ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

Hardware Acceleration of Elliptic Curve Based Cryptographic Algorithms: Design and Simulation

BY

Mubarek Kedir

April, 2008

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

Hardware Acceleration of Elliptic Curve Based Cryptographic Algorithms: Design and Simulation

A thesis submitted to the School of Graduate Studies of Addis Ababa University in partial fulfillment for the Degree of Master of Science in Computer Engineering

By
Mubarek Kedir

Advisor
Dr. Manoj V.N.V

Addis Ababa

April 2008

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

Hardware Acceleration of Elliptic Curve Based Cryptographic Algorithms: Design and Simulation

BY

Mubarek Kedir

Approval by Board of Examiners

Dr. Mengesha Mamo_____	_____
Chairman, Department of Electrical and Computer Engineering	Signature
Dr. Manoj V.N.V_____	_____
Advisor	Signature
_____	_____
External Examiner	Signature
_____	_____
Internal Examiner	Signature

Acknowledgement

I would like to thank all those who helped me to finish this thesis. First, I would like to thank my advisor Dr. Manoj for his support and continuous comments and Prof. Santhanam ,who was alos my advisor, for introducing me to FPGA design using Xilinx. I am deeply grieved to lose Prof. Sanathanam. Second, I would like to thank my family for encouraging me through out the thesis especially my mother, Rukiya and Hanan . My profound gratitude also is to my friends Bisrat, Elias, Qudus and Fetahi for their material support. Acknowledgement is also due to my best friend Fitsum as discussing with him about my work was invaluable. Last but certainly not least, I would like to thank Kibre and Azeb for their motherly advice.

Finally I sincerely thank everybody who contributed to this achievement in direct or indirect manner.

Table of Contents

List of Figures

List of tables

List of Algorithms

Acronyms

ASICs	Application Specific Integrated Circuits
CLB	Configurable Logic Block
ECC	Elliptic Curve Cryptography
ECDH	Elliptic curve based Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
FPGA	Field Programmable Gate Array
GF	Galois Field
HDL	Hardware Description Language
IOB	Input/Output Block
LUT	Look Up Table
NIST	National (American) Institute of Standards and Technology
PI	Programmable interconnect
RSA	Riverst-Shamir-Adleman
VHDL	Very high speed integrated circuits HDL

Abstract

Elliptic curve cryptography (ECC) is an alternative to traditional public key cryptographic systems. Even though, RSA (Rivest-Shamir-Adleman) was the most prominent cryptographic scheme, it is being replaced by ECC in many systems. This is due to the fact that ECC gives higher security with shorter bit length than RSA. In Elliptic curve based algorithms elliptic curve point multiplication is the most computationally intensive operation. Therefore implementing point multiplication using hardware makes ECC more attractive for high performance servers and small devices. In this thesis FPGA accelerator for point multiplication over GF (2^{163}) is proposed. We designed and synthesized the point accelerator using Xilinx XCV2000 FPGA. Binary field arithmetic units from which the point accelerator is built are also designed and synthesized. Experimental results show that a single point multiplication executes in 47μs. This is a 161 fold speed up over software implementation. And it is also better than the fastest hardware accelerator published in the literature.

1 Introduction

1.1 Motivation

As the Internet expands, it will encompass not only server and desktop systems, but also large numbers of small devices such as cell phones. Communications among these systems are usually conducted in an accessible environment such as Internet and wireless networks. These expose them to potential attackers that could tamper with them, eavesdrop communications, alter transmitted data, or attach unauthorized devices to the network. These risks can be mitigated by employing strong cryptography to ensure authentication, authorization, data confidentiality, and data integrity.

Symmetric cryptography, which is computationally inexpensive, can be used to achieve some of these goals. However, it is inflexible with respect to key management as it requires pre-distribution of keys. On the other hand, public key cryptography allows for flexible key management, but requires a significant amount of computation. However, the computational capabilities of low-cost CPUs are very limited in terms of clock frequency, memory size, and power constraints.

Compared to RSA, the prevalent public-key scheme of the Internet today, Elliptic Curve Cryptography (ECC) offers smaller key sizes, faster computation, as well as memory, energy and bandwidth [2].

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 [14]. For each field a specific curve, along with a method for generating a pseudo-random curve, are supplied. These curves have been systematically 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 Statement of the problem

Scalar multiplication is the most time consuming operation in Elliptic curve based cryptosystems. Efficient implementation of ECC algorithms using software is not fast enough on server computers which give service to many users. Implementing this multiplication on hardware makes ECC protocols more attractive. While the general purpose microprocessor is doing its routine task the time consuming operations can be executed using co-processor designed on a special hardware such as FPGA.

1.3 Scope of work and objectives

In this thesis, performance of software implementation of scalar multiplication is measured first. Then hardware units are designed for multiplication, inversion, squaring and addition for binary fields. These finite field arithmetic units are then integrated together to create an elliptic curve cryptographic co-processor capable of computing the scalar multiplication on elliptic curves. Even though design of the co-processor and arithmetic units are optimized for a particular binary field, F(2^163), scalability is considered so that it might be used for the other NIST curves.

To measure the efficiency of the co-processor, the design is translated into a hardware description language (Verilog). Then simulation is done for functionality and timing analysis.

General objective

- To accelerate Elliptic curve cryptographic algorithms on hardware.

Specific objectives

- To implement and measure performance of scalar multiplication on software.

- Design and simulate finite arithmetic units for binary fields
- Integrate the finite arithmetic units into an efficient hardware scalar multiplier.
- Compare performance of the hardware multiplier with the software implementation and other related works.

1.4 Methodology

The following methodology is followed to design and simulate a hardware accelerator for the scalar multiplier.

Literature survey

As both Elliptic curve cryptography and reconfigurable computing are relatively new areas of study, a lot of time is spent on understanding both of them. The following are some of study made.

- Abstract algebra especially finite field arithmetic
- Elliptic curve cryptography
- Reconfigurable computing using FPGA
- Survey of related works

Software implementation of Scalar multiplication

- Using a cryptographic library called MIRACL , the scalar acceleration is implemented on software using C++ to show the effectiveness of the hardware accelerator .

Hardware acceleration on FPGA

- Hardware design and realization of FPGA for binary field arithmetic units and synthesis, timing and functional simulation using Xilinx software tool is done.
- Realization of scalar multiplier using FPGA.
- Comparison between the software and hardware realization.

1.5 Thesis outline

The rest of the work is organized as follows. Section 2 discusses related works. It summarizes the major hardware acceleration on Scalar multiplication. It also discusses those software implementations relevant to this work. Section 3 provides a review of the on mathematical back ground of Elliptic curve cryptography including fields, Groups, rings, binary field arithmetic and curve operations. Section 4 is devoted to hardware acceleration. It explains why hardware acceleration is needed and then presents about hardware design flown using FPGA. Section 5 explains how binary field arithmetic is designed. Section 6 extends the discussion of Section 5 by designing hardware for point multiplication. Section 7 summarizes the results of the synthesized hardware. Resource utilization and timing are also discussed. Finally, Section 8 concludes the paper and presents possible further works.

2 Literature Review

There are a number of works on Elliptic curve cryptography that aided the design and simulation of this work. These include software and hardware implementation of point multiplication, which is the major operation in Elliptic Curve Cryptography, as well as implementation of Galois field arithmetic. This chapter summarizes previous works in these areas.

2.1 Hardware implementation

Hardware implementation of Elliptic cryptographic systems results in higher performance as compared with the software implementation but with relatively low flexibility. Existing hardware implementations vary in the following aspects: GF (2^m), GF (P), key length (from 163 to 233 bits), platforms (FPGA and ASIC). In this section, we review some of the FPGA implementations of ECC over GF (2^m).

Martin Christopher made the first attempt to implement scalar multiplication using FPGA [19]. It is implemented on Xilinx XC4062XLPG475-1 and point multiplication takes 5.65 msec. The latency is almost the same as recent software implementations. The second reconfigurable elliptic curve co-processor is designed over GF (2^{163})[10]. The design consists of main controller, arithmetic unit controller and arithmetic units. The prototype of the processor has been implemented 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. This work used encoding for the scalar multiplier. The encoding is not implemented on hardware. For experimentation, output of software implementation encoding is used. Another hardware accelerator is also implemented over GF (2^{163}) [11]. The accelerator runs at 45 MHZ on Xilinx Virtex FPGA and takes 1.21 msec to perform a 163-bit elliptic scalar multiplication. In addition scalar multiplication is implemented using Montgomery Ladder method [12] and [13]. The method is suitable for parallel implementation of the finite field units. The latter used several multipliers and squaring units in each component of the scalar multiplier. The

resulting design is synthesized on Xilinx XCV2000E and a scalar multiplication takes 53µs. Its resource usage is higher than most works in this area.

In addition to the hardware implementations discussed above, there exist other FPGA implementations for binary fields in the literature, such as [5, 6, 8, 12, 13 and 25].

2.2 Software Implementation

Software implementations of Elliptic curve cryptographic systems are many. To make the implementations efficient various algorithms are suggested for arithmetic and curve level operations. In this section, only those works relevant to this work are summarized.

At the arithmetic level, multiplication and inversion are the two time consuming operations, inversion being many fold slower than multiplication. A lookup table based efficient multiplication is proposed in [21] and implemented and reported in [22]. Inversion can be implemented using square and multiplication method and an efficient method is proposed by T. Itoh and S. Tsujii [17].

An elliptic curve system is implemented for a key exchange protocol [20]. The implementation is simplified by choosing the curve parameter a equal to zero. The system architecture relies on arithmetic in GF (2^{155}) using polynomial representation and an optimized inversion algorithm based on Euclidean division. The implementation performed multiplication of an elliptic curve point in 7.8 milliseconds on a DEC Alpha 3000 RISC machine(64bit, 450MHZ clock speed, 256Mbyte RAM) .

2.3 Summary

Efficient hardware design comprises of two components. The first and obvious component is optimized (high speed with a given target device) hardware designed for the appropriate task. The second and highly important component is the underlying algorithm to be used in the hardware design.

As for the algorithm, we studied many algorithms. Among them a digit serial multiplier which is proposed in [21], efficient inversion algorithm due to Itoh and Tsujii[17] and Montgomery scalar multiplication by Lopez and Dahab [18] are the major ones.

Hardware implementations of scalar multiplication revised in this chapter can generally be grouped into two. The first group is similar to the works in [10]. Point multiplication acceleration is implemented by encoding the scalar multiplier and by using Montgomery scalar multiplication. The encoding is not implemented in hardware. It is good in resource utilization as well as latency. The second group which is similar to the works in [13] uses Montgomery ladder method for scalar multiplication. The algorithm is ideal for parallel computations. This property of the algorithm is used extensively in the design.

Both groups discussed have their own draw backs. The first one uses encoding for the scalar multiplier which complicates the hardware implementation. The second one uses multiple hardware units in the design hierarchy such as multipliers. Our work will alleviate these problems by using the Montgomery ladder method for scalar multiplication and using parallelism but utilizing the resource in an efficient manner.

3 Mathematical Background

Elliptic curve based cryptographic algorithms are implemented using point operations on the Elliptic curve: Addition, doubling and scalar multiplication. Among this scalar multiplication is the fundamental building block. Basically, it is this operation that will be implemented in this work. In this chapter, the mathematics behind elliptic curve scalar multiplication will be discussed.

3.1 Groups

In **[2]** a **group** G, sometimes denoted by {G, ·} is a set of elements with a binary operation, denoted by ·, that associates to each ordered pair (a, b) of elements in G an element (a · b) in G, such that the following axioms are obeyed:

The operator · is generic and can refer to addition, multiplication, or some other mathematical operation.

(A1) Closure: If a and b belong to G, then a · b is also in G.
(A2) Associative: a · (b · c) = (a · b) · c for all a, b, c in G.
(A3) Identity element: There is an element e in G such that a · e = e · a = a for all a in G.
(A4) Inverse element: For each a in G there is an element a' in G such that a · a' = a' · a = e.

If a group has a finite number of elements, it is referred to as a **finite group**, and the **order** of the group is equal to the number of elements in the group. Otherwise, the group is an **infinite group**.

A group is said to be abelian if it satisfies the following additional condition:

(A5) Commutative: a · b = b · a for all a, b in G.

Example:

The set of integers (positive, negative, and 0) under addition is an abelian group. The set of nonzero real numbers under multiplication is an abelian group.

3.1.1 Cyclic Group

We define exponentiation within a group as repeated application of the group operator, so that $a^3 = a \cdot a \cdot a$. Further, we define $a^0 = e$, the identity element; and $a^{-n} = (a')^n$. A group G is cyclic if every element of G is a power a^k (k is an integer) of fixed element a G. The element a is said to generate the group G, or to be a generator of G. A cyclic group is always abelian, and may be finite or infinite.

Example:

The additive group of integers is an infinite cyclic group generated by the element 1. In this case, powers are interpreted additively, so that n is the n^{th} power of 1.

3.1.2 Rings

A **ring** R, sometimes denoted by {R, +, x}, is a set of elements with two binary operations, called addition and multiplication, such that for all a, b, c in R the following axioms are obeyed:

Generally, we do not use the multiplication symbol, x, but denote multiplication by the concatenation of two elements.

(A1-A5) R is an abelian group with respect to addition; that is, R satisfies axioms A1 through A5. For the case of an additive group, we denote the identity element as 0 and the inverse of a as -a.

(M1) Closure under multiplication:	If a and b belong to R, then ab is also in R.
(M2) Associativity of multiplication:	a(bc) = (ab)c for all a, b, c in R.
(M3) Distributive laws:	a(b + c) = ab + ac for all a, b, c in R. (a + b)c = ac + bc for all a, b, c in R.

In essence, a ring is a set in which we can do addition, subtraction [a-b = a + (-b)], and multiplication without leaving the set.

Example:

With respect to addition and multiplication, the set of all n-square matrices over the real numbers is a ring.

A ring is said to be **commutative** if it satisfies the following additional condition:

(M4) Commutativity of multiplication: $ab = ba$ for all a, b in R.

Next, we define an **integral domain**, which is a commutative ring that obeys the following axioms:

(M5) Multiplicative identity:	There is an element 1 in R such that $a1 = 1a = a$ for all a in R.
(M6) No zero divisors:	If a, b in R and $ab = 0$, then either $a = 0$ or $b = 0$.

Example:

Let S be the set of integers, positive, negative, and 0, under the usual operations of addition and multiplication, S is an integral domain.

3.2 Fields

A **field** F, sometimes denoted by {F, +, x}, is a set of elements with two binary operations, called addition and multiplication, such that for all a, b, c in F the following axioms are obeyed:

(A1-M6) F is an integral domain; that is, F satisfies axioms A1 through A5 and M1 through M6.

(M7) Multiplicative inverse:	For each a in F, except 0, there is an element a^{-1} in F such that $aa^{-1} = (a^{-1})a = 1$.

In essence, a field is a set in which we can do addition, subtraction, multiplication, and division without leaving the set. Division is defined with the following rule: $a/b = a(b^{-1})$.

Example:

Familiar examples of fields are the rational numbers, the real numbers, and the complex numbers. Note that the set of all integers is not a field, because not every element of the set has a multiplicative inverse; in fact, only the elements 1 and -1 have multiplicative inverses in the integers.

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

- Prime fields : GF(p) where p is prime
- Binary fields: $GF(2^m)$ where m is large.

The work in this thesis is mainly based on binary field. Therefore, the discussion that follows will be specifically for this field.

As modular arithmetic is used in binary fields, simple definition and notations of modular arithmetic is discussed below.

Modular Arithmetic

Given any positive integer n and any nonnegative integer a, if we divide a by n, we get an integer quotient q and an integer remainder r that obey the following relationship:

$$a = qn + r \quad 0 \le r < n;\ q = \left\lfloor a/n \right\rfloor \ and\ r = a \bmod n \quad \textbf{(3.1)}$$

Where $\lfloor x \rfloor$ is the largest integer less than or equal to x.

Two integers a and b are said to be congruent modulo n, if (a mod n) = (b mod n). This is written as $a \equiv b \pmod n$.

3.2.1 Binary fields

Finite fields of order 2^m are called binary fields or characteristic-two finite fields. One way to construct $F(2^m)$ is to use a polynomial basis representation. Here, the elements of

$F(2^m)$ are the binary polynomials (polynomials whose coefficients are in the field $F(2) = \{0,1\}$) of degree at most m-1:

$$F(2^m) = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2 z^2 + a_1 z + a_0 : a_i \in \{0,1\}\} \quad \textbf{(3.2)}$$

An irreducible binary polynomial *f (z)* of degree *m* is chosen (such a polynomial exists for any *m* and can be efficiently found [1];). Irreducibility of *f (z)* means that *f (z)* cannot be factored as a product of binary polynomials each of degree less than *m*. Addition of field elements is the usual addition of polynomials, with coefficient arithmetic performed modulo 2. Multiplication of field elements is performed modulo the ir*reducible polynomial f (z)*. For any binary polynomial *a(z)*, *a(z)* mod *f (z)* shall denote the unique remainder polynomial r *(z)* of degree less than *m* obtained upon long division of *a(z)* by *f (z)*; this operation is called *reduction modulo f (z)*.

