Design of a Reconfigurable Processor for Elliptic Curve Cryptography over NIST Prime Fields

by

Kendall Ananyi
B.Eng., University of Benin, 2002

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Design of a Reconfigurable Processor for Elliptic Curve Cryptography over NIST Prime

Fields

by

Kendall Ananyi
B.Eng., University of Benin, 2002

Supervisory Committee

Dr. Daler Rakhmatov, Supervisor (Department of Electrical and Computer Engineering)

Dr. Stephen Neville, Departmental Member (Department of Electrical and Computer
Engineering)

Dr. Micaela Serra, Outside Member (Department of Computer Science)

Dr. Kui Wu, External Examiner (Department of Computer Science)

Supervisory Committee

Dr. Daler Rakhmatov, Supervisor (Department of Electrical and Computer Engineering)

Dr. Stephen Neville, Departmental Member (Department of Electrical and Computer Engineering)

Dr. Micaela Serra, Outside Member (Department of Computer Science)

Dr. Kui Wu, External Examiner (Department of Computer Science)

ABSTRACT

Exchange of information must integrate a means of protecting data against unauthorized access. Cryptography plays an important role in achieving information security. It is used for (1) encrypting or signing data at the source before transmission, and then (2) decrypting or validating the signature of the received message at the destination. This thesis focuses on the study of the hardware implementation of a reconfigurable processor supporting elliptic curve cryptography (ECC) over prime fields GF(p). The proposed processor can be reconfigured to work with any of the five prime fields recommended by NIST (192 to 521 bits). Our processor can be programmed to execute any sequence of basic modular operations (add, subtract, multiply, invert) used in higher level ECC arithmetic. The architecture has been prototyped on a Xilinx FPGA. Its performance is competitive with existing hardware implementation, despite the overhead needed to support datapath reconfigurations for different prime sizes.

Table of Contents

List of Tables

List of Figures

Acknowledgment

I would like to thank my supervisor, Dr Daler Rakhmatov, for all his guidance, assistance and time throughout the entire duration of my course.

I am especially indebted to my aunt Julia Shinaba for all the help she gave me during my stay at the University of Victoria and my cousins Seyi and Jide who were very accommodating and understanding throughout the whole period.

I am also grateful to all my friends and especially Febeke Okafor, Maame Duah, Caroline Bagelman, Ayosike Akingbade, Tochukwu Ezebube, Imuetinyan Edigin, Mandela Toyo, Wole Eweje, Ade Adeyemi for all their support throughout my course.

Dedication

This thesis is dedicated to GOD, joined humbly are my parents and my siblings Dunstan, Joy and Nelson who together saw that I made it through graduate school.

1. Introduction

1.1 Elliptic Curve Cryptography

Private information when stored or transmitted from one party to another needs to be secured. This entails restricting its access to authorized users only. Cryptographic algorithms play an important role in achieving information security. There are two families of cryptographic algorithms: (1) private, or symmetric, key cryptographic algorithms and (2) public, or asymmetric, key cryptographic algorithms.

The key used in private key cryptographic algorithms needs to be transferred securely prior to encryption. This posed the risk of interception of the key by an attacker. Public key cryptography was created to solve the problem of key sharing in private key cryptography. Public key cryptography uses a different approach, whereby two keys exist, a private key and a public key (derived from the private key). The public key is made available for users to encrypt information which can only be decrypted by the owner of the private key (which is kept secret). Public key cryptography relies on computational infeasibility of deriving the private key from the public key. Since public key involve more computations than private key cryptographic algorithms, they are mostly used to transmit private keys.

Elliptic curve cryptography (ECC) is a variant of public key cryptography, proposed independently by Kobiltz [1] and Miller [2]. It provides more cryptographic strength per bit of encryption than RSA for the same level of security [7]. ECC has evolved to become widely accepted and standardized by IEEE [3], ANSI [4], NIST [5], ISO [6].

ECC has applications covering various area of security. This section describes some practical situations where ECC can be employed.

- Smart Cards: ECC's high security per bitwidth makes it suitable for securing application in smart cards where memory, bandwidth and computational power

are constrained. They are also used to secure data transmitted between the card and the card reader [8].

- Mobile Devices: ECC is currently used to secure the popular Blackberry. Applications within the mobile device use the provided ECC Application Programming Interface (API) [9] for data encryption/decryption, authentication and certificate management. The ECC API is also used to secure mobile applications before installation on the device and to verify that licenses for each application are obtained [10].

- Online Cheque Clearing: The US Check 21 system [11], which is currently being used by banks for online check clearing, also uses ECC as the cryptographic algorithms to secure transactions [12]. The system eliminates the cost of printing paper cheques as well as transportation costs incurred in exchanging paper cheques. The system uses a digital image of the cheque. The issuing bank which generates the digital check signs the cheque electronically. The signature is verified by the receiving bank to ascertain that it has not been tampered with.

- Key Exchange/Agreement: ECC can be used in protocols such as the Diffie-Hellman private key exchange algorithm [7]. Fast private key algorithms are used for real-time exchange of information as they are less computationally intensive than the public key algorithm. Once the private key is exchanged using ECC secure communication can commence using private key algorithms.

- Web Security: ECC has been used in the implementation of TLS/SSL [7] to secure online transactions as well as Virtual Private Networking tunnels [12].

- Wireless Security: ECC can also be used to replace WEP/WPA security used for user authentication in a wireless network since ECC is much stronger than the current algorithms used in securing WiFi networks [12].

- Voice over Internet Protocol (VOIP): VOIP allows one to make telephone calls using a computer network, by converting the voice to packets which are then

transmitted over the network and reassembled to form sound at the receivers end. ECC can be used to encrypt VOIP packets transmitted over packet switched network. It can also be used in user authentication before communication commences [12].

1.2 Research Contributions

In order to implement ECC an elliptic curve must first be chosen. Various curves have been recommended by IEEE, ANSI and NIST. The curves selected by NIST FIPS 186-2 are a subset of those recommended in ANSI X9.62, and these ANSI curves are a subset of those recommended in IEEE P1363. Thus, ECC implementations using NIST curves conform to all other standards.

This thesis describes a reconfigurable processor for supporting ECC over all five NIST recommended prime fields. The proposed architecture can be configured to perform arithmetic operations over any of the five fields GF(p) with primes p of size 192, 224, 256, 384, and 521 bits. The corresponding modulo p operations (addition, subtraction, multiplication, and inversion, see Chapter 4) form the instruction set of the processor. These instructions can be combined into programs for performing basic ECC operations, such as point addition. Point additions are needed in all methods for point multiplications which is the main ECC operation. In our implementation, selecting different point multiplication algorithm would require only a change in the corresponding sequence of programs for point addition operations. Thus, the flexibility of our ECC processor is characterized by (1) supporting prime fields GF(p) with five different prime sizes, and (2) supporting programmable point addition for implementing any desired point multiplication method. For the sake of efficiency, we have decided to focus only on NIST prime fields, as opposed to other GF(p) alternatives with the same prime size. The main reason behind NIST recommendations is the ability to perform fast modular reductions, and our ECC processor takes full advantage of this. The well-known concepts of concurrency and locality have also been utilized to improve the efficiency of the proposed architecture.

Prime fields GF(p) are not the only choice for ECC. NIST has also recommended several binary fields GF(2^n), whose elements are defined as n-bit binary vectors [5]. There are many hardware-based implementations of ECC over GF(2^n), including [13] that can be reconfigured for different values of n. On the other hand to our knowledge, none of the existing hardware solutions for ECC over GF(p) can handle different sizes of prime p, ranging from 192 to 521 bits. Our proposed ECC processor fills this flexibility gap.

1.3 Thesis Outline

The thesis is divided into seven chapters. In Chapter 2 we introduce the mathematical background of the elliptic curve cryptographic algorithms, the various point representations, point multiplication algorithms used in ECC and briefly discuss ECC. In Chapter 3 we describe the NIST elliptic curve over prime fields and provide specific information on implementing ECC over NIST prime fields. In Chapter 4 we outline previous work and describe the architecture of our proposed ECC processor. In Chapter 5 we present results of the hardware implementation on a Xilinx Virtex-II Pro FPGA. Finally, in Chapter 6 we present the conclusion drawn from our work and summarize possible future research work.

2. Mathematical Background

This chapter provides a broad overview of the ECC mathematics. We explain the fundamental concepts behind elliptic curves over prime fields and describe algorithms for performing operations on elliptic curve points.

2.1 Groups, Rings, Fields

A *group* (G,+) is set G over which the binary operation + on members of set G satisfies the following axioms [14]:

1. Associativity: $a + (b + c) = a + (b + c)$, for all a, b, c $\in$ G.

2. Identity: There exists an element I^+ (identity element) such that $a + I^+ = I^+ + a = a$, for all a $\in$ G.

3. Inverse: For every a $\in$ G there exists an element a^{-1} (inverse element) such that a $+ a^{-1} = a^{-1} + a = I^+$.

If the group is commutative, i.e., $a + b = b + a$, for all a, b $\in$ G, it is referred to as an *Abelian group*.

A *ring* (R, +, $*$) is made up of a set R along with two binary operations $*$ (multiplication) and + (addition) that satisfy the following axioms [14]:

1. Identity: (R , +) is an Abelian group with identity I^+.

2. Multiplicative Identity: There exists element $I^* \neq I^+$ such that $I^* * a = a * I^* = a$, for every a $\in$ R.

3. Multiplicative Associativity: $a * (b * c) = a * (b * c)$, for all a, b, c $\in$ R.

4. Distributivity: a $*$ (b + c) = (a $*$ c) + (b $*$ c) and (b + c) $*$ a = (b $*$ a) + (c $*$ a).

The ring is *commutative* if a $*$ b = b $*$ a, for all a, b $\in$ R. A *field* is a commutative ring whose non-zero elements satisfy axiom 3 over the $*$ operation.

A *finite field* F is a field with a finite number of elements. The number of elements in a finite field is referred to as its order. For every prime power p^m, where p is prime and m is an integer $\geq$ 1, there is a unique finite field of order p^m denoted by GF(p^m). This thesis deals with finite fields GF(p), i.e., m = 1.

2.2 Prime Field Arithmetic

2.2.1 Prime Fields

The *prime field* GF(p) is the finite field whose elements are all the integers between 0 and p − 1 inclusive. For every a $\in$ GF(p) the following operations are defined:

1. Addition: If a, b $\in$ GF(p) then a + b = r, where r $\in$ GF(p) and equals (a + b) modulo p.

2. Multiplication: If a, b $\in$ GF(p) then a $*$ b = r, where r $\in$ GF(p) and equals (a $*$ b) modulo p.

3. Inversion: If a is a non-zero element in GF(p), then a^{-1} = c $\in$ GF(p) such that a $*$ c = 1 modulo p.

2.2.2 Modular Operations over Prime Fields

In this section the modular addition, multiplication and inversion over prime fields GF(p) are illustrated with an example. Consider prime field GF(11), whose elements are {0, 1, 2, ..., 10}. The modular arithmetic examples are as follows:

- Addition: $3 + 9 = 12 \equiv 1 \bmod 11$

- Subtraction: $3 - 9 = (3 + 11) - 9 \equiv 5 \bmod 11$

- Multiplication: $3 * 9 = 27 \equiv 5 \bmod 11$

- Squaring: $5^2 = 25 \equiv 3 \bmod 11$

- Inversion: $3^{-1} = (1+11)/3 \equiv 4 \bmod 11$

2.3 Elliptic Curves

2.3.1 Geometric Interpretation

Elliptic curve E over a field F is given by [15]:

$$E: Y^2 = X^3 + aX + b \qquad (2.1)$$

Points whose coordinates *(x, y)* satisfy equation 2.1, along with the point of infinity P^∞ (which is at the top of the y-axis), make up the set of rational points on curve E [7]. The negative of a point P is its reflection in the x-axis, i.e., *–P = (x, –y)* [16].

2.3.2 Elliptic Curves over GF(p)

For elliptic curves over prime fields GF(p) with p > 3, the parameters *a* and *b* of equation 2.1 should satisfy $4a^3 + 27b^2 \neq 0$ *(mod p)* [17]. The condition is required to ensure that the curve is smooth, and there are no points at which the curve has two or more distinct tangent lines [17]. The allowable point coordinates become integer modulo p.

The standard representation of points on an elliptic curve E over prime fields is the Affine coordinates. Adding two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ in Affine coordinates yields point $P_3 = (x_3, y_3)$, where:

$$x_3 = \mu^2 - x_1 - x_2 \text{ and } y_3 = \mu(x_1 - x_3) - y_1,$$

if $P_1 \neq P_2$, $\mu = \dfrac{y_2 - y_1}{x_2 - x_1}$; otherwise, $\mu = \dfrac{3x_1^2 - 3}{2y_1}$. \hspace{1cm} (2.2)

For example [19], let the elliptic curve over GF(23) be $E: y^2 = x^3 + x + 4$.

- First, we check $4a^3 + 27b^2 = 4 + 432 = 436 \equiv 22 \bmod 23 \neq 0$.

- To determine if point $P_1 = (4, 7)$ is on the curve E, we compute $x^3 + x + 4 = 4^3 + 4 + 4 = 72 \equiv 3 \bmod 23$, and $y^2 = 7^2 = 49 \equiv 3 \bmod 23$.

- To determine if point $P_2 = (13, 11)$ is on the curve E, we compute $x^3 + x + 4 = 13^3 + 13 + 4 = 72 \equiv 6 \bmod 23$, and $y^2 = 11^2 = 121 \equiv 6 \bmod 23$.

- To obtain point $P_3 = (x_3, y_3) = P_1 + P_2$, we compute $x_3 = \left(\dfrac{11-7}{13-4}\right)^2 - 4 - 13 = 3^2 - 4 - 13 \equiv 15 \bmod 23$, and $y_3 = 3(4 - 15) - 7 \equiv 6 \bmod 23$.

2.4 Point Representation in GF(p)

2.4.1 Affine Coordinates

When two points are added or doubled in Affine coordinates [15], we need to perform modular inversion to compute μ, e.g., $(x_2 - x_1)^{-1}$ or $(2y_1)^{-1}$, in equation 2.2. Modular inversion is a computationally intensive operation, and it is required each time a point is added or doubled. There are alternative point representations that favour modular multiplications over inversions. They are useful when the ratio $\dfrac{T_I}{T_M}$ is high, where T_I is

the time it takes to perform inversion, and T_M is the time it takes to perform multiplication. Modular inversion is required only when the point is converted back to Affine representation. Examples of such point representations are presented in the following sections.

2.4.2 Standard Projective Coordinates

The projective equation of the elliptic curve is $Y^2Z = X^3 + aXZ^2 + bZ^3$ [17]. The Projective coordinate point *(X, Y, Z)* is equivalent to the Affine point $(\frac{X}{Z}, \frac{Y}{Z})$, while the negative of the point and the point at infinity are *(X, −Y, Z)* and *(0, 1, 0)* respectively.

To add the points *(X₁, Y₁, Z₁)* and *(X₂, Y₂, Z₂)* to yield point *(X₃, Y₃, Z₃)*, we compute

$$A = X_1Z_2^2 \qquad\qquad B = X_2Z_1^2$$

$$C = A - B \qquad\qquad D = Y_1Z_2^3$$

$$E = Y_2Z_1^3 \qquad\qquad F = D - E$$

$$G = A + B \qquad\qquad H = D + E$$

$$X_3 = F^2 - GC^2 \qquad\qquad I = FC^2 - 2X_3$$

$$Y_3 = (IF - HC^3)/2 \qquad\qquad Z_3 = Z_1Z_2C.$$

To double point *(X₁, Y₁, Z₁)* to yield point *(X₃, Y₃, Z₃)*, we compute:

$$A = 3X_1^2 - 3Z_1^4 \qquad\qquad B = 4X_1Y_1^2$$

$$C = 8Y_1^4 \qquad\qquad X_3 = A^2 - 2B$$

$$Y_3 = A(B - X_3) - C \qquad\qquad Z_3 = 2Y_1Z_1.$$

2.4.3 Jacobian Projective Coordinates

The Jacobian Projective equation of the elliptic curve is the curve $Y^2 = X^3 + aXZ^4 + bZ^6$

[17]. The point (X, Y, Z) is the Affine point $(\dfrac{X}{Z^2}, \dfrac{Y}{Z^3})$, while the negative point and point

at infinity are *(X, –Y, Z)* and *(1, 1, 0)* respectively.

To add the points *(X₁, Y₁, Z₁)* and *(X₂, Y₂, Z₂)* to yield point *(X₃, Y₃, Z₃)*, we compute:

$$U_1 = X_1Z_2^2 \qquad\qquad\qquad U_2 = X_2Z_1^2$$

$$S_1 = Y_1Z_2^3 \qquad\qquad\qquad S_2 = Y_2Z_1^3$$

$$H = U_2 - U_1 \qquad\qquad\qquad R = S_2 - S_1$$

$$X_3 = R^2 - H^3 - 2U_1H^2 \qquad\qquad Y_3 = -S_1H^3 + R(U_1H^2 - X_3)$$

$$Z_3 = Z_1Z_2H.$$

In order to speedup point addition the second point *(X₂, Y₂, Z₂)* is left in Affine coordinates *(X₂, Y₂, 1)*, i.e., *Z₂ = 1*, such point addition is referred to as mixed-Jacobian point addition [17].

