High Performance Elliptic Curve Cryptographic Co-processor

by

Jonathan Lutz

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2003

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In FIPS 186-2, NIST recommends several finite fields to be used in the elliptic curve digital signature algorithm (ECDSA). Of the ten recommended finite fields, five are binary extension fields with degrees ranging from 163 to 571. The fundamental building block of the ECDSA, like any ECC based protocol, is elliptic curve scalar multiplication. This operation is also the most computationally intensive. In many situations it may be desirable to accelerate the elliptic curve scalar multiplication with specialized hardware.

In this thesis a high performance elliptic curve processor is developed which is optimized for the NIST binary fields. The architecture is built from the bottom up starting with the field arithmetic units. The architecture uses a field multiplier capable of performing a field multiplication over the extension field with degree 163 in 0.060 μsec. Architectures for squaring and inversion are also presented. The co-processor uses López and Dahab's projective coordinate system and is optimized specifically for Koblitz curves. A prototype of the processor has been implemented for the binary extension field with degree 163 on a Xilinx XCV2000E FPGA. The prototype runs at 66 MHz and performs an elliptic curve scalar multiplication in 0.233 msec on a generic curve and 0.075 msec on a Koblitz curve.

Acknowledgements

This thesis is sponsored in part by Motorola, Inc. I am particularly grateful to Dan Cronin, Jim Dworkin, and Jeff LaVell of Motorola for their continued support throughout the course of this research effort. Additionally, special thanks is due Professor Anwarul Hasan for his time, guidance, and encouragement. And to my colleagues and friends, Amir and Arash, for the many coffee breaks at Tim Hortons.

To my wife and best friend,

Sarah Joy

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CLB	Configurable Lobic Block
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
FPGA	Field Programmable Gate Array
GF	Galois Field
IOB	Input/Output Block
NAF	Non-adjacent Form
NIST	National Institute of Standards in Technology
τ-NAF	τ-adic Non-adjacent Form
SSL	Secure Socket Layer

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The use of elliptic curves in cryptographic applications was first proposed independently in [17] and [24]. Since then several algorithms have been developed whose strength relies on the difficulty of the discrete logarithm problem over a group of elliptic curve points. Prominent examples include the Elliptic Curve Digital Signature Algorithm (ECDSA) [25], EC El-Gammal and EC Diffie Hellman [14]. In each case the underlying cryptographic primitive is elliptic curve *scalar* multiplication. This operation is by far the most computationally intensive step in each algorithm. In applications where many clients authenticate to a single server (such as a server supporting SSL [8, 27] or WTLS [1]), the computation of the scalar multiplication becomes the bottle neck which limits throughput. In a scenario such as this it may be desirable to accelerate the elliptic curve scalar multiplication with specialized hardware. In doing so, the scalar multiplications are completed more quickly and the

computational burden on the server's main processor is reduced.

Elliptic curve-based cryptosystems are most closely related to algorithms like the Digital Signature Algorithm (DSA) which are based on the discrete logarithm problem. In the DSA, the parameters can be chosen to provide efficient implementations of the algorithm. In the same way, the parameters of ECC based cryptosystems can be selected to optimize the efficiency of the implementation. Unfortunately, the selection of the ECC parameters is not a trivial process and, if chosen incorrectly, may lead to an insecure system. In response to this issue NIST recommends ten finite fields, five of which are binary fields, for use in the ECDSA [25]. For each field a specific curve, along with a method for generating a pseudo-random curve, are supplied. These curves have been intentionally selected for both cryptographic strength and efficient implementation.

Such a recommendation has significant implications on design choices made while implementing elliptic curve cryptographic functions. In standardizing specific fields for use in elliptic curve cryptography (ECC), NIST allows ECC implementations to be heavily optimized for curves over a single finite field. As a result, performance of the algorithm can be maximized and resource utilization, whether it be in code size for software or logic gates for hardware, can be minimized.

1.2 Scope of the Work and Objectives

Presented in this thesis are hardware architectures for multiplication, squaring and inversion over binary finite fields. Each of these architectures is optimized for a specific finite field with the intent that it might be implemented for any of the five NIST

recommended binary curves. These finite field arithmetic units are then integrated together along with control logic to create an elliptic curve cryptographic co-processor capable of computing the scalar multiple of an elliptic curve point. While the co-processor supports all curves over a single binary field, it is optimized for the special Koblitz curves [18].

To demonstrate the feasibility and efficiency of both the finite field arithmetic units and the elliptic curve cryptographic co-processor, the latter has been implemented in hardware using a field programmable gate array (FPGA). The design was synthesized, timed and then demonstrated on a physical board holding an FPGA.

The objectives of the work presented in this thesis are twofold: First to develop a high performance hardware finite field arithmetic units with low resource requirements. Second to integrate the arithmetic units into an efficient hardware elliptic curve scalar multiplier.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 gives an overview of the basic mathematical concepts used in elliptic curve cryptography. This chapter also provides an introduction to the hardware/software system used to implement the elliptic curve scalar multiplier. Chapter 3 presents efficient hardware architectures for finite field multiplication and squaring. A method for high speed inversion is also discussed. In Chapter 4 a hardware architecture of an elliptic curve scalar multiplier is presented. This architecture uses the multiplication, squaring and inversion methods discussed in Chapter 3. Finally Chapter 5 provides concluding remarks and a summary of the

research contributions documented in this thesis.

Chapter 2

Background

The fundamental building block for any elliptic curve-based cryptosystem is elliptic curve scalar multiplication. It is this operation that will be implemented. Provided in this chapter is an overview of the mathematics behind elliptic curve scalar multiplication as well as an introduction to FPGA technology which will be used in the implementation. The chapter is organized as follows: An introduction to concepts in abstract algebra including groups and fields. Next is given an overview of arithmetic over binary finite fields followed by a discussion of arithmetic over elliptic curve groups. The chapter concludes with a brief description of the implementation media used to prototype the elliptic curve scalar multiplier.

2.1 Mathematical Background

Elliptic curve cryptography is built on two underlying algebraic structures. They are finite groups and finite fields. This first section provides an introduction to these

concepts. The definitions and theorems have been gathered from [9], [23] and [29] and are given without proof. These texts as well as [4] and [21] provide further discussion of the mathematics behind elliptic curve cryptography.

2.1.1 Groups

Definition 1. Let G be a set. A *binary operation* on G is a function that assigns each ordered pair of elements in G an element in G.

Definition 2. An *algebraic group* $(G, *)$ is defined by a nonempty set G and a binary operation $*$. $(G, *)$ is said to be a group if the following properties hold:

- Closure: For all elements $a, b \in G$, element $(a * b) \in G$.
- Associativity: For all elements $a, b, c \in G$, $(a * b) * c = a * (b * c)$.
- Identity: There exists an element $e \in G$ such that for any element $a \in G$ $a * e = e * a = a$. The element e is referred to as the group identity.
- Inverse: For every element $a \in G$, the inverse $b = a^{-1}$ is also an element of G. Then $a * b = b * a = e$.

Definition 3. If for all elements $a, b \in G$, $a * b = b * a$, then G is a *commutative* or *Abelian group*.

Theorem 1. There is a single identity in every group G.

Example: The integers form a group under addition. The group $(\mathbb{Z}, +)$ possesses the properties listed in Definition 2 and has the identity $e = 0$.

Example: The set of non-zero integers under multiplication does not form a group. $(\mathbb{Z}^*, \cdot)$ possesses all the properties of a group except one. Elements 1 and -1 are the only elements whose multiplicative inverse is also in $\mathbb{Z}^*$. Element 2, for example, has inverse $1/2 \notin \mathbb{Z}^*$.

Definition 4. The *order* of G, denoted as $|G|$, is the number of elements in the set G.

Definition 5. The *order* of element $g \in G$, denoted as $|g|$, is defined to be the smallest positive integer t such that $g^t = e$.

Definition 6. Element $g \in G$ is said to be a generator of G if every element in G can be expressed by g^i for some integer i. Then $|g| = |G|$.

Example: Consider the group defined by the set $G = \mathbb{Z}_5^* = \{1, 2, 3, 4\}$ under multiplication. Then the order of the group is $|G| = 4$. Since

$$2^0 \mod 5 = 1$$

$$2^1 \mod 5 = 2$$

$$2^2 \mod 5 = 4$$

$$2^3 \mod 5 = 3$$

$$2^4 \mod 5 = 1$$

the order of element 2 is 4. And since

$$4^0 \mod 5 = 1$$

$$4^1 \mod 5 = 4$$

$$4^2 \mod 5 = 1$$

the order of element 4 is 2. Note that element 2 is a generator of the group but 4 is not.

2.1.2 Finite Fields

A finite field can be considered as a finite set whose elements form a group under two binary operations; usually multiplication and addition. More specifically,

Definition 7. $(F, +, \cdot)$ is a field if the following properties hold:

- The elements of F form a group under addition.
- The non-zero elements of F form a group under multiplication.
- The addition and multiplication operations are commutative, i.e. $a + b = b + a$ and $ab = ba$ for all $a, b \in F$.
- The multiplication operation can be distributed through the addition operation, i.e. $a(b + c) = ab + ac$ for all $a, b, c \in F$.

Definition 8. A field F with a finite number of elements is a *finite field*.

Definition 9. The *order* of a field F is the number of elements in F.

Definition 10. A generator of the non-zero elements of a finite field F is said to be a *primitive element* or *generator* of F.

Definition 11. The *characteristic* of a finite field is the smallest positive integer j such that

$$\underbrace{1+1+\cdots+1}_{j \text{ times}} \equiv 0.$$

Example: Consider the field GF(7) containing the elements $0, 1, 2, 3, 4, 5$ and 6. The order of the field is 7 and the characteristic is also 7 since

$$\underbrace{1+1+1+1+1+1+1}_{7 \text{ times}} \equiv 0 \pmod 7.$$

Element 3 generates GF(7) as shown below.

$$\begin{aligned} 3^0 &= 1 \pmod 7 & 3^4 &\equiv 4 \pmod 7 \\ 3^1 &= 3 \pmod 7 & 3^5 &\equiv 5 \pmod 7 \\ 3^2 &= 2 \pmod 7 & 3^6 &\equiv 1 \pmod 7 \\ 3^3 &= 6 \pmod 7 \end{aligned}$$

Definition 12. A unique[1] finite field exists for every prime-power order. These fields are denoted GF(p^m) where p is prime and m is a positive integer.

In cryptographic applications, two types of fields are commonly used. They are

- Prime Fields: GF(p) where p is large
- Binary Fields: GF(2^m) where m is large

[1]Unique in the sense that all fields of a specific prime-power order are isomorphic.

The architectures described in the following chapters perform arithmetic over binary finite fields. Attention will be focused exclusively on this specific case for the duration of the this thesis.

Element Representation: The binary field GF(2^m) contains 2^m elements. Precisely how each element is represented is defined by the basis being used. The two most common representations are *polynomial* basis and *normal* basis. The work discussed in this thesis uses polynomial basis.

Let GF(2)[x] denote the set of polynomials over GF(2). Then for any irreducible polynomial

$$F(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_2x^2 + f_1x + 1$$

with $f_i \in$GF(2), GF(2)[x]/$F(x)$ is a finite field with 2^m elements [23]. Since the field of order 2^m is unique up to isomorphism, the elements of the binary field GF(2^m) can be uniquely represented by the set of polynomials over GF(2) of degree less than m. Furthermore, field addition is performed by adding two such polynomials over GF(2). Field multiplication is performed by straightforward multiplication of two polynomials and reducing mod $F(x)$. The irreducible polynomial $F(x)$ is often referred to as the *reduction polynomial* or *field polynomial*.

Example: Consider the field GF(2^3) with the irreducible polynomial $F(x) = x^3 + x + 1$. The elements of the field are contained in the set

$$\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$$

The element $x+1$ generates GF(2^3) as shown below.

$$(x+1)^0 \equiv 1 \pmod{F(x)}$$
$$(x+1)^1 \equiv x+1 \pmod{F(x)}$$
$$(x+1)^2 \equiv x^2+1 \pmod{F(x)}$$
$$(x+1)^3 \equiv x^2 \pmod{F(x)}$$
$$(x+1)^4 \equiv x^2+x+1 \pmod{F(x)}$$
$$(x+1)^5 \equiv x \pmod{F(x)}$$
$$(x+1)^6 \equiv x^2+x \pmod{F(x)}$$
$$(x+1)^7 \equiv 1 \pmod{F(x)}$$

The characteristic of the field is two since

$$1+1 \equiv 0 \pmod{2}.$$

NIST recommends the fields GF(2^{163}), GF(2^{233}), GF(2^{283}), GF(2^{409}) and GF(2^{571}) for use in the Elliptic Curve Digital Signature Algorithm (ECDSA). These fields and corresponding reduction polynomials are listed in Table 2.1. Note that each of the reduction polynomials listed in the table is either a trinomial or a pentanomial. Also, note that the second leading non-zero coefficient of the polynomial has a relatively small degree when compared to the degree of the whole polynomial. Polynomials were chosen with these properties in order to benefit the resulting implementation of finite field arithmetic.

Table 2.1: NIST Recommended Finite Fields

Field	Reduction Polynomial
$\mathrm{GF}(2^{163})$	$F(x) = x^{163} + x^7 + x^6 + x^3 + 1$
$\mathrm{GF}(2^{233})$	$F(x) = x^{233} + x^{74} + 1$
$\mathrm{GF}(2^{283})$	$F(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
$\mathrm{GF}(2^{409})$	$F(x) = x^{409} + x^{87} + 1$
$\mathrm{GF}(2^{571})$	$F(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

2.2 Arithmetic over Binary Finite Fields

The elements of the binary field $\mathrm{GF}(2^m)$ are interrelated through the operations of addition and multiplication. Since the additive and multiplicative inverses exist for all fields, the subtraction and division operations are also defined. Discussed in this section are basic methods for computing the sum, difference and product of two elements. Also presented is a method for computing the inverse of an element. The inverse, along with a multiplication, is used to implement division.

Addition and Subtraction: If we define the field elements $a, b \in \mathrm{GF}(2^m)$ to be the polynomials $A(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$ and $B(x) = b_{m-1}x^{m-1} + \cdots + b_1 x + b_0$ respectively, then their sum is written

$$S(x) = A(x) + B(x) = \sum_{i=0}^{m-1} (a_i + b_i)x^i. \tag{2.1}$$

Working in a field of characteristic two provides two distinct advantages. First, the bit additions $a_i + b_i$ in (2.1) are performed modulo 2 and translate to an exclusive-OR (XOR) operation. The entire addition is computed by a component-wise XOR operation and does not require a carry chain. The second advantage is that in GF(2) the element 1 is its own additive inverse (i.e. $1 + 1 = 0$ or $1 = -1$). It can be concluded then that addition and subtraction are equivalent.

2.2.1 Multiplication

The product of field elements a and b is written as

$$P(x) = A(x) \times B(x) \mod F(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j x^{i+j} \mod F(x)$$

where $F(x)$ is the field reduction polynomial. By expanding $B(x)$ and distributing $A(x)$ through its terms we get

$$P(x) = b_{m-1}x^{m-1}A(x) + \cdots + b_1 x A(x) + b_0 A(x) \mod F(x).$$

By repeatedly grouping multiples of x and factoring out x we get

$$P(x) = (\cdots(((A(x)b_{m-1})x + A(x)b_{m-2})x + \cdots + A(x)b_1)x + A(x)b_0) \mod F(x). \tag{2.2}$$

Starting with the inner most parenthesis and moving out, Algorithm 1 performs the computation required to compute the right hand side of (2.2). This algorithm can be used to compute the product of a and b.

Algorithm 1 Bit-Level Multiplication

Input: $A(x)$, $B(x)$, and $F(x)$

Output: $P(x) = A(x) \times B(x) \mod F(x)$

$P(x) \leftarrow 0$;

for $i = m - 1$ **downto** 0 **do**

$P(x) \leftarrow xP(x) \mod F(x)$;

if $(b_i == 1)$ **then**

$P(x) \leftarrow P(x) + A(x)$;

Many of the faster multiplication algorithms rely on the concept of group-level multiplication. Let g be an integer less than m and let $s = \lceil m/g \rceil$ (Note that g is different here from previous usage). If we define the polynomials

$$B_i(x) = \begin{cases} \sum_{j=0}^{g-1} b_{ig+j} x^j & 0 \le i \le s-2, \\ \sum_{j=0}^{(m \mod g)-1} b_{ig+j} x^j & i = s-1, \end{cases}$$

then the product of a and b is written

$$P(x) = A(x) \left(x^{(s-1)g} B_{s-1}(x) + \cdots + x^g B_1(x) + B_0(x) \right) \mod F(x).$$

In the derivation of equation (2.2) multiples of x were repeatedly grouped then factored out. This same grouping and factoring procedure will now be implemented for

multiples of x^g arriving at

$$P(x) = (\cdots((A(x)B_{s-1}(x))x^g + A(x)B_{s-2}(x))x^g + \cdots)x^g + A(x)B_0(x) \mod F(x)$$

which can be computed using Algorithm 2.

Algorithm 2 Group-Level Multiplication

Input: $A(x)$, $B(x)$, and $F(x)$

Output: $P(x) = A(x)B(x) \mod F(x)$

$P(x) \leftarrow B_{s-1}(x)A(x) \mod F(x)$;

for $k = s - 2$ **downto** 0 **do**

$P(x) \leftarrow x^g P(x)$;

$P(x) \leftarrow B_k(x)A(x) + P(x) \mod F(x)$;

2.2.2 Inversion

For any element $a \in \mathrm{GF}(2^m)$ the equality $a^{2^m-1} \equiv 1$ holds. When $a \neq 0$, dividing both sides by a results in $a^{2^m-2} \equiv a^{-1}$. Using this equality the inverse, a^{-1}, can be computed through successive field squarings and multiplications. In Algorithm 3 the inverse of an element is computed using this method.

The primary advantage to this inversion method is the fact that it does not require hardware dedicated specifically to inversion. The field multiplier can be used to perform all required field operations.

Algorithm 3 Inversion by Square and Multiply

Input: Field element a

Output: $b \equiv a^{(-1)}$

$b \leftarrow a$;

for $i = 1$ **to** $m - 2$ **do**

  $b \leftarrow b^2 * a$;

$b \leftarrow b^2$;

2.3 Arithmetic over the Elliptic Curve Group

The field operations discussed in the previous section are used to perform arithmetic over an elliptic curve. This thesis is aimed at the elliptic curve defined by the non-supersingular Weierstrass equation for binary fields. This curve is defined by the equation

$$y^2 + xy = x^3 + \alpha x^2 + \beta \tag{2.3}$$

where the variables x and y are elements of the field GF(2^m) as are the curve parameters α and β. The points on the curve, defined by the solutions, (x, y), to (2.3) form an additive group when combined with the "point at infinity". This extra point is the group identity and is denoted by the symbol $\mathcal{O}$. By definition, the addition of two elements in a group results in another element of the group. As a result any point on the curve, say P, can be added to itself an arbitrary number of times and the result will also be a point on the curve. So for any integer k and point P adding P to itself $k - 1$ times results in the point

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}.$$

Given the binary expansion $k = 2^{l-1}k_{l-1} + 2^{l-2}k_{l-2} + \cdots + 2k_1 + k_0$ the scalar multiple kP can be computed by

$$Q = kP = 2^{l-1}k_{l-1}P + 2^{l-2}k_{l-2}P + \cdots + 2k_1P + k_0P.$$

By factoring out 2, the result is

$$Q = (2^{l-2}k_{l-1}P + 2^{l-3}k_{l-2}P + \cdots + k_1P)2 + k_0P.$$

By repeating this operation it is seen that

$$Q = (\cdots((k_{l-1}P)2 + k_{l-2}P)2 + \cdots + k_1P)2 + k_0P$$

which can be computed by the well known (left-to-right) double and add method for scalar multiplication shown in Algorithm 4.

