Hardware Acceleration of Elliptic Curve Cryptographic (ECC) Scalar Multiplication Unit over Binary Polynomial based Galois Field GF (2^m) using Verilog HDL

Iqbalur Rahman Rokon ,Mohammad Abdul Momen, Md. Ishtiaque Mahmood
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh

***Abstract*- This paper describes algorithms and implementation of those algorithms that will hardware accelerate scalar multiplication unit of ECC over binary polynomial based Galois fields in the particular case of the K-163 NIST-recommended curve. The hardware/circuit design has been done in Verilog and synthesized and simulated in *Altera Quantus-II* and *Modelsim*, respectively. In finite field operations, *GF Division* is used instead of *GF Inversion* which makes the division operation in finite field more independent and faster. Furthermore, instead of *double-and-add* algorithm, *Frobenius Map* algorithm is used which makes the hardware faster.**

Keywords- Elliptic Curve Cryptography, Frobenius Map, Hardware Acceleration, Galois Field.

I. INTRODUCTION

In several cryptographic algorithms, signature schemes, public-key encryption or symmetric key generation Elliptic curve (EC) scalar multiplication is basic operation. Traditionally, scalar multiplication is implemented in software using generalpurpose processors or on digital signal processors. In some cases software time constraints cannot met with instruction-set processors and as a result specific hardware or circuit must be designed for executing very complex operations which will take much less time than software.

Now-a-days for developing specific circuits Field Programmable Gate Arrays (FPGA) is used instead of Application Specific Integrated Circuits (ASIC) because of reprogrammable option, small production quantities and much lower engineering cost than ASIC's.

This paper describes algorithms and implementation of those algorithms that will hardware accelerate scalar multiplication unit of ECC over binary polynomial based Galois fields in the particular case of one of the NIST-recommended Koblitzcurves, namely K-163.. The circuit design has been done in Verilog and synthesized and simulated in *Altera Quantus-II* and *Modelsim*, respectively. In the design, efficient bit-series algorithms are compared and implemented for *Galois Field Operations* and *EC Scalar Multiplication Operation* considering among speed, cost and area constrains, so that the proposed circuit requires consumption of both smaller area and less computational time, being m the degree of the irreducible polynomial (m = 163). After implementing the design of EC scalar multiplication, computational time was reduced from 46.6 µs [1] to 14.6 µs.

II. ELLIPTIC CURVE

Suppose K is finite field and elliptic curve E over K is defined by a non-singular Weierstrass equation [2, 3] $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ where a_1, a_2, a_3, a_4, a_6 belong to$K$. Given$L$ of K is an extension field,the following relation defines the corresponding elliptic curve $E(L)$:

$$E(L) = \{(x,y) \in L \times L: \\ y^2 + a_1xy + a_3y = x^3 + a_2x_2 + a_4x + a_6\} \cup \{\infty\}$$

∞ is *point at infinity*, which is an additional point.Given an elliptic curve E *modulo* p, the number of points of L on the curve is denoted by $E(L)$ [4] and is bounded by:

$$p + 1 - 2\sqrt{p} \leq E(L) \leq p + 1 + 2\sqrt{p} \quad (1)$$

Number of points is approximately equal to thenumber of field elements:

$$\#E(L) \cong p(2)$$

Equation of an elliptic curve $E(L)$ over K is

$$y^2 + xy = x^3 + x^2 + 1 \quad (3)$$

$GF(2^{163})$ is the extension field L and Reduction Polynomial representation of $GF(2^{163})$ is used.

$$F(x) = x^{163} + x^7 + x^6 + x^3 + 1 (4)$$

$E(L)$ can be defined as addition operation. Here neutral element is point at infinity ∞ and the point addition is defined as follows:

Let elements of $E(L)$ be $P(x_1, y_l)$ and $Q(x_2, y_2)$; then

$$P + \infty = \infty + P = P,\ (x\ y) + (x,\ x+y) = \infty;$$

if $P \neq Q$ *and* $P \neq -Q$, *then* $P+Q = (x_3, y_3)$ *where*

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \qquad \lambda = \frac{y1 + y2}{x1 + x2} (5)$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y1$$

if $P = Q$ *and* $P \neq -P$, *then* $P + P = (x_3, y_3)$ *where*

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \qquad \lambda = \frac{y1 + y2}{x1 + x2} (6)$$

$y_3 = \lambda(x_1 + x_3) + x_3 + y1$

For the basic operation of ECC We consider a primitive element P and another element T. kP is the scalar product of a natural number k by a curve point P can be defined as

$$T = kP = P + P + \cdots + P \ (K \text{ times})$$

ECC *Scalar Multiplication* is based on Binary Polynomial Based Galois Field arithmetic. Figure 1 depicts this hierarchical structure of arithmetic operations usedfor elliptic curve cryptography over finite fields.

