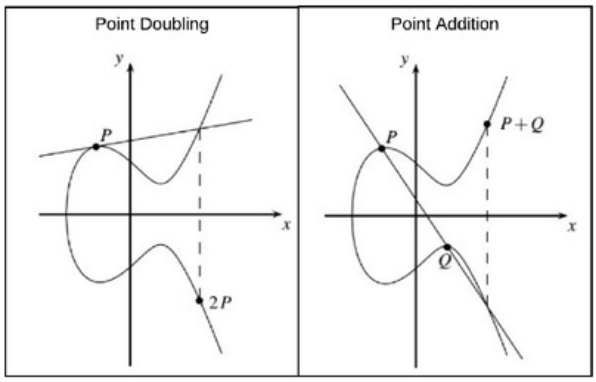


Figure 2.2: Point addition $(P + Q)$ and point doubling $(2P)$ on an elliptic curve [20].

	Affine		Jacobian	
	Point Addition	Point Doubling	Point Addition	Point Doubling
Inversion	1	1	0	0
Multiplication	2	2	15	5
Squaring	1	2	5	5
Addition	9	5	7	4

Table 2.1: Comparison of operations required for point addition and point doubling in affine coordinates and Jacobian coordinates.

ECPA and ECPD will be constructed, as a different number and combination of underlying Galois field operations will be used. One advantage of projective coordinates over affine coordinates is the minimisation of inversion, however many more binary field operations for multiplication and squaring must be used. A summary of the operations used for ECPA and ECPD are shown in table 2.1. Other coordinate systems have also been used in literature such as Lopez-Dahab coordinates and the Montgomery method of point addition and doubling, which can offer even more improvements as shown by implementations in [19] and [4].

2.3.2 Affine Coordinates

Affine coordinates can be used to represent points on an elliptic curve. The set of points for a binary field is the set of solutions to equation 2.1 of a non-supersingular elliptic curve E. A point P is shown by $P(x, y) : x, y \in GF(2^m)$. Elliptic curve point addition and point doubling is shown in equation 2.2 and equation 2.3 respectively [7]. Point addition is used for the addition of two different points, while point doubling is used for the addition

of identical points.

$$E : y^2 + xy = x^3 + ax^2 + b \tag{2.1}$$

$$\begin{aligned} R(x_3, y_3) &= P(x_1, y_1) + Q(x_2, y_2) \in E \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a, \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1, \\ \lambda &= (y_1 + y_2)/(x_1 + x_2) \end{aligned} \tag{2.2}$$

$$\begin{aligned} R(x_2, y_2) &= 2P(x_1, y_1) \in E \\ x_2 &= \lambda^2 + \lambda + a, \\ y_2 &= x_1^2 + \lambda x_2 + x_2, \\ \lambda &= (x_1 + y_1)/x_1 \end{aligned} \tag{2.3}$$

2.3.3 Jacobian Coordinates

A projective coordinate system such as Jacobian coordinates can also be used to perform ECPD and ECPA. The non-supersingular curve used for Jacobian projective coordinates is the set of solutions to equation 2.5. Jacobian coordinates are more appealing than affine coordinates since these two operations can be performed without the need of inversion. Inversion is only needed for converting from Jacobian coordinates to affine coordinates and this can be achieved using equation 2.4. Just like affine coordinates, point addition shown in equation 2.6 is used for the addition of two different points while point doubling shown in equation 2.7, is used for the addition of two identical points [16].

$$\begin{aligned} \text{Affine}(X/Z^2, Y/Z^3) = \text{Jacobian}(X, Y, Z), \\ \textit{where } Z \neq 0 \end{aligned} \tag{2.4}$$

$$E : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6 \tag{2.5}$$

$$
\begin{aligned}
R(X_2, Y_2, Z_2) &= P(X_0, Y_0, Z_0) + Q(X_1, Y_1, Z_1) \in E, \\
U_0 &= X_0 Z_1^2 \\
U_1 &= X_1 Z_0^2 \\
S_0 &= Y_0 Z_1^3 \\
S_1 &= Y_1 Z_0^3 \\
W &= U_0 + U_1 \\
R &= S_0 + S_1 \\
L &= Z_0 W \\
V &= RX_1 + LY_1 \\
Z_2 &= LZ_1 \\
T &= R + Z_2 \\
X_2 &= aZ_2^2 + TR + W^3 \\
Y_2 &= TX_2 + VL^2
\end{aligned}
\tag{2.6}
$$

$$
\begin{aligned}
R(X_2, Y_2, Z_2) &= 2P(X_1, Y_1, Z_1) \in E, \\
X_2 &= X_1^4 + bZ_1^8 \\
Y_2 &= X^4 Z_2 + UX_2 \\
Z_2 &= X_1 Z_1^2 \\
U &= Z_2 + X_1^2 + Y_1 Z_1
\end{aligned}
\tag{2.7}
$$

2.4 Related Work

There have been many implementations of ECSM in the literature which present several approaches for the design of an efficient hardware system. In this section a focus on FPGA implementations of ECSM over a binary field will be explored. One implementation of ECSM is shown by Hossain et al [8], where their design is able to perform one point multiplication in 2.66ms and 5.54ms in the field sizes of $GF(2^{233})$ and $GF(2^{283})$ respectively. The methods used in their implementation present an area-efficient implementation which uses bit-serial multiplication, the modified extended Euclidean algorithm and performs ECPA and ECPD in affine coordinates. The clock cycles required for their binary field operations are shown in table 2.2. Their paper also suggests the use of Jacobian coordinates for further improvement on their design.

Operation	Addition	Multiplication	Squaring	Inversion
Clock cycles	1	233	233	467

Table 2.2: Clock cycles of binary field operations using bit-serial multiplication and the modified extended Euclidean algorithm.

An implementation by Govem in [3] requires 65,000 clock cycles with a computation time of 0.24ms on a Xilinx Virtex-5 device. Lutz and Han implemented their design on the Virtex-E device and were able to achieve a computation time of 0.075ms using Lopez-Dahab projective coordinates in $GF(2^{233})$.

Due to the disadvantages of using inversion in affine coordiantes, there has been a shift to Jacobian coordinates which will require efficient multiplication algorithms. Bit-serial multiplication in $GF(2^m)$ performs a multiplication in m clock cycles. Multiple implementations use digit-serial multiplication as shown in [7] which reduces the latency of multiplication to $\lceil m/d \rceil$ clock cycles. Pan and Lee have worked on improvements on binary field multiplication where they propose digit-serial and digit-parrallel systolic multipliers in [11], where a binary field multiplication can be achieved in $2 \cdot \lceil \sqrt{m/d} \rceil$ clock cycles. Another multiplication algrithm used for ECSM implementations is the Karatsuba-Ofman algorithm which was used in [12], where a ECSM is achieved in 0.059ms in $GF(2^{163})$, 0.084ms in $GF(2^{233})$ and 0.102ms in $GF(2^{283})$. A hardware implementation which also uses digit-serial multiplication is presented in [22] which produces a latency of 0.089ms for ECSM in the binary field of $GF(2^{233})$.

Chapter 3

Experimental Procedures

This chapter presents the resources and experimental procedures used in conducting research for the hardware design, implementation and testing of an ECSM processor. A description of resources used is presented in section 3.1 followed by the experimental procedures in section 3.2.

3.1 Resources

The resources used consist mostly of Xilinx software products which were downloaded from www.xilinx.com/products/design-tools. A free (but restricted) Xilinx 'WebPACK' license was used for this project. The Xilinx Integrated Software Environment (ISE) was used for FPGA design and development using VHDL. The Xilinx ISE was used for compiling the design from VHDL code, waveform simulations using Xilinx ISim, generating synthesis reports, and to generate the programming file that can be used to load a design onto an FPGA hardware device. Xilinx iMPACT or Diligent Adept software can then be used for loading a hardware design onto FPGA devices. To minimise risks associated with the use of software, a private GitHub repository was used for back-up and as a version control system for all VHDL files and Xilinx ISE projects.

3.2 Experimental Procedures

To conduct the experiments for this project, a design flow is required for the construction of a digital system from VHDL for FPGA devices. As identified in the background in chapter 2, this research will require the hardware design for multiple modules such as for Galois field operations (addition, squaring, multiplication and inversion) and the elliptic curve group operations (ECPA, ECPD and ECSM). A complete list of modules to be designed and implemented for both binary fields $GF(2^{233})$ and $GF(2^{288})$ include the following:

- Addition

- Traditional digit-serial multipliers (for the digit sizes of 1, 2, 4, 8, 16, 32, and 64)
- Inversion
- Affine point doubling (APD)
- Affine point addition (APA)
- Jacobian point doubling (JPD)
- Jacobian point addition (JPA)
- ECSM controller in affine coordinates
- ECSM controller in Jacobian coordinates
- Conversion from Jacobian to Affine coordinates
- Serial-in-parallel-out (SIPO)
- Parallel-in-serial-out (PISO)

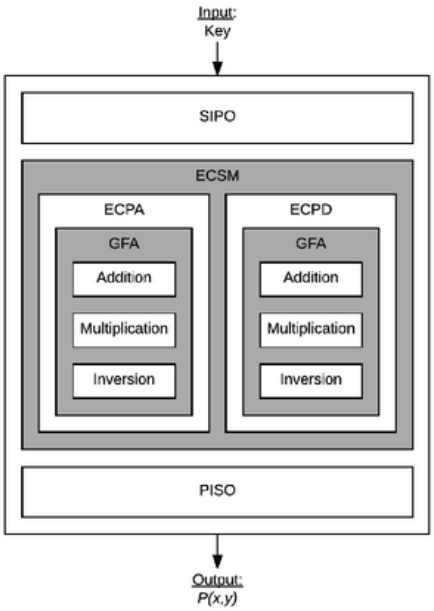


Figure 3.1: Simplified top-level model for the ECSM architecture in FPGA.

The completion of the above modules will result in the initial implementations of an ECSM hardware architecture as shown in figure 3.1. These will then be optimised with the addition of the design and implementation of the following modules which are discussed further in chapter 6:

- Modified digit-serial multipliers (for the digit sizes of 1, 2, 4, 8, 16, 32, and 64)
- Concurrent ECSM controllers for both affine and Jacobian coordinates
- Combined Jacobian point addition and point doubling (PAPD)

The design flow used will be repeated for each module to be constructed which is outlined as follows:

1. Algorithm selection
2. Logic Modeling
3. VHDL coding
4. Syntax and behavioral verification
5. Testing and waveform simulations
6. Synthesis
7. Comparison and analysis of hardware designs

3.2.1 Algorithm selection

This is the first step taken in the design flow to create a hardware module. The present literature related to binary field arithmetic and elliptic curve group operations were analysed for efficient methods, algorithms and hardware architectures that could be used in this research. The timing complexities and area of designs were considered before an algorithm was chosen for implementation.

3.2.2 Logic Modeling

The algorithms selected were then modeled in various ways to assist in easing the transition from an algorithm to a hardware design using VHDL. Trying to implement the VHDL code for large binary fields of $GF(2^{233})$ and $GF(2^{233})$ can be difficult. So smaller field sizes such as $GF(2^4)$ were initially used where examples using the algorithm could be done using pen and paper. Flow-charts were also used which were beneficial in showing the data flow of an algorithm.

3.2.3 VHDL coding

Once an algorithm was modeled, the VHDL code could then be created. The VHDL code was written using Xilinx ISE. A new entity and separate VHDL file was created for each new module required.

3.2.4 Syntax and behavioral verification

VHDL coding is not complete until the syntax and behaviour checks for a hardware design is successful. These checks are available as tools in the Xilinx ISE. If these checks fail then the errors must be corrected by modifying the VHDL code.

3.2.5 Testing and waveform simulations

In this phase, the VHDL code for a module is verified. This requires the creation of a VHDL test bench where inputs to the designed module can be chosen. This test bench is then used for simulations using Xilinx ISim where the waveform output is checked against the expected value(s) and delays. If the results of the simulation are not correct then the VHDL code and logic models must be checked to ensure the correct hardware implementation of an algorithm that is free of errors.

3.2.6 Synthesis

Once the tests are correct, then the hardware design in VHDL can be synthesised to a target FPGA device using the Xilinx Synthesis Tool (XST). The target FPGA device used in this research is the Xilinx Virtex-6 (XC6VLX760-2ff1760). The XST generates a synthesis report for the design which displays the area and timing constraints such as slice registers, slice look-up-tables (LUT), LUT flip-flop pairs, minimum clock periods, and clock frequencies. Minor changes in VHDL code can result in different hardware performances analysed from the synthesis reports in terms of time and area so the design should be optimised for the most efficient results.

3.2.7 Comparison and analysis of hardware designs

Finally the hardware performances shown by the synthesis reports are recorded. Different hardware implementations are then analysed and compared against each design with a focus on area and time. The most efficient design is then chosen to be used in the final ECC processor for $GF(2^{233})$ and $GF(2^{283})$.

For this research, the hardware performances from the synthesis reports were recorded as the results in chapter 7. For the implementation on an FPGA device, a bit file would need to be generated using Xilinx ISE. This bit file is then loaded onto the FPGA from a computer using a JTAG cable and either Xilinx iMPACT or Digilent Adept software.

Chapter 4

Hardware for Galois field arithmetic over $GF(2^m)$

This chapter will show the hardware design and implementation of Galois field arithmetic for this research using VHDL for FPGA devices. Each section will discuss the design process leading up to creation of VHDL code including algorithms chosen, flow chart models, and block diagrams. The related background shown in section 2.2 identifies that addition, multiplication, squaring and inversion modules will be necessary for ECC. The design methods used for this project can be used for any sized binary field. However, only the hardware designs for $GF(2^{233})$ and $GF(2^{283})$ have been implemented as specified by the project scope.

The design of each module is shown in figure 4.1, where each module also has the inputs 'clock', 'run', and 'reset', and output 'done'. The input 'reset' is used to reset all internal signals of a module, 'run' is used to start a module and keep it running for the duration that run is '1'. The 'done' signal then indicates when the module has produced a final output. The designs use an asynchronous reset. These external signals aid the correct timing of Galois field operations in ECPA and ECPD since each operation will have different delays. The modular design of finite field arithmetic will allow for modifications of the system which can easily be achieved by replacing an individual module.

In this project, a specific squaring module has not been created and a multiplication module will be used instead to perform squaring more efficiently. The reason for this is identified from table 2.2 as shown by [8]. This implementation showed that an implementation in $GF(2^{233})$ and $GF(2^{283})$ for bit-serial multiplication requires the same number of clock-cycles and area as squaring. Once the efficiency of multiplication is improved, a multiplication module can then be used to replace the squaring module since it will bring better area and time and thus improve the efficiency of the ECSM hardware architecture. The exclusion of a squaring module will also allow the simpler design of elliptic curve point doubling and point addition as the scheduling and ordering of operations will not be greatly affected if all multiplication and squaring modules have identical latencies.

As shown earlier in section 2.4, inversion is the most expensive operation in Galois field arithmetic over the binary field followed by multiplication and squaring. This is

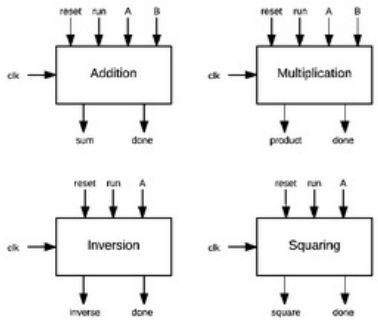


Figure 4.1: The top level block diagrams of Addition, Multiplication, Squaring and Inversion.

summarised in table 2.2 which shows the clock cycles for an implementation of binary field arithmetic in FPGA using bit-serial multiplication and a modified extended Euclidean algorithm [8]. Inversion is shown to have approximately twice the clock cycles required for multiplication and squaring. The goal of implementing efficient Galois field operations will require the selection of algorithms which reduce these clock cycles. It is more difficult to reduce inversion than multiplication.

This chapter focuses on the design and implementation of efficient addition, inversion and multiplication modules. As shown previously in table 2.1, Jacobian coordinates can be used to perform ECPA and ECPD without the need of inversion. However, many more multiplications and squaring are needed which need to be implemented efficiently. In order for an implementation of ECSM in Jacobian coordinates to have an advantage over affine coordinates, the clock cycles required to perform ECPA and ECPD in Jacobian coordinates need to be less than that to achieve ECPA and ECPD in affine coordinates.

4.1 Design and implementation of Galois field arithmetic over binary fields

4.1.1 Addition over binary fields

The addition module was made using bit-wise XOR of two inputs as shown in algorithm 1. A flow chart representing the VHDL representation is shown in figure 4.2. The hardware architecture of this design for addition in $GF(2^m)$ is shown in figure 4.3.

Algorithm 1 Addition in $GF(2^m)$

Input: Binary polynomials $a = (a_{m-1}, ..., a_1, a_0)$, $b = (b_{m-1}, ..., b_1, b_0)$.
Output: $c = a + b$.
1: **for** $i = 0$ to $m - 1$ **do**
2: $c_i = a_i \oplus b_i$.
3: **end for**
4: **return** c.

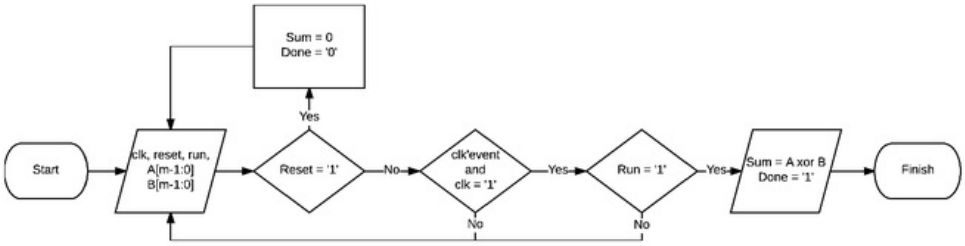


Figure 4.2: Flowchart for the addition module.

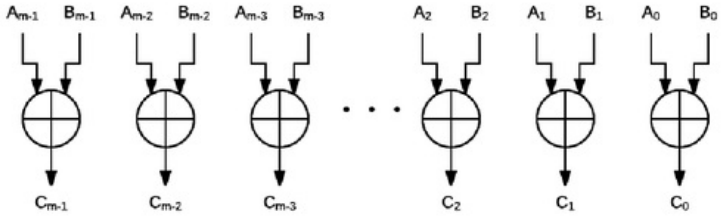


Figure 4.3: Hardware design for addition over the binary field.

4.1.2 Multiplication in $GF(2^m)$

Multiplication is important for efficient ECC implementation over binary fields. Multiplication can be done in multiple ways but bit-serial and digit-serial multiplication were used for this design. A motivation for improving the efficiency of multiplication is for the speed up of ECSM using Jacobian coordinates. By multiplying 2-bits at a time in digit-serial multiplication, the clock cycles required to perform binary field multiplication will be halved. The following sections present the algorithms and hardware architectures which were used to implement bit-serial multiplication and digit-serial multiplication of digit-sizes up to 64-bits using VHDL.

Bit-Serial Multiplication

Bit-serial multiplication creates one partial product for each bit of the input during multiplication. An example of the multiplication logic used is shown by binary multiplication as shown in figure 4.4. In each clock cycle, one partial product is generated and accumulated in the product C. To obtain a result in $GF(2^m)$, in each clock cycle the product C also undergoes reduction. A simple example of the reduction logic is shown in figure 4.5 which focuses on calculating the remainder using binary long division. This illustrates that reduction is achieved by checking each bit with a degree greater than m in the dividend whether or not it is a '1' or a '0'. If it is a '1' then the left-most '1' bit of the dividend and the divisor are aligned and undergo bit-wise XOR. This process is repeated until a remainder is determined. Algorithm 2 was used to implement Galois field multiplication from the least significant bit (LSB) to the most significant bit (MSB) where a multiplication in $GF(2^m)$ will take m clock cycles. This implementation of bit-serial multiplication actually uses the same algorithm as digit-serial multiplication with the digit-size of 1. The next section will discuss the hardware architecture required to perform digit-serial multiplication.

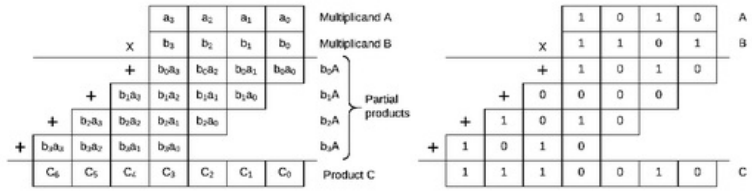


Figure 4.4: General method for Galois field multiplication where an m-bit multiplier and multiplicand will produce m partial products and the product will have a size of $2m$-1 bits. An example is given for the multiplication of 4-bit binary numbers 1010_2 and 1101_2

.

Algorithm 2 Multiplication using right-to-left shift and add

Input: Binary polynomials $a = (a_{m-1}, ..., a_1, a_0)$, $b = (b_{m-1}, ..., b_1, b_0)$ and Reduction Polynomial $F(z) = z^m + r(z)$.
Output: $c = a \cdot b \ mod \ F(z)$.

1: $c \leftarrow 0$.
2: **for** $i = 0$ to $m - 1$ **do**
3: $\quad c \leftarrow c \oplus b_i a$.
4: $\quad a \leftarrow a \cdot z \oplus a_{m-1} r$.
5: **end for**
6: **return** c.