To double the point *(X₁, Y₁, Z₁)* to yield point *(X₃, Y₃, Z₃)*, we compute:

$$A = 3X_1^2 - 3Z_1^4 \qquad\qquad\qquad B = 4X_1Y_1^2$$

$$C = 8Y_1^4 \qquad\qquad\qquad\qquad X_3 = A^2 - 2B$$

$$Y_3 = A(B - X_3) - C \qquad\qquad\qquad Z_3 = 2Y_1Z_1.$$

2.4.4 Chudnovsky Jacobian Projective Coordinates

Point addition in Jacobian coordinates is slower than in standard Projective coordinates. To speed up point additions, the Chudnovsky Jacobian Projective representation can be used. Points are represented by a 5-tuple (X, Y, Z, Z^2, Z^3) [18].

To add the points $(X_1, Y_1, Z_1, Z_1^2, Z_1^3)$ and $(X_2, Y_2, Z_2, Z_2^2, Z_2^3)$ to yield point $(X_3, Y_3, Z_3, Z_3^2, Z_3^3)$, we compute:

$$U_1 = X_1 Z_2^2 \qquad\qquad\qquad U_2 = X_2 Z_1^2$$

$$S_1 = Y_1 Z_2^3 \qquad\qquad\qquad S_2 = Y_2 Z_1^3$$

$$H = U_2 - U_1 \qquad\qquad\qquad R = S_2 - S_1$$

$$X_3 = R^2 - H^3 - 2U_1 H^2 \qquad\qquad Y_3 = -S_1 H^3 + R(U_1 H^2 - X_3)$$

$$Z_3 = Z_1 Z_2 H.$$

To double the point $(X_1, Y_1, Z_1, Z_1^2, Z_1^3)$ to yield the point $(X_3, Y_3, Z_3, Z_3^2, Z_3^3)$, we compute:

$$S = 4 X_1 Y_1^2 \qquad\qquad\qquad M = 3 X_1^2 - 3 Z_1^4$$

$$T = M^2 - 2S \qquad\qquad\qquad X_3 = T$$

$$Y_3 = -Y_1^4 + M(S - T) \qquad\qquad Z_3 = 2 Y_1 Z_1.$$

2.4.5 Comparison of Various Point Representations

Table 1 gives the number of modular operations required for each point representation from slowest to fastest. *I* represents the number of modular inversions and *M* represents the number of modular multiplications performed in each of the point operations

(addition and doubling). The cost of modular addition/subtraction is ignored, as it is small compared to other GF(p) modular operations.

Table 1: Comparison of number of modular operations required for each point representation.

Point Doubling		Point Addition	
Affine	*I, 4M*	Affine	*I, 3M*
Projective	*10M*	Jacobian	*16M*
Chudnovsky	*9M*	Chudnovsky	*14M*
Jacobian	*8M*	Projective	*14M*
		Mixed Jacobian	*11M*

Note that the Jacobian doubling algorithm is the fastest point doubling algorithm of all point representations, as it requires only 8 modular multiplications. The mixed Jacobian representation yields the fastest point addition algorithm of all representation, as it requires only 11 point multiplications.

2.5 Point Multiplication over GF(p)

Point multiplication, or scalar multiplication, is higher in the hierarchy of ECC operations than the point addition and doubling. It is computed by performing a series of point additions and point doublings. A point P is multiplied by a scalar k to yield the point kP. For example to compute *5P* one would double *P* twice to obtain *4P* then add *P* to the intermediate result to obtain *5P*.

Various algorithms exist for point multiplication. When the point P is unknown the *unknown point multiplication* is used, whereas when the point is fixed, the *known point multiplication* is used. The known point multiplication is faster than the unknown point multiplication because certain fixed parameters can be precomputed. In the rest of this section we describe the various point multiplication algorithms.

2.5.1 Binary Point Multiplication Algorithm

Algorithm 1 and **Algorithm 2** are the basic point multiplication algorithms and are analogous to the square and multiply algorithm for exponentiation [14]. They differ in the way the bits of the scalar k are processed. Algorithm 1 processes the bits from the least significant bit and then moves to most significant bit, while **Algorithm 2** reverses the order and starts from the most significant bit and advances to the least significant bit.

Algorithm 1: Right to left binary method for unknown point multiplication [17].

INPUT: $k = (k_{n-1},.....,k_1, k_0)$, $P \in GF(p)$

OUTPUT: kP.

1. $Q \leftarrow P^{\infty}$.

2. For i from 0 to $n - 1$ do

    2.1 If $k_i = 1$ then $Q \leftarrow Q + P$.

    2.2 $P \leftarrow 2P$.

3. Return(Q).

Algorithm 2: Left to right binary method for unknown point multiplication [17].

INPUT: $k = (k_{n-1}, ..., k_1, k_0)$, $P \in GF(p)$

OUTPUT: kP.

1. $Q \leftarrow P^{\infty}$.

2. For i from n --1 to 0 do

    2.1. $Q \leftarrow 2Q$.

    2.2. If $k_i = 1$ then $Q \leftarrow Q + P$.

3. Return(Q).

In order to estimate the time it takes to perform unknown point multiplication, it is assumed that the average number of ones in n-bit scalar k is $n/2$. There are approximately $n/2$ point additions and n point doublings. Therefore the running time of both algorithms is $nA/2 + nD$, where A is the time required to perform point addition and D is the time required to perform point doubling.

2.5.2 Non Adjacent Form (NAF) Point Multiplication

In order to speedup the running time of the unknown point multiplication algorithms two approaches can be taken:

1. Use of a different representation of scalar k with a smaller number of non-zero bits in its representation thus resulting in fewer point doubling operations.

2. Processing multiple bits of k at a time.

NAF point multiplication algorithm takes advantage of the fact that the time to compute

point addition and subtraction are the same. The bits of the NAF representation of k can

be 0, +1 or –1. These bits can be used to determine whether point addition or subtraction

is performed. **Algorithm 3** computes the NAF representation of any positive integer k

while **Algorithm 4** is the binary NAF point multiplication algorithm.

Algorithm 3: NAF computation of an integer [17].

INPUT: k

OUTPUT: NAF(k).

1. i ←0.

*2. **While** $k \geq 1$ **do***

      *2.1. **If** k is odd **then** $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$; **else** $k_i \leftarrow 0$.*

      2.2. $k \leftarrow k/2$, $i \leftarrow i + 1$.

*3. **Return**(k).*

Algorithm 4: Binary NAF multiplication algorithm [17].

INPUT: k = (k_{n-1},.....,k_1, k_0), P$\in$ GF(p).

OUTPUT: kP.

*1.Compute NAF(k) using **Algorithm 3***

2. Q $\leftarrow$ P^{∞}.

*3. **For** i **from** n $-$ 1 **to** 0 **do***

   3.1. Q $\leftarrow$ 2Q.

   *3.2. **If** k_i = 1 **then** Q $\leftarrow$ Q + P; **else if** k_i = -1 **then** Q $\leftarrow$ Q $-$ P.*

*4. **Return**(Q).*

The average number of 1's in the NAF representation is 1/3 and this reduces the number

of point additions required to perform point multiplication to *n/3*. Hence the running time

of **Algorithm 4** is *nA/3 + nD*.

2.5.3 Window Point Multiplication Algorithms

In window methods of point multiplication algorithms multiple bits *w* of scalar *k* are

processed at a time. **Algorithm 5** computes the width-w NAF of a positive integer used in

the window NAF method of point multiplication.

Algorithm 5: Width-w NAF computation of a positive integer [17].

INPUT: positive integer k, window width w

OUTPUT:NAF$_w$(k).

1. i ←0.

*2. **While** k ≥ 1 **do***

    *2.1 **If** k is odd **then** k_i ← k mod 2^w , k ← k - k_i; **else** k_i ← 0.*

    2.2 k ← k /2, i ← i + 1.

*3. **Return**(k).*

Algorithm 6 is the window NAF algorithm for point multiplication. The running time of

the algorithm is approximately $\left[1D + (2^{w-2} - 1)A\right] + \left[\dfrac{m}{w+1}A + mD\right]$, where $m = n + 1$

is the length of the width-w NAF of k and $w + 1$ is the estimated average number of non

zero digits of the width-w NAF of k. The first part of the equation is the time required to

perform precomputation in step 1 and 2 and the second part is the time to perform step 4

of the algorithm.

Algorithm 6: Window NAF method for point multiplication [17].

INPUT: positive integer k, window width w, P ∈ GF(p)

OUTPUT: kP.

1. **Compute** $NAF_w(k)$ using **Algorithm 5**

2. **Compute** $P_i = iP$ for $i \in [1,3,5,....,2^{w-1} - 1]$.

3. $Q \leftarrow \infty$.

4 **For** i **from** n − 1 **to** 0 **do**

     4.1 $Q \leftarrow 2Q$.

     4.2. **If** $k_i = 1$ **then**

          4.2.1. **If** $k_i > 0$ **then** $Q \leftarrow Q + P_{ki}$ **else** $Q \leftarrow Q - P_{-ki}$.

5. **Return(Q).**

Algorithm 6 still processes consecutive bits of *k* which are zero. This can be skipped to give even better performance using a "sliding window" whose width is at most *w* and skipping any sequence of zeroes in the binary representation of *k*. **Algorithm 7** is the sliding widow algorithm. The estimated running time for the sliding window NAF is

$$\left[1D + (\frac{2^w - (-1)^w}{3} - 1)A \right] + \left[\frac{m}{w + v(w)} A + mD \right].$$

Algorithm 7: Sliding window method for point multiplication [17].

INPUT: positive integer k , window width w, P$\in$ GF(p)

OUTPUT:kP.

*1. **Compute** NAF(k) using **Algorithm 3.***

*2. **Compute** $P_i = iP$ for $i \in [1,3,5,......,2(2^w - (-1)^w)/3 - 1]$*

3. $Q \leftarrow \infty, i = n - 1$.

*4 **While** $i \geq 0$ **do***

    *4.1 **If** $k_i = 1$ **then** $t \leftarrow 1$, $u \leftarrow 0$; **else** find the largest $t \leq w$ such that $u \leftarrow (k_i, ...,$*

    $k_{i-t+i})$ *is odd.*

    4.2 $Q \leftarrow 2^t Q$.

    *4.3 **If** $u > 0$ **then** $Q \leftarrow Q + P$; **else** **if** $u < 0$ **then** $Q \leftarrow Q - P_{-u}$.*

    4.4 $i \leftarrow i - t$.

*5. **Return**(Q).*

2.5.4 Known Point Multiplication Algorithm

The known point multiplication algorithms, take advantage of the fact that the fixed points can be precomputed and stored. They can be retrieved each time point multiplication is performed, as opposed to the window methods where the precomputed points are used for that particular point multiplication computation.

For example, if all the doubling in the binary right to left point multiplication algorithms are stored (i.e., points $2P, 2^2P,......, 2^{n-1}P$) then the running time of the algorithm is *mA/2*.

The windowing technique can also be used. Instead of precomputing every 2^i, where $i \in$ *[1,2......, n --1]*, we compute 2^{wi}. The running time becomes $(2^w + d - 3)$, where $d \approx n/w$ and $n \approx m$.

2.6 Elliptic Curve Cryptosystems

In order to use elliptic curve cryptography to encrypt information, the fist step is to map the message or data (plaintext) to a point on the chosen elliptic curve. Elliptic curve operations are then performed on the point to yield the ciphertext. Different elliptic curve cryptosystem use different methods of mapping the plaintext to the curve and different sequences of ECC operations. In this section we describe various systems where elliptic curve cryptography is used for encryption, key exchange, digital signature generation and verification [7].

2.6.1 Elliptic Curve ElGamal Encryption

The elliptic curve ElGamal encryption cryptosystem [7] is used to encrypt information. A simple implementation of the scheme is as follows.

1. A wants to send a message to B.

2. B chooses an elliptic curve E over GF(p), a point α and a secret integer a.

3. B computes $\beta = a\alpha$.

4. B makes α, β public and keeps a private.

5. A expresses her message as a point x on curve E.

6. A chooses a random integer k and computes $y_1 = k\alpha$ and $y_2 = x + k\beta$.

7. A sends y_1 and y_2 (ciphertext) to B.

8. B decrypts the information to get x by computing $x = y_2 - a\,y_1.$

2.6.2 Elliptic Curve Diffie-Hellman Key Exchange

ECC can be employed in key exchange protocols. The Diffie-Hellman key exchange protocol [7] is as follows:

1. A and B want to exchange a key.

2. A and B randomly choose x and y respectively and keep them private to themselves. They both agree on the curve E and point P to use for key exchange.

3. A computes xP, and B computes yP and exchange this information.

4. A then computes $x(yP)$, while B computes $y(xP)$.

5. Both products yield xyP which is used as the key for encryption.

2.6.3 Elliptic Curve Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) [4, 19] is the application of ECC to digital signature generation and verification [14]. The cryptosystem is made up of three procedures:

- Key generation: Used to generate the keys used in digital signing.

- Signature generation: Used to generate the signature for the message using the key obtained from the key generation phase.

- Signature verification: Used by the recipient to verify the integrity of the received message as well as ensure non-repudiation by the sender. The receiver on receiving the message checks the signature and performs the signature verification operation on the received message with the sender's public key and compares the result with the signature to ascertain that the message has not been tampered with.

2.7 Discrete Logarithm Problem in ECC

Elliptic curve cryptography is a public key algorithm and its strength lies in the difficulty of deriving the private key from the public key. The difficulty in obtaining the private key is referred to as the Elliptic Curve Discrete Logarithm Problem (ECDLP) [17].

After point multiplication, i.e. $Q = kP$, both Q and P are the public keys while the integer k is the private key. The ECDLP is the problem of finding k given P and Q.

3. ECC over NIST prime fields

3.1 NIST FIPS 186-2 Standard

The Federal Information Processing Standard (FIPS) 186-2 was issued by the National Institute of Standards and Technology (NIST) in 2000 for applications in which a digital signature was required [5]. Examples include generation and verification of a signature for data integrity assurance and origin authentication. The standard is applicable to all US Federal departments and agencies, but it can also be adopted by private and commercial organizations. Two different digital signature schemes were proposed one utilizing binary fields and the other using prime fields. This thesis focuses on prime fields. **Table 2** lists the various primes recommended by NIST to guard against selecting cryptographically insecure primes in ECC implementations. It also lists the key sizes in RSA that give equivalent security based on the time to attack both algorithm [17].

Table 2: NIST recommended primes [5].

NIST Prime Size (bits)	Equivalent RSA size (bits) [17]	Prime
192	1024	$p192 = 2^{192} - 2^{64} - 1$
224	2048	$p224 = 2^{224} - 2^{96} + 1$
256	3072	$p256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
384	8192	$p384 = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
521	15360	$p521 = 2^{521} - 1$

3.2 Hierarchy of NIST ECC GF(p) Operations

Elliptic curve cryptographic processing is made up of 4 levels:

- Modular operations,

- Point operations,

- Point multiplication, and

- Protocol processing.

At the top of the hierarchy for the NIST FIPS, the protocol processing level is the Elliptic Curve Digital Signature (ECDSA). To digitally sign a message with ECDSA over prime fields, successive point multiplications are performed. Point multiplication itself is performed by a mix of point addition and point doubling operations, at the point operation level of the hierarchy. The point addition and point doubling are performed using modular operations. The modular operations (modular addition, modular subtraction, modular multiplication and modular inversion) are arithmetic operation modulo the chosen NIST prime.

3.3 Modular Arithmetic Operations

3.3.1 Modular Addition and Subtraction

Modular addition is simply *(A+B)* mod p, where p is the selected prime. The computation requires the addition of integers of size 192, 224, 256, 384 or 521 bits depending on the selected prime. Since the final result must be in $[0, p-1]$, we subtract p if necessary. Modular subtraction *(A−B)* mod p can be performed by complementing *B* first and then adding it to *A*.

Algorithm 8: Modular addition [20].

INPUT*: Prime p, integers a and b, m =* $\log_2 p$ *, t = m/32*

OUTPUT *c = (a + b) mod p.*

1. c_i *= Add_with_carry(a_i; b_i).*

2. ***If*** *the carry bit is set,* ***then*** *subtract p from c = (c_{t-1}; ...; c_2; c_1; c_0).*

3. ***If*** *c $\geq$ p* ***then*** *c = c − p.*

4. ***Return****(c).*

Algorithm 9: Modular subtraction [20].