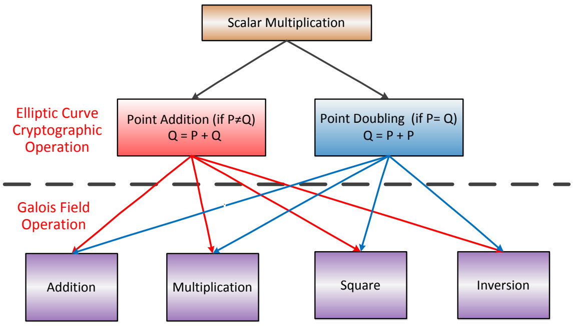

Figure 1: *Basic Hierarchy of Elliptic Curve.*

III. GALOIS FIELD ARITHMETIC

EC over field F_{2^m} includes arithmetic of integer with length m bits. The binary string can be declared as polynomial:
Binary String: $(a_{m-1} \ldots a_1 a_0)$
Polynomial: $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_2x^2 + a_1x + a_0 \, where \, a_i = 0$

A. Addition over GF(2^m)

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

$$C(x) = A(x) + B(x) = \sum_{i=0}^{m-1}(a_i + b_i) \quad (7)$$

The bit additions in $a_i + b_i$ is performed by $modulo$ 2 and translate to an $exclusive - OR \ (XOR)$ operation.

B. Multiplication over GF(2^m)

Multiplication of two elements $a(x), b(x)$ in $GF(2^m)$ can be expressed as

$$C(x) = a(x)b(x) mod\, f(x)$$
$$= a(x)\left(\sum_{i=0}^{m-1} b_i x^i\right) mod\, f(x)$$
$$= \left(\sum_{i=0}^{m-1} b_i\, a(x)x^i\right) mod\, f(x)$$

Therefore, the product $c(x)$ can be computed as

$$c(x) = (b_0a(x) + b_1a(x)x + b_2a(x)x^2 + \cdots + a(x)x^2 + \cdots + b_{m-1}a(x)x^{m-1}) modf(x) \quad (8)$$

In order to compute above equation a quantity of the form $xa(x)$ where $(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0,$ with $a_i \in GF(2)$ has to be reduced modulo $f(x)$. The product $d = xa(x)$ can be computed as follows:

$$d = x(a_0 + a_1x + \cdots + a_{m-1}x^{m-1}) = a_0x + a_1x^2 + \cdots + a_{m-1}x^m \quad (9)$$

Using the fact that $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1x + f_0,$ we have $x^m = f_0 + f_1x + \cdots + f_{m-1}x^{m-1}$, where $f_i s$ are the coefficient of the irreducible polynomial. Substituting this expression in equation (8) we obtain

$$d = d_0 + d_1x + \cdots + d_{m-1}x^{m-1} \quad (10)$$

Where,

$$d_0 = a_{m-1}f_0$$
$$d_i = a_{i-1} + a_{m-1}f_i,\ i = 1,2, \ldots, m-1 \quad (11)$$

Assume that the function,

```
functionProduct_alpha_A(a,f:
poly_vector) return poly_vector
```

implementing Eq. (9) according to Eq. (10) & Eq. (11) and therefore the polynomial $xa(x) mod\, F(x)$ has been defined, where $poly\_vector$ is a bit vector from 0 to $m - 1$.

Assume also that the functions

```
function m2abv(x: bit; y: poly_vector)
return poly_vector
function m2xvv(x, y: poly_vector)
return poly_vector
```

In a least-significant-bit (LSB) multiplier, the coefficients of $b(x)$ are processed starting from the least-significant bit b_0 and continue with the remaining coefficients one at a time in ascending order. Thus multiplication according to this scheme is performed in the following way:

$$c(x) = a(x)b(x) mod\, f(x)$$

$$= (b_0a(x) + b_1a(x)x + b_2a(x)x^2 + \cdots + b_{m-1}a(x)x^{m-1}) mod\, f(x)$$
$$= (b_0a(x) + b_1(a(x)x) + b_2(a(x)x^2) + \cdots + b_{m-1}(a(x)x^{m-1})) mod\, f(x)$$
$$= (b_0a(x) + b_1(a(x)x) + b_2(a(x)x)x + \cdots + b_{m-1}(a(x)x^{m-2})x) mod\, f(x) \quad (12)$$

