Efficient Design and Implementation of Elliptic Curve Cryptography on FPGA

By

Zia Uddin Ahamed Khan

Thesis submitted for the Degree of Doctor of Philosophy

Department of Electronic & Electrical Engineering

The University of Sheffield

October-2015

Abstract

This thesis is concerned with challenging the design space of Elliptic Curve Cryptography (ECC) over binary Galois Field, $GF(2^m)$ in hardware on field-programmable gate array (FPGA) in terms of area, speed and latency. Novel contributions have been made at the algorithmic, architectural and implementation levels that produced leading performance figures in terms of key hardware implementation metrics on FPGA. This demonstrated performance will enable ECC to be deployed across a range of application requiring public key security using FPGA technology. The proposed low area ECC implementation outperforms relevant state of the art in both area-time and area^2-time metrics.

The proposed high throughput ECC implementation adopts a new digit serial multiplier over $GF(2^m)$ incorporating a novel pipelining technique along with algorithmic and architectural level modification to support parallel operations in the arithmetic level. The resulting throughput/area performance outperforms state of the art designs on FPGA to date.

The proposed high-speed only implementation utilises a new full-precision multiplier and smart point multiplication scheduling to reduce the latency. The resulting high speed ECC design with three multipliers achieves the lowest reported latency figure to date with high speed (450 clock cycles to get 2.83 μs on Virtex7).

Finally, the proposed low resources scalable ECC implementation is based on very low latency multiprecision multiplication and low latency multiprecision squaring. The scalable ECC point multiplication design over all NIST curves consumes very low latency and shows the best area-time performance on FPGA to date.

Acknowledgements

I am extremely grateful that my supervisor Dr. Mohammed Benaissa offered me a PhD position in the University of Sheffield. His guidance and expertise in my research field have been played a central role over the four years of my PhD research. Foremost, my sincere thank to my supervisor for everything that he did. I truly appreciate him.

I am very pleased with my examiners, Professor Said Boussakta and Dr Jonathan M. Rigelsford. I honestly thank both of my examiners for their dedication in reading my thesis, for a friendly viva, and their important suggestions to improve my thesis presentation.

I like to thank my colleagues and friends for their support throughout my PhD. I like to acknowledge Dr. M. Nabil Hassan and Dr. J. Chu, who were PhD students of my supervisor, for helping me to start my cryptography research, and Dr. S.Tan, who was a research scholar in my lab, for his support to develop new algorithms for my multiprecision arithmetic operation. I am also grateful to Russ Drieberg, a kind colleague for his work in proofreading my thesis and Sandipan Pal, PhD student, for his support during preparation of my thesis. I would like to thank Dr. Tim Good, Dr. Luke Seed and Neil Powel for their support in the FPGA implementations and presentation of the implementation results. I also like to thank Professor John David, who was head of my department, for supporting me by providing a departmental scholarship in my PhD. I always appreciate Hilary Levesley, PGR administrator, for her outstanding support. I am also thankful to Dr. Wei Liu and Dr. Charith Abhayaratne for their friendly attitude to me.

I would also like to thank my lab mates who became my friends and made my study enjoyable. I am also thankful to my Bangladeshi friends with whom I relished my PhD time.

Last, but not least, thanks to my wife, Farzana Yousuf Anee (MD in internal medicine) and our parents for their ultimate sacrifice and support to complete my PhD. I would also like to thank to my brothers and sisters for their motivation.

To all of you, I am eminently grateful to you. Thank you very much!

Zia Uddin Ahamed Khan

Sheffield

11/06/2016

Contents

Abstract i
Acknowledgements ii
Contents iii
Table of Figures viii
List of Tables x
List of Algorithms xi
Glossary xii
Chapter 1 Introduction of Thesis 1-1
1.1 Overview 1-2
1.2 Motivation 1-3
1.3 Thesis Main Contributions 1-5
1.4 Thesis Outlines 1-9
1.5 Published Papers 1-10
Chapter 2 Background 2-1
2.1 Introduction 2-2
2.2 Cryptography Basics 2-2
2.3 Cryptography Schemes 2-4
2.3.1 Symmetric Key Cryptography 2-4
2.3.2 Public Key Cryptography 2-5
2.3.3 Hash Functions 2-7
2.4 Elliptic Curve Cryptography 2-8
2.5 Finite Field Theory 2-9
2.5.1 Binary extension field $GF(2^m)$ 2-11
2.5.2 Representation of Finite Field 2-13
2.5.3 Finite field arithmetic over $GF(2^m)$ 2-13
2.6 Elliptic Curve Arithmetic 2-19
2.6.1 Elliptic Curve over Binary Fields 2-20
2.6.3 Advantages of Projective Coordinates 2-21
2.6.4 The Main Operation of ECC - Point Multiplication 2-22
2.7 Koblitz Curve 2-27

2.8 Domain Parameters of ECC 2-28
2.9 Elliptic Curve Cryptography Protocols 2-29
2.9.1 Elliptic Curve Key Generation 2-29
2.9.2 Elliptic curve Diffie-Hellman key exchange (ECDH) 2-29
2.9.3 ElGamal Elliptic Curve Cryptosystem 2-30
2.9.4 Elliptic Curve Digital Signature Algorithm (ECDSA) 2-30
2.10: Design and Implementation Issues of ECC 2-32
2.10.1 Implementation of Point Multiplication 2-32
2.10.2 Why Hardware design is suitable for the crypto processor? 2-34
2.10.3 Hardware platform-FPGA 2-34
2.10.4 Design flow 2-36
2.10.5 Design metrics 2-37
Chapter 3 Low Area elliptic Curve Cryptography 3-1
3.1 Introduction 3-2
3.2 Background 3-3
3.2.1 Koblitz Curves based ECC 3-3
3.2.2 Low Area Multiplier 3-5
3.2.3 Low Area Square Circuit 3-5
3.2.4 Inversion operation 3-5
3.3 Low Area ECC Implementation using Montgomery algorithm 3-7
3.4 Low Area ECC Implementation using Binary Algorithm 3-8
3.5 Frobenius Map based Low Area ECC Implementation 3-9
3.6 Arithmetic circuit 3-10
3.6.1 Frobenius mapping unit: 3-11
3.7 Memory unit 3-13
3.8 Interface unit 3-14
3.9 Control Unit 3-15
3.10 Latency of the proposed ECC operation 3-16
3.11 FPGA Implementation Results 3-17
3.11.1 Analysis of the results 3-19
3.12 Conclusions 3-25

Chapter 4 Implementing High Throughput/Area Elliptic Curve Cryptography 4-1
4.1 Introduction 4-2
4.2 Background 4-4
4.3 Resource Constrained High Throughput ECC 4-5
4.4 Proposed throughput/area Efficient ECC processor 4-5
4.4.1 Segmented Pipelining Based Digit Serial Multiplier 4-6
4.4.2 Optimized Memory Unit 4-8
4.4.3 Scheduling for point operations 4-9
4.5 Implementation on FPGA and Results 4-10
4.5.1 Analysis of the Results 4-12
4.6 Conclusion 4-15
Chapter 5 Implementing High Speed Elliptic Curve Cryptography 5-1
5.1 Introduction 5-2
5.2 Background 5-4
5.2.1 High Speed Scalar Point Multiplication 5-4
5.2.2 Field Arithmetic over $GF(2^m)$ 5-6
5.3 Proposed Full-precision Multiplier for High Speed ECC Application 5-6
5.3.1 Multiplier with Segmented Pipelining 5-8
5.4 Proposed High Performance ECC (HPECC) for Point Multiplication 5-9
5.4.1 Point Multiplication without Pipelining Delay 5-9
5.4.2 Multiplier with Segmented Pipelining for HPECC 5-15
5.4.3 Square Circuit, Memory Unit and Control Unit of HPECC 5-15
5.4.4 Critical path delay and clock cycles of the HPEEC 5-17
5.5 Proposed the highest possible frequency based ECC (HFECC) for Point Multiplication 5-18
5.5.1 Low latency Point Multiplication Shecdualing for HFECC 5-18
5.5.2 Square Circuits, field Inversion operation and coordinates conversion of HFECC 5-21
5.5.3 Memory Unit and Control Unit of HFECC 5-22
5.5.4 Pipelining in the ECC Architecture of HFECC 5-22
5.6 Proposed Low Latency ECC (LLECC) Processor for Point Multiplication 5-23

5.6.2 Multiplier with Segmented Pipelining for LLECC 5-28
5.6.3 Square Circuit, Memory Unit and Control Unit of LLECC 5-28
5.6.4 Critical path delay and clock cycles of the LLEEC 5-29
5.7 Implementation Results 5-30
5.7.1 Analysis of Results 5-31
5.7.2 Comparison with state of the art 5-34
5.8 Conclusions 5-38
Chapter 6 Low Latency Multiprecision Arithmetic Circuit based Scalable ECC over $GF(2^m)$ 6-1
6.1 Introduction 6-2
6.2 Background 6-6
6.2.1 Comba Multiprecision Multiplication over $GF(2^m)$ 6-8
6.2.2 Multiprecision squaring over $GF(2^m)$ 6-10
6.2.3 Multiprecision modular reduction over $GF(2^m)$ 6-10
6.2.4 Inversion and Multiprecision Addition over $GF(2^m)$ 6-11
6.3 Implementing Multiprecision Multiplier over $GF(2^m)$ 6-11
6.3.1 Preliminary of the Multiprecision Multiplier over $GF(2^m)$ 6-11
6.3.2 Proposed Comba Multiprecision Multiplier over $GF(2^m)$ 6-13
6.3.3 Implementation of Proposed Comba Multiprecision Multiplier on FPGA 6-17
6.3.4 Implementing Two-and-two *GF2MUL* based Multiprecision Multiplier on FPGA 6-18
6.3.5 Implementing Four-and-Four *GF2MUL* based Multiprecision Multiplier on FPGA 6-20
6.3.6 Implementing On-the-fly Reduction Unit for Multiprecision Multiplier on FPGA 6-20
6.3.7 Analytical Comparison of multiprecision multipliers on FPGA 6-26
6.4 Implementing Proposed Multiprecision Square Circuit 6-31
6.4.1 Novel Architecture of Multiprecision Square Circuit with On-the-Fly Reduction 6-31
6.4.2 Repeated Squaring 6-35
6.4.3 Comparison with relevant square circuit and discussion 6-36
6.5 Proposed Hardware Architecture of Scalable ECC 6-37

6.5.1 Proposed Montgomery Point Multiplication 6-39
6.5.2 Careful Scheduling for Point Multiplication..... 6-40
6.5.3 Proposed Scalable Multiprecision Multiplier Circuit 6-43
6.5.4 Proposed Scalable Multiprecision Square circuit 6-43
6.5.5 SRL16 based Register file 6-45
6.6 Implementation Results..... 6-47
6.7 Conclusion..... 6-50
Chapter 7 Conclusions and Future Research Work 7-1
7.1 Conclusions 7-2
7.1 Future Research Works 7-4
References..... - 1 -

Table of Figures

Figure 2.1 Symmetric key cryptography based communication system 2-4
Figure 2.2 Public key cryptography based communication 2-5
Figure 2.3 ECC hierarchy diagram 2-9
Figure 2.4 Slice in Spartan 3E FPGA 2-36
Figure 3.1 Low area ECC system architecture 3-6
Figure 3.2 Finite field multiprecision multiplier over $GF(2^{163})$ 3-11
Figure 3.3 Finite field square circuit over $GF(2^{163}$ 3-11
Figure 3.4 Frobenius mapping unit 3-12
Figure 3.5 8xm memory unit 3-13
Figure 3.6 8-bit input/output interface 3-14
Figure 3.7 FSM based control unit 3-15
Figure 3.8 Area vs digit size over $GF(2^{163})$ in S3 for Montgomery method 3-20
Figure 3.9 Area-time vs digit size over $GF(2^{163})$ in S3 for Montgomery method 3-20
Figure 3.10 Area vs *kP* algorithm (2 bit digit serial) over $GF(2^{163})$ 3-21
Figure 3.11 Latency vs *kP* algorithm (2-bit digit serial) over $GF(2^{163})$ 3-21
Figure 3.12 Area-time vs kP algorithm (2-bit digit serial) over $GF(2^{163})$ 3-22
Figure 3.13 Performance of bit-serial multiplier based ECC over all NIST curves 3-23
Figure 4.1 Proposed throughput/area efficient ECC architecture (for n=2) 4-6
Figure 4.2 Proposed careful scheduling (4 clock cycles/multiplication) 4-9
Figure 4.3 Frequency vs segment size of the ECC over $GF(2^{163})$ 4-12
Figure 4.4 Area vs segment size of the ECC over $GF(2^{163})$ 4-13
Figure 4.5 Throughput/slices vs segment size of the ECC over $GF(2^{163})$ 4-13
Figure 5.1 Proposed segmented pipelining based full-precision multiplier over $GF(2^m)$ 5-7
Figure 5.2 Proposed high performance ECC architecture 5-11
Figure 5.3 Data flow of HPECC for $k_{i+1} = 1$, $k_i = 1$ and $k_{i-1} = 1$ 5-13
Figure 5.4 Data flow of HPECC for $k_{i+1} = 0$, $k_i=1$ and $k_{i-1} = 1$ 5-14
Figure 5.5 Proposed HFECC architecture for the high speed point multiplication 5-19
Figure 5.6 Main loop operations with the k_i values, including: a) $k_i=1$ and $k_{i+1} =1$ and b) $k_i=1$ and $k_{i+1} = 0$ 5-20
Figure 5.7 Proposed low latency ECC architecture 5-23
Figure 5.8 Data flow of LLECC for $k_{i+1}=1$, $k_i=1$ and $k_{i-1} =1$ 5-25
Figure 5.9 Data flow diagram of *LLECC* for $k_{i+1}=1$, $k_i=0$ and $k_{i-1} =0$ 5-26
Figure 5.10 Area vs ECC architecture over $GF(2^{163})$ 5-32
Figure 5.11 Frequency vs ECC architecture over $GF(2^{163})$ 5-32
Figure 5.12 Latency vs ECC architecture over $GF(2^{163})$ 5-33
Figure 5.13 *kP* time vs ECC architecture over $GF(2^{163})$ 5-33
Figure 5.14 Area-time vs ECC architecture over $GF(2^{163})$ 5-34
Figure 6.1 Row-wise and Column-wise multiprecision multiplication 6-7

Figure 6.2 Proposed Comba multiprecision multiplier over $GF(2^m)$ 6-16
Figure 6.3 Proposed two-and-two *GF2MUL* based Comba multiprecision multiplier over $GF(2^m)$ 6-19
Figure 6.4 Proposed four-and-four *GF2MUL* based Comba multiprecision multiplier over $GF(2^m)$ 6-20
Figure 6.5 Reduction unit of multiprecision multiplication 6-22
Figure 6.6 Scalable multiprecision multiplier reduction over $GF(2^{233})$ 6-25
Figure 6.7 Area, latency, max. frequency vs proposed multiplier over $GF(2^{163})$ 6-28
Figure 6.8 Multiprecision square circuit over $GF(2^{163})$ 6-32
Figure 6.9 Multiprecision square circuit over $GF(2^{233})$ 6-34
Figure 6.10 Proposed scalable ECC architecture 6-38
Figure 6.11 Data flow graph of the proposed combined Montgomery point multiplication 6-41
Figure 6.12 SRL16 based 8xm memory unit 6-46

List of Tables

Table 2.1 Equivalent key size of AES, ECC and RSA/DH and a comparison of computation cost [96]. 2-6
Table 2.2 Number of field operations for a point addition and a point doubling 2-22
Table 2.3 Total field operations for the point multiplication algorithm 2-27
Table 3.1 Latency of MSB multiplier based ECC for Montgomery point multiplication... 3-17
Table 3.2 Implementation results of this proposed low area ECC for point multiplication over $GF(2^{163})$ after place and route..... 3-18
Table 3.3 Performance of the proposed ECC for different digit-size multipliers over $GF(2^{163})$ 3-19
Table 3.4 Implementation results of the proposed bit-serial multiplier based ECC over all NIST curves after place and route 3-22
Table 3.5 Comparison of the propsed ECC with the state of the art over $GF(2^{163})$ after place and route..... 3-24
Table 4.1 Latency, critical path delay and resources of digit serial multipliers over $GF(2^m)$ 4-7
Table 4.2 Latency of ECC for $\lceil m/w \rceil$ = 4, mul= M_4/M_7. add=1. sqr=2 4-10
Table 4.3 Results of our ECC over $GF(2^{163})$ after place and route 4-11
Table 4.4 FPGA implementation results after place and route in Virtex7..... 4-12
Table 4.5 Comparison of state of the art after place and route on FPGA..... 4-14
Table 5.1 Latency, critical path delay (T_{mul}) and resources of the proposed full-precision multiplier and a comparison with the relevant multiplier over $GF(2^m)$..... 5-8
Table 5.2 Critical path delay (T_{ECC}) of the proposed ECC 5-17
Table 5.3 Latency of the proposed ECC ($MUL= M_1=1$, or $M_2=2$, or $M_3=3$, $ADD=1$, $SQR=1$, and $4SQR=1$) 5-29
Table 5.4 Comparison of the results of proposed ECC with the state of the art over $GF(2^m)$ on FPGA after place and route..... 5-35
Table 6.1 *Lv, Mv, Fv* vectors generation over $GF(2^{163})$..... 6-22
Table 6.2 Latency of the proposed multiprecision multiplier..... 6-26
Table 6.3 Area and maximum frequency of the proposed multiprecision multiplier over $GF(2^{163})$ on FPGA 6-26
Table 6.4 Comparison of the proposed multiplier with the relevant multipliars 6-28
Table 6.5 Multiprecision reduction operation over $GF(2^{233})$ on the 465 bits of square output (w = 8 bit) 6-34
Table 6.6 Comparison of the proposed square circuit with the relevant square circuit..... 6-35
Table 6.7 Comparison of the proposed scalable ECC with the state of the art on FPGA after place and route 6-49

List of Algorithms

Algorithm 2.1 Addition or subtraction over $GF(2^m)$ 2-14
Algorithm 2.2 MSB field multiplication over $GF(2^m)$ 2-15
Algorithm 2.3 LSB field multiplication over $GF(2^m)$ 2-16
Algorithm 2.4 Digit serial multiplier over $GF(2^m)$ 2-17
Algorithm 2.5 Fermat's little theorem based inversion over $GF(2^m)$ (m odd) 2-18
Algorithm 2.6 Left to right binary point multiplication algorithm 2-24
Algorithm 2.7 Binary non-adjacent form method for point multiplication 2-25
Algorithm 2.8 Montgomery point multiplication 2-26
Algorithm 2.9 Projective coordinates to affine coordinates conversion 2-27
Algorithm 2.10 Elliptic curve key generation 2-29
Algorithm 2.11 Elliptic curve Diffie-Hellman key exchange (ECDH) 2-30
Algorithm 2.12 ElGamal elliptic curve key exchange 2-30
Algorithm 2.13 ElGamal elliptic curve decryption 2-31
Algorithm 2.14 ECDSA signature generation 2-31
Algorithm 2.15 ECDSA signature verification 2-32
Algorithm 3.1 Montgomery point multiplication (loop operation) 3-7
Algorithm 3.2 Combined doubling and adding operations of Montgomery algorithm 3-8
Algorithm 3.3 Modified LD mix-coordinates algorithm 3-9
Algorithm 3.4 Binary NAF based Frobenius map in the projective coordinates 3-10
Algorithm 3.5 Itoh and Tsujii multiplicative inversion algorithm 3-16
Algorithm 4.1 LD Montgomery point multiplication over $GF(2^m)$ 4-3
Algorithm 4.2 Proposed combined loop operation of the LD Montgomery point multiplication with careful scheduling 4-8
Algorithm 5.1 LD Montgomery point multiplication over $GF(2^m)$ [35] 5-5
Algorithm 5.2 Proposed combined LD Montgomery point multiplication (with each loop for six clock cycles) 5-12
Algorithm 5.3 Proposed combined LD Montgomery point multiplication (main loop) 5-19
Algorithm 5.4 Proposed low latency Montgomery point multiplication (with each loop for two clock cycles) 5-24
Algorithm 6.1 Comba multiprecision multiplication of binary polynomials 6-8
Algorithm 6.2 Multiprecision squaring over $GF(2^m)$ 6-10
Algorithm 6.3 Proposed parallel Comba multiprecision multiplication of binary polynomials 6-13
Algorithm 6.4 Fast reduction modulo $(x) = x163 + x7 + x6 + x3 + 1$ (with $W = 8$)) 6-21
Algorithm 6.5 Proposed fast reduction modulo $(x) = x233 + x74 + 1$ (with $W = 8$) 6-24
Algorithm 6.6 Proposed multiprecision squaring over $GF(2^{163})$ 6-30

Glossary

AES	Advance Encryption Standard
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
DLP	Discrete Logarithmic Problem
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
FFs	Flip-Flops
FF	Finite Field
FPGA	Field Programmable Gate Arrays
FLT	Fermat's Little Theorem
GF	Galois Field
GF2	Galois Field in the field characteristic two
GF2MUL	Multiplication in the field characteristic two
HPECC	High Performance Elliptic Curve Cryptography
IEEE	Institute of Electrical and Electronics Engineers
ISE	Integrated Synthesis Environment
LLECC	Low Latency Elliptic Curve Cryptography
LSB	List Significant Bit
LUT	Look Up Table
MD	Message Digest
MSB	Most Significant Bit
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
PDA	Personal Digital Assistants
PKC	Public Key Cryptography

RAM	Random Access Memory
RFID	Radio Frequency Identification
RSA	Rivest-Shamir-Adleman
SHA-1	Secure Hash Algorithm-1
SSL	Secure Socket Layer
WSNs	Wireless Sensor Nodes
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.
VLSI	Very Large Scale Integration
XOR	Exclusive OR (logical Operation)
τNAF	τ-adic Non-Adjacent Forms
WDDL	Wave Dynamic Differential Logic

Chapter 1 Introduction of Thesis

This chapter portrays the broad outlines of the thesis that are the general idea on the research topic, objectives of the thesis, the main contributions in the thesis, thesis structure, and a list of published and presentable works. In this chapter, the goal of the thesis is an efficient hardware design and implementation of elliptic curve cryptography is discussed. Moreover, Advantages of the hardware platform particularly FPGA platform is focused. Finally, research outlines and list of presentation of research outputs are included in this chapter.

1.1 Overview

Our daily life is now increasingly involved with the communications over wired and wireless networks. In every instant, a big data of many transactions go on in these communications. The wider applications of the communications mostly deal with sensitive data that are protected by adopting several functionalities, including confidentiality, identification and data authentications [1] [2]. These functionalities have been developed based on a scientific process named cryptography. The cryptography process provides secure communications by hiding the original message.

Cryptography can be categorised as symmetric key cryptography (also called private key cryptography) and asymmetric key cryptography (also called public key cryptography). The symmetric key is a classical method of cryptography based on private key, whereas asymmetric key cryptography is based on public key (publicly published key) and private key to secure communications. The symmetric key cryptography requires key management to distribute the private key before establishing a communication. The key management is a problem in the communication system that is solved by public key cryptography [3]. The sender in the asymmetric cryptography knows the public key of the receiver. The sender uses the public key to encrypt a message at the sender end to send. The receiver uses his private key to decrypts at the receiver-end to get original message of sender. Thus, untrusted environment can be utilised to transfer information (i.e. key exchange). The public key cryptography is, thus, widely adopted in the communications protocol such as the Secure Socket Layer (SSL), signed and encrypted mail, and single sign-on.

The underlying security of the public key cryptography depends on a mathematical hard problem [1]. The hard problem is so difficult to solve by current computing systems in a reasonable given time scale. The hard problem of Elliptic Curve Cryptography (ECC) is discrete logarithmic problem. The high security level of problem-based cryptography is attractive and hence, the ECC is increasing popular in many communication systems due to some advantages over the existing public key cryptography (for example, Rivest, Shamir and Adleman (RSA), and Diffie-Hellmann (DH)) (page 2-6). The ECC has high security per bit when ECC is compared with RSA. The ECC based cryptography needs shorter operand than that of RSA [5]. The computational advantages of the ECC make it suitable for both constrained (low area) environment and high resources (high-speed) environments.

Low resource environments such as RFID tags, personal digital assistants (PDA), sensor networks and smart phones are involved with wireless networks. The problem of the wireless network is that it is vulnerable to eavesdropping. The resource constrained wireless system has low resources such as processing capacity, power consumption, and data rate and memory space. Thus, it is a problem to adopt security in low resources wireless communications [5] [6] For the security in the low resource applications, on one hand, symmetric key cryptography is suitable as it requires low computations. But, the problem of key management in symmetric key cryptography makes it complicated for widespread applications, in particular, in where highly reliable communication is required. On the other hand, public key cryptography in the low resources applications has no special requirement of key management as the key is publicly published. Moreover, public key cryptography is now suitable for encryption and authentication in low resource wireless communications, however it requires more resources as its field size is large. Thus, the low end application requires an effective low cost solution for security. The ECC based cryptography is now feasible to enable high security in the pervasive devices [5] [6].

High-speed communications systems such as web server in the internet require high security along with faster computation flexibility. ECC provides very high security as compared to RSA for server end applications for a given key size. Moreover, the faster calculation and lower bandwidth of ECC makes it superior over RSA for internet based network security [7].

1.2 Motivation

Elliptic Curve Cryptography is going to play an important role in the public key cryptography based secure communications system. The standard body, such as the US National Institute of Standard and Technology (NIST) recommends elliptic curves due to shorter parameter than classical cryptography such as RSA. The shorter key, compact bandwidth and high security per bit to enable it to apply in low area and high speed, commercial and governmental applications, wired and wireless communications. The elliptic curve cryptography utilise complex mathematical calculation to enable protected communications.

The main computation of ECC is scalar point multiplication on the elliptic curve. The point multiplication shares the most time of ECC based cryptography protocol processing. The

point multiplication is the generation of public key Q by multiplying a base point P by a key (integer) k, hence $Q = kP$. The point multiplication of ECC is now a highly interested topic in the academia and industry to make it fit in the performance to apply at the application level. Based on the applications, the design and implementations of point multiplication can be categorised such as software, software/hardware and hardware implementations. The software implementation of the point multiplications consumes high latency due to high frequency memory operations involved. The software/hardware implementation is an improvement on software implementations, but still the implementation consumes high latency, as the software and software-hardware implementation are associated with word level computations. Nevertheless, the software and software-hardware design consumes very low power, the high latency thwarts to apply in with both low resources and high performance applications. Thus, hardware design and implementation of point multiplications consumes much fewer clock cycles and shows high speed depend on the optimisation techniques. Moreover, the standalone hardware platform is now an interested topic for both academia and industry as the advancement of hardware cells technology (i.e. low power and high speed VLSI integration) offers high security and optimum performance. Thus, hardware implementations will be ultimate efficient solution of ECC is targeted in the throughout of the thesis.

The hardware platforms such as ASIC and FPGA are considered for the ECC implementation. The FPGA based hardware implementation is a bridge between software and ASIC implementations. The advancement of FPGA technology allows FPGA applications from prototyping to medium scale industrial applications. Moreover, FPGA based hardware solution is a popular hardware platform in academia and industrial application due to some advantages over ASIC hardware solutions. The main advantages of the FPGA are, for example, such as reconfigurability, minor time scale of development, and advances in the cell integrations and low power technology. Particularly, standalone ECC hardware on the FPGA are increasing popular in the both academia and industry as the hardware design can show high performance when high security is a prime concern. In this thesis, the hardware implementation of the ECC processors in the FPGA is utilised to evaluate thesis contributions.

Many hardware implementations are presented in the literature [9] [10] [37] [45] [48-51] [53-64] [66] [71-73] [79-82] [90] [91] [93]. The ECC implementations can be two main spaces of design such as low area ECC and high speed ECC. Each design scheme has its own

design approach. For example, a low area implementation utilises low resources but high latency to fit in the target applications. In the case of high-speed design, the speed of the processing is important. The latency of point multiplication must be reduced to speed up the ECC by exploiting large resources.

Highly efficient ECC shows high efficiency in the product of area and time for ECC. There is a lack of high-efficient ECC design in the literature. Most of the hardware implementations in the ligature are straightforward implementation of an algorithm designed for software implementation. The design utilises very large area without gaining significant speed or they consume very low area with very high latency. Thus, there is a requirement of an optimised design from a hardware perspective. The poor efficiency of ECC prevents to fit in the target application for a required security. Thus, a high-efficient ECC is the vital issue for a cryptography system designer.

The main computation of the elliptic curve cryptography is the scalar point multiplication, $Q=kP$, where k is an integer (a private key) and P is a base point on the elliptic curve and the result of point multiplication is Q , the public key. The efficient point multiplication (kP) design is an important issue in the ECC cryptography. The thesis only focuses on achieving high efficiency point multiplications. The performance of the point multiplications depends on the choice of algorithm, algorithmic modifications on top level and in the low level, arithmetic units. Thus, modifications of algorithm and novel circuit design can yield a highly efficient ECC.

1.3 Thesis Main Contributions

High efficiency ECC offers several advantages to deploy ECC in the different end of applications such as low-end applications (i.e. low area or low-power applications) and high-end applications (i.e. high speed applications). Mainly, the efficient ECC consumes a lower area to compute faster point multiplication [8] [9]. An efficient ECC architecture can provide high security and high speed for the given resource. Thus, a high-efficient design can provide better area-time metric, a standard metric (efficiency) to evaluate the engineering merit of a system architecture design and implementation.

Binary curves based ECC are increasingly more popular for hardware implementation than prime field based ECC. The binary field computation has a simple addition operation due to free from carry propagation delay. The circuit based on binary field is thus suitable to design highly efficient ECC. In this thesis, we consider binary curves that are enlisted by the (NIST) [4].

Hardware implementation for low resource application requires a small area to fit in battery run low resources applications. The most of the implementation presented in the literature used either high area along with poor latency or very low area ECC with very high latency. The high latency may incur a problem for power management. Particularly, the applications of high security requirement utilises large key size of ECC. The key size increases latency geometrically that may thwart to apply in low resources applications. We have developed a low resource friendly ECC that capable of providing high security while utilising low area. The proposed low area design involves several contributions:

To yield low area ECC:

- We consider very low resources field arithmetic circuits such bit serial multiplier. The bit serial multiplier takes *m* clock cycles for a multiplication; the latency is very much lower than word level computation of the multiplication.
- Projective coordinates is a better choice than affine coordinates due to costly inversion operation in each iteration of point multiplication. The projective coordinates based point multiplication need one inversion operation while it is performing projective to affine coordinates conversion.
- We utilise several point multiplication algorithms (binary, Montgomery, Frobenius Map based NAF) to show their individual merits. In particular, the Montgomery ladder point multiplication is an efficient point multiplication algorithm even for low area.
- Field square circuit can be achieved using a multiplier to save area with an overhead of latency. For a high efficiency ECC, a single clocked based square is considered to reduce latency by utilising a small extra area. This square circuit accelerates final inversion operation when a multiplicative inversion is implemented.
- Memory unit is the largest part of a low area ECC. A high performance memory unit can help to get efficient ECC. The Memory unit can be implemented by using either block

RAM or distributed RAM. However, block RAM is a popular option, our distributed RAM based memory unit uses very low area to show very high speed.

- Finite state machine based control unit is considered and allows us to modify point multiplication algorithm to achieve concurrent field operations. Moreover, the dedicated finite state machine (FSM) can increase speed by reducing instruction delay during field operations.
- Our Frobenius map based ECC shows very low latency while utilising a very small area for conversion and can be fitted for a low resource application in which latency is a primary concern.
- Binary implementation always takes lower area, but the latency is not fixed. The latency is changed according to the change of the hamming weight of key.

To yield high throughput ECC:

- High performance (high throughput) ECC for high-speed application, for example, web server needs high performance multiplier. For high-speed applications, there are many works presenting large digit serial multiplier. Most of the work contributes in the high performance ECC by introducing low complexity digit serial multiplier to reduce the latency of point multiplication. Their digit serial multiplier shows poor clock frequency. We implement a novel digit serial multiplier to accelerate ECC operation while the ECC is keeping such area to get the best area time metric. Our novel digit serial multiplier architecture is suitable for the large digit serial multipliers applications where traditional multiplier is still beyond to meet the requirement of area and speed.

To yield high speed ECC:

- High-speed ECC implementation where speed is the only main issue needs very large size multiplier. Moreover, parallel operation of the multiplier reduces latency of the point multiplication further to speed up. In the literature, there are many high-speed works presented, targeted to achieve the highest speed by exploiting bit parallel multipliers or parallel multipliers. Most of the work failed to reach target levels of speed due to poor performance of the multipliers and data dependency in the point multiplication. There is also a gap to reach the theoretical limit of the latency due to poor performance of multiplier and overall ECC architecture. We developed a new full precision multiplier architecture

along with smart ECC architecture to reach the theoretical latency limit. Thus, our ECC can manage a new fastest figure of speed to date.

To yield low latency scalable ECC:

- Scalability of security options is a popular property of a crypto processor to change security while it is required in the future. Moreover, smart devices (smart phone, PDA, RFID tags and wireless sensor nodes (WSNs)) need low resources scalable processor to meet future security requirement without replacing the crypto processor. To develop low resource scalable ECC, there is a requirement of multiprecision multiplication, for example Comba multiprecision multiplier and multiprecision square operation to include all NIST curves in a single processor. There are several scalable ECC implementations showing very poor performance due to high latency in the arithmetic operations with word level operation. We consider Comba multiprecision multiplier is suitable for highly efficient ECC due to its inherent parallelism to improve latency.
- We develop modified Comba multiprecision multiplier to reduce latency abruptly with small overhead of the area. We evaluate the parallelism of Comba by utilising different sets of parallel multipliers.
- A novel scalable reduction circuit is developed, which can perform reduction operations on all NIST curves on the fly during multiplication. Thus, the latency of multiplication is required to accomplish a field multiplication with reduction.
- We also develop very low latency novel multiprecision square circuit. There is no latency for the reduction operations as like as our proposed multiplication. The square circuit requires only the latency that is required to access data from memory. The proposed square circuit takes only three clock cycles delays to get the first word of the square results.
- We propose a repeated squaring circuit that can start new square operation immediately after a squaring operation is finished. The repeated square operation is flexible for all NIST curves without incurring extra clock cycles for reduction and load-unload operation of data from memory. The repeated operation accelerates point multiplication and multiplicative inversion operation due to low latency and repeated square operation without delay
- The novel multiprecision circuits enable inclusion of all NIST curves in the single ECC processor. Moreover, a new low cost memory unit and careful scheduling in the point multiplication is utilised for saving latency to improve the efficiency of the scalable ECC.

The high efficiency of the scalable processor can enable the provision of high security without changing the crypto processor in low-end applications.

1.4 Thesis Outlines

The thesis is organised as follows

Chapter 2 presents necessary preliminaries of the remaining chapters to understand the novel contribution of those chapters. The Galois field arithmetic, elliptic curve arithmetic and application, and hardware platform of ECC such as FPGAs are covered in this background chapter.

In chapter 3, low area elliptic curve cryptography on FPGA is presented. The chapter includes low resources implementation of popular ECC point multiplication algorithms to achieve the best area-time metrics targeted in the low end applications.

A novel digit serial multipliers over $GF(2^m)$ for curve based cryptography application is introduced in chapter 4. The chapter describes the best high throughput per area architecture of ECC based on the digit serial multiplier.

Chapter 5 reports the fastest figure of ECC point multiplications on FPGA over $GF(2^m)$. The pipelining based full-precision multipliers and its parallel operations, and smart pipelined based ECC architecture are presented. The ECC architecture can manage to reach the theoretical limit of latency for the point multiplication utilising careful scheduling in the point operations.

Chapter 6 discusses new modified Comba multiprecision multiplier and new multiprecision square circuit. A novel multiprecision arithmetic based on the two novel circuits drastically reduced latency to achieve the best area-time scalable ECC for the low-end applications.

Finally, the thesis is summarised the main contributions in the conclusions, which is chapter 7 and a discussion of potential future works are included in there as well.

1.5 Published Papers

The contribution of the research work presented in this thesis, the following peer-reviewed academic papers were published/accepted to publish:

1. Z. Khan and M. Benaissa, "Low area ECC implementation on FPGA," in Proc. IEEE 20th Int. Conf. Electronics, Circuits, and Systems, Dec. 8-11, 2013, pp. 581-584.
2. Zia-Uddin-Ahamed Khan and M. Benaissa, "Throughput/Area Efficient ECC Processor using Montgomery Point Multiplication on FPGA," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 62, no. 11, pp. 1078-1082, Nov. 2015.
3. Zia Khan, M. Benaissa, " High Speed ECC Implementation of FPGA over GF(2^m)," in IEEE 25th International Conference on Field-programmable Logic and Applications (FPL), 2-4 Sept. 2015 pp. 1-6.
4. High speed and Low latency ECC implementation on FPGA over GF(2^m): (Accepted to publish in the IEEE Transactions on Very Large Scale Integration Systems)

Chapter 2 Background

This Chapter discusses elliptic curve cryptography application as a public key cryptography. Again, the chapter explains important terminologies of the Elliptic Curve Cryptography principle along with the underlying finite field theory of the elliptic curve cryptography. Finally, the issues involved in the implementation of Elliptic Curve Cryptography for Point multiplication are demonstrated in the chapter.

2.1 Introduction

Cryptography is a technique to change the original message so that the actual meaning of the message is difficult to retrieve. The technique of cryptography is underlying hardness of complex mathematical problem. No one can solve this problem except the target person to get the actual meaning of the sending message using available computing system in the required time scale. Modern cryptography involves with complex mathematical algorithm under the cryptographic protocol to provide authentic communication systems.

In the modern era of cryptography key exchange method, a cryptography algorithm to solve the problem of private key-based, a new milestone of cryptography became known when Whitefield Diffie and Martin Helman [3] proposed key management problem. The new proposed method and its application are termed as public key cryptography (PKC). PKC has two keys such as public key that is known by all users and private key that is saved in secret by a respective user. The sensitive data transmission among all users is performed by using public key. As the transmission is accomplished by using one way function, only the target person is able to get the transmitted data. Thus, public key becomes the main mechanism to secure data transmission in the cryptography based secure communications.

2.2 Cryptography Basics

In a communications system, the sensitive information that is transmitted between sender and receiver is open to access by all. Eavesdroppers can easily get the transmitted information if there is no protection of information. Thus, there is a requirement of security so that third parties are not able to retrieve actual information. Cryptography algorithms is associated with complex algorithms that provide security in the communication system. It is impossible for a third party to retrieve the original data as there is an exponential time requirement to get the original information. Modern PKC provides secure communication by ensuring the following issues [1], [2], [11]:

- Data confidentiality: Confidentiality is a property to secure data so that unauthorised persons cannot receive the transmitted data. Thus, third person cannot eavesdrop the data.

- Data Integrity: the receiver checks the data, whether the data is modified by the eavesdropper (i.e. attacker). In this case, the receiver uses the message to verify the content of the transmitted data has not been altered.
- Data Origin Authentication: The vital secrecy of the transmitted data is authentication. Receiver confirms the original sender identity before data transmission is established.
- Entity Authentication: other users, to authenticate the identity of the user, verify the entity of a user.
- Non-Repudiation: After transmitting data, both sender and receiver cannot reject that they have involved in the communication or denied the content of data in the future.

Fundamentally, cryptography uses keys to encrypt and decrypt information using mathematical techniques so that the above goals are achieved. A typical cryptography based communication system is described below:

Alice (A) wants to send a message (M) is called the plaintext to her friend Bob (B) through a secure communication channel based on cryptography. A third person, Eve (E) is interested to retrieve the message by eavesdropping. Alice wants to ensure security by hiding the plaintext using cryptography tools. The hiding of the plaintext by cryptography tools is performed through a certain algorithm (called cipher) which involves the use of complex mathematical operations. The hiding process of the plaintext is called encryption (*Ec*). The encrypted plaintext is called chippertext (*Ctext*). Again, the reverse operation of the encryption is called decryption (*Dc*). In the decryption process, similar algorithm (decipher) is utilised but operations on the ciphertext to retrieve the plaintext. For encryption and decryption process, the encryption key (*Ke*) is used for encryption of the plaintext and the decryption key (*Kd*) is used for decryption of the ciphertext. The mathematical model of the cryptography is shown below:

$$Ctext = Ec(M) \quad \text{where the key is } Ke \tag{2.1}$$

$$M = Dc(Ctext) \quad \text{where the key is } Kd \tag{2.2}$$

To achieve the goal of security, a distributed sequence of steps called cryptographic protocol is defined precisely in the interactions between two or multiple parties [2].

2.3 Cryptography Schemes

There are mainly three cryptography schemes such as symmetric key cryptography or private key cryptography, asymmetric key cryptography or public key cryptography and hash functions or no key algorithms. Eq. (2.1) and Eq. (2.2) can be established using the cryptographic schemes.

2.3.1 Symmetric Key Cryptography

The symmetric key cryptography algorithm uses the same key for encryption and decryption process. The symmetric key cryptography is illustrated in Figure 2.1 where Alice want to send a message (plaintext) to Bob over symmetric key cryptography based secure communication channel. If Alice uses a key to encrypt the plaintext, then the key must be sent through a secure channel to Bob. Otherwise, a key exchange is used before encryption and decryption is processed. There are several popular symmetric curve cryptography available such as Advance Encryption Standard (AES), A5, and RC5.

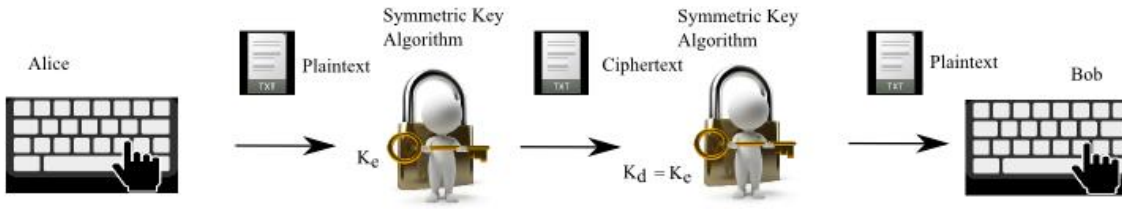


Figure 2.1 Symmetric key cryptography based communication system

Symmetric key cryptography is compact and provides higher security than asymmetric key per bit. Thus, symmetric key cryptography can be used in low end to high speed applications. The advantages of symmetric key cryptography cannot be utilised due some shortcomings [12]. One of the disadvantages of the symmetric key cryptography is the key distribution problem due to both sender and receiver must have same key. If there are few users, then the number key for every two users is less and easy to distribute the keys. But, for a large number of users, the key distribution is not practically suitable. Moreover, the keys are required to be changed frequently which is another problem for key management. Finally, in some cases, authentication and non-repudiation goals are not established by using symmetric key cryptography. The asymmetric key cryptography can overcome the symmetric key cryptography problems is discussed in the next sub-section.

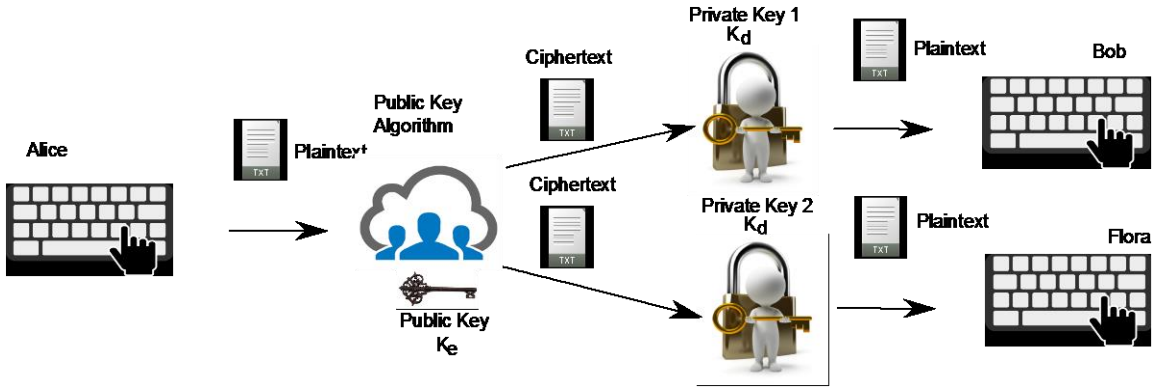


Figure 2.2 Public key cryptography based communication

2.3.2 Public Key Cryptography

Diffie and Hellman proposed public key cryptography. They solved the problem of symmetric key cryptography, including key distribution problem and key management problem [3]. In Figure 2.2, Public key cryptography is explained by using a basic protocol where separate keys are used for encryption (public key, K_e) and decryption (private key, K_d). The public key of a user is available to every user; hence, it is named public. The private key is kept in secret so that only the user knows the key. In the Figure 2.2, Alice wants to send a message (*Plaintext*) to Bob and Flora. To encrypt the messages, Alice uses the public key of Bob to send message to Bob and also uses public key of Flora to send a message to Flora. Bob and Flora can only decrypt their respective message using their own private key; however, both messages are publicly available. The public key cryptography works in one direction, this action is called "one way function" due to that the computation in the reverse direction is not practical. Thus, it is impossible to know (in the target time scale) the private key of a user (i.e. Bob) by another person (i.e. Flora) from the public key of the user (Bob) except the respective user (Bob).

The advantages of public key cryptography over symmetric key cryptography such as key exchange, key management along with confidentiality. The modern public key cryptography also provides authentication by exploiting digital signature. Thus, public key cryptography can prove one of the user would be involved to generate a message called non-repudiation. A notable feature of public key cryptography is to provide sender non-repudiation that is absent in the symmetric key cryptography scheme. There are several mechanisms involved in the public key crypto system such as key exchange protocol, digital signature algorithm and encryption.

Modern public key cryptography provides all desired goals for a given security protocol of the communications system (i.e, IP encryption (IKE/IPSEC), web traffic SLL/TLS and secure electronic mail). For a given security, public key cryptography takes long computation delay. The large size data are involved in the computation make it lower efficiency than private key cryptography. Thus, standalone public key cryptography is infeasible for some computation sensitive applications (low-end applications). In this case, a hybrid cryptosystem where public key cryptography (for key exchange) and private key cryptography (encryptions) can be used together to improve performance.

Table 2.1 Equivalent key size of AES, ECC and RSA/DH and a comparison of computation cost [96].

Symmetric key, (AES)	ECC key ($GF2^m$) / $GF(p)$	RSA/DH key	Ratio of DH Cost : ECC Cost
80 bit	163 bit / 160 bit	1024 bit	3:1
112 bit	233 bit / 224 bit	2048 bit	6:1
128 bit	283 bit / 256 bit	3072 bit	10:1
192 bit	409 bit / 384 bit	7680 bit	32:1
256 bit	571 bit / 521 bit	15360 bit	64:1

Now, There are several popular public key cryptography that are considered for high-security communications included: 1. Integer Factorisation Problem (IFL) based RSA (named from the name of inventors – Rivest, Shamir and Adelman) [13], 2. Discrete Logarithm Problem (DLP) based Diffie-Helman (DH) key exchange protocol and DH Digital Signature Algorithm (DSA), and 3. Elliptic Curve Discrete Logarithm Problem (ECDLP) based Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol and Elliptic Curve Digital Signature Algorithm (ECDSA) [4]. Comparatively, older public key cryptography such as RSA and Discrete Logarithm (DL) are involved with a computationally high intensive operation such as modular exponentiation. The modular operation is exploited large size operand to provide high security as compared to Elliptic Curve Discrete Logarithm Problem based crypto system.

A comparison of equivalent key sizes AES, RSA and ECC recommended by NIST is presented in Table 2.1 [5]. The key size of AES is compact to provide high security while comparing with public key cryptography i.e. RSA, ECC. Key size of conventional encryption algorithm i.e AES is a measure of security to protect attack. To provide an equivalent security of 80 bit long key of AES, RSA needs 1024 bit key, whereas ECC needs 163 bits key [1]. For

high security, for example, the security equivalent of 256 bit key size of AES, the key size of RSA increases abruptly as compared to ECC. Thus, ECC offers higher security per bit than RSA or Diffie-Hellman public key cryptography. The right hand column of Table 2.1 computations cost of between Diffie-Hellman public key cryptography over Elliptic Curve Cryptography [96]. For a typical protocol such as key exchanges in the DH requires a large number bit to transmit each way of a communication channel as compared to ECC. Thus, there is an overhead in bandwidth due to large key size of DH. In this case, ECC is suitable for a channel-constrained environment than first generation public key crypto systems (RSA and DH). Thus, ECC is suitable for both low-end applications (i.e. sensor networks, RFID tags, smart card) and High-speed applications (i.e. Server side).

2.3.3 Hash Functions

The hash function is a cryptography algorithm apart from private key and public key. The hash function is a one way function also called message digests. The hash function uses a hash value instead of a key. The hash value is a fixed length that is generated from a given plaintext [12]. The content of plain text or length of the plaintext is impossible to retrieve from the hash value. Hash is widely used for message integrity along with other cryptosystems. There are several hash functions used in the communication system such as SHA1 [14], MD4, MD5 [15] and the newer one is SHA3 are recommended by NIST [16].

For example, Alice wants to send a message to Bob. She calculates hash value of the message using a hash function. She then encrypts the hash value using asymmetric cryptography algorithm which is called a form of digital signature. Alice also creates an arbitrary session key for symmetric encryption. The key is used for encryption of the message. The private key is encrypted using public key of Bob using public key cryptography. Now a digital envelope is formed, including the message and encrypted session key. Alice then sends the digital envelope and digital signature to Bob. Bob retrieves the session key using his private key. Finally, the message of Alice is decrypted with the help of symmetric key algorithm using the session key. Alice also decrypted the hash value using the Alice public key to verify integrity. Bob uses the decrypted message to generate a hash value using hash algorithm and compare with the value of decrypted hash value. The hybrid procedure ensures Bob several goals such private message (symmetric encryption), the message is only for Bob (Bob’s private

key used to decrypt), the message is not be altered (by matching hash value) and Alice sent the message(Alice public key is used to generate the same hash value).

2.4 Elliptic Curve Cryptography

Elliptic curves have been used as a key tool to solve several problems in mathematics since the middle of the 19th century. The properties of elliptic curves used to solve that are factoring integers, proving Fermat's Theorem, primality testing and currently public key cryptosystems. Koblitz [17] and Miller [18] proposed elliptic curve cryptography in 1985 independently. Since then, elliptic curve cryptography was slowly adopted in commercial industry. Now, ECC is taking place of first generation cryptography, RSA as ECC is adopted by standardizing bodies such as ANSI [19], IEEE [20], ISO [16], and NIST [4].

Elliptic curve cryptography is based on the Discrete Logarithm Problem (DLP). Elliptic curve cryptosystems are implemented in finite field for a group structure. There is a set of elements (points) in the group. The special point, ø is a point at infinity is the identity of the group. The elliptic curve operation in the group is addition of points. The point addition operation is performed underlying arithmetic operation in the finite field is called field arithmetic operation. The idea behind the elliptic curve cryptography is to add a point P itself for k times where k is an integer (scalar) to achieve new point on the elliptic curve , $Q= kP$ is called *scalar point multiplication* as presented in (2.3). It is an easy way to obtain $Q =kP$ using point additions operation. The inverse operation to get the k from the given P and Q is a mathematical problem, *Elliptic Curve Discrete Logarithm Problem* (ECDLP). Unlike other DLP, ECDLP being a harder problem as until now; there is no sub-exponential time algorithm to solve the ECDLP. In $Q= kP$, k is called discrete logarithm problem of Q to the base point P, $k = \log_P Q$. Thus, Elliptic curve crypto systems provide higher security per bit than that of RSA. As a result, the communications channel requires lower bandwidth for ECC, along with the advantage of lower memory requirement than first generation cryptography. For example, for the symmetric curve key of 256 level security, ECC needs a key of 571 bits as compared to RSA with a key of 15360 bits.

$$Q= k.P = P+P+...+ P+P. \tag{2.3}$$

The main underlying operation of the ECC is scalar point multiplication. The point multiplication is consuming most of the computation time during the ECC protocol (encryption and decryption). Thus, the scalar point multiplication is the main building block to implement the elliptic curve cryptography processor. The hierarchy of elliptic curve cryptography

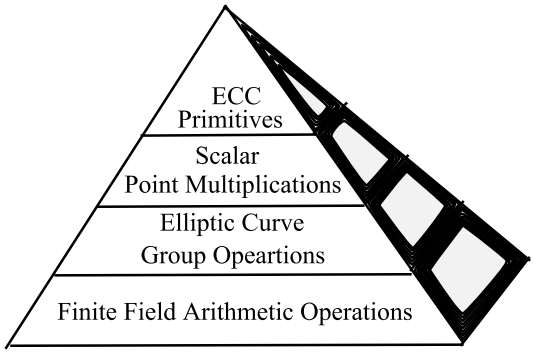


Figure 2.3 ECC hierarchy diagram

operations is shown in Figure 2.3. The point multiplication is achieved using elliptic curve arithmetic operations that are called point addition and point doubling. The underlying of the point operation is finite arithmetic operations that are represented in the bottom layer of the hierarchy diagram.

2.5 Finite Field Theory

The fundamental theory of cryptography engineering is a finite field theory. To explore the ideas and contributions of a cryptography processor, there are requirements of in depth understanding of finite field arithmetic, elliptic curve operations, point multiplications and their implementation issues. Particularly, as the finite field arithmetic circuit design is the crucial part of the ECC processor design, an extensive understanding of finite theory and circuits are the basics for the complete ECC design. Several resources of finite field theory are available in [1],[18], [22],[23] and [24] to read more details.

Modern finite field theory has developed from Galois field theory since the 19th century. The French mathematicians *Evariste Galois* developed the elementary theory, hence, named Galois field. The finite field has been widely considered for the last decades when algebraic geometry and algebra are adopted in the coding theory and cryptographic schemes.

Some fundamental definitions of finite field theory are presented below which will be used in finite field arithmetic in this thesis.

Definition 1 In the finite field theory, a finite field, *F* or a Galois field, *GF* is an algebraic structural group of finite numbers. The algebraic operations such as addition , subtraction, multiplication with and division are performed within elements of the finite field while it is maintaining algebraic laws such as associative, commutative, distributive, existence of an additive identity is 0 and a multiplicative identity is 1, additive inverse, and multiplicative inverse for nonzero elements. The group structure of *GF* also follows the law is called Group, for example, a group *M*. The group, *M* is called a commutative group or Abelian group. Modern cryptography systems are based on the Abelian groups [22].

Definition 2 If *F* is a set of an Abelian group [1] [22]:

- With additive identity, 0 is as identity element.
- With Multiplicative identity element 1.
- Distributive law exists: $(a + b)c = ac + bc$

Definition 3 If the elements of *F* is finite, then the field is called finite. The number of elements is called the order of fields, *q* [1].

Definition 4 In a finite field, *GF(q),* the order of *GF*, the content of q is a prime power is a form of $q = p^m$ where *p* is a prime number is called the characteristic of *GF* and *m* is an integer number [22].

In cryptography and coding theory, there are three main fields as shown below:

- If $m = 1$, then the *GF(p)* is called prime field
- If $m \geq 2$, then the *GF* is called extension field.
    - If $p = 2, m > 1$, then the $GF(2^m)$ is called a binary extension field or characteristic 2 field or simply binary field.
    - If $p > 2, m > 1$, then the $GF(p^m)$ is called an optimal extension field.

The binary field is a popular option over the prime field for the hardware design. This thesis covers all contributions based on *GF(2^m)*. Thus, the rest of the thesis is mainly concerned with the binary field, *GF(2^m)*.

2.5.1 Binary extension field *GF(2^m)*

The field arithmetic operations such as multiplication and addition over a binary extension field are carry free operations. The carry free operation is suitable in the hardware platform as compared to software implementation. Moreover, the carry free operation offers shorter critical path delay and lower resources than a carry based field operation in the hardware platform. Thus, the binary field arithmetic circuit can achieve faster speed with exploiting low resource in any hardware platform. Particularly, the addition over characteristic 2 circuit utilises very simple circuit (exclusive or logic, *xor*) than an integer arithmetic circuit. It can be noted that the general purpose computing system is more suitable integer arithmetic than that of *GF(2^m)*. Thus, binary extension field based arithmetic operations are problematic in the system that is designed for integer arithmetic operations.

In the binary extension field, *GF(2^m)* has 2^m-1 non-zero elements for a *m* length of binary vectors. The bases of the *m* length are $(a_{m-1}, \dots, a_1, a_0)$ as a linear space over *GF(2^m)*. A subtraction is equivalent to addition over *GF(2^m)* is performed bitwise *xor* operation.

Definition 5 A polynomial *A(x)* over a binary extension field is an algebraic structure is of the form $a_m x^m + \cdots + a_1 x + a_0$. The coefficients of the polynomial, a_i's can be integer, real, or complex [25].

Definition 6 The polynomial algebraic operations over *GF(2^m)* maintains finite field; however the algebraic operations are same as normal polynomial algebraic operations [25].

Definition 7 If two polynomials $A(x) = \sum_{i=0}^{m} a_i x^i$ and $B(x) = \sum_{i=0}^{m} b_i x^i$ over a finite field, *GF(q)* [25], then

- The addition : $\big(A(x) + B(x)\big) = \sum_{i=0}^{m} (a_i + b_i) x^i$
- The multiplications: $\big(A(x).B(x)\big) = \sum_{l=0}^{m+n} c_{l_i} x^i$; where $c_l = \sum_{i+j} (a_i . b_j)$

Definition 8 A polynomial *f(x)* over $GF(q)$ is called an irreducible polynomial when $f(x) \neq 1$ and $f(x) = A(x).B(x)$ where $A(x)$ or $B(x)$ must be unit polynomial or constant polynomial [25].

Definition 9 If an irreducible polynomial is $f(x) \in GF(q)$ of degree *m*, then $f(x)$ is used to create an extension field of $GF(q)$. The order of the extension field *m* is defined in the field by $GF(q^m)$ [25].

Definition 10 There are q^m elements over the extension field (q^m). The element can be expressed by a polynomial of degree *m-1* over a subfield $GF(q)$.

Definition 11 An absence of proper subfield is denoted as prime field [25].

Definition 12 If $A(a) = 0$ over $GF(q)$, then a is a root or zero of a polynomial $A(x) \in GF(q)$

Definition 13 For a given irreducible polynomial, $f(x)$ over subfield $GF(2)$ of degree *m* and for a root, $f(x)$ is, $f(x)$ can create a binary extension field, $GF(2^m)$ with a 2^m number of elements. The elements form a set , $(1, a, a^2, \dots, a^{m-1})$ with a basis of the field is called a *polynomial basis* of the field $GF(2^m)$ [12].