Algorithm 4 Scalar Multiplication by Double and Add Method

Input: Integer $k = (k_{l-1}, k_{l-2}, \ldots, k_1, k_0)_2$, Point P

Output: Point $Q = kP$

```
Q ← O;
if (k_{l-1} == 1) then
    Q ← P;
for i = l − 2 downto 0 do
    Q ← DOUBLE(Q);
    if (k_i == 1) then
        Q ← ADD(Q, P);
```

Two basic operations required for elliptic curve scalar multiplication are point `ADD` and point `DOUBLE`. The mathematical definitions for these operations are derived from the curve equation in (2.3). Consider the points P_1 and P_2 represented by the coordinate pairs (x_1, y_1) and (x_2, y_2) respectively. Then the coordinates, (x_a, y_a), of point $P_a = P_1 + P_2$ (or $\texttt{ADD}(P_1, P_2)$) are computed using the equations

$$x_a = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + \alpha$$
$$y_a = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)(x_1 + x_a) + x_a + y_1.$$

Similarly the coordinates (x_d, y_d) of point $P_d = 2P_1$ (or $\texttt{DOUBLE}(P_1)$) are computed using the equations

$$x_d = x_1^2 + \left(\frac{\beta}{x_1^2}\right)$$
$$y_d = x_1^2 + \left(x_1 + \frac{y_1}{x_1}\right) x_d + x_d.$$

So the addition of two points can be computed using two field multiplications, one field squaring, eight field additions and one field inversion. The double of a point can be computed using two field multiplications, one field squaring, six field additions and one field inversion.

2.4 Implementation Media

In the end, the goal of this work is to implement the field and group arithmetic described above using hardware. This can be done using two different hardware technologies.

They are:

- Application Specific Integrated Circuits (ASICs)
- Field Programmable Gate Arrays (FPGAs)

ASICs are typically used when a design is massed produced or when performance is of the utmost importance. FPGAs, on the other hand, lend themselves nicely to research work where a design is being prototyped. The following attributes of the FPGA design flow are particularly advantageous.

1. Relatively small initial setup cost. A single FPGA is inexpensive when compared to the manufacturing cost of an ASIC design.
2. Simplified implementation flow. In most cases, the FPGA vendor (such as Xilinx or Altera) will provide a fully integrated tool flow. This flow will have been fully tested for compatibility with the FPGA and as a result fewer tool related problems can be expected.
3. Fast turn around time. An FPGA can be programmed in less than a minute and can also be reprogrammed many times. An ASIC on the other hand may take months to fabricate.
4. Simplified integration. Whether using an ASIC or FPGA design flow, the design must be integrated into a hardware/software system. It is common for FPGAs to be sold within such a system, minimizing the integration task required of the designer.

It makes sense that most other ECC prototypes have been implemented using FPGA technology. By following suit, results can be more easily compared to those of previously reported work. The following section provides an overview of the fundamental principles on which FPGAs are based. Introduced next is the Rapid-Prototyping Platform which includes the FPGA and hardware/software system used to prototype the design discussed in this thesis.

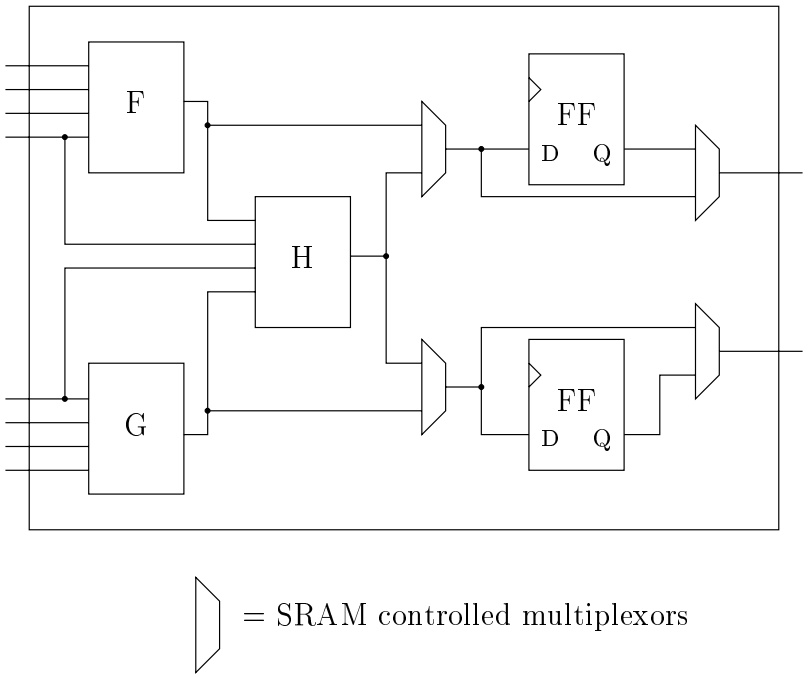
2.4.1 Field Programmable Gate Arrays

An FPGA or field programmable gate array is an integrated circuit consisting of

- Configurable Logic Blocks (CLBs),
- Input/Output Blocks (IOBs) and
- programmable interconnect.

Configurable Logic Blocks: A typical Configurable Logic Block (CLB) is composed of both combinational and sequential logic. The combinational logic can be configured to create any of a number of possible boolean functions. Flip-Flops are provided to support sequential logic and can be utilized or bypassed depending on the configuration. Figure 2.1 shows an example CLB with 8 inputs and 2 outputs. The blocks F, G and H are programmable functions which can be configured to perform any one of a number of different boolean functions. The functions are typically implemented with either look-up tables (LUTs) or logic gates. The actual number of possible boolean functions depends on the implementation. The multiplexors are used to configure the interconnect inside the CLB.

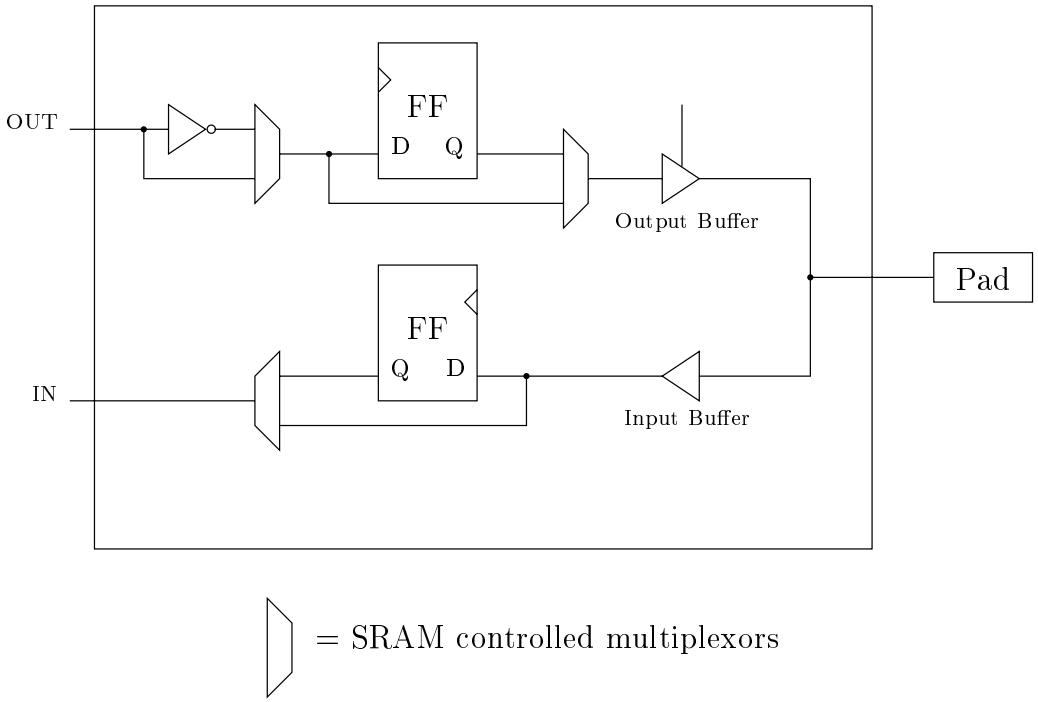
Figure 2.1: Functionality of a CLB



Input/Output Blocks: The Input/Output Blocks (IOBs) are blocks used to connect internal *nets* to external pins or pads on the FPGA. These blocks control the direction of the signal and can also register both input and output data. Figure 2.2 shows an example IOB.

Programmable Interconnect: An FPGA is made of many IOBs and CLBs. These blocks can be configured and connected together to achieve complex functionality. The connections between the blocks are performed by the programmable interconnect. There are several ways in which the CLBs, IOBs and programmable interconnect are organized. One such organization is the symmetric array method. As shown in Figure 2.3, the CLBs are organized in a two dimensional array with

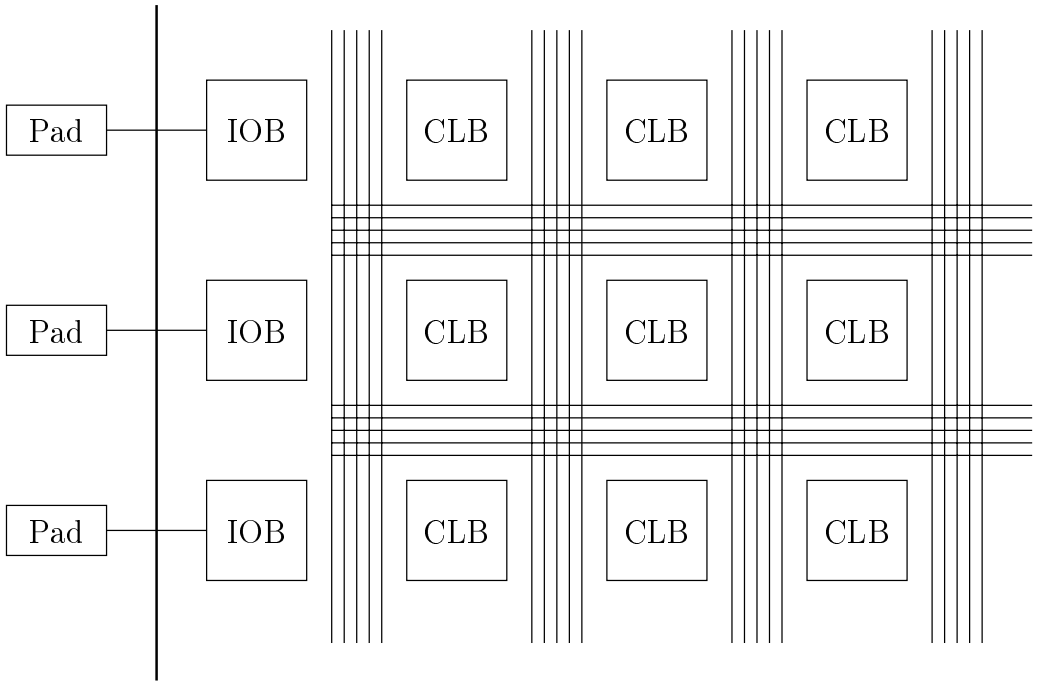
Figure 2.2: Functionality of an IOB



IOBs around the perimeter. The programmable interconnect is routed in between the blocks.

Configuring the FPGA: The configuration of each CLB and IOB as well as the programmable interconnect is defined when a design is loaded into the FPGA. The configuration is typically stored in static RAM cells. This allows the configuration to be preserved through reset of the FPGA while still providing the option of reconfiguration.

Figure 2.3: CLB Organization



2.4.2 The Rapid-Prototyping Platform

The Rapid-Prototyping Platform (RPP) [6, 7] is a hardware/software system provided to Canadian universities by Canadian Microelectronics Corporation (CMC). The major hardware components included in the system are:

- ARM Integrator/AP,
- ARM Integrator/CM7TDMI and
- ARM Integrator/LM-SCV600E+.

The Integrator/CM7TDMI board contains a fully functional ARM7 core. The Integrator/LM-SCV600E+ board holds a Xilnix XCV2000E FPGA. The chips on these two boards

are allowed to communicate through the Integrator/AP board. The common bus between the ARM7 core and the Virtex FPGA is the Arm High Performance Bus (AHB). In this system the ARM7 is the bus master and the design loaded onto the FPGA is the slave. In other words, the ARM7 core initiates bus transactions and the FPGA design responds to them.

The hardware and software design flows of the RPP are thoroughly documented in [6]. Provided here is a brief overview. Hardware flow, the more complicated of the two flows, is summarized in the following steps.

1. HDL (Hardware Description Language) coding. This is done in either VHDL or Verilog HDL.
2. Functional simulation and verification (Cadence Verilog XL).
3. Synthesis (Synopsys FPGA Compiler II).
4. Place/Route (Xilinx Foundation Software).
5. Static Timing Analysis (Xilinx Foundation Software).
6. Generate the bit file (Xilinx Foundation Software).
7. Download the bit file onto the FPGA.

If the design fails to pass static timing analysis, changes may need to be made to the HDL in which case all the steps must be performed again. The software side is less complicated.

1. Write the driver code in C using the ARM Firmware Suite provided with the RPP software environment. The firmware suite provides read and write commands used to access address locations on the AHB bus.

2. Compile the code for the ARM7 core.

3. Download the core into memory on the ARM7 core.

4. Execute the code.

Chapter 3

High Performance Finite Field Arithmetic

In order to optimize the curve arithmetic discussed in Section 2.3 the underlying field operations must be implemented in a fast and efficient way. The required field arithmetic operations are addition, multiplication, squaring and inversion. Each of these operations have been implemented in hardware for use in the prototype discussed in Chapter 4. Generally speaking, field multiplication has the greatest effect on the performance of the entire elliptic curve scalar multiplication.[1] For this reason, focus will be primarily on the field multiplier when discussing hardware architectures for field arithmetic.

This chapter is organized as follows. Section 3.1 presents a hardware architecture designed to perform finite field multiplication. In Section 3.2 the ideas presented for multiplication are extended to create a hardware architecture optimized for squar-

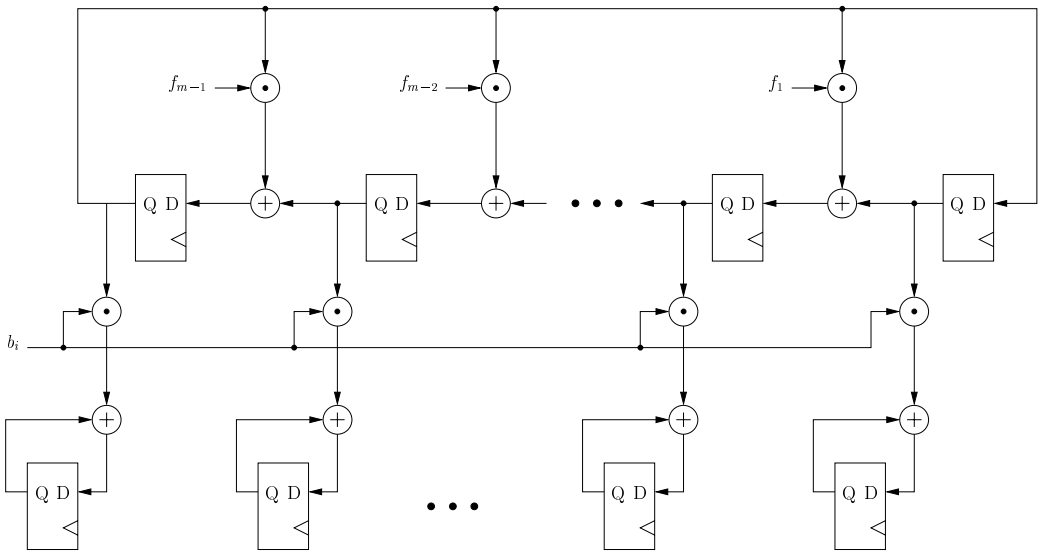
[1]Inversion takes much longer than multiplication, but its effect on performance can be greatly reduced through use of projective coordinates. This is discussed in greater detail in Section 4.1.

ing. Section 3.3 gives a method for inversion due to Itoh and Tsujii. This method does not require any additional hardware but instead uses the multiplication and squaring units described in Sections 3.1 and 3.2. Section 3.4 gives a description of a comparator/adder which both compares and adds finite field elements. Finally, Section 3.5 summarizes results gleaned from a hardware prototype of each arithmetic unit/routine.

3.1 Multiplication

Hardware/software architectures for field multiplication can be roughly categorized into three groups. *Bit Serial* multipliers are based on Algorithm 1 on page 14 where each coefficient of operand b is considered in a separate iteration of the for loop. Such an implementation is resource efficient in that it can be implemented using an m-bit LFSR defined by the reduction polynomial $F(x)$ along with an m bit accumulator. The LFSR and accumulator are connected as shown in Figure 3.1. The disadvantage of such an architecture is the number of iterations required of the for loop. In hardware, the m iterations translate to a minimum of m clock cycles. In contrast, *Bit Parallel* multipliers complete a multiplication in a single iteration. All m-bits of both input operands are considered at the same time and the result is immediately generated. Unfortunately, such a multiplier cannot be implemented in software and may result in a costly design when implemented in hardware. The minimum clock period of such an implementation is also likely to be large. A compromise between these architectures is the *Digit Serial* multiplier. This multiplier is based on Algorithm 2 on page 15 and considers multiple coefficients of operand b in each iteration.

Figure 3.1: LFSR Based Multiplier



A multiplication is completed in $\lceil m/g \rceil$ iterations and requires fewer resources than the bit parallel method.

In [13] a digit serial multiplier is proposed which is based on look-up tables. This method was implemented in software for the field GF(2^{163}) and reported in [16]. To the best of our knowledge this performance of 0.540 μ-seconds for a single field multiplication is the fastest reported result for a software implementation. In this section the possibilities of using this look-up table-based algorithm in hardware will be explored.

First to be described in this section is the algorithm used for multiplication. Then presented is a hardware structure designed to compute $R(x)W(x) \mod F(x)$ where $R(x)$ and $W(x)$ are polynomials with degrees $g-1$ and $m-1$ respectively and $g << m$. A description of the multiplier's data path follows. In conclusion there will be a discussion behind the reasons for the choice of digit sizes.

3.1.1 Algorithm

The computations of

$$P(x) \leftarrow x^g P(x) \mod F(x) \text{ and}$$

$$P(x) \leftarrow B_k(x)A(x) + P(x) \mod F(x)$$

from the **for** loop of Algorithm 2 on page 15 can be broken up into the following steps.

$$V_1 = x^g \sum_{i=0}^{m-g-1} p_i x^i,$$

$$V_2 = x^g \sum_{i=m-g}^{m-1} p_i x^i \mod F(x)$$

$$V_3 = B_k(x)A(x) \mod F(x) \text{ and}$$

$$P(x) = V_1 + V_2 + V_3$$

Note that V_1 is a g-bit shift of the lower $m-g$ bits of $P(x)$. V_2 is a g-bit shift of the upper g bits of $P(x)$ followed by a modular reduction. V_3 requires a polynomial multiplication and reduction where the operand polynomials have degree $g-1$ and $m-1$. Algorithm 2 can be modified to create Algorithm 5.