Algorithm 1: LSB-first multiplier
for i in 0 . . m-1 loop c(i) := 0; end loop;
for i in 0 .. m-1 loop
c := m2xvv(m2abv (b(i) , a,), c)
a := Product_alpha_A(a,f)
end loop;

C. Squaring over GF(2^m)

$a^2 modulo\ f$ computation is done br a specific synthesized circuit [8]. It can be shown that $a^2 = s + t + u\ where,$
$s = s_{162}z^{162} + \cdots + s_1z + s_0$
$With s_j = a_{(j+163/_2)} if j is odd,$
$s_j = a_{(j/_2)} if j is even (13)$
$t = t_{162}z^{162} + \cdots + t_1z + t_0$
$With t_j = 0\ if j < 7,$
$t_7 = a_{82},$
$t_j = a_{(j+156/_2)} if j is even \& j \geq 8,$
$t_j = a_{(j+157/_2)} if j is odd \& j \geq 8, (14)$

$u = u_{162}z^{162} + \cdots + u_1 z + u_0$
$With u_0 = a_{160}$
$u_1 = a_{160} + a_{162}$
$u_2 = a_{161}$
$u_3 = a_{160} + a_{161}$
$u_4 = a_{82} + a_{160}$
$u_5 = a_{161} + a_{162}$
$u_6 = a_{83} + a_{160} + a_{161}$
$u_7 = 0$
$u_8 = a_{84} + a_{160} + a_{161}$
$u_9 = 0$
$u_{10} = a_{85} + a_{161} + a_{162}$
$u_{11} = 0$
$u_{12} = a_{86} + a_{162}$
$u_j = 0 \; if j > 12 \; \& j odd,$
$u_j = a_{\left(j+160/_2\right)} if J > 12 \; and j even$ (15)

All outputs $a^2{}_i$, but $a^2{}_6$, $a^2{}_8$ and $a^2{}_{10}$, are Boolean functions of less than five variables, while $a^2{}_6$, $a^2{}_8$ and $a^2{}_{10}$ are five-variable Boolean functions. Thus, the computation time of a^2 is approximately equal to the computation time of a five-variable Boolean function.

D. Inversion over GF(2^m)

Extended Euclidean algorithm for polynomials

The greatest common divisor (GCD) of 'a' and 'b' ('a' and 'b' are binary polynomials and they are not zero), are denoted by $d = gcd(a, b)$. 'd' is the largest common divisor. In the classical Euclidean algorithm, $deg(b) \geq deg(a)$ computes the gcd of binary polynomials. 'b' is divided by 'a' to obtain a quotient 'q'. A remainder 'r' satisfying $b = qa + r$ and $deg(r) < deg(a)$.
In such state, the problem in determining $gcd(a, b)$ reduces the computation of $gcd(r, a)$, where (r, a) have lower degrees than (a, b). The process should be repeated until one of the arguments reaches to zero. Then the result is immediately obtained since $gcd(0, d) = d$. The algorithm is reached and ended since the degrees of the remainders decrease. The Euclidean algorithm can be extended to find binary polynomials 'x' and 'y' satisfying $ax + by = d$ where $d = (gcd\ a, b)$.

$ax_1 + by_1 = g_1$ (16)
$ax_2 + by_2 = g_2$ (17)

The algorithm ends when u value reaches zero, in the case of $g_2 = \gcd(a, b)$ & $ax_2 + by_2 = d$. The next algorithm is used to compute $gcd(a, b)$ (Hankerson, et al. 2004) (Liu 2007).

Algorithm 2: Inversion in F_{2^m} using the extended Euclidean algorithm

```
for i in 0 .. m loop
s(i) := f(i); r(i) := a(i); v(i) := 0;
u(i) := 0; auxm(i) := 0;
end loop;
u(0) := 1; d := 0;
for i in 1 .. 2*m loop
if r(m) = 0 then
r := rshiftm(r);
u := rshiftm(u);
d := d + 1;
else
if s(m) = 1 then
s := m2xvvm(s,r);
v := m2xvvm(v,u);
end if;
s := rshiftm(s);
if d = 0 then
auxm := s; s := r; r := auxm;
auxm := v; v := u;
u := rshiftm(auxm);
d := 1;
else u := lshiftm(u); d := d - 1; end if;
end if;
end loop;
```

E. Division over $GF(2^m)$

Given three polynomials

$$g = g_{m-1}z^{m-1} + g_{m-2}z^{m-2} + \cdots + g_1 z + g_0$$
$$h = h_{m-1}z^{m-1} + h_{m-2}z^{m-2} + \cdots + h_1 z + h_0$$
$$f = f_{m-1}z^{m-1} + f_{m-2}z^{m-2} + \cdots + f_1 z + f_0$$

