ELLIPTIC CURVE CRYPTOGRAPHIC PROTOCOLS: A SIMULATION AND IMPLEMENTATION OF ECDSA AND ECDH

Nathan Levett

Bachelor of Engineering with Bachelor of Science
Computer Engineering, Mathematics

Department of Electronic Engineering
Macquarie University

April 4, 2017

Supervisor: Mr. MD Selim Hossain
Supervisor: Dr. Yinan Kong

MACQUARIE UNIVERSITY

DEPARTMENT APPROVAL

of a senior thesis submitted by

Nathan Levett

This thesis has been reviewed by the research advisor, research coordinator, and department chair and has been found to be satisfactory.

Date	Supervisor: Mr. MD Selim Hossain Supervisor: Dr. Yinan Kong, Advisor
Date	Department Ugrad Coordinator , Research Coordinator
Date	Department Chair Name, Chair

ACKNOWLEDGMENTS

I would like to acknowledge several fonts of inspiration and encouragement to pursue this thesis, and more largely, this degree. I would like to acknowledge Huawei for providing me the opportunity to be a part of their 2013 Winter Vacation to their Shenzhen campus, which inspired me to choose Computer Engineering as a major. I would also like to acknowledge the Macquarie University PACE Office, for funding my travel to, and participation in, the 2014 Winter Abroad "Digital System Design" unit taught in Changchun, Jilin, which inspired a deep appreciation for HDL's, which inspired half of this thesis. The other half of this thesis, reliant on Field Theory, causes thanks to be due to Dr Chris Cooper of the MQU Mathematics Department, for his inspirational teaching, and his courses and notes in Group, Ring, Field and Galois Theory, during his last semester teaching before retiring. I would also like to acknowledge and thank Dr Yinan Kong for making this possible, all the way from acting as an academic advisor prior to enrolling at MQU, through teaching the Computer Major stream, and designing the thesis goal for this, to suit my interest in computational applications of Galois Theory. Now is the appropriate time to also offer a shout-out to the mathematicians without whose work this area would not exist; Pioneer of Algebra - developer of Group Theory - Evariste Galois, masters of Elliptic Functions - Carl Gustav Jacob Jacobi and Karl Weierstrass, number theorists - Euclid, Etienne Bezout and Pierre de Fermat.

ABSTRACT

Elliptic Curve Cryptography provides a secure means of implementing the Digitial Signature Algorithm protocol through the Elliptic Curve Digitial Signature Algorithm scheme. The ECDSA upholds the data security objectives of information authentication and non-reputability. Similarly, the Elliptic Curve Diffie-Hellman upholds information confidentiality and information integrity. These schemes, whilst theoretically cryptographically secure, are expensive to implement up to certain security standards, with timeliness and size of circuit required to perform either scheme being the main considerations. To have any appreciable speed, they must be implemented in digital hardware design. This project outlines the design methodology and procedure of simulating a generalisable implementation of an Elliptic Curve Cryptographic Processor, able to effectively utilise both the ECDH and ECDSA, over any chosen bit length.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

1.1.1 Pretext: Cryptography in its Historical Context

Cryptography; historically the obscuring of information, typically text, through means of some encryption by an individual, in effort of preventing that information from being accessed by individuals that did not have access to the method of decrypting (through a decryption scheme relative to the encryption scheme) [20]. The depth and breadth of 'cryptography' has increased since it's historic usage, and today the field of Cryptography covers a range of topics related to 'information security.' Typically cryptography refers to 'information security objectives', criteria that establish certain properties of an information's security, the schemes and protocols that enforce or uphold these criteria as they relate to the information security objectives, as well as the mathematical and information theory that underpins these schemes and protocols. In modern usage, it is easier to speak about cryptography more broadly from the top down, to address the topics as per the information security objectives that they relate to. These security objectives typically include the following properties of information; confidentiality, integrity, non-repudability, and authenticity [20].

Confidentiality refers to the obscuring of information such that someone without access to the schemes for encryption/decryption is prevented from observing the information. Integrity refers to the assurance than information has not been altered from it's intended form, that it remains accurate and unchanged. Non-repudability refers to the capability to prevent a progenitor of information from claiming they did not create it. Authenticity refers to the demonstrable authorship of information by a particular source, guaranteeing that they were the progenitor of the information. Note that non-repudability and authenticity work hand in hand; if information is authenticatable as being from a particular source, then it should also serve to make that information non-repudable by the source [1].

1.1.2 Context: Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is the utilisation of the mathematical objects known as Elliptic Curves, to implement cryptographic schemes to provide the information security properties mentioned above (1.1.1), in a Public Key Cryptosystem (PKC), a system that facilitates the use of individual 'private keys' for decryption of messages encrypted with a corresponding 'public key', where the public key is computed in some way from the private key, and shared in an open channel. Primarily, these schemes include the Elliptic Curve Diffie Hellman (ECDH) scheme, along with the Elliptic Curve Digital Signature Algorithm (ECDSA) scheme [11].

ECDH provides a means of establishing a 'shared secret' through utilising the Diffie-Hellman Key Exchange (DHKE) protocol where the corresponding public and private keys are keys generated utilising the structure of Elliptic Curves, wherein the public key is computed from the "Point Multiplication" of an Elliptic Curve's "Base Point" by a randomly generated or precomputed private key. ECDSA utilises the Digital Signature Algorithm (DSA) protocol, again with the public and private keys generated from the structure of an Elliptic Curve, to provide the means of 'signing' some information with a private key, only known to the signer, which can be verified as being signed with the unique private key through a verification algorithm using the public key, which is known to everyone. This is such that a person may sign information with their private key, and anyone is able to verify that the signature belongs to that individual [20].

The DHKE protocol, and hence the ECDH scheme, provides information confidentiality and information integrity, whilst the DSA protocol, and hence the ECDSA scheme, provides information authentication and non-repudability [20] [11]. As such, it is the goal of this project to, broadly speaking, develop a working implementation of the ECDH and ECDSA schemes. This is elaborated on below (1.2).

1.2 Project Goals

This project aims to process, and document, the development of a working simulation model of an implementation of the ECDH and ECDSA schemes. Involved in this is the choice of manner for implementation. Choosing the method of implementation for this project is reliant on complying with the project requirements, and it is a requirement of this project that the method of implementation be through a Hardware Description Language (HDL). Because of this, VHSIC HDL (VHDL) will be the HDL used in this project, although the trade-off between this and other HDL's should be minimal. The project requirement of utilising a HDL for the scheme's implementation is due to the project requirement that the simulation of the implementation must be synthesisable for a Field Programmable Gate Array (FPGA). Hence, the choice of target FPGA to be used influences the choice of Integrated Synthesis Environment (ISE), which influences which HDL's are available, as per those available for synthesis. This project will utilise the Xilinx ISE, which provides both VHDL and Verilog synthesis, and the selection of either is down to preference. This facilitates that the synthesised HDL will be able to target any FPGA for which the Xilinx ISE can Map the synthesis to.

Beyond the selection of a method 'for' implementation, is the selection of what to implement. It is the overall goal of the project to implement the ECDH and ECDSA schemes, but the schemes are reliant on the selection of suitable 'domain parameters' in which to perform the operations that constitute the schemes. This project will be operating under the selection of the domain parameters of the "NISTsecp256r1" curve [25]; NISTsec- a standardised NIST supported curve published by SEC, -p- 'over a prime field', -256- in which the prime is 256 bits in length, -r- that the curve parameters are generated verifiably at random, -1 identity of the curve parameters in the family of curves that are under the same prefixes. However, implementing a specific curve is the least significant aspect of the project, as discussed in the Project Philosophy, wherein the only constraint imparted on the project from the goal of implementing the NISTsecp256r1 curve is that the implementation is able to operate on a 'domain parameter set' of 256 bits.

Hence we can succinctly state the project goals as being to process and document the development of an FPGA-synthesisable simulation of a VHDL implementation of the ECDH and ECDSA schemes, valid up to domain parameters 256 bits in length.

1.3 Project Philosophy and Constraints

Although the original intention of the design process was to establish a working implementation of ECDH and ECDSA over a specific curve for a known target FPGA, which would have prompted a heavy focus of design optimisation for a known target bit length, and design minimisation through use of the "IEEE numeric standard library" for a known maximum on-board multiplier size, an unintended constraint of the project that prevented this was the lack of availability of a Xilinx ISE license key, other than a free license key. This limited the degree to which the Xilinx ISE could be used to synthesise for specific targeted FPGAs, however it did not limit the degree to which the Xilinx simulator, iSim, could be used to process simulated wave configuration models. Hence, early on in the design process, the Project Philosophy shifted from design optimisation and minimisation for a known target bit length and target FPGA, and shifted to a focus on design generalisability and design totality.

This shift in Project Philosophy from optimisation to generalisability yielded the shift in Project goal from being the construction of an ECDH and ECDSA processor of a specific target bit length, optimised to that bit length, to the variable construction of an ECDH and ECDSA processor for any target bit length, generalisable over the choice of target bit length.

Similarly, the Project Philosophy shift from minimisation to totality yielded the shift in Project goal from the use of the "IEEE numeric standard library" to explicitly not using it. Despite that this rampantly increases the SLICE utilisation on the FPGA board, it allows the design to be generalisable to target FPGA boards that support varying fixed multiplier lengths, and it guarantees a more uniform behaviour across different boards, whilst also strictly enforcing a known combinational path length in non-clocked circuit components, a path length that would change more non-deterministically with the use of the IEEE numeric standard library (despite being purely deterministic for a known FPGA, this statement is with regards to the range of all FPGAs).

These changes to the Project Philosophy are represented in the Design Philosophy adopted throughout the project, such that prior to the change, the Design Philosophy was to produce a design that incorporated a hashing processor and cryptographically-secure-pseudo-random-number-generator, however, after adopting the change, the board no longer has room to support these features (for supporting these features with bit sizes up to 256), however this led to an easier adaptation of the "Point Multiplication" algorithm to have a more consistent time and power profile, with the time profile being constant, operating over the entire key length with intentionally no time optimisation (indeed, effort was contributed to intentionally removing time optimisation that was inherent in earlier parts of the design), and the power profile being non-constant, but not being reflective of the key (whilst using a single "Point Addition" and "Point Doubling" circuit).

The end result of this change was that the resultant circuit created by the end of the project was not a NISTsecp256r1 curve processor encompassing ECDH, ECDSA, CSPRNG, and HASH on one board, but a variable (variable at time of synthesise, fixed after synthesis) bit length processor encompassing ECDH and ECDSA, relying on external computation and loading of HASH and CSPRNG values, that is also able to be loaded with any set of Elliptic Curve Domain Parameters with a Prime value that has a bit length equal to the variable bit length (with the additional selected restriction that the "order" of the curve's "base point" must also be of, or less than, the same bit length, to avoid padding). The design process was not void of optimisation however, with the design being optimised for bit lengths that have multiplicity by increasing powers of 2, or "division trees" wherein most divisions are done by 2, where branches that are not multiples of 2 are padded up one bit before splitting again.

1.4 Overview of Chapters

To make this document easier to read, an effort has been made to organise the chapters in an arrangement that flows naturally, however, a brief explanation of each of the chapters is given here to provide an overview from which the main focus of each chapter can be extracted, and the order in which they are read can be comfortably rearranged to suit the readers pre-emptive understanding of the topics.

It is worth acknowledging that Chapters 2, 3, and 4, are all preliminary discussions of the content presented in chapters 5 and 6. Chapters 3 and 4 present content that is, unless otherwise referenced, content produced by the author without prompt from the literature review process. Chapters 5 and 6 are content produced entirely by the author.

1.4.1 Chapter 2: Topic Background

To properly address the development of ECDH and ECDSA schemes in a comprehensive manner, it is necessary to consider both the schemes themselves, and their mathematical underpinnings, as separate topics that both provide parts of ECC. To uphold this, it is necessary to cover these topics sequentially in a top-down fashion addressing cryptographic schemes as the top level, down to the mathematical fundamentals that facilitate the operational constituents of the schemes at the bottom level, then follow this by working backwards to demonstrate how the mathematical objects of elliptic curves genuinely facilitate the cryptographic schemes. This chapter will cover the top-down approach, focusing on the topics from the implementation perspective, as such little to no effort will be given to substantiate the underlying mathematics, and the literature will be taken at face value and regarded as true and whole to the understanding of the authors.

1.4.2 Chapter 3: Mathematical Background

This chapter is the other side of chapter 2, serving as the bottom-up approach, in such as that it's focus is entirely on the underlying mathematics of Elliptic Curves, and covers the mathematical topics from the ground up, making effort to provide extensive, comprehensive, and independently verifiable, mathematical proofs of the mathematics primitives employed and taken at face-value in chapter 2. Hence, for a truly 'bottom-up' approach, it may be easier to read chapter 3 before chapter 2, as the starting point of chapter 2 is the assumption of the truth of everything validated in chapter 3.

1.4.3 Chapter 4: Modelling ECC Protocols

This chapter will serve to address how the ECDH and ECDSA schemes discussed in chapter 2 are best modelled for a HDL implementation. It will cover, as an introduction, equality and inequality operations, linear addition and multiplication, linear division (through repeated combinatorial subtraction or use of a 'clocked subtraction comb'), along with the field arithmetic operations (FAO) (modular addition, modular subtraction, modular multiplication (combinatorially and through a 'clocked addition comb' and a 'clocked subtraction comb')) and how they are carried out in the HDL implementation. It will carry through with these to substantiate working models for the Elliptic Curve Group Operation (ECGO), which can then be extended into the Elliptic Curve Group Action (ECGA). Coupled with this will be techniques of modelling field arithmetic inversion, significantly more complicated than the other FAO, and warranting it's own section. The capabilities to perform both the field inversions and the ECGA will then be used to provide a way of modelling ECGA in an 'Affine Context.' The models presented in the chapter involve both combinatorial and clock typed logic cells, with the clocked logic specifying process structures (conditional and loop structures) including initial case selection, case generation, and terminal case selection.

1.4.4 Chapter 5: VHDL Implementation

This chapter presents the framework of VHDL code used to build a simulation model that demonstrates the functionality of the models discussed in 4. It presents models for all of the fundamental components that are used to construct larger circuits, before moving on to the implementation of those larger circuits. Despite that the Modelling chapter terminates with the presentation of the ECGA "Point Multiplication", this is due to the shift in design representation at this stage from a quantitative circuit designed to perform arithmetic to the next stage up, the circuits that utilise the ECGA in a qualitative manner, operating as the processor that simply controls what data is latched to the inputs of the arithmetic circuits, but itself does not contribute to specific arithmetic operations. Due to this, this chapter will include implementations of circuits that did not have models presented for them in the modelling chapter, as they contribute not to the arithmetic components, but to the control processor that operates the final Elliptic Curve Cryptographic Processor. it is worth mentioning here that the vast majority of the chapter could be considered 'appendix' material, but this project reads far more similarly to a design document than a standard thesis.

1.4.5 Chapter 6: VHDL Simulation

This chapter provides a framework in which to test and evaluate the simulation of the VHDL implementations discussed in 5. It attempts to provide reasoning as to why specific tests are necessary to demonstrate that a circuit is working as intended, but also presents and demonstrates the use of arbitrary tests that themselves had no fundamental reason for being used, other than to evaluate the behaviour of the circuit with random inputs, in the intended input range (such that they are not tests intentionally designed to test the behavior of the circuit in conditions it was not intended to handle). The chapter also provides some comparative evaluations on estimated values of Device Utilisation, "post-synthesis", for the modular arithmetic components (including the 'clocked comb' multiplier and divider components used in the modular multiplier).

1.4.6 Chapter 7: Future Work

This chapter provides a discussion on which areas of the design achieve the project goals, and what parts of the design could be extended. It also details potential research into areas that would have facilitated the implementation of the not included HASH and CSPRNG components. Some areas or components of the final processor were intentionally left non-complete, but the choice to implement them has been provided with input commands set aside for their implementation, and the process structure includes calls that handle these inputs, but the calls have been left non-complete, such that they are implemented as skeleton code.

Chapter 2

Topic Background

2.1 Chapter Introduction

This chapter will serve as the introduction to the topic content related to the work undertaken in this project, beginning with an introduction to the structure of Elliptic Curves and how they are defined for the purposes of ECDH and ECDSA. Following this, the DHKE-protocol/ECDH-scheme and DSA-protocol/ECDSA-scheme will be discussed, addressing how they utilise the specific implementations of Elliptic Curves discussed here. This will then be extended to a discussion of the Elliptic Curve Primitives that underpin the ECDH and ECDSA schemes, before elaborating on how these primitives are themselves constructed from primitive field operations.

2.2 Elliptic Curves

In a heavilly mathematical description, Elliptic Curves, for the purposes of ECC, are sets composed of points that are solutions to one of two cubic equations in two variables, where the points are 2-tuples of elements in a Galois Field GF(P^Q), and the selection of which cubic equation is being satisfied by the points is reliant on the value of P and Q. If $Q = 1$ then the Galois Field is termed a Prime Field, else if $P = 2$ and $Q \geq 2$ then the Galois Field is termed a Binary Field [11], else if $P \geq 3$ and $Q \geq 2$ then the Galois Field is termed a Power Field. In the context of topic literature, at an introductory level, this is usually presented by example of a curve over a Prime Field, as it is more intuitively easy to understand. Hence, before continuing the explanation breakdown of the difference between Prime Fields and Binary Fields, it is worth stating that the final section of this chapter will be the worked construction of a small Elliptic Curve over a Prime Field, and may serve as a suitable example before continuing to read, however appears at the end of the chapter as it relies on the calculations presented here.

2.2.1 Elliptic Curves on Prime Fields

Prime Fields are GF(P), Galois Fields of prime order [11]. The minimal representation for a Prime Field, given that all Fields of the same order are isomorphic, can be expressed as the Field of integers under the modulo operation, where arithmetic is performed modulo the prime order, and the elements of the Field are the congruence classes of integers with all modularly equivalent integers [11];

$$\text{GF}(P) \equiv \mathbb{Z}_P = \{\{a + bp | \forall b \in \mathbb{Z}\} | \forall a \in [0, p-1] \cup \mathbb{N}\} \tag{2.1}$$

The cubic equation over which the points of an Elliptic Curve are found when the Curve is defined on a Prime Field is;

$$y^2 = x^3 + ax + b \leftarrow a, b \in \mathbb{Z}_P \tag{2.2}$$

Such that the 'set generation' notation for an Elliptic Curve E over a Prime Field K is (the set of points in $\mathbb{Z}_P^2$ that satisfy this equation);

$$E_K = \left\{(x, y) \in \mathbb{Z}_P^2 | \forall x, y \in \mathbb{Z}_P \rightarrow y^2 = x^3 + ax + b \leftarrow a, b \in \mathbb{Z}_P\right\} \cup \{\infty\} \tag{2.3}$$

To demonstrate visually what an Elliptic Curve over a Prime Field looks like, presented in figure 6.1 is an example of a curve with some parameters for a and b;

2.2.2 Elliptic Curves on Binary Fields

Binary Fields are GF(2^Q), Galois Fields of an order that is some power of 2 [11]. Binary Fields are minimally representable by the Field composed of polynomials $V(x)$, with degree not greater than $(Q-1)$, also stated $(\delta(V) < Q)$, with coefficients from the Field $\mathbb{Z}_2 \equiv \{0, 1\}$.

$$\text{GF}(2^Q) \equiv \left\{\sum_{j=0}^{Q-1} \left(a_j x^j\right) | \forall j \rightarrow a_j \in \{0, 1\}\right\} \tag{2.4}$$

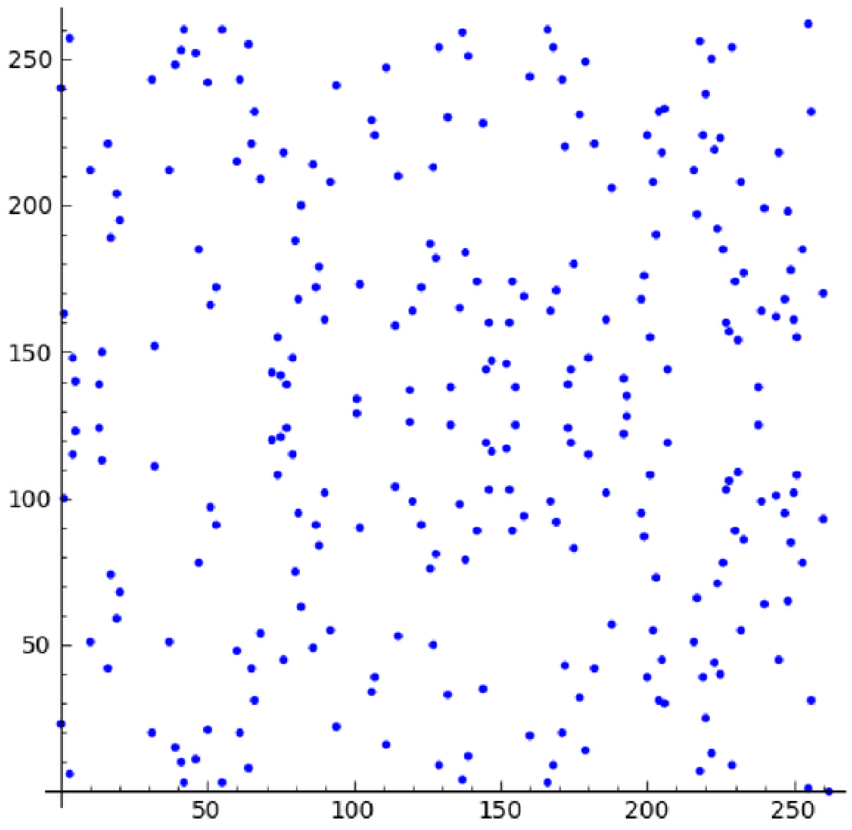
The cubic equation over which the points of an Elliptic Curve are found when the Curve is defined on a Binary Field is;

$$y^2 + axy + by = x^3 + cx^2 + dx + e \leftarrow a, b, c, d, e \in \{0, 1\} \tag{2.5}$$

And the set generation notation for E_K is (the set of points in $\text{GF}(2^Q)^2$ that satisfy this);

$$E_K = \left\{\begin{matrix} (x, y) | \forall x, y \in \text{GF}(2^Q) \\ y^2 + axy + by = x^3 + cx^2 + dx + e \\ a, b, c, d, e \in \{0, 1\} \end{matrix}\right\} \cup \{\infty\} \tag{2.6}$$

Figure 2.1: An example of an Elliptic Curve over a Prime Field, GF(263) [3]



2.3 Elliptic Curve Cryptographic Protocols

The two main 'protocol implementations' in ECC are that of the Elliptic Curve Diffie Hellman (ECDH) and the Elliptic Curve Digital Signature Algorithm (ECDSA) which are schemes specific to application in ECC. These will be individually discussed in their own subsections, however before discussing them, as they both utilise random number generators, it is important to quickly address the significant importance of having 'secure randomness', i.e. not predictable.

2.3.1 Cryptographically Secure Pseudo RNG

Random values serve significant purpose in cryptographic protocols. If a random value is generated once and that same random value is used throughout multiple cycles of a protocol's implementation, or if part of a protocol relies on continuously producing new random values for each new operation of the protocol, it is important that these random values be as genuinely random as is achievable such that values that are produced cannot be determined external to the protocol and used by an adversary to exploit a weakness in the protocol [11]. If the value is to be generated once, we call the state of being genuinely random "verifiably random", and if the value is to be generated multiple times we call this "securely random." ECDSA includes as part of the algorithm a step that produces and uses a new random value each time the algorithm is executed, and it is necessary that this be securely random such that an adversary is not able to predict this value, as being able to predict the value would allow the adversary to recover another user's 'Private Key', which would allow the adversary to 'mimic' the identity of the other person.

Aside from this however, both ECDH and ECDSA begin with either recalling from memory, or generating, an individual's 'Private Key', which, for the protocol to guarantee the information security objectives it is designed to uphold, must be known only to the individual. If the protocols are not set up to include the generation of a 'session key' (a private key unique to a single 'session' of operation of the protocol) then the value stored in memory that the algorithm recalls will still have had had to be produced at random previously to guarantee it is verifiably random. If the protocols are set up to generate a session key each new session, then these session keys must be verifiably random. If neither the constant or session keys are verifiably (or securely) random then an adversary could potentially determine another user's private key, and then mimic their identity.

Hence, all Random Number Generators (RNG) must be 'Securely Random' when they are generating the private key of a user.

2.3.2 Elliptic Curve Diffie Hellman

The ECDH scheme is the ECC variety of the Diffie Hellman Key Exchange (DHKE) protocol. The DHKE protocol is rather simplistic to describe; it begins with a securely randomly generated private key unique to one user, which then undergoes some operation relevant to the cryptographic scheme being employed to generate a corresponding 'Public Key', which is then publicly distributed [11]. Any user A can retrieve another user, B's, public key, at the same time as user B also obtains user A's public key. They then perform the same operation as previously done relevant to the cryptographic scheme using their own private key and the other's public key, such that both obtain a result unique to the combination of their private keys, that they then both have the same result without ever having shared their private keys, and hence have established a "shared secret key", which they can use in the further application of similarly related or unrelated cryptographic schemes, such as utilisation of the shared secret key to generate new domain parameters, adding further security [11].

In the ECC context, the DHKE protocol becomes the ECDH scheme, in which the private key, δ, is some natural number, and the public key is a point in the Elliptic Curve, where the public key Q of user A, Q_A, is determined by performing "Point Multiplication" using their private key δ_A as the 'multiplier' of the Elliptic Curve "Base Point" G, a particular point in the Elliptic Curve chosen such that multiplication of it by some number can be performed to obtain every other point in the Elliptic Curve Group that exists in the cyclic subgroup of the base point, typically as close to the entire group as is possible. Then, their public key is $Q_A = \delta_A * G$. They then share this and obtain other public keys [11]. If they are performing the protocol with another user B, then A knows δ_A, B knows δ_B, and both A and B know Q_A and Q_B. User A then performs $S_A = \delta_A * Q_B$ and user B performs $S_B = \delta_B * Q_A$. This guarantees $S_A = S_B$, that they have evaluated the same point, their unique shared secret key S;

$$S_A = \delta_A * Q_B = \delta_A * \delta_B * G = \delta_B * \delta_A * G = \delta_B * Q_A = S_B \tag{2.7}$$

Hence $S = S_A = S_B$ is now the same point unique to both A and B, and the ECDH protocol then takes the x coordinate of the point S and gives this coordinate as the value of the shared secret that only A and B know.

2.3.3 Elliptic Curve Digital Signature Algorithm

The ECDSA scheme is the ECC variety of the Digital Signature Algorithm (DSA) protocol, which utilises the same construction of public and private keys as outlined in the ECDH. The DSA protocol provides a way for one user, A, to use their private key to 'sign' some data, and any other user to use A's public key to 'verify' that the signature belongs to A. Beyond this, as it must be designed such that the signature cannot simply be reattached to some other data, the signature must be verifiably unique to the combination of the data that was signed, and the private key of user A. The DSA falls into two halves; the signature generation algorithm (SGA) and the signature verification algorithm (SVA) [11].

To provide that the signature is specific to the data being signed, both algorithms must perform some computation or convolution of the data to obtain some value that is, by the computation employed, unique to that data. A typical means of doing this is use of a hashing algorithm, which takes data of some arbitrary size, and results in a piece of data always of a constant size, which provides that the SGA and SVA are able to then rely on an input with a known size. The algorithm then uses a securely randomly generated number in performing an operation relevant to the cryptographic context; in ECC this is point multiplication of the base point G by the RNG value. The result of the operation that utilises the random number and the private key can be extrapolated into more steps; in ECC this involves performing the following operations (written as the entire SGA, although it has so far been described up to step 3) [n is such that $nG = \infty$];

$$\left\{\begin{array}{ll} 1 & e = \text{Initiate HASH}(data) \\ 2 & \text{Securely Randomly Generate } k \rightarrow (0 \leq k \leq n-1),\ \text{calculate } k^{-1} \\ 3 & \text{Compute } kG = (x_1, y_1) \\ 4 & \text{Compute } r = x_1 \text{ Mod } n;\ \text{While } r = 0 \text{ Loop step 2} \\ 5 & \text{Compute } s = k^{-1}\left((e + \delta_A * r)\ \text{ Mod } n\right) \\ 6 & \text{Until } s \neq 0;\ \text{Loop step 2} \\ 7 & \text{Sign with pair } (r, s) \end{array}\right\} \tag{2.8}$$

The SVA, also using the hash value of the data, along with the user's public key, seeks to verify that the signature resulting from the SGA is indeed the result of performing the SGA on the same data and the user's private key. It's algorithm is, presented here [11];

$$\left\{\begin{array}{ll} 1 & e = \text{Initiate HASH}(data) \\ 2 & \text{Calculate } s^{-1} \\ 3 & \text{Compute}:\ \{u_1 = (e * s^{-1} \text{ Mod } n)\}\ \text{ and }\ \{u_2 = (r * s^{-1} \text{ Mod } n)\} \\ 4 & \text{Compute}:\ (u_1 * G) + (u_2 * Q_A) = (x_2, y_2) \\ 5 & \text{Take}:\ v = x_2 \text{ Mod } n \\ 6 & \text{Assert}:\ v = r:\ \text{Accept signature if assertion is true} \end{array}\right\} \tag{2.9}$$

Thus we have both the SGA and SVA for the ECC context, and hence we have the algorithms that constitute the ECDSA protocol.

Hashing Algorithm

The choice of hashing algorithm used in the ECDSA SGA and SVA is insignificant other than it is a cryptographically secure hash function; which does limit the selection. It is desirable to have a hash function that upholds 'pre-image resistant', 'second pre-image resistance', and 'collision resistance.' These are qualities that determine how infeasible it must be to be able to compute, respectively; a message input for a known hash value, a second message with the same hash value as another selected message, two messages that share the same hash value. An easy choice of hashing algorithm is presented in that of the Secure Hash Algorithm family (SHA), although this is down to recommendation by, and having had been guaranteed to uphold the secure hash properties by, NSA FIPS. The SHA family; SHA1, SHA2, SHA3: provides three different incarnations of SHA, for different bit-length hash values, and the choice of using any one of them relies on knowing the desired bit length of the hash value. As the end goal of this project is to produce a working ECDSA scheme on an Elliptic Curve with a Prime with a bit length of 256, the choice of hashing algorithm is down to either SHA2-256, or SHA3-256.

As both SHA2 and SHA3 uphold the requirements of a secure hash, as such as they are both yet to be broken through cryptanalysis, the choice between them is arbitrary. Neither have been implemented in this project, and the final design includes two Hashing registers, a single input register that holds a Hash Digest the bit length of the Prime being used, along with an input scanning channel with strobe IO double the bit length of the Prime being used, with the intention of serving as the input channel for the potential implementation of a Hashing function, however to properly serve as the input for the SHA2 or SHA3 algorithms, the input channel would need a fixed length of 512 bits, which would only be produced by the final design for an implementation that chose a Prime bit length of 256 bits. If anything less is being used then the Hash would need to be computed externally regardless.

2.4 Elliptic Curve Primitives: Point Operations

We must now address the steps in the SGA [step three] and SVA [step four] that utilise "Point Multiplication" (PM); an 'Elliptic Curve Primitive.' This is fundamentally the step in the algorithms that contextualises the algorithms to an ECC setting. Without the PM operation, the algorithms would more closely resemble that of the non-ECC DSA, with only an exponent in step four of the SGA missing. At a scheme level the ECDSA is simply the DSA protocol with the replacement of modular exponentiation by EC PM, thus when we discuss ECC in relation to ECDSA, we are focussed entirely on performing EC PM. Hence, we first discuss PM, before discussing Point Addition and Point Doubling algorithms, despite that they are used by PM.

2.4.1 Point Multiplication

Point Multiplication (PM) on an EC, mathematically speaking, is a 'group action' of an ECG. Speaking contextually for computational applications, PM is an algorithm that uses different cases of the ECGO; both Point Addition (PA) and Point Doubling (PD): to arrive at a value that is effectively "the point added to itself some 'k' times."

$$kQ = \overbrace{Q + ... + Q}^{k \text{ times}} \tag{2.10}$$

Wherein, in the context of the ECGO, the 'addition' of a point and itself is PD (noting that if either the point has $y = 0$ or $Q = \infty$ then $\mathrm{PD}(Q) = \infty$);

$$\mathrm{PD}(Q) = Q + Q \tag{2.11}$$

And the 'addition' of two distinct points is (noting that if $(P = -Q)$ then $\mathrm{PA}(Q, P) = \infty$, if $Q = \infty$ then $\mathrm{PA}(Q, P) = P$, similarly for if $P = \infty$ then $\mathrm{PA}(Q, P) = Q$);

$$\mathrm{PA}(Q, P) = Q + P \tag{2.12}$$

Then PM can be written as an algorithmic function of arguments and expressions;

$$kQ = \mathrm{PM}(k, Q, \mathrm{PA}(\ ), \mathrm{PD}(\ )) \tag{2.13}$$

Before presenting this algorithm, we must observe that an efficient representation for k in binary is due to the implicit binary action of continuously doubling;

$$\left\{ \begin{array}{c} \mathrm{PD}(Q) = Q + Q = 2Q \\ \mathrm{PD}(2Q) = 2Q + 2Q = 4Q \\ \mathrm{PD}(4Q) = 4Q + 4Q = 8Q \\ \mathrm{PD}(2^jQ) = 2^jQ + 2^jQ = 2^{j+1}Q \end{array} \right\} \tag{2.14}$$

Hence, the algorithm makes use of k in binary. The algorithm is then based on continuously doubling the previous result of doubling and including it in the addition if that 'bit' of k is set. To demonstrate this, observe if $k = b_j...b_0$ is the binary representation for k, and $\mathrm{PD}_w(Q) = \mathrm{PD}_{w-1}(\mathrm{PD}(Q))$ [noting $\mathrm{PD}_0(Q) = Q$, and $\mathrm{PD}_1(Q) = Q + Q$], then;

$$kQ = (b_j...b_0)Q = \sum_{h=0}^{j} \left(b_h 2^h Q\right) = \sum_{h=0}^{j} \left(b_h \mathrm{PD}_h(Q)\right) \tag{2.15}$$

In which we now have the equations necessary to write an algorithm for PM;

$$\left\{ \begin{array}{c} \text{Begin} \rightarrow \left\{\text{Input}\left(\text{Integer } k = (b_{\lfloor \mathrm{Log}_2 k \rfloor}...b_0), \text{Point } Q\right)\right\} \\ \text{Do} \rightarrow \{\text{Initialise } \mathrm{PD}_{\mathrm{Temp}} = Q\}, \{\text{Initialise } H = \infty\} \\ \begin{bmatrix} \text{Loop} \\ \forall j \in [0, \lfloor \mathrm{Log}_2 k \rfloor] \\ j_{next} = j + 1 \end{bmatrix} \rightarrow \left\{ \begin{array}{c} \text{If } (b_j = 1) \rightarrow \{H_{\mathrm{Next}} = \mathrm{PA}\left(H, \mathrm{PD}_{\mathrm{Temp}}\right)\} \\ \text{If } (j \neq \lfloor \mathrm{Log}_2 k \rfloor) \rightarrow \{\mathrm{PD}_{\mathrm{Temp:Next}} = \mathrm{PD}(\mathrm{PD}_{\mathrm{Temp}})\} \end{array} \right\} \\ \text{Terminate} \rightarrow \{\text{Return}(H)\} \end{array} \right\} \tag{2.16}$$

2.4.2 Point Addition and Doubling; Affine Plane

Now that we have presented an algorithm for PM, it is necessary to discuss algorithms for PA and PD as they fit into the algorithm. This is first presented in the Affine Plane. First, we observe that for the ECGO (if we assume an ECG set $\bar{E}_K$);

$$\left\{\begin{array}{ll} \text{Closure Statement} & \forall p_j \in \bar{E}_K, * : p_r * p_s \to p_t : \\ \text{Addition : Inverses} & (p_s = -p_r) \to (p_t = \infty) \\ \text{Addition : Input } \infty & (p_s = \infty) \to (p_t = p_r) \\ \text{Addition : Input } \infty & (p_r = \infty) \to (p_t = p_s) \\ \text{Addition} & (p_s \notin \{p_r, -p_r\}) \to (p_t = -((E \cap \mathrm{L}(p_s, p_r)) / \{p_r, p_s\})) \\ \text{Doubling : } (y = 0) & (p_s = p_r, p_s = (x, 0)) \to (p_t = \infty) \\ \text{Doubling : Input } \infty & (p_s = p_r = \infty) \to (p_t = \infty) \\ \text{Doubling} & (p_s = p_r) \to (p_t = -(E \cap \mathrm{L}(\delta(p_r)))) \end{array}\right\} \tag{2.17}$$

In this we see that the only expressions that actually require arithmetic are the unconditional addition and doubling (after having examined the inputs for ∞ or inverses). As this chapter is intended as the topic background and not a mathematical background, from this point we may simply state the equations for PA and PD;

$$\left\{\begin{array}{c} \text{PA} : (x_1, y_1) * (x_2, y_2) \to (x_3, y_3) \\ \left\{x_3 = \left(\frac{y_1 - y_2}{x_1 - x_2}\right)^2 - x_1 - x_2\right\} \\ \left\{y_3 = \left(\frac{y_1 - y_2}{x_1 - x_2}\right)(x_1 - x_3) - y_1\right\} \end{array}\right\} \left\{\begin{array}{c} \text{PD} : (x_1, y_1) * (x_1, y_1) \to (x_3, y_3) \\ \left\{x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1\right\} \\ \left\{y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1\right\} \end{array}\right\} \tag{2.18}$$

2.4.3 Point Addition and Doubling; Jacobian Projection

In the PM algorithm, executing many PA's and PD's, in the Affine equations, requiring a Field Inversion every time. To avoid this, we can extend the Affine equations into the Projective Plane, under some scaling [11]. Here we present the algorithms for the third projective coordinate from the two inputs; firstly PA, and secondly PD [11]

$$\left\{\begin{array}{c} X_3 = \left((Z_2^3 Y_1 - Z_1^3 Y_2)^2 - (X_1 Z_2^2 + X_2 Z_1^2)(Z_2^2 X_1 - Z_1^2 X_2)^2\right) \\ Y_3 = \left((Z_2^3 Y_1 - Z_1^3 Y_2)\left(X_1 Z_2^2 (Z_2^2 X_1 - Z_1^2 X_2)^2 - X_3\right) - Y_1 Z_2^3 (Z_2^2 X_1 - Z_1^2 X_2)^3\right) \\ Z_3 = (Z_1 Z_2 (Z_2^2 X_1 - Z_1^2 X_2)) \end{array}\right\} \tag{2.19}$$

$$\left\{\begin{array}{c} X_3 = \left((3X_1^2 + aZ_1^4)^2 - 8Y_1^2 X_1\right) \\ Y_3 = \left((3X_1^2 + aZ_1^4)^1 (4Y_1^2 X_1 - X_3)^1 - 8Y_1^4\right) \\ Z_3 = (2Y_1 Z_1) \end{array}\right\} \tag{2.20}$$

In which the projection is;

$$\left\{[X_j : Y_j : Z_j] = \left(\left(\frac{X_j}{Z_j^2}\right), \left(\frac{Y_j}{Z_j^3}\right)\right) = (x_j, y_j)\right\} \tag{2.21}$$

2.5 Field Arithmetic

Field arithmetic is the operation of addition, "subtraction", multiplication and "division" with elements of a 'Field', where subtraction and division are 'inverse operators' and equivalently subtraction is effectively addition of the additive inverse, and division is multiplication by a multiplicative inverse. This distinction is necessary, as the concept of 'division' does extend naturally to arithmetic over finite Fields, but not necessarily intuitively [11]. Finite Field Arithmetic, when the order of the Field is prime, is known as Modular Arithmetic. Despite that the mathematical background chapter covers both Prime and Prime power ordered Fields, as this project is aimed at implementation over a Prime Field, we will only present Modular Arithmetic here.

2.5.1 Modular Arithmetic: Addition, Multiplication

Addition, "subtraction", and multiplication extend naturally and intuitively into Modular Arithmetic. We begin by stating what a 'Modulo' Field is; the integers from 0 up to some $p-1$ for some prime p (the 'integer' values are representations for 'congruence classes').

$$\{\mathbb{Z}_p = \{\{a + bp | a, b \in \mathbb{Z}\} \,|\forall a \in (0, p-1)\}\} \tag{2.22}$$

Then the 'minimal representative' for some element $c \in \mathbb{Z}_P$ is an element $A \in [0, p-1] \cup \mathbb{N}$ such that $c = a + bp$. Then, the extension of addition, subtraction, and multiplication, into Modular Arithmetic is simply to find the standard result, and then determine the result's 'congruence class', and the 'minimal representative' of that congruence class. In terms of notation, these are written;

$$\left\{ ((a \bmod p) \begin{bmatrix} + \\ - \\ * \end{bmatrix} (b \bmod p)) \bmod p = (a \begin{bmatrix} + \\ - \\ * \end{bmatrix} b) \bmod p = c | \exists d \rightarrow c = (a \begin{bmatrix} + \\ - \\ * \end{bmatrix} b) + dp \right\} \tag{2.23}$$

2.5.2 Modular Arithmetic: Division; Modular Inversion

Division does not extend as intuitively as the other operations, because the result must be a member of a congruence class, defined by integer representation. Hence, to define Modular Inversion, we must recall that multiplicative inversion is defined as an element b that multiplies a and the result is 1, in which we write $b = a^{-1}$, and the equation for an element and it's inverse is $aa^{-1} = 1$. In modular arithmetic this is written;

$$\{(a * b) \bmod p = 1 \bmod p\} \rightarrow \left\{b \bmod p \equiv a^{-1} \bmod p\right\} \tag{2.24}$$

In which the inverse of an element is another integer that multiplies the element and the resultant congruence class is represented by 1. Finding an elements inverse is rather intensive, and the Extended Euclidean Algorithm is presented in the Mathematical Background.

2.6 A Worked Example over $GF(17)$

Here we present the construction of and elaboration of a worked example of a small Elliptic Curve Group, defined over a prime of 17. If we take equation 2.3 and parametrise it, with $P = 17$, and $a = b = 2$, then we end up with the collection of values;

$$E_K = \left\{(x, y) \in \mathbb{Z}_{17}^2 | \forall x, y \in \mathbb{Z}_{17} \rightarrow y^2 = x^3 + 2x + 2 \leftarrow a, b \in \mathbb{Z}_{17}\right\} \cup \{\infty\} \tag{2.25}$$

We use this to construct a table of the resultant values up to the Prime, showing the values, their inverses, the values of y^2 and $x^3 + 2x + 2$;

Table 2.1: Table of Values for Prime 17, a = b = 2

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
n^{-1}	-	1	9	6	13	7	3	5	15	2	12	14	10	4	11	8	16
n^2	0	1	4	9	16	8	2	15	13	13	15	2	8	16	9	4	1
$n^3 + an + b$	2	5	14	1	6	1	9	2	3	1	2	12	3	15	3	7	16

The numbers appearing in both the last and second last rows are coloured green to indicate that here we have values of x and y that satisfy the equation. For example, when $y = 3$, we have (y^2 mod $17 = 9$), but we also have (y^2 mod $17 = 9$) when $y = 14$, and we also have ($x^3 + 2x + 2$ mod $17 = 9$) when $x = 6$, which gives us the points $(6, 3)$ and $(6, 14)$. We can go ahead and match up the rest of the pair-able values. We know before hand however that each modular value that is pair-able in the last two rows will produce an amount of points on the curve equal to the amount of times it appears in the last row, multiplied by the amount of times it appears in the second last row, as each time we have the value j appearing in ($x^3 + 2x + 2$ mod $17 = j$) we have all the points in (y^2 mod $17 = j$) that form a point with it. We can however also simply state that we have an amount of points equal to twice the amount of pair-able elements in the last row. This is true for any elliptic curve computation unless there is a value of x such that ($x^3 + ax + b$ mod $p = 0$), in which case we would only count that value once, not twice! Hence we can predict that for this, there are going to be 18 points, not including the point at infinity. If we include the point at infinity then this group has 19 points in it, necessarily making it a cyclic group, meaning any selection of "base point" will act as a generator for the entire group! The points on the curve then, are;

$\infty, (0, 6), (0, 11), (3, 1), (3, 16), (5, 1), (5, 16), (6, 3), (6, 14), (7, 6), (7, 11)$
$(9, 1), (9, 16), (10, 6), (10, 11), (13, 7), (13, 10), (16, 4), (16, 13)$

If we pick the base point to be the point $(7, 6)$ (this selection is entirely arbitrary, any point will work) then we could write these out in order that they are as mutliples of this point. But first lets look at doubling the point, and adding its double to find its triple, before meaninglessly stating the order of multiplicities.

$$\left\{\begin{matrix} \text{PD} : (7,7) * (7,6) \rightarrow (5,16) \\ \left\{x_3 = \left(\frac{3*7^2+2}{2*6}\right)^2 - 2*7 = \left(\frac{149}{12}\right)^2 - 14 = (149*10)^2 - 14 = 2 - 14 = -12 = 5\right\} \\ \left\{y_3 = \left(\frac{3*7^2+2}{2*6}\right)(7 - x_3) - 6 = (1490)(7-5) - 6 = (22) - 6 = 5 - 6 = -1 = 16\right\} \end{matrix}\right\} \tag{2.26}$$

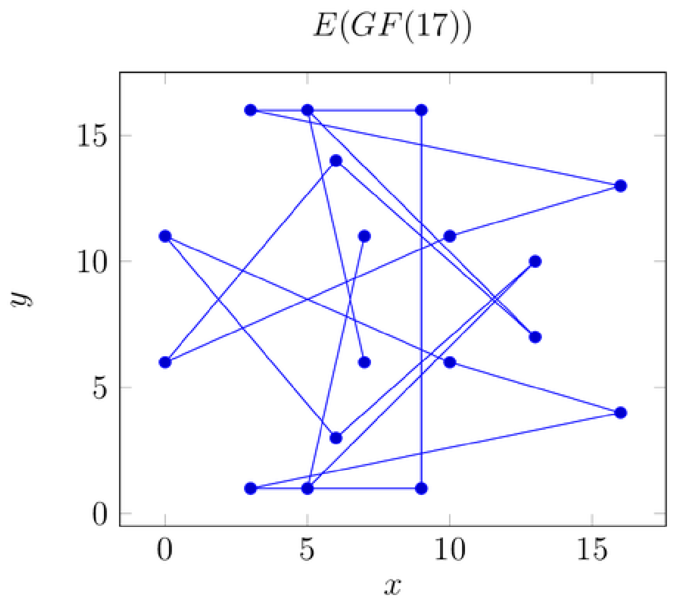
$$\left\{\begin{matrix} \text{PA} : (7,6) * (5,16) \rightarrow (13,7) \\ \left\{x_3 = \left(\frac{6-16}{7-5}\right)^2 - 7 - 5 = \left(\frac{7}{2}\right)^2 + 5 = (63)^2 + 5 = 8 + 5 = 13\right\} \\ \left\{y_3 = \left(\frac{6-16}{7-5}\right)(7 - x_3) - 6 = \left(\frac{7}{2}\right)(7-13) - 6 = (63)(11) - 6 = 13 - 6 = 7\right\} \end{matrix}\right\} \tag{2.27}$$

We can now state the order of the points (with $0Q = 19Q = \infty$);

$$\left\{\begin{matrix} k & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ Q & (7,6) & (5,16) & (13,7) & (6,14) & (0,6) & (10,11) & (16,13) & (3,16) & (9,16) \\ k & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ Q & (9,1) & (3,1) & (16,4) & (10,6) & (0,11) & (6,3) & (13,10) & (5,1) & (7,11) \end{matrix}\right\} \tag{2.28}$$

Which we can also show in a plot of the points and the cyclic multiplicity of the points;

Figure 2.2: An example of an Elliptic Curve over a Prime Field, GF(17)



Chapter 3

Mathematical Background

3.1 Chapter Introduction

This chapter will begin with a mention of the field theory relevant to defining Elliptic Curves "over a field", and performing operations with them. This is then extended into defining Elliptic Curves over general Fields, and how this leads to the construction of an Elliptic Curve Group (ECG). Following is a discussion on the 'group operation' of an ECG and how to contextualise the operation to adapt it for a graphical interpretation, all in an 'Affine Context.' A new topic is then introduced, that of the 'Projective Plane', which has both it's own theory, and a novel purpose, as such as it is not 'mathematically necessary' to validate or perform the operations of an ECG, however it is a required part of the theory of a modern ECG system due to it facilitating faster implementations of ECG than those that do not rely on it. Due to this, it will be discussed after having first completely discussing the 'Affine Context', as it introduces the 'Projective Context.'