In [13] polynomials V_2 and V_3 are computed with the assistance of look-up tables mainly for software implementation. The look-up tables used to compute V_2 and V_3 are referred to as the M-Table and T-Table respectively. The M-Table is addressed by the bit string $(p_{m-1}, p_{m-2}, \ldots, p_{m-g})$ interpreted as the integer $2^{g-1}p_{m-1} + 2^{g-2}p_{m-2} + \cdots + p_{m-g}$. Similarly the T-Table is addressed by the coefficients of $B_k(x)$, or the

Algorithm 5 Efficient Group Level Multiplication

Input: $A(x)$, $B(x)$, and $F(x)$

Output: $P(x) = A(x)B(x) \mod F(x)$

$P(x) \leftarrow B_{s-1}(x)A(x) \mod F(x)$;

for $k = s - 2$ **downto** 0 **do**

$V_1 \leftarrow x^g \sum_{i=0}^{m-g-1} p_i x^i$;

$V_2 \leftarrow x^g \sum_{i=m-g}^{m-1} p_i x^i \mod F(x)$;

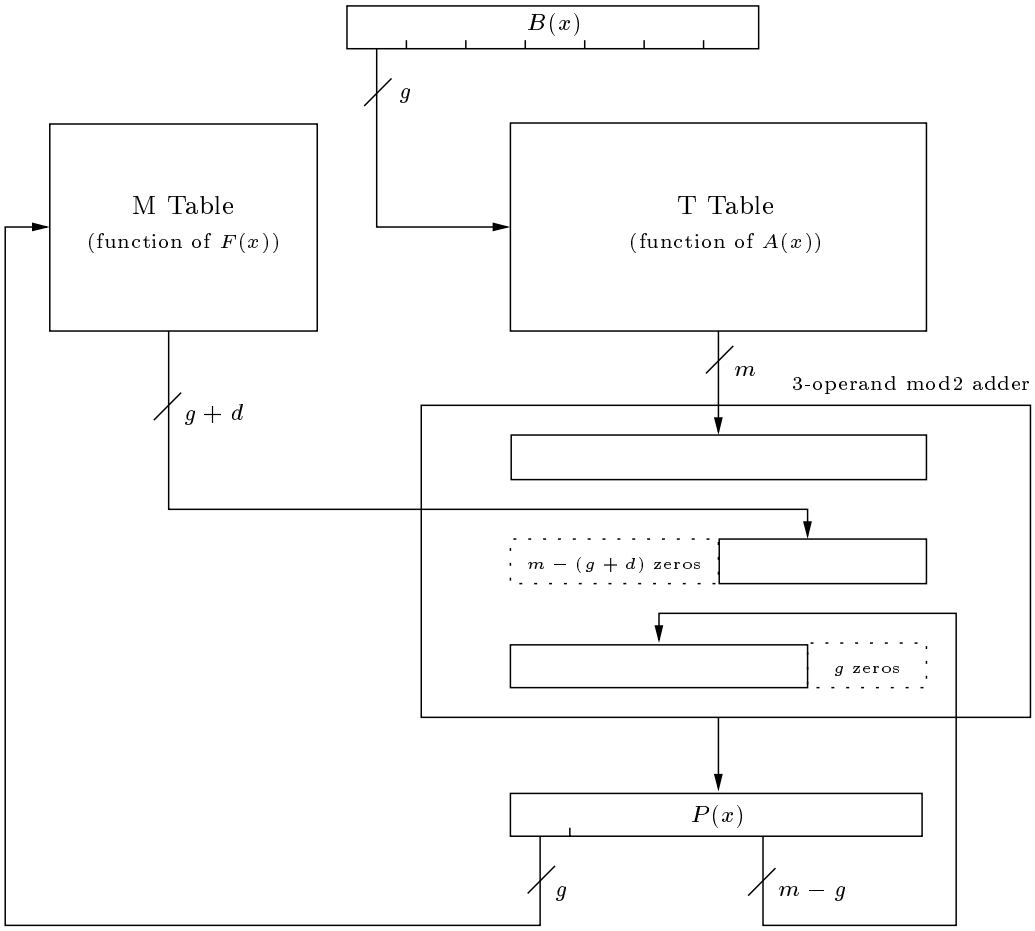
$V_3 \leftarrow B_k(x)A(x) \mod F(x)$;

$P(x) \leftarrow V_1 + V_2 + V_3$;

integer $B_k(x = 2)$. The elements of the M-Table are a function of the reduction polynomial $F(x)$ and can be precomputed. The elements of the T-Table are a function of $A(x)$ and hence are dynamic. These values must be computed at the beginning of every multiplication.

The data path associated with this method is shown in Figure 3.2. The given multiplier is based on this method but is optimized specifically for hardware implementation.

Figure 3.2: The Multiplier Data-Path



3.1.2 Computation of $R(x)W(x) \mod F(x)$

Instead of using tables, the polynomials V_2 and V_3 are computed on the fly. The computation of V_2 and V_3 are similar in that they both require a multiplication of two polynomials followed by a reduction, where the first polynomial has degree $g-1$ and the other has degree less than m. This is obvious for V_3 and can be shown easily for V_2. Note that

$$\begin{aligned} V_2 &= p_{m-1}x^{m+g-1} + \cdots + p_{m-g+1}x^{m+1} + p_{m-g}x^m \mod F(x) \\ &= x^m \left(p_{m-1}x^{g-1} + \cdots + p_{m-g+1}x + p_{m-g}\right) \mod F(x). \end{aligned}$$

The field reduction polynomial $F(x) = x^m + x^d + \cdots + 1$ provides us the equality $x^m \equiv x^d + \cdots + 1$. Substituting for x^m we see that

$$V_2 = \left(x^d + \cdots + 1\right)\left(p_{m-1}x^{g-1} + \cdots + p_{m-g+1}x + p_{m-g}\right) \mod F(x).$$

Provided $d + g < m$, V_2 results in a polynomial of degree less than m which does not need to be reduced. Since d is relatively small for all five NIST polynomials, it is reasonable to assume that $d+g < m$. For the remainder of this work, this assumption will be made.

With this said, the following method can be used to compute both V_2 and V_3. Consider the polynomial multiplication and reduction $R(x)W(x) \mod F(x)$ where

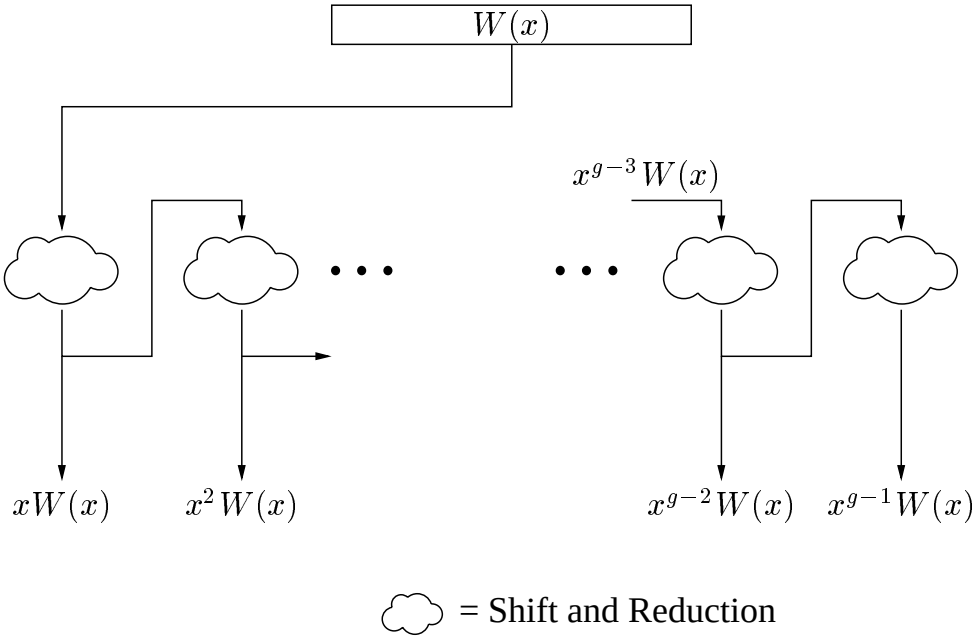
$R(x) = \sum_{i=0}^{g-1} r_i x^i$ and $W(x)$ is a polynomial with degree less than m. Then

$$\begin{aligned} R(x)W(x) \mod F(x) = & r_{g-1}(x^{g-1}W(x) \mod F(x)) \\ & + r_{g-2}(x^{g-2}W(x) \mod F(x)) \\ & \vdots \\ & + r_1(xW(x) \mod F(x)) \\ & + r_0(W(x) \mod F(x)) \end{aligned}$$

The value $x^i W(x) \mod F(x)$ is just a shifted and reduced version of $x^{i-1}W(x) \mod F(x)$. So each value $x^i W(x) \mod F(x)$ can be generated sequentially starting with $x^0 W(x)$ as shown in Figure 3.3. When using a reduction polynomial with a low Hamming weight, such as a trinomial or pentanomial, these terms can be computed quickly at very little cost. Once these values are determined, the final result is computed using a g-input modulo 2 adder. The inputs to the adder are enabled by their corresponding coefficient r_i. This is shown in Figure 3.4. Note that the polynomial $x^i W(x)$ affects the output of the adder only if the coefficient bit r_i is a one. Otherwise the input associated with $x^i W(x)$ is driven with zeros.

Each individual output bit of the g-operand mod 2 adder is computed using $g-1$ XOR gates and g AND gates. The AND gates are used to enable each input bit and the XOR gates compute the mod 2 addition. Figure 3.5 demonstrates how this is done. The depth of the logic in the figure is linearly related to g.

Figure 3.3: Generating $x^i W(x) \mod F(x)$



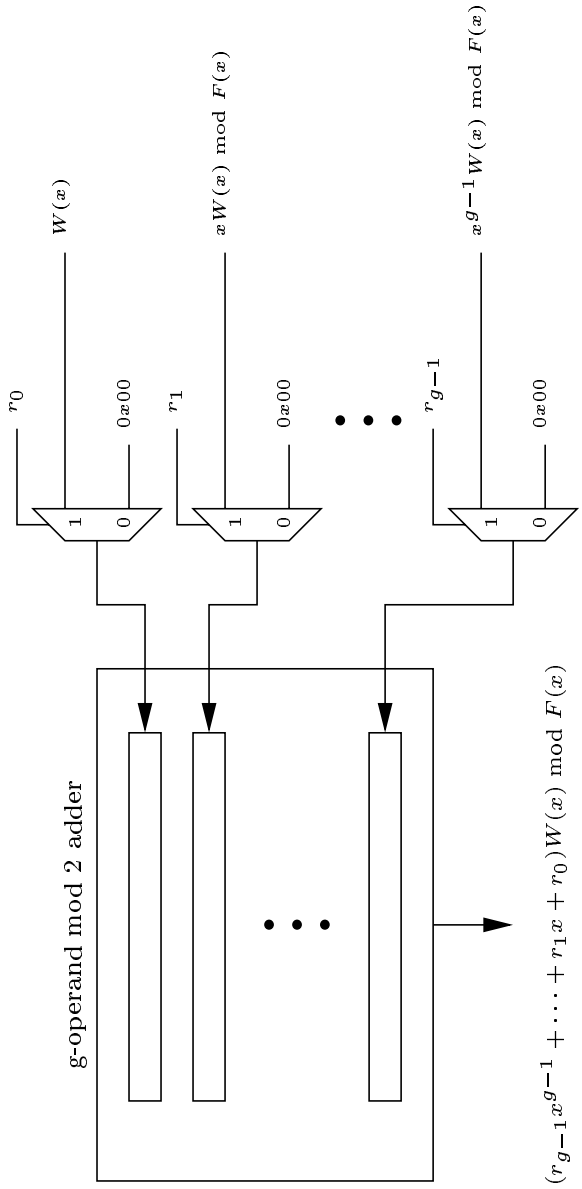


Figure 3.4: Computing $R(x)W(x) \mod F(x)$

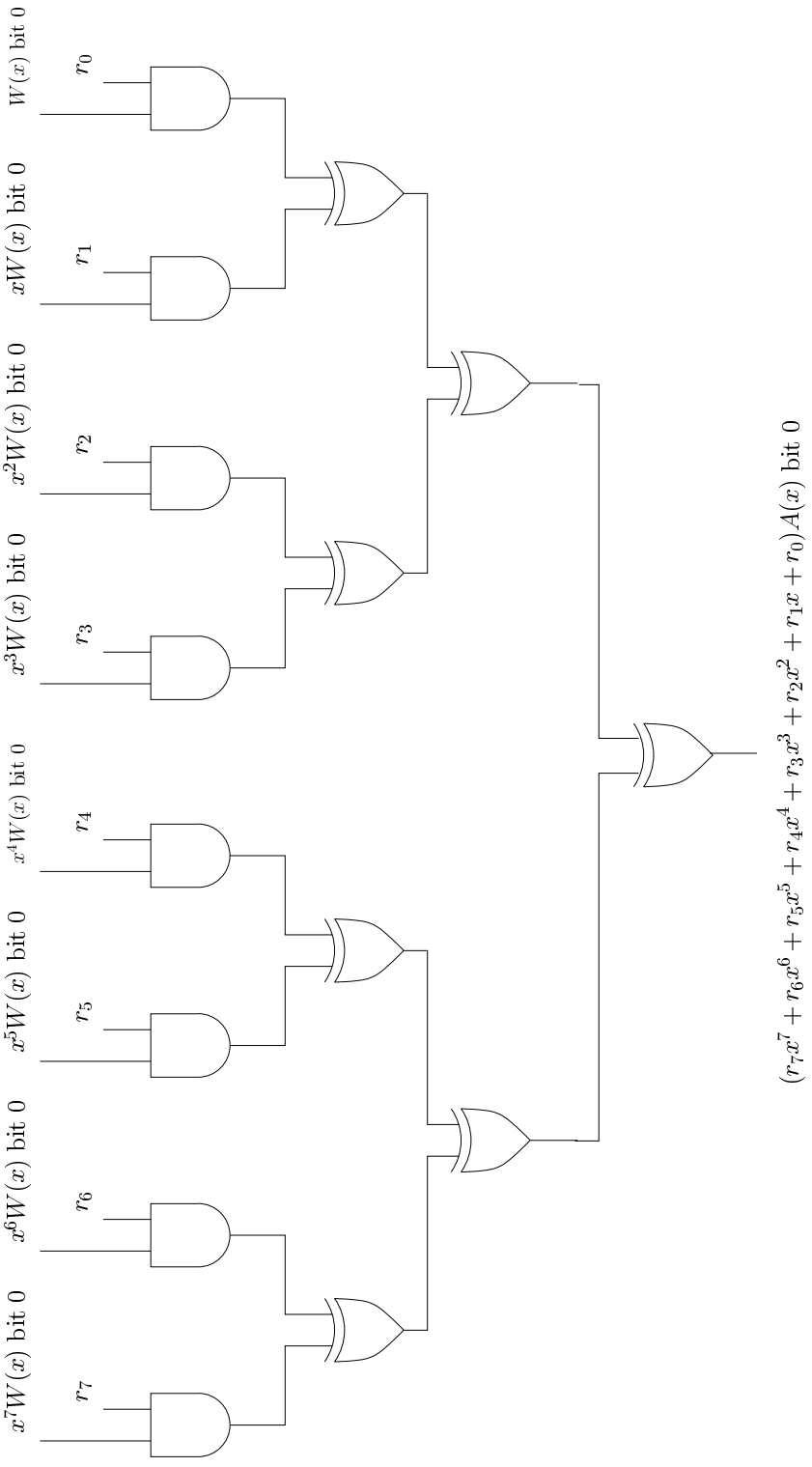


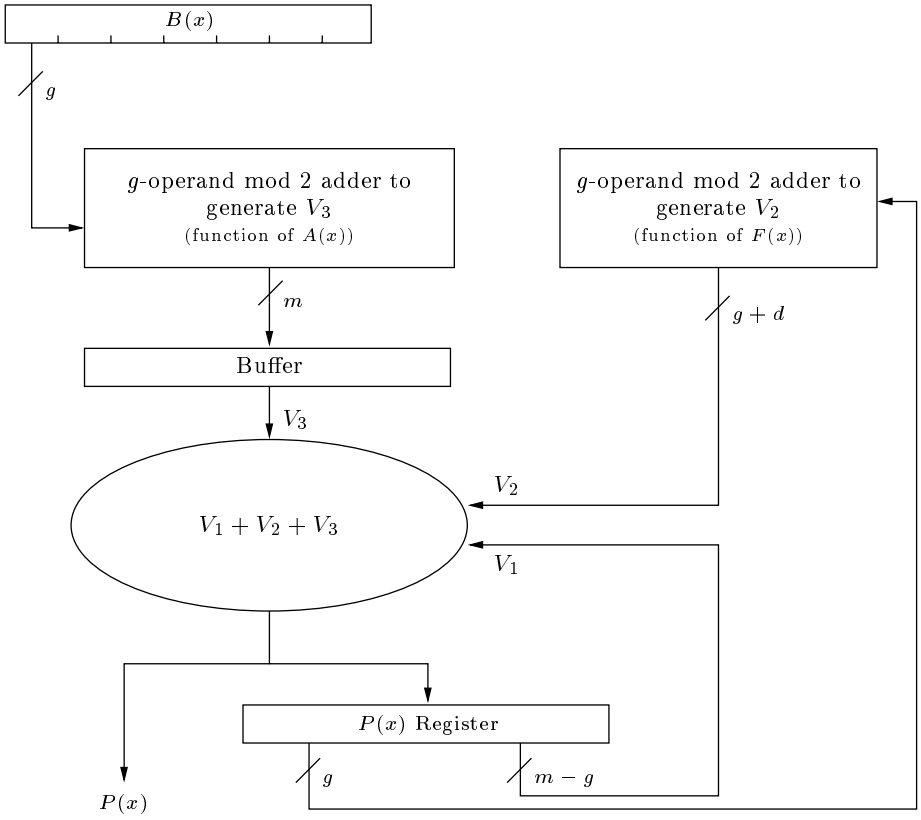
Figure 3.5: Computation of a Single Bit in $R(x)W(x) \mod F(x)$

This method for multiplication is implemented for computation of both V_2 and V_3. In the case of V_3, the polynomial $W(x)$ has degree $m-1$ and will change for every field multiplication. For V_2 the polynomial $W(x)$ has degree d and is fixed. The value d is the degree of the second leading non-zero coefficient of $F(x)$. For reasonable digit sizes this computation can be performed in a single clock cycle.

3.1.3 The Multiplier Data Path

The multiplier's data path connecting the V_2 and V_3 generators along with the adder used to compute $P(x) = V_1 + V_2 + V_3$ is shown in Figure 3.6. A buffer is inserted at the output of the V_3 generator to separate its delay from the delay of the adder for $V_1 + V_2 + V_3$. This, in effect, increases the maximum possible value for the digit size g. If added by itself, this buffer would add a cycle of latency to the multiplier's performance time. This extra cycle is compensated for by bypassing the $P(x)$ register and driving the multiplier's output with the output of the 3-operand mod2 adder. It is important to note that the delay of the 3-operand mod2 adder is being merged with the delay of the bus which connects the multiplier to the rest of the design. In this case the relatively relaxed bus timing had room to accommodate the delay.

Figure 3.6: Modified Multiplier Data-Path



3.1.4 Choice of Digit Size

The multiplier will complete a multiplication in $\lceil m/g \rceil$ clock cycles. Since this is a discrete value, the performance may not change for every value of g. To minimize cost of the multiplier (which increases with g) the smallest digit size g should be chosen for a given performance $\lceil m/g \rceil$. For example, the digit sizes $g = 21$ and $g = 22$ for field size $m = 163$ result in the same performance, $\lceil \frac{163}{21} \rceil = \lceil \frac{163}{22} \rceil = 8$, but $g = 22$ requires a larger multiplier.

A prototype of this multiplier for the field GF(2^{163}) and NIST polynomial has been implemented for each of the digit sizes shown in Table 3.1. For each digit size, the table lists the corresponding cycle performance and resource cost. A maximum digit size of $g = 41$ was chosen for several reasons. First, as the performance cost of the actual field multiplication decreases, the relative cost of loading and unloading the multiplier increases. So as the digit size increases, its affect on the total performance (including time to load and unload the multiplier) decreases. Second, results showed that $g > 41$ had difficulty meeting timing at the target operating frequency of 66 MHz. Instead of spending time redesigning the field multiplier, a maximum digit size of 41 was selected.

