Hardware Implementations of Scalable and Unified Elliptic Curve Cryptosystem Processors

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

By

Kung Chi Cinnati Loi

Permission to Use

Abstract

As the amount of information exchanged through the network grows, so does the demand for increased security over the transmission of this information. As the growth of computers increased in the past few decades, more sophisticated methods of cryptography have been developed. One method of transmitting data securely over the network is by using symmetric-key cryptography. However, a drawback of symmetric-key cryptography is the need to exchange the shared key securely. One of the solutions is to use public-key cryptography.

One of the modern public-key cryptography algorithms is called Elliptic Curve Cryptography (ECC). The advantage of ECC over some older algorithms is the smaller number of key sizes to provide a similar level of security. As a result, implementations of ECC are much faster and consume fewer resources. In order to achieve better performance, ECC operations are often offloaded onto hardware to alleviate the workload from the servers' processors.

The most important and complex operation in ECC schemes is the elliptic curve point multiplication (ECPM). This thesis explores the implementation of hardware accelerators that offload the ECPM operation to hardware. These processors are referred to as ECC processors, or simply ECPs. This thesis targets the efficient hardware implementation of ECPs specifically for the 15 elliptic curves recommended by the National Institute of Standards and Technology (NIST).

The main contribution of this thesis is the implementation of highly efficient hardware for scalable and unified finite field arithmetic units that are used in the design of ECPs. In this thesis, scalability refers to the processor's ability to support multiple key sizes without the need to reconfigure the hardware. By doing so, the hardware does not need to be redesigned for the server to handle different levels of security. Unified refers to the ability of the ECP to handle both prime and binary fields. The resultant designs are valuable to the research community and industry, as a single hardware device is able to handle a wide range of ECC operations efficiently and at high speeds. Thus, improving the ability of network servers to handle secure transaction more quickly and improve productivity at lower costs.

Acknowledgements

This thesis is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Alexander Graham Bell Graduate Scholarship and the University of Saskatchewan College of Graduate Studies and Research Dean's Scholarship. I would like to thank my supervisor, Dr. Seok-Bum Ko, for his support in the past 4 years of my studies and the advice he has given me along the way. Also, thanks to Rylan for his input on the SSL protocol and countless post-midnight online chats to discuss the possibilities of my thesis. A special thanks goes out to Mandy and her family for keeping me sane during the most stressful times of my life. Most importantly, I would also like to thank my family for their continuous support in my life, without them I would not be where I am today!

Contents

Permission to Use **i**

Abstract **ii**

Acknowledgements **iii**

Contents **iv**

List of Tables **vi**

List of Figures **viii**

List of Algorithms **x**

List of Abbreviations **xi**

List of Publications **xiii**

1 Introduction **1**
- 1.1 Literature Review of ECC Processors 4
- 1.2 Motivation 7
- 1.3 Description of the Research and Major Contributions 10
- 1.4 Organization of Thesis 12

2 Background **13**
- 2.1 Introduction to Cryptography 13
- 2.2 Elliptic Curve Cryptography 15
- 2.3 Finite Field Arithmetic 23
  - 2.3.1 Finite Field Arithmetic in Binary Fields 23
  - 2.3.2 Finite Field Arithmetic in Prime Fields 32
- 2.4 Xilinx Virtex-5 FPGA and XtremeDSP Slices 34

3 Scalable ECC Processors for Binary Curves **39**
- 3.1 163-bit ECC Processor 39
- 3.2 Scalable ECC Processor for Binary Curves 42
  - 3.2.1 Scalable ECP over Koblitz Curves 43
  - 3.2.2 Scalable ECP over Pseudo-Random Curves 48
- 3.3 Parallelization of Scalable ECC Processor for Binary Curves 51
  - 3.3.1 Parallelization of Scalable ECP over Koblitz Curves 51
  - 3.3.2 Parallelization of Scalable ECP over Pseudo-Random Curves 61
- 3.4 Low Latency Scalable ECC Processor for Binary Curves 69

4 Scalable ECC Processor for Prime Curves **73**
4.1 Design and Architecture 73
4.2 Implementation Results 81

5 Scalable and Unified ECC Processor **84**
5.1 Design and Architecture 84
5.2 Hardware/Software Co-design 102

6 Conclusion **105**

7 Future Work **109**
7.1 Koblitz τNAF Conversion 109
7.2 Post-Quantum Cryptography 111
7.3 Identity-Based Encryption 112

References **114**

Appendix A Binary Fields Reduction Algorithms **121**

Appendix B Prime Fields Reduction Algorithms **125**

List of Tables

1.1 Comparison between ECC and RSA Security [13] 2
1.2 Summary of ECPs in the Current Literature 8

2.1 NIST Recommended Irreducible Polynomials 27
2.2 Decomposition of $m-1$ for Itoh-Tsujii Algorithm 30
2.3 OPMODE control signals for X Multiplexer 36
2.4 OPMODE control signals for Y Multiplexer 36
2.5 OPMODE control signals for Z Multiplexer 36
2.6 Three-input ALU operations 37
2.7 Two-input ALU operations 37

3.1 Implementation Results and Comparison of the 163-bit ECP 41
3.2 Implementation Results and Comparison of the Koblitz ECP 47
3.3 Implementation Results and Comparison of the Random ECP 50
3.4 Latency in number of clock cycles of the MULT and SA blocks. 56
3.5 Instructions executed by the 1-MULT Koblitz ECP 57
3.6 Instructions executed by the 2-MULT Koblitz ECP 59
3.7 Implementation Results and Comparison of the Parallelized Scalable ECPs for Koblitz Curves 61
3.8 Instructions executed by the 1-MULT Random ECP 65
3.9 Instructions executed by the 2-MULT Random ECP 67
3.10 Implementation Results and Comparison of the Parallelized Scalable ECPs for Pseudo-Random Curves 68
3.11 Implementation Results of the Low-Latency ECPs 72

4.1 Operation sequence for the addition/subtraction/reduction (AR) block for p_{192} 76
4.2 Latency in number of clock cycles of the MULT and AR blocks in the Prime ECP. 77
4.3 Instructions executed during PDBL and PADD states by the Prime ECP 80
4.4 Instructions executed during inversion and FINAL states by the Prime ECP 81
4.5 Implementation Results and Comparison of the Prime ECP 82

5.1 Latencies in clock cycles of the MULT and AR blocks in the Scalable and Unified ECP. 89
5.2 Instructions executed in the PRIME states of the Scalable and Unified ECP 93
5.3 Instructions executed in the KOBLITZ states of the Scalable and Unified ECP 94
5.4 Instructions executed in the RANDOM states of the Scalable and Unified ECP 95
5.5 Latency comparison of the inversion algorithms 96
5.6 Instructions executed in the BINARY INVERSION states of the Scalable and Unified ECP 98
5.7 Instructions executed in the PRIME INVERSION states of the Scalable and Unified ECP 98

5.8 Instructions executed in FINAL states of the Scalable and Unified ECP 99
5.9 Implementation Results and Comparison of the Scalable and Unified ECP 100

List of Figures

1.1 Levels of abstraction in ECC implementations 3

2.1 Simple communication model of an unsecured channel. 13
2.2 Communication model with symmetric-key cryptography. 14
2.3 Diffie-Hellman key exchange. 14
2.4 Example of an elliptic curve ($y^2 = x^3 - 3x + 2$). 16
2.5 Example of point operations on an elliptic curve. 17
2.6 Communication model with Elliptic Curve Diffie-Hellman (ECDH). 22
2.7 Example of a polynomial of degree 324 reduced by $P(t) = t^{163} + t^7 + t^6 + t^3 + 1$ 28
2.8 Simplified block diagram of the DSP48E slice. 34

3.1 ECC processor from Zhang et al. [14] 40
3.2 ALU of the ECC processor from Zhang et al. [14] 40
3.3 Architecture of 2-stage Karatsuba-Ofman multiplier 42
3.4 Block diagram of the Finite Field Arithmetic Unit. 44
3.5 Block diagram of the Koblitz ECP. 45
3.6 Block diagram of the FFAU of the Random ECP. 48
3.7 Block diagram of Parallelized Finite Field Arithmetic Blocks. 53
3.8 The RAM addresses read in the MULT block for 163-bit operation. 54
3.9 Block diagram of the 1-MULT Koblitz ECP. 54
3.10 FSM of the 1-MULT Koblitz ECP. 55
3.11 Data dependency graph of PDQA for 1-MULT Koblitz ECP. 56
3.12 Block diagram of the revised SA block to support 2 MULT blocks. 58
3.13 Block diagram of the 2-MULT Koblitz ECP. 60
3.14 Block diagram of the 1-MULT Random ECP. 62
3.15 FSM of the 1-MULT Random ECP. 63
3.16 Data dependency graph of LOOP for the 1-MULT Random ECP. 63
3.17 Data dependency graph of coordinate conversion for the 1-MULT Random ECP. 64
3.18 Block diagram of the 2-MULT Random ECP. 66
3.19 Block diagram of the Finite Field Arithmetic Blocks in the Low-Latency ECPs. 70
3.20 Block diagram of the Low-Latency Random ECP. 71

4.1 Block diagram of the Finite Field Arithmetic Blocks in the Prime ECP. 74
4.2 Block diagram of the Prime ECP. 77
4.3 Data dependency graph of PDBL of the Prime ECP. 78
4.4 Data dependency graph of PADD of the Prime ECP. 78
4.5 FSM of the Prime ECP. 79

5.1 Block diagram of the MULT block for the Scalable and Unified ECP. 85
5.2 Block diagram blocks used in the MULT block of the Scalable and Unified ECP. 86
5.3 Block diagram of the AR block for the Scalable and Unified ECP. 88
5.4 Block diagram of the Scalable and Unified ECP. 90

5.5 FSM of the Scalable and Unified ECP. 91
5.6 Sub-FSM for the main loop of the Scalable and Unified ECP. 92
5.7 Sub-FSM for inversion of the Scalable and Unified ECP. 97
5.8 Block diagram of the interface between the Microblaze and the ECP. 102
5.9 Block diagram of Xilinx Fast Simplex Link (FSL) [69]. 103

List of Algorithms

2.1 Left-to-right point multiplication 18
2.2 Lopez-Dahab algorithm 20
2.3 τNAF point multiplication on Koblitz Curves 22
2.4 Comba Multiplication 25
2.5 Binary inversion algorithm in binary fields 31
2.6 Binary inversion algorithm in prime fields 33

7.1 τNAF conversion algorithm 110

A.1 Reduction by $P(t) = t^{163} + t^7 + t^6 + t^3 + 1$ 121
A.2 Reduction by $P(t) = t^{233} + t^{74} + 1$ 122
A.3 Reduction by $P(t) = t^{283} + t^{12} + t^7 + t^5 + 1$ 123
A.4 Reduction by $P(t) = t^{409} + t^{87} + 1$ 123
A.5 Reduction by $P(t) = t^{571} + t^{10} + t^5 + t^2 + 1$ 124

B.1 Reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$ 125
B.2 Reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$ 125
B.3 Reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ 125
B.4 Reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ 126
B.5 Reduction modulo $p_{521} = 2^{521} - 1$ 126