Irreducible polynomials that are used in the ECC having zeros in the most of the coefficient and the non-zero coefficients are relied in the lower order. Thus, the sparse irreducible polynomials are suitable for reduction operation in the both hardware and software platforms. The irreducible polynomial can be trinomial or pentanomial as shown below [19, 20, 26].

Trinomial: $f(x) = x^m + x^n + 1$ (2.4)

where $m > n > 1$.

Pentanomial: $f(x) = x^m + x^k + x^l + x^n + 1$ (2.5)

where $m > k > l > m > n > 1$.

2.5.2 Representation of Finite Field

There are three popular basis of the finite field, $GF(q^m)$ that are used for ECC implementation such as polynomial (also called canonical or standard) basis, normal basis and dual basis. The polynomial basis is mostly used in the hardware implementation due to carry free arithmetic operations. The normal basis and dual basis are primarily considered in the software ECC implementation as they have the advantage of squaring by doing a simple shifting operation.

Polynomial basis has several advantages such as carry free operation, low complexity for a larger field size multiplier and even, low complexity for a square circuit in the hardware platform. In the thesis, the hardware ECC implementation exploits the polynomial basis operations over $GF(2^m)$. Thus the rest of the thesis will only consider binary extension field or simply binary field, $GF(2^m)$.

Definition 14 A polynomial basis is formed with a set of elements $(1, a, a^2, \dots, a^{m-1})$ where a is a root of a generator polynomial (also called irreducible polynomial) $f(x)$ [25].

2.5.3 Finite field arithmetic over $GF(2^m)$

In the thesis, the binary field is considered for the design and implementation of field arithmetic circuits to exploit some advantages of hardware platform. Binary filed arithmetic circuits, in particular, allows some flexibility in the hardware design due to carry free arithmetic operations. Thus, multiplication and addition operations are compact and faster. Moreover, square operation in the binary field is linear as the addition of even number of '1' is zero. Thus, square circuit in binary field is a low complexity and faster than multiplier circuit. Again, in the binary field, polynomial arithmetic operation is regular, hence, suitable for pipelining. The binary field arithmetic circuits discussed here briefly will be used throughout the rest of the thesis.

Addition and subtraction

Binary arithmetic field addition and subtraction are same operations. For example, a and b are two polynomials over $GF(2^m)$ with degrees of $(m-1)$. The polynomial a, $a(x) = \sum_{i=0}^{m} a_i x^i$ and the polynomial b, $b(x) = \sum_{i=0}^{m} b_i x^i$ where $a_i, b_i \in \{0,1\}$. The addition and

Algorithm 2.1 Addition or subtraction over $GF(2^m)$

Input: $a(x) = \sum_{i=0}^{m-1} a_i x^i$ and $b(x) = \sum_{i=0}^{m-1} b_i x^i$ over $GF(2^m)$

Output: $c(x) = \sum_{i=0}^{m} c_i x^i$

$c(x) = a(x)\ xor\ b(x)$

Return $c(x)$

subtraction are accomplished bitwise exclusive or, *xor* operation as shown in algorithm 2.1 [1]. Thus, the area-time complexity is neglected for a binary field addition circuit.

Multiplication over GF(2^m)

Binary field multiplication is considered performance-critical arithmetic operation for the ECC implementation when point multiplication is performed in the projective coordinates system. In the field multiplication, two *m*-bit operands (multiplicand and multiplier) are the two finite elements, for example, a and b over $GF(2^m)$ and an irreducible polynomial, $f(x)$ is $(m+1)$-bit length . The product of the multiplication, $C'(x)$ in (2.6) is $2m-1$-bit length is required to reduce to *m*-bit necessary to maintain the field size *m*-bit is shown below:

$$c'(x) = a(x).b(x) = \sum_{i=0}^{m-1} a_i x^i \sum_{i=0}^{m-1} b_i x^i \quad (2.6)$$

$$c(x) = c'^{(x)} Modf(x) \quad (2.7)$$

There are several architectural options for carrying out the Eq. (2.7) [1]. The binary field multiplication architecture can be classified such as a bit serial multiplier, digit serial, bit parallel multiplier and hybrid. Each of the architectures has different area-time complexities are considered on the basis of design space. Bit serial multiplier consumes a very low area, but high latency, whereas bit parallel multiplier has consumed high resources, but very low latency [27, 28]. The consideration of the type of multiplier in an ECC implementation depends on the applications, for example, a low resources ECC implementation is usually considered bit serial or small digit serial, whereas a very high speed ECC implementation may consider large size digit serial multiplier or bit parallel multiplier [28-31]. Apart from these types of multiplier, a multiprecision multiplier involved with word level multiplication may consider for a very low resource applications such as sensor nodes, RFID tag and smart card. Each of the types of

Algorithm 2.2 MSB field multiplication over *GF(2^m)*

Input: $a(x) = \sum_{i=0}^{m-1} a_i x^i$ and $b(x) = \sum_{i=0}^{m-1} b_i x^i$ over *GF(2^m)* and irreducible polynomial $f(x) = x^m + k(x)$.

Output: c$(x) = \sum_{i=0}^{m} c_i x^i = a(x).\mathrm{b}(x)$

Step1 $c \leftarrow 0$

Step2 For *i* from *m-1* to 0 do

Step3.1 $c \leftarrow$leftshift*(c) xor* $c_{m-1}k$.

Step3.2 $c \leftarrow c\ xor\ b_i a$

end if;

Return (c).

multiplier has some merits and demerits can be justified by standard time area metrics when it is comparable to the state-of-the-art.

Bit serial, digit serial and bit parallel multipliers over GF(2^m)

The two operands of a bit serial multiplier over *GF(2^m)* such that one is *m* bit multiplicand and another one is 1 bit multiplier. The multiplication operation is accomplished simple shift and add method. The order of multiplier's bit can be from least significant bit (right) to most significant bit (left) of an operand (multiplier) or most significant bit to least significant bit of an operand (multiplier); thus, named as Least Significant Bit (LSB) multiplier and Most Significant Bit (MSB) multiplier respectively.

In a bit serial multiplier, the latency for both types of bit serial multiplier is *m* clock cycles as each iteration only generates one bit multiplicand and *m* bit multiplier product's results. The algorithms for MSB multiplier and LSB multiplier are shown in the algorithm 2.2 and algorithm 2.3 respectively.

The area for bit serial multiplier is *0(m)* and time complexity is *0(m).* Thus, the bit serial multiplier exploits very low area; however, the bit serial multiplier takes *m* clock cycles delay. To improve latency, a digit serial multiplier is used to reduce latency.

In digit serial multiplier, digit size, *d* is *1>d>m*. In digit serial multiplier, multiple bits (*d* digit)of multiplier multiply *m* size operand in a single clock cycle. Thus, the latency is

Algorithm 2.3 LSB field multiplication over *GF(2^m)*

Input: : $a(x) = \sum_{i=0}^{m-1} a_i x^i$ and $b(x) = \sum_{i=0}^{m-1} b_i x^i$ over *GF(2^m)*

Output: c$(x) = \sum_{i=0}^{m} c_i x^i = a(x).\mathrm{b}(x) mod\ f(x)$

Step1 $R(x) \leftarrow 0$

Step2 $If\ a_0 = '1'$ then

$R(x) \leftarrow b(x)$ else

$R(x) \leftarrow 0$

End if;

Step3 For *i* from *1* to *m-1* do

Step3.1 $b \leftarrow b.x\ mod\ f(x)$

Step3.2 $If\ a_i = '1'$ then

$R(x) \leftarrow b(x)\ xor\ b$

end if;

Return c$(x) = R(x)$

reduced to *m/d* clock cycles which is equal to the number digits, *s*. if an *m* bit multiplicand is *b* and each digit of the multiplicand is *B*, then, $b = \sum_{i=0}^{s-1} B_i x^{di}$. An MSB digit serial multiplier algorithm is shown in the algorithm 2.4. The product of the multiplier is *m+d-1* bit. The process of multiplications is similar to the bit serial multiplier, but the shifting is for *d* bit instead of single bit shifting as a bit serial multiplier. The complexity of area of the digit serial multiplier is *0(m.d)* and time complexity *0(m/d).*

The bit parallel multiplier has higher complexity than sequential multiplier (bit serial or digit serial multiplier). A bit parallel multiplier multiplies *m* bit multiplicand by *m* bit multiplier is a combinational circuit. The product of the multiplier is *2m-1* bit long which takes 1 clock cycle to complete. In the multiplier, the *m* bit multiplicand multiplies each bit of the *m*-bit multiplicand. Thus, a bit parallel multiplier has high area complexity such as m^2 *and* gates and *m(m-1) xor* gates. The whole multiplication is divided into multiply and accumulation. The multiplying *m*-bit multiplicand by each bit of the multiplier is accomplished in parallel, thus the critical path delay for multiplying is one *and* gate delay and the critical path for accumulation is contributed by a chain of addition of the partial results using *m(m-1) xor* gates,

Algorithm 2.4 Digit serial multiplier over *GF(2^m)*

Input: $a(x) = \sum_{i=0}^{m-1} a_i x^i$ and $b(x) = \sum_{i=0}^{m-1} b_i x^i$ over *GF(2^m)* and irreducible polynomial $f(x) = x^m + k(x)$.

Output: c= $a.b$

Step1 $c \leftarrow 0$

Step2 For *i* from *0* to *s-1* do

Step2.1 $c \leftarrow$*c xor* $B_i a$.

Step2.2 $a \leftarrow a.x^d\ f(x)$

end if;

Return $(c\ mod\ f(x))$.

which is equal to $\log_2 m$ of *xor* gate delays. The reduction part of the finite field bit parallel multiplier is accomplished along with same clock cycles. The area and time complexity of the reduction part is added to *GF(2^m)* multiplication depending on the non-zero terms of the irreducible polynomial.

Squaring

Squaring is considered when two inputs of a multiplication are the same due to low cost operation than multiplier. In the case of fixed irreducible polynomial, square circuit can be treated to achieve one clocked cycle squaring operation using very low resource. The square operation, in particular, over *GF(2^m)* is simplified due to $a_i a_j\ xor\ a_j a_i = 0$ where *i* and *j = 0,1,…,m-1*. Particularly, simpler circuit than a multiplier. Thus, interleaving zeros in the operand are used to perform square operation. The only major work of squaring over *GF(2^m)* is reduction operations. The reduction part of the square circuit is further simplified due to near about half of the bits are zeros that are used for reduction. Thus, the area complexity of a square circuit is lower than multiplier and the complexity depends on the irreducible polynomial. The requirement of logic gates for a square circuit over *GF(2^m)* might be as much as *m/4* xor with a time complexity either for a trinomial *f(x)* is one *xor* gate or for a pentanomial *f(x)* is 3 *xor* gates [1].

Inversion

Inversion is one of the main operation in the point multiplication of the ECC. If a is a nonzero element over *GF(2^m)*, then inversion ($a^{-1} \in GF(2^m)$) is performed by calculating unique element such that $a.a^{-1} = 1$ [1, 12]. A standalone inversion circuit is an area-time performance critical operation as compared to a field multiplication. The are-time complexity increases abruptly while the field size of ECC is increasing to enable high security. Thus, the performance of inversion mostly affects the overall ECC operation. There are two ways to accomplish the inversion operation such as a dedicated circuit with a typical latency of *2m* clock cycles and multiplicative inversion [1]. The multiplicative inversion involved with many multiplications and squaring operations contributes higher latency than that of the dedicated inversion circuit. The point multiplications in the standard affine coordinates utilise inversion operation in the each loop operation, whereas point multiplication in projective coordinates is an inversion free point operation with a cost of latency. Thus, there is a trade-off in the area and delay products for an ECC design as a standalone inversion circuit increases area and critical path delay. For a highly efficient circuit, the multiplicative inversion circuit is popular for the projective coordinates based ECC hardware as the existence multiplier and square circuits perform the only required inversion operation during coordinate conversion. A multiplicative inversion algorithm based on Fermat's little theorem is presented in the algorithm 2.5 [1].

Reduction

Algorithm 2.5 Fermat's little theorem based inversion over *GF(2^m)* (*m* odd)

Input: nonzero element , a over *GF(2^m)* and

Output: a^{-1}

Step1 Set $A \leftarrow a^2, B \leftarrow 1, x \leftarrow (m-1)/2$

Step2 while ($x \neq 0$) do

Step2.1 $A \leftarrow A \times A^{2^x}$

Step2.2 if x is even then $x \leftarrow x/2$

Else

$B \leftarrow B \times A, \quad A \leftarrow A^2, \; x \leftarrow (x-1)/2.$

Return (B).

In finite arithmetic, the reduction operation is a part of the each of the arithmetic operations to confine the length of arithmetic results within finite field. For a binary field arithmetic, fast reduction algorithm is widely used in the literature. There are several recommended fast reduction polynomials for ECC over *GF(2^m)* available can be trinomial, $f(x) = x^m + x^n + 1$ and pentanomial $f(x) = x^m + x^k + x^l + x^n + 1$ [1]. The nonzero terms of the reduction polynomial affect the performance of an ECC architecture. If the nonzero terms are in the lower order, then a large amount of the middle terms are zeros which is simple to implement. The reduction operation performs on the accumulated result which is over field size. A shifting and then, addition operation is performed to accomplish the reduction operation. The shifting operation for the trinomial requires two places, whereas shifting operation for pentanomial requires four places before performing final addition [1].

2.6 Elliptic Curve Arithmetic

A general form of the *Weierstarss equation* of an elliptic curve *E* over finite field (*GF*) in the projective is of the form [32]:

$$E:\ Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{2.11}$$

Where $a_1, a_2, a_3, a_4, a_5, a_6 \in GF$.

Again, the *Weierstarss equation* in affine (Euclidean) coordinates is of the form:

$$E:\ y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6 \tag{2.12}$$

Where $a_1, a_2, a_3, a_4, a_5, a_6 \in GF$.

If a point of the elliptic curve with the projective coordinate in (2.11) is (X, Y, Z), then the relative point in the affine coordinates in (2-12) is $(X/Z, Y/Z)$ [32]. Thus, the arithmetic in both coordinates is interchangeable. The designer can take the advantages to choose a coordinate system for the point operations on the Elliptic curve.

As any point on the elliptic curves is (2.11) and (2.12) has no discontinuity, the elliptic curves also are defined as *smooth curves.* For a form of $f(X, Y, Z) = 0$ for the Eq. (2.11) and $f(x, y) = 0$ for the Eq. (2.12), the results of partial derivatives $(\frac{df}{dX}, \frac{df}{dY}, \frac{df}{dZ})$ of (2.11) and

$(\frac{df}{dx}, \frac{df}{dy})$ of (2.12) will not be zero respectively in the case of smooth curves are presented in [32]. Similarly, the smoothness of a curve is defined by $\Delta \neq 0$ where Δ is denoted as *discriminant* of the curve *E* [1, 32].

The discriminant is presented as:

$$\Delta = -n_2^2 n_8 - 8n_4^3 - 27n_6^2 + 9n_2 n_4 n_6$$

where $n_2 = a_1^2 + 4a_2, n_4 = 2a_4 + a_1 a_3, n_6 = a_3^2 + 4a_6,$

$$n_8 = a_1^2 a_6 + 4a_2 a_6 - a_1 a_3\, a_4 + a_2 a_3^2 - a_4^2$$

and where $a_1 a_2 a_3 a_4 a_5 a_6 \in GF$.

2.6.1 Elliptic Curve over Binary Fields

For elliptic curve cryptography application, non-supersingular binary curves are considered instead of supersingular binary curves due to prone to attacks. If E_q where $q = 2^m$, then the curve of field characteristic equal to two is of the form *(GF(2^m))*. A simplified form of the Eq. (2.12) for a non-supersingular elliptic curve over *GF(2^m)* in the affine coordinates is of the form [1]:

$$E_q: y^2 + xy = x^3 + ax^2 + b \tag{2.13}$$

Where a, $b \in GF(2^m)$ and the discriminant, $\Delta = b \neq 0$.

2.6.2 Group Law over *GF(2^m)*

Here are some properties of the binary curves that represent the Abelian group [1, 32, 33]:

- Identity element ∞ (point at infinity)

If *P* is a point on the curves E_q, then $P + \infty = \infty + P = P$.

- Inverse or negative element (-*P*)
- If a point on a binary curve, $P = (x, y) \in E_q$, then negative point is

$-P = (x, x + y) \in E_q$. Thus , $P + (-P) = \infty$ where $(-P)$is called the inverse of P

Or negative of P.

- Point addition of two points on the binary elliptic curve , P and Q
  If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are two different points (i.e. $P \neq \pm Q$)) on the binary curve and $P + Q = (x_3, y_3)$ is the resultant point of addition on that binary curve, then
  $x_3 = \gamma^2 + \gamma + x_1 + x_2 + a$ (2.14)
  $y_3 = \gamma(x_1 + x_3) + x_3 + y_1$ (2.15)
  Where $\gamma = \left(\frac{y_2 + y_1}{x_2 + x_1}\right)$
- Point doubling of a point on the binary elliptic curve, P then
  If $P = (x_1, y_1)$ a point on the binary curve, where $P \neq (-P)$ and $2P = (x_3, y_3)$ is the resultant point of doubling on that binary curve, then
  $x_3 = \gamma^2 + \gamma + a$ (2.16)
  $y_3 = \gamma(x_1 + x_3) + x_3 + y_1$ (2.17)
  Where $\gamma = x_1 + \left(\frac{x_1}{y_1}\right)$

2.6.3 Advantages of Projective Coordinates

Each of the point addition and point doubling equations of the affine coordinates system presented in (2.14)-(2.17) has one inversion operation. Thus, in affine coordinates, the point operations involved with two field inversion operation which is a very costlier operation in terms of resource than the field multiplication. The elliptic curve arithmetic in the projective coordinates avoid the inversion operation by using extra field multiplication [1]. Thus, the existence multiplier can be utilised to avoid the complexity of a standalone inversion circuit to achieve a better area-time trade-off. In elliptic curve point operation, the affine coordinates system is converted into projective coordinates to do point addition and point doubling

Table 2.2 Number of field operations for a point addition and a point doubling

Coordinates System	Point operation	Multiplication	Addition	Squaring	Inversion
Affine	Addition	2	8	1	1
	Doubling	2	6	1	1
Standard Projective	Addition	13	7	1	-
	Doubling	7	4	5	-

operation. After completing point operation, the projective coordinates result is then converted to affine coordinate using one inversion operation. The inversion operation is thus achieved by using multiplicative inversion operation. However, projective coordinates system requires more field square and field addition operations are considered trivial operation as compared to field multiplication. The extra cost in the projective coordinates system is the requirement of extra memory to save intermediate results as compared to affine coordinates is shown in the Table 2.2. For each of the point addition and doubling, the affine coordinates has two multiplications such as to calculate γ and $\gamma(x_1 + x_3)$ that are less than the number multiplication of the respective point operation in projective coordinates.

In this thesis, a projective coordinate system is utilised to design and implement of the ECC hardware processor to reduce complexity. The standard projective coordinates (*X,Y,Z*) , in particular, is considered in this thesis, which affine coordinates is of the form: $x = (X/Z)$ and $y = (Y/Z)$.

2.6.4 The Main Operation of ECC - Point Multiplication

Underlying Elliptic curve Cryptography, the main operation of ECC is the scalar multiplication or commonly, point multiplication [1, 32]. The point multiplication is the foundation of the ECC based cryptography schemes such as ECDSA, Elliptic curve Diffie-Hellman, and elliptic curve encryption. In the cryptography schemes, point multiplication consumes the lion share of the time of the elliptic curve cryptography. Thus, the performance of the cryptography schemes invariably depends on the performance of the point multiplication of ECC. The performance improvement of point multiplication is an academic and industrial interest to evaluate a high efficiency ECC depending on sophisticated research. This thesis

contribution mainly focuses on the highly efficient ECC for the targeted applications (either for low end or high end applications).

The point multiplication, $Q = kP$ is a multiplication by a scalar, k and a base point, *P* on elliptic curves, *E*. There is no straight forward scalar multiplication in the elliptic curve arithmetic. The scalar multiplication is thus achieved by performing repeated addition of *P* to itself for *(k-1)* times to get a resultant point on the elliptic curve, *Q*.

A scalar, *k* is in binary form : $k = 2^{l-1}k_{l-1} + ... + 2^1k_1 + k_0$ (2.18)

Thus, $Q = kP = (2^{l-1}k_{l-1} + ... + 2^1k_1 + k_0)P$

$$= 2^{l-1}k_{l-1}P + ... + 2^1k_1P + k_0P \quad (2.19)$$

where $l \approx m$ and $k_{l-1} = '1'$.

The point multiplication can be performed using different scalar multiplication algorithms. The algorithms are divided into unknown base point, *P* and known base point, *P* (multiple *P* is pre computed) type algorithms. The details of the algorithm are described in [1, 12, 20, 26, 32].

- **Unknown base point, *P* type algorithms:**
  - *Double and Add* or *Binary* Point multiplication algorithm
    - *Right to left* binary point multiplication algorithm
    - *Left to right* binary point multiplication algorithm
  - *Non-Adjacent Form(NAF)* point multiplication algorithm
  - *Addition and Subtraction* point multiplication algorithm
  - *Windowing Methods*
    - *Sliding* window
    - *NAF* window
    - *Width-w NAF* window
  - *Montgomery Point Multiplication algorithm*
- **Known or Fixed base point, *P* type algorithms:**
  - *Fixed-Point windowing* point multiplication
  - *Fixed-Point Comb* point multiplication
  - *Fixed-Point Comb* point (with two table) multiplication

Algorithm 2.6 Left to right binary point multiplication algorithm

Input: a point, $P = (x, y) \in E/GF(2^m)$ and *m* bit binary scalar of some integer, $k = (k_{l-1}, \ldots k_1, k_0)$.

Output: $Q(x_3, y_3) = k.P$

Step1 Set $Q \leftarrow \propto$

Step2 for *i* in $(l-1)$ to *0* do

Step2.1 $Q \leftarrow 2Q$

Step2.2 if $k_i = 1$ then

$Q \leftarrow Q + P$

End if.

Return $Q(x_3, y_3)$.

The popular point multiplications such as *left to right* binary point multiplication algorithm, NAF point multiplication and Montgomery point multiplication algorithms are considered in the thesis to implement ECC over $GF(2^m)$ are discussed below.

Left to Right Binary Point Multiplication

The point multiplication (*kP*) can be achieved by repeated operations of addition and doubling. A more common basic point multiplication is left to right binary point multiplication the algorithm is shown in algorithm 2.6. the inputs of the point multiplication are a base point, *P* and a scalar *k* over $GF(2^m)$. For each value of the *ith* bit of k, doubling point operation is performed. If $k_i = 1$, then a point addition is performed along with the point doubling. For an *m*-bit binary representation of *k*, the binary point multiplication takes *m* iterations of point doubling and *m/2* iterations of point addition, if there is an average hamming weight of *k* is used.

There are several algorithms presented to improve the number iteration of point operation to speed up point multiplication over binary method. Most of them utilised precomputed points to accelerate point multiplication that are not considered in this thesis.

Algorithm 2.7 Binary non-adjacent form method for point multiplication

Input: a point, $P \in E_q$ and a positive integer, $k = (k_{l-1}, \ldots k_1, k_0)$.

Output: $(Q(x_3, y_3) = k.P$

Step1 NAF representation of $k = \sum_i^{l-1} k_i\, 2^i$ where $k_i \in \{0, \pm 1\}$

Step2 Set $Q \leftarrow \propto$

Step3 for *i* in $(l-1)$ to *0* do

Step3.1 $Q \leftarrow 2Q$.

Step3.2 if $k_i = 1$ then $Q \leftarrow Q + P$.

Step3.3 if $k_i = -1$ then $Q \leftarrow Q - P$.

End if.

Return $Q(x_3, y_3)$.

Non-Adjacent Form (NAF)

To improve binary point multiplication, a signed digit representation $k = \sum_i^{l-1} k_i\, 2^i$ where $k_i \in \{0, \pm 1\}$ called *non-adjacent form* is utilised for binary point multiplication is shown in algorithm 2.7 [1].

In Non-adjacent form representation, non-zero values are adjacent to each other i.e. $k_i . k_{i+1} = '0'$. The negative value of k_i implies point subtraction is the same cost operation as addition operation. For example, two points are such that $P_1 = (x_1, y_1)$ and $P_1 = (x_2, y_2)$ and their point subtraction, $(P_1 - P_2)$ is performed as like as addition of $(P_1 + (-P_2))$ where $-P_2 = (x_2, (x_2 + y_2))$ [1]. As the ratio of the number of doubling and addition is more than binary method for a given *k*, the speed of point multiplication is thus improved.

Montgomery Point Multiplication

Montgomery point multiplication is one of the most popular point multiplication algorithms in which point addition and point doubling are performed for every bit, (k_i) of *k*. Lopez and Dahab proposed affine and projective versions of the Montgomery point multiplication, which [34] is also referred as the Lopez-Dahab Montgomery point multiplications [35]. In this thesis, the projective version of Montgomery point multiplication that is widely considered is shown in algorithm 2.8. In the algorithm 2.8, the coordinates *X* and

Algorithm 2.8 Montgomery point multiplication

<table>
<tr><td colspan="2">INPUT: $k = (k_{t-1}, \dots, k_1, k_0)_2$ with $k_{t-1} = 1, P = (x, y) \in E(F_{2^m})$

OUTPUT: kp</td></tr>
<tr><td colspan="2">Initial Step: $P(X_1, Z_1) \leftarrow (x, 1), 2P = \ Q(X_2, Z_2) \leftarrow (x^4 + b, x^2)$

For i from $t-2$ downto 0 do</td></tr>
<tr><td colspan="2">If $k_i = 1$ then</td></tr>
<tr><td>Point addition:$P(X_1, Z_1) = P(X_1, Z_1) + Q(X_2, Z_2)$</td><td>Point Doubling: $Q(X_2, Z_2) = 2Q(X_2, Z_2)$</td></tr>
<tr><td>1. $Z_1 \leftarrow X_2 . Z_1$
2. $X_1 \leftarrow X_1 . Z_2$
3. $T \leftarrow X_1 + Z_1$
4. $X_1 \leftarrow X_1 . Z_1$
5. $Z_1 \leftarrow T^2$
6. $T \leftarrow x . Z_1$
7. $X_1 \leftarrow X_1 + T$
8. Return $P(X_1, Z_1)$</td><td>1. $Z_2 \leftarrow Z_2^2$
2. $T \leftarrow Z_2^2$
3. $T \leftarrow b . T$
4. $X_2 \leftarrow X_2^2$
5. $Z_2 \leftarrow X_2 . Z_2$
6. $X_2 \leftarrow X_2^2$
7. $X_2 \leftarrow X_2 + T$
Return $Q(X_2, Z_2)$</td></tr>
<tr><td colspan="2">Conversion Step: $x_3 \leftarrow X_1/Z_1$; $y_3 \leftarrow \left(\frac{x+X_1}{Z_1}\right)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1 Z_2)](\mathrm{x}Z_1 Z_2)^{-1} + y$.</td></tr>
</table>

Z are used for point multiplication in the projective coordinates after initial operations (affine to projective conversion). After the end of the point multiplication loop operation, the results of projective coordinates are used to convert affine coordinates as shown in the algorithm 2.9 [12]. The loop operation is performed without costly inversion operation; however, one inversion operation is involved when final conversion is performed.

The Montgomery algorithm is increasing popular for both low area and high speed implementation due to some advantages. One of the advantages of the Montgomery point multiplications is a low storage requirement due to point operation performed using *X* and *Z* coordinates. Again, point addition and point doubling operation are performed in every iteration; hence, Montgomery algorithm has a strong resistance against power attack (a side channel attack) [1, 35]. Finally, Montgomery point multiplication is suitable for parallel arithmetic operation to accelerate point operations, in particular, for high-speed design.

Algorithm 2.9 Projective coordinates to affine coordinates conversion

INPUT: $Q(X_1, Z_1)\ and\ Q(X_2, Z_2)$

OUTPUT: $Q(X_3, Y_3)$ where $X_3 \leftarrow X_1/Z_1$; $and\ Y_3 \leftarrow \left(\frac{x+X_1}{Z_1}\right)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y.$

1. $T \leftarrow Z_2.Z_1$	10. $Z_2 \leftarrow Z_2.T$
2. $Z_1 \leftarrow x.Z_1$	11. $X_2 \leftarrow Z_2 + X_2$
3. $Z_1 \leftarrow X_1 + Z_1$	12. $Z_2 \leftarrow x.T$
4. $Z_2 \leftarrow x.Z_2$	13. $Z_2 \leftarrow {Z_2}^{-1}$
5. $X_1 \leftarrow X_1.Z_2$	14. $Z_2 \leftarrow X_2.Z_1$
6. $Z_2 \leftarrow X_2 + Z_2$	15. $X_1 \leftarrow X_1.Z_1$ $\{=X_3\}$
7. $X_2 \leftarrow Z_2.Z_1$	16. $T \leftarrow x + X_1$
8. $Z_2 \leftarrow x^2$	17. $Z_1 \leftarrow Z_2.T$
9. $Z_2 \leftarrow Z_2 + y$	18. $Z_1 \leftarrow Z_1 + y$ $\{=Y_3\}$
	Return $Q(X_3, Z_3)$

A comparison of the requirement of field operation for the point multiplication in projective coordinates is shown in Table 2.3. In the binary method, total latency of the point multiplication depends on the total number of non- zeros values of k_i , whereas in Montgomery, the latency is fixed for a particular field size. Thus, the processing time and power requirement in the Montgomery based point multiplication implementation can be fixed.

Table 2.3 Total field operations for the point multiplication algorithm

Algorithm	#Multiplication	#Addition	#Square	#Inversion
Binary(Projective)	12m	13m	10m	1
Montgomery(Projective)	6m	3m	5m	1

2.7 Koblitz Curve

Here some special elliptic curves are recommended different standard bodies over binary field are called Koblitz curves [20, 26]. The Koblitz curves were proposed by N. Koblitz in 1991 [36]. The advantage of Koblitz curves over standard binary curves is that the doubling point operation can be achieved using another operation called the *Frobenius endomorphism* or Frobenius map [12]. In the Frobenius map, point doubling is replaced by a simple field squaring over point's coordinates. Thus, the latency for a point multiplication under the Frobenius map is reduced with an expanse of extra circuit for the mapping.

Koblitz curve, *E* over *GF*(2^m) is a form of binary curve equation in (2.13) [1]:

$$E: y^2 + xy = x^3 + ax^2 + 1 \qquad 2\text{-}27$$

where $a \in (0,1)$ and $b = 1$.

The squaring operation under the Frobenius map, τ for a point, $P = (x, y) \in$ Koblitz curve, E can be represented as

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

The Frobenius map form is utilised in the Non-adjacent form (NAF) of point multiplication, also called τNAF point multiplication or simply Frobenius map is of the form $k.P$ in NAF $\rightarrow$ τNAF(k).P.

Thus,

$$\tau\text{NAF}(k).P = \left(\sum_{i=0}^{n-1} u_i\, \tau^i\right).P = u_0P + \tau u_1P + \tau^2 u_1P + \cdots + \tau^{n-1}u_{n-1}P \qquad (2\text{-}30)$$

where $u_i \in (0, \pm 1)$; for $i = 0,1,2, \ldots, (n-1)$, $u_{n-1} \neq 0$, the successive value of $u_i$are non-zero and the length of n is equal to the length of τNAF. In the Frobenius NAF form of the point multiplication, total latency of the point multiplication includes $n\tau$ mapping or field coordinates square operation and $\frac{n}{3}$ point additions [1, 37]. Thus, the point multiplication latency in the Frobenius is substantially lower than that of a standard point multiplication algorithm.

2.8 Domain Parameters of ECC

Domain parameters are so designed to protect a cryptosystem from all possible attacks. There is a set of domain parameters that is used by a group of users or a specific user to secure communication based on elliptic curve cryptography. The domain parameters over finite fields, including (q, FR, a, b, P, n, h), are classified into prime field domain parameters and binary field domain parameters [2, 34] where

q = Order of Field,

FR = the field representation of any element under $GF(q)$,

a, b = Two field elements define the base point of an elliptic curve over a finite field for a field characteristic,

P = A base point $(x, y) \in E/GF(q)$ where x, y are the coordinates,

n = A prime number as an order of the P or key length of ECC is a significant parameter for security,

h = Cofactor, $\#Eq/n$ where $\#Eq = n.h;$ n is a prime number and h =small integers such as 1,2,3,4.

The term, $\#Eq$ is chosen a prime or almost prime and $n \geq 160$ to avoid Pohlig-Silver-Hellman and Polar-ρ's methods based solution of the discrete logarithm problem [32, 38, 39].

2.9 Elliptic Curve Cryptography Protocols

2.9.1 Elliptic Curve Key Generation

There are two keys such as public key and private key in the public key cryptography, which can be generated from the domain parameter of an elliptic curve. An algorithm of key generation of the elliptic curve cryptography is presented in algorithm 2.10 for a given set of domain parameters [1].

Algorithm 2.10 Elliptic curve key generation

Input: Domain parameters (q, FR, a, b, P, n, h)
Output: Private key, k and Public key, Q
Step1: Consider a random integer number $k \in (1, n-1)$.
Step2: Calculate a new point, Q from a base point P where $Q = kP$.
Step3: Return the point on elliptic curve, Q as public key and the integer k as private key.

2.9.2 Elliptic curve Diffie-Hellman key exchange (ECDH)

The basic public key algorithm which is proposed by Diffie-Hellman allows key exchange between two users to create a shared key by using Elliptic curve cryptography under public channel is shown in algorithm 2.11 [1, 19, 20, 26].

Algorithm 2.11 Elliptic curve Diffie-Hellman key exchange (ECDH)

Input: Domain parameters (q, FR, a, b, P, n, h)

Output: Shared secret, $Q = k_{alice}k_{bob}P$

Alice end:	Bob end:
Step1: Alice considers a random integer, $k_{alice} \in (1, n-1)$.	Step1: Bob considers a random integer, $k_{bob} \in (1, n-1)$.
Step2: She calculates a new point, $Q_{alice} = k_{alice}P$, and send the $Q_{alice}$to Bob.	Step2: He calculates a new point, $Q_{bob} = k_{bob}P$, and send the $Q_{bob}$to Alice.
Step3: Finally, Alice calculates the shared key, $Q = k_{alice}Q_{bob}$ on arrival of Q_{bob}.	Step3: Finally, Bob calculates the shared key, $Q = k_{bob}Q_{alice}$ on arrival of Q_{alice}.
Return: Q	Return: Q

2.9.3 ElGamal Elliptic Curve Cryptosystem

An ECC based encryption/decryption cryptosystem is proposed by ElGamal is shown in the Algorithms 2.12 and 2.13 [1, 19, 20, 26].

Algorithm 2.12 ElGamal elliptic curve key exchange

Input: Domain parameters (q, FR, a, b, P, n, h) , public key Q, and plain text m

Output: Ciphertext (C_1, C_2)

Step1: Represent the plaintext, m as a point $M \in E/GF(q)$

Step2 Alice selects a random integer number $k_{alice} \in (1, n-1)$.

Step3: Compute, $C_1$from a base point P where $C_1 = k_{alice}P$.

Step4: Compute, $C_2 = (M + k_{alice}Q)$.

Step5: Return (C_1, C_2).

2.9.4 Elliptic Curve Digital Signature Algorithm (ECDSA)

The most popular protocol based on elliptic curve digital signature algorithm (ECDSA) is an elliptic curve equivalent of a digital signature algorithm. The digital signature protocol is widely deployed in cryptography based secure communication systems as the protocol has

Algorithm 2.13 ElGamal elliptic curve decryption

Input: Domain parameters (q, FR, a, b, P, n, h) , private key k_{bob}, and ciphertext $(C_1,\ C_2)$

Output: Plaintext m

Step3: Compute $d_1 = k_{bob} C_1$.

Step4: Compute $M = C_2 - k_{bob}\, C_1)$.

Step3: Extract m from M.

Step5: Return m

been adopted by several standard bodies such as ANSI X9.62, FIPS 186-2, IEEE 1363-2000 and ISO/IEC 15946-2. The elliptic curve digital signature algorithm based signature generation and verification are presented in the Algorithms 2.14 and 2.15 [1, 19, 20, 26]. In this algorithm, a hash function, *H* is considered as pre-image and collision free.

Alice and Bob are two interested parties to set a secure communication using ECDSA and a third party, Eve is an eavesdropper or intruder in the communication channel. Alice wants to send a message with signature to Bob by using Alice's private key. As Bob knows the public key of Alice, thus Bob can verify the signature of Alice by using her public key. It is

Algorithm 2.14 ECDSA signature generation

Input: Domain parameters (q, FR, a, b, P, n, h) , private key k_{alice}, and message m

Output: Signature $(S_1,\ S_2)$

Step1: Select a random integer $k_{alice} \in (1, n-1)$.

Step2: Calculate new point $k_{alice}P = (x_3, y_3)$ and then, set $S_1 = x_3\ (mod\ n)$. if $S_1 = 0$, continue from Step1.

Step3: Compute message digest $H(m)$ where H is a hash function and $H(m)$ is a value which can be achieved using hash algorithm i.e. SHA-1.

Step4: Compute $S_2 = {k_{alice}}^{-1}(H(m) + k_{alice}\, S_1)\ mod\ n$, where k_{alice} is the private key of Alice. If $S_2 = 0$, continue from Step1

Step5: The integers $(S_1,\ S_2)$ are used to verify the signature that is included in the message m. Return$(S_1,\ S_2)$.

difficult for Eve to retrieve the original message without knowing the secret key of Alice. To retrieve the secret key is the discrete logarithm problem of ECC to solve in the given time scale.

Algorithm 2.15 ECDSA signature verification

Input: Domain parameters (q, FR, a, b, P, n, h) , Public key Q_{alice}, message m and signature $(S_1,\ S_2)$

Output: Accept or reject the signature

Step1: If $(S_1,\ S_2) \in (1.n-1)$ else return (reject the signature)
Step2: Compute $S_3 = H(m)$
Step3: Compute $T = {S_2}^{-1} mod\ n$
Step4: Compute $u = S_3 T\ mod\ n$ and $v = S_1 T\ mod\ n$
Step5: Compute $(x_2, y_2) = uP + vQ_{alice}$ and set $S_4 = x_2 mod\ n$.
Step6 If $S_1 \equiv S_4$, return (accept the signature) else return (reject the signature)

2.10: Design and Implementation Issues of ECC

2.10.1 Implementation of Point Multiplication

The computation dominant operation of the ECC public key cryptography scheme is the point multiplication (*kP*). In the thesis, the point multiplication of ECC is designed and implemented in the hardware platform as hardware aspect of design is increasing popular [36] The novel contribution of ECC depends on the efficiency in terms of the area-time metric. The area-time efficient ECC is the product of the resource (area) and the time for the point multiplications. Thus, the main objective is to increase speed by utilisin a low area. The contribution to an efficient hardware invariably depends on mainly arithmetic circuits, and then, point multiplication algorithm, memory unit and control unit. The main strategies for an efficient ECC are considered in the thesis as follows:

In the arithmetic circuit design:

- Utilising very low resource arithmetic circuits such as MSB digit (1 bit, 2 bit and 4 bit) serial for low area design.
- Developing a novel high performance digit serial multiplier to achieve a very high throughput ECC.
- Evaluating the area and speed trade-off for different pipelining techniques in the proposed digit serial multiplier

- Quantifying a high speed and highly efficient ECC using a single full-precision multiplier
- Utilising the proposed low latency squaring circuit for concurrent operation during point multiplication and low latency repeated squaring during multiplicative inversion.
- Developing a new low latency modified Comba multiprecision multiplication algorithm for the efficient low resources hardware implementation.
- Speeding up the modified multiprecision multiplication by introducing novel parallelism techniques in the Comba multiplication.
- Proposing a novel low latency multiprecision squaring circuit for all of the NIST curves to develop low latency scalable ECC.
- Designing novel on the fly reduction circuit for a further reduction in the latency of the modified multiplications circuit for all of the NIST curves.

In the memory unit design:

- Designing a very flexible and high performance memory unit based on distributed RAM for the register file.

In the control unit design:

- Considering FSM (finite state machine) based dedicated control unit based on Moore machine.

In the point multiplication algorithm:

- Modifying the instructions in the point multiplication algorithms to achieve concurrent operations to reduce the number of clock cycles per loop of the point multiplication.
- Combine point multiplication algorithm in the instructions of the point addition and point doubling to save an area of storage and thereby, reducing clock cycles.
- Evaluating merits Binary algorithm (for low area), Frobenius map (low latency) and Montgomery point multiplication algorithms (faster) for low area application by designing efficient arithmetic unit and their inherent parallelism.
- Tracing a very high speed ECC by using the theoretical limit of latency of Montgomery point multiplication under projective coordinates. The limit of the latency is formulated

by exploiting parallelism of the Montgomery Point multiplications using different combinations of high performance full-precision multipliers and concurrent adder and square operations.

In the ECC architecture:

- Choosing low resource circuitry (arithmetic circuits) and high performance memory and parallel field operation for low area ECC design.
- Designing a novel pipelined bit parallel multiplier to get the highest possible speed of the ECC.
- Developing a novel scalable ECC circuit to perform ECC operations under all NIST curves utilising the proposed novel multiprecision multiplier and novel squaring circuit.

2.10.2 Why Hardware design is suitable for the crypto processor?

The hardware implementation is considered in the state of the art due to several advantages over general-purpose processor and embedded processor as follows:

- Hardware implementation has high throughput
- Low power consumption due to stand-alone design, in particular, in reconfigurable hardware.
- Hardware design can be optimised to meet area –time constraints.
- Hardware implementation can provide high physical security
- ECC over $GF(2^m)$ curves can be considered for high speed design due to carry less operation.

2.10.3 Hardware platform-FPGA

Hardware platform can be an Application Specific Integrated Circuit (ASIC) or field programmable logic gate array (FPGA). The first one is so designed for a particular operation. Thus, the computation speed of ASIC based implementation is fast and efficient. The main drawback of the ASIC based implementation is that the design cannot be altered after fabrication. To change the design, there are requirements of redesign and re-fabrication. In particular, to change a small module in a large design is expensive. Moreover, ASIC development time is longer than an FPGA.

In general, a software implementation in the general purpose processor or embedded design is more flexible than other platforms. The software programme is very easy and flexible to alter for a given processor. The main problem of the software implementation is high latency due to word level computations. Again, there is a set of instructions is used to do all operations.

An FPGA is a reconfigurable devise ready to fill the gap between ASIC and software. The FPGA implementation can offer flexibility like software implementation and provide high performance like ASIC. As compared to software, FPGA can be utilised for spatial computation and for changing data path. Again, the FPGA has an advantage over ASIC in reconfigurability – change new hardware in the runtime. Thus, hardware in the FPGA can be exchangeable while it is requiring in the other module. Moreover, FPGA adopts soft processor along with reconfigurable logic is suitable to implement part of the hardware sensitive part of a system [41, 42, 43]. However, ASIC is preferable to complex design; new FPGA has high capacity of reconfigurations, including high density logic block, embedded processors, very high rate bit serial transceiver, clock managers, A/D converter, large storage capacity, various functional blocks. Thus, a large and complex system can be implemented on FPGA such as *system on chip* (*SoC*). The advancement of FPGA offers many features and high density logic block which convinces the academia and industry to choose FPGA for complete solutions. The FPGA is now not only playing a role in the primitive design, but also final product design in their own right to meet rapid demands such as both high performance computing and low resources applications in defence, aerospace, medical and many more.

Many vendors (Xilinx and Altera) offer different FPGA devices based on size, cost, performance and structure where most of the state of the art is implemented on Xilinx FPGA platforms [41]. There are two main types of Xilinx FPGA such as low cost FPGA (Spartan Family) and high performance FPGA (Vertex family). Each FPGA consists of a configurable logic block (CLB) which comprises “slices”. Each slice consists of four Look Up Tables (LUTs) or six Look Up Tables depending on the FPGA family. The older FPGA (Spartan3 family and Virtex4 Family) has two 4-input LUTs in a slice , as shown in Figure 2.4, whereas new FPGA family (Sparatan6 and Virtex5, Virtex7) has two 6-input LUTs in a slice . The six LUTs based FPGA is faster and highly dense with including advanced features [42]. Again, Spartan FPGA family is low power applications friendly whereas Virtex FPGA family provides high frequency operation that is suitable for high performance circuit designs.

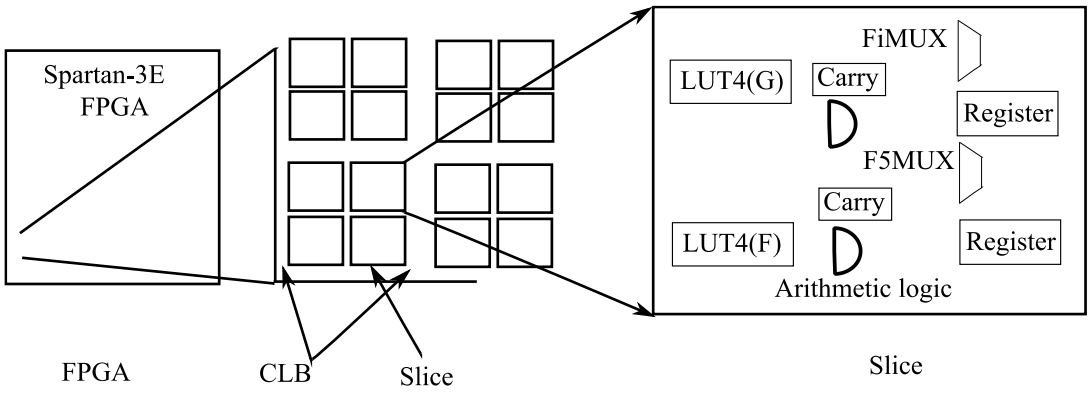


Figure 2.4 Slice in Spartan 3E FPGA

The FPGA is built on Configurable Logic Block (CLB) is comprised four slices in Spartan3E as shown in Figure 2.4. A slice of the FPGA is structured with two storages and two LUTs. The LUTs are utilised for the combinational circuit design and the storage part can be used as flip-flops (or latches). Two fabrics in the FPGA such as block RAM and SRL16 are constructed by using LUTs. Again, several other circuits such arithmetic circuits and wide combinational logic can be utilised by using the multiplexer and carry logic of the slice in the FPGA. Thus, a logic function is implemented by utilising several LUTs in between two Flip Flops (FFs). The chain of LUTs between two FFS in a particular synchronous circuit contributes critical path delay. To increase the frequency, the design of the circuit should use shorter chain can be depicted by using EDA tools such as Xilinx EDA [43].

2.10.4 Design flow

Hardware architecture is described in the Register Transfer Level (RTL) using hardware description language VHDL for FPGA implementation in the thesis. The design and implementation on FPGA mainly depend on the ingenuity and experience of the hardware designer and also depends on many iterations of the design cycle to select the best one for a targeted implementation. The selection of the best circuit may depend on the area, time and power consumptions.

The performance of a circuit depends on the coding in RTL of a logic function and utilisation of the tools. The chain of LUTs used between FFs contributes critical path delay. The chain of LUTs can be breaks using pipelining structure(registers)by expensing a delay (clock cycles). Moreover, the designer’s coding style and selection of circuit can affect the performance of the implementations.

In this thesis, an architecture is divided into the structure (logic function) to write in the RTL coding. The algorithm is implemented as separate logic functions. Data path and control unit are developed separately to optimise. The Xilinx ISE tools such as the ISE10, ISE11, and ISE 13.2 1nd ISe14.5 are used for implementation [43]. The target device is implemented in the FPGA to achieve best area and time performance.

2.10.5 Design metrics

The performance of ECC depends on the performance of the point multiplication. The performance of point multiplication can be defined based on three aspects that are area, speed and power. The metrics (index) of the performance point multiplication can be measured by only speed, or only area, or only power, or area-time, or area^2-time. The area is required to optimise for an area critical application where speed is not the main concern. Similarly, some application required to meet timing constrained to apply the circuit for a high-speed application. In this case, the speed is achieved by using large area. The standard metric is the area-time metric which is a product of area and time. A better area-time metric translates the merit of design efficiency. Throughput is another version of the efficiency that is also used in the thesis. The throughput defines the number of operations in a unit of time.

In this thesis, an efficiency FPGA based ECC implementation for point multiplication will be demonstrated mainly area-time metric for comparing relevant state of the art. Slices, LUTs and FFs, measure the resource of FPGA implementation. The time per clock cycle is measured by nanosecond, *ns* that defines the maximum frequency which is measured by megahertz, MHz. Total time of the point multiplication is calculated by multiplying the total latency (clock cycles) by the time per clock cycles. The power for ECC operation in FPGA is not considered in the thesis due to that the power consumption in FPGA is controlled by the clock trees. Again, power consumption is less for low area implementation in FPGA; thus, in this thesis, smaller circuits that consume low power are considered.

In this thesis, the Xilinx tools is utilised to implement ECC in the low cost FPGA such as Spartan family and high performance FPGA such as Vertex family. The results of the implementations are addressed after place and route(PAR) using different sets of the synthesis, map and implementation properties. The global timing constraints are also utilised repeatedly and change circuits where is possible to meet the time. The Planhead tools for floor planning

is also utilised in the some cases, but the results are not addressed in the thesis due to poor results.

Chapter 3 Low Area elliptic Curve Cryptography

In this chapter, a new compact and stand-alone design of an Elliptic Curve Cryptography (ECC) processor over Galois field GF(2^m) is analysed and implemented on FPGA for the three most popular point multiplication algorithms (the basic binary, Montgomery, and Frobenius map). This chapter demonstrates new concurrency in point addition and point doubling together with novel flexible memory and efficient arithmetic units. Area-time and area^2-time performances are investigated by exploiting a very compact bit/digit serial multiplier. A very low cost 8-bit input-output interface is used that can be embedded with 8-bit processors for low area applications.

3.1 Introduction

In many emerging applications with low resource, but acceptable performance, public key cryptography primitives are key drivers to enable strong security. Elliptic curve cryptography (ECC) is one of the most promising public key cryptography primitives for its smaller key size, which makes it suitable for low area applications. Elliptic curve cryptography was proposed by Koblitz [17] and Miller [18] in 1985 independently, and has been given standardisation by NIST [1]. There two possible design options for low area ECC such prime fields *(Fp)* and binary extension fields ($F2^m$). Prime field is widely used in the software implementation (such as embedded processor) due to build in carry based multiplier and simple square operation (by left shifting data). For the hardware implementation, the binary fields tend to provide faster arithmetic circuits and lower circuit complexity than prime fields mainly due to avoiding the carry propagation. The square circuit in binary circuit is also simple by using interleaving zero followed by reducing operation.

Low-area standalone hardware designed is considered in this work. A standalone hardware can be suitable for low resources application by using low resources arithmetic circuits. In general, the main arithmetic part of ECC field operation is the field multiplier. Bit serial multipliers consume very low area; however, each multiplication takes *m* clock cycles. The clock cycles can be improved abruptly by using small digit serial multipliers. Each of the other arithmetic circuits such as square and adder circuit consumes very low area. For the low area implementation, the major area of ECC processor is used for memory unit. The base point coordinates, key input, curve parameters, and temporary results are required to save during the point multiplication of ECC. Thus, the memory unit consumes more that 50% of the area, in particular, of the low area architecture of ECC [46]. Thus, the memory unit of ECC is required to be small but flexible. To reduce memory requirement, some concurrent operations are required by observing data independence during the point multiplication. Thus, a modification of point multiplications is required to reduce memory size as well as to reduce clock cycles for the point operations.

In this chapter, a hardware design of ECC over $GF2^{163}$, $GF2^{233}$, $GF2^{283}$, *GF2409* and $GF2^{571}$ using Koblitz curves is analysed for the three important algorithms for point multiplication (Montgomery, basic binary and Frobenius map). Here, a novel low area architecture is presented and implemented as a standalone FPGA implementation to investigate

the trade-offs between the area and speed in each case. We investigate these three implementations using bit serial and digit serial (2-bit and 4-bit) modular multipliers on different FPGA devices, Spartan(S) (S3 and S6). We compare our results with the relevant work in the same technology irrespective of low and high speed ECC design. We show our result in terms of area-time and area^2-time metrics. The area^2-time metric is considered in order to depict performance of a low area design while it is comparing with both low area and high speed implementations. Our unified and parallel steps based Montgomery method with 4bit digit serial multiplier (msd4) in S6 shows 0.12 in area-time (slice-sec) and 0.063 (slice2-sec/103) in area^2-time metrics; to our knowledge, is the best area^2-time performance reported for ECC on FPGA. To achieve this level of performance a number of original contributions, as detailed below, were made at the algorithmic level in terms of extracting concurrency, at the arithmetic level in terms of a more efficient arithmetic and crucially at implementation level in terms of a flexible memory design that exploits the concurrency extracted, and a dedicated control design.

3.2 Background

Standalone low area ECC is faster than the ECC implementation in software or software and hardware. The hardware base ECC takes very low clock cycles for point multiplication by avoiding word level computations used in the software-based implementations. The hardware control units can save the instruction delay and memory operations. The characteristic two curves, *GF(2^m)* curves based ECC is increasing common for hardware implementation due to carry free operations. The carry free circuit is not only faster, but also has consume low area and also flexible to consider large curve for high security for low resources applications. Moreover, we can consider Koblitz curve for low area implementation as it has the flexibility for optimisation than binary curve.

3.2.1 Koblitz Curves based ECC

Koblitz curve (E_a) over *GF(2^m)* can be a good option for a low resource application as it is an *anomalous* curve.

$$E_a: y^2 + xy = x^3 + ax + 1 \tag{3.1}$$

Where $a \in \{0,1\}$

Again, the anomalous curve over *GF(q)* can be mapped called Frobenius endomorphism (map) τ which can be represented by the following characteristic equation [3,4]:

$$\tau^2 \pm \tau + q = 0 \tag{3.2}$$

For Koblitz curves over *GF(2^m)* denotes $q = 2$; thus the map equation becomes

$$\tau^2 \pm \tau + 2 = 0 \tag{3.3}$$

If a point, $P(x, y)$, then it can be represented using Frobenius map, $\tau(x, y)$ as follows:

$$\tau(x, y) = \tau P = (x^2, y^2) \text{ and } \tau(\infty) = \infty \tag{3.4}$$

Now, a doubling operation (*2P)* for the anomalous curve is presented utilising Eq.(3.3) is as bellow:

$$\tau^2 \pm \tau P + 2P = 0 \tag{3.5}$$

the equation in Eq. (3.5) is also represented as bellow:

$$2P = \tau(\tau P) + \mu.\tau P \tag{3.6}$$

Where, $\mu = (-1)^{(1-a)}$.

In [4], the Eq. (3.6) translates doubling operation *(2P)*. The doubling operations include twice τ (Frobenius map operator) of *P* followed by doing a point addition between two points $\tau(\tau P)$ and $\mu.\tau P$.

From Eq. (3.6), a quadratic equation is formed:

$$\tau^2 + 2 = \mu.\tau \tag{3.7}$$

The quadratic equation in Eq. (3.7) is also defined as elliptic curve endomorphism [4]. The solution of Eq. (3.7) makes a relation between a squaring map can equivalent to scalar multiplication by the complex number in Eq. (3.6). There are two complex numbers of the quadratic equation and one of the solutions considered is shown below [1]:

$$\tau = \frac{u + \sqrt{-7}}{2} \tag{3.8}$$

Using the solution, squaring map can be included in the scalar point multiplication. The scalar multiplication on the points on the Koblitz curve can be done by using any element of the ring $Z\ (\tau)$. Thus, a complex multiplication can be achieved on Koblitz by using τ.

3.2.2 Low Area Multiplier

The low area multiplier circuit can be multiprecision multipliers or bit serial multiplier. Multiprecision multiplier has high latency to perform a field multiplication than a bit serial multiplier; however, a multiprecision multiplier circuit consumes lower area. The bit serial multipliers latency can be improved further by using small digit serial multiplier with small overhead of the area. If *w* is digit size, then *m/w* clock cycles are required for each multiplications. For a low area implementation, the digit size of *w* can be 2-bit, 4-bit to reduce latency. The most significant bit (MSB) first multiplier or least significant bit (LSB) first multiplier is mostly chosen due to interleave reduction and low area consumption. In the case of interleave reduction, the partial results are reduced on the fly using left to right (for MSB) reduction or right to left reduction method.

3.2.3 Low Area Square Circuit

In low area design, square operation can be considered a special multiplication. As multiplication consumes high latency, a standalone square circuit is generally utilised. A square operation with fixed irreducible polynomial, in particular, can be achieved in a single clock cycle by interleaving zeros followed by reducing operation. The reduction operation is also simple due to $a_i . a_j = a_j . a_i = 0$, where $i, j = 0,1,2 \ldots (m-1)$ and $a_i, a_j \in GF(2^m)$. The critical path delay of the single clocked cycles based square circuit depends on irreducible polynomials. The critical path delay for trinomial irreducible bas square circuit is 3Δ *xor,* whereas for pentanomial the path delay is 4Δ *xor.* Thus, the square circuits is suitable for the multiplicative inversion operation due to single clocked operation.