INPUT*: Prime p, integers a and b, m =* $\log_2 p$ *, t = m/32*

OUTPUT: *c = (a − b) mod p.*

1. c_i*= Subtract_with_borrow(a_i; b_i).*

2. ***If*** *the carry bit is set,* ***then*** *add p to c = (c_{t-1}; ... ; c_2; c_1; c_0).*

3. ***Return****(c).*

Algorithm 8 and **Algorithm 9** are the classical modular addition and subtraction methods, where the n-bit prime is broken down into t words and single precison addition/subtraction is performed with n/t bits each time. We combined both algorithms in our implementation into a single algorithm to perform combined modular addition and subtraction. **Algorithm 10** is the combined modular addition and subtraction procedure.

Algorithm 10: Combined modular addition/subtraction.

INPUT: Prime p, integers a, b and add_sub

OUPUT: c = (a ± b) mod p.

*1. **If** add_sub = 0, **then** c = Add_with_carry(a; b); **else** Subtract_with_borrow(a; b).*

*2. **If** add_sub = 0 and the carry bit is set, **then** subtract p from c;*

> *else if add_sub =1 and the carry bit is set, **then** add p to c.*

*3. **If** c ≥ p, **then** c = c − p.*

*4. **Return** (c).*

3.3.2 Modular Multiplication and Squaring

The classical modular multiplication involves multiplying two n-bit integers A and B and then reducing the integer modulo the n-bit prime. The result of the first multiplication *(AB)* produces a 2n-bit result, which must be reduced modulo p to be a member of GF(p). The multiplication is performed using large bitwidth multipliers. Squaring can be performed by a regular multiplier with identical inputs.

Once the double-precision product from multiplication or squaring is obtained, it must be fed into a reduction circuit in order to convert *(AB)* into a single-precision result *(AB)* mod p that lies in the interval $[0, p − 1]$.

3.3.3 Modular Reduction

Modular reduction can be performed by classical reduction algorithm which involves repeatedly subtracting the prime until the result is less than the prime. Faster algorithms such as the Barrett reduction and Montgomery algorithms have been developed [14].

The NIST primes belong to a class of primes referred to as Generalized Mersenne numbers, and their reduction can be performed efficiently by employing the 5 fast reduction algorithms by Solinas [8]. The algorithms consist of a series of n-bit modular additions. The reduction algorithms are reproduced in **Algorithm 11** – **Algorithm 15** from [21].

Algorithm 11: Fast reduction modulo p192 [21].

> *INPUT*: *Integer $c = (c_{11}; ......; c_4; c_3; c_2; c_1; c_0)$, where each c_i is a 32-bit word*
>
> *OUTPUT*: *c mod p192.*
>
> 1.   *Define 192-bit integers: $s_1 = (c_5; c_4; c_3; c_2; c_1; c_0)$, $s_2 = (0; 0; c_7; c_6; c_7; c_6)$, $s_3 = (c_9; c_8; c_9; c_8; 0; 0)$, $s_4 = (c_{11}; c_{10}; c_{11}; c_{10}; c_{11}; c_{10})$.*
>
> 2.   *Return $(s_1 + s_2 + s_3 + s_4 \bmod p192)$.*

Algorithm 12: Fast reduction modulo p224 [21].

> *INPUT*: *Integer $c = (c_{13}; ... ; c_2; c_1; c_0)$, where each c_i is a 32-bit word.*
>
> *OUTPUT*: *c mod p224.*
>
> 1.   *Define 224-bit integers: $s1 = (c_6; c_5; c_4; c_3; c_2; c_1; c_0)$, $s2 = (c_{10}; c_9; c_8; c_7; 0; 0; 0)$, $s_3 = (0; c_{13}; c_{12}; c_{11}; 0; 0; 0)$, $s_4 = (c_{13}; c_{12}; c_{11}; c_{10}; c_9; c_8; c_7)$, $s5 = (0; 0; 0; 0; c_{13}; c_{12}; c_{11})$.*
>
> 2.   *Return $(s_1 + s_2 + s_3 - s_4 - s_5 \bmod p224)$.*

Algorithm 13: Fast reduction modulo p256 [21].

INPUT: Integer $c = (c_{15}; \dots ; c_2; c_1; c_0)$, where each c_i is a 32-bit word.

OUTPUT: c mod p256.

1.   **Define** 256-bit integers: $s_1 = (c_7; c_6; c_5; c_4; c_3; c_2; c_1; c_0)$, $s2 = (c_{15}; c_{14}; c_{13}; c_{12}; c_{11}; 0; 0; 0)$, $s_3 = (0; c_{15}; c_{14}; c_{13}; c_{12}; 0; 0; 0)$, $s_4 = (c_{15}; c_{14}; 0; 0; 0; c_{10}; c_9; c_8)$, $s_5 = (c_8; c_{13}; c_{15}; c_{14}; c_{13}; c_{11}; c_{10}; c_9)$, $s_6 = (c_{10}; c_8; 0; 0; 0; c_{13}; c_{12}; c_{11})$, $s_7 = (c_{11}; c_9; 0; 0; c_{15}; c_{14}; c_{13}; c_{12})$, $s_8 = (c_{12}; 0; c_{10}; c_9; c_8; c_{15}; c_{14}; c_{14})$, $s_9 = (c_{13}; 0; c_{11}; c_{10}; c_9; 0; c_{15}; c_{14})$.

2.   **Return** $(s_1 + s_2 + s_2 + s_3 + s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9$ mod p256).

Algorithm 14: Fast reduction modulo p384 [21].

INPUT: Integer $c = (c_{23}; \dots ; c_2; c_1; c_0)$, where each c_i is a 32-bit word.

OUTPUT: c mod p384.

1.   **Define** 384-bit integers: $s_1 = (c_{11}; c_{10}; c_9; c_8; c_7; c_6; c_5; c_4; c_3; c_2; c_1; c_0)$, $s_2 = (0; 0; 0; 0; 0; c_{23}; c_{22}; c_{21}; 0; 0; 0; 0)$, $s_3 = (c_{23}; c_{22}; c_{21}; c_{20}; c_{19}; c_{18}; c_{17}; c_{16}; c_{15}; c_{14}; c_{13}; c_{12})$, $s4 = (c_{20}; c_{19}; c_{18}; c_{17}; c_{16}; c_{15}; c_{14}; c_{13}; c_{12}; c_{23}; c_{22}; c_{21})$, $s_5 = (c_{19}; c_{18}; c_{17}; c_{16}; c_{15}; c_{14}; c_{13}; c_{12}; c_{20}; 0; c_{23}; 0)$, $s_6 = (0; 0; 0; 0; c_{23}; c_{22}; c_{21}; c_{20}; 0; 0; 0; 0)$, $s_7 = (0; 0; 0; 0; 0; 0; c_{23}; c_{22}; c_{21}; 0; 0; c_{20})$, $s_8 = (c_{22}; c_{21}; c_{20}; c_{19}; c_{18}; c_{17}; c_{16}; c_{15}; c_{14}; c_{13}; c_{12}; c_{23})$, $s_9 = (0; 0; 0; 0; 0; 0; 0; c_{23}; c_{22}; c_{21}; c_{20}; 0)$, $s_{10} = (0; 0; 0; 0; 0; 0; 0; c_{23}; c_{23}; 0; 0; 0)$.

2.   **Return** $(s_1 + s_2 + s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10}$ mod p384).

Algorithm 15: Fast reduction modulo p521 [5].

INPUT: A binary integer $c = (c_{1041}; \ldots ; c_2; c_1; c_0)$.

OUTPUT: c mod p521.

1.   ***Define*** *521-bit integers:* $s_1 = (c_{1041}; \ldots ; c_{523}; c_{522}; c_{521})$, $s2 = (c_{520}; \ldots ; c_2; c_1; c_0)$.

2.   ***Return*** $(s_1 + s_2$ *mod p521*).

Software implementation of these algorithm resulted in 3-10 times speedup than Barrett Reduction [14]. This also applies to Montgomery reduction, as running time of Montgomery reduction is about the same as that of Barrett reduction [35].

3.3.4 Modular Inversion

Modular inversion is the most computationally expensive of all modular operations, and its actual running time is data-dependent. To perform modular inversion A^{-1} mod p two types of algorithms can be used. They are the extended Euclidean algorithm and the Fermat's theorem based modular inversion algorithm

Algorithm 16 from [20] is a modified version of the extended Euclidean algorithm.

Algorithm 16: Binary extended Euclidean algorithm [20].

INPUT: Prime p, integer 0 < a < p

OUTPUT: a⁻¹ mod p

1. u = a, v = p, A = 1, C = 0.

*2. **While** u ≠ 0 **do***

   *2.1 **While** u is even **do***

      2.1.1. u = u/2.

      *2.1.2. **If** A is even, **then** A = A /2; **else** A = (A + p) /2*

   *2.2 **While** v is even **do***

      2.2.1. v = v/2.

      *2.2.2. **If** C is even, **then** C = C /2; **else** C = (C + p) /2.*

   *2.3 **If** u ≥ v, **then**: u = u – v; A = A – C; **else**: v = v – u; C = C – A.*

*3. **Return** (C mod p).*

To avoid manipulation of signed numbers, we modified **Algorithm 16** so that only

positive integers are used. Our modified version, **Algorithm 17**, is presented below.

Algorithm 17: Modified binary extended Euclidean inversion algorithm.

INPUT *Prime p, a where 0 < a < p*

OUTPUT a^{-1} *mod p*

1. u = a, v = p, A = 1, C = 0.

2. **While** *u ≠ 0* **do**

   2.1 **While** *u is even* **do**

      2.1.1. u = u/2.

      2.1.2. **If** *A is even* **then** *A← A /2;* **else** *A ← (A + p) /2.*

   2.2 **While** *v is even* **do**

      2.2.1. v = v/2.

      2.2.2. **If** *C is even* **then** *C← C /2;* **else** *C ← (C + p) /2.*

   2.3 **If** *u ≥ v* **then**

      2.3.1. u= u – v; **if** *A >C* **then**, *A = A – C;* **else** *A = A + p – C.*

   Else

      2.3.2. v = v – u; **if** *C > A* **then**, *C= C – A;* **else** *C = C + p – A.*

3. **Return** *(C mod p).*

The worst case running time of the algorithm is 2nX where X is the time it takes to perform step 2, and n is the number of bits of the prime.

The second modular inversion technique is based on the Fermat's theorem [17]. Fermat's theorem states that if $a^{p-1} \equiv 1$ mod p for $a \in$ GF(p) then $a^{-1} = \dfrac{1}{a} = \dfrac{a^{p-1}}{a} = a^{p-2}$. In other words, we can also compute the modular inverse of a by modular exponentiation.

Algorithm 18 presents the right to left modular exponentiation method application to Fermat's theorem. The running time of the algorithm is *2nM,* where *M* is the time it takes to perform Modular multiplication. This inversion technique is better than the extended Euclidean inversion algorithm, if *M* is less than *X.*

Algorithm 18: Fermat's theorem based inversion algorithm [14].

INPUT: Prime p = (p_{n-1},.....,p_1, p_0),, integer a: 0 < a < p – 1

OUTPUT: a $^{-1}$ mod p

1. For i from 0 to n – 2 do

     1.1 Q ← a^2.

     1.2 If p_i = 1 then Q ← Qa.

2. Return (Q).

4. Reconfigurable Processor for ECC over NIST Prime Fields

4.1 Design Objectives and Challenges

Since ECC protocols routinely perform point multiplications Q = kP to secure data, an *efficient* implementation of this operation is critical. Since there are many different algorithms for point multiplication and many different elliptic curves with different tradeoffs between the security level and computational efficiency, a *flexible* implementation is also desired. This chapter presents a reconfigurable ECC processor that attempts to strike a balance between the conflicting requirements of efficiency (e.g., speed, area, power) and flexibility (e.g., multiple curves, multiple point multiplication methods).

The proposed architecture performs arithmetic operations over prime fields GF(p). In particular, our processor can be reconfigured for any of the five fields with NIST recommended primes p of size 192, 224, 256, 384, and 521 bits. The corresponding modulo p operations addition, subtraction, multiplication, and inversion form the instruction set of the processor. These instructions can be combined into simple programs for performing basic ECC operations, such as point addition. Point additions are needed in all point multiplication methods. Selecting a different point multiplication algorithm would only require changing the corresponding sequence of programs for point addition operations. Thus, the flexibility of our ECC processor is characterized by (1) supporting prime fields GF(p) with five different prime sizes, and (2) supporting programmable point addition for implementing any desired point multiplication method. For the sake of

efficiency, we have decided to focus only on NIST prime fields, as opposed to other GF(p) alternatives with the same prime size. The main reason behind NIST recommendations is the possibility of fast modular reductions, and our ECC processor takes full advantage of this. The well-known concepts of concurrency and locality have also been utilized to improve the efficiency of the proposed architecture.

None of the existing hardware solutions (described in the next section) for ECC over GF(p) can handle different sizes of prime p, ranging from 192 to 521 bits. The proposed ECC processor fills this flexibility gap.

4.2 Previous Work

4.2.1 Literature Summary

Software-based implementations of ECC, such as [20], are flexible but inefficient as a general-purpose instruction set architecture (ISA) of the underlying hardware is not optimized for computationally intensive cryptographic operations. An ISA can be extended to provide partial support for ECC-related arithmetic operations, as reported in [22, 23]. A more aggressive approach would be to introduce a special arithmetic unit for accelerating modular operations or point addition [24, 25, 26, 27]. Several researchers have also proposed dedicated architectures performing a complete point multiplication operation [28, 29, 30, 31]. Obviously, as an architecture becomes more specific, its efficiency increases and its flexibility decreases.

Table 3 summarizes the hardware solutions specifically optimized for ECC over GF(p), excluding general-purpose ISA extensions.

Table 3: Hardware processors for ECC over GF(p).

Ref.	Prime Size	Point Add.	Point Mult.
[31]	256	Fixed	Fixed
[30]	192	Fixed	Fixed
[29]	192	Fixed	Fixed
[28]	160	Fixed	Fixed
[27]	256	Fixed	Flexible
[26]	256	Fixed	Flexible
[25]	256	Flexible	Flexible
[24]	256	Flexible	Flexible
Ours	192, 224, 256, 384, 521	Flexible	Flexible

Implementations reported in [28, 29, 30, 31] commit to a specific point multiplication method and a specific prime size. A hardwired point multiplication method in such designs cannot be changed. The prime itself can be varied as long as its size remains the same. For example, the architecture from [31] can work with any prime whose size does not exceed 256 bits. Consequently, it can handle NIST primes p192, p224, and p256, but not p384 or p521.

Implementations reported in [26, 27] commit to a specific point addition method and a specific prime size. Such designs allow a user to employ different point multiplication methods; however, the point addition method and the prime size are hardwired and cannot be changed. For example, the architecture from [26] is restricted to 256-bit point operations, i.e., it cannot handle NIST primes p384 and p521.

Implementations reported in [24, 25] are the most flexible architectures that are still relatively efficient. Their purpose is to speed up arithmetic computations over GF(p) for

faster point operations. The control flow of these point operations (e.g., the point multiplication algorithm) is external and can be changed without altering the hardware architecture in question. Our ECC processor falls into this design category. The advantage of the proposed architecture over [24, 25] is in its ability to work with all five NIST primes (including large-sized p384 and p521), which is achieved through efficient reconfigurations. Another unique feature of our design is the utilization of the fast reduction schemes specific to NIST primes. Consequently, conventional Montgomery type modular multiplications [34] can be replaced by regular multiplications followed by fast reductions, which simplifies the processor datapath.

4.2.2 Literature Details

Wolkerstorfer [25] designed a dual-field ECC processor. It utilized a bit-serial multiplier with interleaved modular reduction. The interleaved reduction was rarely accurate, so a final correction circuit using a table lookup was employed to yield the final result. Also, a slightly modified binary extended Euclidean algorithm was used for modular inversion with a signed number representation. To handle binary field a carry save adder was used in their modular adder circuit and signed number representation was used for subtraction.

Quan *et al* [26] described several multiplier architectures operating with large bitwidth data. These multipliers are essential to the ECC over GF(p), and their performance characteristics have direct impact on the computational efficiency of the ECC hardware. They designed a multiplier area-delay estimator and used it to explore the design space of large bitwidth multiplier. From their analysis a multiplier was selected and used in a 256-bit point addition circuit.

Gutub *et al* [27] explored parallelism as a method of improving speed and reducing power consumption. They investigated the power-time tradeoff of using 1, 2 and 4 multipliers within each core in an ECC GF(p) processor. The multiplier and adder were the exact same type as in [29].

Ors *et al* [28] described an implementation based on a 160-bit prime. The hardware was designed to perform low-level modular addition, multiplication (using Montgomery

method), and inversion as well as high-level point addition, doubling (using modified Jacobian coordinates), and multiplication in Projective coordinates. The control structure was hierarchical: blocks for the low level operations were controlled by blocks that performed point addition, which was controlled by the block performing point multiplication. The controlling block actuated a start signal and the controlled block asserted a done signal on completion of computation. The datapath of the design was fixed at design time. The entire datapath was in use during the systems operation even when working with smaller primes.