The quotient $q = gh^{-1}\ modulo\ f$ can be computed with an algorithm based on the following properties(gcd = $greatest\ common\ divider$): given two polynomials $a\ and\ b$ where b is not divisible by z, that is $b_0 = 1$, then

$if\ a\ is\ divisible\ by\ z, that\ is\ a_0 = 0, then\ gcd(a, b) = gcd(a/z, b)$ (18)

$if\ a\ is\ not\ divisible\ by\ z, that\ is\ a_0 = 1, then\ gcd(a, b) = gcd((a + b)/z, b) = gcd((a + b)/z, a)$(19)

The following formal algorithm, in which the function divides by $z(c, J)$ computes, $cz^{-1} modulo\ f$, generates the quotient $q = gh^{-1}\ modulo f$

Algorithm 3: Division of polynomials modulof (binary algorithm)

```
a := f; b := h; c := zero; d := g; alpha := m; beta := m-1;
while beta >= 0 loop
if b(0) = 0 then
b := shift_one(b); d := divide_by_x(d, f); beta :=beta - 1;
else
old_b := b; old_d := d; old_beta := beta;
b := shift_one(add(a, b));
d := divide_by_x(add(c, d),f);
if alpha > beta then
a := old_b; c := old_d; beta := alpha - 1; alpha :=old_beta;
else beta := beta - 1;
end if;
end if;
end loop;
if b(0) = 0 then z := d; else z := c; end if;
```

The sum of two binary polynomials amounts to the component-by-component XOR, and the division by z to a one-bit left shift. The more complex primitive is the division by $z\ mod f$ It is computed as follows:

$cz^{-1} mod\ f = c_0 z^{m-1} + (c_{m-1} + c_0 f_{m-1})z^{m-2} + (c_{m-2} + c_0 f_{m-2})z^{m-3} + \cdots + c_1 + c_0 f_1$(20)

The circuit corresponding to the computation of cz^{-1}or $(c-d)z^{-1}$ *modulo f*, according to the value ofa_0 , is the more time consuming operation). The number of steps of algorithm 3 is smaller than$2m$. Thus, the total computation time of $q = gh^{-1}$ *modulo f* is smaller than $2m$ times the computation time of the circuit, that is $2m$ times the computation time of a five-variable Boolean function if f is assumed to be constant.

In this paper instead of using *GF Inversion*, we used *GF Division* for division operation. By using *GF Inversion*(Figure 2)for a division operation first the denominator (y) had to inverted (y^{-1}) and then it had to multiply (x *{y^{-1}}) to get division output which makes the system slow and dependable. By using *GF Division*(Figure 3), now division operation first of all independent and secondly requires less computation time.

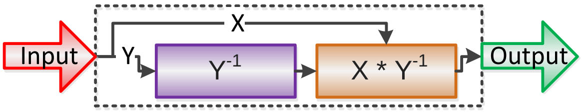

Figure 2: *Division Using GF Inversion*

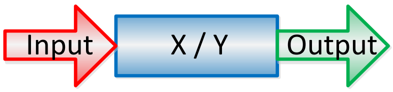

Figure 3: *GF Division*

IV. POINT MULTIPLICATION

The parameter sets of the K-163 binary koblitz curves standardized by NIST [10] for ECC is below (hexadecimal)

p(t) = 800000000000000000000000000000000000000C9, a = 1, G_x =2fe13c0537bbc11acaa07d793de4e6d5e5c94eee8, G_y = 289070fb05d38ff58321f2e800536d538ccdaa3d9, r = 5846006549323611672814741753598448348329118574063

where p(t) is the reduction polynomial, a is the curve coefficient, G_x and G_y is the x and y coordinates of the base generator point G, r is the base point's order.

A. Double-and-Add (basic) Algorithm

Let the binary representation of k be $(k_{m-1}, k_{m-2,\ldots,}k_0)$ that is $k = k_{m-1}*2^{m-1} + k_{m-2}*2^{m-2} + \ldots + k_0*2^0$.Then according to the following schemeKP can be computed (right to left)

$$kp = k_0p + k_1(2p) + k_2(2^2p) + \ldots + k_{m-1}(2^{m-1}p) \quad (21)$$

(21) can be implemented with algorithm 1 which consists of m iteration steps, each of them including atmost two function calls point adding *(Q = Q+P)* and/or point doubling*(P = P+P)*.