3.2.4 Inversion operation

The filed inversion operation is required in every iteration of affine coordinate based point multiplication. A standalone inversion operation the most costly operation in term of

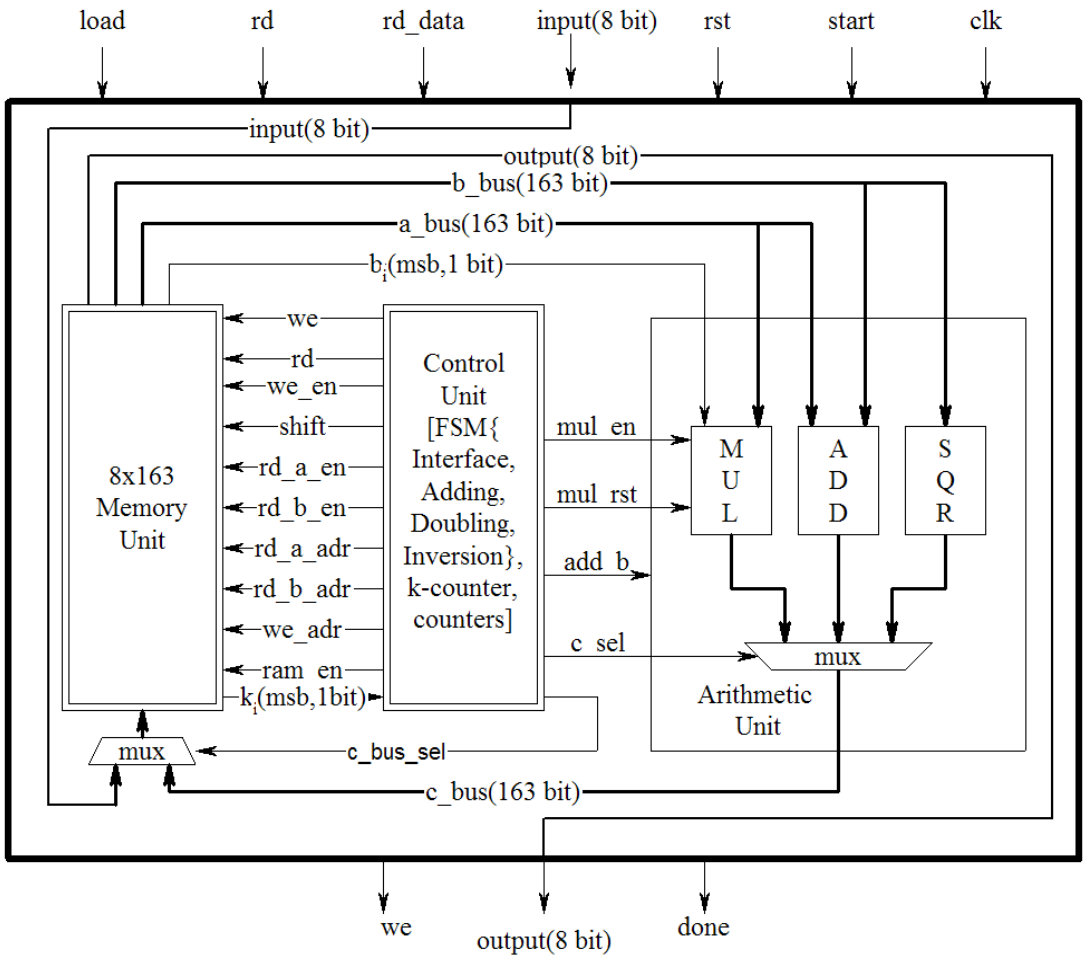


Figure 3.1 Low area ECC system architecture

area-time complexity. Projective coordinates based point operation can be considered to avoid the inversion. There is one inversion operation which is required during final coordinates conversion. The one inversion operation can be utilised multiplicative inversion operation. The multiplicative inversion operation can be achieved by using Fermat's little theorem (FLT) [1]. In FLT, repeated multiplication and squaring operation are performed for an inversion operation. Itoh and Tsujii proposed another version of FLT algorithm which is discussed in section 3.6. The multiplicative inversion which is friendly for low resource ECC design due to the operation is performed by utilising existing resources (multiplier and square circuits) and the latency for one inversion is very low as compared to the total latency of the ECC operation. There are $(m-1)$ number of square operations and total number of multiplications are calculated by $\lfloor \log_2 m - 1 \rfloor + h(m-1) - 1$, where $h(m-1)$ is the Hamming weight and $\lfloor \ \rfloor$ is floor function. There is some other inversion operation over *GF(2^m)* is considered in the literature based on Greatest Common Divisor Algorithm (GCDA) [69]. The GCDA based

inversion operation is not covered in the thesis as the method is not considered due to poor area-time performance.

3.2.5 Scalar point multiplication

The choice of point multiplication algorithm can affect the performance of the ECC. In this work, three popular point multiplication algorithms such as Lopez- Dahab (LD) Montgomery algorithm, binary algorithm and Frobenius map are investigated under projective coordinates.

3.3 Low Area ECC Implementation using Montgomery algorithm

The proposed ECC architecture is based on projective coordinates as shown in Figure 3.1. The design is aimed at exploiting identified concurrency in the point multiplication process, exploiting low resources arithmetic circuit and high performance memory unit to speed up the ECC while the ECC keeping resources low.

The Montgomery method can be one of the best options in the low area design space as only *x* coordinate is used in the point multiplication. The *x* coordinates based point operation can save memory requirement and. both point operations of the method are performed for every

Algorithm 3.1 Montgomery point multiplication (loop operation)

Input : $P(X_1, Z_1)$ and $Q(X_2, Z_2)$ where P and Q are points on ECC over $F_2{}^m$ [1]

Output : $Q(X_2, Z_2) = 2Q(X_2, Z_2)$ and $P(X_1, Z_1) = P(X_1, Z_1) + Q(X_2, Z_2)$.

Point Addition	Point Doubling
Step1: $X_1 \leftarrow X_2.Z_1$	Step1: $T_1 \leftarrow Z_2^2$
Step2: $T_1 \leftarrow X_1.Z_2$	Step2: $T_2 \leftarrow X_2^2$
Step3: $T_2 \leftarrow T_1.X_1$	Step3: $X_2 \leftarrow T_2^2$
Step4: $T_1 \leftarrow T_1 + X_1$	Step4: $Z_2 \leftarrow T_1.T_2$
Step5: $Z_1 \leftarrow T_1^2$	Step5: $T_2 \leftarrow T_1^2$
Step6: $X_1 \leftarrow x.Z_1$	Step6: $T_1 \leftarrow b.T_2$
Step7: $X_1 \leftarrow T_1 + X_1$	Step7: $X_2 \leftarrow T_1 + X_2$
Return(X_1, Z_1)	Return (X_2, Z_2)

bit of key that is partially resistance to power attack. Moreover, the algorithm offers parallel field operation that can save latency for point multiplication. A general version of Montgomery point multiplication algorithm is shown in the algorithm 3.1[68]. In the algorithm 3.1, the steps are serially performed In this work, steps of the Montgomery method are modified by extracting dependencies and enabling independent operations to be performed concurrently..

In the proposed ECC, the high latency field operation is multiplication as compared to the square and addition operation. Thus, addition and squaring operation can be performed concurrently by avoiding data dependency. This is achieved by our proposed unified point doubling and adding (DA) of the Montgomery algorithm as shown in algorithm 3.2 where 1DA, 2DA, 3DA, 4DA and 6DA are parallel operations that contribute to reduce 5 steps in the algorithm presented in the algorithm 3.1 [68]. The concurrency translates to reducing the latency of point multiplication. Again, the parallel operation reduces the number of stages in the finite state machine (FSM) based control unit. Hence, the control unit consumes reduce area and also increases speed by reducing the stages of FSM.

3.4 Low Area ECC Implementation using Binary Algorithm

In the binary method [1], point doubling is performed in every operation and an additional point addition is carried out depending on the *i*th bit of k, $k_i$being 1. The total time required for the point multiplication in (3) depends on the Hamming weight of k.

Algorithm 3.2 Combined doubling and adding operations of Montgomery algorithm

Input : $P(X_1, Z_1)$ and $Q(X_2, Z_2)$ where P and Q are points on ECC over $F_2{}^m$ [1]	
Output : $Q(X_2, Z_2) = 2Q(X_2, Z_2)$ and $P(X_1, Z_1) = P(X_1, Z_1) + Q(X_2, Z_2)$.	
Step1DA: $X_1 \leftarrow X_1.Z_2;\ Z_2 \leftarrow Z_2^2$	Step7DA: $Z_1 \leftarrow T^2$
Step2DA: $Z_1 \leftarrow X_2.Z_1;\ T \leftarrow Z_2^2$	Step8DA: $T \leftarrow x.Z_1$
Step3DA : $T \leftarrow b.T;\quad X_2 \leftarrow X_2^2$	Step9DA: $X_1 \leftarrow X_1 + T$
Step4DA : $Z_2 \leftarrow X_2.Z_2;\ X_2 \leftarrow X_2^2$	Step10DA: Return $P(X_1, Z_1)$ and $Q(X_2, Z_2)$.
Step5DA : $X_2 \leftarrow X_2 + T$	
Step6DA: $X_1 \leftarrow X_1.Z_1;\ T \leftarrow X_1 + Z_1$	

Algorithm 3.3 Modified LD mix-coordinates algorithm

Point addition:

Input $Q(x_q,y_q,z_q)$ and $P(x_p,y_p)$ where Q is in projective coordinate and P is initial point in affine coordinate on ECC curve over $F(2^m)$

Output: $Q(x_q,y_q,z_q)$ $=Q(x_q,y_q,z_q) + P(x_p,y_p)$

Step1: if Q = infinity then return (P)

Step2: If P= infinity then return $(x_p,y_p,1)$

Step3: $T_1 \leftarrow x_p z_q$; $T_2 \leftarrow z_q^2$

Step4 : $x_q \leftarrow x_q + T_1$

Step5 : $T_1 \leftarrow z_q x_q$

Step6 : $z_q \leftarrow T_2 y_p$

Step7 : $y_q \leftarrow y_q + z_q$

Step8 : $z_q \leftarrow T_1^2$

Step9 : $T_2 \leftarrow T_1 y_q$; $T_1 \leftarrow T_1 + T_2$

Step10: $x_q \leftarrow x_q^2$

Step11: $x_q \leftarrow T_1 x_q$; $y_q \leftarrow y_q^2$

Step12 : $x_q \leftarrow x_q + y_q$

Step13 : $x_q \leftarrow x_q + T_2$

Step14 : $T_1 \leftarrow x_p z_q$; $T_2 \leftarrow T_2 + z_q$

Step15 : $T_1 \leftarrow T_1 + x_q$

Step16: $y_q \leftarrow T_1T_2$; $T_1 \leftarrow z_q^2$

Step17 : $T_2 \leftarrow x_p + y_p$

Step18 : $T_1 \leftarrow T_1T_2$

Step19 : $y_q \leftarrow T_1 + y_q$

Step20 : return (x_q,y_q,z_q)

Point doubling:

Input : $P(x_p,y_p,z_p)$ where P is a point on ECC curve over $F(2^m)$

Output: $Q(x_q,y_q,z_q) = 2P$

Step1: if P = infinity then return(infinity)

Step2 : $T_1 \leftarrow z_p^2$

Step3 : $T_2 \leftarrow x_p^2$

Step4 : $z_q \leftarrow T_1 T_2$; $x_q \leftarrow T_2^2$

Step5 : $T_2 \leftarrow T_1^2$

Step6 : $T_1 \leftarrow y_p^2$

Step8 : $x_q \leftarrow x_q + T_2$

Step9 : $T_1 \leftarrow T_1 + z_q$

Step10 : $y_q \leftarrow T_2 z_q$, $T_1 \leftarrow T_1 + T_2$

Step11 : $T_2 \leftarrow x_q T_1$

Step12 : $y_q \leftarrow y_q + T_2$

Step13 : return $Q(x_q,y_q,z_q)$

We consider Lopez-Dahab (LD) mix coordinates based point addition and point doubling algorithms is shown in algorithm 3.3 [1]. Our architecture offers to exploit parallel operation in the following steps in the LD algorithms: steps 3 and 4, 11 and 12, 18 and 21, 22 and 20 in algorithm 3.25 presented in [1] and steps 4 and 5, 11 and 13 in the algorithm 3.24 presented.in [1] respectively are parallel operations as shown in the proposed algorithm 3.3.

3.5 Frobenius Map based Low Area ECC Implementation

Koblitz curves are special ECC curves that can be treated using the Frobenius map to reduce point multiplication time. In this work, Binary NAF point multiplication and computation of the τNAF (k) of an element in $Z[\tau]$ have been implemented simultaneously

Algorithm 3.4 Binary NAF based Frobenius map in the projective coordinates

Input : $Q(x,y,z) = \alpha$, $P(x,y,z) = P(P_x, P_y, 1)$ where $P(P_x, P_y, 1)$ is a initial point , $R_0 = k$, $R_1 = 0$, where $k = R_0 + \tau R_1 \in Z$, $n_i = \tau NAF(k)$;

Output: $Q = kP$;

While $R_0 \neq 0$ or $R_1 \neq 0$ loop

if $R_0(0) = 0$ then $n_i = 0$;

else if then $R_0(1) = R_1(0)$ then $n_i = 1$;	else $n_i = -1$;
if $Q(x,y,z) = \alpha$ (infinity) then	if $Q(x,y,z) = \alpha$(infinity) then
$Q(x,y,z) = P(x,y,z)$;	$Q(x,y,z) = P(x,x+y,z)$;
else	Else
$Q(x,y,z) = Q(x,y,z) + P(x,y,z)$;	$Q(x,y,z) = Q(x,y,z) + P(x,x+y,z)$;
end if;	end if;

end if;

$P(x,y,z) = P(x^2,y^2,z^2)$;

$R_0 = R_0 - n_i$;

$T = R_0/2$; $R_0 = R_0 + T$; $R_1 = -T$;

if ($R_0 = 0$ and $R_1 = 0$) then exit; end loop; Return: $Q(x,y,z)$;

which is shown in algorithm 3.4 [45]. In this work, binary nonadjacent form (NAF) point multiplication and computation of Frobenius map have been implemented simultaneously is shown in the algorithm 3.4 [44][45]. We design a very compact Frobenius mapping unit to generate the non-adjacent form to carry out point multiplication will be discuss in the next section.

3.6 Arithmetic circuit

A 163 bit data path based finite field arithmetic unit is designed with a multiplier, a squarer circuit, an adder circuit and a multiplexer. In our low area implementation, we consider a compact most significant multiplier called bit/digit serial multiplier [1, 46]. A most significant first based bit serial multiplier is shown in the Figure 3.2. We consider 2 bit and 4 bit multipliers (msd2 and msd4) [62] to increase performance while it is exploiting very small resources[57].

In our arithmetic unit, a dedicated squaring circuit is utilised which requires only one clock cycle is shown in Figure 3.3 [1, 46]. The field adder is a simple bit wise xor circuit [1, 46].

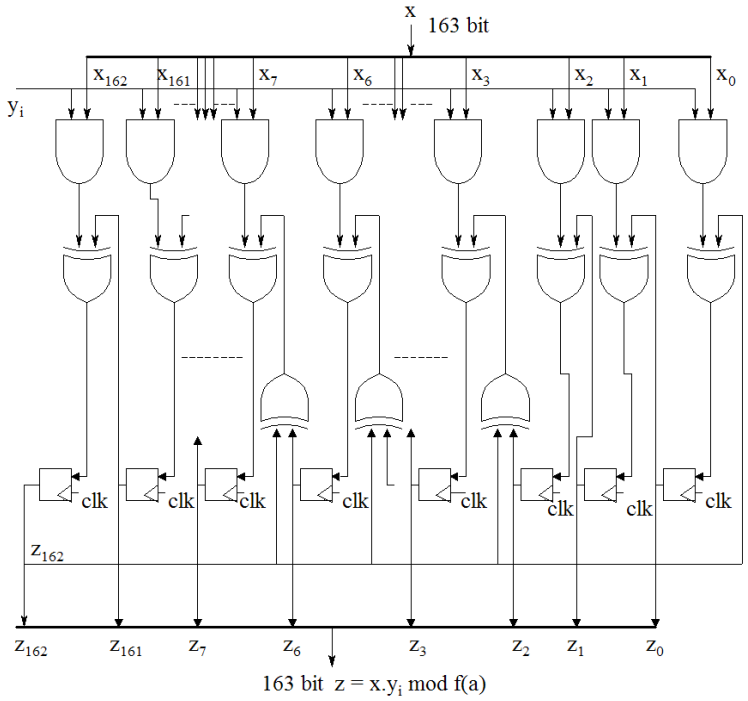


Figure 3.2 Finite field multiprecision multiplier over $GF(2^{163})$

Finally, an inversion operation is required to convert projective to affine coordinates. We use an efficient inversion algorithm proposed by Itoh and Tsujii is shown in the algorithm 3.5 [65].

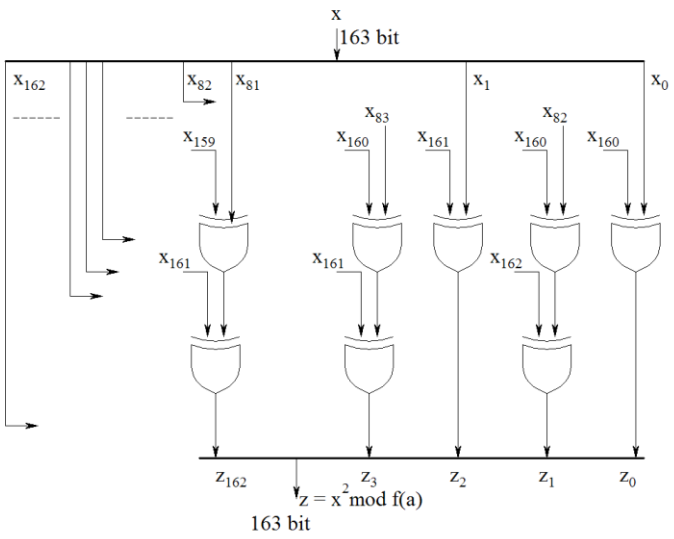


Figure 3.3 Finite field square circuit over $GF(2^{163}$

3.6.1 Frobenius mapping unit:

A compact architecture for the Frobenius mapping circuit is shown in Figure 3.4 In this circuit, we have considered a very low area integer addition (164 bit) circuit called ripple carry

adder. Integer subtraction is accomplished with two's complement. In order to do the two's complement of a particular data, we use modular adder circuit (i.e. xor circuit) for one's complement followed by adding 1 in the ripple carry adder. Thus, we do not use dedicated subtractor to save the area. Again, we use one location of our memory unit for R_1, and we use a dedicated 163 bit shift register for R_0 which is in the same data path of integer adder.

To control Frobenius point multiplication, we utilise the last two bits of R_0 which are considered *ki_1* and *ki_0,* and the last bit R_1 that is provided by the memory unit. The Frobenius point multiplication operation is performed along with mapping as shown in the algorithm 3.4. After each set of control unit bit generation, a loop operation of the point multiplication is accomplished. In the algorithm 3.4. the operation of $R_0=R_1-1$or $R_0=R_1+1$ or $R_0=R_1+0$ can be done by using b_bus for R_1 and by using *add_mod* signal to select 1 or -1 or 0. As a division over binary number is simple, we get $R_0/2$ by using the right shift of the shift register.

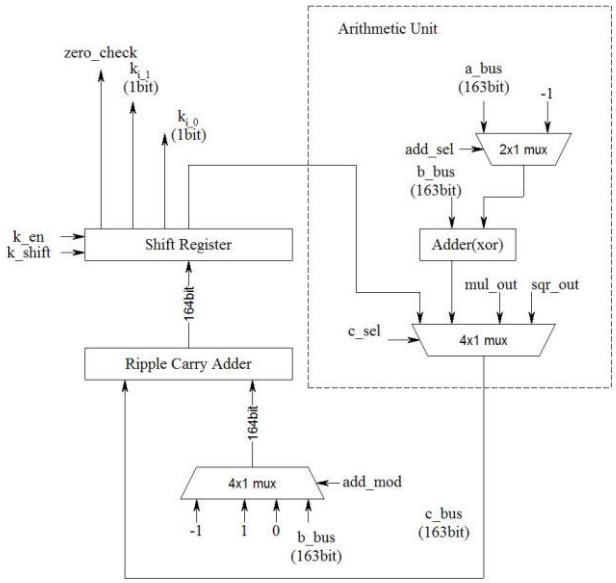


Figure 3.4 Frobenius mapping unit

In this mapping unit, we save area by utilizing serial data path; however, the serial operation takes some extra clock cycles. For example, to do $R_0=R_0+T$, any one of the input data is required to store in the shift register. After then, the b_bus of the memory unit and output of the shift register are used for integer addition followed by saving in the shift register. To check zero, the content of the shift register is checked for zero on the fly. If the content is zero, then the status flag will be 1. We store new value of R_0 and R_1 in the shift register to check $R_0 = 0$ and $R_1 = 0$ in serial.

3.7 Memory unit

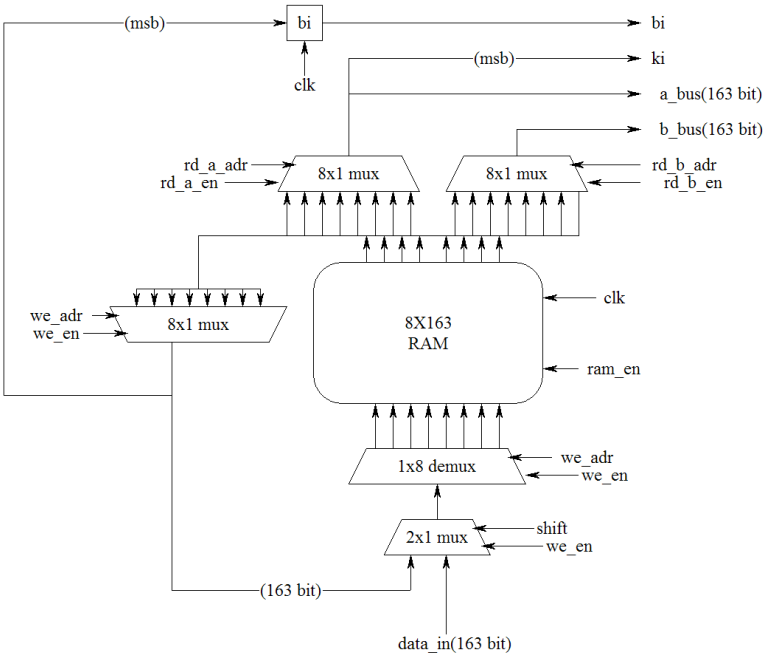


Figure 3.5 8xm memory unit

In this implementation, a generic behavioral 8x*m* memory unit is implemented with an independent one input (*c_bus*) and two outputs (*a_bus*, and *b_bus*) as shown in Figure 3.6. We include more flexibility in the memory than conventional block RAM. Specially, data can be written or shifted in any location (*we_adr*) using *we_en* and *shift* signals. The output data from any two addresses of *rd_a_adr* and *rd_b_adr* can be accessed using their respective enable signals *rd_a_en* and *rd_b_en*. The shifted outputs (*bi* and k_i) are single bit or digit depending on implementation (i.e. bit serial and digit serial). The *bi* and *a_bus* are inputs to the field multiplier. The unused *b_bus* can be used either to add with *a_bus* or to square during multiplication. Thus, concurrent operation can be utilized. Again, the k_i output is used to check the *i*th bit of the key. For the digit serial implementation, the k_i is digit. We save the digit in a digit size register to complete loop operation for each bit of the digit. To access output from the memory unit, the output from the memory is loaded in register and followed by delivering it to the output port. The register increases speed as it is used as a pipelining stage.

3.8 Interface unit

A compact 8 bit interface is adopted in our ECC as shown in Figure 3.6. In this interface circuit, we use shifting and addition using existing hardware i.e. the memory unit is used as registers as well as shifter, the modular addition circuit is used for bit wise xor and counter is used for 8 bit shifting. The two signals *we* and *rd* are used for handshaking for embedding with different speeds.

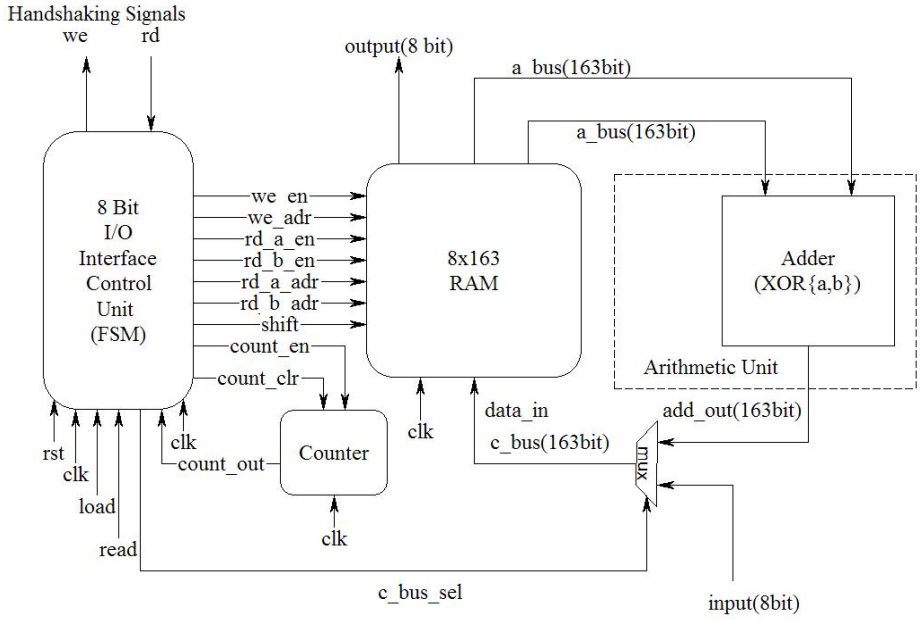


Figure 3.6 8-bit input/output interface

Initially, ECC sends *we = 1* to sender. If *we = 1* , then sender sends 1st 8 bit data, and also sets *rd* signal as high. If *rd* signal is high, ECC saves the data in the least 8 bits position of target location and also sends busy signal , *we= 0*, to sender. During the busy time, the 8 bits block is shifted left by 8 positions with leaving zeros in the least 8 bit position of the target location. If *we= 0*, then the sender changes *rd =0* as it has already sent 1st data, and it prepares next data to send. After arranging next data, the sender sets *rd=1* and waits for *we = 1*.

After saving first data, ECC again sends *we =1* and it waits for next data. If ECC gets *rd= 1*, then the next 8 bit block is stored in the least 8-bit positions of any one of the free memory locations (temporary location). Now, the new data of temporary location are added with data of target location using modular addition. In this addition, shifted 8 bits data of targeted location are added with zeros in the respective position of temporary locations; thus, we get 16 bit data in the target location. After addition, the target location again is shifted left for 8 bits. Again, new 8 bit data are stored in the temporary register to xor (addition) with

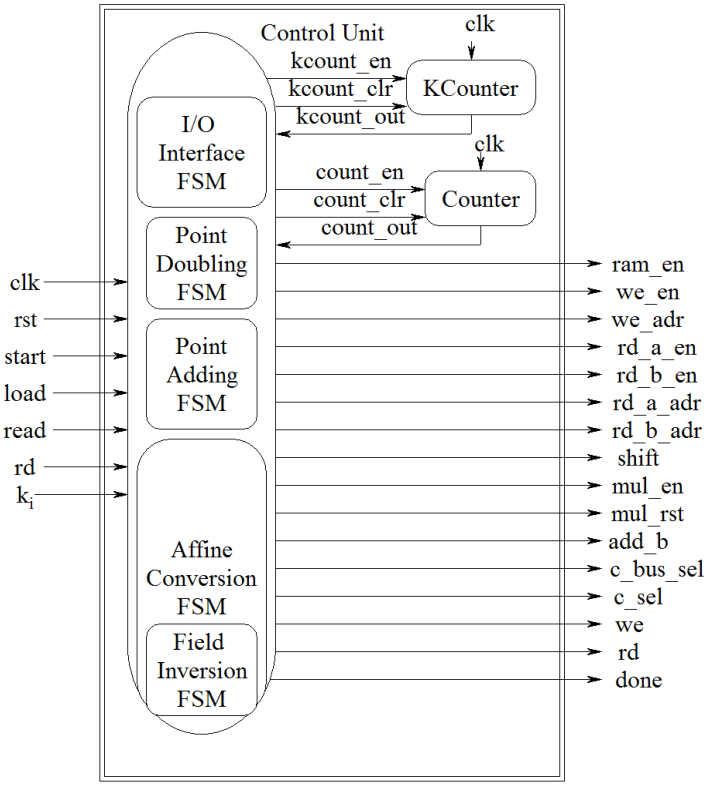


Figure 3.7 FSM based control unit

zeroes of the least 8 bits of target location. In this process, we can save *m* size data in the memory using very low cost interfacing circuit.

3.9 Control Unit

Finally, we design a control unit with a dedicated Finite State Machine (FSM), including all control signals in the same entity.In this architecture, a dedicated Finite State Machine (FSM) is used, as shown in Figure 3.7. The FSM is implemented based on Moore type machine. As soon as, start signal (*start*) and reset (*rst)* are initiated, control unit resets ECC, and it goes to idle state. If *load* =1, then control unit loads affine coordinates (*x* and *y*) and key, *k* with the help of address (*adr*) and handshaking signals (*we* and *rd*). After loading inputs, the control unit initiates projective coordinates to start the point multiplication. The loop operation is controlled by taking inputs k_i value (values for Frobenius) and key counter, K_counter outputs.

Algorithm 3.5 Itoh and Tsujii multiplicative inversion algorithm

Input : X ϵ F_2163 and irreducible polynomial f(α).
Output : Z ← X^{-1} mod f(α) = $x^{2**(m)\ -1}$ where m= 163 , [note :** =to the power]

Step 1: Z ← X^2	[1 square]
Step 2: T← Z.X	[1 Multiplication]
Step 3: Z ← T^{2**2}	[2 square]
Step 4: T ← Z.T	[1 multiplication]
Step 5: Z← T^2	[1 square]
Step 6: T← Z.X	[1 Multiplication]
Step 7: Z ← T^{2**5}	[5 square]
Step 8: T ← Z.T	[1 multiplication]
Step 9: Z ← T^{2**10}	[10 square]
Step 10: T ← Z.T	[1 multiplication]
Step 11: Z ← T^{2**20}	[20 square]
Step 12: T ← Z.T	[1 multiplication]
Step 13: Z ← T^{2**40}	[40 square]
Step 14: T ← Z.T	[1 multiplication]
Step 15: Z← T^2	[1 square]
Step 16: T← Z.X	[1 Multiplication]
Step 17: Z ← T^{2**81}	[81 square]
Step 18: T ← Z.T	[1 multiplication]
Step 19: Z← T^2	[1 square]
Step 20: Return Z	

After the loop operation, the point multiplication in projective coordinates is performed. The result of point multiplication in projective coordinates is converted to affine coordinates by using conversion step. The conversion involved with one multiplicative inversion. The control circuit implements multiplication algorithm proposed by Itoh-Tsujii algorithm is shown in algorithm 3.5. In this method, the total number of multiplications are calculated by $\lfloor \log_2 m - 1 \rfloor + h(m-1) - 1$, where $h(m-1)$ is the Hamming weight and $\lfloor \ \rfloor$ is a floor function. The total number of squaring operations is $(m-1)$. For *GF(2^{163})*, the number multiplications are $\lfloor \log_2 163 - 1 \rfloor + h(163-1) - 1 = 7 + 3 - 1 = 9$ and 162 square operations. At the end conversion, FSM generates a done signal to update the status of completing point multiplication and it holds itself in the idle state. If *read=1*, then the output control unit starts to deliver data from a specific location. The output operation is accomplished with the help of shifting and handshaking signals such as *we* and *rd*.

3.10 Latency of the proposed ECC operation

Latency of the proposed MSB based ECC for point multiplication is the total latency of the load operation, initialisation, point multiplication in the projective coordinates, and affine coordinates conversion. In the load operation, each word takes 8 clock cycles. The load

Table 3.1 Latency of MSB multiplier based ECC for Montgomery point multiplication

Load	8x(m/8)x3
Initialisation	10
Projective coordinates based point multiplication	5(m-1)(m+2) +10(m-1)
Projective to affine conversion	(12 + #mul for inversion)(m+2)
Output	8x(m/8)x2

#mul for inversion = $\lfloor \log_2 m - 1 \rfloor + h(m-1) - 1$, where $h(m-1)$ is the Hamming weight.

operation takes the equivalent latency to input of the three *m* bit inputs, and the output operation takes the latency for two *m* bit outputs. The initialisation takes 10 clock cycles. The bit serial multiplication takes *(m+2)* clock cycles. Each of the squares, addition and *k*-counter update operation takes 2 clock cycles. Thus, in each loop. The three operations (2 add and 1 square) takes 10 clock cycles. The conversion operation has 10 multiplications and 1 inversion operation apart from some other low cost operations (addition and square). The inversion has the multiplications of $\lfloor \log_2 m - 1 \rfloor + h(m-1) - 1$, where $h(m-1)$ is the Hamming weight with *(m-2)* square operations. The total latency for bit serial multiplier based ECC for the Montgomery point multiplication is shown in the Table 3.1.

3.11 FPGA Implementation Results

Our proposed ECC is implemented in a distributed RAM style mainly in the low cost family (XC3S200-5) of Spartan3 (S3). We also implemented the design in the Spartan6 (S6), an advance low cost technology to compare with relevant work. The results of our proposed ECC implementation are shown in Table 3.2, and Table 3.4 after it has been synthesized, mapped, placed and routed in different FPGA technology with successfully using Xilinx ISE 10.1 and 11.4 and 14.5. A worst case key (most hamming weight), k = 011…1 is used in the binary and k = 10…0 is used in the Frobenius version. Our emphasis is on an area optimised ECC as logic power, signal power and clock power depend on the area (slices) of an FPGA design [67]. We introduce the area^2-time metric to compare with any design (low area and high speed) as a fairer metric for the low area design space. An area^2-time metric indicates a prediction of performance while it is considering to implement in the comparable level (i.e. in the high speed level). A smaller value of the metric corresponds to a better performance.

In Table 3.2, our proposed Montgomery implementation outperforms in both area and time the binary and Frobenius based similar implementations. The implementations in Spartan6 shows excellent metrics results as that is a newer technology platform with 6 input LUT. Thus, the Spartan6 device consumes very low resources than that of previous Spartan family. In the case of bit serial implementations, our Montgomery and Binary implementation in Spartan3 consume 1111 slices and 1077 slices respectively, and their times for point multiplication are 1.94 *ms* and 0.99 *ms* respectively. Again, our Montgomery with msd4 in Spartan 3 and Spartan6 utilises only 1293 slices of area with 0.26 *ms* and 545 slices of area with 0.21 *ms* for point multiplication respectively. Our Frobenius Map based implementation takes fewer clock cycles

Table 3.2 Implementation results of this proposed low area ECC for point multiplication over $GF(2^{163})$ after place and route

After Place and Route (distributed ram)		**Frq (MHz)**	**Kilo clock Cycles**	***Kp* Time (ms)**	**Area-Time (Slice x sec)**	**Area^2-Time (Slice^2 x sec)/10^3**	**FPGA**
Slices	**LUTs**						
	Binary algorithm with Bit Serial mul (bsl)						
1077	1869	155	301	1.94	2.089	2.250	S3
	Binary Algorithm with msd2						
1165	2052	160	158	0.99	1.153	1.344	S3
	Montgomery algorithm with Bit serial mul (bsl)						
1111	1878	142	140	0.99	1.100	1.222	S3
	Montgomery algorithm with msd2 mul						
1237	2057	146	73	0.50	0.619	0.765	S3
	Montgomery algorithm with msd4 mul						
1293	2216	146	38	0.26	0.336	0.435	S3
545	1872	181		0.21	0.115	0.062	S6
	Frobenius map with bit serial mul						
1918	3540	60	88	1.39	2.666	5.113	S3
	Frobenius map with msd2 mul						
1918	3566	61	49	0.81	1.554	2.980	S3

LUTs-Look-Up Tables, mul- multiplier, *kp* - point multiplication, S3- Spartan3, S6-Spartan6

for point multiplications; however, the clock frequency is lower as we use a ripple carry adder. The Frobenius map with msd2 in Spartan 3 takes 1918 slices of area and 0.8 millisecond (*ms*).

3.11.1 Analysis of the results

Low power ECC is considered for the applications of a battery run low power device(such as wireless sensor nodes and RFID tags). A low area FPGA implementation can meet the constraint as the lower area design consumes low power. In particular, the power, including logic power, signal power and clock power depend on the area (slices) of an FPGA design [67].

To quantify the results, we have shown our results in terms of area-time and area^2-time metrics. The area-time metric is a standard metric that gives an idea about the optimisation of both area and time of a system. The area-time metric is suitable when a system performance is compared with the performance of a work of same space deign. For example, a low area design can fairly compare with a relevant low area design using the area-time metric. Crucially, the area-time metric of a low area design cannot be compared with the same metric of a high- speed design. Again the speed of a low area design can be increased by using a very few slices. To compare of the performances of the different space designs, the area^2-time metric is a farer metric. The area^2-time metric of a low area design can perform better when the low-area design will be implemented to upgrade in the high-speed design by considering large arithmetic circuit with small overhead of the area. Thus, Area^2-time metric is a suitable index to compare the performance of a low area with the performance of a high-speed design. To show fair comparison, we prefer to compare our results by using both area-time and area^2-time metrices of each of the low area and the high-speed design. The smaller the value of the metric translates the best performance.

Table 3.3 Performance of the proposed ECC for different digit-size multipliers over $GF(2^{163})$

Improvement from (for the case of Montgomery Algorithm)	**% of Improvement in Area x time**	**% of Improvement in Area^{2}x time**
Bit Serial to MSD2	(1.09 -0.62)/1.09=43%	(1.22-0.76)/1.22=37.7%
MSD2 to MSD4	(0.62-0.34)/0.62=45%	(0.76-.43)/0.76=43%
Overall Bit Serial to MSD4	(1.09-0.34)/1.09=68.8%	(1.22-0.43)/1.22=64.75%

The proposed digit serial based implementations show superior performance to bit serial implementations by using a small area overhead. In the case of Montgomery point multiplication, our proposed msd4 based ECC shows 68.8% of area-time and 64.75% of area^2-time metrices better than that of our bit serial implementation is shown in the Table 3.3. The performance of the digit serial implantation increases abruptly due to small area consumption is shown in the Figure 3.8.Thus, small digit serial multiplier improve performance abruptly without increasing significant area (slices). In the Figure 3.9, the area-time performance of the proposed ECC increases linearly due to decrease latency (clock cycles) of point multiplication with an increase of digit size without significantly affecting area and frequency of the proposed ECC.

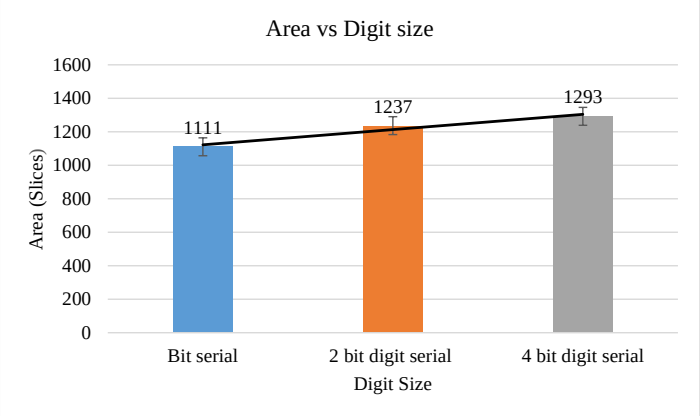


Figure 3.8 Area vs digit size over $GF(2^{163})$ in S3 for Montgomery method

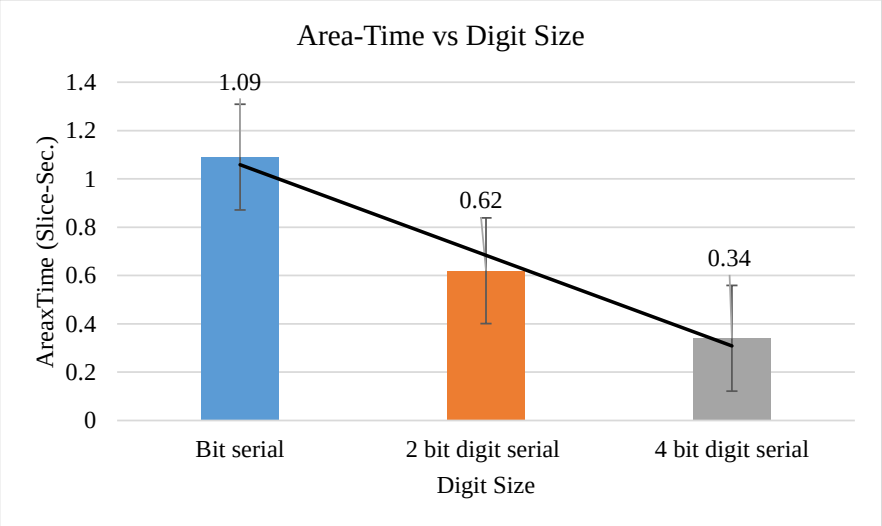


Figure 3.9 Area-time vs digit size over $GF(2^{163})$ in S3 for Montgomery method

The proposed ECC is investigated with three point multiplication algorithms. Each of the algorithms shows different advantages. Binary algorithm based ECC consumes the lowest area than others is shown in Figure 3.10. Again, In Figure 3.11, the Frobenius map based ECC shows very low latency. Notably, the Montgomery based ECC shows the best area- time metric than others algorithm is shown in the Figure 3.12. Thus, binary algorithm based ECC can be suitable for a very area-constraints application and Frobenius map based ECC can be fit for very low latency application and finally, Montgomery algorithm can be preferable for area-time optimised based applications.

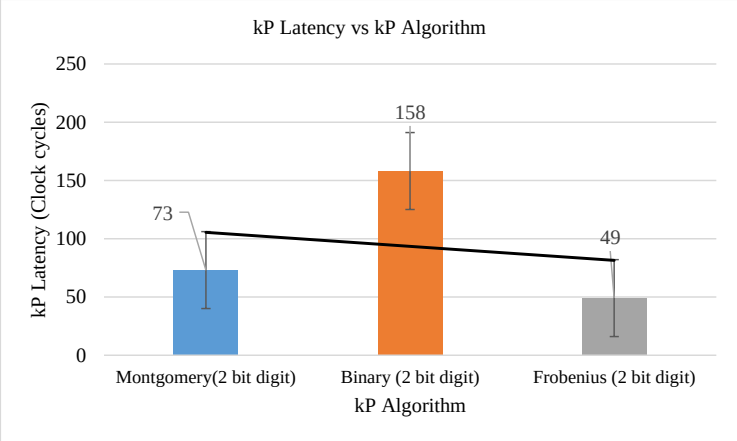


Figure 3.10 Latency vs *kP* algorithm (2-bit digit serial) over $GF(2^{163})$

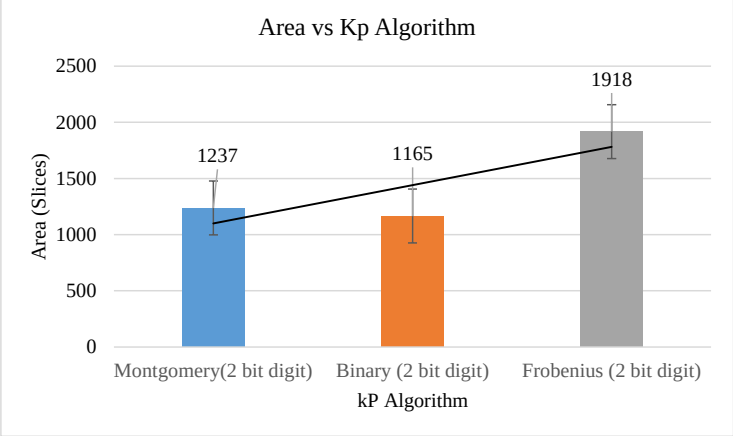


Figure 3.11 Area vs *kP* algorithm (2 bit digit serial) over $GF(2^{163})$

We have implemented MSB multiplier based ECC for Montgomery point multiplication over all NIST curves such as $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$ is shown the Table 3.4.The proposed ECC consumes a very low area to compute fast point multiplication. In general, the lowest field size shows lower complexity when it is comparable

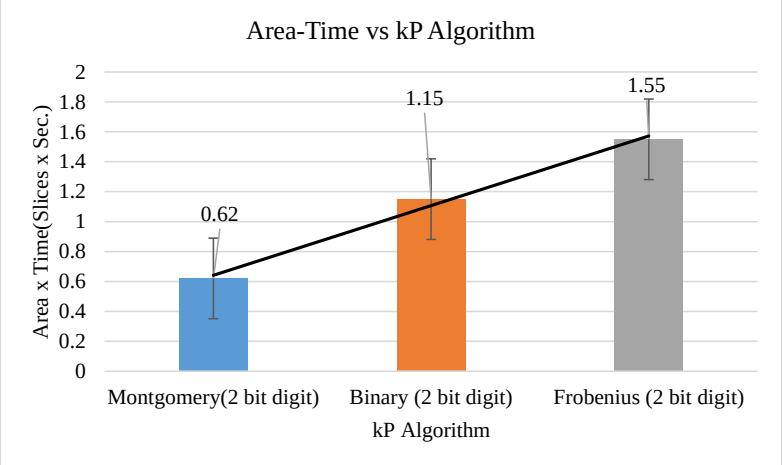


Figure 3.12 Area-time vs kP algorithm (2-bit digit serial) over $GF(2^{163})$

Table 3.4 Implementation results of the proposed bit-serial multiplier based ECC over all NIST curves after place and route

Field size, *m*	Slices	LUTs	FFs	Kilo Clock cycles	Freq., MHz	kP Time, ms	Area x time, Slices-sec	Area^2-Time (Slice^2-sec/10^3)
FPGA: Spartan3								
163	1111	1878	777	140	142	0.99	1.09	1.22
233	1463	2473	1095	281	126	2.23	3.26	4.77
283	1781	3217	1240	410	117	3.50	6.23	11.10
409	2366	4173	1677	852	114	7.45	17.63	41.71
571	3095	5863	2110	1656	107	15.42	47.73	147.71
FPGA: Spartan6								
571	3246	6013	6553	1656	132	12.60	40.90	132.76

LUTs-Look-Up Tables, FFs- Flip Flops, *kp* - point multiplication

to the high field size curve. The performance of all NIST curves is shown in the Figure 3.13. The area-time product increases abruptly as compared to the point multiplication time as shown in the Figure 3.13. As the latency increases exponentially with the increase of field size, the high field shows a poor area-time performances. For example, the point multiplication over $GF(2^{571})$ takes 15.42 *ms* with an area-time product of 47.73, and the point multiplication over $GF(2^{163})$ consumes 0.99 *ms* with 1.09 area-time product. The performance of the ECC can be improved by utilising very low cost digit serial multiplier.

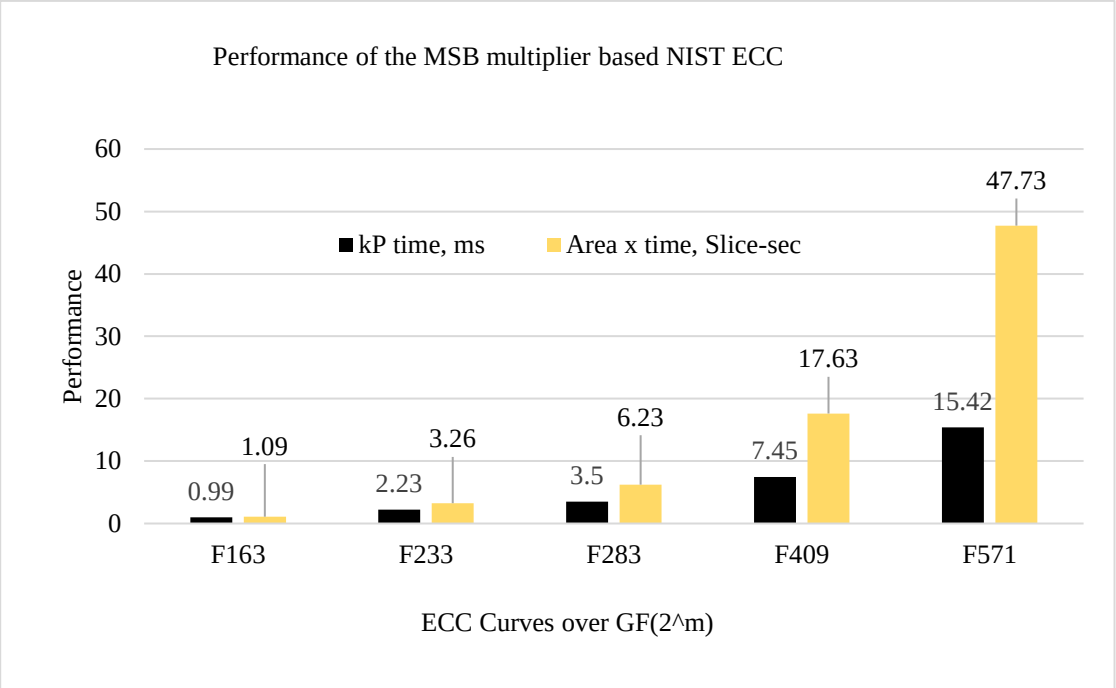


Figure 3.13 Performance of bit-serial multiplier based ECC over all NIST curves

We have reported FPGA based works to date irrespective of design choice, for example, low area and high speed hardware, hardware-software designs, and scalable and non-scalable as shown in Table 3.5. It is difficult to compare among the reported works as there are different design choices and optimisation goals. Again, some of the reported area (slices) does not reflect the actual area of the presented work as the block RAM equivalent slices are not included. The block RAM is used for the memory unit of ECC, but it consumes a major area (about 50% - 60% area depending on the architecture) of an ECC design. Again, some of the reported latency is not included the clock cycles for the input-output and conversion operations. For a fair comparison, the total clock cycles of our memory free design are included the clock cycles for point multiplication, input-output interface and projective to affine conversion.

The figures reported in [50], [51], [54], and [66] do not include the block ram equivalent slices. We compare our low cost Spartan FPGA results with relevant work implementation in the different technology to evaluate the merit of the work, however, the Virtex based ECC is considered for the high-speed design. In the case of low area designs, our ECC clearly outperforms in both area-time and area^2-time metrics the works reported in [45],[49]-[51],[55], and [63]. In the high speed design, our implementation demonstrates the best area^2-time metric. So far, our the area-time metric of msd4 based ECC implemented in the Spartan 3 shows even a better index than the high speed design in [37], [48],[54], [56-58], [62], [64], [66]. Again, the

Table 3.5 Comparison of the propsed ECC with the state of the art over $GF(2^{163})$ after place and route

Ref.	AL	FPGA	Area[slices+ BRAM (b)]	Freq(MHz)	Kp time (ms)	Area-Time(Slice-sec)	Area^2-Time (Slice^2-sec/10^3)
[56]	M	Ve	9,754[e]	66.5	0.1444	1.41	13.74
[57]	M	Ve	12,882[e]	68.9	0.0480	0.618	7.965
[37]	B	Ve	5,008	66	0.075	0.38	1.88
[58]	M	Ve	15,020	77	0.0367	0.55	8.26
[10]	M	Ve	15,368	91.1	0.0330	0.507	7.79
[61]	M	Ve	6432	124	0.0465	0.29	1.92
[66]	B	Ve	2856 +18b	44	0.964	2.753[b]	7.86[b]
[51]	M	Ve	1,101+7b	82	2.460	2.71[b]	2.98[b]
[55]	M	Ve	9,432	49	0.292	2.754	25.98
[49]	B	Ve	3,324	100	2.09	6.95	23.09
[59]	M	V2p	3,416	100	0.041	0.140	0.478
[47]	B	V2p	8,769	153	0.667	5.85	51.29
[50]	M	V2p	1,832+9b	108	29.83	54.65[b]	100[b]
[62]	B	V2p	4,749	100	0.488	2.32	11.00
[54]	M	V2p	8,954+6b	100	0.84	7.52[b]	67.35[b]
[48]	M	V2	18,079	90.2	0.106	1.92	34.65
[52]	M	V2	11,395	133	1.1	12.5	156
[59]	M	V4	4,080	197	0.0206	0.084	0.34
[60]	M	V4	20,807	185	0.0077	0.16	3.33
[49]	B	V4	3,528	100	1.07	3.78	13.32
[51]	M	V4	1,095+2b	150	1.345	1.473[b]	1.613[b]
[66]	B	V4	2,431+8b	155	0.273	0.664[b]	1.613[b]
[53]	M	V4	24,363	143	0.010	0.24	5.94
[10]	-	V4	16,209	154	0.020	0.32	5.14
[55]	M	V4	10,488	99	0.144	1.510	15.84
[61]	M	V5	2,448	357	0.0161	0.039	0.096
[64]	B	V5	1,368[e]	170	---	---	---
[55]	M	S3	10,379	44	0.325	3.37	35.01
[63]	M	S3	723[e]	76.4	17.2	12.44	8.99
[66]	B	S3	2,220+8b	93	0.456	1.012[b]	2.247[b]
[45]	M	S3	3,265	130	1.00	3.27	10.66
8bitsl	**M**	**S3**	**1,111**	**142**	**0.99**	**1.09**	**1.22**
8msd2	**M**	**S3**	**1,237**	**146**	**0.50**	**0.620**	**0.760**
8msd4	**M**	**S3**	**1,293**	**146**	**0.26**	**0.34**	**0.43**
8msd4	**M**	**S6**	**545**	**181**	**0.21**	**0.12**	**0.063**
[45]	B	S3	2,396	125	1.25	3.00	7.18
8bitsl	**B**	**S3**	**1,077**	**155**	**1.94**	**2.09**	**2.23**
8msd2	**B**	**S3**	**1,165**	**160**	**0.99**	**1.15**	**1.34**
[45]	F	S3	2,933	100	0.6	1.76	5.16
8bitsl	**F**	**S3**	**1,918**	**60**	**1.39**	**2.67**	**5.12**
8msd2	**F**	**S3**	**1,918**	**49**	**0.81**	**1.55**	**2.97**

[b] area of BRAM is not included in the area-time and area^2-time metrics, [e] equivalent area, S(3,6) - Sapartan (3,6),V(E,2,2P,4,5)- Virtex (E,2,2P,4,5), Al-Algorithm, Freq-Frequency, M- Montgomery, B- Binary, F-Frobenius, LUTs-Look-Up Tables, FFs- Flip Flops, *kP*- point multiplication

high speed design is generally implemented in the Virtex devices to achieve higher frequency than that of Spartan FPGA. Our proposed low area design is targeted for a low cost FPGA

platform such as Xilinx Spartan family. Again, the ECC in the Spartan 6 shows outperform in the both metrics. The Spartan 6 based ECC shows better area^2-time result than very high speed work presented in [10], [53], [59-61].

3.12 Conclusions

We have implemented Montgomery, binary and Frobenius Map algorithms to investigate area-time and area^2-time performance for ECC over all NIST curves on FPGA. Our Montgomery implementation shows a better area-time and area^2-time metric than that of relevant works reported to date. Specially, we have modified the steps involved in point addition and doubling algorithms as per data independency. Moreover, an area-speed trade-off arithmetic unit, asynchronous accessible memory unit, dedicated FSM based control unit enable it as an optimised ECC. In addition, our ECC offers more flexibility as it is independent of the FPGA fabric and incorporates a small hardware 8-bit I/O interface. We compare our results with relevant works on different Spartan FPGAs (Spartan 3 and Spartan 6). Our Montgomery implementation on Spartan 6 shows the best result achieving 0.21 *ms* for an ECC point multiplication with only 545 slices of the area. To our knowledge, the proposed architecture achieves the best area^2-time metric performance of the low area FPGA to date.

Chapter 4 Implementing High Throughput/Area Elliptic Curve Cryptography

High speed ECC consumes large hardware resources that prohibit its use in resource-constrained applications. Thus, high throughput while it is maintaining low resource is a key issue for Elliptic Curve Cryptography (ECC) hardware implementations in many applications. In this chapter, an ECC processor architecture over Galois Fields is presented that achieves the best reported throughput/area performance on FPGA to date. A novel segmented pipelining digit serial multiplier is developed to speed up ECC point multiplication. To achieve low latency, a new combined algorithm is developed for point addition and point doubling with careful scheduling. A compact and flexible distributed RAM based memory unit design is developed to increase speed while keeping area low. Further optimisations were made via timing constraints and logic level modifications at the implementation level.

4.1 Introduction

Public key based information security networks use cryptography algorithms such as Elliptic Curve Cryptography (ECC) and RSA. ECC has emerged recently as an attractive replacement to the established RSA due to its superior strength-per-bit and reduced cost for equivalent security [39].

High speed ECC is a requirement for matching real-time information security, however, in many applications the hardware resource implications may be prohibitive and the required high speed performance would need to be achieved within a restricted resource performance.

FPGA based Hardware acceleration of ECC has seen a surge of interest recently. There are several state of the art FPGA implementations aimed at the high speed end of the design space [10], [53], [59-61], [71, 72]. Most of these however use increased hardware resource to achieve the speed improvements sacrificing overall efficiency in terms of the throughput/area metric; such efficiency is desirable in many emerging low resource applications in particular in wireless communications. Area optimised high speed ECC design is challenging; there are requirements of algorithmic optimisation, careful scheduling to reduce clock cycles, size of multiplier, critical delay of the logic, and pipelining issues [10], [59].

In ECC, scalar point multiplication (PM) is the main operation. The PM can be implemented over either prime fields, $GF(p)$ or binary extension fields, $GF(2^m)$ adopting either projective coordinates or affine coordinates. Binary extension fields also called finite fields (FFs) are more suited to hardware implementation due to their lower complexity FF multipliers, simple FF adder and single clocked FF squaring circuits. Projective coordinates are suited to throughput/area efficient ECC designs, where the costly inversion operation is avoided and the inversion operation required to convert projective into affine coordinates can be achieved by multiplicative inversion [1], [65].

ECC computations in the projective coordinates system are based on large operand finite field operations of which multiplication is the most frequently performed. The high speed performance of ECC designs therefore would depend mainly on the performance of the FF multipliers. Digit serial FF multipliers are often used to reduce latency; popular multipliers here include the direct method based multipliers and Karatsuba [59], [10]. If the field size is m and the digit size is w of a digit serial multiplier, then the number of clock cycles for each FF multiplication is $s + c$, where $s = m/w$, and c is for clock cycles due to data read-write operations.

Algorithm 4.1 LD Montgomery point multiplication over *GF(2^m)*

INPUT: $k = (k_{t-1}, \dots, k_1, k_0)_2$ with $k_{t-1} = 1, P = (x, y) \in E(F_{2^m})$
OUTPUT: kp

Initial Step: $P(X_1, Z_1) \leftarrow (x, 1), 2P = Q(X_2, Z_2) \leftarrow (x^4 + b, x^2)$
For i from $t-2$ downto 0 do
If $k_i = 1$ then

Point addition: $P(X_1, Z_1) = P(X_1, Z_1) + Q(X_2, Z_2)$	Point Doubling: $Q(X_2, Z_2) = 2Q(X_2, Z_2)$
$Z_1 \leftarrow X_2 . Z_1$	$Z_2 \leftarrow Z_2^2$
$X_1 \leftarrow X_1 . Z_2$	$T \leftarrow Z_2^2$
$T \leftarrow X_1 + Z_1$	$T \leftarrow b . T$
$X_1 \leftarrow X_1 . Z_1$	$X_2 \leftarrow X_2^2$
$Z_1 \leftarrow T^2$	$Z_2 \leftarrow X_2 . Z_2$
$T \leftarrow x . Z_1$	$X_2 \leftarrow X_2^2$
$X_1 \leftarrow X_1 + T$	$X_2 \leftarrow X_2 + T$
Return $P(X_1, Z_1)$	Return $Q(X_2, Z_2)$

Conversion Step: $x_3 \leftarrow X_1/Z_1$; $y_3 \leftarrow \left(\frac{x+X_1}{Z_1}\right)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1 Z_2)](xZ_1 Z_2)^{-1} + y$.

Thus, large digit multipliers can reduce clock cycles (latency) with increasing complexities of area and critical path delay. The critical path delay can be reduced using pipelining with some extra latency [10].