Example: (binary field $F(2^4)$) The elements of $F(2^4)$ are the 16 binary polynomials of degree at most 3:

0	z^2	z^3	$z^3 + z^2$
1	$z^2 + 1$	$z^3 + 1$	$z^3 + z^2 + 1$
z	$z^2 + z$	$z^3 + z$	$z^3 + z^2 + z$
z+1	$z^2 + z + 1$	$z^3 + z + 1$	$z^3 + z^2 + z + 1$

The following are some examples of arithmetic operations in $F(2^4)$ with reduction polynomial $f(z) = z^4 + z + 1$.

- Addition : $(z^3 + z^2 + 1) + (z^2 + z + 1) = z^3 + z$
- Subtraction: $(z^3 + z^2 + 1) - (z^2 + z + 1) = z^3 + z$ (Note that since -1 =1 in F(2), we have a = -a for all a Є F(2)).
- Multiplication: $(z^3 + z^2 + 1) \cdot (z^2 + z + 1) = z^2 + 1$ since

  $$(z^3 + z^2 + 1) \cdot (z^2 + z + 1) = z^5 + z + 1$$

  and $(z^5 + z + 1) \bmod (z^4 + z + 1) = z^2 + 1$.
- Inversion: $(z^3 + z^2 + 1)^{-1} = z^2$ since $(z^3 + z^2 + 1) \cdot (z^2) \bmod (z^4 + z + 1) = 1$

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 the 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 3-1 NIST recommended Finite Fields

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

3.3 Arithmetic over Binary Finite Fields

The elements of the binary field 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.

For the operations to follow let us define field elements a,b $\in$ GF(2^m) to form the polynomials

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

$B(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_2x^2 + b_1x + b_0$ respectively.

3.3.1 Addition/Subtraction

Addition of field elements is performed bitwise, and the sum of A(x) and B(x) given as

$$C(x) = A(x) + B(x) = \sum_{i=0}^{m-1} (a_i + b_i) \quad \textbf{(3.3).}$$

And the algorithm is given below.

Algorithm 3-1 Addition in GF(2^m)

INPUT: Binary polynomials A(x) and B(x) of degrees at most m-1
OUTPUT: C(x) = A(x) + B(x)
    For i from 0 to m-1 do
        $c_i = a_i + b_i$
Return (c)

Working in a field of characteristic two provides two advantages. First, the bit additions in $a_i + b_i$ in Algorithm 3.1 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 to propagate. Hence addition in GF(2^m) is considerably simpler to implement in hardware than in prime fields GF(p). 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.

3.3.2 Multiplication

The product of field elements a and b is written as

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

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

$$C(x) = b_{m-1} x^{m-1} A(x) + b_{m-2} x^{m-2} A(x) + \cdots + b_2 x^2 A(x) + b_1 x A(x) + b_0 A(x) \bmod F(x)$$

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

$$C(x) = (\cdots(((b_{m-1} A(x))x + b_{m-2} A(x))x + \cdots + b_1 A(x))x + b_0 A(x)) \bmod F(x) \quad \textbf{(3.4)}$$

Starting with the inner most parenthesis and moving out, Algorithm 3.2 finds the product of A(x) and B(x).

Many of the faster multiplications algorithms rely on the concept of group-level

Algorithm 3-2 Left-to-right field multiplication in GF(2^m)

INPUT: Binary polynomials A(x) and B(x) of degrees at most m-1
OUTPUT: C(x) = A(x) x B(x) mod F(x)

```
C(x) = 0
for i from m-1 to 0 do
        C(x) = xC(x) mod F(x)
        if(b_i = = 1) then
          C(x) = C(x) + A(x)

Return (c)
```

multiplication (in each iteration more than one bits of b is multiplying with a). Let g be an integer less than m and let $s = \lceil m/g \rceil$ (m is the order of the field, g the number of bits in the digit and s is the number of digits). If we define the polynomials

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

then the product of a and b is written as

$$C(x) = A(x) \times (x^{(s-1)g} B_{s-1}(x) + \cdots + x^g B_1(x) + B_0(x)) \bmod F(x)$$

If grouping similar to equation (3.4) is used and multiplication is done repeatedly with x^g we will get

$$C(x) = (\cdots((A(x)B_{s-1}(x))x^g + A(x)B_{s-2}(x))x^g + \cdots)x^g + A(x)B_0(x) \bmod F(x) \quad \textbf{(3.5)}$$

This will be computed in Algorithm 3.3.

Algorithm 3-3 Group level field multiplication in GF(2^m)

INPUT: Binary polynomials A(x) and B(x) of degrees at most m-1
OUTPUT: C(x) = A(x) x B(x) mod F(x)

C(x) = $B_{s-1}(x)A(x)$ mod F(x)
for t from s-2 to 0 do
    C(x) = $x^g C(x)$
    C(x) = $B_t(x)$ A(x) + C(x) mod F(x)

Return (c)

3.3.3 Squaring

Since squaring a binary polynomial is a linear operation, it is much faster than multiplying two arbitrary polynomials; i.e.

$$A(x)^2 = a_{m-1}x^{2m-2} + \cdots + a_2 x^4 + a_1 x^2 + a_0 \quad \textbf{(3.6)}$$

The binary representation of $A(x)^2$ is obtained by inserting a 0 bit between consecutive bits of the binary representation of A(z) as shown in Figure 3.1.[1]

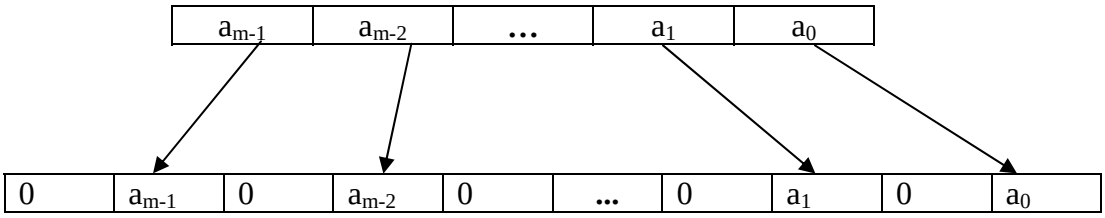


Figure 3-1 squaring a binary polynomial

3.3.4 Inversion

Fermat's theorem states that $a^{2^m-1} \equiv 1$ [2]. When a≠ 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 3.4 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

Algorithm 3-4 field inversion in GF(2^m) by square and multiply method

INPUT: field element a of degree m-1
OUTPUT: $b=a^{-1}$

```
b=a
for i from 1 to m-2 do
        b=b^2 × a

b = b^2

Return (b)
```

hardware dedicated specifically to inversion. The field multiplier can be used to perform all required field operations.

3.4 Elliptic Curve Arithmetic

Cryptographic mechanisms based on elliptic curves depend on arithmetic involving the points of the curve. Curve arithmetic is defined in terms of underlying field operations, the efficiency of which is essential. Efficient curve operations are likewise crucial to performance. [1]

The following figure illustrates module framework required for a protocol such as the Elliptic Curve Digital Signature Algorithm (ECDSA). The curve arithmetic not only is built on filed operations, but in some cases also relies on big number and modular arithmetic. ECDSA, for instance, uses a hash function and certain modular operations, but the computationally-expensive steps involve curve operations.

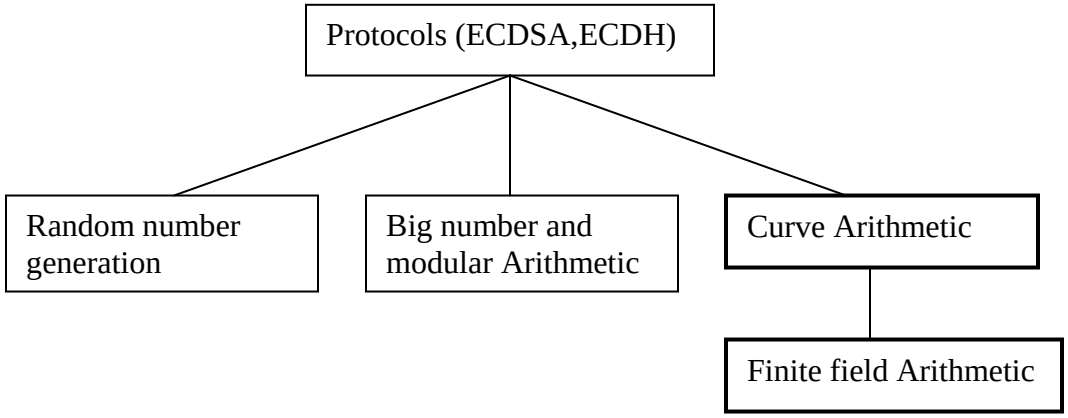


Figure 3-2 ECDSA support modules

The main concern of this work is on curve and finite field arithmetic. The field operations discussed in the previous section are used to perform arithmetic over an elliptic curve. There are different elliptic curves based on the simplified Weierstarass equation. In this thesis the elliptic curve defined by the non-supersingular Weierstrass equation for binary fields is used[14]. This curve is defined by the equation

$$y^2 + xy = x^3 + ax^2 + b \quad \textbf{(3.7)}$$

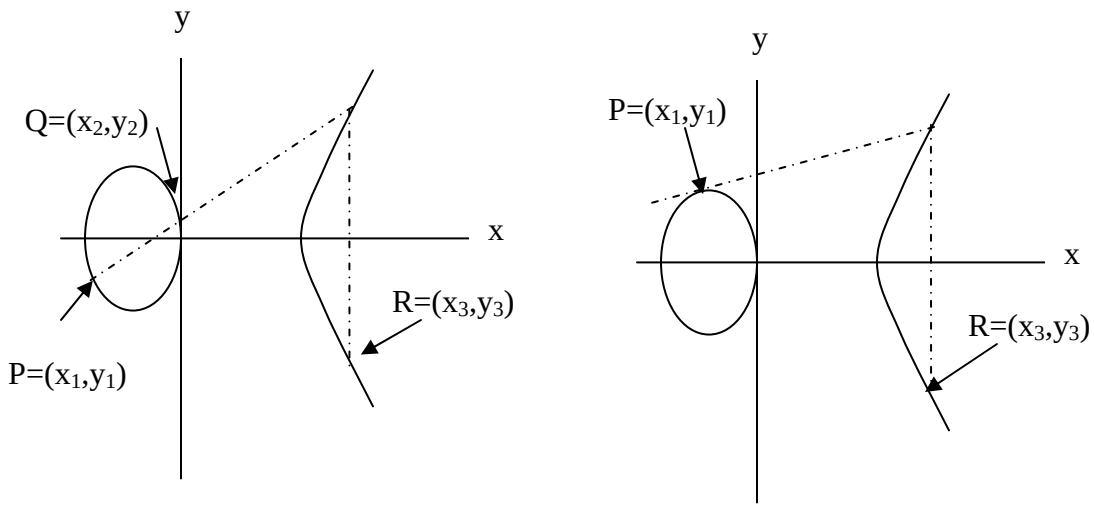
where x and y are elements of the field FG(2^m) and a and b are the curve parameters.

3.4.1 Elliptic Curve Group Law

There is a chord-and-tangent rule for adding two points in the curve equation (3.7) to give a third point on the same curve. Together with addition operation, the set of points on the curve forms an abelian group with the point at infinity, O, serving as its identity. It is such a group that is used in the construction of elliptic curve cryptographic systems [1].

The addition rule is best explained geometrically. Let P = (x1, y1) and Q = (x2, y2) be two distinct points on an elliptic curve E. Then the sum R, of P and Q, is defined as follows. First draw a line through P and Q; this line intersects the elliptic curve at a third point. Then R is the reflection of this point about the x-axis. This is depicted in Figure 3.3(a).

The double R, of P, is defined as follows. First draw the tangent line to the elliptic curve at P. This line intersects the elliptic curve at a second point. Then R is the reflection of this point about the x-axis. This is depicted in Figure 3.3(b).



a) Addition: P+Q=R b) Doubling: P+P=R

Figure 3-3 Geometric addition and doubling of elliptic curve points

Group Law for non-supersingular elliptic curves, $E(GF(2^m))$:

- **Identity** : $P + O = O + P = P$ for all $P \in E(GF(2^m))$
- **Negatives**: If $P = (x,y) \in E(GF(2^m))$, then $(x,y) + (x,x+y) = O$. The point $(x,x+y)$ is denoted by $-P$ and is called the negative of P; note that $-P$ is indeed a point in $GF(2^m)$. Also $-O = O$.
- **Point addition**: Let $P = (x_1, y_1) \in E(GF(2^m))$ and $Q = (x_2, y_2) \in E(GF(2^m))$, where $P \neq Q$. Then $P + Q = (x_3,y_3)$, where

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \textit{and} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

with $\lambda = (y_1 + y_2) \big/ x_1 + x_2$

- **Point doubling**: Let P = $(x_1+y_1) \in E(GF(2^m))$, where $P \neq -P$. Then 2P = (x_3, y_3), where

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + b/x_1^2 \quad and \quad y_3 = x_1^2 + \lambda x_3 + x_3$$

with $\lambda = x_1 + y_1/x_1$.

3.4.2 Scalar multiplication

Scalar multiplication (point multiplication) is the major building block of elliptic curve cryptographic systems. It is basically adding a point to itself arbitrary times and the result 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

$$kP = \underbrace{P + P + \cdots + P}_{k\ Times} \qquad (3.8)$$

Given the binary expansion $k = k_{l-1}2^{l-1} + k_{l-2}2^{l-2} + \cdots + k_2 2^2 + 2k_1 + k_0$ the scalar multiple kP can be compute by

$$Q = kP = k_{l-1}2^{l-1}P + k_{l-2}2^{l-2}P + \cdots + k_2 2^2 P + 2k_1 P + k_0 P$$

By factoring out 2 repeatedly we get

$$Q = (\cdots((k_{l-1}P)2 + k_{l-2})2 + \cdots + k_1 P)2 + k_0 P$$

which can be computed using Algorithm 3.5.