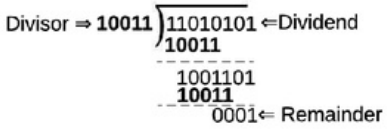


Figure 4.5: General method for reduction of a binary polynomial when multiplication of 2 elements in $GF(2^m)$ produces a product with degree greater than m. An example is given for the reduction of 11010101_2 by $f(x) = 10011_2$ for a result in $GF(2^4)$. This is equivalent to the computation of 11010101_2 mod $10011_2 = 0001_2$.

Traditional Digit-Serial Multiplication

Once the bit-serial multiplier was completed, the digit-serial multiplier was then designed based on algorithm 3 [11]. Digit-serial multipliers will speed up the multiplication process significantly compared to the bit-serial multiplier as multiple partial products are generated and added per clock, thus reducing clock cycles. However, this method of reducing clock cycles will increase the area of hardware used. Different digit sizes were implemented for 2-bit, 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit digit-serial multipliers so that their area-efficiency and delays can be recorded and compared. The most efficient multiplier can then be used for the later stages of the project in order to achieve the best area-time for ECPA and ECPD. The clock cycles required for multiplication over a binary field using traditional digit-serial multipliers will be $\lceil m/d \rceil$, where d is the digit size.

The implementation of traditional digit-serial multiplication used in this project required the use of registers shown in table 4.1. The number of padding bits for register 'Bv' is determined by first calculating $q = \lceil m/d \rceil$. A detailed flow-chart for the design of a traditional digit-serial multiplier in FPGA is shown in figure 4.8.

Register Name	Description	Bit-size
Av	Used to store the new multiplier representing A from 'shiftModA' for the next iteration.	m
RP	Reduction polynomial $f(x)$.	$m+1$
tempA	Used to initialise and perform multiplication.	m
shiftModA	Used to perform $A \cdot x^d \ mod \ F(x)$.	$m+d$
Bv	Used for padding the B input by adding '0' bits to the MSB of B.	$m + (qd - m)$
Cv	Used to store the product during each multiplication iteration.	$m + d$-1

Table 4.1: Registers used for $GF(2^m)$ traditional digit-serial multiplication.

Algorithm 3 Multiplication using right-to-left traditional digit-serial multiplication

Input: Binary polynomials $A = (a_{m-1}, ..., a_1, a_0)$, $B = (b_{m-1}, ..., b_1, b_0)$, reduction polynomial $F(z) = z^m + r(z)$.
Output: $C = A \cdot B \ mod \ F(x)$.

1: $\bar{C} \leftarrow 0$.
2: $B = B_0 + B_1 x^d + ... + B_{q-1} x^{d(q-1)}$, where $B_i = \sum_{j=0}^{d-1} b_{id+j} x^j$.
3: **for** $i = 0$ to $q - 1$ **do**
4: $\quad \bar{C} \leftarrow \bar{C} \oplus B_i A$.
5: $\quad A \leftarrow A \cdot x^d mod F(x)$.
6: **end for**
7: $C \leftarrow \bar{C} \ mod \ f(x)$.
8: **return** C.

A hardware architecture is shown in figure 4.6 which implements the above algorithm. This method of multiplication is efficient in hardware since it uses left shifts and addition. In each iteration, B_i is multiplied with $tempA$ which contains $A \cdot x^{di} mod F(x)$ of the current iteration. While the multiplication of B_i and $tempA$ is performed, the computation of $A \cdot x^{d(i+1)}$ mod $f(x)$ is done in parallel. The current multiplicand $tempA$ is left shifted by d bits followed by reduction mod $f(x)$. At the final iteration, register Cv can have results with binary polynomials of degree greater than m from the accumulation of partial products. So reduction of 'Cv' is also needed to compute the final output C.

The reduction used in the block diagram of multiplication to perform $mod \ f(x)$ in figure 4.6 is shown in figure 4.7. The way reduction was implemented in the hardware architecture for this project was by connecting all bit positions representing a binary polynomial with a degree more than m to be selectors to a multiplexor each. For each '1' bit of Cv greater than C_{m-1}, the contents of Cv from $Cv(C_n \ downto \ C_n - m)$ for the bit-size of m will be added with the reduction polynomial such that $Cv(C_n \ downto \ C_n - m) = Cv(C_n \ downto \ C_n - m) \ xor \ f(x)$, where $n > m$. This allows for reduction of a binary polynomial with degree greater than m to be achieved in one clock cycle in hardware. This approach for reduction follows the same methodology shown previously for binary long division to find the remainder as shown in figure 4.5.

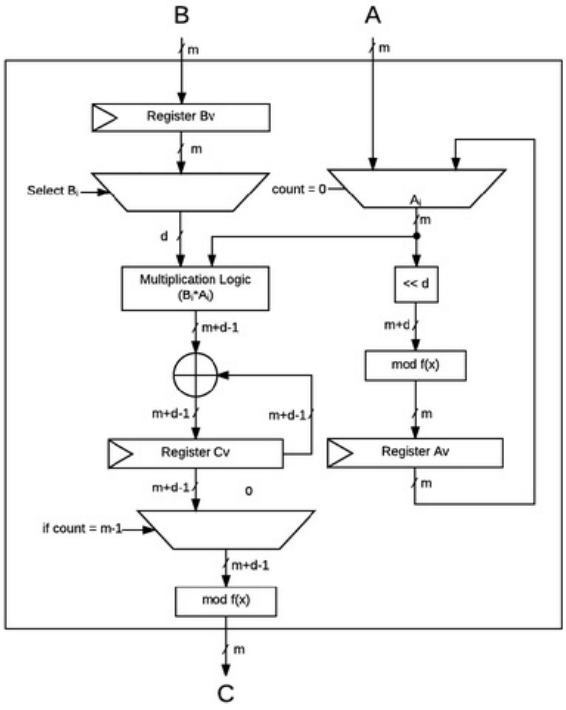


Figure 4.6: Block diagram for multiplication of m-bit binary polynomials where the product $C = A \cdot B \ mod \ f(x)$.

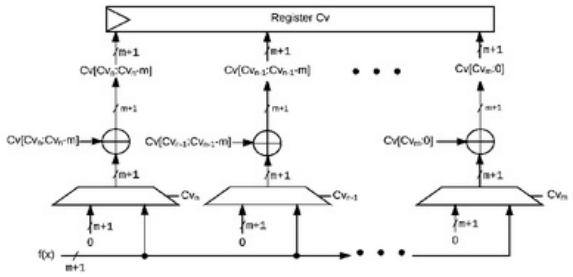


Figure 4.7: Block diagram for the reduction of a binary polynomial greater than m-bits used to perform "$mod \ f(x)$" during multiplication.

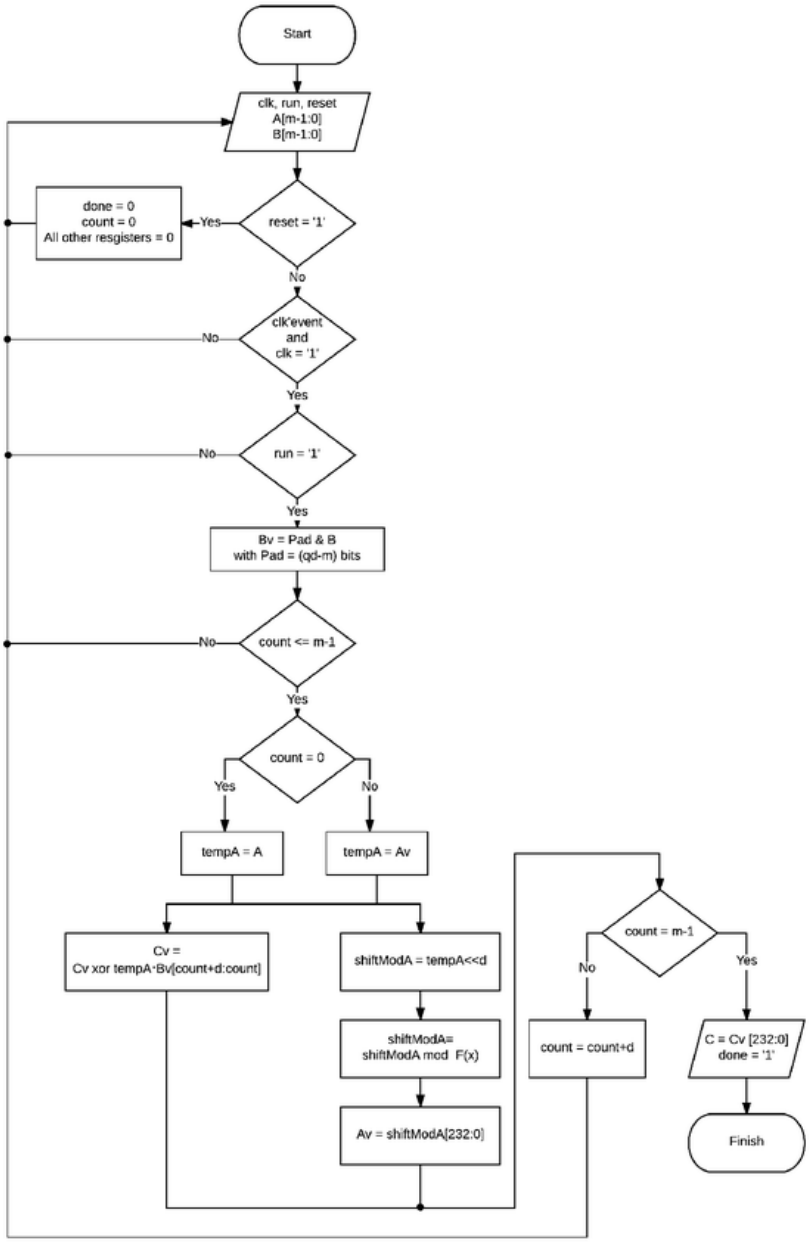


Figure 4.8: Flowchart for the implementation of m-bit digit-serial multiplication with digit size d.

4.1.3 Inversion in $GF(2^m)$

Inversion has the most costly computation compared to the other finite field operations as shown in table 2.2. Inversion is needed to perform division using finite-field operations. The division $A(x)/B(x)$ *mod* $f(x)$ is performed by first finding the inverse B^{-1} and secondly performing the multiplication where $A(x) \cdot B^{-1}(x) = A(x)/B(x)$. The inversion module implemented in this research project consists of 5 registers, *V, Qv, Zv, Pv* and *Count*. Registers *V, Qv, Zv* and *Pv* all have the bit-size of $m + 1$, and *Count* has the size of 2 bits.

The algorithm used takes the input $U(x)$ and returns the output $Z(x)$ which is the modular inverse of $U(x)$. Register Pv is initialised with $U(x)$ padded with a '0' bit. Register Qv is initialised with the reduction polynomial of the binary field used. Register Zv is initialised with the binary value of 1. Register V and *Count* are initialised with the value of zero. The inversion module used for this project is based on algorithm 4. This algorithm used was proposed by Brunner in [1], and implemented in [8] and [5]. The internal register operations in every clock cycle is summarised in table 4.2. The time complexity of this implementation of inversion in terms of clock cycles is $2m + 2$ for the binary field $GF(2^m)$. A detailed flow-chart which shows the hardware implementation in VHDL is shown in figure 4.10 and the hardware representation of this design is then given in figure 4.10.

Register	Next Register State			
	Init	Pv(m)= 0	Pv(m) = 1	
			Count = 0	Count != 0
Pv	'0'& A(x)	$x \cdot Pv$	$x\cdot$ (Pv $\oplus$ Qv)	Pv
Qv	$F(x)$	Qv	Pv	$x \cdot T$
Zv	1	$x \cdot Zv$	$x\cdot$ (V $\oplus$ Zv)	Zv / x mod $F(X)$
V	0	V	Zv	V $\oplus$ Zv
Count	0	Count+1	Count+1	Count-1

Table 4.2: Summary of register operations required when performing inversion.

Algorithm 4 Inversion in GF(2^m) with Modified Euclidean Algorithm.

Input: $U(x) = (u_{m-1}, ..., u_1, u_0)$, reduction polynomial $F(z)$.
Output: $Z(x) = 1U(x)\ mod\ F(x)$.

1: $P_v = `0' \& U(x), Q_v = F(x), Z_v = 1, V = 0; cnt = 0.$
2: **for** $i = 1$ to $2m$ **do**
3: **if** $P_v(m) =' 0'$ **then**
4: $P_v = P_v \cdot x.$
5: $Z_v = Z_v \cdot x.$
6: **if** $Z_v(m) =' 1'$ **then**
7: $Z_v = Z_v + F(x).$
8: **end if**
9: $cnt = cnt + 1.$
10: **else**
11: **if** $Q_v(m) = `1'$ **then**
12: $Q_v = Q_v + P_v.$
13: $V = V + Z_v mod F(x).$
14: **end if**
15: $Q_v = Q_v \cdot x.$
16: **if** $cnt = 0$ **then**
17: $P_v \leftrightarrow Q_v.$
18: $Z_v \leftrightarrow V.$
19: $Z_v \cdot x\ mod\ F(x).$
20: $cnt = cnt - 1;$
21: **else**
22: $Z_v = Z_v / x\ mod\ F(x).$
23: $cnt = cnt - 1.$
24: **end if**
25: **end if**
26: **end for**
27: $Z = Z_v(m - 1 downto 0).$
28: **return** $Z.$

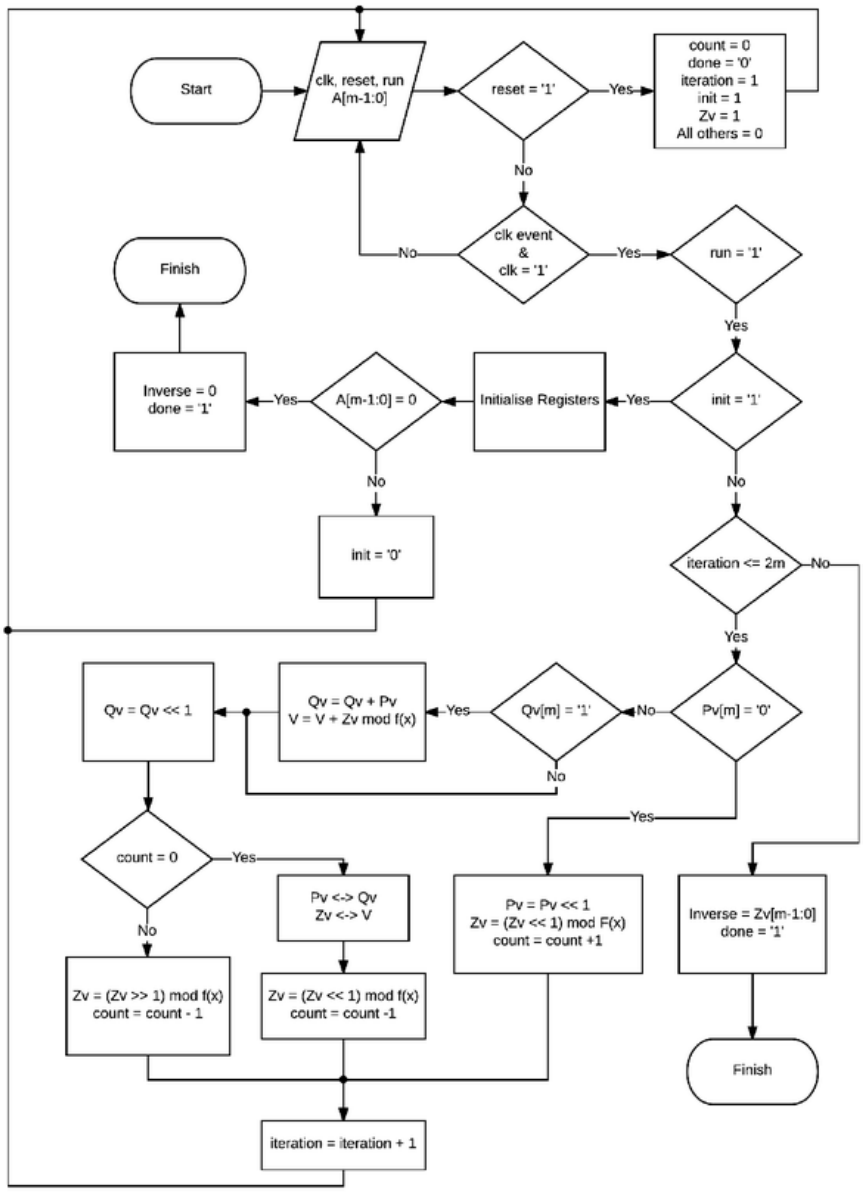


Figure 4.9: Flowchart representing the hardware design of inversion over binary fields.

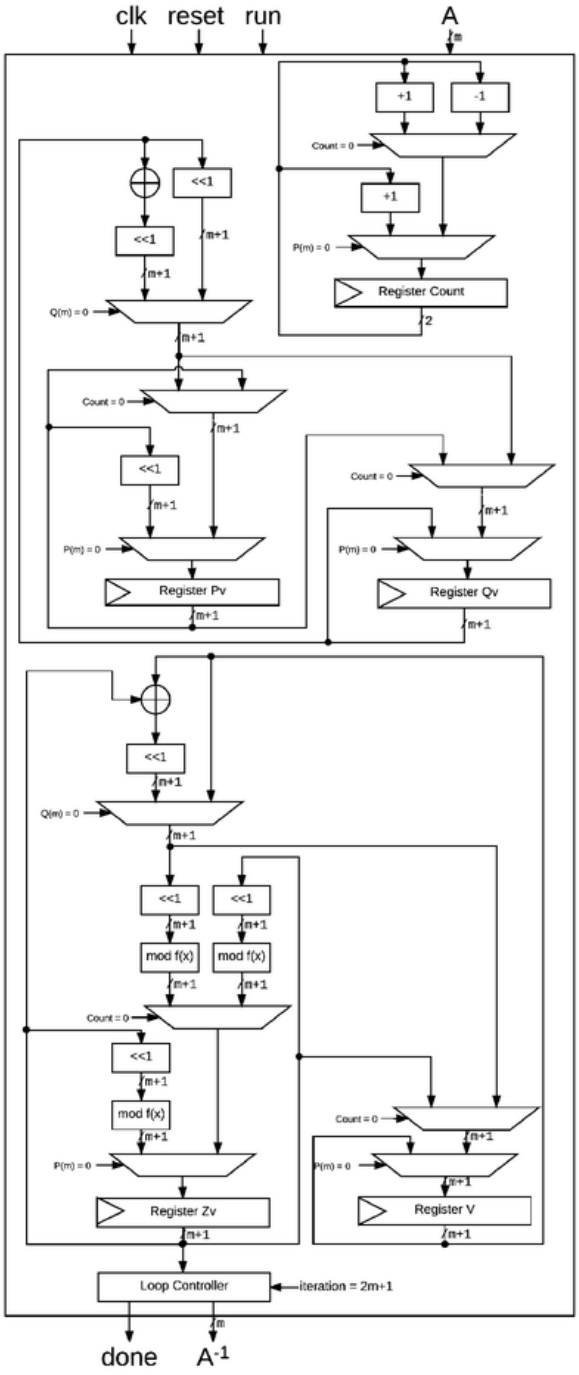


Figure 4.10: Inversion hardware architecture.

4.2 Simulation results for Galois field arithmetic

This section displays the simulation of VHDL modules created for binary field addition, multiplication and inversion. Only some test cases are presented in here, however multiple test cases were conducted in the research. The simulations were created from VHDL test benches using Xilinx ISim. The simulations for the addition modules created are shown in figure 4.11 and 4.12 for the binary field $GF(2^{233})$ and $GF(2^{288})$ respectively.

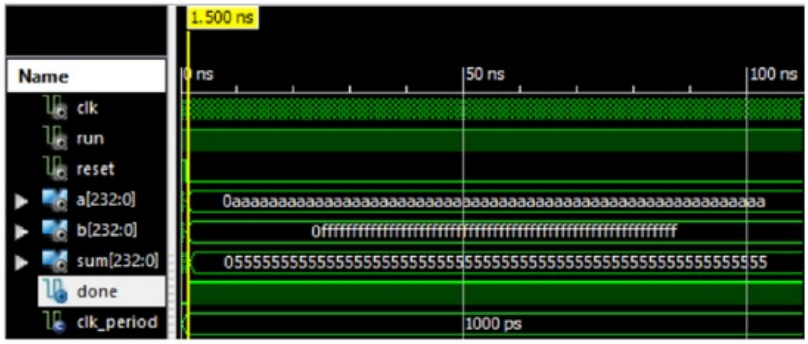


Figure 4.11: The ISim simulation for the addition module in $GF(2^{233})$.

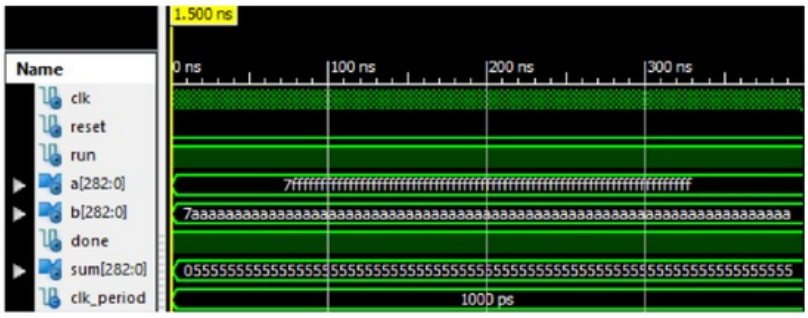


Figure 4.12: The ISim simulation for the addition module in $GF(2^{283})$.