In this chapter, we present an area-time (throughput/slice) efficient ECC processor over binary fields in projective coordinates on FPGA. We implement the Lopez-Dahab (LD) modified Montgomery algorithm for fast PM. We demonstrate a new "no idle cycle" [59] combined point operations (point addition and point doubling) algorithm to remove idle clock cycles in between two successive point operations. We schedule point operations very carefully to avoid the idle clock cycles due to data dependency, read-write operations, and pipelining. In addition, our efficient arithmetic circuit includes a digit serial multiplier, an adder and a square circuit. The presented arithmetic unit can support on- the-fly addition and square operations while it is performing FF multiplication. Moreover, we present an improved Most Significant Digit (MSD) serial multiplier utilizing segmented pipelining similar to the Least Significant Digit (LSD) multiplier presented in [1], [29]. We develop an optimized distributed RAM based memory unit for flexible data access to support reduced data dependency in the arithmetic operations. We adopt the Itoh-Tsujii inversion algorithm for inversion to save area [65], [70]. Finally, we use a dedicated finite state machine based control unit to speed up the control

operations. The proposed architecture is implemented on different FPGA technologies, Virtex4 (V4), Virtex5 (V5) and Virtex7 (V7), and compared to state of the art in terms of a throughput/slices metric. The throughput/area performance in ($1x10^6$/s)/(slices) of our proposed design (19.65 on V4, 65.30 onV5 and 64.70 on V7) outperforms state of the art designs on FPGA to date.

The rest of the chapter is organized as follows. Section 4.2 discusses preliminaries of PM, and the Lopez-Dahab modified Montgomery point multiplication in projective coordinates. Section 4.3 reviews resource constraints in high throughput ECC. Section 4.4 illustrates the proposed design. Section 4.5 presents the results of the FPGA implementation and a comparison with recently published state of the art designs on FPGAs, followed by conclusions in section 4.6.

4.2 Background

Elliptic Curve Cryptography over a binary extension field (2^m) is suitable for hardware implementation. The main operation of the ECC is scalar point multiplication $Q = k.P$, where k is a scalar (integer), P is a point on the elliptic curve, and Q is a new point of the curve after $k.P$ [2]. The Lopez-Dahab (LD) Montgomery algorithm is widely considered in the high performance ECC implementation as shown in algorithm 4.1[10], [53], [59-61], [71,72]. The algorithm has six field multiplication, three addition and five square operations that can be modified for parallel operation. A modified Montgomery point multiplication algorithm, as shown in algorithm 4.2 (page 4-8), has been adopted by many designs in the high performance ECC design [10], [53], [59-61], [71,72] space due to its speed, partial resistance of power attack(due to performing both point operations for every bit of *k*), suitability of parallelisation and low resource friendly.

To enable highly efficient ECC, the implementation requires an optimum balance among latency, frequency and area. The constraints of targeted ECC are low latency, high operating frequency (it is called maximum frequency on FPGA), and lower area. The constraints of the efficient ECC depend on the performance of the point multiplication scheduling and field multipliers. The Montgomery algorithm is widely considered for the friendly point multiplication scheduling. Again, the Montgomery point multiplication depends on the efficient operation of the field multiplier.

The field multiplier can be bit-parallel, digit serial and bit serial. The bit parallel multiplier has high complexity and poor maximum frequency, whereas bit serial multiplier consumes very high latency. Thus, digit serial multiplier is a popular option to design an efficient ECC for the point multiplication. In the literature, digit serial multiplication accomplished by using some multiplication algorithms such as Karatsuba algorithm [59][71] and schoolbook multiplication [61]. The multipliers targeted to reduce the latency and complexity of the multiplier for a high performance ECC architecture. Thus, the underlying success of the efficient design depends on highly efficient digit serial multiplier design. In this chapter, we demonstrate a high performance digit-serial multiplier that accelerates the Montgomery point multiplication algorithm.

4.3 Resource Constrained High Throughput ECC

For a high throughput ECC implementation in the low area end of the design space, there are requirements of optimization of the critical path of the logic, the area of the design, and number of clock cycles (latency) for the point multiplication. Throughput is usually improved via the adoption of large digit size multiplication and parallel operation of multiplications to decrease the latency. However, these steps result in an increased area and critical path delay and therefore affect the throughput per area metric figure. The critical path delay can be minimised via pipelining [10] at the expense of an increase in area and number of clock cycles with the number of pipeline stages inserted in the design. Also, the pipeline stages can generate idle cycles in the data dependable field operations [59]. The number of pipeline stages is an important consideration for area optimized high speed design often requiring a latency versus clock frequency trade-off. The latency due to pipelining can affect the merits of the use of a large digit size multiplier and the parallelisation of multiplication. In general, the area complexity of a high speed ECC design would depend on the digit size of the multipliers used and the level of parallelism adopted, on the size and sophistication of the memory unit, and on the control unit.

4.4 Proposed throughput/area Efficient ECC processor

Our proposed area optimized high throughput architecture is presented in Figure 4.1. The design consists of an efficient arithmetic unit, an optimised memory unit and a dedicated control unit.

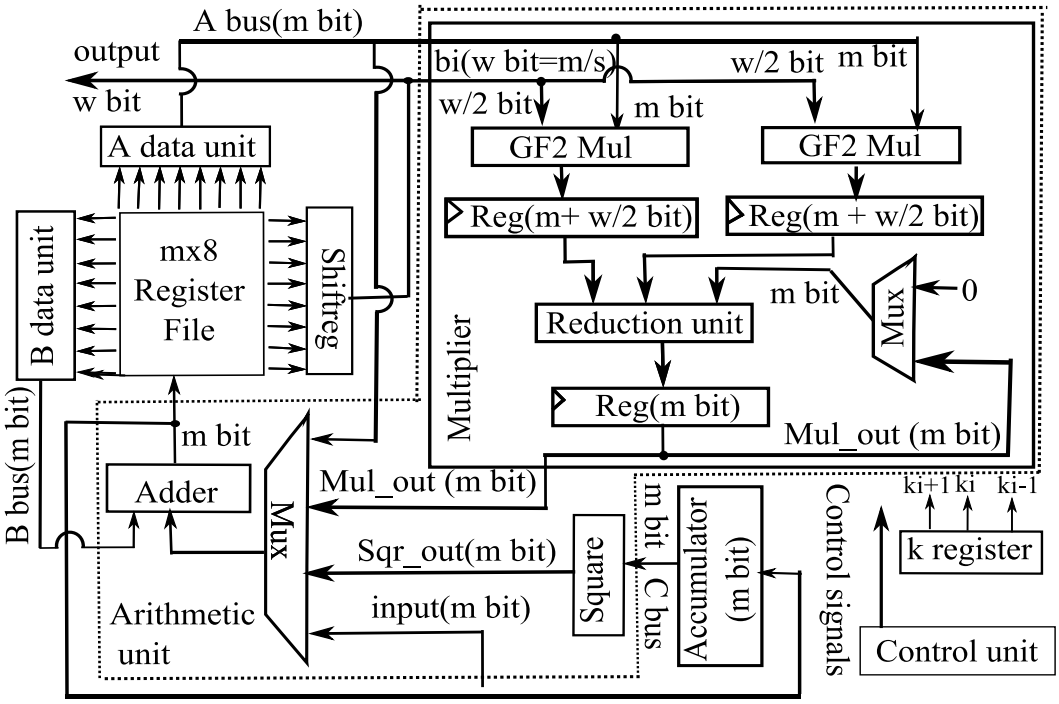


Figure 4.1 Proposed throughput/area efficient ECC architecture (for n=2)

4.4.1 Segmented Pipelining Based Digit Serial Multiplier

The arithmetic unit design consists of a novel most MSD serial multiplier, a square and adder circuit as shown in Figure 4.1.

The performance of ECC depends mainly on the performance of the Digit serial multiplier in particular the speed of the multiplier for a targeted level of latency. Digit serial multiplication for the high speed ECC implementation end tended to be either in direct form (i.e. MSD serial Multiplier) [61] or in bit parallel form (i.e. Karatsuba multiplier) [10], [71]. There are some advantages of Karatsuba multiplication over MSD multiplication. A Karatsuba FF multiplication takes s-1 cycles, where $s = m/w$, and is suitable for pipelining. An MSD FF multiplication takes $m + 1$ cycles where the extra clock cycle delay is due to the reduction register [1], [10] [61], and [70]. However, a pipelined Karatsuba multiplier based ECC implementation has been shown to achieve a lower clock frequency than a direct digit serial multiplier based implementation [59], [61], [71].

For large MSD digit serial based ECC, pipelining is required which can affect latency in the point multiplications. In this work, we apply segmented pipelining to improve performance in MSD multiplication. In the segmented pipelining approach, a w bit multiplier and m bit multiplicand based digit serial multiplication is broken into sub digit serial

Table 4.1 Latency, critical path delay and resources of digit serial multipliers over *GF(2^m)*

Ref	Latency, cc	Critical path delay	#XOR	#AND	#FFs	#Mux
[15]	$2C$	$(2 + \log_2 d)\, T_X$	$m + C(2d + S_1 + S_3) + d + S_2$	CS_3	$(C + 2)m + CS_3$	---
[16]	$m^{\log_4 2}$	$T_A + (2 + 3\log_4 m)\, T_X$	$\frac{69}{20}m^{\log_4 6} - 2m - \frac{1}{4}\, m^{\log_4 2} - \frac{1}{5} + 2m - 2$	$m^{\log_4 6}$	---	---
[17]	$2C$	$T_A + (\log_2 d)\, T_X$	$d(C(d + m) + 3)$	Cdm	---	---
Ours	$\left\lceil \frac{m}{d} \right\rceil$	$T_A + (\log_2(\frac{d}{n}))\, T_X$ or $T_{MUX} + (\log_2(n + k))\, T_X$	M part:dm +Rd part:($nm + kd$),M=*GF2MUL*, Rd=Reduction Unit, and k is the second higher order of irreducible polynomial	M part: dm	M part: $n(m+d-1)$+Rd part: m,	Rd part: m

$n = s =$ #segments, d= digit size, $S_1 = \left\lceil \frac{m}{d} \right\rceil \left(2.5\, d^{\log_2 3} - 3d + 0.5\right) + d^{\log_2 3} - d$, $S_2 = \left\lceil \frac{m}{d} \right\rceil \left(2\, d^{\log_2 3} - 2d\right)$, $S_3 = \left\lceil \frac{m}{d} \right\rceil \left(2\, d^{\log_2 3}\right)$, $C = \left\lfloor \sqrt{\frac{m}{d}} \right\rfloor$. $T_{MUX} = 2x1$ Mux delay. FFs: Flip-Flops

multiplications called segmented multiplications w_1xm, w_2xm,...,w_nxm, where w=w_1+w_2+ …+ w_n. The segmented multiplication product is first saved in the register (*Reg*) before reduction into *m* bits using an interleaved reduction similar to that in the bit serial multiplier in [1]. The reduced *m* bit output of the reduction unit is saved in another *Reg* to use in the next cycles reduction or output. Thus, the proposed multiplication takes *s* +2 clock cycles where 1 extra clock cycle is due to the segmented pipelining, and the other additional clock cycle for pipelining after the reduction unit. A new input of the multiplier is inputted in every *s* clock cycles. Thus, a real time reset is required in every *s* cycles. We use multiplexers to select zero for reset and save one clock cycle for the FF multiplication. Finally, the segmented pipelined multiplier takes one clock cycle for *n* segmentations without increasing area (slices) on the FPGA. The unused flip-flops (FFs) in the combinational circuit of the multiplier are utilized in the pipelining [71].

To evaluate our proposed segmented multiplier, resources and latency complexity analysis is performed and presented in Table 4.1 which also includes comparison to state of the art digit serial multipliers reported in [74-76]. For $s = 4$ or less, our proposed multiplier shows same or better latency using similar or less resources. However, a key advantage of our proposed architecture is that we are able to achieve higher speed for the same (or less) area and the same (or less) latency; this is because the number of segmentations (*n*) with extra FFs can modulate the critical path delay. The value of *n* defines the critical path delay of the multiplier. The path delay is either $T_A + (\log_2(\frac{d}{n}))\, T_X$ for the multiplication part (M) over *GF(2^m)*

Algorithm 4.2 Proposed combined loop operation of the LD Montgomery point multiplication with careful scheduling

For i from $t-2$ down to 0 do If $k_i = 1$ then	
If $k_{i-1} = 1$ then	If $k_{i-1} = 0$ then
Point addition:$P(X_1,Z_1) = P(X_1,Z_1) + Q(X_2,Z_2)$ and Point Doubling: $Q(X_2,Z_2) = 2Q(X_2,Z_2)$.	
St1: $Z_1 \leftarrow X_2.Z_1$.	St1: $Z_2 \leftarrow X_1.Z_2, Z_2 \leftarrow Z_2^2, T \leftarrow Z_2^4$.
St2: $X_1 \leftarrow X_1.Z_2, Z_2 \leftarrow Z_2^2, T \leftarrow Z_2^4$.	St2: $X_2 \leftarrow X_2.Z_1$.
St3: $X_2 \leftarrow b.T + X_2^4, X_2 \leftarrow X_2^2$. St4: $Z_2 \leftarrow X_2.Z_2$.	
St5: $X_1 \leftarrow X_1.Z_1,\ T \leftarrow (X_1 + Z_1)^2, Z_1 \leftarrow T$. St6: $X_1 \leftarrow x.T + X_1$.	
Conversion Step: same as Algorithm 1.	

(*GF2MUL)* or $T_{MUX} + (\log_2(n+k))\ T_X$ for the reduction part (Rd). Thus, our critical path delay can be optimised (to achieve the desirable high speed) by choosing an optimum number of segmentations (*n*). To generalise, from Table 4.1, the best figure latency for a field multiplication [74, 76] is $2\left\lceil\sqrt{\frac{m}{d}}\right\rceil$ where ⌈ ⌉ is a floor function, our multiplier's latency is $\left\lceil\frac{m}{d}\right\rceil$. As a rule of thumb, therefore as long as *m<4d,* our multiplier would achieve comparable or better latency figure. But, what is crucial is that for comparable (less or higher) latency say and same digit size, our design can achieves improved critical path delay $T_A + (\log_2(\frac{d}{n}))\ T_X$ in our case (due to the critical path delay of *GF2MUL*) compared to $T_A + (\log_2 d)\ T_X$ in [74, 76] using an optimum segment size without increasing the latency of the multiplier. Thus, utilising similar area, our multiplier can achieve higher speed. At the extreme, the use a full precision multiplier ($d = m$) with an optimised segmentation would thus lead to the highest speed.

4.4.2 Optimized Memory Unit

High speed and flexible design for the memory unit can improve performance. We consider an optimised distributed RAM based memory unit. There is an 8x*m* size register file in a unit, one *m* bit register (accumulator) and one shift register (Shiftreg). The 8x*m* register file consists of one *m* bit input that can load data in any location of the register file, two *m* bit output buses (A_bus and B_bus) that can access data from any location of the register file. The shift register can store data from any location of the register file to provide *w* size digit (*bi*)

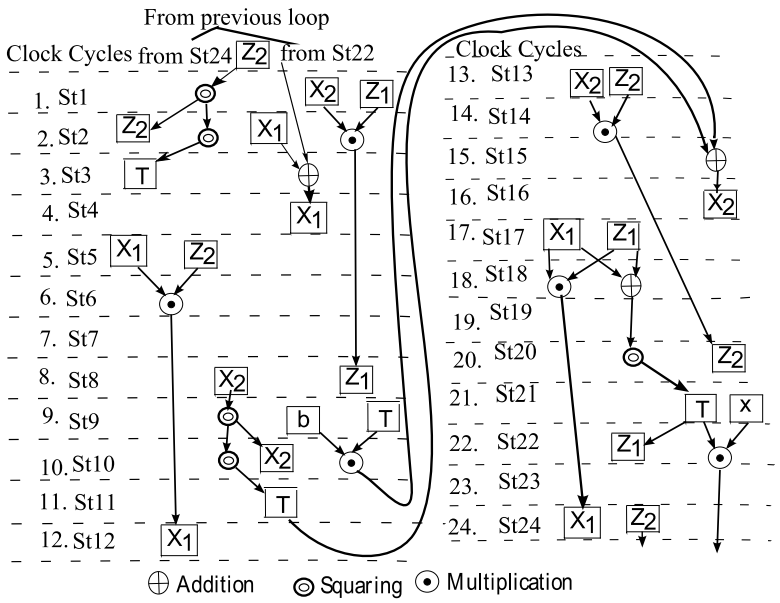


Figure 4.2 Proposed careful scheduling (4 clock cycles/multiplication)

multiplier for the FF multiplication. The accumulator can save a result from the arithmetic unit or new data from the register file to do a square operation. The accumulator and square circuit are connected such that repeated squaring can be done without saving in the register file. The repeated squaring improves latency of multiplicative inversion as proposed in [65]. The memory unit is smartly accessible to write, read shifting operation in any location. The easy accessibility of the memory reduces the number of temporary registers for the PM. The memory unit consumes very low area to provide high-speed data access.

4.4.3 Scheduling for point operations

In this section, we propose a new scheduling in the combined LD Montgomery point multiplication as shown in algorithm 4.2.

To schedule for no idle cycles, we combine the point addition and point doubling algorithms for the current value of $k_i = 1$ as shown in algorithm 4.2. We observe that the product of the last multiplication is X_1 if $k_i = 1$ or X_2 if $k_i = 0$. Thus, the first multiplication of the loop should be independent of the last multiplication. For example, if the last product is X_1, then the next operands of multiplication are X_2 and Z_1. Otherwise, the next operands will be X_1 and Z_2. Thus, the first multiplication depends on the last k_i which means the k_{i+1} bit as shown in algorithm 4.2.

Figure 4.2 illustrate the proposed no idle state schedule using a 41-bit digit size FF multiplier. The 41 bit digit size FF multiplier takes $M = 4$ cycles for actual multiplication, and $c = 4$, with 2 clock cycles for pipelining and 2 clock cycles for unloading from and loading to the memory unit. In a loop, the point operation in the projective coordinates system requires 6 multiplications. To ensure no idle state in the multiplication, a new multiplication is started every 4 clock cycles. Thus, two consequent but independent multiplications are overlapping each other as shown in Figure 4.2 for $k_i = 1$ and $k_{i-1} = 1$.

Table 4.2 Latency of ECC for $\lceil m/w \rceil = 4$, mul= M_4/M_7. add=1. sqr=2

Algorithm	Initial + point operations + Conversion	GF(2163)
[2]	$5 + (6M_7 + 13)(m-1) + (10M_7 + \text{Inv})$	9211
Algorithm2	$5 + (6M_4(m-1) + (7M_4 + 3M_7 + \text{Inv})$	4168

$M_4 = 4$, $M_7 = 7$, Inversion (Inv) = (#Mul for Inversion $\times M_7 + m$)

Again, the adder circuit placed in the common data path is capable of doing addition concurrently. The square operation takes three cycles with 1 cycle to save in the accumulator, 1 clock cycle for squaring, and 1 clock cycle for loading. Repeated squaring can be done without storing in the register file. Thus, double squaring takes 4 clock cycles. Total Latency of the ECC is shown in the Table 4.2.

4.5 Implementation on FPGA and Results

Our proposed efficient ECC processor is implemented over $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$, on different FPGA technologies namely Virtex4 (LX25_12 for f163, and LX100_12 for f233 to f571), Virtex5 (XC5VLX50_3 for f163), and Virtex7 (Vx550T_3 for f163, and V585_T for f233 to f571) using Xilinx tools versions 13.2 and 14.5 respectively. The design was implemented on Virtex4 and Virtex5 technologies to allow for a fair comparison to most relevant works, and on the Virtex7 to evaluate the performance on the newer technology. Where feasible the designs have been implemented in each Virtex family. The FPGA size selected was the smallest in the family that could accommodate the design in terms of area and pin count. We present the implementation results after place and route in Table 4.3 and Table 4.4. The Xilinx tools were used to set high-speed properties and put

Table 4.3 Results of our ECC over $GF(2^{163})$ after place and route

segmentation	Slices (Sls)	LUTs	FFs	Max. Freq., MHz	Total *kP* Time, µs	(10^6/s) /Sls
In virtex4						
No.Seg (41x1)	3623	6793	1348	252	16.51	16.71
2 Seg.(21x2)	3444	6516	1701	276	15.08	19.25
3 Seg.(14x3)	3536	6672	1870	290	14.39	19.65
In virtex5						
No.Seg.(41x1)	1150	3960	1146	286	14.58	59.66
2 Seg. (21x2)	1089	3958	1522	296	14.06	65.30
3 Seg. (14x3)	1185	4027	1701	301	13.85	60.95
In Vertex7						
No.Seg.(41x1)	1341	4406	1300	361	11.56	64.52
2 Seg. (21x2)	1351	4513	1703	364	11.44	64.70
3 Seg. (14x3)	1476	4721	1886	397	10.51	64.48

LUTs-Look-Up Tables, FFs- Flip Flops, *kP*- point multiplication

subsequent timing constraints to improve the area-time product. Table 4.5 also includes area-time performance and comparison to state of the art.

As shown in Table 4.3, the main contribution of the segmentation in the multiplier is an increase in the clock frequency while it is utilizing very small resources (FFs). The clock frequency for 3 segmented (3 Seg.) pipelined multipliers based ECC design is 290 MHz on the Virtex4- that is 38 MHz- more than the respective implementation of non-segmented (No Seg.) multiplier based ECC. Again, the 2 segmented (2 Seg.) pipelined multiplier based ECC shows the best throughput per slice (65.30) is implemented on Virtex5 and the 3 segmented multiplier based ECC on Virtex7 shows the highest performance (only 10.51 µs for an ECC point multiplication). The optimum size of the segments is subject to a trial-error method to achieve high throughput.

Table 4.4 FPGA implementation results after place and route in Virtex7

m(segments size)	Slices (Sls)	LUTs	FFs	Fq.,M Hz	*kP* Time, µs	(10^6/s) /Sls
163(3x14)	1476	4721	1886	397	10.51	65
233(4x14+3)	2647	7895	2832	370	16.01	24
283(5x14+1)	3728	11593	3973	345	20.96	13
409(7x14+5)	6888	20881	6038	316	32.72	4.4
571(10x14+1)	12965	38547	10066	250	57.61	1.3

LUTs-Look-Up Tables, FFs- Flip Flops, *kP*- point multiplication

4.5.1 Analysis of the Results

We present our results of ECC for different segmentations such as non-segmented pipelining multiplier, two-segmented pipelining multiplier and three-segmented pipelining multiplier on the FPGAs as shown in the Table 4.3 after place and route. The segmentation pipelining technique offers a resource free improvement of the performance of multiplier. The main contribution of the segmentation in the multiplier is an increase in the clock frequency by utilizing very small resource (mainly Flip-flops). The clock frequencies for three-segmented pipelined multiplier based ECC are 290 MHz, 301 MHz, and 397 MHz in the Virtex4, Virtex5, and Virtex7 respectively is shown in the Figure 4.3. The frequency of the segmented pipelining based ECC increases by 38 MHz, 15 MHz and 36 MHz more than the respective frequency of non-segmented multiplier based ECC. The critical path for the case 41 bit digit without segmentation has long critical path delay than same digit with segmentation. In the Figure 4.4,

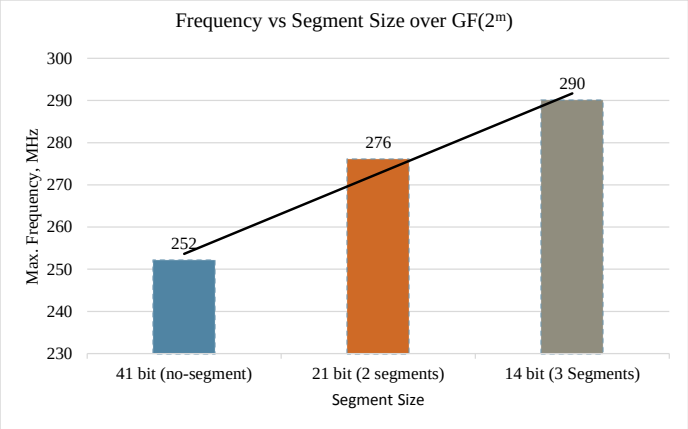


Figure 4.3 Frequency vs segment size of the ECC over $GF(2^{163})$

the increase in the segmentation consumes low slices as the pipelining stage consumes the unused flip-flops in the combinational circuit based multiplier. The tool can able to optimize the area of the two- segment based ECC than that of the others ECC as shown in the Figure 4.4. Thus, the optimum number of segmentation is a matter of trial and error method. For example, the two-segmented pipelined multiplier based ECC shows the best throughput per slices (65.30 as shown in Table 4.3) that is implemented in Virtex5. The performance of the proposed ECC increases with the increase of the number segmentation is shown in the Figure 4.5. In particular, the three-segmented multiplier based ECC in Virtex7 shows high performance (10.51 µs for point multiplication as shown in the Table 4.3) thank to the new

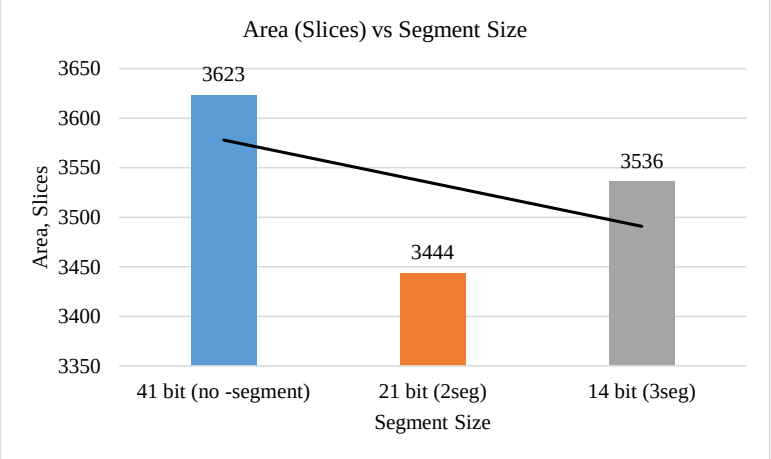


Figure 4.4 Area vs segment size of the ECC over $GF(2^{163})$

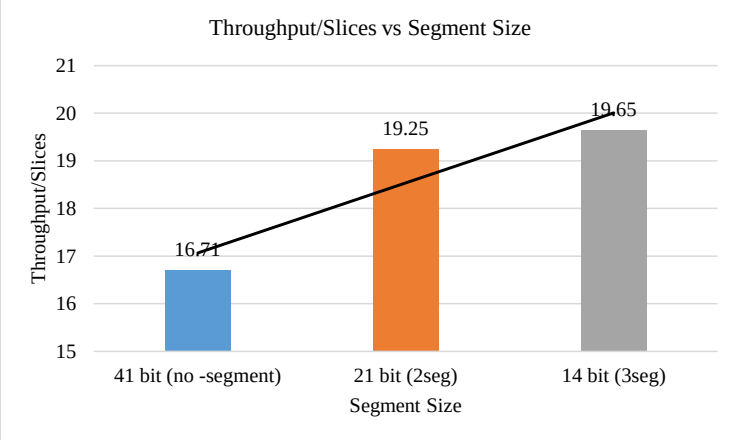


Figure 4.5 Throughput/slices vs segment size of the ECC over $GF(2^{163})$

technology.

Table 4.5 shows comparisons with the relevant high performance ECC designs on FPGAs in term of efficiency metric throughput/area ($1x10^6$/s)/slices (that is a typical area-time

Table 4.5 Comparison of state of the art after place and route on FPGA

Ref. (n, FPGA)	Slices (Sls)	LUTs, FFs	Clk Cs.,Fq. (MHz)	kP time (µs)	(10^6/s) /Sls
GF163					
[7](V4)	4080	7719, 1502	4050, 197	20.56	11.92
[8](V4)	8095	14507,-	1414, 131	10.70	11.55
[9](V4)	16209	26364,7962	3010, 154	19.55	3.16
[11](V4)	20807	-, -	1428, 185	7.72	6.23
[12](V4)	24363	-, -	1446, 143	10.00	4.11
[13](V4)	14203	26557, -	3404, 263	11.60	6.07
[14](V4)	12834	22815, 6683	3379, 196	17.20	4.53
Ours(no, V4)	3623	6793,1348	4168, 252	16.51	16.71
Ours(2, V4)	3444	6516,1701	4168, 276	15.08	19.25
Ours(3, V4)	3536	6672, 1870	4168, 290	14.39	19.65
[10] (V5)	6150	22936, -	1371, 250	5.48	29.67
[8] (V5)	3513	10195, -	1414, 147	9.50	29.96
[14] (V5)	6536	17305, 4075	3379, 262	12.90	11.86
Ours(2, V5)	1089	3958, 1522	4168, 296	14.06	65.30
Ours(3, V7)	1476	4721, 1886	4168, 397	10.51	64.48
GF571					
[7](V4)	34892	66594,6445	14250,107	133	0.22
[10] (V5)	11640	324332,-	44047,127	348	0.25
Ours(11, V4)	35195	61673,10692	14396, 180	79.80	0.36

LUTs: Look_Up_Tables, FFs: Flip-Flops, *kp*: point multiplication

efficiency) over $GF(2^{163})$ and $GF(2^{571})$. Notably, area^{2}xtime metric is not considered as only high speed state-of-art are considered to compare. For $GF(2^{163})$, the previous best optimised work was reported in [59] where one 41bit pseudo-pipelined Karatsuba multiplier was used with a so called "no-idle cycles" point multiplication approach to achieve 11.92 throughput/area figure on Virtex4. Our no-segment based ECC design consumes less area

(3623 slices) and achieves higher clock frequency (252 MHz) than [59] (4080 slices, 197 MHz) and therefore has a 40% higher throughput/area efficiency.

Particularly, our three-segmented based design shows 65% better efficiency than the efficiency of [59] as shown in Table 4.5. Our f571 achieves 180 MHz speed, whereas the work in [59] operates at a max speed of 107 MHz. One potential option of improving the area performance of [59] is to deploy an area efficient Karatsuba multiplier [75]; however, this would be at the expense of increased critical path delay. Another optimized ECC in [71] used full length (164 bit) word serial Karatsuba multiplier with pipelining and implemented on Virtex4 and Virtex5. The work in [71] uses four times bigger multiplier than ours to achieve 11.55 and 29.96 throughput/area on Virtex4 and Virtex5, respectively. Our 3 segmented 41 bit multiplier based design on virtex4 is 70% and the 2 segmented 41 bit multiplier based design on Virtex5 is 118% better than [71]. In [61], the reported best throughput/area efficiency is based on three 33 bit multipliers based ECC on Virtex5 shows 9.86 in throughput/LUTs ($(1x10^6/s)$/LUTs). Our 2 segmented multiplier based ECC shows 17.9 in $(1x10^6/s)$/LUTs is 82% better than the reported most efficient design in [61]. The hardware results presented in [53], [60], [72], and [73] use parallel multipliers to speed up their ECC designs show poor throughput/area efficiency due to the large area consumed. Finally, our single multiplier (41 bit) based ECC implementation on Virtex7 takes 10.51 µs for point multiplication is faster than the reported high speed work in [10], [59], [72], [73], and the work on the Virtex4 reported in [71], and is comparable to the work in [53] while of course using much lower resources.

4.6 Conclusion

We propose a highly efficient FPGA ECC processor design for high speed applications over $GF(2^m)$. Key contributions include a novel high performance segmented pipelining MSD multiplication, a smart no-idle state scheduling that enables the clock cycles for loop operations in the point multiplication to depend only on the actual clock cycles of the FF multiplications, and a highly optimized memory unit design.

To our knowledge, our design achieves the best throughput/area efficiency figure on FPGA reported to date. The best throughput/area design achieved a figure of 65.30 $(1x10^6/s)$/(slices) that is performing an ECC point multiplication in 14.06 µs time whilst utilising only 1089 slices of area. The fastest design achieved is 10.51 µs for a point multiplication using only 1476 slices.

Chapter 5 Implementing High Speed Elliptic Curve Cryptography

High speed hardware implementation of Elliptic Curve cryptography (ECC) is vital for cryptography system design. In this chapter, a novel high speed ECC implementation for point multiplication on Field Programmable Gate Array (FPGA) is proposed. A new segmented pipelined full-precision multiplier is used to reduce the latency and the Lopez-Dahab (LD) Montgomery point multiplication algorithm is modified for careful scheduling to avoid data dependency resulting in a drastic reduction in the number of clock cycles required. The area-time performance of the full-precision multiplier based ECC is evaluated for different combinations of parallelization of the multiplier. The chapter presents novel high speed of ECC, targeting server end applications where speed is a prime concern.

5.1 Introduction

Elliptic curve cryptography (ECC) was proposed by Koblitz [17] and Miller [18] in 1985 individually. Public key cryptography based on ECC provides higher security per bit than its RSA counterpart [39]. ECC has some additional advantages such as a more compact structure, a lower bandwidth, and faster computation that all make ECC usable in both high speed and low resource applications. The National Institute of Standards and Technology (NIST), US has proposed a number of standard Elliptic curves over binary fields $GF(2^m)$ [77]. Binary field curves are suitable for hardware implementation as field arithmetic operations are carry free in the field with characteristics 2. FPGA based ECC hardware design is increasingly popular because of its flexibility, shorter development time scale, easy debugging and continual improvement of the technology (i.e. lower power and higher performance FPGAs).

Many high performance ECC implementations on FPGA have been reported in the literature. Most relevant high performance ECC implementations on FPGA are presented in [10], [53], [59], [60], [61], [71], [72], [73], [79], [80], [81], and [82]. The common optimizing technique of high speed designs is the reduction of latency (number of clock cycles) of a point multiplication. To achieve low latency for a point multiplication, these works adopted either parallel multipliers or large size multipliers at the expense of additional area complexity. Pipelining stages are also often used to increase clock frequency at the expense of few extra clock cycles and area overheads [10, 59]. In addition, the pipelining stages in the multipliers create idle cycles at the point multiplication level if there is data dependency in the instructions. As a result, careful scheduling is required to take full advantage of pipelining. Indeed, recently we have reported the highest throughput and highest speed ECC designs on FPGA in [83, 84] by using novel digit-serial and bit parallel multipliers together with efficient scheduling and pipelining techniques.

The motivation of the work of this chapter is based on our successful contribution in the digit serial multiplier based ECC in [83] to yield the best area-time metric ECC design on FPGA to date. Secondly, we report an even faster ECC design with the lowest ever latency (clock cycles) that achieves the performance of the theoretical limit. These are achieved via a novel pipelining technique that enables high clock frequencies to be attained and via a thorough investigation of the different combinations of the field multipliers to evaluate the performance limits of ECC for high speed applications. Below is a list the key contributions to the results:

- We propose a full precision multiplier with segmented pipelining to reduce both latency and area.
- We propose a one multiplier-based architecture for the ECC design targeted at high performance but with low area (fastest ECC with best area and time complexities).
- We propose a three multipliers-based architecture for the ECC design aimed at the highest possible speed.
- We modify the Montgomery point multiplication algorithm to avoid extra latency due to our two-stage pipelining in the field multiplier and use careful point multiplication scheduling to reduce latency.
- A Moore finite state machine (*FSM*) based control unit is designed to avoid data dependency in the arithmetic operations. The instruction in the *FSM* is delayed by 1 clock cycle which delays the Start of data access from memory. This delay is utilised for controlling local field operations.
- Smart pipelining is adopted to shorten critical path delay in the ECC architecture.
- Data is tapped from different pipeline stages to localize some arithmetic operations and avoid memory input-output operations.
- We propose a repeated square over square circuit (capable to perform a 4-square or quad square operation in a single clock cycle) to reduce latency for the multiplicative inversion operation based on Itoh-Tsujii algorithm [65].
- Finally, we use Xilinx ISE timing closure techniques to achieve the best possible high performance results.

Our proposed high performance one multiplier based architecture takes six cycles for a loop of the Montgomery point multiplication in the projective coordinates without any pipelining delay. By applying the same method, our 2 multipliers based ECC that shows low latency while it is keeping high frequency takes 4 clock cycles for a loop of point multiplication, whereas our low latency ECC (3-multiplier based) takes only two clock cycles. The architectures have been implemented (placed and routed) on Xilinx Virtex4, Virtex5 and Virtex7 FPGA families resulting in the fastest reported implementations to date. Notably, we consider Virtex FPGA family instead of low cost Spartan family because of high performance FPGA.

On the Virtex4, our ECC point multiplication over $GF(2^{163})$ takes 5.32 µs with 13418 slices - is faster than the fastest previously reported Virtex 4 design [60] and also faster than the fastest reported design to date (5.48 µs) which was on a Virtex 5 [61]. On Virtex5, our design over $GF(2^{163})$ is not only even faster at 4.91 µs but also smaller than that of [61] by 1757 slices. Our implementation on the new Virtex7 FPGA technology achieves the best area-time performance with the highest speed to date; an ECC implementation takes only 3.18 µs using 4150 slices. To evaluate scalability of our contributions, we also implemented the proposed one multiplier based architecture over $GF(2^{571})$, the highest security curve in the NIST standard [77], on Virtex 7. This is to our knowledge, the first reported implementation that can complete a point multiplication by taking only 37.54 µs.

A key advantage of our proposed two full-precision multipliers based ECC design is its very low latency whilst enabling high operating frequency; for example, over $GF(2^{163})$ the design requires only 780 clock cycles for an ECC point multiplication. Our FPGA implementation over $GF(2^{163})$ results both on Virtex5 (5.1 µs) and on Virtex7 (3.56 µs) achieve the fastest reported ECC point multiplication on FPGA to date when it is compared with previous works.

Our parallel three multipliers based ECC design is the first reported full-precision parallel architecture which shows the highest speed (2.83 µs) for the point multiplication over $GF(2^{163})$ with the lowest latency (450 clock cycles) on FPGA to date.

The rest of this chapter is organized as follows: Section 2 presents the background of High speed ECC design. In Section 3, the proposed ECC for point multiplication is described where the point multiplication without pipelining delay, the Multiplier with segmented pipelining and other supporting circuits of ECC are discussed. The implementation results are presented, and compared to the state of the art in Section 4. Finally, this chapter is concluded in Section 5.

5.2 Background

5.2.1 High Speed Scalar Point Multiplication

The discrete logarithm problem based elliptic curve cryptography is becoming increasing more popular for public key cryptography in practice than RSA due to its improved security per bit and reduced implementation overheads for both high speed and resource

Algorithm 5.1 LD Montgomery point multiplication over *GF(2^m)* [35]

INPUT: $k = (k_{t-1}, \dots, k_1, k_0)_2$ with $k_{t-1} = 1, P = (x, y) \in E(F_{2^m})$
OUTPUT: kp

Initial Step: $P(X_1, Z_1) \leftarrow (x, 1), 2P = Q(X_2, Z_2) \leftarrow (x^4 + b, x^2)$
For i from $t-2$ downto 0 do
If $k_i = 1$ then

Point addition: $P(X_1, Z_1) = P(X_1, Z_1) + Q(X_2, Z_2)$

$$Z_1 \leftarrow X_2.Z_1$$
$$X_1 \leftarrow X_1.Z_2$$
$$T \leftarrow X_1 + Z_1$$
$$X_1 \leftarrow X_1.Z_1$$
$$Z_1 \leftarrow T^2$$
$$T \leftarrow x.Z_1$$
$$X_1 \leftarrow X_1 + T$$

Return $P(X_1, Z_1)$

Point Doubling: $Q(X_2, Z_2) = 2Q(X_2, Z_2)$

$$Z_2 \leftarrow Z_2^2$$
$$T \leftarrow Z_2^2$$
$$T \leftarrow b.T$$
$$X_2 \leftarrow X_2^2$$
$$Z_2 \leftarrow X_2.Z_2$$
$$X_2 \leftarrow X_2^2$$
$$X_2 \leftarrow X_2 + T$$

Return $Q(X_2, Z_2)$

Conversion Step: $x_3 \leftarrow X_1/Z_1$; $y_3 \leftarrow ((x + X_1)/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$.

constrained applications. The main operation in ECC is called scalar point multiplication, $Q = kP$, where k is a private key (integer), Q is a public key and P is a base point on an elliptic curve, E . The point multiplication, kP is achieved by using scalar point multiplication algorithms utilizing point addition and point doubling depending on the *i*th value of k, k_i [1].

Scalar point multiplication can be affine coordinates based or projective coordinates based. Because of the expensive inversion operation involved in affine coordinates based algorithms, projective coordinates based point multiplication is a more common choice for ECC hardware implementation.

There are several point multiplication algorithms available using to speedup point multiplications such as Montgomery ladder and NAF. The idea behind the speedup is to reduce latency of the point multiplication by exploiting parallelism (Montgomery) or inherent properties of the curve (Koblitz curve in chapter 3). The favourite point multiplication in the literature is the Montgomery method.

In this chapter, the Lopez–Dahab Montgomery (*LD*) point multiplication is considered. This algorithm, requires 6 field multiplications, 5 field squares and four additions as shown in algorithm 5.1 [35]. The *LD* algorithm is generally faster to implement, and leads to improved parallelism and resistance to side channel power attack. A parallel operation of Montgomery

point multiplication can be achieved using either two multipliers or three multipliers. The parallel operations can reduce latency with the overhead of the area. For the bit parallel multipliers based ECC [71], achieved poor frequency due to area complexity and lack of provision of using pipelining. If the number of pipelining is increased, then idle clock cycles may be created due to data dependency in the point multiplications. Thus, both a high performance multiplier and a careful scheduling are very important to speed up point multiplications.

5.2.2 Field Arithmetic over $GF(2^m)$

Field multiplication, field squaring, field addition and field inversion operations are involved in a point operation. Addition and subtraction are equivalent over $GF(2^m)$, which are very simple bitwise *xor* operations.

Field inversion is very costly in term of hardware and delay. In projective coordinates, an inversion operation is used for the projective to affine coordinates conversion which can be achieved with multiplicative inversion. The Itoh-Tsujii algorithm is selected in this work where multiplication and repeated squaring operations are used [65]. In projective coordinates based implementations, overall performance depends on the performance of the field multipliers.

5.3 Proposed Full-precision Multiplier for High Speed ECC Application

For high speed ECC application, the field multiplier is the main part of the arithmetic unit compared to the field squaring and field addition circuits due to the high area and time complexities of the multiplier. The performance of the multiplier affects the overall performance of the ECC implementation. The performance of the multiplier depends mainly on the size of the multiplier. A larger size multiplier reduces latency to speed up the point operation; however, the critical path delay is increased. Thus, pipelining is often adopted to shorten the critical path delay. Moreover, some multiplication algorithms (such as Karatshuba) are used to improve area and time complexity [59, 71, 82]. For the high-speed end of the ECC design space, large digit serial multipliers or bit parallel multipliers are often used. The bit parallel multiplier takes one clock cycle latency, which can be an attractive option to speed up the point multiplication.

The field multiplication for ECC over $GF2^m$ is divided into two parts: the Galois Field characteristic 2 (*GF2)* multiplication part (*GF2MUL*) and the reduction part. For a large size

multiplier, the *GF2MUL* part is costly compared to the reduction part [31]. Thus, the main optimization of the large multiplier is concentrated on the *GF2MUL* part due to long critical path delay. There are several high performance bit parallel multipliers in the literature [71], [78], and [79]. The complexity of a bit parallel multiplier can be quadratic or subquadratic [31]. A quadratic multiplier achieves higher speed by consuming higher area than that of a subquardratic multiplier. Subquadratic multipliers are mostly based on the Karatsuba algorithm to reduce the area complexity at the expense of a lower clock frequency. The performance of the Karatsuba based bit parallel multiplier is improved by adopting pipelining techniques [71].

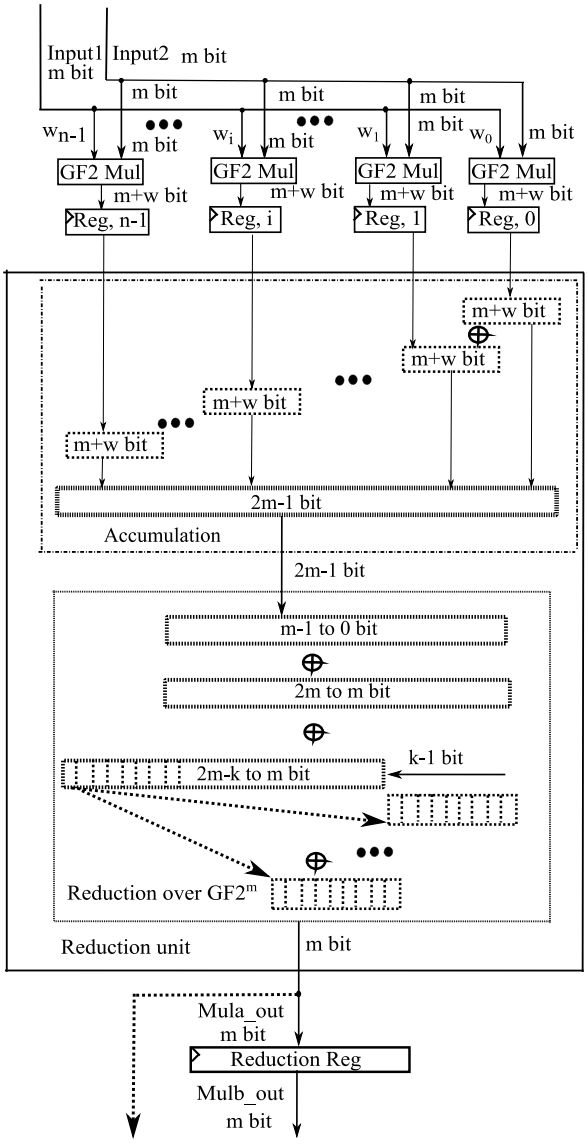


Figure 5.1 Proposed segmented pipelining based full-precision multiplier over *GF(2^m)*

In this section, we present a novel high performance full-precision multiplier (i.e. schoolbook multiplier) with segmented pipelining.

5.3.1 Multiplier with Segmented Pipelining

The proposed full-precision field multiplier with segmented pipelining is shown in Figure 5.1 and consists of two stages pipelining to improve clock frequency. The first stage pipelining is the proposed segmented pipelining to break the critical path delay of the *GF2MUL* part, which is similar to the presented work in [29]. In the segmented pipelining, we divide the *m* bit multiplier operand (*m* in in the Figure 5.1) into *n* number of *w* bit long digit multiplier operands. Then, we multiply the *m* bit multiplicand by each of the *w* bit multipliers. The results of the *w* digit size multiplier is *m+w* bit long. We save each of the results in the *m+w* size pipelining register. Here, we save *n* multiplications results in the *n* number of *m+w* size registers. The outputs of the *m+w* sizes registers are aligned by shifting (logically) *w* bits from each other followed by doing xor operation (addition). The result of the addition is *2m-1* bit long is then reduced to *m-1* bit in the reduction unit. In the reduction unit, we reduce *2m-1* bit to *m-1* bit multiplier output using a fast irreducible reduction polynomial [1][77]. The output of the reduction unit is applied to the second stage pipelining register. Thus, there are two pipelining stages and hence, the proposed multiplier consumes only 2 clock cycles as an initial delay to perform multiplication.

The two stages pipelining is adopted in our proposed one multiplier based high performance ECC (*HPECC*). The pipelining of the multiplier divides the total critical path delay into two parts: the critical path delay of *GF2MUL*, $T_A + (\log_2(m/w))\, T_X$ and the critical path delay of the reduction part, $(\log_2((n+k))\, T_X$ as shown in Table 5.1. Both critical path delays depend on the size of the segment, *w*. Thus, any one of the two critical paths can be the

Table 5.1 Latency, critical path delay (T_{mul}) and resources of the proposed full-precision multiplier and a comparison with the relevant multiplier over $GF(2^m)$

Ref	Type	#XOR	#AND	#FFs	Critical path delay(T_{mul})
[78]	Quadratic	m^2-1	m^2	-	$T_A + (1+\log_2(m))T_X$
[78]	Subquadratic	$5.5m^{\log_2(3)} - 3m + 0.5$	$m^{\log_2(3)}$	-	$T_A + (2\log_2(m)+3)\, T_X$
[79]	Pipelined-Quadratic	m^2+3m	m^2	*40m*	$T_A + ((m/p)+\log_2(m))T_X$
MUL. 2-Stage Pipelining	Fullprecision (Segmented pipelined)	$m^2+nm+3m$	m^2	*n(m+w)+m*	$T_A + (\log_2(^m/_w))\, T_X$ or $(\log_2((n+k))\, T_X$
MUL. 1-Stage Pipelining	Fullprecision (Segmented pipelined)	$m^2+nm+3m$	m^2	*n(m+w)*	$T_A + (\log_2(^m/_w + n + k))\, T_X$

w = segment size, n, #Segments = $^m/_w$, d= digit size, k is the second higher order of irreducible polynomial. T_A and T_X are AND and XOR gates delays respectively. p = # pipelining stages; #FFs=#Flip-flop.

critical path of the multiplier. The optimum critical path can be defined by the optimum size of *w* which can be determined by a trial and error method.

In our proposed low latency ECC (*LLECC*) architecture (as shown in Figure 5.3), we consider only one stage pipelining (segmented pipelining) to achieve 1 clock cycle delay. The critical path delay of the multiplier is the combination of the *GF2MUL* and the reduction part, which is $T_A + (\log_2(m/w + n + k))$. Again, the critical path delay can be modulated by changing the size of the segment of the multiplier. The optimum size of the segment of the multiplier can be also achieved by using a trial and error method. In Table 5.1, we present space and time complexities of our proposed multipliers and we compare these with quadratic and subquadratic multipliers reported in [78], and [79]. In the theoretical analysis of the quadratic and subquadratic multipliers, the quadratic multiplier has twice as high speed as the speed of the subquadratic, but, the quardratic multiplier consumes 2.56 times more area [78]. Moreover, the authors in [78] compare the implementation results of the two multipliers where they show the ratio (Quadratic/Subquadratic) of area is 1.5 and the ratio of delay is 0.625. Their implementation results depict that the bit parallel multiplier can achieve higher speed, however, the area-time product of the subquadratic multiplier outperforms the quadratic multiplier by only 6.65 %. Thus, a quadratic multiplier is a better option for high speed ECC implementation. As shown in Table 5.1, our proposed multiplier has a very short critical path compared to the reported parallel multipliers; hence, our multiplier can show better area-time performance due to its high speed advantage. For the case of space complexity, our proposed multiplier consumes the same resources of *XOR* and *AND* gates as that of the quadratic bit parallel multiplier and uses flip flops (FFs) to reduce the critical path delay.

5.4 Proposed High Performance ECC (HPECC) for Point Multiplication

In this section, we present a careful scheduling in the point addition and point doubling operations, a novel pipelined full-precision multiplier and other supporting units to achieve high speed, low latency while optimizing area complexity.

5.4.1 Point Multiplication without Pipelining Delay

In general, the Montgomery point addition and point doubling in the projective coordinates requires six field multiplications, five field squares and four field-addition

operations equivalent latency if implemented serially according to algorithm 1 [35]. If the field squaring and field addition operations can be operated concurrently with multiplication, then the point operations latency will be equivalent to the latency of the six field multiplications. The six multiplications can for example be computed in two steps by using three multipliers or in three steps by using two multipliers or in six steps by serial multiplications using one multiplier [59], [61] and [73]. Again, the digit size can affect the performance of ECC; for example, a bit serial implementation takes *m* cycles, a digit (*w* bits) serial one takes (m/w) cycles and a bit parallel implementation takes a single clock cycle [10], [70] and [71]. In the case of high speed design, digit serial multipliers are considered to reduce latency. The disadvantage of large digit serial multipliers is lower clock frequency. Thus, pipelining stages are applied to improve clock frequency [10]. The clock frequency can be improved with the increase of the number of pipelining stages in breaking the critical path delay. The main disadvantages of increasing the number of pipelining stages in the high-speed end of the design space are the increase in the number of clock cycles per multiplication and overcoming data dependency [10]. To avoid pipelining delay, optimal scheduling of the field operations of the point multiplication is necessary.

Our first proposed ECC architecture is shown in igure 5.2 over $GF(2^m)$. It comprises a full-precision *m* bit multiplier with two stages pipelining, one squaring circuit, one quad squaring circuit and two additions circuits in order to accomplish point operations (point addition and point doubling) within six clock cycles. To achieve six clock cycles based point operations, we include some strategies in the point operations of the Montgomery point multiplication algorithm as shown in algorithm 2 [83].

In the proposed algorithm, we combine point addition and point doubling to avoid data dependency. In the point multiplication, a particular loop is overlapped with its next loop by 2 clock cycles due to two stages pipelining. Thus, state1 (*st1*) and state2 (*st2*) depend on the previous key bit, k_{i+1}. For example, if previous bit, $k_{i+1} = 1$, then the last output will be X_1 otherwise X_2. The last output of a loop decides the sequence of *st1* and *st2* in the next loop. The rest of the states depend on the current bit of k, k_i. To support a six clock cycle based algorithm, we apply a squarer or double square (Quad Square) or both operations in parallel along with the multiplication. Again, one of the field adders is placed in the common data path

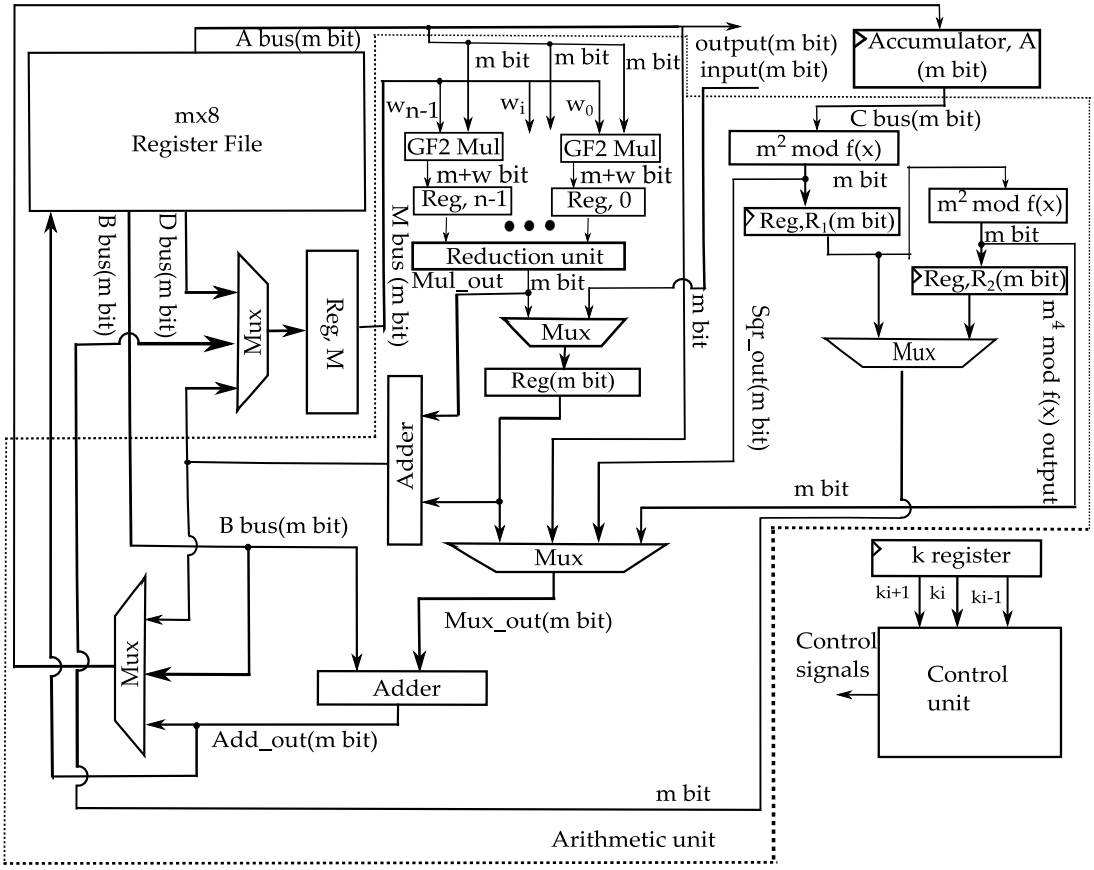


Figure 5.2 Proposed high performance ECC architecture

to add on the fly. The second adder is used to add the two outputs of the multiplier as shown in Figure 5.2. Both adder circuits can add two of their inputs or can transfer either of the inputs, if we need either. Moreover, we can save some intermediate results of field operations in the local registers (R1, R2, M and accumulator, A) to avoid loading/unloading to the main memory. As a result, we can avoid idle clock cycles due to the memory input-output operations.

A data flow diagram is shown in Figure 5.3 to demonstrate the proposed combined point operations. In this diagram, we explain point operations for $k_{i+1} = 1$, $k_i = 1$ and $k_{i-1} = 1$ where k_i the current bit is, k_{i+1} is the previous bit and k_{i-1} is the next bit of key (k). In this data flow diagram, we show the loop operation of the point multiplication in projective coordinates. In our implementation, a multiplication takes three clock cycles due to two stages pipelining and a square operation takes two clock cycles where one clock cycle is used to load in the accumulator (A) register. The addition operation is realized in the common data path and accomplished in the same clock cycles. As we used two stage pipelining and there is a data

Algorithm 5.2 Proposed combined LD Montgomery point multiplication (with each loop for six clock cycles)

For i from $t-2$ down to 0 do

If $k_i = 1$ then

If $k_{i+1} = 1$ then	If $k_{i+1} = 0$ then
Point addition:$P(X_1,Z_1) = P(X_1,Z_1) + Q(X_2,Z_2)$ and Point Doubling: $Q(X_2,Z_2) = 2Q(X_2,Z_2)$	
St1: $Z_1 \leftarrow X_2.Z_1; A \leftarrow Z_2$	St1: $Z_2 \leftarrow X_1.Z_2; A \leftarrow Z_2$
St2: $X_1 \leftarrow X_1.Z_2, Z_2 \leftarrow A^2$; $R_2 \leftarrow A^4$; $A \leftarrow X_2$	St2: $X_2 \leftarrow X_2.Z_1; Z_2 \leftarrow A^2$; $R_2 \leftarrow A^4$; $A \leftarrow X_2$
St3: $X_2 \leftarrow b.R_2 + A^4$; $R_1 \leftarrow A^2$	
St4: $Z_2 \leftarrow R_1.Z_2$, $A \leftarrow X_1 + Z_1$	
St5: $X_1 \leftarrow X_1.Z_1$, $Z_1 \leftarrow A^2$	
St6: $X_1 \leftarrow x.Z_1 + X_1$.	
Conversion Step: $x_3 \leftarrow X_1/Z_1$; $y_3 \leftarrow ((x+X_1)/Z_1)[(X_1+xZ_1)(X_2+xZ_2) + (x^2+y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$.	

dependency in between two loops, we use careful scheduling. In this scheduling, the present loop operation of point multiplication is overlapped with the next loop operations.

