DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI - 600036

ELLIPTIC CURVE CRYPTO PROCESSOR

A Project Report

Submitted by

PATWARDHAN VINOD VIDYADHAR

EE19M055

In the partial fulfilment of requirements

For the award of the degree

Of

MASTER OF TECHNOLOGY

June 2021

CERTIFICATE

This is to undertake that the Project report titled **ELLIPTIC CURVE CRYPTO PROCESSOR**, submitted by me to the Indian Institute of Technology Madras, for the award of M.Tech, is a bona fide record of the research work done by me under the supervision of **Prof Veezhinathan Kamakoti**. The contents of this Project report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

In order to effectively convey the idea presented in this Thesis, the following work of other authors or sources was reprinted in the Thesis with their permission:

- Rebeiro, C.(2009).ARCHITECTURE EXPLORATIONS FOR ELLIPTIC CURVE-CRYPTOGRAPHY ON FPGAS. Doctoral thesis, DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT-Madras, Chennai – 600036

Place: Chennai 600 036

Date: 18 June 2021

PATWARDHAN VINOD VIDYADHAR

EE19M055

Prof Veezhinathan Kamakoti

Research Guide

Dr. Anbarasu Manivannan

Research Co-Guide

ACKNOWLEDGEMENTS

Foremost, I would like to thank my guide Prof Veezhinathan Kamakoti, who gave me an opportunity to be a part of Shakti Project.

I would also like to thank Dr. Anbarasu Manivannan for being my co-guide, which me gave an opportunity for taking up the project outside of Electrical department.

I am grateful to Project Associates Arjun and Varun for their encouragement, advice, and help whenever needed. I appreciate their constant involvement in my project from the scratch.

I would like to take this opportunity to acknowledge my friends for providing all the emotional support and entertainment. I would like to thank my family for their love and encouragement, without their support this M.Tech program would not have been possible.

Patwardhan Vinod Vidyadhar

ABSTRACT

Communications have grown at a breakneck pace in the modern era. Online banking, personal digital assistants, mobile communication, smart-cards, and other applications have highlighted the importance of security in resource-constrained contexts. Despite there being protocols for security purposes such as RSA, the ECC is gaining popularity for its smaller key sizes offering the same level of security.

Hardware efficient architectures are put together to form the processor. For finite field multiplication, General Karatsuba which efficient for smaller bit size multiplications and Simple Karatsuba suitable for large bit size multiplications, are combined together as Hybrid Karatsuba, to gives best performance.

Multiplicative inverse being another crucial operation in ECC, is implemented using Quad-Itoh Tsujii algorithm instead of the conventional Itoh-Tsujii algorithm. This ensures least time for computation.

The above two proposed hardware primitives in Chester's work are implemented to form an accelerator. The performance of this accelerator is significantly enhanced due to these proposed primitives. Further, this accelerator is modified to be resistant from Simple timing and Power analysis side channel attacks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AU	Arithmetic Unit
ASIC	Application Specific Integrated Circuit
ECC	Elliptic Curve Cryptography
ECCP	Elliptic Curve Crypto Processor
EEA	Extended Euclid's Algorithm
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GF	Galois Field
ITA	Itoh-Tsujii Algorithm
LD	Lopez-Dahab
LUT	Look Up Table
RSA	Rivest Shamir Adleman
SPA	Simple Power Analysis
SR-ECCP	SPA Resistant Elliptic Curve Crypto Processor
BSV	Bluespec System Verilog

CHAPTER 1

INTRODUCTION

Communications across wired and wireless networks have increased dramatically in this period. With people preferring online shopping, the daily transactions have increased. These transactions involve sensitive data such as account details, which need to be kept private, and also authorised users should only be able to initiate these transactions. Also, there has been recent rise in *Cryptocurrency* investment, there is a need to ensure that funds can only be spent by their rightful owners. Hence, there is need of robust security framework.

Cryptology is the science concerned with data communication and storage in encrypted form. This ensures only authorised users with the right key can decrypt the encrypted data. Cryptology comprises of *Cryptography* and *Cryptanalysis*. The former entails the study and application of techniques, which encrypt the data, making it accessible only to the intended user. Cryptanalysis, on the other hand, is the study of decrypting crypto systems and recovering the hidden data.

Based on the key, cryptography can be classified as *Symmetric* and *Asymmetric*. In *Symmetric key* algorithms, a single key is use for both encryption and decryption. This ensures fast process but the secret key has to be shared between parties for communication. On the other hand, *Asymmetric key* algorithms uses two keys, public and private. The former is used to encrypt the data and also it is generally known whereas the latter is used to decrypt and is kept confidential. Hence, asymmetric key algorithms are complex and slow but, the underlying primitives used are based on integer factorization and discrete logarithm problems which are hard to crack. ECC also belongs to this category.

Another component of Cryptology, Cryptanalysis deals with exploiting the cryptographic algorithms weakness to get the secured data. Conventional brute force method

to get the encrypted data isn't feasible anymore as it would require large data and also computation time. Hence, recent Cryptanalysis techniques focus on implementation of algorithms to attack. These techniques include gathering information from timing, power, acoustic, radiation characteristics of the systems and using them to get the secret key. Optimised architectures are more prone to these side channel attacks. Hence, we need to take care of these side channel attacks while designing our cryptographic systems.

1.1 Motivation

Efficient implementation of algorithms is required to take care of the complex mathematical computations. There are two ways to achieve this: *Software* and *Hardware*. *Software* implementation [Rebeiro *et al.* (2006)] is low cost and easy to tweak. However, due to architecture limitations of microprocessor on which this is implemented, this scheme cannot perform certain large computations efficiently. Therefore, a dedicated *Hardware* implementation is suitable for public key cryptographic algorithms which involve such large computations. However, design of hardware is expensive and time consuming process, also memory being the problem. Hence, design which can utilise the resources of *FPGA* efficiently and as well take care of timing constraints, be preferred. With the increasing security threats there is also a need to design architectures and algorithms which are not susceptible to attacks.

CHAPTER 2

Mathematical Background

To understand *Elliptic Curve Cryptography* (ECC) we need to know about the underlying mathematical operations of *Finite field, Elliptic curve group, scalar multiplication* and *ECC primitives*. Fig. 2.1 shows the various operations involved in building ECCP.

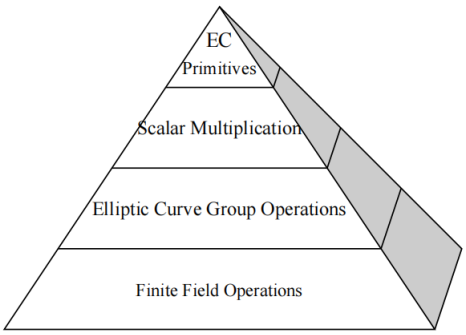

Fig. 2.1: Hierarchy

Rebeiro (2009)

First of all, we need a *field*(or commutative ring) which has finite number of elements to work upon. Hence, we choose finite fields also known as *Galois* fields denoted by $GF(p^m)$. Here p(prime number) is known as the *characteristic* of the field and $m \geq 0$. The *Order* of finite field is the number of elements present in that field and is equal to p^m.

There are two kinds of finite field which are popular in Cryptography: *Prime field* and *Binary finite field*. In prime fields, *m=1*, whereas in binary fields, *p=2*. Binary Galois field ($GF\ (2^m)$) elements can be represented using *m* bits, which is not possible in prime fields. Also, in binary fields the hardware required for arithmetic operations like addition and squaring is quite simpler. Therefore, we choose to work on binary fields.

2.1 Binary Finite Fields

Any *GF(2^m)* element can be expressed as, a(x) = $a_{m-1}\ x^{m-1}$ + . . . + $a_1 x$ + a_0, where the coefficients belong to GF(2).

Alternatively, a(x) can be represented as binary number $(a_{m-1}, ..., a_1, a_0)$ which allows for easy storage and computation. Consider the polynomial in the field $GF(2^{12})$, $x^{11} + x^8 + x^5 + x^3 + x + 1$. This can also be represented as $(100100101011)_2$.

Various arithmetic operations can be performed with ease in Binary finite field. Addition and Subtraction are similar in *GF(2)* as there is no carry forwarding involved, hence these are computed using *XOR* operation. Some of the arithmetic operations are discussed below.

Addition/Subtraction : Consider two elements in the field *GF(2^m)* as shown below,

$$c(x) = \sum_{i=0}^{m-1} c_i x^i \qquad d(x) = \sum_{i=0}^{m-1} d_i x^i$$

the *addition/subtraction* of these is given by

$$c(x) \pm d(x) = \sum_{i=0}^{m-1} (c_i \oplus d_i) x^i \tag{2.1}$$

Squaring : Considering the same element c(x) $\in$ *GF(2^m)* mentioned above, its square is given as follows.

$$c(x)^2 = \sum_{i=0}^{m-1} c_i x^{2i} \textit{mod p(x)} \tag{2.2}$$

After squaring the length of the input increases such that, for *m* bits input the output would be of *2m-1* bits as shown in Fig. 2.2.

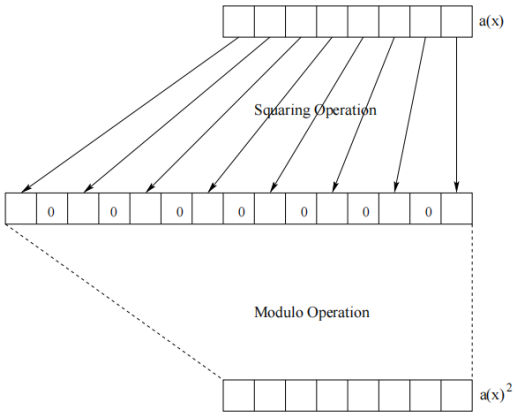

Fig. 2.2: Squarer

Rebeiro (2009)

Multiplication is not so straightforward as addition and squaring. Multiplying the above elements c(x) and d(x) would give,

$$c(x).d(x) = (\sum_{i=0}^{n-1} d(x)c_i x^i) mod\ p(x) \tag{2.3}$$

We will look at finite field multiplier and inversion with great detail in the following chapters.

Modular Reduction : The squaring and multiplication results have bit sizes greater than *m* for *GF(2^m)*. Hence, to get back the result to intended number of bits a modular operation is required. This is done by dividing the output with an irreducible polynomial. [Mahboob (2004)]

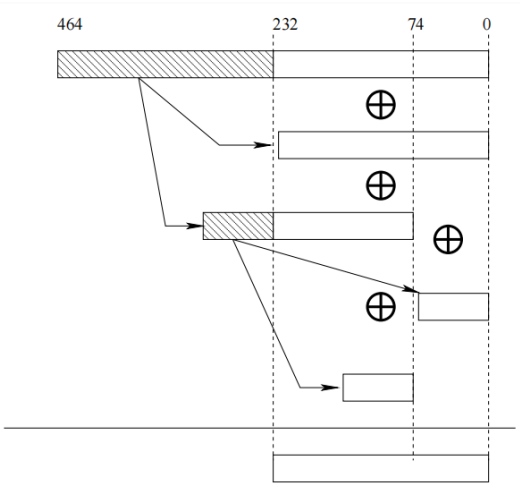

Fig. 2.3: Modular Reduction

Rebeiro (2009)

Consider the irreducible trinomial $x^m + x^n + 1$, which has a root α (1 < n < m/2), giving $\alpha^m + \alpha^n + 1 = 0$. Therefore,

$$\begin{aligned}
\alpha^m &= 1 + \alpha^n \\
\alpha^{m+1} &= \alpha + \alpha^{n+1} \\
&\vdots \\
\alpha^{2m-3} &= \alpha^{m-3} + \alpha^{m+n-3} \\
\alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2}
\end{aligned} \tag{2.4}$$

For example, consider the irreducible trinomial $x^{233} + x^{74} + 1$ [National Institute of Standards and Technology (1994)]. The multiplication or squaring of the elements in *GF(2^{233})* would result in maximum of 465 bits. These 465 bits can be reduced to 233 bits as shown in Fig. 2.3 using Equation 2.4.