4.2.1 Testing of traditional digit-serial multipliers

This subsection shows the waveform simulations for for the digit-serial multipliers. Table 4.3 and table 4.4 show the clock cycles for each digit-size for the traditoinal digitserial multipliers in $GF(2^{233})$ and $GF(2^{283})$ respectively. The clock cycles are reflected with the simulation results. This is shown in figure 4.13 and figure 4.14 for the digit-sizes of 1, 2, 32, and 64 to illustrate the reduction of clock cycles.

Digit-size	Clock cycles
1	233
2	117
4	58
8	30
16	15
32	8
64	4

Table 4.3: Clock cycles for 233-bit traditional digit-serial multipliers.

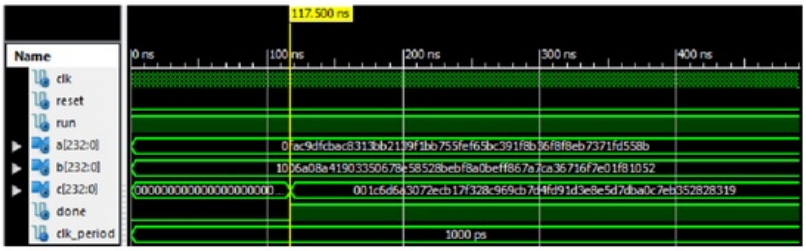
Digit-size	Clock cycles
1	283
2	142
4	71
8	36
16	18
32	9
64	5

Table 4.4: Clock cycles for 283-bit traditional digit-serial multipliers.

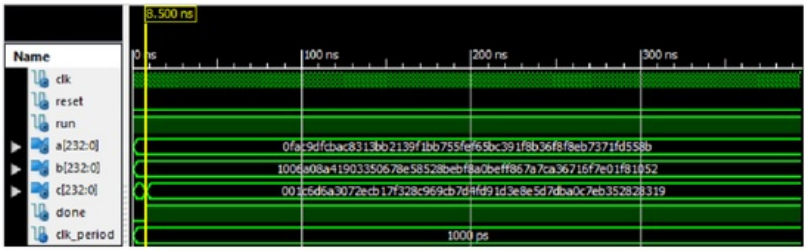
4.2.2 Testing of binary field inversion

Testing of the inversion module was performed by first obtaining a result of inversion as shown in figure 4.15 for $GF(2^{233})$ and in figure 4.17 for $GF(2^{283})$. To check that the inversion module was implemented correctly, it is followed by multiplication of this inverse and its original input. The expected output is equal to 1 $mod\ f(x)$ as shown in figure 4.16 and figure 4.18.

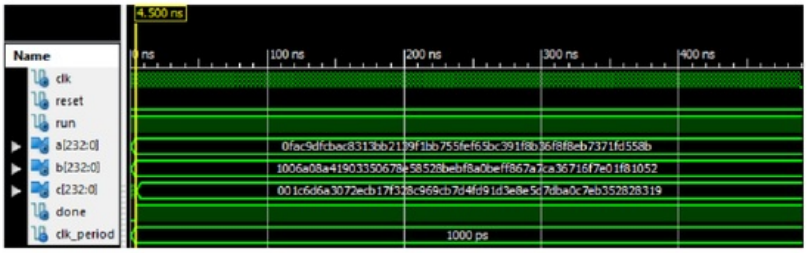
(a) $d = 1$.



(b) $d = 2$.



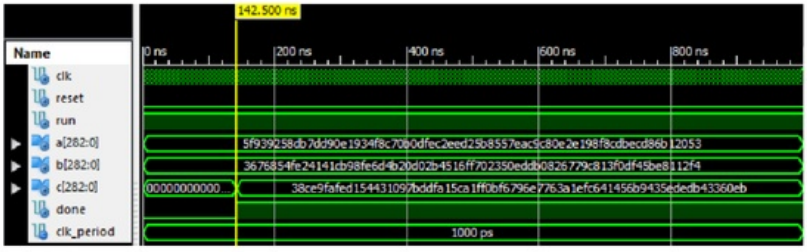
(c) $d = 32$.



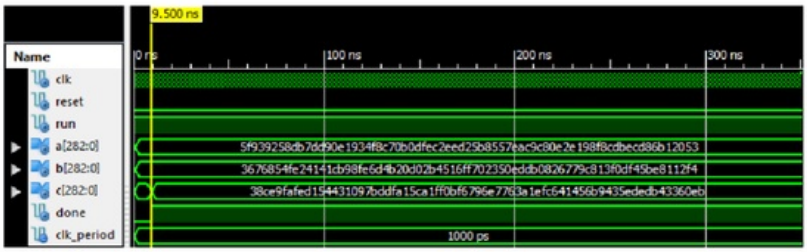
(d) $d = 64$.

Figure 4.13: The simulations for traditional digit-serial multiplication with $d = 1$, 2, 32, and 64 in $GF(2^{233})$.

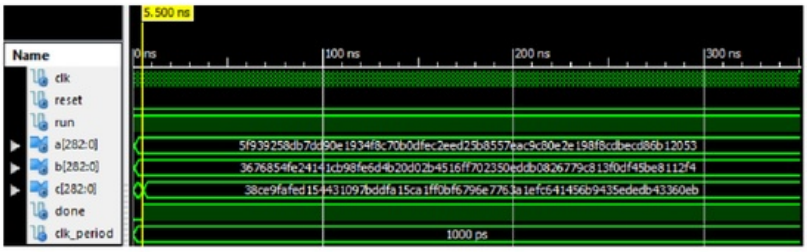
(a) $d = 1$.



(b) $d = 2$.



(c) $d = 32$.



(d) $d = 64$.

Figure 4.14: The ISim simulations for the traditional digit-serial multiplication from top to bottom with $d = 1$, 2, 4, 8, 16 and 32 in in $GF(2^{283})$.

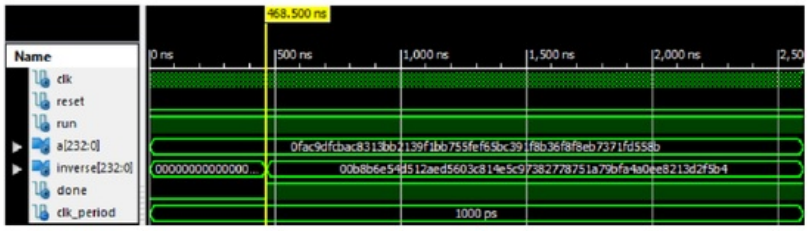


Figure 4.15: The simulation for inversion in $GF(2^{233})$.

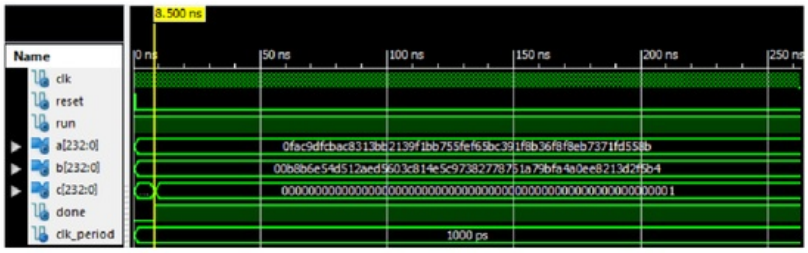


Figure 4.16: The simulation result for multiplication of a binary polynomial and its inverse in $GF(2^{233})$.

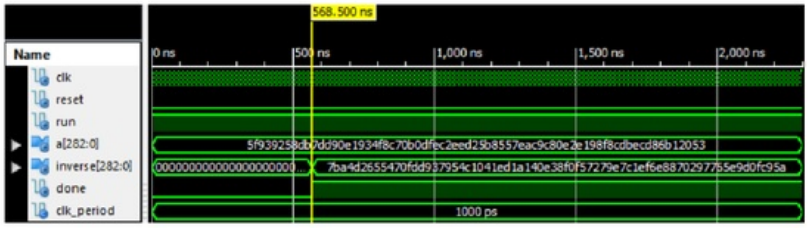


Figure 4.17: The simulation for inversion in $GF(2^{283})$.

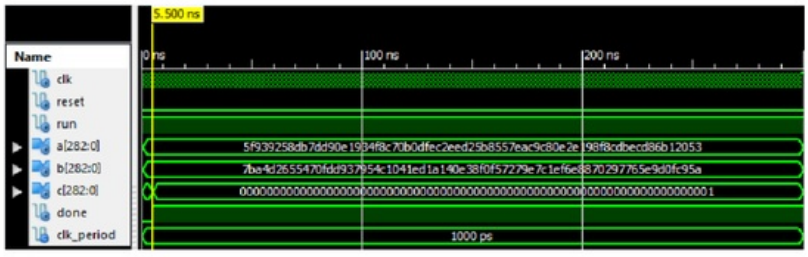


Figure 4.18: The simulation for multiplication of a binary polynomial and its inverse in $GF(2^{283})$.

Chapter 5

Hardware for ECSM over $GF(2^m)$

This chapter presents the design, implementation and testing of ECPD, ECPA and ECSM using an affine coordinate system as well as a Jacobian coordinate system. The design of ECPD and ECPA is based on the underlying finite field arithmetic shown previously in chapter 4.

Different symbols are used to represent addition, multiplication, squaring and inversion in binary fields which are shown in figure 5.1. These will make the architecture diagrams built with finite field arithmetic easier to trace with all wires being input/output signals. A close-up of connections between modules is shown in figure 5.2 where an example of computing A^4 is used. The design and implementation of all elliptic curve group operations including elliptic curve point addition, elliptic curve point addition and elliptic curve scalar multiplication will use the NIST recommended curves for binary field ECC. These NIST recommendations show random elliptic curves and Koblitz elliptic curves over $GF(2^m)$. For this project, only the NIST-recommended random elliptic curves will be used and their associated reduction polynomial $f(x)$, coefficient a, coefficient b, cofactor h and the x and y coordinates of the base point P as shown in appendix A. Of course, the Koblitz elliptic curves can also be used which will require the use of different associated parameters as recommended by NIST in appendix A.

This chapter shows the process of using the binary field arithmetic from chapter 4 for constructing ECPA, ECPD and ECSM for the initial, basic implementation of ECSM in $GF(2^{233})$ and $GF(2^{283})$. Section 5.1 and 5.2 will discuss the hardware design using affine coordinates and Jacobian coordinates respectively. Considerations for hardware implementations and compatability with FPGA devices are then discussed in section 5.3. The ISim simulation waveforms of the VHDL implementations are then presented in section 5.4 for the verification of the hardware design. These initial implementations will then be further optimised in chapter 6.

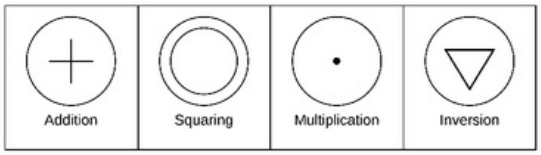


Figure 5.1: Symbols used to represent addition, multiplication, squaring and inversion modules.

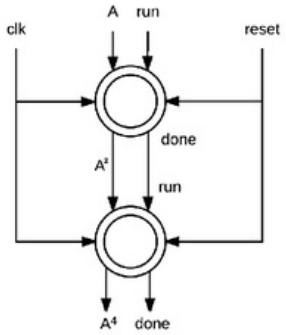


Figure 5.2: Scheduling of sequential finite field operations and the connections between each module. An example for two squaring operations is shown.

5.1 Design and Implementation of affine ECSM

This section presents the design and development of elliptic curve scalar multiplication using the affine coordinate system. Computing affine point addition, point doubling and point multiplication involves the use of coordinate points with two values such as $P(x, y)$.

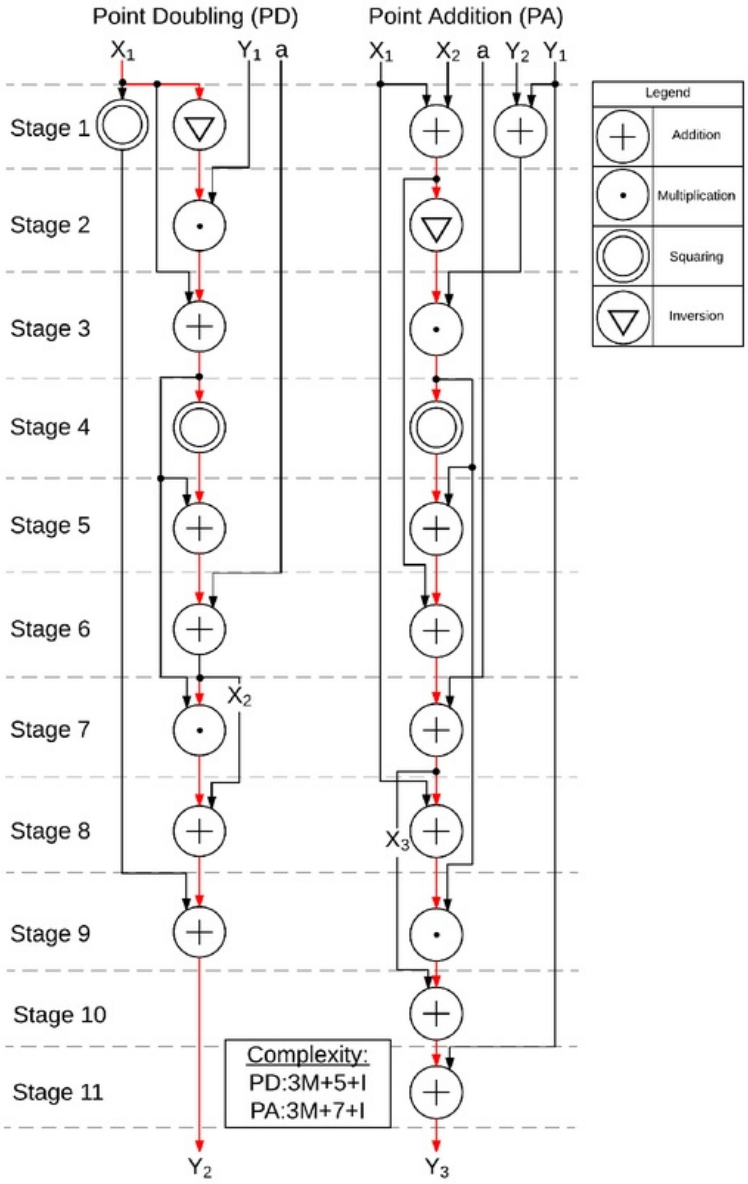


Figure 5.3: Hardware architecture used for affine point doubling (left) and affine point addition (right). The critical path for each is shown by the red arrow. The complexity shown is in terms of clock cycles where M is the clock cycles to perform a single multiplication and I is the clock cycles to perform a single inversion.

5.1.1 Affine Elliptic Curve Point Doubling

Point doubling using affine coordinates is based on equation 2.3. The point $P(x, y)$ is taken as an input where addition, multiplication and inversion modules have been arranged so that the output $2P(x, y) = (x_2, y_2)$ is produced which satisfies this equation. The hardware architecture for the construction of affine ECPD is shown in figure 5.4. Whenever a module such as multiplication or addition requires more than one input, the *done* signal of the module in the previous stage(s) with the longest total delay time is used as the *run* input of the module. This is to ensure that at the time that this module begins its computation, that the correct data inputs are available from all the previous binary field operations. This also allows for any data dependencies between modules to be accounted for. The implementation of affine ECPD requires 5 additions (A), 4 multiplications (M) and 1 inversion (I) and has the critical delay of I+3M+5 clock cyles, where M is the number of clocks to compute one multiplication and I is the clock cycles for a single inversion.

5.1.2 Affine Elliptic Curve Point Addition

Affine ECPA was implemented in a similar way based on equation 2.2. Given two inputs points $P(x_1, y_1)$ and $Q(x_2, y_2)$ the architecture shown in figure 5.4 will produce the output $R(x_3, y_3) = P + Q$. This implementation of affine ECPA requires the use of 8 addition modules, 3 multiplication modules and 1 inversion module. Affine ECPA has the critical delay of $7+3M+$I clock cycles, where M is the clock cycles taken to complete a single multiplication, and I is the clock cycles required for a single inversion.

5.1.3 Affine Elliptic Curve Point Multiplication

Affine ECSM was then implemented using affine ECPD and affine ECPA based on the right-to-left binary method for point multiplication as shown in algorithm 5. A detailed flow chart representing the VHDL implementation is shown in figure 5.5. The hardware block diagram is also shown in figure 5.4.

This proposed design uses 2 registers for the point $P(x, y)$ and it's coordinates, 2 registers for the point $Q(x, y)$ and it's coordinates, a register for the counter to iterate through each bit of the key, and a register for the current state (currentST). The three states used for this implementation is for initialisation (INI), execute-point-addition (EPA) and execute-point-doubling (EPD). The inialisation state resets the counter and temporary values. It also assigns $Q(x, y)$ with the point at infinity for affine coordinates as $(0, 0)$ and assigns $P(x, y)$ with the coordinates of the base point as defined by NIST in appendix A. The algorithm computes the final result by iterating through each bit of the key and doubling point P for every bit. The points P and Q are added together and stored in register $Q(x, y)$ every time the current bit of the key is a '1'. After m iterations, the final output result is the value of register $Q(x, y)$.

The EPA state is used to perform ECPA in affine coordinates. The affine point addition module is given the input points from register $P(x, y)$ and $Q(x, y)$. If the current

$key(count)$ ='1', then register $Q(x, y)$ is updated with the addition of these two points. The new value of the $Q(x, y)$ registers will not always be the outputs from the affine point addition module. The next value of this register will depend on the previous P and Q values. This is determined through the use of a comparison and the selection of possible outputs. This is summarised as follows where the condition is on the left, and the new value of register $Q(x, y)$ is shown on the right column of the table. These conditions are required because the result of the ECPA module will return the point $(0, 0)$ for these conditions which is not correct. In affine coordinates, Q = -P when $Q(x_1, y_1) = P(x_1, y_1 + x_1)$.

Condition	Final result for ECPA
$Q = \infty$	P
$P = \infty$	Q
$P = Q$	Result of the ECPD module
$Q = -P$ or $P = -Q$	Point at infinity ∞
else	Result of the ECPA module

Table 5.1: Summary of conditions and their corresponding results when performing point addition where Q and P are points.

The state then changes to *EPD* once a *done* signal has been received from the affine point addition module or that point addition was not required at all since $key(count) = 0$. A *PA_done2* signal has also been used to indicate that the affine point addition module is not needed since the next value of $P(x, y)$ can be pre-determined from values defined in table 5.1 in one clock cycle. The state *EPD* then assigns the affine point doubling module with the input point from register $P(x, y)$. The resulting outputs from this module is then assigned as the next values for register $P(x, y)$ once the *done* signal of the affine point addition module is indicated. The counter is also incremented by one and the state is then changed to *EPA* when the *done* signal is '1'. The last iteration is when the counter has the value of $m - 1$ and the final result for kP is achieved in register $Q(x, y)$, where k is the key and P is the base point.

This implementation performs ECPA and ECPD sequentially and will have a variable computation time. The worst-case computation time for this implementation is $m*(PD+2)+m*(PA+2)+1$ clock cycles, where PD and PA is the time taken to compute point doubling and point addition respectively and m is the bit-size of the binary field. This is calculated from performing ECPA m times, ECPD m times, and having one clock for the initialisation of registers. An additional 2 clock cycles is also required for each ECPA and ECPD module for assigning inputs to the modules and saving the results in a register. This worst-case computation time is the time taken to perform kP where all the bits of k are '1'. The average computation time to perform ECSM is taken assuming a random key having half '1' bits and half '0' bits to have a computation time to complete ECSM as $m * (PD + 2) + (m/2) * (PA + 2) + 1$ clock cycles.

Algorithm 5 Right-to-left binary method for Point Multiplication

Input: $k = (k_{m-1}, ..., k_1, k_0)_2, P \in E$.
Output: kP.

1: $Q \leftarrow \infty$.
2: **for** $i = 0$ to $t - 1$ **do**
3: **if** $k_i = 1$ **then**
4: $Q \leftarrow Q + P$.
5: **end if**
6: $P \leftarrow 2P$.
7: **end for**
8: **return** Q.