- We see, the starting state, *st1* of a particular loop depends on the hamming weight of previous bit of , k_{i+1}. If the previous bit, $k_{i+1} = 1$ means X_1 is not ready, then, we start from state1 (*st1*) with the multiplication between X_2 and Z_1 instead of X_1 and Z_2. In this case, The state2 (*st2*) is the multiplication in between X_1 and Z_2.
- The X_1 operand of the *st2* is calculated by addition of two outputs (*Mula_out* and *Mulb_out* as in Figure 5.2) of the multipliers where one output(from *Mula_out*) is tapped after the reduction unit (dotted arrow) and the other one from the multiplier output(*Mulb_out*). The other operand of *st2* is Z_2 which is already saved in the memory in st1 to use in *st2*. Here, the delay of the memory operation (accessing Z_2) is utilized to calculate X_1; again, as $k_i = 1$, we need the square and quad square of Z_2. Thus, we save Z_2 in the memory and accumulator simultaneously in st1 to achieve the squaring operations of Z_2 in the *st2*. The output of the square circuit ($A^2 = Z_2^2$) is saved in the

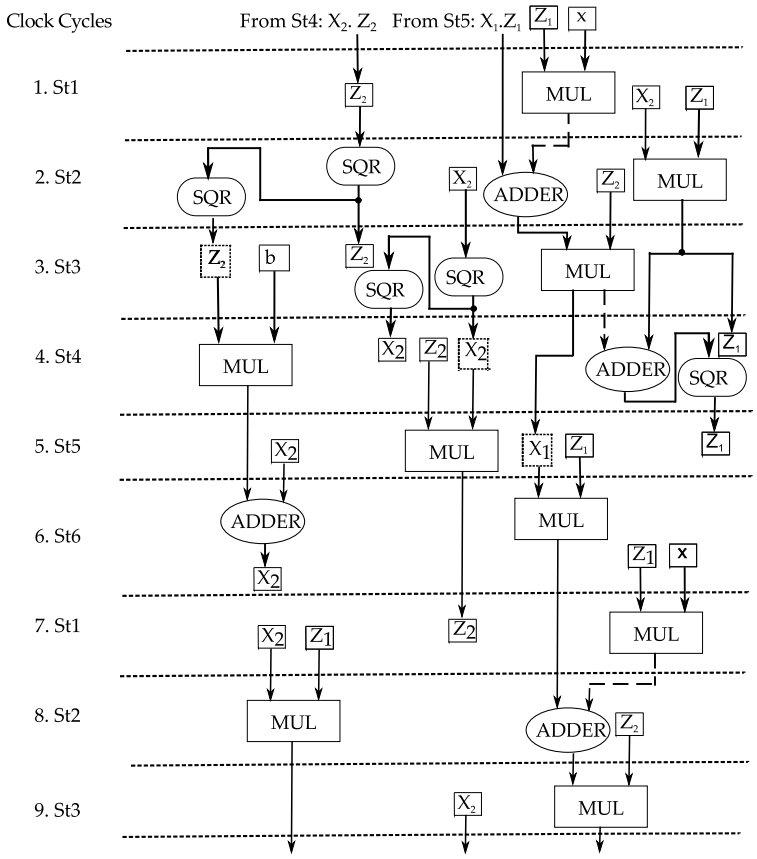


Figure 5.3 Data flow of HPECC for $k_{i+1} = 1$, $k_i = 1$ and $k_{i-1} = 1$

Memory and the output of quad square ($A^4 = Z_2^4$) is saved in the local register, R_2 (dotted box). We can use data from the local register (dotted box) immediately without doing any memory operations to save clock cycles.

- Similarly, during *st2*, *st3* and *st4*, the squaring operations of X_2 is realized by saving in the accumulator through b_bus; in this case, the square output, $A^2 = X_2^2$ is saved in the local register, R_1 , and the quad square output, $A^4 = X_2^4$ is saved in the memory. In *st3* and *st4*, one of the multiplication operands is used from the memory and the other operand from the local registers.

- In st4, Z_1 (result of $X_2.Z_1$) is ready to save in the memory to use in st5. Again in *st4*, the available output, Z_1 is required to add with the multiplication result of X_1 on the fly. At this time, we access (tapping) X_1 from the output of the reduction unit (dotted arrow, one cycle earlier than the normal output) to add with Z_1 followed by saving in the accumulator to do the square operation to get a new Z_1.

- The new Z_1 is ready in *st5* to save in the memory and is required in the *st6* and the next loop. In *st5*, the old Z_1 (saved in *st4*) is used for multiplication with X_1 where X_1 is directly collected from the multiplier output followed by saving in the local register, M. We can manage X_1 to use immediately for multiplication by using the instruction delay (Moore machine based control unit) of accessing the old Z_1 from memory.
- In *st6*, we add X_2 (from memory) on the fly with the multiplier output to get new X_2 followed by saving in the memory. Again, the multiplication in *st6* is in between the base point, x and new Z_1 is completed after two clock cycles. But, a new loop is started after *st6*.

Thus, the st1 of the new loop depends on the last coordinate of the previous loop, X_1 (in this case of $k_{i+1} = 1$, $k_i = 1$ and $k_{i-1} = 1$) which is calculated by adding the results of the multiplications started in *st5* and *st6*.

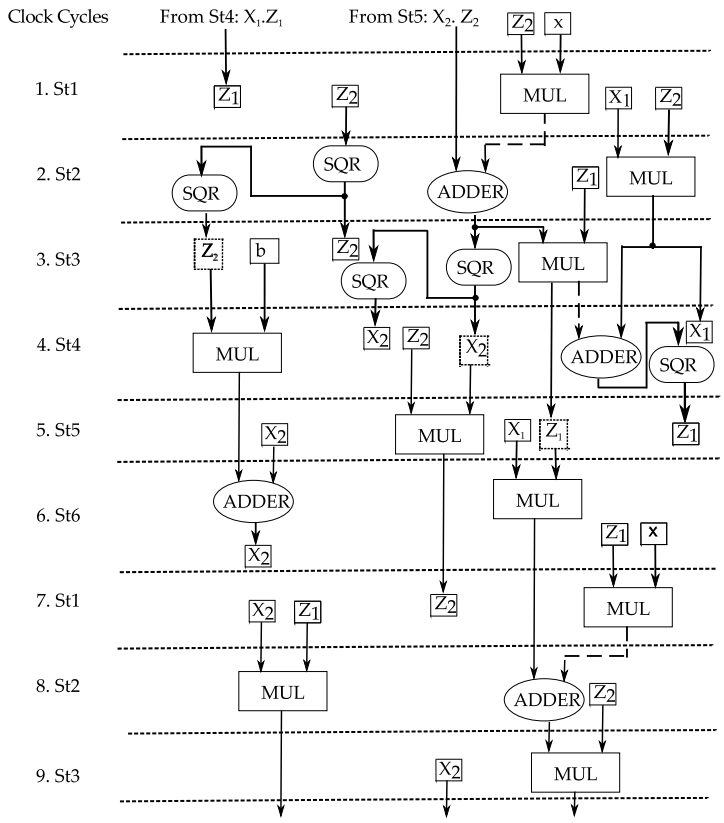


Figure 5.4 Data flow of HPECC for $k_{i+1} = 0$, $k_i=1$ and $k_{i-1} = 1$

In Figure 5.4, we demonstrate the loop of point multiplication for $k_{i+1} = 0$, $k_i = 1$ and $k_{i-1} = 1$. The previous bit of k, is $k_{i+1} = 0$ means coordinate X_2 of the last loop is not ready to start with.

- In this case, the first state (*st1*) is started with multiplication between X_1 and Z_2. In this state, the multiplier output (Z_1) started from st4 of the previous loop is saved in the memory to use in the next state (*st2*). In the same state, we need to start the squaring operation on Z_2. Thus Z_2 is accessed from memory through the a_bus for multiplication and through the b_bus into the accumulator for squaring.

- In *st2*, the multiplication is $X_2 . Z_1$; where X_2 is calculated by adding two outputs of the multiplier and then is saved in the M register for using in the next cycles to multiply with Z_1. In the same time, the calculated X_2 is required and saved in the accumulator for squaring as $k_i = 1$.

The rest of the states of Figure 5.4 are similar to those in Figure 5.3.

5.4.2 Multiplier with Segmented Pipelining for HPECC

We consider the two extreme field sizes in the NIST standard [77] i.e $GF2^{163}$ and $GF2^{571}$ to evaluate the ECC performance. In the implementation over $GF2^{163}$, we select $w = 14$ bit to get 12 of the 14 digit serial multiplication results. The results then are loaded in the twelve 177 bit long registers. Thus the critical path of *GF2MUL* depends on one two input *AND* gate and 13 layers of two input XOR gates to achieve a 14x163 multiplication. Again, the 12 pipelining register outputs are shifted and *XORed* (for accumulation) to get the full-precision multiplication result *(2m-1)* without reduction. The result is then reduced into 163 bit in the reduction unit using the fast irreducible reduction polynomial [77]. The reduced result is saved in the second stage pipelining register. Thus, the architecture works like 12 (14 bit) digit serial multipliers are operating in parallel followed by a full precision reducing operation. The reduction unit consists two parts: the accumulation part and the reduction part. The accumulation part has 11 layers of 2 inputs *XORs* and the reduction part has *k* layers of 2 input *XORs*. Thus, the crical path delay is balanced theoretically. Again, in the ECC implementation over $GF2^{571}$, we also consider the segment size of 14 bit.

5.4.3 Square Circuit, Memory Unit and Control Unit of HPECC

Our proposed high speed ECC design operates by using six clock cycles for each loop of the point multiplication. To achieve the six cycles point multiplication loop, we need a quad

square (4-square) circuit to do a one clock quad square operation. The quad squaring is used in the *st2* and *st3* along with field multiplication as shown in our proposed algorithm 5.2. Again, the latency of the conversion step contributes a significant amount to the total latency of the proposed ECC as the latency of the loop operation is comparable to that of the conversion step. In the conversion step, the inversion operation consumes the major part of the latency in our projective based ECC implementation, a multiplicative inversion is applied for the projective to affine coordinate conversion. Several multiplications and *m* steps repeated squaring operations are required. Thus, we can utilize the quad square circuit for speeding up the inversion by reducing the number of the repeated square operations. In our proposed architecture, we use a register (accumulator) in the arithmetic data path to achieve a repeated quad square operation without loading to the main memory. Thus, we need 1 clock cycle for a 4-square, 2 clock cycles for an 8-square and so on.

We design a friendly memory unit that is developed in a single behavioral entity which comprises an accumulator and 8x*m* register file. The register file is based on distributed RAM to give high performance and flexibility. There are five input-output buses in the memory unit. Particularly, our register file consists of three output buses (A, B, D) and one input bus. Data through *a_bus* and *b_bus* takes one more cycle delay than data through *d_bus*. Data from *D_bus* is dedicated to the multiplier input through the M register. Hence, the two outputs of the memory through *a_bus* and *b_bus*, and the output of *M* (through *D_bus*) are syncronised. The *M* register acts as a pipelining register between the input and the output of the multiplier and also saves local data for the multiplier. The memory unit offers flexibility to access any data from any location of the memory through each of the output buses independently. The memory unit takes one cycle for a write operation and one cycle for a read operation. The accumulator is designed in the same entity of the memory unit and utilizes unused resources (flip-flops) of the memory unit. Apart from our memory unit, we deploy local registers R_1 and R_2 ; R_1 and R_2 are used to save outputs of square and quad square respectively. Thus, the local registers (R_1 and R_2) and M save outputs of concurrent operations to avoid the idle state that is due to the common input bus of the memory unit, and also avoid the data dependency in the successive point operations loop.

Table 5.2 Critical path delay *(T_{ECC})* of the proposed ECC

Ref	Critical path delay
HPECC	T_{mul} or $(\log_2(n+k))\,T_X + T_{adder} + 2T_{mux}$
HFECC	$Path1: T_A + (\log_2(\frac{m}{d}))\,T_X + 4\,T_{mux}$ or $Path2: (\log_2((n+k))\,T_X + T_{adder}(= T_X) + T_{sqr}(= (\log_2((k))T_X) + 3\,T_{mux}$
LLECC	$T_{mul} + T_{adder} + T_{sqr} + 3\,T_{mux}$

$n =$ #Segments, d= digit size, k is the second higher order of irreducible polynomial, T_{mux}= 2x1 mux delay, T_{sqr}=$\log_2(k)$, $T_{adder} = T_x$, HPECC= High performance ECC, HFECC= High Frequency ECC, LLECC= low latency ECC.

A Moore finite state machine based control unit is developed in the single behavioral entity. The pipelined Moore machine takes one clock cycle delay as compared to a Mealy machine while Mealy machine depends on input to change state. The advantage of this initial instruction delay is a more flexible data control as we use an *FSM*. We can utilize the one cycle delay to accomplish some intermediate operations with the help of the local registers. Again, the control unit consists of very few states to complete a point multiplication due to the full-precision multiplier and concurrent operations. As a result, the control unit consumes very low area while helps keeping speed very high.

5.4.4 Critical path delay and clock cycles of the HPEEC

Our proposed high speed ECC (*HPECC*) design uses a segmented pipelining based full-precision multiplier to achieve six clock cycles for each loop of the point multiplication. The critical path delay of the ECC mainly depend on the critical path of the multipliers. Again, the proposed multipliers critical path delay can be the critical path delay of the *GF2MUL* part or the reduction part depending on the size of the segment. As the multiplier output (*Mula_out*) is taped at end of the reduction part, and passed through the adder and multiplexer followed by saving in the *M* register, the critical path delay of the ECC can be the delay of the reduction part+adder + mux. The critical path delay of the ECC architecture is shown in the Table 5.2.

The focus of our proposed ECC is the reduction in the number of clock cycles. Particularly, our design can manage to take 6 clock cycles for each loop of the point multiplication in the projective coordinates. The total clock cycles for point multiplications is the sum of three main parts: affine coordinates to projective coordinates initialization, point multiplication in the projective coordinates and finally projective coordinates to affine coordinates conversion. The total number of clock cycles (*CCs*) for point multiplication = 5 *CCs* (required for initialization)

+ *6xm CCs* (for point multiplication in the projective coordinates) + *CCs* (for the final coordinates conversion = m/2 *CCs* for square + #Mul for inversion x3 +3 *CCs* for Inversion + *28 CCs* for others) + 3 clock cycles for pipelining as shown in Table 5.3. For example, the total clock cycles for point multiplication over $GF2^{163}$= *5+(6x162)+139(=(81+27+3)+28) +3=1119* cycles. Similarly, the latency of *HPECC* processor over $GF2^{571}$ is 3783 clock cycles.

5.5 Proposed the highest possible frequency based ECC (HFECC) for Point Multiplication

High speed ECC can be designed on software, software/hardware, and hardware platforms. For high speed, hardware implementation is attractive to overcome the high latency associated with word level computations of field multiplication [78]. In this section, a high speed hardware implementation on FPGA of ECC is presented comprising two novel full precision pipelined multipliers. A low latency point multiplication scheduling to avoid data dependency in the point operations for the proposed high frequency ECC (HFECC).

We propose a high speed ECC hardware architecture over binary fields $GF(2^m)$. To achieve very low latency, we adopt full precision multiplication. We improve the multiplier's performance by using a novel 2 stages pipelining technique. In our proposed multiplier, we divide a $GF(2^m)$ multiplication into *wxm* multiplications where *w* is the digit size and, where the $n = m/w$ number of *w* digit serial multiplications can be achieved using two stages of pipelining thus enabling very low latency while it is still maintaining high clock frequency. Furthermore, we adopt the Montgomery point multiplication to exploit the underlying parallelism and combine point addition and point doubling operations to concurrently deploy two full precision field multipliers to increase speed. As there is data dependencies in the Montgomery point multiplication, we modify the schedule of point operations to avoid these data dependencies. Also, we use local registers as part of the pipelining to save intermediate data; this avoids expensive memory operations and further improves the critical path delay in the overall ECC architecture. We designed a distributed logic based Memory unit, a dedicated finite state machine (FSM) based control unit, and a novel 4-squarer (square over square) based inversion operation to accelerate the overall performance of the proposed ECC.

5.5.1 Low latency Point Multiplication Shecdualing for HFECC

For a high speed ECC implementation, a careful point multiplication scheduling is required to avoid data dependency due to pipelining stages in the field multiplier. The merit of pipelining

Algorithm 5.3 Proposed combined LD Montgomery point multiplication (main loop)

For i from $t-2$ down to 0 do If $k_i = 1$ then	
If $k_{i+1} = 1$ then	If $k_{i+1} = 0$ then
Point addition:$P(X_1,Z_1) = P(X_1,Z_1) + Q(X_2,Z_2)$ and Point Doubling: $Q(X_2,Z_2) = 2Q(X_2,Z_2)$	
St1: $Z_1 \leftarrow R_y.\ R_2$; $\{as\ R_y = X_2\ and\ \ R_2 = Z_1\}$ $R_1 \leftarrow {R_y}^2$; $R_4 \leftarrow {R_y}^4$; $R_c \leftarrow Z_2$; $\{R_y = X_1\} \leftarrow R_{m1} + R_{m2}$; St2: $X_1 \leftarrow R_y.Z_2$; $R_2 \leftarrow {R_c}^2$; $R_3 \leftarrow {R_c}^4$;	St1: $X_1 \leftarrow R_y.\ R_2$; $\{as\ R_y = X_1\ and\ \ R_c = R_2 = Z_2\}$ $R_2 \leftarrow {R_c}^2$; $R_3 \leftarrow {R_c}^4$; $\{R_y = X_2\} \leftarrow R_{m1} + R_{m2}$; St2: $Z_1 \leftarrow R_y.Z_1$; $R_1 \leftarrow {R_y}^2$; $R_4 \leftarrow {R_y}^4$;
St3: $R_y \leftarrow b.R_3 + R_4$; $Z_2 \leftarrow R_1.R_2$; $R_2 \leftarrow (X_1 + Z_1)^2$ St4: $R_{m1} \leftarrow x.\ R_2$; $R_{m2} \leftarrow X_1.Z_1$;	

can be exploited by observing data dependency in the successive field operations. We propose a combined Montgomery point multiplication algorithm to comply with the pipelining delay of our proposed multiplier as shown in algorithm 5.3. In the proposed algorithm, the last two multiplications results are required to be added concurrently to start a new loop and pipelining delay needs to be overcome. This is achieved by having a different Start Operation (from state 1, *St1*) for the new loop to the previous loop. The values of the Key select the Starting Operation. Two full precision multipliers are used to achieve 4 clock cycles for the main loop

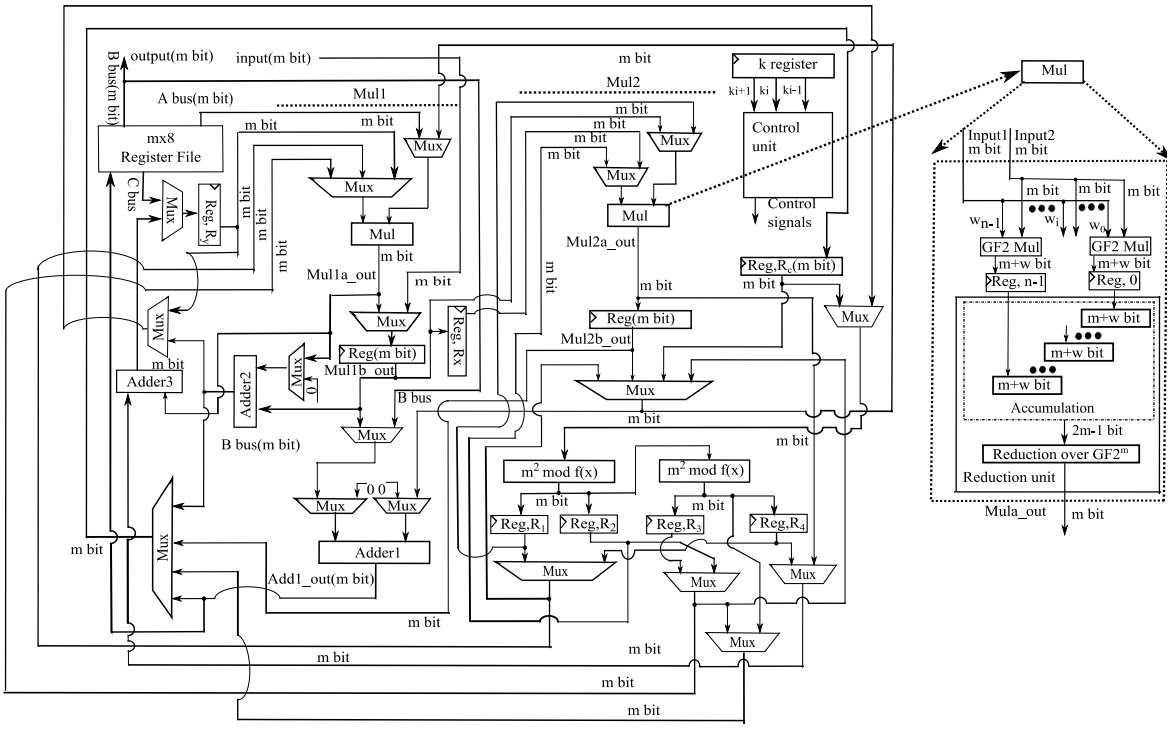
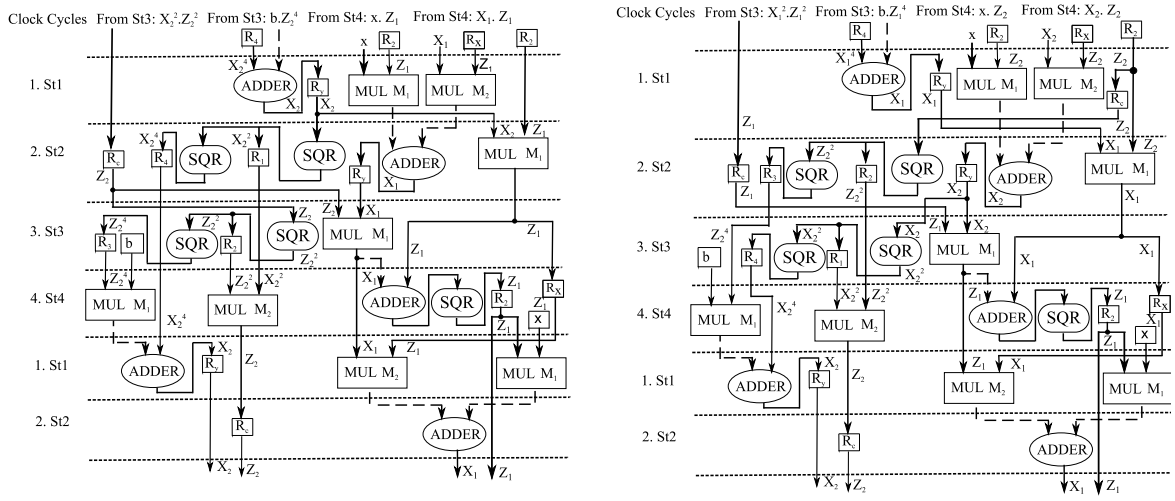


Figure 5.5 Proposed HFECC architecture for the high speed point multiplication

operation. The loop operation is folded over the next loop to save latency due to pipelining. Our proposed multiplier can support the proposed 4 clock cycles algorithm as the multiplier has two outputs; output *a* (*mul1a* for multiplier1 (*mul1*) and *mul2a* for multiplier 2 (mul2) and output *b* (*mul1b* and *mul2b*). Note that multiplier takes 1 clock cycle to have an output from output *a* and 2 clock cycles from output *b*.

The proposed 4 clock cycles based loop operation is illustrated in Figure 5.6a and in Figure 5.6b for the current bit of key, $k_i = 1$. In state 1 (*st1*) and state (*st2*), we consider 1 multiplication and in state 3(*st3*) and state 4(*st4*), two multiplications for each state are performed. The Figure 5.6a shows the main loop operation for the case of the previous bit of the key, $k_{i+1} = 1$:

- In *st1*, $X_2.Z_1$ is performed, the input X_2 is calculated and saved in the local register R_y to input in the multiplier 1(*mul1*) as shown in Figure 5.5. Z_1 is the other input of multiplier1 saved in the register 2 (R_2) as also shown in Figure 5.5. The content of R_y is squared and 4-squared to achieve X_2^2 and X_2^4. The results of the squares are saved in R_1 and R_4 respectively.
- In *st2*, the $X_1.Z_2$ multiplication is performed. The input X_1 is a result of the addition of the mul1a_*out* of the *mul1* and *mul2a_out* of the *mul2* which is saved in the R_y to input in the mul1. The other input of the mul1 (Z_2) is inputted into *mul1* through R_c. The content of R_c (Z_2) is squared and 4-squared are saved in the R_2 and R_3 respectively.
- The two multiplications $b.Z_2^4$ and $X_2^2.Z_2^2$ are performed in the *st3* in *mul1* and *mul2* respectively. The results of $b.Z_2^4$ will be added with X_2^4 (the content of R_4) in the



a) k_i=*1* and k_{i+1} =1 b) k_i=1 and k_{i+1} = 0

Figure 5.6 Main loop operations with the k_i values, including: a) k_i=*1* and k_{i+1} =1 and b) k_i=1 and k_{i+1} = 0

following loop to generate new X_2 . Again, the other multiplication gives a new output of Z_2.

- The final state involves two multiplications ($x.Z_1$ (=$(X_1 + Z_1)^2$) and ($X_1.Z_1$) using two multipliers simultaneously. Out of the two inputs of *mul1*, x is an output of the memory and Z_1 is the new Z_1 output. The new Z_1which is just the squared of the addition result of the multiplication output (Z_1) of the *st1*and the output of X_1 the multiplication of *st2*. For $X_1.Z_1$, the outputs *st1* and *st2* are inputted locally to *mul2*. To synchronize, the *st1* multiplication output, $Z_1$is saved in the local register R_x to input in the *mul2* at *st4*, and the X_1, the output multiplication result of *st2* is directly inputted to *mul2*. The outputs of the two multiplications at *st4* are finally added in the following loop to generate new X_1.

The last multiplication result is X_1 for $k_i = 1$ or X_2 for $k_i = 0$. The last result of the previous loop will be inputted as a multiplication operand at *st2*. For example, if X_1 is the last ouput of the previous loop, then *st2* is considered for the multiplication of $X_1.Z_2$. Similarly, if X_2 is the last output of the previous loop, then the multiplication at *st2* will be $X_2.Z_1$. The changing multiplications based on the previous value of *k*, k_{i+1} is shown in Figure 5.6b and in algorithm 2. The other difference of Figure 5.6b from Figure 5.6a is that the squaring operation on $Z_2$is considered first as the X_2 (last output of the previous loop) is not ready at *st1*.

5.5.2 Square Circuits, field Inversion operation and coordinates conversion of HFECC

The field square circuit consumes very low resource and performs a square operation in a single clock cycle. To achieve 4 clock cycles based main loop operation of the point multiplication, we need to do 4-squared in a single clock cycle. Thus, we need two square circuits connected as cascaded to get both square and 4-square in the same clock cycle.

A standalone field inversion is a costly operation in the ECC implementation. As we consider projective coordinates based ECC design, we need to compute one inversion operation during coordinates conversion. We consider the widely used multiplicative inversion operation proposed by Itoh-Tsujii [65]. For an inversion operation over $GF(2^m)$, a repeated square and multiplication operation is required [65]. For example, there are 9 multiplication and 162 square operations required for the inverse operation over $GF(2^{163})$. As there is a dependency in data in the consequent field operation during inversion, each multiplication takes extra latency due to pipelining. We can improve the inversion latency by using a 4-square circuit to speed up the repeated squaring operations.

The final step of the Montgomery point multiplication is the coordinates conversion which involves field inversion, multiplication, squaring and addition operations. For the low latency and high speed ECC operation, the optimization part in the conversion coordinates step is the inversion operation. As we use a 4-square circuit we save clock cycles during the inversion operation. Our conversion step takes 125 clock cycles out of which 104 clock cycles are for the inversion. The total latency of the proposed ECC is shown in Table 5.3 (page 5-28). The total latency over $GF(2^{163})$ is as follows:

Total clock cycles for point multiplication (PM) is 780 = 7 clock cycles for initialization + 4x162 clock cycles for main loop of the PM + 125 clock cycles for conversion.

5.5.3 Memory Unit and Control Unit of HFECC

We developed a distributed logic based memory unit to get a high throughput ECC. Our memory unit is *8xm* size with three outputs buses (a, b and c) and one input bus as shown in Figure 5.5. We can access data separately using each of the buses. Initially, we save the inputs data in the memory. After the start of the point multiplication loop, variable data are saved in the local registers. The constant data are accessed from memory. The local registers R_x, R_y R_c, R_1, R_2, R_3, and R_4 save intermediate data to save latency for memory access.

We design a dedicated control circuit of our ECC with a finite state machine (FSM) modelled in the same entity. We control the state of point multiplication using the counter inputs, and three key bits (previous bit k_{i+1}, current bit k_i, and next bit of k, k_{i-1}). As the proposed design is based on very low latency and high speed, we perform several arithmetic operations concurrently by controlling multiplexers.

5.5.4 Pipelining in the ECC Architecture of HFECC

We introduce a novel pipelining in the ECC architecture to break the long critical path delay. We use some local registers such as R_x, R_y R_c, R_1, R_2, R_3, and R_4 to save local variables to input into multiplication. The local registers offer flexibility to do some low cost arithmetic operations such as addition or squaring by accessing data from faster output ports (*mul1a* and *mul2a*) of the proposed multiplier. Thus, we can save clock cycles for addition and squaring by concurrent operation followed by saving in the local registers. By doing this, the overall two clock cycles latency of our proposed multiplier is preserved. The critical path delay of the ECC architecture is shown in Table 5.2 (page 5-17). The value of *n* defines the critical path delay. There are two possible critical path delays, *path1* and *path2* as shown in Table 5.2 (page 5-17).

The selection of the optimum size of *n* is matter of a trial and error method as different cell technologies have different flexibilities.

5.6 Proposed Low Latency ECC (LLECC) Processor for Point Multiplication

The speed of ECC can be improved for high speed applications by reducing latency of the point multiplication. Parallel full-precision multipliers can reduce latency to speed up the point operations. We propose a high speed ECC processor for point multiplication utilizing three full-precision multipliers to achieve the lowest latency high speed ECC as shown in Figure 5.7.

5.6.1 Low Latency Montgomery Point Multiplication

Montgomery Point multiplication offers flexibility of parallel field operations. There are six field multiplications in the projective coordinates based Montgomery point multiplication as shown in algorithm 5.1. In theoretically, the six multiplications can be

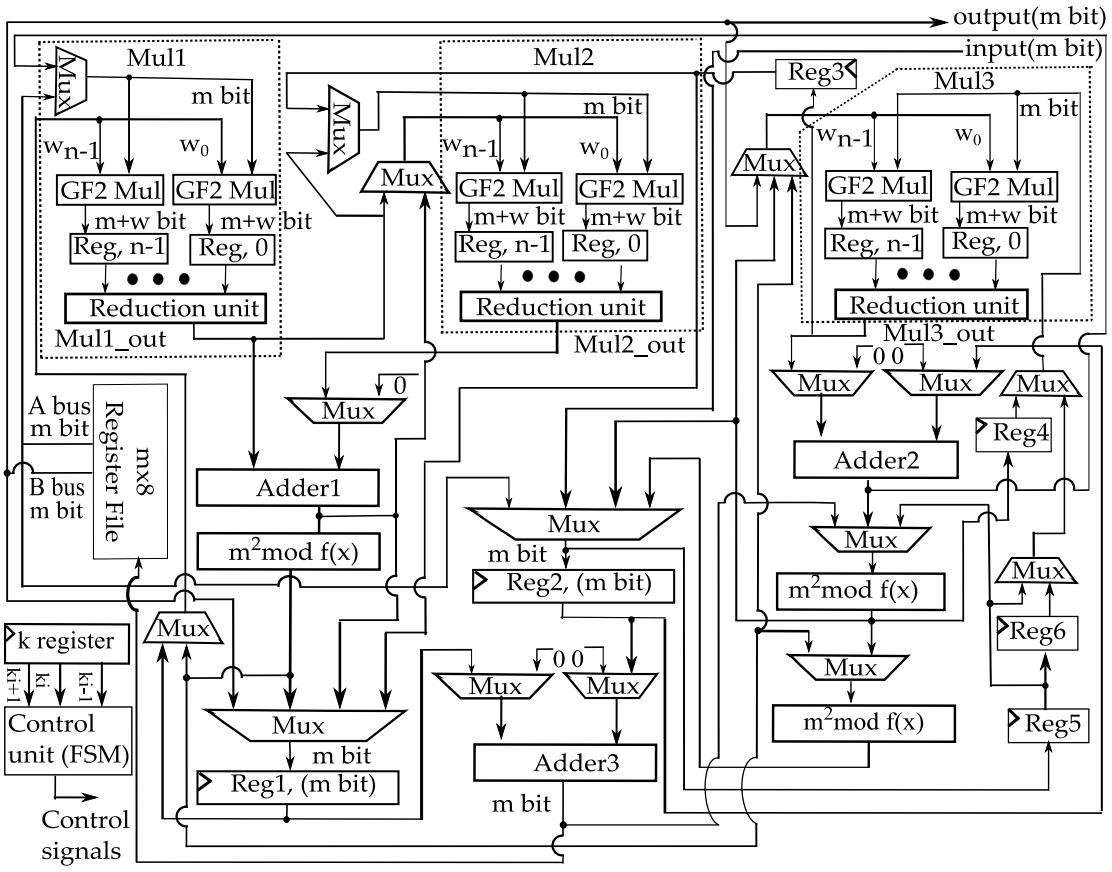


Figure 5.7 Proposed low latency ECC architecture

Algorithm 5.4 Proposed low latency Montgomery point multiplication (with each loop for two clock cycles)

For i from $t-2$ down to 0 do

If $k_i = 1, k_{i+1} = 1$ and $k_{i-1} = 1$ then{ No transition}

Point addition:$P(X_1, Z_1) = P(X_1, Z_1) + Q(X_2, Z_2)$ and Point Doubling: $Q(X_2, Z_2) = 2Q(X_2, Z_2)$

	Mul1	Mul2	Mul3
St1:	$Z_1 \leftarrow X_2.\ R_1; \{R_1 = Z_1\}$	$X_1 \leftarrow X_1.R_3;$	$Z_2 \leftarrow X_2^2.R_4;\ R_2 \leftarrow X_2^4;$
St2:	$X_1 \leftarrow (x.(X_1 + Z_1)^2$	$+ \quad X_1 \leftarrow X_1.Z_1);$	$X_2 \leftarrow b.R_5 + R_2;$
	$R_1 \leftarrow (X_1 + Z_1)^2$	$R_3 \leftarrow Z_2$	$R_5 \leftarrow {Z_2}^4; R_4 \leftarrow {Z_2}^2;$

else If $k_i = 1, k_{i+1} = 1$ and $k_{i-1} = 0$ then{Transition :$k_i = 1$ to $k_i = 0$}

St1: $Z_1 \leftarrow X_2.\ R_1; \{R_1 = Z_1\}\quad X_1 \leftarrow X_1.R_3;\quad Z_2 \leftarrow X_2^2.R_4;\ R_2 \leftarrow X_2^4;$

St2: $X_1 \leftarrow (x.(X_1 + Z_1)^2 \quad + \quad X_1 \leftarrow X_1.Z_1);\qquad X_2 \leftarrow b.R_5 + R_2;$

$R_1 \leftarrow (X_1 + Z_1)^2 \qquad R_3 \leftarrow Z_2 \quad R_5 \leftarrow ((X_1 + Z_1)^2)^2; \{R_5 = {Z_1}^2\}$

If $k_i = 0, k_{i+1} = 1$ and $k_{i-1} = 0$ then {Transition : $k_i = 1$ to $k_i = 0$}

St1: $X_2 \leftarrow X_2.\ R_1; \{R_1 = Z_1\}\ Z_2 \leftarrow X_1.R_3;\qquad Z_1 \leftarrow X_1^2.R_5;\ R_2 \leftarrow X_1^4;$

$R_4 \leftarrow {R_5}^2; \{R_4 = {Z_1}^4\}$

St2: $X_2 \leftarrow (x.(X_2 + Z_2)^2 \quad + \quad X_2 \leftarrow X_2.Z_2);\qquad X_1 \leftarrow b.R_4 + R_2;$

$R_1 \leftarrow (X_2 + Z_2)^2 \qquad R_3 \leftarrow Z_1 \qquad R_5 \leftarrow {Z_1}^4; R_4 \leftarrow {Z_1}^2;$

If $k_i = 0, k_{i+1} = 0$ and $k_{i-1} = 0$ then{ No transition}

St1: $Z_2 \leftarrow X_1.\ R_1; \{R_1 = Z_2\}\ X_2 \leftarrow X_2.R_3;\qquad Z_1 \leftarrow X_1^2.R_4;\ R_2 \leftarrow X_1^4;$

St2: $X_2 \leftarrow (x.(X_2 + Z_2)^2 \quad + \quad X_2 \leftarrow X_2.Z_2);\qquad X_1 \leftarrow b.R_5 + R_2;$

$R_1 \leftarrow (X_2 + Z_2)^2 \qquad R_3 \leftarrow Z_1 \qquad R_5 \leftarrow {Z_1}^4; R_4 \leftarrow {Z_1}^2;$

Conversion Step: As shown in the Algorithm 5.1.

achieved in two steps by using three full-precision multipliers as shown in algorithm 5.4. To achieve the theoretical limit of the loop operation, an ECC architecture needs single clocked field multipliers along with concurrent square and addition operations, all with careful scheduling. In our implementation here, we target and achieve this limit, which to our knowledge, no previously reported implementation has achieved to date due to the hitherto restrictive performance of the field multiplier. We propose a modified Montgomery point multiplication loop based on two steps

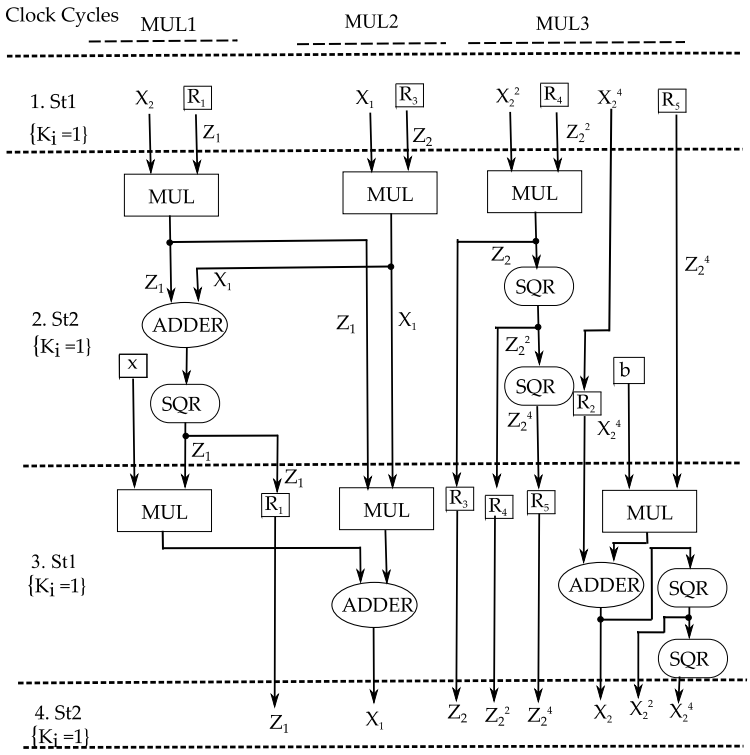


Figure 5.8 Data flow of LLECC for $k_{i+1}=1$, $k_i=1$ and $k_{i-1}=1$

utilizing three full precision multipliers as shown in algorithm 5.4. In each state of the proposed algorithm, three multiplications outputs are concurrently used for additions, square and square over square (*4-square*) to generate the required output for the next states as shown in Figure 5.7. Apart from this, we utilize a smart scheduling to avoid data dependency in the successive loop. We show data flow diagrams to illustrate the point operations for the different combinations of the previous, current and next values of k_i in Figure 5.8 and Figure 5.9.

The data flow diagram shown in Figure 5.8 is for the values of $k_{i+1} = 1$, $k_i = 1$ and $k_{i-1} = 1$. In this case, the point operations of the previous loop, current loop and next loop are the same, hence, there is no transition of the point operations in the successive loops. There are only two states (*st1* and *st2*) for each loop to accomplish the field operations (i.e. multiplication, square and addition) for a point multiplication loop operation. The field multiplication takes 1 clock cycle delay due to one stage pipelining; however, the field square and field adder have only combinational circuit delay and can be performed in the same clock cycle. In Figure 5.8. 1, the data diagram shows the utilization of three full-precision multipliers called *Mul1*, *Mul2* and *Mul3* in each state to accomplish three multiplications. As the multiplier, adder and square circuits are cascaded, we can achieve different field operations in the same clock cycle by tapping the results respectively.

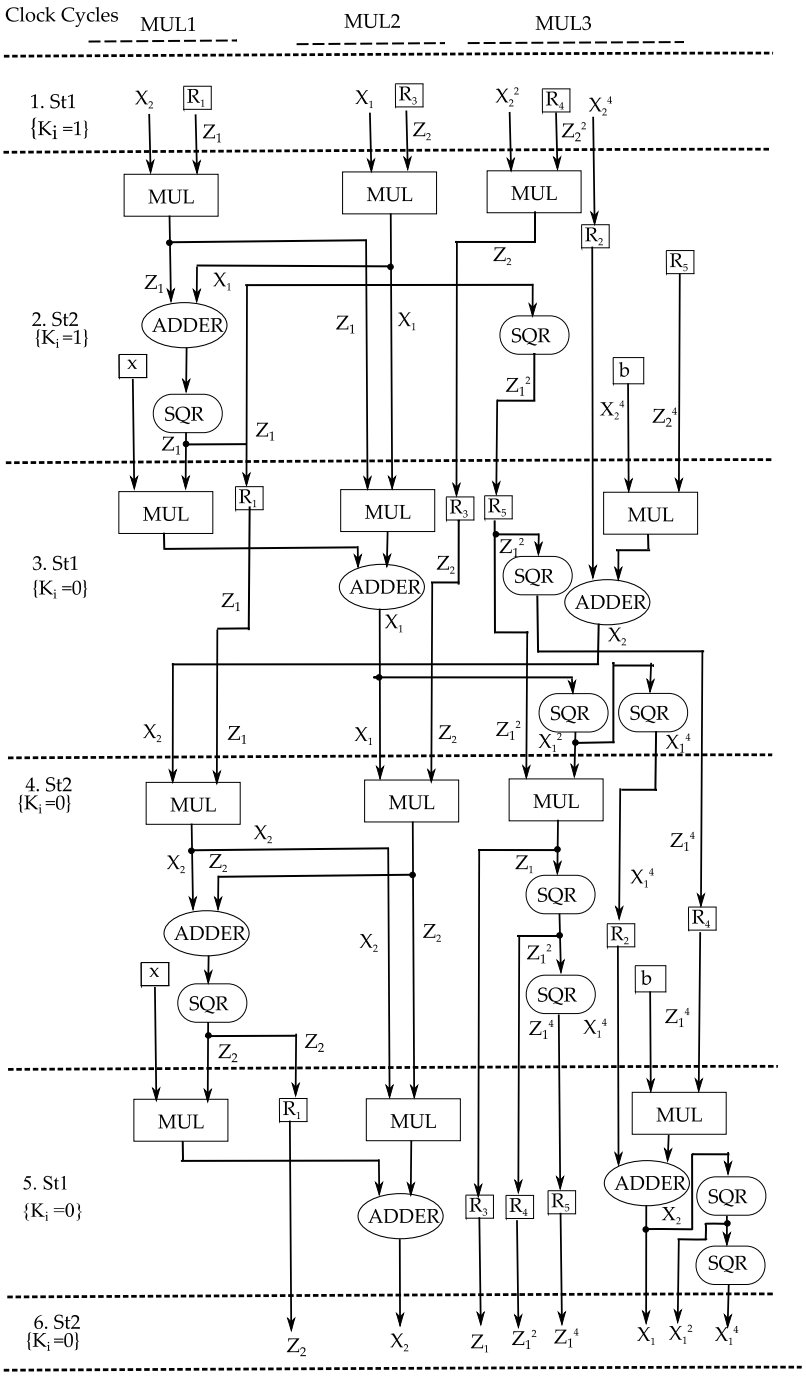


Figure 5.9 Data flow diagram of *LLECC* for $k_{i+1}=1$, $k_i=0$ and $k_{i-1}=0$

- In *st1*, *Mul1* and *Mul2* outputs ($i.e. Z_1 and\ X_1$) are added and squared to get new Z_1 on the fly. The Z_1 is immediately used in the next loop as an input to *Mul1* and also Z_1 is saved in Register 1 (R_1) to use in the next loop. Again, the output of *Mul3* is Z_2 is squared and 4-squared in the same clock to get Z_2^2 and Z_2^4. After then, the three outputs (Z_2, Z_2^2 and Z_2^4) are saved in R_3, R_5 and R_4 register respectively to use in the next loop.

- In state *st2*, we get output X_1 by adding the outputs of *Mul1* and *Mul2* and we also get X_2 by adding the output of *Mul3* and the content of R_2 (X_2^4). The X_2 and its square, X_2^2 are directly applied as an input of *Mul1* and *Mul3* respectively in the *st1* of the next loop and also X_2 is squared over squared (*4-square*) to get X_2^4 output in the same clock cycle is saved in the R_2 for the next operation.

Thus, all inputs that are required to begin the next loop are ready. The dataflow diagram is the same for the combination of values $k_{i+1} = 0$, $k_i = 0$ and $k_{i-1} = 0$ except that the variables are changed as shown in algorithm 5.4.

In Figure 5.9, a data flow diagram of the loop of point multiplication is presented for the values of $k_{i+1} = 1$, $k_i = 0$ and $k_{i-1} = 0$. The diagram shows three consequent loops (for six clock cycles) of data flow to illustrate the transition from the loop of $k_i = 1$ to the loop of the $k_i = 0$.

- In *st1* and*2st2*, the point operations for the value of $k_i = 1$ is performed. As the next loop for $k_i = 0$, the squared outputs of the loop ($k_i = 1$) should be Z_1^2, Z_1^4, X_1^2, and X_1^4 instead of Z_2^2,Z_2^4, X_2^2, and X_2^4. In the loop, Z_1^2 is calculated and saved in R_5 in the *st2*. Again, the output X_1 of the loop will be squared and 4-squared to get X_1^2 and X_1^4 in the *st1* of the next loop ($k_i = 0$).

- In *st1* of the loop of $k_i = 0$ (at clock cycle 3), the X_1^2 is used as *Mul3* input, the X_1^4 is saved in R_2. In the same state, the content of R_5 (Z_1^2) is squared to get Z_1^4 and saved in R_4.Thus, the second loop for $k_i = 0$ can be started with three multipliers inputs X_2 Z_1, X_1. Z_2 and Z_1^2. X_1^2 after the previous loop ($k_i = 1$). In this case, the loop ($k_i = 0$) inputs of *Mul1* and *Mul2* are the same as the inputs of the previous loop ($k_i = 1$) due to the last output (the addition of R_2 and *Mul3)* of the previous loop is X_2; however, the outputs of the multipliers are different than that of the previous loop.

- Now, the final loop is for $k_i = 0$ (at *st5* and *st6*) is similar to Figure 5.8, (no transition) except that the variables are changed as shown in algorithm 5.4.

Thus, the loop of the point operations can be accomplished utilizing only two clock cycles for any set of values of k_{i+1}, k_i and k_{i-1}.

5.6.2 Multiplier with Segmented Pipelining for LLECC

Parallel multipliers are used to reduce latency for point multiplication in ECC implementations and the majority of reported designs in the literature are based on digit serial multipliers instead of bit parallel multipliers [53], [60-61], [72-73]. Bit parallel multipliers take larger area and critical path delay as the size of the multiplier is large due to the large field sizes of the ECC curves [31]. The subquardratic bit parallel multiplier can be suitable for a high speed ECC design, however, pipelining is required to improve speed [71]. The adoption of the pipelining in the proposed 3 multiplier-based ECC is limited as the loop operation takes place within two clock cycles only. Thus, only one stage pipelining can be adopted to improve the performance of the multiplier providing a smart scheduling is devised to overcome the data dependency. The limitation of pipelining is a serious bottleneck for the traditional bit parallel and subquadratic multipliers to achieve significant performance. This is overcome in our proposed segmented pipelining technique by implementing *n* pipelines in parallel, achieving an overall single stage only pipelining as shown in Figure 5.7. This makes the proposed full-precision multiplier suitable for the very low latency loop while the ECC is still maintaining a high performance. The high performance can allow high security ECC curves to be deploy in more applications.

In our proposed low latency ECC (*LLECC*) architecture (as shown in Figure 5.7on page 5-23), we consider *LLECC* implementation over $GF(2^{163})$ where we use three parallel multipliers where each of them is a 163 bit full-precision multiplier with 14 bit segmented pipelining.

5.6.3 Square Circuit, Memory Unit and Control Unit of LLECC

Our proposed least latency ECC (*LLECC*) takes 2 clock cycles for a loop operation of the Montgomery point multiplications. To accomplish 2 clock cycles based loop operation, we need to process the multiplier output in same clock cycle by cascading the adder and square circuits. Thus, in Figure 5.7 (page 5-23), there are several extra adder, square circuits and local registers (R_1, R_2, R_3, R_4, R_5, and R_6) are considered to calculate some instructions of the point operation on the fly as compared to Figure 5.2 (page 5-11). The main memory architecture adopted is the same as that of the distributed based memory of Figure 5.2 (page 5-11) used to enhance speed. Our main memory saves the initial input and the final outputs, and during a loop operation, the memory supplies the constant values (*x, y, b*) as most of the calculated outputs are saved in the local registers to reduce the delay for memory access.

We also use a separate shift register (*k* register) to save the key of the ECC. The shift register shifts 1 bit in every two cycles to generate a new set of values for k_{i+1}, k_i and k_{i-1} used in the control unit as shown in Figure 5.7 (page 5-23). The control unit of the *LLECC* is also based on a finite state machine (*FSM*) that controls the two clock cycles based point operations and is simpler than the control unit of the *HPECC* as most of the operation are performed concurrently.

5.6.4 Critical path delay and clock cycles of the LLEEC

In the proposed low latency ECC (*LLECC*) architecture, we perform several instructions in the same cycle by cascading the multiplier, adder and square circuits as shown in Figure 5.7(page 5-23). The critical path delay of the *LLECC* is the path delay of *GF2MUL* + the reduction part + adder + square + *3x1 mux* as shown in Table 5.2 (page 5-17). The critical path delay can be optimised by selecting the size of *w* through a trial and error approach.

The total clock cycles of ECC mainly depends on the latency of the loop operation of the point multiplication. We achieve *2* clock cycles for each loop operation for the Montgomery point multiplication in projective coordinates which is the theoretical limit of the Montgomery point multiplication algorithm under projective coordinates. Again, the coordinates conversion circuit includes the costly inversion operation. We adopt multiplicative inversion to reduce area and time complexities overheads [65]. As the total latency of the point multiplication in projective coordinates based on the two clocked cycles loop operations is comparable to the latency of the final conversion operation, reducing the clock cycles for the conversion operation is required. The inversion operation involved in the conversion step consumes most of the clock cycles and is thus the focus for optimisation. We use a 4-square circuit to speed up the multiplicative inversion operation. The total clock cycles(CCs) for point multiplications of the *LLECC = 5 CCs* for initialisation + *4 CCs* to start of the loop+ *mx2 CCs* for loop operations

Table 5.3 Latency of the proposed ECC (*MUL= M_1=1, or M_2=2, or M_3=3, ADD=1, SQR=1, and 4SQR =1*)

ECC	Initial + point operations + Conversion	GF(2^{163})	GF(2^{571})
HPECC_1M	$5 + (6M_1)(m-1) + (7M_2$+ Inv1+$3M_3 + 8)$	1099	3783
HFECC_2M	$7 + 4M_1(m-1) + (7M_2$ +Inv2+$3M_2 + 1)$	780	
LLECC_3M	$5 + (4 + 2M_1(m-1) + 4) + (7M_2$ +Inv3+$3M_2 + 3)$	450	-

Inv1= m/2 *CCs* for square + #Mul for inversion x3 +3 *CCs;* Inv2 =m/2 *CCs* for square + #Mul for inversion (M_2) x3 +5 *CCs*; Inv3= m/2 *CCs* for square + #Mul for inversion (M_1) x3.

+ 4 CCs to exit loop+ *CCs* for Coordinates conversion(= (*m/2*) for square + *#mulx1) CCs* for inversion +23 others) as shown in Table 5.3. For example, the total clock cycles for $GF2^{163}$ = *5+4+162x2+4+113(= (81 + 9) +23) = 450* clock cycles.

5.7 Implementation Results

The results of our proposed high speed ECC implementation on Virtex4 (XC4VLX60), Virtex5 (XC5VLX50) and Virtex7(XC7V330T) for *HPECC* and again, Virtex5 (XC5VLX110) and Virtex7 (XC7V690T) for *LLECC* over $GF(2^{163})$, and Virtex7 (XC7VX980T) for *HPECC* over $GF(2^{571})$ using Xilinx ISE 14.5 tool after place and route are shown in Table 5.4 (page 5-35). The presented results are achieved with the use of high speed timing closure techniques. Where feasible the designs have been implemented in each Virtex family. The FPGA size selected was the smallest in the family that could accommodate the design in terms of area and pin count. We used repeated place and route for different timing constraints to achieve the best possible result.

The high performance ECC implementations over $GF(2^{163})$ based on one multiplier (*HPECC_1M*) on Virtex4, Virtex5 and Virtex7 consume 12964 slices, 4393 slices and 4150 slices and can operate at maximum clock frequencies of 210 MHz, 228 MHz and 352 MHz respectively. The achievement of high frequency is due to the design of the high performance field multiplier.

Our proposed HFECC architecture over $GF(2^m)$ based on Montgomery point multiplication is implemented on FPGA. We have considered the same platform to compare with previous presented high speed works. We have used VHDL language to code the ECC model and Xilinx ISE version 14.5 design software to synthesize and Xilinx Virtex5 XC5VLX110 and Xilinx Virtex7 XCV7VX550T for implementation. We consider the elliptic curve over $GF(2^{163})$ to implement for fair comparison of our results with the relevant state of the art.

The implementation results of our proposed architecture are presented in Table 5.4 after place and route (PAR). The implementation consumes 10,363 slices to compute point multiplication in 5.10 µs on Virtex5. The Virtex7 implementation shows the fastest point multiplication to date at 3.50 µs and consumes 8736 slices.

The novelty of the proposed *HFECC* architecture is based on the novel pipelined full precision multipliers, smart point multiplication scheduling, and adoption of proper pipelining in the ECC architecture. In addition, we have exploited distributed logic based memory unit,

dedicated FSM based control unit, 4-squared based repeated square operation for multiplicative inversion for improving the overall *HFECC* performance. The combination of these ideas yielded the remarkable lowest latency figure, compared to previous works, of only 780 clock cycles whilst still achieving a high clock frequency of 223 MHz on Virtex7.

Our Low latency ECC based on three parallel multipliers (*LLECC_3M*) improves speed by reducing latency with an area overhead. The proposed *LLECC* on Virtex7 can manage 159 MHz frequency by consuming the same area of the Virtex5 (113 MHz and 11777 Slices).

5.7.1 Analysis of Results

In our proposed ECC processor, we utilise the popular Montgomery point multiplication algorithm to exploit parallel operations to reduce latency. In the point multiplication algorithm, the loop operation (for each value of k_i) in the projective coordinates requires 6 field multiplications, 5 field squares and 3 field addition operations. The most significant operation in the point multiplication is field multiplication while square and addition are concurrently performing; however, last two multiplication results are required to add to start a new loop is a dependency of the algorithm. The 6 multiplications can be achieved using one multiplier or two parallel multipliers or three parallel multipliers.

In this ECC architecture, we use a novel full-precision multiplier consuming one clock cycle per multiplication; however, two clock cycles is required as an initial delay to start point multiplication. Our proposed *HPECC* architecture utilises one full-precision to complete a loop of the point multiplication by using 6 clock cycles. Again, our proposed *HFECC* uses two parallel field multipliers to reduce further latency. The ECC architecture utilises two stages of pipelining to shorten short critical path delay. The ECC architecture can manage 4 clock cycles for a loop operation; however, the two stages of pipelining are utilised. Our *LLECC* implementation utilises a segmented pipelined based full-precision multiplier consuming one clock cycle per operation due to one stage pipelining. In the *LLECC* architecture, we utilise three multipliers to reduce latency towards the theoretical limit. The ECC architecture can manage two clock cycles per loop operation of the point multiplication.

In Figure 5.10, the bar chart illustrates the area consumption of the three different ECC architecture i.e. *HPECC*, *HFECC* and *LLECC*. The field multiplier of the ECC consumes a bigger area than the other module (for example, memory unit). As the unit of area for ECC each increases as per use of the number field multiplier. Thus, the area for ECC linearly

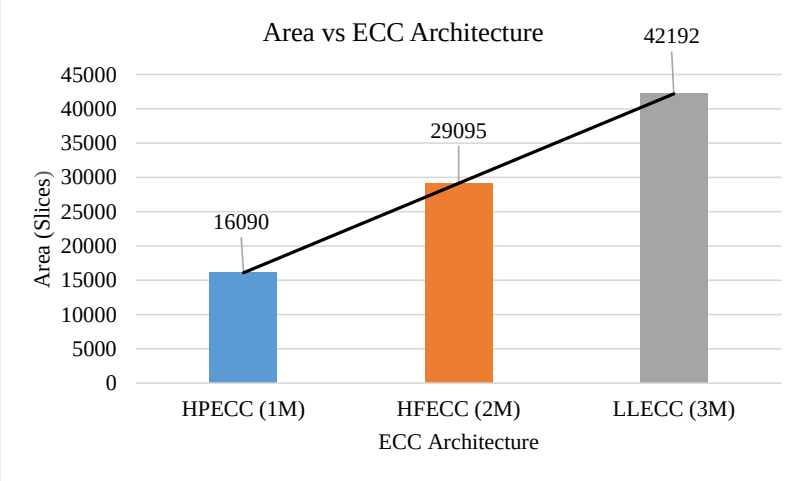


Figure 5.10 Area vs ECC architecture over $GF(2^{163})$

increases. Thus, for a high speed ECC implementation, number of field multiplier consideration dominates resource requirement than square, addition and memory unit

In Figure 5.11, the bar graph demonstrates the frequency of ECC for three different architectures implemented in the Virtex5. The *HPECC* shows the highest frequency (228 MHz) due to a standalone segmented pipelined based Full-precision with lower area complexity than the two stage pipelined based *HFECC* (the Full-precision multiplier exploits one stage pipelining). The *HFECC* is able to sustain the maximum possible frequency (153 MHz) by utilising the two-stage pipeline in the ECC; however, using a one-stage pipeline based multiplier. Hence, the ECC is named high-frequency ECC, *HFECC*. The *LLECC* has one stage pipelining in the ECC architecture with large area complexity shows the low frequency, but it is still comparable to the frequency of the *HPECC* and *HFECC*.