2.2 Elliptic Curves

Definition 2.2.1 The simplified form of the *Weierstrass* equation yields an elliptic curve (in *GF(2^m)*). The Weierstrass equation in its simplest form is :

$$y^2 + xy = x^3 + ax^2 + b \tag{2.5}$$

with the coefficients a and b in *GF(2^m)* and if $b \neq 0$ it is known as non-singular curve.

Elements of Elliptic curve doesn't form a group as such, unless, we introduce one more element known as *point at infinity* ($\mathcal{O}$). It also acts as an identity element of the group. Given below are the operations that can be performed on this group.

Point Inversion : The inverse of a point P(x_1, y_1) is found as shown in Fig. 2.4. The coordinates of $-$P are ($x_1, x_1 + y_1$).

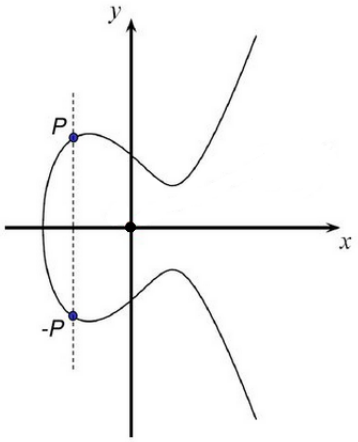

Fig. 2.4: Inverse of an elliptic curve point.

Rebeiro (2009)

Point Doubling : To compute the double of a point P(x_1, y_1) on an elliptic curve, a tangent to the curve is drawn passing through this point. The tangent would intersect the curve at another point which is $-$2P as shown in Fig. 2.5 taking the inverse of this point would give us the required result. The equations for 2P(x_3, y_3) are as follows:

$$x_3 = \lambda^2 + \lambda + a = {x_1}^2 + \frac{b}{{x_1}^2}$$
$$y_3 = {x_1}^2 + \lambda x_3 + x_3 \tag{2.6}$$

where $\lambda = x_1 + (y_1/x_1)$.

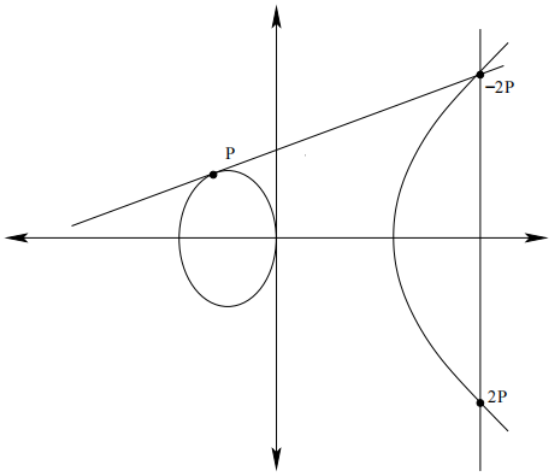

Fig. 2.5: Double of an elliptic curve point

Rebeiro (2009)

Point Addition : The addition of points on elliptic curve say P(x_1, y_1) and Q(x_2, y_2) is found out as shown in Fig. 2.6. A line is drawn passing through these points, intersecting the curve at a third point, which is the inverse of (P + Q). Taking the inverse of this point would give us the result. Suppose R(x_3, y_3) = (P + Q), then

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \tag{2.7}$$

where $\lambda = (y_1 + y_2)/(x_1 + x_2)$. For P = −Q, the addition would result in $\mathcal{O}$.

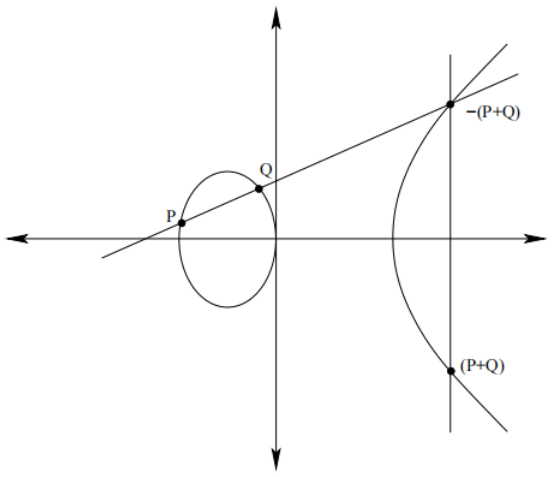

Fig. 2.6: Addition of elliptic curve points

Rebeiro (2009)

Scalar/Point Multiplication : Point multiplication of a point(P) on elliptic curve with a scalar(k) is nothing but adding P with itself $k-1$ times. Since this would take a lot of time to compute, many algorithms are suggested in literature. Among these we are implementing *Left to Right* or *MSB first* algorithm, given in Algorithm 2.1.

Algorithm 2.1 : Left to Right algorithm

Input : Basepoint P = (p_x, p_y) and Scalar k = $(k_{m-1}, k_{m-2}, ..., k_0)_2$, where $k_{m-1} = 1$

. **Output :** Point on the curve Q + kP

```
1 Q = P
2 for i = m-2 to 0 do
3     Q = 2 . Q
4     if k_i = 1 then
5         Q = Q + P
6     end
7 end
```

Mukhopadhyay and Chakraborty (2014)

The Table 2.1 illustrates Algorithm 2.1, considering *k* = 25 or $(11001)_2$.

Table 2.1: Left to right algorithm illustration

i	k_i	Operation	Q
3	1	Double and Add	3P
2	1	Double only	6P
1	1	Double only	12P
0	0	Double and Add	**25P**

For every iteration *i* doubling operation takes place, while addition is carried out only when k_i = 1. From Equations 2.6 and 2.7 we can see that doubling as well as addition require 2 multiplications along with 1 inversion. The number of additions is directly proportional to the number of 1's present in the scalar key, which would also mean the number of inversions going up.

2.2.1 Projective Coordinate Representation

Finite field inversion is the most complicated operation. So, there is need to reduce these operations to the extent possible. The equations discussed till now in section 2.2 dealt with only two point coordinate system (x, y) also known as *Affine* coordinates. By, introducing a third point in the coordinate system i.e (X, Y, Z) there is a scope to reduce the number of Inversions. This coordinate system is known as *Projective* representation. In general, x and y are replaced with $\frac{X}{Z^c}$ and $\frac{Y}{Z^d}$ respectively. Various representations are suggested in literature to make one to one correspondence from affine to projective representations by changing the values of *c* and *d*. In this work, c = 1 and d = 2 and the resulting points are known as *López-Dahab(LD)* coordinates[Menezes *et al.* (1996)].

Equations 2.5, 2.6 and 2.7 are modified accordingly ($x \rightarrow \frac{X}{Z^c}, y \rightarrow \frac{Y}{Z^d}$) to get Equation 2.8, 2.9 and 2.10.

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^4 \tag{2.8}$$

For *Point Inversion*, with P(X_1, Y_1, Z_1), the inversion point $-$P has coordinates ($X_1, X_1Z_1 + Y_1, Z_1$). In LD, the point at infinity, $\mathcal{O}$ is represented as (1, 0, 0).
When the point P in LD is doubled, the result is the point 2P with coordinates (X_3, Y_3, Z_3), given by the equations below. The number of multiplications have gone up by 3 whereas the inversions have reduced to zero.

$$\begin{aligned} Z_3 &= {X_1}^2.{Z_1}^2 \\ X_3 &= {X_1}^4 + b.{Z_1}^4 \\ Y_3 &= b.{Z_1}^4.Z_3 + X_3.(a.Z_3 + {Y_1}^2 + b.{Z_1}^4) \end{aligned} \tag{2.9}$$

The equations for *Addition* involving one point in affine, Q(x_2, y_2) and another in LD coordinate systems, P(X_1, Y_1, Z_1), P + Q = (X_3, Y_3, Z_3) (with Q $\neq \pm$P) are given as follows:

$$\begin{aligned} A &= y_2.{Z_1}^2 + Y_1 \\ B &= x_2.Z_1 + X_1 \\ C &= Z_1.B \\ D &= B^2.(C + a.{Z_1}^2) \\ Z_3 &= C^2 \\ E &= A.C \\ X_3 &= A^2 + D + E \\ F &= X_3 + x_2.Z_3 \\ G &= (x_2 + y_2).{Z_3}^2 \\ Y_3 &= (E + Z_3).F + G \end{aligned} \tag{2.10}$$

The number of multiplications have gone up by 7 whereas the inversions have reduced to zero.
The advantage of using LD coordinates is that, no intermediate inversions are required other than one inversion at the end to convert the LD to affine coordinate representation.

CHAPTER 3

Finite Field Multiplier

We have increased the number of finite field multiplications by opting LD coordinate system. So, we have to choose the multiplier which is efficient, as the multiplier has longest latency and it decides the operating frequency. Finite field multiplication essentially involves two steps. Firstly, multiplying two elements of field which results in output not belonging to *GF(2^m)* . Secondly, the result produced is reduced using an irreducible polynomial to get the output in *GF(2^m)*.
Consider C(x), D(x) and P(x) $\in$ *GF(2^m)*, where P(x) is an irreducible polynomial. Let M′(x) = C(x) . D(x). The result M′(x) $\notin$ *GF(2^m)*, hence it is reduced as shown below.

$$\begin{aligned} M(x) &= M'(x) \textit{mod P(x)} \\ &= C(x).D(x) \textit{mod P(x)} \end{aligned} \tag{3.1}$$

After going through a lot of literature work, Karatsuba Multiplier was found to be efficient for high performance applications as it has a sub-quadratic space complexity (O($m^{log_2 3}$), m being the number of operand bits). The same was implemented in Rebeiro (2009)

3.1 Karatsuba Multiplication

This multiplier performs small bit size multiplications by splitting the operands and finally combining the partial results appropriately. The *m* bit multiplicands C(x) and D(x) are split into two *m/2* bits as shown in Equation 3.2. C_h, C_l, D_h and D_l are of *m/2* bits each.

$$\begin{aligned} C(x) &= C_h x^{m/2} + C_l \\ D(x) &= D_h x^{m/2} + D_l \end{aligned} \tag{3.2}$$

As we can see from Equation 3.3, we don't need *m* bit multiplication to get the result rather using three *m/2* multiplications would get the job done.

$$
\begin{aligned}
M'(x) &= (C_h x^{m/2} + C_l)(D_h x^{m/2} + D_l) \\
&= C_h D_h x^m + (C_h D_l + C_l D_h) x^{m/2} + C_l b_l \\
&= C_h D_h x^m \\
&+ ((C_h + C_l)(D_h + D_l) + C_h D_h + C_l D_l) x^{m/2} \\
&+ C_l D_l
\end{aligned}
\tag{3.3}
$$

The generated partial products are then combined to get the final result as shown below in the Fig. 3.1.

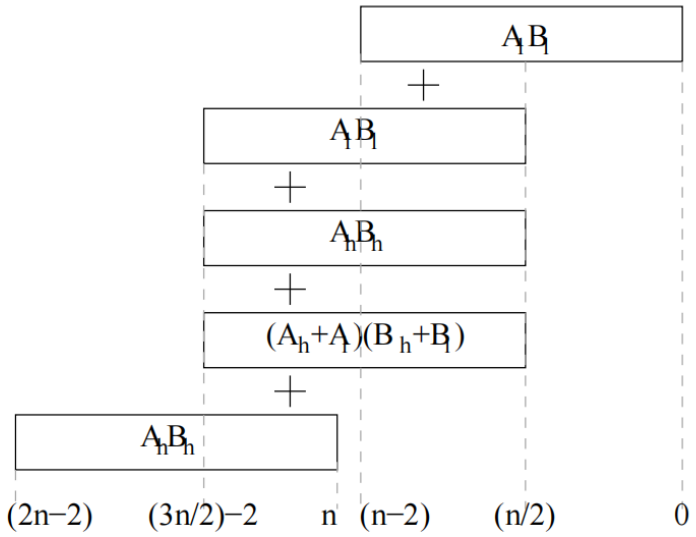

Fig. 3.1: Computing M ′(x)

Rebeiro (2009)

This M ′(x) is further reduced by irreducible polynomial as given in Section 2.1. This splitting of multiplicands can be further carried out with *m/4* bits each. Then we would require a total of nine *m/4* multiplications. This recursive application of Karatsuba multiplication can be applied till the multiplicands are of 2 bits. Then the final recursion can be achieved using *AND* gates. Hence, *m* being a power of 2 would be best suited for Karatsuba multiplication and is known as *Basic Karatsuba Multiplier*.