3.2 Squaring

While squaring is a specific case of general multiplication and can be performed by the multiplier, performance can be improved significantly by optimizing the architecture specifically for the case of squaring. The square of an element a represented by $A(x)$ involves two mathematical steps. The first is the polynomial multiplication of $A(x)$

Table 3.1: Performance/Cost Trade-off for Multiplication over GF(2^{163})

Digit Size	Performance in clock cycles	# LUTs	# Flip Flops
$g = 1$	163	677	670
$g = 4$	41	854	670
$g = 28$	6	3,548	670
$g = 33$	5	4,040	670
$g = 41$	4	4,728	670

resulting in

$$A^2(x) = a_{m-1}x^{2m-2} + \cdots + a_2x^4 + a_1x^2 + a_0.$$

The second is the reduction of this polynomial modulo $F(x)$. If the terms with degree greater than $m-1$ are separated and x^{m+1} is factored out where possible the result will be $A^2(x) = A_h(x)x^{m+1} + A_l(x)$ where

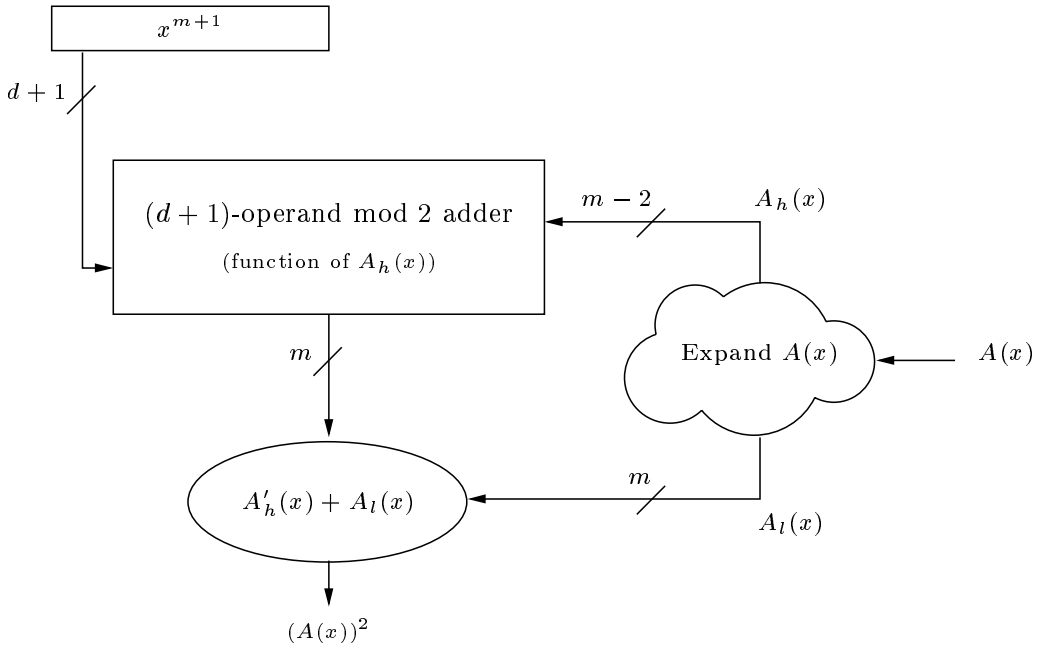
$$A_h(x) = a_{m-1}x^{m-3} + \cdots + a_{(\frac{m+3}{2})}x^2 + a_{(\frac{m+1}{2})}$$
$$A_l(x) = a_{(\frac{m-1}{2})}x^{m-1} + \cdots + a_1x^2 + a_0,$$

The polynomial $A_l(x)$ has degree less than m and does not need to be reduced. The product $A_h(x)x^{m+1}$ may have degree as large as $2m-2$. The reduction polynomial gives us the equality $x^m = x^d + \cdots + 1$. Multiplying both sides by x, we get $x^{m+1} = x^{d+1} + \cdots + x$. So

$$A_h(x)x^{m+1} = A_h(x)\left(x^{d+1} + \cdots + x\right).$$

This multiplication can be performed using a method similar to the one described in Section 3.1. The same architecture used to compute $R(x)W(x) \mod F(x)$ in the multiplier is used here to compute $x^{m+1}A_h(x)$. The digit size is set to $g = d + 2$ and the elements of g-operand mod 2 adder are generated from $A_h(x)$. $A_h(x)$ is in turn generated by expanding $A(x)$ (i.e. inserting zeros between the coefficient bits of $A(x)$). Since the digit size is set to $d + 2$, the multiplication is completed in a single cycle. This method only works if $d + 2 < m$ which is the case for each of the NIST polynomials. Figure 3.7 shows the data flow for the squaring operation. Note that the flow does not include any buffers and so is implemented in pure combinational logic.

Figure 3.7: Data-Path of the Squaring Unit



The prototype of this squaring unit for field GF(2^{163}) using the NIST reduction

polynomial runs at 66 MHz and is capable of performing a squaring operation in a single clock cycle. This implementation requires 330 LUTs and 328 Flip Flops.

3.3 Inversion

The inversion method described in Algorithm 3 on page 16 requires $m-1$ squarings and $m-2$ multiplications. In order to accurately estimate the cycle performance of the inversion, consideration must be given to the performance of the multiplication and squaring units as well as the time required to load and unload these units. The architecture of the elliptic curve scalar multiplier will be discussed in detail in Chapter 4. For now, it is sufficient to know that the arithmetic units are loaded using two independent m bit data buses and unloaded using a single m bit data bus. The operands are stored in a dual port memory which takes two clock cycles to read from and one cycle to write to. These combined makes three cycles that are required to both load and unload any arithmetic unit. Further analysis assumes that these three cycles remain constant for all m. If C_s and C_m denote the number of clock cycles required to complete a squaring and multiplication respectively, then an inversion can be completed in

$$(C_s + 3)(m-1) + (C_m + 3)(m-2)$$

clock cycles. For the field GF(2^{163}) where $C_s = 1$ and $C_m = 4$, this translates to 1775 clock cycles.

Performance can be improved by using Algorithm 6 due to Itoh and Tsujii [15]. This algorithm is derived from the equation $a^{(-1)} \equiv a^{2^m-2} \equiv \left(2^{2^{m-1}-1}\right)^2$ which is

true for any element $a \in \mathrm{GF}(2^m)$. From

$$a^{2^t-1} \equiv \begin{cases} \left(a^{2^{t/2}-1}\right)^{2^{t/2}} \left(a^{2^{t/2}-1}\right) & \text{for } t \text{ even,} \\ a\left(a^{2^{t-1}-1}\right)^2 & \text{for } t \text{ odd,} \end{cases} \tag{3.1}$$

the computation required for the exponentiation $2^{2^{m-1}-1}$ can be iteratively broken down. Algorithm 6 requires $\lfloor \log_2(m-1) \rfloor + H(m-1) - 1$ multiplications and $m-1$ squarings. Using the notation defined earlier, this translates to

$$(C_s + 3)(m-1) + (C_m + 3)(\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)$$

clock cycles. For $\mathrm{GF}(2^{163})$ this translates to 711 clock cycles.

Algorithm 6 Optimized Inversion by Square and Multiply [15]

Inputs: Field element a,

Binary representation of $m-1 = (m_{l-1}, \ldots, m_2, m_0)_2$

Output: $b \equiv a^{(-1)}$

$b \leftarrow a^{m_{l-1}}$;

$e \leftarrow 1$;

for $i = l-2$ **downto** 0 **do**

  $b \leftarrow b^{2^e} b$;

  $e \leftarrow 2e$;

  if $(m_i == 1)$ **then**

    $b \leftarrow b^2 a$;

    $e = e + 1$;

$b \leftarrow b^2$;

Now, the majority of the time spent for each squaring operation is used to load and unload the squaring unit (three out of the four cycles). Algorithm 6 requires several sequences of repetitive squaring (i.e. computations of the form x^{2^t}). These repeated squarings do not require intermediate values to be stored outside the squaring unit. By modifying the squaring unit to support the *re-square* of an element, most of the memory accesses otherwise required to load and unload the squaring unit are eliminated. In fact, the squaring unit only needs to be loaded and unloaded once for each multiplication. Hence the number of clock cycles is reduced to

$$\begin{aligned}(C_s(m-1) + 3(\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)) \\ + (C_m + 3)(\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)\end{aligned}$$

clock cycles. For the field GF(2^{163}) with $C_s = 1$ and $C_m = 4$, this results in 252 clock cycles.

This is a competitive value since a typical hardware implementation of the Extended Euclidean Algorithm (EEA) is expected to complete an inversion in approximately $2m$ clock cycles or 326 cycles for GF(2^{163}). This corresponds to a 60 clock cycle reduction or 20% performance improvement without requiring hardware dedicated specifically for inversion. Table 3.2 lists the performance numbers of the previously mentioned inversion methods when implemented over the field GF(2^{163}).

The actual time to complete an inversion using the ECC co-processor architecture discussed in Chapter 4 is 259 clock cycles. The 7 extra cycles are due to control related instructions executed in the micro-sequencer.

Table 3.2: Comparison of Various Inversion Methods for GF(2^{163})

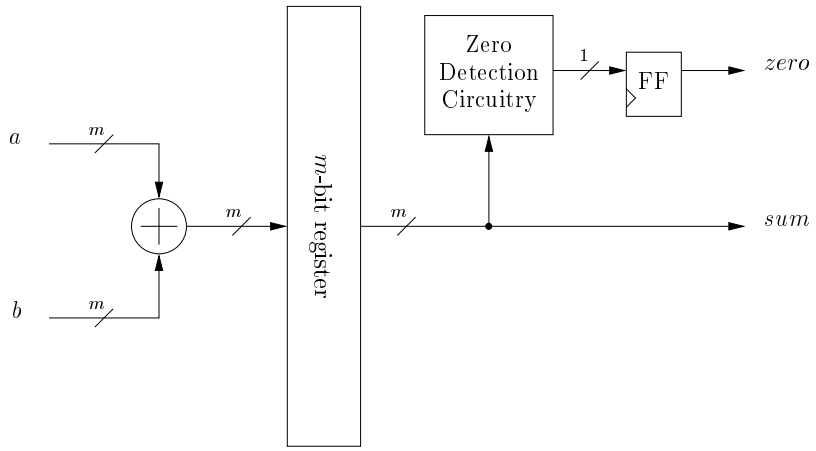
Method	# Squarings	# Multiplications	# Cycles
Square & Multiply	$m-1$	$m-2$	1127
Itoh & Tsujii	$m-1$	$\lfloor \log_2(m-1) \rfloor + H(m) - 1$	711
Itoh & Tsujii w/ *re-square*	$m-1$	$\lfloor \log_2(m-1) \rfloor + H(m) - 1$	252
EEA	-	-	326

3.4 Comparator/Adder

The primary purpose of the Comparator/Adder is to compute the sum of two field elements. This is done with an array of m exclusive OR gates. To minimize register usage as well as time to complete the addition, the sum of the two operands is the only value stored in a register. In this way, the sum is available immediately after the operands are loaded into the Comparator/Adder. In other words, it takes zero clock cycles to complete a finite field addition.

In addition to computing the sum of two finite field elements, the Comparator/Adder also acts as a comparator. The comparison is performed by taking the logical NOR of all the bits in the sum register. If the result is a one, then the sum is zero and the two operands are equal. If operand a is set to zero, then operand b can be tested for zero. The logic depth for the zero detect circuitry (the m-bit NOR gate) is $\log_2(m)$ and is registered before being sent out of the module. Figure 3.8 provides a functional diagram of the Comparator/Adder.

Figure 3.8: Data-Path of the Comparator/Adder



3.5 Concluding Remarks

In this chapter, we have discussed hardware architectures designed to perform finite field addition, multiplication and squaring. Also discussed was an efficient method for inversion which uses the squaring and multiplication units. The performance results associated with these arithmetic units are summarized in Table 3.3.

Table 3.3: Performance of Finite Field Operations

Operation ($g = 41$)	# Cycles	# Cycles Including Initial and Final Data Movement
Multiplication	4	7
Squaring	1	4
Addition	0	3
Inversion	256	259

Chapter 4

A Co-processor Architecture for ECC Scalar Multiplication

In the recent past, several articles have proposed various hardware architectures/accelerators for ECC. These elliptic curve cryptographic accelerators can be categorized into three functional groups. They are

1. Accelerators which use general purpose processors to implement curve operations but implement the finite field operations using hardware. References [2] and [32] are examples of this. Both of these implementations support the composite field GF(2^{155}).

2. Accelerators which perform both the curve and field operations in hardware but use a small field size such as GF(2^{53}). Architectures of this type include those proposed in [30] and [10]. In [30], a processor for the field GF(2^{168}) is synthesized, but not implemented. Both works discuss methods to extend their implementation to a larger field size but do not actually do so.

3. Accelerators which perform both curve and field operations in hardware and use fields of cryptographic strength such as GF(2^{163}). Processors in this category include [3, 12, 19, 26, 28].

The work discussed in this chapter falls into category three. The architectures proposed in [26] and [28] were the first reported cryptographic strength elliptic curve co-processors. Montgomery scalar multiplication with an LSD multiplier was used in [28]. In [26] a new field multiplier is developed and demonstrated in an elliptic curve scalar multiplier. In both [19] and [3] parameterized module generation is discussed. To the best of our knowledge the architecture proposed in [12] offers the fastest scalar multiplication using FPGA technology at 0.144 milliseconds. This architecture uses Montgomery scalar multiplication with López and Dahab's projective coordinates. They use a shift and add field multiplier but also compare LSD and Karatsuba multipliers.

In this chapter a hardware architecture for elliptic curve scalar multiplication is proposed. The architecture uses projective coordinates and is optimized for scalar multiplication over the Koblitz curves. The arithmetic routines discussed in Chapter 3 are used to perform the field arithmetic. This architecture has been implemented and demonstrated on an FPGA.

The chapter is organized as follows. Section 4.1 introduces projective coordinates and discusses some of the reasons for using a projective system. Section 4.2 presents two methods for recoding the scalar. They are non-adjacent form (NAF) and τ-adic non-adjacent form (τ-NAF). Then in Section 4.3 the ideas described in 4.1 and 4.2 are implemented in a co-processor architecture for scalar multiplication. The data path

and different levels of control are outlined there. Section 4.4 discusses the prototype of the scalar multiplier. Finally in Section 4.5 concludes with results gathered from the prototype.

4.1 Projective Coordinates

Projective coordinates allow the inversion required by each `DOUBLE` and `ADD` to be eliminated at the expense of a few extra field multiplications. The benefit is measured by the ratio

$$\frac{\text{Time to Complete Inversion}}{\text{Time to Complete Multiplication}}. \tag{4.1}$$

The inversion algorithm proposed by Itoh and Tsujii [15] will be used and therefore, the ratio in (4.1) is guaranteed to be larger than $\lfloor \log_2(m-1) \rfloor$ and could be larger depending on the efficiency of the squaring operations. Therefore, projective coordinates will provide us the best performance for NIST curves. Several flavors of projective coordinates have been proposed over the last few years. The prominent ones are *Standard* [22], *Jacobian* [5, 14] and López & Dahab [20] projective coordinates.

If the affine representation of P be denoted as (x, y) and the projective representation of P be denoted as (X, Y, Z), then the relation between affine and projective coordinates for the Standard system is

$$x = \tfrac{X}{Z} \quad \text{and} \quad y = \tfrac{Y}{Z}.$$

For Jacobian projective coordinates the relation is

$$x = \tfrac{X}{Z^2} \quad \text{and} \quad y = \tfrac{Y}{Z^3}.$$

Finally for López & Dahab's, the relation between affine and projective coordinates is

$$x = \frac{X}{Z} \quad \text{and} \quad y = \frac{Y}{Z^2}.$$

For López & Dahab's system the projective equation of the elliptic curve in (2.3) then becomes

$$Y^2 + XYZ = X^3Z + \alpha X^2Z^2 + \beta Z^4.$$

It is important to note that when using the left-to-right double and add method for scalar multiplication all point additions are of the form $\texttt{ADD}(P, Q)$. The base point P is never modified and as a result will maintain its affine representation (i.e. $P = (x, y, 1)$). The constant Z coordinate significantly reduces the cost of point addition (from 14 field multiplications down to 10). The addition of two distinct points $(X_1, Y_1, Z_1)+(X_2, Y_2, 1) = (X_a, Y_a, Z_a)$ using *mixed* coordinates (one projective point and one affine point) is then computed by

$$\begin{aligned}
A &= Y_2 \cdot Z_1^2 + Y_1 & E &= A \cdot C \\
B &= X_2 \cdot Z_1 + X_1 & X_a &= A^2 + D + E \\
C &= Z_1 \cdot B & F &= X_a + X_2 \cdot Z_a \\
D &= B^2 \cdot (C + \alpha \cdot Z_1^2) & G &= X_a + Y_2 \cdot Z_a \\
Z_a &= C^2 & Y_a &= E \cdot F + Z_a \cdot G
\end{aligned} \tag{4.2}$$

Similarly, the double of a point $(X_1, Y_1, Z_1) + (X_1, Y_1, Z_1) = (X_d, Y_d, Z_d)$ is computed

by

$$\begin{aligned} Z_d &= Z_1^2 \cdot X_1^2 \\ X_d &= X_1^4 + \beta \cdot Z_1^4 \\ Y_d &= \beta \cdot Z_1^4 \cdot Z_d + X_d \cdot (\alpha \cdot Z_d + Y_1^2 + \beta \cdot Z_1^4) \end{aligned} \tag{4.3}$$

In Table 4.1, the number of field operations required for the affine, Standard, Jacobean and López & Dahab coordinate systems are provided. In the table the symbols $\mathcal{M}$, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{I}$ denote field multiplication, squaring, addition and inversion respectively.

Table 4.1: Comparison of Projective Point Systems

System	Point Addition	Point Doubling
Affine	$2\mathcal{M} + 1\mathcal{S} + 8\mathcal{A} + 1\mathcal{I}$	$3\mathcal{M} + 2\mathcal{S} + 4\mathcal{A} + 1\mathcal{I}$
Standard	$13\mathcal{M} + 1\mathcal{S} + 7\mathcal{A}$	$7\mathcal{M} + 5\mathcal{S} + 4\mathcal{A}$
Jacobian	$11\mathcal{M} + 4\mathcal{S} + 7\mathcal{A}$	$5\mathcal{M} + 5\mathcal{S} + 4\mathcal{A}$
López & Dahab	$10\mathcal{M} + 4\mathcal{S} + 8\mathcal{A}$	$5\mathcal{M} + 5\mathcal{S} + 4\mathcal{A}$

The projective coordinate system defined by López and Dahab will be used since it offers the best performance for both point addition and point doubling.

4.2 Scalar Multiplication using Recoded Integers

The binary expansion of an integer k is written as $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, 1\}$. For the case of elliptic curve scalar multiplication the length l is approximately equal

to m, the degree of the extension field. Assuming an average Hamming weight, a scalar multiplication will require approximately $l/2$ point additions and $l-1$ point doubles. Several recoding methods have been proposed which in effect reduce the number of additions. In this section two methods are discussed; NAF [11, 31] and τ-adic NAF [18, 31].

4.2.1 Scalar Multiplication using Binary NAF

The symbols in the binary expansion are selected from the set $\{0, 1\}$. If this set is increased to $\{0, 1, -1\}$ the expansion is referred to as *signed binary* (SB) representation. When using this representation, the double and add scalar multiplication method must be slightly modified to handle the -1 symbol (often denoted $\bar{1}$). If the expansion $k'_{l-1}2^{l-1}+\cdots+k'_1 2+k'_0$ where $k'_i \in \{0, 1, \bar{1}\}$ is denoted by $(k'_{l-1}, \ldots, k'_1, k'_0)_{SB}$, then Algorithm 7 computes the scalar multiple of point P. The negative of the point

Algorithm 7 Scalar Multiplication for Signed Binary Representation

Input: Integer $k = (k'_{l-1}, k'_{l-2}, \ldots, k'_1, k'_0)_{\mathrm{SB}}$, Point P

Output: Point $Q = kP$

$Q \leftarrow \mathcal{O}$;
if $(k'_{l-1} \neq 0)$ **then**
  $Q \leftarrow k'_{l-1}P$;
for $i = l-2$ **downto** 0 **do**
  $Q \leftarrow$ `DOUBLE`(Q);
  if $(k'_i \neq 0)$ **then**
    $Q \leftarrow$ `ADD`$(Q, k'_i P)$;

(x, y) is $(x, x + y)$ and can be computed with a single field addition. The signed

binary representation is redundant in the sense that any given integer has more than one possible representation. For example, 17 can be represented by $(1001)_{\mathrm{SB}}$ as well as $(101\bar{1})_{\mathrm{SB}}$.

Interest here is in a particular form of this signed binary representation called NAF or non-adjacent form. A signed binary integer is said to be in NAF if there are no adjacent non-zero symbols. The NAF of an integer is unique and it is guaranteed to be no more than one symbol longer than the corresponding binary expansion. The primary advantage gained from NAF is its reduced number of non-zero symbols. The average Hamming weight of a NAF is approximately $l/3$ [31] compared to that of the binary expansion which is $l/2$. As a result, the running time of elliptic curve scalar multiplication when using binary NAF is reduced to $(l+1)/3$ point additions and l point doubles. This represents a significant reduction in run time.

In [31], Solinas provides a straightforward method for computing the NAF of an integer. This method is given here in Algorithm 8.