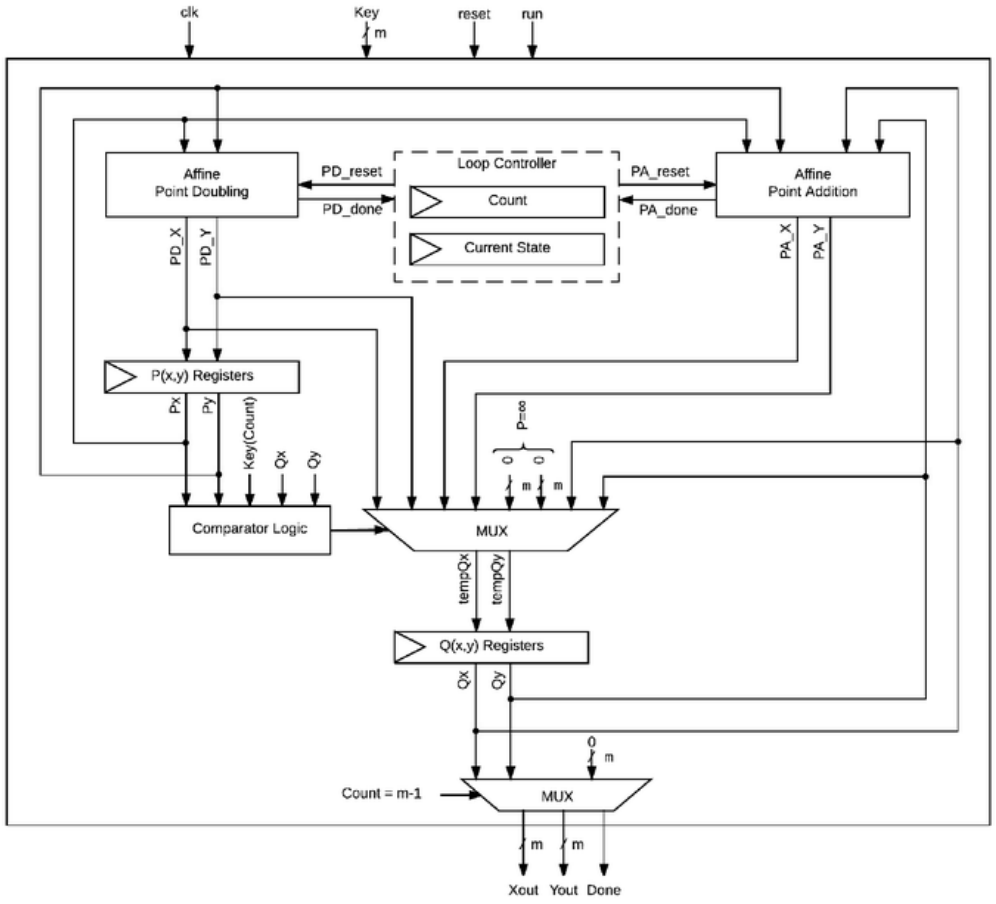


Figure 5.4: Hardware block diagram used for the implementation of sequential affine ECSM.

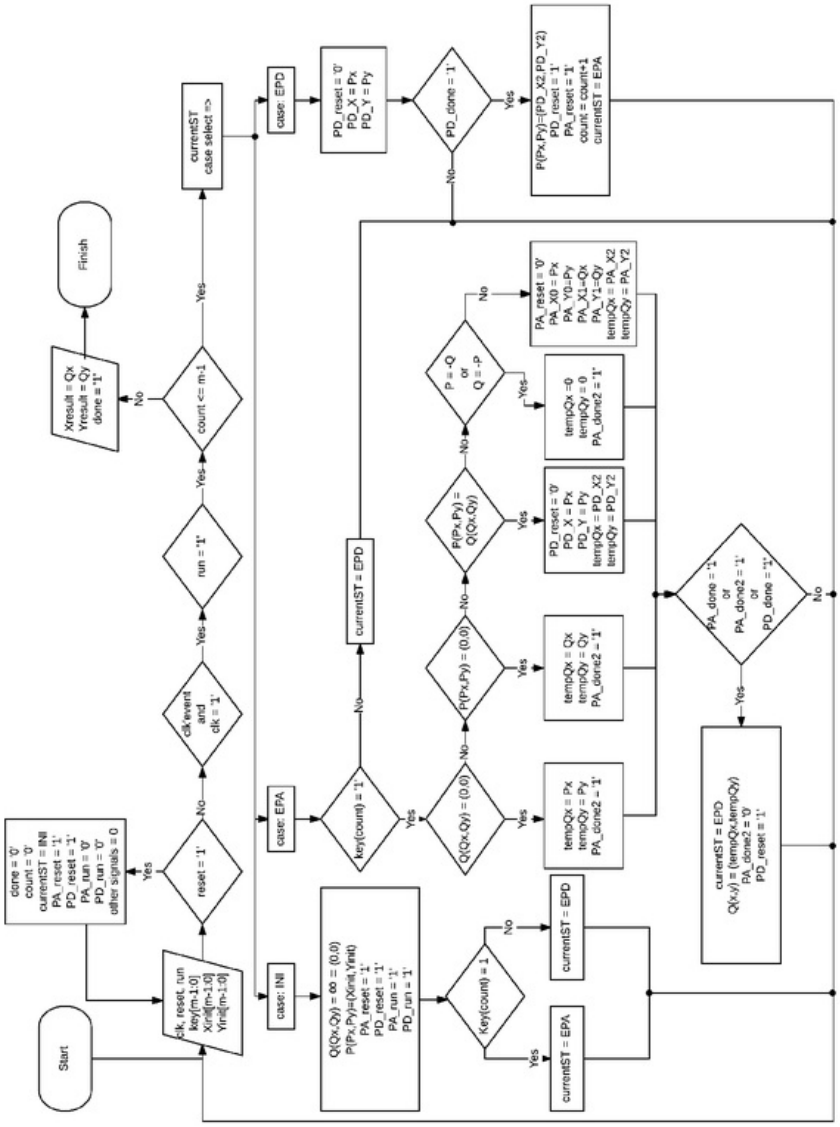


Figure 5.5: Flowchart representing the hardware design of VHDL for the affine controller in $GF(2^m)$.

5.2 Design and implementation of Jacobian ECSM

This section describes the hardware design for elliptic curve scalar multiplication using Jacobian coordinates. The completed hardware modules for Galois field arithmetic discussed in section 4 will be used to construct Jacobian ECPD and Jacobian ECPA. The design and implementation of ECSM in Jacobian coordinates follows closely with the approach used for affine coordinates. However, Jacobian coordinates involve computations with points with 3 coordinate values such as $P(x, y, z)$. Upon completion of ECSM using Jacobian coordinates, conversion between affine coordinates and Jacobian coordinates is then required as shown in equation 2.4. This section will first present the design and development of Jacobian ECPD in section 5.2.1 followed by Jacobian ECPA in section 5.2.2. A controller was then designed to perform ECSM using Jacobian coordinates in section 5.2.3. Finally a A top-level module consisting of this controller and a conversion module will then be used for the complete elliptic curve scalar multiplication with a final output in affine coordinates as shown in section 5.2.3.

5.2.1 Jacobian Elliptic Curve Point Doubling

Jacobian ECPD uses the underlying Galois field arithmetic to satisfy equation 2.7. This equation was used to design the hardware architecture as shown in figure 5.6. Given the input point $P(X, Y, Z)$, the output will be $2P(X, Y, Z)$. The Jacobian point doubling architecture requires 10 multiplication modules and 4 addition modules. The critical path for point doubling in Jacobian coordinates is the delay of 2 additions and 5 multiplication binary field operations. This design can also be optimised for the NIST recommended random curves where the coefficient $b = 1$ will allow the removal of one multiplication operation. This will allow for the optimised delay of 4M+2 clock cycles, where M is the clock cycles required to complete one multiplication.

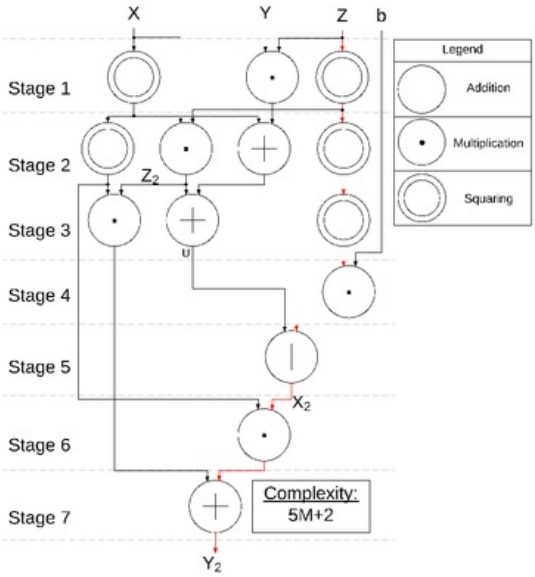


Figure 5.6: Hardware design of ECPD in Jacobian coordinates. The critical path is shown by red wiring. The complexity given is in terms of clock cycles where M is the clock cycles required for a single multiplication.

5.2.2 Jacobian Elliptic Curve Point Addition

Jacobian ECPA was also implemented based on equation 2.7. The Jacobian ECPA hardware architecture designed requires 20 multiplication modules and 7 addition modules as shown in figure 5.7. The critical path to perform ECPA is the delay of 7 multiplications and 3 additions. This design can be further optimised depending on the elliptic curve and the coefficients used. The NIST recommended random curves have the coefficient $a = 1$ which will reduce the delay of ECPA by one multiplication. This allows for an optimised delay for Jacobian elliptic curve point addition of 6M+5 clock cycles, where M is the time taken to complete one multiplication.

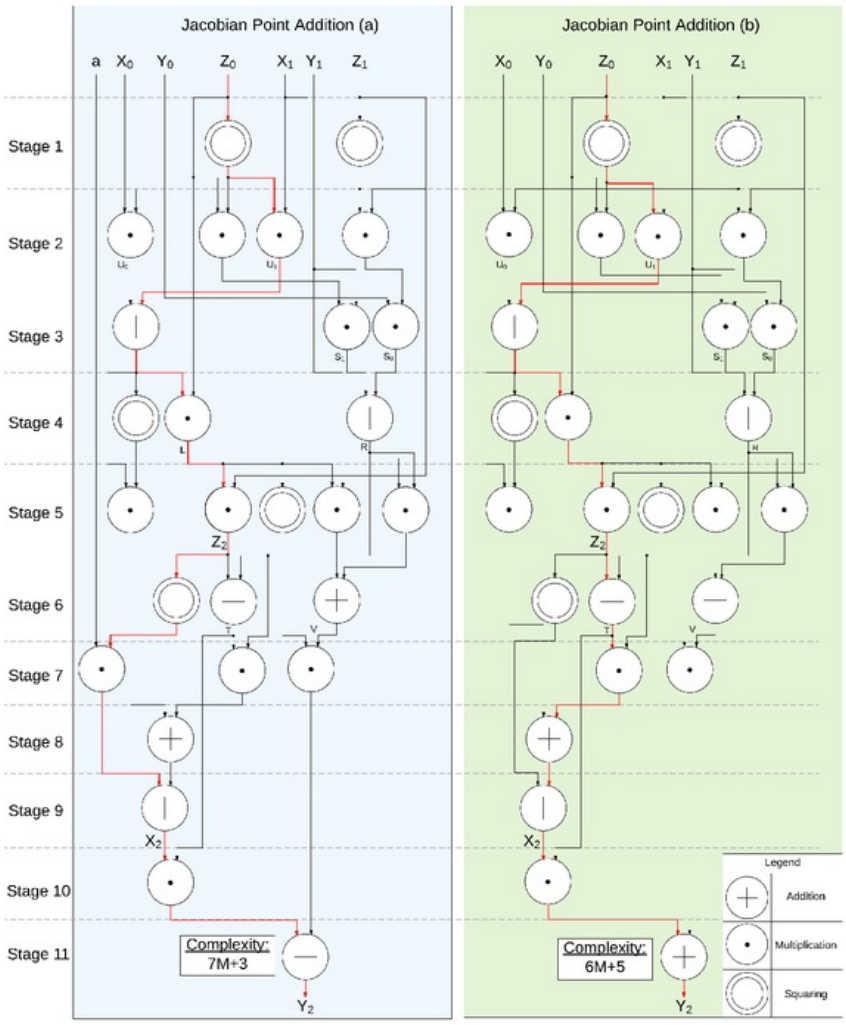


Figure 5.7: Hardware design of ECPA in Jacobian coordinates. The critical path is shown in red. The complexity given is in terms of clock cycles where M is the number of clock cycles to perform one multiplication.

5.2.3 Jacobian Elliptic Curve Point Multiplication

The next level of operation to be implemented is Jacobian ECPM. There are multiple methods for the implementation of ECSM as outlined in section 2.3. The algorithm used for the implementation of ECSM was the same as that used for affine coordinates using

the double-and-add method shown in algorithm 5. This is the simplest algorithm to implement ECSM which performs point doubling m times and point addition for every '1' bit of the key. This method performs point addition and point doubling sequentially.

Minor changes to the hardware architecture was required from the implementation of affine coordinates. Firstly, 6 registers were needed to represent the x, y, and z coordinates for the point P and Q. The second change that was needed is for the conditions outlined in table 5.1. A point at infinity for Jacobian coordinates is the point $(1, 1, 0)$ and the negative of a point $A(x_1, y_1, z_1)$ is given as $B(x_1, x_1 + y_1, z_1)$. The design of ECSM in hardware needs a controller component to perform ECSM and a conversion component for the conversion to affine coordinates.

Jacobian Controller

The design and implementation of Jacobian point addition and point doubling has been presented in the previous sections. A Jacobian controller will be needed to schedule when ECPA and ECPD is performed and the accumulation of the results of these components in registers untill a final output is computed. The design of the Jacobian controller and its connections are shown in figure 5.8. A detailed flow chart to implement algorithm 5 is shown in figure 5.9 for the design in VHDL.

This design uses the same states to control what function is currently being performed during ECSM. Just like for affine coordinates, the states are INI, EPA, and EPD as shown in figure 5.10. The state INI is used for the initialisation of the internal registers. The point $Q(Qx, Qy, Qz)$ is initialised to be a point at ∞ , in Jacobian coordinates this is $Q(1, 1, 0)$ [7]. The point $Q(x, y, z)$ is used for adding and accumulating the points indicated by the current bit iteration of the key while $P(Px, Py, Pz)$ doubles for every iteration.

The next state is EPA when $key(count)$ ='1' otherwise it is EPD. The EPA state checks if the current bit of $key(count)$ of the iteration is a '1' and performs ECPA with the points Q and P. Point Addition is performed by assigning the Jacobian ECPA component inputs with P and Q coordinates and setting the 'reset' to 0 to allow the component to function. The controller then waits for the *done* signal of the Jacobian point addition module to update the registers representing the point $Q(x, y, z)$ depending on the previous conditions and values of the points P and Q. The state is then changed to EPD which performs ECPD the same way. The input of the ECPD component are set with point P coordinates while the *reset* of the component is set to 0 to allow computation of $2P$. Once the *done* signal of the point doubling component is set, the registers representing the coordinates of $P(x, y, z)$ are updated with the new output of the ECPD component. When the *done* signal of the Jacobian ECPD module is recieved, the ECPA and ECPD *reset* signals are then set to '1' and the counter is incremented for the next bit of the key. The state is then changed to EPA to perform ECPA then EPD.

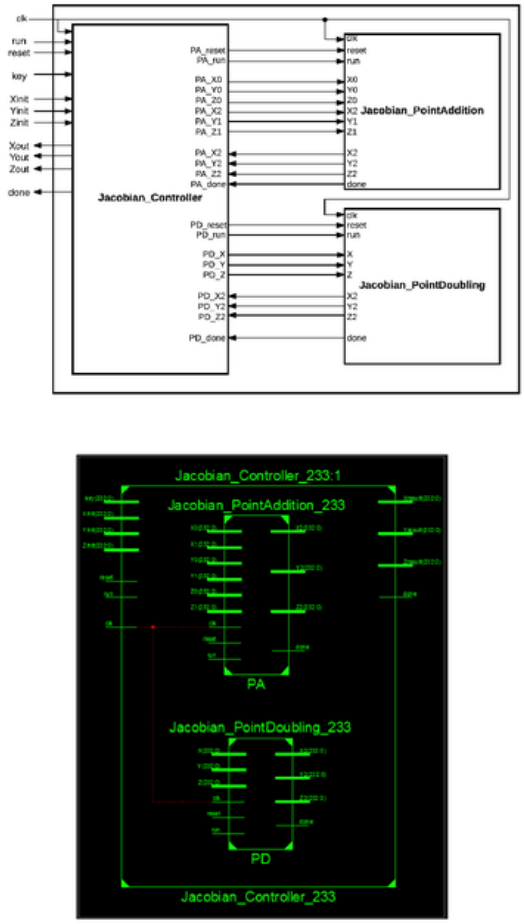


Figure 5.8: The high-level architecture of the Jacobian controller for elliptic curve scalar multiplication.

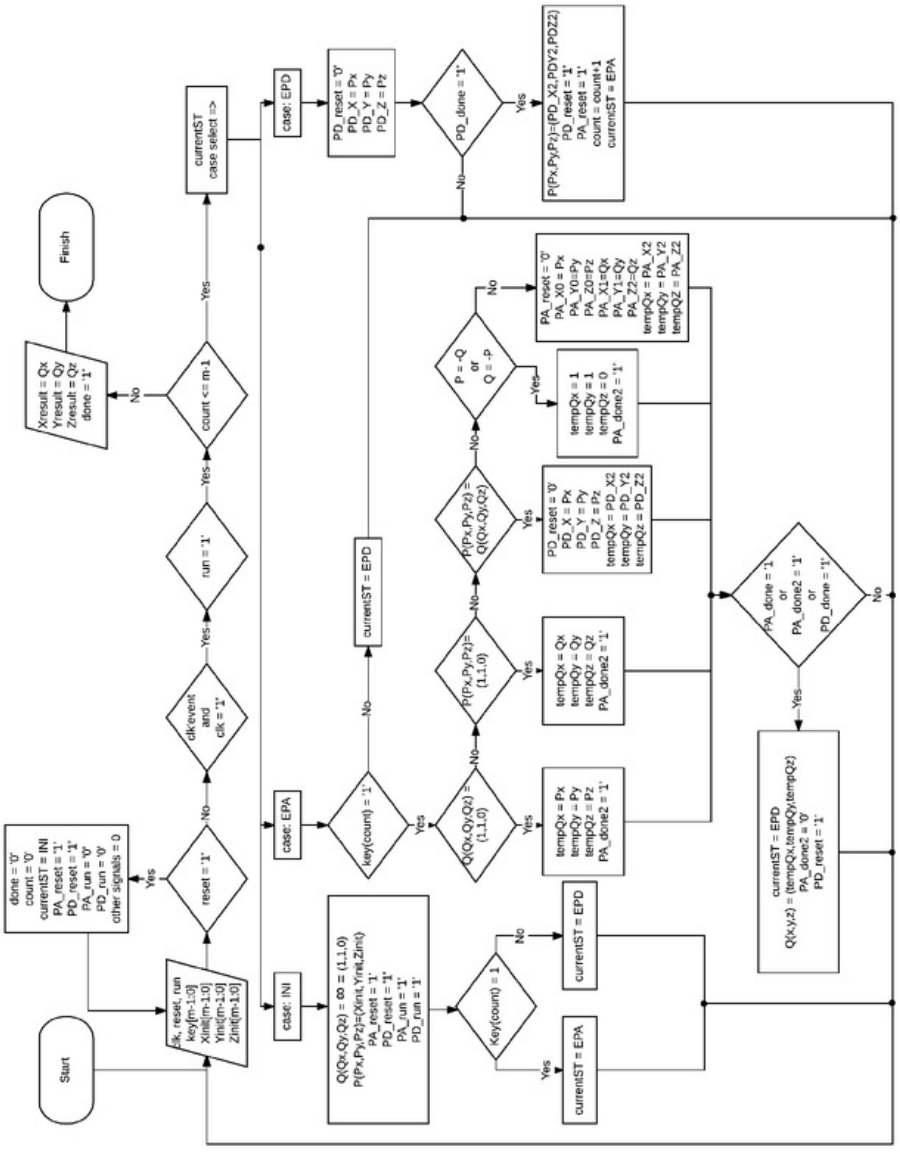


Figure 5.9: Flowchart representing the hardware design of ECSM in Jacobian coordinates.

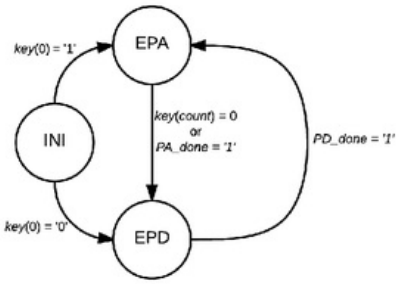


Figure 5.10: State diagram used in the Jacobian controller.

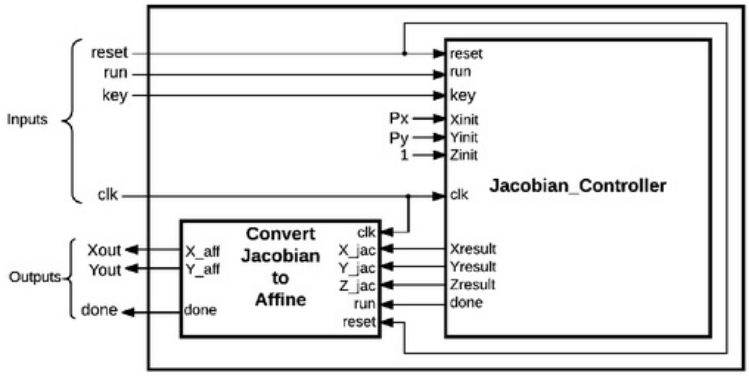


Figure 5.11: The top level block diagram showing the final conversion for Jacobian to affine coordinates once point multiplication has completed.