3.2 Karatsuba Multipliers in ECC

Since the fields used in ECC have a prime degree, the Basic Karatsuba cannot be applied for finite field multiplication. There are two ways to use this multiplication technique for our ECCP:

- Sequential circuit approach, where the output is fed back into the same circuit at every clock cycle. This approach requires less hardware but at the expense of consuming more clock cycles.
- Combinational circuit approach, which can produce the output in one clock cycle at the expense of large area.

Second approach is implemented in this work as we are interested in making a performance optimised ECC processor.

The *Simple Karatsuba multiplier*[Weimerskirch and Paar (2006)] is similar to basic Karatsuba multiplier. The *m* bit multiplicands are divided as shown in Equation 3.1. Hence, we need three *m/2* bit multiplications to generate partial products.

General Karatsuba multiplier[Weimerskirch and Paar (2006)], does follow recursion but instead of splitting the multiplicands into two terms it does it in more than two.

The proposed multiplier in [Rebeiro (2009)] is a combination of the above two variants, which I too have implemented. This combined multiplier is known as *Hybrid Karatsuba Multiplier*. For, multiplicands with bits sizes < 29, the general Karatsuba algorithm is applied as it ensures maximum LUT's utilisation. And for bit sizes greater than 29, simple Karatsuba is invoked as it makes sure least gate count for higher bi multiplications. The recursion for 233 bit Hybrid Karatsuba multiplier is as shown below in Fig. 3.2.

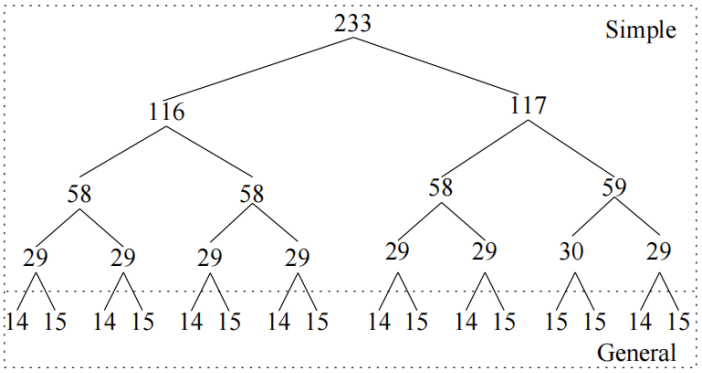

Fig. 3.2: Hybrid Karatsuba Multiplier for 233 bits

Rebeiro (2009)

Algorithms for implementing general Karatsuba and Hybrid Karatsuba are given below in Algorithm 3.1 ad 3.2 respectively.

Algorithm 3.1: gkmul (General Karatsuba Multiplier)

Input: A and B $\Longrightarrow$ m bits

. **Output:** C $\Longrightarrow$ (2m $-$ 1) bits

/* **Define :** $M_x \rightarrow A_x B_x$ */

/* **Define :** $M_{(x,y)} \rightarrow (A_x + A_y)(B_x + B_y)$ */

1 **begin**

2 **for** i = 0 to m $-$ 2 **do**

3 $C_i = C_{2m-2-i} = 0$

4 **for** j = 0 to [i/2] **do**

5 **if** i = 2 j **then**

6 $C_i = C_i + M_j$

7 $C_{2m-2-i} = C_{2m-2-i} + M_{m-1-j}$

8 **else**

9 $C_i = C_i + M_j + M_{i-j} + M_{j,i-j}$

10 $C_{2m-2-i} = C_{2m-2-i} + M_{m-1-j} + M_{m-1-i+j} + M_{m-1-j,m-1-i+j}$

11 **end**

12 **end**

13 **end**

14 $C_{m-1} = 0$

15 **for** j = 0 to [(m-1)/2] **do**

16 **if** m-1 = 2 j **then**

17 $C_{m-1} = C_{m-1} + M_j$

18 **else**

19 $C_{m-1} = C_{m-1} + M_j + M_{m-1-j} + M_{(j,m-1-j)}$

20 **end**

21 **end**

22 **end**

Mukhopadhyay and Chakraborty (2014)

Algorithm 3.2: hmul (Hybrid Karatsuba Multiplier)

Input: A and B $\Longrightarrow$ m bits

. **Output:** C $\Longrightarrow$ (2m $-$ 1) bits

1 **begin**

2 **if** m < 29 **then**

3 **return** gkmul (A,B,m)

4 **else**

5 l = [m/2]

6 A $'$ = $A_{[m-1,...,l]}$ + $A_{[l-1,...,0]}$

7 B $'$ = $B_{[m-1,...,l]}$ + $B_{[l-1,...,0]}$

8 C_{p1} = hmul ($A_{[l-1,...,0]}$,$B_{[l-1,...,0]}$,l)

9 C_{p2} = hmul (A $'$,B $'$,l)

10 C_{p3} = hmul ($A_{[m-1,...,l]}$,$B_{[m-1,...,l]}$,m-l)

11 **return** (C_{p3} < < 2l) + (C_{p1} + C_{p2} + C_{p3}) < < l + C_{p1}

. /* < < indicates left shift */

12 **end**

13 **end**

Mukhopadhyay and Chakraborty (2014)

CHAPTER 4

Finite Field Inversion

In *GF(2^m)*, the inverse of any non zero element *a* is such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$, with $a^{-1} \in$ *GF(2^m)*. In spite of reducing the number of inversions by introducing LD coordinate system, we cannot completely eliminate field inversion operation. Therefore, it is necessary to design an efficient multiplicative inverse technique. *Extended Euclidean algorithms* (EEA) and the *Itoh-Tsujii* (ITA) are the most widely used algorithms to compute multiplicative inverse. In this work, Itoh-Tsujii algorithm is implemented as it is faster compared to EEA, but at the expense of using multiplier which consumes large area . Since we require finite field multiplier in our ECCP, we don't need extra multiplier for computing inversion separately. The same multiplier unit can be used to compute inversion. Hence, ITA is the best choice.

4.1 Itoh-Tsujii Algorithm

According to Fermat's little theorem, the inverse of an element $a \in GF(2^m)$ can be determined as given in Equation 4.1. This forms the basis for ITA.

$$a^{-1} = a^{2m-2} \tag{4.1}$$

Implementing a^{-1} using the rudimentary method necessitates $(m-1)$ squares and $(m-2)$ multiplications. The number of multiplications need to be reduced as it has the longer latency. This can be achieved by using *addition chains*. A sequence of integers ($n \in N$) can form an *addition chain* if the following properties are satisfied.

U = (u_0 u_1 u_2 . . . u_r).

- $u_o = 1$
- $u_r = n$
- $u_i = u_j + u_k$, where $k \leq j < i$

If j = i $-$ 1 in above condition then the addition chains are known as *Brauer chains*. There are many ways to form an addition chain for given *n* but the one with minimum length is said to be optimal.

Reusing the notations from [Rodríguez-Henríquez *et al.* (2007)] for $\beta_k(a)$, Equation 4.1 can be expressed as below.

$$a^{-1} = (a^{2^{m-1}-1})^2$$

$$\beta_k(a) = a^{2^k-1} \in GF(2^m), k \in \mathbb{N}$$

$$a^{-1} = [\beta_{m-1}(a)]^2$$

$\beta_{k+j}(a) \in$ *GF(2^m)* can be resolved as shown in Equation 4.2. This property along with addition chain is used in recursive manner to compute multiplicative inverse. (Note: $\beta_k(a)$ is denoted as β_k)

$$\beta_{k+j}(a) = (\beta_j)^{2^k}\beta_k = (\beta_k)^{2^j}\beta_j \tag{4.2}$$

For $a \in GF(2^{233})$, the inverse is obtained by squaring $\beta_{232}(a)$, where β_{232} (a) = $a^{2^{232}-1}$. A Brauer chain for 232 is as considered below.

$$U_1 = (\ 1\ \ 2\ \ 3\ \ 6\ \ 7\ \ 14\ \ 28\ \ 29\ \ 58\ \ 116\ 232\ ) \tag{4.3}$$

$\beta_{232}(a)$ computation is shown in Table 4.1. It requires a total of 231 squarings and 10 multiplications.

In general for *GF(2^m)*, with addition chain of length *l*, we need a total of m$-$1 squarings and l$-$1 multiplications. The addition chain length,l is related to m as l $\leq$ [log_2m]. As a result, the ITA requires a substantially smaller number of multiplications than the conventional technique.

Table 4.1: Generic ITA for $GF(2^{233})$

	β_{u_i} (a)	$\beta_{u_j+u_k}$ (a)	Exponentiation
1	β_1(a)		a
2	β_2(a)	$\beta_{1+1}(a)$	$(\beta_1)^{2^1}\beta_1 = a^{2^2-1}$
3	β_3(a)	$\beta_{2+1}(a)$	$(\beta_2)^{2^1}\beta_1 = a^{2^3-1}$
4	β_6(a)	$\beta_{3+3}(a)$	$(\beta_3)^{2^3}\beta_3 = a^{2^6-1}$
5	β_7(a)	$\beta_{6+1}(a)$	$(\beta_6)^{2^1}\beta_1 = a^{2^7-1}$
6	β_{14}(a)	$\beta_{7+7}(a)$	$(\beta_7)^{2^7}\beta_7 = a^{2^{14}-1}$
7	β_{28}(a)	$\beta_{14+14}(a)$	$(\beta_{14})^{2^{14}}\beta_{14} = a^{2^{28}-1}$
8	β_{29}(a)	$\beta_{28+1}(a)$	$(\beta_{28})^{2^1}\beta_1 = a^{2^{29}-1}$
9	β_{58}(a)	$\beta_{29+29}(a)$	$(\beta_{29})^{2^{29}}\beta_{29} = a^{2^{58}-1}$
10	β_{116}(a)	$\beta_{58+58}(a)$	$(\beta_{58})^{2^{58}}\beta_{58} = a^{2^{116}-1}$
11	β_{232}(a)	$\beta_{116+116}(a)$	$(\beta_{116})^{2^{116}}\beta_{116} = a^{2^{232}-1}$

4.2 Quad Itoh-Tsujii Algorithm

The generic ITA discussed earlier uses squarer circuits for computation, but we can also use any 2^n circuit in that place. In general, for any a $\in$ *GF(*2^m*)*, k $\in \mathbb{N}$, we define

$$\alpha_k(a) = a^{2^{nk}-1}$$

The following properties are used to compute the inverse.

- For any element a $\in$ GF(2^m), $\alpha_{k_1}(a) = a^{2^{nk_1}-1}$ and $\alpha_{k_2}(a) = a^{2^{nk_2}-1}$ then

  $$\alpha_{k_1+k_2}(a) = (\alpha_{k_1}(a))^{2^{nk_2}}\ \alpha_{k_2}(a)$$

  where k_1, k_2 and $n \in \mathbb{N}$

- For any element a $\in$ GF(2^m), its inverse is given by

  $$a^{-1} = \begin{cases} [\alpha_{\frac{m-1}{n}}(a)]^2 & \text{when} \quad n|(m-1) \\ [(\alpha_q(a))^{2^r}\beta_r(a)]^2 & \text{when} \quad n \nmid (m-1) \end{cases}$$

  where $nq + r = m - 1$ and n, q and r $\in \mathbb{N}$.

In Quad ITA, we set $n = 2$, such that $\alpha_k(a) = a^{4^k-1}$. Now to compute the inverse of any element a $\in$ *GF(*2^{233}*)*, we need to compute $[\alpha_{\frac{233-1}{2}}(a)]^2 = [\alpha_{116}(a)]^2$ (by property mentioned above).

The computation of $[\alpha_{116}(a)]$ is given in Table 4.2. Squaring this would result in the final inverse. The algorithm to compute this is also given below in Algorithm 4.1.