Algorithm 3-5 Scalar multiplication using 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 from l-2 to 0 do
        Q = DOUBLE(Q)
        if ( k_i= = 1) then
            Q = ADD(Q,P)
Return (Q)
```

3.4.3 Projective coordinates

The coordinate system used in section 3.4.1 for Elliptic curve operation is called affine coordinates. In this coordinate system, according to the group law of points on elliptic curve E, we can see that both point addition and point doubling need a Galois field inversion. Galois field inversion is much more expensive than Galois field multiplication. Using projective coordinates can eliminate the use of Galois field inversion in point addition and point doubling. The point addition and point doubling in projective coordinates can be computed as following [18]:

Point addition in projective coordinates:

$$Z3 = (X1.Z2 + X2.Z1)^2$$
$$X3 = x.Z3 + (X1.Z2).(X2.Z1) \quad (3.9)$$

where (X3, Z3) is the result of the point addition in projective coordinate, and (X1, Z1) (X2; Z2) are the projective coordinates of P and Q, respectively.

Point doubling in projective coordinates:

$$Z = X1^4 + b.Z1^4$$
$$X = Z1^2.X1^2 \quad (3.10)$$

where (X, Z) is the result of the point doubling in projective coordinates, and (X1, Z1) is the projective coordinates of P.

4 Hardware Acceleration Overview

General purpose processors are not optimized for cryptographic arithmetic [4]. They also cannot provide the amount of parallelism that is required to compute field arithmetic in scalar multiplication which is required in elliptic curve based cryptographic systems. This results in degradation of performance when compared to hardware implementation. It is, therefore, important to use hardware implementation to avoid such draw backs. This can be done by the use of two different hardware technologies.

They are:

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

ASICs are typically used when a design is to be produced in mass 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.

- Relatively small initial setup cost: A single FPGA is inexpensive when compared to the manufacturing cost of an ASIC design.
- Simplified implementation flow: In most cases, the FPGA vendor 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.
- 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.
- 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.

4.1 Basic FPGA concepts

The basic FPGA architecture consists of a two dimensional array of logic blocks and flip-flops with means for the user to configure (i) the function of each logic block, (ii) the inputs/outputs and (iii) the interconnection between blocks (Figure 4.1). Families of FPGAs differ from each other by the physical means for implementing user programmability, arrangement of interconnection wires, and basic functionality of the logic blocks.

***Programming Methods**:*

SRAM Based (e.g., XilinxTM): FPGA connections are achieved using pass-transistors, transmission gates, or multiplexers that are controlled by SRAM cells. This technology allows fast in-circuit reconfiguration. The major disadvantages are the size of the chip, required by the RAM technology, and the needs for some external source (usually external nonvolatile memory chips) to load the chip configuration. The FPGA can be programmed an unlimited number of times.

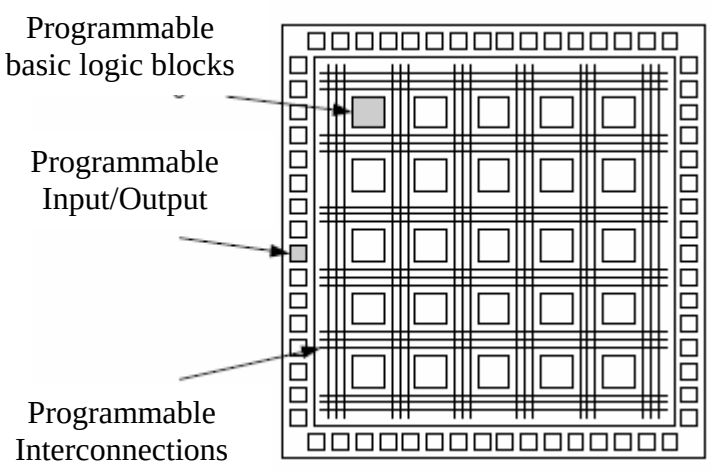


Figure 4-1 Basic architecture of FPGA

Look-Up Tables:

The way logic functions are implemented in a FPGA is another key feature. Logic blocks that carry out logical functions are look-up tables (LUTs), implemented as memory, or multiplexer and memory. Figure 4-2 shows these alternatives, together with an example

of memory contents for some basic operations. A 2^n x 1 ROM can implement any n-bit function. Typical sizes for n are 2, 3, 4, or 5.

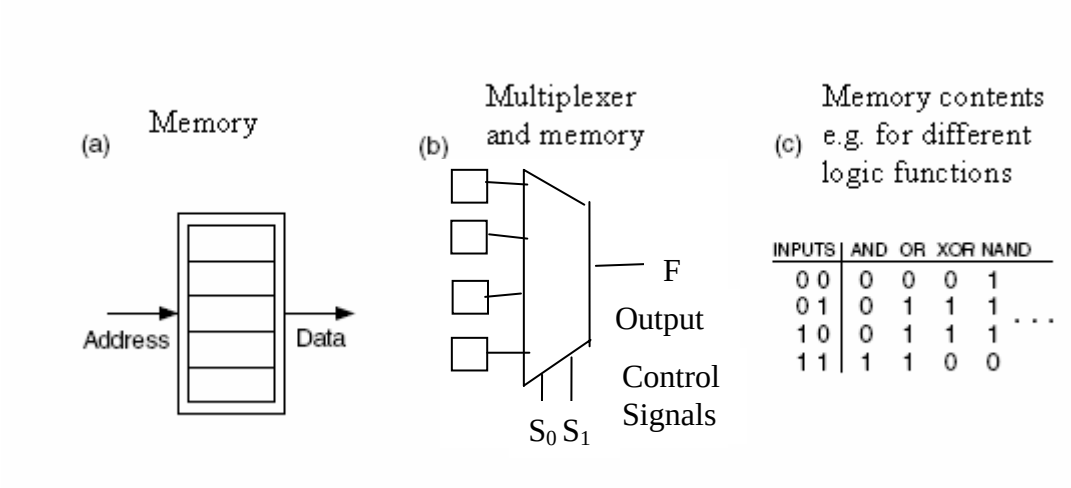


Figure 4-2 FPGA Look-up table (LUT)

FPGA Logic Block:

A simplified FPGA logic block can be designed with a LUT, typically a 4-input LUT, implementing a combinational logic function, and a register that optionally stores the output of the logic generator (Figure 4.3).

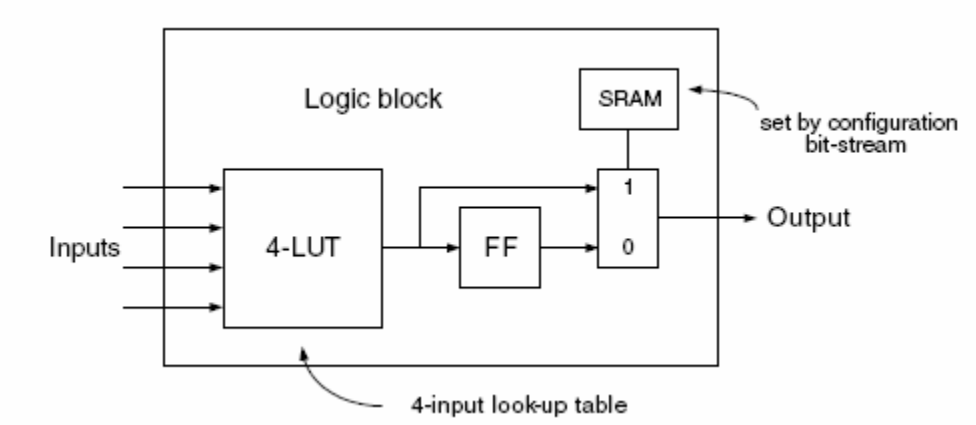


Figure 4-3 A basic FPGA logic block

4.2 Xilinx FPGA

The virtex II device family is a powerful architecture sharing most of the capabilities and basic concepts of Virtex. Spartan III is the low-cost version of Virtex II. Finally, Virtex II-Pro features additional hardwired Power-PC processors.

All Xilinx FPGAs contain the same basic resources (Figure 4.4):

- Configurable logic blocks (CLBs), containing combinational logic and register resources.
- Input/output blocks (IOBs), interface between the FPGA and the outside world.
- Programmable interconnections (PIs).
- RAM blocks.
- Other resources: three-state buffers, global clock buffers, boundary scan logic, and so on.

Furthermore, Virtex II and Spartan III devices contain resources such as dedicated multipliers and a digital clock manager (DCM). The Virtex II-Pro also includes embedded Power-PC processors and full-duplex high-speed serial transceivers.

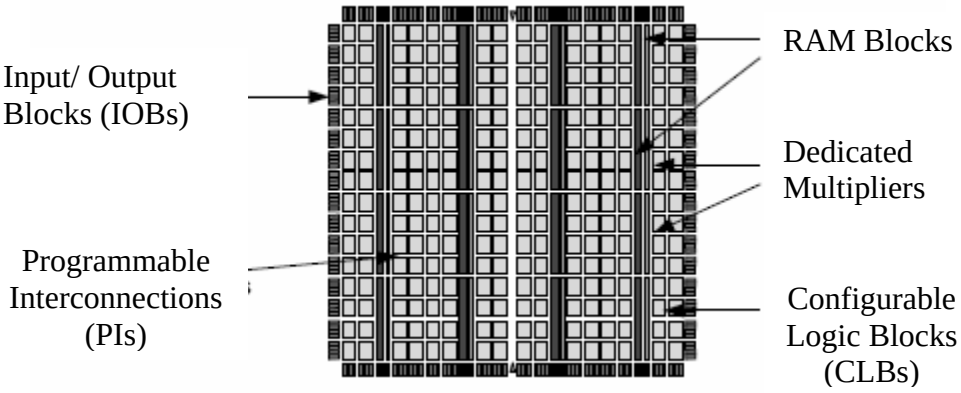


Figure 4-4 Example of distribution of CLBs, IOBs, PIs, RAM blocks, and multipliers in vertex II

4.2.1 Configurable Logic Blocks (CLBs)

The basic building block of Xilinx (CLBs) is the slice. Virtex and Spartan II hold two slices in one CLB, while Virtex II and Spartan III hold four slices per CLB. Each slice contains two 4-input function generators (F/G), carry logic, and two storage elements.

Each function generator output drives both the CLB output and the D-input of a flip-flop. Besides the four basic function generators, the Virtex/Spartan II CLB contains logic that combines function generators to provide functions of five or six inputs. The look-up tables and storage elements of the CLB have the following characteristics:

- Look-Up Tables (LUTs): Xilinx function generators are implemented as 4-input look-up tables. Beyond operating as a function generator, each LUT can be programmed as a (16x1)-bit synchronous RAM. Furthermore, the two LUTs can be combined within a slice to create a (16x2)-bit or (32x1)-bit synchronous RAM, or a (16x1)-bit dual-port synchronous RAM. Finally, the LUT can also provide a 16-bit shift register, ideal for capturing high-speed data.
- Storage Elements: The storage elements in a slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D-inputs can be driven either by the function generators within the slice or directly from the slice inputs, bypassing the function generators. As well as clock and clock enable signals, each slice has synchronous set and reset signals.

4.2.2 Input/Output Blocks (IOBs)

The Xilinx IOB includes inputs and outputs that support a wide variety of I/O signaling standards. The IOB storage elements act either as D-type flip-flops or as latches. For each flip-flop, the set/reset (SR) signals can be independently configured as synchronous set, synchronous reset, asynchronous preset, or asynchronous clear. Pull-up and pull-down resistors and an optional weak-keeper circuit can be attached to each pad. IOBs are programmable and can be categorized as follows:

- Input Path: A buffer in the IOB input path is routing the input signals either directly to internal logic or through an optional input flip-flop.
- Output Path: The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop. The 3-state control of the output can also be routed directly from the internal logic or through a flip-flop that provides synchronous enable and disable signals.

- Bidirectional Block: This can be any combination of input and output configurations.

4.2.3 RAM Blocks

Xilinx FPGA incorporates several large RAM memories (block select RAM). These memory blocks are organized in columns along the chip. The number of blocks, ranging from 8 up to more than 100, depends on the device size and family. In Virtex/Spartan II, each block is a fully synchronous dual-ported 4096-bit RAM, with independent control signals for each port. The data width of the two ports can be configured independently. In Virtex II/Spartan III, each block provides 18-kbit storage.

4.2.4 Programmable Routing

Adjacent to each CLB stands a general routing matrix (GRM). The GRM is a switch matrix through which resources are connected ; the GRM is also the means by which the CLB gains access to the general-purpose routing. Horizontal and vertical routing resources for each row or column include:

- Long Lines: bidirectional wires that distribute signals across the device.
- Vertical and horizontal long lines span the full height and width of the device.
- Hex Lines route signals to every third or sixth block away in all four directions.
- Double Lines: route signals to every first or second block away in all four directions.
- Direct Lines: route signals to neighboring blocks—vertically, horizontally, and diagonally.
- Fast Lines: internal CLB local interconnections from LUT outputs to LUT inputs.

The routing performance factor of internal signals is the longest delay path that limits the speed of any worst-case design. Consequently, the Xilinx routing architecture and its place-and-route software were defined in a single optimization process. Xilinx devices provide high-speed, low-skew clock distribution. Virtex provides four primary global nets that drive any clock pin; instead, Virtex II has 16 global clock lines—eight per quadrant.

4.2.5 Arithmetic Resources in Xilinx FPGAs

Modern FPGAs have special circuitry to speed-up arithmetic operations. Therefore adders, counters, multipliers, and other common operators work much faster than the same operations built from LUTs and normal routing only.
Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. There is one carry chain per slice; the carry chain height is 2 bits per slice. The arithmetic logic includes one XOR gate that allows a 1-bit full adder to be implemented within the available LUT. In addition, a dedicated AND gate improves the efficiency of multiplier implementations.

4.3 FPGA Design flow

Figure 4-5 depicts FPGA design flow. Brief description of the flow phases is also given

- **Design Entry**: creation of design files using schematic editor or hardware description language (Verilog, VHDL).
- **Design Synthesis**: a process that starts from a high level of logic abstraction (typically Verilog or VHDL) and automatically creates a lower level of logic abstraction using a library of primitives.
- **Partition (or Mapping):** a process of assigning to each logic element a specific physical element that actually implements the logic functions in a configurable device.
- **Place**: maps logic into specific locations in the target FPGA chip.
- **Route**: connections of the mapped logic.
- **Program Generation**: a bit-stream file is generated to program the device.
- **Device Programming**: downloading the bit-stream to the FPGA.

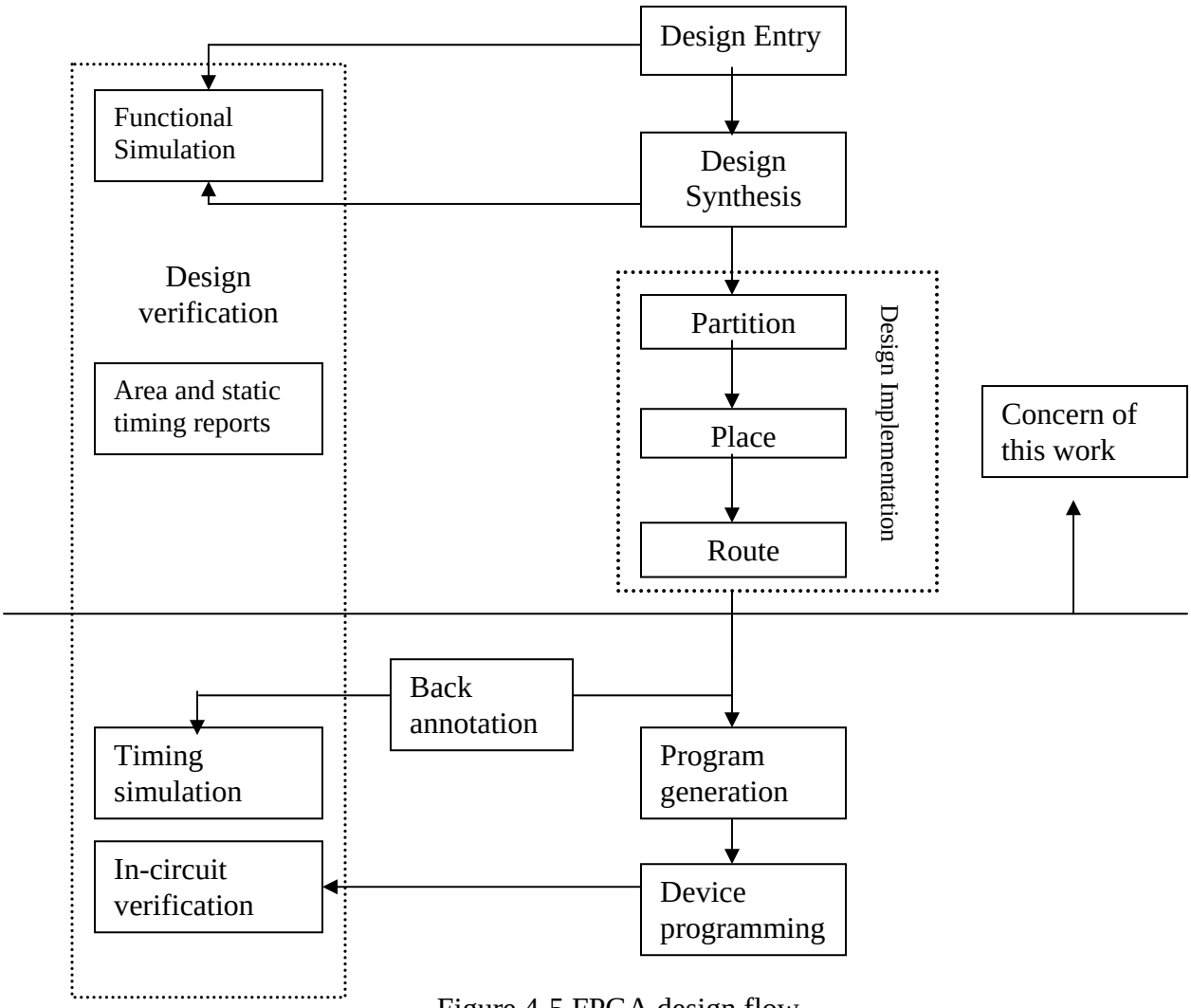


Figure 4-5 FPGA design flow

- **Design Verification**: simulation is used to check functionalities. The simulation can be done at different levels. The functional or behavioral simulation does not take into account component or interconnection delays. The timing simulation uses back-annotated delay information extracted from the circuit. Other reports are generated to verify other implementation results, such as maximum frequency and delay and resource utilization.

The partition (or mapping), place, and route processes are commonly referred to as design implementation.

In this work those steps, above the horizontal line (in Figure 4-5), up to the design implementation are done.

4.4 Hardware Description Language

A hardware description language (HDL) is a computer language designed for formal description of electronic circuits. It can describe a circuit operation, its structure, and the input stimuli to verify the operation (using simulation). An HDL model is a text-based description of the temporal behavior and/or the structure of an electronic system. In contrast to a software programming language, the HDL syntax and semantics include explicit notations for expressing time and concurrencies, which are the primary attributes of hardware.

The two main players in this field are VHDL and Verilog. Verilog is chosen, in this work, as the hardware description language due to its simplicity in that its syntax is very similar to the c/c++ software programming language.

5 Hardware Design for Finite Field Arithmetic

The operations in GF(2^m) are typically easier to implement in hardware than the arithmetic in finite fields of characteristic greater than 2. Because bit-wise addition in GF(2^m) does not have any carry propagation. The field arithmetic operations considered for this work are addition, multiplication, squaring and inversion. Among these operations, excluding inversion, field multiplication is the most repeated and resource consuming one and more focus is given to it.

5.1 Addition

As it will be recalled from the discussion in section 3.3.1 addition or subtraction is a simple bit-wise XOR operation. And such operations can be executed in a single clock cycle (other than the cycles that we need to load and unload data from and to registers).

5.2 Multiplication

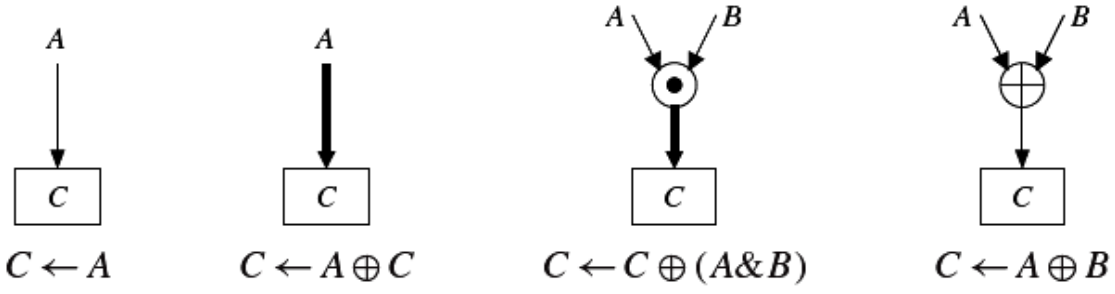
We discuss the design of a hardware circuit to multiply elements in a binary field of GF(2^m) and consider the case where the elements of GF(2^m) are represented with respect to a polynomial basis. If F(x) is the reduction polynomial, then we write

$$F(x) = x^m + r(x), \text{ where } \deg r \le m-1 \quad \textbf{(5.1)}$$

Moreover, if $r(x) = r_{m-1}r^{m-1} + r_{m-2}x^{m-2} + \cdots + r_2x^2 + r_1x + r_0$, then we represent r(x) by the binary vector

$$r = (r_{m-1}, r_{m-2}, \cdots, r_2, r_1, r_0).$$

In Figures 5.1, the following symbols are used to denote operations on bits A,B,C:



[1] Hardware architectures for field multiplication can be roughly categorized into three groups. Bit serial multipliers are based on Algorithm 3.2 in section 3.3.2. It generates one bit of the product at each clock cycle. Algorithm 5.1, which multiplies a multiplicand a Є GF(2^m) and a multiplier b Є GF(2^m), processes the bits of b from left (most significant) to right (least significant). The multiplier, called a most significant bit first(MSB) multiplier, is depicted in Figure 5.1 for the case m = 5. In this figure b is a shift register and c is a shift register whose low-end bit is tied to 0.

Algorithm 5-1 Most significant bit first (MSB) multiplier for GF(2^m)

INPUT: a = (a_{m-1},...,a_1,a_0) , b = (b_{m-1},...,b_1,b_0) , and the reduction polynomial
F(x) = x^m + r(x)
OUTPUT: c = a.b
c=0
for i from m-1 to 0 do
c = leftshift(c) + c_{m-1}r
c = c + b_ia
Return (c)

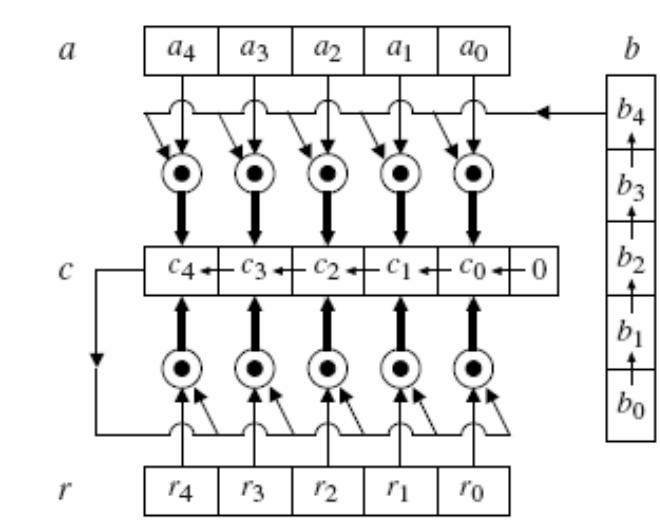


Figure 5-1 Most significant bit first (MSB) multiplier for GF(2^m)

The disadvantage of such architecture is the number of iterations required for the loop. In hardware, these 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. A compromise between these architectures is the Digit Serial multiplier. This multiplier is based on Algorithm 3.3 in section 3.3.2. While the complexity of the circuit increases with g as compared with bit-serial multiplier, a g-fold speedup for multiplication can be achieved. However, it requires fewer resources than the bit parallel method.

In [21] an efficient method for digit serial multiplier is proposed for software implementation. It uses two pre-computed tables. Based on this algorithm a hardware multiplier is implemented in hardware [10]. It is based on this work that we designed our multiplier.

5.2.1 Efficient Digit Serial Multiplier

[10] Algorithm 3.3 which is discussed in section 3.3.2 is modified as in algorithm 5.2.

Algorithm 5-2 modified group level field multiplication GF(2^m)
INPUT: Binary polynomials A(x) and B(x) of degrees at most m-1
OUTPUT: C(x) = A(x) x B(x) mod F(x)

$$C(x) = B_{s-1}(x)A(x) \bmod F(x)$$

for t from s-2 to 0 do

$$V1 = x^g \sum_{i=0}^{m-g-1} c_i x^i$$

$$V2 = x^g \sum_{i=m-g}^{m-1} c_i x^i \bmod F(x)$$

$$V3 = B_t(x)A(x) \bmod F(x)$$

$$C(x) = V1(x) + V2(x) + V3(x)$$

Return (c)

Note that *V1 is* a *g*-bit shift of the lower $m - g$ bits of C(x). *V2* is a *g*-bit shift of the upper *g* bits of C(x) followed by a modular reduction. V_3 requires a polynomial multiplication and reduction where the operand polynomials have degrees $g - 1$ and $m - 1$. In the next section we will discuss how we can compute V2 and V3.

The computation of V2 and V3 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 V3 and can be shown easily for V2. Note that

$$\begin{aligned} V2 &= c_{m-1}x^{m+g-1} + \cdots + c_{m-g+1}x^{m+1} + c_{m-g}x^{m} \bmod F(x) \\ &= x^{m}(c_{m-1}x^{g-1} + \cdots + c_{m-g+1}x^{1} + c_{m-g}) \bmod F(x) \end{aligned}$$

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 get

$$V2 = (x^d + \cdots + 1)(c_{m-1}x^{g-1} + \cdots + c_{m-g+1}x^1 + c_{m-g}) \bmod F(x) \quad \textbf{(5.2)}$$

Provided d+g < m , which is true for all NIST curves, V2 results in a polynomial of degree less than m which does not need to be reduced.

Let us denote the two polynomials for the multiplication of R(x) and W(x) to compute both V2 and V3. 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

$$\begin{aligned} R(x)W(x) \bmod F(x) = {} & r_{g-1}(x^{g-1}W(x) \bmod F(x)) + r_{g-2}(x^{g-2}W(x) \bmod F(x)) \\ & + \cdots + r_1(xW(x) \bmod F(x)) + r_0(xW(x) \bmod F(x)) \end{aligned} \quad \textbf{(5.3)}$$

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 5-2.

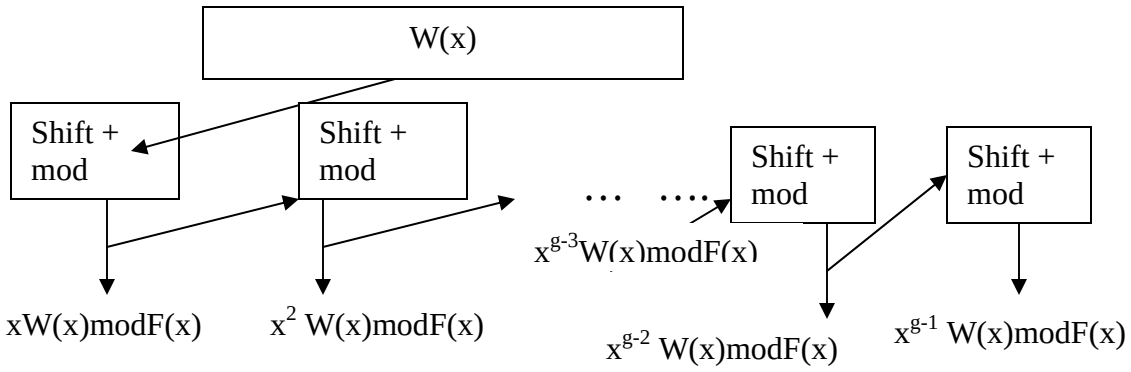


Figure 5-2 Generating $x^i W(x) \bmod F(x)$

When using NIST reduction polynomials these terms can be computed quickly at very little cost. Once these values are determined, the final result is computed in a g-input modulo 2 adder. The inputs to the adder are enabled by their corresponding coefficient r_i. This is shown in Figure 5-3 . The polynomial $x^i W(x)$ affects the output of the adder only if the coefficients bit r_i is a one. Otherwise the input associated with $x^i W(x)$ is driven with zeros.

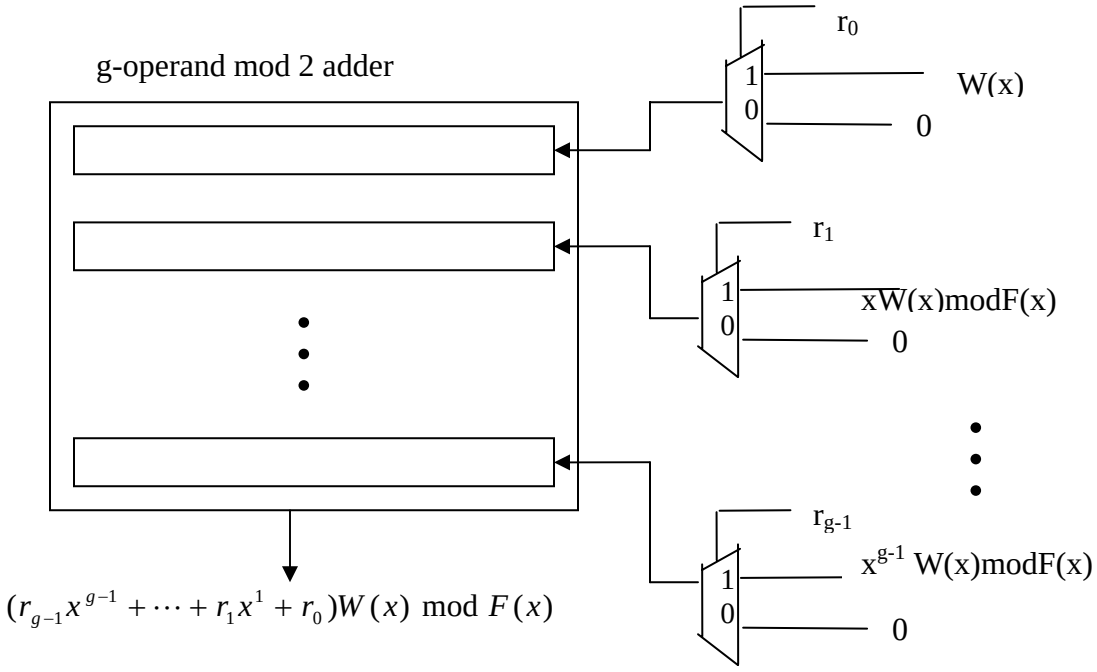


Figure 5-3 Computing R(x)W(x) mod F(x)

This method for multiplication is implemented for computation of both V2 and V3. In the case of V3, the polynomial W(x) has degree m-1 and will change for every field multiplication. For V2 the polynomial W(x) has degree d and is fixed. The value d is 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.

5.2.2 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 $g = 33$ and $g = 40$ for field size m = 163 result in the same performance, $\lceil 163/33 \rceil = \lceil 163/40 \rceil = 5$ but g = 40 requires a larger multiplier.

5.3 Squaring

In binary field, squaring is not complex as compared with multiplication. Basically, the square of an element a represented by A(x) involves two mathematical steps. The first is the polynomial multiplication of A(x) resulting in

$$A(x)^2 = a_{m-1}x^{2m-2} + \cdots + a_2 x^4 + a_1 x^2 + a_0 \quad \textbf{(5.4)}$$

The second is the reduction of this polynomial modulo F(x). If we split this polynomial into a non-reducible lower part and reducible higher part we get

$$A_h(x) = a_{m-1}x^{m-3} + \cdots + a_{(m+3/2)}x^2 + a_{(m+1/2)}$$
$$A_l(x) = a_{(m-1/2)}x^{m-1} + \cdots + a_1 x^2 + a_0$$

And $A^2(x) = A_h(x)x^{m+1} + A_l(x)$. The product $A_h(x)x^{m+1}$ may have degree as large as 2m-2. The reduction polynomial gives us the equality $x^m \equiv x^d + \cdots + 1$ and multiplying both sides by x we get $x^{m+1} \equiv x^{d+1} + \cdots + x$. Therefore

$$A_h(x)x^{m+1} = A_h(x)(x^{d+1} + \cdots + x) \quad \textbf{(5.5)}$$

This multiplication can be performed using a method similar to the one described in Section 5.2.1. The same architecture used to compute $R(x)$ $W(x)$ mod $F(x)$ in the multiplier is used here to compute $A_h(x)x^{m+1}$. 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.

5.4 Inversion

Inversion is the most complex operation in Galois field arithmetic. The inversion method discussed in Algorithm 3.4 using the square and multiply method requires m-1 squarings and m-2 multiplications. This is 162 squarings and 161 multiplications for m=163. Actually, the number of multiplications can be reduced due to the following features [17].

$$a^{2^t-1} \equiv \begin{cases} (a^{2^{t/2}-1})^{2^{t/2}}(a^{2^{t/2}-1}) & for\ t\ even \\ a(a^{2^{t-1}-1})^2 & for\ t\ odd \end{cases} \quad \textbf{(5.6)}$$

Based on the above relation we derive the following algorithm to compute inversion in GF(2^{163}).

We can see from algorithm 5-3 that inversion requires only 9 multiplications and 162 squarings. If multiplication takes 4 clock cycles (7 including data move) and squaring 1 clock cycle (4 including data move) inversion takes 711 clock cycles.

Algorithm 5-3 Inversion using Itoh and Tsujii GF(2^{163})

$$T0 = a$$
$$T1 = T0^2 = a^2$$
$$T1 = T1 * T0 = a^{2^2-1}$$
$$T2 = T1^2$$
$$T2 = T2^2$$
$$T1 = T1 * T2 = a^{2^4-1}$$
$$T2 = T1^2$$
$$T1 = T2 * T0 = a^{2^5-1}$$
$$T2 = T1^{2^5} \quad // 5 \ squarings$$
$$T1 = T2 * T1 = a^{2^{10}-1}$$
$$T2 = T1^{2^{10}} \quad // 10 \ squarings$$
$$\vdots$$
$$\vdots$$
$$T1 = T1 * T2 = a^{2^{20}-1}$$
$$T2 = T1^{2^{20}} \quad // 20 \ squarings$$
$$T1 = T1 * T2 = a^{2^{40}-1}$$
$$T2 = T1^{2^{40}} \quad // 40 \ squarings$$
$$T1 = T1 * T2 = a^{2^{80}-1}$$
$$T2 = T1^2$$
$$T1 = T2 * T0 = a^{2^{81}-1}$$
$$T2 = T1^{2^{81}} \quad // 81 \ squarings$$
$$T1 = T1 * T2 = a^{2^{162}-1}$$
$$T1 = T1^2 = a^{2^{163}-2}$$

5.4.1 Efficient realization of Inversion

To avoid idle clock cycles for data movement, output of the squaring is made available just after a single clock and used as input for squaring or multiplication. The output of the multiplication is also used directly as input to the next squaring. Except in few places, especially the multipliers are initialized; the idle cycles are utilized efficiently. This resulted in inversion taking only 230 cycles.

6 Hardware Design for Scalar Multiplication

6.1 Introduction

An important element of hardware design is to determine those layers of the hierarchy that should be implemented in hardware [1]. A typical design hierarchy of Elliptic curve algorithms is depicted in Figure 6-1. The top level of the system contains cryptographic protocols. In an ECC based SSL connection, the cipher suite uses ECDH for key exchange and ECDSA for authentication of the public key. Point multiplication is utilized in both of the ECDH and ECDSA protocol. The secondary level in the design hierarchy is point multiplication. Point multiplication is composed of point doubling and point addition. Point multiplication, point doubling and point addition are operations involving the points on the elliptic curve. The bottom level of the ECC system is Galois field arithmetic including Galois field multiplication, Galois field inversion and Galois field squaring and Galois field addition.

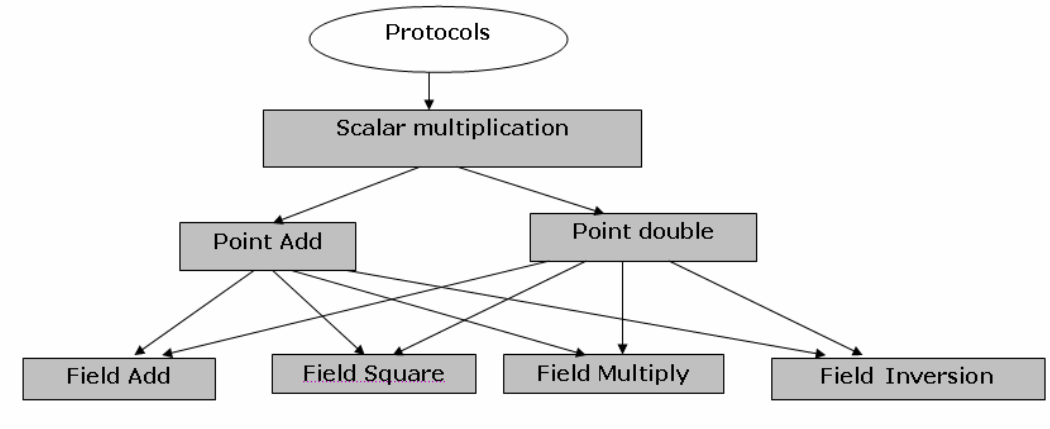


Figure 6-1 Design hierarchy of Elliptic curve algorithms

Clearly, finite field arithmetic must be designed into any hardware implementation. One possibility of hardware design is to accelerate finite field arithmetic only, and then use off-the-shelf microprocessor to perform the higher-level functions of elliptic curve point arithmetic. It is important to note that an efficient finite field multiplier does not necessarily yield an efficient point multiplier: all layers of the hierarchy need to be optimized. This is because executing field operations in parallel that is possible at the

curve operation level in hardware will not be possible if implementation such operations is done in software.

Moving point addition and doubling and then point multiplication to hardware provides a more efficient ECC processor at the expense of more complexity. In all cases a combination of both efficient algorithms and hardware architectures is required. Our design focuses on all but the protocol level of the elliptic curve cryptosystem.

The basic method for computing scalar multiplication or point multiplication is the well known “add-and-double” method discussed in section 3.3.2 which requires m point doublings and m/2 point additions on the average. Lopez and Dahab proposed a fast algorithm of point multiplication over GF(2^m) without pre-computation based on Montgomery ladder method[18]. One advantage of using this algorithm is that fewer field multiplications will be involved on average than in the traditional method. Secondly, since projective instead of affine coordinates are used, inversion is performed at the coordinate transformation step. In addition, it is secure against side channel attack. Therefore, we adopt it for our scalar multiplier [1].

6.2 Montgomery Scalar multiplication algorithm

Algorithm 5-2 shows Montgomery scalar multiplication for non-supersingular elliptic curves over binary fields. In this algorithm Madd (X1, Z1, X2, Z2), Mdouble (X1, Z1) and M_{xy}(X1,Z1, X2, Z2) are functions for point addition, point doubling and conversion of projective coordinates to affine coordinates.

Algorithm 6-1 Scalar multiplication in projective coordinates

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

X1 = x, Z1 = 1, X2 = x^4 + b, Z2 = x^2
If(k = 0 or x = 0) then
x=0 , y=0
end if

for i = l-2 to 0 do
if (k_i = 1) then
(X1,Z1) = Madd (X1, Z1, X2, Z2), (X2, Z2) = Mdouble(X2, Z2)
else
(X2, Z2)= Madd(X2, Z2, X1, Z1), (X1, Z1) = Mdouble(X1, Z1)
end if
end for

Q = M_{xy}(X1, Z1, X2, Z2)

return Q

The functions Madd, Mdouble and M_{xy} are implemented as follows

Madd(X1,Z1,X2,Z2)

{

Z = (X1*Z2 + X2*Z1)^2

X = x*Z + (X1*Z2) * (X2*Z1)

Return (X, Z)

}

Mdouble(X1, Z1)

{

X = X1^4 + b*Z1^4

Z = Z1^2 * X1^2

}

$M_{xy}(X1, Z1, X2, Z2)$

{

$x_k = X1/Z1$

$y_k = (x + x_k) *[(y + x^2)*Z1*Z2 + (X_2 +xZ2) (X1 + xZ1)] * (1/(x*Z1*Z2)) + y$

}

6.3 Hardware realization

Since finite field multiplier is the bottleneck of scalar multiplication, it requires special consideration for realizing high performance architecture for scalar multiplication. Consider a word serial finite field multiplier. It can be divided into two functional units: the multiplication core and the input/output buffers. When data is being loaded to the input buffer or the result is unloaded from the output buffer, the multiplier core is essentially idle. Our goal is to utilize the multiplier in such a way so that it effectively becomes the sole component that determines the time duration of each pass of the loop in the scalar multiplication algorithm.

This can be achieved by performing a field addition and/or a squaring in parallel with a field multiplication. For this the combined execution time for the addition and squaring is assumed to be less than or equal to that of multiplication. Since the multiplier is a finite state machine and performs the multiplication in a certain number of clock cycles, the multiplier should be fed with data at equal pace.

6.3.1 Merging of two execution paths

In Algorithm 6-1, depending on the value of k_i either the first or the second if-else statement is executed. The operations are the same in both paths, but the inputs and outputs of Madd() and Mdouble( ) functions are different. In order to keep the algorithm uniform and suitable for pipelining we merge the two k_i dependent execution paths in Algorithm 6-1. Since point addition is commutative, the inputs to Madd( ) function remains the same. The output variable, however, depends on k_i. It is sufficient to swap

X1 with X2 and Z1 with Z2 before and after any calculation, if k_i equals one. Doing so, the input to Mdouble( ) are changed to X2 and Z2 accordingly. After calculation, the variables need to be swapped back to their original states. If two consecutive bits are one, then a pair of swapping can be eliminated. This modification is shown in Algorithm 6-2.