Jacobian Coordinate Conversion

The Jacobian controller currently takes a key as an input and computes the output point $P(x, y, z)$ in Jacobian coordinates so conversion to affine coordinates is required. The high level diagram showing the interconnections for the final conversion from Jacobian coordinates to affine coordinates is shown in figure 5.11. The internal Galois field operations to perform the conversion are shown in figure 5.12, where the critical path is shown by red arrows. The construction of the conversion component requires the use of 5 multiplication modules and 1 inversion module. The delay of this conversion component will be the delay of 3 multiplications and 1 inversion. The design of this hardware architecure is based on equation 2.5. Given the input point $P(P_x, P_y, P_z)$ in Jacobian

coordinates, the conversion to affine coordinates produces the output point $Q(Q_x, Q_y)$, where $Q_y = P_y/(P_z^3)$ and $Q_x = (P_x)/(P_z^2) = (P_xP_z)/(P_z^3)$.

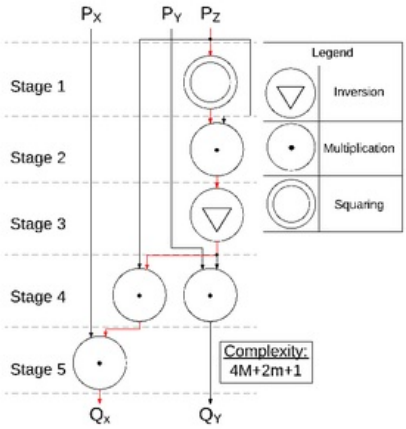


Figure 5.12: The top level architecture showing the conversion of Jacobian coordinates to affine coordinates using Galois field operations.

5.3 FPGA implementation of ECSM

The hardware design for the computation of affine ECSM and Jacobian ECSM has been presented in section 5.1 and 5.2. Both of these designs will require input pins for the *run, reset,* and *key* signals and output pins for the *done* signal as well as the X and Y coordinates of the ECSM output. For the key, X, and Y the number of pins for each of these input/output (I/O) signals will be m for $GF(2m)$. This results in a total of $3m + 3$ I/O pins with a total of 702 pins for the ECSM in $GF(2^{233})$ and 852 I/O for ECSM in $GF(2^{233})$.

The number of I/O pins vary between different FPGA devices where the number of pins may be too limited to support the design. In order to make the hardware design used in this research to be compatible with any FPGA device, serial inputs will need be used for the key, and serial outputs for the resulting coordinates of ECSM. This will reduce the number of I/O pins to just 7. The hardware architecture is shown in figure 5.13 where a serial-in-parrallel-out (SIPO) module is used to accumulate the input for the ECSM module and a parallel-in-serial-out (PISO) module is used to output the final X and Y coordinate sequentially. The SIPO module has a delay of m+1 clock cycles and the PISO module has the delay of m+2 clock cycles.

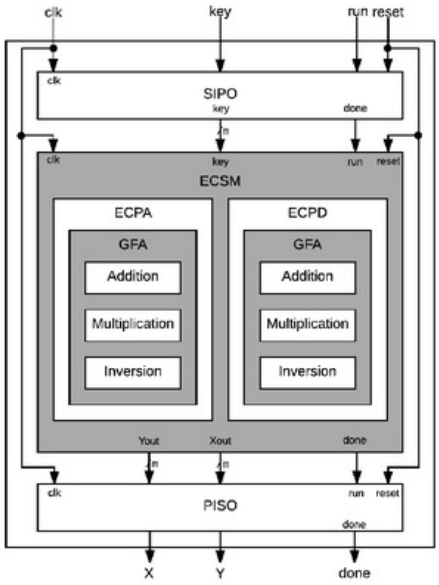


Figure 5.13: The top level architecture with PISO and SIPO for the implementation on an FPGA device.

5.4 Simulations for ECSM over binary fields

This section shows the tests use to verify the hardware implementation of ECSM using affine and Jacobian coordinates. It does not display all the tests used, but important test cases will be discussed here.

5.4.1 Testing of ECSM over $GF(2^{233})$

. Affine point addition and affine point doubling were tested through the use of the individual computations using the previously implemented Galois field modules. The test for affine point doubling is shown in figure 5.14 where the base point P defined by NIST in appendix A was used to compute $2P$. The affine point addition module was then used to compute $3P = 2P + 1P$ as shown in figure 5.15. The result of ECSM in affine coordinates using was then obtained when the key had a value of 3 as shown in figure 5.16 which produces the expected outputs as confirmed by the affine point addition test.

Small key sizes were easily tested through the use of the affine ECPA and affine ECPD tests. However, special test cases were used to have confidence in the correct operation of ECSM architectures as shown in figure 5.17. The order of a curve is shown as n in

appendix A. When the key $k = n$,the expected result is the point at infinity which is $(0, 0)$ for affine coordinates. When $k = n + 1$, the expected result is $\infty + P{=}1P$ which is the base point of an elliptic curve. Another case is when $k = n-1 = \infty-1P = -P$. Negative P is obtained when given the point $P(x_1, y_1)$, $-P = (x_1, x_1+y_1)$. The serial output of the FPGA implementation was then checked as shown in figure 5.21 which shows the output given when $k = 3$.

The tests for ECSM using Jacobian coordinates were conducted in a similar way to affine coordinates. Simulations for Jacobian EPCD and ECPA are shown in figure 5.18 and figure 5.19 respectively. This was then used to check the point for $3P$ using the Jacobian ECSM architecture in figure 5.16. The test cases for the order of the curve were then simulated as shown in figure 5.20. These tests verify the correct implementation of Jacobian coordinates as these produce the same output point for a key as those produced by affine ECSM. Finally, the tests for ECSM in the binary field $GF(2^{283})$ will be briefly shown here. The testing methods for the verification of the 283-bit hardware implementation was identical to the process used to test the 233-bit implementation. The simulations for $GF(2^{283})$ are shown with the key $k = 3, n - 1, n$, and $n + 1$ for affine coordinates in figure 5.23 and for Jacobian coordinates in figure 5.24.

Figure 5.14: The simulation for $GF(2^{233})$ ECPD using affine coordinates.

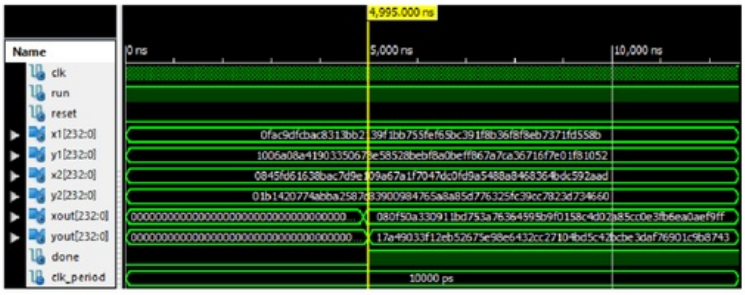


Figure 5.15: The simulation for $GF(2^{233})$ ECPA using affine coordinates.

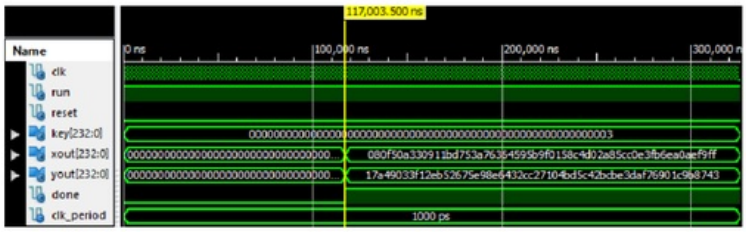
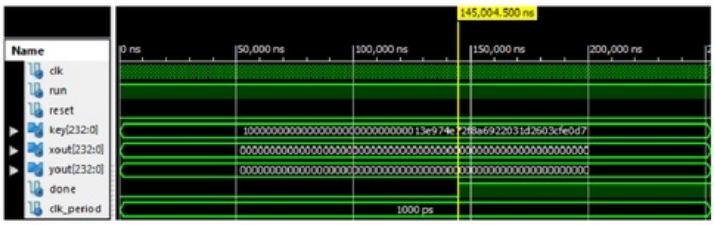


Figure 5.16: The simulation for ECSM using affine coordinates with $k = 3$ in in $GF(2^{233})$.

(a) $k = n - 1$.



(b) $k = n$.

(c) $k = n + 1$.

Figure 5.17: The ISim simulations for ECSM using affine coordinates with $k = n - 1$, n, and $n + 1$ in $GF(2^{233})$.

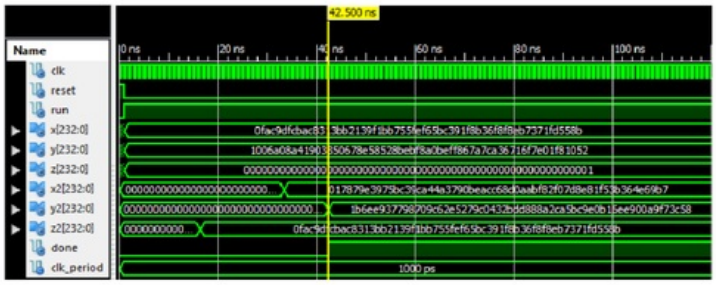


Figure 5.18: Simulation results for Jacobian point doubling in $GF(2^{233})$.

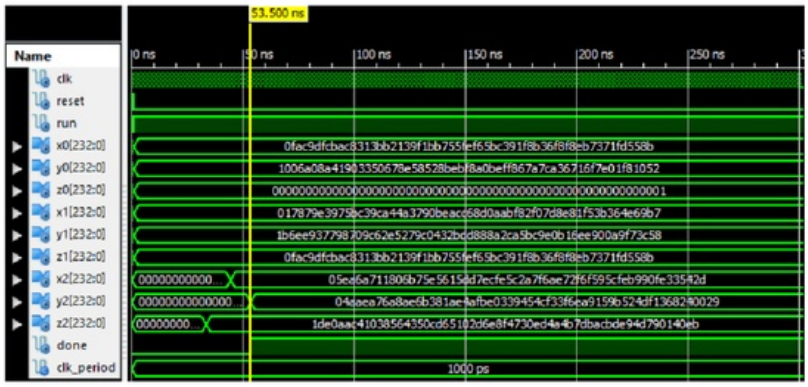


Figure 5.19: Simulation results for Jacobian point addition in $GF(2^{233})$.

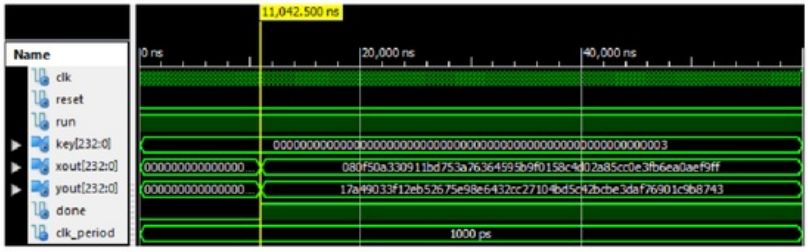
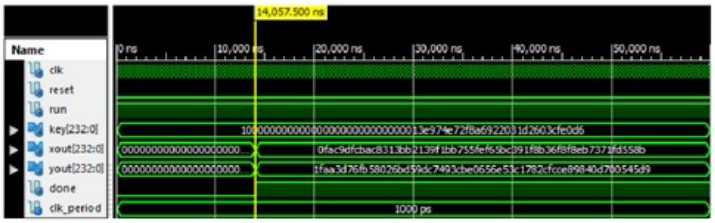
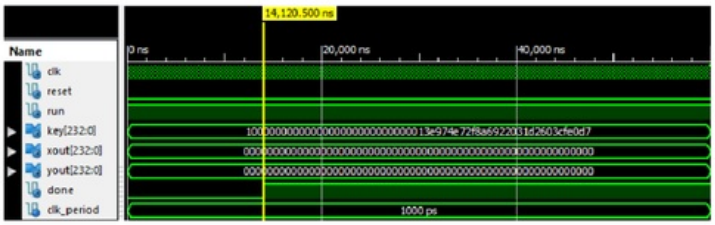


Figure 5.20: The simulations for ECSM using Jacobian coordinates with $k = 3$ in in $GF(2^{233})$.

Figure 5.21: Simulation results for the serial output of Jacobian ECSM with k=3 in $GF(2^{233})$.



(a) $k = n - 1$.

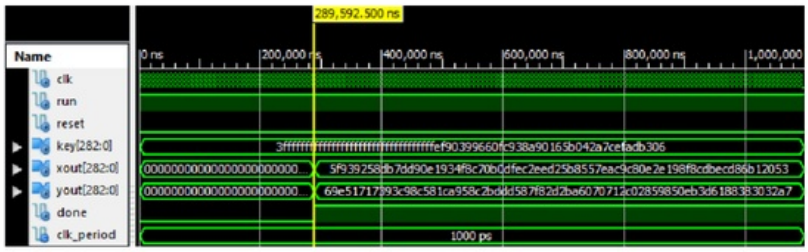


(b) $k = n$.

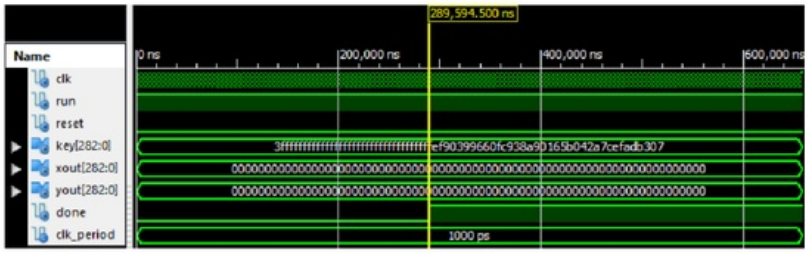
(c) $k = n + 1$.

Figure 5.22: The simulations for ECSM using Jacobian coordinates with $k = n - 1$, n, and $n + 1$ in in $GF(2^{233})$.

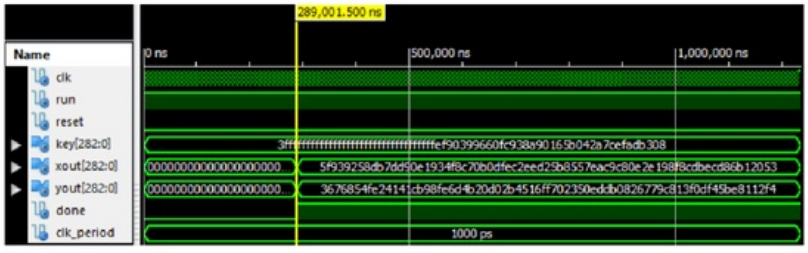
(a) $k = 3$.



(b) $k = n - 1$.



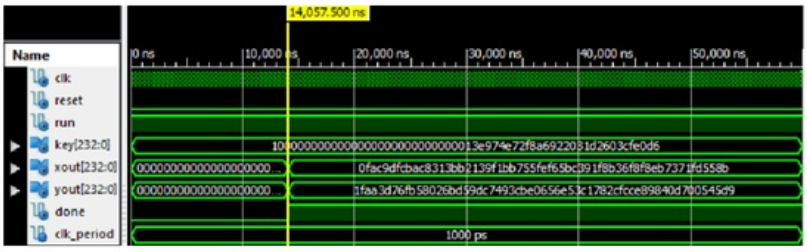
(c) $k = n$.



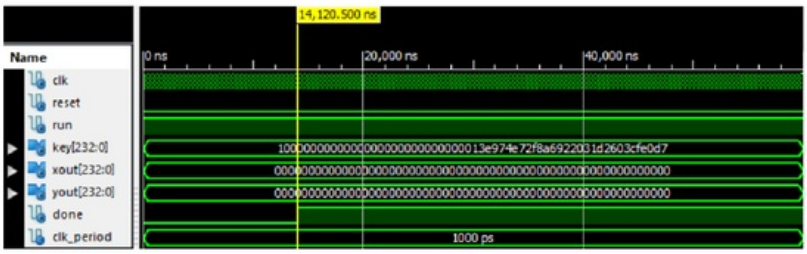
(d) $k = n + 1$.

Figure 5.23: The simulations for affine ECSM with $k = n - 1$,n, and $n + 1$ in $GF(2^{283})$.

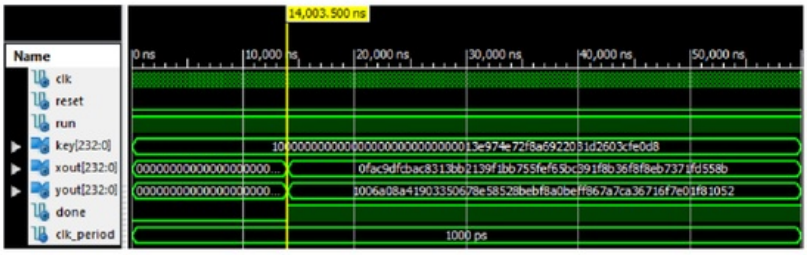
(a) $k = 3$.



(b) $k = n - 1$.



(c) $k = n$.



(d) $k = n + 1$.

Figure 5.24: The simulations for Jacobian ECSM with k = n-1, n, and n+1 in in $GF(2^{283})$.

Chapter 6

Optimised ECC operations in $GF(2^m)$

Each level in the hierarchy of ECC operations can be further improved. The initial hardware implementations for ECSM using Jacobian coordinates and affine coordinates have been presented previously in chapter 5. This chapter presents methods which have been explored in this project which aimed to optimise the overall efficiency of ECSM in hardware. Optimisation of binary field multiplication is presented in section 6.1 and optimisation for elliptic curve group operations are shown in section 6.2.

6.1 Modfied Binary Field Multiplication

This section presents a modified pipe-lined digit-serial multiplier based on the hardware architecture proposed by Pan and Lee in [11]. The modified multiplication uses algorithm 6 and is able to perform multiplication in $GF(2^m)$ in $2 * \lceil \sqrt{m/d} \rceil + 1$ clock cycles.

6.1.1 Design and implementation of modified multipliers

Recall the traditional digit-serial multipliers implemented in section 4.1.2, the clock cycles required to perform binary field multiplication is defined by $q = \lceil m/d \rceil$. A pair of integers p, and k are now used such that $kp = q$ and $k = \sqrt{q}$. Zeros need to also be padded to the multiplicand B so that $q = \lceil m/d \rceil = kp$ can be satisfied. The hardware design of the pipe-lined digit-serial multiplier is shown in figure 6.1. This top level hardware design uses k processing elements (PE) to perform the inner loop (lines 6 to 9) of algorithm 6.

Algorithm 6 Modified digit-serial multiplication

Input: Binary polynomials $A = (a_{m-1}, ..., a_1, a_0)$, $B = (b_{m-1}, ..., b_1, b_0)$, reduction polynomial $f(z) = z^m + r(z)$.
Output: $C = A \cdot B \ mod \ f(x)$.

1: $\bar{C} \leftarrow 0$.
2: $B = \sum kp - 1^{i=0} B_i x^{id}$, where $B_i = \sum_{j=0}^{d-1} b_{id+j} x^j$.
3: **for** $i = 0$ to $p - 1$ **do**
4: $\quad D \leftarrow B$
5: $\quad A \leftarrow x^{kd} A \bmod f(x)$
6: $\quad$ **for** $j = 0$ to $k - 1$ **do**
7: $\quad\quad \bar{C} \leftarrow \bar{C} \oplus B_{ik+j} D$.
8: $\quad\quad D \leftarrow D \cdot x^d \bmod f(x)$.
9: $\quad$ **end for**
10: **end for**
11: $C \leftarrow \bar{C} \ mod \ f(x)$.
12: **return** C.

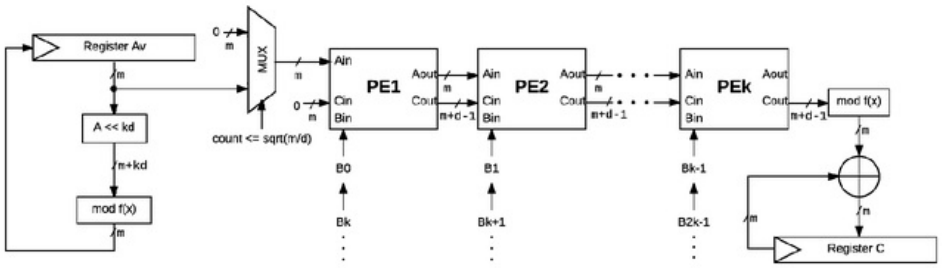


Figure 6.1: Modified digit-serial multiplier hardware architecture.

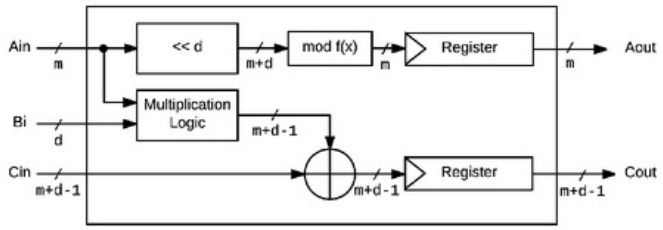


Figure 6.2: Block diagram for the processing elements used in the modified digit-serial multiplier.