4.2.2 Scalar Multiplication using τ-NAF

Anomalous Binary Curves (ABC's), first proposed for cryptographic use in [18], provide an efficient implementation when the scalar is represented as a complex algebraic number. ABC's, often referred to as the Koblitz curves, are defined by

$$y^2 + xy = x^3 + \alpha x^2 + 1 \tag{4.4}$$

with $\alpha = 0$ or $\alpha = 1$. The advantage provided by the Koblitz curves is that the `DOUBLE` operation in Algorithm 7 can be replaced with a second operation, namely

Algorithm 8 Generation of Binary NAF [31]

Input: Positive integer k

Output: $k' = \text{NAF}(k)$

$i \leftarrow 0$;

while $(k > 0)$ **do**

  if $(k \equiv 1 \pmod 2)$ **then**

    $k'_i \leftarrow 2 - (k \mod 4)$;

    $k \leftarrow k - k'_i$;

  else

    $k'_i \leftarrow 0$;

  $k \leftarrow k/2$;

  $i \leftarrow i + 1$;

Frobenius mapping, which is easier to perform.

If point (x, y) is on a Koblitz curve then it can be easily checked that (x^2, y^2) is also on the same curve. Moreover, these two points are related by the following Frobenius mapping

$$\tau(x, y) = (x^2, y^2)$$

where τ satisfies the quadratic equation

$$\tau^2 + 2 = \mu\tau. \tag{4.5}$$

In (4.5), $\mu = (-1)^{1-\alpha}$ and α is the curve parameter in (4.4) and is 0 or 1 for the Koblitz curves.

The integer k can be represented with radix τ using signed representation. In this

case, the expansion is written

$$k = \kappa_{l-1}\tau^{l-1} + \cdots \kappa_1\tau + \kappa_0,$$

where $\kappa_i \in \{0, 1, \bar{1}\}$. Using this representation, Algorithm 7 can be rewritten, replacing the DOUBLE(Q) operation with τQ or a Frobenius mapping of Q. The modified algorithm is shown in Algorithm 9. Since τQ is computed by squaring the coordinates of Q, this suggests a possible speed up over the DOUBLE and ADD method.

Algorithm 9 Scalar Multiplication for τ-adic Integers

Input: Integer $k = (\kappa_{l-1}, \kappa_{l-2}, \ldots, \kappa_1, \kappa_0)_\tau$, Point P

Output: Point $Q = kP$

```
Q ← O;
if (κ_{l-1} ≠ 0) then
    Q ← κ_{l-1}P;
for i = l − 2 downto 0 do
    Q ← τQ;
    if (κ_i ≠ 0) then
        Q ← ADD(Q, κ_iP);
```

This complex representation of the integer can be improved further by computing its non-adjacent form. Solinas proved the existence of such a representation in [31] by providing an algorithm which computes the τ-adic non-adjacent form or τ-NAF of an integer. This algorithm is provided here in Algorithm 10. In most cases, the input to Algorithm 10 will be a binary integer, say k (i.e. $r_0 = k$ and $r_1 = 0$). If k has length l then TNAF(k) will have length $2l$, roughly twice the length of NAF(k).

Algorithm 10 Generation of τ-adic NAF [31]

Input: $r_0 + r_1\tau$ where $r_0, r_1 \in \mathbb{Z}$

Output: u =TNAF$(r_0 + r_1\tau)$

$i \leftarrow 0$;

while ($r_0 \neq 0$ or $r_1 \neq 0$) **do**

if ($r_0 \equiv 1 \pmod 2$) **then**

$u_i \leftarrow 2 - (r_0 - 2r_1 \mod 4)$;

$r_0 \leftarrow r_0 - u_i$;

else

$u_i \leftarrow 0$;

$t \leftarrow r_0$;

$r_0 \leftarrow r_1 + \mu r_0/2$;

$r_1 \leftarrow -t/2$;

$i \leftarrow i + 1$;

The length of the representation generated by Algorithm 10 can be reduced by either preprocessing the integer k, as is done in [31], or by post processing the result. A method for post processing the output of Algorithm 10 is presented here.

Remember that $\tau(x, y) = (x^2, y^2)$. Since $z^{2^m} = z$ for all $z \in \mathrm{GF}(2^m)$, it follows that

$$\tau^m(x, y) = (x^{2^m}, y^{2^m}) = (x, y).$$

This relation gives us the general equality

$$(\tau^m - 1)P \equiv 0$$

where P is a point on a Koblitz curve. As a result, any integer k expressed with radix τ can be reduced modulo $\tau^m - 1$ without changing the scalar multiple kP. This reduction is performed easily with a few polynomial additions. Consider the τ-adic integer

$$u = u_{2m-1}\tau^{2m-1} + \cdots + u_{m+1}\tau^{m+1} + u_m\tau^m + u_{m-1}\tau^{m-1} + \cdots + u_1\tau + u_0.$$

Factoring out τ^m wherever possible, the result is

$$\begin{aligned} u = &(u_{2m-1}\tau^{m-1} + \cdots + u_{m+1}\tau + u_m)\tau^m \\ &+(u_{m-1}\tau^{m-1} + \cdots + u_1\tau + u_0) \end{aligned}$$

Substituting τ^m with 1 and combining terms results in

$$u = ((u_{2m-1} + u_{m-1})\tau^{m-1} + \cdots + (u_{m+1} + u_1)\tau + (u_m + u_0).$$

The output of Algorithm 10 is approximately twice the length of the input but may be slightly larger. Assuming the length of the input to be approximately m symbols, the reduction method must be capable of reducing τ-adic integers with length slightly greater $2m$. Algorithm 11 describes this method for reduction.

Algorithm 11 Reduction mod τ^m

Input: $u = u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0$ with $m \leq l < 3m$

Output: v =REDUCE_TM(u)

$v \leftarrow 0$;

if $(l > 2m)$ **then**

$v \leftarrow (u_{l-1}\tau^{l-2m-1} + \cdots + u_{2m+1}\tau + u_{2m})$;

if $(l > m)$ **then**

$v \leftarrow v + (u_{2m-1}\tau^{m-1} + \cdots + u_{m+1}\tau + u_m)$;

$v \leftarrow v + (u_{m-1}\tau^{m-1} + \cdots + u_1\tau + u_0)$;

Now the result of Algorithm 11 has length m but is no longer in τ-adic NAF form. There may be adjacent non-zero symbols and the symbols are not restricted to the set $\{0, 1, \bar{1}\}$.

The input of Algorithm 10 is of the form $r_0 + r_1\tau$ where $r_0, r_1 \in \mathbb{Z}$. The output is

the τ-adic representation of the input. For $v \in \mathbb{Z}[\tau]$ we can write

$$\begin{aligned} v &= v_{m-1}\tau^{m-1} + \cdots + v_2\tau^2 + v_1\tau + v_0 \\ &= v_{m-1}\tau^{m-1} + \cdots + v_2\tau^2 + \text{TNAF}(v_1\tau + v_0) \end{aligned}$$

Now the two least significant symbols of v are in τ-adic NAF. Repeating this procedure for every bit in v the entire string can be converted to τ-adic NAF. This process is described in Algorithm 12.

Algorithm 12 Regeneration of τ-adic NAF

Input: $v = v_{m-1}\tau^{m-1} + \cdots + v_1\tau + v_0$

Output: w =REGEN_TNAF(v)

$w \leftarrow v$;
$i \leftarrow 0$;
while ($w_j \neq 0$ for some $j \geq i$) **do**
  if ($w_i == 0$) **then**
    $i \leftarrow i + 1$;
  else
    $t_0 \leftarrow w_i$;
    $t_1 \leftarrow w_{i+1}$;
    $w_i \leftarrow 0$;
    $w_{i+1} \leftarrow 0$;
    $w \leftarrow w$+TNAF($t_1\tau + t_0$);
    $i \leftarrow i + 1$;

The output of Algorithm 12 is in τ-adic NAF and has a length of approximately m symbols. If the result is larger than m symbols, it is possible to repeat Algorithms 11

and 12 to further reduce the length. Algorithms 10, 11 and 12 have been implemented in C and were used to generate test vectors for the prototype discussed later in this chapter. During testing, it was found that a single pass of these algorithms generates a τ-adic representation with average length of m and a maximum length of $m + 5$[1].

Like radix 2 NAF the τ-adic NAF uses the symbol set $\{1, 0, \bar{1}\}$ and has an average Hamming weight of approximately $l/3$ for an l-bit integer [31]. So Algorithm 9 has a running time of $l/3$ point additions and $l - 1$ Frobenius mappings.

4.2.3 Summary and Analysis

A point addition using López & Dahab's projective coordinates requires ten field multiplications, four field squarings and eight field additions. A point double requires five field multiplications, five field squarings and four field additions. Using this information, the run time for scalar multiplication can be written in terms of field operations. Typically scalar multiplication is measured in terms of field multiplications, inversions and squarings, ignoring the cost of addition. In the case of this architecture, field multiplication and squaring are completed quickly enough that the cost of field addition becomes significant. The run times using binary, binary NAF and τ-adic NAF representations are shown in Table 4.2. These values are based on the curve addition and doubling equations defined in (4.2) and (4.3) assuming arbitrary curve parameters α and β and the average Hamming weights discussed in the previous sections. For the case of τ-NAF, a Frobenius mapping is assumed to require three squaring operations. The symbols $\mathcal{M}$, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{I}$ correspond to field multiplication, squaring, addition and inversion respectively. In each case it is assumed that the length of the

[1]These are empirical rather than analytical results.

integer is approximately equal to m.

Table 4.2: Cost of Scalar Multiplication in terms of Field Operations

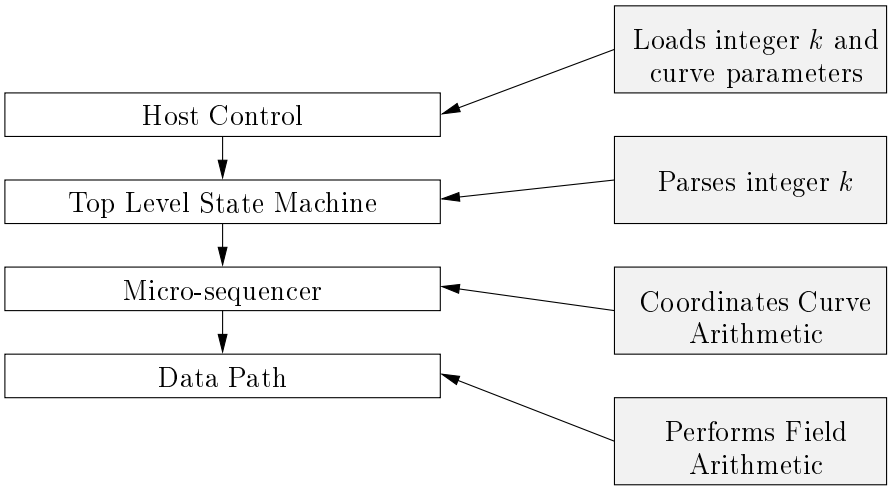
	Generic m	$m = 163$
Binary	$(10\mathcal{M} + 7\mathcal{S} + 8\mathcal{A})m + \mathcal{I}$	$1630\mathcal{M} + 1141\mathcal{S} + 1304\mathcal{A} + \mathcal{I}$
NAF	$(\frac{25}{3}\mathcal{M} + \frac{19}{3}\mathcal{S} + \frac{20}{3}\mathcal{A})m + \mathcal{I}$	$1359\mathcal{M} + 1033\mathcal{S} + 1087\mathcal{A} + \mathcal{I}$
τ-NAF	$(\frac{10}{3}\mathcal{M} + \frac{13}{3}\mathcal{S} + \frac{8}{3}\mathcal{A})m + \mathcal{I}$	$544\mathcal{M} + 706\mathcal{S} + 435\mathcal{A} + \mathcal{I}$

4.3 Co-processor Architecture

The architecture, which is detailed in this section, consists of several finite field arithmetic units, field element storage and control logic. All logic related to finite field arithmetic is optimized for specific field size and reduction polynomial. Internal curve computations are performed using López & Dahab's projective coordinate system. While generic curves are supported, the architecture is optimized specifically for the special Koblitz curves.

The processor's architecture consists of the data path and two levels of control. The lower level of control is composed of a micro-sequencer which holds the routines required for curve arithmetic such as `DOUBLE` and `ADD`. The top level control is implemented using a state machine which parses the scalar and invokes the appropriate routines in the lower level control. This hierarchical control is shown in Figure 4.1.

Figure 4.1: Co-Processor's Hierarchical Control Path

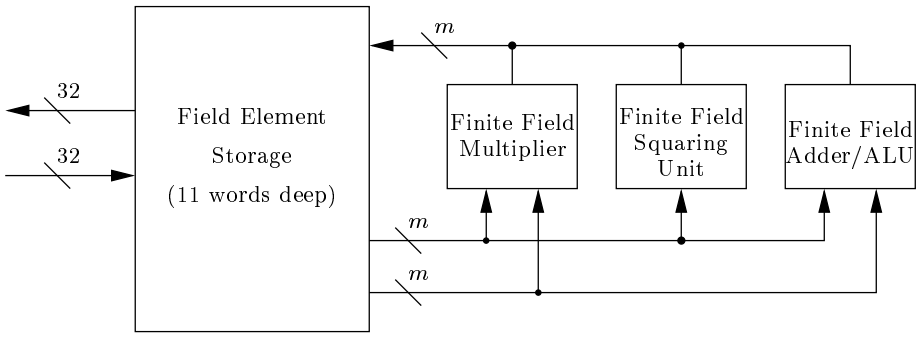


4.3.1 The Data Path

The data path of the co-processor consists of three finite field arithmetic units as well as space for operand storage. The arithmetic units include a multiplier, adder, and squaring unit. Each of these are optimized for a specific field and corresponding field polynomial. In an attempt to minimize time lost to data movement, the adder and multiplier are equipped with dual input ports which allow both operands to be loaded at the same time (the squaring unit requires a single operand and cannot benefit from an extra input bus). Similarly, the field element storage has two output ports used to supply data to the finite field units. In addition to providing field element storage, the storage unit provides the connection between the internal m-bit data path and the 32-bit external world. Figure 4.2 shows how the arithmetic units are connected to the storage unit.

The internal m-bit busses connecting the storage and arithmetic units are con-

Figure 4.2: Co-Processor Data-Path



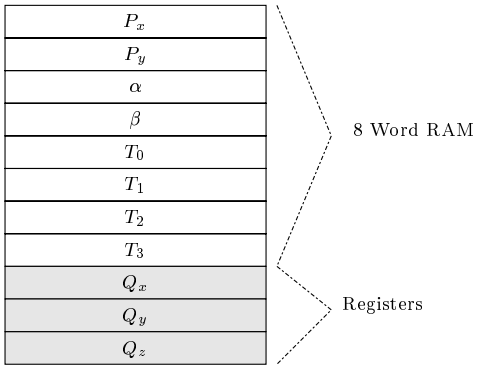
trolled to perform sequences of field operations. In this way the underlying curve operations `DOUBLE` and `ADD` as well as field inversion are performed.

Field Element Storage: The field element storage unit provides storage for curve points and parameters as well as temporary values. Parameters required to perform elliptic curve scalar multiplication include the field elements α and β and coordinates of the base point P. Storage will also be required for the coordinates of the scalar multiple Q. The point addition routine developed for this design also requires four temporary storage locations for intermediate values. Figure 4.3 shows how the storage space is organized.

The top eight field element storage locations are implemented using 32-bit dual-port RAMs generated by the Xilinx Coregen tool and the bottom three storage locations[2] are made of register files with 32-bit register widths. The dual 32-bit/m-bit interface support is achieved by instantiating $\lceil \frac{m}{32} \rceil$ dual-port storage blocks (either memories or register files) with 32-bit word widths as shown in Figure 4.4. The figure assumes $m = 163$. If the 32-bit storage locations in Figure 4.4 are viewed as a

[2]These locations are shaded gray in Figures 4.3 and 4.4.

Figure 4.3: Field Element Storage



matrix then the rows of the matrix hold the m-bit field words. Each 32-bit location is accessible by the 32-bit interface and each m-bit location is accessible by the m-bit interface. For simplicity sake the field elements are aligned at 32 byte boundaries.

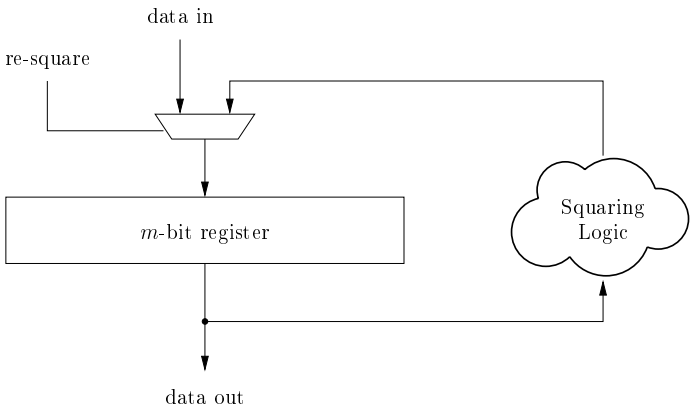
Figure 4.4: 32-bit/163-bit Address Map

RAM 5	RAM 4	RAM 3	RAM 2	RAM 1	RAM 0	
addr 5	addr 4	addr 3	addr 2	addr 1	addr 0	address 0
addr 13	addr 12	addr 11	addr 10	addr 9	addr 8	address 1
addr 21	addr 20	addr 19	addr 18	addr 17	addr 16	address 2
addr 29	addr 28	addr 27	addr 26	addr 25	addr 24	address 3
addr 37	addr 36	addr 35	addr 34	addr 33	addr 32	address 4
addr 45	addr 44	addr 43	addr 42	addr 41	addr 40	address 5
addr 53	addr 52	addr 51	addr 50	addr 49	addr 48	address 6
addr 61	addr 60	addr 59	addr 58	addr 57	addr 56	address 7
addr 69	addr 68	addr 67	addr 66	addr 65	addr 64	address 8
addr 77	addr 76	addr 75	addr 74	addr 73	addr 72	address 9
addr 85	addr 84	addr 83	addr 82	addr 81	addr 80	address 10

Computation of τQ: In addition to providing storage, the registers in the bottom three m-bit locations are capable of squaring the resident field element. This is accomplished by connecting the logic required for squaring directly to the output of the storage register. The squared result is then muxed in to the input of the

storage register and is activated with an enable signal. Figure 4.5 provides a diagram of this connection. This allows the squaring operations required to compute τQ to be performed in parallel. Furthermore, it eliminates the data movement otherwise required if the squaring unit were to be loaded and unloaded for each coordinate of Q. This provides significant performance improvement when using Koblitz curves.

Figure 4.5: Efficient Frobenius Mapping



4.3.2 The Micro-sequencer

The micro-sequencer controls the data movement between the field element storage and the finite field arithmetic units. In addition to the fundamental load and store operations, it supports control instructions such as jump and branch. The following list briefly summarizes the instruction set supported by the micro-sequencer.

- ld: Load operand(s) from storage location into specified field arithmetic unit.
- st: Store result from field arithmetic unit into specified storage location.
- j: Jump to specified address in the micro-sequencer.

- jr: Jump to specified micro-sequencer address and push current address onto the program counter stack.
- ret: Return to micro-sequencer address. The address is supplied by the program counter stack.
- bne: Branch if the last field elements loaded into the ALU are NOT equal.
- nop: Increment program counter but do nothing.
- set: Set internal counter to specified value.
- rsq: Resquares the contents of the squaring unit.
- dbnz: Decrement internal counter and branch if the new value of the counter is zero. This opcode also causes the contents of the squaring unit to be resquared.

A two-pass perl assembler was developed to generate the micro-sequencer bit code. The assembler accepts multiple input files with linked addresses and merges them into one file. This file is then used to generate the bit code. The multiple input file support allows different versions of the ROM code to be efficiently managed. Different implementations of the same micro-sequencer routine can be stored in different files allowing them to be easily selected at compile time.

Micro-Sequencer Routines

The micro-sequencer supports the curve arithmetic primitives, field inversion as well as a few other miscellaneous routines. The list below provides a summary of routines developed for use in the design.