Algorithm 6-2 Modified Montgomery multiplication in projective coordinates

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

X1 = x, Z1 = 1, X2 = x^4 + b, Z2 = x^2
If(k = 0 or x = 0) then
x=0 , y=0
end if

if(k_{l-2} = 1) then
swap(X1, X2), swap(Z1, Z2)
end if
for i = l-2 to 0 do

(X2, Z2)= Madd(X2, Z2, X1, Z1), (X1, Z1) = Mdouble(X1, Z1)
If ((i !=0 and k_i != k_{i-1}) or (i = 0 and k_i = 1)) then
Swap(X1, X2), Swap(Z1, Z2)
end if
end for

Q = M_{xy}(X1, Z1, X2, Z2)

return Q

6.3.2 Parallel execution

If the finite field operations required for each Madd(.) and Mdouble ( ) are performed in sequence, then each pass of the main loop of Algorithm 6-2 will require 6M + 3A + 5S clock cycles. Where M, A, and S are clock cycles required for field multiplication, addition and squaring operations. Note that this is repeated for the entire iteration of the Montgomery algorithm which is the number of bits in the scalar multiplier. In this work two field multipliers are used and computations of squaring and addition are done in the

idle cycle of the multipliers. This architecture depicted in Fig 6-1 reduces the clock cycles to 3M + A for a single iteration.

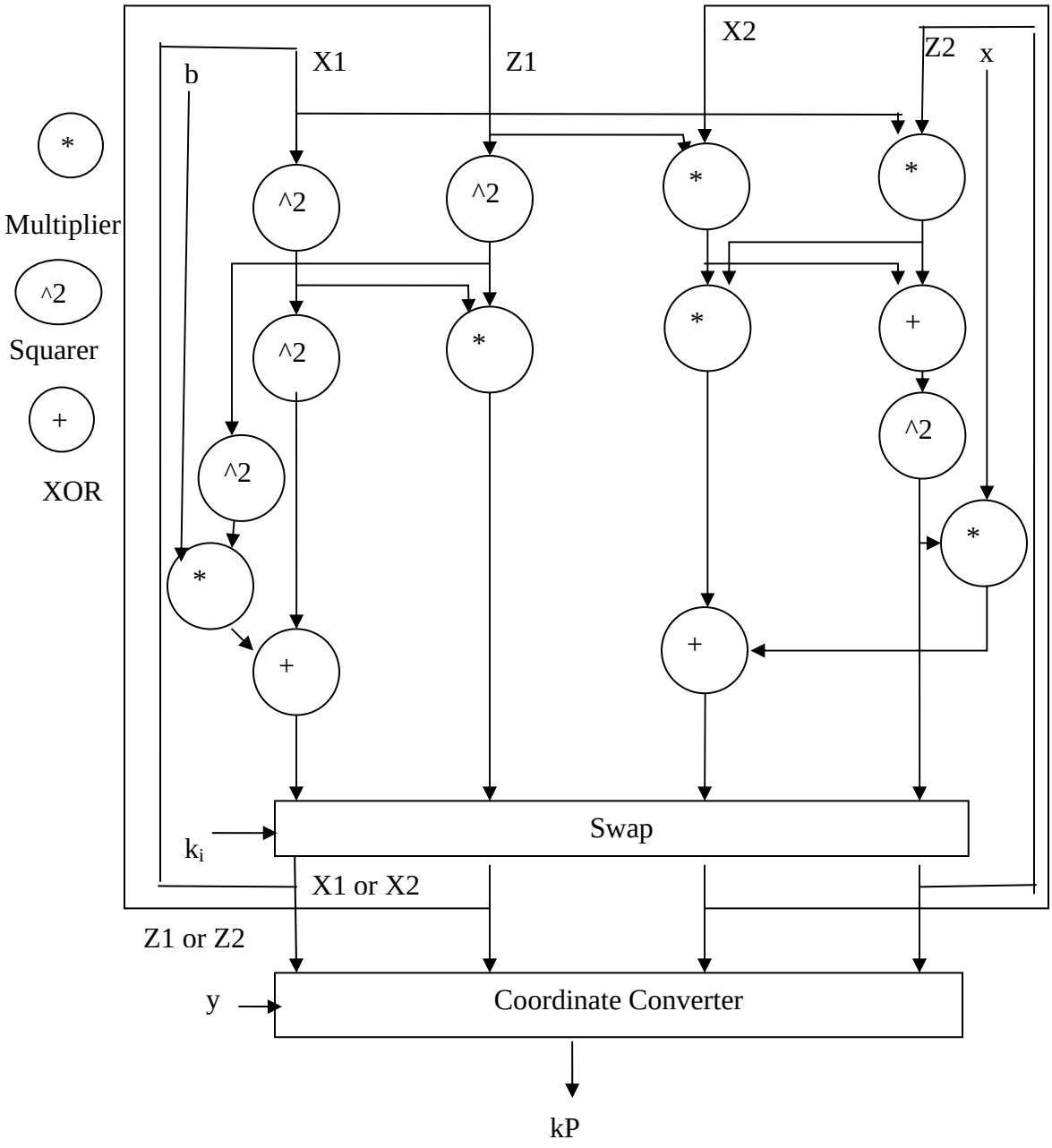


Figure 6-2 Proposed architecture for scalar multiplication

6.3.3 Realizing the coordinate converter

The coordinate converter maps the four outputs in projective coordinates X1, Z1, X2, Z2 into affine coordinate $x_{k,}$ y_k. It is complicated as compared with point doubling and point adding. However, as it is used only once at the end of point adding and doubling the previous resources are utilized efficiently here. Using a field inverter, the previous two finite multipliers, finite squaring and adding units, its realization is shown in Figure 6-3. Note that there is only one inversion in the hardware realization.

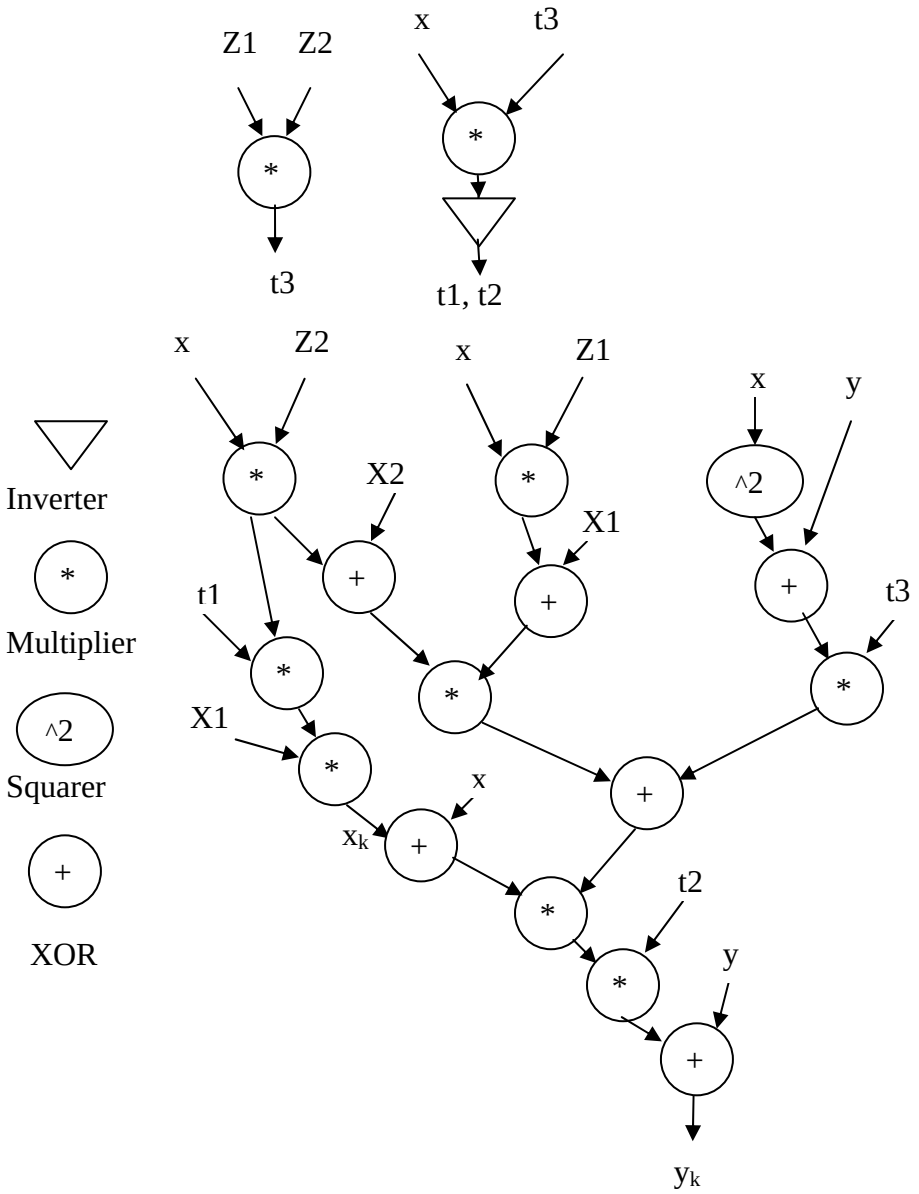


Figure 6-3 Hardware realization of the coordinate converter

7 Results and Discussions

The prototype of all the field arithmetic units and the scalar multiplier is realized in FPGA. For the hardware description Verilog HDL is used. Synthesis is done on the target device Xilinx XC2V2000. To show the effectiveness of the hardware acceleration the scalar multiplication is also implemented in software. MIRACL[23], an efficient cryptographic library is used to serve the purpose. All the codes were compiled using Visual Studio 6.0 and performance is measured on a Pentium IV 2.8 GHZ and 2GB RAM computer.

7.1 Experimental results

7.1.1 Results for Finite field Arithmetic

Field multiplier is synthesized for different digit sizes. The resource utilization and the maximum frequency in which the multiplier runs are summarized in Table 7-1.

Table 7-1 Performance and resource utilization for multiplication over GF (2^{163})

Multiplier Digit size (g)	**Maximum Frequency(MHZ)**	**# CLB slices**	**# Flip Flops**	**# LUT**
1	265.463	196(1%)	339(1%)	343(1%)
4	184.604	351(3%)	181(0%)	670(3%)
14	109.875	1321(12%)	185(0%)	2538(11%)
16	231.69	1229(11%)	284(1%)	2367(11%)
28	102.972	2059(19%)	187(0%)	3884(18%)
32	227.528	2384(22%)	401(0%)	4547(21%)
33	102.852	4511(41%)	204(0%)	8560(40%)
41	98.630	5378(50%)	212(0%)	10283(47%)

According to this table, the number of flip flops used in the synthesis result is below 2 % for all digit sizes experimented. Usage of CLB slices and LUTs increases with increase of multiplier digit size where as the operating frequency decreases with digit size except for bits 16 and 32. Another important observation made is bit sizes 16 and 32 give optimal synthesis results in terms of operating frequency. For instance, if we compare results for 32 and 41, we can observe that using digit size of 32 will give us twice the operating frequency of that of 41. It will also allow us to use multiple field multipliers in the point multiplier designed in the previous chapter. This is depicted in Figure 7-1.

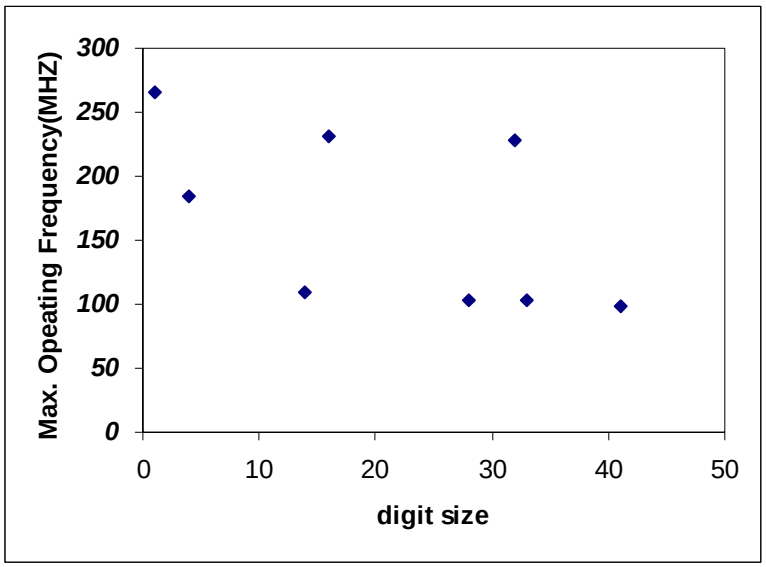


Figure 7-1 Maximum Operating frequency vs digit size

Also synthesized are the squarer and the inversion units and the result is shown in Table 7-2. As can be seen from the table, squaring uses the least resources among all the arithmetic units. Observation similar to that of multiplication can be made from this table about the optimal digit size for inversion. This is so because the inverter uses a multiplier and a squarer.

Table 7-2 Performance and resource utilization for Inversion and squiring over GF (2^{163})

Operation	Maximum Frequency	# CLB slices	# Flip Flops	# LUT
Inversion(g=1)	234.028	737(6%)	1216(5%)	1114(5%)
Inversion(g=4)	184.775	793(7%)	1060(4%)	1437(6%)
Inversion(g=14)	106.643	1757(16%)	1064(4%)	3312(15%)
Inversion(g=16)	202.102	1657(15%)	1075(4%)	3109(14%)
Inversion(g=28)	102.972	2480(23%)	1059(4%)	4651(21%)
Inversion(g=32)	188.875	2802(26%)	1118(5%)	5270(24%)
Inversion(g=33)	102.628	4938(45%)	1083(5%)	9146(43%)
Inversion(g=41)	99.514	5813(54%)	1091(5%)	11070(51%)
Squaring	-	95(0%)	-	165(0%)

7.1.2 Results for Scalar multiplication

The synthesis result of scalar multiplication for the different digit sizes is listed in table 7-3. We can see from the table that a scalar multiplication GF(2^{163}) takes 46.7µs.

Table 7-3 Performance and resource utilization of Scalar multiplier over GF (2^{163})

Multiplier Digit size	Maximum Frequency(MHz)	Latency per kP (µs)	# CLB slices	# Flip Flops	# LUT
1	203.421	298.4	1679(15%)	1393(6%)	3357(16%)
4	156.624	147.8	2178(21%)	1237(6%)	4117(19%)
14	99.745	123.3	4483(42%)	1323(6%)	8609(40%)
16	173.317	89.5	4191(39%)	1643(8%)	8006(37%)
28	96.832	80.3	6660(62%)	1433(6%)	12588(59%)
32	166.325	46.7	7300(67%)	1918(9%)	14527(68%)

The scalar multiplication is implemented also using a cryptographic library called MIRACL and the performance is found to be 7.6msec on Pentium IV computer with

2.8GH clock speed and 2GB RAM. This means the FPGA accelerated the point multiplication by 161 fold.

Our result is also compared with other works and it is reported in Table 7-4. The slowest implementation among this works is Orlando & Parr's design[25]. However, the implementation is done on a different FPGA. Then comes that of N. Gura, et. al. the resource utilization as well as the timing is higher than the rest of the works[8]. Among all the works J. Luaz' design is the most efficient in resource utilization and a single scalar multiplication takes 75µs[10]. However, this work uses encoding for the scalar multiplier. The encoder is not implemented in hardware. For testing purpose the encoding is done using software. Chang Chu's designed a hardware in which point multiplier takes only 53µs[13]. However, it is the worst in resource utilization. This is due to allocation of separate multiplier for all the units in the design hierarchy.

Table 7-4 Comparison with other Published results

Implementation	**FPGA**	**# Flip Flops**	**# LUT**	**kP (µs)**
Orlando & Parr[25]	Xilinx XCV400E	-	-	210
N. Gura, et. al.[8]	Xilinx XCV2000E	6442	19508	144
J. Luaz [10]	Xilinx XCV2000E	1930	10017	75
Chang Chu[13]	Xilinx XCV2000E	7467	25768	53
Our design	Xilinx XCV2000	1918	14527	47

All in all our work is the best among the ones listed above. A scalar multiplication takes 47μs and its resource utilization is 1918 Flop Flops and 14527 LUTs on Xilinx XCV2000 FPGA.

8. Conclusion and Further works

RSA is the most widely used public key cryptosystem. Recently, however, ECC is becoming the most prominent one. This is so because ECC is efficient for software as well as hardware realization. In addition, it gives a better security with shorter bit length than RSA.

In this work, hardware is designed and realization is done on FPGA for both curve and arithmetic operations. First we tried to implement efficient hardware for digit serial finite field multiplier based on the works of [21]. Then other field arithmetic operations are designed: squaring and Inversion. While realizing the inversion idle cycles are utilized to execute squaring resulting in a better implementation than the best result reported in the literature.

After completing the finite field design, a scalar multiplier is designed. We adopted Montgomery scalar multiplication algorithm due to Lopez and Dahab [18]. Modifications are made on this algorithm to efficiently realize it on hardware. The first modification made is swapping to make the execution path for point doubling and point adding a single path. Then the point doubling and point add operations are implemented using two multipliers, a squarer and XORs. These curve operations are implemented in parallel and the squaring and XOR is done in the idle cycle of the multiplication.

The designed scalar multiplier over GF(2^{163}) is realized using Xilinx XCV2000 FPGA. A single point multiplication takes 47 micro seconds with resource utilization of 14527 LUTs and 1918 Flip Flops. This result is 161 fold faster than the software implementation and better in latency than the best result reported in the literature utilizing half of the resources used in the latter.

There are other tasks for further work. The first one is loading the binary of our program on a real FPGA. After this work is properly completed, the actual performance of the

hardware accelerator can be tested for a particular cryptographic algorithm, for instance ECDSA. Finally, support for the rest of NIST curves could be tried on a single FPGA.

Bibliography

[1] Darrel H. ,etal , Guide to Elliptic Curve cryptography, SPRINGER ROFESSIONAL COMPUTING, 2004

[2] M. Joye and J.-J. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs Quisquater (Eds.): CHES 2004, LNCS 3156, pp. 119–132, 2004

[3] William Stallings, Cryptography and Network Security Principles and Practices, 4th edition, Prince Hall, 2005

[4] J. Deschamps and etal. , Synthesis of Arithmetic circuits: FPGA, ASIC and embedded Systems, John Willey & Sons, 2006

[5] Chanho L. and Jeongho L. , Design of an Elliptic Curve Cryptography Processor Using a Scalable Finite Field Multiplier in GF(2^{193}), Journal of the Korean Physical Society, Vol. 44, No. 1, January 2004, pp. 39-45

[6] Vipul Gupta,etal, Performance Analysis of Elliptic curve cryptography for SSL , Sun Microsystems Laboratories , 2001.

[7] L. Goubin and M. Matsui , Hardware/Software Co-Design of Elliptic Curve Cryptography on an 8051 Microcontroller , CHES 2006, LNCS 4249, pp. 430–444, 2006. International Association for Cryptologic Research 2006

[8] N. Gura, et. al.,An End-to-End Systems Approach to Elliptic Curve Cryptography , Springer-vertag, pp. 349–365, 2003.

[9] ALI M.Z, A MODULAR RECONFIGURABLE ARCHITECTURE FOR ASYMMETRIC AND SYMMETRIC KEY CRYPTOGRAPHY, MS thesis, King Faisal University of Petroleum and Minerals, 2007.

[10] Johnatan Luaz, High Performance Elliptic Curve Cryptographic Co-processor, M.Sc thesis, Univeristy of waterloo, 2003

[11] J. Riley and M.J. Shulte, A Hardware Accelerator for Elliptic Curve Cryptography over GF(2^M), Univesity of Wiscosin,2004

[12] Jian Huang , FPGA IMPLEMENTATIONS OF ELLIPTIC CURVE CRYPTOGRAPHY AND TATE PAIRING OVER BINARY FIELD, M.Sc Thesis, University of North Texas, 2007

[13] Chang Chu ,HARDWARE ARCHITECTURES OF ELLIPTIC CURVE BASED CRYPTOSYSTEMS OVER BINARY FIELDS, George Mason University, PH.D dissertation, 2007

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

[15] Ken Eguro, Scott Hauck, “Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development”, Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 111-120,2003.

[16] Steve Kilts, Advanced FPGA design: Architecture, Implementation and Optimization, John Wiley & Sons, 2007

[17] T. Itoh and S. Tsujii, A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases, Information and Computation, vol. 78, 1988, pp. 171 - 177.

[18] J. Lopez and R. Dahab, Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. Cryptographic Hardware and Embedded Systems -CHES'99, LNCS 1717, pp. 316 - 327.

[19] Martin Christopher, Elliptic Curve Cryptosystems on Reconfigurable Hardware, M.Sc thesis, Worcester Polytechnic Institute, 1998

[20] J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In Advances in Cryptography | CRYPTO '97, pages 342-356. Springer-Verlag, 1997.

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

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

[23] MIRACL, **M**ultiprecision **I**nteger and **R**ational **A**rithmetic **C**/C++ **L**ibrary, http://www.shamus.ie/

[24] F. Rodriguez-Henriquez ,etal, Cryptographic Algorithms on Reconfigurable Hardware, Springer, 2006

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

Appendix A - Random Elliptic curve parameters over F(2^{163})

The following parameters are given for this curve:

m The extension degree of the binary field

F(z) The reduction polynomial of degree m.

S The seed selected to randomly generate the coefficients of the elliptic curve.

a,b The coefficients of the elliptic curve $y^2 + xy = x^3 + ax^2 + b$

n The(prime) order of the base point P

h The cofactor

x,y The x and y coordinates of P.

B-163: m=163, $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$, a=1, h=2

S = 0x 85E25BFE 5C86226C DB12016F 7553F9D0 E693A268

b= 0x 000000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD

n= 0x 000000004 00000000 0000000000 000292FE 77E70C12 A4234C33

x= 0x 000000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36

y= 0x 000000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1

Appendix B – Verilog Test benches and Sample Simulation results

Here binary test fixtures and test fixtures and behavioural test are presented for bianary field multiplier, binary field squarer and binary field inverter. And alos presented is similar result for the scalar multiplier.

1. Binary field Multiplier (g=32)