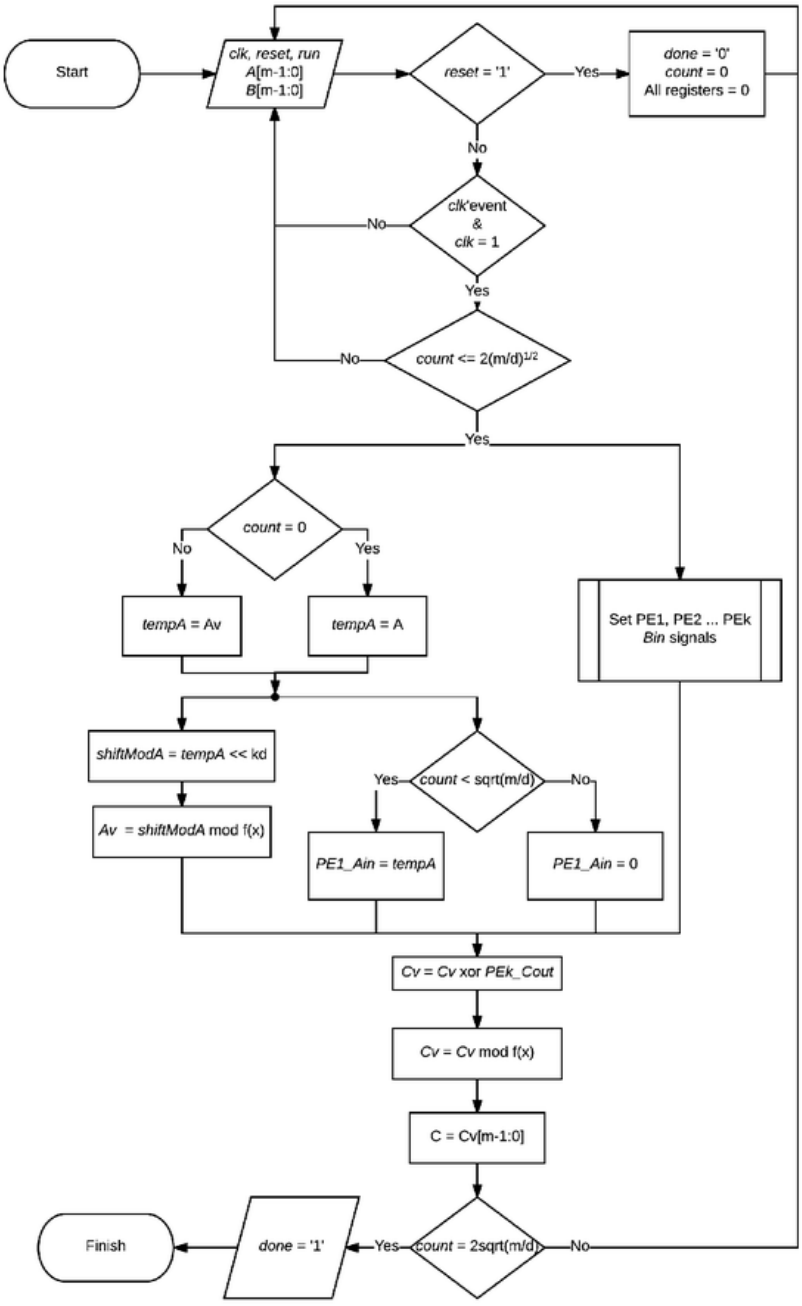


Figure 6.3: Flowchart representing the VHDL implementation of the modified digit-serial multiplier.

This modified multiplier is composed of three main parts consisting of updating register Av, the processing elements, and the reduction and accumulation of the *Cout* partial products. In each clock cycle, register Av is updated with $Ax^{kd} \ mod \ f(x)$. The first PE then has the value of register Av as an input for $\sqrt{m/d}$ clock cycles followed by zeros untill the multiplication is complete. The the first PE (PE1) also has a constant *Cin* input of 0. The remaining PEs have the *Ain* and *Cin* inputs of the previous processing element's *Aout* and *Cout* respectively. Figure 6.2 shows the internal hardware of each processing element. Given the inputs *Ain*, *Bin*, and *Cin*, the processing element computes the outputs $Aout = Ain \cdot x^d \ mod \ f(x)$ and $Cout = Cin \oplus (Ain \cdot Bin)$. Finally, register Cv is used to add the partial products from the last processing element's (PE_k) *Cout* such that $Cv = \sum_0^{2k} Cout$. The number of processing elements and clock cycles for each digit-size in $GF(2^{233})$ and $GF(2^{283})$ are shown in table 6.2.

The design for a modified digit-serial multiplier in $GF(2^{233})$ with the digit-size of 8 ($d = 8$) will be described here. The number of processing elements required is given by $k = \sqrt{\lceil m/d \rceil} = \sqrt{\lceil 233/8 \rceil} = 6$. The number of clock cycles is given by $2k = 12$ with the addition of one clock cycle for initalisation, making it a total of 13 clock cycles to complete a single multiplication. As shown in the flowchart in figure 6.3, there is a sub-process for setting the PE *Bin* signals. For this example, the scheduling of *Bin* inputs for each processing element is shown in table 6.1. The scheduling of *Ain* inputs for the first processing element is also shown.

Cycle	PE1_Ain	PE1	PE2	PE3	PE4	PE5	PE6
0	A0	0	0	0	0	0	0
1	A1	B[7:0]	0	0	0	0	0
2	A2	B[55:48]	B[15:8]	0	0	0	0
3	A3	B[103:96]	B[63:56]	B[23:16]	0	0	0
4	A4	B[151:144]	B[111:104]	B[71:64]	B[31:24]	0	0
5	A5	B[199:192]	B[159:152]	B[119:112]	B[79:72]	B[39:32]	0
6	0	B[247:240]	B[207:200]	B[167:160]	B[127:120]	B[87:80]	B[47:40]
7	0	0	B[255:240]	B[215:208]	B[175:168]	B[135:128]	B[95:88]
8	0	0	0	B[263:256]	B[223-216]	B[183:176]	B[143:136]
9	0	0	0	0	B[271:264]	B[231:224]	B[191:184]
10	0	0	0	0	0	B[279:272]	B[239:232]
11	0	0	0	0	0	0	B[287:280]
12	0	0	0	0	0	0	0

Table 6.1: The scheduling of *Bin* inputs for each processing element in $GF(2^{233})$ when d=8.

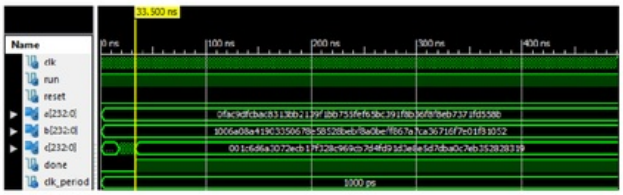
$GF(2^{233})$		
Digit-size	#PEs	Clock cycles
1	16	33
2	11	23
4	8	17
8	6	13
16	4	9
32	3	7
64	2	5

$GF(2^{283})$		
Digit-size	#PEs	Clock cycles
1	17	35
2	12	25
4	9	19
8	6	13
16	5	11
32	3	7
64	3	7

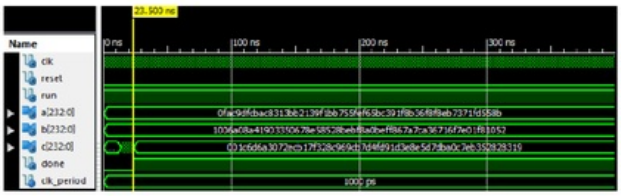
Table 6.2: The number of processing elements (PEs) and clock cycles for each digit-size implemented in $GF(2^{233})$ and $GF(2^{283}$.

6.1.2 Simulation of modified multipliers

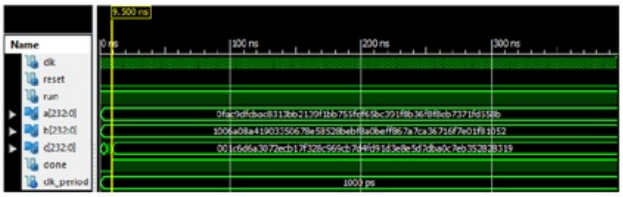
This section shows the simulations for the verification of modified digit-serial multipliers. These simulations were compared with results of those used in the traditional digit-serial multipliers but a focus here is shown on the clock cycles required to complete a single multiplication. The simulations for the modified digit-serial multipliers for $d = 1$, 2, 16, and 32 are shown in figure 6.4 for $GF(2^{233})$ while figure 6.5 shows the simulations for $GF(2^{283})$. These results reflect the calculated clock cycles as shown in table 6.2.



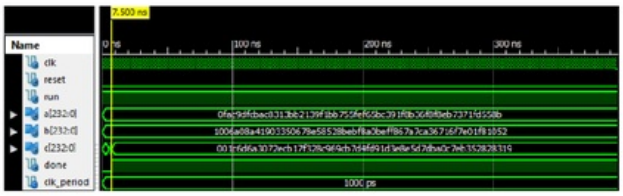
(a) $d = 1$.



(b) $d = 2$.

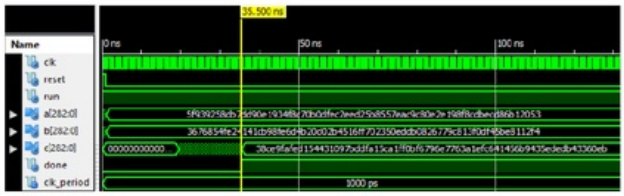


(c) $d = 16$.

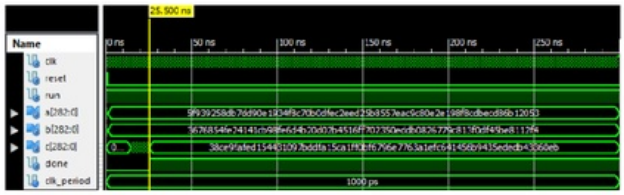


(d) $d = 32$.

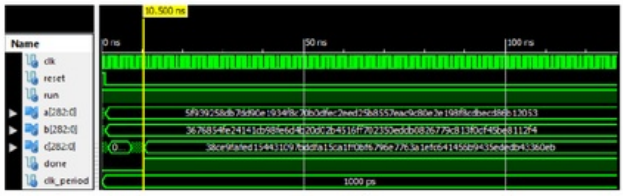
Figure 6.4: The simulations for the modified digit-serial multiplication with $d = 1$, 2, 16, and 32 in in $GF(2^{233})$.



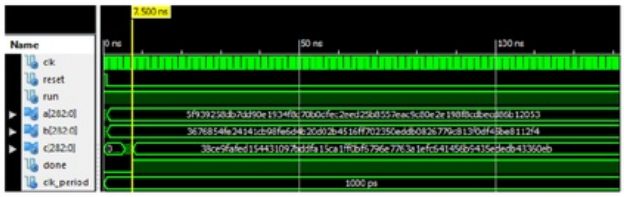
(a) $d = 1$.



(b) $d = 2$.



(c) $d = 16$.



(d) $d = 32$.

Figure 6.5: The simulations for the modified digit-serial multiplication with $d = 1$, 2, 16, and 32 in $GF(2^{283})$.

6.2 Binary field Elliptic Curve Group Operations

The previous implementation of ECSM using the double and add method in chapter 5 performs ECSM with an inconsistent computation time which is unfavourable in hardware design. This section proposes methods to improve the efficiency of this ECSM implementation in FPGA. The first improvement in section 6.2.1 modifies the sequential double and add method so that these operations are performed concurrently. The use of elliptic curve point doubling and elliptic curve point addition was then combined and optimised in section 6.2.2.

6.2.1 Concurrent ECPA and ECPD

Previously, the method for elliptic curve scalar multiplication using algorithm 5 produced a variable time which is undesirable for hardware implementations. Rather than performing point addition *then* point doubling, concurrent techniques in hardware have been used to perform elliptic curve point doubling and elliptic curve point addition simultaneously. This brings advantages such as a decrease in total time, and that ECSM can be performed with a constant computation time. The initial implementation of ECSM will have the the worst case computation time of $m * (PA + 2) + m * (PD + 2) + 1$, where PA is the clock cycles required to perform ECPA, and PD is the clock cycles required to perform ECPD. The time to execute ECSM using the concurrent double and add will now be $m * (PA + 2) + 1$ since each iteration of the key will wait for one execution of point addition regardless of the current bit of the key being a '1' or a '0' or the conditions of elliptic curve point addition.

This concurrent module is shown in figure 6.6 which uses the same hardware architecture for Jacobian point addition and point doubling as presented in figure 5.7 and figure 5.6 respectively. Given two input points P and Q, this module will compute the outputs $2P$ and $Q + P$. The done signal is associated with the last output of the internal point addition module since it has the longest critical path. The done signal will be used to indicate the completed execution of ECPA and ECPD.

This module was then used for the design of a concurrent Jacobian ECSM architecture proposed in figure 6.8. This hardware architecture produces results in Jacobian coordinates. Conversion from Jacobian to affine coordinates will be required as presented in section 5.2.3. This architecture uses a register for the count, a register for the current state (currentST), and 6 registers for the x, y and z coordinates of point P and point Q. Point P is used to double the initial input point for each iteration of the key bits while point Q is used for the accumulation of point addition for each of bit of the key that is "1". At the complete iteration of all key bits, $Q(x, y, z)$ registers contain the final result of $k * P$.

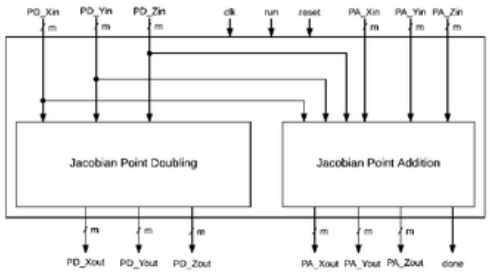


Figure 6.6: Block diagram combining Jacobian ECPD and ECPA modules.

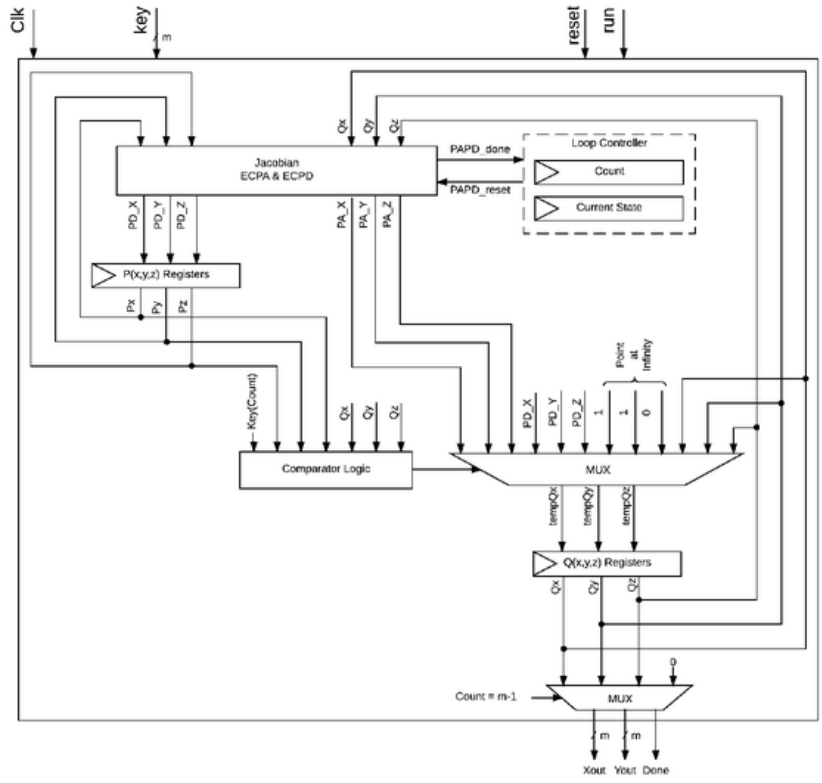


Figure 6.7: Hardware architecture for concurrent ECSM using the "double-and-add" algorithm in Jacobian coordinates.

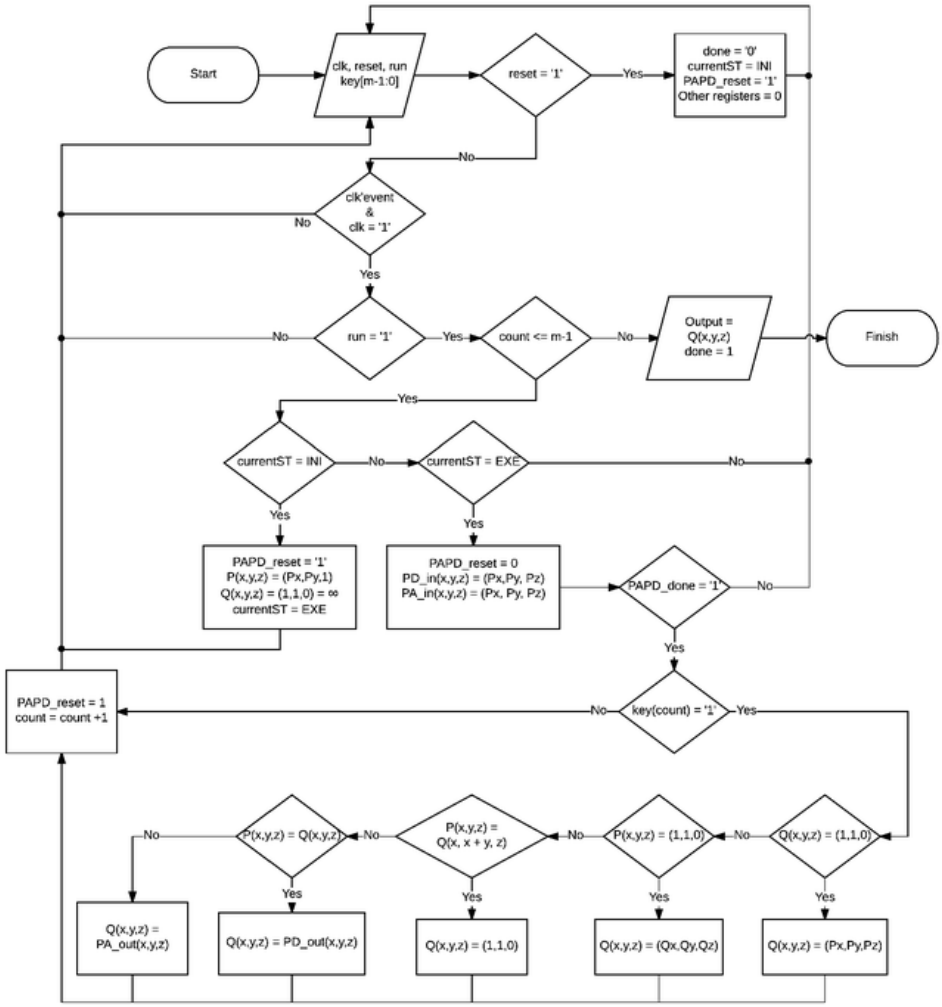


Figure 6.8: Flowchart representing the VHDL implementation of concurrent ECSM using Jacobian coordinates.

6.2.2 Optimised Jacobian ECSM

Combined Jacobian ECPD and ECPA

This subsection uses an optimised Jacobian point addition and point doubling which takes the input $A(X, Y, Z)$, and generates the coordinate outputs for $2A$ and $2A + P$, where

P is the base point given by NIST. One example is for the input P, the outputs will be $2P$ and $3P$. Another example is for the input $3P$, the outputs will be $6P$ and $7P$. This combined hardware architecture will be referred to as the Jacobian point addition and point doubling (PAPD) module. The hardware architecture will always compute the "double" and "double plus P", where P is the base point determined by NIST in appendix A. This optimised PAPD module is shown in figure 6.9. This architecture has reduces the area of binary field operations to just 23 multiplication modules and 11 addition modules which is a lot less than that shown previously for Jacobian ECPA and ECPD.

Optimised Jacobian Controller

This optimised implementation performs the 'double and add' method from left to right and ECSM can be achieved in $(m-1)*(PAPD+2)$ clock cycles, where $PAPD$ is the latency required for the PAPD architecture. Just like the Jacobian ECSM architecture presented in chapter 5.2, the conversion from Jacobian to affine coordinates must also be performed afterward. The hardware architecture for the optimised Jacobian controller using the Jacobian PAPD module is shown in figure 6.10.

This hardware architecture uses two counters to control the ECC operations during ECSM. A counter *count* is used to iterate through the bits of the key, and another counter *count_PAPD* which is used to indicate when the Jacobian PAPD module is complete. $PA(x,y,z)$ will be used to refer to the X_3, Y_3, and Z_3 coordinates which result from the Jacobian PAPD module as shown in figure 6.9. $PD(x,y,z)$ will be used to denote the values for X_2, Y_2, and Z_2 of the PAPD module. A simplified high-level flowchart is shown in figure 6.11. The registers for $Q(x,y,z)$ are initialised with the base point as specified by NIST in appendix A if the $key(m-1) =' 1'$, otherwise it is initialised with $(0,0,0)$. *Count* is initialised with $m-2$ while *count_PAPD* is initialised with 0. Another register is used to store the results of $PD(x,y,z)$ of the PAPD module.