- `POINT_ADD` (P, Q): Adds the elliptic curve points P and Q where P is represented in affine coordinates and Q is represented using projective coordinates. The result is given in projective coordinates.

- `POINT_SUB` (P, Q): Computes the difference $Q - P$. P is represented using affine coordinates and Q is represented using projective coordinates. The result is given in projective coordinates. This routine calls the `POINT_ADD` routine.

- `POINT_DBL` (Q): Doubles the elliptic curve point Q. Both Q and the result are in projective coordinates.

- `INVERT` (X): Computes the inverse of the finite field element X.

- `CONVERT` (Q): Computes the affine coordinates of an elliptic curve point Q given the point's projective coordinates. This routine calls the `INVERT` routine.

- `COPY_P2Q` (P, Q): Copies the x and y coordinates of point P to the x and y coordinates of point Q. The z coordinate of point Q is set to 1.

- `COPY_MP2Q` (P, Q): Computes the x and y coordinates of point $-P$ and copies them to the x and y coordinates of point Q. The z coordinate of point Q is set to 1.

Several versions of the `POINT_ADD` routine have been developed. The most generic one supports any curve over the field GF(2^m). In this version, the values of α and β are used when computing the sum of two points. This curve also checks if $Q \neq P$, $Q \neq -P$ and $Q \neq \mathcal{O}$. The second version of the point addition routine is optimized for a Koblitz curve by assuming α and β are equal to the NIST recommended values.

The number of field multiplications required to compute the addition of two points is reduced from 10 to 9. The third version of the routine is optimized for a Koblitz curve and also forgoes the checks of point Q. If the base point P has a large prime order and the integer k is less than this order[3], it will never be the case that $Q = \pm P$ or $Q = \mathcal{O}$. This final version of the routine is the fastest of the three routines and is the one used to achieve the results reported at the end of this chapter. The assembly code for each of these routines is included in the appendix.

4.3.3 Top Level Control

The routines listed above along with the `POINT_FRB`(Q) operation are invoked by the top level state machine. The `POINT_FRB`(Q) routine computes the Frobenius map of the point Q. This operation is not as complex as the other operations and is not implemented in the micro-sequencer. It is invoked by the top level state machine all the same.

The state machine parses the scalar k and calls the routines as needed. Since integers in NAF and τ-NAF require use of the symbol -1 (denoted $\bar{1}$), the scalar requires more than just an m-bit register for storage. In the implementation given here, each symbol in the scalar is represented using two bits; one for the magnitude and one for the sign. Table 4.3 provides the corresponding representation. For each bit k_i in the scalar k the magnitude is stored in the register $k_i^{(m)}$ and the sign is stored in register $k_i^{(s)}$. Table 4.4 provides example representations for integers in binary form, NAF, and τ-adic NAF using $m = 8$.

[3]These are fair assumptions since the security of the ECC implementation relies on these properties.

Table 4.3: Representation of the Scalar k

Symbol	Magnitude	Sign
0	0	-
1	1	0
$\bar{1}$	1	1

Table 4.4: Example Representations of the Scalar

k	$k^{(m)}$	$k^{(s)}$
$(01001100)_2$	$(01001100)_2$	$(00000000)_2$
$(0100\bar{1}010)_{NAF}$	$(01001010)_2$	$(00001000)_2$
$(0100\bar{1}010)_{\tau-NAF}$	$(01001010)_2$	$(00001000)_2$

The top level state machine is designed to support binary, NAF and τ-adic NAF representations of the scalar. This effectively requires the state machine to perform Algorithms 4, 7 and 9. By taking advantage of the similarities between these algorithms, the top level state machine can perform this task with the addition of a single mode. This is shown in Algorithm 13. The algorithm is written in terms of the underlying curve and field primitives provided by the micro-sequencer (listed in Section 4.3.2).

The first step of Algorithm 13 is to search for the first non-zero bit in $k^{(m)}$. Once found, either P or $-P$ is copied to Q depending on the sign of the non-zero bit. The **while** loop then iterates over all the remaining bits in the scalar performing "doubles and adds" or "Frobenius mappings and adds" depending on the mode. Since the curve

Algorithm 13 State Machine Algorithm

Inputs: $k^{(m)} = (k_{l-1}^{(m)}, k_{l-2}^{(m)}, \ldots, k_1^{(m)}, k_0^{(m)})_2$,
$k^{(s)} = (k_{l-1}^{(s)}, k_{l-2}^{(s)}, \ldots, k_1^{(s)}, k_0^{(s)})_2$,
Point P and *mode* (NAF or τ-NAF)

Output: Point $Q = kP$

$i \leftarrow l - 1$;
while $(k_i^{(m)} == 0)$ **do**
  $k \leftarrow i - 1$;
if $(k_i^{(s)} == 1)$ **then**
  COPY_MP2Q(P, Q);
else
  COPY_P2Q(P, Q);
$i \leftarrow i - 1$;
while $(i \geq 0)$ **do**
  if $(mode == \tau\text{-NAF})$ **then**
    $Q \leftarrow$ POINT_FRB(Q);
  else
    $Q \leftarrow$ POINT_DBL(Q);
  if $(k_i^{(m)} == 1)$ **then**
    if $(k_i^{(s)} == 1)$ **then**
      $Q \leftarrow$ POINT_SUB(Q, P);
    else
      $Q \leftarrow$ POINT_ADD(Q, P);
  $i \leftarrow i - 1$
$Q \leftarrow$ CONVERT(Q);

arithmetic is performed using projective coordinates, the result must be converted to affine coordinates at the end of computation.

4.3.4 Choice of Field Arithmetic Units

The use of redundant arithmetic units, specifically field multipliers, has been suggested in [3] and should be considered when designing an elliptic curve scalar multiplier. It seems the advantage provided remains purely theoretical. This can be seen by examining the top performing ECC multipliers in [12] and [28], both of which use a single field multiplier. Reasons for doing the same for this ECC accelerator are twofold. (1) One of the limiting factors for the performance of the design is data movement. As shown in Figures 4.6 and 4.7 the bus usage for point addition and point doubling is very high (83% and 80% respectively). If another multiplier is added to the design there may not be enough free bus cycles to capitalize on the extra computational power. For the field GF(2^{163}), the multiplier computes a product in four clock cycles and requires three cycles to load and unload the unit. If a second multiplier is added, then two multiplications can be completed in four cycles but six cycles are required to unload the multiplier. (2) Many of the multiplications in point addition and point doubling are dependent on each other and must be performed in sequence. For this reason, the second multiplier may sit idle much of the time. The combination of these observations seems to argue against the use of multiple field multiplication units in the design.

Figure 4.6: Utilization of Finite Field Units for Point Addition

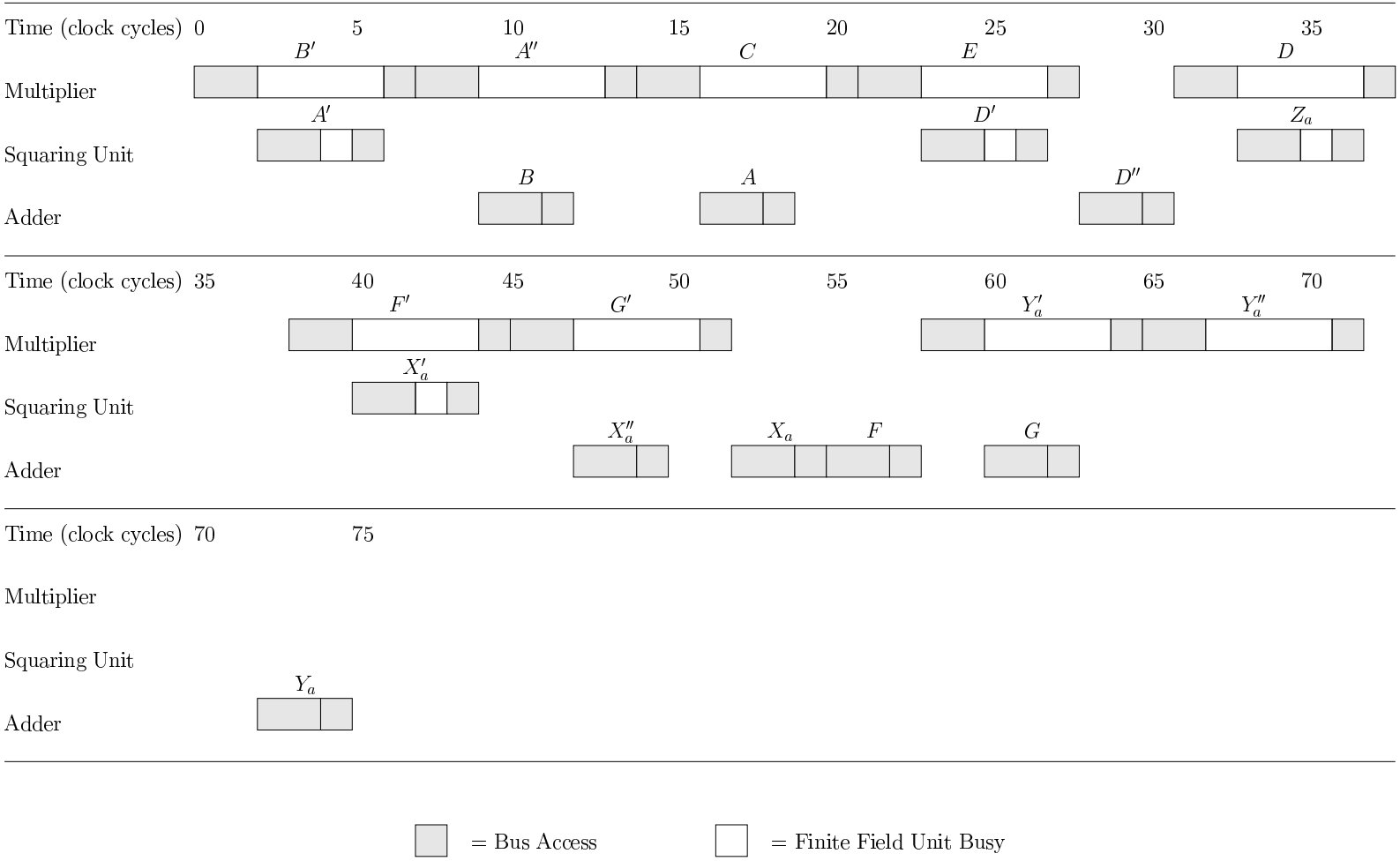


Figure 4.7: Utilization of Finite Field Units for Point Doubling

4.3.5 Usage Model

The following steps should be performed when using the module to compute the scalar multiple of an elliptic curve point.

- Load the base point P.
- Load the magnitude and sign of the scalar.
- Set the mode.
- Start computation.
- Wait for completion.
- Read out the resulting point Q.

During computation the base point P is preserved. If several scalar multiples of P need to be computed, P only needs to be loaded once. The same is true of the curve parameters α and β.

4.4 FPGA Prototype

A prototype of the architecture has been implemented for the field GF(2^{163}) using the NIST recommended field polynomial. The design was coded using Verilog HDL and synthesized using Synopsys FPGA Compiler II. Xilinx' Foundation software was used to place, route and time the netlist. The prototype was designed to run at 66 MHz on a Xilinx' Virtex 2000E FPGA.

The resulting design was verified on the Rapid Prototyping Platform (RPP) provided by Canadian Microelectronics Corporation (CMC) [6, 7]. The hardware/software system includes an ARM Integrator/LM-XCV600E+ (board with a Virtex 2000E FPGA) and an ARM Integrator/ARM7TDMI (board with an ARM7 core) connected by the ARM Integrator/AP board. The design was connected to an AHB slave interface which made it directly accessible by the ARM7 core. Stimulated by compiled C-code, the core read from and wrote to the prototype. The Integrator/AP's system clock had a maximum frequency of 50 MHz. In order to run our design at 66 MHz it was necessary to use the oscillator generated clock provided with the Integrator/LM-SCV600E+. The data headed to and coming from the design was passed across the two clock domains.

4.5 Results

Table 4.5 shows the performance in clock cycles of the prototypes field and curve operations. These values were gathered using a field multiplier digit size of $g = 41$. Note that the multiple instantiations of the squaring logic allow for the Frobenius mapping of a projective point to be completed in a single cycle. This significantly improves the performance of scalar multiplication when using the Koblitz curves.

The prototype of the scalar multiplier has been implemented using several digit sizes in the field multiplier. Table 4.6 reports the area consumption and resulting performance of the architecture given the different digit sizes. Table 4.7 provides a comparison of published performance results for scalar multiplication. The performance of 0.144 ms reported in [12] is the fastest reported scalar multiplication using

Table 4.5: Performance of Field and Curve Operations

Operation ($g = 41$)	# Cycles
Point Addition	79
Point Subtraction	87
Point Double	68
Frobenius Mapping	1

FPGA technology. The design presented in this thesis provides almost double (0.075 ms) the performance for the specific case of Koblitz curves.

The co-processor discussed in this thesis requires approximately half the CLBs used in the co-processor of [12] using the same FPGA. It must be noted that the co-processor presented in [12] is robust in that it supports all fields up to GF(2^{256}). In applications where support for a only single field size is required it is overkill to support elliptic curves over many fields. In scenarios such as this, this new elliptic curve co-processor offers an improved cost effective solution.

Table 4.6: Performance and Cost Results for Scalar Multiplication

Multiplier Digit Size	# LUTs	# FFs	Binary (ms)	NAF (ms)	τ-NAF (ms)
$g = 4$	6,144	1,930	1.107	0.939	0.351
$g = 14$	7,362	1,930	0.446	0.386	0.135
$g = 19$	7,872	1,930	0.378	0.329	0.113
$g = 28$	8,838	1,930	0.309	0.272	0.090
$g = 33$	9,329	1,930	0.286	0.252	0.083
$g = 41$	10,017	1,930	0.264	0.233	0.075

Table 4.7: Comparison of Published Results

Implementation	Field	FPGA	Scalar Mult. (ms)
S. Okada et. al. [26]	GF(2^{163})	Altera EPF10K250	45
Leong & Leung [19]	GF(2^{155})	Xilinx XCV1000	8.3
M. Bednara et. al. [3]	GF(2^{191})	Xilinx XCV1000	0.27
Orlando & Paar [28]	GF(2^{167})	Xilinx XCV400E	0.210
N. Gura et. al. [12]	GF(2^{163})	Xilinx XCV2000E	0.144
Our design ($g = 14$)	GF(2^{163})	Xilinx XCV2000E	0.135
Our design ($g = 41$)	GF(2^{163})	Xilinx XCV2000E	0.075

Chapter 5

Concluding Remarks

5.1 Summary and Contributions

In this thesis, the development of an elliptic curve cryptographic co-processor has been discussed. The co-processor takes advantage of multiplication and squaring arithmetic units which are based on the look-up table-based multiplication algorithm proposed in [13]. Field elements are represented with respect to the polynomial basis. While the base point and resulting scalar are given in affine coordinates, internal arithmetic is performed using projective coordinates. This choice of coordinate system allows the scalar multiple of a point to be computed with a single field inversion alleviating the need for a highly efficient inversion method. The processor was designed to support signed, unsigned and τ-NAF integer representation. All curves over a specific field are supported, but the architecture is optimized specifically for the Koblitz curves.

The feasibility and efficiency of the co-processor architecture has been demonstrated through a prototype implementation on an FPGA. The prototype has resulted

in record performance for elliptic curve scalar multiplication over the field GF(2^{163}). Contributions achieved in this work are as follows:

- A new high performance, low cost implementation of the field multiplier from [13].
- A new architecture designed for efficient Frobenius mappings through multiple instantiations of squaring logic.
- A high performance implementation of Itoh & Tsujii's inversion method.
- Overall performance for the elliptic curve co-processor is 0.075 micro-seconds for a single elliptic curve scalar multiplications.

5.2 Future Work

In the future it is intended to extend field support to several field sizes. Ideally, the architecture would support all NIST recommended fields simultaneously. Logic would be reused wherever possible. Extra logic would be limited to certain parts of the squaring and multiplication units which are dependent on the field reduction polynomial.

Appendix A

Micro-code supporting Curve Arithmetic and Field Inversion

This appendix includes the assembly code written to support elliptic curve point addition, point doubling, and field inversion, along with a few other operations. Note that there are multiple point addition and inversion routines.

A.1 Point Addition

The following three routines perform elliptic curve point addition. The first is the most generic and supports all curves with arbitrary α and β. The second routine is optimized for the NIST Koblitz over GF(2^{163}). The third routine is also optimized for the NIST Koblitz curve, but also forgoes integrity checking of point Q.

A.1.1 Generic Point Addition

```
//----------------------------
// Generic Point Add Routine
//----------------------------

                              // Is Q == identity?
ld  (ADD, QX, ZRO);  PTADD    // Is x1 == 0?
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
bne (ADD, PTADD_L3);          // (Q!=identity)->cont. with add.
ld  (ADD, QY, ZRO);           // Is y1 == 0?
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
bne (ADD, PTADD_L3);          // (Q!=identity)->cont. with add.
ld  (ADD, PX, ZRO);           // x2 + 0
st  (ADD, QX);                // Read x2 into location for x1
ld  (ADD, PY, ZRO);           // x2 + 0
st  (ADD, QY);                // Read y2 into location for y1
st  (ONE, QZ);                // Set z1 to a one
ret ();                       // Return

                              // Start the Point Addition
ld  (MLT, PX, QZ);   PTADD_L3 // Start B'  = x2*z1
ld  (SQR, QZ);                // Start A'  = z1^2
st  (SQR, T0);                // Read  A'
st  (MLT, T3);                // Read  B'
ld  (MLT, T0, PY);            // Start A'' = y2*A'
ld  (ADD, T3, QX);            // Start B   = B' + x1

                              // Is px == qx?
nop ();                       // dead cycle
```

```
nop ();                         // dead cycle
nop ();                         // dead cycle
nop ();                         // dead cycle
bne (ADD, PTADD_L1);            // If B' != x1 then branch


                                // Is py == qy?
st  (MLT, T2);                  // Read  A''
ld  (ADD, T2, QY);              // Start A'' + y1 ?= 0
nop ();                         // dead cycle
nop ();                         // dead cycle
nop ();                         // dead cycle
nop ();                         // dead cycle
bne (ADD, PTADD_L2);            // If A'' != y1 then branch

                                // Case: P == Q
jr  (PTDBL);                    // Jump to Point Double Routine
ret ();                         // We are done... so return

                                // Case: P == -Q
st  (ZRO, QX);          PTADD_L2 // x1 = 0
st  (ZRO, QY);                  // y1 = 0
ret ();                         // Return

                                // Case: P != Q and P != -Q
st  (ADD, T1);          PTADD_L1 // Read  B
st  (MLT, T2);                  // Read  A''
ld  (MLT, QZ, T1);              // Start C   = z1*B
ld  (ADD, T2, QY);              // Start A   = A'' + y1
st  (ADD, T2);                  // Read  A
st  (MLT, QZ);                  // Read  C
ld  (MLT, QZ, T2);              // Start E   = A*C
ld  (SQR, T1);                  // Start D'  = B^2
st  (SQR, T1);                  // Read  D'
st  (MLT, T3);                  // Read  E
ld  (MLT, A,  T0);              // Start D'' = a*A'
```

```
st  (MLT, T0);              // Read  D''
ld  (ADD, QZ, T0);          // Start D''' = C + D''
st  (ADD, T0);              // Read  D'''
ld  (MLT, T1, T0);          // Start D   = D'*D'''
ld  (SQR, QZ);              // Start z3   = C^2
st  (SQR, QZ);              // Read  z3
st  (MLT, T0);              // Read  D
ld  (MLT, PX, QZ);          // Start F'  = x2*z3
ld  (SQR, T2);              // Start x3' = A^2
st  (SQR, QX);              // Read  x3'
st  (MLT, QY);              // Read  F'
ld  (MLT, PY, QZ);          // Start G'  = y2*z3
ld  (ADD, QX, T0);          // Start x3''= x3' + D
st  (ADD, QX);              // Read  x3''
st  (MLT, T1);              // Read  G'
ld  (ADD, QX, T3);          // Start x3  = x3'' + E
st  (ADD, QX);              // Read  x3
ld  (ADD, QY, QX);          // Start F   = F' + x3
st  (ADD, QY);              // Read  F
ld  (MLT, T3, QY);          // Start y3' = E*F
ld  (ADD, T1, QX);          // Start G   = G' + x3
st  (ADD, T1);              // Read  G
st  (MLT, QY);              // Read  y3'
ld  (MLT, T1, QZ);          // Start y3''= z3*G
st  (MLT, T1);              // Read  y3''
ld  (ADD, QY, T1);          // Start y3  = y3' + y3''
st  (ADD, QY);              // Read  y3
ret ();                     // Return to base
```