```
`timescale 1ns / 1ps

module multiply_st_v;
        parameter m=163;
        // Inputs
        reg [162:0] a;
        reg [162:0] b;
        reg clk;
        reg reset;
        reg start;
        // Outputs
        wire done;
        // Bidirs
        wire [162:0] c;
        integer i;
        // Instantiate the Unit Under Test (UUT)
        multiplier_163_7_6_3 uut (
                .a(a),
                .b(b),
                .clk(clk),
                .reset(reset),
                .start(start),
                .c(c),
                .done(done)
        );
```

```
always
        #5 clk=!clk;
        initial begin
                // Initialize Inputs
                a = 0;
                b = 0;
                clk = 0;
                reset = 0;
                start = 0;
                // Wait 100 ns for global reset to finish
                #100;
                reset =1;
                #10 reset=0;

                b=163'h6237e711bf388df9c46fce237e711bf388df9c43a;
                a=163'h57;

                #10 start=1;
                #10 start=0;
                // Add stimulus here
        for(i=0;i<=m;i=i+1)
  begin
        #10
        if(done==1)
   $display($time,"a=%h \n b=%h \n c=%h ",a[162:0],b[162:0],c[162:0]);
  end
end
endmodule
```

Test result is provided on the next page.

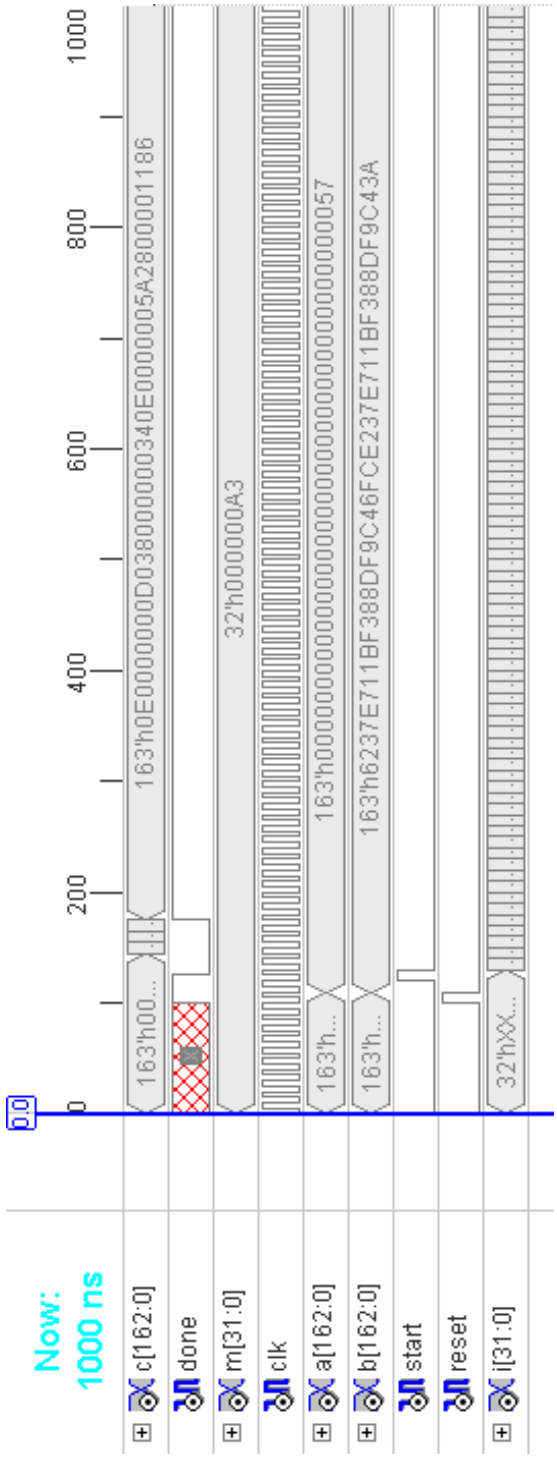


Figure B-1 Binary field multiplication simulation result

2. Binary field squarer