In each iteration, Q is used to accumulate the addition of points depending on the value of $key(count)$ and the current values for $PD(x,y,z)$. The conditions used to determine the next values for the $Q(x,y,z)$ registers is shown by table 6.3 which represents the comparator logic and outputs of the multiplexor connected to it as shown in figure 6.10. After each iteration of the key, *count_PAPD* is set to 0 and the PAPD module is run. Since the critical path of the PAPD module is $7M+7$, an additional clock is given for saving the values in a register. ECSM is completed once the *count* is 0, and $count_PAPD$ is $7M+8$.

Conditions		Next Register Value for $Q(x,y,z)$
$key(count) =$ '1'	$PD(x,y,z) = 1P$	$1P$
	$PD(x,y,z) = (0,0,0)$	$2P$
	all others	$PA(x,y,z)$
$key(count) =$ '0'	-	$PD(x,y,z)$

Table 6.3: The conditions for ECPA which determine the next values for the registers representing $Q(x,y,z)$.

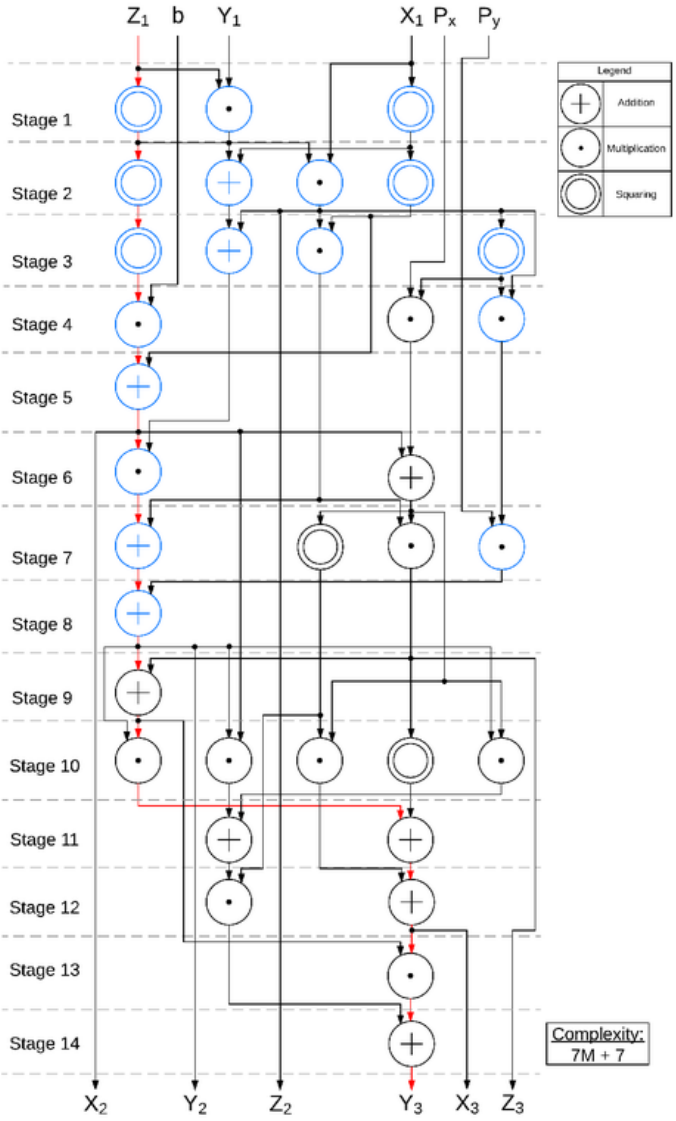


Figure 6.9: Hardware architecture for the optimised Jacobian PAPD. The blue modules for the Galois field operator modules are used to perform Jacobian point doubling, where as the black modules are used to compute Jacobian point addition.

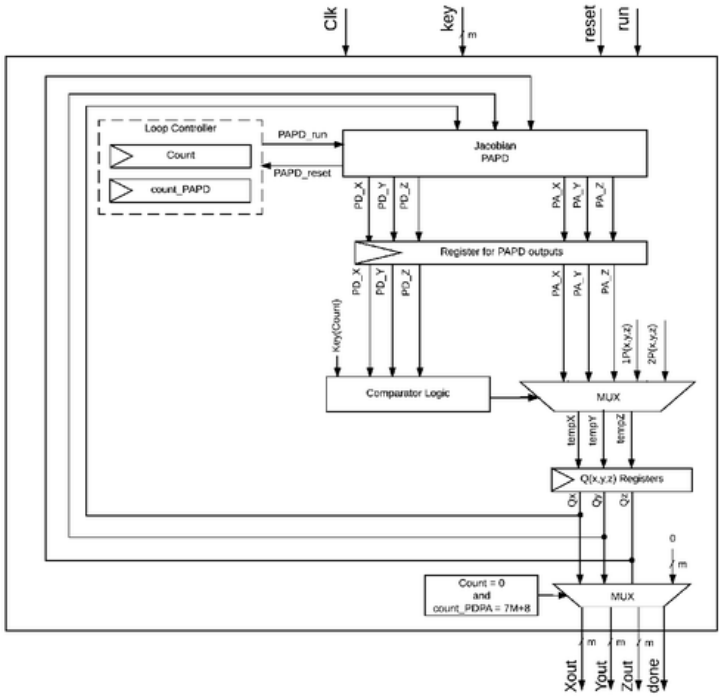


Figure 6.10: Hardware architecture for ECSM using the Jacobian PAPD module.

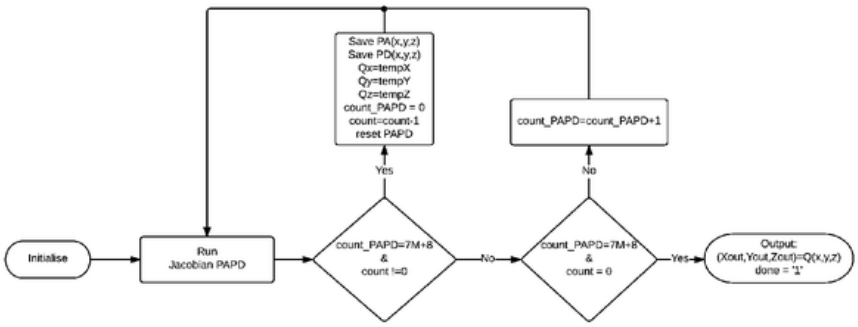


Figure 6.11: Simplified flow-chart for the optimised Jacobian ECSM using PAPD. The temporary values which update registers $Q(x, y, z)$ is shown by the conditions in table 6.3.

6.2.3 Simulations for ECSM optimisations.

This section presents the simulations using Xilinx ISim for the optimisations of ECSM over the binary field. The focus of the simulations shown here is to show the clock cycles required to perform a single ECSM. The actual output values of ECSM were checked with tests for ECSM in chapter 5. Figures 6.12 and 6.13 show the concurrent Jacobian implementation of ECSM with $k = 3$ and $k = n - 1$ respectively. From these simulations it is shown that a constant computation time for ECSM is obtained regardless of the key. The 283-bit implementation of concurrent Jacobian ECSM is also shown in figure 6.15 and figure 6.16 for k=3 and $k = n - 1$ respectively. The simulations for the optimised Jacobian ECSM using the PAPD module is then shown in figure 6.14 for $GF(2^{233})$ and in figure 6.17 for the implementation in $GF(2^{283})$.

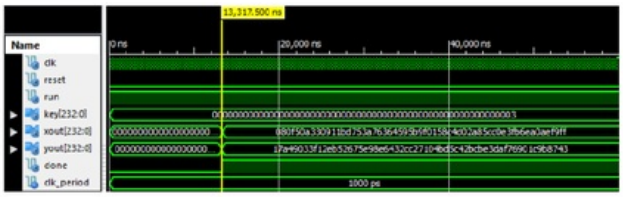


Figure 6.12: Simulation results for concurrent Jacobian ECSM with $k = 3$.

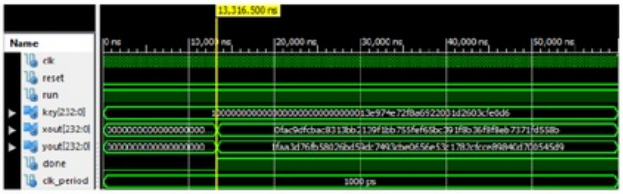


Figure 6.13: Simulation results for concurrent Jacobian ECSM with $k = n$-1.

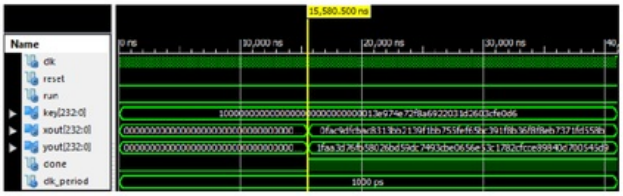


Figure 6.14: Simulation results for Jacobian ECSM using the PAPD module with $k = n$-1.

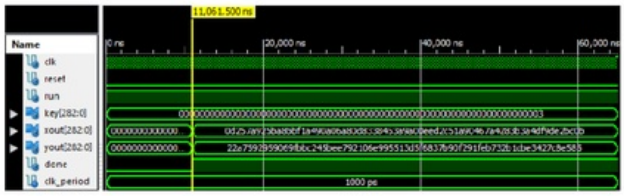


Figure 6.15: Simulation results for concurrent Jacobian ECSM with $k = 3$.

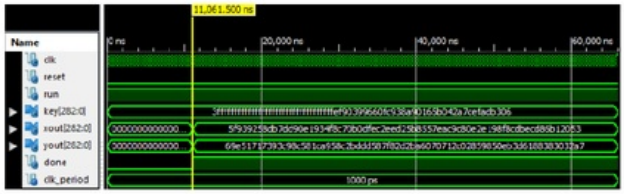


Figure 6.16: Simulation results for concurrent Jacobian ECSM with $k = n$-1.

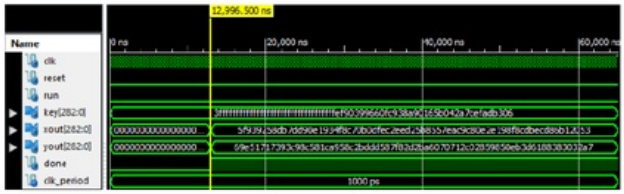


Figure 6.17: Simulation results for Jacobian ECSM using the PAPD module with $k = n$-1.

Chapter 7

Results and discussion of ECC operations

This chapter presents the results and analysis of the hardware design and implementations used in this project. The technology used for synthesis of the design in FPGA is the Xilinx Virtex-6 (XC6VLX760-2ff1760) device. The aim of this research was to produce an efficient implementation of ECSM where a focus was used on reducing the latency in terms of clock cycles through the use of efficient algorithms and concurrent hardware architectures. The hardware implementation of Galois field operations over a binary field presented in chapter 4 will be analysed in terms of area and time in section 7.1 for inversion, traditional digit-serial multipliers, and modified digit-serial multipliers. The results for the hardware implementation of ECSM in $GF(2^{233})$ and $GF(2^{283})$ are then shown in section 7.2, which show the area and time complexities for the basic methods of implementing ECSM in affine coordinates and Jacobian coordinates in chapter 5 and the optimisations in chapter 6.

7.1 Results for Galois Field Arithmetic in $GF(2^m)$

7.1.1 Inversion

Binary Field	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (μs)	Area-Time (slices×time)
$GF(2^{233})$	468	3.709	269.640	542	1.74	940.81
$GF(2^{283})$	568	3.718	268.962	618	2.11	1,305.11

Table 7.1: Synthesis results for inversion in $GF(2^{233})$ and $GF(2^{283})$

7.1.2 Traditional Digit-Serial Multipliers

Digit-serial multipliers with the digit sizes of 1, 2, 4, 8, 16, 32 and 64 bit hardware implementations were proposed previously in section 4.1.2. The results shown by the

synthesis reports show that an increase in area from higher digit-sizes for a reduction of time is able to achieve much better area-time than bit-serial multiplication. As the digit size doubles, the clock-cycles required to complete a single multiplication is halved due to the fact that clock cycles is determined by m/d for traditional digit-serial multipliers. The trade-off for this reduction of clock-cycles is the increase of area. The results for the $GF(2^{233})$ implementation of traditional digit-serial multipliers is shown in table 7.2. These results reveal that area-time is improved from digit-sizes 1 until 32, where the trade-off for more area and less time is no longer worth it for the 64-bit digit-serial multiplier. The synthesis results for the $GF(2^{283})$ bit implementation is shown in table 7.4. This table shows a similar trend where the area-time is improved as the digit-size increases for all digit-sizes implemented.

7.1.3 Modified Digit-Serial Multipliers

The modified digit-serial multipliers were implemented as discussed in section 6.1 for the digit sizes of 1, 2, 4, 8, 16, 32 and 64 bits. The $GF(2^{233})$ and $GF(2^{283})$ implementation results of these modified multipliers are shown in table 7.3 and table 7.5 respectively. The clock cycles required to complete a single multiplication using the modified digit-serial multiplier is given by $2*\lceil sqrtm/d \rceil$. The modified multiplier with the best area-time is for the digit-size of 4 for both the 233-bit implementation and the 283-bit implementation.

7.1.4 Comparison of multipliers

Both the traditional digit-serial multipliers and the modified digit-serial multipliers have their own advantages and disadvantages. Graphs which compare the multipliers in $GF(2^{233})$ in terms of time, area and area-time are shown in figure 7.1. The comparison of $GF(2^{283})$ is then shown in figure 7.2. The multiplier with the best area in terms of slices is achieved by the traditional bit-serial multipliers ($d = 1$). However, this has the slowest computation time. For high-speed implementations, the modified digit serial multipliers should be used for the best computation time at the expense of a lot of area being used. The implementation of modified digit-serial multipliers offer better area-time than traditional digit-serial multipliers in $GF(2^{233})$ for the digit-sizes of 1, 2, 3 and 4. In $GF(2^{283})$ the modified multipliers also offer better area-time for the digit-sizes of 1, 2, 4, 8 and 32. For optimal area-time the traditional digit-serial multipliers should be used rather than the modified digit-serial multipliers, where the best overall area-time is achieved for the digit-size of 32 for $GF(2^{232})$ and 64 for $GF(2^{283})$. In order to achieve an efficient implementation of ECSM, the multipliers with the most optimal area-time were then used to implement ECPA, ECPD. The results of these implementations are discussed in the following section.

Digit-size	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (ns)	Area-Time (slices×time)
1	233	3.085	324.180	320	718.81	230,017.60
2	117	3.131	319.425	350	366.33	128,214.45
4	58	3.310	302.143	512	195.29	99,988.48
8	30	3.588	278.707	785	107.64	84,497.40
16	15	3.288	304.179	1,377	53.03	73,015.43
32	8	3.535	282.852	1,525	28.28	43,127.00
64	4	3.607	277.230	3,340	14.43	48,189.52

Table 7.2: Synthesis results for the 233-bit implementation of traditional digit-serial multipliers.

Digit-size	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (ns)	Area-Time (slices×time)
1	33	2.533	394.86	1,750	83.59	146,280.75
2	23	2.042	489.61	1,430	46.97	67,161.38
4	17	1.779	561.97	1,806	30.24	54,618.86
8	13	1.651	605.53	3,015	18.16	54,755.42
16	9	2.031	492.45	4,071	18.28	74,413.81
32	7	2.195	455.48	5,501	15.37	84,522.87
64	5	2.620	381.74	6,530	13.10	85,543.00

Table 7.3: Synthesis results for the 233-bit modified digit-serial multipliers.

Digit-size	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (ns)	Area-Time (slices×time)
1	283	3.309	302.19	584	936.45	546,885.05
2	142	3.599	277.88	354	511.06	180,914.53
4	71	3.648	274.15	508	259.01	131,576.06
8	36	3.933	254.27	743	141.59	105,199.88
16	18	3.857	259.27	1,470	69.43	102,056.22
32	9	4.057	246.51	2,028	36.51	74,048.36
64	5	3.717	269.02	3,765	18.59	69,972.53

Table 7.4: Synthesis results for the 283-bit implementation of traditional digit-serial multipliers.

Digit-size	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (ns)	Area-Time (slices×time)
1	35	2.155	464.08	2,117	75.43	159,674.73
2	25	1.965	508.80	1,928	49.13	94,713.00
4	19	1.607	622.37	2,347	30.53	71,660.95
8	13	1.801	555.32	3,633	23.41	85,059.43
16	10	2.042	489.79	5,003	20.42	102,161.26
32	7	2.196	455.41	5,839	15.37	90,587.20
64	7	2.613	382.71	10,024	18.29	183,348.98

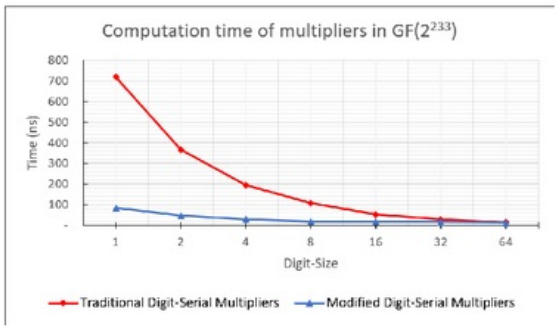
Table 7.5: Synthesis results for the 283-bit modified digit-serial multipliers.

ECSM Method	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (μs)	Area-Time (slices×time)
Affine (sequential)	174,885.00 (average)	4.346	230.08	11,652	760.05	8,856,105.05
Affine (concurrent)	116,734	4.271	234.12	10,851	498.57	5,409,992.99
Jacobian (sequential)	17,188 (average)	4.784	209.02	45,821	82.23	3,767,741.33
Jacobian (concurrent)	13,315	4.710	212.31	44,433	62.71	2,786,555.61
Jacobian (PAPD)	15,579	4.027	248.35	35,997	62.74	2,258,330.58

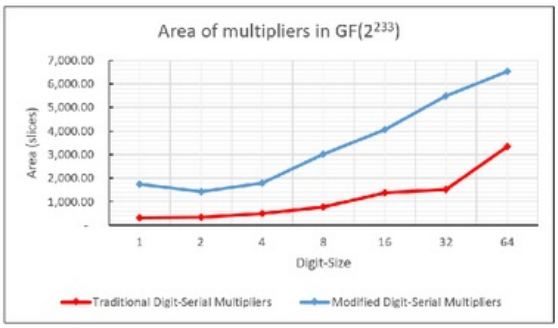
Table 7.6: Synthesis results for the ECSM over the binary field $GF(2^{233})$.

ECSM Method	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (μs)	Area-Time (slices×time)
Affine (sequential)	251,035 (average)	4.614	216.71	21,309	1,158.28	24,681,692.42
Affine (concurrent)	167,537	4.414	226.58	20,101	739.51	14,864,856.70
Jacobian (sequential)	14,050 (average)	4.590	217.86	86,412	64.49	5,572,666.67
Jacobian (concurrent)	11,060	4.824	207.298	85,612	53.35	4,567,694.71
Jacobian (PAPD)	12,996	4.689	213.266	74,824	60.94	4,559,643.17

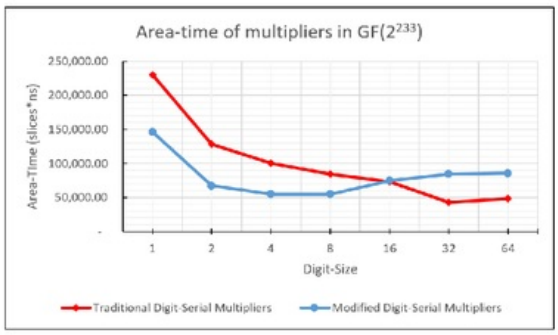
Table 7.7: Synthesis results for the ECSM over the binary field $GF(2^{283})$.



(a)

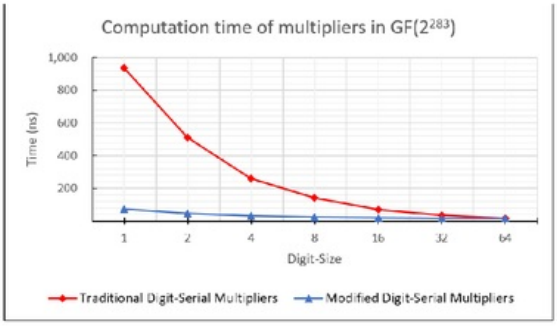


(b)

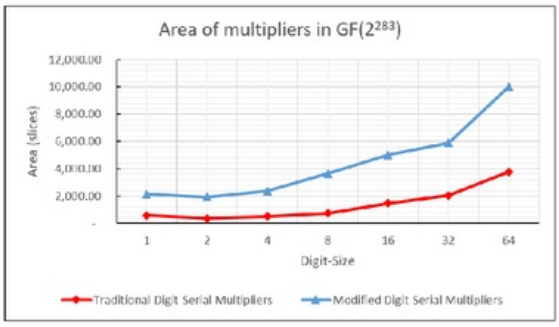


(c)

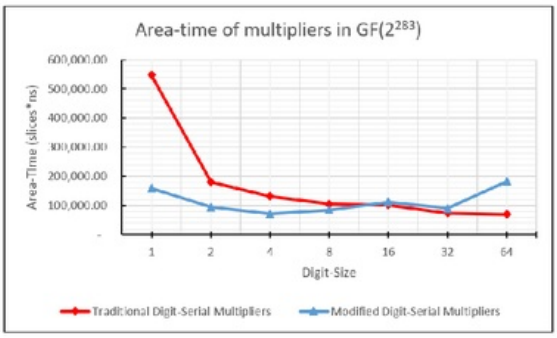
Figure 7.1: Comparison of traditional and modified digit-serial multipliers with various digit-sizes in $GF(2^{233})$. (a) shows the computation time, (b) shows the slices occupied, and (c) shows the area-time.