Table 4.2: Quad-Itoh Tsujii for $GF(2^{233})$

	α_{ui} (a)	$\alpha_{u_j+u_k}$ (a)	Exponentiation
1	α_1(a)		a^3
2	α_2(a)	$\alpha_{1+1}(a)$	$(\alpha_1)^{4^1}\alpha_1 = a^{4^2-1}$
3	α_3(a)	$\alpha_{2+1}(a)$	$(\alpha_2)^{4^1}\alpha_1 = a^{4^3-1}$
4	α_6(a)	$\alpha_{3+3}(a)$	$(\alpha_3)^{4^3}\alpha_3 = a^{4^6-1}$
5	α_7(a)	$\alpha_{6+1}(a)$	$(\alpha_6)^{4^1}\alpha_1 = a^{4^7-1}$
6	α_{14}(a)	$\alpha_{7+7}(a)$	$(\alpha_7)^{4^7}\alpha_7 = a^{4^{14}-1}$
7	α_{28}(a)	$\alpha_{14+14}(a)$	$(\alpha_{14})^{4^{14}}\ \alpha_{14}$ = $a^{4^{28}-1}$
8	α_{29}(a)	$\alpha_{28+1}(a)$	$(\alpha_{28})^{4^1}\alpha_1 = a^{4^{29}-1}$
9	α_{58}(a)	$\alpha_{29+29}(a)$	$(\alpha_{29})^{4^{29}}\alpha_{29} = a^{4^{58}-1}$
10	α_{116}(a)	$\alpha_{58+58}(a)$	$(\alpha_{58})^{4^{58}}\alpha_{58} = a^{4^{116}-1}$

In general, for *l* length addition chain Quad-ITA would require $(l-1)$ multiplications and 2 multiplications for precomputation. Therefore, a total of $l+1$ multiplications and $[\frac{m-1}{2} - 1]$ quad operations are required to compute inverse.

Algorithm 4.1: qitmia *(Quad-ITA)*

Input: The element a $\in$ $GF(2^m)$ and the Brauer chain

U = {1 , 2 , . . . , $\frac{m-1}{2}$, m-1 }

Output: The multiplicative inverse a^{-1}

begin

l = length (U)

a^2 = hmul(a , a); /* hmul : Hybrid Karatsuba multiplier */

$\alpha_{u_1} = a^3 = a^2$. a

foreach $u_i \in$ U ($2 \leq$ i $\leq$ l-1) **do**

p = u_{i-1}

q = u_i - u_{i-1}

α_{u_i} = hmul ($\alpha_p^{4^q}$,α_q)

end

α^{-1} = hmul ($\alpha_{u_{l-1}}$, $\alpha_{u_{l-1}}$)

end

Mukhopadhyay and Chakraborty (2014)

4.3 Quad Block

The Hardware architecture for Quad block is as shown in Fig. 4.1. The quad circuits are cascaded and the individual quad circuit outputs are connected to the multiplexer inputs. The *qsel* decides which of the 14 powers gets selected as output. The number of cascades id decided such that the delay of all the cascades can be at max equal to that of Karatsuba multiplier.

Let $(Delay)_q$ be the delay of one quad circuit and let us consider u_s number of such cascades. Let $(Delay)_m$ be the multiplier delay. Then,

$$(Delay)_q.u_s \leq (Delay)_m$$

.

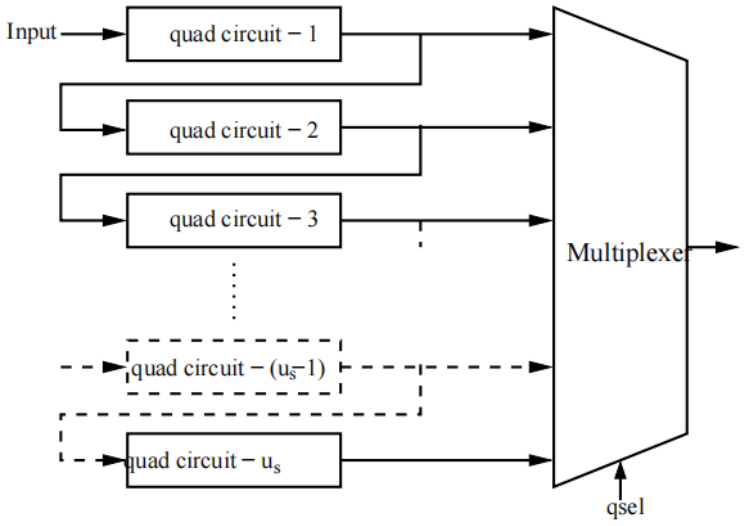

Fig. 4.1: Quad block

Rebeiro (2009)

Each quad circuit is as simpler as squarer circuit discussed before. Here, the input to is spread out by inserting three zeros in between adjacent bits which would result in the output $\notin GF(2^{233})$.

Therefore, to get the result in $GF(2^{233})$ modular reduction needs to be performed on the result obtained after inserting zeros.

CHAPTER 5

Elliptic Curve Crypto Processor

The basis for any cryptography application involving elliptic curves is the *Scalar/-Point multiplication*. So, the main purpose of this work is to design a hardware which computes that. The elliptic curve is chosen from the National Institute of Standards and Technology (1994) standard curves over $GF(2^{233})$.

$$y^2 + xy = x^3 + ax^2 + b \tag{5.1}$$

The same can be represented in LD coordinate system as follows.

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^4 \tag{5.2}$$

The Algorithm 2.1 discussed earlier is implemented in this Processor using the primitive operations such as point doubling, point addition.

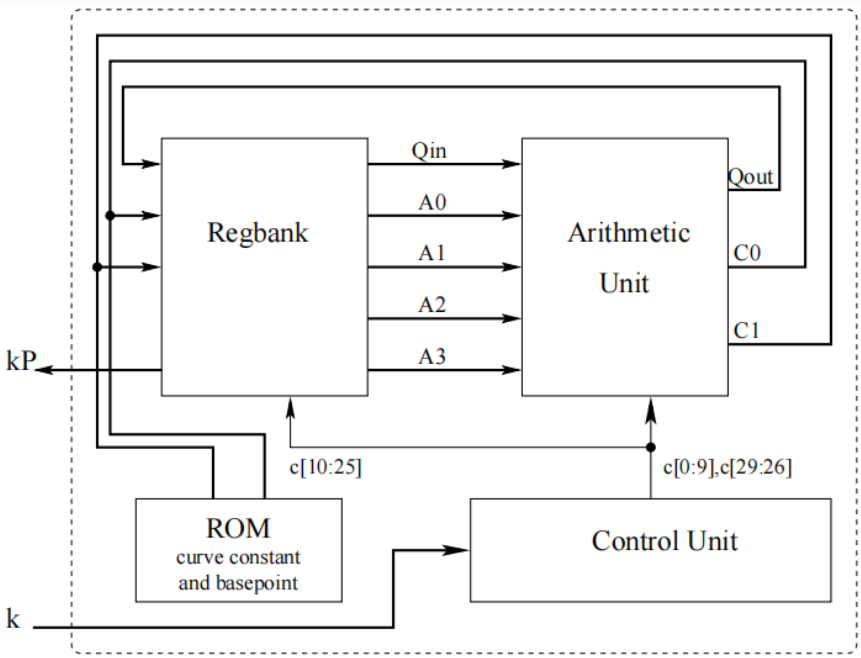

Fig. 5.1: Elliptic Curve Crypto Processor

Rebeiro (2009)

The block diagram of the processor is shown in Fig. 5.1. It takes key, *k* and base-point, *P* as the inputs and produces *kP* as the output after performing scalar multiplication. The elliptic curve constants such as x and y coordinates of *basepoint*, P and constant, *b* are stored in ROM. At every clock cycle, control signals are generated which determine the following:

- Data that should be fed to Arithmetic unit (AU).
- Computation that should be performed by the AU on the data it received.
- AU results to be stored in which registers.

5.1 Blocks of Cryptoprocessor

5.1.1 Register Bank

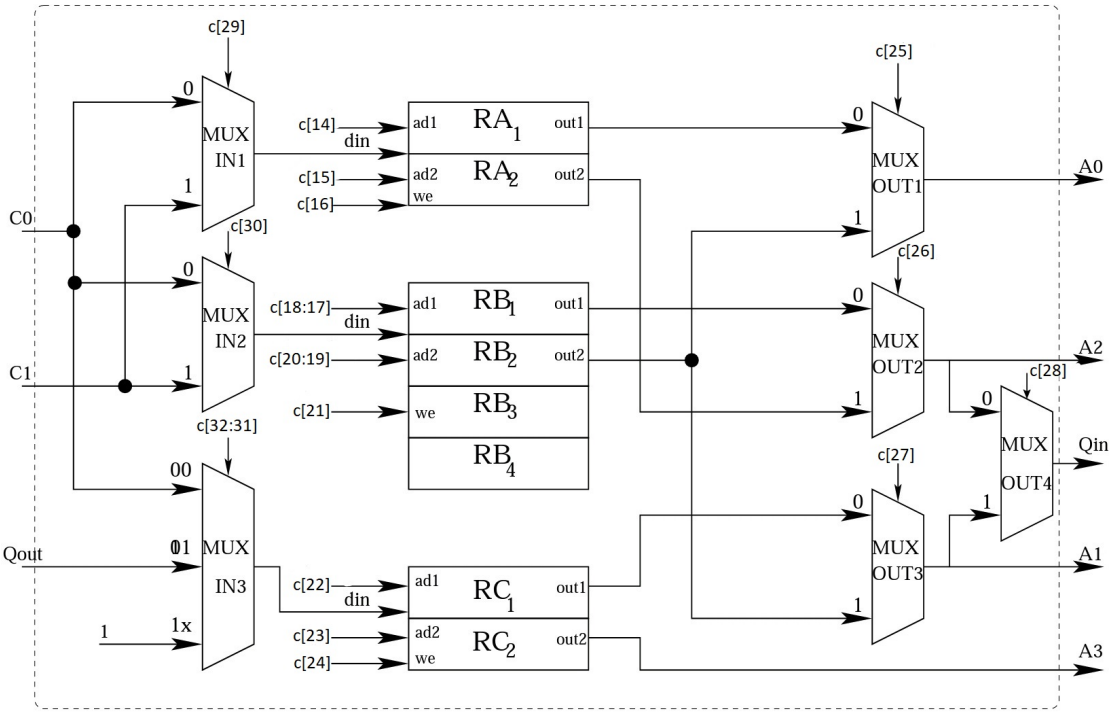

Fig. 5.2: Register File for Elliptic Curve Crypto Processor

The register bank can be divided into three banks, bank A, bank B, and bank C as shown in above Fig. 5.2. Since, this processor is designed to implement ECC algorithm for

233 bits, all the eight registers are of 233 bits size.

Reading: Each register is a dual ported which can facilitate *asynchronous* read. The values on *out1* and *out2* correspond to the addresses on *ad1* and *ad2*.

Writing: A value *din* can be written into register only if *we* signal is high. The address line *ad1* decides which register gets written.

At every clock cycle, this register module gives out five values A0, A1, A2, A3, Qin depending on the select lines of out multiplexers. It also takes three inputs from Arithmetic unit which are then passed on for updation of registers based on select lines of input multiplexers.

Table 5.1 describes the role of each register.

Table 5.1: Role of Registers

Register	Description
RA_1	1. Initially to store P_x. 2. Stores the x coordinate of the result. 3. Intermediate result storage.
RA_2	Stores P_x
RB_1	1. Initially to store P_y. 2. Stores the y coordinate of the result. 3. Intermediate result storage.
RB_2	Stores P_y
RB_3	Intermediate result storage.
RB_4	Stores the curve constant b.
RC_1	1. Initialised to 1. 2. Stores z coordinate of the projective result. 3. Intermediate result storage.
RC_2	Intermediate result storage.

5.1.2 Arithmetic Unit

It is capable of performing finite field operations discussed in previous chapters. It comprises of the following blocks:

- Squarer
- Adder: which performs bitwise XOR operation
- Hybrid Karatsuba multiplier
- Quad block: comprising of 14 cascade quad circuits as shown in Fig. 4.1.

The quad block is used only during the final conversion of result. The AU takes five inputs from the register bank and produce three results namely, C0, C1 and Qout.
Fig 5.3 shows the AU structure. Select lines of mux A and B, decide which data should be fed into Karatsuba multiplier. And the select lines of mux C and D, decide which data is output.