3.2 Field Theory

A Field, K, is a set $\bar{K}$, together with two binary operations; $+$ and $*$, which obey the Field Axioms; Closure, Associativity, (unique) Identity, (unique) Inverse, and Commutativity, on both operators, along with Distribution of $*$ over $+$, with the caveat that $+$ operates on $\bar{K}$, while $*$ operates on $\bar{K}/\{0\}$. [26]

$$\left\{K = \left\{\bar{K}, +, *\right\}\right\} \tag{3.1}$$

Aside from the axiom of Distribution, this can be more succinctly written as the "Abelian Group" of $+$ on $\bar{K}$;

$$\left\{\bar{K}, +\right\} \tag{3.2}$$

Along with the "Abelian Group" of $*$ on $\bar{K}/\{0\}$.

$$\left\{\bar{K}/\left\{0\right\}, *\right\} \tag{3.3}$$

3.2.1 Finite Fields

Fields on a set K are said to have 'order' $|\bar{K}|$, (order $\equiv$ cardinality) which can be either infinite or finite [26]. The rationals, $\mathbb{Q}$, form the 'smallest' (explicitly; the 'least ordered' cardinal infinity) infinite Field (explicitly; as a representation of an isomorphism class), and is fundamental to the construction of 'larger' infinite Fields. Finite Fields are Fields whose order is finite, a restriction significant enough that all Finite Fields are easily classifiable (down to isomorphism classes) simply by knowing the order [26]. The order of a Finite Field is described here; the order of a Finite Field is such that it is some prime, raised to some power. [26]

$$\left\{\left|\bar{K}\right| = p^q, p \in \text{Prime}, q \in \mathbb{N}\right\} \rightarrow \{\text{Char}(K) = p\} \tag{3.4}$$

The Finite Field, or Galois Field ($GF(p)$), with an order that is some prime raised to unitary power is the isomorphism class represented by the Integers 'modulo' the prime, such that the elements of the Field are the p unique 'modular congruence classes' under the operation of 'modulo' the prime. [26]

$$\{GF\left(p\right) \equiv \mathbb{Z}_p\}\left\{\mathbb{Z}_p = \{\{a + bp | a, b \in \mathbb{Z}\} \,|\forall a \in (0, p-1)\}\right\} \tag{3.5}$$

The Finite Field, or Galois Field ($GF(p^q)$), with an order that is some prime raised to a non unitary power is the isomorphism class represented by the Integers 'modulo' the prime, under $(q-1)$ Field Extensions on x, such that the elements of the Field are polynomials in one variable, of degree $(q-1)$, whose coefficients are elements of $\mathbb{Z}_p$, where the elements of $\mathbb{Z}_p$ are similarly modular congruence classes. [26]

$$\left\{GF\left(p^q\right) \equiv \mathbb{Z}_p\left[x\right]^{q-1}\right\}\left\{\mathbb{Z}_p\left[x\right]^{q-1} = \left\{\left\{\sum_{j=0}^{q-1}\left(a_j x^j\right)\right\} |a_j \in \mathbb{Z}_p\right\}\right\} \tag{3.6}$$

3.2.2 Arithmetic over a Finite Field

The operations of addition, subtraction and multiplication easily extend to Finite Fields, as operations performed under 'modular arithmetic.' Modular arithmetic reduces to performing an operation to obtain a numerical result, and then determining the 'least representative' of the congruence class that the result is an element of, where the least representative is 'a', such that; $\{a \in \{b + cp|\forall c \in \mathbb{Z}\} \cup (0, p-1)\}$ where b was the standard numerical result [24]. A more succinct way of writing this congruence relation is;

$$\{a \equiv b \bmod p\} \equiv \{a \bmod p = b \bmod p\} \equiv \{a \in \{b + cp|\forall c \in \mathbb{Z}\}\} \tag{3.7}$$

With this, it is easy to express modular addition and subtraction on Finite Fields;

$$\left\{\left(\left(\sum_{j=0}^{q-1} x^j\left(a_j \bmod p\right)\right) \pm \left(\sum_{j=0}^{q-1} x^j\left(b_j \bmod p\right)\right)\right) \bmod p = \left(\sum_{j=0}^{q-1} x^j\left(a_j \pm b_j\right) \bmod p\right) |a, b \in \mathbb{Z}_p\right\} \tag{3.8}$$

In the case of Prime Fields we can simplify this (and also state it for modular multiplication simultaneously); [24]

$$\left\{((a \bmod p)\begin{bmatrix}+\\-\\*\end{bmatrix}(b \bmod p)) \bmod p = (a\begin{bmatrix}+\\-\\*\end{bmatrix}b) \bmod p | a, b \in \mathbb{Z}_p\right\} \tag{3.9}$$

Modular multiplication in a non prime ordered Field is similarly; [24]

$$\left\{\left(\left(\sum_{j=0}^{q-1} x^j (a_j \bmod p)\right) * \left(\sum_{j=0}^{q-1} x^j (b_j \bmod p)\right)\right) \bmod p = \left(\sum_{j=0}^{q-1}\sum_{h=0}^{q-1} x^{j+h}(a_j * b_h) \bmod p\right)\right\} \tag{3.10}$$

$$\left\{\left(\sum_{j=0}^{q-1}\sum_{h=0}^{q-1} x^{j+h}(a_j * b_h) \bmod p\right) = \left(\sum_{j=0}^{2q-2}\left(x^j\left(\sum_{j=g+h}^{\forall g,h\in\mathbb{N}}(a_g * b_h)\right) \bmod p\right)\right)\right\} \tag{3.11}$$

The (Multiplicative) Inverse Element

The inverse of an element, a, in a Finite Field is an element, a^{-1}, such that the modular multiplication of an element by its inverse is equal to the modular congruence class of 1;

$$\{(a * b) \bmod p = 1 \bmod p\} \to \left\{b \bmod p \equiv a^{-1} \bmod p\right\} \tag{3.12}$$

Although the statement of the modular multiplicative inverse is easy is formulate, finding the inverse of an arbitrary element requires significant work. Relevant to the inverse element is the statement of "Fermat's Little Theorem", and the Extended Euclidean Algorithm. Given the computational dissimilarity between finding the Inverse over Prime Galois Fields versus over Prime Power Galois Fields, this section will only address Prime Fields (although Prime Power Fields can be derived through simply replacing integer arithmetic with the natural projection in polynomial arithmetic, with the measure suitably replaced by polynomial degree). Thus, the statement of Fermat's Little Theorem is; [19]

$$\left\{\begin{matrix}\forall p \in \text{Primes}\\ \forall a \in \mathbb{Z}\end{matrix} \to (a^p \equiv a \bmod p) \to \left(a^{p-1} \equiv aa^{p-2} \equiv 1 \bmod p\right) \to \left(a^{p-2} \equiv a^{-1}\right)\right\} \tag{3.13}$$

Which provides a functional expression for an elements multiplicative inverse over some Modulo Prime. However, if P is sufficiently large, this calculation is expensive, which leads to a need to address the Extended Euclidean Algorithm, an iterative simultaneous assertion of the Euclidean Algorithm and Bezout's Identity; [13]

$$\{\forall a, b \in \mathbb{Z}, \exists x, y \in \mathbb{Z} \to ax + by = \gcd(a, b)\} \tag{3.14}$$

In the context of Modular Multiplicative Inverse on Prime Fields, we can assert; [13]

$$\{\forall a, b \in \mathbb{Z}, \text{iff } \{\exists x, y \in \mathbb{Z} \to ax + by = 1\} \to a, b \text{ Coprime}\} \tag{3.15}$$

But observe that the statement $(ax + by = 1)$ in modular arithmetic, if we select that one of the choices for a or b, say a, is some prime, and the other, say b, is some value less than that prime, reduces to $(by \equiv 1)$.

Hence if we choose some prime a, and some value less than the prime b (a value substantiating that it is the 'least representative' for a congruence class), we can say that the statement $(by \equiv 1)$, which is equivalent to saying $(y \equiv b^{-1})$, reduces to finding the coefficients of Bezout's Identity for $(ax + by = 1)$, guaranteed to exist given the coprimality of a and b [13]. Hence, the Extended Euclidean Algorithm, as it applies to modular multiplicative inversion, is finding these integer solution x and y, or rather, as the coefficient of a is not important, simply finding the value of y. [11]

$$\left\{\begin{array}{c} \text{Begin} \rightarrow \{\text{Input}\,(\text{Prime } a, \text{Value } b,)\} \\ \text{Do} \rightarrow \{u \leftarrow b, v \leftarrow a, x \leftarrow 1, y \leftarrow 0\} \\ \text{Loop}\,[\text{While } \ (u \neq 1)] \rightarrow \left\{\begin{array}{c} q_{var} \leftarrow \left(\frac{(v-(v \bmod u))}{u}\right) \\ (r_{var} \leftarrow v - qu)\,, (z_{var} \leftarrow y - qx) \\ v \leftarrow u \\ y \leftarrow x \\ u \leftarrow r_{var} \\ x \leftarrow z_{var} \end{array}\right\} \\ \text{Terminate} \rightarrow \{\text{Return}(x)\} \end{array}\right\} \tag{3.16}$$

However, given the potentially computationally difficult size of x in the algorithm, it is appropriate to form a 'binary' version of the Extended Euclidean Algorithm, where all divisions are done by 2 on even numbers, such that they are simple left right operations;

$$\left\{\begin{array}{c} \text{Begin} \rightarrow \{\text{Input}\,(\text{Prime } p, \text{Value } b,)\} \\ \text{Do} \rightarrow \{u \leftarrow b, v \leftarrow p, x \leftarrow 1, y \leftarrow 0\} \\ \begin{array}{c} \text{Loop} \rightarrow \\ \left[\begin{array}{c} \text{While} \\ \text{OR} \left\{\begin{array}{c} (u \neq 1) \\ (v \neq 1) \end{array}\right\} \end{array}\right] \end{array} \left\{\begin{array}{cc} \begin{array}{c} \text{If} \\ \left(\text{OR} \left\{\begin{array}{c} (u \bmod 2 = 0) \\ (v \bmod 2 = 0) \end{array}\right\}\right) \end{array} & \left\{\begin{array}{cc} \begin{array}{c} \text{If} \\ (u \bmod 2 = 0) \end{array} & \left(u \leftarrow \frac{u}{2}, \left\{\begin{array}{cc} \begin{array}{c} \text{If} \\ (x \bmod 2 = 0) \end{array} & x \leftarrow \frac{x}{2} \\ \text{Else} & x \leftarrow \frac{(x+p)}{2} \end{array}\right\}\right) \\ \begin{array}{c} \text{If} \\ (v \bmod 2 = 0) \end{array} & \left(v \leftarrow \frac{v}{2}, \left\{\begin{array}{cc} \begin{array}{c} \text{If} \\ (y \bmod 2 = 0) \end{array} & y \leftarrow \frac{y}{2} \\ \text{Else} & y \leftarrow \frac{(y+p)}{2} \end{array}\right\}\right) \end{array}\right\} \\ \text{Else} & \left\{\begin{array}{cc} \text{If } (u \geqslant v) & \left(u \leftarrow u - v, \left\{\begin{array}{cc} \left(\begin{array}{c} \text{If} \\ (x > y) \end{array}\right) & (x \leftarrow x - y) \\ \text{Else} & (x \leftarrow x + p - y) \end{array}\right\}\right) \\ \text{Else} & \left(v \leftarrow v - u, \left\{\begin{array}{cc} \left(\begin{array}{c} \text{If} \\ (y > x) \end{array}\right) & (y \leftarrow y - x) \\ \text{Else} & (y \leftarrow y + p - x) \end{array}\right\}\right) \end{array}\right\} \end{array}\right\} \\ \text{Terminate} \rightarrow \left\{\begin{array}{cc} \text{If } (u = 1) & \text{Return}(x) \\ \text{Else (when } (v = 1)) & \text{Return}(y) \end{array}\right\} \end{array}\right\} \tag{3.17}$$

This algorithm given here is a one-step loop-unrolled version of the algorithm given in [12], itself taken from the Binary Extended Euclidean Algorithm presented in [11]. Presenting this algorithm concludes the content necessary for Finite Field Arithmetic, with Inversion in Prime Fields.

3.3 Elliptic Curve Equations: Weierstrassian Curves

Elliptic Curves are a variety of objects, with polynomial, geometric, and group structure representations, of which the particular degree of restrictedness on the representation used being inherited from the context in which they are used. In the most general polynomial sense, an Elliptic Curve is a Cubic Plane Curve whose Kernel defines a Unitary Algebra; that is, it is a set composed of the solutions to a Cubic Plane Curve, with the added restriction that the curves be non-self-intersecting. A general Cubic Plane Curve;

$$\left\{a + bx + cy + dx^2 + ey^2 + fxy + gx^3 + hy^3 + ixy^2 + jyx^2 = 0\right\} \tag{3.18}$$

Reduces to the Elliptic Curve Equation (with different a, b); [11]

$$\left\{y^2 = x^3 + ax + b | \left(4a^3 + 27b^2\right) \neq 0\right\} \tag{3.19}$$

This definition has very little restrictedness, as, as far as it stands in this expression, it is defined for any value of x and y in the reals, $(x, y \in \mathbb{R})$, and over the reals, all real solutions are part of the set of points that form the curve. The first level of restriction would be to limit the points on the curve to only rational points, $(x, y \in \mathbb{Q})$, or only integer points, $(x, y \in \mathbb{Z})$, which leads to a more enhanced definition, ; $(x, y \in K)$. [19]

$$\left\{(x, y) \left| y^2 = x^3 + ax + b \right| x, y \in \bar{K}\right\} \tag{3.20}$$

In which we are saying that an Elliptic Curve is the set of points that are solutions of the Elliptic Curve Equations, that are points in a Field K, which again, leads to a more enhanced representation, in which we say an Elliptic Curve E_K is an Elliptic Curve E defined over some Field K, such that the coefficients a and b are elements of K, and E_K is the set of all points that are solutions to E, and composed of elements that are themselves elements of K; [11] [19] [13]

$$\left\{\forall a, b \in \bar{K}\right\} \rightarrow E_K(a, b) = \left\{(x, y) \left| y^2 = x^3 + ax + b \right| x, y \in \bar{K}\right\} \tag{3.21}$$

Although we would typically only write E_K, without the specification in the set generation of E_K that a and b are elements in K, as an Elliptic Curve is fixed for a particular choice of a and b, we can say, to avoid later confusion; [11]

$$E_K = \left\{(x, y) \left| \exists\, a, b \in \bar{K} \right| y^2 = x^3 + ax + b | x, y \in \bar{K}\right\} \tag{3.22}$$

Noting another convention of writing E_K is $E(K)$, although typically when we say E_K we are talking about a particular curve in which we have chosen a and b, where as when we say $E(K)$ we are typically referring to the set of all curves defined on K; [11] [19]

$$E(K) = \left\{\left\{(x, y) | y^2 = x^3 + ax + b | x, y \in \bar{K}\right\} | \forall a, b \in \bar{K}\right\} \tag{3.23}$$

So now we have a way of expressing an Elliptic Curve E_K for a Field K, but our initial expression for the curve was for one defined over a Field whose order was infinite. If we wish to generalise the Elliptic Curve to be definable over Finite Fields, there is not much effort required. We note that, as mentioned in the section on Finite Fields, that a Field K of order $|\bar{K}| = p^q$ has 'characteristic', or "Char()", $\text{Char}(K) = p$. We mentioned above, that we may restrict the points on the curve to be from some general Field other than $\mathbb{R}$ or $\mathbb{Q}$, and offered $\mathbb{Z}$ as an alternative, but $\mathbb{Z}$ is not a Field, so we cannot generalise to this. However, $\mathbb{Z}_p$ is a Field, so we can generalise to this. Note that $\text{Char}(\mathbb{Z}_p) = p$. The current expression for an Elliptic Curve holds when the characteristic is a prime greater than 3, but it takes two special forms when the characteristic is either 3, or 2 [11] (despite that $\mathbb{Z}_1$ arithmetic over $\mathbb{R}$ defines a topology [19], it does not define a Field [13]). For the following, we assert that $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t \in \bar{K}\}$. Then we are ready to state the three "Weierstrass Elliptic Curves." [11]

$$\begin{array}{rc} \text{Cubic} & \{a + bx + cy + dx^2 + ey^2 + fxy + gx^3 + hy^3 + ixy^2 + jyx^2 = 0\} \\ \text{Weierstrassian Curve on } \mathbb{Z}_2 & \{y^2 + kxy + ly = x^3 + mx^2 + nx + o\} \\ \text{Weierstrassian Curve on } \mathbb{Z}_3 & \{y^2 = 4x^3 + px^2 + 2qx + r\} \\ \text{Weierstrassian Curve on } \mathbb{Z}_k \ \forall k > 3 & \{y^2 = x^3 - sx - t\} \end{array} \tag{3.24}$$

We now have the three Elliptic Curve Equations, according to our choice of Field!

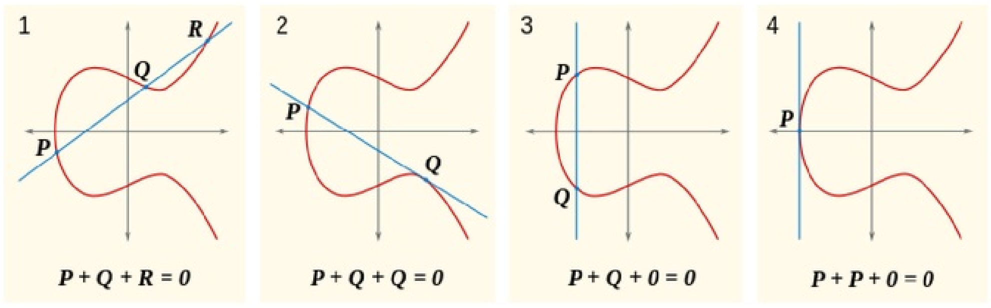
3.4 Elliptic Curve Group: Elliptic Curve over a Field

We have so far explored the polynomial representation of an Elliptic Curve. The 'group' representation of an Elliptic Curve is another way of representing an Elliptic Curve, in the context of Finite Groups. Indeed it is the reason why we began by discussing Finite Fields before discussing Elliptic Curves, as the representation we are interested in for the context of Elliptic Curve Cryptography is the Elliptic Curve Group (ECG) structure, a Finite Group defined over the appropriate Weierstrass Elliptic Curve polynomial, for some choice of Finite Field. Hence, to appropriately discuss the topic of ECGs, it was necessary to define Finite Field arithmetic, and Weierstrass Curves. We are now ready to define the underlying set of an ECG, before using this set to establish the appropriate group operator. In the previous section we defined E_K as the set of all points in K^2 that were solutions to the curve E, where E had constant coefficients that were themselves elements of K. We use a modified form of this to define the set of elements of an ECG;

$$\bar{E_K} = \{(x, y) \mid (y^2 \equiv (x^3 + ax + b) \bmod p), (x, y \in \mathbb{Z}_p)\} \cup \{\infty\} \tag{3.25}$$

Such that the set of an ECG is composed of points in K^2 that are solutions to the 'modular expression' of E, modulo the prime P that defines the Field $K = \mathbb{Z} \equiv GF(P)$. This, in union with a 'point at infinity' (the group identity element) is the complete set of elements of an ECGs set [11]. Now that we have defined the set, we must define the Elliptic Curve Group Operator (ECGO). Before defining this, it is worth noting the geometric interpretation, and how it relates to defining the ECGO.

Figure 3.1: Geometric Representation of the Group Operator [6]



Geometrically, the Group Operator takes two points on E, constructs the line between them, determines the third point on the line that is also on E, and returns the 'inverse' of this point. There are several caveats to this, and they are best explained with reference to the figure presented here. In this figure we see the different cases for how to interpret the group operator. The image labelled '3' demonstrates something of fundamental importance, that of how to interpret a point's 'inverse'. The inverse of a point $(x, y) \in \bar{K}$, $\mathrm{Char}(\bar{K}) = p$ is the point $(x, -y) = (x, p - y)$. This is shown in image '3' as the points P and Q as being each other's inverse. By direct consequence of the group structure, the group operator acting on a point (x, y) and its inverse $(x, p-y)$ will result in the group identity ∞. The left most image demonstrates the three points of intersection on a line and E, where-in, if we were to perform the Group Operation on any two of the three points, the result would be the 'inverse' of the third point. The accompanying equation describes this as the three points adding to the group identity, as the result of any two is the inverse of the one remaining, hence this reduces to a point and its inverse, which then results in the group identity. Image '2' demonstrates the interpretation of the Group Operator acting on a point and itself. If this is the case, the line is constructed by localised differentiation of E, and the line constructed from this derivative then intersects E at another point, again, the result of the Operator on a point and itself is the inverse of the result of intersection of the line with E. Lastly is image '4' which combines '2' and '3' to demonstrates that the local derivative of a point with $y = 0$ constructs a line that only meets the point at infinity, and hence, any point with $y \in \{0, p\}$ when the operator acts on this point and itself, the result is the identity.

We now have the definition for the ECGs underlying set and a description for the operator that we can use to summarise an ECG! [11]

$$E_K = \left\{ \begin{array}{c} \bar{E_K} = \{(x, y) \mid (y^2 \equiv (x^3 + ax + b) \bmod p), (x, y \in \mathbb{Z}_p)\} \cup \{\infty\} \\ \left\{ \forall p_j \in \bar{E_K}, * : p_r * p_s \to p_t : \begin{array}{c} (p_s = -p_r) \to (p_t = \infty) \\ (p_s = p_r) \to (p_t = -(E \cap \text{Ł}(\delta(p_r)))) \\ (p_s = p_r, p_s = (x, 0)) \to (p_t = \infty) \\ (p_s \notin \{p_r, -p_r\}) \to (p_t = -((E \cap \text{Ł}(p_s, p_r)) / \{p_r, p_s\})) \end{array} \right\} \end{array} \right\} \tag{3.26}$$

3.4.1 Elliptic Curve Group Operator on Affine Points

Now that we have a succinct expression for the ECG E_K we need to demonstrate that the group operator that we have so far described works. Before doing this we distinguish the cases from one another. There is no specific name for the group operator when performed on a point and its inverse, or on a point and itself when that point had a y value, $y \equiv 0$ Mod P, but the other two cases do have specific names. When we are performing the group operation on a point and itself (other than cases with $y \equiv 0$ Mod P), we call the group operator "Point Doubling" (PD), and when we are performing the group operation on a point and some other point that is neither the point itself or its inverse, we refer to the group operation as "Point Addition" (PA). To sustain the case that the geometric interpretation of the Elliptic Curve E_K defines a group operator, we must prove that PA and PD both have closure over the Field K, as the other group properties directly relate to the closure of the operator over the Elliptic Curve E on K, hence we need only demonstrate closure. To do this, we simply need to construct a "closed equipped action" on the set E_K that validates the geometric description presented above. [26]

The statement of PA and PD as closed equipped actions is; PA : $(x_1, y_1) * (x_2, y_2) \rightarrow (x_3, y_3)$ and PD : $(x_1, y_1) * (x_1, y_1) \rightarrow (x_3, y_3)$. To construct PA and PD as closed equipped actions over the geometry of E [24], we begin by observing that PA and PD utilise the intersection of points on E and the line formed by the points under PA/point under PD. Hence, we begin by stating this intersection;

$$\bigcap \begin{matrix} \{y^2 = x^3 + ax + b\} \\ \{y = \lambda x + y_1 - \lambda x_1 = y_1 + \lambda(x - x_1)\} \end{matrix} \equiv \left\{x^3 + ax + b = (y_1 + \lambda(x - x_1))^2\right\} \tag{3.27}$$

We now have an expression for the intersection of E and some line with gradient λ, and now we perform arithmetic to arrive at;

$$\left\{\begin{aligned} x^3 + ax + b &= (\lambda x + y_1 - \lambda x_1)^2 \\ &= y_1^2 + 2y_1\lambda\,(x - x_1) + \lambda^2\,(x - x_1)^2 \\ &= y_1^2 + 2y_1\lambda x - 2y_1\lambda x_1 + \lambda^2 x^2 - \lambda^2 2xx_1 + \lambda^2 x_1^2 \\ 0 &= x^3 - \lambda^2 x^2 + x(a - 2y_1\lambda + \lambda^2 2x_1) + (b - \lambda^2 x_1^2 - y_1^2 + 2y_1\lambda x_1) \end{aligned}\right\} \tag{3.28}$$

When compared with a general cubic with roots x_1, x_2, x_3;

$$\left\{0 = (x - x_1)(x - x_2)(x - x_3) = x^3 - (x_1 + x_2 + x_3)x^2 + x(x_1x_2 + x_1x_3 + x_2x_3) - x_1x_2x_3\right\} \tag{3.29}$$

We now have a general cubic with three roots and the cubic expression for the intersection of E and some line with gradient λ, of which we can equate coefficients, and arrive at an expression for the x_3 result of PA and PD on points p_1 and p_2 with x_1 and x_2, reliant on some λ.

$$\left\{\lambda^2 = x_1 + x_2 + x_3\right\} \rightarrow \left\{x_3 = \lambda^2 - x_1 - x_2\right\} \tag{3.30}$$

We can then insert this result for x_3 into the expression of the line, to obtain y_3;

$$\{-y_3 = \lambda x_3 + y_1 - \lambda x_1 = y_1 + \lambda(x_3 - x_1)\} \rightarrow \{y_3 = \lambda(x_1 - x_3) - y_1\} \tag{3.31}$$

We now have an expression for a point (x_3, y_3) obtained from either PA or PD; [11]

$$\left\{ \begin{matrix} \{x_3 = \lambda^2 - x_1 - x_2\} \\ \{y_3 = \lambda(x_1 - x_3) - y_1\} \end{matrix} \right\} \tag{3.32}$$

However we still need to select our λ value. If we are doing PA, then λ is simply the gradient of the line between the two points (x_1, y_1) and (x_2, y_2); [11]

$$\left\{ \lambda = \left(\frac{y_1 - y_2}{x_1 - x_2} \right) \right\} \tag{3.33}$$

However if we are doing PD, then our selection of λ relies on implicit differentiation. [11]

$$\left\{ y^2 = x^3 + ax + b \right\} \rightarrow \left\{ \left\{ \frac{\partial\left(y^2\right)}{\partial x} = 3x^2 + a \right\} \cup \left\{ \frac{\partial\left(y^2\right)}{\partial y} = 2y \right\} \right\} \rightarrow \left\{ \lambda = \frac{\partial y}{\partial x} = \left(\frac{3x^2 + a}{2y} \right) \right\} \tag{3.34}$$

By the nature of their construction as presented here, PA and PD are both closed equipped actions [26] that guarantee the group operator is a closed operator [26] over the geometry of E, and is hence a valid choice for group operator!

3.4.2 PA and PD in Affine Space

Now that we have demonstrated the selection of group operator is valid, it suffices to state the result for PA and PD in closed expressions. [11]

Point Addition

$$\left\{ \begin{matrix} \text{PA} : (x_1, y_1) * (x_2, y_2) \rightarrow (x_3, y_3) \\ \left\{ x_3 = \left(\frac{y_1 - y_2}{x_1 - x_2} \right)^2 - x_1 - x_2 \right\} \\ \left\{ y_3 = \left(\frac{y_1 - y_2}{x_1 - x_2} \right) (x_1 - x_3) - y_1 \right\} \end{matrix} \right\} \tag{3.35}$$

Point Doubling

$$\left\{ \begin{matrix} \text{PD} : (x_1, y_1) * (x_1, y_1) \rightarrow (x_3, y_3) \\ \left\{ x_3 = \left(\frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1 \right\} \\ \left\{ y_3 = \left(\frac{3x_1^2 + a}{2y_1} \right) (x_1 - x_3) - y_1 \right\} \end{matrix} \right\} \tag{3.36}$$

3.4.3 Generalising the Group Operator to a Group Action

We can finish this section by exclaiming that, without presenting the algorithm here, it would be possible to construct an algorithm that takes a point P on E_K and some integer q, and performs a "Point Multiplication" (PM), and returns the result of the point added to itself q times, through a combination of PAs and PDs. As an operator that takes a point and an integer, and utilises only the Group Operator, PM satisfies the requirements for a Group Equipped with an Action, i.e., PM is a Group Action. [26]

3.5 Projective Plane

Up to now we have considered the points of E_K as points in Affine coordinates, the direct product space of K^2. As the operations of PA and PD require a Field Inversion to compute, and our end goal is being able to perform PM, utilising many successive PA and PD operations, we desire a coordinate system that allows us to do PA and PD without a Field Inversion, as an inversion is logically expensive to perform, and becomes cumbersome if it is needed for each PA and PD operation, such that we can perform all the required PA and PD operations, and perform a single Field Inversion at the end of the PM algorithm.

To facilitate this, we must introduce Projective Coordinates, a coordinate system of points in a Projective Plane. To properly discuss projective planes, and save from later confusion, it is necessary to address here the ambiguity of a Projective Plane's dimension. The cardinal naming convention for a Projective Plane used here is such that a Projective Plane written P^Q has dimension $(Q + 1)$. The reasoning for this is that a Projective Plane P^Q, over the Field K, $P^Q(K)$, is composed of elements that are $(Q + 1)$-tuples of elements of K, and hence the objects of P^Q have dimension $(Q+1)$. This is to distinguish from the fact that subsets of P^Q are piecewise isomorphic to direct product spaces of K, with the largest direct product space of K being isomorphic to a subset of P^Q is K^Q. Because of this, if we are talking about the dimension of the direct product space K^Q, the dimension is simply Q, but the dimension of the Projective Plane P^Q is $(Q + 1)$, despite being isomorphic to $K^Q \cup J$ where $\forall h \in \mathbb{N} \rightarrow K^{Q+h} \not\subset J$. This distinction is necessary as when describing the Projective Plane in generality we will write P^Q, for dimension $(Q+1)$, instead of P^{Q+1}. (* $(J+1)$ Dimensional Projective Plane $\equiv$ Projective J Plane).

We are now in a position to define the Projective Plane P^Q defined on a Field K, as $P^Q(K)$ being a collection that is isomorphic to the union of all direct product spaces of the Field K for direct product spaces of sizes 0 up to Q.

$$\left\{ P^Q(K) \equiv \bigcup_{j=0}^{Q} K^j \right\} \tag{3.37}$$

We can alternatively define $P^Q(K)$ by the construction of its elements, wherein its elements are $(Q+1)$-tuples of elements of K, not including the $(Q+1)$-tuple that is composed solely of 0's. Note that the $(Q + 1)$-tuples are elements of the direct product space (of dimension $(Q + 1)$) of K.

$$\left\{ P^Q(K) \equiv K^{Q+1} / \{0\}^{Q+1} \left| K^{Q+1} = \{(k_1, ..., k_{Q+1}) \left| \forall j, k_j \in K \} \right. \right. \right\} \tag{3.38}$$

Which can alternatively be written as the set composed of $(Q+1)$-tuples written as Projective Coordinates, not including the Projective Coordinate of only 0's.

$$\left\{ P^Q\left(K\right) \equiv \left\{\left[k_1 : ... : k_{Q+1}\right] \left|\forall j, k_j \in K\right.\right\} / \left[0\right]^{Q+1} \right\} \tag{3.39}$$

Where the projective coordinates are themselves sets composed of all points in the Projective Plane that are 'projections' of any other point that is in the set of that projective coordinate. So, a projective coordinate is a set of points that projectively scale to each other, under some scale λ.

$$\left\{[k_1 : ... : k_{Q+1}] = \left\{\left(\lambda^{A_1}k_1, ..., \lambda^{A_Q}k_Q, \lambda k_{Q+1}\right) | \lambda \in (K/\{0\}), \forall j, A_j \in K, \forall j, A_j \text{ is constant}\right\}\right\} \tag{3.40}$$

In summary, a Projective Plane $P^Q(K)$ is populated by points that are $(Q+1)$-tuples of elements of K, organised into sets based on some choice of projective scaling, wherein the sets are composed of the points that scale to one-another, and these sets are the Projective Coordinates.

3.5.1 Projective Coordinates

To discuss Projective Coordinates with ease, we must first address the property of Projective Coordinates that a coordinate is "representative" of a set of points, and that any member point in a coordinate may be that coordinate's "representative." As the coordinate is composed of points that map to each other under some projective scaling, any one point will scale to all the other points represented by the coordinate, and as such any point can be chosen as the representative. Observe also that for each projective coordinate, with the last term, $k_{Q+1} \neq 0$, that all points represented by the coordinate have $k_{Q+1} \neq 0$, and that there will be one representative with $k_{Q+1} = j, \forall j \in K/0$, which provides us that there will be one representative for a projective coordinate with $k_{Q+1} \neq 0$ such that $k_{Q+1} = 1$. This is stated here as the equality between some arbitrary representation for a projective coordinate and its representation with $k_{Q+1} = 1$, by appropriate selection of λ.

$$\left\{[k_1 : ... : k_Q : k_{Q+1}] = \left[\left(\frac{k_1}{k_{Q+1}^{A_1}}\right) : ... : \left(\frac{k_Q}{k_{Q+1}^{A_Q}}\right) : 1\right] | \lambda = k_{Q+1} \neq 0\right\} \tag{3.41}$$

This is of fundamental importance, as we have no restriction placed on the other Q terms in the $(Q+1)$-tuple projective coordinate, so we may state that the set of all projective coordinates for which $k_{Q+1} \neq 0$ is isomorphic to the Affine Plane K^Q.

$$\left\{\{[k_1 : ... : k_Q : k_{Q+1}] \, | k_{Q+1} \neq 0\} \equiv K^Q\right\} \tag{3.42}$$

Similarly, for projective coordinates with $k_{Q+1} = 0$, and $k_Q \neq 0$, we can state a similar equality to a representation of the coordinate for which $k_Q = 1$, again by selection of λ.

$$\left\{[k_1 : ... : k_{Q-1} : k_Q : 0] = \left[\left(\frac{k_1}{k_Q^{\left(\frac{A_1}{A_Q}\right)}}\right) : ... : \left(\frac{k_{Q-1}}{k_Q^{\left(\frac{A_{Q-1}}{A_Q}\right)}}\right) : 1 : 0\right] \Big| \begin{matrix}\lambda^{A_Q} = k_Q \neq 0 \\ \lambda = \sqrt[A_Q]{k_Q} \neq 0\end{matrix}\right\} \tag{3.43}$$

Which again provides another isomorphic map!

$$\left\{\{[k_1 : ... : k_{Q-1} : k_Q : 0]\,|k_Q \neq 0\} \equiv K^{Q-1}\right\} \tag{3.44}$$

We can generalise this to any point with $k_{V+1} = 0$, and $k_V \neq 0$;

$$\left\{[k_1 : ... : k_V : 0 : ... : 0] = \left[\left(\frac{k_1}{k_V^{\left(\frac{A_1}{A_V}\right)}}\right) : ... : \left(\frac{k_{V-1}}{k_V^{\left(\frac{A_{V-1}}{A_V}\right)}}\right) : 1 : 0 : ... : 0\right] |\begin{matrix}\lambda^{A_V} = k_V \neq 0 \\ \lambda = \sqrt[A_V]{k_V} \neq 0\end{matrix}\right\} \tag{3.45}$$

Which provides a generalised isomorphic map!

$$\left\{\{[k_1 : ... : k_V : 0 : ... : 0]\,|k_V \neq 0\} \equiv K^{V-1}\right\} \tag{3.46}$$

Of importance is the result for $V = 2$;

$$\left\{[k_1 : k_2 : 0 : ... : 0] = \left[\left(\frac{k_1}{k_2^{\left(\frac{A_1}{A_2}\right)}}\right) : 1 : 0 : ... : 0\right] |\begin{matrix}\lambda^{A_2} = k_2 \neq 0 \\ \lambda = \sqrt[A_2]{k_2} \neq 0\end{matrix}\right\} \tag{3.47}$$

Which gives us an isomorphic relation to a direct product space of dimension 1.

$$\left\{\{[k_1 : k_2 : 0 : ... : 0]\,|k_2 \neq 0\} \equiv K^1\right\} \tag{3.48}$$

And when $V = 1$;

$$\{[k_1 : 0 : ... : 0] = [1 : 0 : ... : 0]\,|k_1 \neq 0\} \tag{3.49}$$

We obtain a dimensionless direct product space, i.e., a set composed of no objects, that is a 'point with no value.'

$$\left\{\{[k_1 : 0 : ... : 0]\,|k_1 \neq 0\} \equiv K^0\right\} \tag{3.50}$$

We are now able to succinctly summarise the result of all these isomorphisms that allows us to trade operations in an Affine Q Plane for operations in a Projective Q Plane.

$$\left\{P^Q(K) \equiv \bigcup_{j=0}^{Q} K^j = \left(\left(K^Q\right) \cup \left(\bigcup_{j=0}^{Q-1} K^j\right)\right) = \left(\text{(Affine Q space)} \cup \left(\bigcup_{j=0}^{Q-1} \text{(j Content at } \infty)\right)\right)\right\} \tag{3.51}$$

Where the notion of 'content at infinity' relates to the notion of the 'point at infinity' of the Elliptic Curve Group, as the incomplete isomorphic map from a Q dimensional Affine Plane to a $(Q+1)$ dimensional Projective Plane is completed by including all lower dimensional Affine Planes in the isomorphic map, but operations that are minimally generalised over an Affine Q Plane do not have extensions to lower dimensional Affine Planes, and hence the complete isomorphic map from the union set of all Affine J Planes, $\forall J \in (0, Q)$ to the Projective Q Plane includes both the Affine Q Plane that we are generalising operations over into the Projective Plane, along with Affine J Planes, $\forall J \in (0, Q-1)$ in which the Projective Plane operations work, but do not have corresponding operations under the restriction of being mapped back to an Affine Q Plane. Hence, we refer to the subsets of P^Q that are piecewise isomorphic to K^J, $\forall J \in (0, Q-1)$, as 'content at infinity.' Note that content may be stated as; point, line, plane, volume, content.

3.5.2 Projective Plane in 3 Dimensions

To substantiate the use of the Projective Plane for performing PA and PD, we begin by stating the 3 Dimensional Projective Plane;

$$\left\{ P^2 (K) = \left(K^2 \cup K^1 \cup K^0\right) = \left(\bigcup \begin{matrix} \text{The Affine 2D Plane} \\ \text{The Line at } \infty \\ \text{The Point at } \infty \end{matrix} \right) \right\} \tag{3.52}$$

Which we also write as the set of all Projective Coordinates; [11]

$$\left\{ P^2 (K) \equiv \{[x : y : z] \,|x, y, z \in K\} \,/\, [0]^3 \right\} \tag{3.53}$$

For which each projective coordinate is the set of points that scale under λ; [11]

$$\left\{ [x : y : z] = \left\{ \left(\lambda^a x, \lambda^b y, \lambda z\right) |\lambda \in (K/\,\{0\})\,, a, b \in K, a, b \text{ constant} \right\} \right\} \tag{3.54}$$

And each coordinate is represented by its equivalence where $\forall p \in P^2, p_z \neq 0 \rightarrow p_z \equiv 1$; [11]

$$\left\{ [x : y : z] = \left[\left(\frac{x}{z^a}\right) : \left(\frac{y}{z^b}\right) : 1 \right] |\lambda = z \neq 0 \right\} \tag{3.55}$$

Then we have facilitated a Projective Plane with Projective Coordinates that are mapped to, from points in the Affine Plane; [11]

$$\{(x, y) = [x : y : 1]\} \tag{3.56}$$

As well as a mapping from Projective Coordinates to points in the Affine Plane; [11]

$$\left\{ [x : y : z] = \left(\left(\frac{x}{z^a}\right), \left(\frac{y}{z^b}\right) \right) \right\} \tag{3.57}$$

3.5.3 Elliptic Curve Group Operation in the Projective Plane

To transform the Affine Plane equations for PA and PD into their Projective equivalents, we begin by succinctly stating a summary of the Affine Plane equations for PA and PD, based on the selection of φ (where φ is a symbolic replacement for λ in 3.4.1);

$$\left\{ \begin{matrix} \left\{ x_3 = (\varphi)^2 - x_1 - x_2 \right\} \\ \left\{ y_3 = (\varphi)\,(x_1 - x_3) - y_1 \right\} \\ \left\{ \varphi \in \left\{ \left(\frac{y_1 - y_2}{x_1 - x_2}\right), \left(\frac{3x_1^2 + a}{2y_1}\right) \right\} \right\} \end{matrix} \right\} \tag{3.58}$$

As well as stating the coordinate transformation, from Affine Coordinate Equations to Projective Plane Equations, that we will be using; a substitution for Affine coordinates of points by the representation for those Affine coordinates under the map from a general Projective coordinate;

$$\left\{ [X_j : Y_j : Z_j] = \left(\left(\frac{X_j}{Z_j^a}\right), \left(\frac{Y_j}{Z_j^b}\right) \right) = (x_j, y_j) \right\} \tag{3.59}$$

Which, after performing the substitution, yields the following Projective Plane equations!

$$\left\{\begin{array}{c} \left\{\frac{X_3}{Z_3^a} = (\varphi)^2 - \frac{X_1}{Z_1^a} - \frac{X_2}{Z_2^a}\right\} \\ \left\{\frac{Y_3}{Z_3^b} = (\varphi)\left(\frac{X_1}{Z_1^a} - \frac{X_3}{Z_3^a}\right) - \frac{Y_1}{Z_1^b}\right\} \\ \left\{\varphi \in \left\{\left(\frac{\frac{Y_1}{Z_1^b} - \frac{Y_2}{Z_2^b}}{\frac{X_1}{Z_1^a} - \frac{X_2}{Z_2^a}}\right), \left(\frac{3\left(\frac{X_1}{Z_1^a}\right)^2 + a}{2\frac{Y_1}{Z_1^b}}\right)\right\}\right\} \end{array}\right\} \tag{3.60}$$

To which we also state three expressions for φ, the first two if the equation is that of PA, according to the selection for the powers of λ, and thirdly if the equation is that of PD;

$$\left\{\varphi \in \left\{\begin{array}{ccc} \left(\frac{Z_1^{a-b} Z_2^a Y_1 - Z_2^{a-b} Z_1^a Y_2}{Z_2^a X_1 - Z_1^a X_2}\right) & \left(\frac{Z_2^b Y_1 - Z_1^b Y_2}{Z_1^{b-a} Z_2^b X_1 - Z_2^{b-a} Z_1^b X_2}\right) & \left(\frac{3 Z_1^b X_1^2 + a Z_1^{b+2a}}{2 Y_1 Z_1^{2a}}\right) \\ \text{iff } a > b & \text{iff } a < b & \text{iff Doubling} \end{array}\right\}\right\} \tag{3.61}$$

Now that we have expressions in Projective Plane equations for the Group Operator (and φ) when the operation is Point Addition and Point Doubling, we can choose values for the powers of λ, that govern the explicit scaling of choice of Projective Plane, and manipulate the equations to be solutions for the resultant Projective Coordinate that do not involve Field Inversion, i.e. they are solutions for X_3, Y_3, Z_3, composed only of Field Addition, Subtraction, Multiplication.