(a)



(b)



(c)

Figure 7.2: Comparison of traditional and modified digit-serial multipliers with various digit-sizes in $GF(2^{283})$. (a) shows the computation time, (b) shows the slices occupied, and (c) shows the area-time.

Binary Field	Clock cycles	Clock period (ns)	Frequency (MHz)	Slices	Time (μs)	Area-Time (slices×time)
$GF(2^{233})$	16,048	4.033	247.928	35,615	64.72	2,305,059.21
$GF(2^{283})$	13,565	4.689	213.266	69,908	63.61	4,446,588.17

Table 7.8: Synthesis results for the most efficient ECSM over the binary field $GF(2^{233})$ and $GF(2^{283})$ with serial inputs and outputs.

7.2 Comparison of ECSM hardware implementations over $GF(2^m)$

Operation	Complexity in terms of clock cycles
Addition	1
Traditional digit-serial multiplication	$\lceil m/d \rceil$
Modified digit-serial multiplication	$2 * \lceil \sqrt{m/d} \rceil + 1$
Inversion	2m+2
Affine Point Doubling (APD)	3M+5+Inversion
Affine Point Addition (APA)	3M+7+Inversion
Jacobian Point Doubling (JPD)	5M+2
Jacobian Point Addition (JPA)	6M+5
Jacobian PA and PD (PAPD)	7M+7
Conversion	4M+Inversion
Affine ECSM (sequential)	(m/2)*(APA+2)+m*(APD+2)+1
Affine ECSM (concurrent)	m*(APA+2)+1
Jacobian ECSM (sequential)	(m/2)*(JPA+2)+m*(JPD+2)+1+Conversion
Jacobian ECSM (concurrent)	m*(JPA+2)+1+Conversion
Optimised Jacobian ECSM	(m-1)*(PAPD+2)+Conversion
Serial-in-parallel-out (SIPO)	$m + 1$
Parallel-in-serial-out (PISO)	$m + 2$

Table 7.9: Summary of latencies for each operation involved in ECC where m is the bit-size of the binary field, M is the clock cycles required to perform one multiplication by either traditional digit-serial multiplication or the modified digit-serial multiplication, and d is the chosen digit-size.

In order to achieve the most efficient implementation of ECSM, the most efficient implementation of Galois field arithmetic should also be used. This research project focused on using different multipliers where the traditional digit-serial multipliers have been revealed to produce the best area-time with the digit-size of 32 for $GF(2^{233})$ and the digit size of 64 for $GF(2^{233})$. The following implementations of ECSM used these multipliers for the construction of affine point addition, affine point doubling, Jacobian point addition,

Jacobian point doubling and for Jacobian to affine conversion modules. The following methods of ECSM were then constructed for sequential affine and Jacobian elliptic curve scalar multiplication as shown in chapter 5, where these implementations were then optimised for a concurrent implementation of ECSM and also a combined Jacobian point addition and point doubling implementation as presented in chapter 6. The results for the $GF(2^{233})$ implementations of ECSM are shown in table 7.6 and the results for the $GF(2^{283})$ implementations are shown in table 7.7.

The total number of clock cycles required to perform a single ECSM are summarised in table 7.9 which shows the general formulas required to calculate the latency for each operation needed. The total clock cycles will vary depending on the multiplier used, the coordinate system, and the method of using ECPA and ECPD modules for performing ECSM. The clock cycles required for a single ECSM can be calculated by using table 7.9 and substituting the methods used. The total clock cycles calculated are also reflected as shown in the simulation results.
An example of calculating the clock cycles for the implementation of ECSM over the binary field $GF(2^{233})$ using the Jacobian coordinate system, concurrent ECSM methods and the traditional digit-serial multiplier with a digit-size of 32 is as follows:

Total clock cycles =

$$
\begin{aligned}
&= m*(JPA+2)+Conversion+1 && \text{, where } JPA = 6M+5+2\\
&= m*(6M+5+2)+Conversion && \text{, where } Conversion = 4M+Inversion\\
&= m*(6M+5+2)+4M+Inversion && \text{, where } Inversion = 2m+2\\
&= m*(6M+5+2)+4M+2m+2 && \text{, where } M = \lceil m/d \rceil\\
&= m*(6(\lceil m/d \rceil)+5+2)+4(\lceil m/d \rceil)+2m+2 && \text{, where } m = 233 \text{ and } d = 32\\
&= 233*(6*8+5+2)+4*8+2*233+2\\
&= 13,315
\end{aligned}
$$

Several observations can be drawn from table 7.6 and table 7.7 to indicate the advantages of the different types of ECSM methods to achieve the most efficient hardware implementation. As discussed in section 6.2.1, a concurrent implementation of ECSM using affine and Jacobian coordaintes can be achieved with a huge reduction of time, and a slight reduction of area. For the most area-efficient implementation, the affine coordinate system should be used since a total of 7 multiplication modules, 13 addition modules, and 2 inversion modules are required to construct ECPA and ECPD. This is significantly less than that of a the Jacobian implementation for a total of 34 multiplication modules, 11 addition modules, and 1 inversion module is required to construct the Jacobian ECPA, ECPD, and conversion from Jacobian to affine coordinates. Although an implementation in affine coordinates uses the least area, its computation time is extremely slow where ECSM takes 498.57 μs in $GF(2^{233})$ and 739.51 μs in $GF(2^{283})$. This is up to 13 times slower than an implementation in Jacobian coordinates for both concurrent implementations of ECSM.

The implementation of ECSM with the best time is the concurrent version of Jacobian coordinates which is shown for both $GF(2^{233})$ and $GF(2^{283})$ where a computation time of 62.71 μs and 53.35 μs is achieved respectively. It would be expected that the larger field of $GF(2^{283})$ would have a longer computation time. However, this is due to the fact that a 64-bit traditional digit-serial multiplier has the best area-time for $GF(2^{283})$ while the 32-bit traditional digit-serial multiplier had the best area-time for binary field $GF(2^{233})$. This means that the multiplication used in the $GF(2^{233})$ takes 8 clock cycles compared with the 5 clock cycles in $GF(2^{283})$.

Finally the most efficient implementation of ECSM is achieved using the optimised Jacobian ECSM with the combined Jacobian PAPD module. This is shown in table 7.6 and table 7.7 where the best area-time for an ECC hardware architecture is obtained. Table 7.8 shows the final results including the SIPO and PISO modules. The results show that a single scalar multiplication in $GF(2^{233})$ takes 16,048 clock cycles, has the computation time of 64.72μs and occupies 35,615 slices. A single ECSM is also performed in 13,565 clock cycles in 63.61μs and occupies 69,908 slices for the binary field $GF(2^{283})$.

The final results from this work is then compared with recent hardware implementations of ECSM found in the literature as mentioned in chapter 2. Table 7.10 shows a comparison for the computation time to perform a single ECSM. It shows that the design of ECSM used for this research is faster than most designs except for that by Sutter [23]. The work in this paper performs ECSM slightly faster than Lutz and Hassan who used Itoh and Tsuji's method for inversion and Lopez-Dahab coordinates for ECSM in $GF(2^{163})$. Their results indicate a slower ECSM architecture where the work presented in this research for larger binary fields $GF(2^{233})$ and $GF(2^{288})$ is still faster. However, the circuit designed by Sutter is 3 times faster than this implementation in $GF(2^{233})$ and almost twice as fast for the $GF(2^{283})$ implementation of ECSM. Their work uses a similar approach by using digit-serial binary operations for multiplication, division and use the Montgomery ladder method for ECSM. This indicates the disadvantages of the hardware implementation of the current system where better inversion techniques can be used as well as the use of improved coordinate systems.

Circuits	Bit-Size	Device	Time (μs)
This design	233	Virtex-6	64.72
This design	283	Virtex-6	63.61
Govem [3]	233	Virtex-5	240
Sutter [23]	233	Virtex-5	17.8
Sutter [23]	283	Virtex-5	33.6
Lutz and Hasan [13]	163	Virtex-E	75
Loi [12]	163	Virtex-5	59.15
Loi [12]	233	Virtex-5	84.19
Loi [12]	283	Virtex-5	102.10
Shu [22]	163	Virtex-E	48
Shu [22]	233	Virtex-E	89

Table 7.10: Comparison of the ECSM implementations with other work.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The primary aim of this project was for an efficient FPGA implementation of ECSM over the binary fields $GF(2^{233})$ and $GF(2^{283})$. A literature review was conducted to analyse the hardware architectures and algorithms used at each level of the elliptic curve hierarchy of operations from the Galois field arithmetic to ECPA, ECPD and ECSM methods. Several algorithms and hardware implementation methods were then chosen to be implemented, where the efficiency in terms of area and time were measured and analysed in chapter 7.

Initial Galois field operations were designed and implemented as presented in chapter 4 with a focus on the implementation of efficient multipliers. For all the hardware implementations used in this research project, addition was implemented using bit-wise XOR, and inversion was implemented using the modified extended Euclidean algorithm. Binary field multiplication was designed for several digit-sizes for $d = 1$, 2, 4, 8, 16, 32, and 64 so that the most efficient multiplier can be used for constructing the hardware to perform elliptic curve group operations.

Two coordinate systems were then compared, namely the affine coordinates and the Jacobian coordinates as presented in chapter 5. These required different ECPA and ECPD modules which were constructed with the most efficient binary field operations explored in this research project for multiplication, inversion and addition. It is concluded that an implementation in affine coordinates produces results with the best area but will have a significantly longer computation time than the implementation in Jacobian coordinates due to an inversion required every time ECPA and ECPD is performed. With the current Galois field modules designed in this research the Jacobian hardware implementation of ECSM obtains the faster computation time and efficient area-time.

The optimisation of ECC operations were then presented in chapter 6. These optimisations aimed to obtain more efficient multipliers by reducing the number of clock cycles required through the development of modified digit-serial multipliers with various digit sizes of $d = 1$, 2, 4, 8, 16, 32, and 64. The comparison of these modified-multipliers and the traditional digit-serial multipliers shows that for the binary fields $GF(2^{233})$ and $GF(2^{283})$, the traditional digit-serial multipliers are able to obtain the best efficient area-time out

of the digit-sizes explored. The modified digit-serial multiplications would be suitable for high-speed based applications of ECC.

Finally, an efficient hardware architecture for computing ECSM over the binary fields $GF(2^{233})$ and $GF(2^{283})$ has been successfully implemented using FPGA as shown in this research. Two types of multipliers were explored being the traditional digit-serial multipliers and the modified digit-serial multipliers so that the efficiency and trade-off in terms of area and time can be analysed. It was concluded that the the traditional digit-serial multipliers with the digit size of 32-bits for $GF(2^{233})$ and 64-bits for $GF(2^{283})$ obtain the best area-time. On a Xilinx Virtex-6 device, elliptic curve scalar multiplication is performed using the optimised Jacobian PAPD method. This ECSM processor performs a single point multiplication in 62.74μs occupying 35,997 slices for $GF(2^{233})$. For an implementation in $GF(2^{283})$, a single ECSM takes 60.94μs and has the area of 69,908 slices.

8.2 Future Work

The outcome of the research presented in this thesis is that an efficient ECSM hardware architecture for the binary fields $GF(2^{233})$ and $GF(2^{283})$ was achieved. The methods explored had a high focus on improving binary field multiplication and the comparison between affine coordinates and Jacobian coordinates. The results obtained were limited to the binary field of $GF(2^{233})$ and $GF(2^{283})$. To obtain more complete conclusions, hardware implementations for ECSM in the remaining NIST recommended curves must be conducted for $GF(2^{163})$, $GF(2^{409})$ and $GF(2^{571})$. Accurate conclusions may then be drawn once these binary fields have been implemented such as the use of modified digit-serial multipliers are expected to have better performance in larger binary fields.

The ECSM hardware design for use in ECC in this research can also be further optimised. There are multiple appealing algorithms and hardware architectures which were reviewed in the literature that were not used in this design. At the Galois field arithmetic level, the use of digit-parrallel multipliers and their efficiency in terms of area and time can also be analysed and compared with the results of this research. Another area of focus to improve the efficiency of inversion or division algorithms which is the most costly binary field operation which makes affine-coordinates unappealing. Further improvements have also been identified from exploration of different coordinate systems which will reduce the number of underlying binary field operations and overall area such as Lopez-Dahab projective coordinates or the Mongtomery method for ECSM.

Chapter 9

Abbreviations

AES	Advanced Encryption Standard
APA	Affine Point Addition
APD	Affine Point Addition
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECPA	Elliptic Curve Point Addition
ECPD	Elliptic Curve Point Doubling
ECPM	Elliptic Curve Point Multiplication
ECSM	Elliptic Curve Scalar Multiplication
EPA	Execute Point Addition
EPD	Execute Point Addition
EXE	Execute
FFA	Finite Field Arithmetic
FPGA	Field-Programmable Gate Array
GF	Galois Field
GFA	Galois Field Arithmetic
INI	Initialise
ISE	Integrated Software Environment
JPA	Jacobian Point Addition
LUT	Look Up Table
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
PA	Point Addition
PAPD	Point Addition and Point Doubling
PD	Point Doubling
PE	Processing Element
PISO	Parallel-In-Serial-Out
RST	Reset

RSA	Ron Rivest, Adi Shamir and Leonard Adleman cryptosystem
SIPO	Serial-In-Parallel-Out
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XOR	Exclusive-or
XST	Xilinx Synthesis Tool

Appendix A

NIST curves for the binary field $GF(2^{233})$ and $GF(2^{283})$

B-233: $m = 233$, $f(z) = z^{233} + z^{74} + 1$, $a = 1$, $h = 2$	
$S =$	0x 74D59FF0 7F6B413D 0EA14B34 4B20A2DB 049B50C3
$b =$	0x 00000066 647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD
$n =$	0x 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7
$x =$	0x 000000FA C9DFCBAC 8313BB21 39F1BB75 5FEF65BC 391F8B36 F8F8EB73 71FD558B
$y =$	0x 00000100 6A08A419 03350678 E58528BE BF8A0BEF F867A7CA 36716F7E 01F81052
B-283: $m = 283$, $f(z) = z^{283} + z^{12} + z^7 + z^5 + 1$, $a = 1$, $h = 2$	
$S =$	0x 77E2B073 70EB0F83 2A6DD5B6 2DFC88CD 06BB84BE
$b =$	0x 027B680A C8B8596D A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A F6263E31 3B79A2F5
$n =$	0x 03FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFEF90 399660FC 938A9016 5B042A7C EFADB307
$x =$	0x 05F93925 8DB7DD90 E1934F8C 70B0DFEC 2EED25B8 557EAC9C 80E2E198 F8CDBECD 86B12053
$y =$	0x 03676854 FE24141C B98FE6D4 B20D02B4 516FF702 350EDDB0 826779C8 13F0DF45 BE8112F4

K-233: $m = 233$, $f(z) = z^{233} + z^{74} + 1$, $a = 0$, $b = 1$, $h = 4$	
$n =$	0x 00000080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF
$x =$	0x 00000172 32BA853A 7E731AF1 29F22FF4 149563A4 19C26BF5 0A4C9D6E EFAD6126
$y =$	0x 000001DB 537DECE8 19B7F70F 555A67C4 27A8CD9B F18AEB9B 56E0C110 56FAE6A3
K-283: $m = 283$, $f(z) = z^{283} + z^{12} + z^7 + z^5 + 1$, $a = 0$, $b = 1$, $h = 4$	
$n =$	0x 01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFE9AE 2ED07577 265DFF7F 94451E06 1E163C61
$x =$	0x 0503213F 78CA4488 3F1A3B81 62F188E5 53CD265F 23C1567A 16876913 B0C2AC24 58492836
$y =$	0x 01CCDA38 0F1C9E31 8D90F95D 07E5426F E87E45C0 E8184698 E4596236 4E341161 77DD2259

Figure A.1: The NIST recommended random and Koblitz curves and parameters for the binary field $GF(2^{233})$ and $GF(2^{283})$ [7].

Appendix B

Supervisor Consultations

Appendix B shows the attendance sheet required for this project for supervisor consultations and participation.

B.1 Supervisor Consultation Attendance Form

Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
1	3/08/16	discussed about PA multiplication & inversion		3/08/16
2	10/08/16	discussed digit-serial multiplication		10/08/16
3	17/08/16	bit serial mult work, trying digit serial ckt		17/08/16
4	24/08/16	digit serial mult working PD, PA almost done - PM to be done.		24/08/16
5	31/08/16	PM in jac working trying inv & PDPA module		31/08/16
6	07/08/16	trying PDPA for PM & inversion to be done.		07/09/16
7	14/09/16	discussed optimization of mult. & PM in jac		14/09/16
8	21/09/16	implemented new field multi & trying PM using this.		21/09/16
9	28/09/16	PM in jac & affine done using PDPA. diagram & block diagram		28/09/16
10	5/10/16	trying to make all figures. Almost done.		5/10/16
11	12/10/16	finished most of the compass, writing thesis		12/10/16
12	19/10/16	writing thesis 75% almost done.		19/10/16
13	26/10/2016	writing thesis Almost done -		26/10/16
14	2/11/2016	Thesis writing is nearly done		02/11/16

Figure B.1: Attendance sheet for the weekly consultation meetings with the supervisor.

Bibliography

[1] H. Brunner, A. Curiger, and M. Hofstetter, "On computing multiplicative inverses in gf (2 m)," *IEEE Transactions on computers*, vol. 42, no. 8, pp. 1010–1015, 1993.

[2] W. N. Chelton and M. Benaissa, "Fast elliptic curve cryptography on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 198–205, 2008.

[3] B. Gövem, K. Järvinen, K. Aerts, I. Verbauwhede, and N. Mentens, "A fast and compact fpga implementation of elliptic curve cryptography using lambda coordinates," in *International Conference on Cryptology in Africa*. Springer, 2016, pp. 63–83.

[4] Z. Guitouni, R. Chotin-Avot, M. Machhout, H. Mehrez, and R. Tourki, "High performances asic based elliptic curve cryptographic processor over gf (2m)," *International Journal of Computer Applications*, no. 4, pp. 1–10, 2011.

[5] J.-H. Guo and C.-L. Wang, "Systolic array implementation of euclid's algorithm for inversion and division in gf (2 m)," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1161–1167, 1998.

[6] D. Gustavo and J. Deschamps, "Efficient elliptic curve point multiplication using digit-serial binary field operations," *Industrial Electronics, IEEE Transactions on*, vol. 60, no. 1, pp. 217–225, 2013.

[7] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[8] M. S. Hossain, E. Saeedi, and Y. Kong, "High-speed, area-efficient, fpga-based elliptic curve cryptographic processor over NIST binary fields," in *2015 IEEE International Conference on Data Science and Data Intensive Systems*. IEEE, 2015, pp. 175–181.

[9] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in gf (2m) using normal bases," *Information and computation*, vol. 78, no. 3, pp. 171–177, 1988.

[10] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, pp. pp. 417–426, 1987.

[11] J. P. Lee and C. Yng, "Low-latency digit-serial and digit-parallel systolic multipliers for large binary extension fields," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 60, no. 12, pp. 3195–3204, 2013.

[12] K. C. Loi, S. An, and S.-B. Ko, "Fpga implementation of low latency scalable elliptic curve cryptosystem processor in gf (2 m)," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 822–825.

[13] J. Lutz and A. Hasan, "High performance fpga based elliptic curve cryptographic co-processor," in *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, vol. 2. IEEE, 2004, pp. 486–492.

[14] V. Miller, "Uses of elliptic curves in cryptography," in *Advances in Cryptology-CRYPTO'85 Proceedings*. Springer Berlin Heidelberg, 1985, pp. pp. 417–426.

[15] *FIPS 186-2*, National Institute of Standards and Technology (NIST) Std., 2000.

[16] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for gf (2m)," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2000, pp. 41–56.

[17] C. Paar and J. Pelzl, *Understanding cryptography*. Springer Science & Business Media, 2009.

[18] C. Rebeiro, "Revisiting the itoh-tsujii inversion algorithm for fpga platforms," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 8, pp. 1508–1512, 2011.

[19] C. Rebeiro, S. S. Roy, and D. Mukhopadhyay, "Pushing the limits of high-speed gf (2 m) elliptic curve scalar multiplication on fpgas," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 494–511.

[20] K. B. Resnahan, "Elliptic curve cryptography," September 2016, accessed: 25-10-2016. [Online]. Available: http://www.slideshare.net/Kellybresnahan/elliptic-curve-cryptography-66406021

[21] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *CACM*, vol. 21, no. 2, pp. 120–126, 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[22] C. Shu, K. Gaj, and T. El-Ghazawi, "Low latency elliptic curve cryptography accelerators for nist curves over binary fields," in *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.* IEEE, 2005, pp. 309–310.

[23] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Efficient elliptic curve point multiplication using digit-serial binary field operations," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 1, pp. 217–225, 2013.