List of Abbreviations

AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
AU	Arithmetic Unit
BRAM	Block RAM
CLB	Configurable Logic Block
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECP	Elliptic Curve Processor
ECPM	Elliptic Curve Point Multiplication
FF	Finite Field
FFAU	Finite Field Arithmetic Unit
FIFO	First-In First-Out
FIPS	Federal Information Processing Standards
FPGA	Field-Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
GF	Galois Field
HSC	Hardware/Software Co-design
IBE	Identity-Based Encryption
LD	Lopez-Dahab
LSB	Least-Significant Bit
LSD	Least-Significant Digit
LUT	Look-Up Table
MSB	Most-Significant Bit
MSD	Most-Significant Digit
NIST	National Institute of Standards and Technology
PADD	Point Addition
PDA	Personal Digital Assistant
PDBL	Point Doubling
PFRB	Point Frobenius Endomorphism
PKG	Private Key Generator
RAM	Random Access Memory
RFID	Radio Frequency Identification
RNS	Residue Number System
RSA	Rivest-Shamir-Adleman
SA	Square-Add

SECG	Standards for Efficient Cryptography Group
SIPO	Serial-In Parallel-Out
SSL	Secure Socket Layer
TLS	Transport Layer Security
XOR	Exclusive OR

List of Publications

- [50] K.C.C. Loi and S. B. Ko, "Improvements for High Performance Elliptic Curve Cryptosystems Processor over $GF(2^{163})$", *International Symposium on Electronic System Design (ISED)*, pp. 140 – 144, December 2012.
- [51] K.C.C. Loi and S.B. Ko, "High Performance Scalable Elliptic Curve Cryptosystem Processor for Koblitz Curves", *Microprocessors and Microsystems*, Elsevier, Volume 37, Issues 4 – 5, pp. 394 – 406, June – July 2013.
- [52] K.C.C. Loi and S.B. Ko, "High Performance Scalable Elliptic Curve Cryptosystem Processor in $GF(2^m)$", *International Symposium on Circuits and Systems (ISCAS)*, pp. 2585 – 2588, May 2013.
- [53] K.C.C. Loi, S. An and S.B. Ko, "FPGA Implementation of Low Latency Scalable Elliptic Curve Cryptosystem Processor in $GF(2^m)$", *International Symposium on Circuits and Systems (ISCAS)*, pp. 822 – 825, June 2014.
- [54] K.C.C. Loi and S.B. Ko, "Scalable Elliptic Curve Cryptosystem FPGA Processor for NIST Prime Curves", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, accepted for future publication.
- [67] K.C.C. Loi and S.B. Ko, "Parallelization of Scalable Elliptic Curve Cryptosystem Processors in $GF(2^m)$", *IEEE Transactions on Computers*, draft submitted to journal for peer review.
- [68] K.C.C. Loi and S.B. Ko, "Efficient Scalable and Unified Elliptic Curve Cryptography Coprocessor using DSP Slices on FPGAs", *IEEE Transactions on Industrial Electronics*, draft submitted to journal for peer review.

Chapter 1
Introduction

The growth of secure online transactions in recent years has created a demand for higher security needs for information transmitted over the Internet, which requires servers of online service providers to process a large number data coming from all the users. In addition, in today's fast-paced society, increasing security at the expense of the users' long wait times for the information to be processed is not desirable. Thus, this thesis looks to improve on the current cryptographic processors to provide a high-speed and secured communication channel for data transmission.

One of the protocols used to establish such a secure channel is the Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS) protocol [1]. The SSL protocol can be separated into two stages: handshaking and bulk-data. In the handshaking stage, the client and the server exchange messages in order to establish a shared secret key using public-key cryptography. The shared secret key is in turn used in the bulk-data stage, which uses private-key cryptography. Generally, public-key cryptography is only used during the handshaking stage due to its computational complexity compared to private-key cryptography. However, even though SSL reduces the number of public-key cryptographic operations by transmitting data using private-key cryptography and by supporting reuse of previously established keys, public-key cryptography is still the most time consuming operation in the transmission channel [2]. Thus, in order to accelerate processing of the public-key cryptography operations, secure server systems offload these complex operations into faster running hardware platforms, such as SSL accelerator cards or standalone SSL processing units [3]. Companies, such as Elliptic Technologies[1] and Broadcom[2], have created such products that either support

[1]`http://www.elliptictech.com/`
[2]`http://www.broadcom.com/`

Table 1.1: Comparison between ECC and RSA Security [13]

Symmetric	ECC	RSA	Protection Lifetime
80	163	1024	Until 2010
112	233	2048	Until 2030
128	283	3072	Beyond 2030
192	409	7680	
256	571	15360	

the complete SSL protocol [4, 5, 6, 7] or simply some of the underlying operations [8, 9].

In recent years, the use of elliptic curve cryptography (ECC) for public-key cryptography has been increasingly popular among researchers and industry members. ECC was independently proposed by Miller [10] and Koblitz [11] in the 1980s. The advantage of ECC over the more commonly used Rivest-Shamir-Adleman (RSA) [12] algorithm for public-key cryptography is that ECC allows for reduced key sizes while providing a similar level of security. Table 1.1 [13] shows the comparison of the key sizes for symmetric-key and public-key cryptosystems and their respective protection lifetimes. The security levels shown assume that the adversary has computational powers limited to the state-of-the-art at the time (i.e. computational security), as opposed to informational theoretic security where the algorithms can be proven to be secure. The shorter key sizes allow implementations of ECC to be more efficient either in terms of higher throughput or lower area. Higher throughput implementations can be found in applications where high speeds are required, such as network servers, at the expense of area and power consumptions [14, 15]. In more restricted environments, such as Personal Digital Assistant (PDA), cell phones and Radio Frequency Identification (RFID) readers [16, 17], the design goal becomes to reduce the area and power consumptions at the expense of lower throughput. Due to these advantages, ECC has been adopted by many standards, such as National Institute of Standards and Technology (NIST) [18], Standards for Efficient Cryptography Group (SECG) [19], and Federal Information Processing Standards (FIPS) 186-3 [20].

In particular, NIST [18] and SECG [19] recommend the use of different specific named curves to be used for various purposes. For example, the United States National Security

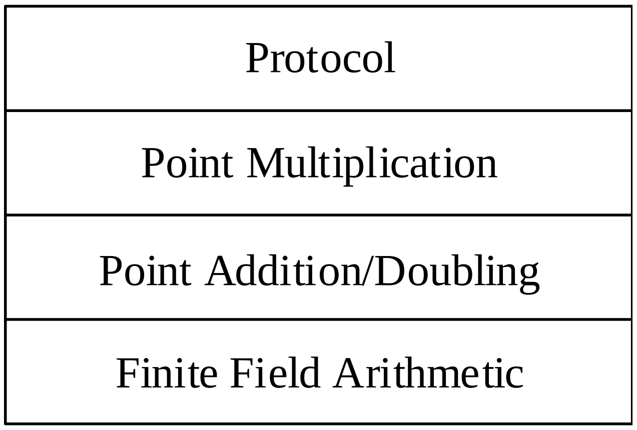


Figure 1.1: Levels of abstraction in ECC implementations

Agency (NSA) chose to use prime field elliptic curves of key size 256, 384 and 521 published by NIST [18] for both classified and unclassified information [21]. These recommended curves can be divided into two categories: prime Galois fields (or prime fields), $GF(p)$, and binary Galois fields (or binary fields), $GF(2^m)$. Furthermore, the binary field curves can be further divided into pseudo-random and Koblitz curves. In order to maximize the usability of a server, it should be able to support all the named curves recommended by the standards. Thus, this thesis investigates scalable and unified ECC processor architectures that can support all the curves recommended by NIST [18].

Offloading complex public-key cryptography operations to hardware can take place in any level of abstraction shown in Figure 1.1. In other words, hardware accelerator implementations can range from offloading only the finite field arithmetic operations to hardware and implementing all the overlaying levels in software to implementing a complete protocol, such as Elliptic Curve Digital Signature Algorithm (ECDSA) or Elliptic Curve Diffie-Hellman (ECDH), including all its underlying levels in hardware, such that the interface merely needs to input the message to be signed, in case of ECDSA, and the module outputs the resultant signed message.

In this thesis, scalability is defined as the ability for the design to be able to support a range of key sizes without the need to redesign or reprogram the hardware, and a unified design is one that is able to compute both $GF(p)$ and $GF(2^m)$ operations using the same hardware. Designing a scalable and unified ECC processor improves the efficiency of the hardware utilization, since the same hardware is able to handle both primes and binary fields and a range of key sizes without the need for any human intervention.

The remainder of this chapter is organized as follows: Section 1.1 looks at the current literature on scalable and unified ECC processors; Section 1.2 further discusses the motivation of this thesis; Section 1.3 describes the research problem and the major contributions of this thesis; and Section 1.4 outlines the organization of the remainder of this thesis.

1.1 Literature Review of ECC Processors

Savas et al. [22] proposed a scalable and unified multiplier in 2000, which uses the Montgomery multiplication algorithm and it can handle operands of any size, but requires precomputations and transformations. The design has been implemented on application-specific integrated circuit (ASIC). Furthermore, it only offloads the finite field arithmetic. In [23], two more dual-field multiplier architectures are presented by Savas et al. that use precomputation technique and dual-radix design to achieve faster computation times in both fields at the expense of larger area utilization on ASIC.

In 2004, Chelton and Benaissa [24] propose a scalable arithmetic unit that can operate over any field in $GF(2^m)$, but not prime fields. It proposes architectures for both a scalable multiplier and a scalable divider that can be used in a ECC processor, but does not design the complete processor.

In [25], Tanimura et al. propose a hardware implementation of the Montgomery multiplication in ASIC using 4 parallel radix-2^{16} multipliers in $GF(p)$ and a radix-2^{64} multiplier in $GF(2^m)$ in order to balance the time delay difference when computing prime field and binary field arithmetic in hardware.

In 2009, Chiou et al. [26] propose a scalable unified multiplier architecture for both fields, but it only supports prime fields of modulo $P = 2^m - 1$ and all-one polynomials (AOP), which most NIST [18] and SECG [19] recommended curves are not.

In all of the above describe implementations, only the finite field arithmetic is offloaded into hardware. Hardware implementations of designs that also include higher levels of abstraction are described below.

In 2001, Goodman and Chandrakasan [27] describes a microcode-based design that implements finite field arithmetic in both fields, point addition, point doubling and point multiplication. The design is implemented using ASIC and provides an energy-efficient solution. It also uses the Montgomery multiplication algorithm.

In 2003, Satoh and Takano [28] propose a scalable dual-field processor that supports any prime or binary field, but only shows results up to 256 bits. The design uses Montgomery multiplication algorithm and is implemented on ASIC. It also supports protocol level operations, such as DSA and ECDSA.

In 2008, Wang et al. [29] present a coprocessor that operates over both fields that is capable of performing both RSA and ECC operations. The implemented arithmetic unit includes multiplication, addition, subtraction and inversion for both prime and binary fields. However, the design only supports 1 field size at a time. Implementation results are presented for both field programmable gate array (FPGA) and ASIC platforms.

In 2008, Lai and Huang [30] proposed a dual-field coprocessor that supports arbitrary prime or binary fields and arbitrary elliptic curves. It optimizes the scheduling of the point operations to increase performence. However, the buffer size of the processor limits the key size of the processor. The paper only presents results up to 256-bit key sizes. Lai and Huang have also presented 2 other designs that improve on the first one in [31, 32].