3.5.4 Elliptic Curve Group Operation in Jacobian Projection

One particularly favourable selection for the powers of λ is known as Jacobian Coordinates [11]; here-after referred to as the Jacobian Projection. The Jacobian Projection selects the powers a and b of λ in the Projective Plane Scaling, to be $a = 2$ and $b = 3$. We now fix these selections and solve the PA and PD equations;

Point Addition in Jacobian Projection: Derivation

To derive the expressions for PA, we begin by solving X_3;

$$\left\{\begin{array}{ll} \frac{X_3}{Z_3^2} & = \left(\left(\frac{Z_2^3 Y_1 - Z_1^3 Y_2}{Z_1^1 Z_2^3 X_1 - Z_2^1 Z_1^3 X_2}\right)^2 - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2}\right) \\ & = \left(\frac{\left(Z_2^3 Y_1 - Z_1^3 Y_2\right)^2}{\left(Z_1^1 Z_2^3 X_1 - Z_2^1 Z_1^3 X_2\right)^2} - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2}\right) \\ & = \left(\frac{(Z_2^3 Y_1 - Z_1^3 Y_2)^2 - X_1 Z_2^2 (Z_2^2 X_1 - Z_1^2 X_2)^2 - X_2 Z_1^2 (Z_2^2 X_1 - Z_1^2 X_2)^2}{Z_1^2 Z_2^2 (Z_2^2 X_1 - Z_1^2 X_2)^2}\right) \end{array}\right\} \tag{3.62}$$

At which point we must state a selection for Z_3;

$$\left\{\begin{array}{ll} Z_3^2 & = \left(Z_1^1 Z_2^3 X_1 - Z_2^1 Z_1^3 X_2\right)^2 \\ & = \left(Z_1^1 Z_2^1 \left(Z_2^2 X_1 - Z_1^2 X_2\right)\right)^2 \\ & = \left(Z_1^2 Z_2^2 \left(Z_2^2 X_1 - Z_1^2 X_2\right)^2\right) \\ Z_3 & = \left(Z_1 Z_2 \left(Z_2^2 X_1 - Z_1^2 X_2\right)\right) \end{array}\right\} \tag{3.63}$$

Which provides us with our expression for X_3;

$$\left\{X_3 = ((Z_2^3Y_1 - Z_1^3Y_2)^2 - X_1Z_2^2(Z_2^2X_1 - Z_1^2X_2)^2 - X_2Z_1^2(Z_2^2X_1 - Z_1^2X_2)^2)\right\} \tag{3.64}$$

Now we must solve Y_3;

$$\left\{\begin{array}{ll} \frac{Y_3}{Z_3^3} & = \left(\frac{(Z_2^3Y_1 - Z_1^3Y_2)}{(Z_1^1Z_2^3X_1 - Z_2^1Z_1^3X_2)}\right)\left(\frac{X_1}{Z_1^2} - \frac{X_3}{Z_3^2}\right) - \frac{Y_1}{Z_1^3} \\ & = \left(\frac{X_1(Z_2^3Y_1 - Z_1^3Y_2)}{Z_1^2Z_3^1} - \frac{X_3(Z_2^3Y_1 - Z_1^3Y_2)}{Z_3^3} - \frac{Y_1}{Z_1^3}\right) \\ & = \left(\frac{X_1Z_1^1Z_3^2(Z_2^3Y_1 - Z_1^3Y_2) - X_3Z_1^3(Z_2^3Y_1 - Z_1^3Y_2) - Y_1Z_3^3}{Z_1^3Z_3^3}\right) \end{array}\right\} \tag{3.65}$$

After extracting the Z_3 terms now solving;

$$\left\{\begin{array}{ll} Y_3 & = \left(\frac{X_1Z_1^1Z_1^2Z_2^2(Z_2^2X_1 - Z_1^2X_2)^2(Z_2^3Y_1 - Z_1^3Y_2) - X_3Z_1^3(Z_2^3Y_1 - Z_1^3Y_2) - Y_1Z_1^3Z_2^3(Z_2^2X_1 - Z_1^2X_2)^3}{Z_1^3}\right) \\ & = \left(X_1Z_2^2\left(Z_2^2X_1 - Z_1^2X_2\right)^2\left(Z_2^3Y_1 - Z_1^3Y_2\right) - X_3\left(Z_2^3Y_1 - Z_1^3Y_2\right) - Y_1Z_2^3\left(Z_2^2X_1 - Z_1^2X_2\right)^3\right) \\ & = \left(\left(Z_2^3Y_1 - Z_1^3Y_2\right)\left(X_1Z_2^2\left(Z_2^2X_1 - Z_1^2X_2\right)^2 - X_3\right) - Y_1Z_2^3\left(Z_2^2X_1 - Z_1^2X_2\right)^3\right) \end{array}\right\} \tag{3.66}$$

Point Addition in Jacobian Projection: Derived

This derivation yields the resultant expressions for PA under a Jacobian Projection; [11]

$$\left\{\begin{array}{c} X_3 = \left(\left(Z_2^3Y_1 - Z_1^3Y_2\right)^2 - \left(X_1Z_2^2 + X_2Z_1^2\right)\left(Z_2^2X_1 - Z_1^2X_2\right)^2\right) \\ Y_3 = \left(\left(Z_2^3Y_1 - Z_1^3Y_2\right)\left(X_1Z_2^2\left(Z_2^2X_1 - Z_1^2X_2\right)^2 - X_3\right) - Y_1Z_2^3\left(Z_2^2X_1 - Z_1^2X_2\right)^3\right) \\ Z_3 = \left(Z_1Z_2\left(Z_2^2X_1 - Z_1^2X_2\right)\right) \end{array}\right\} \tag{3.67}$$

Point Doubling in Jacobian Projection: Derivation

Now we derive expressions for PD, starting again with X_3;

$$\left\{\begin{array}{ll} \frac{X_3}{Z_3^2} & = \left(\frac{3Z_1^3X_1^2 + aZ_1^7}{2Y_1Z_1^4}\right)^2 - 2\frac{X_1}{Z_1^2} \\ & = \frac{\left(3Z_1^3X_1^2 + aZ_1^7\right)^2}{\left(2Y_1Z_1^4\right)^2} - 2\frac{X_1}{Z_1^2} \\ & = \frac{\left(3X_1^2 + aZ_1^4\right)^2 - 8Y_1^2X_1}{4Y_1^2Z_1^2} \end{array}\right\} \tag{3.68}$$

From which we can select Z_3 and assert X_3;

$$\left\{\begin{array}{c} X_3 = \left(\left(3X_1^2 + aZ_1^4\right)^2 - 8Y_1^2X_1\right) \\ Z_3 = \left(2Y_1Z_1\right) \end{array}\right\} \tag{3.69}$$

And now we solve Y_3;

$$\left\{ \frac{Y_3}{Z_3^3} = \left(\frac{3Z_1^3X_1^2 + aZ_1^7}{2Y_1Z_1^4}\right)\left(\frac{X_1}{Z_1^2} - \frac{X_3}{Z_3^2}\right) - \frac{Y_1}{Z_1^3} \right\} \tag{3.70}$$

After extracting the Z_3 terms;

$$\left\{ \begin{array}{l} Y_3 = Z_3^2\left(3X_1^2 + aZ_1^4\right)\left(\frac{X_1}{Z_1^2} - \frac{X_3}{Z_3^2}\right) - \frac{Z_3^3Y_1}{Z_1^3} \\ \quad = \left(3X_1^2 + aZ_1^4\right)\left(\frac{Z_3^2X_1}{Z_1^2} - X_3\right) - \frac{Z_3^3Y_1}{Z_1^3} \\ \quad = \left(3X_1^2 + aZ_1^4\right)\left(4Y_1^2X_1 - X_3\right) - 8Y_1^4 \end{array} \right\} \tag{3.71}$$

Point Doubling in Jacobian Projection: Derived

Which now allows us to present the equations for PD under a Jacobian Projection; [11]

$$\left\{ \begin{array}{c} X_3 = \left(\left(3X_1^2 + aZ_1^4\right)^2 - 8Y_1^2X_1\right) \\ Y_3 = \left(\left(3X_1^2 + aZ_1^4\right)^1\left(4Y_1^2X_1 - X_3\right)^1 - 8Y_1^4\right) \\ Z_3 = \left(2Y_1Z_1\right) \end{array} \right\} \tag{3.72}$$

3.6 Chapter Summary

This chapter has now laid out definitions and treatment for Fields, both infinite fields and finite fields (Galois Fields), how to construct the elements and do arithmetic with those elements for Galois Fields for a particular prime, or prime power, cardinality. Following, the three Weierstrass Elliptic Curve equations where defined and related to choice of Galois Field over which the curves are defined. We then proceeded to define Elliptic Curve Groups over Prime Galois Fields as solution sets to the Weierstrass Elliptic Curve equations whose solutions were composed of elements in Galois Fields with prime cardinality. We then used these solution sets to define the Group Structure on Elliptic Curves, defining the Group Operator (and Group Action), before extending the group operator into a Projective Plane context. Hence, in totality, this chapter has demonstrated the Projective Plane equations for the Group Operator of an Elliptic Curve Group defined over a Galois Field with Prime cardinality.

Chapter 4

Modelling ECC Protocols

4.1 Chapter Introduction

This chapter will cover the modelling of all levels of the ECDH and ECDSA schemes, starting with the fundamental cells for field operations. Following this is the construction of a model for performing Field Inversion, that of finding the multiplicative inverse for a field element. Next, models for performing the Elliptic Curve Group Operation are discussed, in terms of 'Point Addition' and 'Point Doubling.' These will be modelled under the utilisation of the Jacobian Variety of the Projective Plane, see 3.5.4. After finishing the discussion of the Projective model, this will be utilised with the Field Inverter to produce a model for performing the Elliptic Curve Group Action in the Affine Context.

4.2 Field Arithmetic Processing

Field Arithmetic Primitives, the smallest and most fundamental components, have a flow on effect for their design to the rest of the design. We can break this up into individually modelling addition, subtraction, and multiplication. Given multiplication relies on multiple addition blocks, and subtraction relies on several addition blocks (to enforce modular behaviour rather than linear subtraction), it is pertinent to discuss addition first.

4.2.1 FAP: Addition

To model addition, we must recall the fundamental logic of a 'Full Adder' (FA) cell;

$$\left\{\begin{array}{c} \text{Inputs} : a, b, c_{in} \\ sum = a \text{ xor } b \text{ xor } c_{in} \\ c_{out} = (a \text{ and } b) \text{ or } (a \text{ and } c_{in}) \text{ or } (b \text{ and } c_{in}) \\ \text{Outputs} : sum, c_{out} \end{array}\right\} \tag{4.1}$$

A generic length FA logic cell can be stipulated in, generic terms, if we have a chain of FA in which the c_{out} from one FA cell is connected to the c_{in} of the next FA cell.

$$\left\{\begin{array}{c} \text{Inputs}: a_k...a_0, b_k...b_0, c_{in} \\ \text{Internal}: \text{ Carry Chain } c_{(k-1)}...c_0 \text{ where } c_{-1} = c_{in} \\ c_j = (a_j \text{ and } b_j) \text{ or } (a_j \text{ and } c_{(j-1)}) \text{ or } (b_j \text{ and } c_{(j-1)}) \\ sum_j = a_j \text{ xor } b_j \text{ xor } c_{(j-1)} \\ c_{out} = (a_k \text{ and } b_k) \text{ or } (a_k \text{ and } c_{(k-1)}) \text{ or } (b_k \text{ and } c_{(k-1)}) \\ \text{Outputs}: sum_k...sum_0, c_{out} \end{array}\right\} \tag{4.2}$$

This is the easiest way to express a chain of FA, with $(k+1)$ FA cells. Note that a chain of h FA cells is able to perform addition on numbers from 0 up to $2^h - 1$, or in the range of -2^{h-1} to $2^{h-1} - 1$. We can modify this slightly such that if we have a need for a maximum fixed length, that we can simply change the identification of the carry out bit to be a final summation bit, as when summing two values over h bits, the result will necessarily be $(h+1)$ bits. Hence we describe an Adder cell (ADDR) as a chain of FA, disregarding the carry in. Then, an h bit ADDR is described as;

$$\left\{\begin{array}{c} \text{Inputs}: a_{(h-1)}...a_0, b_{(h-1)}...b_0 \\ \text{Internal}: \text{ Carry Chain } c_{(h-2)}...c_0 \text{ where } c_{-1} = 0 \\ c_j = (a_j \text{ and } b_j) \text{ or } (a_j \text{ and } c_{(j-1)}) \text{ or } (b_j \text{ and } c_{(j-1)}) \\ sum_j = a_j \text{ xor } b_j \text{ xor } c_{(j-1)} \\ sum_h = (a_{(h-1)} \text{ and } b_{(h-1)}) \text{ or } (a_{(h-1)} \text{ and } c_{(h-2)}) \text{ or } (b_{(h-1)} \text{ and } c_{(h-2)}) \\ \text{Outputs}: sum_h...sum_0 \end{array}\right\} \tag{4.3}$$

This h bit ADDR cell is however optimisable for faster results. Noting that the Carry Chain's values will take time to propagate, the total delay of this logic is the delay of a single carry operation multiplied by the number of FA cells. We can use an implementation appraoch similar to the Kogge-Stone (KS) adder logic block, which is simply a fixed bit length block with all of the internal logic typically written loop unwrapped. By knowing some fixed length, we can optimise the use of KS style optimisation by mixing the use of KS style cells, and MUX'ing the results. This project was initially to be done over 256 bits, so to construct a 256 KS style MUX'd cell (256KSMUXADDDR), it was observed that $256 = 4^4$, a fact that can be exploited to effectively 'nest' KS adders and continuously MUX the results. This optimisation was later disregarded but is left here as the final result mimics the behaviour presented in this optimisation by use of a fixed length terminal block that performs generic loop unwrapped addition, taking some higher order bit length input and splitting it in half.

First, observe that if we have f cells, broken up into g lots of h cells, then each block of h cells will have a delay of $h * \delta(carry)$. Once the first block of h has passed its delay time, so too will have the other blocks, and hence it is down to $g-1$ propagations of the MUX's selection of its outputs. Hence the total delay will be $h * \delta(carry) + (g-1) * \delta(MUX - select)$. If we nest this and say f cells are broken up into g lots of g lots of h blocks (in which if f were unchanged, our h has decreased and g has increased) then the internal delay for a single g block of h is still $h * \delta(carry) + (g-1) * \delta(MUX - select)$, but in this time all the other blocks have also completed their delay times, and so now it is simply another $(g-1) * \delta(MUX - select)$ delay for each of the outer blocks to propagate their concatenated MUX chains; giving a total delay of $h*\delta(carry)+2*(g-1)*\delta(MUX-select)$. We can extend this to say if we have g blocks of h where the blocks are at a nesting level j, then the delay for this will be $h * \delta(carry) + j * (g-1) * \delta(MUX - select)$, however this is at a trade-off with the available room to accommodate a circuit of this size as each degree of nesting effectively multiplies the size of the cell. A choice was been made to exploit the fact that $256 = 4^4$ to construct a 3 layer nesting of 4 blocks of 4.

$$\text{4KS} = \left\{\begin{array}{c} \text{Inputs}: A = a_3a_2a_1a_0,\ B + b_3b_2b_1b_0,\ c_{in} \\ \text{Internal}:\ \text{Carry Chain } c_2c_1c_0 \text{ where } c_{-1} = c_{in} \\ sum_j = a_j \text{ xor } b_j \text{ xor } c_{(j-1)} \\ c_j = (a_j \text{ and } b_j) \text{ or } (a_j \text{ and } c_{(j-1)}) \text{ or } (b_j \text{ and } c_{(j-1)}) \\ c_{out} = (a_3 \text{ and } b_3) \text{ or } (a_3 \text{ and } c_2) \text{ or } (b_3 \text{ and } c_2) \\ \text{Outputs}: S = sum_3sum_2sum_1sum_0,\ c_{out} \end{array}\right\} \tag{4.4}$$

We then MUX two of the 4KS cells;

$$\text{4KSMUX} = \left\{\begin{array}{c} \text{Inputs}: A = a_3a_2a_1a_0,\ B = b_3b_2b_1b_0,\ c_{in} \\ \text{Subcircuits}: X\ (4KS),\ Y\ (4KS) \\ \text{Subcircuit}: X.A \text{ connects } A,\ X.B \text{ connects } B,\ X.c_{in} \text{ connects } 0 \\ \text{Subcircuit}: Y.A \text{ connects } A,\ Y.B \text{ connects } B,\ Y.c_{in} \text{ connects } 1 \\ S \text{ connects } X.S \text{ when } (c_{in} = 0) \text{ else } Y.S \\ c_{out} \text{ connects } X.c_{out} \text{ when } (c_{in} = 0) \text{ else } Y.c_{out} \\ \text{Outputs}: S = sum_3sum_2sum_1sum_0,\ c_{out} \end{array}\right\} \tag{4.5}$$

We then concatenate 4 of the 4KSMUX;

$$\text{16KS} = \left\{\begin{array}{c} \text{Inputs}: A = a_{15}...a_0,\ B = b_{15}...b_0,\ c_{in} \\ \text{Subcircuits}: X_0\ (\text{4KSMUX}),\ X_1\ (\text{4KSMUX}),\ X_2\ (\text{4KSMUX}),\ X_3\ (\text{4KSMUX}) \\ X_j.A \text{ connects } A[0 + j4, 3 + j4] \\ X_j.B \text{ connects } B[0 + j4, 3 + j4] \\ X_j.c_{in} \text{ connects } X_{j-1}.c_{out} \\ S[0 + j4, 3 + j4] \text{ connects } X_j.S \\ c_{out} \text{ connects } X_3.c_{out} \\ \text{Outputs}: S = sum_{15}...sum_0,\ c_{out} \end{array}\right\} \tag{4.6}$$

We then MUX two of the 16KS cells to obtain a 16KSMUX;

$$\text{16KSMUX} = \left\{ \begin{array}{c} \text{Inputs} : A = a_{15}...a_0,\ B = b_{15}...b_0,\ c_{in} \\ \text{Subcircuits} : X\ (16KS),\ Y\ (16KS) \\ \text{Subcircuit} : X.A \text{ connects } A,\ X.B \text{ connects } B,\ X.c_{in} \text{ connects } 0 \\ \text{Subcircuit} : Y.A \text{ connects } A,\ Y.B \text{ connects } B,\ Y.c_{in} \text{ connects } 1 \\ S \text{ connects } X.S \text{ when } (c_{in} = 0) \text{ else } Y.S \\ c_{out} \text{ connects } X.c_{out} \text{ when } (c_{in} = 0) \text{ else } Y.c_{out} \\ \text{Outputs} : S = sum_{15}...sum_0,\ c_{out} \end{array} \right\} \tag{4.7}$$

We then concatenate 4 of the 16KSMUX to obtain a 64KS;

$$\text{64KS} = \left\{ \begin{array}{c} \text{Inputs} : A = a_{63}...a_0,\ B = b_{63}...b_0,\ c_{in} \\ \text{Subcircuits} : X_0\ (\text{16KSMUX}),\ X_1\ (\text{16KSMUX}),\ X_2\ (\text{16KSMUX}),\ X_3\ (\text{16KSMUX}) \\ X_j.A \text{ connects } A[0 + j16, 3 + j16] \\ X_j.B \text{ connects } B[0 + j16, 3 + j16] \\ X_j.c_{in} \text{ connects } X_{j-1}.c_{out} \\ S[0 + j16, 3 + j16] \text{ connects } X_j.S \\ c_{out} \text{ connects } X_3.c_{out} \\ \text{Outputs} : S = sum_{63}...sum_0,\ c_{out} \end{array} \right\} \tag{4.8}$$

We then MUX two of the 64KS cells to obtain a 64KSMUX;

$$\text{64KSMUX} = \left\{ \begin{array}{c} \text{Inputs} : A = a_{63}...a_0,\ B = b_{63}...b_0,\ c_{in} \\ \text{Subcircuits} : X\ (64KS),\ Y\ (64KS) \\ \text{Subcircuit} : X.A \text{ connects } A,\ X.B \text{ connects } B,\ X.c_{in} \text{ connects } 0 \\ \text{Subcircuit} : Y.A \text{ connects } A,\ Y.B \text{ connects } B,\ Y.c_{in} \text{ connects } 1 \\ S \text{ connects } X.S \text{ when } (c_{in} = 0) \text{ else } Y.S \\ c_{out} \text{ connects } X.c_{out} \text{ when } (c_{in} = 0) \text{ else } Y.c_{out} \\ \text{Outputs} : S = sum_{63}...sum_0,\ c_{out} \end{array} \right\} \tag{4.9}$$

We then concatenate 4 of the 64KSMUX to obtain a 256KS;

$$\text{256KS} = \left\{ \begin{array}{c} \text{Inputs} : A = a_{255}...a_0,\ B = b_{255}...b_0,\ c_{in} \\ \text{Subcircuits} : X_0\ (\text{64KSMUX}),\ X_1\ (\text{64KSMUX}),\ X_2\ (\text{64KSMUX}),\ X_3\ (\text{64KSMUX}) \\ X_j.A \text{ connects } A[0 + j64, 3 + j64] \\ X_j.B \text{ connects } B[0 + j64, 3 + j64] \\ X_j.c_{in} \text{ connects } X_{j-1}.c_{out} \\ S[0 + j64, 3 + j64] \text{ connects } X_j.S \\ c_{out} \text{ connects } X_3.c_{out} \\ \text{Outputs} : S = sum_{255}...sum_0,\ c_{out} \end{array} \right\} \tag{4.10}$$

We now have a 256KS ADDR. Despite the naming convention on the subcircuits up to this point, that the 256KS does not actually have a MUX component in the highest level, because it is built from MUX'd circuits and it was our final goal, we call it the 256KSMUX-ADDR cell. This now has an overall circuit delay of $4 * \delta(\text{carry}) + 9 * \delta(\text{MUX.select})$. Despite this is good way to optimise time, the size is somewhat disproportionate, with 2 lots of the 4KS in the 4KSMUX, 16 of the 4KS in the 16KSMUX, 128 of the 4KS in the 64KSMUX, and 512 of the 4KS in the completed 256KSMUXADDR. This was used in the initial design phase to construct an ADDR circuit optimised to 256 bits, and is presented here as an optimisation, but was later replaced by a circuit description that split bitlengths into halves, rather than quadrants. Without replicating more circuit descriptions, we will simply describe it as a circuit that takes any bit length, splits it in half, and continuously splits the split branches in half again and again until they reach some terminal length upon which it enacts the KS style addition. This is such that the behaviour when the terminal length is set to 4 and the input bit length is 256, the circuit generated is almost identical to that described in the optimisation process presented here. We will simply refer to this as an N-MUXADDR;

4.2.2 FAP: Modular Addition

Now that we have an N bit MUXADDR, we must be able to enforce modular addition with it. If we take the N-MUXADDR, and enforce a carry-in of 0, and take the carry out bit as the (N+1)th bit of the addition, then this is simple addition. We replace the notation of N-MUXADDR under these stipulations and will simply from now on call it the N-ADDR (with (N+1) output bits).

$$\text{N}-\text{ADDR} = \left\{ \begin{array}{c} \text{Inputs}: A = a_{(N-1)}...a_0,\ B = b_{(N-1)}...b_0 \\ \text{Subcircuits}: X\ (\text{N}-\text{MUXADDR}) \\ X.A \text{ connects } A,\ X.B \text{ connects } B,\ X.c_{in} \text{ connects } 0 \\ S_{(N-1)}...S_0 \text{ connects } X.S,\ S_N \text{ connects } X.c_{out} \\ \text{Outputs}: S \end{array} \right\} \tag{4.11}$$

To enforce modularity, we must check for equality with, or inequality to, the prime value that defines the Field. If the addition of two values is equal to the prime, then the modulo result is 0, and if the addition is greater than the prime, then the modulo result in that value subtract the prime. To facilitate this, the modular addition cell will require two of the N-ADDR cells, one that performs the addition on the inputs A and B, and the other that unconditionally takes the result from the first cell as one of its inputs, with the other input being the 2's compliment of the prime. Once the delay of the first N-ADDR has passed, then sub-circuits must evaluate if the result if greater or equal to the value of the prime. While these are being checked, the second N-ADDR cell works unconditionally. Once the total time delay of either (which ever is greater) has passed, both the result of the addition of the inputs, and their addition minus the prime, should be known, along with whether the addition is equal to or greater than the prime, and hence the circuit can select which ADDR's value to place on the output. Thus we have a logic cell for modular addition, with some constant prime (although easily generalisable);

$$\begin{pmatrix} \text{N} \\ \text{MOD} \\ \text{ADDR} \end{pmatrix} = \left\{ \begin{matrix} \text{Inputs} : A = a_{(N-1)}...a_0,\ B = b_{(N-1)}...b_0 \\ \text{Subcircuits} : X\ (\text{N} - \text{ADDR}),\ Y\ (\text{N} - \text{ADDR}) \\ \text{Internal} : \text{Sum}_{(A+B):N}...\text{Sum}_{(A+B):0},\ \text{Sum}_{(A+B-P):N}...\text{Sum}_{(A+B-P):0} \\ \text{Constants} : \text{Prime } P = p_{(N-1)}...p_0,\ \text{Prime's 2's Compliment } Q = q_{(N-1)}...q_0 \\ X.A \text{ connects } A,\ X.B \text{ connects } B,\ X.S \text{ connects } \text{Sum}_{(A+B)} \\ Y.A \text{ connects } \text{Sum}_{(A+B)},\ Y.B \text{ connects } Q,\ Y.S \text{ connects } \text{Sum}_{(A+B-P)} \\ S \text{ connects } X.S \text{ when } X.S < P \text{ else } 0 \text{ when } X.S = P \text{ else } Y.S \\ \text{Outputs} : S[(N-1):0] = sum_{(N-1)}...sum_0 \end{matrix} \right\} \tag{4.12}$$

And thus we have the description for the N-MODADDR circuit. The only part to consider about this that has not yet been mentioned is the logic to determine the equality or inequality between two numbers.

4.2.3 FAP: Equality and Inequality

Testing the equality and inequality between two numbers will be presented here generally; noting that the notation $[X|Y](Z)$ is the action of X over the conditional Y on the expressions Z; if we say $X = x_k...x_0$ and $Y = y_k...y_0$;

$$\left\{ \begin{matrix} X = Y & \begin{bmatrix} AND \\ \forall h \in [0,k] \end{bmatrix} (x_h \text{ xnor } y_h) \\ X > Y & \begin{bmatrix} \text{OR} \\ \forall h \in [0,k] \end{bmatrix} \left(x_h \text{ and } !y_h \text{ and } \begin{bmatrix} \text{AND} \\ \forall j \in [h+1,k] \end{bmatrix} (x_j \text{ xnor } y_j) \right) \end{matrix} \right\} \tag{4.13}$$

Where the expression for the equality is such that all (generalised with an 'AND') the bits of X and Y are the same, using an xnor to test similarity of the individual bits. The expression for the inequality holds that for any bit (generalised with an 'OR' over all bits), if that bit is set in X and off in Y (the expression "x_h and $!y_h$") and all (generalised with an 'AND') of the bits higher than it are equal in X & Y, then X is greater than Y.

4.2.4 FAP: Subtraction

Subtraction over integers is easy to implement similarly to the initial logic cell presented in addition, however this does not implement well for modular subtraction. If we have two numbers, and we wish to subtract one from the other using adders, we must first use an adder to obtain the 2's compliment form of the subtrahend. We must then also in this time determine if the subtrahend is larger or smaller or equal to the minuend. Obviously, if they are equal then the result is 0. If the subtrahend is smaller than the minuend then we must simply add the 2's compliment of the subtrahend to the minuend. If the subtrahend is greater than the minuend we must add the 2's compliment of the subtrahend to the result of adding the minuend and the prime (that defines the Field). The first step of this algorithm is to simultaneously find the 2's compliment of the subtrahend and determine if the subtrahend is greater than or equal to the minuend, and also to add the prime to the minuend [uses two ADDR's and equality checkers]. The second step is to then add the 2's complimented form of the subtrahend to either the minuend or the result of adding the minuend and the prime, and the result of this will be the modular subtraction.

$$\begin{pmatrix}\text{N}\\\text{MOD}\\\text{SUBT}\end{pmatrix} = \left\{\begin{matrix}\text{Inputs}: A = a_{(N-1)}...a_0,\ B = b_{(N-1)}...b_0\\ \text{Constants}: \text{Prime } P = p_{(N-1)}...p_0,\ \text{Subcircuits}: B_{Neg}\ (\text{N}-\text{MODADDR})\\ \text{Subcircuits}: X\ (\text{N}-\text{MODADDR}),\ Y\ (\text{N}-\text{MODADDR})\\ \text{Internal}: \text{B}_{Inv} = !B,\ \text{B}_{(Neg):N}...\text{B}_{(Neg):0},\ \text{Sw}_N...\text{Sw}_0,\ \text{Sum}_{(A+P):N}...\text{Sum}_{(A+P):0}\\ B_{Neg}.A \text{ connects } \text{B}_{Inv},\ B_{Neg}.B \text{ connects } "...01",\ B_{Neg}.S \text{ connects } B_{Neg}\\ Y.A \text{ connects } A,\ Y.B \text{ connects } P,\ Y.S \text{ connects } \text{Sum}_{(A+P)}\\ Sw \text{ connects } A \text{ when } A >= B \text{ else } \text{Sum}_{A+P}\\ X.A \text{ connects } B_{Neg},\ X.B \text{ connects } Sw,\ X.S \text{ connects } \text{Dif}_{(A-B)}\\ \text{Outputs}: D[(N-1):0] = \text{Dif}_{(A-B):N}...\text{Dif}_{(A-B):0}\end{matrix}\right\} \tag{4.14}$$

4.2.5 FAP: Multiplication

Field multiplication is somewhat more involved, requiring a generalised amount of ADDR circuits just to perform the multiplication;

$$\begin{pmatrix}\text{N}\\\text{MULT}\end{pmatrix} = \left\{\begin{matrix}\text{Inputs}: A = a_{(N-1)}...a_0,\ B = b_{(N-1)}...b_0\\ \text{Subcircuits}: \forall k \in [1,(N-1)] \rightarrow X_k\ (\text{N}-\text{ADDR})\\ \text{Internal}: \text{Mult}_{(AB):(2N-1)}...\text{Mult}_{(AB):0},\ \text{AndArr}_{(N-1):(N-1)}...\text{AndArr}_{0:0}\\ \forall k \forall j \rightarrow \text{AndArr}_{k:j} \text{ connects } (A_k \text{ and } B_j)\\ \forall k \in [1,(N-1)] \rightarrow X_k.A \text{ connects } \text{AndArr}_{k:[0,(N-1)]}\\ X_k.B \text{ connects } X_{k-1}.S[1,N],\ \text{where } X_0.S[0,(N-2)] \text{ is } \text{AndArr}_{0:[1,(N-1)]}\\ \text{Mult}_{(AB):0} \text{ connects } \text{AndArr}_{0:0},\ \forall k < (N-1) \rightarrow \text{Mult}_{(AB):k} \text{ connects } X_k.S[0]\\ \forall k \in [(N-1),(2N-1)] \rightarrow \text{Mult}_{(AB)}[(N-1),(2N-1)] \text{ connects } X_{(N-1)}.S\\ \text{Outputs}: M[(N-1):0] = \text{Mult}_{(AB):(2N-1)}...\text{Mult}_{(AB):0}\end{matrix}\right\} \tag{4.15}$$

Now that we have the multiplication as a strict result, we must be able to decrease by factors of the modulo prime, which requires the introduction of a specialised logic cell, the "Piecewise Comparator Reducer." The piecewise comparator checks the multiplicand in segments of bit length of the prime plus an additional bit, so for an N bit prime, the comparator must check (N+1) bits for prime modularity.

$$\begin{pmatrix}\text{Piece}\\\text{Compare}\end{pmatrix} = \left\{\begin{matrix}\text{Inputs}: A = a_N...a_0\\ \text{Subcircuits}: X\ (\text{N}-\text{ADDR}),\ \text{Internal}: \text{Modd}_{(N-1)}...\text{Modd}_0\\ \text{MuxAdd}_{(N-1)}...\text{MuxAdd}_0,\ \text{Subp}_1\text{Subp}_0\\ \text{Constants}: \text{Prime } P = p_{(N-1)}...p_0,\ \text{PrimeDoub } R = r_{(N-1)}...r_0\\ \text{Prime's 2's Compliment } Q = q_{(N-1)}...q_0\\ \text{PrimeDoub's 2's Compliment } T = t_{(N-1)}...t_0\\ \text{MuxAdd connects } T \text{ when } A \geq R \text{ else } Q \text{ when } A \geq P \text{ else } "0"\\ \text{Subp}_1 \text{ connects } '1' \text{ when } A \geq R \text{ else } '0'\\ \text{Subp}_0 \text{ connects } '1' \text{ when } ((A \geq P) \text{ and } (!\text{Subp}_1)) \text{ else } '0'\\ X.A \text{ connects } A,\ X.B \text{ connects MuxAdd},\ X.S \text{ connects Modd}\\ \text{Outputs}: M[(N-1):0] = \text{Modd}_{(N-1)}...\text{Modd}_0,\ \text{Subp}\end{matrix}\right\} \tag{4.16}$$

Now that we have a piecewise comparator, and the linear multiplier, all that remains is to combine them to obtain a modular multiplier, with the additional functionality of producing not just the modulo value (remainder through piecewise division), but also the modulo division value. Before combining them, to make the modular multiplier as self contained as possible in it's logic description, we define a modular reducer by overlaying multiple piecewise comparators, which will be referred to as the DIVD:QR logic block.

$$\begin{pmatrix}\text{DIVD}\\\text{Quotient}\\\text{Remainder}\end{pmatrix} = \left\{\begin{matrix}\text{Inputs}: D = d_{(2N-1)}...d_0\\ \text{Subcircuits}: \forall k \in [(N-1),0] \rightarrow X_k\ \text{(PieceCompare)}\\ \text{Internal}: \mathrm{Q}_{(N-1)}...\mathrm{Q}_0,\ \mathrm{R}_{(N-1)}...\mathrm{R}_0\\ \forall k \in [(N-1),0] \rightarrow X_k.A[N,1]\ \text{connects}\ X_{(k+1)}.M\\ X_k.A[0]\ \text{connects}\ d_k,\ \text{where}\ X_N.M\ \text{is}\ d_{(2N-1)},...,d_N\\ \forall k \in [(N-1),0] \rightarrow Q_k = (X_k.S[0]\ \text{or}\ X_{k-1}.S[1]),\ \text{where}\ X_{-1}.S[1]\ \text{is}\ '0'\\ \text{Outputs}: R[(N-1):0] = X_0.M,\ Q[(N-1):0]\end{matrix}\right\} \quad (4.17)$$

Now we may define the modular multiplier by simply combining the linear multiplier and the DIVD logic blocks, with the multiplicand as the input to the DIVD.

$$\begin{pmatrix}\text{N}\\\text{MOD}\\\text{MULT}\end{pmatrix} = \left\{\begin{matrix}\text{Inputs}: A = a_{(N-1)}...a_0,\ B = b_{(N-1)}...b_0\\ \text{Subcircuits}: M\ (\text{N} - \text{MULT}),\ D\ (\text{DIVDQR})\\ M.A\ \text{connects}\ A,\ M.B\ \text{connects}\ B\\ D.D\ \text{connects}\ M.M,\ D.Q\ \text{connects OPEN}\\ \text{Outputs}: R[(N-1):0] = D.R\end{matrix}\right\} \quad (4.18)$$

4.3 Field Inversion

Now that we have the 256 bit modular adder, subtractor and multiplier, we need to construct the Finite Field equivalent of division; Finite Field Multiplicative Modular Inversion: also known as Field Inversion. This is necessarily a clocked process, and as such the main logic cell will rely on continuously taking outputs and remapping them as inputs to an internal circuit. Firstly we describe the outer logic cell. The presence of the clocking is used in controlling the timing of looping of the 'while' structure.

$$\begin{pmatrix}\text{N}\\\text{MOD}\\\text{INVR}\end{pmatrix} = \left\{\begin{matrix}\text{Input}: A = a_{(N-1)}...a_0,\ \text{Constants}: \text{Prime}\ P = p_{(N-1)}...p_0\\ \text{Internal}: U = u_{(N-1)}...u_0,\ V = v_{(N-1)}...v_0,\ X = x_{(N-1)}...x_0,\ Y = y_{(N-1)}...y_0\\ \text{Subcircuits}: H\ \text{(InverterRound)}\\ \begin{pmatrix}\text{INITIALISE}\\ (U = A)\ \ (X = \ '0...01')\ \ (V = P)\ \ (Y = \ '0...0')\end{pmatrix}\\ \begin{pmatrix}\text{[WHILE [(H.Unext} \neq \ '0...01')\ \text{AND}\ (\text{H.Vnext} \neq \ '0...01')]]\\ (\text{H.U connects H.Unext})\ \ (\text{H.V connects V.Unext})\\ (\text{H.X connects H.Xnext})\ \ (\text{H.Y connects H.Ynext})\end{pmatrix}\\ \begin{pmatrix}\text{[WHILE [(H.Unext} =' 0...01')\ \text{OR}\ (\text{H.Vnext} =' 0...01')]]\\ \text{IF H.Unext} = \ '0...01'\ \ \text{Outputs}:\ \text{E} = \text{H.Xnext}\\ \text{IF H.Vnext} = \ '0...01'\ \ \text{Outputs}:\ \text{E} = \text{H.Ynext}\end{pmatrix}\end{matrix}\right\} \quad (4.19)$$

And now to define the logic of a single inverter round, as described in the chapter on the mathematical background, is a single cycle of the bitwise variety of the Extended Euclidean Algorithm;

$$
\begin{array}{c}
\mathrm{N-INVERTERROUND} = \\
\left\{
\begin{array}{c}
\text{Inputs} : U = u_{(N-1)}...u_0,\ V = v_{(N-1)}...v_0,\ X = x_{(N-1)}...x_0,\ Y = y_{(N-1)}...y_0 \\
\text{Constants} : \text{Prime } P = p_{(N-1)}...p_0 \\
\left(\begin{array}{c}
\text{Internal} : \text{XandPandYneg}[(N-1),0],\ \text{YandPandXneg}[(N-1),0] \\
\text{Unext}[(N-1),0],\ \text{Vnext}[(N-1),0],\ \text{Xnext}[(N-1),0],\ \text{Ynext}[(N-1),0] \\
\text{Uinv} = !U,\ \text{Vinv} = !V,\ \text{Xinv} = !X,\ \text{Yinv} = !Y,\ \text{Uneg}[(N-1),0] \\
\text{Vneg}[(N-1),0],\ \text{Xneg}[(N-1),0],\ \text{Yneg}[(N-1),0],\ \text{VandUneg}[(N-1),0] \\
\text{UandVneg}[(N-1),0],\ \text{XandP}[(N-1),0],\ \text{YandP}[(N-1),0]
\end{array}\right) \\
\left(\begin{array}{c}
\text{Subcircuits} :\ \mathrm{N-ADDR} \\
A_{Uneg},\ A_{Vneg},\ A_{Xneg},\ A_{Yneg},\ A_{VandUneg},\ A_{UandVneg} \\
A_{XandP},\ A_{YandP},\ A_{XandPandYneg},\ A_{YandPandXneg}
\end{array}\right) \\
\left(\begin{array}{cc}
\left(\begin{array}{c}
\forall K \in \{U,V,X,Y\} \\
A_{Kneg}.A \text{ connects Kinv} \\
A_{Kneg}.B \text{ connects } '0...01' \\
A_{Kneg}.S \text{ connects } Kneg
\end{array}\right) &
\left(\begin{array}{c}
\forall K,J \in \{U,V\},\ K \neq J \\
A_{KandJneg}.A \text{ connects K} \\
A_{KandJneg}.B \text{ connects Jneg} \\
A_{KandJneg}.S \text{ connects KandJneg}
\end{array}\right) \\
\left(\begin{array}{c}
\forall K \in \{X,Y\} \\
A_{KandP}.A \text{ connects K} \\
A_{KandP}.B \text{ connects P} \\
A_{KandP}.S \text{ connects KandP}
\end{array}\right) &
\left(\begin{array}{c}
\forall K,J \in \{X,Y\},\ K \neq J \\
A_{KandPandJneg}.A \text{ connects KandP} \\
A_{KandPandJneg}.B \text{ connects Jneg} \\
A_{KandPandJneg}.S \text{ connects KandPandJneg}
\end{array}\right)
\end{array}\right) \\
\left(\begin{array}{c}
[\text{IF}\ [(U[0] =\ '0')\ \text{ OR }\ (V[0] =\ '0')]] \\
\left\{\begin{array}{c}
\text{If } (U[0] =\ '0') \\
\left(\begin{array}{c}
\text{Unext} = \text{RightShift}(U) \\
\left\{\begin{array}{cc}
(\text{If } (X[0] =\ '0')) & \text{Xnext} = \text{RightShift}(X) \\
\text{Else} & \text{Xnext} = \text{RightShift(XandP)}
\end{array}\right\}
\end{array}\right)) \\
\text{If } (V[0] =\ '0') \\
\left(\begin{array}{c}
\text{Vnext} = \text{RightShift}(V) \\
\left\{\begin{array}{cc}
(\text{If } (Y[0] =\ '0'))) & \text{Ynext} = \text{RightShift}(Y) \\
\text{Else} & \text{Ynext} = \text{RightShift(YandP)}
\end{array}\right\}
\end{array}\right)
\end{array}\right\} \\
\text{ELSE} \\
\left\{\begin{array}{c}
\text{If } (U \geqslant V) \\
\left(\begin{array}{c}
\text{Unext} = \text{UandVneg} \\
\left\{\begin{array}{cc}
(\text{If } (X > Y)) & (\text{Xnext} = \text{XandYneg}) \\
\text{Else} & (\text{Xnext} = \text{XandPandYneg})
\end{array}\right\}
\end{array}\right) \\
\text{Else} \\
\left(\begin{array}{c}
\text{Vnext} = \text{VandUneg} \\
\left\{\begin{array}{cc}
(\text{If } (Y > X)) & (\text{Ynext} = \text{YandXneg}) \\
\text{Else} & (\text{Ynext} = \text{YandPandXneg})
\end{array}\right\}
\end{array}\right)
\end{array}\right\}
\end{array}\right) \\
\text{Output} :\ \text{Unext},\ \text{Vnext},\ \text{Xnext},\ \text{Ynext}
\end{array}
\right\}
\end{array}
\tag{4.20}
$$

4.4 Elliptic Curve Group Operation

Now that the N-MODADDR, N-MODSUBT, N-MODMULT and N-MODINVR have been described, we have the necessary building blocks for the logic required to facillitate Elliptic Curve Group Operations. We now describe the logic cells for Point Addition and Point Doubling in Jacobian coordinates, using the ADDR, SUBT and MULT cells.

4.4.1 Point Addition in Jacobian

We begin by recalling the equations for Point Addition in Jacobian coordinates;

$$\left\{\begin{array}{c}
X_3 = \left(\left(Z_2^3Y_1 - Z_1^3Y_2\right)^2 - \left(X_1Z_2^2 + X_2Z_1^2\right)\left(Z_2^2X_1 - Z_1^2X_2\right)^2\right) \\
Y_3 = \left(\left(Z_2^3Y_1 - Z_1^3Y_2\right)\left(X_1Z_2^2\left(Z_2^2X_1 - Z_1^2X_2\right)^2 - X_3\right) - Y_1Z_2^3\left(Z_2^2X_1 - Z_1^2X_2\right)^3\right) \\
Z_3 = \left(Z_1Z_2\left(Z_2^2X_1 - Z_1^2X_2\right)\right)
\end{array}\right\} \tag{4.21}$$

We now write the equations for PA as a set of binary operations: Piecewise Actions;

$$\left\{\begin{array}{cc}
(V_A = (Z_1 * Z_1) = Z_1^2) & (V_B = (Z_2 * Z_2) = Z_2^2) \\
(V_C = (V_A * X_2) = Z_1^2X_2) & (V_D = (V_B * X_1) = Z_2^2X_1) \\
(V_E = (Z_1 * Z_2) = Z_1Z_2) & (V_F = (V_D - V_C) = (Z_2^2X_1 - Z_1^2X_2)) \\
[Z_3 = (V_F * V_E)] & (V_G = (V_C + V_D) = (X_1Z_2^2 + X_2Z_1^2)) \\
(V_H = (V_F * V_F) = (Z_2^2X_1 - Z_1^2X_2)^2) & (V_I = (V_G * V_H)) \\
(V_J = (V_H * V_F) = (Z_2^2X_1 - Z_1^2X_2)^3) & (V_K = (V_D * V_H) = X_1Z_2^2(Z_2^2X_1 - Z_1^2X_2)^2) \\
(V_L = (Y_1 * Z_2) = Y_1Z_2) & (V_M = (Y_2 * Z_1) = Y_2Z_1) \\
(V_N = (V_L * V_B) = Z_2^3Y_1) & (V_O = (V_M * V_A) = Z_1^3Y_2) \\
(V_P = (V_N - V_O) = (Z_2^3Y_1 - Z_1^3Y_2)) & (V_Q = (V_P * V_P) = (Z_2^3Y_1 - Z_1^3Y_2)^2) \\
[X_3 = (V_Q - V_I)] & (V_R = (V_N * V_J) = Y_1Z_2^3(Z_2^2X_1 - Z_1^2X_2)^3) \\
(V_S = (V_K - X_3)) & (V_T = (V_P * V_S)) \\
[Y_3 = (V_T - V_R)] &
\end{array}\right\} \tag{4.22}$$

We can now write these as 'Parallel Piecewise Actions', with 'Staggered Dependencies.'

$$\left\{\begin{array}{c}
\text{Stage X} : \{\text{Action}\} \\
\text{Stage 1}: \ \{V_A, V_B, V_E, V_L, V_M\} \\
\text{Stage 2}: \ \{V_C, V_D, V_N, V_O\} \\
\text{Stage 3}: \ \{V_F, V_G, V_P\} \\
\text{Stage 4}: \ \{Z_3, V_H, V_Q\} \\
\text{Stage 5}: \ \{V_I, V_J, V_K\} \\
\text{Stage 6}: \ \{X_3, V_R\} \\
\text{Stage 7}: \ \{V_S\} \\
\text{Stage 8}: \ \{V_T\} \\
\text{Stage 9}: \ \{Y_3\}
\end{array}\right\} = \left\{\begin{array}{c}
\text{Stage X} : \{\text{MULT}\}[\text{ADDR/SUBT}] \\
\text{Stage 1}: \ \{V_A, V_B, V_E, V_L, V_M\} \\
\text{Stage 2}: \ \{V_C, V_D, V_N, V_O\} \\
\text{Stage 3}: \ [V_F, V_G, V_P] \\
\text{Stage 4}: \ \{Z_3, V_H, V_Q\} \\
\text{Stage 5}: \ \{V_I, V_J, V_K\} \\
\text{Stage 6}: \ \{V_R\}\,[X_3] \\
\text{Stage 7}: \ [V_S] \\
\text{Stage 8}: \ \{V_T\} \\
\text{Stage 9}: \ [Y_3]
\end{array}\right\} \tag{4.23}$$

We can now write the 'Staggered Dependencies' with Dominating MULT operations.

$$\left\{\begin{array}{c}\left\{\begin{array}{c}\{V_A, V_B, V_E, V_L, V_M\} \to \{V_C, V_D, V_N, V_O\} \to [V_F, V_G, V_P] \to \{Z_3, V_H, V_Q\} \\ \to \{V_I, V_J, V_K\} \to \begin{pmatrix}\{V_R\} \\ [X_3] \to [V_S]\end{pmatrix} \to \{V_T\} \to [Y_3]\end{array}\right\} \\ \left\{\begin{array}{c}\{V_A, V_B, V_E, V_L, V_M\} \to \{V_C, V_D, V_N, V_O\} \to [V_F, V_G, V_P] \to \{Z_3, V_H, V_Q\} \\ \to \{V_I, V_J, V_K\} \to [X_3] \to [V_S] \to \{V_R, V_T\} \to [Y_3]\end{array}\right\}\end{array}\right\} \tag{4.24}$$

Which we can now use to construct the Jacobian PA logic cell;

$$\mathrm{N-JPA} = \left\{\begin{array}{c}\text{Inputs}: P[(3N-1),0] = P.X[(N-1),0], P.Y[(N-1),0], P.Z[(N-1),0] \\ \text{Inputs}: Q[(3N-1),0] = Q.X[(N-1),0], Q.Y[(N-1),0], Q.Z[(N-1),0] \\ \text{Internal}: A,\ B,\ C,\ D,\ E,\ F,\ G,\ H,\ I,\ J,\ K,\ L,\ M,\ N,\ O,\ P,\ Q,\ R,\ S,\ T,\ X,\ Y,\ Z \\ \text{Subcircuits}:\ \mathrm{N-MODMULTs};\ V_A,\ V_B,\ V_C,\ V_D,\ V_E,\ V_H,\ V_I,\ V_J,\ V_K \\ \text{Subcircuits}:\ \mathrm{N-MODMULTs};\ V_L,\ V_M,\ V_N,\ V_O,\ V_Q,\ V_R,\ V_T,\ Z_3 \\ \text{Subcircuits}:\ \mathrm{N-MODSUBTs};\ V_F,\ V_P,\ V_S,\ X_3,\ Y_3 \\ \text{Subcircuits}:\ \mathrm{N-MODADDRs};\ V_G \\ V_A.A \to P.Z,\ V_A.B \to P.Z,\ V_A.R \to A,\ V_B.A \to Q.Z,\ V_B.B \to Q.Z,\ V_B.R \to B \\ V_C.A \to A,\ V_C.B \to Q.X,\ V_C.R \to C,\ V_D.A \to B,\ V_D.B \to P.X,\ V_D.R \to D \\ V_E.A \to P.Z,\ V_E.B \to Q.Z,\ V_E.R \to E,\ V_H.A \to F,\ V_H.B \to F,\ V_H.R \to H \\ V_I.A \to G,\ V_I.B \to H,\ V_I.R \to I,\ V_J.A \to H,\ V_J.B \to F,\ V_J.R \to J \\ V_K.A \to D,\ V_K.B \to H,\ V_K.R \to K,\ V_L.A \to P.Y,\ V_L.B \to Q.Z,\ V_L.R \to L \\ V_M.A \to Q.Y,\ V_M.B \to P.Z,\ V_M.R \to M,\ V_N.A \to L,\ V_N.B \to B,\ V_N.R \to N \\ V_O.A \to M,\ V_O.B \to A,\ V_O.R \to O,\ V_Q.A \to P,\ V_Q.B \to P,\ V_Q.R \to Q \\ V_R.A \to N,\ V_R.B \to J,\ V_R.R \to R,\ V_T.A \to P,\ V_T.B \to S,\ V_T.R \to T \\ Z_3.A \to F,\ Z_3.B \to E,\ Z_3.R \to Z,\ V_G.A \to C,\ V_G.B \to D,\ V_G.S \to G \\ V_F.A \to D,\ V_F.B \to C,\ V_F.D \to F,\ V_P.A \to N,\ V_P.B \to O,\ V_P.D \to P \\ V_S.A \to K,\ V_S.B \to X_3,\ V_S.D \to S,\ X_3.A \to Q,\ X_3.B \to I,\ X_3.D \to X \\ Y_3.A \to T,\ Y_3.B \to R,\ Y_3.D \to Y \\ \text{Output}: W = X, Y, Z\end{array}\right\} \tag{4.25}$$

4.4.2 Point Doubling in Jacobian

We begin by recalling the equations for Point Doubling in Jacobian coordinates;

$$\left\{\begin{array}{c}X_3 = \left((3X_1^2 + aZ_1^4)^2 - 8Y_1^2X_1\right) \\ Y_3 = \left((3X_1^2 + aZ_1^4)^1 (4Y_1^2X_1 - X_3)^1 - 8Y_1^4\right) \\ Z_3 = (2Y_1Z_1)\end{array}\right\} \tag{4.26}$$