```
`timescale 1ns / 1ps
module sq_st();
parameter m=163;
parameter g=6;
reg [m-1:0]a,b;
wire [m-1:0]z1;

 square_163_7_6_3 s1(a,z);

 initial
 begin
        a=163'h66748f33a4799d23cce91e4beef25f7792fbbc92e;
        b=163'h5237B;
 end

 initial
 $monitor($time," a=%h \n b=%h \n z=%h",a,b,z);
Endmodule
```

Test result is provides on the next page.

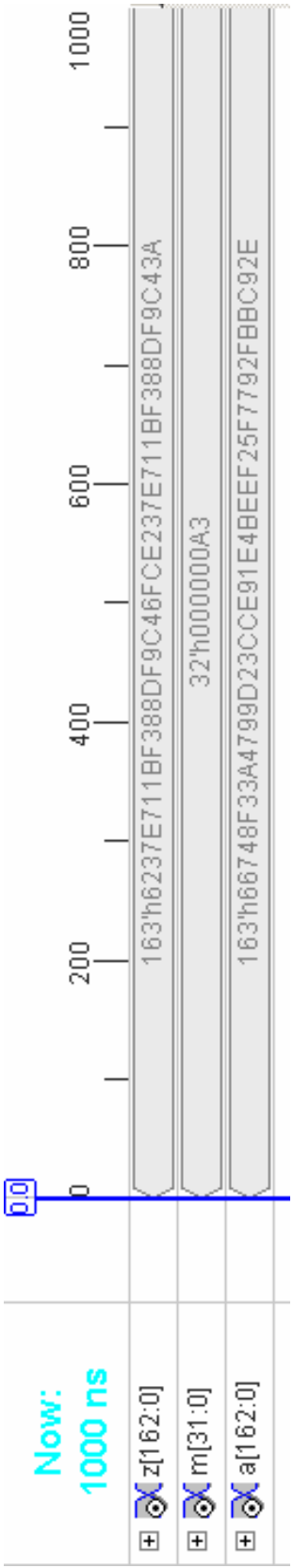


Figure B-2 Binary field squaring simulation result

3. Binary field Invertor

```
`timescale 1ns / 1ps
module inverse_st_v;

        // Inputs
        reg [162:0]a;
        reg clk;
        reg reset;
        reg start;

        // Outputs
        wire [162:0] c;
        wire done;

        integer i;

        // Instantiate the Unit Under Test (UUT)
        invert uut (
                .a(a),
                .clk(clk),
                .reset(reset),
                .start(start),
                .c(c),
                .done(done)
        );

  always
         #5 clk=!clk;

        initial begin
                // Initialize Inputs
```

```
        a = 0;
        //b = 0;
        clk = 0;
        reset = 0;
        start = 0;


        // Wait 100 ns for global reset to finish
        #100;
         reset =1;
        #10 reset=0;

        a=163'h57;
        //b=163'h57;

        #10 start=1;
        #10 start=0;
        // Add stimulus here
    for(i=0;i<=230;i=i+1)
  begin
    #10
       $display($time,"a=%h \n  c=%h ",a,c);
  end
    end
endmodule
```

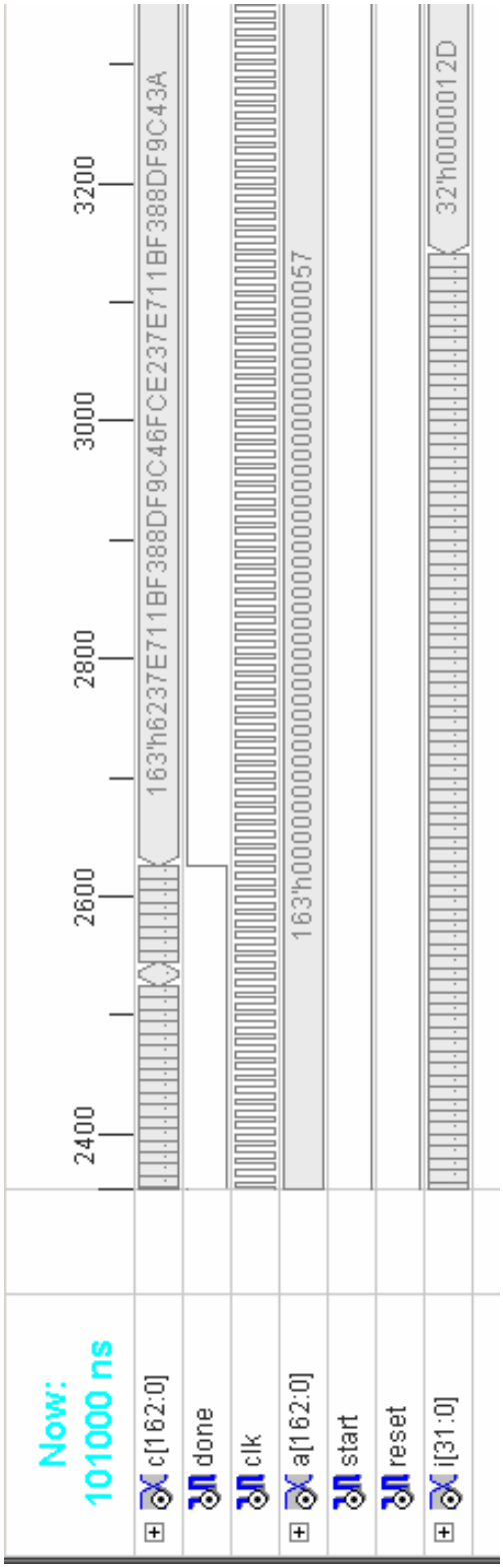


Figure B-3 Binary field inversion simulation result

Scalar multiplier

The following is Verilog test bench for scalar multiplier.

```
`timescale 1ns / 1ps
module monte_pm_st_v;

    // Inputs
    reg [162:0] xp;
    reg [162:0] yp;
    reg [162:0] k;
    reg clk;
    reg reset;
    reg start;

    // Outputs
    wire [162:0] xq;
    wire [162:0] yq;
    wire q_infinity;
    wire done;
    integer i;
    // Instantiate the Unit Under Test (UUT)
    montgomery_point_multiplication uut (
        .xp(xp),
        .yp(yp),
        .k(k),
        .clk(clk),
        .reset(reset),
        .start(start),
        .xq(xq),
        .yq(yq),
        .q_infinity(q_infinity),
        .done(done)
```

```
        );
        always
         #5 clk=!clk;

        initial begin
                // Initialize Inputs
                xp = 0;
                yp = 0;
                k=0;
                clk = 0;
                reset = 0;
                start = 0;
                // Wait 100 ns for global reset to finish
                #100;
                reset =1;
                #10 reset=0;
                xp=163'h3F0EBA16286A2D57EA0991168D4994637E8343E36;
                yp=163'h0D51FBC6C71A0094FA2CDD545B11C5C0C797324F1;
                k=163'h6237e711bf388df9c46fce237e711bf388df9c43a;
                #10 start=1;
                #10 start=0;
                // Add stimulus here
        for(i=0;i<=100000;i=i+1)
  begin
        #10
        if(done==1)
  $display($time,"xp=%h \n yp=%h \n k=%h \n xq=%h \n yq=%h ",xp,yp,k,xq,yq);
        end
        end
  endmodule
```

Test result is provided on the next page.

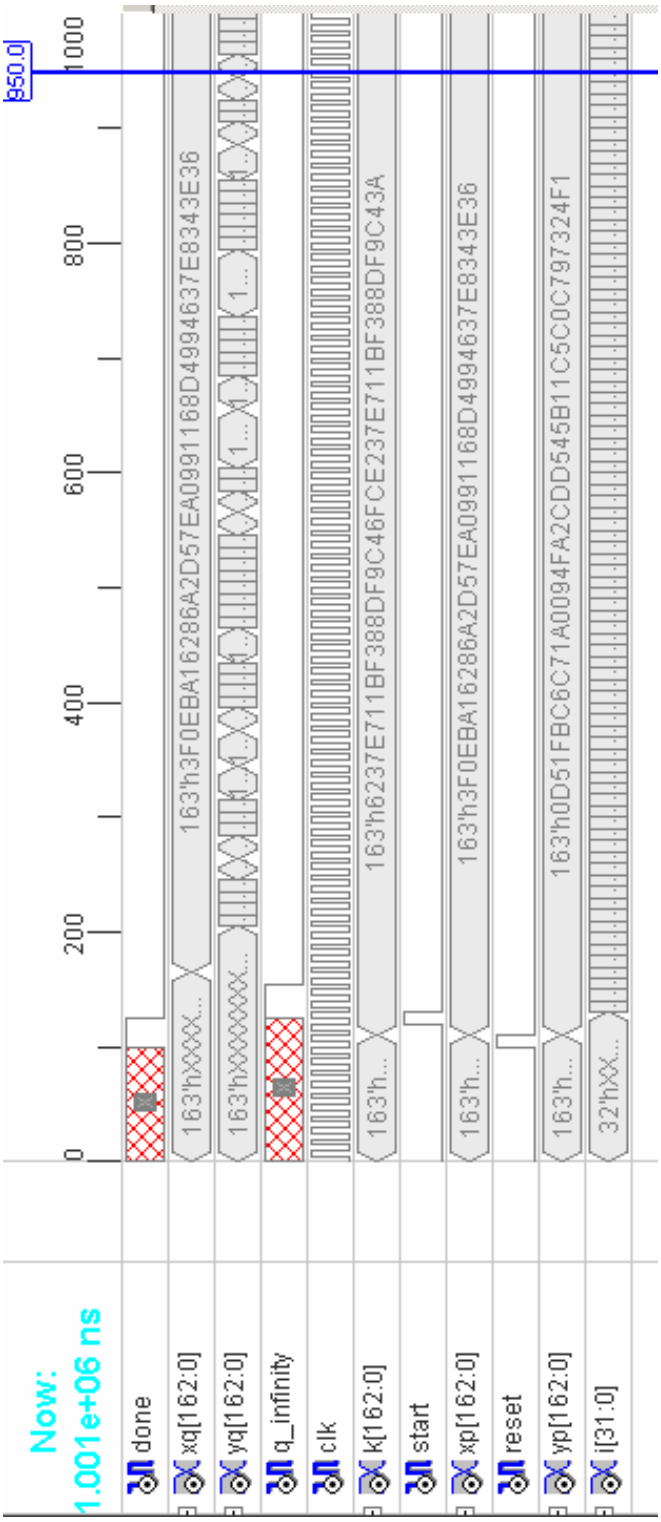


Figure B-4 Scalar multiplication simulation result

Appendix C – sample Verilog code (binary field multiplier)

Putting all the codes of the compete design will make this paper bulky. For, instance inversion which is implemented using a multiplier and a squarer has 937 lines of code. Therefore, only a sample code of binary field multiplier is included.

```
`timescale 1ns / 1ps
module  data_path_multiplier ( a, b, clk, ce_c, clear_c, load_a, shift_a, c, equal_zero );
    parameter m = 163;//constant
    parameter logm = 8;//constant


        parameter g=32;
        parameter s=6;
        parameter f=163-(s-1)*g;

   input [(m - 1):0]  a ;
   input [(m - 1):0]  b ;
   input clk;
   input ce_c;
   input clear_c;
   input load_a;
   input shift_a;
   inout [(m - 1):0]  c ;
   output equal_zero;


   wire [(m - 1):0] next_c;
   //reg [(m - 1):0] int_a;
   reg [(logm - 3):0] count;
   reg [(m - 1):0] c_=163'b0;
```

```
    wire [(m - 1):0] c;
    wire equal_zero;



//digit serial
//reg [m-1:0]temp;
//reg [m-1:0]v1,v2,v3;

reg [g-1:0]bk;
wire [m-1:0]v11,v22,v33,p1;

wire [m-1:0]w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10;
wire [m-1:0]w11,w12,w13,w14,w15,w16,w17,w18,w19,w20;
wire [m-1:0]w21,w22,w23,w24,w25,w26,w27,w28,w29,w30,w31;

  //w finding can be optimized further
  assign w0=a;
  assign w1={{w0[161:7]},{w0[6:0],1'b0} ^ ({8{w0[m-1]}} & 8'b11001001)};
  assign w2={{w1[161:7]},{w1[6:0],1'b0} ^ ({8{w1[m-1]}} & 8'b11001001)};
  assign w3={{w2[161:7]},{w2[6:0],1'b0} ^ ({8{w2[m-1]}} & 8'b11001001)};
  assign w4={{w3[161:7]},{w3[6:0],1'b0} ^ ({8{w3[m-1]}} & 8'b11001001)};
  assign w5={{w4[161:7]},{w4[6:0],1'b0} ^ ({8{w4[m-1]}} & 8'b11001001)};
  assign w6={{w5[161:7]},{w5[6:0],1'b0} ^ ({8{w5[m-1]}} & 8'b11001001)};
  assign w7={{w6[161:7]},{w6[6:0],1'b0} ^ ({8{w6[m-1]}} & 8'b11001001)};
  assign w8={{w7[161:7]},{w7[6:0],1'b0} ^ ({8{w7[m-1]}} & 8'b11001001)};
  assign w9={{w8[161:7]},{w8[6:0],1'b0} ^ ({8{w8[m-1]}} & 8'b11001001)};
  assign w10={{w9[161:7]},{w9[6:0],1'b0} ^ ({8{w9[m-1]}} & 8'b11001001)};

  assign w11={{w10[161:7]},{w10[6:0],1'b0} ^ ({8{w10[m-1]}} & 8'b11001001)};
  assign w12={{w11[161:7]},{w11[6:0],1'b0} ^ ({8{w11[m-1]}} & 8'b11001001)};
  assign w13={{w12[161:7]},{w12[6:0],1'b0} ^ ({8{w12[m-1]}} & 8'b11001001)};
```

```
assign w14={{w13[161:7]},{w13[6:0],1'b0} ^ ({8{w13[m-1]}} & 8'b11001001)};
assign w15={{w14[161:7]},{w14[6:0],1'b0} ^ ({8{w14[m-1]}} & 8'b11001001)};
assign w16={{w15[161:7]},{w15[6:0],1'b0} ^ ({8{w15[m-1]}} & 8'b11001001)};
assign w17=w16<<1 ^ ({8{w16[m-1]}} & 8'b11001001);
assign w18=w17<<1 ^ ({8{w17[m-1]}} & 8'b11001001);
assign w19=w18<<1 ^ ({8{w18[m-1]}} & 8'b11001001);
assign w20=w19<<1 ^ ({8{w19[m-1]}} & 8'b11001001);

assign w21=w20<<1 ^ ({8{w20[m-1]}} & 8'b11001001);
assign w22=w21<<1 ^ ({8{w21[m-1]}} & 8'b11001001);
assign w23=w22<<1 ^ ({8{w22[m-1]}} & 8'b11001001);
assign w24=w23<<1 ^ ({8{w23[m-1]}} & 8'b11001001);
assign w25=w24<<1 ^ ({8{w24[m-1]}} & 8'b11001001);
assign w26=w25<<1 ^ ({8{w25[m-1]}} & 8'b11001001);
assign w27=w26<<1 ^ ({8{w26[m-1]}} & 8'b11001001);
assign w28=w27<<1 ^ ({8{w27[m-1]}} & 8'b11001001);
assign w29=w28<<1 ^ ({8{w28[m-1]}} & 8'b11001001);
assign w30=w29<<1 ^ ({8{w29[m-1]}} & 8'b11001001);

assign w31=w30<<1 ^ ({8{w30[m-1]}} & 8'b11001001);

assign v33=((bk[0]==1'b1)?w0:0) ^ ((bk[1]==0)?0:w1) ^ ((bk[2]==0)?0:w2)
        ^ ((bk[3]==0)?0:w3) ^ ((bk[4]==0)?0:w4) ^ ((bk[5]==0)?0:w5)
        ^ ((bk[6]==0)?0:w6) ^ ((bk[7]==0)?0:w7) ^ ((bk[8]==0)?0:w8)
        ^ ((bk[9]==0)?0:w9) ^ ((bk[10]==0)?0:w10) ^ ((bk[11]==0)?0:w11)
        ^ ((bk[12]==0)?0:w12) ^ ((bk[13]==0)?0:w13) ^ ((bk[14]==0)?0:w14)
                                ^ ((bk[15]==0)?0:w15) ^ ((bk[16]==0)?0:w16) ^
((bk[17]==0)?0:w17)
                                ^ ((bk[18]==0)?0:w18) ^ ((bk[19]==0)?0:w19) ^
((bk[20]==0)?0:w20)
```

```
                                ^ ((bk[21]==0)?0:w21) ^ ((bk[22]==0)?0:w22) ^
((bk[23]==0)?0:w23)
                                ^ ((bk[24]==0)?0:w24) ^ ((bk[25]==0)?0:w25) ^
((bk[26]==0)?0:w26)
                                ^ ((bk[27]==0)?0:w27) ^ ((bk[28]==0)?0:w28)^
((bk[29]==0)?0:w29)
                                ^ ((bk[30]==0)?0:w30) ^ ((bk[31]==0)?0:w31);



 assign p1=c[(m-1)-:g];
 assign v11=c[0+:m-g]<<g;
 assign v22=p1[m-1:0]<<7 ^ p1[m-1:0]<<6 ^ p1[m-1:0]<<3 ^ p1[m-1:0];


 assign next_c= (~load_a)&(v11 ^  v22) ^ v33;  //check effect of load_a


 always @ (posedge clk ) // begin

          if ((load_a == 1'b1))
             //int_a <= a;
                                    begin
                                      bk=b[(m-1)-:f];  // f=163-(s-1)*g
                                    end



          else if ((shift_a == 1'b1))
             begin
```

```
                bk=b[(g*(count)-1)-:g];

              end

// end always

always @ (posedge clk ) // begin

          if ((load_a == 1'b1))
             //count <= (m - 1);
               count<=s-1;

          else if ((shift_a == 1'b1))
             count <= (count - 1);