Chen et al. [33] propose a unified design for both RSA and ECC. The authors implement a microcode-based architecture, where 3 tiers of instruction sets can be executed depending on desired level of abstraction in RSA or ECC operations. Furthermore, the proposed design does not support prime field operations in ECC. The processor is optimized for modular exponentiation in $GF(p)$ and arithmetic in $GF(2^m)$. Therefore, in order to support $GF(p)$ in ECC, it would require support for addition, subtraction, inversion, in addition to multiplication that it currently implements. By doing so, the critical path of the processor would increase, considerably due to the carry propagation of the addition and subtraction.

In [34], Chen et al. propose a 160-bit and 256-bit unified ECC processor implemented on ASIC that can support both fields. It uses a radix-4 division unit to increase the speed of the calculations. However, the processor is not a scalable design.

Lee et al. [35] present a dual-field heterogeneous processing element architecture, where one processing unit performs multiplication-addition/subtraction and the other evaluates all of the above plus division.

There are also architectures in the current literature, which are not unified but are scalable. These designs either implement the ECC processor (ECP) in binary or in prime fields. The binary field ECPs are discussed first. In 2006, Benaissa and Lim [36] proposed a scalable ECC processor that uses a digit-serial multiplier and squaring unit. Thus, the design is scalable and can support multiple fields. In 2009, Hassan and Benaissa [37] proposed a scalable ECC processor that uses the PicoBlaze soft-core microcontroller in Xilinx FPGAs to implement the design using the hardware/software co-design (HSC) approach. The motivation is to design a processor that can handle the elliptic curves up to the 193-bit key size suggested in SECG [19] without the need to reconfigure the hardware. The design goal is to reduce area consumption for area constrained platforms, such as RFID, mobile handsets, smart cards, and wireless sensor networks [37]. Since then, Hassan and Benaissa have also proposed scalable designs that support curves up to 571 bits recommended by the National Institute of Standards and Technology (NIST) [38, 17, 39] also for area-constrained environments.

The design proposed in [40] is also a HSC scalable ECC processor, where the authors use the on-chip PowerPC in Virtex-4 FX series to build the system. However, the reconfiguration of the portion that computes the elliptic curve point multiplication (ECPM) is dynamically reloaded at run time using partial reconfiguration technology on Virtex FPGAs.

The following ECP designs present scalable ECPs in prime fields. In 2006, McIvor et al. [41] presented an ECP that can compute the ECPM over prime fields with less than 256 bits based on a new unified modular inversion algorithm. It is one of the fastest prime field ECPs at the time and it can perform a 256-bit ECPM in 3.86 ms on a Virtex-2 Pro FPGA. One of the strengths of [41] is that it can evaluate ECPM for any curve. However, it cannot evaluate all NIST recommended prime field curves because it can only support up to 256-bits. If the design is extended to 521 bits, hardware utilization will increase immensely and the maximum clock frequency might also suffer due to more difficult routing in the FPGA. Thus, in [42], Ananyi et al. proposed a scalable ECP that can support curves up to 521 bits, but only for NIST recommended prime fields, as opposed to any curve up to 521 bits. The advantage of only supporting NIST recommended curves is the ability to take advantage of the prime moduli selected by NIST, which can be reduced very easily, requiring fewer hardware resources compared to using Montgomery multiplications and inversions as in [41].

Similar to [41], the authors in [42] chose a very wide datapath, using 265 bits for the modular adder, subtractor and multiplier and using 521 bits for the modular inverter. As a result, the implemented design on a Xilinx Virtex-4 FPGA uses 20,793 slices and 32 DSP48 blocks and only runs at 60 MHz. It can compute ECPM for 192-, 244-, 256-, 384-, and 521-bit curves in 4.8 ms, 5.8 ms, 6.9 ms, 19.9 ms, and 45.6 ms, respectively. In 2011, the authors in [43] developed MicroECC, which only has 16- or 32-bit datapaths to make the implementation much smaller and faster.

There are also designs that are neither scalable nor unified. These target a specific curve and can have very optimized designs. In [14], the authors present a highly efficient ECP for the 163-bit pseudo-random curve recommended by NIST. The design uses parallel cores to perform arithmetic. The ECPs in [44] are specifically designed for 163, 233 and 283-bit Koblitz curves recommended by NIST. The design uses 4 parallel multipliers to improve the performance of the ECPM operation. In 2008, Güneysu and Paar [45] optimized the architecture of the ECP using high performance Digial Signal Processing (DSP) slices on FPGAs. In [46], the authors developed a side channel attack resistant ECP using the double-and-add-always algorithm. The authors of [47] and [48] use the residue number system (RNS) to parallelize the operations in the ECP.

A summary of the works reviewed in this section is provided in Table 1.2. The table describes the technology used and the level of abstraction implemented on hardware. It indicates whether or not the designs are scalable and the finite field that is supported. Finally, some brief remarks are provided about each work.

1.2 Motivation

In the works described in Section 1.1, there are no designs that are scalable and unified for all 15 elliptic curves recommended by NIST [18] on the same hardware. Many ECC implementations in literature are highly optimized designs. Many of the architectures presented are flexible, where a different field or key length can be implemented by scaling the proposed architecture accordingly. However, this results in larger designs when implementing the larger

Table 1.2: Summary of ECPs in the Current Literature

Work	Tech.	Abst.	Scalable	Field	Remarks
[22]	ASIC	Multiplier	Yes	Unified	• Montgomery multiplication.
[23]	ASIC	Multiplier	Yes	Unified	• 2 multipliers: precomputation; duual-radix
[24]	FPGA	AU	Yes	Binary	• HSC: Control in software, arithmetic in hardware
[25]	ASIC	Multiplier	Yes	Unified	• 4 parallel radix-2^{16}. multipliers in prime • Radix-2^{64} multiplier in binary.
[26]	N/A	Multiplier	Yes	Unified	• prime number has the form $2^m - 1$. • Irreducible polynomial is an all one polynomial.
[27]	ASIC	ECPM	Yes	Unified	• Microcode-based design. • Energy efficient.
[28]	ASIC	Protocol	Yes	Unified	• Arbitrary prime number and irreducible polynomial. • Uses 64-bit multipliers.
[29]	ASIC & FPGA	ECPM	No	Unified	• Supports RSA and ECC. • Uses signed-digit number representation.
[30]	ASIC & FPGA	ECPM	No	Unified	• Arbitrary elliptic curve and finite field. • 4 32-bit multipliers and 4 64-bit adders.
[31]	ASIC	ECPM	No	Unified	• AHB interface to easily integrate into existing systems. • Improvement from [30]
[32]	ASIC	ECPM	No	Unified	• Energy-adaptive design. • Improves inversion. Improvement from [31]
[33]	ASIC	ECPM	Yes	Unified	• Microcode-based design. • Only binary ECC and prime RSA.
[34]	ASIC	ECPM	No	Unified	• Fast radix-4 unified division
[35]	ASIC & FPGA	ECPM	Yes	Unified	• Power-Analysis Resistant. • Heterogeneous dual-processing-element
[36]	FPGA	ECPM	Yes	Binary	• Word-level algorithm for multiplication and squaring.
[37]	FPGA	ECPM	Yes	Binary	• HSC using PicoBlaze. • Low area design. SEGC curves.
[38]	FPGA	ECPM	Yes	Binary	• HSC using PicoBlaze. • Low area design. • SEGC curves. • Word-level modular reduction
[17]	FPGA	ECPM	Yes	Binary	• HSC using PicoBlaze. • NIST pseudo-random curves
[39]	FPGA	ECPM	Yes	Binary	• HSC using PicoBlaze. • NIST Koblitz curves
[40]	FPGA	ECPM	Yes	Binary	• Dynamic partial reconfiguration. • Up to 283 bits.
[41]	FPGA	ECPM	Yes	Prime	• Up to 256 bits. • Improved modular inversion.
[42]	FPGA	ECPM	Yes	Prime	• NIST prime curves up to 521 bits. • Large inversion module.
[43]	FPGA	ECPM	Yes	Prime	• HSC approach. • Fast reduction algorithm.
[14]	FPGA	ECPM	No	Binary	• 163-bit pseudo-random curves. • Parallel cores
[44]	FPGA	ECPM	No	Binary	• 163, 233 and 283-bit Koblitz curves. • Parallelization of instructions.
[45]	FPGA	ECPM	No	Prime	• Uses DSP blocks. • Shows 224 and 256-bit implementations.
[46]	ASIC & FPGA	ECPM	No	Prime	• Parallelization techniques for affine coordinate ECPM.
[47]	FPGA	ECPM	No	Prime	• RNS to speed up ECPM.
[48]	FPGA	ECPM	No	Prime	• RNS-based design.

key sizes, such as 571 bits. Even though these designs have high speeds, it is not practical to deploy a hardware accelerator that supports different types of curves on the same hardware using these architectures.

This thesis improves on the state-of-the-art implementations of the ECC processors described in Section 1.1 in order to achieve faster computational times, which in turn enhances the users' experience by reducing the wait times for a secure connection. In addition, this thesis will also look to integrate scalable and unified architectures into the design of ECC processors to enhance their capabilities. In this thesis, the point multiplication level and all the levels below it in Figure 1.1 are implemented in hardware.

In server-side applications, scalability, high-throughput and low latency are some of the factors that determine the performance of a cryptosystem processor. As previously mentioned, during the handshaking step in the SSL protocol, ECC scalar multiplications occurs frequently between the client and the server. Every time during the handshaking step, the client and the server must agree on the key length and thus the elliptic curve to be used for the ECDSA and ECDH. As a result, the server must have the ability to support a range of curves, in order to be able to accept requests from different clients requiring different security levels, and it must be able to respond to requests promptly. Due to the long computation times of the scalar multiplication in the software environment, many servers look to offload this computationally intensive operation into a separate hardware environment, called hardware accelerators. Thus, it is important that these servers are complemented by ECC processors that are scalable and have high throughput and low latency.

In the current literature, some authors also present ECC architectures that can support any field size. However, this thesis only investigates curves recommended by NIST [18]. The advantage of selecting only a particular set of curves is that the processor architecture can be more optimized to yield lower latencies and smaller hardware resource utilization, while maintaining its practicality since NIST curves are widely used.

1.3 Description of the Research and Major Contributions

In this thesis, the ECC processor designs are scalable, where the key length can be changed on-the-fly during run-time. In other words, the same hardware design has the ability to support multiple curves. The advantage of these scalable ECC processors is the ability to share the hardware resources for computing the underlying finite field arithmetic operations among different key lengths. In server-side applications, where the support of a wide range of key lengths is important, scalable ECC processors can support various key lengths with the same hardware, whereas the high-speed and optimized implementations in literature require different hardware for different key lengths.

Furthermore, due to the resource sharing, the total hardware utilized to implement all NIST curves in the same hardware is much smaller than implementing a ECC processor for each curve independently. For example, the architecture presented by Sutter et al. [49] uses 6,150, 8,134, 7,069, 10,236, 11,640 slices for 163, 233, 283, 409, 571-bit ECC, respectively, on a Virtex-5 FPGA. If the server-side application is to support all 5 key lengths, the resultant hardware would require 43,229 slices. Furthermore, if the design in [49] were to implement all 5 processors on the same FPGA, the routing delay would increase dramatically as the FPGA would start to fill up, causing the clock frequencies to not be as high as reported.