We now write the equations for PD as a set of binary operations: Piecewise Actions;

$$\left\{\begin{array}{cc}
(V_A = (Y_1 * Z_1) = Y_1Z_1) & [Z_3 = (V_A + V_A) = 2Y_1Z_1] \\
(V_B = (X_1 * X_1) = X_1^2) & (V_C = (Y_1 * Y_1) = Y_1^2) \\
(V_D = (Z_1 * Z_1) = Z_1^2) & (V_E = (V_D * V_D) = Z_1^4) \\
(V_F = (V_C * V_C) = Y_1^4) & (V_G = (V_F + V_F) = 2Y_1^4) \\
(V_H = (V_G + V_G) = 4Y_1^4) & (V_I = (V_H + V_H) = 8Y_1^4) \\
(V_J = (a * V_E) = aZ_1^4) & (V_K = (V_B + V_B) = 2X_1^2) \\
(V_L = (V_K + V_B) = 3X_1^2) & (V_M = (V_L + V_J) = (3X_1^2 + aZ_1^4)) \\
(V_N = (V_M * V_M)) & (V_O = (X_1 + X_1) = 2X_1) \\
(V_P = (V_O + V_O) = 4X_1) & (V_Q = (V_P + V_P) = 8X_1) \\
(V_R = (V_Q * V_C) = 8Y_1^2X_1) & [X_3 = (V_N - V_R)] \\
(V_S = (V_P * V_C) = 4Y_1^2X_1) & (V_T = (V_S - X_3)) \\
(V_U = (V_T * V_M)) & [Y_3 = (V_U - V_I)]
\end{array}\right\} \tag{4.27}$$

We can now write these as 'Parallel Piecewise Actions', with 'Staggered Dependencies.'

$$\left\{\begin{array}{c}
\text{Stage X : }\{\text{Action}\} \\
\text{Stage 1 : }\{V_A, V_B, V_C, V_D, V_O\} \\
\text{Stage 2 : }\{Z_3, V_E, V_F, V_K, V_P\} \\
\text{Stage 3 : }\{V_G, V_J, V_L, V_Q, V_S\} \\
\text{Stage 4 : }\{V_H, V_M, V_R\} \\
\text{Stage 5 : }\{V_I, V_N\} \\
\text{Stage 6 : }\{X_3\} \\
\text{Stage 7 : }\{V_T\} \\
\text{Stage 8 : }\{V_U\} \\
\text{Stage 9 : }\{Y_3\}
\end{array}\right\} = \left\{\begin{array}{c}
\text{Stage X : }\{\text{MULT}\}[\text{ADDR/SUBT}] \\
\text{Stage 1 : }\{V_A, V_B, V_C, V_D\}\,[V_O] \\
\text{Stage 2 : }\{V_E, V_F\}\,[Z_3, V_K, V_P] \\
\text{Stage 3 : }\{V_J, V_S\}\,[V_G, V_L, V_Q] \\
\text{Stage 4 : }\{V_R\}\,[V_H, V_M] \\
\text{Stage 5 : }\{V_N\}\,[V_I] \\
\text{Stage 6 : }[X_3] \\
\text{Stage 7 : }[V_T] \\
\text{Stage 8 : }\{V_U\} \\
\text{Stage 9 : }[Y_3]
\end{array}\right\} \tag{4.28}$$

We can now write the 'Staggered Dependencies' with Dominating MULT operations.

$$\left\{\begin{array}{c}
\left(\begin{array}{c}\{V_A, V_B, V_C, V_D\} \\ [V_O] \to [V_P] \to [V_Q]\end{array}\right) \to \left(\begin{array}{c}\{V_E, V_F, V_S, V_R\} \\ [Z_3, V_K] \to [V_L]\end{array}\right) \to \left(\begin{array}{c}\{V_J\} \\ [V_G] \to [V_H] \to [V_I]\end{array}\right) \\
\to [V_M] \to \{V_N\} \to [X_3] \to [V_T] \to \{V_U\} \to [Y_3]
\end{array}\right\} \tag{4.29}$$

Which we can now use to construct the Jacobian PD logic cell;

$$\text{N}-\text{JPD} = \left\{\begin{array}{c} \text{Inputs}: P[(3N-1),0] = P.X[(N-1),0], P.Y[(N-1),0], P.Z[(N-1),0] \\ \text{Constant}: ECCA[(N-1),0] \\ \text{Internal}: A,\ B,\ C,\ D,\ E,\ F,\ G,\ H,\ I,\ J,\ K,\ L,\ M,\ N,\ O,\ P,\ Q,\ R,\ S,\ T,\ U,\ X,\ Y,\ Z \\ \text{Subcircuits}:\ \text{N}-\text{MODMULTs};\ V_A,\ V_B,\ V_C,\ V_D,\ V_E,\ V_F,\ V_J,\ V_N,\ V_R,\ V_S,\ V_U \\ \text{Subcircuits}:\ \text{N}-\text{MODADDRs};\ V_G,\ V_H,\ V_I,\ V_K,\ V_L,\ V_M,\ V_O,\ V_P,\ V_Q,\ Z_3 \\ \text{Subcircuits}:\ \text{N}-\text{MODSUBTs};\ V_T,\ X_3,\ Y_3 \\ V_A.A \to P.Y,\ V_A.B \to P.Z,\ V_A.R \to A,\ V_B.A \to P.X,\ V_B.B \to P.X,\ V_B.R \to B \\ V_C.A \to P.Y,\ V_C.B \to P.Y,\ V_C.R \to C,\ V_D.A \to P.Z,\ V_D.B \to P.Z,\ V_D.R \to D \\ V_E.A \to D,\ V_E.B \to D,\ V_E.R \to E,\ V_F.A \to C,\ V_F.B \to C,\ V_F.R \to F \\ V_J.A \to NISTA,\ V_J.B \to E,\ V_J.R \to J,\ V_N.A \to M,\ V_N.B \to M,\ V_N.R \to N \\ V_R.A \to Q,\ V_R.B \to C,\ V_R.R \to R,\ V_S.A \to P,\ V_S.B \to C,\ V_S.R \to S \\ V_U.A \to T,\ V_U.B \to M,\ V_U.R \to U,\ V_G.A \to F,\ V_G.B \to F,\ V_G.S \to G \\ V_H.A \to G,\ V_H.B \to G,\ V_H.S \to H,\ V_I.A \to H,\ V_I.B \to H,\ V_I.S \to I \\ V_K.A \to B,\ V_K.B \to B,\ V_K.S \to K,\ V_L.A \to K,\ V_L.B \to B,\ V_L.S \to L \\ V_M.A \to L,\ V_M.B \to J,\ V_M.S \to M,\ V_O.A \to P.X,\ V_O.B \to P.X,\ V_O.S \to O \\ V_P.A \to O,\ V_P.B \to O,\ V_P.S \to P,\ V_Q.A \to P,\ V_Q.B \to P,\ V_Q.S \to Q \\ Z_3.A \to A,\ Z_3.B \to A,\ Z_3.S \to Z,\ V_T.A \to S,\ V_T.B \to X_3,\ V_T.D \to T \\ X_3.A \to N,\ X_3.B \to R,\ X_3.D \to X,\ Y_3.A \to U,\ Y_3.B \to I,\ Y_3.D \to Y \\ \text{Output}: W = X, Y, Z \end{array}\right\} \tag{4.30}$$

4.5 Elliptic Curve Point Multiplication: Jacobian

Now that we have described both the PA and PD logic in Jacobian coordinates and given their logic cells, we may now construct a logic cell for PM in Jacobian coordinates;

$$\text{N}-\text{JPM} = \left\{\begin{array}{c} \text{Inputs}: K[(N-1),0],\ P.X[(N-1),0],\ P.Y[(N-1),0],\ P.Z[(N-1),0] \\ \text{Subcircuits}: X\ N-JPA,\ Y\ N-JPD \\ \text{Internal}: A[(3N-1),0] = A.X[(N-1),0], A.Y[(N-1),0], A.Z[(N-1),0] \\ \text{Internal}: D[(3N-1),0] = D.X[(N-1),0], D.Y[(N-1),0], D.Z[(N-1),0] \\ Y.P \text{ connects } D,\ X.P \text{ connects } D,\ X.Q \text{ connects } A \\ \text{Initialise}: D = P,\ A = '0...01', '0...01', '0...0' \\ \left((\forall j \in [0,(N-1)]) \begin{pmatrix} \text{IF } K[j] = '1' & A \leftarrow X.W \\ \text{ALWAYS} & D \leftarrow Y.W \end{pmatrix} \right) \\ \text{Output}: M_X = A.X,\ M_Y = A.Y,\ M_Z = A.Z \end{array}\right\} \tag{4.31}$$

4.6 Elliptic Curve Point Multiplication: Affine

Now that we have the logic cell for Jacobian PM, we can use this with the Field Inversion logic cell to construct a logic cell for PM in Affine coordinates;

$$\begin{Bmatrix} \mathrm{N-APM} = \\ \text{Inputs} : K[(N-1),0],\ P[511,0] = P.X[255,0], P.Y[(N-1),0] \\ \text{Subcircuits} : X\ \mathrm{N-JPM},\ Y\ \mathrm{N-MODINVR} \\ Z_2\ \mathrm{N-MODMULT},\ Z_3\ \mathrm{N-MODMULT} \\ W_X\ \mathrm{N-MODMULT},\ W_Y\ \mathrm{N-MODMULT} \\ X.K \text{ connects } K,\ X.P \text{ connects } P,' 0...01' \\ Y.A \text{ connects } X.M_Z,\ Z_2.A \text{ connects } Y.E,\ Z_2.B \text{ connects } Y.E \\ Z_3.A \text{ connects } Z_2.M,\ Z_3.B \text{ connects } Y.E \\ W_X.A \text{ connects } X.M_X,\ W_X.B \text{ connects } Z_2.M,\ W_X.S \text{ connects } M_X \\ W_Y.A \text{ connects } X.M_Y,\ W_Y.B \text{ connects } Z_3.M,\ W_Y.S \text{ connects } M_Y \\ \text{Output} : M = M_X, M_Y \end{Bmatrix} \tag{4.32}$$

We have now now provided Logic Cell Models for Field Arithmetic Processes; Addition, Subtraction, Multiplication, Inversion: as well as ECGO functionality of PA and PD, used to implement a logic cell model for PM in Jacobian and then Affine coordinates. These models serve as a half-way point between the theory and the implementation. Their description and validity is reliant on the theory behind ECC, however from this point forward, the implementations of protocols will relate to these logic cell models, and transporting them into a HDL setting.

Chapter 5

VHDL Implementation

5.1 Chapter Introduction

This chapter utilises the models presented in 4 to develop their implementation in VHDL. Relying heavily on the models previously discussed, this chapter is devoted to the encapsulation of those models for a HDL implementation, and how these implementations sufficiently produce the expectation of the modelling results. First to be implemented will be the Field Arithmetic Primitives of Field operations; Addition, Subtraction, Multiplication. This is followed by the implementation of the Field Inversion operation. The Field Arithmetic Primitives are then used to devise the implementations of the Elliptic Curve Group Operation, which is then extended to the Elliptic Curve Group Action, in the context of the Jacobian Variety, followed by the Affine Variety.

5.2 Evolution of Code Convention

Whilst the modelling chapter reflected the old design, and was not replaced with the newer design given their similarity, this is also because the final design will be presented here, and can be understood from the HDL context rather than the cell logic context. To explain the naming convention used here, the Field Arithmetic Primitives in their design comprise the Field Arithmetic Processor (FAP) designated units, which are then used by the ECC designated units. The naming convention of referring to the cells with the prefix of "GENERIC" is a reflection on the several design stages, for which each design was translated into a different format, starting with the structural "STRUC" units, numeric "NUMER" units and example "EXAMP" units. Here, we only present the finished product, the Generic units, after first discussing the evolution from STRUC to NUMER to GENERIC.

The main measure that brings difference to the different design strategies is in the Port declaration of the entities. STRUC units were based on the initial design goal of a fixed length optimised processor, and hence they all involved static Port declarations with fixed length IO, giving all components a static interface that necessarily matched. NUMER units were constructed during the first attempt to, and lean in the direction of obtaining, generalisability. They utilised a user defined package that contains a variable which specifies a target bit-length, however they inherited much of the entity architecture of the STRUC units, and the variable bit length was constrained to values larger than 16 that were divisible by 4. This gave them the potential to have variable length IO ports, however they all still only employed Port declarations, and so on component declaration had only static interfaces, despite that all components would simultaneously share the same variable length IO, and hence have matching interfaces, the interfaces within components was static. EXAMP units were the original incarnations of the smaller sized units that matched the same internal architecture as the STRUC units, but on a smaller scale, fixed to a bit length of 5, using the worked example of an Elliptic Curve Group presented in Chapter 2, however their behavior began to be differentiable from the behavior of the NUMER units.

The GENERIC units were the final resolution to the need to become absolutely generalisable, and enforce consistent behavior between the actual and test simulations. Their name is indicative of their ascension from the NUMER code to a more generically resizable code. They introduced and made extensive use of "Generic" declarations in entity declarations. By having a generic declaration which includes reference to a universally controlled variable to maintain consistent IO lengths and unit interfaces, it is possible to enforce that all units will have matching IO lengths globally, however it is also possible for several top level units with different generic map values to be utilised simultaneously, to compare different behavior for different sizes of the same unit, in a single simulation. However, the main benefit of employing generic maps is that it provides a means for self referentiallity. This can be exploited to make units that recursively call themselves, which, along with the benefit of "if ... generate" commands, provides a means of designing logic descriptions that can decide from some parameter and their own length, whether to split their lengths somehow, and mimic their own design on a smaller scale, or to terminate at their current length.

This has been used to create the final design, which uses units with generic entity declaration and matches it to generic maps in component calls, in both length matching, and length variable calls. With regards to units that call themselves recursively, they employ a naming convention of "Splitting Length" and "Terminal Length" "if ... generate" clauses. They both utilise the same variable, defined in the VECTOR_STANDARD package under the name "MultLen" as the measure for whether they should split or terminate. This, along with the other variable "VecLen" which defines the total bit length being used, provides the means to produce units which rely on these two, and to simply use them as global control variables to determine the best "terminal length" configuration for some bit length. Despite that there is a significant amount of code, given that it is the goal and result of this project, it is not deferred to an appendix, but presented here in this chapter, although some code has been clipped to fit line lengths.

5.3 Package Declarations

The design uses two project defined packages, which contain variables used in both generic entity declarations, and as data for Elliptic Curve Domain Parameter (ECDP) sets. The VECTOR_STANDARD package defines constant STD_LOGIC_VECTOR values, whilst the ECC_STANDARD contains ECDP sets that are used in designing the protocol units.

5.3.1 VECTOR STANDARD

This package defines two global control variables, the bit-length controlling VecLen, and terminal case bit-length selecting MultLen, along with three STD_LOGIC_VECTORs defined by the VecLen variable; numeric representations of the 'unit' and 'zero' vectors, and an all impedance driving vector.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
--A change file that declares basic vectors, up to a specific length
package VECTOR_STANDARD is
--Change this to define the standard length used
constant VecLen : natural := 112;
--A VecLen bit vector populated by only zeroes
constant ZeroVector : STD_LOGIC_VECTOR ((VecLen - 1) downto 0)
                := (others => '0');
--A VecLen bit vector populated by only zeroes except the first bit
constant UnitVector : STD_LOGIC_VECTOR ((VecLen - 1) downto 0)
                := (0 => '1', others => '0');
--A VecLen bit vector populated by only 'Z's, to drive a high impedence.
constant ImpedeVector : STD_LOGIC_VECTOR ((VecLen - 1) downto 0)
                := (others => 'Z');
--Used in determining the terminal length of multipliers.
constant MultLen : natural := 7;
end VECTOR_STANDARD;
package body VECTOR_STANDARD is
end VECTOR_STANDARD;
```

5.3.2 ECC STANDARD

This package defines several array types, each an array of 6 elements, all of the same bit length. These types are then used to instantiate constant values of the defined types, such that each instance of a particular array type is given the name of the curve of which it contains the ECDP set of, with each set containing the curve's Prime, equation defining a and b values, the curve's base point Gx and Gy, and the order, N, of the base point.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.VECTOR_STANDARD.ALL;

--A package containing declarations of constant parameter sets
--Of Elliptic Curve Domain Parameters.
package ECC_STANDARD is

-----------------------------------------------------
-----------Declare types of generic size!-----------
-----------------------------------------------------

type ECC_Parameters_521 is array (0 to 5) of STD_LOGIC_VECTOR(520 downto 0);
type ECC_Parameters_384 is array (0 to 5) of STD_LOGIC_VECTOR(383 downto 0);
type ECC_Parameters_256 is array (0 to 5) of STD_LOGIC_VECTOR(255 downto 0);
type ECC_Parameters_224 is array (0 to 5) of STD_LOGIC_VECTOR(223 downto 0);
type ECC_Parameters_192 is array (0 to 5) of STD_LOGIC_VECTOR(191 downto 0);
type ECC_Parameters_160 is array (0 to 5) of STD_LOGIC_VECTOR(159 downto 0);
type ECC_Parameters_128 is array (0 to 5) of STD_LOGIC_VECTOR(127 downto 0);
type ECC_Parameters_112 is array (0 to 5) of STD_LOGIC_VECTOR(111 downto 0);
type ECC_Parameters_5 is array (0 to 5) of STD_LOGIC_VECTOR(4 downto 0);

----------------------------------
-----------Naming model-----------
----------------------------------

----Curve_Name
--constant Curve_Name : ECC_Parameters_BitLength := (
--X"Hex String", --Prime
--X"Hex String", --A
--X"Hex String", --B
--X"Hex String", --GX
--X"Hex String", --GY
--X"Hex String");--N

-------------------------------------
-----------List of Curves!-----------
-------------------------------------

--Curves whichs were implemented have been omitted to save space
constant SECp521r1 : ECC_Parameters_521;
constant SECp384r1 : ECC_Parameters_384;
constant SECp256r1 : ECC_Parameters_256;
constant SECp256k1 : ECC_Parameters_256;
constant SECp224r1 : ECC_Parameters_224;
constant SECp192k1 : ECC_Parameters_192;
constant SECp192r1 : ECC_Parameters_192;
constant SECp128r1 : ECC_Parameters_128;
constant SECp128r2 : ECC_Parameters_128;
constant SECp112r1 : ECC_Parameters_112;
--SECp224k1: Not implemented, N is longer in bitlength than the prime.
--SECp160k1: Not implemented, N is longer in bitlength than the prime.
--SECp160r1: Not implemented, N is longer in bitlength than the prime.
--SECp160r2: Not implemented, N is longer in bitlength than the prime.

--Example of a curve entered with parameters in Hex strings
constant SECp112r2 : ECC_Parameters_112 := (
X"DB7C_2ABF_62E3_5E66_8076_BEAD_208B", --Prime
X"6127_C24C_05F3_8A0A_AAF6_5C0E_F02C", --A
X"51DE_F181_5DB5_ED74_FCC3_4C85_D709", --B
X"4BA3_0AB5_E892_B4E1_649D_D092_8643", --GX
X"ADCD_46F5_882E_3747_DEF3_6E95_6E97", --GY
X"36DF_0AAF_D8B8_D759_7CA1_0520_D04B");

--Example of a curve entered with parameters in binary strings
constant M17 : ECC_Parameters_5 := (
"10001", --Prime
"00010", --A
"00010", --B
"00111", --GX
"00110", --GY
"10011");--N

end ECC_STANDARD;
package body ECC_STANDARD is
end ECC_STANDARD;
```

5.4 FAP: Equality and Inequality

This section contains the code used in combinatorial logic realisation of a single bit representation for whether or not one STD_LOGIC_VECTOR is equal to another or not, and another single bit representation of whether one STD_LOGIC_VECTOR is greater than another or not. As it did not fit elsewhere, this section also includes the description of the RAM logic used to latch data from IO pins.

5.4.1 Equality Test

The unit that tests for equality between two STD_LOGIC_VECTOR simply utilises repeated and nested XNOR logic to determine whether they are equal bitwise, where each bit is xnor'd with the corresponding bit position, and all the outputs from the xnor's are AND'd together, to determine that all bits are bitwise equal.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_EQUALITY is
         Generic (N : natural := VecLen); --
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC);
end GENERIC_FAP_EQUALITY;

architecture Behavioral of GENERIC_FAP_EQUALITY is

component GENERIC_FAP_EQUALITY
        generic (N : natural);
        port (A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
         B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
         E : out  STD_LOGIC);
end component;

signal EQBits : STD_LOGIC_VECTOR (2 downto 0);

begin

terminal_structure_1 : if (N = 1) generate
begin
        E <= A(0) xnor B(0);
end generate terminal_structure_1;
terminal_structure_2 : if (N = 2) generate
begin
        E <= (A(0) xnor B(0)) and (A(1) xnor B(1));
end generate terminal_structure_2;

terminal_structure_3 : if (N = 3) generate
begin
        E <= (A(0) xnor B(0)) and (A(1) xnor B(1)) and (A(2) xnor B(2));
end generate terminal_structure_3;
```

```
recursive_structure_trip : if (((N mod 3) = 0) and (N > 3)) generate
begin
        GFE_RSE_Lower : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A(((N/3)-1) downto 0),
                                B => B(((N/3)-1) downto 0),
                                E => EQBits(0));
        GFE_RSE_Middr : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A(((2*N/3)-1) downto (N/3)),
                                B => B(((2*N/3)-1) downto (N/3)),
                                E => EQBits(1));
        GFE_RSE_Upper : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A((N-1) downto (2*N/3)),
                                B => B((N-1) downto (2*N/3)),
                                E => EQBits(2));
        E <= EQBits(0) and EQBits(1) and EQBits(2);
end generate recursive_structure_trip;

recursive_structure_trip_uni : if (((N mod 3) = 1) and (N > 3)) generate
begin
        GFE_RSE_Lower : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A((((N-1)/3)-1) downto 0),
                                B => B((((N-1)/3)-1) downto 0),
                                E => EQBits(0));
        GFE_RSE_Middr : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A(((2*(N-1)/3)-1) downto ((N-1)/3)),
                                B => B(((2*(N-1)/3)-1) downto ((N-1)/3)),
                                E => EQBits(1));
        GFE_RSE_Upper : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A(((N-1)-1) downto (2*(N-1)/3)),
                                B => B(((N-1)-1) downto (2*(N-1)/3)),
                                E => EQBits(2));
        E <= EQBits(0) and EQBits(1) and EQBits(2) and (A(N-1) xnor B(N-1));
end generate recursive_structure_trip_uni;

recursive_structure_trip_duo : if (((N mod 3) = 2) and (N > 3)) generate
begin
```

```
        GFE_RSE_Lower : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A((((N-2)/3)-1) downto 0),
                                B => B((((N-2)/3)-1) downto 0),
                                E => EQBits(0));
        GFE_RSE_Middr : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A(((2*(N-2)/3)-1) downto ((N-2)/3)),
                                B => B(((2*(N-2)/3)-1) downto ((N-2)/3)),
                                E => EQBits(1));
        GFE_RSE_Upper : GENERIC_FAP_EQUALITY
                generic map (N  => (N/3))
                port map (A => A(((N-2)-1) downto (2*(N-2)/3)),
                                B => B(((N-2)-1) downto (2*(N-2)/3)),
                                E => EQBits(2));
        E <= EQBits(0) and EQBits(1) and EQBits(2) and (A(N-1)
                 xnor B(N-1)) and (A(N-2) xnor B(N-2));
end generate recursive_structure_trip_duo;

end Behavioral;
```

5.4.2 Inequality Test: Outer Cell

The unit that tests for inequality between two STD_LOGIC_VECTOR requires an 'outer' and an 'inner' cell, where in the inner cell can call itself recursively but the outer cell cannot call itself, however calls the inner cell. This is such that the inner cell handles the case-wise calculation of whether or not one particular region of bits is greater than another or not, while the outer cell determines all of the bit-wise equalities and hands this data to each call of the inner cell.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_INEQUALITY_OUTER is
          Generic (N : natural := VecLen); --
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           G : out  STD_LOGIC;
                          E : out  STD_LOGIC);
end GENERIC_FAP_INEQUALITY_OUTER;

architecture Behavioral of GENERIC_FAP_INEQUALITY_OUTER is
```

```
signal EquiAbove : STD_LOGIC_VECTOR (((((N/16)+1)*16)-1) downto 0);
signal XNORTable : STD_LOGIC_VECTOR (((((N/16)+1)*16)-1) downto 0);
signal ANDTable1 : STD_LOGIC_VECTOR (((((N/16)+1)*8)) downto 0);
signal ANDTable2 : STD_LOGIC_VECTOR (((((N/16)+1)*4)) downto 0);
signal ANDTable3 : STD_LOGIC_VECTOR (((((N/16)+1)*2)) downto 0);
signal ANDTable4 : STD_LOGIC_VECTOR ((N/16)+1 downto 0);
-- serves to return a logic value from a logical index
signal IndexR : STD_LOGIC_VECTOR (1 downto 0);

component GENERIC_FAP_INEQUALITY_INNER
         Generic (N : natural := VecLen); --
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           EquiAbove : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           G : out  STD_LOGIC);
end component;

begin

ANDTable4(N/16 + 1) <= '1';
ANDTable3((((N/16)+1)*2)) <= '1';
ANDTable2((((N/16)+1)*4)) <= '1';
ANDTable1((((N/16)+1)*8)) <= '1';
IndexR(0) <= '0'; IndexR(1) <= '1';

EquiAboveGen : for K in 0 to ((((N/16)+1)*16)-1) generate
begin
        EquiAboveGenLess : if (K < N) generate --If in actual range
        begin
                XNORTable(K) <= A(K) xnor B(K); --use natural
        end generate EquiAboveGenLess;

        EquiAboveGenGrEq : if (K >= N) generate --If above actual range
        begin
                XNORTable(K) <= '1'; --use padding
        end generate EquiAboveGenGrEq;

        AND1Gen : if ((K mod 2) = 0) generate
        begin
                ANDTable1(K/2) <= XNORTable(K) and XNORTable(K+1);
        end generate AND1Gen;
        AND2Gen : if ((K mod 4) = 0) generate
        begin
                ANDTable2(K/4) <= ANDTable1(K/2) and ANDTable1((K/2)+1);
        end generate AND2Gen;

        AND3Gen : if ((K mod 8) = 0) generate
        begin
                ANDTable3(K/8) <= ANDTable2(K/4) and ANDTable2((K/4)+1);
        end generate AND3Gen;

        AND4Gen : if ((K mod 16) = 0) generate
        begin
                ANDTable4(K/16) <= (ANDTable3(K/8) and ANDTable3((K/8)+1))
                                and ANDTable4((K/16)+1);
        end generate AND4Gen;
end generate EquiAboveGen;

AETGen0 : for K in 0 to (N/16) generate
begin
AETGen1 : for J in 0 to 1 generate
begin
AETGen2 : for H in 0 to 1 generate
begin
AETGen3 : for G in 0 to 1 generate
begin
AETGen4 : for L in 0 to 1 generate
begin
AETIFGen1 : if ((J=0) and (H=0) and (G=0) and (L=0)) generate
begin
        EquiAbove(K*16) <= ANDTable4(K);
end generate AETIFGen1;
AETIFGen2 : if ((J=1) or (H=1) or (G=1) or (L=1)) generate
begin
        EquiAbove(K*16+J*8+H*4+G*2+L) <= (((XNORTable(K*16+J*8+H*4+G*2+L)
                and ANDTable1(K*8+J*4+H*2+G+L)) and (((not IndexR(G)) and
                (not IndexR(L)) and ANDTable2(K*4+J*2+H)) or
                ((IndexR(G) or IndexR(L)) and ANDTable2(K*4+J*2+H+1))))
                and ((((not IndexR(H)) and (not IndexR(G)) and (not IndexR(L))
                and ANDTable3(K*2+J)) or (((IndexR(H)) or (IndexR(G)) or (IndexR(L)))
                and ANDTable3(K*2+J+1))) and ANDTable4(K+1)));
end generate AETIFGen2;
end generate AETGen4;
end generate AETGen3;
end generate AETGen2;
end generate AETGen1;
end generate AETGen0;

E <= EquiAbove(0);

--The above generates the EquiAbove vector to hand to
--the inner cell which calls itself recursively,
--to prevent the inner cell from generating an entire
--equality check structure for each bit length
GFII : GENERIC_FAP_INEQUALITY_INNER
         Generic map (N => N)
    Port map (A => A,
                                  B => B,
                                  EquiAbove => EquiAbove(N-1 downto 0),
                                  G => G);

end Behavioral;
```

5.4.3 Inequality Test: Inner Cell

The inner cell of the inequality testing takes the bit-wise equality evaluation and recursively hands this data to itself, to prevent itself from recalculating all of the upper equalities each call to itself.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_INEQUALITY_INNER is
         Generic (N : natural := VecLen); --
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           EquiAbove : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           G : out  STD_LOGIC);
end GENERIC_FAP_INEQUALITY_INNER;

architecture Behavioral of GENERIC_FAP_INEQUALITY_INNER is
```

```
component GENERIC_FAP_INEQUALITY_INNER
         Generic (N : natural := VecLen); --
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           EquiAbove : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           G : out  STD_LOGIC);
end component;

signal GBit : STD_LOGIC;

begin

terminal_structure_1 : if (N = 1) generate
begin
        G <= (A(0) and (not B(0)));
end generate terminal_structure_1;

terminal_structure_2 : if (N = 2) generate
begin
        G <= (((A(0) and (not B(0))) and EquiAbove(1)) or ((A(1) and (not B(1)))));
end generate terminal_structure_2;

terminal_structure_3 : if (N = 3) generate
begin
        G <= (((A(0) and (not B(0))) and EquiAbove(1)) or ((A(1)
                and (not B(1))) and EquiAbove(2)) or ((A(2) and (not B(2)))));
end generate terminal_structure_3;

recursive_structure : if (N > 3) generate
begin
GFIII : GENERIC_FAP_INEQUALITY_INNER
        generic map (N  => (N-3))
        port map (A => A((N-1) downto 3),
                        B => B((N-1) downto 3),
                        EquiAbove => EquiAbove((N-1) downto 3),
                        G => GBit);
        G <= (((A(0) and (not B(0))) and EquiAbove(1)) or ((A(1) and (not B(1)))
        and EquiAbove(2)) or ((A(2) and (not B(2))) and EquiAbove(3)) or GBit);
end generate recursive_structure;

end Behavioral;
```

5.4.4 Relator Cell

The relator cells uses both the equality and inequality cell as components, and relies on a generic numerical input to control which cell is connected upwards via port mapping to the inputs and outputs of the relator cell.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_RELATIONAL is
         Generic (N : Natural := VecLen;
                         --0 for just equality, 1 for Greater Than test
                         VType : Natural := 1);   : Default 1
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end GENERIC_FAP_RELATIONAL;

architecture Behavioral of GENERIC_FAP_RELATIONAL is

component GENERIC_FAP_INEQUALITY_OUTER
         Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           G : out  STD_LOGIC;
                   E : out  STD_LOGIC);
end component;
component GENERIC_FAP_EQUALITY
         Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC);
end component;

begin

EquiaGen : if (VType = 0) generate --make a copy of the equality checker
begin
        X : GENERIC_FAP_EQUALITY generic map (N => N)
                port map (A => A, B => B, E => E);
        G <= '0';
end generate EquiaGen;
```

```
GreatGen : if (VType = 1) generate --make a copy of the inequality checker
begin
        X : GENERIC_FAP_INEQUALITY_OUTER
                generic map (N => N)
                port map (A => A, B => B, G => G, E => E);
end generate GreatGen;

end Behavioral;
```

5.4.5 Ram Utility Cell

The RAM cell utilises an INOUT port as a data bus to simulate the behavior of isolated registers under the assumption that they behave as RAM blocks.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_UTIL_RAM_CLOCKED is
         Generic (N : natural := VecLen);
    Port ( RW : in  STD_LOGIC;
                --RW is Writing High, Reading Low.
                --To block 'X' collisions on rising_edge(CLK),
                --Reading is only enhabled if Data is latched with (others => 'Z')
           Data : inout  STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
                          CLK : in STD_LOGIC);
end GENERIC_UTIL_RAM_CLOCKED;

architecture Behavioral of GENERIC_UTIL_RAM_CLOCKED is

--Guarantees that a consistent length Impede Vector is available.
constant ImpedeInternal : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => 'Z');

signal DataInternal : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');

begin
```

```
process(CLK)
begin
        if (rising_edge(CLK)) then
                if (RW = '1') then
                        DataInternal <= Data;
                        Data <= ImpedeInternal;
                elsif ((RW = '0') and ((Data = DataInternal)
                                or (Data = ImpedeInternal))) then
                        Data <= DataInternal;
                else
                        Data <= ImpedeInternal;
                end if;
        end if;
end process;

end Behavioral;
```

5.5 FAP: Linear Addition

The linear addition logic unit relies on a use of an external unit that calls an internal unit which calls itself recursively, with the internal component being a Ripple Carry Adder (RCA) that breaks into halves of itself, where the callee RCA calls a single RCA on its lower half, and two RCA on its upper half for fixed carry in of both on and off, which then have their outputs case selected by the carry out result from the lower half. Thus we have the RCA circuit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_LINADDRMUX_INTERNALRCA is
        Generic (N : natural := VecLen;
                M : natural := MultLen); --Terminal Length
        Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Cin : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Cout : out  STD_LOGIC);
end GENERIC_FAP_LINADDRMUX_INTERNALRCA;

architecture Behavioral of GENERIC_FAP_LINADDRMUX_INTERNALRCA is
```

```
component GENERIC_FAP_LINADDRMUX_INTERNALRCA
        Generic (N : natural := VecLen;
                M : natural := MultLen); --Terminal Length
        Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Cin : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Cout : out  STD_LOGIC);
end component;

signal CarryOn : STD_LOGIC;
signal CarryOff : STD_LOGIC;
signal CarryInternalTerminal : STD_LOGIC_VECTOR ((M-1) downto 0);
signal CarryInternalSplit : STD_LOGIC;
signal SummationOn : STD_LOGIC_VECTOR ((N-1) downto (N/2));
signal SummationOff : STD_LOGIC_VECTOR ((N-1) downto (N/2));

begin

terminal_structure_FA : if (N = 1) generate
begin
        Cout <= ((A(0) and B(0)) or (A(0) and Cin) or (B(0) and Cin));
        S(0) <= (A(0) xor B(0) xor Cin);
end generate terminal_structure_FA;

terminal_structure : if ((N <= M) and (N > 1)) generate
begin
CarryInternalTerminal(0) <= ((A(0) and B(0)) or (A(0) and Cin) or (B(0) and Cin));
S(0) <= (A(0) xor B(0) xor Cin);
ADDRGen : for K in 1 to (N-2) generate
begin
CarryInternalTerminal(K) <= ((A(K) and B(K)) or (A(K) and
        CarryInternalTerminal(K-1)) or (B(K) and CarryInternalTerminal(K-1)));
S(K) <= (A(K) xor B(K) xor CarryInternalTerminal(K-1));
end generate ADDRGen;
Cout <= ((A(N-1) and B(N-1)) or (A(N-1) and CarryInternalTerminal(N-2))
        or (B(N-1) and CarryInternalTerminal(N-2)));
S(N-1) <= (A(N-1) xor B(N-1) xor CarryInternalTerminal(N-2));
end generate terminal_structure;
```

```
--Recursively break into halves
recursive_structure : if (N > M) generate
begin
        --Make one copy of the first block
        ADDR1 : GENERIC_FAP_LINADDRMUX_INTERNALRCA
                Generic map (N => (N/2))
                Port map ( A => A(((N/2)-1) downto 0),
                          B => B(((N/2)-1) downto 0),
                          Cin => Cin,
                          S => S(((N/2)-1) downto 0),
                          Cout => CarryInternalSplit);
        --Make the "OFF" Carry Block
        ADDR2_0 : GENERIC_FAP_LINADDRMUX_INTERNALRCA
                Generic map (N => (N/2))
                Port map ( A => A(((2*(N/2))-1) downto (N/2)),
                          B => B(((2*(N/2))-1) downto (N/2)),
                          Cin => '0',
                          S => SummationOff(((2*(N/2))-1) downto (N/2)),
                          Cout => CarryOff);
        --Make the "ON" Carry Block
        ADDR2_1 : GENERIC_FAP_LINADDRMUX_INTERNALRCA
                Generic map (N => (N/2))
                Port map ( A => A(((2*(N/2))-1) downto (N/2)),
                          B => B(((2*(N/2))-1) downto (N/2)),
                          Cin => '1',
                          S => SummationOn(((2*(N/2))-1) downto (N/2)),
                          Cout => CarryOn);
        --The carry out of the entire block from the first block's carry out
        Cout <= (CarryInternalSplit and CarryOn) or ((not CarryInternalSplit)
                and CarryOff);
        --The summation of the second half of the block
        SumGen1 : for K in (N/2) to ((2*(N/2))-1) generate
        begin
                S(K) <= (CarryInternalSplit and SummationOn(K)) or
                        ((not CarryInternalSplit) and SummationOff(K));
        end generate SumGen1;

end generate recursive_structure;

end Behavioral;
```

5.5.1 Linear Addition

The outer cell of the linear addition circuit simply changes the interface of the RCA to be one with a constantly off carry in bit, and the carryout forms part of the summation output.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_LINADDRMUX is
        Generic (N : natural := VecLen;
                M : natural := MultLen); --Terminal Length
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end GENERIC_FAP_LINADDRMUX;

architecture Behavioral of GENERIC_FAP_LINADDRMUX is

component GENERIC_FAP_LINADDRMUX_INTERNALRCA
        Generic (N : natural;
                                M : natural); --Terminal Length
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Cin : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Cout : out  STD_LOGIC);
end component;

begin

X : GENERIC_FAP_LINADDRMUX_INTERNALRCA
               Generic map (N => N, M => M)
               Port map (A => A, B => B, Cin => '0',
                       S => S((N-1) downto 0), Cout => S(N));

end Behavioral;
```

5.5.2 2’s Compliment

To support the later functionality of modular subtraction, it is important to implement linear subtraction, or, at least, a unit that is able to output the 2’s Complimented form of an input, such that addition can be done with 2’s Complimented form. This cell simply inverts an input and adds the UnitVector to it.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_2SCOMPLIMENT is
         Generic (N : natural := VecLen);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           C : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end GENERIC_2SCOMPLIMENT;

architecture Behavioral of GENERIC_2SCOMPLIMENT is

component GENERIC_FAP_LINADDRMUX
         Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;

signal CTemp : STD_LOGIC_VECTOR (N downto 0);
signal AINV : STD_LOGIC_VECTOR ((N-1) downto 0);
constant InternalUnitVector : STD_LOGIC_VECTOR ((N-1) downto 0)
                        := (0 => '1', others => '0');

begin

AINV <= (not A);

X : GENERIC_FAP_LINADDRMUX
                Generic map (N => N)
                Port map ( A => AINV,
                  B => InternalUnitVector,
                  S => CTemp);
C <= CTemp((N-1) downto 0);

end Behavioral;
```

5.6 FAP: Linear Multipliers Design Alternatives

The design so far up until now has been mostly combinatorial, but the stage of implementing the design for a linear multiplier is where the necessity for distinguishing between clocked and combinatorial circuits arises, as the size of the combinatorial form is related roughly to the square of the bit length in amounts of Full Adder circuits, using concatenated ADDR cells to form the MULT cell. However the clocked comb form requires a single ADDR, which shifts along the length of the output. We simply present both options.

5.6.1 Combinatorial Multiplication

The combinatorial form is presented here, with significant reductions, presenting the combinatorial form with no optimisation, and then presenting two of the optimisation cases in the code, the Karatsuba style optimisation, and the Toom Cook style optimisation.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODMULT_MULTCOMB is
         Generic (N : Natural := VecLen;
                  M : Natural := MultLen;
                  T : Natural := 1);
                  --TYPE CASTS: 0: Does concatenated addition only,
                  --1 (default) Karatsuba-Ofman reuction
                  --2 Toom-Cook reduction
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Product : out  STD_LOGIC_VECTOR (((2*N)-1) downto 0));
end GENERIC_FAP_MODMULT_MULTCOMB;

architecture Behavioral of GENERIC_FAP_MODMULT_MULTCOMB is

component GENERIC_FAP_MODMULT_MULTCOMB
         Generic (N : Natural := VecLen;
                  M : Natural := MultLen;
                  T : Natural := 1);
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
                MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
                Product : out  STD_LOGIC_VECTOR (((2*N)-1) downto 0));
end component;
```

```
component GENERIC_FAP_LINADDRMUX
         Generic (N : natural;
                                  M : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;
component GENERIC_2SCOMPLIMENT
         Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           C : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;
--Used in Terminal Length Calculations
signal ClipATerminalLength : STD_LOGIC_VECTOR((M-1) downto 0);
signal ClipBTerminalLength : STD_LOGIC_VECTOR((M-1) downto 0);
signal ClipPTerminalLength : STD_LOGIC_VECTOR(((2*M)-1) downto 0);
type AndArray is array ((M-1) downto 0) of STD_LOGIC_VECTOR((M-1) downto 0);
signal AndArr : AndArray;
type AddrArray is array ((M-2) downto 0) of STD_LOGIC_VECTOR(M downto 0);
signal AddrArr : AddrArray;
--Used in splitting length calculations.
signal ProductLALB : STD_LOGIC_VECTOR((2*(N/2)-1) downto 0);
signal ProductMAMB : STD_LOGIC_VECTOR((2*(N-(N/2))-1) downto 0);
signal ProductLAMB : STD_LOGIC_VECTOR((2*(N-(N/2))-1) downto 0);
signal ProductMALB : STD_LOGIC_VECTOR((2*(N-(N/2))-1) downto 0);
signal LowerAPadded : STD_LOGIC_VECTOR(((N-(N/2))-1) downto 0);
signal LowerBPadded : STD_LOGIC_VECTOR(((N-(N/2))-1) downto 0);
begin
TerminalGenNoReduction : if (T = 0) generate
begin
--If T is 0 then put all the inputs onto a port map that maps M to N and
--changes T such that this calls the TerminalGenReduction on the whole length
        TerminalMultNoReductionPortal : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => N,
                        M => N,
                        T => 1)
                Port Map ( MultiplicandA => MultiplicandA,
                         MultiplicandB => MultiplicandB,
                         Product => Product);
end generate TerminalGenNoReduction;
```

```
TerminalGenReduction : if ((N <= M) and ((T = 1) or (T = 2)
                or (T = 3) or (T = 4))) generate
begi
        ClipToLengthUpTo : for K in 0 to (N-1) generate
        begin
                ClipATerminalLength(K) <= MultiplicandA(K);
                ClipBTerminalLength(K) <= MultiplicandB(K);
        end generate ClipToLengthUpTo;
        ClipToLengthUpFrom : for K in N to (M-1) generate
        begin
                ClipATerminalLength(K) <= '0';
                ClipBTerminalLength(K) <= '0';
        end generate ClipToLengthUpFrom;
        AndArrGenK : for K in 0 to (M-1) generate
        begin
                AndArrGenJ : for J in 0 to (M-1) generate
                begin
                  AndArr(K)(J) <= ClipATerminalLength(K) and ClipBTerminalLength(J);
                end generate AndArrGenJ;
        end generate AndArrGenK;
        AddrArr(0) <= ('0' & AndArr(0));
        TerminalAddrGen : for K in 0 to (M-3) generate
        begin
                TerminalAddrX : GENERIC_FAP_LINADDRMUX
                        Generic Map (N => M, M => M)
                        Port Map (A => AddrArr(K)(M downto 1),
                                B => AndArr(K+1),
                                S => AddrArr(K+1));
                ClipPTerminalLength(K) <= AddrArr(K)(0);
        end generate TerminalAddrGen;
        ClipPTerminalLength(M-2) <= AddrArr(M-2)(0);
        TerminalAddrFinal : GENERIC_FAP_LINADDRMUX
                        Generic Map (N => M, M => M)
                        Port Map (A => AddrArr(M-2)(M downto 1),
                                B => AndArr(M-1),
                                S => ClipPTerminalLength(((2*M)-1) downto M-1));
        Product <= ClipPTerminalLength(((2*N)-1) downto 0);
end generate TerminalGenReduction;
end Behavioral;
```

Karatsuba Optimisation

```
--KO
signal AdditionLAMA : STD_LOGIC_VECTOR((N-(N/2)) downto 0);
signal AdditionLBMB : STD_LOGIC_VECTOR((N-(N/2)) downto 0);
signal AdditionLLMM : STD_LOGIC_VECTOR((2*(N-(N/2))) downto 0);
signal AdditionLLMMInAPort : STD_LOGIC_VECTOR(((2*(N-(N/2)))-1) downto 0);
signal ProductLALBconcatMAMB : STD_LOGIC_VECTOR(((2*N)-1) downto 0);
signal ProductADDRLAMAandLBMB : STD_LOGIC_VECTOR(((2*(N-(N/2)+1))-1) downto 0);
signal ProductMixedDifferenceSubtrahend : STD_LOGIC_VECTOR((2*N-1) downto 0);
signal ProductMixedDifferenceSubtrahendInv : STD_LOGIC_VECTOR((2*N-1) downto 0);
signal ProductMixedDifference : STD_LOGIC_VECTOR(((2*(N-(N/2)+1))-1) downto 0);
signal ProductMixedDifferencePadded : STD_LOGIC_VECTOR(((2*N)-1) downto 0);
constant Padding : STD_LOGIC_VECTOR(((2*N)-1) downto 0) := (others => '0');
signal ProductLLMMLMLM : STD_LOGIC_VECTOR((2*N) downto 0);
signal ProductInternal : STD_LOGIC_VECTOR((2*N) downto 0);
constant CappedModulus : STD_LOGIC_VECTOR((2*N-1) downto 0) := (others => '1');
SplittingGenKaratsubaOfman : if ((N > M) and (T = 1)) generate
begin
        --Multiply the lower half of A with the lower half of B
        SplittingKOMultLALB : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N/2), M => M, T => T)
                Port Map ( MultiplicandA => MultiplicandA(((N/2)-1) downto 0),
                        MultiplicandB => MultiplicandB(((N/2)-1) downto 0),
                        Product => ProductLALB);
        --Multiply the upper half of A with the upper half of B
        SplittingKOMultMAMB : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N-(N/2)), M => M, T => T)
                Port Map ( MultiplicandA => MultiplicandA((N-1) downto (N/2)),
                        MultiplicandB => MultiplicandB((N-1) downto (N/2)),
                        Product => ProductMAMB);
--Concatenate the result of multiplying the two
--top halves and the two bottom halves.
        ProductLALBconcatMAMB <= ProductMAMB & ProductLALB;
--Establish inputs to the Karatsuba Adders:
--When N is even, copy the natural lengthed lower halves
--which is the same as the natural lengthed upper halves.
        SplittingKOIfNEven : if ((N mod 2) = 0) generate
        begin
                LowerAPadded <= MultiplicandA(((N/2)-1) downto 0);
                LowerBPadded <= MultiplicandB(((N/2)-1) downto 0);
                AdditionLLMMInAPort <= ProductLALB;
        end generate SplittingKOIfNEven;
```