Orlando and Paar [29] proposed a processor that utilized a high radix Montgomery multiplier (for modular multiplication) and an inverter based on Fermat's theorem (for inversion). Their high radix Montgomery multiplication required precomputation of a large number of variables depending on the word length of the prime and as such required a large memory to store these points. The processor's controller was hardwired to use standard Projective coordinates for point representation i.e. it would not be able to take advantage of any other point representations.

Shuhua and Yuefei [30] explored time-area tradeoffs for an ECC processor working with an arbitrary 192-bit prime. They optimized Montgomery multiplication to deliver a 6-ms point multiplication in their FPGA implementation. They compared their design to the processor in [29] for the p192 NIST prime, which could perform point multiplications in 4 ms. Even though [30] was slower than [29], the former used only 1/6 of the resources required by the latter. Their design was also able to use multiple cores for execution to speedup computation. In order to support larger primes, an increase in the memory size is required.

Satoh and Takano [31] proposed a dual-field ECC processor based on a Montgomery multiplier that could perform modular multiplications in both prime and binary fields. The processor used NAF point multiplication algorithm for point multiplication and mixed coordinates for its point representation. The processor could work with variable

length prime by storing the size of the prime in a register before commencing operation. Their ASIC implementation could handle operand sizes from 160 to 256 bits.

McIvor *et al* [32] designed a processor that utilized the finely interleaved operand scanning (FIOS) Montgomery multiplication [33], which allowed for the variable size of the data in multiples of 32-bit words. The size of the datapath however had to be fixed at design time and would require reconfiguration of the FPGA in response to change the size of the prime. In their study, they compared the FIOS Montgomery multiplier with standard Montgomery multiplier for binary point multiplication in standard Projective coordinates. Both multipliers used the 16 x 16 bit multipliers blocks in the Xilinx FPGA and were cascaded to form larger multipliers. From the study the ordinary Montgomery multiplier offered a better throughput rate when used in ECC point addition but the FIOS implementation provided better flexibility.

4.3 Processor Architecture

4.3.1 Execution Environment

Our proposed execution environment allows for dynamic optimization of point multiplication $Q = kP$ based on the value of k. This environment includes the following components:

- **Scheduler:** software procedure generating an optimized sequence of point operations translated into the corresponding sequences of modular arithmetic operations;

- **Processor:** reconfigurable hardware unit which could be mapped to FPGA or ASIC executing the generated sequences of modular arithmetic operations;

- **Supervisor:** general-purpose microprocessor running the scheduler and controlling the processor.

The supervisor is responsible for computing $Q = kP$ once P and k are known. By default, it can execute any available software implementation of point multiplication. Alternatively, it can take advantage of the processor for more efficient point additions and point doublings. Each of these point operations can be performed by generatig a program made up of a sequence of modular arithmetic operations, such as modular addition, subtraction, multiplication, and inversion. The supervisor can assign multiple programs (i.e., multiple point operations) to the processor. These programs are generated by the scheduler, which is a software procedure executed by the supervisor. The responsibility of the scheduler is to analyze the value of k (e.g., scan ahead a subset of its bits) and optimize the sequence of point operations accordingly.

Note that the described execution environment offers a high degree of flexibility. One extreme is to perform point multiplication entirely in software using the supervisor, without utilizing the scheduler and the processor. The other extreme is to perform all necessary operations using only the processor, whose programming is performed by the scheduler based on the value of k. As a compromise, computations can be partitioned between the supervisor and the processor based on desired flexibility or speed. A good execution scenario should achieve high efficiency of point multiplications, i.e., the total latency and energy consumption should be minimized. In this thesis we focus on the processor design described next.

4.3.2 Hardware Organization

Our proposed processor is composed of five major units: Main Memory, two Operand Registers (A and B), Control Unit (CU), and Functional Unit (FU). The top-level block

diagram is shown in **Figure 1**. The supervisor communicates with the processor by asserting certain signals and loading a program (a sequence of processor instructions) into the internal main memory of the processor.

The I/O signals to/from the processor are as follows:

- **SUPERVISOR:** asserted to indicate that the main memory is accessed externally by the supervisor to write instructions and write/read data,

- **WE:** high/low when the main memory is written/read externally by the supervisor,

- **DIN:** external 32-bit data input (also used to load instructions into the main memory),

- **DOUT:** external 32-bit data output,

- **ADDR:** external 9-bit address,

- **GLOBAL START:** asserted to start program execution,

- **BUSY:** asserted to indicate that the processor is busy (the main memory is in use internally),

- **GLOBAL RESET:** asserted to reset the processor.
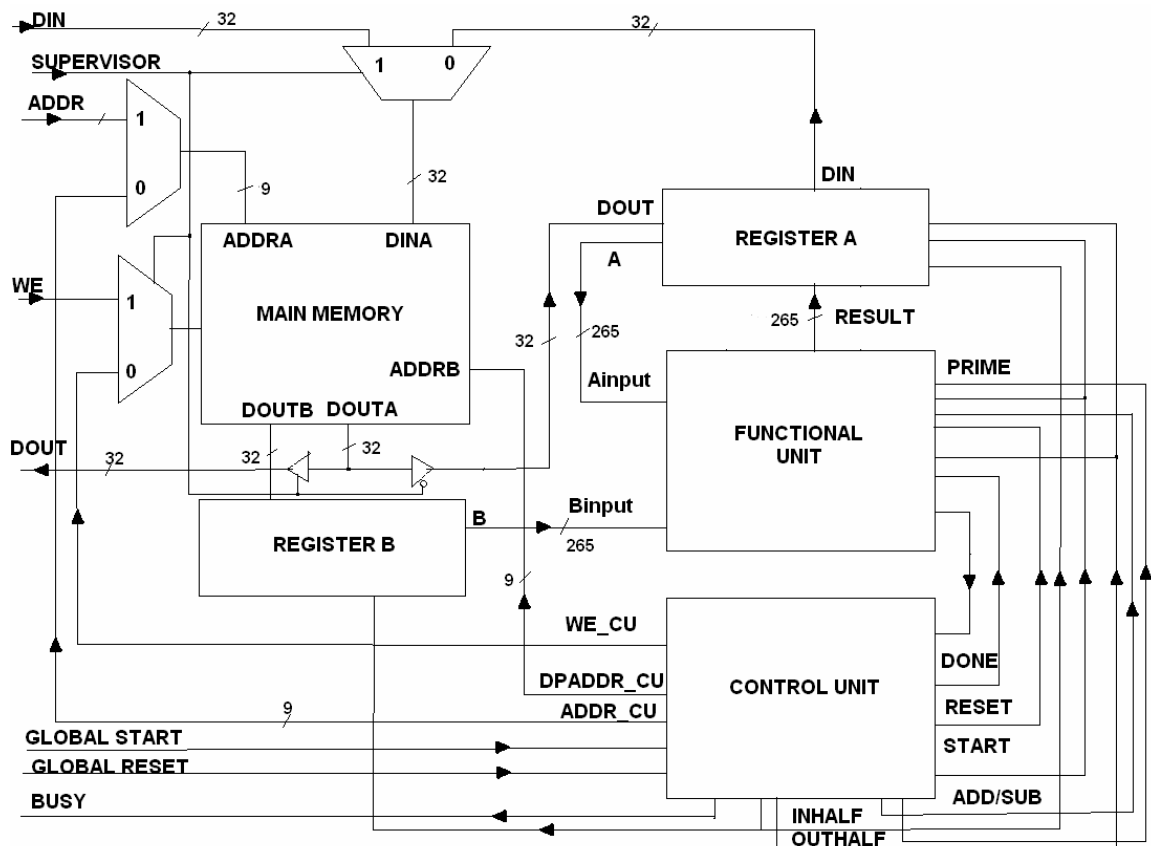
Figure 1: Top-level block diagram of the processor.

4.3.2.1 Main Memory

The main memory is a 512×32-bit dual port memory used to store both instructions and data. The memory can be accessed internally by the processor (reading/writing data and reading instructions) and externally by the supervisor (writing/reading data and writing instructions). Location 0 is reserved for the exclusive use by the supervisor, this location is not accessed by the processor. It indicates that one of the operands for a binary operation, or the single operand of a unary operation is already stored in operand register A, thus that operand does not need to be fetched from main memory. The available 511 memory locations are sufficient to hold data and instructions to perform ECC point

operations. For example, the main memory can hold twenty-five 521-bit operands and still have room for eighty-five 32-bit instructions.

4.3.2.2 Operand Register A and B

Operand registers are 2×265 bits registers used to store the large bitwidth operands of an instruction after they are retrieved from the main memory. They accelerate the operation of the functional unit, as it can access data 265 bits at a time from register A and B, as opposed to 32 bits at a time from the main memory. For 384-bit and 521-bit operations they take two clock cycles to transfer the most significant part (MSP) and least significant part (LSP) of the data to the functional units. The operand registers can also serve as a temporary storage for the results of execution of an instruction to be used as an operand for the next successive instruction. Thus, these registers reduce memory access time for the next operation. This feature is especially useful in NIST elliptic curve operations, since the program is usually sequential with no branch instructions, and the output of one instruction is often one of the inputs of the next instruction.

4.3.2.3 Control Unit

This unit fetches and decodes instructions from the main memory and sends appropriate control signals to the functional unit and main memory. It contains a microprogram, a 32-bit instruction register, and a 9-bit program counter. When the GLOBAL START signal is asserted and the SUPERVISOR signal goes low, the control unit asserts the BUSY signal and begins reading instructions from the main memory. Once it encounters a stop instruction, control is passed back to the supervisor. More details are provided in section 4.5.

4.3.2.4 Functional Unit

This unit houses various blocks that perform modular addition/subtraction, multiplication with reduction, and inversion. These blocks are controlled by the control unit and receive the operands from registers A and B. More details are provided in section 5.4.

4.3.3 Instruction Set of Proposed Processor

Processor instruction words are divided into four fields as shown in **Table 4**. The Opcode field is used to represent the instructions type, the Operand1 and Operand2 fields represent the address of the operands of a binary operation. For unary operations, the single operand address is stored in the Operand1 field. The Result field contains the address of the memory location where the result is to be stored

Table 4: Format of 32-bit instruction.

Opcode	Operand1	Operand2	Result
5 bits	9 bits	9 bits	9 bits

The opcode field is 5-bits wide as described in **Table 5**.

Table 5: Opcode field of instruction.

Opcode	Description	Notation
00xxx	Modular Addition	ADD
01xxx	Modular Multiplication	MULT
10xxx	Modular Inversion	INV
11xxx	Modular Subtraction	SUB
11110	Jump	JUMP
11111	Stop	STOP

The 'xxx' portion of the opcodes shown in **Table 5** encodes the prime in operation: 000 for p192, 001 for p224, 010 for p256, 011 for p384, and 100 for p521. The operands and the result are as wide as the selected prime and stored in the main memory as n successive 32-bit words, where n = 6, 7, 8, 12, or 17 for p192, p244, p256, p384, and p521 respectively. The 9-bit Operand1, Operand2, and Result fields contain the address of the first 32-bit word of the corresponding data.

Once the instruction is fetched and decoded, the control unit asserts the reset signal for the appropriate functional unit block responsible for computing a modular operation. Then, the control unit asserts the START signal of the block and waits for the DONE signal to be asserted by the block upon completion of instruction execution. The asserted DONE signal indicates that the next instruction may be fetched from the main memory. This signal greatly simplifies the control unit design and decouples the control flow from timing variations of the individual blocks. In other words, whether the processor executes slow 521-bit inversion or fast 192-bit modular subtraction, the control state

diagram remains the same. Moreover, if a block is modified and its timing is changed, the same control unit can be used, which makes the overall architecture reusable.

The instructions are sequentially fetched and executed until a stop or a jump is encountered. If a jump instruction is encountered, the appropriate jump address is loaded into the program counter. If a stop instruction is encountered, execution of the program is stopped until the GLOBAL START signal is received from the supervisor. In other words, if the supervisor does not need to perform ECC computations, the processor remains in a low-power mode without any switching activity.

4.3.4 Programming Example

Using the processors instruction set, we present examples of point doubling and point addition here. The point doubling is performed in Jacobian Projective coordinates (section 2.4.3) and point addition is performed in mixed Jacobian representation.

4.3.4.1 Point Doubling Example

Let X1, Y1, Z1 represent the addresses of the Jacobian coordinates of point P. Let T1, T2, T3, and T4 represent the addresses of the variables used to hold intermediate results during computation. Let X3, Y3, Z3 represent the addresses of the Jacobian coordinates of the resulting point Q = 2P. An example of the processor instructions used to perform point doubling is given below.

1. MULT   Z1, Z1, T1

2. SUB   X1, T1, T2

3. ADD   X1, T1, T1

4. MULT   T1, T2, T2

5. ADD   T2, T2, T3

6. ADD   T3, T2, T2

7. ADD   Y1, Y1, Y3

8. MULT   Y3, Z1, Z3

9. MULT   Y3, Y3, Y3

10. MULT   Y3, X1, T3

11. MULT   Y3, Y3, Y3

12. MULT   Y3, T4, Y3

13. MULT   T2, T2, X3

14. ADD   T3, T3, T1

15. SUB   X3, T1, X3

16. SUB   T3, X3, T1

17. MULT   T1, T2, T1

18. SUB   T1, Y3, Y3

Note that the above sequence involves nine multiplications and nine additions/subtractions.

4.3.4.2 Point Addition Example

Let X1, Y1, Z1 represent the addresses of the Jacobian Projective coordinates of point P_1 and X2, Y2, Z2 represent the addresses of the Jacobian Projective coordinates of point P_2. Let T1, T2, T3 and T4 represent the addresses of the variables used to hold intermediate results during computation. Let X3, Y3, Z3 represent the address of the Jacobian

Projective coordinates of the resulting point $Q = P_1 + P_2$. An example of the processor instructions performing point doubling is given below.

1. MULT   Z1, Z1, T1

2. MULT   T1, Z1, T2

3. MULT   T1, X1, T1

4. MULT   T2, Y2, T2

5. SUB   T1, X1, T1

6. SUB   T2, Y1, T2

7. MULT   Z1, T1, Z3

8. MULT   T1, T1, T3

9. MULT   T1, T3, T4

10. MULT   T3, X1, T3

11. ADD   T3, T3, T1

12. ADD   T2, T2, X3

13. SUB   X3, T1, X3

14. SUB   X3, T4, X3

15. SUB   T3, X3, T3

16. MULT   T3, T2, T3

17. MULT   T4, Y1, T4

18. MULT   T3, T4, Y3

Note that the above sequence involves seven multiplications and eight additions/subtractions.

4.3.5 Reconfiguration

The latency variations are inevitable as the functional unit is *reconfigurable*. Since several primes are supported, the amount of time it takes to perform a modular operation depends on the used prime. For example, a multiplication modulo p384 may require twice as much processing as that of modulo p192, hence the execution latency of a **multiply** instruction may vary. Note that the selected prime is encoded in the instruction's opcode field and sent to the functional unit as the 3-bit signal PRIME. If this signal is changed the functional unit is reconfigured: the necessary adjustments of the datapath operations are performed by local controllers. Thus, conventional downloads of configuration bitstreams are not needed, thus there are no delay or energy penalties associated with processor reconfigurations.

4.4 Functional Unit

4.4.1 Reconfigurable Datapath

During the FU design one must make a critical decision on the datapath width, which greatly impacts the area, speed, and power consumption of the processor.