Algorithm 4: Scalar multiplication *(Q = kP)*

```
Q:= point at infinity;
for i in .. m-1 loop
P := P + P;
if k(i) 1 then Q := Q + P; end if;
end loop;
```

B. Frobenius Map

The chosen curve is a Koblitz curve for which an interesting property can be used. Define the Frobenius map [11] X from E(L) to E(L):

$$\tau(\infty) = (\infty), \tau(x, y) = (x^2, y^2) \quad (22)$$

It can be demonstrated that$P + P = -\tau^2(P) + \mu\tau(P)$ with $\mu = 1$ if $c = 1$ and $\mu = -1$ if $c = 0$

More generally, it is possible to express kP under the form:

$$kP = r_{t-1}\tau^{t-1}(P) + r_{t-2}\tau^{t-2}(P) + \ldots + r_1\tau(P) + r_0P \text{ with } ki \in \{-1, 0, 1\} \quad (23)$$

to which correspond the formal algorithms 2 in which frobenius is a function computing relations (8), which in turn includes three computation primitives: *adding* (whenki = 1), *subtracting* (when ki = -1) and τ. The difference with the basic algorithm is that point doubling has been substituted by the Frobenius map computation, that is squaring, an easy operation over a binary field. Obviously it remains to express *kP* under the form (4). Given two integers a and b, define an application $cc = a + bt$ from *E(L)* to *E(L)*: $\alpha(P) = aP + b\,\tau(P)$. Then look for two integers a' and b' such that

$$\alpha(P) = \alpha'(\tau(P)) + rP$$
$$\text{where } \alpha' = a' + b'\tau \text{ and } r \in \{-1, 0, 1\} \quad (24)$$

To summarize:
1. If a is even, then $r = 0$, $b' = -a/2$, and $a' = b - \mu b' = b + \mu a/2$.
2. If a is odd and $a_1 \oplus b_0 = 0$, then $r = 1$, $b' = -(a-1)/2$, and $a' = b - \mu b' = b + \mu(a-1)/2$.
3. If a is odd and $a_1 \oplus b_0 = 1$, then $r = -1$, $b' = -(a+1)/2$ and $a' = b - \mu b' = b + \mu(a+1)/2$.

Equation defines a kind of integer division of α by τ, that is $\alpha = \alpha'\tau + r$ with $r \in \{-1, 0, 1\}$

By repeatedly using the previous relation, an expression of α can be computed:

$$\alpha = \alpha_1\tau + r_0$$
$$\alpha_1 = \alpha_2\tau + r_1$$
$$\ldots$$
$$\alpha_{t-1} = \alpha_t\tau + r_{t-1} \quad (25)$$

with$r_i \in \{-1, 0, 1\}$. Thus (multiply the second equation by τ, the third one by τ^2, and so on, and sum up the t equations)

$$\alpha = r_0 + r_1\tau + \ldots + r_{t-1}\tau_{t-1} + \alpha_t\tau_t \quad (26)$$

Algorithm 5: Point multiplication ($Q = kP$), Koblitz curve

```
Q :=point_at_infinity;
for i in 0 .. t-1 loop
    if r(i) = 1 then Q := Q + P;
elsif r(i) = -1 then Q := Q-P;
    end if;
    P := frobenius(P);
end loop;
```

Assume that algorithm 5 is used. The coefficients *ki* can be computed in parallel with the other operations of algorithm 5. As the value of t is not known in advance, the computation is performed as long *as* $a_j = a_j + b_j t \neq 0$, that is $a_j \neq 0$ and $b_j \neq 0$. Initially $a_0 = k$, that is $a_0 = k$ and $b_0 = 0$:

Algorithm 6:Point multiplication *(Q = kP)*, Koblitz curve, τ -ary representation
Q := point_at_infinity; a := k; b := 0;
if a /= O then
loop
if a(0) = 0 then r_i := 0;
elsif (a(1) + b(0)) mod 2 = 0 then
r_i := 1; Q := Q + P;
else r_i := -1; Q := Q - P; end if;
old_a := a; a := b + (old_a -r i)/2;
b (r i - old a)/2;
if a = 0 and b = 0 then exit;
end loop;
end if;

To summarize, doubling has been substituted by squaring, an operation executable in one clock cycle. Furthermore, among two successive coefficients k_i, at least one is equal to 0, so that the number of non-zero coefficients K_i is smaller than m. Thus, the computation of *KP* includes at most *m* operations (adding or subtracting), so that the total computation time should be roughly half the computation time of the basic algorithm.