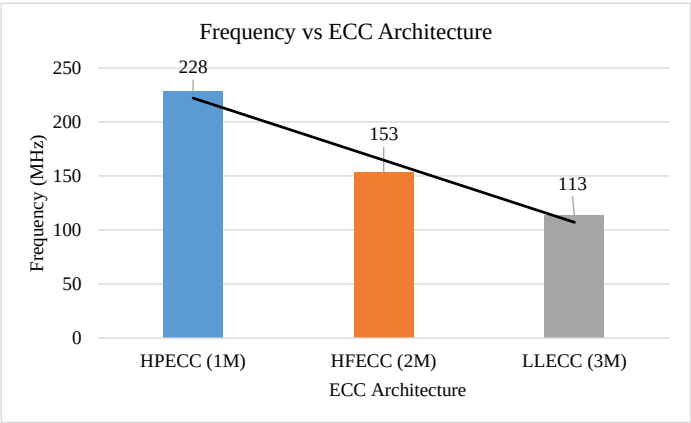


Figure 5.11 Frequency vs ECC architecture over $GF(2^{163})$

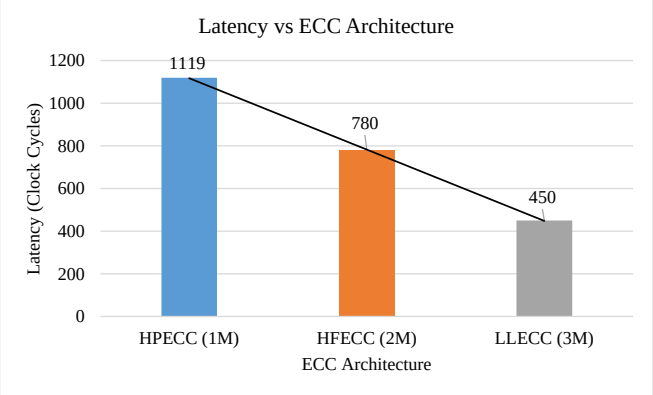


Figure 5.12 Latency vs ECC architecture over $GF(2^{163})$

Latency is the crucial requirement for achieving high speed. The parallel multipliers based ECC (*HFECC* and *LLECC*) that shows very low latency is illustrated in Figure 5.12. The low latency is achieved with an expense of large area based multiplier. The proposed novel pipelined based multipliers can operate at a high frequency to utilise the advantages of the reduction of latency. The *LLECC* shows, in particular, very low latency (450 clock cycles) is the theoretical requirement of a loop operation of the point multiplication. For the $GF(2^{163})$, a loop operation consumes 162x2 = 324 clock cycles . Thus, the latency for the final coordinates conversion is becoming significant can affect the ECC performance. In the ECC, the largest multiplier can reduce latency of point operations instead affecting conversion.

In Figure 5.13. the bar graph shows the point multiplication time of three ECC architectures. The high frequency of *HPECC* contributes to reduce the time for point multiplication sharply. The *HPECC* keeps high frequency by using two stage pipelined based

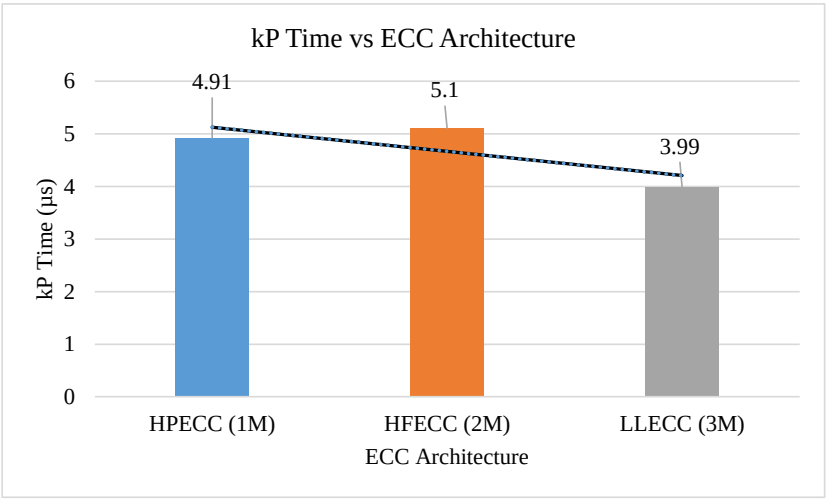


Figure 5.13 *kP* time vs ECC architecture over $GF(2^{163})$

standalone multiplier. The *LLECC* shows the best figure for point multiplication by using the advantages of low latency and the high operating frequency of the multiplier. The largest complexity based ECC can reduces latency but operating frequency will be reducing that affect the overall performance.

The bar chart in Figure 5.14 shows the area-time metric of three ECC architectures. The low value of the area-time shows the best efficiency. The *HPECC* shows the best figure of performance (thus, it is named high-performance ECC) due to high operating frequency and low area complexity. The area-time metric of *HFECC* is comparable to the metric of the *LLECC*. The largest complexity (large multiplier) based ECC can improve *kp* time but showing poor performance. Finally, data dependency can cause worse performance (i.e. HFECC)

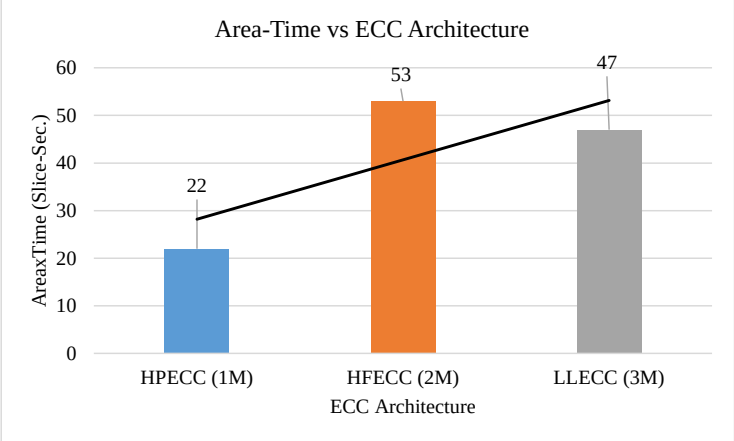


Figure 5.14 Area-time vs ECC architecture over $GF(2^{163})$

5.7.2 Comparison with state of the art

Table 5.4 provides detailed comparison to state of the art using the same technology (for a fair comparison). To evaluate performance in a new technology, the high speed design is implemented on Virtex7 and results included.

Our previous high throughput design presented [83] is the best reported implementation in terms of area-time metric; our *HPECC* implementation presented here over $GF(2^{163})$ on *Virtex7* achieves a better metric value (area-time metric of 13) even using a full precision multiplier. Our high speed ECC implementation, *HFECC* presented in [84] is the new high

Table 5.4 Comparison of the results of proposed ECC with the state of the art over $GF(2^m)$ on FPGA after place and route

Ref.	Slices (Sls)	FFs	LUTs	Freq. (MHz)	kP Time (µs)	Sls x Time x 10^{-3}	Latency, Clock Cycles	FPGA	Resources: Multipliers(Mul)
ECC over $GF2^{163}$									
[59]	4080	1502	7719	197	20.56	84	4050	V4	41 bit Karatsuba Mul
[71]	8095	-	14507	131	10.70	87	1429	V4	163 bit Karatsuba Mul
[10]	16209	7962	26364	154	19.55	317	3010	V4	55 bit mul
[60]	20807	-	-	185	7.72	161	1428	V4	3 Core 82 bit Mul
[53]	24363	-	-	143	10.00	244	1446	V4	3 GNB 55 bit Mul
[72]	17929	-	33414	250	9.60	172	2751	V4	3 Digit Serial 55 bit Mul
[73]	12834	6683	22815	196	17.20	221	3372	V4	2 GNB 55 bit Mul
[80]	8070	-	14265	147	9.70	78	1429	V4	163 bit Karatsuba Mul
[81]	10417	-	-	121	9.00	94	1091	V4	163 bit Karatsuba Mul
[82]	-	-	27889	133	16.00	-	2128	V4	163 bit Karatsuba Mul
[83]	3536	1870	6672	290	14.39	51	4168	V4	41 bit Digit Serial Mul
HPECC_1M	**12964**	**3077**	**23468**	**210**	**5.32**	**69**	**1119**	**V4**	**163 bit Mul**
[61]	6150	-	22936	250	5.48	34	1371	V5	3 Digit Serial 81 bit Mul
[71]	3513	-	10195	147	9.50	33	1429	V5	163 bit Karatsuba Mul
[73]	6536	4075	17305	262	12.90	84	3379	V5	2 GNB 55 bit Mul
[80]	3446	-	10176	167	8.60	30	1429	V5	163 bit Karatsuba Mul
[82]	-	-	18505	199	11.00	-	2189	V5	163 bit Karatsuba Mul
[83]	1089	1522	3958	296	14.06	15	4168	V5	41 bit Digit Serial Mul
HPECC_1M	**4393**	**3090**	**16090**	**228**	**4.91**	**22**	**1119**	**V5**	**163 bit Mul**
HFECC_2M	**10363**	**6529**	**29095**	**153**	**5.10**	**53**	**780**	**V5**	**2x163 bit Mul**
LLECC_3M	**11777**	**3403**	**42192**	**113**	**3.99**	**47**	**450**	**V5**	**3x163 bit Mul**
[83]	1476	1886	4721	397	10.51	16	4168	V7	41 bit Digit Serial Mul
HPECC_1M	**4150**	**3747**	**14202**	**352**	**3.18**	**13**	**1119**	**V7**	**163 bit Mul**
HFECC_2M	**8736**	**6529**	**27105**	**223**	**3.50**	**31**	**780**	**V7**	**2x163 bit Mul**
LLECC_3M	**11657**	**7969**	**41090**	**159**	**2.83**	**33**	**450**	**V7**	**3x163 bit Mul**
ECC over $GF2^{571}$									
[59]	34892	6445	66594	107	133.00	4641	14231	V4	143 bit Karatsuba Mul
[83]	12965	10066	38547	250	57.61	747	14420	V7	143 bit Digit Serial Mul
HPECC_1M	**50336**	**29217**	**141078**	**111**	**34.05**	**1815**	**3783**	**V7**	**571 bit Mul**

HPECC_1M: High Performance ECC with 1 Multiplier, HFECC_2M: High Frequency ECC with 2 Multiplier, LLECC_3M: Low Latency ECC with 3 Multipliers, LUTs: Look_Up_Tables, FFs: Flip-Flops, *kp*: point multiplication,

speed figure on FPGA design to date. Our proposed design, *HPECC* in this work outperforms, *HFECC* [84] in both speed and area-time metrics.

For Virtex4, the previous highest speed implementation is presented in [60] and consumed 20807 slices to achieve 7.72 µs using three 82 bit parallel multiplier cores. Our *HPECC* implementation on Virtex4 consumes 38% less area and shows 31% speed improvement. Again, our work uses less arithmetic (163 bit multiplier) resource to gain 2.33 times improvement in the area-time metric (Slices x Time x 10^{-3}) as compared to the work in [60]. In [72], the authors presented a high speed design that used 17929 slices to attain 9.60 µs for the point multiplication time; meanwhile, our proposed work on Virtex4 is 45% faster than that in [72] and consuming less area by 4965 slices. The work presented in [53] uses three 55

bit multipliers consumed two times the area to achieve 10 µs, whereas our design can show two times better speed. The most relevant work is presented in [71] where the authors using a 163 bit multiplier with four stage pipelining to achieve maximum clock frequency 131 MHz. Our design is based on 163 bit multiplier with two stages pipelining achieved a clock frequency of 210 MHz that is 60% clock frequency (79 MHz) speed up improvement. Again, our ECC implementation (5.32 µs *kp* tim, 12964 slices and 69 area-time metric) is twice as fast with only 60% more slices; this translates to 21% improvement in the area-time metric than the reported efficient design in [71] with 10.70 µs *kp* time, 8095 slices and 87 area-time metric. Our design shows 18% better area-time metric than the previous best optimized design presented in [59] with 20.56 µs *kp* time, 4080 slices and 84 area-time metric. The work presented in [10] used pipelining technique to achieve high clock frequency. Our proposed ECC uses 2 stages pipelining to get 36% improvement in clock frequency speed over [10]. The work in [80] is the previous version of [71] and the work in [81] and [82] are a similar implementation to [71] and [80] with optimised LUTs. In comparison with [80], [81] and [82], our work shows better results than the best results they presented.

For Virtex5, the best reported performance result over $GF(2^{163})$ is 5.48 µs and is presented in [61] with 6150 slices. Our proposed ECC consumes only 4393 slices to compute a point multiplication in 4.91 µs is better in both speed (10%) and area (29%) than that in [61]. Our state of the art achieves double the speed of [71] but consuming only 25 % more slices. The presented work in [73] consumes 6536 slices to get a speed of 12.9 µs; our area-time metric is 3.81 times better than that in [73].

For the new Virtex7 technology, our *HPECC* design shows both the fastest (3.18 µs) and the best area-time metric, reported on FPGA hardware for ECC point multiplication to date. The proposed *HPECC* architecture over $GF(2^{571})$, the highest security NIST curve is the first reported full precision multiplier based implementation and sets a new time record for point multiplication (37.5 µs on Virtex7).

For *HFECC*, we compare our implementation results to the results of the most relevant high performance ECC results in the literature as shown in Table 5.4. The most relevant works are presented in [61], [71], and [73] and are implemented on Virtex5. In [71], Rebeiro et al. proposed an area efficient High speed ECC processor based on full precision pipelined Karatsuba multiplier. They improved the performance of their proposed multiplier using LUTs based pipelining and adopted a scheduling to achieve low latency point multiplications. [71]

implemented their ECC on both Virtex4 and Virtex5 FPGAs. Their Virtex5 implementation takes 9.5 µs which is comparable to their Virtex4 implementation's speed of 10.7µs. The work presented in [61] uses three 81 bit parallel digit serial multipliers to achieve the previously fastest figure of 5.48 µs with the highest clock frequency of 250 MHz for point multiplication on Virtex5. Their digit serial multiplier based ECC however takes 1371 cycles latency due to data dependencies. The work in [73] utilizes three 55 bit multipliers to achieve PM time of 12.9 µs on Virtex5. Our proposed HFECC implementation result shows a new fastest figure of 5.1 µs for point multiplication on Virtex5 despite running at a frequency of 153 MHz.

Several high performances ECC implementations were on the old technology, Virtex4, are reported in [10] [53], [59-60], [71-73], and [81]. For a fair comparison with the works presented on Virtex4, we compare the speed and latency of the works with our results. The fasted reported work onVirtex4 is presented in [60] shows 7.72 µs for point multiplication utilizing three 82 digit multipliers. The recently published ECC hardware in [81] utilized a 163 bit Karatsuba multiplier to achieve 9 µs with a 1091 clock cycles latency to compute PM. Again, the work in [71] consumes 1414 clock cycles and 10.7 µs for point multiplication on Virtex4 which is within only 10% in performance difference compared to their Virtex5 implementation. Again, the work in [72] requires 9.6 µs for point multiplication with a latency of 1751 clock cycles. Our proposed HFECC on Virtex5 shows 34% improved speed when compared to the fastest reported previous work in [60] on Virtex4.

Crucially, our implementation on Virtex 7 achieves the highest speed (3.5 µs) ECC implementation on FPGA to date with a 36% improvement in speed over the previous fastest reported design presented in [61] and has the best area-time performance.

Our low latency ECC (*LLECC*) requiring only two clock cycles for Montgomery point multiplication is the first implementation in the literature with such schedule. The proposed *LLECC* design has the lowest latency figure (450 clock cycles for the curve over $GF(2^{163})$) reported to date while the ECC is still achieving a high clock frequency thanks to the novel pipelining technique in the field multiplier and the smart breaking of the long critical path delay by inserting local registers. Furthermore, the *LLECC* over $GF(2^{163})$ implemented on Virtex7 shows the fastest ever figure for point multiplication (2.83 µs) on FPGA at the theoretical limit of performance. Notably, the minimum clock cycles requirement (theoretical limit) for a loop operation of the Montgomery point multiplication algorithm is two clock cycles to calculate 6 field multiplication, 5 field squaring and 4 addition operations.

5.8 Conclusions

This chapter presented a very high speed elliptic curve cryptography processor for point multiplication on FPGA based on a novel two-stage pipelined full-precision standalone multiplier in *HPECC*, a two-stage pipelined ECC architecture using 2 full precision multipliers in *HFECC* and three one-stage pipelined full-precision multipliers in *LLECC*. In the each of the three cases, a careful scheduling is utilised for the combined Montgomery point multiplication algorithm. The proposed *HPECC* processor has very low latency while the ECC is maintaining a very high clock frequency (353 MHz on Virtex7) and low area. The proposed *HPECC* processor outperforms state of the art both in the speed and area-time metrics. Our implementation *HPECC* on Virtex7 shows the fastest speed (3.18 µs) to date for a hardware ECC implementation for point multiplication.

We propose a novel Montgomery point multiplication based high speed ECC architecture *(HFECC*) over $GF(2^m)$. We exploit novel pipelined full precision multipliers to reduce latency as well as achieve high throughput. A careful scheduling is utilized to avoid data dependency in the main loop of the point multiplication. Moreover, we adopt pipelining registers in the ECC architecture to keep low critical path delay. Our FPGA implementations over $GF(2^{163})$ on Virtex5 and Virtex7 both outperform the reported state of the art. Our Virtex7 based implementation requires 3.5 µs that is a new high speed figure for point multiplication on FPGA.

To our knowledge, the *LLECC* used 3 parallel multipliers to achieve the best latency figure (450 clock cycles) in the literature together with a new milestone for high speed ECC point multiplication (2.83 µs) on Virtex7 which is roughly two times faster than the previous fastest time(5.48 µs) on FPGA.

Chapter 6 Low Latency Multiprecision Arithmetic Circuit based Scalable ECC over $GF(2^m)$

This chapter presents *a low latency hardware implementation of scalable ECC over $GF(2^m)$ to be applied in secure constrained applications. The key requirement to develop a scalable ECC is a multiprecision arithmetic circuit, in particular, a multiprecision multiplier. The research work presents a parallel Comba multiprecision multiplier with on the fly reduction is scalable for all NIST curves. The arithmetic circuit also included low latency square circuit is presented in the chapter. In this chapter, the scalable hardware ECC can meet highest security using same the module is vital for the low-end application.*

6.1 Introduction

Elliptic curve cryptography (ECC) now being rapidly deployed in the public key cryptography (PKC) systems. The shorter key length of the ECC offers low bandwidth and low complexity in the communication system. Thus, ECC based PKC can apply in the low resources applications such as RFID tags, sensor networks and smart card. There are five NIST curves over binary field $(GF(2^m))$ such as $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$. The security of ECC depends on the size of the field. The complexity of ECC implementation increases with the increase of the field. In the near future, the requirement of security is increasing. Thus, high security based communication is required using the same ECC module. In particular, the low resource applications such as battery run devices have resource constraints (area, time and power) to adopt high security curves using existing resources. The area and processing time of the high security curve increase abruptly with the increase of the field size.

In previous chapters, we have reported that high security curve (i.e. $GF(2^{571})$) consumes more resources. For a low resource application, small curve usually suitable, hence, they are mostly considered for low-end cryptography operation. As the security requirement increases over time, a higher security module would be required to replace low security cryptography module. Thus, the critical issues can be solved by designing a low resource scalable ECC. The scalable module can provided high security without replacing the module. Again, the scalable ECC processor requires high latency to provide high security. Thus, a low resource along with low latency scalable ECC can be an ultimate solution.

The main operation of ECC is the point multiplication on the Elliptic curve. The performance of the point multiplication is important to yield an ECC processor to fit, in particular, in cryptographic applications. The point multiplication is accomplished by point addition and point doubling operations on the elliptic curve. The underlying operations of the point operations are the field arithmetic operations such as multiplication, squaring, addition and inversion. The field inversion is the most non-trivial operation, in particular, for the low resources applications. In the low complexity application, multiplicative inversion is applied instead of standalone inversion operation by using multiplication and squaring. The field multiplication is thus important module to achieve high efficiency ECC as the square and addition are linear operations that are simple to implement over $GF(2^m)$.

To achieve scalability of security in the same ECC processor, we need to use multiprecision arithmetic circuit to offer processing of the different elliptic curves in the same

module. There are several scalable ECC processor implementations are reported in [9, 63, 66, 89-93]. The reported scalable works are implemented in the different platform such as software only, hardware/software and hardware only. The main target of the implementations are to optimize the crucial field operations such as field multiplication. The field multiplication is achieved using multiprecision multiplication to operate large operand multiplication in the small processor. Thus, there is word level field multiplication is performed to achieve full operands multiplication using a multiprecision multiplication algorithm. The word level multiplication requires frequent memory operations to load partial products. The most of scalable works are utilised Comba algorithm based multiprecision multiplier. The Comba multiprecision is suitable to reduce memory operations as compared to schoolbook multiplications.

The software implementation of ECC in the embedded processor consumes high latency (order of ten clock cycles) thwart the merits of low die size of embedded processor in the low resources ECC applications. The high latency of a multiprecision multiplication can affect the total point multiplications time. To reduce latency some works using instruction extension sets to do some operations using the integral hardware module (i.e. coprocessor). The coprocessor is the hardware part to perform field multiplication that is controlled by the main processor. The hardware/software design still involves with instruction delay to accomplish ECC operation. Thus, to achieve significant improvement in the point multiplication time, dedicated hardware implementation is preferable. The hardware implementation can be a good solution to reduce the latency as well as reduce the area of ECC to achieve highly efficient scalable ECC processor.

To achieve a highly efficient scalable ECC, there are requirements of modification of field arithmetic algorithm to use the flexibility/scalability of the hardware design. Most of the state of the art fails to reduce latency by exploiting concurrency in the hardware implementation of algorithmic modification [9, 91]. However, some modification is adopted in the [9, 63, 66, 89-93] to modify the multiprecision algorithm to reduce latency; the latency of their processor is high to compute the point multiplication. Moreover, in [66, 93], they utilised large parameter (32 bit instead of 8 bit) to reduce latency. The multiprecision multiplication in 8-bit data path is more complex than that of 32-bit data path based multiprecision multiplier. The requirement of algorithmic modification is thus vital in the 8 bit data path based processor to get efficient ECC.

The scalable ECC over the five NIST curves is complex due to the mix of trivalent and

pentavalent irreducible polynomial based modular reduction operations. For example, some implementation considers only pentavalent to reduce complexity [9, 63, 89, 91, 92]. Again, the complexity is more in the case of ECC with 8-bit data path architecture while the ECC is considering for the 16-bit or 32-bit data path architecture. For example, the implementation in [66, 93] consider 32 bit data path to reduce complexity of control unit. Finally, the complexity is more in the case of hardware implementation than the software implementation due to low level operations. Thus, a hardware implementation of scalable ECC over all NIST curves in the 8-bit data path is still an open problem to improve performance.

In this chapter, we consider Comba multiprecision multiplier and multiprecision square to implement all of NIST curves included $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$ with the 8-bit data path in the hardware platform, FPGA. To achieve high performance scalable ECC, we present a modified Comba algorithm and introduce a novel multiprecision square circuit. Moreover, we develop low area memory unit by avoiding block RAM of FPGA to investigate the actual consumption of the area. The main contributions of the chapter are as below:

- A modified Comba algorithm for multiprecision multiplication is proposed to reduce the latency significantly.
- In the modified algorithm, both loop operations of the Comba algorithm are performed concurrently.
- The proposed multiplier perform as a standalone multiplier without accessing main memory while it is performing multiplication. To avoid the main memory operation, two operands are saved in the local memory.
    - The proposed multiplier utilizes two *w*x*w* (*w* is word size) to perform two loops of Comba algorithm concurrently. We propose a more parallel operations of the Comba algorithm to reduce latency by exploiting very small area overhead. For example, the latency of the proposed multiplier can be reduced sharply further by adopting another set of multiplier (two *w*x*w* multipliers).
    - Another novelty of the modification of Comba algorithm is to achieve on the fly reduction operation. Thus, the latency of the multiprecision multiplier depends on the latency of the Comba multiplication, *GF2MUL* operations.
    - The uniqueness of the reduction operation is that the both trinomial and pentanomial irreducible polynomials based multiprecision multiplication can be achieved on the fly reduction operation. In the NIST curves, the non-zeros terms

of the pentanomial irreducible polynomial are lying on the right most side and hence, the big middle part is zeros. Thus, the pentanomial irreducible polynomial based ECC architecture is simple and regular than the trinomial-based ECC.

- We propose a novel multiprecision circuit square circuit that consumes very low latency to perform square operation with reductions. The proposed square circuit consume only $s+2$ clock cycles where, $s=\frac{m}{w}$, including loading and storing operations.
- We propose a square circuit consumes same latency to perform a square operation for the both cases of the trinomial and pentanomial irreducible polynomial based curves.
- Another originality of the proposed square circuit is that repeated square operation can be performed without any delay for the both pentanomial and trinomial cases. The repeated circuit can accelerate the multiplicative inversion operations, hence, point multiplication of the ECC.
- The adding operation can be performed along with multiplication. Thus, there is not latency for addition in the proposed ECC architecture.
- A novel scalable ECC architecture is implemented to perform the ECC operation over all NIST curves utilizing same work.
- We consider Montgomery point multiplication algorithm due to faster computation of the point multiplication. We utilise a careful scheduling in the point multiplication to avoid dependency. Thus, the loop operation of the Montgomery algorithm depends only on the latency of the field multiplications.
- We utilise a novel SRL16 based memory unit to quantify actual area overhead of a scalable ECC. In general, low resource ECC, in particular, scalable ECC consumes most of the area (50-70% of the ECC) due to register file (memory). The block RAM of the FPGA can be used to save logic cells (slices). Thus, our proposed architecture translates actual figure the scalable ECC architecture.
- We use a separate individual control circuit for standalone multiplier, standalone square circuit and multiplicative inversion operations are based on FSM (Finite State Machine). An FSM based top-level control unit controls the point multiplication.

The proposed scalable ECC is implemented on a low cost FPGA such as Spartan3 and

Spartan6. Notably, the Spartan FPGA family consumes low power than high performance FPGA family such as Vertex. Our proposed 8-bit data path based Scalable ECC can compute point multiplication over all NIST curves such as $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$. Our proposed novel multiprecision multiplier and novel square circuit are utilised to get a low latency arithmetic unit to enable the scalable field operations of the ECC. We implemented parallel version Comba algorithm for different sets of parallel multipliers such as 2 *w*x*w* multipliers, 4 *w*x*w* multipliers and 8 *w*x*w* multipliers to evaluate the merit of parallelism of the Comba algorithm. In this case, the proposed on the fly reduction is used to reduce latency of the proposed multiplier. The reduction circuit can manage on the fly reduction for the both trinomial and pentanomial irreducible polynomial based curves is the first time reported low latency hardware implementation to date. The proposed low latency square circuit consumes only $s+2$ clock cycles for any curves of NIST. The proposed square circuit consumes only *2* clock cycles delay between access of input and load of the output. The proposed square operation is the fastest multiprecision squaring operation to the author's knowledge.

The proposed scalable ECC is implemented in the FPGA to explore the actual area (slices of FPGA) requirement for a scalable ECC without using block RAM and compare with the state of the art. Especially, our proposed ECC consumes very low latency to show better area-time time products as compared to relevant works. Moreover, the proposed ECC outperforms the most relevant works in the speed.

The chapter is organized as follows: Section 2 discusses a background of multiprecision arithmetic circuit and its application in the design of scalable ECC. A description of novel multiprecision multipliers and parallelism of the Comba multiplication algorithm with on the fly reduction is presented in the section 3. In Section 4, a novel squaring circuit is illustrated included a new repeated squaring for all curves. The proposed low latency scalable ECC implementation is demonstrated, including SRL16 based memory unit in the section 5. In section 6. We analyze our results and compare them with state of the art followed by a conclusion in the section 7.

6.2 Background

Public key cryptography based on scalable elliptic curve cryptography is applicable for low resource applications such as RFID tags, wireless sensor nodes and smart card. For the low resources applications, ECC implementation require to meet some constraints mainly area and latency to fit in those applications. Several scalable ECC reported in the literature are

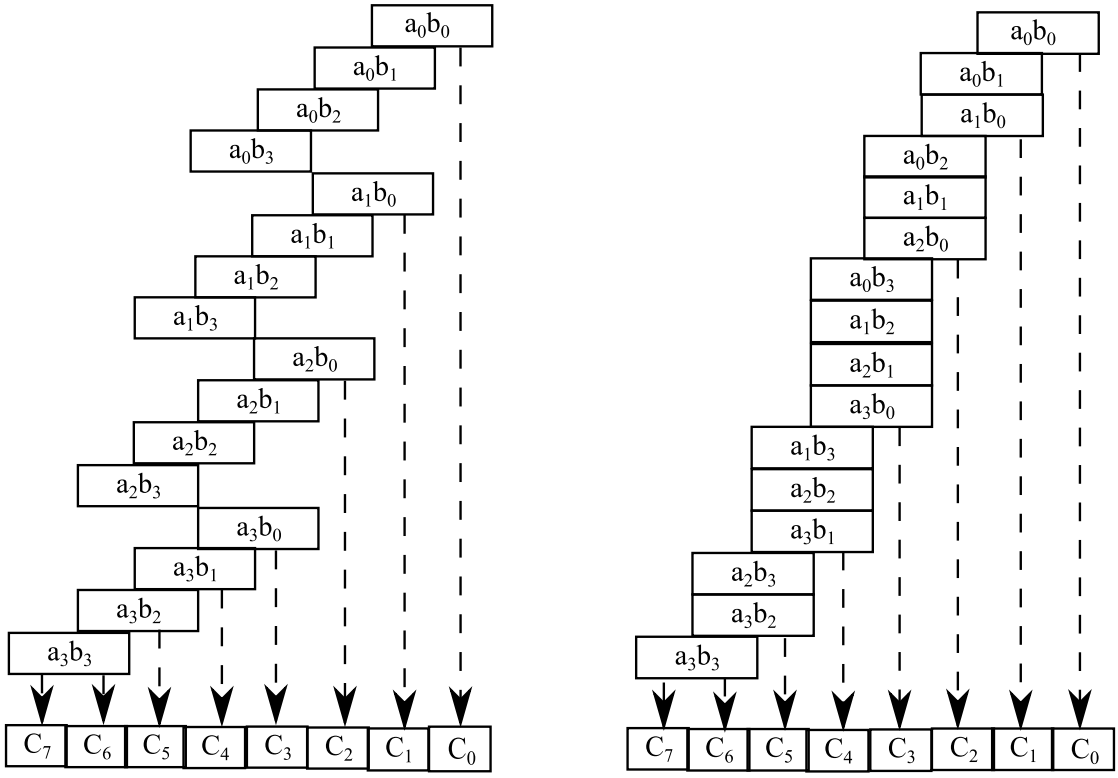


Figure 6.1 Row-wise and Column-wise multiprecision multiplication

implemented in the software, hardware-software and hardware only. In [9, 63, 89, 90, 92], authors presented scalable ECC targeteting for low resources applications. Their works consumes many clock cycles, thus, their ECC consumes large time to compute point multiplication. Again, they did not consider all NIST curves in [9, 63, 89, 91, 92] to reduce complexity except in [90]. The author in [66, 93] consider scalable ECC hardware implementation for server side application by considering 32-bit multiprecision multiplier and large-digit size reduction circuit.

Low resource curve based scalable elliptic curve cryptography (ECC) requires multiprecision arithmetic circuit to enable a flexible high security. There are the field multiplication, square, addition and inversion operations are involved in the point multiplication of ECC. For the low resource application, the area requirement is more profound than the speed of the cryptography processor. The high complexity part is the field multiplier when the multiplier is considered for doing the inversion operation. The multiplication mainly contributes a major area when the inversion is performed by utilizing multiplicative inversion in the point multiplication. To reduce the area of the multipliers, a low resource multiplier such as a bit-serial multiplier, small digit-serial multiplier and multiprecision multiplier is

Algorithm 6.1 Comba multiprecision multiplication of binary polynomials

Input: $A(x) = A_{s-1}, \dots A_1, A_0$and $B(x) = B_{s-1}, \dots B_1, B_0$, each represented by an array of *s* single-precision(i.e *w*-bit) words

Output: Product $C(x) = A(x).B(x) = C_{2s-1}, C_{2s-2}, \dots \dots, C_0$

$(H, L) \leftarrow 0$
For i from 0 by 1 to s-1 do
  For j from 0 by 1 to i do
  $(H, L) \leftarrow (H, L)(A_j . B_{i-j})$
  end for
$C_i \leftarrow L$
$L \leftarrow H, H \leftarrow 0$
end for
For i from s by 1 to 2s-2 do
  For j from i-s+1 by 1 to s-1 do
  $(H, L) \leftarrow (H, L)(A_j . B_{i-j})$
  end for
$C_i \leftarrow L$
$L \leftarrow H, H \leftarrow 0$
end for
$C_{2s-1} \leftarrow L$
Return $C(x) = C_{2s-1}, C_{2s-2}, \dots \dots, C_0$

considered. For the scalable ECC, the bit or digit serial multiplier based ECC consumes large area. Thus, multiprecision arithmetic processor can offer different field sizes to ensure scalable security in the same arithmetic module. The multiprecision multiplier is a key driver to enable high security in the low resources application, for example, wireless sensor nodes and RFID tags. The field of Multiprecision multiplier can be binary field and prime field. The polynomial multiplication in the binary field is suited for hardware implementation as the multiplication is less complex and faster due to carry free operation is involved.

6.2.1 Comba Multiprecision Multiplication over *GF(2^m)*

Multiprecision multiplication can be considered for software or hardware-software or hardware to implement large size (*m*) multiplication. Especially, a low-resource flexible cryptography processor requires a small resource to accomplish large multiplication is now increasingly considered for the hardware implementation to get highly efficient cryptography systems. In this multiplication, the *m* size data divided into *s* number of *w* size (i.e. processor data path) word. Using the multiprecision method, the *w* digit size single precision multiplication (*GF2MUL*) is performed and accumulate the result in the multiprecision techniques to obtain a *m*x*m* result is presented in the *2s-1* number of *w* word size data.

Multiprecision multiplication on the curve-based cryptography over the binary field is accomplished in the two steps. Firstly, word level multiplication and secondly, reduction

operation on the accumulated results. The first part can be mainly standard row base multiplication (pen and pencil method) and standard column based multiplication is shown in the Figure 6.1 respectively. The standard column based multiplication is better than the row based multiplications for flexible (scalable) cryptography processors. Comba proposed a column-based multiplication is presented for binary field in [88] is shown in the algorithm 6.1. There are some other high complexity methods such Karatsuba-Ofman method [69] and row style method (operand scanning algorithm) [87] that are not suitable for low-end application. The Karatsuba-Ofman method involves with divide-and-conquer approach that requires large memory space and repeated memory operations. Again, the operand scanning method has a crucial problem of loading and storing operations of large partial results.

In the multiprecision multiplier the full-length operand, *m* is divided into *s* number of *w* size words, $s = m/w$. Each of *w* size word of an operand multiplies each of *w* size of the other operand. Then, the *GF2MUL* result is accumulated after addition operation with previous results. The accumulated result is 2*w* size. There are total $2s-1$ number of words of multiplier result (*Cs*) at end of multiplication. The $2s-1$ size multiplication result is required to reduce *m* size multiplication result. To reduce the result, the trivalent and pentavalent irreducible polynomial based well-known fast reduction methods are utilised [1]. Typically, in the case of multiprecision multiplication, the *GF2MUL* needs $2s^2$ operations to complete the multiplication without reduction. Again, to store the *GF2MUL* results, it needs $2s$ storing operations. After then, the reduction unit takes large latency to achieve *s* number of words (*m)* size of reduced multiplication result.

Multiprecision multiplier performs on small word size operation is suitable in the device constrained (low power and low area) applications. The Multiprecision multiplier can be implemented in the embedded processor (software), hardware-software and hardware only design. The advantage of multiprecision circuit is to provide high security by implementing large field size of ECC with small resources. The main disadvantage is the increase of latency with the increase of the field size of ECC. The latency of complete field multiplication includes read-write operation, *GF2MUL* multiplication and reduction operations. There several state of the art of multiprecision multiplier are presented to reduce latency by offering parallel hardware implementation [63, 91]. The state of the art presented in [63] considers hardware-software implementation. They consider hybrid–Comba multiprecision multiplication to reduce intermediate read write operation. They utilize a low area without significant improvement in

Algorithm 6.2 Multiprecision squaring over $GF(2^m)$

Input: $a(x) = a_{s-1}, a_{s-2}, \ldots\ldots, a_0\ over\ GF(2^m)$ where $s = \frac{m}{w}, w$ = word size.

Output: $C(x) = (a(x))^2 = C_{s-1}, C_{s-2}, \ldots\ldots, C_0$

St1: $T(u, v) \leftarrow (0.0)$

St2: For I in 0 to s-1 (by one) do

$T(u, v) \leftarrow a_i\ AND\ a_i$

$C(2i) \leftarrow v\ and\ C(2i + 1)\ ) \leftarrow u$

$T(u, v) \leftarrow (0.0)$

end for;

St3: Return $C(x)$

Step4: Return C(x) = (C_{2s-1} , …, C_1, C_0)

the overall latency. In [91], the authors improved latency using standalone hardware implementation with parallel *GF2MUL* multipliers. In [66, 93], the authors targets high speed scalable ECC is implemented in the hardware design. They focus on high speed is implemented in 32 bit data path for server end applications. They mainly improve the latency of *GF2MUL* multiplication using large digit multiplication and the latency of the reduction by considering large-digit size reduction operation.

6.2.2 Multiprecision squaring over $GF(2^m)$

Both digit squaring and full-precision squaring over $GF(2^m)$ in the hardware implementation are free due to simple interleave zero operation. The crucial operation of the squaring is the reduction operation. However, the dedicated (Full-precision) square needs single clock cycle to square an *m* size operand; the dedicated squaring circuit is not suitable for the scalable ECC due to large operand size and different irreducible polynomials (pentanomial and trinomial). The multiprecision squaring with the interleaving zero operation of the input is shown in the algorithm 6.2. The latency of multiprecision square circuit mainly depends on read-write operation and linear squaring by interleaving zero followed by reducing operation [88].

6.2.3 Multiprecision modular reduction over $GF(2^m)$

In the finite arithmetic operations such as multiplication and square operation requires modular reduction using irreducible polynomials, *f(x)* [1, 26]. The modular reduction operation performs on the result of size of *2m-1* bit of multiplication or square operation to reduce *m* bit

size of output. The reduction operation can be bit level or word level and full-precision reduction operations [1]. For the multiprecision arithmetic operation, word level reduction operation is suitable than full precision reduction operation due to area complexity for storage large outputs (*2m-1 bit*). The multiprecision reduction operation can be left-to-right reduction operation or right-to-left reduction operation or fast reduction [88]. For example, the left-to-right word level reduction is more suitable in the software platform than the fast reduction for NIST curves as the middle term, k of $f(x)$ is closer to the other term [88] [1]. In the case of hardware implementation, a word level left-to-right fast reduction can be better than software. The fast reduction method can be suitable to design in the hardware architecture due to that the flexibility of controlling data and the flexibility to modify the design in the hardware [9, 63, 66, 88-93].

6.2.4 Inversion and Multiprecision Addition over $GF(2^m)$

Inversion in the finite field theory can be achieved using a standalone inversion circuit (for example, the greatest divisor algorithm [94]) or using the multiplicative inversion [1]. In the projective coordinates based ECC, the multiplicative inversion operation is performed by using the existing resources (multiplier and square circuits). A basic algorithm for the multiplicative inversion is Fermat's little theorem (FLT) is cheaper than standalone inversion circuit [1]. Itoh and Tsujii in [95] proposed a modified version of the FLT is widely considered for the hardware implementation.

An addition operation in the finite field is simple bitwise *xor* operation of the two operands over $GF(2^m)$; thus, addition operation does not need reduction operation. For an addition in the word level operation, the total latency depends on the latency of data accessing from memory and output storing to memory.

6.3 Implementing Multiprecision Multiplier over $GF(2^m)$

6.3.1 Preliminary of the Multiprecision Multiplier over $GF(2^m)$

Hardware based Multiprecision multiplication can be efficient, in particular, for the low resources application; however, the multiprecision multiplication is widely considered for the software implementation [9, 63, 66, 88-93]. Again, for a low resource with flexible (scalable) cryptography processor, the ECC processor requires a small resource to accomplish large multiplication with low delay (i.e. latency). Thus, the hardware implementation is increasingly popular to consider a highly efficient cryptography system [9, 63, 66, 89-93]. Moreover, for

low power application, the hardware design can meet the requirement by optimizing area, time, and latency.

In this chapter, we concentrate on Comba multiprecision multiplication algorithm over $GF(2^m)$ is presented in [88]. The Comba multiprecision is also called column-wise or product-scanning multiplication is popular for its less number of memory accesses of data as shown in the algorithm 6.1. In the Comba algorithm, the multiprecision multiplier is divided into two nested loops to define the indices of multiplier and multiplicand in the column-wise fashion. Each nested loop has two loops such as outer loop and inner loop. The outer loop indexes the multiplier operand and inner loop indexes the relative multiplicand operand. If the inner loop is operated in a conventional processor, then there are total s^2 operations such as two-load operations in the each loop, a single precision *GF2MUL* operation, and an *xor* operation to get partial sums. After each of the outer loop, a column of multiplication is completed is required to store the result. Thus, there are total latency for a conventional embedded processor is such as $2s^2$ load operations, s^2 of single precision *GF2MUL* operations and a $2s$ number of store operations is shown in the Table 1 in [88] . Hardware based multiprecision circuit can improve latency abruptly as well as increase speed with small resource utilization [9, 63, 66, 90-93] is suitable for low resource application.

There are few hardware-based design of the Comba multiprecision multiplier is available in the literature. In [63, 91] a hardware Comba multiprecision multiplier is implemented to enable a low resource ECC cryptography processor. The work in [91] is a hardware-software ECC implementation while the multiplication is operating in the hardware based multiprecision multiplier. They improve the latency of the *GF2MUL* multiplier by considering additional *GF2MUL* multiplier as per their proposed hybrid Comba algorithm. Again, they use separate reduction unit to reduce the results of multiprecision multiplier by using the right-to-left reduction style. Their total latency for a multiprecision multiplication is $2s^2 + \#gf2mul + 2s + 21 + 3s$ where $2s^2 + \#gf2mul + 2s$ clock cycles for Comba *GF2MUL* where # means "number of" operation and a latency for reduction is $21 + 3s$ clock cycles. The latency for their *GF2MUL* depends on consideration of parallel operation of the multiplier. The work presented in [93] and [66] consider Comba multiprecision hardware multiplier for the high-speed ECC design. They consider 32-bit word size multiplier to reduce latency for both multiprecision multiplication and the reduction of the result. Their design consumes clock cycles of $1+s^2+2$ for Comba multiplication and an additional latency for reduction. Their

architecture started reduction operations start by shifting 160 bits of the multiplier output results. Thus, their proposed reduction unit consumes extra latency along with extra hardware for storing of large output and large multiplexer due to five curves outputs.

6.3.2 Proposed Comba Multiprecision Multiplier over $GF(2^m)$

Algorithm 6.3 Proposed parallel Comba multiprecision multiplication of binary polynomials

Input: A (x) = (A_{s-1}, …, A_1, A_0) and B(x) = (B_{s-1}, …, B_1, B_0) , each represented by an array of s single-precision(i.e w-bit) words
Output: Product C(x) = A(x) . B(x) = (C_{2s-1} , …, C_1, C_0)

Step1: (pH, pL) ← 0	Step1: (qH, qL) ← 0
i' = s-1	i'' =s
Step2: For j from 0 by 1 to s-1 do	Step2:For j from 1 by 1 to s-1 do
(pH, pL) ← (pH, PL) *XOR* (A_j *AND* $B_{i'-j}$)	(qH, qL) ← (qH, qL) *XOR*(A_j *AND* $B_{i''-j}$)
end for	end for
pT_L ← pL	C_s ← qL *XOR* pT_H
pT_H ← pH	qT_H ← qH; qTL ← 0
Step3: For i' from s-2 by 1 to 0 do	Step3: For i'' from 2s-2 by 1 to s+1do
(pH, pL) ← 0	(qH, qL) ← 0
For j from 0 by 1 to i' do	For j from i''-s+1 by 1 to s-1 do
(pH, pL) ← (pH, PL) *XOR* (A_j *AND* $B_{i'-j}$)	(qH, qL) ← (qH, qL) *XOR* (A_j *AND* $B_{i''-j}$)
end for	end for
pT_L ← pL; $C_{i'+1}$ ← pT_L *XOR* pH	If i''= s +1 then
end for	C_{s+1} ← qL *XOR* qT_H
C_0 ← pT_L	$C_{i''+1}$ ← qT_L *XOR* qH
	Else if i''= s +1 then
	qT_L ← qL; $C_{i''+1}$ ← qT_L *XOR* qH
	End if
	end for

Step4: Return C(x) = (C_{2s-1} , …, C_1, C_0)

Polynomial multiprecision multiplication can be utilized by several schemes. The easiest scheme is shift-and-add method that takes long delay to complete multiplications. Comba based multiprecision multiplier is faster as it uses less memory access. The Comba algorithm presented in algorithm 6.1 is generally considered for software implementation. Again, to accelerate the multiprecision multiplication, a hardware implementation of the multiplier is suitable to exploit the parallelism of the Comba algorithm. The hardware multiprecision operation can be optimised more effectively than software platform when it is considering parallel operation in the Comba algorithm. Moreover, the latency of multiplication increases exponentially with an increase of the field size. Thus, the hardware design of the multiplier can offer faster multiplication with small area overhead than software to multiply large operand in the multiprecision style. A duplicating *GF2MUL* multiplier to operate in parallel can reduce the latency for *GF2MUL* operation. The utilization of the parallel multipliers can be one of the several options, including the parallel multipliers to multiply the words in the same column, called mix row-column (hybrid method) and the parallel multipliers to multiply the words of

different columns. The hybrid Comba (row-column mix) method approach of parallelization is presented in the [63-91] to reduce the latency for the Comba *GF2MUL* operation. In their proposed design, the Comba *GF2MUL* multiplication is performed first; after then, they use separate reduction operation to reduce *2s-1* size result into *s* size result.

The parallel operation of Comba proposed in [63-91] can accelerate particular column operation. Their Comba method reduces latency; however, there are several optimisations are required to consider improving latency by using the advantage of parallelism of Comba algorithm such as

- The Comba multiplier outputs are required to reduce in the reduction unit. In the hybrid method, a parallel operation can accelerate a particular column operation. After then, the data is stored until the required outputs are available to start the reduction operation. If the parallel operation increases, then the latency of the *GF2MUL* operation decreases, but the latency of the reduction operation will remain constant. Again, if the parallel multipliers are used to get two separate columns, then the different outputs can be achieved in the same time. The separate data can help to reduce data dependency to start reduction.
- To increase parallel operation, the different column operations may require some extra hardware such as registers.
- The reduction operation in the hybrid method is performed after completing Comba operation. In this case, the large number of products is required to save in the registers to start the reduction. Hence, the method requires a big storage.

Our proposed modified Comba multiprecision algorithm is a column-based multiplication is presented in the algorithm 6.3. In our proposed algorithm, we consider several techniques to utilize the resources in a better way to reduce latency such as:

- Our proposed multiprecision multiplier needs at least two multipliers. The two multipliers multiply two columns from two separate upper loops. Thus, we get two different column results using the two multiplier to reduce dependency in the reduction operation.
- We utilise local registers to supply inputs to multiply. To reduce the latency of loading inputs from main memory to local memory, we consider middle two columns of the Comba algorithm for the first multiplication. We access data from main memory as per

middle column of Comba to save the inputs in the local register. We can get multiplier input after one clock cycle of the loading operation. In this case, loading and multiplication are performed simultaneously.

- After first step, we move to multiply the left most column to last column under the left-side upper loop and the right-side upper loop continues the next column operation as shown in the algorithm. For example, left-side upper loop starts from the C_{2s-1} column and the right-side upper loop starts from the C_{s-1} column of the Comba algorithm. As the most significant outputs are available that are required to reduce the outputs of right-side upper loop, the data dependency is very less to start fast reduction operation.
- The left-side upper loop (column from $C_{2s-1\ to}\ C_s$) is calculated earlier than then the right side upper loop (column from C_{s-1} *to 0*). The proposed multiplier provides the require data concurrently to get reduced output. Thus, multiplication and reduction operation perform concurrently.
- In the proposed method, the reduction operation performs in the word level without storing large number of products. Thus, the proposed method saves area of the memory.
- The reduction operation performs on the fly with Comba multiplication; thus, we manage to reduce the latency of reduction operation.
- The results of the reduction operation are saved concurrently in the memory; thus, the proposed algorithm saves the latency of loading results after multiplication.

Our proposed multiplier needs minimum two *GF2MUL* multipliers to multiply two columns simultaneously. There are two upper loops in the standard Comba algorithm. One of the upper loops is dedicated to compute the products of C_{2s-1} to C_s and the other loop involves to calculate the products of Cs_{-1} to *0*. In our proposed algorithm, two multipliers multiply two columns. One multiplier multiplies a column of Comba from one upper loop, and the other one perform a column of Comba from another upper loop independently. Thus, in our proposed method, the two multipliers generate two products. In the first step, the Comba multiplication start with the columns such as *s*, and *s-1*. The *s* column consists all words of the multiplier and multiplicand. Thus, we can save the two operands in the local memory when it is performing the multiplication. After then, two upper loops included left upper loop, *2s-1* to *s+1* and right upper loop, *s-1* to *0* are performed multiplication in the left to right Comba style. Thus, the carry-word of the column C_s is required to save to use later. In the each loop operation, we get a column of the multiplication result of size of *2w*. The *2w* result includes a *w* size partial

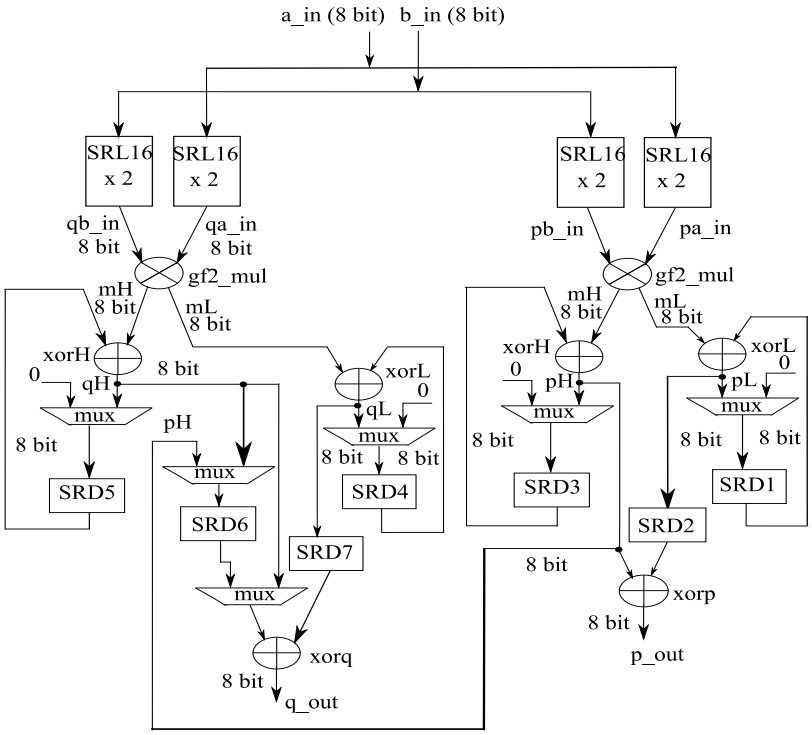


Figure 6.2 Proposed Comba multiprecision multiplier over $GF(2^m)$

product (C_i) and a carry word. Carry word is added to the previous partial product of multiplication in the next loop operation as shown in the algorithm.

In our proposed algorithm, left to right Comba multiplication is considered achieving on the fly reduction. The left-side upper loops are finished earlier than the right side upper loops. Thus, the products of the left side upper loop can be used for the generation of reduction vector to utilise simultaneous reduction operation.

Our proposed algorithm can be modified for more parallel operations. Notably, the proposed basic algorithm involves with the two multiprecision multipliers to compute one column for each of the upper loops. To do the more parallel, we need to adopt two *GF2MUL* multipliers for each case to accelerate the multiplication for a given data path. For example, to achieve parallelization, we need *2*x multipliers where *x* can be 1,2,3,4 or more with an overhead of area complexity. The parallel multipliers of each upper loop perform multiplication on the consequent column at a time. For example, if we use two multiplier for each upper loop, then, the parallel operation is performed on the two columns such as C_i and C_{i-1} where $i = (2s-1$ to $0)$ of the each loop. The limit of the consideration of more parallel operations of the proposed multiplier depends on the area and time tradeoff.

6.3.3 Implementation of Proposed Comba Multiprecision Multiplier on FPGA

We propose a standalone architecture of the multiprecision multiplication circuit to support on the fly reduction. The proposed 8-bit data-path multiplier circuit is depicted in Figure 6.2. The proposed Comba algorithm has two upper loops and each upper loop has one inner loop. Both upper loops complete *2s-1* numbers of inner loop. The left-side circuit performs the inner loops of *2s-1* to *s* (*q-block*) and the right-side circuit performs the inner loops of *s-1* to *0* (*p-block*).

The proposed multiplier architecture consists two *GF2MUL* multipliers to perform multiplication on the two columns where each column for each of the upper loops. In the parallel, each of the multiplier operates independently utilizing separate *2xm* register file. In the beginning step, two operands (*a* and *b*) are loaded in the local register file. The data from local registers are used for multiplication. Thus, we can start the multiplication after one clock cycle of the start of the load operation. The load operation and multiplication operation are performed simultaneously in the first step. In this step, one multiplier (left side) performs the *GF2MUL* operation on *s* column and the other multiplier performs the *GF2MUL* operation on the *s-1* column of the Comba algorithm. We consider the s-1column for the first *GF2MUL* operation as the column consists all *s-1* operands of *a* and *b*. Thus, *a* and *b* inputs are simultaneously loaded in the local register when the multiplication is performed. After 1st step, the *p-block* and *q-block* perform multiprecision multiplication from left to right Comba style multiplication operation. A dedicated finite state machine for each multiplier block. Thus, the separate loops are performed independently (is not shown in Figure 6.2).

The Comba multiprecision algorithm is a column-wise multiplication included multiplication and accumulation. Two *w*-size operands are inputted to each of the multiplier (*GF2MUL*) from local registers and the output of multiplier is *2w* (two word size) result. The *2w*-size result is divided into upper word (i.e. *pH)* and lower word (i.e. *pL*) as shown in the Figure 6.2. The accumulator performs a *2w*-size addition (i.e. *xorH* and *xorL*) using the inputs of current *GF2MUL* result and previous accumulated result. Finally, the addition result is saved in the accumulator registers (i.e. *SRD5, SRD4, SRD3,* and *SRD1*) as shown in Figure 6.2. A sequence of the *GF2MUL* multiplication and accumulation are accomplished in a column of Comba algorithm. Each of the columns produces a *2w* size (for example, 16 bit for 8 bit data path) of the result. At the end of each column operation, the upper word (i.e. *pH)* is used to generate new products and lower word (i.e. *pL*) is stored to add to the upper word of the next

loop output. In particular, the upper word (i.e, *pH* for *p* part and *qH* for *q* part) is performed *xor* with previous lower words (*pL* for *p* part and *qL* for *q* part) to give final multiplication results in the left to right Comba style algorithm. The previous lower words are stored in the temporary registers (i.e. *SRD2* and *SRD7*) along with the clear operations of the accumulator register after every inner loop of Comba algorithm operation. The upper word is always added with the previous calculated lower word except the leftmost column (*2s-1*) *and* right most column *(0th*). In the case of the leftmost column, *pH* of the result is the multiplication result of the most significant column, C_{2s-1}. Similarly, in the case of right most column (*0th*) of Comba algorithm, the lower part of the multiplication output, *pL* is the result of the 0th column, C_0.

In the left to right method of Comba, the column of *2s-2* result is appeared after the multiplication of the column of *2s-3* is completed and so on. Again, in our modified algorithm, the *sth* column result, C_s is calculated by using *pH* of *s-1th* column and *pL* of *sth* column. The *pH* of *sth* column is saved in the temporary resister, *SRD6* before starting of the left-to-right multiplication (from the column of *2s-1* to *s+1*) in the step2 of the proposed algorithm. At the end of the left-side upper loop, after completing *s+1th* column, we get the result of *(s+2)th* column and in the next clock cycle, we get the result *(s+1)th* column by using stored *pH* of *sth* column from *SRD6* (as the *sth* column is already calculated in the beginning step). Thus, our left-side loop (*q* part for *2s-1* to *s*) completes Comba multiplication well before (*s-4* clock cycles) the finishing of the right-side part (*p* part for *s-1* to *0*). The advance calculation of the columns of *2s-1* to *s* reduce data dependency for the fast reduction operation.