Thus, it is important to investigate the on-the-fly scalability of the ECC processor implementations in order to be able to support a wider range of key lengths on the same hardware for server-side applications, where both high-speed and scalability, are important performance factors.

The main objectives of this thesis are:

- Research different architectures of finite field arithmetic to be used for implementing scalable ECC processors;
- Research different architectures of ECC processors in both prime and binary fields;
- Implement hardware scalable ECC processors in both prime and binary fields;
- Implement a scalable and unified ECC processor for all curves recommended by NIST.

The foundation of the ECPs is the finite field arithmetic, as shown in Figure 1.1. Thus, the efficiency of the ECC processor is highly dependent on the architecture of the finite field arithmetic units. In this thesis, much of the focus is on improving the efficiency of the ECC processing by modifying the architecture of the finite field arithmetic units.

The major contributions of this thesis are:

- Design of scalable finite field arithmetic blocks for scalable ECPs;
- Efficient parallelization of multiplication and ECPM operations;
- Efficient use of DSP48E slices for the scalable prime field ECP;
- Efficient use of DSP48E slices for the scalable and unified ECP.

In this thesis, the objective of implementing a scalable ECP is accomplished by the design of scalable finite field arithmetic blocks. These blocks implement digit-wise operations in finite fields in order to allow for hardware resource sharing among the different bit lengths. The reduction operations for each finite field is also optimized, when possible, to be implemented in the same hardware. Subsequently, the parallelization of these finite field arithmetic blocks is explored. There are 2 levels of parallelization deployed. Firstly, the algorithm used for multiplication is parallelized to reduce the latency of the operation. Secondly, the operations required for the ECPM (i.e. point addition and point doubling operations) are parallelized by separating the multiplication from the addition, subtraction and reduction. By doing so, the multiplication operation, which requires a high number of clock cycles can be evaluated in parallel with multiple execution of the other instructions to reduce the overall latency.

The implementation of the ECP for prime fields explores the use of DSP48E slices that exist in Xilinx Virtex-5 FPGAs. These DSP48E slices have built-in hardware multipliers and arithmetic logic units (ALU) that can be used to improve the performance of the ECP. The DSP48E slices also have the ability to perform logical operations, such as exclusive-OR (XOR), which is used efficiently in the implementation of the scalable and unified ECP that requires both prime and binary field additions.

In this thesis, a series of ECC processors have been designed and implemented targeting FPGA platform. Some of the designs described in this thesis have been published in [50, 51, 52, 53, 54]. As previously mentioned, main objective of the thesis is to implement a scalable and unified ECC processor with a high throughput that is suitable for server-side security applications.

1.4 Organization of Thesis

The subsequent chapters are organized as follows: Chapter 2 discusses background information about finite field arithmetic, elliptic curve cryptography and provides an overview of the Xilinx Virtex-5 FPGA used in the thesis; Chapter 3 presents the design and architecture of scalable ECPs for binary fields; Chapter 4 presents the design and architecture of a scalable ECP for prime fields; Chapter 5 presents the design and architecture of a scalable and unified ECP for both binary and prime fields; Chapter 6 provides a conclusion to the thesis. Chapter 7 proposes potential future work for this thesis.

Chapter 2
Background

This chapter describes some of the background theory and information related to cryptography and Elliptic Curve Cryptography (ECC). Much of the information presented in this chapter is described in [55]. Following the levels of abstraction shown in Figure 1.1, Section 2.1 presents an introduction to cryptography and some protocols used in ECC, Section 2.2 presents information specific to ECC and Section 2.3 reviews some of the finite field arithmetic operations that will be used in the designs of this thesis. In Section 2.4, an overview of Xilinx Virtex-5 FPGAs is provided and the architecture of the Xilinx XtremeDSP slices is described.

2.1 Introduction to Cryptography

Consider the simple communication model presented in Figure 2.1, where Alice and Bob send messages to each other through a channel. In the case of an unsecured channel, an eavesdropper, Eve, can very easily monitor the communication channel and have access to all the interaction between Alice and Bob. Consider the case of an online banking system, where Alice is a bank client logging into her computer and Bob is the bank's server. If the information is transferred through the unsecured channel, a malicious party, Eve, can very

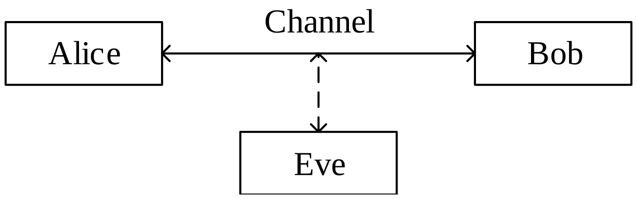


Figure 2.1: Simple communication model of an unsecured channel.

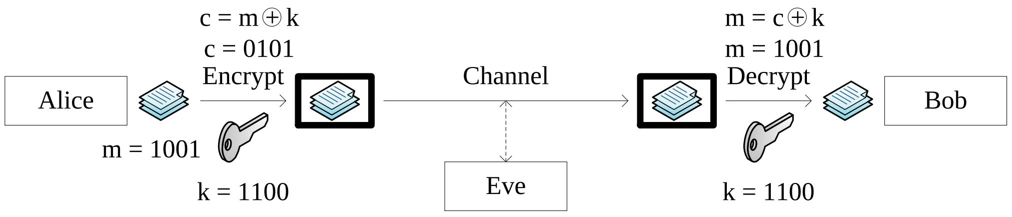


Figure 2.2: Communication model with symmetric-key cryptography.

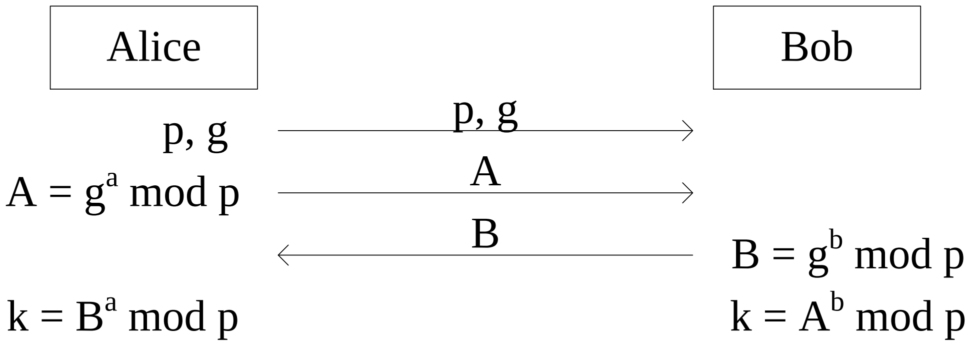


Figure 2.3: Diffie-Hellman key exchange.

easily obtain Alice's online banking information.

One way to secure the information is by using symmetric-key cryptography. Figure 2.2 shows an example of symmetric-key cryptography, where Alice sends the binary message, $m = 1001$, to Bob. First, she encrypts the message by performing an exclusive OR (XOR) operation with the binary secret shared key, $k = 1100$, to generate the binary cipher, $c = 0101$. The encrypted cipher is sent to Bob through the channel and Bob decrypts the message by once again performing an XOR on the cipher with the secret shared key to retrieve the original message. By doing so, no one can retrieve the original message unless he/she is in possession of the key, k. However, in the situation described above, the problem still remains in how Alice and Bob can share the secret key securely, without anyone else knowing.

Public-key cryptography provides a solution for solving the problem of exchanging keys securely. In 1976, Diffie and Hellman [56] proposed a scheme to share keys between two parties, called the Diffie-Hellman key exchange, shown in Figure 2.3. To begin the exchange, Alice selects a large prime number, p, and a large base integer, g. She sends these values to Bob through an unsecured channel. Alice also selects another large integer, a, evaluates $A = g^a \mod p$ and transmits the value of A to Bob. Simultaneously, after Bob receives the values of p and g, he selects a large integer, b, evaluates $B = g^b \mod p$ and transmits B to

Alice. Finally, Alice computes $k = B^a \mod p$ and Bob computes $k = A^b \mod p$ to obtain the secret shared key. The final k value is common for both Alice and Bob because $k = B^a \mod p = (g^b)^a \mod p = (g^a)^b \mod p = A^b \mod p$. Thus, k can be used as the shared secret key to establish a secured communication channel using symmetric-key cryptography between Alice and Bob.

In the Diffie-Hellman key exchange, only the values p, g, A and B are transmitted on the unsecured channel, Eve cannot compute the value of k, without knowing a or b. Furthermore, computing the value of a from A, p, and g is extremely difficult and is usually referred to as the discrete logarithm problem. In the above described scheme, a and b are referred to as the private keys, which are never shared to the public, and A and B are referred to as public keys.

2.2 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is a type of public-key cryptography that is based on a set of operations on elliptic curves. The description of the background in ECC in this section is based on the information in [55].

As previously mentioned, ECC operations recommended by NIST can be subdivided into prime fields, $GF(p)$, or binary fields, $GF(2^m)$. It changes the underlying finite field operations, but it also changes the elliptic curves that the operations are defined over.

The equation of the elliptic curve can be given by the Weierstrass equation. The simplified version of the Weierstrass equation for binary fields is:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{2.1}$$

where a and b are elements of the binary field, $GF(2^m)$. For prime fields, the Weierstrass equation becomes:

$$E_p : y^2 = x^3 + ax + b \tag{2.2}$$

where a and b are elements of the prime field, $GF(p)$. In NIST, these equations are further simplified by setting one of the coefficients as a constant. Thus, for binary pseudo-random

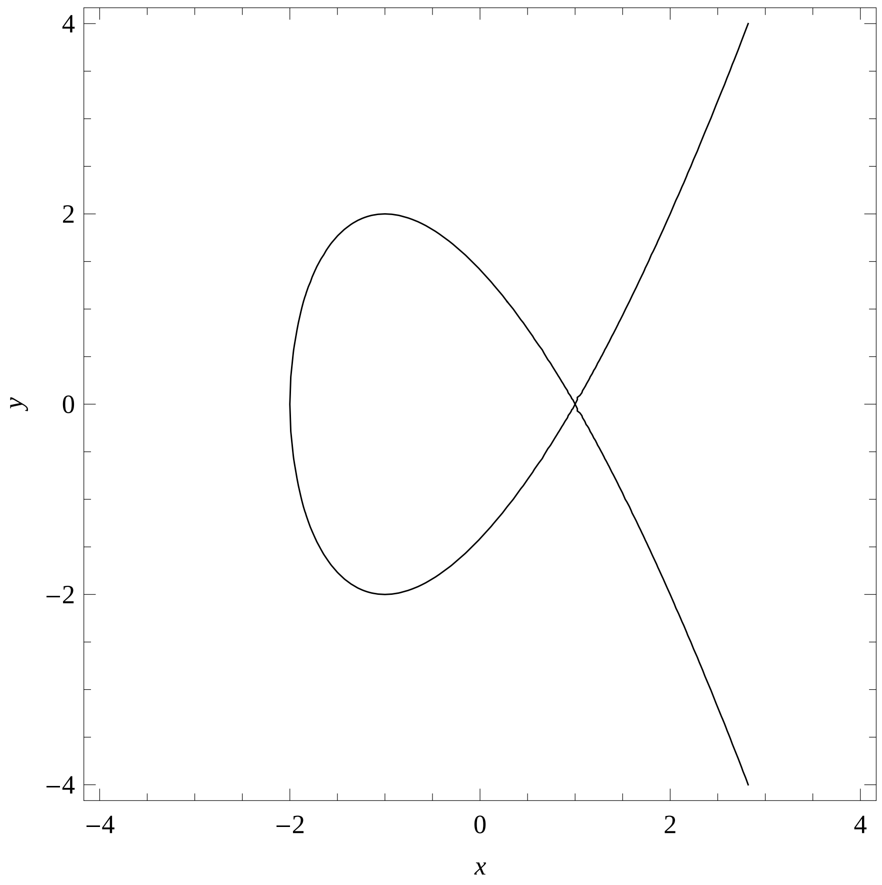


Figure 2.4: Example of an elliptic curve ($y^2 = x^3 - 3x + 2$).