The algorithm 7, deduced from algorithms 5 and 6, computes $Q = kP$, with $k < n$ and P of order n. At the end of step number i of algorithm 5, $Q = k_0P + k_l\ \tau(P) + k_2\tau^2(P) + \ldots + k_{i-1}\ \tau^{i-1}(P)$. If can be shown that, unless all coefficients $k_0, k_1, \ldots, k_{i-1}$, are equal to 0, $Q \neq \infty$. As before, instead of defining a specific representation for 0o, Boolean flags *Q* infinity and *R* infinity are used. Figure 4 reflects full operation of algorithm 7.

Algorithm 7: Point multiplication (k < 2m), Frobenius map
Q_infinity true; xxP =xP; yyP = yP;
a = k; b= zero;
while ((a /= zero) or (b /= zero)) loop
if a(0) = then r_i = 0;
elsif a(l) = b(0) then r_i = 1;
if Q infinity then
(xQ,yQ) =(xxP,yyP); Q_infinity = false;
else (xQ,yQ) = adding ((xxP , yyP) , (xQ , yQ));
end if;
else r_i = -1;
if Q_infinity then xQ = xxP
yQ = xxP + yyP; Q_infinity = false;
else (xQ,yQ) = adding ((xxP , xxP + yyP) , (xQ , yQ));
end if;
end if;
xxP = square(xxP); yyP = square(yyP); old a = a;
a = b + (old_a - r_i)/2; b = (r_i - old_a)/2;
if a = 0and b = 0 then exit
end loop;

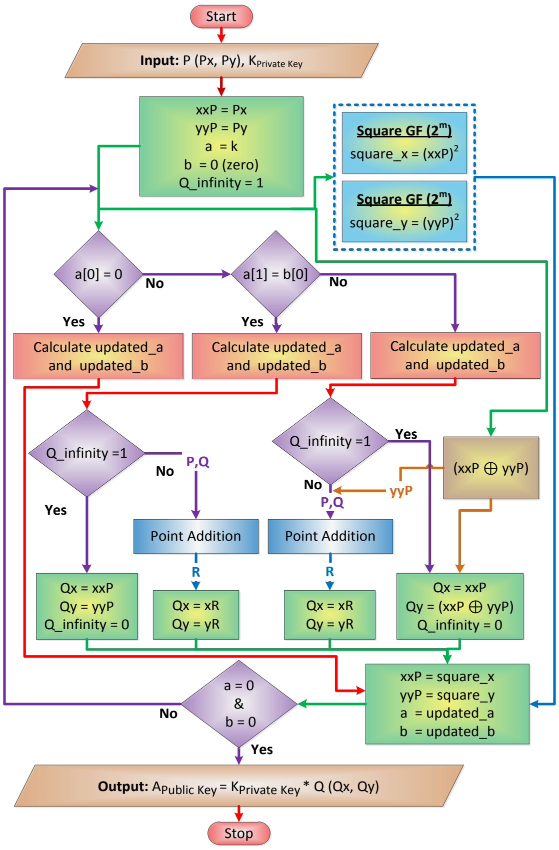

Figure 4: *Flowchart Point multiplication (k < 2m), Frobenius map*

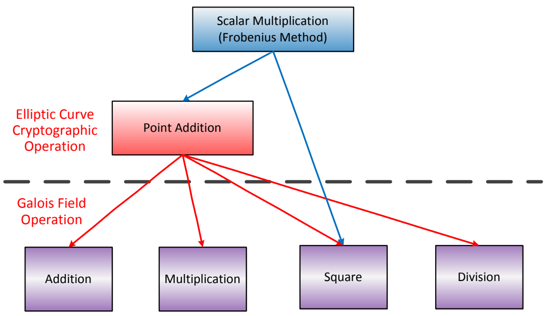

Figure 5: *Design Hierarchy of Point multiplication (Frobeniusmap)*

After applying Frobenius map as a point multiplication algorithm and above stated Galois Field algorithms the design hierarchy changes drastically (Figure 5). Only for Frobenius map speed has boosted about 50% just because of we are using *GF Square* instead of *point doubling*. Figure 5 Design Hierarchy of Point multiplication (Frobenius map).

V. HARDWARE DESIGN OF THE SYSTEM

A. Top View

Figure 6 shows the block diagram of *Scalar Multiplication* and Table I is the pin description *Scalar*

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 4 Issue 05, May-2015

*Multiplication*block.