```
--Establish inputs to the Karatsuba Adders:
--When N is odd, copy the natural lengthed lower
--halves, padded to the natural lengthed upper halves.
        SplittingKOIfNOdd : if ((N mod 2) = 1) generate
        begin
                LowerAPadded <= ('0' & MultiplicandA(((N/2)-1) downto 0));
                LowerBPadded <= ('0' & MultiplicandB(((N/2)-1) downto 0));
                AdditionLLMMInAPort <= ("00" & ProductLALB);
        end generate SplittingKOIfNOdd;
--Add the upper half and lower half of A
        SplittingKOAddrAHALVES : GENERIC_FAP_LINADDRMUX
                        Generic Map (N => (N-(N/2)), M => M)
                        Port Map (A => MultiplicandA((N-1) downto (N/2)),
                                B => LowerAPadded,
                                S => AdditionLAMA);
--Add the upper half and lower half of B
        SplittingKOAddrBHALVES : GENERIC_FAP_LINADDRMUX
                        Generic Map (N => (N-(N/2)), M => M)
                        Port Map (A => MultiplicandB((N-1) downto (N/2)),
                                B => LowerBPadded,
                                S => AdditionLBMB);
--Multiply these additions of the two halves each of A and B.
        SplittingKOMultADDRHALVES : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N-(N/2)+1),
                                M => M,
                                T => T)
                Port Map ( MultiplicandA => AdditionLAMA,
                                     MultiplicandB => AdditionLBMB,
                                     Product => ProductADDRLAMAandLBMB);
--Add together the result of multiplying the lower
--halves and the upper halves each by themselves.
        SplittingAddrLLandMM : GENERIC_FAP_LINADDRMUX
                        Generic Map (N => (2*(N-(N/2))), M => M)
                        Port Map (A => AdditionLLMMInAPort,
                                B => ProductMAMB,
                                S => AdditionLLMM);
--Pad the length of the results of the previous addition
--and multiplication to be at their appropriate bit locations.
        ProductMixedDifferencePadded <= Padding((2*N-1) downto
                ((N/2)+(2*(N-(N/2)+1)))) & ProductADDRLAMAandLBMB &
                Padding(((N/2)-1) downto 0);
        ProductMixedDifferenceSubtrahend <= (Padding((2*N-1) downto
                ((N/2)+(2*(N-(N/2))+1))) & AdditionLLMM &
                Padding(((N/2)-1) downto 0));
--Add the result of the concatenated upper and lower
--multiples with the result of multiplying the upper
--and lower halves added to each other.
        SplittingKOAddrLALBMAMBandLAMALBMB : GENERIC_FAP_LINADDRMUX
                        Generic Map (N => (2*N), M => M)
                        Port Map (A => ProductLALBconcatMAMB,
                                B => ProductMixedDifferencePadded,
                                S => ProductLLMMLMLM);
--Take the additive inverse of the sum of the lower and upper multiples
        InverterKO : GENERIC_2SCOMPLIMENT
                        Generic Map (N => (2*N))
                        Port Map (A => ProductMixedDifferenceSubtrahend,
                                C => ProductMixedDifferenceSubtrahendInv);
--Add (Subtract) this inverse of the sum of the lower
--and upper multiples from the result
--of the concatenation added with the multiple
--of the upper and lower halves multiplying each other.
        SplittingKOSubtADDRHALVESbyLLandMM : GENERIC_FAP_LINADDRMUX
                Generic Map (N => (2*N), M => M)
                Port Map (A => ProductLLMMLMLM(((2*N)-1) downto 0),
                        B => ProductMixedDifferenceSubtrahendInv,
                        S => ProductInternal);
--Put the lengthed Product on the output.
        Product <= ProductInternal((2*N-1) downto 0);
end generate SplittingGenKaratsubaOfman;
```

Toom Cook Optimisation

```
--TC
signal AddrLMandML : STD_LOGIC_VECTOR((2*(N-(N/2))) downto 0);
signal AddrLMandMLPadded : STD_LOGIC_VECTOR(((2*N)-1) downto 0);

SplittingGenToomCook : if ((N > M) and (T = 2)) generate
begin
--Multiply the lower half of A with the lower half of B
        SplittingTCMultLALB : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N/2),
                                        M => M,
                                        T => T)
                Port Map ( MultiplicandA => MultiplicandA(((N/2)-1) downto 0),
                        MultiplicandB => MultiplicandB(((N/2)-1) downto 0),
                        Product => ProductLALB);
```

```
--Multiply the upper half of A with the upper half of B
        SplittingTCMultMAMB : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N-(N/2)),
                                                 M => M,
                                                 T => T)
                Port Map ( MultiplicandA => MultiplicandA((N-1) downto (N/2)),
                        MultiplicandB => MultiplicandB((N-1) downto (N/2)),
                        Product => ProductMAMB); --(2*(N-(N/2)) in size
--Concatenate the result of multiplying the
--two top halves and the two bottom halves.
        ProductLALBconcatMAMB <= ProductMAMB & ProductLALB;
--Establish inputs to the Toom-Cook Mults:
--When N is even, copy the natural lengthed lower halves
--which is the same as the natural lengthed upper halves.
        SplittingTCIfNEven : if ((N mod 2) = 0) generate
        begin
                LowerAPadded <= MultiplicandA(((N/2)-1) downto 0);
                LowerBPadded <= MultiplicandB(((N/2)-1) downto 0);
        end generate SplittingTCIfNEven;
--Establish inputs to the Toom-Cook Mults:
--When N is odd, copy the natural lengthed lower
--halves, padded to the natural lengthed upper halves.
        SplittingTCIfNOdd : if ((N mod 2) = 1) generate
        begin
                LowerAPadded <= ('0' & MultiplicandA(((N/2)-1) downto 0));
                LowerBPadded <= ('0' & MultiplicandB(((N/2)-1) downto 0));
        end generate SplittingTCIfNOdd;
--Multiply the lower half of A with the upper half of B
        SplittingTCMultLAMB : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N-(N/2)),
                                M => M,
                                T => T)
                Port Map ( MultiplicandA => LowerAPadded,
                                MultiplicandB => MultiplicandB((N-1) downto (N/2)),
                                Product => ProductLAMB); --(2*(N-(N/2)) in size
--Multiply the upper half of A with the lower half of B
        SplittingTCMultMALB : GENERIC_FAP_MODMULT_MULTCOMB
                Generic Map (N => (N-(N/2)),
                                M => M,
                                T => T)
                Port Map ( MultiplicandA => MultiplicandA((N-1) downto (N/2)),
                                MultiplicandB => LowerBPadded,
                                Product => ProductMALB); --(2*(N-(N/2)) in size
```

```
--Do the additions on each of the padded halves piecewise
        SplittingTCAddrMMandML : GENERIC_FAP_LINADDRMUX
                Generic Map (N => (2*(N-(N/2))), M => M)
                Port Map (A => ProductMALB,
                                B => ProductLAMB,
                                S => AddrLMandML);
        AddrLMandMLPadded <= Padding(((2*N)-1) downto
                ((N/2)+(2*(N-(N/2)))+1)) & AddrLMandML &
                Padding(((N/2)-1) downto 0);
--Do the total addition overall for the whole thing.
        SplittingTCAddrLLandLMandMLandMM : GENERIC_FAP_LINADDRMUX
                Generic Map (N => 2*N, M => M)
                Port Map (A => ProductLALBconcatMAMB,
                                B => AddrLMandMLPadded,
                                S => ProductInternal);
--Put the lengthed Product on the output.
        Product <= ProductInternal((2*N-1) downto 0);
end generate SplittingGenToomCook;
```

5.6.2 Clocked Comb Multiplication

Here we present the "clocked comb" version of the multiplier, and the unit chosen in the final design implementation.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODMULT_MULTCOMB_CLOCKED is
         Generic (N : Natural := VecLen;
                M : Natural := MultLen;
                AddrDelay : Time := 30 ns);
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Product : out  STD_LOGIC_VECTOR (((2*N)-1) downto 0);
           CLK : in STD_LOGIC;
           StableOutput : out STD_LOGIC);
end GENERIC_FAP_MODMULT_MULTCOMB_CLOCKED;

architecture Behavioral of GENERIC_FAP_MODMULT_MULTCOMB_CLOCKED is
```

```
component GENERIC_FAP_LINADDRMUX
         Generic (N : natural;
                M : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;
component GENERIC_FAP_RELATIONAL
         Generic (N : Natural;
                VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;
signal AndRegister : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal LatchedMultA : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal LatchedMultB : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal LatchedAddrA : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal LatchedAddrB : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal LatchedAddrS : STD_LOGIC_VECTOR (N downto 0) := (others => '0');
signal LatchedProduct : STD_LOGIC_VECTOR ((2*N) downto 0) := (others => '0');
signal StableAdder : STD_LOGIC := '0';
signal StableOutputInner : STD_LOGIC := '0';
signal BitPositionVector : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
Constant BitPositionVectorIsZero : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (others => '0');
Constant BitPositionVectorIsUnit : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (1 => '1', others => '0');
signal InternalClock : STD_LOGIC := '0';
signal BitPositionBit : STD_LOGIC := '0' ;
signal BitPositionBitVector : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (others => '0');
signal Equalities : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
begin
StableOutput <= StableOutputInner;
LatchedAddrA <= AndRegister;
ProdGen : for K in 0 to ((2*N)-1) generate
begin Product(K) <= (LatchedProduct(K+1) and StableOutputInner);
end generate ProdGen;
AddrGen : for K in 0 to (N-1) generate
begin AndRegister(K) <= (MultiplicandB(K) and BitPositionBit);
end generate AddrGen;
```

```
BitPosGen : for K in 0 to (N-1) generate
begin BitPositionBitVector(K) <= MultiplicandA(K) and BitPositionVector(K);
end generate BitPosGen;
InternalClock <= transport (not InternalClock) after AddrDelay;
process(CLK)
begin
 if (rising_edge(CLK)) then
  if ((Equalities(1) and Equalities(2)) = '1') then --comparator --comparator
   if (Equalities(3) = '0') then
    if (StableAdder = '0') then
     if (Equalities(0) = '1') then
      BitPositionBit <= '0';
     else
      BitPositionBit <= '1';
     end if;
     LatchedAddrB <= LatchedProduct((2*N) downto (N+1));
     LatchedProduct(((2*N)-1) downto 0) <= '0' &
             LatchedProduct(((2*N)-1) downto 1);
     StableAdder <= '1';
    else
     LatchedProduct((2*N) downto N) <= LatchedAddrS;
     BitPositionVector <= BitPositionVector((N-2) downto 0) & '0';
     StableAdder <= '0';
    end if;
   else
    StableOutputInner <= '1';
   end if;
  else
   LatchedMultA <= MultiplicandA;
   LatchedMultB <= MultiplicandB;
   StableOutputInner <= '0';
   StableAdder <= '0';
   BitPositionVector <= BitPositionVectorIsUnit;
   if (MultiplicandA(0) = '1') then
    LatchedProduct(2*N) <= '0';
    LatchedProduct(((2*N)-1) downto N) <= MultiplicandB;
    LatchedProduct((N-1) downto 0) <= (others => '0');
   else
    LatchedProduct <= (others => '0');
   end if;
end if; end if; end process;
```

```
ADDR : GENERIC_FAP_LINADDRMUX
        Generic Map (N => N, M => M)
        Port Map (A => LatchedAddrA, B => LatchedAddrB, S => LatchedAddrS);

EG0 : GENERIC_FAP_RELATIONAL
        Generic Map (N => N,
                VType : Natural);
    Port Map ( A => BitPositionBitVector,
            B => BitPositionVectorIsZero,
            E => Equalities(0),
            G => open);

EG1 : GENERIC_FAP_RELATIONAL
        Generic Map (N => N,
                VType : Natural);
    Port Map ( A => MultiplicandA,
            B => LatchedMultA,
            E => Equalities(1),
            G => open);

EG2 : GENERIC_FAP_RELATIONAL
        Generic Map (N => N,
                VType : Natural);
    Port Map ( A => MultiplicandB,
            B => LatchedMultB,
            E => Equalities(2),
            G => open);

EG3 : GENERIC_FAP_RELATIONAL
        Generic Map (N => N,
                VType : Natural);
    Port Map ( A => BitPositionVector,
            B => BitPositionVectorIsZero,
            E => Equalities(3),
            G => open);

end Behavioral;
```

5.7 FAP: Linear Division Design Alternatives

We now present the code used to design the linear division logic, beginning with the description of the Piecwise comparator that acts either concatenated with itself to form combinatorial division, or can be used as a "clocked comb."

5.7.1 Piece-wise Comparator

We now describe the entirely combinatorial logic of the Piecewise comparator, a logic block that takes and evaluates some $(N + 1)$ bits against an N bit prime.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE is
         Generic (N : Natural := VecLen;
                  M : Natural := MultLen);
    Port ( ValueIn : in  STD_LOGIC_VECTOR (N downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           ValueModded : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           SubbedByM : out  STD_LOGIC;
           SubbedBy2M : out  STD_LOGIC;
           SubbedBy3M : out  STD_LOGIC);
end GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE;

architecture Behavioral of GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE is

component GENERIC_FAP_LINADDRMUX
         Generic (N : natural;
                  M : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;

component GENERIC_FAP_RELATIONAL
         Generic (N : Natural;
                  VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;
```

```
component GENERIC_2SCOMPLIMENT
         Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           C : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;

signal ModulusOnce : STD_LOGIC_VECTOR (N downto 0);
signal ModulusTwice : STD_LOGIC_VECTOR (N downto 0);
--If ModulusThrice(N+1) is 1 then this exceeds the range
--of the value in and SubbedBy3M is impossible.
signal ModulusThrice : STD_LOGIC_VECTOR ((N+1) downto 0);
signal ModulusOnceInv : STD_LOGIC_VECTOR (N downto 0);
signal ModulusTwiceInv : STD_LOGIC_VECTOR (N downto 0);
signal ModulusThriceInv : STD_LOGIC_VECTOR (N downto 0);
signal WasSubbedByM : STD_LOGIC;
signal WasSubbedBy2M : STD_LOGIC;
signal WasSubbedBy3M : STD_LOGIC;
signal Equia : STD_LOGIC_VECTOR (3 downto 0);
signal Great : STD_LOGIC_VECTOR (3 downto 0);
signal MODIN : STD_LOGIC_VECTOR (N downto 0);
signal MODOUT : STD_LOGIC_VECTOR ((N+1) downto 0);

begin

--Construct the three multiples of the Modulus to be compared to
ModulusOnce <= '0' & Modulus;
ModulusTwice <= Modulus & '0';
TripleModulus : GENERIC_FAP_LINADDRMUX
        Generic Map (N => (N+1),
                     M => M)
   Port Map ( A => ModulusOnce,
              B => ModulusTwice,
              S => ModulusThrice);
--Perform the comparisons on the three Modular multiples.
CompareOnce : GENERIC_FAP_RELATIONAL
        Generic Map (N => (N+1),
                     VType => 1)
   Port Map ( A => ValueIn,
              B => ModulusOnce,
              E => Equia(0),
              G => Great(0));
```

```
CompareTwice : GENERIC_FAP_RELATIONAL
        Generic Map (N => (N+1),
                VType => 1)
   Port Map ( A => ValueIn,
                B => ModulusTwice,
                E => Equia(1),
                G => Great(1));
CompareThrice : GENERIC_FAP_RELATIONAL
   Generic Map (N => (N+1),
                VType => 1)
   Port Map ( A => ValueIn,
                B => ModulusThrice(N downto 0),
                E => Equia(3),
                G => Great(3));
--Buffer the actual Equia and Great for the 3*M
--check with whether the top bit is on or off;
Equia(2) <= (Equia(3) and (not ModulusThrice(N+1)));
Great(2) <= (Great(3) and (not ModulusThrice(N+1)));
--Universally compliment all three modular multiples
ComplimentOnce : GENERIC_2SCOMPLIMENT
    Generic Map (N => (N+1))
    Port Map ( A => ModulusOnce,
               C => ModulusOnceInv);
ComplimentTwice : GENERIC_2SCOMPLIMENT
    Generic Map (N => (N+1))
    Port Map ( A => ModulusTwice,
               C => ModulusTwiceInv);
ComplimentThrice : GENERIC_2SCOMPLIMENT
    Generic Map (N => (N+1))
    Port Map ( A => ModulusThrice(N downto 0),
               C => ModulusThriceInv);
--Determine the 'Was Subbed By' outputs
WasSubbedByM <= ((Equia(0) or Great(0)) and (not WasSubbedBy2M)
        and (not WasSubbedBy3M));
WasSubbedBy2M <= ((Equia(1) or Great(1)) and (not WasSubbedBy3M));
WasSubbedBy3M <= (Equia(2) or Great(2));
--Determine the modular multiple from knowing the E and G relations.
MODINGEN : for K in 0 to N generate
begin
        MODIN(K) <= ((ModulusOnceInv(K) and WasSubbedByM) or
                (ModulusTwiceInv(K) and WasSubbedBy2M) or
                (ModulusThriceInv(K) and WasSubbedBy3M));
end generate MODINGEN;
```

```
--Add the modular inverse for the apprpriate modular multiple
ModAddr : GENERIC_FAP_LINADDRMUX
         Generic Map (N => (N+1),
                M => M)
    Port Map ( A => MODIN,
                B => ValueIn,
                S => MODOUT);
--Put on the output
ValueModded <= MODOUT((N-1) downto 0);
SubbedByM <= WasSubbedByM;
SubbedBy2M <= WasSubbedBy2M;
SubbedBy3M <= WasSubbedBy3M;

end Behavioral;
```

5.7.2 Combinatorial Division

We can now use the Piecewise comparator to present combinatorial division;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODMULT_DIVDCOMB is
         Generic (N : Natural := VecLen;
                M : Natural := MultLen);
    Port ( Dividend : in  STD_LOGIC_VECTOR (((2*N)-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Remainder : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Quotient : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end GENERIC_FAP_MODMULT_DIVDCOMB;
architecture Behavioral of GENERIC_FAP_MODMULT_DIVDCOMB is
component GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE
         Generic (N : Natural;
                M : Natural);
    Port ( ValueIn : in  STD_LOGIC_VECTOR (N downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           ValueModded : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           SubbedByM : out  STD_LOGIC;
           SubbedBy2M : out  STD_LOGIC;
           SubbedBy3M : out  STD_LOGIC);
end component;
type PiecewiseData is array ((N-1) downto 0) of STD_LOGIC_VECTOR((N-1) downto 0);
```

```
signal SubbedByM : STD_LOGIC_VECTOR ((N-1) downto 0);
signal SubbedBy2M : STD_LOGIC_VECTOR ((N-1) downto 0);
signal SubbedBy3M : STD_LOGIC_VECTOR ((N-1) downto 0);
signal PieceWiseDataBus : PieceWiseData;
begin
--Instantiate the first comparator with the
--highest (N+1) bits of the Dividend attached
PieceWiseCompareN : GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE
        Generic Map (N => N, M => M)
        Port Map (ValueIn => Dividend(((2*N)-1) downto (N-1)),
           Modulus => Modulus,
           ValueModded => PieceWiseDataBus(N-1),
           SubbedByM => SubbedByM(N-1),
           SubbedBy2M => SubbedBy2M(N-1),
           SubbedBy3M => SubbedBy3M(N-1));
--Generate the cascade of Piecewise Comparators
--Generate the remaining 255 comparators, establishing all interconnections
PieceWiseCompareXGen : for k in (N-2) downto 0 generate
begin
        PieceWiseCompareX : GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE
        Generic Map (N => N, M => M)
        Port Map (ValueIn(N downto 1) => PieceWiseDataBus(k+1),
           ValueIn(0) => Dividend(k),
           Modulus => Modulus,
           ValueModded => PieceWiseDataBus(k),
           SubbedByM => SubbedByM(k),
           SubbedBy2M => SubbedBy2M(k),
           SubbedBy3M => SubbedBy3M(k));
end generate PieceWiseCompareXGen;
--The remainder after integer division is the
--contents of the final modulo comparator.
Remainder <= PieceWiseDataBus(0);
--Now deal with the quotient from the SubbedBy signals. The first bit
--only looks at the SubbedByP of the last
--comparator, the rest use a generate.
Quotient(0) <= (SubbedByM(0) or SubbedBy3M(0));
QuotientGenerate : for k in 1 to (N-1) generate
begin
        Quotient(k) <= (SubbedByM(k) or SubbedBy2M(k-1) or
                (SubbedBy3M(k-1) or SubbedBy3M(k)));
end generate QuotientGenerate;
end Behavioral;
```

5.7.3 Clocked Comb Division

We are now in a position to present the Clocked Comb version of this above description, which acts over an input of double the length of the VecLen variable, and reduces it to an output of the length of VecLen, by using the Piecewise comparator in a Clocked Comb.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODMULT_DIVDCOMB_CLOCKED is
    Generic (N : Natural := VecLen;
                M : Natural := MultLen;
                CompDelay : Time := 30 ns);
    Port ( Dividend : in  STD_LOGIC_VECTOR (((2*N)-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC;
           Remainder : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Quotient : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           StableOutput : out STD_LOGIC);
end GENERIC_FAP_MODMULT_DIVDCOMB_CLOCKED;

architecture Behavioral of GENERIC_FAP_MODMULT_DIVDCOMB_CLOCKED is

component GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE
         Generic (N : Natural;
                M : Natural);
    Port ( ValueIn : in  STD_LOGIC_VECTOR (N downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           ValueModded : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           SubbedByM : out  STD_LOGIC;
           SubbedBy2M : out  STD_LOGIC;
           SubbedBy3M : out  STD_LOGIC);
end component;

component GENERIC_FAP_RELATIONAL
         Generic (N : Natural;
                VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;
```

```
signal SubbedByM : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal SubbedBy2M : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal SubbedBy3M : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal LatchedProductSpace : STD_LOGIC_VECTOR (((2*N)-1) downto 0)
        := (others => '0');
signal LatchedDividend : STD_LOGIC_VECTOR (((2*N)-1) downto 0)
        := (others => '0');
signal LatchedModulus : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (others => '0');
signal StableCompare : STD_LOGIC := '0';
signal StableOutputInner : STD_LOGIC := '0';
signal LatchedValueIn : STD_LOGIC_VECTOR (N downto 0) := (others => '0');
signal LatchedValueModded : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (others => '0');
signal LatchedSubbedByM : STD_LOGIC := '0';
signal LatchedSubbedBy2M : STD_LOGIC := '0';
signal LatchedSubbedBy3M : STD_LOGIC := '0';
signal BitPositionVector : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (0 => '1', others => '0');
Constant BitPositionVectorIsZero : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (others => '0');
Constant BitPositionVectorIsUnit : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (0 => '1', others => '0');
signal InternalClock : STD_LOGIC := '0';
signal Equalities : STD_LOGIC_VECTOR(2 downto 0) := (others => '0');

begin

StableOutput <= StableOutputInner;

RemGen : for K in 0 to (N-1) generate
begin
        Remainder(K) <= (LatchedProductSpace(K+(((2*N)-1)-N+1))
                and StableOutputInner);
end generate RemGen;

InternalClock <= transport (not InternalClock) after CompDelay;
```

```
process(CLK)
begin
 if         (rising_edge(CLK)) then
  if ((Equalities(1) and Equalities(2)) = '1') then --comparator --comparator
  --Carry on the action with the next stage
   if(Equalities(0) = '0') then
    if (StableCompare = '0') then
     LatchedValueIn <= LatchedProductSpace(((2*N)-1) downto (((2*N)-1)-N));
     LatchedProductSpace <= LatchedProductSpace(((2*N)-2) downto 0) & '0';
     StableCompare <= '1';
    else
     LatchedProductSpace(((2*N)-2+1) downto (((2*N)-1)-N+1)) <= LatchedValueModded;
     BitPositionVector <= BitPositionVector((N-2) downto 0) & '0';
     StableCompare <= '0';
    end if;
   else
    StableOutputInner <= '1';
   end if;
  else
   --Reset to the new input and refresh the process
   LatchedDividend <= Dividend;
   LatchedProductSpace <= Dividend;
   LatchedModulus <= Modulus;
   StableOutputInner <= '0';
   StableCompare <= '0';
   BitPositionVector <= BitPositionVectorIsUnit;
  end if;
 end if;
end process;



COMP : GENERIC_FAP_MODMULT_DIVDCOMB_PIECEWISE
        Generic Map (N => N, M => M)
        Port Map (ValueIn => LatchedValueIn,
           Modulus => Modulus,
           ValueModded => LatchedValueModded,
           SubbedByM => open,
           SubbedBy2M => open,
           SubbedBy3M => open);
```

```
EG0 : GENERIC_FAP_RELATIONAL
        Generic Map (N => N,
                VType => 0)
        Port Map ( A => BitPositionVector,
           B => BitPositionVectorIsZero,
           E => Equalities(0),
           G => open);

EG1 : GENERIC_FAP_RELATIONAL
        Generic Map (N => (2*N),
                VType => 0)
        Port Map ( A => Dividend,
           B => LatchedDividend,
           E => Equalities(1),
           G => open);

EG2 : GENERIC_FAP_RELATIONAL
        Generic Map (N => N,
                VType => 0)
        Port Map ( A => Modulus,
           B => LatchedModulus,
           E => Equalities(2),
           G => open);

end Behavioral;
```

5.8 FAP: Modular Operations

Now that we have described the units for linear addition, 2's complimenting, and multiplication and division combing, we are in a position to produce unit descriptions for the modular operations that comprise the Field Arithmetic Processor (FAP). We begin with Modular Addition and Subtraction, and then construct Modular Multiplication as the stabilised connection between the Multiplier Comb and the Division Comb.

5.8.1 Modular Addition

Modular Addition utilises several linear adders and relational logic to concurrently perform linear addition on different combinations of the input and the additive inverse of the modulus, with this value calculated by connecting the Modulus up to the 2's Compliment logic.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODADDR is
         Generic (N : natural := VecLen;
                M : natural := MultLen); --Terminal Length
    Port ( SummandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           SummandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0); --Modulo.
           Summation : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end GENERIC_FAP_MODADDR;

architecture Behavioral of GENERIC_FAP_MODADDR is

component GENERIC_FAP_LINADDRMUX
         Generic (N : natural;
                M : natural := MultLen); --Terminal Length
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;

component GENERIC_FAP_RELATIONAL
         Generic (N : Natural;
                VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;

component GENERIC_2SCOMPLIMENT
         Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           C : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;

signal MInv : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MRel : STD_LOGIC_VECTOR (N downto 0);
signal AandB : STD_LOGIC_VECTOR (N downto 0);
signal AandBClipped : STD_LOGIC_VECTOR ((N-1) downto 0);
signal FinalIn : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Equia : STD_LOGIC;
signal Great : STD_LOGIC;
signal SInternal : STD_LOGIC_VECTOR (N downto 0);

begin

InvOfModulus : GENERIC_2SCOMPLIMENT
    Generic map (N => N)
    Port map ( A => Modulus,
               C => MInv);

AandBMap : GENERIC_FAP_LINADDRMUX
    Generic map (N => N, M => M)
    Port map ( A => SummandA,
               B => SummandB,
               S => AandB);

MRel <= '0' & Modulus;

Relator : GENERIC_FAP_RELATIONAL
    Generic map (N => (N+1),
               VType => 1)
    Port map ( A => AandB,
               B => MRel,
               E => Equia,
               G => Great);

FinalsGen : for K in 0 to (N-1) generate
begin
        FinalIn(K) <= ((Equia or Great) and MInv(K));
end generate FinalsGen;

AandBClipped <= AandB(N-1 downto 0);

ModAddr : GENERIC_FAP_LINADDRMUX
    Generic map (N => N, M => M)
    Port map ( A => AandBClipped,
               B => FinalIn,
               S => SInternal);

Summation <= SInternal((N-1) downto 0);

end Behavioral;
```

5.8.2 Modular Subtraction

Modular Subtraction is similarly accomplished with linear additions and relation checking logic;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODSUBT is
    Generic (N : natural := VecLen;
                M : natural := MultLen);
    Port ( Minuend : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Subtrahend : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0); --Modulo.
           Difference : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end GENERIC_FAP_MODSUBT;

architecture Behavioral of GENERIC_FAP_MODSUBT is

component GENERIC_FAP_LINADDRMUX
    Generic (N : natural;
                M : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;

component GENERIC_FAP_RELATIONAL
    Generic (N : Natural;
                VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;

component GENERIC_2SCOMPLIMENT
        Generic (N : natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           C : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;

signal SubtrahendInv : STD_LOGIC_VECTOR ((N-1) downto 0);
signal FinalIn : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Equia : STD_LOGIC;
signal Great : STD_LOGIC;
signal MinuendAndModulus : STD_LOGIC_VECTOR (N downto 0);
signal DInternal : STD_LOGIC_VECTOR (N downto 0);

begin

InvOfSubtrahend : GENERIC_2SCOMPLIMENT
    Generic map (N => N)
    Port map ( A => Subtrahend,
               C => SubtrahendInv);

Relator : GENERIC_FAP_RELATIONAL
    Generic map (N => N,
               VType => 1)
    Port map ( A => Minuend,
               B => Subtrahend,
               E => Equia,
               G => Great);

MinandSubtrMap : GENERIC_FAP_LINADDRMUX
    Generic map (N => N, M => M)
    Port map ( A => Minuend,
               B => Modulus,
               S => MinuendAndModulus);

FinalsGen : for K in 0 to (N-1) generate
begin
        FinalIn(K) <= (Equia and Subtrahend(K)) or (Great and Minuend(K))
               or ((not Equia) and (not Great) and MinuendAndModulus(K));
end generate FinalsGen;

MinsubSubtrMap : GENERIC_FAP_LINADDRMUX
    Generic map (N => N, M => M)
    Port map ( A => FinalIn,
               B => SubtrahendInv,
               S => DInternal);

Difference <= DInternal((N-1) downto 0);

end Behavioral;
```

5.8.3 Modular Multiplication

Now we can state modular multiplication as the port mapping between the multiplier comb and the division comb. A reference to a cell not used in the final design has been omitted in the code provided here, however this is where the generic port mapping of the "Toomed" variable originated from, in reference to an "if ... generate" clause that decided whether to produce a non optimised or a Toom Cook optimised Clocked Comb cell, however the Toom Cook optimised cell was not used in the final design so it has been omitted here. The code provided here is the result of setting the Toomed variable to false.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK is
         Generic (N : Natural := VecLen;
                 M : Natural := MultLen;
                 AddrDelay : Time := 30 ns;
                 CompDelay : Time := 30 ns;
                 Toomed : Boolean := false);
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in STD_LOGIC_VECTOR ((N-1) downto 0);
           Product : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC;
           StableOutput : out STD_LOGIC);
end GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK;

architecture Behavioral of GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK is

component GENERIC_FAP_MODMULT_MULTCOMB_CLOCKED
    Generic (N : Natural;
                 M : Natural;
                 AddrDelay : Time);
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Product : out  STD_LOGIC_VECTOR (((2*N)-1) downto 0);
           CLK : in STD_LOGIC;
           StableOutput : out STD_LOGIC);
end component;

component GENERIC_FAP_MODMULT_DIVCOMB_CLOCKED
    Generic (N : Natural;
                 M : Natural;
```

```
                CompDelay : Time);
    Port ( Dividend : in  STD_LOGIC_VECTOR (((2*N)-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC;
           Remainder : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Quotient : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           StableOutput : out STD_LOGIC);
end component;

signal ProductofMultComb : STD_LOGIC_VECTOR (((2*N)-1) downto 0);
signal StableMultComb : STD_LOGIC;
signal StableDivdComb : STD_LOGIC;
signal StableTradeOver : STD_LOGIC;

begin

MULTCELL : GENERIC_FAP_MODMULT_MULTCOMB_CLOCKED
                Generic Map (N => N,
                        M => M,
                        AddrDelay => AddrDelay)
                Port Map ( MultiplicandA => MultiplicandA,
                        MultiplicandB => MultiplicandB,
                        Product => ProductofMultComb,
                        CLK => CLK,
                        StableOutput => StableMultComb);


DIVDCELL : GENERIC_FAP_MODMULT_DIVDCOMB_CLOCKED
        Generic Map (N => N,
                        M => M,
                        CompDelay => CompDelay)
        Port Map ( Dividend => ProductofMultComb,
                        Modulus => Modulus,
                        CLK => CLK,
                        Remainder => Product,
                        Quotient => open,
                        StableOutput => StableDivdComb);
```

```
process(CLK)
begin
        if(rising_edge(CLK)) then
                if ((StableTradeOver and StableDivdComb and StableMultComb) = '1') then
                        StableOutput <= '1';
                else
                        StableTradeOver <= (StableDivdComb and StableMultComb);
                        StableOutput <= '0';
                end if;
        end if;
end process;

end Behavioral;
```

5.9 FAP: Modular Inversion

Now that we have provided all the modular arithmetic operations it is time to provide the code for the Modular Inversion logic. First, we present the Outer Cell, which continuously latches the output and puts it on the input, of the Inner cell.

5.9.1 Outer Cell: Binary Extended Euclidean Algorithm

The Outer cell of the Binary Extended Euclidean Algorithm loop unwrapped.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.VECTOR_STANDARD.ALL;
use work.ECC_STANDARD.ALL;

entity GENERIC_FAP_MODINVR_CLOCKED is
         Generic (N : natural := VecLen;
                 M : natural := MultLen;
                 AddrDelay : Time := 30 ns);
    Port ( Element : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Inverse : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC);
end GENERIC_FAP_MODINVR_CLOCKED;

architecture Behavioral of GENERIC_FAP_MODINVR_CLOCKED is
```

```
--------------------------------
-----COMPONENT DECLARATIONS-----
--------------------------------

component GENERIC_FAP_MODINVR_INTERNAL_CLOCKED
    Generic (N : natural;
          M : natural;
          AddrDelay : Time);
    Port ( U : in STD_LOGIC_VECTOR((N-1) downto 0);
           V : in STD_LOGIC_VECTOR((N-1) downto 0);
           X : in STD_LOGIC_VECTOR((N-1) downto 0);
           Y : in STD_LOGIC_VECTOR((N-1) downto 0);
           Uout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Vout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Xout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Yout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Modulus : in STD_LOGIC_VECTOR((N-1) downto 0);
           CLK : in STD_LOGIC);
end component;

component GENERIC_FAP_RELATIONAL
         Generic (N : Natural;
                VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;

component GENERIC_FAP_MODADDR
    Generic (N : natural;
                M : natural); --Terminal Length
    Port ( SummandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           SummandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0); --Modulo.
           Summation : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;
```

```
------------------------------
-----SIGNAL DECLARATIONS-----
------------------------------

signal U : STD_LOGIC_VECTOR ((N-1) downto 0);
signal V : STD_LOGIC_VECTOR ((N-1) downto 0);
signal X : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Y : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Uout : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Vout : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Xout : STD_LOGIC_VECTOR ((N-1) downto 0);
signal Yout : STD_LOGIC_VECTOR ((N-1) downto 0);

signal StableInverse : STD_LOGIC;
signal InternalInverse : STD_LOGIC_VECTOR((N-1) downto 0);
signal PreviousElement : STD_LOGIC_VECTOR((N-1) downto 0);
signal PreviousModulus : STD_LOGIC_VECTOR((N-1) downto 0);
signal StableElement : STD_LOGIC;
signal CHANGELOCK_Element : STD_LOGIC_VECTOR(1 downto 0) := "11";
signal StableModulus : STD_LOGIC;
signal UnitaryU : STD_LOGIC;
signal UnitaryV : STD_LOGIC;

signal XoutModded : STD_LOGIC_VECTOR ((N-1) downto 0);
signal YoutModded : STD_LOGIC_VECTOR ((N-1) downto 0);

begin

STABLEELE : GENERIC_FAP_RELATIONAL
         Generic Map (N => N,
                  VType => 0)
    Port Map ( A => Element,
              B => PreviousElement,
              E => StableElement,
              G => open);

STABLEMOD : GENERIC_FAP_RELATIONAL
         Generic Map (N => N,
                  VType => 0)
    Port Map ( A => Modulus,
              B => PreviousModulus,
              E => StableModulus,
              G => open);
```

```
UISUNIT : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 0)
    Port Map ( A => Uout,
           B => UnitVector,
           E => UnitaryU,
           G => open);

VISUNIT : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 0)
    Port Map ( A => Vout,
           B => UnitVector,
           E => UnitaryV,
           G => open);

XOUTMOD : GENERIC_FAP_MODADDR
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( SummandA => Xout,
           SummandB => ZeroVector,
           Modulus => Modulus,
           Summation => XoutModded);

YOUTMOD : GENERIC_FAP_MODADDR
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( SummandA => Yout,
           SummandB => ZeroVector,
           Modulus => Modulus,
           Summation => YoutModded);

InternalPart : GENERIC_FAP_MODINVR_INTERNAL_CLOCKED
        Generic Map (N => N, M => M, AddrDelay => AddrDelay)
        Port Map (U => U, V => V, X => X, Y => Y,
           Uout => Uout, Vout => Vout, Xout => Xout,
           Yout => Yout, Modulus => Modulus, CLK => CLK);

InvGen : for K in 0 to (N-1) generate
begin
        Inverse(K) <= (InternalInverse(K) and StableInverse);
end generate InvGen;
```

```
process(CLK)
begin
        if        (rising_edge(CLK)) then
                if ((StableElement and StableModulus) = '1')  then
                        --If end condition met and inputs are stable, put output
                        if ((UnitaryU and CHANGELOCK_Element(1) and
                                CHANGELOCK_Element(0)) = '1') then
                                InternalInverse <= XoutModded;
                                StableInverse <= '1';
                        --If end condition met and inputs are stable, put output
                        elsif ((UnitaryV and CHANGELOCK_Element(1) and
                        CHANGELOCK_Element(0)) = '1') then
                                InternalInverse <= YoutModded;
                                StableInverse <= '1';
                        else
                                U <= Uout;
                                V <= Vout;
                                X <= Xout;
                                Y <= Yout;
                                if (CHANGELOCK_Element = "00") then
                                        CHANGELOCK_Element <= "01";
                                elsif (CHANGELOCK_Element = "01") then
                                        CHANGELOCK_Element <= "10";
                                elsif (CHANGELOCK_Element = "10") then
                                        CHANGELOCK_Element <= "11";
                                end if;
                        end if;
                else --Else, reset the signals and begin updates again.
                        PreviousElement <= Element;
                        PreviousModulus <= Modulus;
                        U <= Element;
                        V <= Modulus;
                        X <= UnitVector;
                        Y <= ZeroVector;
                        StableInverse <= '0';
                        CHANGELOCK_Element <= "00";
                end if;
        end if;
end process;

end Behavioral;
```

5.9.2 Inner Cell: One round of the BEEA

The Inner cell of the Binary Extended Euclidean Algorithm loop unwrapped.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_FAP_MODINVR_INTERNAL_CLOCKED is
         Generic (N : natural := VecLen;
                 M : natural := MultLen;
                 AddrDelay : Time := 30 ns);
    Port ( U : in STD_LOGIC_VECTOR((N-1) downto 0);
           V : in STD_LOGIC_VECTOR((N-1) downto 0);
           X : in STD_LOGIC_VECTOR((N-1) downto 0);
           Y : in STD_LOGIC_VECTOR((N-1) downto 0);
           Uout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Vout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Xout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Yout : out STD_LOGIC_VECTOR((N-1) downto 0);
           Modulus : in STD_LOGIC_VECTOR((N-1) downto 0);
           CLK : in STD_LOGIC);
end GENERIC_FAP_MODINVR_INTERNAL_CLOCKED;

architecture Behavioral of GENERIC_FAP_MODINVR_INTERNAL_CLOCKED is

---------------------------------
-----COMPONENT DECLARATIONS-----
---------------------------------
component GENERIC_FAP_RELATIONAL
    Generic (N : Natural;
        VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;
component GENERIC_FAP_LINADDRMUX
    Generic (N : natural;
        M : natural); --Terminal Length
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           S : out  STD_LOGIC_VECTOR (N downto 0));
end component;
```

```
-----------------------------
-----SIGNAL DECLARATIONS-----
-----------------------------

--Inverses
signal NotU : STD_LOGIC_VECTOR ((N-1) downto 0);
signal NotV : STD_LOGIC_VECTOR ((N-1) downto 0);
signal NotX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal NotY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal InvU : STD_LOGIC_VECTOR (N downto 0);
signal InvV : STD_LOGIC_VECTOR (N downto 0);
signal InvX : STD_LOGIC_VECTOR (N downto 0);
signal InvY : STD_LOGIC_VECTOR (N downto 0);
--Output signals for U and V

signal UsumInvV : STD_LOGIC_VECTOR (N downto 0);
signal VsumInvU : STD_LOGIC_VECTOR (N downto 0);
--Output signals for X

signal XsumInvY : STD_LOGIC_VECTOR (N downto 0);
signal XsumPrime : STD_LOGIC_VECTOR (N downto 0);
signal XsumPrimeShift : STD_LOGIC_VECTOR (N downto 0);
signal XsumPrimesumInvY : STD_LOGIC_VECTOR (N downto 0);
--Output signals for Y

signal YsumInvX : STD_LOGIC_VECTOR (N downto 0);
signal YsumPrime : STD_LOGIC_VECTOR (N downto 0);
signal YsumPrimeShift : STD_LOGIC_VECTOR (N downto 0);
signal YsumPrimesumInvX : STD_LOGIC_VECTOR (N downto 0);

--Output control signals
signal Control_UorVEven : STD_LOGIC;
signal Control_UgoreV : STD_LOGIC;
signal Control_UgV : STD_LOGIC;
signal Control_UeV : STD_LOGIC;
signal Control_XgY : STD_LOGIC;
signal Control_XeY : STD_LOGIC;
signal Control_YgX : STD_LOGIC;
signal Control_UorVEqualUnit : STD_LOGIC;
signal Control_UeUnit : STD_LOGIC;
signal Control_VeUnit : STD_LOGIC;
```

```
begin

NotU <= (not U);
NotV <= (not V);
NotX <= (not X);
NotY <= (not Y);
INVRU : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                 M => M) --Terminal Length
    Port Map ( A => NotU,
                 B => UnitVector,
                 S => InvU);
INVRV : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                 M => M) --Terminal Length
    Port Map ( A => NotV,
                 B => UnitVector,
                 S => InvV);
INVRX : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                 M => M) --Terminal Length
    Port Map ( A => NotX,
                 B => UnitVector,
                 S => InvX);
INVRY : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                 M => M) --Terminal Length
    Port Map ( A => NotY,
                 B => UnitVector,
                 S => InvY);
--Output signals for U and V
UINVV : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                 M => M) --Terminal Length
    Port Map ( A => U,
                 B => InvV((N-1) downto 0),
                 S => UsumInvV);
VINVU : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                 M => M) --Terminal Length
    Port Map ( A => V,
                 B => InvU((N-1) downto 0),
                 S => VsumInvU);
```

```
--Output signals for X
XINVY : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( A => X,
                B => InvY((N-1) downto 0),
                S => XsumInvY);
XPRIME : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( A => X,
                B => Modulus,
                S => XsumPrime);
XPRIMEINVY : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( A => XsumPrime((N-1) downto 0),
                B => InvY((N-1) downto 0),
                S => XsumPrimesumInvY);
XsumPrimeShift <= "0" & XsumPrime(N downto 1);
--Output signals for Y
YINVX : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( A => Y,
                B => InvX((N-1) downto 0),
                S => YsumInvX);
YPRIME : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( A => Y,
                B => Modulus,
                S => YsumPrime);
YPRIMEINVX : GENERIC_FAP_LINADDRMUX
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( A => YsumPrime((N-1) downto 0),
                B => InvX((N-1) downto 0),
                S => YsumPrimesumInvX);
YsumPrimeShift <= "0" & YsumPrime(N downto 1);

--Output control signals; COMPARE [U>=V], [X>=Y] in Greater Than
Control_UorVEven <= ((not U(0)) or (not V(0)));
```

```
UVGreat : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 1)
    Port Map ( A => U,
           B => V,
           E => Control_UeV,
           G => Control_UgV);
XYGreat : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 1)
    Port Map ( A => X,
           B => Y,
           E => Control_XeY,
           G => Control_XgY);
UisUnit : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 1)
    Port Map ( A => U,
           B => UnitVector,
           E => Control_UeUnit,
           G => open);
VisUnit : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 1)
    Port Map ( A => U,
           B => UnitVector,
           E => Control_VeUnit,
           G => open);
Control_UgoreV <= (Control_UgV or Control_UeV);
Control_YgX <= ((not Control_XgY) and (not Control_XeY));
Control_UorVEqualUnit <= (Control_UeUnit or Control_VeUnit);

process(CLK)
begin
if (rising_edge(CLK)) then
        if ((Control_UorVEven and (not U(0)) and
         (not Control_UorVEqualUnit)) = '1') then
                Uout <= "0" & U((N-1) downto 1);
        elsif (((not Control_UorVEven) and Control_UgoreV and
         (not Control_UorVEqualUnit)) = '1') then
                Uout <= UsumInvV((N-1) downto 0);
        else
                Uout <= U;
        end if;
        if ((Control_UorVEven and (not V(0)) and
         (not Control_UorVEqualUnit)) = '1') then
                Vout <= "0" & V((N-1) downto 1);
        elsif (((not Control_UorVEven) and (not Control_UgoreV) and
         (not Control_UorVEqualUnit)) = '1') then
                Vout <= VsumInvU((N-1) downto 0);
        else
                Vout <= V;
        end if;
        if ((Control_UorVEven and (not U(0)) and (not X(0)) and
         (not Control_UorVEqualUnit)) = '1') then
                Xout <= "0" & X((N-1) downto 1);
        elsif ((Control_UorVEven and (not U(0)) and X(0) and
         (not Control_UorVEqualUnit)) = '1') then
                Xout <= XsumPrimeShift((N-1) downto 0);
        elsif (((not Control_UorVEven) and Control_UgoreV and Control_XgY and
         (not Control_UorVEqualUnit)) = '1') then
                Xout <= XsumInvY((N-1) downto 0);
        elsif (((not Control_UorVEven) and Control_UgoreV and
         (not Control_XgY) and (not Control_UorVEqualUnit)) = '1') then
                Xout <= XsumPrimesumInvY((N-1) downto 0);
        else
                Xout <= X;
        end if;
        if ((Control_UorVEven and (not V(0)) and (not Y(0)) and
         (not Control_UorVEqualUnit)) = '1') then
                Yout <= "0" & Y((N-1) downto 1);
        elsif ((Control_UorVEven and (not V(0)) and Y(0) and
         (not Control_UorVEqualUnit)) = '1') then
                Yout <= YsumPrimeShift((N-1) downto 0);
        elsif (((not Control_UorVEven) and (not Control_UgoreV) and
         Control_YgX and (not Control_UorVEqualUnit)) = '1') then
                Yout <= YsumInvX((N-1) downto 0);
        elsif (((not Control_UorVEven) and (not Control_UgoreV) and
         (not Control_YgX) and (not Control_UorVEqualUnit)) = '1') then
                Yout <= YsumPrimesumInvX((N-1) downto 0);
        else
                Yout <= Y;
        end if;
end if;
end process;
end Behavioral;
```

5.10 Elliptic Curve Group Operation

Now that we have the units for Modular Addition, Subtraction, and Multiplication, we are in a position to utilise these components to construct the units for Jacobian Point Addition (JPA) and Jacobian Point Doubling (JPD), and then turn these into Jacobian Point Multiplication. Then, along with Modular Inversion, we can turn these into the Affine Point Multiplication Algorithm!