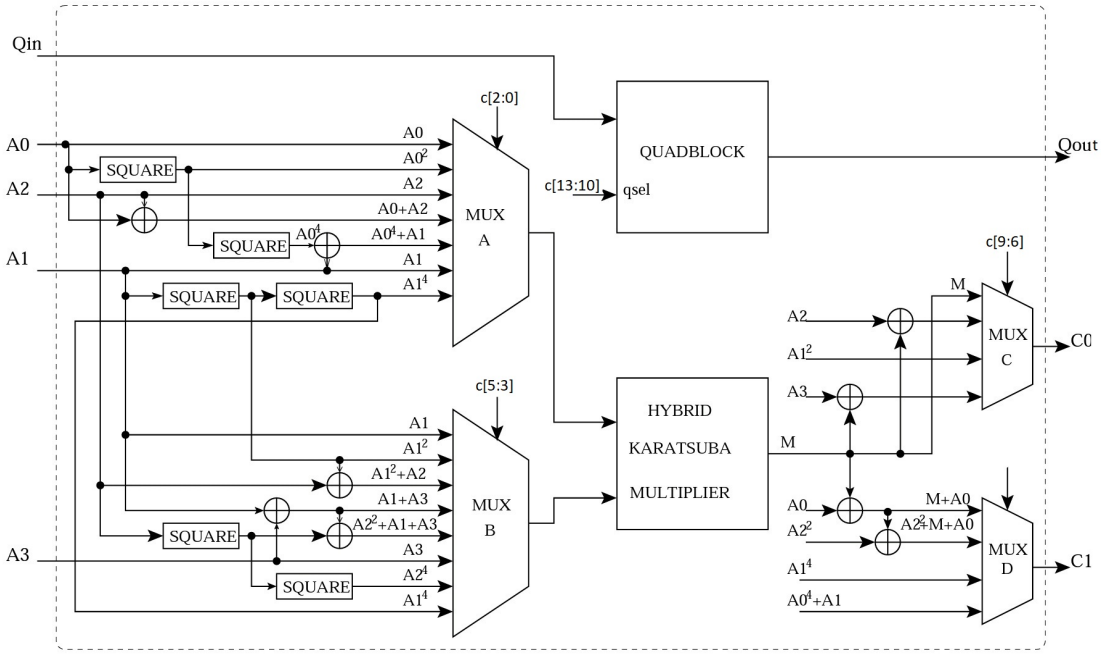

Fig. 5.3: Arithmetic Unit

5.1.3 Control Unit

The control unit generates 33 bits control word(c) every clock cycle depending on the operation to be performed. The data flow is controlled by these control words, which also determine the operations that are done on the data. The Arithmetic unit is controlled by bits c[13:0] and the remaining bits control the reading and writing of register bank.

5.2 Implementing Point Arithmetic

5.2.1 Point Doubling

As mentioned in chapter 2, the doubling operation is performed in projective representation. The Equation 5.3 are implemented here. We need a total of five multiplications but since the value of *a* is 1 in our chosen elliptic curve, we need only four multiplications. As we have only one multiplier in our design and to ensure its high utilisation efficiency, we need to schedule a multiplication operation every clock cycle. Hence, we require atleast four clock cycles to finish the doubling operation. The Algorithm 5.1 depicts this implementation.

$$\begin{aligned} Z_3 &= X_1^2.Z_1^2 \\ X_3 &= X_1^4 + b.Z_1^4 \\ Y_3 &= b.Z_1^4.Z_3 + X_3.(a.Z_3 + Y_1^2 + b.Z_1^4) \end{aligned} \tag{5.3}$$

Algorithm 5.1: Implementation of Point Doubling

Input: LD Point P(X_1 , Y_1 , Z_1) $\Longrightarrow$ (RA_1 , RB_1 , RC_1)

. The curve constant b $\Longrightarrow$ RB_4.

. **Output:** LD Point 2P(X_3 , Y_3 , Z_3) $\Longrightarrow$ (RA_1 , RB_1 , RC_1).

1 RB_3 = RB_3 . RC_1^4

2 RC_1 = RA_1^2 . RC_1^2

3 RA_1 = RA_1^4 . RB_3

4 RB_1 = RB_3 . RC_1 + RA_1 . (RC_1 + RB_1^2 + RB_3)

The operations mentioned in above algorithm can be parallelised as shown in Table 5.2. Since, AU is capable of giving out three outputs, one of these can give out multiplication result and the others can give out quad or squarer or adder results.

Table 5.2: Scheduling of Point Doubling operations

Clock	Operation 1 (C0)	Operation 2 (C1)
1	$RC_1 = RA_1^2.RC_1^2$	$RB_3 = RC_1^4$
2	$RB_3 = RB_3.RB_4$	
3	$RC_2 = (RA_1^4 + RB_3).(RC_1 + RB_1^2 + RB_3)$	$RA_1 = (RA_1^4 + RB_3)$
4	$RB_1 = RB_3.RC_1 + RC_2$	

5.2.2 Point Addition

The following Equation 5.4 are implemented in ECCP. The equations involve point addition of two points, among which one is in affine coordinate and the other in projective coordinate system.

$$
\begin{aligned}
A &= y_2.{Z_1}^2 + Y_1 \\
B &= x_2.Z_1 + X_1 \\
C &= Z_1.B \\
D &= B^2.(C + a.{Z_1}^2) \\
Z_3 &= C^2 \\
E &= A.C \\
X_3 &= A^2 + D + E \\
F &= X_3 + x_2.Z_3 \\
G &= (x_2 + y_2).{Z_3}^2 \\
Y_3 &= (E + Z_3).F + G
\end{aligned}
\tag{5.4}
$$

Algorithm 5.2: Implementation of Point Addition

Input: LD point P$(X_1, Y_1, Z_1) \Longrightarrow (RA_1, RB_1, RC_1)$.

Affine point Q$(x_2, y_2) \Longrightarrow (RA_2, RB_2)$.

Output: LD Point P+Q=$(X_3, Y_3, Z_3) \Longrightarrow (RA_1, RB_1, RC_1)$.

1 $RB_1 = RB_2.{RC_1}^2 + RB_1$

2 $RA_1 = RA_2.RC_1 + RA_1$

3 $RB_3 = RC_1.RA_1$

4 $RA_1 = {RA_1}^2.(RB_3 + {RC_1}^2)$

5 $RC_1 = {RB_3}^2$

6 $RC_2 = RB_1.RB_3$

7 $RA_1 = {RB_1}^2 + RA_1 + RC_2$

8 $RB_3 = RA_1 + RA_2.{RC_1}^2$

9 $RB_1 = (RA_2 + RB_2).{RC_1}^2$

10 $RB_1 = (RC_2 + RC_1).RB_3 + RB_1$

The Algorithm 5.2 depicts the implementation of point addition. The total number of multiplications involved according to Equation 5.4 are nine, but since *a* is 1 we need only eight multiplication operations. And these are scheduled in eight clock cycles. Similar to point doubling, operations in addition algorithm can also be parallelised as shown in Table 5.3.

Table 5.3: Parallel LD Point Additionon the ECCP

Clock	Operation 1 (C0)	Operation 2 (C1)
1	$RB_1 = RB_2.{RC_1}^2 + RB_1$	-
2	$RA_1 = RA_2.RC_1 + RA_1$	-
3	$RB_3 = RC_1.RA_1$	-
4	$RA_1 = {RA_1}^2.(RB_3 + {RC_1}^2)$	-
5	$RC_2 = RB_1.RB_3$	$RA_1 = {RB_1}^2 + RA_1 + RB_1.RB_3$
6	$RC_1 = {RB_3}^2$	$RB_3 = RA_1 + RA_2.{RB_3}^2$
7	$RB_1 = (RA_2 + RB_2).{RC_1}^2$	-
8	$RB_1 = (RC_2 + RC_1).RB_3 + RB_1$	-

5.3 Finite State Machine

The ECCP involves the following steps:

- *Initialisation*: In this state, the curve constants such as x & y coordinates of base point and constant *b* are stored in registers. This operation takes three clock cycles depicted by three different states. Also, the position of leading '1' in the scalar key is found out.
- *Point Double*: We need four clock cycles for this operation, which are shown as four states in Fig. 5.4.
- *Point Addition*: This operation require eight clock cycles for computation and hence shown using eight different states.
- *Inversion*: Final scalar multiplication result is obtained in LD coordinates and to get back the result in affine coordinate we need this operation. This involves 24 clock cycles in total.

Once the initialisation is done, the scalar multiplication algorithm starts with state D1. Once the state D4 is reached, the current bit of key, k_i is checked.
If its 1 then the next state would be A1 else D1. This flow is continued until the last bit of key is reached. If last bit is 0 then after D4 the next state would be inversion I1. If the last bit is 1 after completing the addition the next state would be I1.

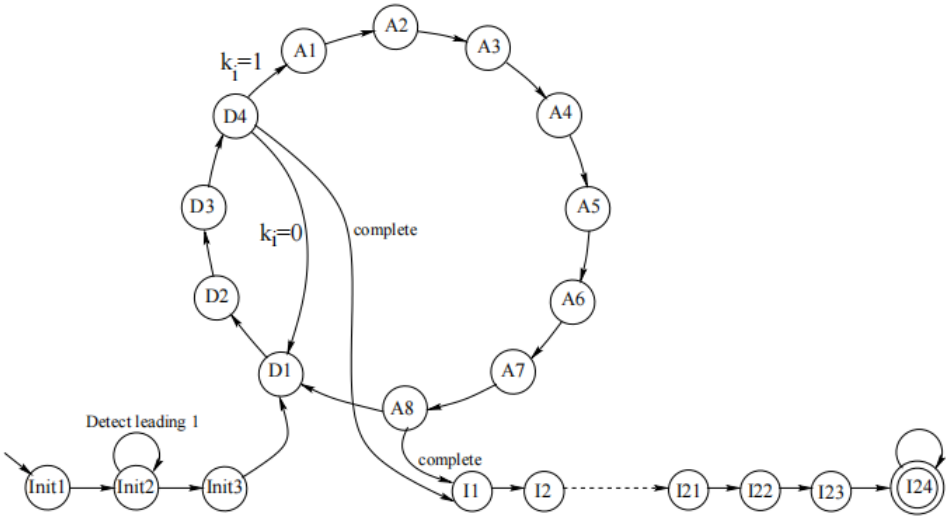

Fig. 5.4: FSM for ECCP

Rebeiro (2009)

Consider the following example to know how to compute the number of clock cycles required.

Let scalar key, k = 8'b00101001.

The leading '1' is found at position $i = 5$. Hence the algorithm starts from $i = 4$. There are five bits after the leading one. And for each bit doubling is performed. There are only two 1's so the addition states are traversed twice. 3 clock cycles for initialisation and 24 clock cycles at the end for inversion.

$$\begin{aligned} ClockCycles &= 3 + 4(5) + 8(2) + 24 \\ &= 63 \end{aligned} \tag{5.5}$$

The register module outputs and inputs at each state of FSM are given in Table 5.4. The corresponding control words generated for these states are given in Table 5.5.