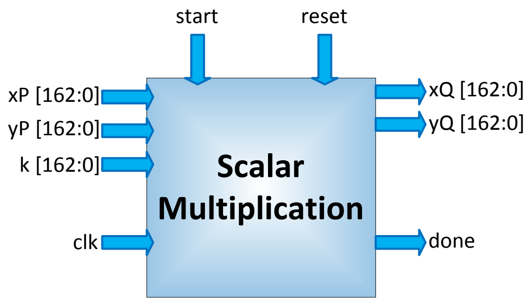

Figure 6: Block Diagram of Scalar Multiplication

Table I
Pin Description of Top Block

Pin Name	Input/Output	Description
xP [162:0]	Input	Curve Generator Point Co-ordinate x
yP [162:0]		Curve Generator Point Co-ordinate y
K [162:0]		Private Key
reset		Reset Flag
start		Start Flag
clk		System Clock
xQ [162:0]	Output	Public Key/Scalar Multiplication Co-ordinate x
yQ [162:0]		Public Key/Scalar Multiplication Co-ordinate y
done		Done Flag

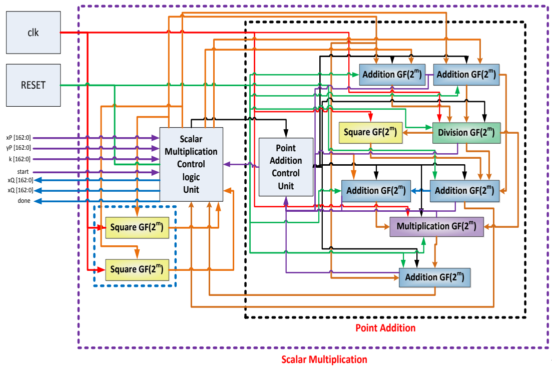

Figure 7: *Internal Block diagram of Scalar Multiplication*

Figure 7 displays the entire hardware internal block diagram of *Scalar Multiplication Unit.* In this diagram, how the input/output pins are connected to *Scalar Multiplication Control Logic Unit (SMCLU)* (Figure 8)*and the SMCLU* is connected to the *Point Addition* and *GF Square* unit is shows. Furthermore, displays how the input/output connections of *Point Addition* and GF Arithmetic Operations are controller by *Point Addition Control Unit*(Figure 9).

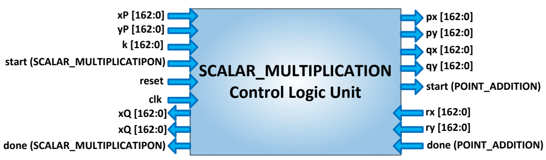

Figure 8: *Scalar Multiplication Control Logic Unit*

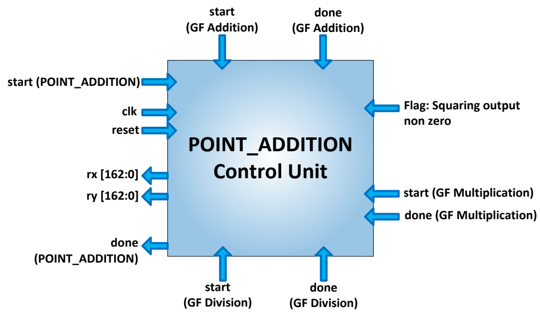

Figure 9: *Point Addition Control Unit*

VI. EXPERIMENTAL RESULTS AND DISCUSSION

A. *Synthesis Result*

Altera Quantus-II was used to Analyze and synthesize the design. Synthesis report is shown in Table II, III and IV. Figure 10 shows the RTL view of *Altera Quantus-II* synthesis tool.

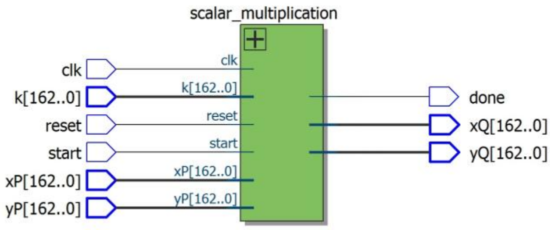

Figure 10: *Register Transfer Levelviewof Scalar Multiplication from Altera Quantus-II*

Table II
Quantus II Flow Summary

Flow Status	
Quartus II 64-Bit Version	14.1.0 Build 186 12/03/2014 SJ Web Edition
Revision Name	scalar_multiplication
Top-level Entity Name	scalar_multiplication
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	3,982
Total registers	6576
Total pins	819

Table III
Resource Usage summary

Resource	**Usage**
Estimate of Logic utilization (ALMs needed)	3585
Combinational ALUT usage for logic	4932
-- 7 input functions	1
-- 6 input functions	181
-- 5 input functions	846
-- 4 input functions	1492
-- <=3 input functions	2412
Dedicated logic registers	6576
I/O pins	819

Resource	Usage
Maximum fan-out node	clk~input
Maximum fan-out	6576
Total fan-out	40417
Average fan-out	3.07

Table IV
General Register Statistics

Statistic	Value
Total registers	6576
Number of registers using Synchronous Clear	2612
Number of registers using Synchronous Load	1469
Number of registers using Clock Enable	3770

B. *Simulation Result*

*ModelSim*was used to simulate the design. Figure 11 and 12 gives a full view of the simulation of *Scalar Multiplication*and *Point Addition*, respectively.