Our datapath of the circuit is split into smaller word-sized parts, to enable the shut down of unused portions in order to reduce switching activity. This is depicted in **Figure 2** where the decoder enables the *load_enable* signal to the registers. The decoder input is the 3-bit *select_prime* used to encode the prime. The *load_enable* signal also controls the 1-bit registers that connect the carry-out of the previous section to its carry-in.
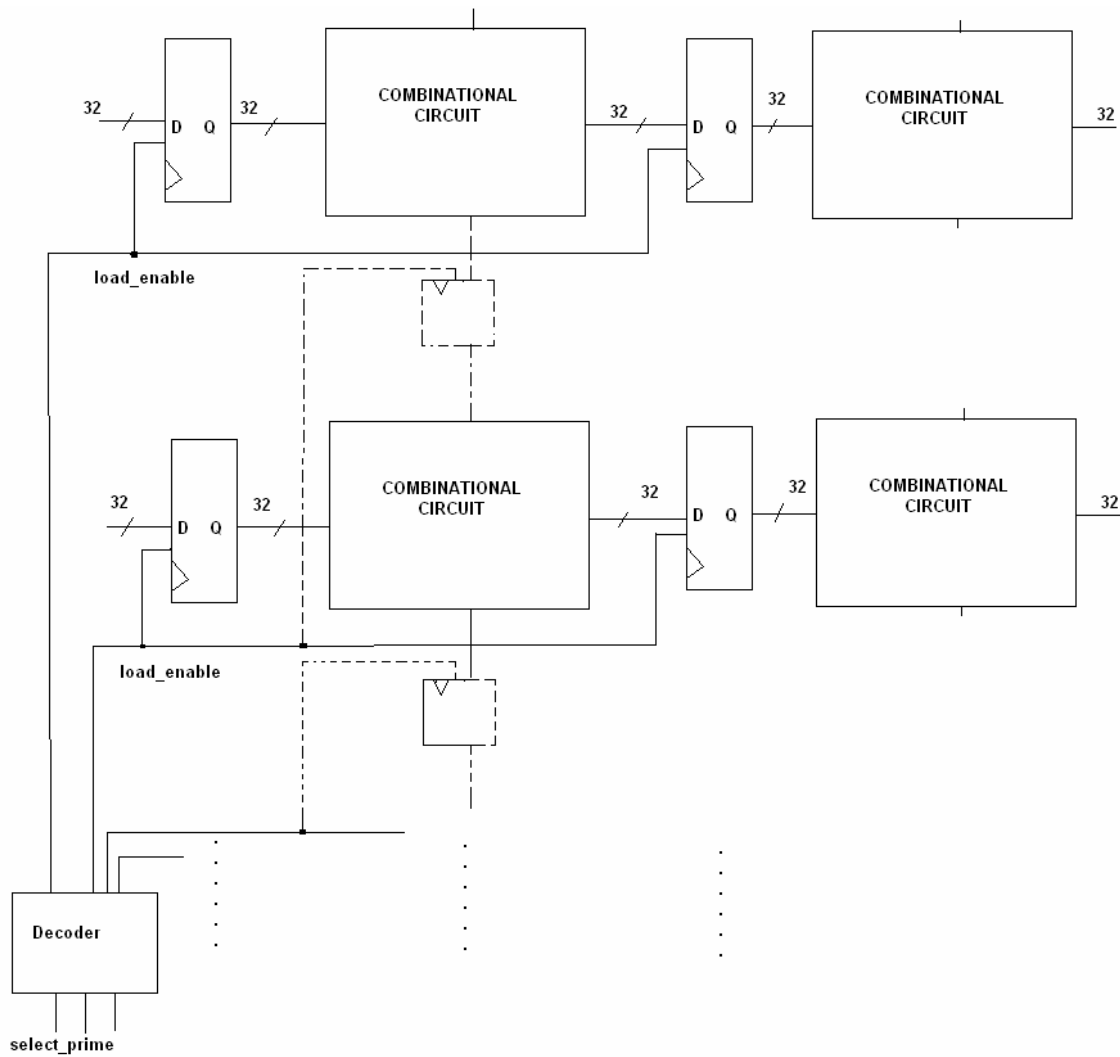
Figure 2: Connection within blocks in FU to reduce switching activity in unused parts of circuit.

Various choices exist for the size of the datapath of our circuit as well as the word size of the circuit. The selected datapath size and word size should:

- minimize the number of unused bits as a result of the circuit handling different prime sizes, and

- minimize control signals needed to shut down the unused part of the datapath.

The datapath of the functional unit is 265-bit wide. The reason behind this design decision is as follows. The NIST prime sizes shown in **Table 6** are divisible by 32, except for p521 whose size is nine bits longer than 512, the closest multiple of 32. The size of operands A and B is determined by the size of the used prime and it ranges from six 32-bit words (prime p192) to sixteen 32-bit words with an extra 9-bit portion (prime p521).

Table 6: Number of unused bits and number of execution passes for different datapath widths.

Datapath	p192		p224		p256		p384		P521		Average	
Width	EP	UB	EP	UB	EP	UB	EP	UB	EP	UB	EP	UB
32	6	0	7	0	8	0	12	0	17	23	10.0	4.6
64	3	0	4	32	4	0	6	0	9	55	5.2	17.4
128	2	64	2	32	2	0	3	0	5	119	2.8	43
192	1	0	2	160	2	128	2	0	3	55	2.0	68.6
224	1	32	1	0	2	192	2	64	3	151	1.8	87.8
256	1	64	1	32	1	0	2	128	3	247	1.6	94.2
384	1	192	1	160	1	128	1	0	2	247	1.2	145.4
512	1	320	1	288	1	256	1	128	2	503	1.2	299
521	1	329	1	297	1	265	1	137	1	0	1.0	205.6
265	**1**	**73**	**1**	**41**	**1**	**9**	**2**	**146**	**2**	**9**	**1.4**	**55.6**

Table 6 compares various datapath width options in terms of the number of execution passes (EP) and the number of unused bits (UB). For example, processing 224-bit operands with the 64-bit datapath requires four passes, i.e., EP = 4. The total number of bits that can be processed with these four passes is $64 \times 4 = 256$. Since only 224 bits are

needed, thirty-two bits are unused, i.e., UB = 32. Our assumption is that each prime is equally likely to be selected so the average values of EP and UB will be used to make our decision on the datapth size. The average values of EP and UB over all primes for the 64-bit datapath are 5.2 and 17.4 respectively. Ideally, both EP and UB should be small, which can be interpreted as an heuristic measure of the datapath efficiency. We have selected the width of 256 + 9 = 265 bits, which yields EP = 1.4 and UB = 55.6 on average. The alternatives with the smaller EP (i.e., better execution speed, e.g., 384-bitwidth) have much larger UB, while the alternatives with the smaller UB (i.e., better area utilization, e.g., 128-bitwidth) have much larger EP. In other words, the 265-bit offers a balanced compromise between execution speed and area utilization.

As not all datapath bits are used all the time, the registers of the unused datapath portions can be disabled to reduce unnecessary switching activity. The word size decision was based on the least number of *load_enable* signals (See **Figure 2**) for the chosen datapath size of 265 bits.

Table 7: Number of control signals for different word sizes.

Word Size	Control Signals
2	133
4	67
8	34
16	17
32	9
64	5
128	2

Table 7 shows some of the choices. The word size of 32 bits offers a good compromise between the number of control signals and the number of bits that cannot be disabled. For example, while working with p224, 64-bit words will enable $64 \times 4 = 256$ bits of the datapath, i.e., 32 bits will not be disabled. On the other hand, using 16-bit words offers no advantage over 32-bit words in terms of the number of not-disabled bits, while increasing the number of control signals. Hence, our datapath has been divided into 6, 7, 8, 12 and 17 32-bit parts for p192, p224, p256, p384, p521 respectively. Consequently, we designed 32-bit components for our registers, multiplexers, counters, comparators.

The individual blocks within the functional unit are the Combined Modular Adder/Subtractor, the Modular Inverter and the Modular Multiplier. Each block has been designed to include a dedicated local controller. The inputs and outputs of each block are registered to allow independent operation with respect to the other blocks. In other words, the datapath allows for parallel instruction execution, even though our control unit currently schedules instructions sequentially. It also allowed us design, test and optimize each block independently before integrating them into the entire processor. The individual blocks within the functional unit shown in **Figure 3** are described next.
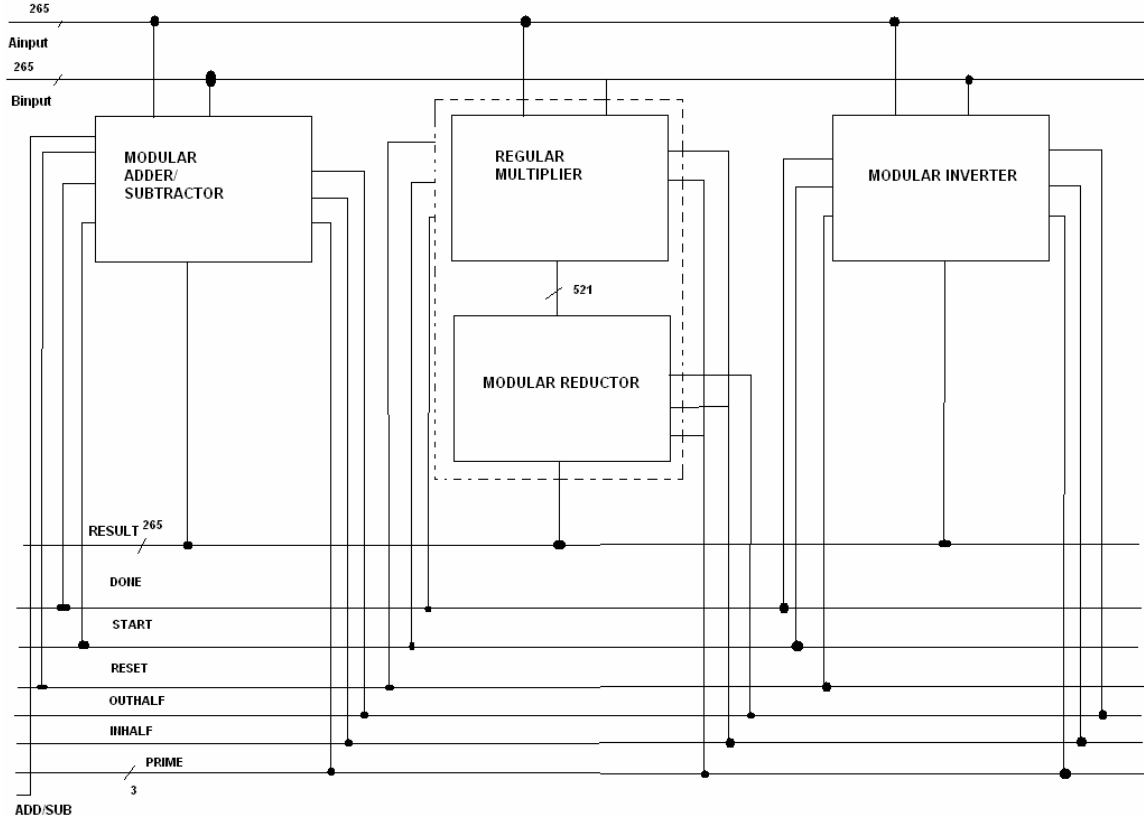
Figure 3: Organization of blocks within functional unit.

4.4.2 Combined Modular Adder/Subtractor

Our combined modular adder/subtractor (CMAS) uses **Algorithm 10** from Section 3.3.1

The CMAS block diagram is given in **Figure 4**, where one can see two pairs of serially

connected registers. The first register is used for pipelining while the second register is

needed when working with p384 and p521. In those two cases the data is comprised of

the 256-bit least significant part (LSP) and the remaining most significant part (MSP),

which is 128-bit wide for 384-bit data and 265-bit wide for 521-bit data. The second

register delays the LSP till the MSP of the intermediate result is available. Steps 2 and 3

of Algorithm 1 are performed only after the MSP has been added/subtracted or compared

− hence, the need for the delay register. A 1-bit registers also stores the carry-out from the

LSP for use as a carry-in of the MSP. The block takes 10 cycles to complete modular addition for p192, p224 and p256 and 11 cycles for p384 and p521.
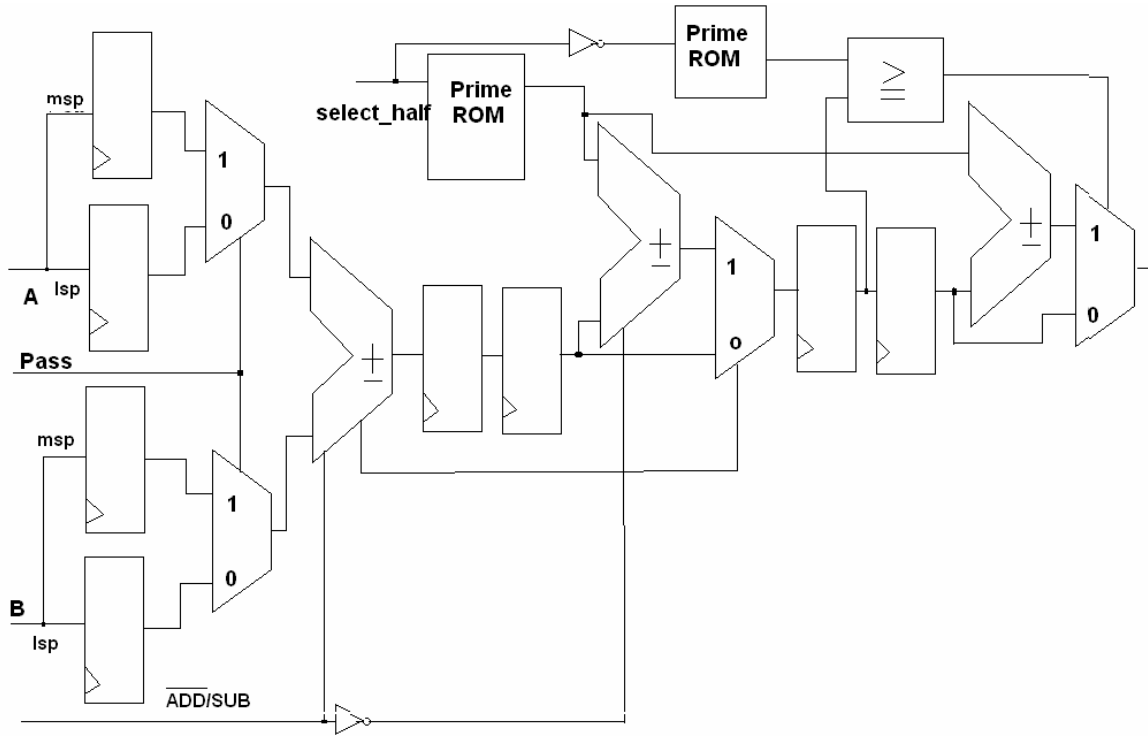
Figure 4: Block diagram of modular adder/subtractor.

4.4.3 Modular Inverter

Our proposed inverter is a hardware implementation of **Algorithm 17** which is a modified version of the inversion algorithm from [6]. The modifications to the algorithm are such that we work only with unsigned integers, thus eliminating the need for signed binary shifters or signed comparators. In our implementation we take advantage of the fact that step 2.1 and step 2.2 of **Algorithm 17** can be performed in parallel. Since modular inversion is an expensive operation with data dependent delays, we deliberately avoided hardware time-multiplexing for 384 and 521-bit computations. We used a datapath with 521-bit adders/subtractors, so that each addition/subtraction can be

performed in one pass. Note that the unused portions of the datapath are disabled when working with primes other than p384 and p521. The 521-bit adder/subtractor is constructed by pipelining a 256-bit adder/subtractor (eight 32-bit components) and a 265-bit adder (eight 32-bit components with one 9-bit component), as shown in **Figure 5**. We also took advantage of the fact that inversion is a unary operation and requires only one operand: both LSP and MSP of 384- or 521-bit data can be read and written simultaneously using both ports of the main memory, thus saving communication time.
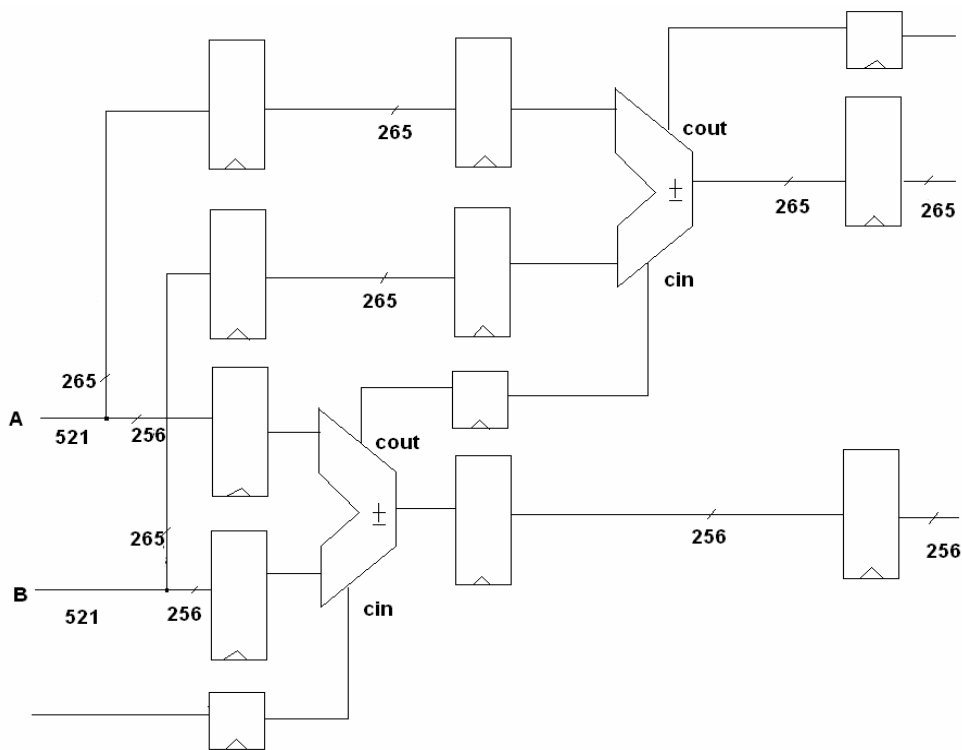
Figure 5: Pipelining of 521-bit adder/subtractor inside inverter.

4.4.4 Modular Multiplier

In order to perform modular multiplication, we designed a multiplier followed by a modulo reduction circuit. As in the other blocks we used time multiplexing. A 256-bit

multiplier was used for multiplication over p192, p224 and p256. For larger bitwidth, a divide and conquer multiplier was used, which performed intermediate multiplications using the 256-bit multiplier.