Table 5.4: Register Module Input and Outputs,[Mukhopadhyay and Chakraborty (2014)]

State	A0	A1	A2	A3	Qin	Register Module Inputs
Init1	-	-	-	-	-	**C0**:RA_1=P_x;**C1**:RB_1=P_y;$RC_1 = 1$
Init2	-	-	-	-	-	**C0**:RA_2=P_x;**C1**:RB_2=P_y
Init3	-	-	-	-	-	**C1**:$RB_4 = b$
D1	RA_1	RC_1	-	-	-	**C0**:RC_1=RA_1^2 . RC_1^2;**C1**:RB_3=RC_1^4
D2	-	RB_4	RB_3	-	-	**C0**: RB_3 = $RB_3.RB_4$
D3	RA_1	RB_3	RB_1	RC_1	-	**C0**:RC_2= (RA_1^4 + RB_3).(RC_1 + RB_1^2 + RB_3)
						C1: RA_1 = (RA_1^4 + RB_3)
D4	RB_3	RC_1	-	RC_2	-	**C0**: RB_1 = RB_3 . RC_1 + RC_2
A1	RB_2	RC_1	RB_1	-	-	**C0**:RB_1 = $RB_2.RC_1{}^2$+RB_1
A2	RA_1	RC_1	RA_2	-	-	**C0**:$RA_1 = RA_2.RC_1 + RA_1$
A3	RA_1	-	-	RC_1	-	**C0**:$RB_3 = RC_1.RA_1$
A4	RA_1	RC_1	RB_3	-	-	**C0**:$RA_1 = RA_1{}^2.(RB_3 + RC_1{}^2)$
A5	RA_1	RB_3	RB_1	-	-	**C0**:$RC_2 = RB_1.RB_3$
						C1:$RA_1 = RB_1{}^2 + RA_1 + RB_1.RB_3$
A6	RA_1	RB_3	RA_2	-	-	**C0**:$RC_1 = RB_3{}^2$;**C1**:$RB_3 = RA_1 + RA_2.RB_3{}^2$
A7	RB_2	RC_1	RA_2	-	-	**C0**:$RB_1 = (RA_2 + RB_2).RC_1{}^2$
A8	RB_3	RC_1	RB_1	RC_2	-	**C0**:$RB_1 = (RC_2 + RC_1).RB_3 + RB_1$
I1	-	RC_1	-	-	-	**C0**:$RC_1 = RC_1{}^2.RC_1$
I2	-	RC_1	-	-	-	**C0**:RB_3 = $RC_1{}^4$. RC_1
I3	-	RC_1	RB_3	-	-	**C0**:RB_3 = $RB_3{}^4$. RC_1
I4	-	-	-	-	RB_3	**Qout**:RC_2=$RB_3{}^3$
I5	-	RC_2	RB_3	-	-	**C0**:RB_3=$RC_2.RB_3$
I6	-	RC_1	RB_3	-	-	**C0**:RB_3=$RB_3{}^4.RC_1$
I7	-	-	-	-	RB_3	**Qout**:RC_2=$RB_3{}^7$
I8	-	RC_2	RB_3	-	-	**C0**:RB_3=$RC_2.RB_3$
I9	-	-	-	-	RB_3	**Qout**:RC_2=$RB_3{}^14$
I10	-	RC_2	RB_3	-	-	**C0**:RB_3=$RC_2.RB_3$
I11	-	RC_1	RB_3	-	-	**C0**:RB_3=$RB_3{}^4.RC_1$
I12	-	-	-	-	RB_3	**Qout**:RC_2=$RB_3{}^{14}$
I13	-	-	-	-	RC_2	**Qout**:RC_2 = $RC_2{}^{14}$
I14	-	RC_2	RB_3	-	-	**C0**:RB_3=$RC_2{}^4.RB_3$
I15	-	-	-	-	RB_3	**Qout**:RC_2=$RB_3{}^{14}$
I16	-	-	-	-	RC_2	**Qout**:RC_2 = $RC_2{}^{14}$
I17	-	-	-	-	RC_2	**Qout**:RC_2 = $RC_2{}^{14}$
I18	-	-	-	-	RC_2	**Qout**:RC_2 = $RC_2{}^{14}$
I19	-	-	-	-	RC_2	**Qout**:RC_2 = $RC_2{}^2$
I20	-	RC_2	RB_3	-	-	**C0**:RB_3=$RC_2.RB_3$
I21	-	RB_3	-	-	-	**C0**:RC_1=$RB_3{}^2$
I22	RA_1	RC_1	-	-	-	**C0**:RA_1=$RA_1.RC_1$
I23	RB_1	RC_1	-	-	-	**C0**:RB_1=$RB_1.RC_1{}^2$

Table 5.5: Control Words,[Mukhopadhyay and Chakraborty (2014)]

State	Regfile MUXIN $[c_{32}:c_{29}]$	Regfile MUXOUT $[c_{28}:c_{25}]$	Regbank signals $[c_{24}:c_{14}]$	Quadblock $[c_{13}:c_{10}]$	AU Mux C and D $[c_9:c_6]$	AU MUX A and B $[c_5:c_0]$
Init1	1010	00xx	1x01xx001x0	xxxx	0000	000000
Init2	1010	00xx	0xx1xx011x1	xxxx	xxxx	xxxxxx
Init3	1x10	xxxx	0xx1xx110xx	xxxx	xxxx	xxxxxx
D1	001x	00x0	1x01xx100x0	xxxx	1000	001001
D2	000x	x10x	0xx111100xx	xxxx	xx00	000010
D3	00x1	0100	101010001x0	xxxx	1100	100100
D4	000x	00x1	010110000xx	xxxx	xx11	000000
A1	000x	0001	0x0101000xx	xxxx	xx01	001000
A2	00x1	0010	0x00xx00110	xxxx	00xx	000010
A3	00xx	00x0	00x1xx100x0	xxxx	xx00	101000
A4	00x0	0000	0100xx101x0	xxxx	xx00	010001
A5	00x1	0100	1x1010001x0	xxxx	0100	000010
A6	001x	0100	1x011010010	xxxx	0010	001010
A7	000x	0011	0x01010001x	xxxx	xx00	001011
A8	000x	0001	010110000xx	xxxx	xx01	011000
I1	00xx	00xx	1x0xxxxx0xx	xxxx	xx00	001101
I2	000x	000x	0x01xx100xx	xxxx	xx00	000110
I3	000x	000x	xx01xx100xx	xxxx	xx00	110101
I4	01xx	000x	1x10xx100xx	0011	xxxx	xxxxxx
I5	000x	000x	0x11xx100xx	xxxx	xx00	000010
I6	000x	000x	0x01xx100xx	xxxx	xx00	110101
I7	01xx	000x	1x10xx100xx	0111	xxxx	xxxxxx
I8	000x	00xx	0x11xx100xx	xxxx	xx00	000010
I9	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I10	000x	00xx	0x11xx100xx	xxxx	xx00	000010
I11	000x	000x	0x01xx100xx	xxxx	xx00	110101
I12	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I13	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I14	000x	000x	0x11xx100xx	xxxx	xx00	111010
I15	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I16	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I17	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I18	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I19	01xx	100x	1x10xxxx0xx	0010	xxxx	xxxxxx
I20	000x	000x	0x11xx100xx	xxxx	xx00	000010
I21	000x	010x	1x0010xx0xx	xxxx	xx10	xxxxxx
I22	00x0	00x0	0x00xxxx1x0	xxxx	xx00	000000
I23	000x	00x1	0x0100xx0xx	xxxx	xx00	001000
I24	000x	0000	0xx0xx000x0	xxxx	xxxx	xxxxxx

5.4 Results

The Elliptic Curve chosen for the implementation is given below along with base-point and constants.National Institute of Standards and Technology (1994)

$$y^2 + xy = x^3 + ax^2 + b$$

where,

x = 233'h0fac9dfcbac8313bb2139f1bb755fef65bc391f8b36f8f8eb7371fd558b

y = 233'h1006a08a41903350678e58528bebf8a0beff867a7ca36716f7e01f81052

a = 1

b = 233'h066647ede6c332c7f8c0923bb58213b333b20e9ce4281fe115f7d8f90ad

The design was simulated using Bluespec Compiler and synthesised using Xilinx Vivado. The results for both are reported subsequently.

5.4.1 BSV Simulation results

The design was implemented using BSV. The simulation results for key values 2 and $2^{233} - 1$ are given below in Fig 5.5 and Fig 5.6 respectively.

```
vinod@battlestation:~/scratch/vinod/Elliptic/ECCP$ make compile link simulate
Compiling for Bluesim ...
bsc -steps-max-intervals 100 -u +RTS -K100M -RTS -sim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -elab -keep-fires -aggressive-co
nditions -no-warn-action-shadowing -show-range-conflict -g mktest  TB.bsv
checking package dependencies
All packages are up to date.
Compiling for Bluesim finished
Linking for Bluesim ...
bsc +RTS -K100M -RTS -e mktest -sim -o mktest_bsim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -keep-fires
Bluesim object reused: build_bsim/mktest.{h,o}
Bluesim object reused: build_bsim/mkECC.{h,o}
Bluesim object created: build_bsim/model_mktest.{h,o}
Simulation shared library created: mktest_bsim.so
Simulation executable created: mktest_bsim
Linking for Bluesim finished
Bluesim simulation ...
./mktest_bsim  -V
---------- Test Bench-------------

Px = 0845fd61638bac7d9e109a67a1f7047dc0fd9a5488a8468364bdc592aad
Py = 01b1420774abba2587c83900984765a8a85d776325fc39cc7823d734660
Cycles =                  33

Bluesim simulation finished
vinod@battlestation:~/scratch/vinod/Elliptic/ECCP$
```

Fig. 5.5: Simulation result for $key = 2$

```
vinod@battlestation:~/scratch/vinod/Elliptic/ECCP$ make compile link simulate
Compiling for Bluesim ...
bsc -steps-max-intervals 100 -u +RTS -K100M -RTS -sim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -elab -keep-fires -aggressive-co
nditions -no-warn-action-shadowing -show-range-conflict -g mktest  TB.bsv
checking package dependencies
compiling TB.bsv
code generation for mktest starts
Elaborated module file created: build_bsim/mktest.ba
All packages are up to date.
Compiling for Bluesim finished
Linking for Bluesim ...
bsc +RTS -K100M -RTS -e mktest -sim -o mktest_bsim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -keep-fires
Bluesim object reused: build_bsim/mkECC.{h,o}
Bluesim object created: build_bsim/mktest.{h,o}
Bluesim object created: build_bsim/model_mktest.{h,o}
Simulation shared library created: mktest_bsim.so
Simulation executable created: mktest_bsim
Linking for Bluesim finished
Bluesim simulation ...
./mktest_bsim  -V
---------- Test Bench-------------

Px = 0c478f35043e97f650ba9035ee8acfe27d264c3c6fb634074d6c4c2311c
Py = 158c45f38a18fc6ec457d699dae7e7c28d215eb8d6892c2643bfa60c75f
Cycles =                2813

Bluesim simulation finished
vinod@battlestation:~/scratch/vinod/Elliptic/ECCP$
```

Fig. 5.6: Simulation result for $key = 2^{233} - 1$

We can see from the above simulation results that the number of clock cycles required to compute the scalar multiplication varies with the key value. We can conclude that this design is prone to timing attacks. This issue is addressed in next chapter and architecture is modified accordingly.

5.5 Synthesis Report

The implemented BSV code was converted to verilog and then synthesised using *Xilinx Vivado* software. The resources utilised by the Elliptic curve crypto processor are as shown below in Fig. 5.7.

Overall, **22296 LUTs** and **1713 Flip-flops** are required for this design.

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Sat Jun 12 23:58:41 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkECC
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs |  FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| mkECC    | (top)  |      22296 |      22296 |       0 |    0 | 1713 |      0 |      0 |            0 |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
```

Fig. 5.7: ECCP resource utilisation

The resources utilised by the individual blocks within this processor are discussed in following sections.

5.5.1 Arithmetic Unit

The designed AU is a combinational circuit capable of giving the output within a clock cycle. However, the clock cycle is dictated by the Hybrid Karatsuba Multiplier as it has the longest latency. The overall AU consumes **19444 LUTs** for its operation. The synthesis report is shown in Fig. 5.8.

The resources utilised by constituents of AU are given below:

- Hybrid Karatsuba Multiplier consumes a total of **12804 LUTs**. Its synthesis report is shown in Fig. 5.9.
- Quad block which comprises of 14 cascade quad circuits require **3505 LUTs** as shown in Fig. 5.10.
- Squarer block require **79 LUTs** for computation as shown in Fig. 5.11.

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Mon Jun 14 11:49:49 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkALU
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
| Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs | FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
| mkALU    |  (top) |      19444 |      19444 |       0 |    0 |   0 |      0 |      0 |            0 |
+----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
```

Fig. 5.8: AU resource utilisation

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Sat Jun 12 11:27:29 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkHkmul
| Device       : 7al00tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
| Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs | FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
| mkHkmul  |  (top) |      12804 |      12804 |       0 |    0 |   0 |      0 |      0 |            0 |
+----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
```

Fig. 5.9: Hybrid Karatsuba Multiplier resource utilisation

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Sat Jun 12 11:28:42 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkQuadBlk
| Device       : 7al00tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+-----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
|  Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs | FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+-----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
| mkQuadBlk |  (top) |       3505 |       3505 |       0 |    0 |   0 |      0 |      0 |            0 |
+-----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
```

Fig. 5.10: Quad block resource utilisation

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Sat Jun 12 11:32:16 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkSquarer
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+-----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
|  Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs | FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+-----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
| mkSquarer |  (top) |         79 |         79 |       0 |    0 |   0 |      0 |      0 |            0 |
+-----------+--------+------------+------------+---------+------+-----+--------+--------+--------------+
```

Fig. 5.11: Squarer circuit resource utilisation

5.5.2 Register Module

Register module is a sequential block which has storage elements. It requires **2340 LUTs** and **1864 FFs** as shown in Fig. 5.12.

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Sun Jun 13 00:05:00 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkRegBlk
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs |  FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| mkRegBlk |  (top) |       2340 |       2340 |       0 |    0 | 1864 |      0 |      0 |            0 |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
```

Fig. 5.12: Register module resource utilisation

CHAPTER 6

Side Channel Attack Resistant ECCP

Side channel attack on systems can be of many types. In this work, attack based on *simple power analysis(SPA)/timing* is analysed and rectified. To make our design resistant to this attack, the architecture is tweaked to result in a new version of existing processor called as *SPA resistant ECCP*(SR-ECCP).