curves, the equation is:

$$E : y^2 + xy = x^3 + x^2 + b \tag{2.3}$$

where b is a constant depending on the curve. For binary Koblitz curves, the equation is:

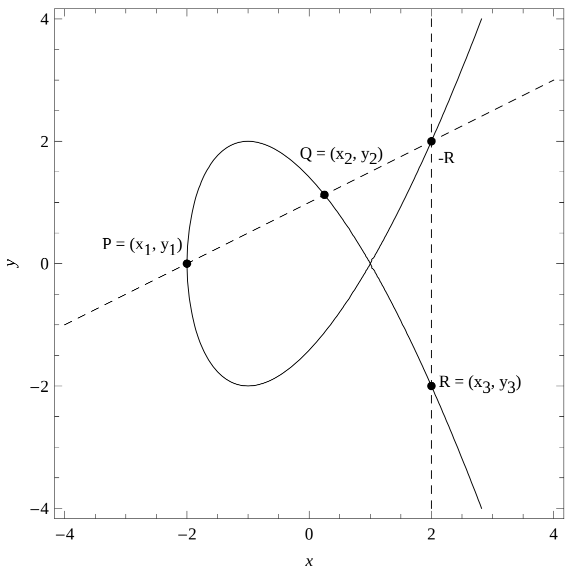
$$E_a : y^2 + xy = x^3 + ax^2 + 1 \tag{2.4}$$

where $a = 0$ or 1. For prime curves, the equation is:

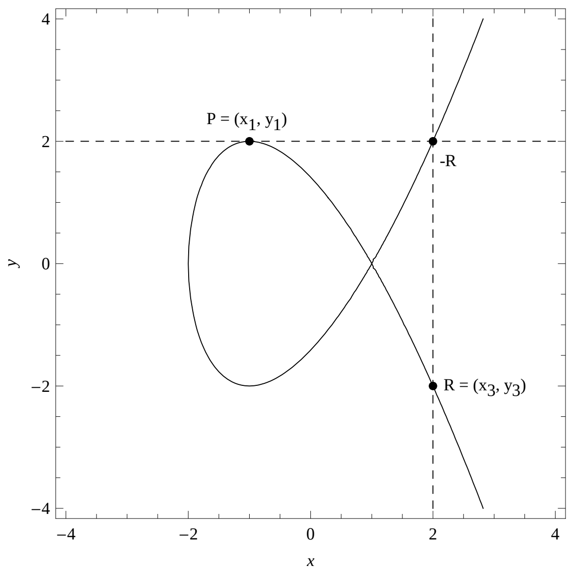
$$E_p : y^2 = x^3 - 3x + b \tag{2.5}$$

where b is also a constant depending on the curve. This thesis focuses only on the curves that are recommended by NIST [18]. Thus, the remainder of this thesis describes only operations related to the above mentioned NIST curves.

ECC operations can be understood by using geometry on the plot of the curve. Consider the elliptic curve in Figure 2.4, with equation $y^2 = x^3 - 3x + 2$. Two operations can be defined



(a) Example of point addition on an elliptic curve ($y^2 = x^3 - 3x + 2$).



(b) Example of point doubling on an elliptic curve ($y^2 = x^3 - 3x + 2$).

Figure 2.5: Example of point operations on an elliptic curve.

on the curve, namely point addition (PADD) and point doubling (PDBL). The graphical interpretation of the point addition is shown in Figure 2.5(a). Given two points, $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, where $P \neq \pm Q$, the addition of the two points, $R = (x_3, y_3) = P + Q$, is given by drawing a straight line between P and Q and extending the line until it intersects the curve on a third point, $-R$, and then negating the point. The negative of a point, (x, y), in binary fields is $(x, x + y)$ and $(x, -y)$ in prime fields.

Figure 2.5(b) shows the point doubling operation. Given a point, P, where $P \neq -P$, the double of the point, $R = 2P$, is given by drawing a line tangent to the elliptic curve at point, P, extending the line until it intersects with the curve at a second point, $-R$, and then negating the point.

Point addition and point doubling can also be represented mathematically, but the expressions differ for binary fields and prime fields. In binary fields, point addition is defined as:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{and} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \tag{2.6}$$

where $\lambda = \frac{y_1+y_2}{x_1+x_2}$, and point doubling is defined as:

$$x_3 = x_1^2 + \frac{b}{x_1^2} \quad \text{and} \quad y_3 = x_1^2 + \lambda x_3 + x_3 \tag{2.7}$$

where $\lambda = x_1 + \frac{y_1}{x_1}$. In prime fields, point addition is defined as:

$$x_3 = \lambda^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1 \tag{2.8}$$

where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ and point doubling is defined as:

$$x_3 = \lambda^2 - 2x_1 \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1 \tag{2.9}$$

where $\lambda = \frac{3x_1^2 + a}{2y_1}$.

Using the point addition and doubling operations, the scalar multiplication, or point multiplication, (ECPM) operation is given by:

$$Q = kP = \underbrace{P + P + \cdots + P}_{k \text{ times}} \tag{2.10}$$

where P is a point on the elliptic curve, and k is a scalar integer value. The resultant point, Q, will also be on the elliptic curve.

The trivial way of computing the scalar multiplication is the double-and-add operation, where a sequence of point doubling and point addition operations are evaluated based on the binary representation of k. Algorithm 2.1, which is modified from Algorithm 3.27 in [55], shows the left-to-right version of the double-and-add algorithm. $Q \leftarrow 2Q$ is the point doubling operation and $Q \leftarrow Q + P$ is the point addition operation.

Algorithm 2.1 Left-to-right point multiplication

```
Input: $k = (k_{t-1}, \ldots, k_1, k_0)$, $P$ – a point on $E_a$
Output: $Q = kP$
  $Q \leftarrow \infty$
  for $i$ from $t-1$ down to 0 do
    $Q \leftarrow 2Q$
    if $k_i = 1$ then
      $Q \leftarrow Q + P$
    end if
  end for
  return $Q$
```

As shown in Algorithm 2.1, the point doubling operation is performed in every iteration of the **for** loop and point addition is performed for every non-zero bit of k. Thus, the efficiency of the ECPM operation depends highly on the implementation of Equations (2.6),(2.7),(2.8), and (2.9). Given that finite field division or inversion are the most complex operations among finite field arithmetic operations, the implementation of PADD and PDBL operations in affine coordinates, where a point is represented by (x, y), is not very efficient and results in hardware implementations that either have long latencies or require a large area. Thus, the use of projective coordinates are proposed to simplify the PADD and PDBL operations. In this thesis, the latency of the processor is defined by the time of the completion of an ECPM operation.

In binary fields, the 2 most commonly used projective coordinates are the Lopez-Dahab (LD) projective coordinates and the standard projective coordinates. The Lopez-Dahab projective coordinates represent a point using 3 coordinates, (X, Y, Z). The conversion from LD coordinates back to affine coordinates is given by $(x, y) = (X/Z, Y/Z^2)$. Generally, when using LD coordinates, PADD uses a mixed coordinate addition, where a point in LD coordinates, (X_1, Y_1, Z_1) is added to a point in affine coordinates, (X_2, Y_2), to result in a point in LD coordinates, (X_3, Y_3, Z_3), and PADD becomes as follows:

$$\begin{aligned} Z_3 &= (Z_1(X_2Z_1 + X_1))^2 \\ X_3 &= (Y_2Z_1^2 + Y_1)^2 + (X_2Z_1 + X_1)^2(Z_1(X_2Z_1 + X_1) + aZ_1)^2 \\ &\quad +(Y_2Z_1^2 + Y_1)(Z_1(X_2Z_1 + X_1)) \\ Y_3 &= ((Y_2Z_1^2 + Y_1)(Z_1(X_2Z_1 + X_1)) + Z_3)(X_3 + X_2Z_3) + (X_2 + Y_2)Z_3^2 \end{aligned} \tag{2.11}$$

and the PDBL of a point in LD coordinates, (X_1, Y_1, Z_1), to result in a point in LD coordinates, (X_3, Y_3, Z_3), is given by:

$$\begin{aligned} Z_3 &= X_1^2 \cdot Z_1^2 \\ X_3 &= X_1^4 + b \cdot Z_1^4 \\ Y_3 &= bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4) \end{aligned} \tag{2.12}$$

Notice that in Equations (2.11) and (2.12), there are no longer any finite field divisions, at the expense of more multiplications. As long as the complexity of the division outweighs that of the several multiplication, performing ECPM in projective coordinates is more efficient than in affine coordinates.

The standard projective coordinates also represent a point with 3 coordinates, (X, Y, Z), where the conversion back to affine coordinate is given by $(x, y) = (X/Z, Y/Z)$. Most commonly, these coordinates are used in combination with the Lopez-Dahab (LD) algorithm [57] given in Algorithm 2.2. The advantage of using the LD algorithm is that only the X and Z coordinates are needed in the main loop. The x and y affine coordinates are recovered in the conversion step.

Algorithm 2.2 Lopez-Dahab algorithm

```
Input: $k = (k_{t-1}, \ldots, k_1, k_0)$ with $k_{t-1} = 1$, $P(x, y)$, $b$ – curve specific coefficient
Output: $Q(x_0, y_0) = kP$
  // Initialization - Affine to Projective Conversion
  // and processing $k_{t-1} = 1$
  $(X_1, Z_1) \leftarrow (x, 1)$, $(X_2, Z_2) \leftarrow (x^4 + b, x^2)$
  // Main Loop
  for $i$ from $t - 2$ down to 0 do
    if $k_i = 1$ then
      $(X_1, Z_1) \leftarrow Madd(X_1, X_2, Z_1, Z_2, x)$
      $(X_2, Z_2) \leftarrow Mdouble(X_2, Z_2, b)$
    else
      $(X_2, Z_2) \leftarrow Madd(X_1, X_2, Z_1, Z_2, x)$
      $(X_1, Z_1) \leftarrow Mdouble(X_1, Z_1, b)$
    end if
  end for
  // Mxy - Projective to Affine Conversion
  $x_0 \leftarrow \frac{X_1}{Z_1}$
  $y_0 \leftarrow \frac{1}{x}(x + \frac{X_1}{Z_1})[(x + \frac{X_1}{Z_1})(x + \frac{X_2}{Z_2}) + x^2 + y] + y$
  return $Q(x_0, y_0)$
```

In Algorithm 2.2, *Madd* is defined as:

$$
\begin{aligned}
&(X, Z) \leftarrow Madd(X_1, X_2, Z_1, Z_2, x) \\
&\{ \\
&\quad X \leftarrow X_1X_2Z_1Z_2 + x(X_1Z_2 + X_2Z_1)^2 \\
&\quad Z \leftarrow (X_1Z_2 + X_2Z_1)^2 \\
&\}
\end{aligned}
\tag{2.13}
$$

and *Mdouble* is defined as:

$$
\begin{aligned}
&(X, Z) \leftarrow Mdouble(X_1, Z_1, b) \\
&\{ \\
&\quad X \leftarrow X_1^4 + bZ_1^4 \\
&\quad Z \leftarrow X_1^2Z_1^2 \\
&\}
\end{aligned}
\tag{2.14}
$$

Once again, Equations (2.13) and (2.14) show that no finite field division is required when using the LD algorithm at the expense of more multiplications.