The reduction circuit implements the NIST reduction algorithms introduced in Section 3.3.3. Consequently, our modular multiplier does not use Montgomery multiplication, unlike other hardware implementations reported in literature [24, 28, 29, 30, 31]. To the best of our knowledge, our multiplier with fast reduction is the first implementation of its kind.

4.4.4.1   256-bit Multiplier

In order to decide which multiplier is suitable for NIST ECC over prime fields, we considered certain desired characteristics of the hardware implementation below.

1. Take advantage of parallelism by processing multiple bits at a time, to speedup multiplication.

2. All bits of the result must be available before reduction can commence, as irregular word permutations are required.

3. The multiplier should have a regular repeating structure and not use lots of routing resources.

4. Pipelining is unlikely to be advantageous because of data dependencies. As one can see in the examples of point addition and doubling (Sections 4.3.4.1 and 4.3.4.2) there are very few occasions requiring successive multiplications with independent operands.

Radix-2 sequential multipliers are unsuitable as multipliers for large bitwidth multiplications, because their time complexity is comparable to Montgomery multiplication [34], and we would like to perform modular multiplication faster.

Large bitwidth multiplication is actually a series of repeated addition and speedup usually comes in the form of performing simultaneous addition of operands together using tree or array multipliers or dividing the numbers to be added into smaller parts and adding them together using high radix multipliers. Tree multiplier structures do not use regular repeating structures. Consequently, they utilize connections of various lengths that can lead to hazards and signal skew and hence make placing and routing more difficult in comparison to array multipliers [36]. Array multipliers use shorter wires and have a regular structure, but are slower than the tree multipliers unless pipelining is used.

Bit serial multipliers also have a regular repeating structure and use short interconnections between modules. We chose a semisystolic bit serial multiplier over the array multiplier because the structure of semisystolic multipliers can be altered to process more bits at a time. An alternative is tree multipliers that would require large arrays to perform the same multiplication and the resulting circuit might have long critical paths.

We designed a hardware multiplier which performed multiplications up to 256 bits wide with a high-radix semisystolic multiplier variant. Our 256-bit multiplier, shown in **Figure 6,** requires only 8 cycles to multiply two 256-bit numbers and an additional 12 cycles to shift out the final result. The nature of the intermediate results of the 8 simultaneous 32-bit multiplications required a different connection of the carry signal and the shifting of the cumulative total. Two stages of addition were used with the carry-out of the first stage

connected to the carry-in of the second stage to the left, while the carry-out resulting from the second stage addition was fed back to the carry-in of the first stage.
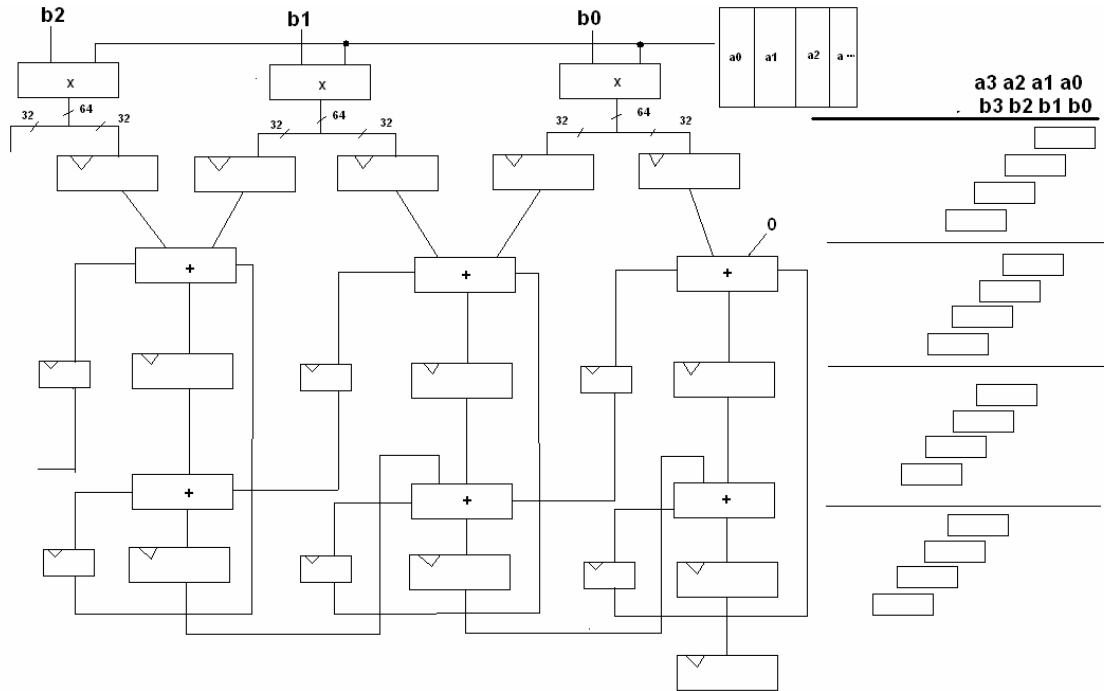
Figure 6: Block diagram of semisystolic high-radix multiplier.

For larger bitwidth multiplications (384 and 521 bits), we designed a larger divide-and-conquer multiplier shown in **Figure 7** that utilized the 256-bit multiplier with an extra 256-bit adder and an extra register for storing intermediate results.
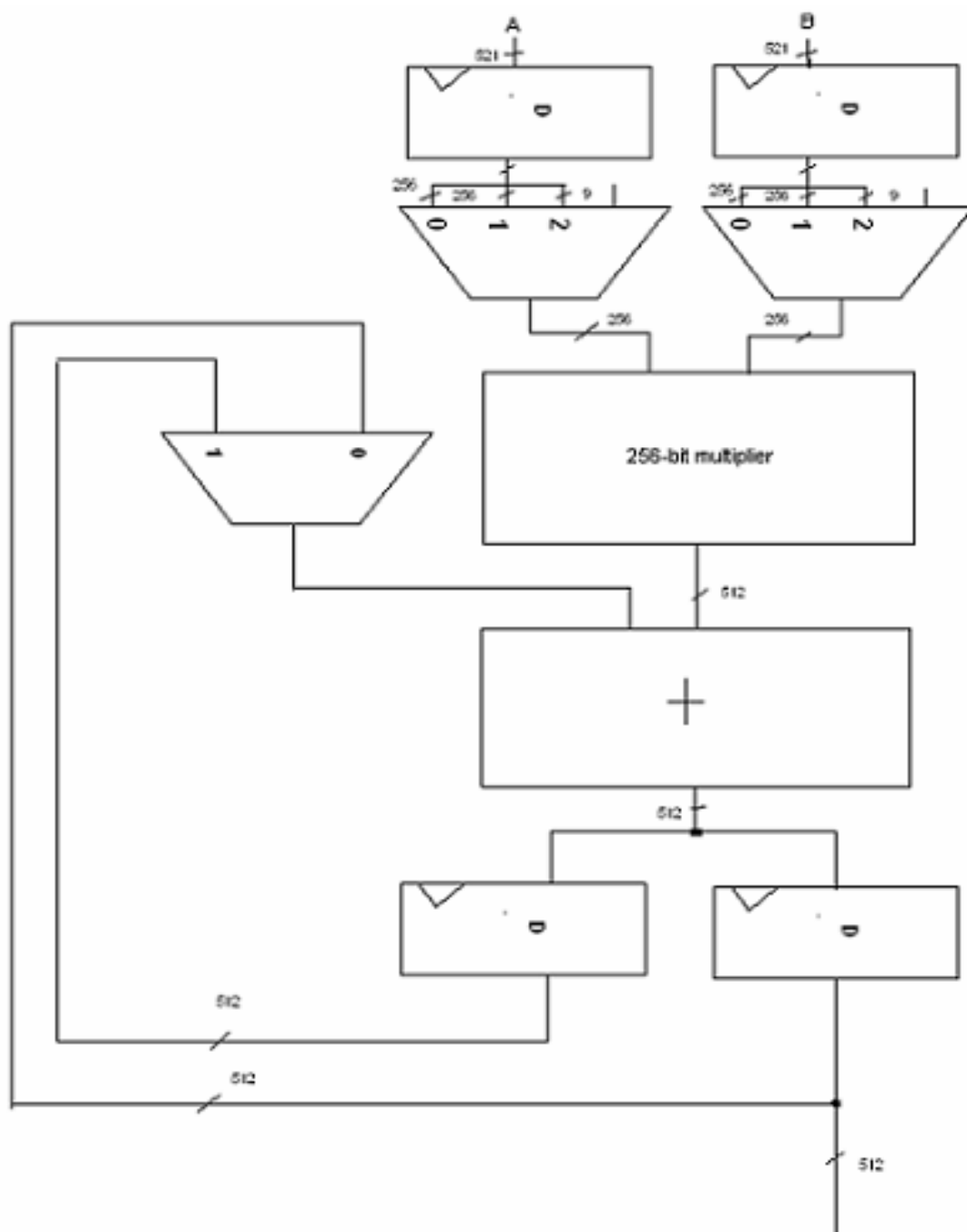
Figure 7: Divide-and-conquer multiplier.

For the 384-bit multiplication, the multiplier and multiplicand is divided into 2 parts, one 256 bits wide and the other 128 bits wide:

$$\{m_2,\ m_1,\ m_0\} = \{a_1,\ a_0\} \times \{b_1,\ b_0\},\ \text{where}$$

$$m_0 = (a_0b_0)_l,$$

$$m_1 = (a_0b_0)_h + (a_0b_1)_l + (a_1b_0)_l,$$

$$m_2 = (a_0b_1)_h + (a_1b_0)_h + (a_1b_1).$$

Subscript h is the higher 256-bit result of ($a_i \times b_i$), and l is the lower 256-bit result of the computation $a_i \times b_i$. This 384-bit multiplication requires four 256-bit multiplications and takes 127 clock cycles to complete and transfer the result across the 521-bit connection to the modular reduction circuit.

For 521-bit multiplication the input operands are split into 3 parts, each 256 bits wide:

$$\{m_4,\ m_3,\ m_2,\ m_1,\ m_0\} = \{a_2,\ a_1,\ a_0\} \times \{b_2,\ b_1,\ b_0\},\ \text{where}$$

$$m_0 = (a_0b_0)_l,$$

$$m_1 = (a_0b_0)_h + (a_1b_0)_l + (a_0b_1)_l,$$

$$m_2 = (a_1b_0)_h + (a_2b_0)_l + (a_0b_1)_h + (a_1b_1)_l + (a_0b_2)_l,$$

$$m_3 = (a_2b_0)_h + (a_1b_1)_h + (a_2b_1)_l + (a_0b_2)_h + (a_1b_2)_l,$$

$$m_4 = (a_2b_1)_h + (a_1b_2)_h + a_2b_2,$$

The a_2b_2 term represents the product of the most significant 9-bit parts. 521-bit multiplication requires nine 256 bit multiplications and takes 260 clock cycles to complete and transfer the result across the 521-bit connection to the modular reduction circuit.

4.4.4.2 Modular Reduction

The reduction circuit is a hardware implementation of the NIST modular reduction algorithm presented in Section 3.3.3. **Algorithm 19** presents the combined version of **Algorithm 11 – Algorithm 15.**

Algorithm 19: Combined NIST reduction algorithm.

INPUT: Integer $c = (c_{33}; \ldots ; c_2; c_1; c_0)$, where each c_i is a 32-bit word, p is selected prime

OUTPUT: c mod p.

1. ***Define*** *521-bit integers: sa_1, sa_2, sa_3 , sa_4 , sa_5, sa_6 , sa_7 , sa_8 , ss_1, ss_2 , ss_3 ,ss_4.*

2. ***Compute*** *$A = sa_1 + sa_2 + sa_3 + sa_4 + sa_5 + sa_6 + sa_7 + sa_8$.*

3. ***Compute*** *$B = ss_1 + ss_2 + ss_3 + ss_4$.*

4. ***Return*** *$A - B$ mod p.*

In **Algorithm 19** the *sa* terms are the s-terms from **Algorithm 11 – Algorithm 15** that must be added while the *ss* terms are the s-terms that must be subtracted. In order to reduce area of the hardware implementation, regular adders were used to perform the addition and performed a final reduction with a smaller reduction circuit similar to the one already being used to obtain the final result. Use of regular adders allowed the design a faster circuit since we could now perform addition in parallel using a tree structure for multi-operand addition.

For the various primes the number of *sa* terms to be added to yield *A* are 4, 3,7, 8 and 2 for p192, p224, p256, p384, p521 respectively, while for the *ss* terms are 0, 2, 4, 3 and 0

respectively. Thus, the largest number of different terms to be added to yield A is 8, while for B it is 4. This resulted in three and two stages of additions for the tree adder for A and B respectively. The final result for A is at most three bits greater than the prime size (addition of eight n-bit integers yields a $n + 3$ bit result), while for B it at most two bits greater than the prime size (addition of four n-bit integers yields a $n + 2$ bit result). The datapath of the final circuit was extended by three bits to accommodate the result before being reduced modulo the selected prime.

Our final circuit uses time multiplexing similar to the other circuits for operation greater than 256 bits. The 256-bit datapath is also split into eight 32-bit sections and an extra 12-bit section to support 521-bit operation, with the additional 3 bits to accommodate accumulated carries.

Figure 8 shows the 32-bit section of the reduction circuit used to add *sa* terms, while **Figure 9** shows the 32-bit sections used to add the *ss* terms before they are then subtracted.
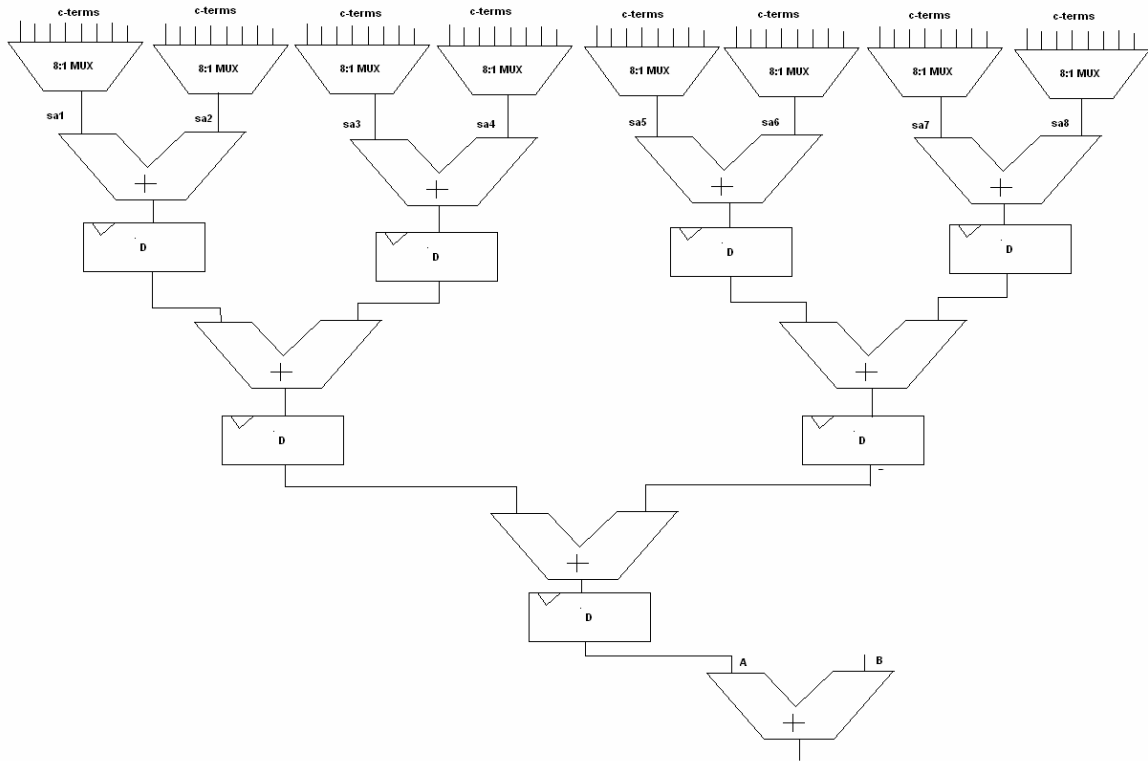
Figure 8: 32-bit section of modular reduction circuit (Part A).
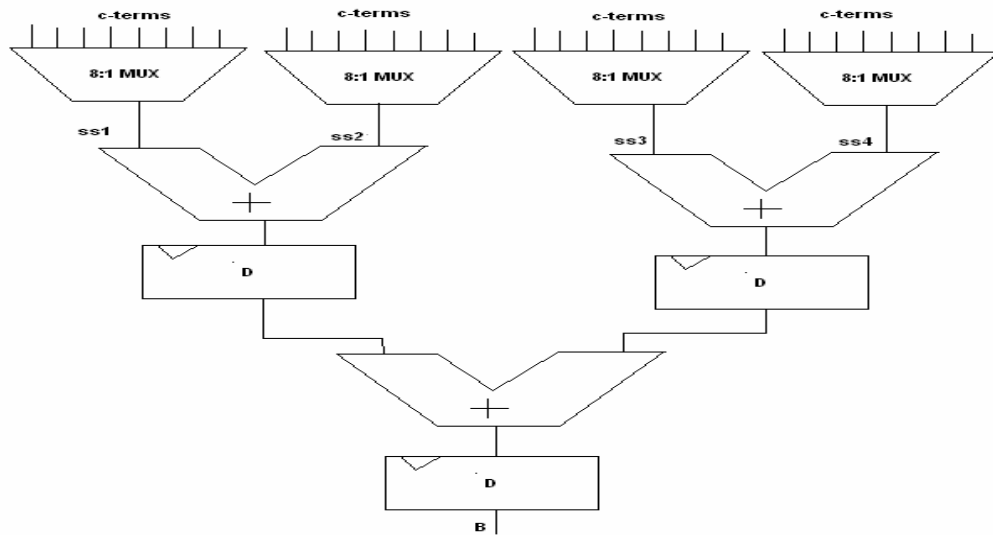
Figure 9: 32-bit section of modular reduction circuit (Part B).