6.3.4 Implementing Two-and-two *GF2MUL* based Multiprecision Multiplier on FPGA

We can improve performance of the proposed by parallel operation of the proposed multiprecision multiplier with small hardware resource overhead. We can use additional *GF2MUL* multipliers without using extra *2m*–size register file. We propose a parallel multiplier architecture with two parallel multipliers for each of the upper loop of Comba algorithm is depicted in Figure 6.3. The multipliers perform the *GF2MUL* operation on the two consequent columns of each upper loop. We can manage two different sequences of the two consequent columns by using a delay circuit. For example, if there are two consequent columns such as *s-1th* column and *s-2th* column then, the sequence of inputs of the *s-1th* column are a_0b_{s-1}, $a_1b_{s-2, \ldots}$ $a_{s-1}b_0$. Again, the sequence of inputs of the *s-2th* column are a_0b_{s-2}, $a_1b_{s-3, \ldots}$ $a_{s-2}b_0$. The *s-1th* column has one more sequence than *s-2th* column. We can use a simple delay circuit to delay

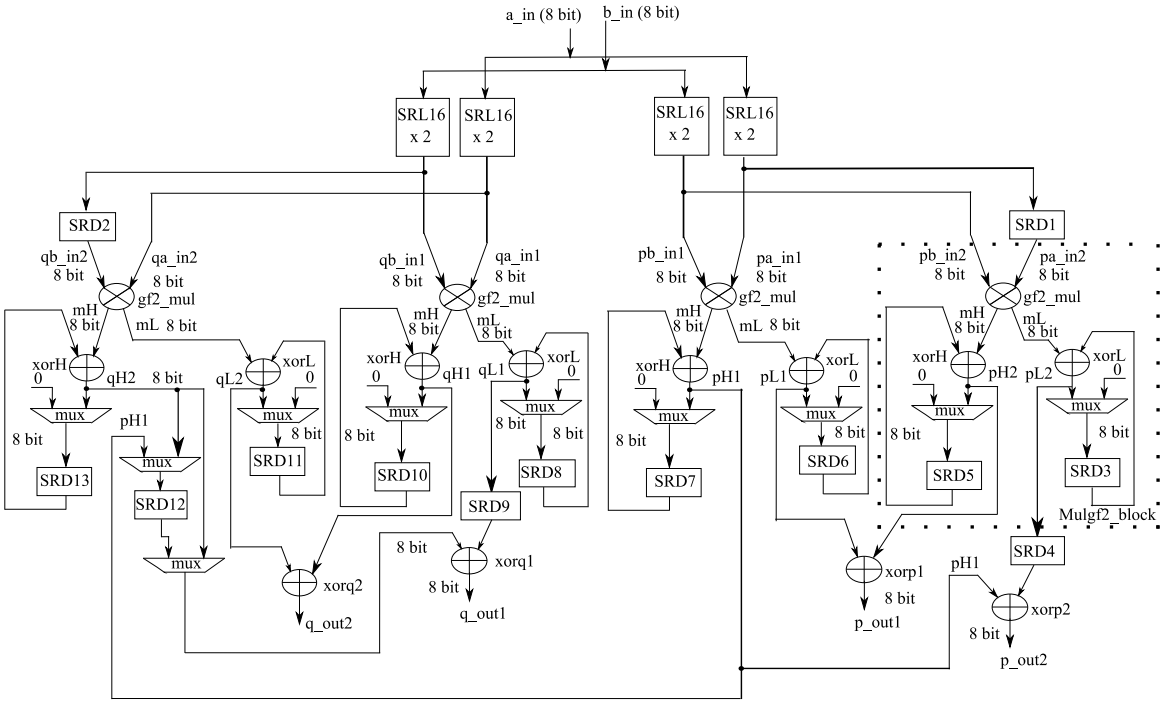


Figure 6.3 Proposed two-and-two *GF2MUL* based Comba multiprecision multiplier over $GF(2^m)$

the '*a'* input for one clock cycle. The delayed 'a' input is aligned with 'b' input to maintain the sequence of inputs of the *s-2th* column. Thus, we need to put a register to delay '*a'* input in the right-side upper loop. Similarly, we need to put a register delay on the 'b' inputs of the left-side upper loop to get the required sequence of inputs of the next column (the column with a smaller sequence of inputs). In the right side, we use a delay (*SRD1*) on the input of '*a',* whereas in the left side, we use a delay (*SRD2*) on the input of '*b'*. Thus, the smaller column is started the *GF2MUL* operation after one clock cycle of the column of longer sequences. The both columns are finished *GF2MUL* operations at the same time.

According to our proposed algorithm, in the first step (loading and multiply), the proposed architecture completes the *GF2MUL* operation on the column of *s+1* and *s* in the left-side upper loop and *s-1* and *s-2* in the right-side upper loop. We get multiplication results of column of *s+1th*, *sth* and *s-1th* in the beginning step. We save *pH* of *s+1* in a temporary register, *SRD12* as shown in the Figure 6.3. After then, the left to right Comba style multiplications is performed by considering two columns in each time (using two multipliers) from the leftmost columns of left-side upper loop such as *2s-1th* and *2s-2th* and so on. In the same time, two other multipliers are multiplying two consequent columns of the right-side upper loop such as *s-2th* and *s-3th* and so on. Thus, two inner loops of an upper loop are performed at each iteration.

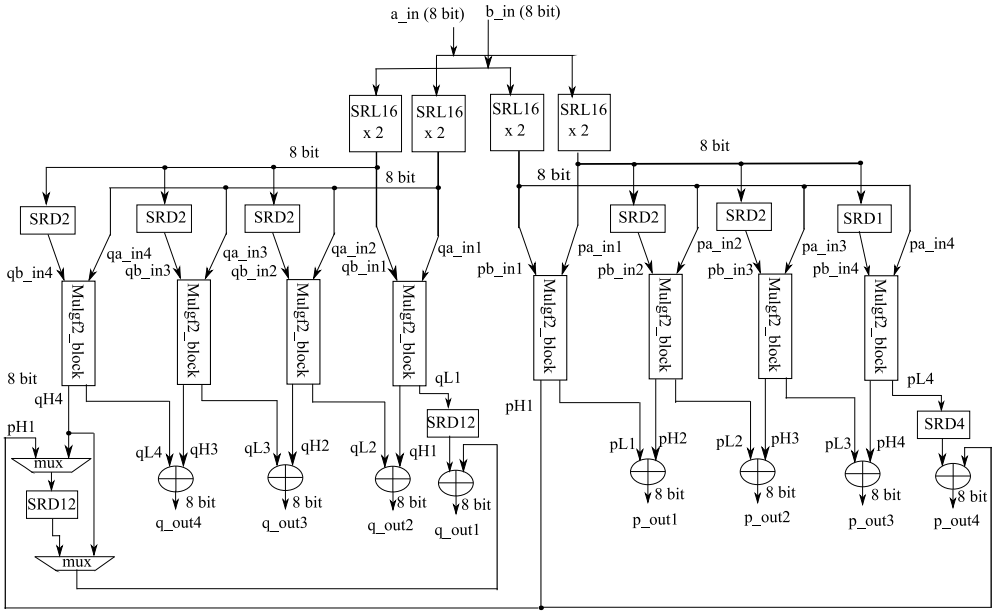


Figure 6.4 Proposed four-and-four *GF2MUL* based Comba multiprecision multiplier over $GF(2^m)$

6.3.5 Implementing Four-and-Four *GF2MUL* based Multiprecision Multiplier on FPGA

A further parallel architecture is proposed utilizing 4 multipliers for each of the two upper loops of the Comba algorithm is shown in Figure 6.4 In the proposed architecture, we extend the proposed architecture in Figure 6.3 to improve performance and also evaluation of the proposed parallel approach. In this architecture, the inputs of *a* of the longer-sequence column are delayed by 1, 2, and 3 clock cycles in p-block to retain the sequences of the 4 consecutive columns of the right-side upper loop of the Comba algorithm. Similarly, *b* inputs of the column with longer sequences in left-side upper loop (the q block) is delayed by 1, 2 and 3 clock cycles to maintain sequence of the consecutive 4 inner loops (columns) of left-side upper loop. Thus, the proposed architecture can produce 4 multiplication results in each of two blocks in each iteration. The circuits can calculate *GF2MUL* operations with ½ of latency of the proposed architecture of Figure 6.3 and with ¼ of latency of the architecture proposed in Figure 6.2 (page 6-16) with small resource overhead (registers for delay and multiplexers to serial outputs).

6.3.6 Implementing On-the-fly Reduction Unit for Multiprecision Multiplier on FPGA

We propose a novel architecture of reduction circuit as an integral part of the multiprecision multiplier over $GF(2^m)$. In this architecture, we propose a multiprecision reduction unit to reduce *2s-1* number of words of multiplier result to *s-1* number of words of multiprecision multiplier output in the left to right fast reduction approach. The proposed reduction circuit can be implemented to accomplish fast reduction method over all the binary

Algorithm 6.4 Fast reduction modulo $\int(x) = x^{163} + x^7 + x^6 + x^3 + 1$ (with $W = 8$))

Input: A binary polynomial degree of at most 324 $c(x)$ is divided into two binary polynomials: $q(x)$ of degree from 163 at most 324 are q[20],…, q[1],q[0]= (p[20] & x07) and $p(x)$ of degree from 0 to at most 162 are p[20], p[1],p[0].
Output: $c(x)$ mod $\int(x)$.

Step1:	For i 20 down to 0 and q_{-1}, $q_{-2 = x"00"}$ are extra q inputs after q[0]

	Lv <= q[i]<<5
	Mv <= q[i]<<4 xor q[i] xor q[i]<<3 xor Lv
	Fv <= q[i]<<5 xor q[i]<<4 xor Mv
	If i=< 18 then
	For j 20 down to 0
	If j=< 1 then c(j) <= Fv[j] xor p[j] xor Ev (j)
	Else
	c(j) <= Fv[j] xor p[j]
	end
	end
	Ev(1) <= (00$q_{325}q_{324}q_{323}q_{322}q_{321}q_{320}$) xor (0000$q_{325}q_{324}q_{323}q_{322}$)
	Ev(0) <= ($q_{319}q_{319}$00q_{319}00q_{319}) xor (q_{321}(q_{325} xor q_{322} xor q_{320})(q_{324}xor q_{322} xor q_{321}))(q_{323}xor q_{321} xor q_{320})(q_{322} xor q_{320}) (q_{325} xor q_{322} xor q_{321}))(q_{324}xor q_{321} xor q_{320}) (q_{323}xorq_{320}))

Step2:	Return c[20] , …c[1],c[0]

elliptic curves of NIST such $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$. The fast reduction operation is performed in the word level using the available outputs of the multipliers. As the left-side upper loop outputs are available before the right-side upper loop outputs, thus, the reduction operation is performed on the fly along with the Comba multiprecision multiplication.

The reduction operation over $GF(2^{163})$ and $GF(2^{233})$ are discussed in Figure 6.5 and Figure 6.6 (page 6-24)to demonstrate pentanomial and trinomial based reduction operation respectively. In the pentanomial based reduction polynomial, the second highest non-zero terms of the NIST fast-reducible polynomials are lying closer to each other of the non-zeros terms. As the position of highest second-nonzero term is at the 12^{th} bit for 571, 10^{th} bit for 283 and the 7^{th} bit for 163, the dependency of data is very low to generate reduction vectors (to add to the outputs of left-side upper loops). For example, three consequent outputs (3 words for 8-bit data path) of the left-side loop in 571 are required to generate reduction vectors.

In the Figure 6.5, the inputs (p and q) of the reduction circuit are the p and q outputs of *p-block* and *q-block* of the multiprecision multipliers respectively. The q inputs are inputted from C_{2s-1} to C_s and the p inputs are inputted from C_{s-1} to C_0. In the case of the parallel multiprecision multipliers, the parallel inputs of p and q are required to convert parallel to serial

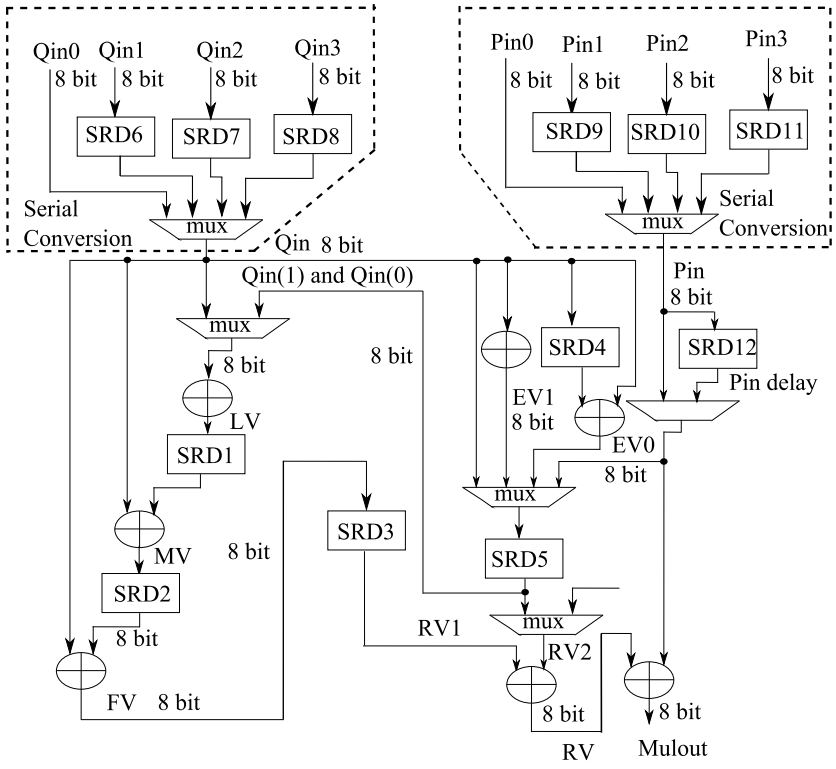


Figure 6.5 Reduction unit of multiprecision multiplication

as shown in Figure 6.5. Our proposed pentanomial based reduction operation over $GF(2^{163})$ is presented in the algorithm 6.4. The proposed reduction architecture consists two main blocks. The left side (main block) block generates the regular reduction vectors such as *Lv*, *Mv* and *Fv* vectors as shown in the algorithm 6.4. The extra vectors such as *Ev1* and *Ev0* vectors are also generated in a separate block followed by saving in a local register file. The extra vectors are used to reduce the least consequent words (*Ev1* and *Ev0* vectors over $GF(2^{163})$ are used to reduce 1^{st} and 0^{th} words) as shown in the algorithm 6.4.

For each of the *q* inputs, new *Fv*, *Mv* and *Lv* vector is generated. The typical vectors of *Fv*, *Mv* and *Lv* for a *q* input, q_y as shown in a tabular form in the Table 6.1. In the Table 6.1, the vector, *Lv* for a *q* input (q_y) is saved in the delay register to add to next partial *Mv* vector (will be generated by the q_x input) to get a next *Mv* vector. The current *Mv* vector is the addition of the partial *Mv* vector of q_y and the old *Lv* vector (generated by q_z). The current *Mv* vector is

Table 6.1 *Lv*, *Mv*, *Fv* vectors generation over $GF(2^{163})$

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FV								*MV*								*LV*							
			q_z								q_{y7}	q_{y6}	q_{y5}	q_{y4}	q_{y3}	q_{y2}	q_{y1}	q_{y0}	q_x				
q_z								q_{y7}	q_{y6}	q_{y5}	q_{y4}	q_{y3}	q_{y2}	q_{y1}	q_{y0}	q_x							
q_z					q_{y7}	q_{y6}	q_{y5}	q_{y4}	q_{y3}	q_{y2}	q_{y1}	q_{y0}	q_x										
q_z				q_{y7}	q_{y6}	q_{y5}	q_{y4}	q_{y3}	q_{y2}	q_{y1}	q_x	q_x											

saved in a delay register. Similarly, the current *Fv* vector is the addition of the partial vector of the q_y and the old *Mv* vector. The *Fv* vector is saved in the *SRD3* to add to the *p* input to generate a reduced multiplier output (except the last several outputs as shown in the algorithm 6.1) as shown in the Figure 6.5. To generate last two final vectors over $GF(2^{163})$, we need to add *Fv* vectors with the respective extra vectors (i.e. *Ev1* and *Ev0*) to generate final vectors to add with the last two *p* inputs as shown in the Figure 6.5. Notably, the extra vectors, *Ev1* and *Ev0* are generated by using C_{2s-1} and C_{2s-2} of *q* inputs. The extra vectors are generated and saved in the *SRD5* in the beginning of Comba multiplication. Finally, the proposed reduction operation is performed concurrently with the Comba multiprecision multiplication.

The reduction outputs depend on the generation of *Fv* vectors. To generate a *Fv* vector, the required *q* inputs should be available well before (i.e. 3 consequent words for the case $GF(2^{163})$) the targeted *p* input. If the *p* input is available before the required *Fv* vectors, then the *p* input is delayed by using a delay register. In the case parallel multipliers (Figure 6.3 and Figure 6.4) based Comba operation, the delay may incur few clock cycles is negligible as compared to total clock cycles for a multiplication.

The trinomial based fast reduction operation of over $GF(2^{233})$ and $GF(2^{409})$ is also performed in the left-to-right fast reduction. The crucial problem in the reduction operation using the trinomial irreducible polynomials is the difference between the consequent non-zeros terms in the polynomial. For example, the second highest non-zero bits of the trinomial reduction polynomials for $GF(2^{233})$ and for $GF(2^{409})$ are at 74^{th} bit and 87^{th} bit respectively. The difference incurs extra latency to complete reduction operations. We propose a reduction method over a trinomial curve to achieve on the fly reduction is shown in the algorithm 6.5. In the algorithm 6.5, we present reduction operation over $GF(2^{233})$ which is implemented in the hardware as shown in Figure 6.6. In the figure, the multiplier inputs are inputted as p inputs from 232 bit to 0 bit (in words, 29 to 0 as shown in the algorithm 6.5) and q inputs from 465 to 233 bits (in words, 29 to 0 as shown in the algorithm 6.5). As the q inputs are available as a sequence of words of 29,…, 0; the first reduction-vector generation depends on the 19th word due to second non-zero term of the polynomial. We need to delay 29 to 20 words (in the *SRD1* as shown in Figure 6.6) to align with the sequence of 19,.., 0 to start vector generation (in step1

Algorithm 6.5 Proposed fast reduction modulo $\int(x) = x^{233} + x^{74} + 1$ (with $W = 8$)

Input: A binary polynomial degree of at most 465 $c(x)$ is divided into two binary polynomials: $q(x)$ of degree from 233 at most 465 are q[29],…, q[1],q[0]= (p[29] & x03) and $p(x)$ of degree from 0 to at most 232 are p[29], p[1],p[0].
Output: $c(x)$ mod $\int(x)$.

```
Step1:   For  i= 29 to 0 then by one
          If i= 29 to 20 then
          Lv[i]<= q[i]
         T[t] <= q[i]
          end if
           if  i= 19 to 0 then
           Lv[i]<= q[i]
         end if
         end for
Step2:   For n= 29 down to 0 then by one
         If n= 29 to 11then
         Mv <= Lv[n]  <<7 xor q[n-10]<<1          {n-10= 20,…,1}
         Fv[n]  <= Lv[n] >>1 xor q[n-10]>>7 xor Mv
         End if
Step 3:  If n= 10 then
         Mv <= Lv[n]  <<7 xor q[n-10](6 t0 1) &0x"00")
         Fv[n]  <= Lv[n] >>1 xor q[n-10]>>7 xor Mv
         End if
Step4:   If i=9 to 0 then
         Mv <= Lv[n] <<7
         Fv[n]  <= Lv[n] >>1 xor Mv
         End for
Step5:   For j= 29 down to 0 by one
          If j= 29 to 19 then
          c(j) <= Fv[j]  xor p[j]
            else If j= 18 to 10 then
             c(j) <= Fv[j] xor p[j] xor Ev1(j-9)
                else if j=< 9  to 0 then
                c(j) <= Fv[j] xor p[j] xor Ev1(j-9) xor Ev0 (j)
         end if
Step 6:  For m= 9 down to 0 by one
         T2 <= T[m]<<2
         Ev1[m] <=T[m-1]>>6 xor T2
         Ev0[m] <=T[m]
         End for
Step7:   Return c[29] , …c[1],c[0]
```

and step2 as show in the algorithm 6.5). Again, we to save 29 to 20 words (in the *SRD5* as shown in Figure 6.6) to generate extra vectors (*Ev1* and *Ev0* as shown in the step1 of algorithm 6.5).

A reduction vector (*Fv*) is generated by using current input and previous input *q*. The reduction vector is added with the *p* inputs to get reduction output. The *Mv* vector is generated using current input followed by saving in the delay register. The delayed *Mv* vector is added to current input to generate *Fv* vector. Thus, the *Fv* vector is the resultant addition of current input and *Mv* vector.

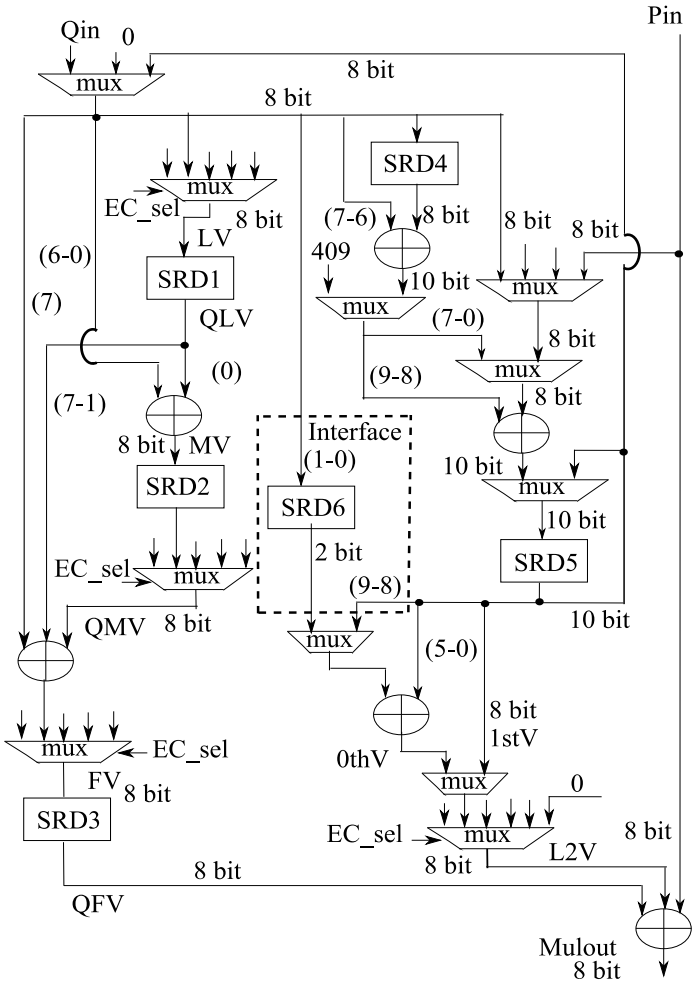


Figure 6.6 Scalable multiprecision multiplier reduction over $GF(2^{233})$

The final reduction outputs from 29 to 20 depend on the *Fv* and *p* inputs. The extra vector *Ev1* (1st vector as shown in the Figure 6.6) is required to add during 19 to 9 and extra vector *Ev0* (0th vector as shown in the Figure 6.6) is required to add during 9-0 inputs of *p* inputs as shown in the algorithm 6.5. At the *9th* input, the *Ev1* and *Ev0* (both are exclusive each other) added with *p* inputs. We need to remove undesired bits of *Ev1* and *Ev2* (in step 3) as shown in the 'interface' of Figure 6.6. As the *q* inputs of the reduction unit such as 29 to 20 are inputted in the starting, thus, the extra vectors such as *Ev1*and *Ev0* are generated concurrently to add with the targeted *p* inputs as shown in the algorithm 6.5. However, the both *q* and *p* inputs are inputted concurrently; the *q* inputs are calculated (due to a lower sequence of column) well before the *p* inputs. The required reduction vector *(Fv)* is calculated followed by saving in the *SRD3* to add to the respective *p* input. Thus, the trinomial-based reduction algorithm can manage on the fly reduction with the Comba multiprecision multiplication.

The reduction algorithm for $GF(2^{409})$ is similar to the $GF(2^{233})$ except the power of the second non-zero element of the irreducible polynomial.

6.3.7 Analytical Comparison of multiprecision multipliers on FPGA

The proposed multiprecision multiplier with reduction is implemented on Sparatan3 and Sparatan6 FPGAs to evaluate the performance of proposed parallel architectures using Xilinx tool, ISE14.5. In the Table 6.2 shows the theoretical latency and in the Table 6.3 presents resources (Slices, Luts and FFs after Synthesis), maximum frequency and latency over $GF(2^{163})$ of the proposed multiplier *(Mul(1&1)* and its parallel forms such as *Mul(2&2)* and *Mul(4&4)* respectively.

Table 6.2 Latency of the proposed multiprecision multiplier

Arithmetic Operation	#load	#Comba Mul	#Reduction	# Clk Cycles(s=21)
Multiprecision Mul with 2 Multipliers (*Mul(1&1))*	1	((s(s+1))/2) + 2	0	234
Multiprecision Mul with 4 Multipliers (*Mul(2&2))*	1	[{((s+1)/2)(s +1)}/2] +3	0	125
Multiprecision Mul with 8 Multipliers (*Mul(4&4))*	1	([{(s+3)/4}(s +1)]/2) +7	3	77

Table 6.3 Area and maximum frequency of the proposed multiprecision multiplier over $GF(2^{163})$ on FPGA

FPGA	Slices (Sls)	LUTs	FFs	Max. Freq, MHz	# Clk Cycles(s=21)
Mul(1&1)					
S3	391	652	120	103	234
S6	162	444	137	172	
Mul(2&2)					
S3	611	1014	130	102	125
S6	238	701	145	176	
Mul(4&4)					
S3	891	1417	139	97	77
S6	251	918	139	162	

S3- Spartan 3, S6- Spartan 6

In our proposed architecture, the latency to complete a multiprecision multiplication with reduction depends on the latency of the *p* inputs provided by Multiplier. For our proposed architecture of Figure 6.2 (page 6-16), (*Mul(1&1))*, provide *p* inputs over $GF(2^{163})$ in the sequence of clock cycles such as 1, *22,20,19,18,…,3,2,1,and 1*. At the start of the multiplication, we need 1 clock cycle for loading in the local register and in the beginning step, we need *22* clock cycles where *21* clock cycles for Comba operation in the right to left style and to switch left to right style of Comba, it takes 1 extra clock cycle. After then, the Comba multiplication follows a sequence as like as *20,19,…3,2,1*. Finally, we need 1 extra clock cycle to get the final *p* multiplication result (right most column). We can write the sequence as like

as (*21+20+19+…+ 3+2+1)+3.* Thus, Our total latency of the multiprecision multiplication with reduction of the proposed *Mul(1&1)* is (s(s +1)/2) +3 ≈ $s^2/2$ where $s^2>>3$.

The parallel architecture as shown in the Figure 6.3 (*Mul(2&2)*), provides *p* inputs(two inputs at a time) in a sequence such as *22, 19, 17… 5, 3, 1,1,* and *1*. In the Figure 6.3 (page 6-19), the proposed architecture takes 1 clock cycle for loading, 1 clock cycle for switching Comba style and 1 clock cycle for final output as like as Figure 6.2 (page 6-16) and the Figure 6.3 takes 1 extra clock cycle for parallel to serial conversion. Therefore, in our proposed architecture in Figure 6.3, the *Mul(2&2)* takes (*((s/2)(s +1))/2) +4* clock cycles.

Finally, our proposed architecture in Figure 6.4 (page 6-20) (*Mul(4&4))* follows a sequence of *p* inputs such as *25, 17, 13, 9, 5, 1, 1, 2,* and *3.* Similar to previous architecture, the proposed multiplier consumes 1 clock cycle for loading. After then, the multiplier takes 25 clock cycles for the first 4 columns of the proposed Comba algorithm. In this step, the Comba operation performs multiplication in the right to left style. After then, the Comba multiplication is performed in the left to right style. Thus, out of 25 clock cycles, 3 clock cycles are used to convert parallel to serial and to switch the multiplication style. In the last step, 2 clock cycles are used for serialisation of the multiplier results to apply in the reduction unit. Apart from this extra delay in the multiplication, there is a delay in the reduction unit to generate reduction vectors for the last three outputs because of delay reduction vector generation (the last *q* inputs and p inputs are available in the same time). Our reduction unit takes 3 clock cycles to generate respective reduction vectors to reduce last three *p* inputs. Thus, *Mul(4&4)* consumes the total latency for multiprecision multiplication with reduction is (*(((s+3)/4)(s +1))/2) +11* clock cycles.

We illustrate the results of the multipliers are shown in the Figure 6.7 as area vs proposed multiplier, latency vs proposed multiplier and max. frequency vs proposed multiplier over $GF(2^{163})$. In the Figure 6.7, the *Mul(1&1)* included 2x8bit *GF2MUL* for the Comba operation and reduction circuit consumes 391 slices. The parallel multipliers, *Mul(2&2)* and *Mul(4&4)* consumes 611 and 891 slices respectively. The *Mul(2&2)* uses twice as resources of *GF2MUL* as the resources of *Mul(1&1) and* The *Mul(4&4)* uses four times more resources of *GF2MUL* than *Mul(1&1).* The parallel version consumes comparatively less slices to provide better latency performance as shown in the Figure 6.7. For example, The *Mul*(2&2) consumes very low latency, 125 clock cycles and the *Mul(4&4)* utilizes only 77 clock cycles. Thus, the latency reduces abruptly by consuming small resources. The frequency of the three multipliers

is similar as shown in the Figure 6.7. Finally, the latency can be reduced significantly with small overhead of resources; thus, the proposed parallel operation can improve performance of ECC.

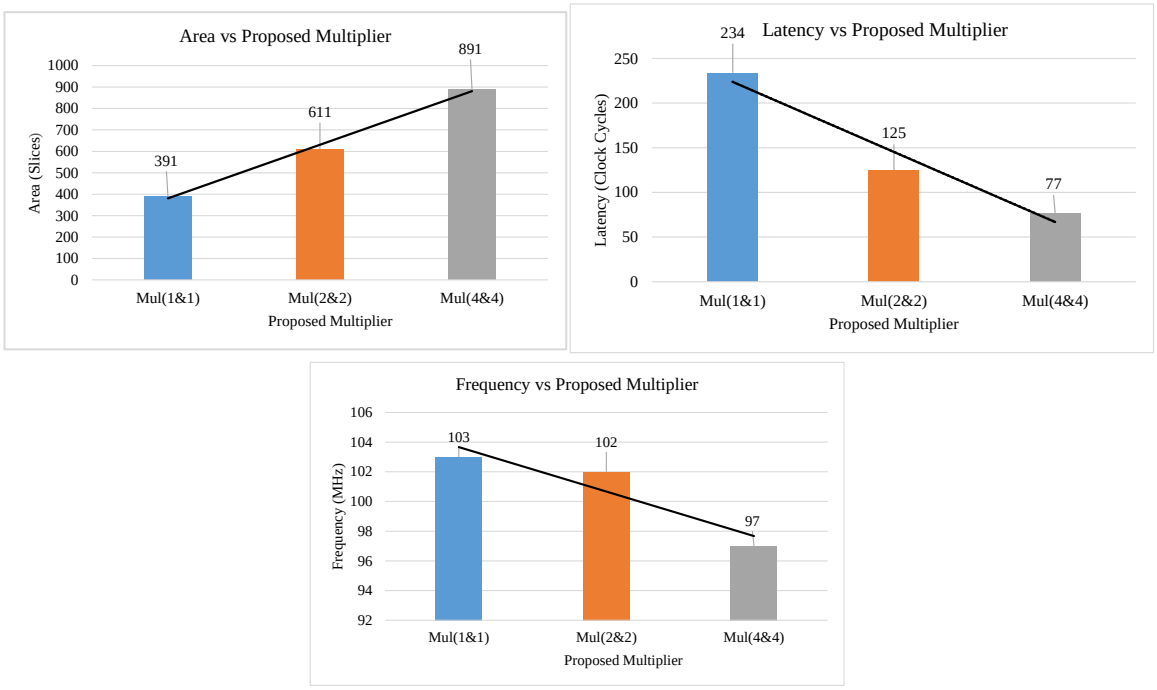


Figure 6.7 Area, latency, max. frequency vs proposed multiplier over $GF(2^{163})$

The latency of the proposed Comba multiplication with reduction is compared with previous reported relevant multipliers is shown in Table 6.4. In this table, the Comba

Table 6.4 Comparison of the proposed multiplier with the relevant multipliars

Ref.	Multiplication latency, t_{mul}				GF(2^{163}), Clock cycles	GF(2^{571}), Clock cycles	Platform:(resources)
	load	Comba Mul	Reduction	others	$s = 21$	$s = 72$	Platform & Mul
[88]	$2s^2$	$3s^2 + 2s$	$5s^2 - 2s$	0	4,410	51,840	S (ISE) (16-bit ALU)
[89]	$20 + 110s^2 + 28s$		45(s+1)	-	50,108	575,561	S: (8-bit ALU, PicoBlaze)
[66]	1	s^2	2 $+ reduction$	3	447+ $reduction$	5187+ $reduction$	H (1x32-bit GF2MUL and full-precision reduction)
[91]	$2(14s^2 + 14s)$		2(131+30(s-3))	-	13,738	149,500	H/S: (3x8-bit GF2MUL)
[63]	$2s^2 + \#mulgf2 + 2s$		$21 + 3s$	-	1,169	12,525	H: (3x8-bit GF2MUL)
ours	**1**	$(s^2 + s)/2$	**0**	**2**	**234**	**2,631**	**H (2x8-bit GFmul2)**

H= Hardware, S=Software, Hardware/Software= H/S, $\#GF2MUL$= estimated

multiprecision multiplier is implemented in the software [88, 89], hardware-software [91] and hardware only platform [63, 66, 93,]. The software implementation in [88] is implemented in

a 16-bit data path based embedded processor. The latency is very high for both Comba multiplication and reduction due to frequent memory access and instruction delay. They consider reduction operation in the left to right method for pentanomial-based curve consumes higher latency than their trinomial-based reduction operation. In [89], a pure software implementation of Comba implementation is implemented in the embedded processor, PicoBlaze. The PicoBlaze implementation consumes very high latency for the both Comba multiplications and right to left reduction operations. To improve the latency problem in the software implementation, the Comba multiprecision multiplication is implemented in the hardware as an integral part of the embedded processor (using an instruction set extension) called hardware-software implementation. In [91], a Comba multiplication is implemented in the hardware and the embedded processor performs the control operation. In the [91], three *GF2MUL* units (3x8 *GF2MUL*) are used to improve their performance (in particular, latency) using small area overhead. Their implementation still consumes large number of clock cycles (13,738 clock cycles over $GF(2^{163})$) to compute multiprecision multiplication, however, they can manage their improvement of latency when it is compared with their software implementation in [89] (50,108 clock cycles). To reduce latency, a standalone Comba multiprecision multiplier is required to implement in the hardware platform (i.e. FPGA). The hardware implementation consumes low latency due to the very low word level memory operation and dedicated control unit (low or no instruction delay). Moreover, the latency can be reduced abruptly by duplication of resources with small area overhead. In [63], a hardware implementation is presented based on three *GF2MUL* multipliers. The hardware implementation consumes lower latency such as 1169 clock cycles for multiprecision multiplication with reduction as compared to their both software [88, 89] and hardware-software [91] implementations. In [66], the Comba multiprecision multiplier is implemented utilizing 32-bit data path that is targeted for the high speed design. The large digit (32 bit) based multiplier is simpler to implement than 8bit data path based. If the architecture in [66] would be designed for 8-bit data path, then their architecture potentialy consumes higher latency than our proposed multiplier.

Our proposed Comba multiplication is implemented in the hardware as a standalone multiplier. The proposed multiplier need minimum two *GF2MUL* multiplier. The standalone multiplier can start multiplication by using one clock cycle for the loading operation. The both multipliers perform multiplication on the two separate upper loops of Comba individually to

Algorithm 6.6 Proposed multiprecision squaring over $GF(2^{163})$

Input: A (x) = ($A_{s-1},\ldots, A_1, A_0$) represented by an array of s single-precision (i.e. w-bit) words
Output: Product $C(x) = A(x)^2 \bmod f(x)$

Step1. (pH, pL) ← 0; (qH, qL) ← 0
For i from s-1 by 1 to (s-1)/2{20 t0 10}
Step2: (qH, qL) ← $(A_i)^2$ [interleave zero]
If i=(s-1)/2
(qH, qL) ← (qH, qL) & (x07,00)
End if
Lv← qL<<5
Mv← qL>>3 xor qL xor qL<<3 xor qL<<4 xor qH<<5
Fv← qH>>3 xor qH xor qH<<3 xor qH<<4 xor qL>>5 xor qL>>4 xor Lv
FFv← qH>>5 xor qH>>4 xor Mv
Step3: (pH, pL) ← $(A_{(i-((s-1)/2))})^2$ [interleave zero]
j= 2i -(s-1)
If j=0 then
C_1 ← C_1 xor Ev1{(0,0,0,q324,0,q322,0,q320) xor (0,0,0,0,0,q324,q322)}
C_0 ← FFv xor pH xor EV0{(0,(q322 xor q320),(q324 xor q322), q320,(q322 xor q320) ,q322 ,(q324 xor q320), q320)}
else
C_j ← FFv xor pH
C_{j-1} ← Fv xor pL
End if
end for
Return C(x) = ($C_{s-1},\ldots, C_1, C_0$)

provide the necessary multipliers outputs to the start reduction. Thus, Comba multiplication and reduction are performing in concurrently. The proposed hardware design reduces the latency lower than the theoretical limit as the multiprecision multiplication performs by free reduction operation. For example, the theoretical latency for the multiprecision multiplications is included a load operation(*s* clock cycles), the *GF2MUL* operations (s^2), a latency for reduction (#clock cycles for reduction) followed by storing (*s* clock cycles). If two *GF2MUL* are used, then the theoretical latency of Comba *GF2MUL* operation is $s^2/2$ excluding other latency (latency of the load, reduction and store operations). Our proposed architecture can save latency for loading, reduction, and storing to achieve a very low latency of $\approx \frac{s^2}{2}$ clock cycles. For example our, multiplier with 8 bit data path consumes $234 \approx 221$ (where $s=21$) clock cycles for Comba multiplication with reduction operation over $GF(2^{163})$. Our proposed parallel multiplier reduces drastically the clock cycles using less resources (2x8-bit *GF2MUL*) as compared to the presented multipliers (3x8-bit *GF2MUL*) in [63].

We save both the latency of the reduction operation and a large area to store large multiplier outputs for reduction [63]; however, we use extra hardware for local memory to save inputs of the multiplier. Moreover, our proposed multiplier performs the reduction operation on the fly

irrespective of the position of second non-zero element in the reduction polynomial. Thus, our proposed method can be used for the multiplication with on the fly reduction operation independent of the structure of reduction polynomial. Again, as our standalone multiplier used local memory for the operands (multiplicand and multiplier), thus, the main memory can be utilized for other operations concurrently with the multiplication.

6.4 Implementing Proposed Multiprecision Square Circuit

Multiprecision square operation is a linear operation over $GF(2^m)$ is achieved by interleaving zero in the input followed by reducing operation in the digit level. The main operation of the squaring over $GF(2^m)$ is the reduction operation. In general, the digit squaring operation of an input with *s* words require *2s* clock cycles (2 clock cycles for each word squaring). After the digit squaring, the *2s* words are saved for the reduction operation. Thus, squaring operation with reduction may share a significant latency [89] of the total latency of ECC point multiplication, in particular, in the software [88, 89] or in the hardware-software platform [91]. The hardware based squaring is free operation (only need load and interleave zero) as compared to software based implementation (need load, *GF2MUL* operations and storage of *2s* words of squaring output for the reduction operation) [63, 66, 88-91]. There are several hardware based square circuits reported in [63, 66, 91]. However, their works show low latency of squaring operation due to hardware implementation. They did not consider further major optimization of square circuit to reduce latency. The latency of the reduction is required to be low in the case of low latency multiprecision multiplication. The latency of the squaring with reduction operation can be improved in the hardware implementation by utilizing the flexibility of FSM based addressing is discussed in this section.

6.4.1 Novel Architecture of Multiprecision Square Circuit with On-the-Fly Reduction

The digit (word) square operation produces two digit size outputs for each digit operation. After squaring of *s* number of words, we get *2s-1* size outputs. As the square of each word (*w*) generates two-word size result, the digit squaring performs in the every two-clock cycles. For example, the squaring of the 0th word of *s* input words produces two words output such as 0th and 1st words of the *2s-1* result. the next input (1st digit) is delayed by one cycle to square that produce 2nd and 3rd words of output. Thus, *s* number words take *2s* clock cycles for squaring operation.

We propose a novel multiprecision square algorithm is shown in the algorithm 6.6 that can square *s* number words by using only *s* clock cycles along with on the fly reduction operation.

In this algorithm, we square *s-1* words by addressing in a sequence of *s-1, (s-1)-((s-1)/2), (s-2), (s-2)-((s-1)/2)… 1, (s-1)/2, 0* to square *s* words by using *s* clock cycles as shown in the step2 and step3 of the algorithm. We merge two sequences such *as s-1,-, s-2,-,…,-, 0* and *(s-1)-((s-1)/2), -, (s-2)-((s-1)/2),-,…, -,(s-1)/2* so that the proposed alternate sequence still allows successive inputs of a particular sequence to do square in every two clock cycles. The alternate sequences offers several advantages such as :

- The square operation of *s* words can be achieved using only *s* clock cycles.
- The reduction operation can be started after one-clock cycle of staring.
- The *2s-1* outputs does not require saving for the reduction operation.
- Finally, no delay for loading outputs.

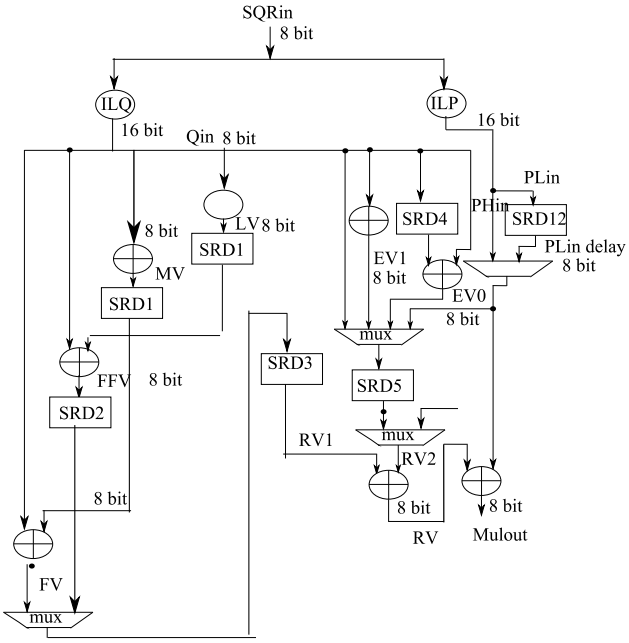


Figure 6.8 Multiprecision square circuit over $GF(2^{163})$

In the algorithm 6.6, a multiprecision square operation with reduction operation over $GF(2^{163})$ is presented. A hardware implementation of the algorithm is shown in the Figure 6.8. In the figure, the square operation of each word (8 bit) produces a 16-bit output. The sequence of input words is *20, 10, 19, 8,…10,0.* The results of square of the words, 20 to 10 are used to generate reduction vectors (*Rv*) as shown in the figure. The *Rv* is generated using fast reduction method [1, 88] as shown in the algorithm to add with the square outputs of words of 10 to 0. For example, if we square 10^{th} word of input, then we get two-word size output. The two-word size output is divided into two words such as $10^{th}H$ *and* $10^{th}L$ words. The *H* part contains 15 - 8 bit and the *L* part consists 7-0 bit. Similarly, if we square the 20^{th} word of input, then we get

$20^{th}H$ and $20^{th}L$. The $20^{th}H$ and $20^{th}L$ words are the required inputs to generate the reduction first reduction vector that is *FV* in the Figure 6.8. After then, the first reduction vector is added with $10^{th}L$ (right-side of the Figure 6.8) to get the most significant square output (20^{th}) word of the reduction. Thus, the reduction operation is started after one clock cycle of the 1st input (*s-1th* word*).* In left to right fast reduction algorithm, the extra vectors *Ev1* and *Ev0* (as shown in algorithm 6.6) depends on the most significant output words. Thus, the *Ev1 and Ev0* vectors are ready to add with the least significant word (1st and 0th words for the case of $GF(2^{163})$ as shown in the Figure 6.8) as shown in the algorithm 6.6.

Trinomial irreducible polynomial based reduction operation has more complexity than the pentanomial irreducible based reduction operation because of the second highest order term of the irreducible polynomial. For example, in the case of trinomial based reduction polynomial, the position of the second highest order (non-zero term) is at 74 and 87 in the case of $GF(2^{233})$ and $GF(2^{409})$ respectively. Thus, there is a dependency in generating reduction vector due to the position of the second non-zero terms. For example, to generate 1st reduction vector for the left to right fast reduction operation (similar to the algorithm 6.6) the sequence of *2s-1. 2s-2, …, s-1* outputs require to align with a sequence of outputs of *(2s-1-k/w), ((2s-1)-(k/w)-1)),…, (s-1).* (Where *k* is the second higher order). Thus, the reduction operation can be started upon the availability of output sequence of *(2s-1-k/w), ((2s-1)-(k/w)-1)),…,(s-1).* The dependency can induce extra latency for the reduction operation.

We can utilise our proposed algorithm 6.6 of the pentavalent curves, to accomplish a square operation in the case of trinomial-based reduction operation by considering extra register. A hardware architecture of trinomial irreducible polynomial based reduction operation is shown in Figure 6.9 and the reduction operation is illustrated in Table 6.5 over $GF(2^{233})$.The left side of Figure 6.9, continuous vector, *QV* is generated whereas right side of the figure is generating the last two vector(*L2V*). To start reduction operation, we need the three sequences of square results such as 1*) 2s-1, 2s-2,…, s-1,;2) (2s-1)-(k/w), (2s-2)-(k/w),…, s-1;* and 3) *s-1, s-2,...,0* as shown in the Table 6.5. We can get two sequences (2nd and 3rd sequences) as square outputs by accessing data from main memory in the alternate addressing as shown in the algorithm 6.6. We utilise a second register to supply the necessary operand to generate the sequence of *2s-1, 2s-2,…, s-1,;2) (2s-1)-(k/w), (2s-2)-(k/w),…, s-1*. Thus, we need to store the *k/w* numbers of most significant operands in a temporary register (*Temp* in Figure 6.9) before the start of the squaring operation. The temporary register provide the operands to generate part

of the 1st sequence *(2s-1, 2s-2 .. (2s-1)-(k/w))* and we also reuse the operands to generate reduction vectors during the reduction operation.

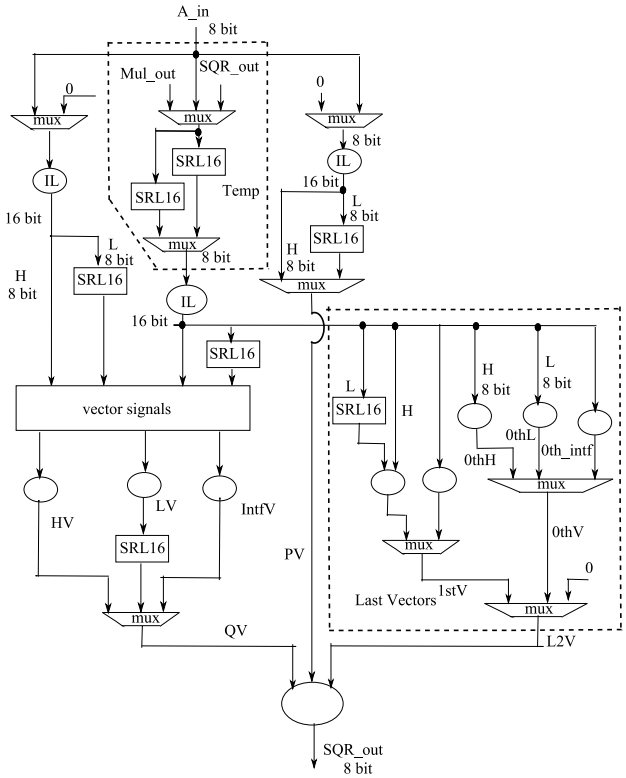


Figure 6.9 Multiprecision square circuit over $GF(2^{233})$

In the Table 6.5, we illustrate left to right reduction operation. We access the digits from main memory is as the sequence of *20, 14, 19,13,…,14,0* and from the *Temp* register is as the sequence of *29,29,28,28,…20,20*. We get two words size output for each digit squaring

Table 6.5 Multiprecision reduction operation over $GF(2^{233})$ on the 465 bits of square output (w = 8 bit)

29			28			…	18		…	9			…	0	
239,…,232			231,…,224			…	151,…144		…	79,…72			…	7,…0	
7	6,…,1	0	7	6,…,1	0	…	7,…,0	1,0	…	7	6,…,2	1,0	…	7	6,…,0
14th H			14th L			…	9th L		…	4th H			…	0th L	
239,…,232			231,…,224			…	151,…, 144		…	79	78,…,74	73,72	…	7	6,…,0
24th H		24th L							…	14th H		-	…	-	
398,…, 392		391	390,…,384		383	…	….		…	238-233		-	…	-	
									…				…		
	29th L		28th H			…	…	…	…	…	….		…	14th H	
472	471,…,465		464	463,…, 457		…	…	…	…	…	…		…	240	239,…,233
											29th L		…	24th H	
									…		-	465,464	…	399,…,392	
							29th L	…	…	24th H					
							…,465,464	…	…	397,…,292		-			
VH(s-1)			VL(s-2)				1stV((k/w)-1)			0thVi((k/w)-1)				VL(0)	

such *H* part (15-0) and *L* part (7-0) as shown in the table. The reduction vectors are generated (*Rv*) by aligning 1st and 2nd sequence to add with the 3rd sequence in reduction operation. The vector is generated by aligning the square results as shown in the table. For example, the most significant vectors uses $29^{th}L$ and 24^{th} to generates vectors (*Qv* in the Figure 6.9) to add with *Pv* inputs $(14^{th}H)$ to get the 1st output of reduction operation. We can the 1st reduction output one clock cycle of the starting. Two extra vectors 1st and 0th as shown in the Figure 6.9 (similar to the *Ev1* and *Ev0* as shown on the algorithm 6.6) are generated using the sequence of input as $29^{th}L$ *to* $24^{th}H$ as shown in the Table 6.5. The first extra vectors (as like as *Ev1*) are added from *(2k-1)th* bit to k^{th} bit and second extra vectors, 0th (as like as *Ev0*) are added from *k-1th* bit to *0* bit as shown in the Table 6.5. For example, the 1st extra vectors are added from the word for 18th to 9th word and second extra vectors (0th) are added from 9th to 0 words over $GF(2^{233})$. Especially, at 9^{th} word, part of the both extra vectors are used exclusively. Thus, the unnecessary part of the vectors are padded with zeros at the 9^{th} word. The trinomial-based reduction operation is performed in the same latency($s+2$ clock cycles) of the pentanomial based reduction. The reduction operation based on trinomial polynomial utilise extra register to show the same performance as like as the pentanomial based reduction operation.

6.4.2 Repeated Squaring

The repeated squaring is required in the case of multiplicative inversion operation. In the case of repeated squaring, extra latency may require to start a new square while the

Table 6.6 Comparison of the proposed square circuit with the relevant square circuit

Ref.	Squaring latency, t_sqr				GF(2^{163}), Clock cycles)	GF(2^{571}), Clock cycles)	Platform:(resources)
	load	Digit wise squaring	Reduction	others	$s = 21$	$s = 72$	
[88]	$2s$ +2s		2s+2s+s+2s		231	972	S: 16bit RISC Processor
[89]	$(20 + 110)s$		45(s+1)	-	3,720	11,225	S: (8-bit ALU, PicoBlaze)
[66]	1	$2s$	2 $+ reduction$	3	48 $+ reduction$	150 $+ reduction$	H: (32-bit data path and full-precision square)
[91]	$2(24 + 21s)$		2(131+30(s-3))	-	2,272	7,474	H/S: (8-bit data path and Multiprecision square)
[63]	$2s + \left(\frac{s}{3}\right) + 2s$		3s+21	-	175	549	H: (8-bit data path and Multiprecision square)
ours	**1**	$\boldsymbol{s + 1}$	**0**	**0**	**23**	**74**	**H: (8-bit data path and Multiprecision square)**

H= Hardware, S=Software, Hardware/Software= H/S

operation is depending on the previous output results. In particular, the trinomial based reduction operation has a dependency to repeat the square operation on the previous squaring result. In our proposed square operation, the starting square operation depends on the most significant words. Thus, the pentanomial based square operation can be started immediately as there is no requirement of storing operation as like as a trinomial. In the case of trinomial, as the most significant words of square output are outputted first, the required most significant outputs are saved in the alternate temporary register to use in the next square operation. In the Figure 6.9, we use two duplicate *Temp* registers (2 *SRL16*) in the Temp block (dashed line) to store most significant words alternatively. One of the *Temp* registers contains current square operands, and the other *Temp* register is used to store the most significant results of the current square to use in the next squatting operation. The alternate use of the temporary register helps to avoid data dependency to start new squaring. Thus, our proposed architecture allows the repeated square operation immediately irrespective of the structure of irreducible polynomials.

6.4.3 Comparison with relevant square circuit and discussion

In Table 6.6, we compare the performance of our proposed multiprecision square circuit with the relevant square circuits. Most of the multiprecision square circuits in the literature consume high latency for square operation; however, the square operation is a linear time operation. In general, the software based squaring [88, 89] consumes higher latency than hardware implementation [63, 66, 91] due to instruction delay. The software implementation in [88] consumes 231 clock cycles over $GF(2^{163})$ for the case of 8 bit data path, whereas the work in [89] consumes high latency (3720 clock cycles) due to the different software platform. The latency of [89] is improved in their hardware–software design in [91]. The hardware implementation of square circuit in [63] shows very lower latency (175 clock cycles) than their implementation of the software, [89] and software-hardware, [91]. The hardware implementation in [66] presents multiprecision square operation followed by reducing operation. Their design consumes 45 clock cycles only for digit squaring without reduction. Our proposed design reduces latency more than half of the previous multiprecision square circuit. The latency performance over $GF2^{571}$ shows similar performance as like as $GF(2^{163})$. Our proposed square consumes 1 clock cycles for load operation, and *s* clock cycles for digit squaring operation. The reduction operation of the proposed design is performed concurrently with the digit squaring irrespective of the structure polynomial (pentanomial and trinomial). Moreover, the repeated squaring is achieved immediately after previous square operation

without consuming the delay for loading. The proposed square circuit offers some advantages such as:

- The reduction operation performs concurrently along with the squaring operation. Thus, the square results (*2s-1* words) are not required to store for the reduction operation.
- The latency of the proposed square circuit is considerably reduced to the theoretical limit of latency for the multiprecision squaring with reduction.
- The square operation consumes same latency for the both pentavalent and trivalent irreducible polynomials based square operation.
- The square circuit consumes very low latency; hence, the squaring operation can operate concurrently with the field multiplication during ECC point multiplication.
- The repeated squaring (without delay of loading) can accelerate the inversion operation.
- ECC with a low latency multiplication (using the parallel architecture of multiprecision multipliers) can improve efficiency using the proposed very low latency squaring operation.

6.5 Proposed Hardware Architecture of Scalable ECC

Low resource scalable ECC is a promising solution for the low-end applications (i.e. wireless sensor nodes, smart cards, radio identification (RFID) tags and mobile devices) to provide high security using the same crypto processor. The large operand size of the ECC is a crucial challenge to adopt in the constraint environment due to their limited storage and low computation power. The challenge is also becoming worse while it is requiring scalability of security.

Low resource ECC implementation can be software only (embedded processor), software-hardware or hardware only. Each of them had some advantages and disadvantages. Software only implementation uses an embedded processor to implement all of the ECC operations using word level instructions. The hardware-software implementation is a combination of embedded processor and hardware coprocessor (using instruction set extension-based processor, RISC processor). The main processor (embedded processor) controls some of the arithmetic part (i.e. field multiplication) as a peripheral. The hardware-software based ECC processor is faster than software only implementation. The separate hardware module is working as a dedicated module to improve performance.

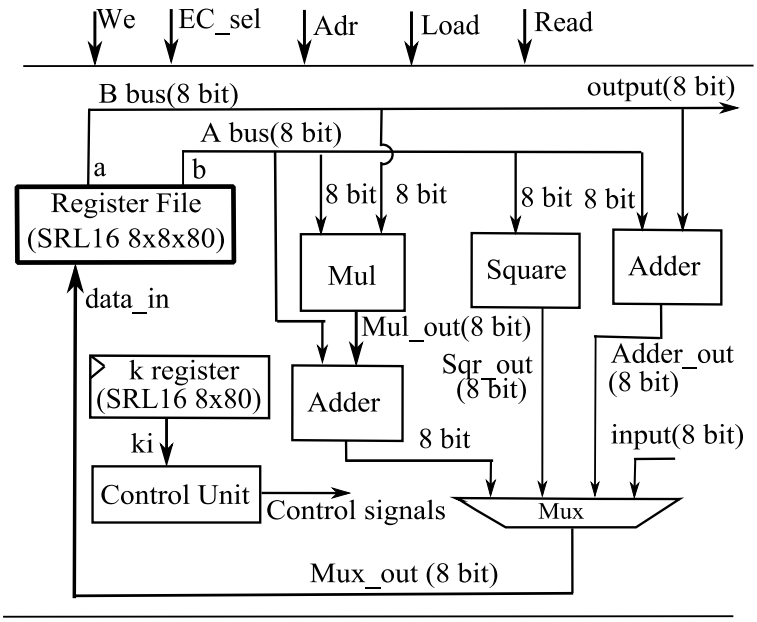


Figure 6.10 Proposed scalable ECC architecture

Hardware solution of scalable cryptography is an attractive topic in the academic research and industrial application due to both high performance and low power operations. In particular, at present, for a sensitive data transmission, the cryptography requires high security day by day. ECC processor with the scalability of security can meet the changing security requirement using the same module. The high security can be increased by increasing field size. Thus, the same crypto processor with higher security with flexibility (Scalability) is very important. For an increase of field size, in particular, the computation requirement increases geometrically. The high workload for point multiplication of ECC can deteriorate the performance of cryptography operation. In particular, the software and hardware-software based implementations consume higher latency per operation than that of the hardware-based processor. Thus, the hardware solution can meet the requirement for a real time solution of security with scalability. Thus, hardware solution is increasing popular for ECC implementation even in the low resources applications.

Several scalable hardware, in particular, FPGA implementations for point multiplication are reported in [9, 63, 66, 89-93]. Most of them considered low security curves or selected some specific curves instead of all NIST curves [9, 63, 89-92]. In [90], they presented hardware-software ECC point multiplication implementation on all NIST curves. In their implementation, they used hardware circuit for multiprecision arithmetic (Mul, Sqr and Add) and reduction by using controlling signal from Picoblaze microcontroller. Their hardware/software design work consumes high latency for point multiplication due to high

latency arithmetic operations; however, they improved the latency of point multiplication as compared to their software-based implementation in [89]. A pure hardware based scalable ECC is presented in [9] by considering Elliptic curves such as 113, 131, 163, 193. They uses lower field size curves to evaluate their performance. Their hardware-based implementation improves their latency as compared to the respective software implementation in [9]. Still, their hardware scalable design consumes large delays for point multiplications. All NIST curves are considered in [66, 93] are targeted for server end applications. The reported work show low latency using the advantages of large digit (32 bit) size data path and large digit-size reduction operation.