A.1.2 Koblitz Curve Point Addition

```
//---------------------------------
// Koblitz Curve Point Add Routine
//---------------------------------

                              // Is Q == identity?
ld  (ADD, QX, ZRO);  PTADD    // Is x1 == 0?
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
bne (ADD, PTADD_L3);          // (Q!=identity)->cont. with add.
ld  (ADD, QY, ZRO);           // Is y1 == 0?
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
bne (ADD, PTADD_L3);          // (Q!=identity)->cont. with add.
ld  (ADD, PX, ZRO);           // x2 + 0
st  (ADD, QX);                // Read x2 into location for x1
ld  (ADD, PY, ZRO);           // x2 + 0
st  (ADD, QY);                // Read y2 into location for y1
st  (ONE, QZ);                // Set z1 to a one
ret ();                       // Return

                              // Start the Point Addition
ld  (MLT, PX, QZ);   PTADD_L3 // Start B'  = x2*z1
ld  (SQR, QZ);                // Start A'  = z1^2
st  (SQR, T0);                // Read  A'
st  (MLT, T3);                // Read  B'
ld  (MLT, T0, PY);            // Start A'' = y2*A'
ld  (ADD, T3, QX);            // Start B   = B' + x1

                              // Is px == qx?
nop ();                       // dead cycle
```

```
nop ();                         // dead cycle
nop ();                         // dead cycle
nop ();                         // dead cycle
bne (ADD, PTADD_L1);            // If B' != x1 then branch


                                // Is py == qy?
st  (MLT, T2);                  // Read  A''
ld  (ADD, T2, QY);              // Start A'' + y1 ?= 0
nop ();                         // dead cycle
nop ();                         // dead cycle
nop ();                         // dead cycle
nop ();                         // dead cycle
bne (ADD, PTADD_L2);            // If A'' != y1 then branch

                                // Case: P == Q
jr  (PTDBL);                    // Jump to Point Double Routine
ret ();                         // We are done... so return

                                // Case: P == -Q
st  (ZRO, QX);          PTADD_L2 // x1 = 0
st  (ZRO, QY);                  // y1 = 0
ret ();                         // Return

                                // Case: P != Q and P != -Q
st  (ADD, T1);          PTADD_L1 // Read  B
st  (MLT, T2);                  // Read  A''
ld  (MLT, QZ, T1);              // Start C   = z1*B
ld  (ADD, T2, QY);              // Start A   = A'' + y1
st  (ADD, T2);                  // Read  A
st  (MLT, QZ);                  // Read  C
ld  (MLT, QZ, T2);              // Start E   = A*C
ld  (SQR, T1);                  // Start D'  = B^2
st  (SQR, T1);                  // Read  D'
st  (MLT, T3);                  // Read  E
ld  (ADD, QZ, T0);              // Start D'' = C +  (aA') but a = 1
```

```
st  (ADD, T0);              // Read  D''
ld  (MLT, T1, T0);          // Start D   = D'*D''
ld  (SQR, QZ);              // Start z3   = C^2
st  (SQR, QZ);              // Read  z3
st  (MLT, T0);              // Read  D
ld  (MLT, PX, QZ);          // Start F'  = x2*z3
ld  (SQR, T2);              // Start x3' = A^2
st  (SQR, QX);              // Read  x3'
st  (MLT, QY);              // Read  F'
ld  (MLT, PY, QZ);          // Start G'  = y2*z3
ld  (ADD, QX, T0);          // Start x3''= x3' + D
st  (ADD, QX);              // Read  x3''
st  (MLT, T1);              // Read  G'
ld  (ADD, QX, T3);          // Start x3  = x3'' + E
st  (ADD, QX);              // Read  x3
ld  (ADD, QY, QX);          // Start F   = F' + x3
st  (ADD, QY);              // Read  F
ld  (MLT, T3, QY);          // Start y3' = E*F
ld  (ADD, T1, QX);          // Start G   = G' + x3
st  (ADD, T1);              // Read  G
st  (MLT, QY);              // Read  y3'
ld  (MLT, T1, QZ);          // Start y3''= z3*G
st  (MLT, T1);              // Read  y3''
ld  (ADD, QY, T1);          // Start y3  = y3' + y3''
st  (ADD, QY);              // Read  y3
ret ();                     // Return to base
```

A.1.3 Efficient Koblitz Curve Point Addition

```
//-----------------------------------------------------
// Koblitz Curve Point Add Routine with out checking Q
//-----------------------------------------------------

                              // Start the Point Addition
ld  (MLT, PX, QZ);   PTADD    // Start B'  = x2*z1
ld  (SQR, QZ);                // Start A'  = z1^2
st  (SQR, T0);                // Read  A'
st  (MLT, T3);                // Read  B'
ld  (MLT, T0, PY);            // Start A'' = y2*A'
ld  (ADD, T3, QX);            // Start B   = B' + x1
st  (ADD, T1);                // Read  B
st  (MLT, T2);                // Read  A''
ld  (MLT, QZ, T1);            // Start C   = z1*B
ld  (ADD, T2, QY);            // Start A   = A'' + y1
st  (ADD, T2);                // Read  A
st  (MLT, QZ);                // Read  C
ld  (MLT, QZ, T2);            // Start E   = A*C
ld  (SQR, T1);                // Start D'  = B^2
st  (SQR, T1);                // Read  D'
st  (MLT, T3);                // Read  E
ld  (ADD, QZ, T0);            // Start D'' = C +  (aA') but a = 1
st  (ADD, T0);                // Read  D''
ld  (MLT, T1, T0);            // Start D   = D'*D''
ld  (SQR, QZ);                // Start z3   = C^2
st  (SQR, QZ);                // Read  z3
st  (MLT, T0);                // Read  D
ld  (MLT, PX, QZ);            // Start F'  = x2*z3
ld  (SQR, T2);                // Start x3' = A^2
st  (SQR, QX);                // Read  x3'
st  (MLT, QY);                // Read  F'
ld  (MLT, PY, QZ);            // Start G'  = y2*z3
ld  (ADD, QX, T0);            // Start x3''= x3' + D
```

```
st  (ADD, QX);              // Read  x3''
st  (MLT, T1);              // Read  G'
ld  (ADD, QX, T3);          // Start x3  = x3'' + E
st  (ADD, QX);              // Read  x3
ld  (ADD, QY, QX);          // Start F   = F' + x3
st  (ADD, QY);              // Read  F
ld  (MLT, T3, QY);          // Start y3' = E*F
ld  (ADD, T1, QX);          // Start G   = G' + x3
st  (ADD, T1);              // Read  G
st  (MLT, QY);              // Read  y3'
ld  (MLT, T1, QZ);          // Start y3''= z3*G
st  (MLT, T1);              // Read  y3''
ld  (ADD, QY, T1);          // Start y3  = y3' + y3''
st  (ADD, QY);              // Read  y3
ret ();                     // Return to base
```

A.2 Point Doubling

The following routine computes the double of an elliptic curve point.

```
//------------------------
// Point Double Routine
//------------------------

                              // Is Q == identity?
ld  (ADD, QX, ZRO);  PTDBL    // Is x1 == 0?
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
bne (ADD, PTDBL_L1);          // (Q!=identity)->cont. with add.
ld  (ADD, QY, ZRO);           // Is y1 == 0?
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
nop ();                       // dead cycle
bne (ADD, PTDBL_L1);          // (Q!=identity)->cont. with add.
ret ();                       // Return to base

ld  (SQR, QZ);       PTDBL_L1 // Start z3'    = z1^2
st  (SQR, T0);                // Read  z3'
ld  (SQR, QX);                // Start z3''   = x1^2
st  (SQR, QX);                // Read  z3''
ld  (MLT, QX, T0);            // Start z3     = z3'*z3''
ld  (SQR, T0);                // Start x3'    = z3'^2
st  (SQR, T0);                // Read  x3'
st  (MLT, QZ);                // Read  z3
ld  (MLT, B,  T0);            // Start x3''   = b*x3'
st  (MLT, T0);                // Read  x3''
ld  (MLT, T0, QZ);            // Start y3'    = x3''z3
ld  (SQR, QX);                // Start x3'''  = z3''^2
st  (SQR, QX);                // Read  x3'''
```

```
st  (MLT, T1);              // Read  y3'
ld  (ADD, QX, T0);          // Start x3     = x3''' + x3''
st  (ADD, QX);              // Read  x3
ld  (SQR, QY);              // Start y3''   = y1^2
st  (SQR, QY);              // Read  y3''
ld  (ADD, QY, T0);          // Start y3'''  = y3'' + x3''
st  (ADD, QY);              // Read  y3'''
ld  (MLT, A,  QZ);          // Start y3^(4) = a * z3
st  (MLT, T2);              // Read  y3^(4)
ld  (ADD, QY, T2);          // Start y3^(5) = y3''' + y3^(4)
st  (ADD, QY);              // Read  y3^(5)
ld  (MLT, QX, QY);          // Start y3^(6) = x3*y3^(5)
st  (MLT, QY);              // Read  y3^(6)
ld  (ADD, QY, T1);          // Start y3     = y3^(6) + y3'
st  (ADD, QY);              // Read  y3
ret ();                     // Return to base
```

A.3 Field Inversion

The following two routines perform inversion over the field GF(2^{163}). This second routine relies on the fact that the `dbnz` opcode also re-squares the contents of the squaring unit.

A.3.1 Inversion by Square and Multiply

```
//------------------------
// Field Inversion
//------------------------

set (CTR1, 162);    FLDINV    // Set the counter
st  (ONE, T1);
ld  (SQR, T1);      FLDINV_L1 // Square T0
st  (SQR, T1);                //
ld  (MLT, T1, T0);            // Mult T0
st  (MLT, T1);                //
dbnz(CTR1, FLDINV_L1);        // Repeat 162 times
ld  (SQR, T1);                // One more squaring
st  (SQR, T1);
ret ();
```

A.3.2 Inversion by Itoh and Tsujii

```
//----------------------------------
// Field Inversion by Itoh & Tsujii
//----------------------------------


ld  (SQR, T0);      FLDINV    // -- square
st  (SQR, T1);                //
ld  (MLT, T1, T0);            // -- mult
st  (MLT, T1);                // T1 = a^(2^2 - 1)

ld  (SQR, T1);                // -- square
nop ();                       //
nop ();                       //
rsq ();                       // -- square
st  (SQR, T2);                //
ld  (MLT, T1, T2);            // -- mult
st  (MLT, T1);                // T1 = a^(2^4 - 1)

ld  (SQR, T1);                // -- square
st  (SQR, T2);                //
ld  (MLT, T2, T0);            // -- mult
st  (MLT, T1);                // T1 = a^(2^5 - 1)

ld  (SQR, T1);                // -- square
nop ();                       //
nop ();                       //
rsq ();                       // -- square
rsq ();                       // -- square
rsq ();                       // -- square
rsq ();                       // -- square
st  (SQR, T2);                //
ld  (MLT, T1, T2);            // -- mult
st  (MLT, T1);                // T1 = a^(2^10 - 1)
```

```
ld  (SQR, T1);                // -- square
nop ();                       //
set (CTR1, 8);                // -- 9 squarings
dbnz(CTR1, FLDINV_L1); FLDINV_L1 //
st  (SQR, T2);                //
ld  (MLT, T1, T2);            // -- mult
st  (MLT, T1);                // T1 = a^(2^20 - 1)

ld  (SQR, T1);                // -- square
nop ();                       //
set (CTR1, 18);               // -- 19 squarings
dbnz(CTR1, FLDINV_L2); FLDINV_L2 //
st  (SQR, T2);                //
ld  (MLT, T1, T2);            // -- mult
st  (MLT, T1);                // T1 = a^(2^40 - 1)

ld  (SQR, T1);                // -- square
nop ();                       //
set (CTR1, 38);               // -- 39 squarings
dbnz(CTR1, FLDINV_L3); FLDINV_L3 //
st  (SQR, T2);                //
ld  (MLT, T1, T2);            // -- mult
st  (MLT, T1);                // T1 = a^(2^80 - 1)

ld  (SQR, T1);                // -- square
st  (SQR, T2);                //
ld  (MLT, T2, T0);            // -- mult
st  (MLT, T1);                // T1 = a^(2^81 - 1)

ld  (SQR, T1);                // -- square
nop ();                       //
set (CTR1, 79);               // -- 80 squarings
dbnz(CTR1, FLDINV_L4); FLDINV_L4 //
st  (SQR, T2);                //
ld  (MLT, T1, T2);            // -- mult
st  (MLT, T1);                // T1 = a^(2^162 - 1)
```

```
ld  (SQR, T1);                  // -- square
st  (SQR, T1);                  // T1 = (a^162) = (a^-1)
ret ();
```

A.4 Coordinate Conversion

The following routine converts a point from its projective representation to its affine representation.

```
//------------------------
// Convert to Affine
//------------------------

ld  (ADD, QZ, ZRO); CNVAFF    // Copy z1 to T0
st  (ADD, T0);                //
jr  (FLDINV);                 // Compute (1/z1)
ld  (MLT, T1, QX);            // Start x1*(1/z1)
ld  (SQR, T1);                // Start (1/z1)^2
st  (SQR, T0);                // Read (1/z1)^2
st  (MLT, QX);                // Read x1 = x1*(1/z1)
ld  (MLT, T0, QY);            // Start y1*(1/z1)^2
st  (MLT, QY);                // Read y1 = y1*(1/z1)^2
st  (ONE, QZ);                // Set z1 to 1
ret ();
```

A.5 Copy Routines

The following two routines are used to initialize the Q register at the beginning of a scalar multiplication. The first loads Q with P and the second loads Q with $-P$.

A.5.1 Copy P to Q

```
//------------------------
// Copy P to Q
//------------------------

                              // Is Q == identity?
ld  (ADD, PX, ZRO);  CPYP2Q   // x2 + 0
st  (ADD, QX);                // Read x2 into location for x1
ld  (ADD, PY, ZRO);           // x2 + 0
st  (ADD, QY);                // Read y2 into location for y1
st  (ONE, QZ);                // Set z1 to a one
ret ();                       // Return
```

A.5.2 Copy $-P$ to Q

```
//------------------------
// Copy -P to Q
//------------------------

                              // Is Q == identity?
ld  (ADD, PX, ZRO);  CPYMP2Q  // x2 + 0
st  (ADD, QX);                // Read x2 into location for x1
ld  (ADD, PY, PX);            // x2 + y2
st  (ADD, QY);                // Read x2+y2 into location for y1
st  (ONE, QZ);                // Set z1 to a one
ret ();                       // Return
```

Appendix B

Tool Related Scripts and Setup Files

This appendix includes several tool related scripts and setup files which were used in the development of the ECC co-processor discussed in this thesis.

B.1 Synthesis Scripts

Listed in this section are two scripts used to synthesize the design. The file synth_compile.fst is the top level script which includes synt_constraints.fst. These scripts were written for Synopsys' FPGA Compiler II.

B.1.1 Synthesis Compile Scripts

```
#
# pmult_compile.fst
#
# This script synthesizes the pmult_top design for the Xilinx
# Vertex E FPGA.
#
# To run the script:
#
#    fc2_shell -f pmult_compile.fst
#

#
# Define variables
#
set proj pmult_proj
set top AHBAHBTop
set target VIRTEXE
set chip pmult_ahb
set export_dir exports
set report_dir reports

#
# Remove any old version of the project,
# and create the new project
# comment out this section to work on
# an existing project
#
exec rm -rf $proj
create_project -dir . $proj

#
# Setup project variables
#
proj_export_timing_constraint = "yes"
proj_enable_vpp = "yes"
```

```
#
# Setup default variables
#
default_clock_frequency = 66

#########################################################
#
# Identify the design source files
#
#########################################################

set SOURCEDIR  /secure2/jlutz/kp_unit/design
set AHBDIR     $SOURCEDIR/ahb_if/rtl_v
set PMULTDIR   $SOURCEDIR/pmult/rtl_v
set MULTDIR    $SOURCEDIR/mult/rtl_v
set ALUDIR     $SOURCEDIR/alu/rtl_v

# AHB files
add_file -format Verilog $AHBDIR/pmult_glue.v
add_file -format Verilog $AHBDIR/APBRegs.v
add_file -format Verilog $AHBDIR/APBIntcon.v
add_file -format Verilog $AHBDIR/AHB2APB.v
add_file -format Verilog $AHBDIR/AHBAPBSys.v
add_file -format Verilog $AHBDIR/AHBZBTRAM.v
add_file -format Verilog $AHBDIR/AHBDecoder.v
add_file -format Verilog $AHBDIR/AHBMuxS2M.v
add_file -format Verilog $AHBDIR/AHBAHBTop.v

# Top pmult files
add_file -format Verilog $PMULTDIR/pmult_defines.v
add_file -format Verilog $PMULTDIR/pmult_biu.v
add_file -format Verilog $PMULTDIR/pmult_logic.v
add_file -format Verilog $PMULTDIR/pmult_ptmlt_ctl.v
add_file -format Verilog $PMULTDIR/pmult_ram.v
add_file -format Verilog $PMULTDIR/pmult_q.v
```

```
add_file -format Verilog $PMULTDIR/pmult_top.v
add_file -format Verilog $PMULTDIR/pmult_useq.v

# ALU file(s)
add_file -format Verilog $ALUDIR/alu_top.v
add_file -format Verilog $ALUDIR/square_core.v

# Mult files
add_file -format Verilog $MULTDIR/mult_defines.v
add_file -format Verilog $MULTDIR/m_table.v
add_file -format Verilog $MULTDIR/mult_ctrl.v
add_file -format Verilog $MULTDIR/mult_top.v
add_file -format Verilog $MULTDIR/t_table.v


# The Memories
add_file -format Verilog $SOURCEDIR/pmult/user_cell/ram_8x32_d.v
add_file -format Verilog $SOURCEDIR/pmult/user_cell/ram_256x32_s_dist.v

add_file -format EDIF $SOURCEDIR/pmult/user_cell/ram_8x32_d.edn
add_file -format EDIF $SOURCEDIR/pmult/user_cell/ram_256x32_s_dist.edn


#
# Analyze all the source files and display the progress
#

analyze_file -progress

#
# Create a chip targetted for $target with the default part and
# speed grade.  The chip will be named $chip.  $top indicates
# the top level design.
#

create_chip -progress -target $target -name $chip $top
```

```
#
# Set the current chip to add constraints
#
current_chip $chip

#
# Read the constraints file
#
source synth_constraints.fst

#########################################################
#
# Optimize the current chip
#
#########################################################
set opt_chip [format "%s-Optimized" $chip]
optimize_chip -progress -name $opt_chip


#########################################################
#
# Generate Reports
#
#########################################################

# Set current chip
current_chip $opt_chip

# Create the reports directory
exec rm -rf $report_dir
exec mkdir -p $report_dir

# Show any error and warning messages for the chip
list_message > $report_dir/$top.errors_warnings.rpt
```

```
# Create a timing report
report_timing > $report_dir/$top.timing.rpt

# Create a few other reports
report_chip -force > $report_dir/$top.chip.rpt
report_project -all > $report_dir/$top.project.rpt


#########################################################
#
# Export Verilog netlist, PPR netlist and constraints
# to $export_dir
#
#########################################################

# create the export directory
exec rm -rf $export_dir
exec mkdir -p $export_dir

# Export synopsys db files
export_chip -dir $export_dir -db


# export edif netlists
export_chip -progress -dir $export_dir

# export verilog netlists
export_chip -progress -dir $export_dir -simulation \
VERILOG -primitive -timing_constraint

#
# Save and close the project
#
close_project

quit
```

B.1.2 Synthesis Constraints Script

```
#
# synth_constraints.fst
#
# This script sets constraints. It is called by the
# synth_compile.fst scripts.
#



set PMULT_TOP /$chip/uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top
set APB_TOP   /$chip/uAHBAPBSys

#
# Specify the clock waveform
#
set_clock -period 30 -rise 0 -fall 15 HCLK_PORT
set_clock -period 15 -rise 0 -fall 8 ECMULT_CLK_PORT



# Eliminate the boundaries of the field units. Hopefully this
# will allow synopsys to generate the fastest logic.
set_module_primitive optimize "$PMULT_TOP/pmult_logic/mult_top"
set_module_primitive optimize "$PMULT_TOP/pmult_logic/mult_top/t_table"
set_module_primitive optimize "$PMULT_TOP/pmult_logic/alu_top"
set_module_primitive optimize "$PMULT_TOP/pmult_logic/pmult_ram"
set_module_primitive optimize "$PMULT_TOP/pmult_logic"
set_module_primitive optimize "$PMULT_TOP/pmult_useq"
set_module_primitive optimize "$PMULT_TOP"
```

B.2 Place and Route Scripts

Listed in this section are several scripts which were used to place and route the design. The first is the top level script which takes the design as a synthesized netlist and returns the final bit file. The second scripts is the User Constraints File (UCF) file which is used to constrain the design.