6.1 Timing analysis

From the FSM discussed in last chapter, we can see that the changing of states depend on the key bit(0 or 1), which makes the design prone to attacks. To demonstrate this, lets consider the example for two different key values.

1. key, k_1=8'b00101001 Double states are traversed five times while addition states twice (since there are two 1's after leading MSB 1). Hence the number of clock cycles required for this key is given below.

$$\begin{aligned} ClockCycles &= 3 + 4(5) + 8(2) + 24 \\ &= 63 \end{aligned} \tag{6.1}$$

2. key, k_2=8'b00111101 Double states are traversed five times while addition states four times (since there are four 1's after leading MSB 1). Hence the number of clock cycles required for this key is given below.

$$\begin{aligned} ClockCycles &= 3 + 4(5) + 8(4) + 24 \\ &= 79 \end{aligned} \tag{6.2}$$

Therefore, we can see that in spite of the key lengths being of same size, the number of clock cycles required vary. Hence, this design is prone to timing attacks or even in terms of power consumption, as more computation is involved for $k_i = 1$ than $k_i = 0$.

6.2 Solution for timing attack

To overcome the timing attacks, we need to make the FSM free from key bit dependency. This can be done by performing addition for every bit as doubling is done and considering the double or addition result based on the value of bit in process. For this we need to have an intermediate registers to store double result. This solution is shown pictorially in Fig 6.1.

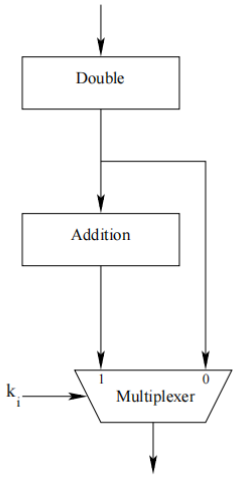

Fig. 6.1: Always Add Method to Prevent SPA

Rebeiro (2009)

The above proposed solution works only if the leading MSB '1' is at the same position. Since the left to right algorithm for scalar multiplication starts after the leading MSB '1'. To illustrate this problem consider the following example.

1. key, k_1=8'b01001001 Here the leading 1 is fount at 6^{th} bit position. So the double and addition states are traversed six times. Hence the number of clock cycles required for this key is given below.

$$\begin{aligned} ClockCycles &= 3 + 4(6) + 8(6) + 24 \\ &= 99 \end{aligned} \tag{6.3}$$

2. key, k_1=8'b00101001 Here the leading 1 is fount at 5^{th} bit position. So the double and addition states are traversed five times. Hence the number of clock cycles required

for this key is given below.

$$\begin{aligned} ClockCycles &= 3 + 4(5) + 8(5) + 24 \\ &= 87 \end{aligned} \tag{6.4}$$

We can see from the above example that just removing bit dependency from FSM doesn't make the design side channel resistant.

To overcome this we require *dummy double and addition operations* for the bits preceding the leading MSB '1'. Only initialisation of the register bank takes place when the leading 1 is being processed . Therefore, for key length l, $l - 1$ times double and addition takes place. Thus ensuring same clock cycle requirement for all key of same size. For 8 bit key sizes,

$$\begin{aligned} ClockCycles &= 3 + 4(7) + 8(7) + 24 \\ &= 111 \end{aligned} \tag{6.5}$$

irrespective of the key value the 111 clock cycles are required.

6.3 Modified ECCP

The above proposed solution is implemented in this modified version. To store the intermediate double result, bank D is added to the Register module. These registers store double result for every bit. When the current bit is 0, the next cycle would require the values stored in bank D, else the addition result from bank A, B and C are read. The modified bit independent FSM is shown in Fig. 6.2. Dummy double and additions are performed unless a leading '1' is detected. Once the leading '1' is detected, the state sequence is changed to Init1....Init3 for register initialisation and again back to D1. Once all bits are processed, inversion starts with the values stored in bank A, B and C if the last bit is 1 else from the values stored in bank D.

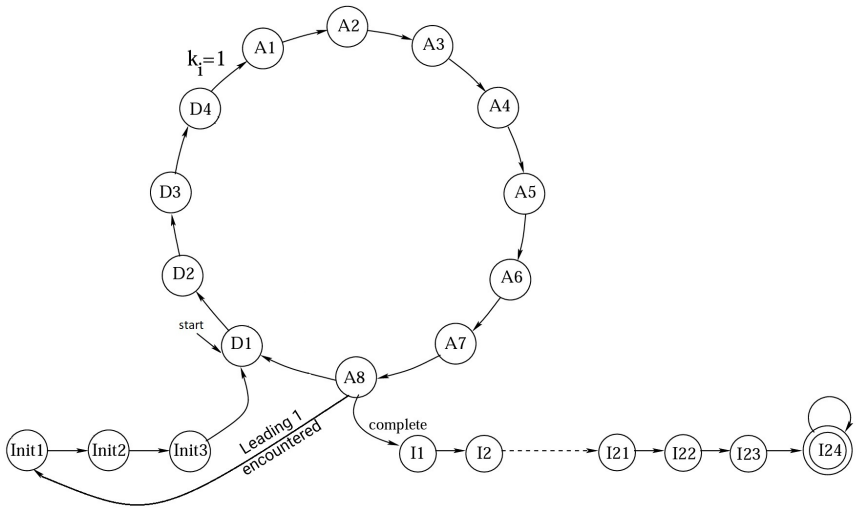

Fig. 6.2: Modified FSM

Rebeiro (2009)

The modified register module is shown in Fig. 6.3 . Additional muxes are required to read and write the appropriate values into registers.

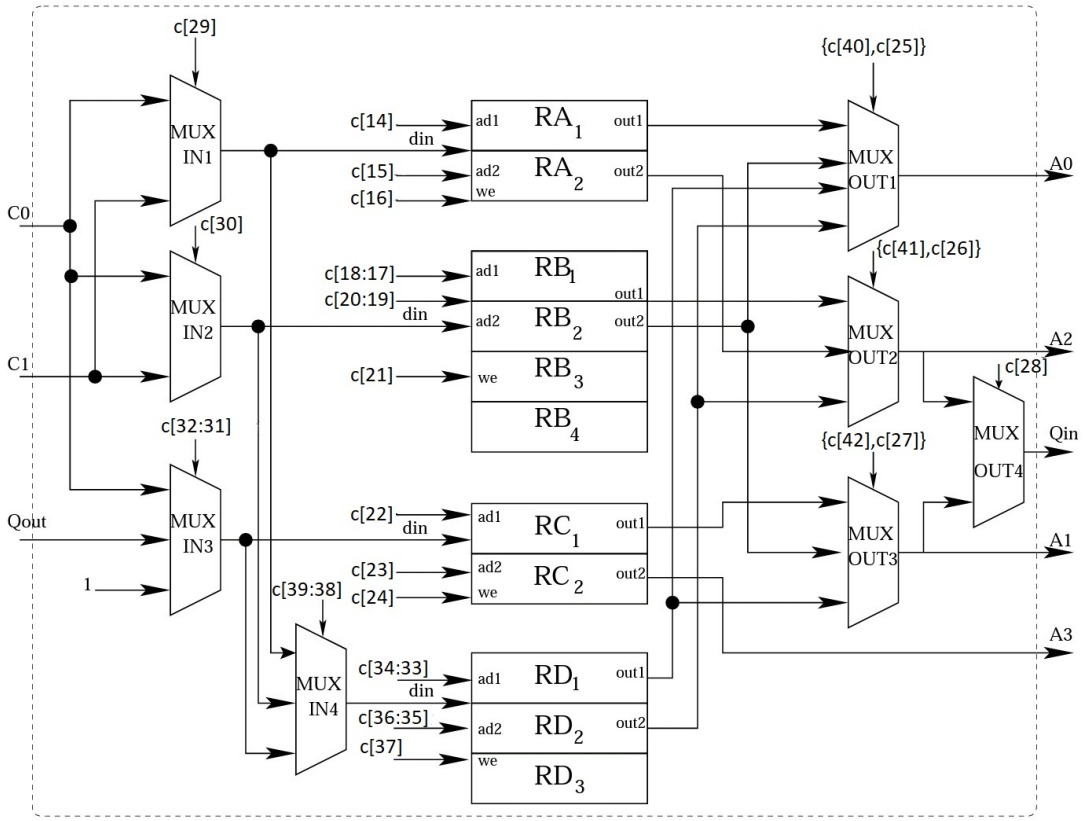

Fig. 6.3: Modified Register Module

Since additional muxes are introduced, the control words which were of 33 bits earlier, would now be of 43 bits size. The extra 10 MSB bits correspond to the additional hardware. Control words vary for each state depending on the preceding bit. If the preceding bit is 1, then we have to read from bank A, B and C. If the preceding bit is 0, then we have to read from bank D. These control words are given in Table 6.1 and 6.2.

Table 6.1: Control Words when the preceding key bit is 1

State	Additional lines $[c_{42}:c_{33}]$	Regfile MUXIN $[c_{32}:c_{29}]$	Regfile MUXOUT $[c_{28}:c_{25}]$	Regbank signals $[c_{24}:c_{14}]$	Quadblock $[c_{13}:c_{10}]$	AU Mux C and D $[c_9:c_6]$	AU MUX A and B $[c_5:c_0]$
Init1	xxx001xx00	1010	00xx	1x01xx001x0	xxxx	0000	000000
Init2	xxx011xx01	1010	00xx	0xx1xx011x1	xxxx	xxxx	xxxxxx
Init3	xxx101xx10	1x10	xxxx	0xx1xx110xx	xxxx	xxxx	xxxxxx
D1	0x0101xx10	001x	0010	1x01xx100x0	xxxx	1000	001001
D2	00xxx0xxxx	000x	x10x	0xx111100xx	xxxx	xx00	000010
D3	000001xx00	00x1	0100	101010001x0	xxxx	1100	100100
D4	000011xx01	000x	00x1	010110000xx	xxxx	xx11	000000
A1	xxxxxxxxxx	000x	0001	0x0101000xx	xxxx	xx01	001000
A2	xxxxxxxxxx	00x1	0010	0x00xx00110	xxxx	00xx	000010
A3	xxxxxxxxxx	00xx	00x0	00x1xx100x0	xxxx	xx00	101000
A4	xxxxxxxxxx	00x0	0000	0100xx101x0	xxxx	xx00	010001
A5	xxxxxxxxxx	00x1	0100	1x1010001x0	xxxx	0100	000010
A6	xxxxxxxxxx	001x	0100	1x011010010	xxxx	0010	001010
A7	xxxxxxxxxx	000x	0011	0x01010001x	xxxx	xx00	001011
A8	xxxxxxxxxx	000x	0001	010110000xx	xxxx	xx01	011000
I1	000xxxxxxx	00xx	00xx	1x0xxxx0xx	xxxx	xx00	001101
I2	000xx0xxxx	000x	000x	0x01xx100xx	xxxx	xx00	000110
I3	000xx0xxxx	000x	000x	xx01xx100xx	xxxx	xx00	110101
I4	000xx0xxxx	01xx	000x	1x10xx100xx	0011	xxxx	xxxxxx
I5	000xx0xxxx	000x	000x	0x11xx100xx	xxxx	xx00	000010
I6	000xx0xxxx	000x	000x	0x01xx100xx	xxxx	xx00	110101
I7	000xx0xxxx	01xx	000x	1x10xx100xx	0111	xxxx	xxxxxx
I8	000xx0xxxx	000x	00xx	0x11xx100xx	xxxx	xx00	000010
I9	000xx0xxxx	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I10	000xx0xxxx	000x	00xx	0x11xx100xx	xxxx	xx00	000010
I11	000xx0xxxx	000x	000x	0x01xx100xx	xxxx	xx00	110101
I12	000xx0xxxx	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I13	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I14	000xx0xxxx	000x	000x	0x11xx100xx	xxxx	xx00	111010
I15	000xx0xxxx	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I16	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I17	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I18	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I19	000xx0xxxx	01xx	100x	1x10xxxx0xx	0010	xxxx	xxxxxx
I20	000xx0xxxx	000x	000x	0x11xx100xx	xxxx	xx00	000010
I21	000xx0xxxx	000x	010x	1x0010xx0xx	xxxx	xx10	xxxxxx
I22	000xx0xxxx	00x0	00x0	0x00xxxx1x0	xxxx	xx00	000000
I23	000xx0xxxx	000x	00x1	0x0100xx0xx	xxxx	xx00	001000
I24	000xx0xxxx	000x	0000	0xx0xx000x0	xxxx	xxxx	xxxxxx