We propose a scalable ECC for point multiplication over all of the NIST curves such $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$. The scalable ECC is implemented on FPGA hardware using 8 bit data path architecture is shown in the Figure 6.10. We consider combined Montgomery point multiplication algorithm in projective coordinate to perform point multiplication. The point multiplication result of the projective coordinates is then converted into affine coordinates. In this scalable architecture, we utilise a novel multiprecision multiplier, a novel multiprecision square circuit and an adder circuit for the field arithmetic unit as shown in Figure 6.10. We utilize two level control circuits included the top level control unit to accomplish point operations (group operations) and low level control units for field arithmetic operations. We utilize a low resource register file to saves the parameter of the elliptic curve (coordinates of the base points, constant, b) (page 2-28). We use separate register for key is used to control top control unit to accomplish point multiplications. Our proposed scalable ECC over all NIST curves shows very latency for the point multiplication using low arithmetic resources to enable an efficient scalable ECC for the low-end applications.

6.5.1 Proposed Montgomery Point Multiplication

The Montgomery point multiplication algorithm is widely considered for the ECC implementation. The point multiplication algorithm has some advantages such as low memory requirement due to x and z coordinates (in the projective coordinates) are used for the point multiplications. Moreover, the algorithm can compute faster point operation than the basic point multiplication. Again, the algorithm has partial side channel resistance (power attack) due to the point addition and point doubling of point multiplication is performed in every bit of key input.

We propose a combined point addition and point doubling based Montgomery point multiplication algorithm for the scalable ECC implementation is shown in the algorithm 6.7. In our proposed algorithm, we compute the square and addition operations concurrent with the multiprecision multiplication as the multiprecision multiplication is a higher latency operation than square or adder operation. Moreover, In the algorithm, we need to add last two multiplication results to get last output X_1/X_2 at the end of the each of point multiplication loops. Our proposed architecture support to add the result of multiplication in *st5* with the result of multiplication in the *st6* concurrently. Thus, the architecture can manage to save the latency of an addition. Thus, our proposed ECC depends on the latency of the multiplications. Again, the memory operations is a word level load and unload operations. The load and unload in the same location is complex. The complexity may increase the requirement of memory. Our architecture utilizes the memory by using free location. We, thus, perform some transfer operations (in the *st2* and *st5* as shown in the algorithm) to use the register effectively when the multiplication is performed.

The loop of point multiplication is performed in the projective coordinates and the result of point multiplication in the projective coordinates is then converted to affine coordinates. The conversion operation includes a costly multiplicative inversion operation. The inversion operation is utilised by repeated squaring and multiplication operations using the Itoh-Tsujii algorithm [65].

6.5.2 Careful Scheduling for Point Multiplication

We present a careful scheduling for the point multiplication is shown in the Figure 6.11. Our proposed algorithm is performed in the six steps to complete a loop operation of the point multiplication. Each step is involved with one of the six field multiplications of Montgomery algorithm. The square, addition and transfer between register are performed while the multiplication is performed. As multiplication is the high latency operation, thus, the latency of each step is the latency of multiprecision multiplication. The latency of a loop of the point multiplication in projective coordinates is as same as the latency of six multiplications as shown in the Figure.6.11 (for the case of $k_i = 1$).

- In *st1*, X_2 and Z_1 are loaded from the register file to local register of the multiplier. The *GF2MUL* operation is started after one clock cycle of the loading operation. Thus, the loading and multiplication are performed simultaneously. After the loading operation, the memory is free for other operations such as square, addition, and transfer operations

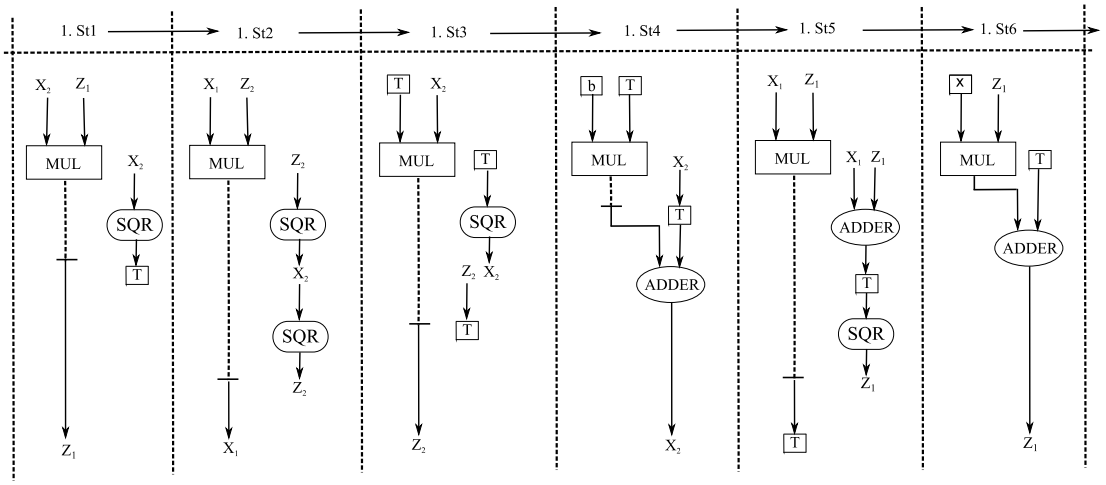


Figure 6.11 Data flow graph of the proposed combined Montgomery point multiplication

as long as new results of the multiplier are ready to load in the main memory. The X^2 operation is started after loading operation is completed (dash line). The *X* is accessed from the main memory to square and the results of the square are saved in the address of *T* (a location of the memory). The gap between the end of load operation and the start of reduction of multiplier is used for accessing main memory operation to square. After finishing the square operation, the reduction operation is performed. Thus, the start of the reduction operation depends on the parallel operation. After the loading operation, there is a common delay of *2s-4* clock cycles to start the reduction. In this case, the starting of reduction is delayed if more than one parallel operation is performed. In the case of trinomial curves, several parallel operations can be performed before the first output of the multiplier as there is a big gap to start reduction (mainly due to the order of second non-zero term in irreducible polynomial).

- The *st2* is started immediately the multiplication of Step1 is finished. In the Step2, X_1 and Z_2 are accessed from main memory to load in the multiplier. After loading, the Z_2 content is free to square by accessing from main memory followed by saving at the address of X_2. Now, we square again the content of X_2 to get finally Z_2^4. The new square result of X_2 is saved at the address Z_2 of the register file.
- The *st3* performs the multiplication, $Z_2 \leftarrow T.X_2$. After loading of multiplier operands, the *T* content is accessed from memory. Thus, the content of *T* is squared and saved at X_2. Now, the *T* address is free. The content of Z_2 is required in the next step. We

also need the Z_2 to save current multiplication result. Thus, the content of Z_2 is transferred to *T* location. Now, the Z_2 location is free to save multiplication output.

- In *st4*, *b* and *T* are multiplied and the result of the multiplication is added with the content of X_2 followed by saving in the X_2. To save the addition result at X_2, the content of X_2 is required to transfer in the other empty location (*T*) before the start of reduction operation. The addition operation performs on the fly using the result of the ongoing multiplication and the content of *T* (the transferred content of X_2) as shown in the solid line. The addition result is saved at X_2.
- The multiplication X_1 and Z_1 is performed in the *st5*. The same contents (X_1 and Z_1) are required to add and then, squared. After loading X_1 and Z_1 in the multiplier, the same contents are accessed from the memory to add in the word level. The result of addition is saved immediately in the temporary location (*T*) to use again for squaring. The results of the squaring operation of the *T* is saved in the Z_1. After square operation, the location of *T* is free to save the output of multiplier reduction.
- The final step, the coordinate *x* is multiplied by Z_1 and the result of the multiplication is added with the result of the previous multiplier result (*T*) on the fly. The content of *T* is accessed word by word to add with a sequence of respective reduction output. The result of the addition is saved as Z_1 in the register file.

After finishing the step, new loop is started from the Step1 of the proposed algorithm. The multiplication operands depend on the value k_i, the *i*th value of *k*, key. The change of address due to change of k_i value is performed by observing three consequent values such as previous value, k_{i+1}, current value, k_i and future value, k_{i-1}. The new loop operation started immediately after the *st6*. Thus, the latency of each loop is the latency of six multiprecision multiplications. Moreover, the careful scheduling utilised unused locations to save the intermediate values. In particular, the scheduling can manage all necessary square and addition operation on the fly by smartly using the transfer operation during multiplication. In the each loop operation, the key counter, *K_counter* is updated to transfer control to complete conversion (affine conversion) operation.

The projective to affine conversion is a one-way operation. To complete conversion operation, the latency of the conversion is due to mainly 10 field multiplications and one inversion operation. The inversion operation is accomplished with a multiplicative inversion. The Itoh–Tsujii method, a modified Fermat's little theorem (FLT) is widely considered

involved with repeated multiplication and square operations. In this method, total number of multiplications are calculated by $\lfloor \log_2 m - 1 \rfloor + h(m-1) - 1$, where $h(m-1)$ is the Hamming weight and $\lfloor \ \rfloor$ is floor function. The total number of squaring operations is $(m-1)$ [9, 16]. For $GF(2^{163})$, the number multiplications are $\lfloor \log_2 163 - 1 \rfloor + h(163-1) - 1 = 7 + 3 - 1 = 9$ and the total number of square operations are (163-1)= 162. Thus, the latency of square operation to complete inversion over $GF(2^{163})$ is equivalent to 16 multiplications where the latency for a square is 23 clock cycles and a multiplication is 234 clock cycles. Again, the latency square operation of inversion over $GF(2^{571})$ is also equivalent to 16 multiplications where the latency of the square is 74 clock cycles and the multiplication is 2631 clock cycles. Total equivalent multiplication work load in the conversion over $GF(2^{163})$ = (10 + 9 +16 =35) =35 multipliers. For the case $GF(2^{571})$, the total number of equivalent multiplications is (10 + 13+16) = 39. Thus, the conversion operation has high impact on the low order curves than higher order curves. For example, the ratio of latency of (conversion operation/loop operations) over $GF(2^{163})$ is 0.036 (3.60%), and the ratio over $GF(2^{571})$ is 0.0114 (1.14%).

6.5.3 Proposed Scalable Multiprecision Multiplier Circuit

The proposed scalable ECC utilizes novel multiprecision multiplier for scalable multiplication. The multiplication has two parts, including *GF2MUL* operation followed by reducing operation. The proposed modified Comba multiplication is utilised for *GF2MUL* operation and fast reduction method is used for the reduction operation.

The proposed multiplication algorithm utilised two upper loop operations using two *GF2MUL* multipliers. The multiplier architecture in Figure 6.2 illustrates the multiplier architecture. The *GF2MUL* operation is performed over all of the NIST curves. The reduction operation of the multiplier results of *2s-1* words is reduced concurrently with the *GF2MUL* operation. However, the operation of reduction over pentanomial and trinomial has different complexities, our scalable reduction can manage all reduction operations without extra latency. Apart from pentanomial, the trinomial reduction operation requires special care to clear unnecessary bits during the reduction vectors generation.

6.5.4 Proposed Scalable Multiprecision Square circuit

Multiplier can perform a square operation by taking same operand as multiplicand and multiplier. As multiplication has higher latency than a square operation, thus a separate square operation is commonly considered in the ECC crypto processor. Moreover, square operation

over $GF(2^m)$ is achieved by simply interleaving zeros in the inputs. Thus, the only main operation of the square circuit is the reduction operation.

In the ECC, square operation can be achieved concurrently with the multiplication as the latency of multiprecision multiplier has many times higher than the latency of multiprecision square circuit. Thus, the latency of the square is not significant in the most of the scalable ECC design when it is compared with the latency of multiprecision multiplier. If parallel multipliers are considered to improve the latency of multiprecision multiplier, then the latency of the square circuit will be comparable to the latency of the multiplier. In this case, an efficient and low latency square circuit is required.

Small data path (8 bit data path) require a large number of operations to reduce *2s-1* words of square result. The complexity increases with increase of the field size. In particular, the reduction operation of trinomial-based curves has more complexity than the pentanomial based curves.

Thus, small data path based square circuit has more complexity in the controlling of data than the large word size based scalable square circuit. Even, the complexity increase high when all of NIST standard curves are considered in the same squaring module. The complexity is increased mainly due to pentavalent (for $GF(2^{163})$, $GF(2^{283})$, *and* $GF(2^{571})$ and trivalent *(*for $GF(2^{233})$, and $GF(2^{409})$) irreducible polynomials are involved. The second higher order of trivalent (87 for $GF(2^{409})$) is multiples of the second highest order (7 for $GF(2^{163})$) of the pentavalent irreducible polynomial. The differences in the second highest order values of the irreducible polynomials create complexity in the area, latency and controlling of the multiprecision square circuit.

There are several square circuit are presented in the literature [9, 63, 66, 88-93], they use common reduction circuit for the multiplier and square circuit. Their square operation consumes large latency. In [66, 93], a large digit (32 bit data path) based square operation followed by a reducing operation of the common reduction unit.

We propose an 8 bit data path scalable square circuit utlising low latency novel digit square circuit over $GF(2^m)$ as shown in the Figure 6.8 over $GF(2^{163})$, and in Figure 6.9 over $GF(2^{233})$. Our proposed scalable square circuit can support square operation over all NIST binary curves such as $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$. In the proposed scalable multiprecision square circuit, we adopt trivalent and pentavalent irreducible based

ECC curves consuming same latency, $2s+2$ clock cycles to complete scalable square with reduction.

The difficulty in the design of scalable ECC, including trivalent irreducible based ECC curves is due large value of the second highest order (k) of the irreducible polynomial. To start left to right reduction, first output is ready on the available (2s- *(kth* order/w) -1)th word. This latency can be overcome by saving the most significant words in the temporary registers before the square operation. After then, the temporary register and main memory provide the necessary square outputs (interleave zero) to get reduced output after 2 clock cycles of the starting.

The latency of squaring is also significant for the point multiplication as our proposed ECC architecture utlises very low latency multiprecision multiplier. Our low latency square circuit can further reduce total point multiplication by providing low latency repeated squaring. In the repeated squaring, after one square finish, the next square can start immediately (same for the both trivalent and pentavalent cases).

6.5.5 SRL16 based Register file

Multiprecision arithmetic circuit based ECC need a large area to save the ECC parameter and intermediate results. The Memory unit or register file of ECC is the largest part (about 50-70% of the total area) of a low area ECC architecture. The memory unit can be designed by using different style such distributed logic (LUTs) or block memory (Block RAM (BRAM)) [95]. The look up tables (LUTs) based memory unit is faster with an overhead of extra logic cells. The BRAM is a storage fabric of the FPGA technology is suitable to store data without taking extra logic cell. Again, SRL16 is FPGA fabric that is a 16x1 shift register depends on the logic cell available in the most of the FPGAs to develop a register file. The SRL16 consumes a very low area per bit as compared to direct logic based memory unit [95]. However; SRL16 consumes a very low area; there is hardly found the SRL16 based register file in the literature for an ECC architecture due to its data shifting property. In the literature, the memory unit is designed by using dual port BRAM as an advantage of the FPGA. The area of the BRAM does not reflect in the area consumptions (slices).

We design a novel architecture *8xm* size of register file based on SRL16 is shown in Figure 6.12. The register file has 8 locations and each of them can able to store the largest NIST curve, m= 571. The building block of the register file is a shift register, SRL16 (1 bit x16 locations). We create a module 8x16 bit storage using the 8 of 1x16 SRL16. We use 5 of 8x16

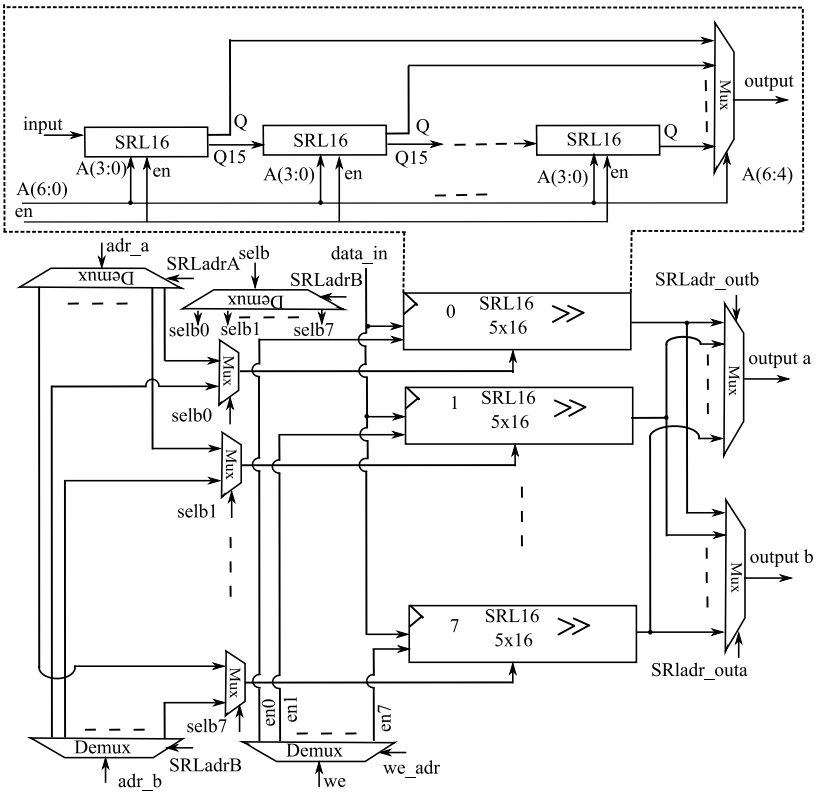


Figure 6.12 SRL16 based 8xm memory unit

storage modules to connect in a cascaded form to allocate 571 bit in a matrix of 8x80 bits. To store a 571 bits data, the 8x80 module takes 8 bit data in each clock and shifts the data to allow next input as long as the last 8 bit ($\lceil 571/8 \rceil th$) data is inputted. To read 571 data, an address of SRL16 is used to read 8 bit data from the 8x80 modules. The address is divided into two parts. The least significant 4-bit is the address of 16 locations of a particular SRL16 and the most significant 3-bit is used to select one of the 5 of 8x16 modules to read 8 bit data in each clock as shown in the Figure 6.12.

The eight numbers of 8x80 modules are grouped to form an 8xm size register file. The register file consists one input data bus and two output data buses using two addresses. The address *SRLadrA* defines the any of the 8 locations of the register file to access data as an output, *a_out*, and similarly, the address *SRLadrB* defines any of the 8 locations to access data as an output, *b_out*. The *adr_a* and *adr_b* are used to address any 8-bit data (word) out of 80 locations to read as *a_out* and *b_out* respectively.

The SRL16 based register file can access two different data from two separate locations and save the input in another location concurrently. The limitation of the register file is that

data cannot be read from and save in the same register location concurrently due shifting nature of the SRL16.

Our proposed SRL16 based register file consumes a very low area to save 8x571 bit data. In the synthesis result on Sparatan3 FPGA using ISE 14.5 tool, the register file consumes only 472 slices (816 LUTs) with a maximum frequency of 316 MHz (3.16 ns).

6.5.6 Interface to outer world

The proposed scalable ECC can compute point multiplication on all NIST curves by selecting curve by using *EC-sel*. A finite state machine is used for interfacing input and output with the outer world. To save an input in a location of a particular ECC, the *load* signal =1 and the *adr* is used to select a location in the memory unit to save input. The ECC parameters are saved in the register file and the key is saved in the key register. After load operation, *Load = 0*, then , the control unit starts point multiplication. After the end of the point multiplication, done =1. To get output, the *rd=1* and the *adr* are used to take the output from the particular location.

6.6 Implementation Results

The propose scalable ECC over $GF(2^m)$ is modelled in VHDL coding and synthesised, mapped and implemented using Xilinx ISE 14.5 tool. We implement the proposed scalable ECC on the low cost FPGAs, such as Spartan3 and Spartan6. In Table 6.7, the utilization of area (Slices, LUTs, FFs, Block RAM), maximum frequency, latency, and area-time metric of the proposed scalable ECC for the point multiplication are presented after place and route. In the table, the results of the most relevant scalable ECC over $GF(2^m)$ are also presented for a fair comparison.

Our proposed Scalable ECC utilises novel low latency multiprecision multiplier, novel low latency multiprecision square circuit to get a low latency point multiplication. In the scalable ECC, the low latency based arithmetic circuits consumes a very low area to perform point multiplication over all NIST binary curves such as $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$. We utilize a novel memory unit (register file) using low cost SRL16 instead of block RAM to evaluate clear picture of the performance of proposed scalable ECC. Moreover, the proposed scalable ECC is based on 8-bit data path architecture that is targeted in the low end applications. Our proposed architecture consumes the area of 2377 slices with maximum frequency of 41 MHz on low cost Spartan3 and 1260 slices with a maximum

frequency of 63.74 MHz on Spartan6. Our proposed scalable ECC consumes very low latency for point multiplication is 236k (k for kilo) clock cycles over $GF(2^{163})$ and 9090k clock cycles over $GF(2^{571})$. Our low latency based ECC shows the area-time metric (slices x kP time x 10^{-3}) of 13.74 over $GF(2^{163})$ *and* 554 over $GF(2^{571})$ on Spartan3 and the are-time metric of 4.66 over $GF(2^{163})$ *and* 179.71 over $GF(2^{571})$ on Spartan6. Thus, our proposed scalable ECC shows better metric in the advance technology .

To compare with state of the art, we implement our proposed design on Sparatn3 for fair a comparison with state of the art. In general, the comparison of our proposed scalable ECC with reported work in the Table 6.7 is difficult due different design considerations and goals. Most of the reported scalable circuits use FPGA fabric (i.e Block RAM) to optimised their area. The area presented by their results does reflect the actual area of their ECC. Moreover, Memory is the largest part of ECC (50-70% of the total area) , thus, the area (slices) information and their corresponding metric does reflect actual performance of the reported works. We reported first time the actual figure of the area of the proposed scalable ECC to evaluate its application in the low-end application. Our proposed ECC consumes a very low area and very low latency to show better performance (area-time metric) than the most relevant scalable hardware ECC.

In [91], a scalable hardware-software ECC implementation is presented as 8-bit architecture on FPGA(Spartan3) over only the pentavalent irreducible based elliptic curves such as $GF(2^{131})$, $GF(2^{163})$, $GF(2^{283})$, *and* $GF(2^{571})$ to avoid complexity. Their arithmetic part is implemented in the hardware and the control part is implemented in the software (microcontroller). They consider *tNAF* method for point multiplication to reduce the latency of point multiplication abruptly. They mapped key in the offline to consider very simple field squaring of the coordinates in place of point doubling operation. Thus, their implementation saved resources and latency for mapping circuit to improve the performance of the point multiplication. Their implementation utilises three 8-bit *GF2MUL* resources to accelerate field multiplication(i.e. *GF2MUL* operation). The multiplier resources are also used for squaring operation to reduce latency of the squaring operation. The ECC in [91] consumes area of 392 slices and 4 BRAMs and maximum frequency 91 MHz. The point multiplication time over $GF(2^{163})$ and $GF(2^{571})$ are 87 ms and 2,740 ms respectively. Our proposed implementation shows 15 times in the case of $GF(2^{163})$ and 12 times $GF(2^{571})$ faster time for point multiplication than the ECC in [91]. Our proposed ECC consumes low resources (two 8-bit

Table 6.7 Comparison of the proposed scalable ECC with the state of the art on FPGA after place and route

Ref	Data Path (bit)	Field size, m	Slices (Sls)	LUTs	FFs	Block RAM	Max. Freq, MHz	Total Clk Cycles	Total *kP* Time, ms	Area-time, Sls x Time x 10^{-3}	Remark: FPGA and Arithmetic Resources
[9]	8	113	471	813	518	4	76.33	--	--	--	Spartan3 Hardware: 1x8-bit *mulgf2*
		131						--	--	--	
		163						--	--	--	
		193						--	--	--	
[91]	8	163	392	517	299	4	91.00	7,917,000	87.00	*34.10	Spartan3 Hardware/Software: 3x8-bit *mulgf2*
		283						33,761,000	371.00	*145.43	
		571						249,340,000	2740.00	*1074.08	
[90]	8	163	452	578	244	4	80.93	15,943,210	197.00	*89.04	Spartan3 Hardware/Software: 1x8-bit *mulgf2*
		233						42,083,600	520.00	*235.04	
		283						73,160,720	904.00	*408.61	
		409						151,339,100	1870.00	*845.24	
		571						210,418,000	2600.00	*1175.20	
[93]	32	163	2418	---	--	--	79.64	68,813	0.86	*2.08	Spartan3 Hardware: 1x32-bit *mulgf2*
		233						155,857	1.96	*4.74	
		283						200,182	2.51	*6.07	
		409						550,370	6.91	*16.71	
		571						1,312,461	16.48	*39.85	
[63]	8	131	543	847	417	4	76.44	732,295	9.58	*5.20	Sparatn3 Hardware: 3x8-bit *mulgf2*
		163						1,311,710	17.16	*9.32	
		283						5,812,497	76.04	*41.29	
		571						44,916,144	587.60	*319.07	
ours	8	163	2377	4,269	555	0	40.75	235,687	5.78	13.74	Spartan3 Hardware: 2x8-bit *mulgf2*
		233						667,866	16.39	38.96	
		283						1,155,523	28.35	97.39	
		409						3,429,119	84.14	200.00	
		571						9,090,409	223.06	553.98	
ours	8	163	1260	2,961	698	0	63.74	235,687	3.70	4.66	Spartan6 Hardware: 2x8-bit *mulgf2*
		233						667,866	10.48	13.21	
		283						1,155,523	18.13	22.84	
		409						3,429,119	53.80	67.79	
		571						9,090,409	142.63	179.71	

* memory resources (equivalent slices) are not included in the area-time product. LUTs: Look_Up_Tables, FFs: Flip-Flops, *kp*: point multiplication

GF2MUL instead of three 8-bit *GF2MUL*) to get very low latency ECC to enable far better performance(are-time metric) than the low area scalable ECC in [91]. The works in [90] and [91] show very poor performances due to very high latency for the point multiplication.

In [93], the authors consider scalable hardware ECC targeting for server-end applications. The ECC implemented using 32-bit *GF2MUL* circuit with three stages pipelining to reduce the latency of the multiplication geometrically. They use a large reduction circuit to support the fast reduction operation of the pentavalent and trivalent-based multiplication result. The 32-bit structure is prohibited to use in the low area application due to large area 2418 slices and several block RAMs for memory(not shown) with a frequency of 80 MHz. Their

implementation shows better results by using the advantages of 32-bit architecture. Our 8-bit architectures shows a comparable performance when the 32-bit based ECC is compared with our state of the art.

In [63], a standalone hardware version is presented for the point multiplication. They consider only pentavalent irreducible based Elliptic curves (i.e. $GF(2^{131})$, $GF(2^{163})$, $GF(2^{283})$, *and* $GF(2^{571})$) to enable a low area ECC architecture. They used a microprogramming approach of control circuit to reduce the area. They use three 8-bit *GF2MUL* unit to reduce the latency of point multiplication. Moreover, the three parallel multipliers also accelerate their square operation. Their area-efficient hardware ECC utilises area of 543 slices and 4 BRAM with 76 MHz. The work takes the total time to compute point multiplication over $GF(2^{163})$ and over $GF(2^{571})$ are 17.16ms and 587 ms. Our proposed ECC can compute the point multiplication as 3 times faster than that of the ECC presented in [63]. Their ECC shows poor performance than our two *GF2MUL*-based ECC; however, they use three *GF2MUL* multipliers to accelerate the point multiplication.

Our proposed scalable ECC consumes very lower latency than relevant scalable ECC to date. In particular, the latency of the relevant ECC adopted three *GF2MUL* multipliers in [63] over $GF(2^{163})$ (1,211,710 clock cycles) is 6 times more and over $GF(2^{571})$ (44,916,144 clock cycles) is 5 times more than our proposed two *GF2MUL* multipliers based ECC over $GF(2^{163})$ (235,687 clock cycles) and over $GF(2^{571})$ (9,090,409 clock cycles) respectively. Our novel multiprecision multiplier and novel multiprecision square circuit contributes to reduce the latency of point multiplication promptly. Moreover, our proposed careful scheduling of point multiplication and repeated squaring operation step up the speed of the ECC by reducing latency for a loop of the point multiplication.

6.7 Conclusion

Scalable elliptic curve cryptography offers an important flexibility in the cryptography based secure communications system to meet upcoming high security requirement. Hardware platform (i.e. FPGA) offers better optimisation than software platform because of simplicity to duplicate resources for parallel operation with small area overhead and easy to control a complex system to generate a fair result.

In this Chapter, we propose a low latency novel multiprecision multiplier and a novel square circuit to enable a low-latency standalone hardware implementation of scalable ECC

for the low resource applications. We propose a parallel Comba multiprecision multiplier using two *GF2MUL* that can compute a multiplication with free (on the fly) reduction using only $s^2/2$ latency (the theoretical latency). We also proposed a parallel approach of our proposed architecture that can also perform multiplication with on the fly reduction. For example, if we just duplicate the two of *GF2MUL* to four of *GF2MUL*, then the proposed architecture can compute the multiplication using half of the latency of two *GF2MUL* based multiplier. Moreover, we present a novel very low latency square circuit latency to do digit square operation. The proposed multiprecision square circuit can perform repeated squaring immediately after a square operation without extra latency for loading. Our proposed square circuit can provide the first result after three clock cycles, including one clock cycle to load and two clock cycles for squaring with reduction. Thus, the proposed square circuit consumes $s+2$ clock cycles for s (where $s=m/w$, $word=w$) words squaring. The low latency of the square circuit can be significant when it is considered in a low latency multiplication (more parallel *GF2MUL*) based ECC.

In the proposed low latency scalable ECC, we utilise combined Montgomery point multiplication using a careful scheduling. The scheduling can manage to perform concurrent operations so that the latency of the each loop operation depends on the six field multiplications. Moreover, the scheduling smartly manages to utilise the memory resources by transferring intermediate results to the empty register. In addition, we propose a low resource SRL16 (except BRAM) based register file (memory unit) to evaluate actual resource constraint for the low resource applications.

Finally, we implemented a standalone scalable ECC on the low cost FPGA such as Spartan3 and Spatan6 instead of Virtex FPGA. Our implementation shows several time faster point multiplication than the most relevant hardware scalable ECC. For the point multiplication over $GF(2^{163})$ and $GF(2^{571})$, our implementation takes 5.78 ms and 223.06 ms respectively on Spartan3 and 3.70 ms and 142.63 ms respectively on Spartan6 that translates the fastest time to date when the ECC is compared with the reported scalable ECC. Again, the speed of our ECC is many times faster than the software or hardware-software implementation because of the low latency ECC. The main contribution behind the performance is the utilisation of novel multiprecision multiplier and square circuit. Finally, the proposed ECC with scalability/flexibility can be one of the best solutions for the low resource applications to meet high security requirement.

Chapter 7 Conclusions and Future Research Work

This chapter presents a summary of the contribution of the thesis and forecasts some potential future research work. The chapter summarize the main contributions of the thesis that are separately discussed in chapter3 to chapter6. The presented research work can open up some potential future works that can be explored.

7.1 Conclusions

The research works presented in the thesis focus on the efficient elliptic curve cryptography on FPGA to apply in the public key cryptography effectively at different application levels. The research presents several novel contributions to meet constraints such as area, time, latency, and frequency for the low area to high-speed ECC applications. Although many relevant works are presented in the literature, most of the work fails to meet the required constraints. The previous hardware implementations consider straightforward algorithms that are mostly developed for the software environment. Again, the hardware platform is vital for public key cryptography because of expensive computation overhead. Specially, the hardware platform on FPGA is popular as the FPGA plays a role to bridge the gulf between software and hardware. Moreover, FPGA hardware implementation has small development time and viability in the commercial applications. Thus, to enable an efficient hardware ECC, distinct contributions are required that are from arithmetic level to point multiplication level.

Low area ECC design needs mainly to reduce the area. The reduction of area may incur large latency that thwarts the advantage of the low area. The thesis presents a low area ECC implementation in Chapter 3 over $GF(2^m)$ on FPGA for the device-constrained applications. The proposed low area ECC consumes a very low area to compute the fastest point multiplication than the relevant works to date. Thus, the proposed ECC shows the best performance by showing the best both area-time and area^2-time metric when the low area ECC is compared with the state-of-the-art to date. The underlying contributions of the performance of the low area ECC are low cost arithmetic unit, flexible memory unit, simple interface circuit and finally, dedicated control unit. The low cost arithmetic unit utilises MSB first bit/small digit (4-bit) serial multiplier, dedicated square circuit, and multiplicative inversion operation. The low area design considers modified Montgomery algorithm, basic binary algorithm and Frobenius map based point multiplications. The proposed ECC shows different aspects of the merits such as the Montgomery algorithm based ECC shows very quick point multiplication, the binary algorithm based ECC consumes a very low area and the Frobenius map based ECC consumes very low latency. The intermediate results of the work were published at the IEEE international conference on ICECS 2013.

High speed ECC utilises digit serial multiplier to enable high throughput ECC. Most of the work in the literature has some problems, including consumption of large area, high latency for the point multiplication and poor maximum frequency to meet the required throughput. The

thesis presents in Chapter 4 a breakthrough of high speed digit-serial multiplier over $GF(2^m)$ to apply in the high throughput ECC implementation. The proposed multiplier utilises a novel segmented pipelining technique that exhibits high performance to apply in a low latency and high-speed multiplication. In particular, a careful scheduling and low resource arithmetic unit contribute to achieve a very high frequency to operate point multiplication while the ECC is consuming very low resources. The proposed multipliers based ECC implementation shows the best throughput/area result to date as compared to the reported relevant works. The works were published in IEEE Transactions on Circuits and Systems-II.

The speed of ECC is an important parameter for the server end ECC applications. Many high-speed works focus on the stepping up the speed by using large resources without major development in the arithmetic level and point multiplication level. The published works fail to reduce the latency due to overuse of pipelining and they fail to increase the frequency due to high complexity to permit high efficiency ECC. The thesis shows in Chapter 5 a new milestone to achieve the fastest speed for point multiplication of ECC over $GF(2^m)$. The key contribution of the fastest design is the utilising a novel high performance full-precision multiplier and careful scheduling of the combined Montgomery point multiplication algorithm. The work utilises three different sets of the novel full-precision multiplier to achieve the high speed individually as compared to the reported high speed ECC on FPGA. The one-full-precision multiplier based ECC consumes a very low area to achieve faster speed than the previous reported fastest speed. The performance of one-multiplier based ECC in the area-time metric shows the best metric than any other relevant high-speed works. The two-multiplier based ECC shows better speed by exploiting low latency and high operating frequency on FPGA. The final version of the high speed ECC considers three full-precision multipliers. The three multipliers based ECC outperforms in speed by achieving theoretical limit of latency with a fair operating frequency. The two multiplier based high speed ECC result was presented at the international conference on FPL2015. The rest of the work is accepted to publish in IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

The shorter key length of ECC offers the flexibility to scalable security in the low resource ECC applications. The low area scalable ECC is becoming an interesting research area, thanks to multiprecision multiplier. The Comba based multiprecision multiplier is widely used to develop scalable ECC. In Chapter 6, the research reveals a new progress in the multiprecision arithmetic based scalable ECC by introducing a new technique in the multiprecision

multiplication and a new multiprecision squaring circuit over $GF(2^m)$. The proposed multipliers can compute Comba multiprecision multiplication using a close to theoretical latency, with on the fly reduction operation as a bonus. The multipliers offer a parallel version of multiplication with concurrent reduction operations utilising small overhead of the area. Again, the proposed multiprecision square circuit with the reduction consumes very low latency to accelerate the point multiplication of the ECC. The low latency arithmetic operations contribute to achieve a very low latency scalable ECC which can perform point multiplication over all NIST curves. A careful scheduling and a combined Montgomery point multiplication are adopted to achieve low latency loop operation of the point multiplication. The actual logic consumption of the ECC is explored by using SRL16 based memory unit to measure the feasibility of ECC in the low resource applications. The proposed scalable ECC shows the fastest computation of the point multiplication when the ECC is compared with the most relevant work to date. Moreover, the proposed scalable ECC outperforms in performance of the area-time to date. This work is planned to submit for publication in a peer-reviewed journal.

7.1 Future Research Works

More investigation upon the existent research

The thesis contributions lead some potential future researches to explore the ECC implementations. The low area ECC of the thesis shows very low area to achieve high speed. The small increase of area may increase performance. For example, the Montgomery and basic binary algorithm based ECC can increase speed by using parallel multipliers with the help of some low cost shift registers. Again, the Frobenius map based work consumes the very low latency as well as low area. The operating frequency of the Frobenius map can be increased by using pipelining registers in the integer adder (ripple carry adder) with very small area overhead.

The high throughput/area based ECC consumes a very low area to achieve very high speed. The digit serial multiplier used in the work consumes four clock cycles for multiplication. The proposed pipelining technique consumes three clock cycles delay. Thus, a further investigation can be done by using a multiplier with three clock cycles delay without any major modification to get high throughput ECC. For example, the ECC over $GF(2^{163})$ utilising a 41-bit digit serial multiplier consumes 4 clock cycles for each multiplication. The ECC implementation over

GF(2^m) implementation can be achieved using 55-bit digit serial multiplier by utilising the limit of the pipelining delay.

The proposed high-speed ECC techniques adopt very high performance full-precision multiplier. The full precision multipliers based ECC shows very high frequency on the FPGA. The existence FPGA family supports the three multipliers based ECC over *GF*(2^{163}) operating with a high frequency. The three multiplier based ECC over *GF*(2^{571}) can be considered to implement in the future FPGA technology to generate the fastest ECC for the highest security application.

The thesis presented low latency scalable work can open up several future research works by using the merits of the work. The novel technique of Comba multiprecision multiplier with reduction consumes same latency for any irreducible polynomial. The multiplier can utilise any ECC curves to achieve very low latency ECC. The proposed multiprecision square circuit consumes very low latency can also be utilised to accelerate exponentiation in any type of the irreducible based ECC.

The multiprecision-based scalable ECC can increase its performance by using parallel multipliers. For example, the proposed implementation can be increased performance by adopting another two multipliers (for *GF2MUL* operation). Moreover, proposed scalable ECC can be considered for 16-bit data path and 32-bit data path without parallel or with parallel multipliers to quantify the merits of the proposed low latency multipliers and square circuits.

ECC with Side-channel resistance

The countermeasure of side channel attack is an extra overhead of ECC. However, FPGA has resistance against low-level physical attack and black box attack, there are several attacks available on FPGA based ECC implementation such as advance physical attack, fault injections, side channel attacks. To overcome part of the attacks, partial power attack resistant is exercised using algorithmic modification (i.e. Montgomery point multiplication) in the ECC operations. For a robust ECC implementation, the workload may increase double of the existence ECC operation Again, The resources complexity may increase double while a side-channel protection is exercised by using Wave Dynamic Differential Logic (WDDL) based protection. Our proposed efficient ECC can allow countermeasure of side channel attacks because of high efficient ECC. Thus, a further investigation is required to evaluate the performance of ECC with countermeasure of the side channel attacks**.**

Considering different platforms

Now, the software and hardware-software platforms offer some flexibilities such as *GF2MUL* operation in the software environment, multicore operation, instruction set extension based embedded processor with flexible addressing, an FPGA with soft-core (Picoblaze and Microblaze), and FPGA with embedded processor (i.e. system on chip). Our proposed multiprecision algorithms (multiplication and square) are flexible to consider in the implementation in both software and hardware-software platforms. In particular, the proposed parallel operation of the multiprecision multiplication can accelerate ECC operations in the software and software-hardware platforms using the flexibilities of the advancement as a matter of further investigation.

Future public key cryptography based on existing work

Several novel arithmetic operations are proposed in the research work. The underlying technique of the proposed arithmetic circuit can be utilised for integer arithmetic operations. For example, the proposed multiprecision multiplier and proposed digit serial multiplier can be utilised to implement large modular integer multiplication for future cryptography applications (i.e. post-quantum cryptography).

References

[1] D. R. Hankerson, A. J. Menezes and S.A. Vanstone, “Guide to Elliptic Curve Cryptography,” Spinger Verlag, 2004.

[2] B. Schneier. Applied Cryptography. John Wiley, New York, NY, USA, 2nd edition, 1996.

[3] W. Diffie and M. E. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information theory, IT-22(6): 644-654, November, 1976.

[4] FIPS 186-4: Digital Signature Standard (DSS). National Institute of Standards and Technology Gaithersburg, MD 20899-8900 Issued July 2013. Available to download at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[5] A. K. Lenstra and E. R. Verheul, “Selecting cryptographic key sizes,” J. Cryptol., vol. 14, no. 4, pp. 255–293, 2001.

[6] D. J. Malan, M. Welsh, and M. D. Smith, “A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography,” 2004 First Annu. IEEE Commun. SoC. Conf. Sens. Ad Hoc Commun. Networks, 2004. IEEE SECON 2004.

[7] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, “Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks,” Secur. Priv. Ad-Hoc Sens. Networks, pp. 6–17, 2006.

[8] N. Guillermin, “A high speed coprocessor for elliptic curve scalar multiplications over Fp,” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2010, vol. 6225 LNCS, pp. 48–64.

[9] M. N. Hassan and M. Benaissa, “Small footprint implementations of scalable ECC point multiplication on FPGA,” in IEEE International Conference on Communications, 2010.

[10] W. N. Chelton and M. Benaissa, “Fast elliptic curve cryptography on FPGA,” IEEE Trans. Very Large Scale Integr. Syst., vol. 16, no. 2, pp. 198–205, 2008.

[11] W. Stallings, Cryptography and Network Security- Principles and Practices. Fifth Edition, Publisher: Prentice Hall 2006.

[12] F. R-Henriquez, N.A. Saqib, A. Diaz-Perez and C.K. Koc, “Cryptographic Algorithm on Reconfigurable Hardware” Springer series on signals and communication technology, publisher Springer 2006.

[13] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Communications of the ACM, vol. 21, no. 2. pp. 120–126, 1978.

[14] P. Gallagher, “Secure Hash Standard (SHS) FIPS PUB 180-4,” Processing, vol. FIPS PUB 1, no. October, 2012.

[15] R. Rivest, “The MD5 message-digest algorithm,” Network WorkingGroup, Request for Comments (RFC) 1321, Apr. 1992.

[16] M. J. Dworkin **“**SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” publish August 04, 2015. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.202.

[17] N. Koblitz, “Elliptic curve cryptosystems,” Mathematics of Computation, vol. 48, no. 177. pp. 203–203, 1987.

[18] V. Miller, “Use of elliptic curves in cryptography,” Adv. Cryptol. — CRYPTO ’85 Proc., vol. 218, pp. 417–426, 1986.

[19] ANSI X9.62: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). American National Standard Institute, New York, USA, 1999.

[20] IEEE P1363-2000: IEEE Standard Specifications for Public-Key Cryptography. IEEE Computer Society Press, Silver Spring, MD, USA, 2000.

[21] ISO/IEC 15946-5:2009: Information Technology-Security Techniques-Cryptographic Techniques based on Elliptic Curves. International Organisation for Standardisation, Geneva, Switzerland, 2009.

[22] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, vol. 106. 1997.

[23] E. R. Berlekamp, “Algebraic Coding Theory”, McGraw-Hill Book Company, New York, 1968.

[24] R. E. Blahut, “Theory and Practice of Error Control Codes”, Addison-Wesley Publication Co. 1983.

[25] R. Lidl, and H. Niederreiter, “Introduction to finite fields and their applications,” Cambridge University Press, Cambridge, UK, Revised Additions 1994.

[26] National Institute of Standards and Technology (NIST). “Recommended Elliptic Curves for Federal Government use,” Available at http://csrc.nist.gov/encryption/2000.

[27] G. Orlando and C. Paar, “A super-serial Galois fields multiplier for FPGAs and its application to public-key algorithms,” Seventh Annu. IEEE Symp. Field-Programmable Cust. Comput. Mach. (Cat. No.PR00375), 1999.

[28] J. Großschadl, “a low-power bit-serial multiplier for finite fields GF(2^m), “ In Proceedings of the 34th IEEE Int. Symposium on Circuits and Systems (ISCAS-2001), vol. IV, pp. 37-40, 2001.

[29] S. Kumar, T. Wollinger, and C. Paar, “Optimum digit Serial GF(2^m) multipliers for curve-based cryptography,” IEEE Trans. Comput., vol. 55, no. 10, pp. 1306–1311, 2006.

[30] L. Song and K. Parhi, “Low-energy digit-serial/parallel finite field multipliers,” J. VLSI signal Process. Syst. signal, vol. 166, pp. 149–166, 1998.

[31] H. Fan, M. A. Hasan, "A survey of some recent bit-parallel multipliers," Elsevier, Vol. 32, pp. 5-43, 2015.

[32] I. Blake, G. Seroussi and N. Smart, “Elliptic Curves in Cryptography,” London Mathematical Society, Lecture Note Series 265. Cambridge University Press, 1999.

[33] N. Koblitz, “Algebraic Aspects of Cryptography,” Springer-Verlag, ACM, vol. 3, 1998.

[34] P. L. Montgomery, “Speeding the Pollard and elliptic curve methods of factorization,” Mathematics of Computation, vol. 48, no. 177. pp. 243–243, 1987.

[35] J. López and R. Dahab, “Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation,” in Lecture Notes in Computer Science: Advances in Cryptology - CRYPTO 1999 Proceedings, 1999, pp. 316–327.

[36] N. Koblitz, “CM-Curves with Good Cryptographic Properties,” in CRYPTO, vol. 576 of Lectures Notes in Computer Science, pp. 195-249, Mar. 2003.

[37] J. Lutz and A. Hasan, “High Performance FPGA based Elliptic Curve Cryptographic Co-Processor,” in International Conference on Information Technology: Coding and Computing - ITCC, Vol.2. 2004, pp. 486–492.

[38] J. Lopez and R. Dahab, “An overview of Elliptic Curve Cryptography,” Technical report, May 2000.

[39] N. Koblitz, A. Menezes, and S. Vanstone, “The State of Elliptic Curve Cryptography,” Des. Codes Cryptogr., vol. 193, no. 2, pp. 173–193, 2000.

[40] G. Meurice de Dormale and J. J. Quisquater, “High-speed hardware implementations of Elliptic Curve Cryptography: A survey,” J. Syst. Archit., vol. 53, no. 2–3, pp. 72–84, 2007.

[41] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," ACM Computing Surveys, vol. 34, no. 2. pp. 171–210, 2002.

[42] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," Foundations and Trends® in Electronic Design Automation, vol. 2, no. 2. pp. 135–253, 2007.

[43] Xilinx Documentations. [online] Available at http://www.xilinx.com/support.html.

[44] J.A. Solinas, "An improved Algorithm for Arithmetic on a Family of Elliptic Curves," Advances in Cryptology- CRYPTO'97, pp. 357-371, Springer, 1997.

[45] J.-P, Deschamps and G.Sutter,"Elliptic-urve Point-Multiplication Over GF(2^{163})," in Proc. IEEE Conf. on PL, pp. 25-30, 2008.

[46] S.Kumar and C. Paar. "Are Standards Compliant Elliptic Curve Cryptosystems Feasible on RFID?," Printed handout of Workshop on RFID Security (RFID Sec 06), 2006.

[47] L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede, "Flexible Hardware Architectures For Curve-Based Cryptography," in Proc. IEEE ISCAS , 2006.

[48] K. Jarvinen , M. Tommiska, and J. Skytta, " A Scalable Architecture For Elliptic Curve Point Multiplication," in Proc. IEEE int. Conf. on Field-Programmable Technology, pp. 303–306, 2004.

[49] M. Morales-Sandoval, C. Feregrino-Uribe, R. Cumplido, and I. Algredo-Badillo, "A Reconfigurable GF(2^m) Elliptic Curve Cryptographic Coprocessor," in proc. IEEE 2011 VII Southern Conf. on SPL, Cordoba, pp. 209–214, 2011.

[50] J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi and I. Verbauwhede, "A Compact FPGA-Based Architecture For Elliptic Curve Cryptography Over Prime Fields," in proc. 21st IEEE Int. Conf. on ASAP, pp. 313–316, 2010.

[51] S. Antao, R. Chaves and L. Sousa," Compact and Flexible Microcoded Elliptic Curve Processor for Reconfigurable Devices," in proc.17th IEEE Symp. on FCCM , pp. 193–200, 2009.

[52] G. Zied, M. Mohsen and T. Rached, "On The Hardware Design of Elliptic Curve Public Key Cryptosystems Using Programmable Cellular Automata" in proc. IEEE 2nd Int.Conf. on SCS, pp. 1–6, 2008.

[53] H. M. Choi, C. P. Hong and C. H. Kim "High Performance Elliptic Curve Cryptographic Processor Over GF(2^{163})," in proc. 4th IEEE Intl. Symposium on Electronic Design, Test & Applications, DELTA, 2008, pp. 290 – 295.

[54] K. Sakiyama, E. D. Mulder, B. Preneel and I. Verbauwhede, “A Parallel Processing Hardware Architecture For Elliptic Curve Cryptosystems,” in proc. IEEE Intl. Conf. on ICASSP, Vol.3, 2006.

[55] S. l. Antao, R. Chaves, and L. Sousa, “ Efficient FPGA Elliptic Curve Cryptographic Processor over GF(2^m) ,” in proc. IEEE Int. Conf. on ICECE Technology, FPT 2008, Vol.2, pp. 486–492, 2008.

[56] N. Gura, S. C. Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila, “An End-to-End Systems Approach to Elliptic Curve Cryptography,” in CHES ’02, Springer-Verlag, pp. 349–365, 2003.

[57] C. Shu, K. Gaj, and T. El-Ghazawi, "Low latency elliptic curve cryptography accelerators for NIST curves over binary fields," Proc. IEEE Intl. Conf. on Field-Programmable Technology, pp. 309-310, 2005.

[58] W. Chelton and M. Benaissa, "High-Speed Pipelined EGG Processor on FPGA," IEEE Workshop on Signal Processing Systems Design and Implementation, SIPS '06, pp. 136-141, Oct. 2006.

[59] B. Ansari, M. A. Hasan, , "High-Performance Architecture of Elliptic Curve Scalar Multiplication," IEEE Trans. On Computers, vol.57, no.11, pp. 1443-1453, Nov.2008.

[60] Y. Zhang, D. Chen, Y. Choi, L. Chen and S.-B. Ko, “A high performance ECC hardware implementation with instruction-level parallelism over GF(2^{163}),” Microprocess and. Microsyst., vol. 34, no. 6, pp. 228–236, 2010.

[61] G. Sutter, J. Deschamps, J Imana, "Efficient Elliptic Curve Point Multiplication using Digit Serial Binary Field Operations," IEEE Trans. on Industrial Electronics, vol. 60, no.1, pp. 217-225, 2013.

[62] K.Sakiyama, L.Batina, B. Preneel, and I. Verbauwhede, “Superscalar coprocessor for high-speed curve-based cryptography,” in Proc. of CHES, ser. LNCS, vol. 4249. Springer-Verlag, pp. 415–429. 2006.

[63] M. Hassan and M. Benaissa, "Efficient Time-Area Scalable ECC Processor Using μ-Coding Technique", in Arithmetic of Finite Fields, LNCS, Springer, vol. 6087, 2010, pp. 250-268.

[64] M. Amara, and A. Siad, “Hardware implementation of arithmetic for elliptic curve cryptosystems over GF(2^m),” in proc 2011 IEEE conf. on worldcis, pp. 73–78,21-23, 2011.

[65] T.Itoh and S. Tsujii,”A fast algorithm for computing multiplicative inverses in GF (2^m) using normal bases,” info. Comput., vol. 78, no.3, pp. 171-177, 1988.

[66] K.C.C. Loi and S.-B. Ko, “High performance Scalable Elliptic Curve Cryptosystem Processor for Koblitz Curves,” Microprocessors and Microsystems, vol. 37, pp. 394-406, 2013.

[67] L. Deng, K Sobti and C. Chakrabarti, “Accurate models for estimating area and power of FPGA implementations,” IEEE Int. Conf. on ICASSP, pp. 1417-1420, 2008.

[68] P. Tuyls and L. Batina, “RFID-tags for Anti-Counterfeiting,” CT-RSA 2006, Springer Verlag. pp. 115-131, 2006.

[69] D. Knuth, “The Art of Computer Programming”, Vol.2, Semi-numerical Algorithm, 3rd e, Addison-Wesley, 1997.

[70] Z. Khan and M. Benaissa, "Low area ECC implementation on FPGA," in Proc. IEEE 20th ICECS, Dec. 8-11, 2013, pp. 581-584.

[71] S. Roy, C. Rebeiro, and D. Mukhopadhyay, “Theoretical Modeling of Elliptic Curve Scalar Multiplier on LUT-Based FPGAs for Area and Speed,” IEEE Trans. VLSI Syst., vol. 21, no. 5, pp. 901–909, May. 2013.

[72] H. Mahdizadeh, and M. Masoumi, “Novel Architecture for Efficient FPGA Implementation of Elliptic Curve Cryptographic Processor Over GF(2163),” IEEE Trans. VLSI Systems, vol. 21, no. 12, pp. 2330-2333, Dec. 2013.

[73] R. Azarderakhsh and A. Reyhani-Masoleh, “Efficient FPGA implementations of point multiplication on binary Edwards and generalized Hessian curves using Gaussian normal basis,” IEEE Trans. VLSI Systems, vol. 20, no. 8, pp. 1453-1466, Aug. 2012.

[74] J.-S. Pan, R Azarderakhsh, M. M. Kermani, C.-Y Lee, W.-Y. Lee, C. W. Chiou, and J.-M Lin, "Low-Latency Digit-Serial Systolic Double Basis Multiplier Over GF(2m) Using Subquadratic Toeplitz Matrix-Vector Product Approach," IEEE Trans. on Comp., Vol. 63, no. 5, pp. 1169-1181, 2014.

[75] C.-Y. Lee, C.-S. Yang, B. K. Meher, P. K. Meher, and J.-S. Pan, "Low-Complexity Digit-Serial and Scalable SPB/GPB Multipliers over Large Binary Extension Fields using (b,2)-Way Karatsuba Decomposition," IEEE Trans. Circuits and Syst.-I, vol. 61, no. 11, pp. 3115-3124, 2014.

[76] C.-Y. Lee, "Super Digit-Serial Systolic Multiplier over GF(2^m)," The Sixth ICGEC., Aug.25-28, 2012, pp. 509-513.

[77] U. S. Department of Comerce/NIST, “National Institute of Standards and Technology,” Digital Signature Standard, FIPS Publications 186-2, January 2000.

[78] M. A. Hasan, A.H. Namin, and C. Negre., "Toeplitz Matrix Approach for Binary Field Multiplication Using Quadrinomials," IEEE Transactions on VLSI Systems, vol. 20, no. 3, pp. 449-458, March, 2012.

[79] B. Rashidi, R.R. Farashahi, S.M. Sayedi, “High-speed and pipelined finite field bit-parallel multiplier over GF(2^m) for elliptic curve cryptosystems,” in Proc. 11th Int. ISC Conf. on Info. Security and Cryptology (ISCISC), 2014, pp. 15-20.

[80] C. Rebeiro, S. Roy, and D. Mukhopadhyay, “Pushing the Limits of High-Speed GF(2^m) Elliptic Curve Scalar Multiplication on FPGAs,” lecture Notes in Comp. Sc.–CHES 2012 vol. 7428, pp. 496-511.

[81] S. Liu, L. Ju, X. Cai, Z. Jia, Z. Zhang, “High Performance FPGA Implementation of Elliptic Curve Cryptography over Binary Fields,”in proc. 13th IEEE Int. Conf. on Trust, Security and Privacy in Comp. and Communications(TrustCom), 2014, pp. 148-155.

[82] A.P. Fournaris, J. Zafeirakis, and O. Koufopavlou, "Designing and Evaluating High Speed Elliptic Curve Point Multipliers," in proc. 17th Euromicro Conf. on, Digital System Design (DSD), 2014, pp. 169-174.

[83] Z. Zia-Uddin-Ahamed Khan and M. Benaissa, “Throughput/Area Efficient ECC Processor using Montgomery Point Multiplication on FPGA,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 62, no. 11, pp. 1078-1082, Nov. 2015.

[84] Z.U.A. KHAN, M. Benaissa, “High Speed ECC Implementation of FPGA over GF(2^m),” In proc. on 25th International Conference on Field-programmable Logic and Applications (FPL), 2-4 Sept. 2015, pp. 1-6.

[85] S. Bartolini, I. Branovic, R. Giorgi, E. Martinelli, “Effects of Instruction-Set Extensions on an Embedded Processor: A Case Study on Elliptic Curve Cryptography over GF(2^m),” IEEE Trans.Computers, vol.57, no.5, pp. 672-685, May 2008.

[86] P. Comba, “Exponentiation cryptosystems on the IBM PC”, IBM Systems Journal, 29(4), pp. 526-538, 1990.

[87] C. K. Koc and T. Acar, “Montgomery multiplication in GF(2^k)”, Designs, Codes and Cryptography, 14(1), pp. 57-68, April 1998.

[88] J. Groszschaedl, G.A. Kamendje, "Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m)," in Proc. on IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors, 24-26 June 2003, pp. 455-468.

[89] Hassan, M.N.; Benaissa, M., "Embedded Software Design of Scalable Low-Area Elliptic-Curve Cryptography," in IEEE Embedded Systems Letters, vol.1, no.2, pp. 42-45, Aug. 2009.

[90] M. N. Hassan and M. Benaissa, “A scalable hardware/software co-design for elliptic curve cryptography on picoblaze microcontroller,” in proc. on IEEE Int. Symp. Circuits Syst. Nano-Bio Circuit Fabr. Syst., 2010, pp. 2111–2114.

[91] M. N. Hassan, M. Benaissa, and a. Kanakis, “Flexible hardware/software co-design for scalable elliptic curve cryptography for low-resource applications,” in proc. on Int. Conf. Appl. Syst. Archit. Process., no. 2, pp. 285–288, 2010.

[92] M. N. Hassan and M. Benaissa, "Low Area-Scalable Hardware/Software Co-Design for Elliptic Curve Cryptography," in proc. on 3rd Int. Conf. in New Technologies, Mobility and Security (NTMS), 2009, pp. 1-5.

[93] K.C.C. Loi, and K. S-B. Ko, "High performance scalable elliptic curve cryptosystem processor in GF(2^m)," in proc. on IEEE Int. Sympo. on Circuits and Systems (ISCAS), 19-23 May 2013, pp. 2585-2588,

[94] J. Stein, “Computational problems associated with Racah algebra,” Journal of Computational Physics, vol.1, pp. 397-405, 1967.

[95] XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices [online], XILINX Inc, March 20, 2013 Available:http://-www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/xst.pdf.

[96] None (2015) Cerberus FTP Server supports Elliptic Curve Cryptography, [online] Available at: https://www.cerberusftp.com/products/features/cerberus-ftp-server-elliptical-curve-cryptography.html