When Koblitz curves recommended by NIST [18] are used, the ECPM algorithm can be further optimized by using the τ-adic non-adjacent form (τNAF), which rewrites k into the form $k = \sum_{i=0}^{l-1} u_i \tau^i$, where $u_i \in \{0, \pm 1\}$ and $u_{l-1} \neq 0$, $\tau = \frac{\mu+\sqrt{-7}}{2}$, $\mu = (-1)^{1-a}$, $a = 0$ or 1, and l is the bit length of k in τ-adic form. Using LD coordinates and τNAF representation of k, Algorithm 2.1 is modified and shown in Algorithm 2.3, which is modified from Algorithm 3.66 in [55]. The major differences between Algorithm 2.1 and Algorithm 2.3 are that the latter performs $Q \leftarrow \tau Q$ for PDBL, which is simply a finite field squaring on each coordinate, and the need to add or subtract a point in PADD, which can be done efficiently because the negative of a point with affine coordinates (x, y) is given by $-(x, y) = (x, x + y)$. The operation $Q \leftarrow \tau Q$ is called Frobenius endomorphism (PFRB). The conversion of k from binary to τNAF is out of the scope of this thesis. More information can be found in [58] and in Section 7.1 where future work is discussed.

In prime fields, one of the most efficient projective coordinates used is the Jacobian projective coordinates. It also uses 3 coordinates, (X, Y, Z) and the conversion to affine coordinates is given by $(x, y) = (X/Z^2, Y/Z^3)$. Using Jacobian coordinates, mixed Jacobian-affine coordinates PADD becomes:

$$\begin{aligned} X_3 &= (Y_2Z_1^3 - Y_1)^2 - (X_2Z_1^2 - X_1)^2(X_1 + X_2Z_1^2) \\ Y_3 &= (Y_2Z_1^3 - Y_1)(X_1(X_2Z_1^2 - X_1)^2 - X_3) - Y_1(X_2Z_1^2 - X_1)^3 \\ Z_3 &= (X_2Z_1^2 - X_1)Z_1 \end{aligned} \tag{2.15}$$

and PDBL becomes:

$$\begin{aligned} X_3 &= (3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2 \\ Y_3 &= (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 &= 2Y_1Z_1 \end{aligned} \tag{2.16}$$

In this thesis, the ECPM in elliptic curves is performed by using the double-and-add algorithm and the Equations (2.15) and (2.16), where $a = -3$ as recommended by NIST [18].

Using the ECPM operation, the Diffie-Hellman key exchange scheme can be modified to use ECC. The scheme is usually referred to as Elliptic Curve Diffie-Hellman (ECDH)

Algorithm 2.3 τNAF point multiplication on Koblitz Curves

```
Input: $k$ – a binary integer, $P(x, y)$ – a point on $E_a$
Output: $Q = kP$
  Compute $\tau\text{NAF}(k) = \sum_{i=0}^{l-1} u_i\tau^i$
  // Perform the first point addition of $Q \leftarrow \infty \pm P$
  if $u_{l-1} = 1$ then
    $Q(X_3, Y_3, Z_3) \leftarrow P(x, y)$
  else
    $Q(X_3, Y_3, Z_3) \leftarrow P(x, x + y)$
  end if
  for $i$ from $l - 2$ down to 0 do // Main loop
    // Perform PFRB ($Q \leftarrow \tau Q$)
    $Q(X_3, Y_3, Z_3) \leftarrow Q(X_3^2, Y_3^2, Z_3^2)$
    // Perform PADD
    if $u_i = 1$ then
      $Q(X_3, Y_3, Z_3) \leftarrow Q(X_3, Y_3, Z_3) + P(x, y)$
    end if
    if $u_i = -1$ then
      $Q(X_3, Y_3, Z_3) \leftarrow Q(X_3, Y_3, Z_3) + P(x, x + y)$
    end if
  end for
  return $Q(x_3, y_3) \leftarrow Q(X_3/Z_3, Y_3/Z_3^2)$
```

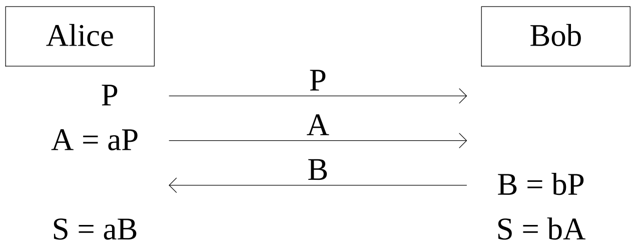


Figure 2.6: Communication model with Elliptic Curve Diffie-Hellman (ECDH).

and is shown in Figure 2.6. In ECDH, Alice initiates communication with Bob by selecting a base point, P, and transmits the coordinates of A to Bob. Alice also selects a large integer, a, and evaluates ECPM for $A = aP$. Simultaneously, Bob selects a large integer, b, evaluates $B = bP$ and transmits the coordinates of B to Alice. Finally, Alice and Bob compute $S = aB$ and $S = bA$ to arrive at the shared secret, S, which is equivalent because $S = aB = a(bP) = b(aP) = bA$.

In Transport Layer Security (TLS) or Secure Socket Layer (SSL) protocol, ECDH and

Elliptic Curve Digital Signature Algorithm (ECDSA) can be used during the handshaking step for key exchange. Thus, the ECPM operation is crucial in the efficiency of the operation in secure communication over the network.

2.3 Finite Field Arithmetic

Galois fields (GF), or finite fields, are a number set with a finite number of elements. Operations on one or more elements, result in another element in the field. In this thesis, binary fields refer to $GF(2^m)$ and prime fields refer to $GF(p)$.

Finite field arithmetics are the operations that can be performed on the finite field elements. Generally, the operations that need to be implemented are finite field addition/subtraction, finite field squaring, finite field multiplication and finite field division or finite field inversion.

This section is divided into 2 subsections to discuss finite field arithmetic in binary fields and in prime fields separately.

2.3.1 Finite Field Arithmetic in Binary Fields

In binary fields, an element can be represented in polynomial basis or normal basis. In polynomial basis representation, an element in the finite field is represented by a bit string, $(a_{m-1}, \cdots, a_2, a_1, a_0)$, which correspond to the polynomial [18]:

$$a_{m-1}t^{m-1} + \cdots + a_2t^2 + a_1t^1 + a_0 \tag{2.17}$$

The field arithmetic is evaluated as polynomial arithmetic modulo $P(t)$, where $P(t)$ is an irreducible polynomial of order m, called the field polynomial. In normal basis representation, the finite field element is represented by a bit string, $(a_0, a_1, a_2 \cdots, a_{m-1})$, which is defined as follows [18], given an element θ:

$$a_0\theta + a_1\theta^2 + a_2\theta^{2^2} + \cdots + a_{m-1}\theta^{2^{m-1}} \tag{2.18}$$

The advantage of normal basis representation is the simplicity of the square operation, which is simply a cyclic right shift. However, the multiplication operation becomes much more complex compared to the polynomial basis representation. This thesis focuses mainly on the polynomial representation of binary finite field elements.

The remainder of this section describes some algorithms to evaluate finite field arithmetic in binary polynomial representation. Among the finite field operations, addition is the most trivial because it can be evaluated by simply using an XOR operation between the operands. Finite field squaring without the reduction step can also be very easily implemented by using the following property:

$$A(t)^2 = a_{m-1}t^{2m-2} + \cdots + a_1t^2 + a_0 \mod P(t) \tag{2.19}$$

which is simply interleaving 0 bits and operand bits. The reduction step refers to the mod $P(t)$ operation in Equation (2.19).

The next most complex finite field operation is multiplication. In general, hardware multiplication implementations can be divided into 3 categories: bit-serial, bit-parallel, digit-serial. Bit-serial implementations consume the least amount of hardware resources, but requires m clock cycles per multiplication [59], where m is the bit length. An example of a bit-serial implementation is the shift-and-add algorithm, where the multiplicand bits are shifted every clock cycle and accumulated if the multiplier bit is non-zero. Bit-parallel implementations require only 1 clock cycle but generally require more hardware resources and are more difficult to make them support multiple fields simultaneously [60]. The Karatsuba-Ofman multiplier is an example of bit-parallel multiplier. Digit-serial implementations are a compromise between the bit-serial and bit-parallel implementations [61]. Digit-serial implementations allow for the multiplier to support fields of different bit lengths by simply processing different number of digits, so it is very suitable for a scalable design.

In this thesis, the Comba and the Karatsuba-Ofman algorithms for multiplication will be described with some details as these algorithms are used in the following chapters.

Digit-serial finite field multiplication can be defined as follows:

$$\begin{aligned} Z(t) &= [A(t) \times B(t)] \mod P(t) \\ &= \left[\sum_{i=0}^{s-1} a_i t^{iw} \times \sum_{j=0}^{s-1} b_j t^{jw}\right] \mod P(t) \\ &= \left[\sum_{i=0}^{s-1}\sum_{j=0}^{s-1} a_i b_j t^{(i+j)w}\right] \mod P(t) \\ &= C(t) \mod P(t) \end{aligned} \tag{2.20}$$

where $s = \lceil m/w \rceil$ and w is the number of bits in a digit.

The Comba algorithm, as shown in Algorithm 2.4 (Modified from Algorithm 2 in [62]), is a digit-wise multiplication algorithm and it processes the operands digit-per-digit. As shown in Algorithm 2.4, the Comba algorithm is divided into 2 **for** loops: the least-significant digits (LSDs) loops and the most-significant digits (MSDs) loops. In each inner loop, the digits are multiplied together and accumulated. The main features of the Comba algorithm are that the result is produced LSD-first, and once each iteration of the outer loop is completed, the computation of the i^{th} digit is complete and can be outputted.