An 8:1 multiplexer supplies the input to the adders. The inputs of the multiplexers are mapped to the corresponding 32-bit words as follows:

- Input 0: c_i of 192-bit c,

- Input 1: c_i of 224-bit c,

- Input 2: c_i of 256-bit c,

- Input 3: c_i of 384-bit c, first pass (LSP),

- Input 4: c_i of 521-bit c, first pass (LSP),

- Input 5: c_i of 384-bit c, second pass (MSP),

- Input 6: c_i of 521-bit c, second pass (MSP),

- Input 7: unconnected.

Symbol c_i is taken from **Algorithm 11 – Algorithm 15** to represent the 32-bit section of the s-term to be added for p192, p224 and p256 and pass 1 of p384, p521, or the second pass value for p384 and p521. Subscript i denotes the section being added.

The subtraction $(A - B)$ in step 4 of **Algorithm 19** could result in a negative result. We compute the equivalent positive result by adding 8p to the negative result to yield a positive number, which is at most 3-bits greater than the size of the prime. The result obtained after the sign adjustment (if the need arises) is then passed to a smaller reduction circuit similar to the reduction circuit designed above.

For example for p192 adding s_1, s_2, s_3, s_4, yields a value of A that is at most 2 bits greater than $2^{192} - 1$. A then consists of seven c-terms (c_6, c_5,..., c_0), while $c_8, c_9, \ldots, c_{11}$ are all equal to zero. This new value of A and B (computed the same way) is then used with

Algorithm 19 to reduce the result modulo p192. Modular reduction takes 21 clock cycle

for p192, p224, p256 and 22 cycles for p384 and p521.

4.5 Control Unit

4.5.1 Controller Organization

The control unit controls instruction execution by asserting control signals connected to

the functional unit. **Figure 10** shows the block diagram of the control unit datapath.
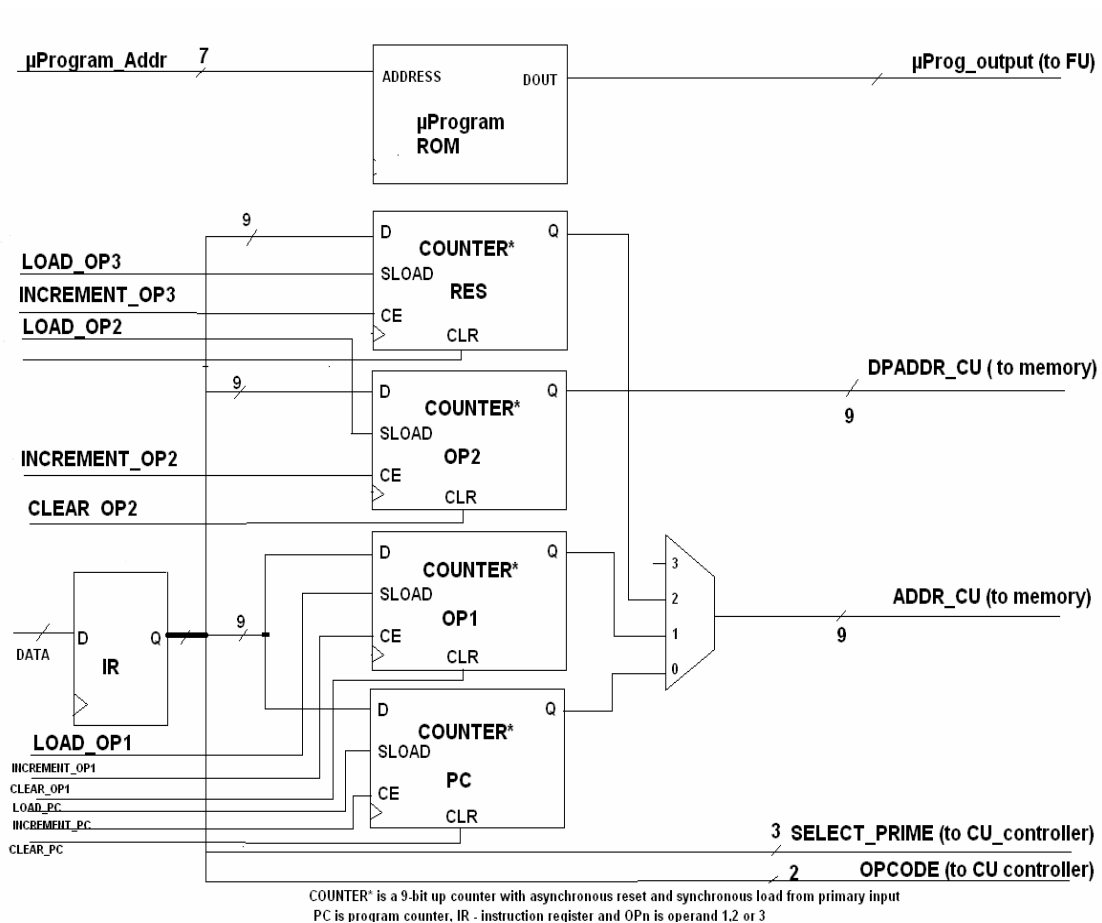
Figure 10: Datapath of control unit.

The instruction register (IR) holds 32-bit instruction fetched from main memory which is then split into four parts. The program counter (PC) controls the order of program execution. Counter OP1 and OP2 is used to hold the address of the both operands in a binary operation and initially contains the address of the least significant word of the operand and is incremented to retrieve successive words of the operands. RES counter hold the address of the result and is incremented to store successive words of the results. The controller also houses the microprogram which contains the microinstructions required to execute each instruction in the instruction set of the processor. The input signals to the control unit are:

- **GLOBAL START:** Asserted by the supervisor to start the operation of the processor

- **GLOBAL RESET:** Resets the processor.

- **SUPERVISOR:** Asserted by the processor to write instructions and data to the main memory. The control unit hands over access to the main memory to the supervisor.

- **SELECT_PRIME:** Prime in operation decoded from instruction. This signal is also connected to the functional unit as signal **PRIME** in **Figure 1** and **Figure 3**.

- **DONE:** Indicates to the controller that computation within the block in the functional unit has completed. This signal is the **DONE** in Figure 1 and Figure 3 coming from the functional unit

The output signals are:

- **BUSY:** Asserted while an instruction is being executed.

- **LOAD_OP1:** Load enable signal of the counter OP1 which stores the 9 bit address of the first operand.

- **LOAD_OP2:** Load enable signal of the counter OP2, which stores the 9 bit address of the second operand in a binary operation, or the least significant word (LSW) of the MSP of an inversion (unary) operation.

- **LOAD_OP3:** Load enable signal of the counter RES which stores the 9 bit address of the LSW of the result.

- **LOAD_PC:** Load enable signal of the program counter (when enabled allows the program counter to be incremented).

- **INCREMENT_OP1:** Increments counter OP1 each time a new word of the first operand is fetched from memory.

- **INCREMENT_OP2:** Increments counter OP2 each time a new word of the second operand is fetched from memory.

- **INCREMENT_OP3:** Increments the counter RES each time a new word of the result is stored in memory.

- **CLEAR_OP1/2/3:** Clears the count of the counter OP1, OP2 and RES respectively.

- **WE_CU:** Write enable signal for the control unit. It is one of the inputs to the 2-1 multiplexer that is used to select which write enable signal controls the write enable signal of the main memory. The other input is the write enable control signal from the supervisor.

- **µProg_FU:** This is the multi signal output of the microprogam ROM and are connected to the functional unit. These signals are used by the control unit to control the functional unit and registers. They are described in section 4.5.3.

4.5.2 Instruction Fetch-Decode-Execute Cycle

The fetch-decode-execute cycle of each instruction can be divided into three parts:

a) Instruction fetch: After the supervisor asserts the start signal, the controller within the control unit fetches the 32-bit instruction one at a time from the main memory into the instruction register. This operation takes two clock cycles.

b) Decode: The instruction is decoded in one clock cycle by transferring the different parts of the instruction as described in **Table 4** in the instruction register into the appropriate counters. For example the 9-bit Operand1 is transferred to counter OP1. The operands are then fetched from the main memory into the register, if the operand field is not equal to zero. If the operand field is equal to zero, then the data in the register (result of prior instruction execution) is used.

c) Execute: The execution time is dependent on the instruction and the prime. The controller within the control unit uses the opcode to determine which routine of the microprogram is used to control the execution of the instruction. After instruction execution the result is stored in the register. Depending on the value of the 'Result" field of the instruction, the data is either written to memory or left in the register (0 represents register while any other value is the address of the LSW of the location where the result is to be stored).

4.5.3 Microprogram

The microinstructions control data flow within the blocks in the functional unit during instruction execution. The output signals of the microprogram are:

- **SEL_mm/mas/inv:** Controls the tristate buffers at the output of each block within the functional unit. The signal is mutually exclusive as the asserted signal allows the result to be written to the register. The three signals are collectively called **INHALF** in **Figure 1** and **Figure 3.**

- **START_mm/mas/inv:** Starts execution of the instruction in the corresponding block. The three signals are collectively called **START** in **Figure 1** and **Figure 3.**

- **RESET_mm/mas/inv:** Resets the block prior to instruction execution. The three signals are collectively called **RESET** in **Figure 1** and **Figure 3.**

- **SELECT_OUT_mm/mas/inv:** selects each half of the result (384-bit and 521-bit primes only) when data is transferred to the register. The register can only receive 265 bit of data at a time. The three signals are collectively called **OUTHALF** in **Figure 1** and **Figure 3.**

- **ADD_SUB_mas:** Indicates to the modular adder/subtractor if a modular addition or subtraction is to be performed. This signal is called **ADD/SUB** in **Figure 1** and **Figure 3** for brevity**.**

4.6 Timing Summary

The number of cycles required to perform each operation within the processor for each of the NIST primes is presented in

Table *8*.

Table 8: Number of clock cycles required to perform operations.

Operation	p192	p224	p256	p384	p521
Modular Add/Sub.	11	11	11	13	13
Modular Multiplication.	56	56	56	148	285
Modular Inversion (Worst Case)	3072	3584	4096	6144	8336
Memory R/W (Operands and Result)	6	7	8	12	17
Fetch and Decode of Instructions	6	6	6	6	6

4.7 Strengths of the Proposed Architecture

The strengths of our proposed processor can be summarized as follows:

- It is flexible enough to handle any method for computing point multiplication $Q = kP$, which depends on the representation of k and P.

- It is programmable achieved by executing a sequence of instructions to perform point addition and doubling using different point representations.

- It can accelerate modular arithmetic over five GF(p) fields of different sizes, thus supporting five different levels of security.

- It allows for concurrent execution of instructions since the blocks within the functional units operate independently.

- It exploits locality by utilizing registers between the main memory and the functional unit thereby limiting data transfer from and to the main memory only to fetching of instructions, retrieving of operands, and storing of results (not used as operands for the next instruction).

- While the processor is executing its assigned set of programs, the supervisor can run the scheduler to generate the next set of programs. Thus, the latency overhead of the supervisor running the scheduler is hidden by useful computations performed by the processor in most cases.

4.8 Weaknesses of the Proposed Architecture

The current design of the proposed architecture has the following weaknesses:

- The controller supports only sequential instruction issue for the sake of simplicity. The processors datapath however, supports concurrent instruction execution. Implementing a controller that would exploit instruction-level parallelism is one of the proposed future extensions to this work.

- The processor has been designed with NIST primes in mind. This choice was deliberate: it offers flexibility in security levels, conforms with most ECC standards and takes advantage of fast reduction algorithms. Nevertheless, the processor can be adapted to support other primes, as only the reduction circuit is specific to NIST primes.

5. Implementation Results

5.1 Implementation Platform

In this chapter we describe an implementation of our ECC processor architecture using Xilinx FPGA and design tools.

5.1.1 Design Steps

First, separate blocks were designed for all the basic GF(p) operations, which eliminated the dependency of hardware on a particular choice of scalar and point representations. Each block was designed with its own controller, so that its operation was independent of the others. Then all blocks were put together to create a complete processor with a global controller for sequencing processor instructions.

Second, the blocks were described using Verilog HDL. In order to verify functional correctness of the blocks, the output of various test vectors for each of the individual primes were compared to the output of an existing software implementation of the modular operations [37]. We tested the point multiplication in both Affine and Jacobian Projective coordinates using appropriate instruction sequences.

Third, after each block was synthesized, it was placed and routed. After successful placing and routing, the simulation results from the behavioral simulation were compared against post-place and route simulation to ensure that our circuit still functioned correctly. This step was repeated for each block in the processor and then the entire architecture was put together and synthesized, placed and routed, and verified.

Finally, we obtained accurate post place-and-route data and performance estimates for our processor. Also we obtained estimates for the running time of the processor performing point representation in Affine and Projective coordinates.

5.1.2 Design Tools

The entire processor was coded in Verilog within the Xilinx Project Navigator development environment using IEEE 1364-2001 Verilog standard to allow for code portability. Test benches were generated using Xilinx Testbench Waveform tool while behavioral simulation was performed using Mentor Graphics' ModelSim XE III tool. Functional verification was performed by comparing simulation results with those of an existing software implementation.

Synthesis was performed using Xilinx XST. The generated netlist generated was placed and routed using Xilinx PAR tool. Finally post-place-and-route simulation was carried out using Mentor Graphics' ModelSim XE III tool, and the output was compared to the result obtained from behavioral simulations.

5.2 Prototype Performance

Our implementation uses 32 18x18 hardware multipliers, occupies 22,526 slices, and runs at 40.33 MHz. **Table 9** shows the area and maximum clock frequency of the various blocks within the processor obtained from the post place and report for each block. Each block was placed and routed separately.

Table 9: Implementation results.

Block	Slices	Max. Clock (MHz)
Modular Adder/Subtractor	3,193	40.87
256-bit Multiplier	1,355	59.97
Modular Reductor	8,420	40.43
Entire Modular Multiplier	12,537	40.47
Inverter	10,169	41.05

Figure 11 shows the slice distribution of the various blocks within the functional unit. The modular multiplier is the largest circuit occupying 4 times more area than the modular adder/subtractor.

12%

39%

49%

☐ Modular Adder/Subtractor
■ Modular Multiplier
☐ Inverter
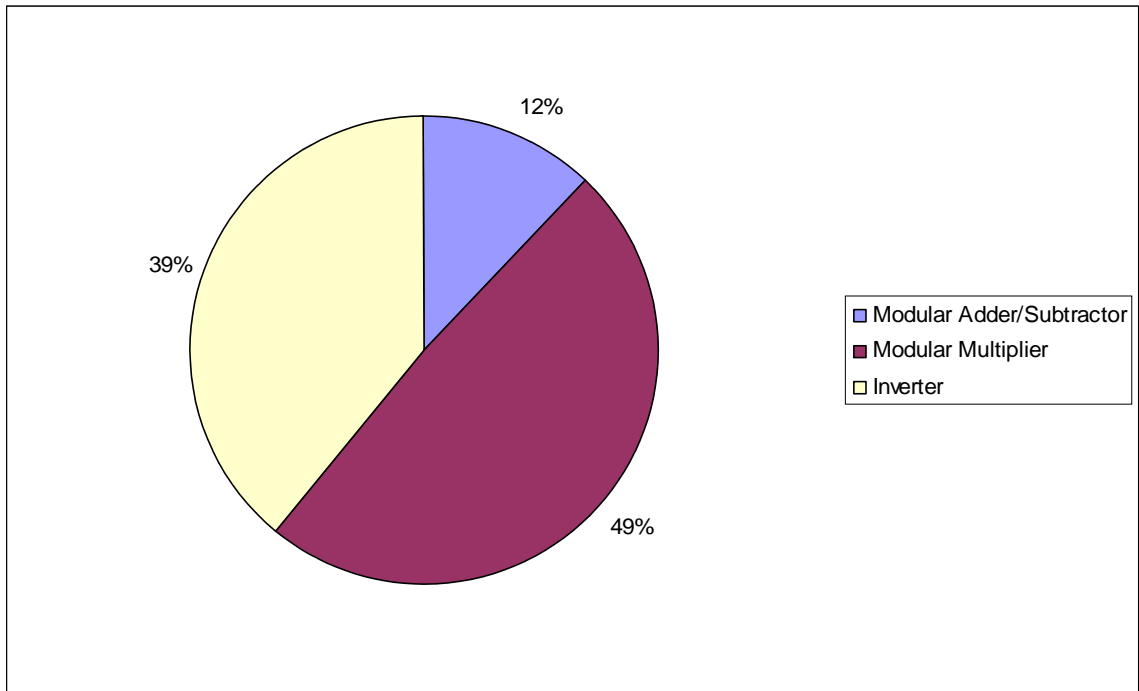
Figure 11: Slice distribution of blocks within functional unit.