5.10.1 ECC: Jacobian Point Addition

We recall the description for JPA provided in the Mathematical Background Chapter, and use it to construct a unit for JPA that only relies on 5 multiplier cells at a time, to minimise area consumption.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED is
         Generic (NGen : natural := VecLen;
                 MGen : Natural := MultLen;
                 AddrDelay : Time := 30 ns;
                 CompDelay : Time := 30 ns;
                 Toomed : Boolean := false);
    Port ( AX : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AZ : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           BX : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           BY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           BZ : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CX : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CY : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CZ : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED;

architecture Behavioral of GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED is
```

```
--------------------------------
-----COMPONENT DECLARATIONS-----
--------------------------------
component GENERIC_FAP_MODADDR
    Generic (N : natural;
                M : natural); --Terminal Length
    Port ( SummandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           SummandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0); --Modulo.
           Summation : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;

component GENERIC_FAP_MODSUBT
    Generic (N : natural;
                M : natural);
    Port ( Minuend : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Subtrahend : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0); --Modulo.
           Difference : out  STD_LOGIC_VECTOR ((N-1) downto 0));
end component;

component GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic (N : Natural;
                M : Natural;
                AddrDelay : Time;
                CompDelay : Time;
                Toomed : Boolean);
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in STD_LOGIC_VECTOR ((N-1) downto 0);
           Product : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC;
           StableOutput : out STD_LOGIC);
end component;

component GENERIC_FAP_RELATIONAL
    Generic (N : Natural;
        VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;
```

```
-----------------------------
-----SIGNAL DECLARATIONS-----
-----------------------------

--Generic
signal A : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal B : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal C : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal D : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal E : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal F : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal G : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal H : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal I : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal J : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal K : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal L : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal M : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal N : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal O : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal P : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal Q : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal R : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal S : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal T : STD_LOGIC_VECTOR((NGen-1) downto 0);--
--9th is the stable output
signal STAGE_STABILITIES : STD_LOGIC_VECTOR(9 downto 0);

--BUFF's
signal XBUFF : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal YBUFF : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal ZBUFF : STD_LOGIC_VECTOR((NGen-1) downto 0);

--Travelling Multipliers
signal Mult1A : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult1B : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult1P : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal P1_Stability : STD_LOGIC;--
signal Mult2A : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult2B : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult2P : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal P2_Stability : STD_LOGIC;--
signal Mult3A : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult3B : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult3P : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal P3_Stability : STD_LOGIC;--
signal Mult4A : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult4B : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult4P : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal P4_Stability : STD_LOGIC;--
signal Mult5A : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult5B : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Mult5P : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal P5_Stability : STD_LOGIC;--
--Producing a period of four clock cycles of nop behaviour to
--allow the multipliers to effect the instability of the outputs.
signal CLK_Stability : STD_LOGIC_VECTOR(1 downto 0);
--Determining the stability of the inputs
signal AX_Previous : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal AY_Previous : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal AZ_Previous : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal BX_Previous : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal BY_Previous : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal BZ_Previous : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal AX_Stability : STD_LOGIC;
signal AY_Stability : STD_LOGIC;
signal AZ_Stability : STD_LOGIC;
signal BX_Stability : STD_LOGIC;
signal BY_Stability : STD_LOGIC;
signal BZ_Stability : STD_LOGIC;

signal AZ_Zero : STD_LOGIC;
signal BZ_Zero : STD_LOGIC;

begin

------------------------------
-----Stage X Mult's Ports-----
------------------------------
```

```
--1P(1A,1B)
MULT_1P : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => NGen,
                 M => MGen,
                 AddrDelay => AddrDelay,
                 CompDelay => CompDelay,
                 Toomed => Toomed)
    Port Map ( MultiplicandA => Mult1A,
                 MultiplicandB => Mult1B,
                 Modulus => Modulus,
                 Product => Mult1P,
                 CLK => CLK,
                 StableOutput => P1_Stability);

--2P(2A,2B)
MULT_2P : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => NGen,
                 M => MGen,
                 AddrDelay => AddrDelay,
                 CompDelay => CompDelay,
                 Toomed => Toomed)
    Port Map ( MultiplicandA => Mult2A,
                 MultiplicandB => Mult2B,
                 Modulus => Modulus,
                 Product => Mult2P,
                 CLK => CLK,
                 StableOutput => P2_Stability);

--3P(3A,3B)
MULT_3P : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => NGen,
                 M => MGen,
                 AddrDelay => AddrDelay,
                 CompDelay => CompDelay,
                 Toomed => Toomed)
    Port Map ( MultiplicandA => Mult3A,
                 MultiplicandB => Mult3B,
                 Modulus => Modulus,
                 Product => Mult3P,
                 CLK => CLK,
                 StableOutput => P3_Stability);
```

```
--4P(4A,4B)
MULT_4P : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => NGen,
                M => MGen,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( MultiplicandA => Mult4A,
                MultiplicandB => Mult4B,
                Modulus => Modulus,
                Product => Mult4P,
                CLK => CLK,
                StableOutput => P4_Stability);

--5P(5A,5B)
MULT_5P : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => NGen,
                M => MGen,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( MultiplicandA => Mult5A,
                MultiplicandB => Mult5B,
                Modulus => Modulus,
                Product => Mult5P,
                CLK => CLK,
                StableOutput => P5_Stability);

------------------------------------
-----Stage 3 Addr-Subt's Ports-----
------------------------------------

--F(D,C)
SUBT_F : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                M => MGen)
    Port Map ( Minuend => D,
                Subtrahend => C,
                Modulus => Modulus, --Modulo.
                Difference => F);
```

```
--G(C,D)
ADDR_G : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                M => MGen) --Terminal Length
    Port Map ( SummandA => C,
                SummandB => D,
                Modulus => Modulus, --Modulo.
                Summation => G);
--P(N,O)
SUBT_P : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                M => MGen)
    Port Map ( Minuend => N,
                Subtrahend => O,
                Modulus => Modulus, --Modulo.
                Difference => P);

-------------------------
-----Stage 6 X Port-----
-------------------------

--X(Q,I)
SUBT_X : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                M => MGen)
    Port Map ( Minuend => Q,
                Subtrahend => I,
                Modulus => Modulus, --Modulo.
                Difference => XBUFF);

---------------------------
-----Stage 7 Subt Port-----
---------------------------

--S(K,XBUFF)
SUBT_S : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                M => MGen)
    Port Map ( Minuend => K,
                Subtrahend => XBUFF,
                Modulus => Modulus, --Modulo.
                Difference => S);
```

```
-------------------------
-----Stage 9 Y Port-----
-------------------------

--Y(T,R)
SUBT_Y : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                 M => MGen)
    Port Map ( Minuend => T,
                 Subtrahend => R,
                 Modulus => Modulus, --Modulo.
                 Difference => YBUFF);

-------------------------
-----Zero Check Port-----
-------------------------

AZZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => AZ,
            B => ZeroVector,
            E => AZ_Zero,
            G => open);

BZZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => BZ,
            B => ZeroVector,
            E => BZ_Zero,
            G => open);

AX_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => AX,
            B => AX_Previous,
            E => AX_Stability,
            G => open);
```

```
AY_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => AY,
           B => AY_Previous,
           E => AY_Stability,
           G => open);

AZ_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => AZ,
           B => AZ_Previous,
           E => AZ_Stability,
           G => open);

BX_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => BX,
           B => BX_Previous,
           E => BX_Stability,
           G => open);

BY_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => BY,
           B => BY_Previous,
           E => BY_Stability,
           G => open);

BZ_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => BZ,
           B => BZ_Previous,
           E => BZ_Stability,
           G => open);
```

```
-------------------------
-----Control Process-----
-------------------------

OutGen : for K in 0 to (NGen-1) generate
begin
        CX(K) <= (((XBUFF(K) and ((not AZ_Zero) and (not BZ_Zero))) or
                (BX(K) and AZ_Zero) or (AX(K) and BZ_Zero)) and STAGE_STABILITIES(9));
        CY(K) <= (((YBUFF(K) and ((not AZ_Zero) and (not BZ_Zero))) or
                (BY(K) and AZ_Zero) or (AY(K) and BZ_Zero)) and STAGE_STABILITIES(9));
        CZ(K) <= (((ZBUFF(K) and ((not AZ_Zero) and (not BZ_Zero))) or
                (BZ(K) and AZ_Zero) or (AZ(K) and BZ_Zero)) and STAGE_STABILITIES(9));
end generate OutGen;

StableOutput <= STAGE_STABILITIES(9);

process(CLK)
begin
 if(rising_edge(CLK)) then
  if((AX_Stability and AY_Stability and AZ_Stability and
   BX_Stability and BY_Stability and BZ_Stability) = '1') then
   --Now when the inputs are stable.
   if(((not CLK_Stability(1)) and (not CLK_Stability(0))) = '1') then
    CLK_Stability(0) <= '1';
   elsif(((not CLK_Stability(1)) and (CLK_Stability(0))) = '1') then
    CLK_Stability(1) <= '1';
    CLK_Stability(0) <= '0';
   elsif(((CLK_Stability(1)) and (not CLK_Stability(0))) = '1') then
    CLK_Stability(0) <= '1';
    --Stage 9 Stabilities: YBUFF (Not Mult)
   elsif ((STAGE_STABILITIES(8) and (not STAGE_STABILITIES(9))) = '1') then
    STAGE_STABILITIES(9) <= '1';
    --Don't save anything, stability has been reached.
    --Stage 8 Stabilities: T, R (Mult)
   elsif ((STAGE_STABILITIES(7) and P1_Stability and P2_Stability) = '1') then
    STAGE_STABILITIES(8) <= '1';
    --Save TR
    T <= Mult1P;
    R <= Mult2P;
    --Don't make anything, next round is just subt.
    CLK_Stability <= "00";
    --Stage 7 Stabilities: S (Not Mult)
   elsif ((STAGE_STABILITIES(6) and (not STAGE_STABILITIES(7))) = '1') then
```

```
    STAGE_STABILITIES(7) <= '1';
    --Don't save anything
    --Make RT
    --T(P,S)
    Mult1A <= P;
    Mult1B <= S;
    --R(N,J)
    Mult2A <= N;
    Mult2B <= J;
    CLK_Stability <= "00";
    --Stage 6 Stabilities: XBUFF (Not Mult)
   elsif ((STAGE_STABILITIES(5) and (not STAGE_STABILITIES(6))) = '1') then
    STAGE_STABILITIES(6) <= '1';
    --Don't save anything
    --Don't make anything for the next round
    CLK_Stability <= "00";
    --Stage 5 Stabilities: I, J, K (All mults)
   elsif ((STAGE_STABILITIES(4) and P1_Stability and
    P2_Stability and P3_Stability) = '1') then
    STAGE_STABILITIES(5) <= '1';
    --Save IJK
    I <= Mult1P;
    J <= Mult2P;
    K <= Mult3P;
    --Don't make anything for the next round.
    CLK_Stability <= "00";
    --Stage 4 Stabilities: ZBUFF, H, Q (All mults)
   elsif ((STAGE_STABILITIES(3) and P1_Stability and
    P2_Stability and P3_Stability) = '1') then
    STAGE_STABILITIES(4) <= '1';
    --Save ZHQ
    ZBUFF <= Mult1P;
    H <= Mult2P;
    Q <= Mult3P;
    --Make IJK
    --I(G,H)
    Mult1A <= G;
    Mult1B <= Mult2P;
    --J(H,F)
    Mult2A <= Mult2P;
    Mult2B <= F;
    --K(D,H)
    Mult3A <= D;
```

```
 Mult3B <= Mult2P;
 CLK_Stability <= "00";
 --Stage 3 Stabilities: F, G, P (No mults)
elsif ((STAGE_STABILITIES(2) and (not STAGE_STABILITIES(3))) = '1') then
 STAGE_STABILITIES(3) <= '1';
 --Nothing to save
 --Make ZHQ
 --ZBUFF--Z(F,E)
 Mult1A <= F;
 Mult1B <= E;
 --H(F,F)
 Mult2A <= F;
 Mult2B <= F;
 --Q(P,P)
 Mult3A <= P;
 Mult3B <= P;
 CLK_Stability <= "00";
 --Stage 2 Stabilities: C, D, N, O (All mults)
elsif ((STAGE_STABILITIES(1) and P1_Stability and
 P2_Stability and P3_Stability and P4_Stability) = '1') then
 STAGE_STABILITIES(2) <= '1';
 --Save CDNO
 C <= Mult1P;
 D <= Mult2P;
 N <= Mult3P;
 O <= Mult4P;
 --Don't make anything for the next round,
 --it's only subt/addr
 CLK_Stability <= "00";
 --Stage 1 Stabilities: A, B, E, L, M (All mults)
elsif((STAGE_STABILITIES(0) and P1_Stability and P2_Stability and
 P3_Stability and P4_Stability and P5_Stability) = '1') then
 STAGE_STABILITIES(1) <= '1';
 --Save ABELM
 A <= Mult1P;
 B <= Mult2P;
 E <= Mult3P;
 L <= Mult4P;
 M <= Mult5P;
 --Make CDNO
 --C(A,BX)
 Mult1A <= Mult1P;
 Mult1B <= BX;
```

```
    --D(B,AX)
    Mult2A <= Mult2P;
    Mult2B <= AX;
    --N(L,B)
    Mult3A <= Mult4P;
    Mult3B <= Mult2P;
    --O(M,A)
    Mult4A <= Mult5P;
    Mult4B <= Mult1P;
    CLK_Stability <= "00";
   elsif (STAGE_STABILITIES(0) = '0') then
    STAGE_STABILITIES(0) <= '1';
   end if;
  else
   --If neither input is a point on the line at infinity,
   --and the inputs have changed, then rehash the process.
   STAGE_STABILITIES <= (others => '0');
   AX_Previous <= AX;
   AY_Previous <= AY;
   AZ_Previous <= AZ;
   BX_Previous <= BX;
   BY_Previous <= BY;
   BZ_Previous <= BZ;
   --Stack the first lot on the stages
   --Make ABELM
   --A(AZ,AZ)
   Mult1A <= AZ;
   Mult1B <= AZ;
   --B(BZ,BZ)
   Mult2A <= BZ;
   Mult2B <= BZ;
   --E(AZ,BZ)
   Mult3A <= AZ;
   Mult3B <= BZ;
   --L(BZ,AY)
   Mult4A <= BZ;
   Mult4B <= AY;
   --M(AZ,BY)
   Mult5A <= AZ;
   Mult5B <= BY;
   CLK_Stability <= "00";
  end if;
 end if; end process; end Behavioral;
```

5.10.2 ECC: Jacobian Point Doubling

And now we recall the description of JPD in the Mathematical Background Chapter, and use it to construct a unit for JPD that only relies on 4 multiplier units, to conserve space. We can also eliminate space in this by declaring that it contains all the same component declarations, almost all the same signals (with the two unique to this declared in it), and we can skip the instantiation of the four modular multipliers, as they are identical to those written in the JPA.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;
entity GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED is
         Generic (NGen : natural := VecLen; MGen : Natural := MultLen;
                 AddrDelay : Time := 30 ns; CompDelay : Time := 30 ns;
                 Toomed : Boolean := false);
    Port ( AX : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AZ : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CX : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CY : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CZ : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           ECC_A : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED;

architecture Behavioral of GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED is

------------------------------ ***********************
-----SIGNAL DECLARATIONS----- *Signals unique to JPD*
------------------------------ ***********************

signal U : STD_LOGIC_VECTOR((NGen-1) downto 0);--
signal AY_Zero : STD_LOGIC;
```

```
------------------------------
-----Stage 1 Addr's Ports-----
------------------------------

--O(AX,AX)
ADDR_O : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( SummandA => AX,
               SummandB => AX,
               Modulus => Modulus, --Modulo.
               Summation => O);

------------------------------
-----Stage 2 Addr's Ports-----
------------------------------
--K(B,B)
ADDR_K : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( SummandA => B,
               SummandB => B,
               Modulus => Modulus, --Modulo.
               Summation => K);
--P(O,O)
ADDR_P : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( SummandA => O,
               SummandB => O,
               Modulus => Modulus, --Modulo.
               Summation => P);

--ZBUFF(A,A)
ADDR_Z : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( SummandA => A,
               SummandB => A,
               Modulus => Modulus, --Modulo.
               Summation => ZBUFF);
```

```
------------------------------
-----Stage 3 Addr's Ports-----
------------------------------

--G(F,F)
ADDR_G : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                M => MGen) --Terminal Length
    Port Map ( SummandA => F,
                SummandB => F,
                Modulus => Modulus, --Modulo.
                Summation => G);

--L(K,B)
ADDR_L : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                M => MGen) --Terminal Length
    Port Map ( SummandA => K,
                SummandB => B,
                Modulus => Modulus, --Modulo.
                Summation => L);

--Q(P,P)
ADDR_Q : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                M => MGen) --Terminal Length
    Port Map ( SummandA => P,
                SummandB => P,
                Modulus => Modulus, --Modulo.
                Summation => Q);

------------------------------
-----Stage 4 Addr's Ports-----
------------------------------

--H(G,G)
ADDR_H : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                M => MGen) --Terminal Length
    Port Map ( SummandA => G,
                SummandB => G,
                Modulus => Modulus, --Modulo.
                Summation => H);
```

```
--M(L,J)
ADDR_M : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( SummandA => L,
                 SummandB => J,
                 Modulus => Modulus, --Modulo.
                 Summation => M);
-----------------------------
-----Stage 5 Addr's Port-----
-----------------------------
--I(H,H)
ADDR_I : GENERIC_FAP_MODADDR
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( SummandA => H,
                 SummandB => H,
                 Modulus => Modulus, --Modulo.
                 Summation => I);
---------------------------
-----Stage 6 X's Ports-----
---------------------------
--XBUFF(N,R)
SUBT_X : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( Minuend => N,
                 Subtrahend => R,
                 Modulus => Modulus, --Modulo.
                 Difference => XBUFF);
----------------------------
-----Stage 7 Subt Ports-----
----------------------------
--T(S,XBUFF)
SUBT_T : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                 M => MGen) --Terminal Length
    Port Map ( Minuend => S,
                 Subtrahend => XBUFF,
                 Modulus => Modulus, --Modulo.
                 Difference => T);
```

```
---------------------------
-----Stage 9 Y's Ports-----
---------------------------
--YBUFF(U,I)
SUBT_Y : GENERIC_FAP_MODSUBT
    Generic Map (N => NGen,
                M => MGen) --Terminal Length
    Port Map ( Minuend => U,
                Subtrahend => I,
                Modulus => Modulus, --Modulo.
                Difference => YBUFF);
-------------------------
-----Zero Check Port-----
-------------------------
AZZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => AZ,
           B => ZeroVector,
           E => AZ_Zero,
           G => open);

AYZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => AY,
           B => ZeroVector,
           E => AY_Zero,
           G => open);
AX_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => AX,
           B => AX_Previous,
           E => AX_Stability,
           G => open);
AY_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => AY,
           B => AY_Previous,
           E => AY_Stability,
           G => open);
```

```
AZ_STABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => AZ,
           B => AZ_Previous,
           E => AZ_Stability,
           G => open);
-------------------------
-----Control Process-----
-------------------------
OutGen : for K in 0 to (NGen-1) generate
begin
        CX(K) <= (((XBUFF(K) and ((not AZ_Zero) and (not AY_Zero))) or
         (UnitVector(K) and (AZ_Zero or AY_Zero))) and STAGE_STABILITIES(9));
        CY(K) <= (((YBUFF(K) and ((not AZ_Zero) and (not AY_Zero))) or
         (UnitVector(K) and (AZ_Zero or AY_Zero))) and STAGE_STABILITIES(9));
        CZ(K) <= (((ZBUFF(K) and ((not AZ_Zero) and (not AY_Zero))) or
         (ZeroVector(K) and (AZ_Zero or AY_Zero))) and STAGE_STABILITIES(9));
end generate OutGen;
StableOutput <= STAGE_STABILITIES(9);


process(CLK)
begin
 if(rising_edge(CLK)) then
  if((AX_Stability and AY_Stability and AZ_Stability) = '1') then
  --Now when the inputs are stable.
   if(((not CLK_Stability(1)) and (not CLK_Stability(0))) = '1') then
    CLK_Stability(0) <= '1';
   elsif(((not CLK_Stability(1)) and (CLK_Stability(0))) = '1') then
    CLK_Stability(1) <= '1';
    CLK_Stability(0) <= '0';
   elsif(((CLK_Stability(1)) and (not CLK_Stability(0))) = '1') then
    CLK_Stability(0) <= '1';
   --Stage 9 Stabilities: Y (Subt)
   elsif ((STAGE_STABILITIES(8) and (not STAGE_STABILITIES(9))) = '1') then
    STAGE_STABILITIES(9) <= '1';
    --Save nothing
    --Make nothing
    --Stage 8 Stabilities: U (Mult);
   elsif ((STAGE_STABILITIES(7) and P1_Stability) = '1') then
    STAGE_STABILITIES(8) <= '1';
    --Save U
 U <= Mult1P;
 --Make nothing
 CLK_Stability <= "00";
--Stage 7 Stabilities: T (Subt)
elsif ((STAGE_STABILITIES(6) and (not STAGE_STABILITIES(7))) = '1') then
 STAGE_STABILITIES(7) <= '1';
 --Save nothing
 --Make U
 --U(T,M)
 Mult1A <= T;
 Mult1B <= M;
 CLK_Stability <= "00";
--Stage 6 Stabilities: X (Subt)
elsif ((STAGE_STABILITIES(5) and (not STAGE_STABILITIES(6))) = '1') then
 STAGE_STABILITIES(6) <= '1';
 --Save nothing
 --Make nothing
 CLK_Stability <= "00";
--Stage 5 Stabilities: N (Mult); I (Addr)
elsif ((STAGE_STABILITIES(4) and P1_Stability) = '1') then
 STAGE_STABILITIES(5) <= '1';
 --Save N
 N <= Mult1P;
 --Make nothing
 CLK_Stability <= "00";
 --Stage 4 Stabilities: R (Mult); H, M (Addrs)
elsif ((STAGE_STABILITIES(3) and P1_Stability) = '1') then
 STAGE_STABILITIES(4) <= '1';
 --Save R
 R <= Mult1P;
 --Make N
 --N(M,M)
 Mult1A <= M;
 Mult1B <= M;
 CLK_Stability <= "00";
--Stage 3 Stabilities: J, S (Mults); G, L, Q (Addrs)
elsif ((STAGE_STABILITIES(2) and P1_Stability and P2_Stability) = '1') then
 STAGE_STABILITIES(3) <= '1';
 --Save JS
 J <= Mult1P;
 S <= Mult2P;
 --Make R
 --R(Q,C)
```

```
    Mult1A <= Q;Mult1B <= C;
    CLK_Stability <= "00";
   --Stage 2 Stabilities: E, F (Mults); K, P, Z (Addrs)
   elsif ((STAGE_STABILITIES(1) and P1_Stability and P2_Stability) = '1') then
    STAGE_STABILITIES(2) <= '1';
    --Save EF
    E <= Mult1P;
    F <= Mult2P;
    --Make JS
    --J(ECC_A,E)
    Mult1A <= ECC_A;Mult1B <= Mult1P;
    --S(P,C)
    Mult2A <= P; Mult2B <= C;
    CLK_Stability <= "00";
   --Stage 1 Stabilities: A, B, C, D (Mults); O (Addr)
   elsif((STAGE_STABILITIES(0) and P1_Stability and P2_Stability and
    P3_Stability and P4_Stability) = '1') then
    STAGE_STABILITIES(1) <= '1';
    --Save ABCD
    A <= Mult1P; B <= Mult2P;
    C <= Mult3P; D <= Mult4P;
    --Make EF
    --E(D,D)
    Mult1A <= Mult4P;  Mult1B <= Mult4P;
    --F(C,C)
    Mult2A <= Mult3P; Mult2B <= Mult3P;
    CLK_Stability <= "00";
   elsif (STAGE_STABILITIES(0) = '0') then
    STAGE_STABILITIES(0) <= '1';
   end if;
  else
   --If neither input is a point on the line at infinity,
   --and the inputs have changed, then rehash the process.
   STAGE_STABILITIES <= (others => '0');
   AX_Previous <= AX; AY_Previous <= AY; AZ_Previous <= AZ;
   --Stack the first lot on the stages
   --Make ABCD
   --A(AY,AZ)
   Mult1A <= AY; Mult1B <= AZ;
   --B(AX,AX)
   Mult2A <= AX; Mult2B <= AX;
   --C(AY,AY)
   Mult3A <= AY; Mult3B <= AY;
   --D(AZ,AZ)
   Mult4A <= AZ; Mult4B <= AZ;
   CLK_Stability <= "00";
  end if;
 end if;
end process;
end Behavioral;
```

5.11 Elliptic Curve Group Action

Now that we have demonstrated the logic for the JPA and JPD units, it is time to use both of these to establish the Jacobian Point Multiplier (JPM) cell. This is one of the major milestone units in the design process.

5.11.1 ECC: Jacobian Point Multiplication

Here we present the JPM unit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;

entity GENERIC_ECC_JACOBIAN_POINT_MULTIPLY_CLOCKED is
    Generic (NGen : natural := VecLen;
        MGen : Natural := MultLen;
        AddrDelay : Time := 30 ns;
        CompDelay : Time := 30 ns;
                Toomed : Boolean := false);
    Port ( KEY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           JPX : in STD_LOGIC_VECTOR ((NGen-1) downto 0);
           JPY : in STD_LOGIC_VECTOR ((NGen-1) downto 0);
           JPZ : in STD_LOGIC_VECTOR ((NGen-1) downto 0);
           JQX : out STD_LOGIC_VECTOR ((NGen-1) downto 0);
           JQY : out STD_LOGIC_VECTOR ((NGen-1) downto 0);
           JQZ : out STD_LOGIC_VECTOR ((NGen-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           ECC_A : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end GENERIC_ECC_JACOBIAN_POINT_MULTIPLY_CLOCKED;

architecture Behavioral of GENERIC_ECC_JACOBIAN_POINT_MULTIPLY_CLOCKED is
```

```
--------------------------------
-----COMPONENT DECLARATIONS-----
--------------------------------

component GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED is
    Generic (NGen : natural := VecLen;
        MGen : Natural := MultLen;
        AddrDelay : Time := 30 ns;
        CompDelay : Time := 30 ns;
        Toomed : Boolean := false);
    Port ( AX : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AZ : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CX : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CY : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CZ : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           ECC_A : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end component;

component GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED is
    Generic (NGen : natural := VecLen;
        MGen : Natural := MultLen;
        AddrDelay : Time := 30 ns;
        CompDelay : Time := 30 ns;
        Toomed : Boolean := false);
    Port ( AX : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           AZ : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           BX : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           BY : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           BZ : in  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CX : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CY : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CZ : out  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((NGen-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end component;
```

```
component GENERIC_FAP_RELATIONAL
    Generic (N : Natural;
        VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;

------------------------------
-----SIGNAL DECLARATIONS-----
------------------------------
--MemoryDataLine
signal MemDoublingX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemDoublingY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemDoublingZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemAddingX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemAddingY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemAddingZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemNonceX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemNonceY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal MemNonceZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
--Temporal Adder Inputs and Outputs for the port map
signal ADDRAX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRAY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRAZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRBX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRBY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRBZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRCX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRCY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal ADDRCZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
--Temporal Doubling Inputs and Outputs for the port map
signal DOUBAX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal DOUBAY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal DOUBAZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal DOUBCX : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal DOUBCY : STD_LOGIC_VECTOR ((NGen-1) downto 0);
signal DOUBCZ : STD_LOGIC_VECTOR ((NGen-1) downto 0);
--Used to hold the result of the 'stable' output
signal InternalJQX : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal InternalJQY : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal InternalJQZ : STD_LOGIC_VECTOR((NGen-1) downto 0);
```

```
--Used to control whether the stabalised outputs are displayed
--or not, turns to '1' when they are stable.
signal StableOutputInner : STD_LOGIC;
--Used to record the previous state of the input, to reflush and restart
--the procedure when new inputs are given
signal PreviousJPX : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal PreviousJPY : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal PreviousJPZ : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal PreviousKEY : STD_LOGIC_VECTOR((NGen-1) downto 0);
--Hold the index and used to check that the entirety of K has been realised
signal Jindex : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal JComplete : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal Jnext : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal CLKDBL : STD_LOGIC; --Used to offset the input to output timing for
--inputs to the addr and doub cells, to place inputs on them in one clock
--cycle and then continuously check the output stability in
--every following clock cycles
--Stability
signal StableADDR : STD_LOGIC;
signal StableDOUB : STD_LOGIC;
signal JPX_Stable : STD_LOGIC;
signal JPY_Stable : STD_LOGIC;
signal JPZ_Stable : STD_LOGIC;
signal KEY_Stable : STD_LOGIC;
signal J_Finished : STD_LOGIC;
signal Adding_Round : STD_LOGIC;
signal Adding_Round_Next : STD_LOGIC;
signal JK : STD_LOGIC_VECTOR((NGen-1) downto 0);
signal JKnext : STD_LOGIC_VECTOR((NGen-1) downto 0);

begin

------------------------------
-----Stability Check Port-----
------------------------------

JPXSTABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                VType => 0)
    Port Map ( A => JPX,
          B => PreviousJPX,
          E => JPX_Stable,
          G => open);
```

```
JPYSTABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => JPY,
           B => PreviousJPY,
           E => JPY_Stable,
           G => open);

JPZSTABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => JPZ,
           B => PreviousJPZ,
           E => JPZ_Stable,
           G => open);

KSTABLE : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => KEY,
           B => PreviousKEY,
           E => KEY_Stable,
           G => open);

ADDRROUND : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => JK,
           B => ZeroVector,
           E => Adding_Round,
           G => open);

ADDRROUNDNEXT : GENERIC_FAP_RELATIONAL
    Generic Map (N => NGen,
                 VType => 0)
    Port Map ( A => JKnext,
           B => ZeroVector,
           E => Adding_Round_Next,
           G => open);
```

```
-------------------------
-----Control Process-----
-------------------------

--Display the internal outputs only when they are stable
OutGen : for K in 0 to (NGen-1) generate
begin
        JQX(K) <= (InternalJQX(K) and StableOutputInner);
        JQY(K) <= (InternalJQY(K) and StableOutputInner);
        JQZ(K) <= (InternalJQZ(K) and StableOutputInner);
        ADDRAX(K) <= ((MemAddingX(K) and ((not Adding_Round) or
         (not Adding_Round_Next))) or (MemNonceX(K) and
          (Adding_Round and Adding_Round_Next)));
        ADDRAY(K) <= ((MemAddingY(K) and ((not Adding_Round) or
         (not Adding_Round_Next))) or (MemNonceY(K) and
          (Adding_Round and Adding_Round_Next)));
        ADDRAZ(K) <= ((MemAddingZ(K) and ((not Adding_Round) or
         (not Adding_Round_Next))) or (MemNonceZ(K) and
          (Adding_Round and Adding_Round_Next)));
end generate OutGen;

StableOutput <= StableOutputInner;

--Connecting the data bus lines of the ports.
ADDRBX <= MemDoublingX;
ADDRBY <= MemDoublingY;
ADDRBZ <= MemDoublingZ;
DOUBAX <= MemDoublingX;
DOUBAY <= MemDoublingY;
DOUBAZ <= MemDoublingZ;
JK <= (Jindex and KEY);
JKnext <= (Jnext and KEY);

process(CLK)
begin
 if (rising_edge(CLK)) then
  if ((JPX_Stable and JPY_Stable and JPZ_Stable and KEY_Stable) = '1') then
  --If the Inputs are stable
   if (J_Finished = '1') then --If end condition met, put output
    InternalJQX <= MemAddingX;
    InternalJQY <= MemAddingY;
    InternalJQZ <= MemAddingZ;
    StableOutputInner <= '1';
```

```
elsif (CLKDBL = '0') then
 CLKDBL <= '1';
elsif ((StableADDR and StableDOUB) = '1') then
--Else, if the cells are stable, do the update.
 if (Adding_Round = '0') then
 --if adding bit, then do the adding (zero from 'not' equal to the ZSeroVector)
  MemAddingX <= ADDRCX;
  MemAddingY <= ADDRCY;
  MemAddingZ <= ADDRCZ;
 else
  MemNonceX <= ADDRCX;
  MemNonceY <= ADDRCY;
  MemNonceZ <= ADDRCZ;
 end if;
 --Do the doubling unambiguously.
 MemDoublingX <= DOUBCX;
 MemDoublingY <= DOUBCY;
 MemDoublingZ <= DOUBCZ;
 if (Jindex(NGen-1) = '1') then
  J_Finished <= '1';
 end if;
 Jindex <= Jindex((NGen-2) downto 0) & "0";
 Jnext <= Jnext((NGen-2) downto 0) & "0";
 JComplete <= (JComplete or JK);
 CLKDBL <= '0';
end if;
else --Else, reset the signals and begin updates again.
 --Stability Checkers
 PreviousJPX <= JPX;
 PreviousJPY <= JPY;
 PreviousJPZ <= JPZ;
 PreviousKEY <= KEY;
 StableOutputInner <= '0';
 --Inititialise N (Doubling Feedback) and
 --Q (Adder Feedback) and the index
 MemDoublingX <= JPX;
 MemDoublingY <= JPY;
 MemDoublingZ <= JPZ;
 MemAddingX <= UnitVector;
 MemAddingY <= UnitVector;
 MemAddingZ <= ZeroVector;
 MemNonceX <= UnitVector;
 MemNonceY <= UnitVector;
```

```
    MemNonceZ <= ZeroVector;
    Jindex <= UnitVector;
    JComplete <= ZeroVector;
    Jnext <= UnitVector((NGen-2) downto 0) & "0";
    CLKDBL <= '0';
    J_Finished <= '0';
   end if;
  end if;
end process;
--The adder cell
ADDR : GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED
    Generic Map (NGen => NGen, MGen => MGen,
                AddrDelay => AddrDelay, CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( AX => ADDRAX,
                AY => ADDRAY,
                AZ => ADDRAZ,
                BX => ADDRBX,
                BY => ADDRBY,
                BZ => ADDRBZ,
                CX => ADDRCX,
                CY => ADDRCY,
                CZ => ADDRCZ,
                Modulus => Modulus,
                CLK => CLK,
                StableOutput => StableADDR);
--The doubling cell
DOUB : GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED
    Generic Map (NGen => NGen, MGen => MGen,
                AddrDelay => AddrDelay, CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( AX => DOUBAX,
                AY => DOUBAY,
                AZ => DOUBAZ,
                CX => DOUBCX,
                CY => DOUBCY,
                CZ => DOUBCZ,
                Modulus => Modulus,
                ECC_A => ECC_A,
                CLK => CLK,
                StableOutput => StableDOUB);
end Behavioral;
```

5.11.2 ECC: Affine Point Multiplication

Now that we have a cell for JPM and Modular Inversion we can finally produce a cell for Affine Point Multiplication!

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;
entity GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED is
    Generic (N : natural := VecLen;
        M : Natural := MultLen;
        AddrDelay : Time := 30 ns;
        CompDelay : Time := 30 ns;
        Toomed : Boolean := false);
    Port ( KEY : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           APX : in STD_LOGIC_VECTOR ((N-1) downto 0);
           APY : in STD_LOGIC_VECTOR ((N-1) downto 0);
           AQX : out STD_LOGIC_VECTOR ((N-1) downto 0);
           AQY : out STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((N-1) downto 0);
           ECC_A : In  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED;

architecture Behavioral of GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED is
component GENERIC_ECC_JACOBIAN_POINT_MULTIPLY_CLOCKED
    Generic (NGen : natural;
        MGen : Natural;
        AddrDelay : Time;
        CompDelay : Time;
        Toomed : Boolean);
    Port ( KEY : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           JPX : in STD_LOGIC_VECTOR ((N-1) downto 0);
           JPY : in STD_LOGIC_VECTOR ((N-1) downto 0);
           JPZ : in STD_LOGIC_VECTOR ((N-1) downto 0);
           JQX : out STD_LOGIC_VECTOR ((N-1) downto 0);
           JQY : out STD_LOGIC_VECTOR ((N-1) downto 0);
           JQZ : out STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : In  STD_LOGIC_VECTOR ((N-1) downto 0);
           ECC_A : In  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : IN STD_LOGIC;
           StableOutput : out STD_LOGIC);
end component;
```

```
component GENERIC_FAP_MODINVR_CLOCKED
         Generic (N : natural := VecLen;
                M : natural := MultLen;
                AddrDelay : Time := 30 ns);
    Port ( Element : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Inverse : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC);
end component;

component GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
         Generic (N : Natural;
                M : Natural;
                AddrDelay : Time;
                CompDelay : Time;
                Toomed : Boolean);
    Port ( MultiplicandA : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           MultiplicandB : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           Modulus : in STD_LOGIC_VECTOR ((N-1) downto 0);
           Product : out  STD_LOGIC_VECTOR ((N-1) downto 0);
           CLK : in STD_LOGIC;
           StableOutput : out STD_LOGIC);
end component;

component GENERIC_FAP_RELATIONAL
    Generic (N : Natural;
                VType : Natural);
    Port ( A : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           B : in  STD_LOGIC_VECTOR ((N-1) downto 0);
           E : out  STD_LOGIC;
           G : out  STD_LOGIC);
end component;

signal StableOutputInner : STD_LOGIC;
signal Stability_JPM : STD_LOGIC;
signal JQX : STD_LOGIC_VECTOR((N-1) downto 0);
signal JQY : STD_LOGIC_VECTOR((N-1) downto 0);
signal JQZ : STD_LOGIC_VECTOR((N-1) downto 0);
signal AQXInner : STD_LOGIC_VECTOR((N-1) downto 0);
signal AQYInner : STD_LOGIC_VECTOR((N-1) downto 0);
signal Inverse : STD_LOGIC_VECTOR((N-1) downto 0);
signal InverseSquared : STD_LOGIC_VECTOR((N-1) downto 0);
signal InverseCubed : STD_LOGIC_VECTOR((N-1) downto 0);
signal Stability_Inverse : STD_LOGIC;
signal Stability_Squared : STD_LOGIC;
signal Stability_Cubed : STD_LOGIC;
signal Stability_AQX : STD_LOGIC;
signal Stability_AQY : STD_LOGIC;
signal Stability_JPM_Held3 : STD_LOGIC_VECTOR(2 downto 0);
signal Stability_Inverse_Held3 : STD_LOGIC_VECTOR(2 downto 0);
signal Infinity_JQZ : STD_LOGIC;
signal Infinity_APX : STD_LOGIC;
signal Infinity_APY : STD_LOGIC;
signal Infinity_Input : STD_LOGIC;

begin

StableOutput <= StableOutputInner;
Infinity_Input <= (Infinity_APX and Infinity_APY);

AQGen : For K in 0 to (N-1) generate
begin
        AQX(K) <= (AQXInner(K) and (StableOutputInner and
                (not Infinity_JQZ) and (not Infinity_Input)));
        AQY(K) <= (AQYInner(K) and (StableOutputInner and
                (not Infinity_JQZ) and (not Infinity_Input)));
end generate AQGen;

JPM : GENERIC_ECC_JACOBIAN_POINT_MULTIPLY_CLOCKED
    Generic Map (NGen => N,
                MGen => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map (KEY => KEY,
                JPX => APX,
                JPY => APY,
                JPZ => UnitVector,
                JQX => JQX,
                JQY => JQY,
                JQZ => JQZ,
                Modulus => Modulus,
                ECC_A => ECC_A,
                CLK => CLK,
                StableOutput => Stability_JPM);
```

```
MODINV : GENERIC_FAP_MODINVR_CLOCKED
    Generic Map (N => N,
                 M => M,
                 AddrDelay => AddrDelay)
    Port Map (Element => JQZ,
              Inverse => Inverse,
              Modulus => Modulus,
              CLK => CLK);

JQZISZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => JQZ,
            B => ZeroVector,
            E => Infinity_JQZ,
            G => open);

APXISZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => APX,
            B => ZeroVector,
            E => Infinity_APX,
            G => open);

APYISZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => APY,
            B => ZeroVector,
            E => Infinity_APY,
            G => open);

INVISZERO : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => Inverse,
            B => ZeroVector,
            E => Stability_Inverse,
            G => open);
```

```
MULT_INV2 : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => N,
                 M => M,
                 AddrDelay => AddrDelay,
                 CompDelay => CompDelay,
                 Toomed => Toomed)
    Port Map ( MultiplicandA => Inverse,
                 MultiplicandB => Inverse,
                 Modulus => Modulus,
                 Product => InverseSquared,
                 CLK => CLK,
                 StableOutput => Stability_Squared);

MULT_INV3 : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => N,
                 M => M,
                 AddrDelay => AddrDelay,
                 CompDelay => CompDelay,
                 Toomed => Toomed)
    Port Map ( MultiplicandA => InverseSquared,
                 MultiplicandB => Inverse,
                 Modulus => Modulus,
                 Product => InverseCubed,
                 CLK => CLK,
                 StableOutput => Stability_Cubed);

MULT_AX : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => N,
                 M => M,
                 AddrDelay => AddrDelay,
                 CompDelay => CompDelay,
                 Toomed => Toomed)
    Port Map ( MultiplicandA => JQX,
                 MultiplicandB => InverseSquared,
                 Modulus => Modulus,
                 Product => AQXInner,
                 CLK => CLK,
                 StableOutput => Stability_AQX);
```

```
MULT_AY : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => N,
                M => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( MultiplicandA => JQY,
                MultiplicandB => InverseCubed,
                Modulus => Modulus,
                Product => AQYInner,
                CLK => CLK,
                StableOutput => Stability_AQY);

process(CLK)
begin
 if        (rising_edge(CLK)) then
  if (Stability_JPM = '1') then
   if (Stability_JPM_Held3(1) = '1') then
    Stability_JPM_Held3(2) <= '1';
   elsif (Stability_JPM_Held3(0) = '1') then
    Stability_JPM_Held3(1) <= '1';
   else
    Stability_JPM_Held3(0) <= '1';
   end if;
  else
   Stability_JPM_Held3 <= "000";
  end if;
  if (((not Stability_Inverse) or (Infinity_JQZ and
   Stability_JPM_Held3(2))) = '1') then
   if (Stability_Inverse_Held3(1) = '1') then
    Stability_Inverse_Held3(2) <= '1';
   elsif (Stability_Inverse_Held3(0) = '1') then
    Stability_Inverse_Held3(1) <= '1';
   else
    Stability_Inverse_Held3(0) <= '1';
   end if;
  else
   Stability_Inverse_Held3 <= "000";
  end if;
  StableOutputInner <= ((Stability_JPM_Held3(2) and
   Stability_Inverse_Held3(2) and Stability_Squared) and
    (Stability_Cubed and Stability_AQX and Stability_AQY));
end if; end process; end Behavioral;
```

5.12 Elliptic Curve Diffie Hellman

The ECDH cell can be realised simply through a renaming of the IO pins of the Affine Point Multiplier;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.VECTOR_STANDARD.ALL;
use work.ECC_STANDARD.ALL;

entity GENERIC_ECC_ECDH_CLOCKED_PARAMETERISED is
    Generic (N : natural := VecLen; --VecLen  must match Parameter Size
             M : Natural := MultLen;
             AddrDelay : Time := 30 ns;
             CompDelay : Time := 30 ns;
             Toomed : Boolean := false;
             --Parameter Size must match VecLen
             ParameterSet : ECC_Parameters_5 := M17);
    Port ( KEY_PRIVATE : in STD_LOGIC_VECTOR ((N-1) downto 0);
        KEY_PUBLIC : in STD_LOGIC_VECTOR (((2*N)-1) downto 0);
        SHARED_SECRET : out STD_LOGIC_VECTOR (((2*N)-1) downto 0);
        CLK : IN STD_LOGIC;
        StableOutput : out STD_LOGIC);
end GENERIC_ECC_ECDH_CLOCKED_PARAMETERISED;

architecture Behavioral of GENERIC_ECC_ECDH_CLOCKED_PARAMETERISED is

COMPONENT GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED
    Generic (N : natural := VecLen;
             M : Natural := MultLen;
             AddrDelay : Time;
             CompDelay : Time;
             Toomed : Boolean);
    Port ( KEY : in  STD_LOGIC_VECTOR ((N-1) downto 0);
       APX : in STD_LOGIC_VECTOR ((N-1) downto 0);
       APY : in STD_LOGIC_VECTOR ((N-1) downto 0);
       AQX : out STD_LOGIC_VECTOR ((N-1) downto 0);
       AQY : out STD_LOGIC_VECTOR ((N-1) downto 0);
       Modulus : In  STD_LOGIC_VECTOR ((N-1) downto 0);
       ECC_A : In  STD_LOGIC_VECTOR ((N-1) downto 0);
       CLK : IN STD_LOGIC;
       StableOutput : out STD_LOGIC);
END COMPONENT;
```

```
signal KEY_PUBLIC_X : STD_LOGIC_VECTOR ((N-1) downto 0);
signal KEY_PUBLIC_Y : STD_LOGIC_VECTOR ((N-1) downto 0);
signal SHARED_SECRET_X : STD_LOGIC_VECTOR ((N-1) downto 0);
signal SHARED_SECRET_Y : STD_LOGIC_VECTOR ((N-1) downto 0);

begin

KEY_PUBLIC_X <= KEY_PUBLIC(((2*N)-1) downto N);
KEY_PUBLIC_Y <= KEY_PUBLIC((N-1) downto 0);
SHARED_SECRET <= SHARED_SECRET_X & SHARED_SECRET_Y;

CELL : GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED
    Generic Map (N => N,
                M => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( KEY => KEY_PRIVATE,
       APX => KEY_PUBLIC_X,
       APY => KEY_PUBLIC_Y,
       AQX => SHARED_SECRET_X,
       AQY => SHARED_SECRET_Y,
       Modulus => ParameterSet(0),
       ECC_A => ParameterSet(1),
       CLK => CLK,
       StableOutput => StableOutput);

end Behavioral;
```

Now, we have completed one of the units that comprise the ECC schemes, we have succesfully constructed the ECDH unit. Rather than seperately construct a ECDSA SGA unit and an SVA unit, the final unit we produce is a Cryptographic Processor, such that performs all the functions, whilst resource sharing, to be area concious of the synthesis.

5.13 Elliptic Curve Cryptographic Processor

We have now arrived at the discussion of the construction of the end goal of the project, a circuit that is capable of processing ECDH, ECDSA-SGA, and ECDSA-SVA. The cryptographic processor is designed such that it does not itself contain a hashing component, or a cryptographically secure pseudo random number generator, but requires that these be loaded externally. However, it does provide for a user to selectively load different Elliptic Curve Domain Parameter sets, operating under the condition that the bit length of the prime chosen is the bit length of the processor, and that the bit length of the "base point"'s order also be of the same bit length (or less). It can have any standard or non standard Elliptic Curve loaded into it, and it will operate on those conditions until a new curve set is given to it. Effort has been contributed to make it "interruptible safe" such that it can swap from any active task if the command input changes mid way through some other process. This section of the chapter will operate differently as it will contain more commentary on the code, rather than just presenting it.

5.13.1 Architecture: Entity Declaration

```
entity GENERIC_ECC_ECDSA_CLOCKED_SHARED is
    Generic (N : natural := VecLen; --VecLen  must match Parameter Size
        M : Natural := MultLen;
        AddrDelay : Time := 30 ns;
        CompDelay : Time := 30 ns;
        Toomed : Boolean := false;
        ParameterSet : ECC_Parameters_5 := M17); --Parameter Size must match VecLen
    Port ( DATABUS : inout  STD_LOGIC_VECTOR (((2*N)-1) downto 0) := (others => '0');
        RW : in STD_LOGIC;
        --0: Reading From
        --1: Writing To
        LacthToReadFrom : in STD_LOGIC_VECTOR (3 downto 0);
        OtherUserRegister : in STD_LOGIC_VECTOR(3 downto 0);
        --Indicates to the ECDH, Public and Shared Key
        --registers what user is being flagged. [NOT IMPLEMENTED]
        HashStrobeIn : in STD_LOGIC_VECTOR(1 downto 0);
        --Indicates to the HASH next input is ready (0), or that
        --the the Last input is ready (1) [NOT IMPLEMENTED]
        HashStrobeOut : Out STD_LOGIC := '0';
        --Used by the HASH to request the next input [NOT IMPLEMENTED]
        Command : in STD_LOGIC_VECTOR (3 downto 0);
        CLK : IN STD_LOGIC;
        StableOutput : out STD_LOGIC := '0';
        Error : out STD_LOGIC_VECTOR(3 downto 0) := "0000");
end GENERIC_ECC_ECDSA_CLOCKED_SHARED;
```

5.13.2 Architecture: Commands, Errors, LatchToReadFrom

Of the control registers, the LatchToReadFrom indicates, during a Read/Write Command operation, which register is being targeted to save the data currently on the DATABUS port. The registers that is targets are (given in file);

```
--0000: Curve_Prime [((2*N)-1) downto N] & Curve_A [(N-1) downto 0]
--0001: Curve_GX [((2*N)-1) downto N] & Curve_GY [(N-1) downto 0]
--0010: Curve_B [((2*N)-1) downto N] & Curve_N [(N-1) downto 0]
--0011: -
--0100: Key_Private [(N-1) downto 0]
--0101: Key_Public_User_X [((2*N)-1) downto N] &
--      Key_Public_User_Y [(N-1) downto 0]
--                Note this is the Public key corresponding to
--                the private key
--0110: Key_Public_Other_X [((2*N)-1) downto N] &
--      Key_Public_Other_Y [(N-1) downto 0]
--                Note that this is not the public key corresponding
--                to the private key. It is another entity's
--                public key for use in DHKE
--0111: Key_Shared_X [((2*N)-1) downto N] & Key_Shared_Y [(N-1) downto 0]
--1000: Signature_R [((2*N)-1) downto N] & Signature_S [(N-1) downto 0]
--1001: Signature_K [(N-1) downto 0]
--                A register that can hold a precomputed Signature_K
--1010: HASH_Total [(N-1) downto 0]
--                A register that can hold a precomputed hash value
--1011: HASH_Input [((2*N)-1) downto 0]
```

Where the first three input codes are all double input commands, wherein the entire DATABUS is saved to both ports listed, mapped to from the bit range specified here. It also gives the capacity to store an externally computer Private and Public key, or it gives the opportunity to compute the Public Key corresponding to some input Private Key. It also alloes for loading of another user's Public Key, for calculation of the ECDH shared secret. It also stores the Signature values of (R, S) in registers to be read out. It also offers a registed for a precomputed Hash input, as well as leaving available an unimplemented Hashing input channel, which would utilise the HashStrobeIn and HashStrobeOut bits in the entity declaration, to control the incoming flow of the next section of an ongoing Hash stream.

The Command input port gives a user the capacity to control what action the processor is taking. The Commands offer a range of computation commands available to the user, involving commands that determine a user's public key, compute a shared secret through ECDH, computes a signature pair and verifies a signature pair. The commands as per the in file comments;

```
--0000: Not operating anything, In RW mode
--0001: [Not Implemented] Generate Key_Private; PRNG(1,(N-1))
--0010: Generate Key_Public
--0011: ECDH
--0100: [Not Implemented] HASH
--0101: [Not Implemented] Generate Signature_K; PRNG(1,(N-1))
--0110: ECDSA-SGA
--0111: ECDSA_SVA
--1000: [Not Used] --1001: [Not Used] --1010: [Not Used] --1011: [Not Used]
--1100: [Not Implemented] Error Check: Key Parameters; Other's
--1101: [Not Implemented] Error Check: Key Parameters; User's
--1110: [Not Implemented] Error Check: Curve Parameters;
--                         ((4*A^3 + 27*B^2) mod Prime != 0)
--1111: Error Check: Curve Parameters; (N*(GX,GY) = infinity)
```

Lastly we have the Error output, which was largely left unimplemented as it did not have a fixed interface requirement to provide Error information, but the sections currently exist as skeleton stubs. The Error output stubs are;

```
--0000: Default (Nothing Wrong) (Note Flag)
--0001: Illegal Read Attempt Private Key flushed to
--      Databus (Attack Flag: Flush 'Z' Drive recommended)
--0010: Invalid Curve Parameter: (N*(GX,GY) != infinity)
--      (Error Flag: IRQ recommended)
--0011: Invalid Curve Parameter: ((4*A^3 + 27*B^2) mod Prime = 0)
--      (Critical Error Flag: IRQ Mandatory)
--0100: Illegal Write Attempt: User's Public Key (Warning Flag
--      [should use Generate Key_Public command])
--0101: Illegal Write Attempt: Shared Key (Warning Flag
--      [should use ECDH command])
--0110: Invalid Public Key: User's (Error Flag: IRQ Recommended)
--0111: Invalid Public Key: Other's (Error Flag: IRQ Recommended)
--1000: Invalid SGA Attempt: Signature_K prompts Signature_R = 0
--      (Error Flag: IRQ requests new Signature_K)
--1001: Invalid SGA Attempt: Signature_K prompts Signature_S = 0
--      (Error Flag: IRQ requests new Signature_K)
--1010: Invalid SVA Attempt: Signature_R is 0 (Error Flag:
--      IRQ requests Signature rewrite)
--1011: Invalid SVA Attempt: Signature_S is 0 (Error Flag:
--      IRQ requests Signature rewrite)
--1100: Invalid SVA Attempt: Signature is Invalid (Message Flag)
--1101: Valid SVA Attempt: Signature is valid (Message Flag)
```

5.13.3 Architecture: Components and Signals

Full components are not listed here, only the caller reference to the component, by its name.