Table 6.2: Control Words when the preceding key bit is 0

State	Additional lines $[c_{42}:c_{33}]$	Regfile MUXIN $[c_{32}:c_{29}]$	Regfile MUXOUT $[c_{28}:c_{25}]$	Regbank signals $[c_{24}:c_{14}]$	Quadblock $[c_{13}:c_{10}]$	AU Mux C and D $[c_9:c_6]$	AU MUX A and B $[c_5:c_0]$
Init1	xxx001xx00	1010	00xx	1x01xx001x0	xxxx	0000	000000
Init2	xxx011xx01	1010	00xx	0xx1xx011x1	xxxx	xxxx	xxxxxx
Init3	xxx101xx10	1x10	xxxx	0xx1xx110xx	xxxx	xxxx	xxxxxx
D1	1x11010010	001x	0010	1x01xx100x0	xxxx	1000	001001
D2	000xx0xxxx	000x	x10x	0xx111100xx	xxxx	xx00	000010
D3	0110010100	00x1	0100	101010001x0	xxxx	1100	100100
D4	000011xx01	000x	00x1	010110000xx	xxxx	xx11	000000
A1	xxxxxxxxxx	000x	0001	0x0101000xx	xxxx	xx01	001000
A2	xxxxxxxxxx	00x1	0010	0x00xx00110	xxxx	00xx	000010
A3	xxxxxxxxxx	00xx	00x0	00x1xx100x0	xxxx	xx00	101000
A4	xxxxxxxxxx	00x0	0000	0100xx101x0	xxxx	xx00	010001
A5	xxxxxxxxxx	00x1	0100	1x1010001x0	xxxx	0100	000010
A6	xxxxxxxxxx	001x	0100	1x011010010	xxxx	0010	001010
A7	xxxxxxxxxx	000x	0011	0x01010001x	xxxx	xx00	001011
A8	xxxxxxxxxx	000x	0001	010110000xx	xxxx	xx01	011000
I1	100xx00010	00xx	00xx	1x0xxxxx0xx	xxxx	xx00	001101
I2	000xx0xxxx	000x	000x	0x01xx100xx	xxxx	xx00	000110
I3	000xx0xxxx	000x	000x	xx01xx100xx	xxxx	xx00	110101
I4	000xx0xxxx	01xx	000x	1x10xx100xx	0011	xxxx	xxxxxx
I5	000xx0xxxx	000x	000x	0x11xx100xx	xxxx	xx00	000010
I6	000xx0xxxx	000x	000x	0x01xx100xx	xxxx	xx00	110101
I7	000xx0xxxx	01xx	000x	1x10xx100xx	0111	xxxx	xxxxxx
I8	000xx0xxxx	000x	00xx	0x11xx100xx	xxxx	xx00	000010
I9	000xx0xxxx	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I10	000xx0xxxx	000x	00xx	0x11xx100xx	xxxx	xx00	000010
I11	000xx0xxxx	000x	000x	0x01xx100xx	xxxx	xx00	110101
I12	000xx0xxxx	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I13	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I14	000xx0xxxx	000x	000x	0x11xx100xx	xxxx	xx00	111010
I15	000xx0xxxx	01xx	000x	1x10xx100xx	1110	xxxx	xxxxxx
I16	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I17	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I18	000xx0xxxx	01xx	100x	1x10xxxx0xx	1110	xxxx	xxxxxx
I19	000xx0xxxx	01xx	100x	1x10xxxx0xx	0010	xxxx	xxxxxx
I20	000xx0xxxx	000x	000x	0x11xx100xx	xxxx	xx00	000010
I21	000xx0xxxx	000x	010x	1x0010xx0xx	xxxx	xx10	xxxxxx
I22	001xx000xx	00x0	00x0	0x00xxxx1x0	xxxx	xx00	000000
I23	001xx00100	000x	00x1	0x0100xx0xx	xxxx	xx00	001000
I24	000xx0xxxx	000x	0000	0xx0xx000x0	xxxx	xxxx	xxxxxx

6.4 Results

6.4.1 BSV Simulation results

The design was implemented using BSV. The simulation results for key values 2 and $2^{233} - 1$ are given below in Fig 6.4 and Fig 6.5 respectively.

```
vinod@battlestation:~/scratch/vinod/Elliptic/SCR$ make compile link simulate
Compiling for Bluesim ...
bsc -steps-max-intervals 100 -u +RTS -K100M -RTS -sim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -elab -keep-fires -aggressive-co
nditions -no-warn-action-shadowing -show-range-conflict -g mktest  TB.bsv
checking package dependencies
All packages are up to date.
Compiling for Bluesim finished
Linking for Bluesim ...
bsc +RTS -K100M -RTS -e mktest -sim -o mktest_bsim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -keep-fires
Bluesim object reused: build_bsim/mktest.{h,o}
Bluesim object reused: build_bsim/mkECC.{h,o}
Bluesim object created: build_bsim/model_mktest.{h,o}
Simulation shared library created: mktest_bsim.so
Simulation executable created: mktest_bsim
Linking for Bluesim finished
Bluesim simulation ...
./mktest_bsim  -V
---------- Test Bench-------------

Px = 0845fd61638bac7d9e109a67a1f7047dc0fd9a5488a8468364bdc592aad
Py = 01b1420774abba2587c83900984765a8a85d776325fc39cc7823d734660
Cycles =                2814

Bluesim simulation finished
vinod@battlestation:~/scratch/vinod/Elliptic/SCR$
```

Fig. 6.4: Simulation result for $key = 2$

```
vinod@battlestation:~/scratch/vinod/Elliptic/SCR$ make compile link simulate
Compiling for Bluesim ...
bsc -steps-max-intervals 100 -u +RTS -K100M -RTS -sim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -elab -keep-fires -aggressive-co
nditions -no-warn-action-shadowing -show-range-conflict -g mktest  TB.bsv
checking package dependencies
compiling TB.bsv
code generation for mktest starts
Elaborated module file created: build_bsim/mktest.ba
All packages are up to date.
Compiling for Bluesim finished
Linking for Bluesim ...
bsc +RTS -K100M -RTS -e mktest -sim -o mktest_bsim -simdir build_bsim -bdir build_bsim -info-dir build_bsim -keep-fires
Bluesim object reused: build_bsim/mkECC.{h,o}
Bluesim object created: build_bsim/mktest.{h,o}
Bluesim object created: build_bsim/model_mktest.{h,o}
Simulation shared library created: mktest_bsim.so
Simulation executable created: mktest_bsim
Linking for Bluesim finished
Bluesim simulation ...
./mktest_bsim  -V
---------- Test Bench-------------

Px = 0c478f35043e97f650ba9035ee8acfe27d264c3c6fb634074d6c4c2311c
Py = 158c45f38a18fc6ec457d699dae7e7c28d215eb8d6892c2643bfa60c75f
Cycles =                2814

Bluesim simulation finished
vinod@battlestation:~/scratch/vinod/Elliptic/SCR$
```

Fig. 6.5: Simulation result for $key = 2^{233} - 1$

We can see from the above simulation results that irrespective of key value, the number of clock cycles required for scalar multiplication are same.

6.4.2 Synthesis Report

In SR-ECCP, the Register module was modified to accommodate *bank D* for storing doubling operation result for every key bit iteration. So, extra hardware and control lines were introduced to take care of this. The AU and its sub-modules are not modified,

hence the resource utilisation of those remain the same as given in section 5.5.1.

This modified register module require **3980 LUTs** and **2563 FFs** as shown in Fig. 6.6.

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Mon Jun 14 12:45:21 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkRegBlk
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs |  FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| mkRegBlk |  (top) |       3980 |       3980 |       0 |    0 | 2563 |      0 |      0 |            0 |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
```

Fig. 6.6: Modified Register Module resource utilisation

Overall, the side channel resistant version require **24417 LUTs** and **2662 FFs**. The synthesis report is given below in Fig. 6.7.

```
report_utilization -hierarchical
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.2 (lin64) Build 2258646 Thu Jun 14 20:02:38 MDT 2018
| Date         : Sat Jun 12 11:22:38 2021
| Host         : vinod-Dell-System-XPS-L502X running 64-bit Ubuntu 16.04.7 LTS
| Command      : report_utilization -hierarchical
| Design       : mkECC
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Utilization by Hierarchy

1. Utilization by Hierarchy
---------------------------

+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| Instance | Module | Total LUTs | Logic LUTs | LUTRAMs | SRLs |  FFs | RAMB36 | RAMB18 | DSP48 Blocks |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
| mkECC    |  (top) |      24417 |      24417 |       0 |    0 | 2662 |      0 |      0 |            0 |
+----------+--------+------------+------------+---------+------+------+--------+--------+--------------+
```

Fig. 6.7: SR-ECCP resource utilisation

REFERENCES

1. **Mahboob, A.** (2004). *EFFICIENT HARDWARE AND SOFTWARE IMPLEMENTATION OF ELLIPTIC CURVE CRYPTOGRAPHY*. Doctoral thesis, NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY.

2. **Menezes, A. J.**, **S. A. Vanstone**, and **P. C. V. Oorschot** (1996). *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition. ISBN 0849385237.

3. **Mukhopadhyay, D.** and **R. Chakraborty** (2014). *Hardware Security: Design, Threats, and Safeguards*. CRC Press. ISBN 9781439895849. URL `https://books.google.co.in/books?id=22TNBQAAQBAJ`.

4. **National Institute of Standards and Technology** (1994). Digital signature standard (dss). FIPS Publication 186.

5. **Rebeiro, C.** (2009). *ARCHITECTURE EXPLORATIONS FOR ELLIPTIC CURVE CRYPTOGRAPHY ON FPGAS*. Doctoral thesis, DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT-Madras, Chennai – 600036.

6. **Rebeiro, C.**, **A. D. Selvakumar**, and **A. S. L. Devi** (2006). Bitslice implementation of aes. *In* **D. Pointcheval**, **Y. Mu**, and **K. Chen** (eds.), *CANS*, volume 4301 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-49462-6. URL `http://dblp.uni-trier.de/db/conf/cans/cans2006.html#RebeiroSD06`.

7. **Rodríguez-Henríquez, F.**, **G. Morales-Luna**, **N. Saqib**, and **N. C. Cortés** (2007). Parallel itoh–tsujii multiplicative inversion algorithm for a special class of trinomials. *Designs, Codes and Cryptography*, **45**, 19–37.

8. **Weimerskirch, A.** and **C. Paar** (2006). Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, **2006**, 224.