Algorithm 2.4 Comba Multiplication

```
Input: $A = (A_{s-1}, \ldots, A_1, A_0)$ and
  $B = (B_{s-1}, \ldots, B_1, B_0)$
Output: $Z = A \cdot B = (Z_{2s-1}, \ldots, Z_1, Z_0)$
  $(U, V) \leftarrow (0, 0)$
  for $i$ from 0 to $s-1$ do // LSDs outer loop
    for $j$ from 0 to $s-1$ do // LSDs inner loop
      $(U, V) \leftarrow (U, V) + A_j \cdot B_{i-j}$
    end for
    $Z_i \leftarrow V$
    $V \leftarrow U$, $U \leftarrow 0$
  end for
  for $i$ from $s$ to $2s-2$ do // MSDs outer loop
    for $j$ from $i-s+1$ to $s-1$ do // MSDs inner loop
      $(U, V) \leftarrow (U, V) + A_j \cdot B_{i-j}$
    end for
    $Z_i \leftarrow V$
    $V \leftarrow U$, $U \leftarrow 0$
  end for
  $Z_{2s-1} \leftarrow V$
  return $Z = (Z_{2s-1}, \ldots, Z_1, Z_0)$
```

The Karatsuba-Ofman multiplication algorithm is a bit-parallel algorithm and it uses the divide-and-conquer method to reduce the operand sizes of the multiplication. The Karatsuba-Ofman algorithm can be defined as follows [55]:

$$\begin{aligned} A(t) \cdot B(t) &= (A_1 t^l + A_0) \cdot (B_1 t^l + B_0) \\ &= \alpha t^{2l} + [\beta + \alpha + \gamma] t^l + \gamma \end{aligned} \tag{2.21}$$

where

$$\alpha = A_1 \cdot B_1, \quad \beta = (A_1 + A_0) \cdot (B_1 + B_0), \quad \gamma = A_0 \cdot B_0,$$

$A(t) = A_1 t^l + A_0$, $B(t) = B_1 t^l + B_0$, $l = \lceil n/2 \rceil$, n is the degree of $A(t)$ and $B(t)$, and A_1, A_0, B_1, B_0 are binary polynomials in t of degree less than l. As a result, a multiplication of 2 n-bit operands is reduced to 3 multiplications of $\lceil n/2 \rceil$-bit operands. The Karatsuba-Ofman algorithm can also be modified to split the operands into 3 operands instead of 2. By doing so, the algorithm is defined as follows [55]:

$$\begin{aligned} A(t) \cdot B(t) &= (A_2 t^{2r} + A_1 t^r + A_0) \cdot \\ &\quad (B_2 t^{2r} + B_1 t^r + B_0) \\ &= \alpha_2 t^{4r} + [\beta_2 + \alpha_2 + \alpha_1] t^{3r} \\ &\quad + [\beta_1 + \alpha_2 + \alpha_1 + \alpha_0] t^{2r} \\ &\quad + [\beta_0 + \alpha_1 + \alpha_0] t^r + \alpha_0 \end{aligned} \tag{2.22}$$

where

$$\begin{aligned} \alpha_2 &= A_2 \cdot B_2, \quad \beta_2 = (A_2 + A_1) \cdot (B_2 + B_1), \\ \alpha_1 &= A_1 \cdot B_1, \quad \beta_1 = (A_2 + A_0) \cdot (B_2 + B_0), \\ \alpha_0 &= A_0 \cdot B_0, \quad \beta_0 = (A_1 + A_0) \cdot (B_1 + B_0), \end{aligned}$$

$A(t) = A_2 t^{2r} + A_1 t^r + A_0$, $B(t) = B_2 t^{2r} + B_1 t^r + B_0$, $r = \lceil n/3 \rceil$, n is the degree of $A(t)$ and $B(t)$, and $A_2, A_1, A_0, A_2, B_1, B_0$ are binary polynomials in t of degree less than r. By separating the operands into 3 parts, a multiplication of 2 n-bit operands is reduced to 6 multiplications of $\lceil n/3 \rceil$-bit operands. These two methods of the Karatsuba-Ofman multiplication can be applied recursively to reduce the complexity of the multiplication.

The modulo $P(t)$ operation, required by both finite field multiplication and squaring, is called reduction. $P(t)$ is an irreducible polynomial chosen for each specific curve and it is shown in [18]. The five irreducible polynomials recommended by NIST [18] are shown

Table 2.1: NIST Recommended Irreducible Polynomials

m	Irreducible Polynomial, $P(t)$
163	$t^{163}+t^7+t^6+t^3+1$
233	$t^{233}+t^{74}+1$
283	$t^{283}+t^{12}+t^7+t^5+1$
409	$t^{409}+t^{87}+1$
571	$t^{571}+t^{10}+t^5+t^2+1$

in Table 2.1. The same polynomial is used for both pseudo-random and Koblitz curves. When performing a m-bit polynomial multiplication or squaring as described above, the result is $2m-1$ bits wide, which falls outside of the finite range of elements in the finite field. Thus, the reduction operation is performed to reduce the result back to the finite range. One of the methods to evaluate the reduction step is shown in Algorithm 2.41 – 2.45 in [55]. Algorithm A.1 to Algorithm A.5 in Appendix A show these algorithms for the 5 irreducible polynomials recommended by NIST [18]. The algorithms in [55] modify the reduction operation into a series of binary field additions, where the operands are shifted digits of the product.

Consider the example of 163-bit NIST recommended binary field, where $P(t)=t^{163}+t^7+t^6+t^3+1$. Given that $a(t)$ and $b(t)$ are polynomials of degree 162, their product yields a polynomial of up to degree 324 (i.e. $c(t)=a(t)\times b(t)=c_{324}t^{324}+c_{323}t^{323}+\cdots+c_1t^1+c_0$). Given that $t^{163}+t^7+t^6+t^3+1 \pmod{P(t)}=0 \Rightarrow t^{163}=t^7+t^6+t^3+1 \pmod{P(t)}$, so the following equalities can be derived:

$$\begin{array}{lcllllllll}
c_{324}t^{324} & = & c_{324}t^{168} & + & c_{324}t^{167} & + & c_{324}t^{164} & + & c_{324}t^{161} & \pmod{P(t)} \\
c_{323}t^{323} & = & c_{323}t^{167} & + & c_{323}t^{166} & + & c_{323}t^{163} & + & c_{323}t^{160} & \pmod{P(t)} \\
 & \vdots & & & & & & & & \\
c_{165}t^{165} & = & c_{165}t^{9} & + & c_{165}t^{8} & + & c_{165}t^{5} & + & c_{165}t^{2} & \pmod{P(t)} \\
c_{164}t^{164} & = & c_{164}t^{8} & + & c_{164}t^{7} & + & c_{164}t^{4} & + & c_{164}t & \pmod{P(t)} \\
c_{163}t^{163} & = & c_{163}t^{7} & + & c_{163}t^{6} & + & c_{163}t^{3} & + & c_{163} & \pmod{P(t)}
\end{array} \tag{2.23}$$

Considering each column in Equation (2.23) as a polynomial, one can see that each column is a shifted version of $c_{324}t^{161}+c_{323}t^{160}+\cdots+c_{164}t+c_{163}$. Figure 2.7 shows the same example

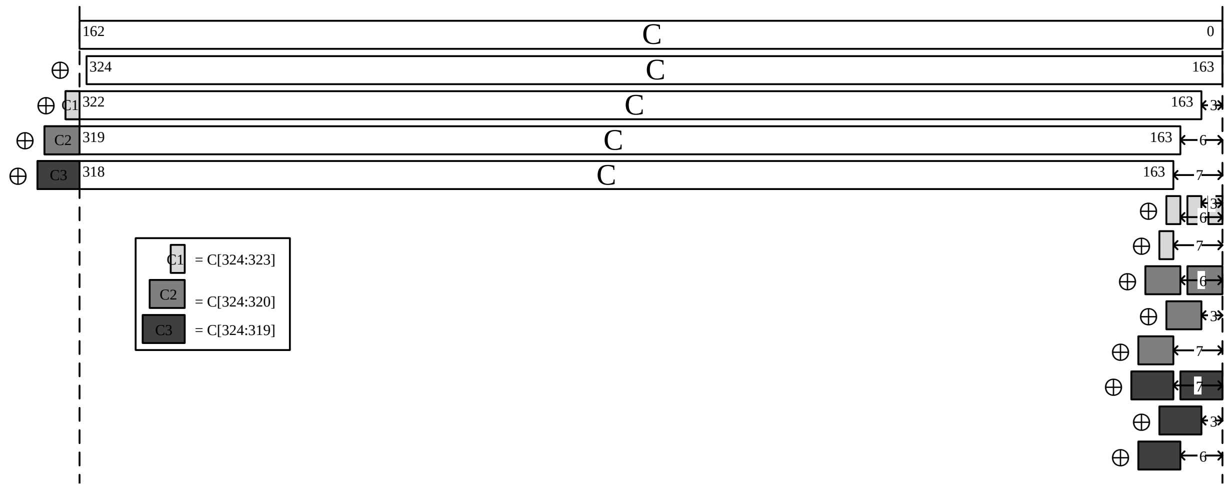


Figure 2.7: Example of a polynomial of degree 324 reduced by $P(t) = t^{163} + t^7 + t^6 + t^3 + 1$

in a diagram. When shifting $c_{324}t^{161} + c_{323}t^{160} + \cdots + c_{164}t + c_{163}$ by 3, it yields the terms $c_{323}t^{163}$ and $c_{324}t^{164}$, which has a degree greater than 162, so it must be reduced again, as shown by block 'C1' in Figure 2.7. Similarly, blocks 'C2' and 'C3' are further reduced when $c_{324}t^{161} + c_{323}t^{160} + \cdots + c_{164}t + c_{163}$ is shifted by 6 and 7, respectively.

Another method of performing the reduction operation is by using a reduction matrix for each finite field, such that the reduction operation is defined as follows:

$$D(t) = R \times C(t) \tag{2.24}$$

where $C(t)$ is a binary column matrix of the coefficients of the polynomial to be reduced, $(c_{2m-2}, \ldots, c_1, c_0)$, R is the $m \times 2m - 1$ reduction matrix and $D(t)$ is the reduced column matrix, $(d_{m-1}, \ldots, d_1, d_0)$. The multiplication and addition operations in the matrix multiplication are performed in $GF(2)$. The reduction matrices are generated by taking an identity matrix of size $2m - 1$, and repeatedly eliminating any non-zero elements on the upper half of the matrix, using the irreducible polynomial, $P(t)$. The resultant reduction matrix indicates the coefficients in the polynomial to be reduced, $C(t)$, that needs to be added together in $GF(2)$, to produce each coefficient of $D(t)$.

As an example, consider the irreducible polynomial, $P(t) = t^5 + t^3 + t^2 + t + 1$, where $m = 5$. To generate the R matrix, begin with an identity matrix of size $2m - 1 = 9$ and perform the following steps:

$$\begin{tikzcd}
\left[\begin{array}{c}100000000\\010000000\\001000000\\000100000\\\hline 000010000\\000001000\\000000100\\000000010\\000000001\end{array}\right] \arrow[r] & \left[\begin{array}{c}100000000\\010000000\\001000000\\000000000\\\hline 000010000\\000101000\\000100100\\000100010\\000100001\end{array}\right] \arrow[r] & \left[\begin{array}{c}100000000\\010000000\\000000000\\000000000\\\hline 001010000\\001101000\\001100100\\001100010\\000100001\end{array}\right] \arrow[r] & \left[\begin{array}{c}100000000\\000000000\\000000000\\010000000\\\hline 011010000\\011101000\\011100100\\001100010\\000100001\end{array}\right] \arrow[r] & \left[\begin{array}{c}100000000\\000000000\\000000000\\000000000\\\hline 011010000\\001101000\\001100100\\011100010\\010100001\end{array}\right] \arrow[d] \\
 & R = \left[\begin{array}{c}011010000\\101101000\\001100100\\011100010\\110100001\end{array}\right] & \left[\begin{array}{c}000000000\\000000000\\000000000\\000000000\\\hline 011010000\\101101000\\001100100\\011100010\\110100001\end{array}\right] \arrow[l] & \left[\begin{array}{c}000000000\\000000000\\100000000\\000000000\\\hline 111010000\\001101000\\101100100\\111100010\\110100001\end{array}\right] \arrow[l] & \left[\begin{array}{c}000000000\\000000000\\100000000\\100000000\\\hline 111010000\\101101000\\001100100\\011100010\\010100001\end{array}\right] \arrow[l]
\end{tikzcd} \tag{2.25}$$

In each step, a non-zero element above the horizontal line is replaced by the modulo 2 addition of the column below the horizontal line and a shifted version of the column matrix $[\ 0 \quad 1 \quad 1 \quad 1 \quad 1\ ]^T$. The process is repeated until there are no more non-zero elements above the horizontal line and the R matrix is the $m \times 2m-1$ matrix below the horizontal line. Using the reduction matrix, the polynomial $D(t)$ can be represented by a linear combination of the coefficients of $C(t)$. Thus, reduction can be evaluated by a series of binary field additions (i.e. XOR operations).