```
COMPONENT GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED;
COMPONENT GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED;
COMPONENT GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED;
COMPONENT GENERIC_FAP_MODINVR_CLOCKED;
COMPONENT GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK;
COMPONENT GENERIC_FAP_RELATIONAL;
COMPONENT GENERIC_FAP_MODADDR;
COMPONENT GENERIC_UTIL_RAM_CLOCKED;
```

The signals can be broken up into catagories of their intended usage;

```
signal DATABUSIN : STD_LOGIC_VECTOR (((2*N)-1) downto 0);
signal CLK_QUAD : STD_LOGIC_VECTOR (1 downto 0) := "00";

-----------------------
--Command Stabilities--
-----------------------
signal COMMAND_STABILITY : STD_LOGIC;
signal COMMAND_PREVIOUS : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
--001: Generate_Key_Private; PRNG(1,(N-1))
signal STABILITY_Generate_Key_Private : STD_LOGIC := '0';
--010: Generate_Key_Public
signal STABILITY_Generate_Key_Public : STD_LOGIC := '0';
--011: ECDH
signal STABILITY_ECDH : STD_LOGIC := '0';
--100: HASH
signal STABILITY_HASH : STD_LOGIC := '0';
--101: Generate_Signature_K; PRNG(1,(N-1))
signal STABILITY_Generate_Signature_K : STD_LOGIC := '0';
--110: ECDSA_SGA
signal STABILITY_ECDSA_SGA : STD_LOGIC := '0';
--111: ECDSA_SVA
signal STABILITY_ECDSA_SVA : STD_LOGIC := '0';
```

```vhdl
-----------------
--DSA Registers--
-----------------
signal STABILITY_ECDSA_ROUNDS : STD_LOGIC_VECTOR(21 downto 0) := (others => '0');
signal DSA_PM_ByG_X : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_PM_ByG_Y : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_PM_ByQ_X : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_PM_ByQ_Y : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_PA_GwQ_X : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_PA_GwQ_Y : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTU_A : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTU_B : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTU_P : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTU_M : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_U : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_U2 : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTU_STABLE : STD_LOGIC;
signal DSA_MULTV_A : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTV_B : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTV_P : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTV_M : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_V : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_V2 : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_MULTV_STABLE : STD_LOGIC;
signal DSA_ADDRW_A : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_ADDRW_B : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_ADDRW_S : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_W : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal SIGNATURE_STABLILITY_CHECK : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal SIGNATURE_STABLE : STD_LOGIC;
signal SIGNATURE_R : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JPA_JQX : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JPA_JQY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JPA_JQZ : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JPA_STABLE : STD_LOGIC;
signal DSA_JQX : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JQY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JQZ : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JQZ_INV : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JQZ_INV_SQUARED : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal DSA_JQZ_INV_CUBED : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal SIGNATURE_VERIFY : STD_LOGIC;
signal SIGNATURE_VERIFIED : STD_LOGIC;
```

```
-------------------------------------------------------------------
--Provides A way of locking operations that are clock sensitive--
-------------------------------------------------------------------
signal OP_UNIVERSAL : STD_LOGIC := '0';
signal OP_PHASELOCK : STD_LOGIC_VECTOR (1 downto 0) := "11";
signal SGA_PHASELOCK : STD_LOGIC := '0';
signal SVA_PHASELOCK : STD_LOGIC := '0';
signal SGA_TERMINAL : STD_LOGIC := '0';
signal SVA_TERMINAL : STD_LOGIC := '0';
signal JPM_PHASELOCK : STD_LOGIC := '0';
signal APM_PHASELOCK : STD_LOGIC := '0';
signal APM_TERMINAL : STD_LOGIC := '0';
signal JPA_SVA_SPECIAL : STD_LOGIC;
----------------
--PM REGISTERS--
----------------
signal PM_Modulus : STD_LOGIC_VECTOR ((N-1) downto 0);
signal PM_ECC_A : STD_LOGIC_VECTOR ((N-1) downto 0);
signal PM_StableOutput : STD_LOGIC;
signal PM_KEY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_MOD : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_ECA : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_APX : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_APY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_AQX : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_AQY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_ASTO : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JPX : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JPY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JPZ : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JQX : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JQY : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JQZ : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal PM_JSTO : STD_LOGIC_VECTOR((N-1) downto 0) := (others => '0');
signal INV_ELEMENT : STD_LOGIC_VECTOR ((N-1) downto 0) := UnitVector;
signal INV_INVERSE : STD_LOGIC_VECTOR ((N-1) downto 0);
signal INV_MODULUS : STD_LOGIC_VECTOR ((N-1) downto 0)
        := (1 => '1', others => '0');
signal DSA_INVERSE : STD_LOGIC_VECTOR ((N-1) downto 0);
signal INV_INVERSE_STABLE : STD_LOGIC;
signal APM_ROUNDS : STD_LOGIC_VECTOR(6 downto 0) := (others => '0');
```

```
--MemoryDataLine
signal MemDoublingX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemDoublingY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemDoublingZ : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemAddingX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemAddingY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemAddingZ : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemNonceX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemNonceY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal MemNonceZ : STD_LOGIC_VECTOR ((N-1) downto 0);
--Temporal Adder Inputs and Outputs for the port map
signal ADDRAX : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal ADDRAY : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal ADDRAZ : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal ADDRBX : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal ADDRBY : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal ADDRBZ : STD_LOGIC_VECTOR ((N-1) downto 0) := (others => '0');
signal ADDRCX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal ADDRCY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal ADDRCZ : STD_LOGIC_VECTOR ((N-1) downto 0);
--Temporal Doubling Inputs and Outputs for the port map
signal DOUBAX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal DOUBAY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal DOUBAZ : STD_LOGIC_VECTOR ((N-1) downto 0);
signal DOUBCX : STD_LOGIC_VECTOR ((N-1) downto 0);
signal DOUBCY : STD_LOGIC_VECTOR ((N-1) downto 0);
signal DOUBCZ : STD_LOGIC_VECTOR ((N-1) downto 0);
--Used to control whether the stabalised outputs are
--displayed or not, turns to '1' when they are stable.
signal StableOutputJPM : STD_LOGIC;
--Used to record the previous state of the input, to reflush
--and restart the procedure when new inputs are given
signal PreviousJPX : STD_LOGIC_VECTOR((N-1) downto 0);
signal PreviousJPY : STD_LOGIC_VECTOR((N-1) downto 0);
signal PreviousJPZ : STD_LOGIC_VECTOR((N-1) downto 0);
signal PreviousKEY : STD_LOGIC_VECTOR((N-1) downto 0);
--Hold the index and used to check that the entirety of K has been realised
signal Jindex : STD_LOGIC_VECTOR((N-1) downto 0);
signal JComplete : STD_LOGIC_VECTOR((N-1) downto 0);
signal Jnext : STD_LOGIC_VECTOR((N-1) downto 0);
signal CLKDBL : STD_LOGIC;
--Used to offset the input to output timing for inputs to the
--addr and doub cells, to place inputs on them in one clock
--cycle and then continuously check the output stability in
--every following clock cycles
-------------
--Stability--
-------------
signal StableADDR : STD_LOGIC;
signal StableDOUB : STD_LOGIC;
signal J_Finished : STD_LOGIC;
signal Adding_Round : STD_LOGIC;
signal Adding_Round_Next : STD_LOGIC;
signal JK : STD_LOGIC_VECTOR((N-1) downto 0);
signal JKnext : STD_LOGIC_VECTOR((N-1) downto 0);
-------
--RW!--
-------
signal RW_MODE : STD_LOGIC := '0';
signal RW_UNIVERSAL : STD_LOGIC := '0';
signal RW_PHASELOCK : STD_LOGIC_VECTOR (1 downto 0) := "11";
signal RW_IO_TRADEOVER : STD_LOGIC := '0';
signal RW_Curve_Prime : STD_LOGIC := '0';
signal DATA_Curve_Prime : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Curve_A : STD_LOGIC := '0';
signal DATA_Curve_A : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Curve_GX : STD_LOGIC := '0';
signal DATA_Curve_GX : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Curve_GY : STD_LOGIC := '0';
signal DATA_Curve_GY : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Curve_B : STD_LOGIC := '0';
signal DATA_Curve_B : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Curve_N : STD_LOGIC := '0';
signal DATA_Curve_N : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Private : STD_LOGIC := '0';
signal DATA_Key_Private : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Public_User_X : STD_LOGIC := '0';
signal DATA_Key_Public_User_X : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Public_User_Y : STD_LOGIC := '0';
signal DATA_Key_Public_User_Y : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Public_Other_X : STD_LOGIC := '0';
signal DATA_Key_Public_Other_X : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Public_Other_Y : STD_LOGIC := '0';
signal DATA_Key_Public_Other_Y : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Shared_X : STD_LOGIC := '0';
signal DATA_Key_Shared_X : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Key_Shared_Y : STD_LOGIC := '0';
signal DATA_Key_Shared_Y : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Signature_R : STD_LOGIC := '0';
signal DATA_Signature_R : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Signature_S : STD_LOGIC := '0';
signal DATA_Signature_S : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_Signature_K : STD_LOGIC := '0';
signal DATA_Signature_K : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_HASH_Total : STD_LOGIC := '0';
signal DATA_HASH_Total : STD_LOGIC_VECTOR ((N-1) downto 0) := ZeroVector;
signal RW_HASH_Input : STD_LOGIC := '0';
signal DATA_HASH_Input : STD_LOGIC_VECTOR (((2*N)-1) downto 0) := (others => '0');
```

5.13.4 Operation: RW and IO Port Maps

The RW_X pins, and DATA_X IO ports are all connected to GENERIC_UTIL_RAM_CLOCKED components. Presented here is the logic for the RW pins, that selects them to be open either if a RW cycle currently has them selected, or if an internal process is writing data to the register.

```
-----------------------------------------------------------------------------
---------------------------DATA CELLS: RW PINS-------------------------------
-----------------------------------------------------------------------------

RW_MODE <= ((not Command(0)) and (not Command(1)) and
        (not Command(2)) and (not Command(3)));
RW_UNIVERSAL <= (RW and (RW_PHASELOCK(1) xor RW_PHASELOCK(0)) and RW_MODE);
OP_UNIVERSAL <= (OP_PHASELOCK(1) xor OP_PHASELOCK(0));
RW_Curve_Prime <= (RW_UNIVERSAL and ((not LacthToReadFrom(3)) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1)) and
        (not LacthToReadFrom(0))));
RW_Curve_A <= (RW_UNIVERSAL and ((not LacthToReadFrom(3)) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1)) and
        (not LacthToReadFrom(0))));
RW_Curve_GX <= (RW_UNIVERSAL and ((not LacthToReadFrom(3)) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1))
        and LacthToReadFrom(0)));
RW_Curve_GY <= (RW_UNIVERSAL and ((not LacthToReadFrom(3)) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1))
        and LacthToReadFrom(0)));
```

```
RW_Curve_B <= (RW_UNIVERSAL and ((not LacthToReadFrom(3)) and
        (not LacthToReadFrom(2)) and LacthToReadFrom(1) and
        (not LacthToReadFrom(0))));
RW_Curve_N <= (RW_UNIVERSAL and ((not LacthToReadFrom(3)) and
        (not LacthToReadFrom(2)) and LacthToReadFrom(1) and
        (not LacthToReadFrom(0))));
RW_Key_Private <= ((RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and (not LacthToReadFrom(1)) and
        (not LacthToReadFrom(0)))) or (OP_UNIVERSAL and
        STABILITY_Generate_Key_Private and (not command(2)) and
        (not command(1)) and command(0)));
RW_Key_Public_User_X <= ((RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and (not LacthToReadFrom(1)) and
        LacthToReadFrom(0))) or (OP_UNIVERSAL and
        STABILITY_Generate_Key_Public and (not command(2)) and
        command(1) and (not command(0))));
RW_Key_Public_User_Y <= ((RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and (not LacthToReadFrom(1)) and
        LacthToReadFrom(0))) or (OP_UNIVERSAL and
        STABILITY_Generate_Key_Public and (not command(2)) and
        command(1) and (not command(0))));
RW_Key_Public_Other_X <= (RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and LacthToReadFrom(1)
        and (not LacthToReadFrom(0))));
RW_Key_Public_Other_Y <= (RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and LacthToReadFrom(1)
        and (not LacthToReadFrom(0))));
RW_Key_Shared_X <= ((RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and LacthToReadFrom(1) and
        LacthToReadFrom(0))) or (OP_UNIVERSAL and STABILITY_ECDH
        and (not command(2)) and command(1) and command(0)));
RW_Key_Shared_Y <= ((RW_UNIVERSAL and ((not LacthToReadFrom(3))
        and LacthToReadFrom(2) and LacthToReadFrom(1) and
        LacthToReadFrom(0))) or (OP_UNIVERSAL and STABILITY_ECDH
        and (not command(2)) and command(1) and command(0)));
RW_Signature_R <= ((RW_UNIVERSAL and (LacthToReadFrom(3) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1)) and
        (not LacthToReadFrom(0)))) or (OP_UNIVERSAL and STABILITY_ECDSA_SGA
        and command(2) and command(1) and (not command(0))));
RW_Signature_S <= ((RW_UNIVERSAL and (LacthToReadFrom(3) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1)) and
        (not LacthToReadFrom(0)))) or (OP_UNIVERSAL and STABILITY_ECDSA_SGA
        and command(2) and command(1) and (not command(0))));
```

```
RW_Signature_K <= ((RW_UNIVERSAL and (LacthToReadFrom(3) and
        (not LacthToReadFrom(2)) and (not LacthToReadFrom(1)) and
        LacthToReadFrom(0))) or (OP_UNIVERSAL and STABILITY_Generate_Signature_K
        and command(2) and (not command(1)) and command(0)));
RW_HASH_Total <= (RW_UNIVERSAL and (LacthToReadFrom(3) and
        (not LacthToReadFrom(2)) and LacthToReadFrom(1) and
        (not LacthToReadFrom(0))));
RW_HASH_Input <= ((RW_UNIVERSAL and (LacthToReadFrom(3) and
        (not LacthToReadFrom(2)) and LacthToReadFrom(1) and LacthToReadFrom(0)))
        or (OP_UNIVERSAL and STABILITY_HASH and command(2)
        and (not command(1)) and (not command(0))));

---------------------------------------------------------------------------
--------------------------DATA CELLS: DATA MUXS--------------------------
---------------------------------------------------------------------------

RW_IO_TRADEOVER <= ((RW_PHASELOCK(1) and (not RW_PHASELOCK(0)))
        or (OP_PHASELOCK(1) and (not OP_PHASELOCK(0))));
DATA_Curve_Prime <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Curve_Prime and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Curve_A <= DATABUSIN((N-1) downto 0) when
        ((RW_Curve_A and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Curve_GX <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Curve_GX and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Curve_GY <= DATABUSIN((N-1) downto 0) when
        ((RW_Curve_GY and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Curve_B <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Curve_B and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Curve_N <= DATABUSIN((N-1) downto 0) when
        ((RW_Curve_N and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Key_Private <= DATABUSIN((N-1) downto 0) when
        ((RW_Key_Private and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Key_Public_User_X <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Key_Public_User_X and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Key_Public_User_Y <= DATABUSIN((N-1) downto 0) when
        ((RW_Key_Public_User_Y and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Key_Public_Other_X <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Key_Public_Other_X and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Key_Public_Other_Y <= DATABUSIN((N-1) downto 0) when
        ((RW_Key_Public_Other_Y and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Key_Shared_X <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Key_Shared_X and RW_IO_TRADEOVER) = '1') else (others => 'Z');
```

```
DATA_Key_Shared_Y <= DATABUSIN((N-1) downto 0) when
        ((RW_Key_Shared_Y and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Signature_R <= DATABUSIN(((2*N)-1) downto N) when
        ((RW_Signature_R and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Signature_S <= DATABUSIN((N-1) downto 0) when
        ((RW_Signature_S and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_Signature_K <= DATABUSIN((N-1) downto 0) when
        ((RW_Signature_K and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_HASH_Total <= DATABUSIN((N-1) downto 0) when
        ((RW_HASH_Total and RW_IO_TRADEOVER) = '1') else (others => 'Z');
DATA_HASH_Input <= DATABUSIN(((2*N)-1) downto 0) when
        ((RW_HASH_Input and RW_IO_TRADEOVER) = '1') else (others => 'Z');

---------------------------------------------------------------------------
--------------------------------DATA CELLS---------------------------------
---------------------------------------------------------------------------

RAM_Curve_Prime : GENERIC_UTIL_RAM_CLOCKED
   Generic Map (N => N)
    Port Map ( RW => RW_Curve_Prime,
           Data => DATA_Curve_Prime,
           CLK => CLK);

RAM_Curve_A : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Curve_A,
           Data => DATA_Curve_A,
           CLK => CLK);

RAM_Curve_GX : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Curve_GX,
           Data => DATA_Curve_GX,
           CLK => CLK);

RAM_Curve_GY : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Curve_GY,
           Data => DATA_Curve_GY,
           CLK => CLK);
```

```
RAM_Curve_B : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Curve_B,
           Data => DATA_Curve_B,
           CLK => CLK);

RAM_Curve_N : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Curve_N,
           Data => DATA_Curve_N,
           CLK => CLK);

RAM_Key_Private : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Private,
           Data => DATA_Key_Private,
           CLK => CLK);

RAM_Key_Public_User_X : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Public_User_X,
           Data => DATA_Key_Public_User_X,
           CLK => CLK);

RAM_Key_Public_User_Y : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Public_User_Y,
           Data => DATA_Key_Public_User_Y,
           CLK => CLK);

RAM_Key_Public_Other_X : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Public_Other_X,
           Data => DATA_Key_Public_Other_X,
           CLK => CLK);

RAM_Key_Public_Other_Y : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Public_Other_Y,
           Data => DATA_Key_Public_Other_Y,
           CLK => CLK);
```

```
RAM_Key_Shared_X : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Shared_X,
           Data => DATA_Key_Shared_X,
           CLK => CLK);

RAM_Key_Shared_Y : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Key_Shared_Y,
           Data => DATA_Key_Shared_Y,
           CLK => CLK);

RAM_Signature_R : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Signature_R,
           Data => DATA_Signature_R,
           CLK => CLK);

RAM_Signature_S : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Signature_S,
           Data => DATA_Signature_S,
           CLK => CLK);

RAM_Signature_K : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_Signature_K,
           Data => DATA_Signature_K,
           CLK => CLK);

RAM_HASH_Total : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => N)
    Port Map ( RW => RW_HASH_Total,
           Data => DATA_HASH_Total,
           CLK => CLK);

RAM_HASH_Input : GENERIC_UTIL_RAM_CLOCKED
    Generic Map (N => (2*N))
    Port Map ( RW => RW_HASH_Input,
           Data => DATA_HASH_Input,
           CLK => CLK);
```

5.13.5 Operation: Non-IO Port Maps

The non IO port maps are connections to either multipliers, the JPA, JPD or Inverter cells, or relators and ADDRs. Here we present the Non-IO port maps.

```
---------------------------------------------------
-----JACOBIAN POINT MULTIPLIER CELL PORT MAPS-----
---------------------------------------------------

ADDRROUND : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 0)
    Port Map ( A => JK,
            B => ZeroVector,
            E => Adding_Round,
            G => open);

ADDRROUNDNEXT : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                VType => 0)
    Port Map ( A => JKnext,
            B => ZeroVector,
            E => Adding_Round_Next,
            G => open);

--The adder cell
ADDR : GENERIC_ECC_JACOBIAN_POINT_ADDITION_CLOCKED
    Generic Map (NGen => N,
                MGen => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( AX => ADDRAX,
                AY => ADDRAY,
                AZ => ADDRAZ,
                BX => ADDRBX,
                BY => ADDRBY,
                BZ => ADDRBZ,
                CX => ADDRCX,
                CY => ADDRCY,
                CZ => ADDRCZ,
                Modulus => PM_MOD,
                CLK => CLK,
                StableOutput => StableADDR);
```

```
--The doubling cell
DOUB : GENERIC_ECC_JACOBIAN_POINT_DOUBLE_CLOCKED
    Generic Map (NGen => N,
                MGen => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( AX => DOUBAX,
                AY => DOUBAY,
                AZ => DOUBAZ,
                CX => DOUBCX,
                CY => DOUBCY,
                CZ => DOUBCZ,
                Modulus => PM_MOD,
                ECC_A => PM_ECA,
                CLK => CLK,
                StableOutput => StableDOUB);

--Connecting the data bus lines of the ports.

DOUBAX <= MemDoublingX;
DOUBAY <= MemDoublingY;
DOUBAZ <= MemDoublingZ;
JK <= (Jindex and PM_KEY);
JKnext <= (Jnext and PM_KEY);

--Determine the ADDR inputs each round
JPA_SVA_SPECIAL <= ((STABILITY_ECDSA_ROUNDS(8) and (not
        STABILITY_ECDSA_ROUNDS(10)) and (not Command(3)))
        and (Command(2) and Command(1) and Command(0)));
OutGen : for K in 0 to (N-1) generate
begin
        ADDRAX(K) <= ((((MemAddingX(K) and ((not Adding_Round) or
                (not Adding_Round_Next))) or (MemNonceX(K) and (Adding_Round
                and Adding_Round_Next))) and (not JPA_SVA_SPECIAL)) or
                (DSA_PM_ByQ_X(K) and JPA_SVA_SPECIAL));
        ADDRAY(K) <= ((((MemAddingY(K) and ((not Adding_Round) or
                (not Adding_Round_Next))) or (MemNonceY(K) and (Adding_Round
                and Adding_Round_Next))) and (not JPA_SVA_SPECIAL)) or
                (DSA_PM_ByQ_Y(K) and JPA_SVA_SPECIAL));
        ADDRAZ(K) <= ((((MemAddingZ(K) and ((not Adding_Round) or
                (not Adding_Round_Next))) or (MemNonceZ(K) and (Adding_Round
                and Adding_Round_Next))) and (not JPA_SVA_SPECIAL)) or
                (UnitVector(K) and JPA_SVA_SPECIAL));
        ADDRBX(K) <= ((MemDoublingX(K) and (not JPA_SVA_SPECIAL)) or
                (DSA_PM_ByG_X(K) and JPA_SVA_SPECIAL));
        ADDRBY(K) <= ((MemDoublingY(K) and (not JPA_SVA_SPECIAL)) or
                (DSA_PM_ByG_Y(K) and JPA_SVA_SPECIAL));
        ADDRBZ(K) <= ((MemDoublingZ(K) and (not JPA_SVA_SPECIAL)) or
                (UnitVector(K) and JPA_SVA_SPECIAL));
end generate OutGen;

---------------------------------------------------------------------------
----------------------MODULO INVERTER and FAP RELATOR---------------------
---------------------------------------------------------------------------

inverter : GENERIC_FAP_MODINVR_CLOCKED
    Generic Map (N => N,
                                M => M,
                                AddrDelay => AddrDelay)
    Port Map ( Element => INV_ELEMENT,
           Inverse => INV_INVERSE,
           Modulus => INV_MODULUS,
           CLK => CLK);

inverter_stability : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => INV_INVERSE,
           B => ZeroVector,
           E => INV_INVERSE_STABLE,
           G => open);

signature_stability : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => SIGNATURE_STABLILITY_CHECK,
           B => ZeroVector,
           E => SIGNATURE_STABLE,
           G => open);

signature_verification : GENERIC_FAP_RELATIONAL
    Generic Map (N => N,
                 VType => 0)
    Port Map ( A => DSA_W,
           B => DATA_Signature_R,
           E => SIGNATURE_VERIFY,
           G => open);

-----------------------------------------------------------------------------
-------------------------------MULTIPLIERS----------------------------------
-----------------------------------------------------------------------------

multv : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => N,
                M => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( MultiplicandA => DSA_MULTV_A,
           MultiplicandB => DSA_MULTV_B,
           Modulus => DSA_MULTV_M,
           Product => DSA_MULTV_P,
           CLK => CLK,
           StableOutput => DSA_MULTV_STABLE);

multu : GENERIC_FAP_MODMULT_CLOCKEDCOMBS_TOOMCOOK
    Generic Map (N => N,
                M => M,
                AddrDelay => AddrDelay,
                CompDelay => CompDelay,
                Toomed => Toomed)
    Port Map ( MultiplicandA => DSA_MULTU_A,
           MultiplicandB => DSA_MULTU_B,
           Modulus => DSA_MULTU_M,
           Product => DSA_MULTU_P,
           CLK => CLK,
           StableOutput => DSA_MULTU_STABLE);

addrw : GENERIC_FAP_MODADDR
    Generic Map (N => N,
                M => M) --Terminal Length
    Port Map ( SummandA => DSA_ADDRW_A,
           SummandB => DSA_ADDRW_B,
           Modulus => DATA_Curve_N, --Modulo.
           Summation => DSA_ADDRW_S);
```

5.13.6 Operation: Control Process

The main clocked process utilises the command input and several registers to determine what to do each cycle, utilising "phase locking" registers that trap the operation in a particular cycle to operate over several adjacent clock cycles. This, along with the conditionals that check the command input provides a process that can effectively produce the Cryptographic processor. Because the Process itself is so large, it has been broken down here, and this section contains only the body of the Control Process!

```
CONTROL_PROCESS : process(CLK)
begin
 if (rising_edge(CLK)) then
  if (RW_PHASELOCK = "00") then
   DATABUSIN <= DATABUS;
   RW_PHASELOCK <= "01";
  elsif (RW_PHASELOCK = "01") then
   RW_PHASELOCK <= "10";
  elsif (RW_PHASELOCK = "10") then
   RW_PHASELOCK <= "11";
  elsif (OP_PHASELOCK = "00") then
   OP_PHASELOCK <= "01";
  elsif (OP_PHASELOCK = "01") then
   OP_PHASELOCK <= "10";
  elsif (OP_PHASELOCK = "10") then
   OP_PHASELOCK <= "11";
  elsif ((Command_Previous = Command) and (JPM_PHASELOCK = '1') and
          (APM_PHASELOCK = '1')) then
   --Do standard Jacobian PM
  elsif ((Command_Previous = Command) and (JPM_PHASELOCK = '0') and
          (APM_PHASELOCK = '1')) then
   --Do standard Affine PM using the JPM results
  elsif ((Command_Previous = Command) and (SGA_PHASELOCK = '1') and
          (SGA_TERMINAL = '0')) then
   --Implement ECDSA-SGA
  elsif ((Command_Previous = Command) and (SVA_PHASELOCK = '1') and
          (SVA_TERMINAL = '0')) then
   --Implement the SVA
  elsif (((not Command(3)) and (not Command(2))) = '1') then
   if (((not Command(1)) and (not Command(0))) = '1') then
   --000: Not operating anything, In RW mode
   elsif (((not Command(1)) and Command(0)) = '1') then
   --001: Generate Key_Private; PRNG(1,(N-1))
   elsif ((Command(1) and (not Command(0))) = '1') then
   --010: Generate Key_Public
  elsif ((Command(1) and Command(0)) = '1') then
  --011: ECDH
  end if;
 elsif (((not Command(3)) and Command(2)) = '1') then
  if (((not Command(1)) and (not Command(0))) = '1') then
  --100: HASH
  elsif (((not Command(1)) and Command(0)) = '1') then
  --101: Generate Signature_K; PRNG(1,(N-1))
  elsif ((Command(1) and (not Command(0))) = '1') then
  --110: ECDSA-SGA
  elsif ((Command(1) and Command(0)) = '1') then
  --111: ECDSA_SVA
  end if;
 elsif (Command(3) = '1') then
  if (Command(2) = '1') then
   if (Command(1) = '1') then
    if (Command(0) = '1') then
    --Command "1111": Error Check: Curve Parameters;
    --        (N*(GX,GY) = infinity)
    else
    --Command "1110": Error Check: Curve Parameters;
    --        ((4*A^3 + 27*B^2) mod Prime != 0)
    end if;
   else
    if (Command(0) = '1') then
    --Command "1101": Error Check: Key Parameters; User's
    else
    --Command "1100": Error Check: Key Parameters; Other's
    end if;
   end if;
  else
   if (Command(1) = '1') then
    if (Command(0) = '1') then
    --Command "1011"
    else
    --Command "1010"
    end if;
   else
    if (Command(0) = '1') then
    --Command "1001"
    else
    --Command "1000"
    end if; end if; end if; end if; end if; end process;
```

5.13.7 Operation: Control Process: RW

Now we can analyse the RW part of the control process! Firstly, we note that the control process begins with a series of if's surrounding the use of a "RW_PHASELOCK" register, which is switched on and initiated each time a write event is taking place, which guarantees the order of events that lead up to storing the data in a RAM block, by timing the Impedence drive internally and the placing of data on the data bus line one clock cycle apart. Here, we present the code for the explicit RW phase of the control process!

```
--000: Not operating anything, In RW mode
SGA_PHASELOCK <= '0';
SVA_PHASELOCK <= '0';
if (RW = '1') then
 StableOutput <= '0';
 DATABUS <= (others => 'Z');
 if (((not LacthToReadFrom(3)) and LacthToReadFrom(2) and
         (not LacthToReadFrom(1)) and LacthToReadFrom(0)) = '1') then
  Error <= "0100"; --0100: Illegal Write Attempt: User's Public Key
  --         (Warning Flag [should use Generate Key_Public command])
 elsif (((not LacthToReadFrom(3)) and LacthToReadFrom(2) and
         LacthToReadFrom(1) and LacthToReadFrom(0)) = '1') then
  Error <= "0101"; --0101: Illegal Write Attempt: Shared Key
  --         (Warning Flag [should use ECDH command])
 end if;
 RW_PHASELOCK <= "00";
else
 StableOutput <= '1';
 if (((not LacthToReadFrom(3)) and (not LacthToReadFrom(2)) and
         (not LacthToReadFrom(1)) and (not LacthToReadFrom(0))) = '1') then
  DATABUS <= DATA_Curve_Prime & DATA_Curve_A; --Curve_Prime and Curve_A
 elsif (((not LacthToReadFrom(3)) and (not LacthToReadFrom(2)) and
         (not LacthToReadFrom(1)) and LacthToReadFrom(0)) = '1') then
  DATABUS <= DATA_Curve_GX & DATA_Curve_GY; --Curve_GX and Curve_GY
 elsif (((not LacthToReadFrom(3)) and (not LacthToReadFrom(2)) and
         LacthToReadFrom(1) and (not LacthToReadFrom(0))) = '1') then
  DATABUS <= DATA_Curve_B & DATA_Curve_N; --Curve_B and Curve_N
 elsif (((not LacthToReadFrom(3)) and LacthToReadFrom(2) and
         (not LacthToReadFrom(1)) and (not LacthToReadFrom(0))) = '1') then
  DATABUS <= ZeroVector & DATA_Key_Private; --Key_Private
  Error <= "0001"; --0001: Illegal Read Attempt Private Key flushed to Databus
  --         (Attack Flag: Flush 'Z' Drive recommended)
 elsif (((not LacthToReadFrom(3)) and LacthToReadFrom(2) and
         (not LacthToReadFrom(1)) and LacthToReadFrom(0)) = '1') then
  DATABUS <= DATA_Key_Public_User_X & DATA_Key_Public_User_Y;
 elsif (((not LacthToReadFrom(3)) and LacthToReadFrom(2) and
         LacthToReadFrom(1) and (not LacthToReadFrom(0))) = '1') then
  DATABUS <= DATA_Key_Public_Other_X & DATA_Key_Public_Other_Y;
 elsif (((not LacthToReadFrom(3)) and LacthToReadFrom(2) and
         LacthToReadFrom(1) and LacthToReadFrom(0)) = '1') then
  DATABUS <= DATA_Key_Shared_X & DATA_Key_Shared_Y;
 elsif ((LacthToReadFrom(3) and (not LacthToReadFrom(2)) and
         (not LacthToReadFrom(1)) and (not LacthToReadFrom(0))) = '1') then
  DATABUS <= DATA_Signature_R & DATA_Signature_S;
 elsif ((LacthToReadFrom(3) and (not LacthToReadFrom(2)) and
         (not LacthToReadFrom(1)) and LacthToReadFrom(0)) = '1') then
  DATABUS <= ZeroVector & DATA_Signature_K;
 elsif ((LacthToReadFrom(3) and (not LacthToReadFrom(2)) and
         LacthToReadFrom(1) and (not LacthToReadFrom(0))) = '1') then
  DATABUS <= ZeroVector & DATA_HASH_Total;
 elsif ((LacthToReadFrom(3) and (not LacthToReadFrom(2)) and
         LacthToReadFrom(1) and LacthToReadFrom(0)) = '1') then
  DATABUS <= DATA_HASH_Input;
 end if;
end if;
```

5.13.8 Operation: Control Process: ECDH

We now present the code that sets up inputs in registers to be used by the cell unwrapped version of Jacobian Point Multiplier and Modular Inverter, for the purpose of computing the Shared Secret Key!

```
--011: ECDH
SGA_PHASELOCK <= '0';
SVA_PHASELOCK <= '0';
if (not (Command_Previous = Command)) then
 Error <= "0000";
 StableOutput <= '0';
 STABILITY_ECDH <= '0';
 --Inputs to the APM: Initiate
 APM_PHASELOCK <= '1';
 JPM_PHASELOCK <= '1';
 APM_TERMINAL <= '0';
 StableOutputJPM <= '0';
 --Inputs to the JPM: Initiate
 MemDoublingZ <= UnitVector;
 MemAddingX <= UnitVector;
 MemAddingY <= UnitVector;
 MemAddingZ <= ZeroVector;
 MemNonceX <= UnitVector;
 MemNonceY <= UnitVector;
 MemNonceZ <= ZeroVector;
 Jindex <= UnitVector;
 JComplete <= ZeroVector;
 Jnext <= UnitVector((N-2) downto 0) & "0";
 CLKDBL <= '0';
 J_Finished <= '0';
 --Inputs to the APM: Select
 PM_KEY <= DATA_Key_Private;
 PM_MOD <= DATA_Curve_Prime;
 PM_ECA <= DATA_Curve_A;
 MemDoublingX <= DATA_Key_Public_Other_X; --APX
 MemDoublingY <= DATA_Key_Public_Other_Y; --APY
else
 if (APM_TERMINAL = '1') then
  STABILITY_ECDH <= '1';
  StableOutput <= '1';
  DATABUSIN <= DSA_U & DSA_V; --DATA_Key_Public_User_X --DATA_Key_Public_User_Y
  OP_PHASELOCK <= "00";
 end if;
end if;
```

5.13.9 Operation: Control Process: SGA

Firstly, we present the code in the main control process that sets up the SGA algorithm, and locks the process to this task unless the command input changes.

```
--110: ECDSA-SGA
if (not (Command_Previous = Command)) then
 SVA_PHASELOCK <= '0';
 STABILITY_ECDSA_SVA <= '0';
 --Prepare ECDSA registers
 STABILITY_ECDSA_ROUNDS <= (others => '0');
 Error <= "0000";
 StableOutput <= '0';
 --Switch on the SGA algorithm
 SGA_PHASELOCK <= '1';
 SGA_TERMINAL <= '0';
end if;
```

Then we are now able to reproduce the code that enacts the SGA!

```
--Implement ECDSA-SGA
if (STABILITY_ECDSA_ROUNDS(15) = '1') then
        --Assert Signature_S != 0 (Recompute Signature_K otherwise)
        --CAPTURE OUTPUTS
        if ((not SIGNATURE_STABLE) = '1') then --for 'Not' Equal ZeroVector
                STABILITY_ECDSA_SGA <= '1';
                DATABUSIN <= Signature_R & DSA_U; --DATA_Signature_R_S
                OP_PHASELOCK <= "00";
                CLK_QUAD <= "00";
                StableOutput <= '1';
                SGA_TERMINAL <= '1';
        else
                Error <= "1001"; --1001: Invalid SGA Attempt
        end if;
elsif (STABILITY_ECDSA_ROUNDS(14) = '1') then
        --Assert Signature_S != 0 (Recompute Signature_K otherwise)
        --SETUP INPUTS
        SIGNATURE_STABLILITY_CHECK <= DSA_U;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(15) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(13) = '1') then
        --Compute (MULT) S = (K * (E + (Key_Private * R))) mod N
        --CAPTURE OUTPUTS
        if ((DSA_MULTU_STABLE) = '1') then
                DSA_U <= DSA_MULTU_P; --DSA_U now contains Signature_S
                STABILITY_ECDSA_ROUNDS(14) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(12) = '1') then
        --Compute (MULT) S = (K * (E + (Key_Private * R))) mod N
        --SETUP INPUTS
        DSA_MULTU_A <= DSA_INVERSE;
        DSA_MULTU_B <= DSA_W;
        DSA_MULTU_M <= DATA_Curve_N;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(13) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(11) = '1') then
        --Compute (ADDR) (E + (Key_Private * Signature_R)) mod Curve_N
        --CAPTURE OUTPUTS
        DSA_W <= DSA_ADDRW_S; --DSA_W is now (E + (Key_Private * Signature_R))
        STABILITY_ECDSA_ROUNDS(12) <= '1';
        CLK_QUAD <= "00";
elsif (STABILITY_ECDSA_ROUNDS(10) = '1') then
        --Compute (ADDR) (E + (Key_Private * Signature_R)) mod Curve_N
        --SETUP INPUTS
        DSA_ADDRW_A <= DATA_HASH_Total;
        DSA_ADDRW_B <= DSA_V;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(11) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(9) = '1') then
        --Compute (MULT) (Key_Private * Signature_R) mod Curve_N
        --CAPTURE OUTPUTS
        if ((DSA_MULTV_STABLE) = '1') then
                DSA_V <= DSA_MULTV_P;
                STABILITY_ECDSA_ROUNDS(10) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(8) = '1') then
        --Compute (MULT) (Key_Private * Signature_R) mod Curve_N
        --SETUP INPUTS
        DSA_MULTV_A <= DATA_Key_Private;
        DSA_MULTV_B <= SIGNATURE_R;
        DSA_MULTV_M <= DATA_Curve_N;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(9) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(7) = '1') then
        --Assert Signature_R != 0 (Recompute Signature_K otherwise)
        --CAPTURE OUTPUTS
        if ((not SIGNATURE_STABLE) = '1') then --for 'Not' Equal ZeroVector
                STABILITY_ECDSA_ROUNDS(8) <= '1';
                SIGNATURE_R <= DSA_W;
                CLK_QUAD <= "00";
        else
                Error <= "1000"; --1000: Invalid SGA Attempt
        end if;
elsif (STABILITY_ECDSA_ROUNDS(6) = '1') then
        --Assert Signature_R != 0 (Recompute Signature_K otherwise)
        --SETUP INPUTS
        SIGNATURE_STABLILITY_CHECK <= DSA_W;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(7) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(5) = '1') then
        --Compute Signature_R = (Signature_K * G).X mod Curve_N
        --CAPTURE OUTPUTS
        DSA_W <= DSA_ADDRW_S; --DSA_W is now Signature_R
        STABILITY_ECDSA_ROUNDS(6) <= '1';
        CLK_QUAD <= "00";
elsif (STABILITY_ECDSA_ROUNDS(4) = '1') then
        --Compute Signature_R = (Signature_K * G).X mod Curve_N
        --SETUP INPUTS
        DSA_ADDRW_A <= DSA_PM_ByG_X;
        DSA_ADDRW_B <= ZeroVector;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(5) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(3) = '1') then
        --Compute 1 PM (over Curve) of (Signature_K * G)
        --CAPTURE OUTPUTS
        if (APM_TERMINAL = '1') then
                DSA_PM_ByG_X <= DSA_U;
                DSA_PM_ByG_Y <= DSA_V;
                STABILITY_ECDSA_ROUNDS(4) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(2) = '1') then
        --Compute 1 PM (over Curve) of (Signature_K * G)
        --SETUP INPUTS
        if (CLK_QUAD = "00") then
                --Inputs to the APM: Select
                PM_KEY <= DATA_Signature_K; --K
                PM_MOD <= DATA_Curve_Prime;
                PM_ECA <= DATA_Curve_A;
                MemDoublingX <= DATA_Curve_GX; --APX
                MemDoublingY <= DATA_Curve_GY; --APY
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                --Inputs to the JPM: Initiate
                MemDoublingZ <= UnitVector;
                MemAddingX <= UnitVector;
                MemAddingY <= UnitVector;
                MemAddingZ <= ZeroVector;
                MemNonceX <= UnitVector;
                MemNonceY <= UnitVector;
                MemNonceZ <= ZeroVector;
                Jindex <= UnitVector;
                JComplete <= ZeroVector;
                Jnext <= UnitVector((N-2) downto 0) & "0";
                CLKDBL <= '0';
                J_Finished <= '0';
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
```

```
                --Inputs to the APM: Initiate
                APM_PHASELOCK <= '1';
                JPM_PHASELOCK <= '1';
                APM_TERMINAL <= '0';
                StableOutputJPM <= '0';
                CLK_QUAD <= "11";
        elsif (APM_TERMINAL = '1') then
                STABILITY_ECDSA_ROUNDS(3) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(1) = '1') then
        --Compute Inverse (over Curve_N) of Signature_K
        --CAPTURE OUTPUTS
        if (INV_INVERSE_STABLE = '0') then
                DSA_INVERSE <= INV_INVERSE;
                STABILITY_ECDSA_ROUNDS(2) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(0) = '1') then
        --Compute Inverse (over Curve_N) of Signature_K
        --SETUP INPUTS
        INV_ELEMENT <= DATA_Signature_K;
        INV_MODULUS <= DATA_Curve_N;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(1) <= '1';
        end if;
else
        --Take E from DATA_HASH_Total
        --Take Signature_K from DATA_Signature_K
        CLK_QUAD <= "00";
        STABILITY_ECDSA_ROUNDS(0) <= '1';
end if;
```

5.13.10 Operation: Control Process: SVA

Ee present the code in the main control process that sets up the SVA algorithm, and locks the process to this task unless the command input changes

```
--111: ECDSA_SVA
if (not (Command_Previous = Command)) then
 SGA_PHASELOCK <= '0';
 STABILITY_ECDSA_SGA <= '0';
 --Prepare ECDSA registers
 STABILITY_ECDSA_ROUNDS <= (others => '0');
 StableOutput <= '0';
 Error <= "0000";
 SIGNATURE_VERIFIED <= '0';
 --Switch on the SVA algorithm
 SVA_PHASELOCK <= '1';
 SVA_TERMINAL <= '0';
end if;
```

Then we are now able to reproduce the code that enacts the SVA!