Table 10 summarized the area, maximum clock frequency, and implementation medium of other processors reported in literature. We give this data before commencing comparison of each implementation with our proposed processor.

Table 10: Implementation details of various processors reported in literature.

	Clock	Area(Slices)	Prime Size	Medium
Ours	40.0	22526	521	Virtex II 2VP 100
Paar [29]	40.0	11416	192	XCV 1000E
Ors [28]	91.3	*	160	XCV 1000E
Satoh [31]	137.7	*	256	0.13 μm CMOS
McIvor [32]	45.0	*	256	XC2VP125-7-ff1696
Shuhua [30]	50.0	*	192	XC2V 1000-5-BG256
Buell [26]	100.0	33792	256	XC2V6000
Wolkerstorfer [25]	*	*	256	0.35 μm CMOS
Daly [24]	31.92	3109	256	XC2V2000-6

* Data not available

5.2.1 Performance of Modular Operations

Table 11 shows the timing of each operation in microseconds when the processor is run at 40 MHz.

Table 11: Timing (in μs) of modular operations at 40 MHz clock.

	p192	p224	p256	p384	p521
Modular Add/Sub.	0.25	0.25	0.25	0.27	0.27
Modular Mult.	1.40	1.40	1.40	3.70	7.12
Modular Inversion	76.80	89.60	102.40	153.60	208.40

Our architecture takes advantage of the fast NIST reduction algorithms. Our reduction circuit uses regular adders followed by another small reduction circuit. An alternative implementation of the NIST reduction algorithm would use just modular adders without an extra reductor. These two implementation choices are compared in **Table 12** in terms of area and latency.

Table 12: Comparison of implementations of modular reduction.

Modular Reduction Implementation	Area (Slices)	Timing in μs					Max Clock Freq (MHz)
		P192	p224	p256	p384	p521	
Ours	8420	0.47	0.52	0.56	0.61	0.67	40.43
Modular Adder based	3193	1.07	1.39	3.54	4.23	0.68	40.87
Our Speedup Factor		2.31	2.71	6.30	6.92	1.03	

Note that our proposed modular reduction implementation is faster than the other implementation for all primes. The speedup is more predominant for primes requiring more input permutations and modular addition/subtractions. The speedup, however, comes at the expense of 2.6 times more area.

In order to take into consideration the effect of the area used by both implementation we compare the Area-Time efficiency (AT^2) [36]. **Table 13** presents the Area-Time efficiency for both implementations, where A is the area in slices, and T is the time of execution in μs. The results show that it is efficient to use our implementation for all primes except p521, which required only 1 modular addition and a few input permutations.

Table 13: Comparison of area-time efficiency of modular reduction.

Modular Reduction Implementation	Area-Time Efficiency (AT^2)				
	p192	p224	p256	p384	p521
Ours	1860	2272	2725	3220	3755
Modular Adder based	3701	6211	40191	57211	1499
Our Efficiency Factor	1.99	2.73	14.75	17.77	0.40

Table 14 gives the results of the comparison of the time to perform modular multiplication using existing processors listed in **Table 10**.

Table 14: Comparison of modular multiplication delays.

	Maximum Clock Frequency (MHz)	Prime Size	Clock Cycles	Time (µs)	Our Speedup Factor
Ors [28]	91.31	192*	580	6.35*	4.54
Satoh [31]	137.7	192	995	7.22	5.16
Satoh [31]	137.7	224	1,389	10.08	7.21
Satoh [31]	137.7	256	1,549	11.25	8.04
McIvor [32]	45.0	256	32	0.71	0.51

* 160-bit ECC processor, but formulas for estimating delays of all operations for higher bitwidth processors were providedin [28] and used here.

The results in **Table 14** indicate that our implementation is faster than all modular multipliers except for the one by McIvor [32]. Thier multiplier, however, ignores the cost of precomputations associated with Montgomery multiplication, which is considerable.

For example, in the Satoh implementation [31] the cost is 950, 1323 and 1483 cycles for 192, 224 and 256 bit multiplications.

In **Table 15** we compare the delay of our modular inverter with that of extended Euclidean algorithm (EEA) based inverters and other implementations using Montgomery Multiplication and Fermat's theorem. The Wolkerstorfer implementation (EEA based) [25] does not mention the maximum clock frequency, so we used our maximum clock frequency to compute the delay.

Table 15: Comparison of modular inversion delays.

	Clock	Prime Size	Clock Cycles	Time (µs)	Our Speedup Factor
Ors [28]	91.3	192	167,040	1.83	23.82
Wolkerstorfer [25]	40.0	192	14,000	0.35	4.56
Satoh [31]	137.7	192	379,637	2.76	35.89
Satoh [31]	137.7	224	618,734	4.49	50.15
Wolkerstorfer [25]	40.0	224	16,500	0.41	4.60
Satoh [31]	137.7	256	788,016	5.72	55.89
McIvor [32]	45.0	256	16,256	0.36	3.52
Wolkerstorfer [25]	40.0	256	19,400	0.49	4.73

Table 15 shows that our EEA based modular inverter is 4.5 times faster than another EEA based inverter [25] and up to 55 times faster than a Fermat's based Montgomery modular inverter [31].

5.2.2 Performance of Point Operations

In **Table 16** we present the worst case performance of point addition and point doubling using Affine and Jacobian Projective coordinates (point doubling in Jacobian coordinates and point addition in Affine-Jacobian coordinates). We include the time it takes to fetch each instruction and its operand from memory and the time it takes writes the result back to memory without using of any of the registers in its operation.

Table 16: Estimated point addition and doubling timing (in µs) at 40 MHz clock.

Point Representation	Point Operation	p192	p224	p256	p384	p521
Affine Coordinates	Doubling	22.95	23.85	24.75	49.50	84.83
Affine Coordinates	Addition	25.25	26.15	27.05	56.30	98.48
Jacobian Projective	Doubling	90.25	103.70	117.15	180.55	252.30
Jacobian Projective	Addition	87.00	100.30	113.60	174.00	241.58

To compare the worst case performance of our proposed implementation we employ unknown point multiplication method, whose timing is computed by adding delays of half-bitwidth point additions and full-bitwidth point doublings. By half- and full-bitwidth we mean the number of non-zero bits in the binary representation of scalar k.

Table 17 shows our delay estimates for point multiplications in Affine coordinates and Jacobian Projective coordinates (see Section 4.3.4), using the worst case delays, without use of the register for point addition and point doubling from **Table 16**.

Table 17: Estimated point multiplication timing (in ms) with 40 MHz clock.

Coordinates	p192	p224	p256	p384	p521
Affine	25.68	34.46	44.53	102.74	194.38
Jacobian Projective	6.83	8.27	9.79	29.81	69.84

In **Table 18** we compare the performance of Unknown point multiplication of our proposed processor given in **Table 17** with other implementations reported in literature.

Table 18: Comparison of performance of our proposed implementation with existing implementations.

Reference	Prime Size	Clock Frequency	Point Mult. (ms)	Normalized Point Mult. @ 40MHz (ms)	Point Mult. Ours (ms)	Our Speedup Factor
Ors [28]	160	91.3	14.41	32.89	6.83	4.82
Paar [29]	192	40	3	3.00	6.83	0.44
Satoh [31]	192	137.7	1.44	4.95	6.83	0.73
Shuhua [30]	192	50	6	7.50	6.83	1.1
McIvor [32]	256	45	4.8	5.40	9.79	0.55
Buell [26]	256	100	9.43	23.58	9.79	2.41
Wolkerstorfer [25]	256	40	28.75	28.75	9.79	2.94
Daly [24]	256	50	17.58	21.98	9.79	2.24

Not surprising, our FPGA implementation is much slower than the ASIC processor from [31]. Much of the penalty comes from the FPGA fabric itself, e.g., the Xilinx synthesis tool used 18-bit hardware multipliers with an additional logic to perform 32-bit multiplications. A custom implementation of our architecture proposed as future work with 32-bit multipliers will likely to have better performance figures. Nevertheless, our implementation outperforms many other FPGA-based processors, despite its inherent

overhead needed to support different primes. Note that our estimates include the time to read and write data from memory which gives the worst case computation and these operations take up between 20-40% of the computation time. This could be greatly reduced with the use of the registers and optimized code.

6. Conclusion and Future Work

6.1 Conclusion

In this thesis we presented the design of an elliptic curve cryptographic processor capable of low-level ECC operations over NIST-recommended prime fields. Our implementation exhibits good efficiency despite the overhead needed to support datapath reconfigurations for different prime sizes. The proposed architecture is the first of its kind to support all NIST prime fields. This flexibility is achieved by changing only 3 bits of the instruction of the processor and can be used to perform higher level ECC operations using any point representation or scalar. Our architecture has been prototyped on a Xilinx Virtex-II Pro FPGA, occupying 22526 slices and running at 40.33 MHz

The main contributions in this thesis are:

- Efficient processor architecture that supports all five NIST primes with sizes ranging from 192 to 521 bits.

- Programmable processor architecture whose instructions set allows a user to automatically perform a sequence of modular operations. It is flexible in performing higher level operations (similar to a full software implementation), while taking advantage of concurrent execution (similar to a hardware implementation).

- Hardware implementation of the fast reduction algorithms for NIST recommended primes, featuring a single hardware unit for all five dissimilar reduction algorithms and using regular adders (instead of larger modular adders) for intermediate computation.

- Efficient modular inversion implementation using only positive integers.

6.2 Future Work

Possible future research that can be carried out to extend this work is as follows:

- Enabling concurrent instruction execution to take advantage of the fact that each unit within the functional unit can operate independently of the others.

- Designing the compiler for generating processor instructions.

- Custom implementation of the processor which would have better performance than the FPGA prototype.

- Improving resilience to implementation attacks [17].

Bibliography

[1]   N. Kobiltz, "Elliptic curve cryptosystems", Mathematics of Computation, 48, 1987, pp. 203-209.

[2]   V. Miller, "Uses of elliptic curves in cryptography", *Advances in Cryptology '85*, LNCS 218, 1986, pp. 417-426.

[3]   IEEE P1363, Standard Specifications for Public Key Cryptography.

[4]   ANSI X 9.62, *Public Key cryptography for the Financial Services Industry: Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.

[5]   National Institute of Standards and Technology, *Digital Signature Standard*, FIPS Publication 186-2, 2000.

[6]   ISO/IEC 14888-3, *Information Technology – Security Techniques – Digital Signatures with Appendix – Part 3: Certificate Based-Mechanisms*, 1998.

[7]   W. Trappe and L. Washington, "*Introduction to Cryptography with Coding Theory*", Prentice Hall, New Jersey, 2002.

[8]   "The Elliptic curve cryptosystem for smart cards", ECC White Papers, http://www.certicom.com.

[9]   "How to design an API", http://openide.netbeans.org/tutorial/api-design.html

[10]   "Certicom Security for the RIM BlackBerry", ECC White Papers, http://www.certicom.com.

[11]   "US check 21 frequently asked questions" http://www.federalreserve.gov/paymentsystems/truncation/faqs2.htm.

[12]   "ECC in action: real-world applications of elliptic curve cryptography", ECC White papers, http://www.certicom.com.

[13]   J. Goodman and A. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor, *IEEE Journal on Solid-State Circuits*, vol. 36, no. 11, pp. 1808-1820, Nov. 2001.

[14]   P. van Oorschot, A. Menezes, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press Inc., Florida , 1996.

[15]   I. Blake, G. Seroussi and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, New York, 1999.

[16]   "ECC tutorial," http://www.certicom.com.

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

[18]   H. Cohen, A. Miyaji and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates", *Advances in Cryptology '98*, LNCS 1514, 1998, 51-65.

[19]   D. Johnson, A. Menezes "The Elliptic Curve Digital Signature Algorithm (ECDSA)", Technical Report CORR 99-34, Dept of C& O, University of Waterloo, 1999.

[20]   M. Brown, D. Hankerson, J. Lopez, and A. Menezes, "Software implementation of the NIST elliptic curves over prime fields", http://www.cacr.math.uwaterloo.ca.

[21]   J. Solinas, "Generalized Mersenne numbers", Technical Report CORR 99-39, Dept. of C & O, University of Waterloo, 1999.

[22]   J. Groszschaedl and E. Savas, "Instruction set extensions for fast arithmetic in finite field GF(p) and GF(2^m)", *Cryptographic Hardware and Embedded Systems*, 2004, pp. 133-147.

[23]   H. Eberle et al., "Accelerating next-generation public-key cryptosystems on general-purpose CPUs", *IEEE Micro*, pp. 52-59, Mar. 2005.

[24]   A. Daly, W. Marnane, T. Kerins and E. Popovici, "An FPGA implementation of a GF(p) ALU for encryption processors", *Elsevier Microprocessors and Microsystems 28*, 2004, pp. 252-260.

[25] J. Wolkerstorfer, "Dual-field arithmetic unit for GF(p) and GF(2m)," *Cryptographic Hardware and Embedded Systems*, 2002, pp. 500-514.

[26] G. Quan, D. A. Buell, J. P. Davis and S. Devarkal, "High-level synthesis for large bit-width multipliers on FPGAs: a case study", *International Conference on Hardware/Software Codesign and System Synthesis,* 2005, pp. 213-218.

[27] A. Gutub et al., "Pipelining GF(p) elliptic curve cryptography computation," *Cryptographic Hardware and Embedded Systems*, 2006, pp. 93- 99.

[28] S. B. Ors, L. Batina, B. Preneel and J. Vandewalle, "Hardware implementation of an elliptic curve processor over GF(p)", *IEEE. Application-Specific Systems, Architecture and Processors*, 2003

[29] G. Orlando and C. Paar, "A scalable GF(p) elliptic curve processor architecture for programmable hardware", *Cryptographic Hardware and Embedded Systems,* LNCS 2162, 2001, pp. 356-371.

[30] W. Shuhua and Z. Yuefei, "A timing-and-area tradeoff GF(p) elliptic curve processor architecture for FPGA", *International Conference on Communications, Circuits and Systems*, 2005, pp. 1308-1312.

[31] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Trans. on Computers*, April 2003, pp. 449-460.

[32] C. McIvor, M. McLoone, and J. McCanny, "FPGA Montgomery modular architecture suitable for ECCs over GF(p)", *IEEE International Conference on Circuits and Systems,* May 2004.

[33] C.K. Koc, T. Acar, B.S. Kaliski, "Analysing and Comparing Montgomery Multiplication Algorithms", *IEEE Micro*, Vol. 16, No. 3, pp 26-33, June 1996.

[34] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519-521, Mar. 1985.

[35] A. Bossaelaers, R. Govaerts and J. Vandewalle, "Comparison of three modular reduction functions", *Advances in Cryptology,* LNCS 773, 175-186.

[36] B. Parhami, *Computer Arithmetic: Algorithm and Hardware Designs*, Oxford University Press*,* New York, 2000.

[37]   M. Khabbazian, "Software Elliptic Curve Cryptography", Masters Thesis, Dept. of Electrical and Computer Engineering, University of Victoria, 2004.

Vita

Surname: Ananyi   Given Names: Kendall

Place of Birth: Lagos, Nigeria

Educational Institutions Attended:

University of Victoria   2004 - 2006

University of Benin, Edo State, Nigeria   1998 - 2003

Degrees Awarded:

B.Eng.   University of Benin, Edo State, Nigeria   2002

Honors and Awards:

University of Victoria Fellowship   2004 - 2005

Chevron-Texaco National University Scholarship Scheme   1997 – 2002

Dean's Award,1st year, Dept. of Electrical & Elect. Eng. U of Benin 1998

Best Student, High School Certificate Examination (S.S.C.E)   1996

Publication:

K. Ananyi and D. Rakhmatov, "Reconfigurable Processor for NIST Prime Field ECC," *IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM-2006; Page(s): -- (to appear), April 2006.

<u>UNIVERSITY OF VICTORIA PARTIAL COPYRIGHT LICENSE</u>

Title of Thesis/Dissertation:

Design of a Reconfigurable Processor for Elliptic Curve Cryptography over NIST Prime Fields

Author:

Kendall Ananyi

Signed:August 28, 2006