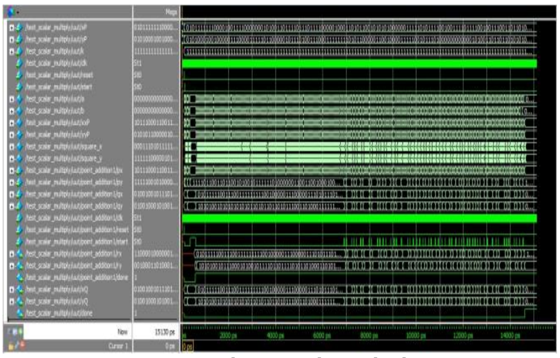

Figure 11: *Simulation Scalar multiplication*

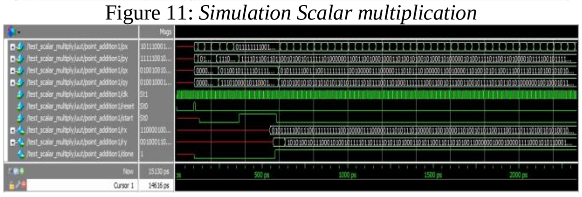

Figure 12: *Simulation Point Addition*

VII. CONCLUSION

This paper represents an implementation of hardware accelerated Elliptic Curve co-processor components, *Scalar Multiplication* and *Point addition.* By only using *Frobenius Map*for *Scalar Multiplication*, the computation is accelerated about 50%. Furthermore, in *Galois Field* operations most efficient algorithms are used for *GF Addition*, *GF Multiplication*, *GF Square* and *GF Division.* As a result, after implementing the algorithms, computational time for *Scalar Multiplication* has been reduced from 46.6 µs [1] to 14.6 µs which is about 68.67% faster.In order to further accelerate the computation process more efficient algorithms are required perform the field arithmetic operations and *Point/Scalar Multiplication*operation. At the circuit level, some optimizations are required which will even accelerate the process.

ACKNOWLEDGMENT

Authors would like to express their gratitude towards Almighty for giving them the opportunity to do this work. They would also like to thank Dr. Arshad M. Chowdhury, Chairman of the ECE Department at North South University.

REFERENCE

[1]. MubarekKedir (2008) *Hardware Acceleration of Elliptic Curve Based Cryptographic Algorithms: Design and Simulation*
[2]. D.Hankerson, A.J.Menezes, and S.Vanstone, Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004.
[3]. F.Rodriguez-Henriquez, N.A.Saqib, A.Diaz Perez, and Q.K.Koq, Cryptographic Algorithms on Reconfigurable Hardware, Springer, 2007.
[4]. I.VBlake, G.Seroussi, and N.Smart, Elliptic Curves in Cryptography, Cambridge University Press, 2002.
[5]. N.Koblitz, A course in Number Theory and Cryptography,Springer, 1994.
[6]. Ch.-H.Wu, Ch.-M.Wu, M.-D.Shieh, and Y.-T.Hwang, "Novel Algorithms and VLSI Design for Division over GF(2m)", IEICE Transactions Fundamentals, vol.E85-A, no 5, May 2002, pp. 1129-1139.
[7]. J.-P.Deschamps and G.Sutter, "Hardware Implementation of Finite-Field Division", ActaApplicandaeMathematicae, vol.93, no 1-3, September 2006, pp.119-147.
[8]. J.-P.Deschamps, "Squaring over GF(2163)", UniversityRovira i Virgili, Inter Research Report DESS21, Oct 2006.
[9]. Paar, C., &Pelzl, J. (2009). Understanding cryptography a textbook for students and practitioners. Berlin: Springer.
[10]. NIST Koblitz Curves Parameters. (n.d.). Retrieved May 16, 2015, from http://en.wikisource.org/wiki/NIST_Koblitz_Curves_Parameters
[11]. Deschamps, J., & A, J. (2009). Hardware implementation of finite-field arithmetic. New York: McGraw-Hill.