```
--Implement the SVA
if (STABILITY_ECDSA_ROUNDS(21) = '1') then
        --Assert that r = w (if yes, then success)
        --CAPTURE OUTPUTS
        SIGNATURE_VERIFIED <= SIGNATURE_VERIFY;
        STABILITY_ECDSA_SVA <= '1';
        StableOutput <= '1';
        SVA_TERMINAL <= '1';
        if (SIGNATURE_VERIFY = '0') then
                Error <= "1100"; --1100: Invalid SVA Attempt
        else
                Error <= "1101"; --1101: Valid SVA Attempt
        end if;
        CLK_QUAD <= "00";
elsif (STABILITY_ECDSA_ROUNDS(20) = '1') then
        --Assert that r = w (if yes, then success)
        --SETUP INPUTS
        --Inputs are fixed to the DSA_W and DATA_Signature_R registers,
        --hold one round of the STABILITY_ECDSA to let Relator stabilise
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(21) <= '1';
        end if;
```

```
elsif (STABILITY_ECDSA_ROUNDS(19) = '1') then
        --Compute w = (uG+vQ).X mod Curve_N
        --CAPTURE OUTPUTS
        DSA_W <= DSA_ADDRW_S;
        STABILITY_ECDSA_ROUNDS(20) <= '1';
        CLK_QUAD <= "00";
elsif (STABILITY_ECDSA_ROUNDS(18) = '1') then
        --Compute w = (uG+vQ).X mod Curve_N
        --SETUP INPUTS
        DSA_ADDRW_A <= DSA_U;
        DSA_ADDRW_B <= ZeroVector;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(19) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(17) = '1') then
        --Compute APX and APY (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if ((DSA_MULTV_STABLE and DSA_MULTU_STABLE) = '1') then
                DSA_V <= DSA_MULTV_P; --DSA_V now contains APY
                DSA_U <= DSA_MULTU_P; --DSA_U now contains APX
                STABILITY_ECDSA_ROUNDS(18) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(16) = '1') then
        --Compute APX and APY (over Curve) between uG and vQ
        --SETUP INPUTS
        DSA_MULTU_A <= DSA_JQX;
        DSA_MULTU_B <= DSA_JQZ_INV_SQUARED;
        DSA_MULTU_M <= DATA_Curve_Prime;
        DSA_MULTV_A <= DSA_JQY;
        DSA_MULTV_B <= DSA_JQZ_INV_CUBED;
        DSA_MULTV_M <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
```

```
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(17) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(15) = '1') then
        --Compute JPZ_INV_CUBED (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if ((DSA_MULTV_STABLE) = '1') then
                DSA_JQZ_INV_CUBED <= DSA_MULTV_P;
                STABILITY_ECDSA_ROUNDS(16) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(14) = '1') then
        --Compute JPZ_INV_CUBED (over Curve) between uG and vQ
        --SETUP INPUTS
        DSA_MULTV_A <= DSA_JQZ_INV;
        DSA_MULTV_B <= DSA_JQZ_INV_SQUARED;
        DSA_MULTV_M <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(15) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(13) = '1') then
        --Compute JPZ_INV_SQUARED (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if ((DSA_MULTU_STABLE) = '1') then
                DSA_JQZ_INV_SQUARED <= DSA_MULTU_P;
                STABILITY_ECDSA_ROUNDS(14) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(12) = '1') then
        --Compute JPZ_INV_SQUARED (over Curve) between uG and vQ
        --SETUP INPUTS
        DSA_MULTU_A <= DSA_JQZ_INV;
        DSA_MULTU_B <= DSA_JQZ_INV;
        DSA_MULTU_M <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(13) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(11) = '1') then
        --Compute JPZ_INV (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if (INV_INVERSE_STABLE = '0') then
                DSA_JQZ_INV <= INV_INVERSE;
                STABILITY_ECDSA_ROUNDS(12) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(10) = '1') then
        --Compute JPZ_INV (over Curve) between uG and vQ
        --SETUP INPUTS
        INV_ELEMENT <= DSA_JQZ;
        INV_MODULUS <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(11) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(9) = '1') then
        --Compute JPA (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if (StableADDR = '1') then
                DSA_JQX <= ADDRCX;
                DSA_JQY <= ADDRCY;
                DSA_JQZ <= ADDRCZ;
                STABILITY_ECDSA_ROUNDS(10) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(8) = '1') then
        --Compute JPA (over Curve) between uG and vQ
        --SETUP INPUTS
        --Inputs tied permantently to the adder
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(9) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(7) = '1') then
        --Compute PM (over Curve) of vQ (Q, the persons public key)
        --CAPTURE OUTPUTS
        if (APM_TERMINAL = '1') then
                DSA_PM_ByQ_X <= DSA_U;
                DSA_PM_ByQ_Y <= DSA_V;
                STABILITY_ECDSA_ROUNDS(8) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(6) = '1') then
        --Compute PM (over Curve) of vQ (Q, the persons public key)
        --SETUP INPUTS
        if (CLK_QUAD = "00") then
                --Inputs to the APM: Select
                PM_KEY <= DSA_V2; --U = ((S Inverse) * Hash)
                PM_MOD <= DATA_Curve_Prime;
                PM_ECA <= DATA_Curve_A;
                MemDoublingX <= DATA_Key_Public_Other_X; --APX
                MemDoublingY <= DATA_Key_Public_Other_Y; --APY
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                --Inputs to the JPM: Initiate
                MemDoublingZ <= UnitVector;
                MemAddingX <= UnitVector;
                MemAddingY <= UnitVector;
                MemAddingZ <= ZeroVector;
                MemNonceX <= UnitVector;
                MemNonceY <= UnitVector;
                MemNonceZ <= ZeroVector;
                Jindex <= UnitVector;
                JComplete <= ZeroVector;
                Jnext <= UnitVector((N-2) downto 0) & "0";
                CLKDBL <= '0';
                J_Finished <= '0';
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                --Inputs to the APM: Initiate
                APM_PHASELOCK <= '1';
                JPM_PHASELOCK <= '1';
                APM_TERMINAL <= '0';
                StableOutputJPM <= '0';
                CLK_QUAD <= "11";
        elsif (APM_TERMINAL = '1') then
                --DSA_U & DSA_V; --vQ.X --vQ.Y
                STABILITY_ECDSA_ROUNDS(7) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(5) = '1') then
        --Compute PM (over Curve) of uG
        --CAPTURE OUTPUTS
        if (APM_TERMINAL = '1') then
                DSA_PM_ByG_X <= DSA_U;
                DSA_PM_ByG_Y <= DSA_V;
                STABILITY_ECDSA_ROUNDS(6) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(4) = '1') then
        --Compute PM (over Curve) of uG
        --SETUP INPUTS
        if (CLK_QUAD = "00") then
                --Inputs to the APM: Select
                PM_KEY <= DSA_U2; --U = ((S Inverse) * Hash)
                PM_MOD <= DATA_Curve_Prime;
                PM_ECA <= DATA_Curve_A;
                MemDoublingX <= DATA_Curve_GX; --APX
                MemDoublingY <= DATA_Curve_GY; --APY
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                --Inputs to the JPM: Initiate
                MemDoublingZ <= UnitVector;
                MemAddingX <= UnitVector;
                MemAddingY <= UnitVector;
                MemAddingZ <= ZeroVector;
                MemNonceX <= UnitVector;
                MemNonceY <= UnitVector;
                MemNonceZ <= ZeroVector;
                Jindex <= UnitVector;
                JComplete <= ZeroVector;
                Jnext <= UnitVector((N-2) downto 0) & "0";
                CLKDBL <= '0';
                J_Finished <= '0';
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                --Inputs to the APM: Initiate
                APM_PHASELOCK <= '1';
                JPM_PHASELOCK <= '1';
                APM_TERMINAL <= '0';
                StableOutputJPM <= '0';
                CLK_QUAD <= "11";
        elsif (APM_TERMINAL = '1') then
                --DSA_U & DSA_V; --uG.X --uG.Y
                STABILITY_ECDSA_ROUNDS(5) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(3) = '1') then
        --Compute 2 MULTS (over Curve_N) of u(Signature_S_Inv * E) and
        --v(Signature_S_Inv * Signature_R)
        --CAPTURE OUTPUTS
        if ((DSA_MULTV_STABLE and DSA_MULTU_STABLE) = '1') then
                DSA_U2 <= DSA_MULTU_P; --U = ((S Inverse) * Hash)
                DSA_V2 <= DSA_MULTV_P; --V = ((S Inverse) * R)
                STABILITY_ECDSA_ROUNDS(4) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(2) = '1') then
        --Compute 2 MULTS (over Curve_N) of u(Signature_S_Inv * E) and
        --v(Signature_S_Inv * Signature_R)
        --SETUP INPUTS
        DSA_MULTV_A <= DSA_INVERSE;
        DSA_MULTV_B <= DATA_Signature_R;
        DSA_MULTV_M <= DATA_Curve_N;
        DSA_MULTU_A <= DSA_INVERSE;
        DSA_MULTU_B <= DATA_HASH_Total;
        DSA_MULTU_M <= DATA_Curve_N;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(3) <= '1';
        end if;
elsif (STABILITY_ECDSA_ROUNDS(1) = '1') then
        --Compute Inverse (over Curve_N) of Signature_S
        --CAPTURE OUTPUTS
        if (INV_INVERSE_STABLE = '0') then
                DSA_INVERSE <= INV_INVERSE;
                STABILITY_ECDSA_ROUNDS(2) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (STABILITY_ECDSA_ROUNDS(0) = '1') then
        --Compute Inverse (over Curve_N) of Signature_S
        --SETUP INPUTS
        INV_ELEMENT <= DATA_Signature_S;
        INV_MODULUS <= DATA_Curve_N;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                STABILITY_ECDSA_ROUNDS(1) <= '1';
        end if;
else
        --Take E from DATA_HASH_Total
        --Take Signature_R from DATA_Signature_R
        --Take Signature_S from DATA_Signature_S
        if (DATA_Signature_R = ZeroVector) then
                Error <= "1010"; --1010: Invalid SVA Attempt: R is 0
        elsif (DATA_Signature_S = ZeroVector) then
                Error <= "1011"; --1011: Invalid SVA Attempt: S is 0
        else
                CLK_QUAD <= "00";
                STABILITY_ECDSA_ROUNDS(0) <= '1';
        end if;
end if;
```

5.13.11 Operation: Control Process: JPM, APM

Lastly, we present the control process that enacts JPM and APM, underpinning the SGA and SVA algorithms! This is depth one unwrapped so as to share the resources of the internal components of the APM more readilly. Here, we firstly present the JPM.

```vhdl
--Do standard Jacobian PM
if (J_Finished = '1') then --If end condition met, put output
        DSA_JQX <= MemAddingX;
        DSA_JQY <= MemAddingY;
        DSA_JQZ <= MemAddingZ;
        StableOutputJPM <= '1';
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                --At the end, switch off the JPM_PHASELOCK
                JPM_PHASELOCK <= '0';
                APM_ROUNDS <= (others => '0');
                CLK_QUAD <= "00";
        end if;
elsif (CLKDBL = '0') then
        CLKDBL <= '1';
elsif ((StableADDR and StableDOUB) = '1') then --Else, if cells stable, update.
        if (Adding_Round = '0') then --if adding bit, then do the adding
                MemAddingX <= ADDRCX;
                MemAddingY <= ADDRCY;
                MemAddingZ <= ADDRCZ;
        else
                MemNonceX <= ADDRCX;
                MemNonceY <= ADDRCY;
                MemNonceZ <= ADDRCZ;
        end if;
        --Do the doubling unambiguously.
        MemDoublingX <= DOUBCX;
        MemDoublingY <= DOUBCY;
        MemDoublingZ <= DOUBCZ;
        if (Jindex(N-1) = '1') then
                J_Finished <= '1';
        end if;
        Jindex <= Jindex((N-2) downto 0) & "0";
        Jnext <= Jnext((N-2) downto 0) & "0";
        JComplete <= (JComplete or JK);
        CLKDBL <= '0';
end if;
```

Now we present the APM.

```
--Do standard Affine PM using the JPM results
if (APM_ROUNDS(6) = '1') then
        --Compute APX and APY (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if ((DSA_MULTV_STABLE and DSA_MULTU_STABLE) = '1') then
                DSA_V <= DSA_MULTV_P; --DSA_V now contains APY
                DSA_U <= DSA_MULTU_P; --DSA_U now contains APX
                CLK_QUAD <= "11";
                --At the end, switch off the APM_PHASELOCK
                APM_PHASELOCK <= '0';
                --At the end, switch on the APM_TERMINAL
                APM_TERMINAL <= '1';
        end if;
elsif (APM_ROUNDS(5) = '1') then
        --Compute APX and APY (over Curve) between uG and vQ
        --SETUP INPUTS
        DSA_MULTU_A <= DSA_JQX;
        DSA_MULTU_B <= DSA_JQZ_INV_SQUARED;
        DSA_MULTU_M <= DATA_Curve_Prime;
        DSA_MULTV_A <= DSA_JQY;
        DSA_MULTV_B <= DSA_JQZ_INV_CUBED;
        DSA_MULTV_M <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                APM_ROUNDS(6) <= '1';
        end if;
elsif (APM_ROUNDS(4) = '1') then
        --Compute JPZ_INV_CUBED (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if ((DSA_MULTV_STABLE) = '1') then
                DSA_JQZ_INV_CUBED <= DSA_MULTV_P;
                APM_ROUNDS(5) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (APM_ROUNDS(3) = '1') then
        --Compute JPZ_INV_CUBED (over Curve) between uG and vQ
        --SETUP INPUTS
        DSA_MULTV_A <= DSA_JQZ_INV;
        DSA_MULTV_B <= DSA_JQZ_INV_SQUARED;
        DSA_MULTV_M <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                APM_ROUNDS(4) <= '1';
        end if;
elsif (APM_ROUNDS(2) = '1') then
        --Compute JPZ_INV_SQUARED (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if ((DSA_MULTU_STABLE) = '1') then
                DSA_JQZ_INV_SQUARED <= DSA_MULTU_P;
                APM_ROUNDS(3) <= '1';
                CLK_QUAD <= "00";
        end if;
elsif (APM_ROUNDS(1) = '1') then
        --Compute JPZ_INV_SQUARED (over Curve) between uG and vQ
        --SETUP INPUTS
        DSA_MULTU_A <= DSA_JQZ_INV;
        DSA_MULTU_B <= DSA_JQZ_INV;
        DSA_MULTU_M <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                APM_ROUNDS(2) <= '1';
        end if;
elsif (APM_ROUNDS(0) = '1') then
        --Compute JPZ_INV (over Curve) between uG and vQ
        --CAPTURE OUTPUTS
        if (INV_INVERSE_STABLE = '0') then
                DSA_JQZ_INV <= INV_INVERSE;
                APM_ROUNDS(1) <= '1';
                CLK_QUAD <= "00";
        end if;
else
        --Compute JPZ_INV (over Curve) between uG and vQ
        --SETUP INPUTS
        INV_ELEMENT <= DSA_JQZ;
        INV_MODULUS <= DATA_Curve_Prime;
        if (CLK_QUAD = "00") then
                CLK_QUAD <= "01";
        elsif (CLK_QUAD = "01") then
                CLK_QUAD <= "10";
        elsif (CLK_QUAD = "10") then
                CLK_QUAD <= "11";
        else
                APM_ROUNDS(0) <= '1';
        end if;
end if;
```

5.13.12 Operation: Control Process: Public Key Generation

Lastly, we present the code that calculates the Public Key, assuming a known Private Key.

```
--010: Generate Key_Public
SGA_PHASELOCK <= '0';
SVA_PHASELOCK <= '0';
if (not (Command_Previous = Command)) then
        Error <= "0000";
        StableOutput <= '0';
        --Inputs to the APM: Initiate
        APM_PHASELOCK <= '1';
        JPM_PHASELOCK <= '1';
        APM_TERMINAL <= '0';
        StableOutputJPM <= '0';
        --Inputs to the JPM: Initiate
        MemDoublingZ <= UnitVector;
        MemAddingX <= UnitVector;
        MemAddingY <= UnitVector;
        MemAddingZ <= ZeroVector;
        MemNonceX <= UnitVector;
        MemNonceY <= UnitVector;
        MemNonceZ <= ZeroVector;
        Jindex <= UnitVector;
        JComplete <= ZeroVector;
        Jnext <= UnitVector((N-2) downto 0) & "0";
        CLKDBL <= '0';
        J_Finished <= '0';
        --Inputs to the APM: Select
        PM_KEY <= DATA_Key_Private;
        PM_MOD <= DATA_Curve_Prime;
        PM_ECA <= DATA_Curve_A;
        MemDoublingX <= DATA_Curve_GX; --APX
        MemDoublingY <= DATA_Curve_GY; --APY
else
        if (APM_TERMINAL = '1') then
                STABILITY_Generate_Key_Public <= '1';
                StableOutput <= '1';
                DATABUSIN <= DSA_U & DSA_V; --DATA_Key_Public_User_X
                                            --DATA_Key_Public_User_Y
                OP_PHASELOCK <= "00";
        end if;
end if;
```

Chapter 6

VHDL Simulation

6.1 Chapter Introduction

This chapter addresses the implementations presented in chapter 5, and presents systems for evaluation of the implementations according to the component simulations. Its structure similarly follows that of chapter 5, inheriting the section and subsection structure directly, as it relates implicitly. However not all circuits have been simulated in the same way. Some have been simulated case wise evaluation, others have been simulated scanning all inputs. Others have been simulated for behavioural checking, whilst many have no simulation files as they were considered too small individually, and that their behvaiour directly impacts larger circuits that would require them to be behaving properly to be able to work themselves, and hence, the correct operation of these larger circuits is taken as the test evaluation of the smaller circuit components as well. In some cases, synthesis information is provided.

6.2 FAP: Equality and Inequality

The main testing philosophy employed in relation to the Equality and Inequality circuits is to simply assert whether they correctly identify that one input is larger than, or equal to, another. This should be true over the range of the bit length, hence, it is pertinent to test the case that any non zero number is larger than the ZeroVector, and that a vector of all 1's is larger than everything not equal to it. It is then necessary to check varying bit lengths. The simulation code used employed that both the Equality and Inequality circuits are part of the Relator circuit, so a single combined test was run on them, involving the testing inputs up to 256 bits. In the testbench file, for instance, the input commands were used;

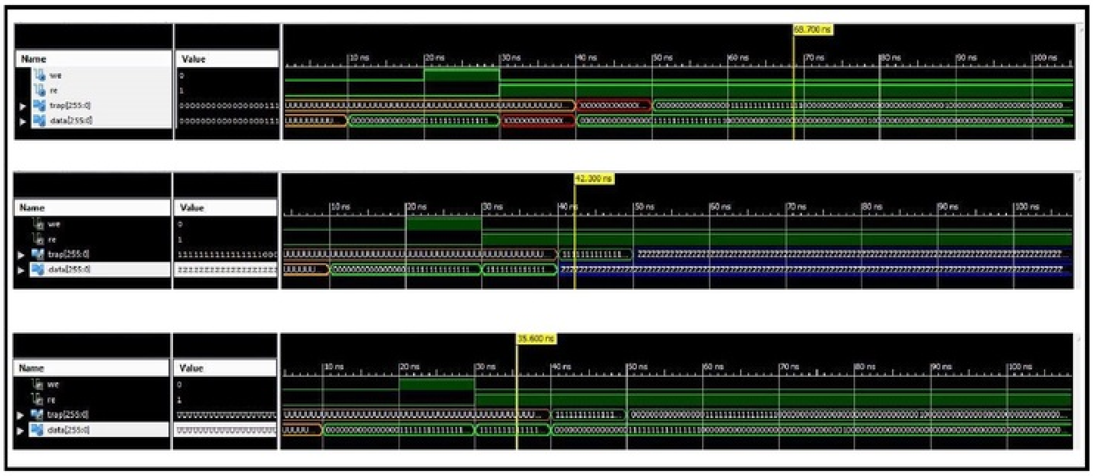
```
wait for 30 ns;
--Check Return 1 (Second Return)
A <= X"FFFF_FFFF_0000_0001_0000_0000_0000_0000"
& X"0000_0000_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF";
```

```
B <= X"0000_0000_0000_0000_0000_0000_0000_0000"
& X"0000_0000_0000_0000_0000_0000_0000_0000";
wait for 30 ns;
--Check Return 1 (Third Return)
A <= X"FFFF_FFFF_0000_0001_0000_0000_0000_0000"
& X"0000_0000_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF";
B <= X"0000_0000_FFFF_FFFE_FFFF_FFFF_FFFF_FFFF"
& X"FFFF_FFFF_0000_0000_0000_0000_0000_0001";
```

6.3 Ram Utility Cell

In simulating the behaviour of the RAM cell, several tests were undertaken to get the timing correct to enforce that the cell would internally latch an impedence drive on its output line. Here we show the progression from double driving of the data bus, to permanently locking the impedance drive latch, to finally working.

Figure 6.1: Demonstration of the progression from X to Z to {0,1}



6.4 Modular Addition

Modular Addition was tested over several bit sizes, always including checks that the addition of zero and anything would be the modulo class of the non zero item, that addition of zero and zero is zero, that addition of zero and the modulus is zero, that addition of two non-zero items whose sum is the modulus results in zero, and that the result of adding anything above this results in the correct modulo class. The modular addition circuit was then run through several synthesis tests to determine the appropriate size of splitting lengths for each bit length of SEC supported primes. Presented here are the synthesis results in table 6.1. The 'Time' is indicative of the shortest combinatorial path length. Similarly for MODSUBT.

6.5 Modular Subtraction

Modular Subtraction was also tested extensively on small prime fields, before being run under similar synthesis tests, however less than that conducted on the MODADDR circuit, for optimised bit lengths. The results of the synthesis tests are presented here, in table 6.2

6.6 Modular Multiplication

Similar to the above, the MODMULT (tabel 6.5) cell was also run through synthesis tests, however, they were first undertaken on the MULTCOMB (tabel 6.3) and DIVDCOMB (tabel 6.4) cells. The 'Time' is indicative of the minimum period length.

6.7 FAP: Modular Inversion

The modular inversion cell was simulated on a small curve, on the curve presented in the worked example in Chapter 2. Here, we present the wave configuration for both, that demonstrates their operation, the outer cell scanned over the inputs, and the inner cell for one cycle of operation.

6.7.1 Outer Cell: Binary Extended Euclidean Algorithm

We see the wave profile for all inputs that are in the range of the modulus (figure 6.2);

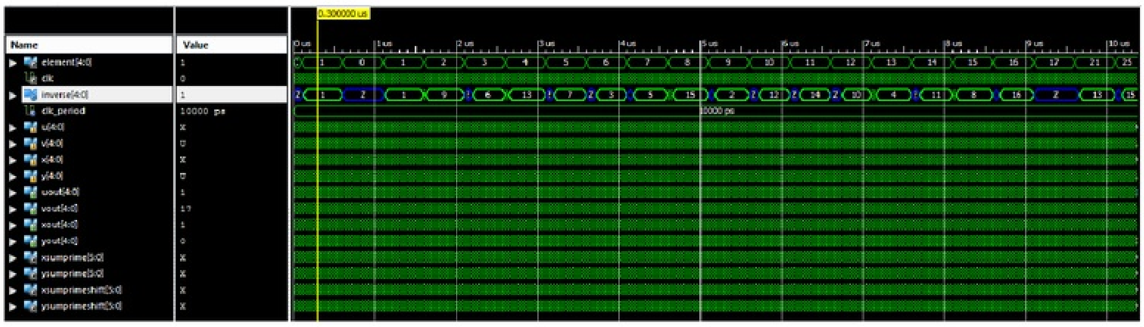
6.7.2 Inner Cell: One round of the BEEA

Here we see the register trade over during the inner cell calculating the inverse(figure 6.3);

6.8 Elliptic Curve Group Operation

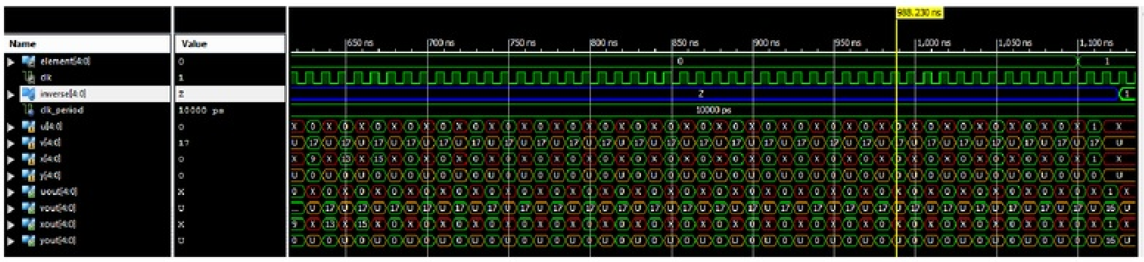
The completed JPA and JPD circuits were tested on the same small curve, and the results agreed with calculated values. Presented here are simulation wave profiles of their operation over sample inputs in the expected range.

Figure 6.2: Demonstration of the BEEA operating over Mod 17



Name	Value
element[4:0]	1
clk	0
inverse[4:0]	1
clk_period	10000 ps
u[4:0]	X
v[4:0]	U
x[4:0]	X
y[4:0]	U
uout[4:0]	1
vout[4:0]	17
xout[4:0]	1
yout[4:0]	0
xsumprime[5:0]	X
ysumprime[5:0]	X
xsumprimeshift[5:0]	X
ysumprimeshift[5:0]	X

Figure 6.3: Demonstration of the BEEA Inner cell alternating its registers



Name	Value
element[4:0]	0
clk	1
inverse[4:0]	Z
clk_period	10000 ps
u[4:0]	0
v[4:0]	17
x[4:0]	0
y[4:0]	0
uout[4:0]	X
vout[4:0]	U
xout[4:0]	X
yout[4:0]	U

6.8.1 ECC: Jacobian Point Addition

See figure 6.4.

6.8.2 ECC: Jacobian Point Doubling

See figure 6.5.

6.9 Elliptic Curve Group Action: JPM

The simulation of the Point Multiplication in the Jacobian Context was similarly run over the M17 curve presented here, for select inputs, and the simulation was as presented here in figure 6.6.

Figure 6.4: Demonstration of the JPA on the M17 curve

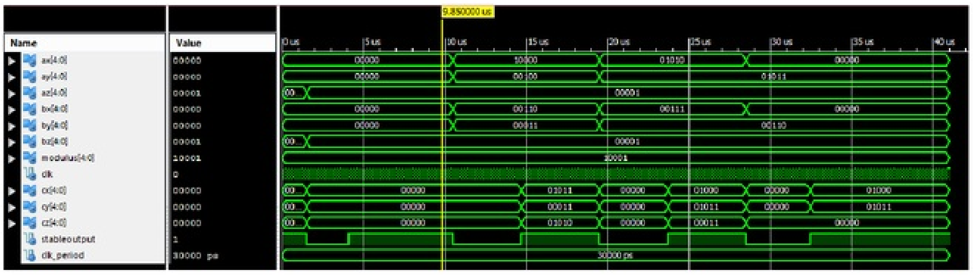
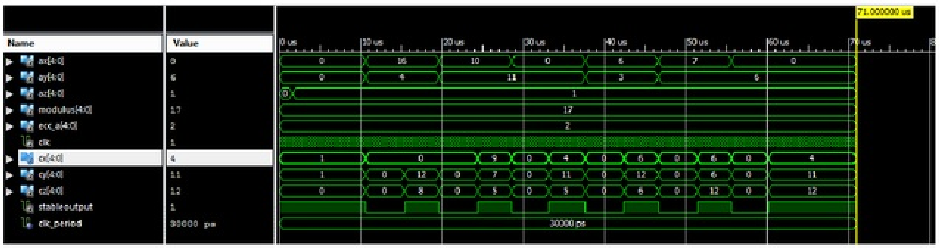


Figure 6.5: Demonstration of the JPD on the M17 curve



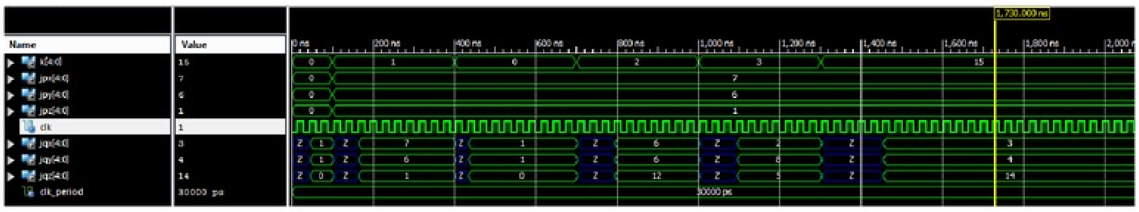
6.10 Elliptic Curve Diffie Hellman

The ECDH cell, utilising an Affine Point Multiplier, was put through a scanning test evaluation of all possible inputs over the M17 curve presented here, and all possible configurations resulted in the same shared key. The test was set up with two ECDH cells and two APM cells, with their Public and Private keys shared explicitly and the test ran over all possible Private Key values for the two ECDH circuits, and each iteration produced a shared key collision. Presented here is the code used to connect the circuits;

```
-- Instantiate the Unit Under Test (UUT)
uut: GENERIC_ECC_ECDH_CLOCKED_PARAMETERISED PORT MAP (
                 KEY_PRIVATE => KEY_PRIVATE_U,
                 KEY_PUBLIC => KEY_PUBLIC_U,
                 SHARED_SECRET => SHARED_SECRET_U,
                 CLK => CLK,
                 StableOutput => StableOutput_U
          );

-- Instantiate the Unit Under Test (UUT)
vut: GENERIC_ECC_ECDH_CLOCKED_PARAMETERISED PORT MAP (
                 KEY_PRIVATE => KEY_PRIVATE_V,
                 KEY_PUBLIC => KEY_PUBLIC_V,
```

Figure 6.6: Demonstration of the JPM on the M17 curve



```
                 SHARED_SECRET => SHARED_SECRET_V,
                 CLK => CLK,
                 StableOutput => StableOutput_V
          );

at : GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED_PARAMETERISED
 Port Map ( KEY => A_PRIVATE,
                  AQX => A_PUBLIC(((2*VecLen)-1) downto VecLen),
                  AQY => A_PUBLIC((VecLen-1) downto 0),
                  CLK => CLK,
                  StableOutput => A_STABLE);

bt : GENERIC_ECC_AFFINE_POINT_MULTIPLY_CLOCKED_PARAMETERISED
 Port Map ( KEY => B_PRIVATE,
                  AQX => B_PUBLIC(((2*VecLen)-1) downto VecLen),
                  AQY => B_PUBLIC((VecLen-1) downto 0),
                  CLK => CLK,
                  StableOutput => B_STABLE);

KEY_PUBLIC_U <= A_PUBLIC;
KEY_PRIVATE_U <= B_PRIVATE;
KEY_PUBLIC_V <= B_PUBLIC;
KEY_PRIVATE_V <= A_PRIVATE;

eq : GENERIC_FAP_RELATIONAL
 Generic Map (N => (2*VecLen),
        VType => 0)
 Port Map ( A => SHARED_SECRET_U,
                  B => SHARED_SECRET_V,
                  E => SecretsEqual,
                  G => open);
```

The test then used an automatic counting technique to self evaluate and target collision over the simulation;

```
CLK_process :process
begin
        CLK <= '0';
        wait for CLK_period/2;
        CLK <= '1';
        wait for CLK_period/2;
        SecretsSame <= (SecretsEqual and StableOutput_V and StableOutput_U and A_
end process;

count_proc: process(SecretsSame)
begin
        if (rising_edge(SecretsSame)) then
           SecretsThatMatch <= SecretsThatMatch + 1;
        end if;
end process;
```

6.11 Elliptic Curve Cryptographic Processor

The final design of the ECCP was tested by periodically writing a Private Key in, using the Command prompt to recalculating a new Public Key, sign some stored Hash value with some stored pseudo random multiplying value, then read out the Public Key of the user to some other register, then write that value back in to the register of the Public Key of some other individual, and then run the SVA, to check that the signature was produced correctly, and check that it does not validate some other Public Key. This was performed on several different configurations. Presented here is a sample of the SGA and SVA operating on M17. Seen in figure 6.7.

Table 6.1: MODADDR Synthesis results

MODADDR			
VecLen	MultLen	Time (ns)	Slice LUT
4	4	4.436	13
5	5	4.249	19
112	7	17.892	1048
112	14	20.421	936
112	28	24.006	887
112	56	34.021	836
112	112	58.214	735
128	4	19.607	1259
128	8	21.825	1225
128	16	21.62	1087
128	32	26.028	1041
128	64	38.309	977
128	128	65.908	849
160	5	21.155	1570
160	10	24.153	1522
160	20	24.961	1351
160	40	31.339	1299
160	80	47.081	1218
160	160	82.329	1060
192	6	22.622	2303
192	12	26.613	1830
192	24	28.281	1633
192	48	36.53	1569
192	96	55.883	1466
192	192	98.866	1278
224	7	22.45	2741
224	14	27.227	2542
224	28	29.474	1936
224	56	39.16	1865
224	112	62.497	1748
224	224	110.848	1531
256	4	25.019	3307
256	8	27.086	3257
256	16	29.212	2860
256	32	32.262	2164
256	64	44.508	2077
256	128	70.939	1951
256	256	128.783	1709
384	6	27.039	6449
384	12	32.497	4914
384	24	36.67	4321
384	48	42.97	3303
384	96	62.543	3180
384	192	104.372	2976
384	384	191.549	2606
521	521	266.367	3451

Table 6.2: MODSUBT Synthesis results

MODSUBT			
VecLen	MultLen	Time (ns)	Slice LUT
4	4	3.373	17
5	5	3.581	22
112	7	13.398	1155
112	28	16.414	941
112	112	32.116	708
128	4	14.242	1381
128	32	18.278	1090
128	128	36.498	822
160	5	15.001	1791
160	20	16.592	1435
160	160	44.766	1031
192	6	16.029	2309
192	24	19.127	1713
192	192	53.702	1225
224	7	17.836	2721
224	28	21.36	2009
224	224	62.364	1437
256	4	19.826	3231
256	32	23.716	2275
256	256	71.155	1618
384	6	23.194	6451
384	48	31.331	3484
384	384	104.517	2491
521	521	143.54	3209

Table 6.3: MULTCOMB Synthesis tests

MULT COMB					
VecLen	MultLen	Period (ns)	Slice LUT	Slice REG	Minimum Arrival (ns)
4	4	2.05	44	26	1.949
5	5	2.009	79	32	1.909
112	7	4.092	1371	674	3.932
112	28	4.373	1346	674	4.062
112	112	3.974	1326	675	3.855
128	4	4.108	1474	771	4.002
128	32	4.234	1427	774	4.023
128	128	3.997	1599	771	3.914
160	5	4.361	1950	964	4.255
160	20	4.196	1895	963	4.086
160	160	4.594	1750	963	4.511
192	6	4.582	2533	1155	4.461
192	24	4.499	2422	1155	4.386
192	192	4.633	2187	1155	4.601
224	7	4.544	2915	1346	4.361
224	28	4.995	2900	1347	4.869
224	224	4.506	2642	1351	4.427
256	4	4.656	3378	1540	4.544
256	32	4.629	3159	1538	4.488
256	256	4.618	3082	1539	4.545
384	6	5.148	5572	2311	4.927
384	48	7.473	4600	2309	7.331
384	384	4.971	4538	2315	4.867
521	521	5.491	6168	3128	5.277

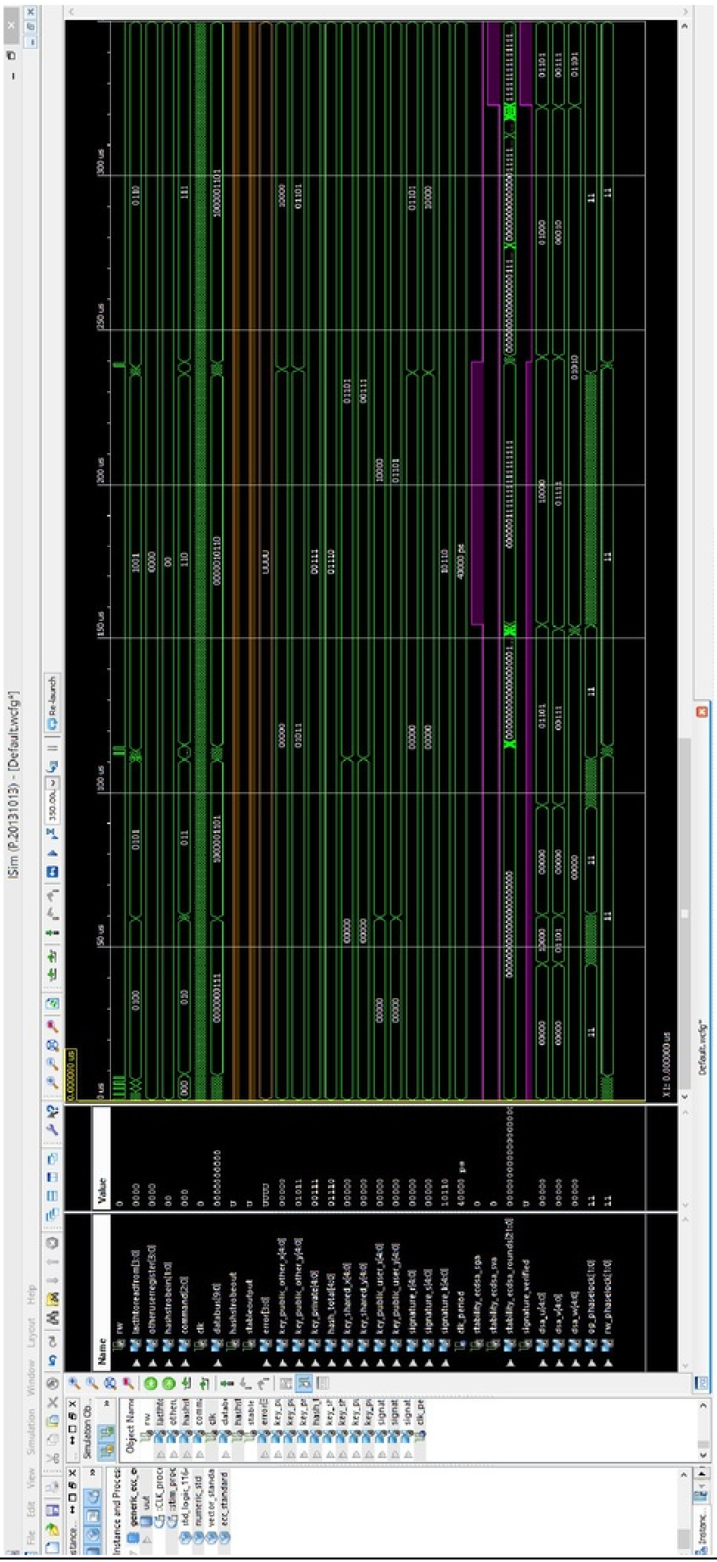
Table 6.4: DIVDCOMB Synthesis Tests

DIVD COMB					
VecLen	MultLen	Period (ns)	Slice LUT	Slice REG	Minimum Arrival (ns)
4	4	2.618	49	31	3.084
5	5	2.709	88	38	3.208
112	7	8.09	2931	817	13.052
112	28	7.813	2592	810	14.394
112	112	7.517	2491	787	20.898
128	4	7.927	3539	899	13.679
128	32	8.172	2897	916	16.839
128	128	8.493	2838	899	23.666
160	5	8.469	4514	1123	14.693
160	20	8.413	3837	1132	14.326
160	160	8.281	3620	1125	28.504
192	6	8.79	5294	1347	14.994
192	24	8.771	4682	1347	15.061
192	192	8.567	4562	1358	34.196
224	7	9.197	6293	1571	15.897
224	28	9.179	5298	1571	16.964
224	224	9.287	5001	1572	39.426
256	4	9.202	7701	1795	16.302
256	32	8.89	6853	1795	20.064
256	256	10.089	5939	1800	43.561
384	6	9.815	13762	2691	16.912
384	48	9.538	10404	2691	25.95
384	384	9.705	8957	2710	63.137
521	521	9.999	13181	3807	85.822

Table 6.5: MODMULT Synthesis Tests

MODMULT					
VecLen	MultLen	Time (ns)	Slice LUT	Slice REG	Minimum Arrival (ns)
4	4	2.624	102	59	3.084
5	5	2.857	135	73	3.375
112	7	8.167	3945	1484	12.736
112	28	8.009	3539	1472	14.679
112	112	7.517	3456	1465	20.898
128	4	7.927	4724	1671	13.679
128	32	8.286	4143	1689	18.37
128	128	8.493	3983	1671	23.666
160	5	8.469	6104	2087	14.693
160	20	8.413	5369	2096	15.448
160	160	8.274	5075	2090	28.503
192	6	8.79	7246	2503	14.999
192	24	8.929	6394	2503	17.52
192	192	9.054	6293	2509	33.202
224	7	9.197	8572	2919	15.897
224	28	9.045	7447	2919	19.743
224	224	15.006	6973	2923	39.575
256	4	9.142	11586	3335	15.35
256	32	9.011	9735	3335	20.137
256	256	9.916	8244	3345	43.578
384	6	9.815	18328	4999	16.912
384	48	9.533	14892	4999	25.926
384	384	11.387	13036	5002	63.186
521	521	10.267	17620	6882	85.685

Figure 6.7: Demonstration of the ECCP on the M17 curve


ISim (P.20131013) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
Re-launch
Default.wcfg*
X1: 0.000000 us

Name	Value
rw	0
lacthtoreadfrom[3:0]	0000
otheruserregister[3:0]	0000
hashstrobein[1:0]	00
command[2:0]	000
clk	0
databus[9:0]	0000000000
hashstrobeout	U
stableoutput	U
error[3:0]	UUUU
key_public_other_x[4:0]	00000
key_public_other_y[4:0]	01011
key_private[4:0]	00111
hash_total[4:0]	01110
key_shared_x[4:0]	00000
key_shared_y[4:0]	00000
key_public_user_x[4:0]	00000
key_public_user_y[4:0]	00000
signature_r[4:0]	00000
signature_s[4:0]	00000
signature_k[4:0]	10110
clk_period	40000 ps
stability_ecdsa_sga	0
stability_ecdsa_sva	0
stability_ecdsa_rounds[21:0]	0000000000000000000000
signature_verified	U
dsa_u[4:0]	00000
dsa_v[4:0]	00000
dsa_w[4:0]	00000
op_phaselock[1:0]	11
rw_phaselock[1:0]	11

Chapter 7

Future Work

7.1 Chapter Introduction

This chapter is devoted to discussing the ways in which the work carried out in this project could be extended or continued. It also serves as the conclusion to the process undertaken.

7.2 In Review of the Project

A key issue in the delivery of this project was the lack of availability and difficulty in obtaining anything other than a free Xilinx License key, which severely limited the amount of simulation elaboration that could be undertaken. All the testing and evaluation for this circuit was carried out on a small Elliptic Curve. Despite, however, that it was not possible to obtain longer simulations for larger bit sizes over such a large process (license key timed lock out) it was possible to simulate it on small curves, and still obtain synthesis information about larger sizes, although they were only estimated values, as the free license key does not include the capacity to place and route on most boards. Hence, running longer simulations on multiple curves would be a top priority in expanding the content of this thesis. Also, this thesis was executed with minimal direction from a supervisor, and shifted focus during its execution to more closely align with realistic goals obtainable on a free license key. However, the originally intended goal of producing a circuit capable of handling the NISTsecp256r1 curve can still be satisfied by the design, given a synthesis evaluation of the ECCP with a bit length of 256 bits, over a splitting length of 4 bits, the device utilisation of a Virtex-7 board was estimated to be at 88%, although the splitting length of 4 it by far the least optimal for size, and most optimal for speed. However, sadly, access to ModelSim was not possible, so it would not be feasible to state any well determined timing information about the circuit.

7.3 HASH and CSPRNG

The two components originally intended to be included in the design that were never included, the Hashing processor, utilising the SHA2 algorithm, and some Cryptographically Secure Pseudo Random Number Generator, could be designed and supplemented in, depending on their area impact, however up until and if such an extension were viable, this is worked around through external computation of these, passed into the registers, and then used in the computation. Despite that the interfacing with an external component would make the device slow, the intention of the design is that it can be used prior to synthesis to determine and match a desired trade off between speed and area, with the design gaining more noticeable speed benefits the longer the bit length applied to it.

7.4 Elliptic Curve HASH and CSPRNG?

Another thing worth considering is the potential to utilise multiple smaller curves (given the design allows for curve interchangeability post synthesis) using only a handful of actual circuits, that comb over some data and design a cryptographically secure hashing function using Elliptic Curves. Alternatively, using them on some seed value to design an ECCSPRNG?

Chapter 8

Abbreviations

APA	Affine Point Addition
APD	Affine Point Doubling
APM	Affine Point Multiplication
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECCP	Elliptic Curve Cryptographic Processor
ECDH	Elliptic Curve Diffie Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECG	Elliptic Curve Group
ECGA	Elliptic Curve Group Action
ECGO	Elliptic Curve Group Operator
DHKE	Diffie Hellman Key Exchange
DSA	Digital Signature Algorithm
GF	Galois Field
HDL	Hardware Description Language
JPA	Jacobian Point Addition
JPD	Jacobian Point Doubling
JPM	Jacobian Point Multiplication
PA	Point Addition
PD	Point Doubling
PM	Point Multiplication
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Bibliography

[1] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley Publishing, 2008, 9780470068526.

[2] M. Askar and T. S. Celebi, "Design and fpga implementation of hash processor," uluslararasi Katilimli Bilgi Guvenligi Ve Kriptoloji Konferansi.

[3] J. Bauer. (2015) Elliptic curve cryptography tutorial. [Online]. Available: http://www.johannes-bauer.com/compsci/ecc/

[4] T. S. Celebi, "Design and fpga implementation of hash processor," Master's thesis, GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF MIDDLE EAST TECHNICAL UNIVERSITY, 2007.

[5] P. Choi, J. T. Kong, and D. K. Kim, "Analysis of hardware modular inversion modules for elliptic curve cryptography," department of Electronic Engineering Hanyang University, Seoul, Korea.

[6] W. Commons. (2017, March) Elliptic curve. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic_curve

[7] A. Corbellini. (2015, May) Elliptic curve cryptography: Ecdh and ecdsa. [Online]. Available: http://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

[8] N. Ghanmy, L. Chaari-Fourati, and L. Kamoun, "Enhancement security level and hardware implementation of ecdsa," electronic and Information Technology Laboratory (LETI), SFAX University, Tunisia.

[9] N. Ghanmy, N. Khlif, L. Fourati, and L. Kamoun, "Hardware implementation of elliptic curve digital signature algorithm on koblitz curves," in *IET International Symposium on Communication Systems*, ser. Networks and Digital Signal Processing, no. 8th. IEEE.

[10] A. A. A. Gutub, "New hardware algorithms and designs for montgomery modular inverse computation in galois fields gf(p) and gf(2)," Ph.D. dissertation, Oregon State University, 2002.

[11] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[12] M. S. Hossain and Y. Kong, "High-performance fpga implementation of modular inversion over f256 for elliptic curve cryptography," in *International Conference on Data Science and Data Intensive Systems.* IEEE, 2015.

[13] K. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, 2nd ed. Springer-Verlag, 1990.

[14] K. Jarvinen, "Design and implementation of a sha-1 hash module on fpgas," helsinki University of Technology, Otakaari, Espoo, Finland.

[15] ——, "Final project report: Cryptoprocessor for elliptic curve digital signature algorithm," helsinki University of Technology, Signal Processing Laboratory, Otakaari, Espoo, Finland.

[16] S. Lad, V. Gupta, and A. S. Kalambe, "Fpga implementation of point multiplication on koblitz curves using vhdl," *International Journal of Engineering Research and Technology*, vol. 2, no. 11, pp. 3886 – 3888, November 2013, iSSN: 2278-0181.

[17] G. Lai, "Analysis of modular inverse gf(p) implementations," department of Electrical Engineering and Computer Science, Oregon State University, Corvallis, Oregon, USA.

[18] H. Q. Li and C. Y. Miao, "Hardware implementation of hash function sha-512," school of Information and Communication Engineering, Tianjin Polytechnic University, Tianjin, China.

[19] C. T. Long, *Elementary Introduction to Number Theory*, 2nd ed. Lexington, 1972.

[20] A. J. Menezes, S. A. Vanstone, and P. C. VanOorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, 1996, 0849385237.

[21] G. d. D. Meurice, P. Bulens, and J. J. Quisquater, "Efficient modular division implementation: Ecc over gf(p) affine coordinates application," universite Catholique de Louvain, UCL Crypto Group, Laboratoire de Microelectronique (DICE), Louvain-La-Neuve, Belgium.

[22] NIST, *Secure Hash Standard*, NIST FIPS, 2015.

[23] K. K. Parhi, "Fast vlsi binary addition," dept. of Electrical and Computer Engineering University of Minnesota, Minneapolis, USA.

[24] M. M. Postnikov, *Foundations of Galois Theory.* Dover, 2004.

[25] C. Research, *Standards for Efficient Cryptography*, Certicom Corporation, 2000.

[26] J. Rotman, *Galois Theory*, 2nd ed. Springer-Verlag, 1998.

[27] Z. Shi, C. Ma, J. Cote, and B. Wang, *Introduction to Hardware Security and Trust*. Springer, 2012, ch. Hardware Implementation of Hash Functions.

[28] J. Shokrollahi, "Efficient implementation of elliptic curve cryptography on fpgas," Ph.D. dissertation, Mathematisch-Naturwissenschaftlichen Fakultat der Rheinischen Friedrich-Wilhelms-Universitat, Bonn, 2006.

[29] Xilinx, *Constraints Guide*, Xilinx, 2008.