// end always

assign equal_zero=(count==6'b000000) ? 1'b1 : 1'b0;


always @ (posedge clk ) // begin

          if ((clear_c == 1'b1))
             c_ <= {m {1'b0}};

          else if ((ce_c == 1'b1))
                c_ <= next_c;


// end always
    assign c = c_;
```

```
endmodule

module  multiplier_163_7_6_3 ( a, b, clk, reset, start, c, done );
        parameter m = 163;//constant
     parameter logm = 8;//constant

    input [(m - 1):0]  a ;
    input [(m - 1):0]  b ;
    input clk;
    input reset;
    input start;
    inout [(m - 1):0]  c ;
    output done;


    reg ce_c;
    reg clear_c;
    reg load_a;
    reg shift_a;
    wire equal_zero;
    reg [1:0] current_state;
    wire [(m - 1):0] c;
    reg done;


 data_path_multiplier main_component(.a(a), .b(b), .clk(clk), .ce_c(ce_c),
.clear_c(clear_c), .load_a(load_a), .shift_a(shift_a), .c(c), .equal_zero(equal_zero));

   always @ (posedge clk or posedge reset ) begin
```

```
    if ((reset == 1'b1))
        current_state <= 0;

    else
        case (current_state)
            0 :     if ((start == 1'b0))
                        current_state <= (current_state + 1);


            1 :     if ((start == 1'b1))
                        current_state <= (current_state + 1);


            2 :     current_state <= (current_state + 1);
            3 :     if ((equal_zero == 1'b1))
                        current_state <= 0;

        endcase
        end// always

always @ ( clk or reset or current_state or equal_zero ) // begin
    begin
        case (current_state)
            0,1 :
                begin
                    ce_c <= 1'b0;
                    clear_c <= 1'b0;
                    load_a <= 1'b0;
                    shift_a <= 1'b0;
                    done <= 1'b1;
                end
```

```
        2 : begin
              ce_c <= 1'b0;
              clear_c <= 1'b1;
              load_a <= 1'b1;
              shift_a <= 1'b0;
              done <= 1'b0;
            end
        3 : begin
              ce_c <= 1'b1;
              clear_c <= 1'b0;
              load_a <= 1'b0;
              shift_a <= 1'b1;
              done <= 1'b0;
            end
      endcase

   end
// end always
 endmodule
```

Appendix D – scalar multiplier netlist

The following pictures are part of the scalar multiplier netlist.

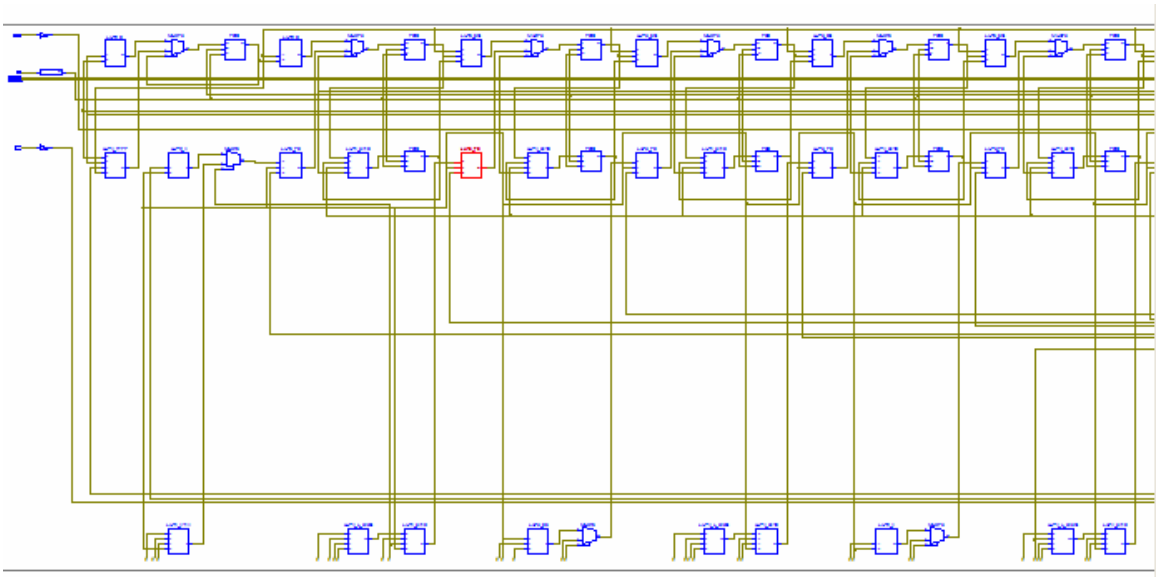


Figure D-1 scalar multiplier netlist (sample 1)

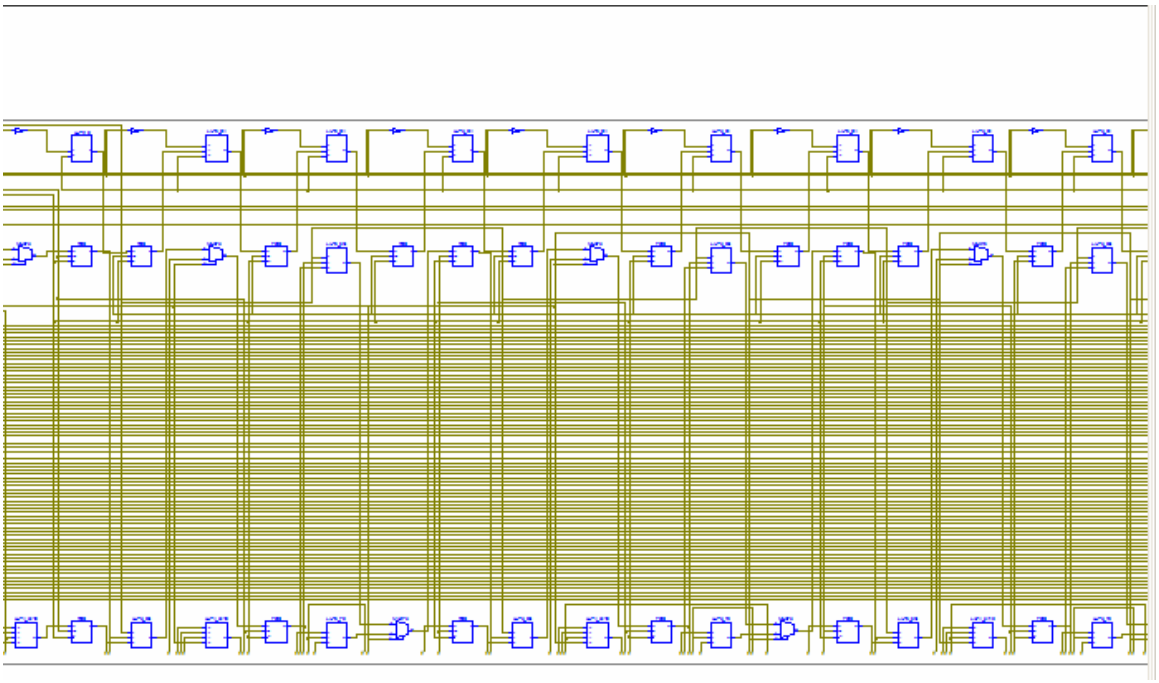


Figure D-2 scalar multiplier netlist (sample 2)

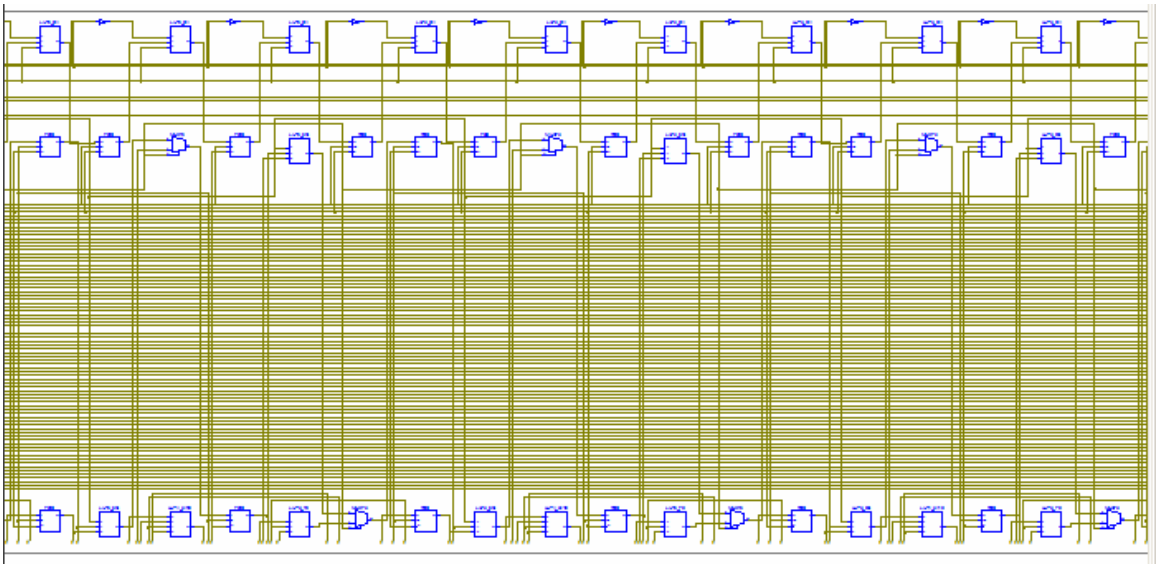


Figure D-3 scalar multiplier netlist (sample 3)

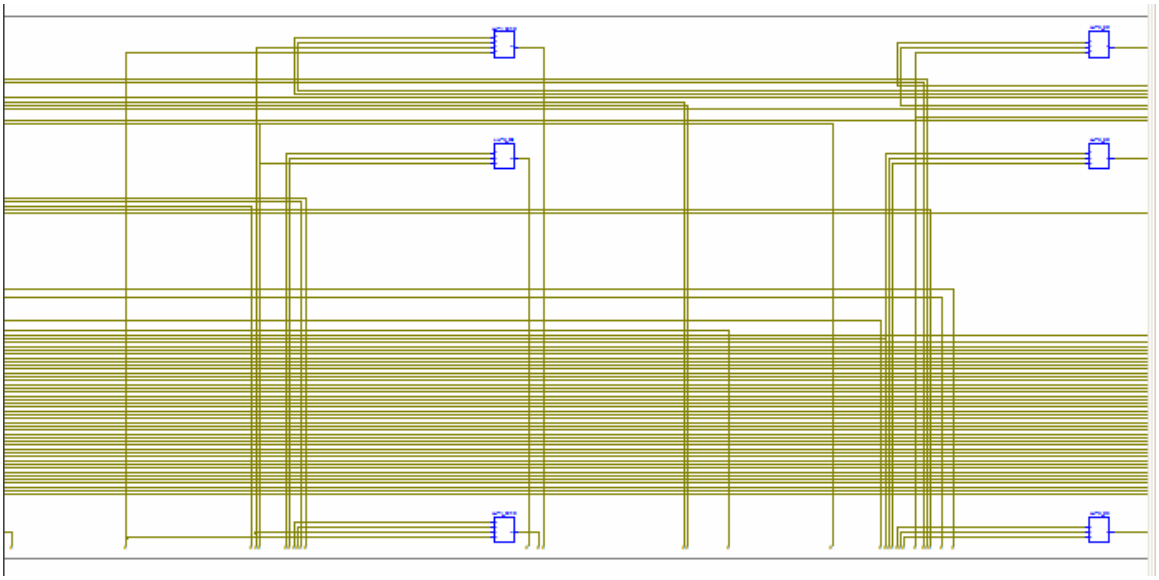


Figure D-4 scalar multiplier netlist (sample 4)

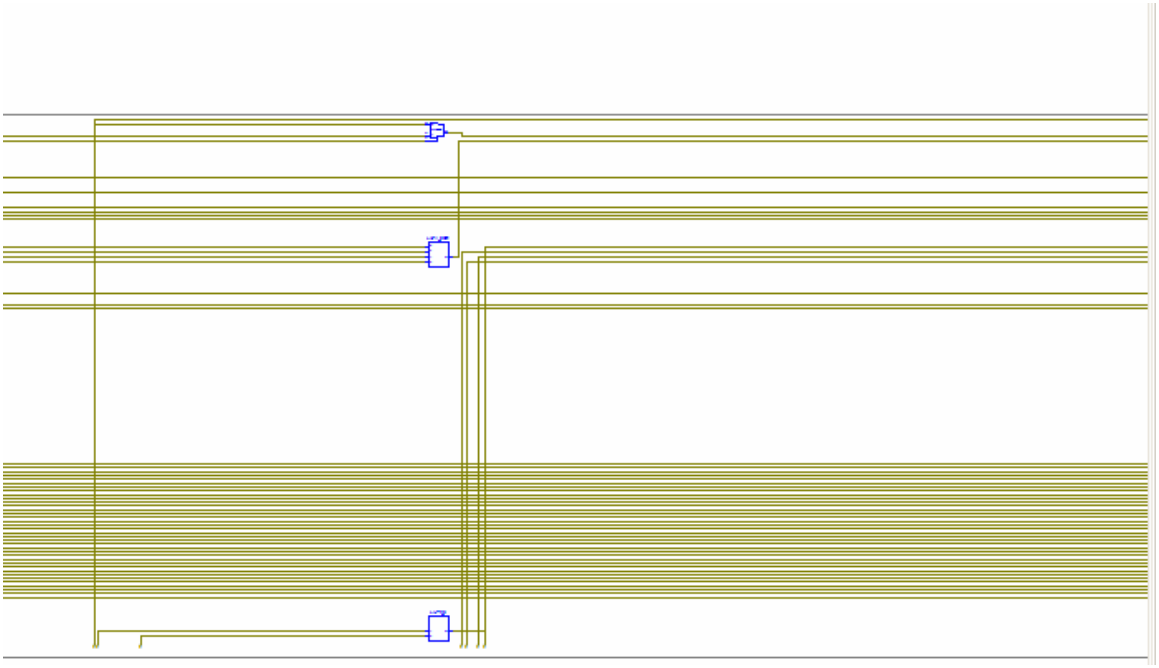


Figure D-5 scalar multiplier netlist (sample 5)

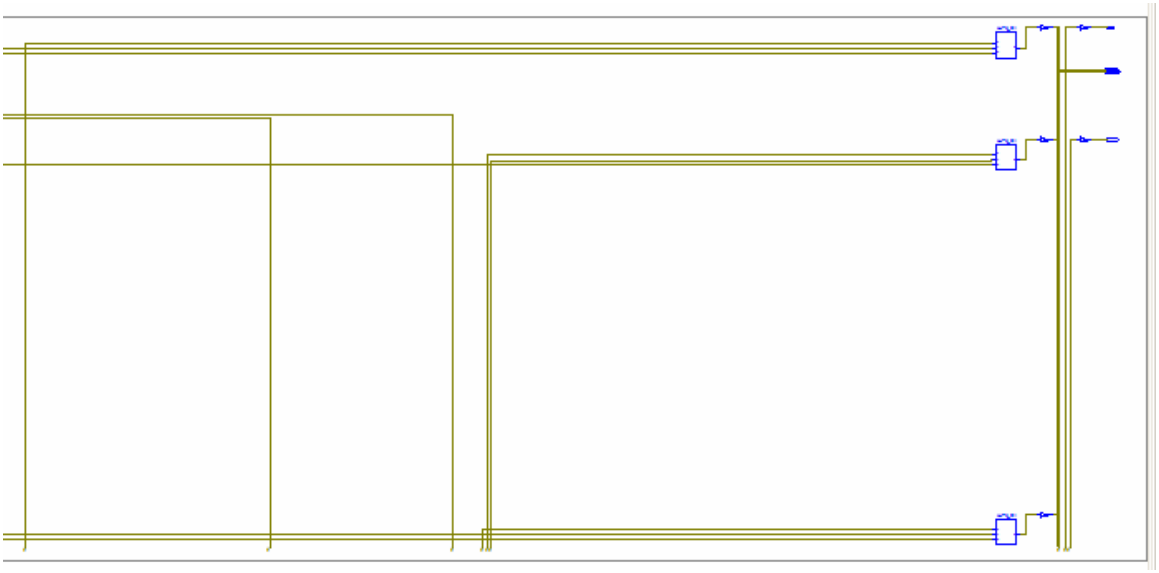


Figure D-6 scalar multiplier netlist (sample 6)

DECLARATION

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Dr Manoj V.N.V, has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

Mubarek Kedir

April, 2008.