B.2.1 Top Level Place and Route Script

```
#! /bin/csh -f

setenv BASE_NAME AHBAHBTop

#
# Merge the RAM edn files and the synopsys generated edf
# files into one ngd file.
#
ngdbuild        -p xcv2000e-6-fg680 \
                -aul \
                -sd ../../../pmult/user_cell \
                -uc $BASE_NAME.ucf \
                -dd . \
                $BASE_NAME.edf \
                $BASE_NAME.ngd

#
# Map the design to gates on the Virtex E FPGA
#
map             -p xcv2000e-6-fg680 \
                $BASE_NAME.ngd \
                -o map.ncd \
                $BASE_NAME.pcf

#
```

```
# Place and Route the Design
#
par             -w \
                -pl 5 \
                -rl 5 \
                map.ncd \
                $BASE_NAME.ncd \
                $BASE_NAME.pcf

#
# Run Static timing analysis
#
trce            $BASE_NAME.ncd \
                $BASE_NAME.pcf \
                -e 1000 \
                -o $BASE_NAME.twr

trce            $BASE_NAME.ncd \
                $BASE_NAME.pcf \
                -e 100 \
                -skew \
                -o $BASE_NAME-skew.twr


#
# Dump a verilog netlist and SDF file for timed simulation
#
ngdanno         -o $BASE_NAME.nga \
                -s 6 \
                -p $BASE_NAME.pcf \
                -report \
                $BASE_NAME.ncd \
                map.ngm

ngd2ver         -aka \
                -log $BASE_NAME.ngd2ver \
```

```
                -ne \
                -tm $BASE_NAME \
                -verbose -ul -w \
                -sdf_path . \
                $BASE_NAME.nga \
                $BASE_NAME.v



#
# Generate the bit file to be downloaded onto the FPGA
#
bitgen          $BASE_NAME.ncd \
                $BASE_NAME.bit \
                -l -m -w \
                -f bitgen.ut
```

B.2.2 User Constraints File

```
##################################################################
# Clock Information
##################################################################
NET "HCLK_PORT" TNM_NET = "HCLK_PORT" ;
TIMESPEC TS_HCLK_PORT = PERIOD "HCLK_PORT" 30 ns HIGH 50% ;

NET "ECMULT_CLK_PORT" TNM_NET = "ECMULT_CLK_PORT" ;
TIMESPEC TS_ECMULT_CLK_PORT = PERIOD "ECMULT_CLK_PORT" 15 ns HIGH 50% ;


##################################################################
# Group Information
##################################################################


INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_q/
qx_reg_reg<*>" TNM = "q_regs" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_q/
qy_reg_reg<*>" TNM = "q_regs" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_q/
qz_reg_reg<*>" TNM = "q_regs" ;

INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/mult_top/
a_reg<*>" TNM = "ffu_inputs" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/mult_top/
b_reg<*>" TNM = "ffu_inputs" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/alu_top/
a_reg<*>" TNM = "ffu_inputs" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/alu_top/
b_reg<*>" TNM = "ffu_inputs" ;


INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_biu/
ram_read_en_a_reg" TNM = "strg_read_ens" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_biu/
```

```
ram_read_en_b_reg" TNM = "strg_read_ens" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_biu/
q_read_en_a_reg" TNM = "strg_read_ens" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/pmult_biu/
q_read_en_b_reg" TNM = "strg_read_ens" ;

INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/mult_top/
t_table/t_odata_reg<*>" TNM = "mult_t_odata" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/mult_top/
a_reg<*>" TNM = "mult_a_reg" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/pmult_top/pmult_logic/mult_top/
b_reg<*>" TNM = "mult_b_reg" ;


INST "uAHBAPBSys/uAPBRegs/pmult_glue/ecm_odata_reg_reg<*>"
TNM = "ecmult_clk_buffer" ;

INST "uAHBAPBSys/uAPBRegs/pmult_glue/write_data_reg_reg<*>"
TNM = "ahb_clk_buffers" ;
INST "uAHBAPBSys/uAPBRegs/ecm_addr_reg<*>" TNM = "ahb_clk_buffers" ;
INST "uAHBAPBSys/uAPBRegs/pmult_glue/start_write_*" TNM =
"ahb_clk_buffers";
INST "uAHBAPBSys/uAPBRegs/pmult_glue/start_read_*" TNM =
"ahb_clk_buffers";


################################################################
# Path Information
################################################################

TIMESPEC TS_strg_rdens_2_ffus = FROM "strg_read_ens"
TO "ffu_inputs" 30 ns ;

TIMESPEC TS_strg_2_ffus = FROM "q_regs" TO "ffu_inputs" 30 ns ;

TIMESPEC TS_ttable_a = FROM "mult_a_reg" TO "mult_t_odata" 15 ns ;
```

```
TIMESPEC TS_ttable_b = FROM "mult_b_reg" TO "mult_t_odata" 15 ns ;

TIMESPEC TS_hclk2ecclk = FROM "ahb_clk_buffers"
TO "ECMULT_CLK_PORT" 60 ns;
TIMESPEC TS_ecclk2hclk = FROM "ecmult_clk_buffer"
TO "PADS" 60 ns;

TIMESPEC TS_P2P = FROM PADS TO PADS 30 ns ;
OFFSET = IN 30 ns BEFORE "HCLK_PORT" ;
OFFSET = OUT 30 ns AFTER "HCLK_PORT" ;


################################################################
# Port Information
################################################################

NET HADDR<31>     LOC=m2;
NET HADDR<30>     LOC=m1;
NET HADDR<29>     LOC=l4;
NET HADDR<28>     LOC=l3;
NET HADDR<27>     LOC=l2;
NET HADDR<26>     LOC=l1;
NET HADDR<25>     LOC=k4;
NET HADDR<24>     LOC=k3;
NET HADDR<23>     LOC=k2;
NET HADDR<22>     LOC=k1;
NET HADDR<21>     LOC=j4;
NET HADDR<20>     LOC=j3;
NET HADDR<19>     LOC=j2;
NET HADDR<18>     LOC=j1;
NET HADDR<17>     LOC=h4;
NET HADDR<16>     LOC=h3;
NET HADDR<15>     LOC=h2;
NET HADDR<14>     LOC=h1;
NET HADDR<13>     LOC=g4;
NET HADDR<12>     LOC=g3;
```

```
NET HADDR<11>     LOC=g2;
NET HADDR<10>     LOC=g1;
NET HADDR<9>      LOC=f4;
NET HADDR<8>      LOC=f3;
NET HADDR<7>      LOC=f2;
NET HADDR<6>      LOC=f1;
NET HADDR<5>      LOC=e3;
NET HADDR<4>      LOC=e2;
NET HADDR<3>      LOC=e1;
NET HADDR<2>      LOC=d3;
NET HADDR<1>      LOC=d2;
NET HADDR<0>      LOC=d1;

NET HDATA<31>     LOC=y1;
NET HDATA<30>     LOC=w1;
NET HDATA<29>     LOC=ab2;
NET HDATA<28>     LOC=aa4;
NET HDATA<27>     LOC=aa3;
NET HDATA<26>     LOC=w4;
NET HDATA<25>     LOC=w3;
NET HDATA<24>     LOC=w2;
NET HDATA<23>     LOC=v5;
NET HDATA<22>     LOC=v4;
NET HDATA<21>     LOC=v3;
NET HDATA<20>     LOC=v2;
NET HDATA<19>     LOC=v1;
NET HDATA<18>     LOC=u5;
NET HDATA<17>     LOC=u4;
NET HDATA<16>     LOC=u3;
NET HDATA<15>     LOC=u2;
NET HDATA<14>     LOC=u1;
NET HDATA<13>     LOC=t4;
NET HDATA<12>     LOC=t3;
NET HDATA<11>     LOC=t2;
NET HDATA<10>     LOC=t1;
NET HDATA<9>      LOC=r4;
```

```
NET HDATA<8>      LOC=r3;
NET HDATA<7>      LOC=r2;
NET HDATA<6>      LOC=p2;
NET HDATA<5>      LOC=p1;
NET HDATA<4>      LOC=n4;
NET HDATA<3>      LOC=n3;
NET HDATA<2>      LOC=n2;
NET HDATA<1>      LOC=n1;
NET HDATA<0>      LOC=m3;

NET CTRLCLK1<18>  LOC=av10;
NET CTRLCLK1<17>  LOC=au10;
NET CTRLCLK1<16>  LOC=at10;
NET CTRLCLK1<15>  LOC=aw9;
NET CTRLCLK1<14>  LOC=av9;
NET CTRLCLK1<13>  LOC=au9;
NET CTRLCLK1<12>  LOC=at9;
NET CTRLCLK1<11>  LOC=aw8;
NET CTRLCLK1<10>  LOC=av8;
NET CTRLCLK1<9>   LOC=au8;
NET CTRLCLK1<8>   LOC=at8;
NET CTRLCLK1<7>   LOC=aw7;
NET CTRLCLK1<6>   LOC=av7;
NET CTRLCLK1<5>   LOC=au7;
NET CTRLCLK1<4>   LOC=at7;
NET CTRLCLK1<3>   LOC=aw6;
NET CTRLCLK1<2>   LOC=av6;
NET CTRLCLK1<1>   LOC=au6;
NET CTRLCLK1<0>   LOC=at6;
NET PWRDNCLK1     LOC=av19;

NET CTRLCLK2<18>  LOC=av15;
NET CTRLCLK2<17>  LOC=au15;
NET CTRLCLK2<16>  LOC=at15;
NET CTRLCLK2<15>  LOC=aw14;
NET CTRLCLK2<14>  LOC=av14;
```

```
NET CTRLCLK2<13>  LOC=au14;
NET CTRLCLK2<12>  LOC=at14;
NET CTRLCLK2<11>  LOC=aw13;
NET CTRLCLK2<10>  LOC=av13;
NET CTRLCLK2<9>   LOC=au13;
NET CTRLCLK2<8>   LOC=at13;
NET CTRLCLK2<7>   LOC=aw12;
NET CTRLCLK2<6>   LOC=av12;
NET CTRLCLK2<5>   LOC=au12;
NET CTRLCLK2<4>   LOC=aw11;
NET CTRLCLK2<3>   LOC=av11;
NET CTRLCLK2<2>   LOC=au11;
NET CTRLCLK2<1>   LOC=at11;
NET CTRLCLK2<0>   LOC=aw10;
NET PWRDNCLK2     LOC=at21;

NET SDATA<31>     LOC=aw36;
NET SDATA<30>     LOC=av36;
NET SDATA<29>     LOC=au36;
NET SDATA<28>     LOC=aw35;
NET SDATA<27>     LOC=av35;
NET SDATA<26>     LOC=aw34;
NET SDATA<25>     LOC=av34;
NET SDATA<24>     LOC=au34;
NET SDATA<23>     LOC=at34;
NET SDATA<22>     LOC=aw33;
NET SDATA<21>     LOC=av33;
NET SDATA<20>     LOC=au33;
NET SDATA<19>     LOC=at33;
NET SDATA<18>     LOC=aw32;
NET SDATA<17>     LOC=av32;
NET SDATA<16>     LOC=au32;
NET SDATA<15>     LOC=at32;
NET SDATA<14>     LOC=aw31;
NET SDATA<13>     LOC=av31;
NET SDATA<12>     LOC=au31;
```

```
NET SDATA<11>     LOC=at31;
NET SDATA<10>     LOC=aw30;
NET SDATA<9>      LOC=av30;
NET SDATA<8>      LOC=au30;
NET SDATA<7>      LOC=ar4;
NET SDATA<6>      LOC=ah1;
NET SDATA<5>      LOC=ag2;
NET SDATA<4>      LOC=ad3;
NET SDATA<3>      LOC=r1;
NET SDATA<2>      LOC=p3;
NET SDATA<1>      LOC=p4;
NET SDATA<0>      LOC=c2;

#NET SADDR<20>     LOC=al36; # reserved for expansion
NET SADDR<19>     LOC=am39;
NET SADDR<18>     LOC=am38;
NET SADDR<17>     LOC=am37;
NET SADDR<16>     LOC=am36;
NET SADDR<15>     LOC=an39;
NET SADDR<14>     LOC=an38;
NET SADDR<13>     LOC=an37;
NET SADDR<12>     LOC=an36;
NET SADDR<11>     LOC=ap39;
NET SADDR<10>     LOC=ap38;
NET SADDR<9>      LOC=ap37;
NET SADDR<8>      LOC=ap36;
NET SADDR<7>      LOC=ar39;
NET SADDR<6>      LOC=ar38;
NET SADDR<5>      LOC=ar37;
NET SADDR<4>      LOC=ar36;
NET SADDR<3>      LOC=at39;
NET SADDR<2>      LOC=at38;

NET SMODE         LOC=ah37;
NET SnWR          LOC=aj38;
NET SnWBYTE<3>    LOC=ak39;
```

```
NET SnWBYTE<2>    LOC=ak38;
NET SnWBYTE<1>    LOC=ak37;
NET SnWBYTE<0>    LOC=ak36;
NET SnCE          LOC=al39;
NET SCLK          LOC=al38;
NET SnOE          LOC=aj36;
NET SnCKE         LOC=aj39;
NET SADVnLD       LOC=aj37;

NET SW<7>         LOC=au17;
NET SW<6>         LOC=at17;
NET SW<5>         LOC=ar17;
NET SW<4>         LOC=aw16;
NET SW<3>         LOC=av16;
NET SW<2>         LOC=au16;
NET SW<1>         LOC=at16;
NET SW<0>         LOC=aw15;
NET LED<8>        LOC=b37;
NET LED<7>        LOC=at19;
NET LED<6>        LOC=aw18;
NET LED<5>        LOC=av18;
NET LED<4>        LOC=au18;
NET LED<3>        LOC=at18;
NET LED<2>        LOC=ar18;
NET LED<1>        LOC=aw17;
NET LED<0>        LOC=av17;
NET nPBUTT        LOC=AU19;

NET FnOE          LOC=aw29;   # tie high
NET FnWE          LOC=at30;   # tie high

NET nLMINT        LOC=AF38;

NET HDRID<3>      LOC=af36;
NET HDRID<2>      LOC=ag39;
NET HDRID<1>      LOC=ag38;
```

```
NET HDRID<0>      LOC=ag37;

NET RTCK          LOC=ac37;
NET TDO           LOC=ad39;
NET TCK           LOC=ac35;
NET TDI           LOC=ad38;

NET HCLK_PORT     LOC=a20;
NET HRESETn       LOC=ag36;

NET HSIZE<1>      LOC=ak4;
NET HSIZE<0>      LOC=ak3;
NET HTRANS<1>     LOC=ak2;
NET HTRANS<0>     LOC=ak1;
NET HRESP<1>      LOC=an3;
NET HRESP<0>      LOC=an2;
NET HREADY        LOC=an1;
NET HWRITE        LOC=am4;

#not currently used
NET HBUSREQ               LOC=a4;
NET HLOCK             LOC=a5;

NET ECMULT_CLK_PORT LOC=aw19;
```

Bibliography

[1] *Wireless Application Protocol - Version 1.0*, 1998.

[2] G. B. Agnew, R.C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Slected Areas in Communications*, 11:804–813, June 1993.

[3] Marcus Bednara, Michael Daldrup, Joachim von zur Gathen, Jamshid Shokrollahi, and Jurgen Teich. Implementation of elliptic curve cryptographic coprocessor over GF(2^m) on an FPGA. In *International Parallel and Distributed Processing Symposium: IPDPS Workshops*, April 2002.

[4] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[5] D. Chudnovsky and G. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factoring tests. *Advances in Applied Mathematics*, 1987.

[6] Canadian Microelectronics Corporation. *CMC Rapic-Prototyping Platform: Design Flow Guide*, 2002.

[7] Canadian Microelectronics Corporation. *CMC Rapic-Prototyping Platform: Installation Guide*, 2002.

[8] T. Dierks and C. Allen. *The TLS Protocol - Version 1.0 IETF RFC 2246*, 1999.

[9] Joseph A. Gallian. *Contemporary Abstract Algebra*. Houghton Mifflin Company, 1998.

[10] Lijun Gao, Sarvesh Shrivastava, and Gerald E. Sobelman. Elliptic curve scalar multiplier design using FPGAs. In *Cryptographic Hardware and Embedded Systems (CHES)*, 1999.

[11] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.

[12] Nils Gura, Sheueling Chang Shantz, Hans Eberle, Summit Gupta, Vipul Gupta, Daniel Finchelstein, Edouard Goupy, and Douglas Stebila. An end-to-end systems approach to elliptic curve cryptography. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[13] M. Anwarul Hasan. Look-up table-based large finite field multiplication in memory constrained cryptosystems. *IEEE Transactions on Computers*, 49(7), July 2000.

[14] IEEE. *P1363: Editorial Contribution to Standard for Public Key Cryptography*, February 1998.

[15] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and Computing*, 78(3):171–177, 1988.

[16] Brian King. An improved implementation of elliptic curves over GF(2^n) when using projective point arithmetic. In *Selected Areas in Cryptography*, 2001.

[17] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 1987.

[18] Neal Koblitz. CM curves with good cryptographic properties. In *Advances in Cryptography, Crypto '91*, pages 279–287. Springer-Verilag, 1991.

[19] Philip H. W. Leong and Ivan K. H. Leung. A microcoded elliptic curve processor using FPGA technology. *IEEE Transactions on VLSI Systems*, 10(5), October 2002.

[20] Julio Lopez and Ricardo Dahab. Improved algorithms for elliptic curve arithmetic in GF(2^n). In *Selected Areas in Cryptography*, pages 201–212, 1998.

[21] Robert J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1989.

[22] Alfred Menezes. Elliptic curve public key cryptosystems. *Kluwer Academic Publishers*, 1993.

[23] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press LLC, 1997.

[24] Victor Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptography, Crypto '85*, 1985.

[25] NIST. *FIPS 186-2 draft, Digital Signature Standard (DSS)*, 2000.

[26] Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka. Implementation of elliptic curve cryptographic coprocessor over GF(2^m) on an FPGA. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 25–40. Springer-Verlag, 2000.

[27] OpenSSL. *See http://www.openssl.org.*

[28] Gerardo Orlando and Christof Paar. A high-performance reconfigurable elliptic curve processor for GF (2^m). In *Cryptographic Hardware and Embedded Systems (CHES)*, 2000.

[29] Arash Reyhani-Masoleh. *Low Complexity and Fault Tolerant Arithmetic in Binary Extended Finite Fields.* PhD thesis, University of Waterloo, 2001.

[30] Martin Christopher Rosner. Elliptic curve cryptosystems on reconfigurable hardware. Master's thesis, Worcester Polytechnic Institute, 1998.

[31] Jerome A. Solinas. Improved algorithms for arithmetic on anomalous binary curves. In *Advances in Cryptography, Crypto '97*, 1997.

[32] S. Sutikno, R. Effendi, and A. Surya. Design and implemntation of arithmetic processor $F_{2^{155}}$ for elliptic curve cryptosystems. In *IEEE Asia-Pacific Conference on Circuits adn Systems*, pages 647–650, November 1998.