Finally, the most complex finite field arithmetic operation discussed in this thesis is the finite field division. Alternatively, finite field inversion can also be used instead of division. Inversion is defined as an element $b = a^{-1} \mod P(t)$, where $a\dot{b} = 1 \mod P(t)$. One method of evaluating finite field inversion is by using the Itoh-Tsujii algorithm [63]. The Itoh-Tsujii algorithm is based on the property that $a^{-1} = a^{2^m-2} \pmod{P(t)}$ in binary fields. Defining

Table 2.2: Decomposition of $m-1$ for Itoh-Tsujii Algorithm

m	Decomposition
163	162, 81, 80, 40, 20, 10, 5, 4, 2, 1
233	232, 116, 58, 29, 15, 14, 7, 6, 3, 2, 1
283	282, 141, 140, 70, 35, 17, 16, 8, 4, 2, 1
409	408, 204, 102, 51, 50, 25, 24, 12, 6, 3, 2, 1
571	570, 285, 284, 142, 71, 70, 35, 34, 17, 16, 8, 4, 2, 1

$\beta_k(a) = a^{2^k-1}$, the following property can be derived:

$$\begin{aligned} \beta_{k+j}(a) &= a^{2^{k+j}-1} \\ &= \frac{(a^{2^k})^{2^j}}{a} \\ &= (\frac{a^{2^k}}{a})^{2^j} \cdot \frac{a^{2^j}}{a} \\ &= (a^{2^k-1})^{2^j} \cdot a^{2^j-1} \\ &= \beta_k(a)^{2^j} \cdot \beta_j(a) \end{aligned} \tag{2.26}$$

Thus, the inverse of an element a^{-1} can be computed by $a^{-1} = a^{2^m-2} = (a^{2^{m-1}-1})^2 = (\beta_{m-1}(a))^2$ and repeatedly decompose the value $m-1$ by using Equation (2.26). Table 2.2 shows the decomposition for each m recommended by NIST [18].

To apply Table 2.2, consider the example of using $m = 163$. The finite field inversion of an element, a^{-1} is given by $(\beta_{162}(a))^2$, and $\beta_{162}(a)$ can be evaluated as follows:

$$\begin{aligned} \beta_{162}(a) &= (\beta_{81}(a))^{2^{81}} \cdot \beta_{81}(a) \\ \beta_{81}(a) &= (\beta_{80}(a))^{2^{1}} \cdot \beta_{1}(a) \\ \beta_{80}(a) &= (\beta_{40}(a))^{2^{40}} \cdot \beta_{40}(a) \\ \beta_{40}(a) &= (\beta_{20}(a))^{2^{20}} \cdot \beta_{20}(a) \\ \beta_{20}(a) &= (\beta_{10}(a))^{2^{10}} \cdot \beta_{10}(a) \\ \beta_{10}(a) &= (\beta_{5}(a))^{2^{5}} \cdot \beta_{5}(a) \\ \beta_{5}(a) &= (\beta_{4}(a))^{2^{1}} \cdot \beta_{1}(a) \\ \beta_{4}(a) &= (\beta_{2}(a))^{2^{2}} \cdot \beta_{2}(a) \\ \beta_{2}(a) &= (\beta_{1}(a))^{2^{1}} \cdot \beta_{1}(a) \\ \beta_{1}(a) &= a^{2^1-1} = a \end{aligned} \tag{2.27}$$

By doing so, finite field inversion can be evaluated by a series of multiplications and repeated squarings. Furthermore, notice that by decomposing the values as given in Table 2.2, in order to compute the next $\beta_{k+j}(a)$ value, only the current $\beta_k(a)$ and $\beta_1(a) = a$ values need to be stored. All other temporary values can be overwritten. Using the method shown in [64], the Itoh-Tsujii can be further optimized for the 409 and 571 key sizes.

Another method of evaluating finite field inversion is by using the binary inversion algorithm shown in Algorithm 2.49 in [55] and reproduced in Algorithm 2.5. The binary inversion algorithm is derived from the inversion based on the extended Euclidean algorithm, which is beyond the scope of this thesis. It is important to note in Algorithm 2.5 that the only operations required are additions and right-shift operations (division by t), which can be very easily accomplished in hardware. Furthermore, upon careful analysis of the algorithm, one can see that at the completion of the **if** statement at the end of the **while** loop, only one of u and v is divisible by t, but not both. Thus, only one of the two inner **while** loops is entered in each iteration of the outer **while** loop. Since each inner **while** loop divides u or v by t, it reduces them by 1 bit during each iteration. Thus, the outer **while** loop executes a maximum of $2m$ times, where m is the bit length of a.

Algorithm 2.5 Binary inversion algorithm in binary fields

```
Input: Irreducible polynomial, $f$, and binary field element, $a = a_{m-1}t^{m-1} + \cdots + a_1t^1 + a_0$
Output: $a^{-1} \bmod f$
  $u \leftarrow a$, $v \leftarrow p$, $g_1 \leftarrow 1$, $g_2 \leftarrow 0$
  while $u \neq 1$ and $v \neq 1$ do
    while $t$ divides $u$ do
      $u \leftarrow u/t$
      if $t$ divides $g_1$ then $x_1 \leftarrow g_1/t$ else $g_1 \leftarrow (g_1 + f)/t$ end if
    end while
    while $t$ divides $v$ do
      $v \leftarrow v/t$
      if $t$ divides $x_2$ then $g_2 \leftarrow g_2/t$ else $g_2 \leftarrow (g_2 + f)/t$ end if
    end while
    if $\deg(u) \geq \deg(v)$ then $u \leftarrow u + v$, $g_1 \leftarrow g_1 + g_2$ else $v \leftarrow v + u$, $g_2 \leftarrow g_2 + g_1$ end if
  end while
  if $u = 1$ then return $(g_1)$ else return $(g_2)$ end if
```

2.3.2 Finite Field Arithmetic in Prime Fields

Prime field arithmetics are operations performed on a closed set of integers. The operation that keeps all values within range is the modulo operation. Thus, prime field operations are exactly the same as integer operations, followed by a modulo p, where p is a prime number. Therefore, values in prime fields can be represented as binary integers of length m.

Addition and subtraction in prime fields are performed as integer addition or subtraction. Thus, the difference between prime field and binary field additions is that in prime fields there needs to be a carry chain to propagate the carry, whereas in binary fields, a simple bit-wise XOR operation would suffice. Multiplication can use similar algorithms as in binary fields by using Comba algorithm or Karatsuba-Ofman algorithm.

Since all operations are followed by a modulo p operation, the algorithm to perform the modulo operation has an impact on the performance of all prime field operations. Fortunately, the 5 prime numbers recommended by NIST have been carefully selected to make the reduction step more efficient. The following are the 5 NIST primes:

$$\begin{aligned} p_{192} &= 2^{192} - 2^{64} - 1 \\ p_{224} &= 2^{224} - 2^{96} - 1 \\ p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\ p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \\ p_{521} &= 2^{521} - 1 \end{aligned} \tag{2.28}$$

In order to see the efficiency of taking the modulo of the above prime numbers, consider the example shown in [55]. For $p_{192} = 2^{192} - 2^{64} - 1$, the following equations can be written:

$$\begin{aligned} 2^{192} &= 2^{64} + 1 \pmod{p_{192}} \\ 2^{256} &= 2^{128} + 2^{64} \pmod{p_{192}} \\ 2^{320} &= 2^{256} + 2^{128} = 2^{128} + 2^{64} + 1 \pmod{p_{192}} \end{aligned} \tag{2.29}$$

Consider an integer c, where $0 \le c < p_{192}^2$. It can be written in base-2^{64} as:

$$c = c_5 2^{320} + c_4 2^{256} + c_3 2^{192} + c_2 2^{128} + c_1 2^{64} + c_0 \tag{2.30}$$

where c_i is a 64-bit integer, so using Equation (2.29) to reduce Equation (2.30), results in the following:

$$\begin{aligned} c &= c_5 2^{128} &+& c_5 2^{64} &+& c_5 \\ &+ c_4 2^{128} &+& c_4 2^{64} && \\ &+ && c_3 2^{64} &+& c_3 \\ &+ c_2 2^{128} &+& c_1 2^{64} &+& c_0 \pmod{p_{192}} \end{aligned} \tag{2.31}$$

Thus, the reduction of c becomes a sum of 4 192-bit integers. Using a similar method, the reduction algorithm for the other NIST primes can also be derived. The reduction algorithms for all 5 NIST primes can be found in Appendix B.

Similar to binary finite fields, inversion or division is also the most complex operation in prime fields. In prime fields, the inverse of an element a is defined as, $b = a^{-1} \mod p$, if $a \cdot b = 1 \mod p$. One method of evaluating the inverse of an element is the binary inversion algorithm shown in Algorithm 2.22 in [55] and shown in Algorithm 2.6.

Algorithm 2.6 Binary inversion algorithm in prime fields

```
Input: Prime number, $p$, and prime field element, $a$
Output: $a^{-1} \mod p$
  $u \leftarrow a$, $v \leftarrow p$, $x_1 \leftarrow 1$, $x_2 \leftarrow 0$
  while $u \neq 1$ and $v \neq 1$ do
    while $u$ is even do
      $u \leftarrow u/2$
      if $x_1$ is even then $x_1 \leftarrow x_1/2$ else $x_1 \leftarrow (x_1 + p)/2$ end if
    end while
    while $v$ is even do
      $v \leftarrow v/2$
      if $x_2$ is even then $x_2 \leftarrow x_2/2$ else $x_2 \leftarrow (x_2 + p)/2$ end if
    end while
    if $u \geq v$ then $u \leftarrow u - v$, $x_1 \leftarrow x_1 - x_2$ else $v \leftarrow v - u$, $x_2 \leftarrow x_2 - x_1$ end if
  end while
  if $u = 1$ then return $(x_1 \mod p)$ else return $(x_2 \mod p)$ end if
```

As in the binary fields case, the binary inversion algorithm evaluates inversion with only a series of addition, subtraction and division by 2, which is simply a right-shift operation. Using the same analysis as in the binary fields case, one can see that the algorithm also requires $2m$ iterations, where m is the bit length of p.

Another method of evaluating inversion in prime fields is by using Fermat's Little Theo-

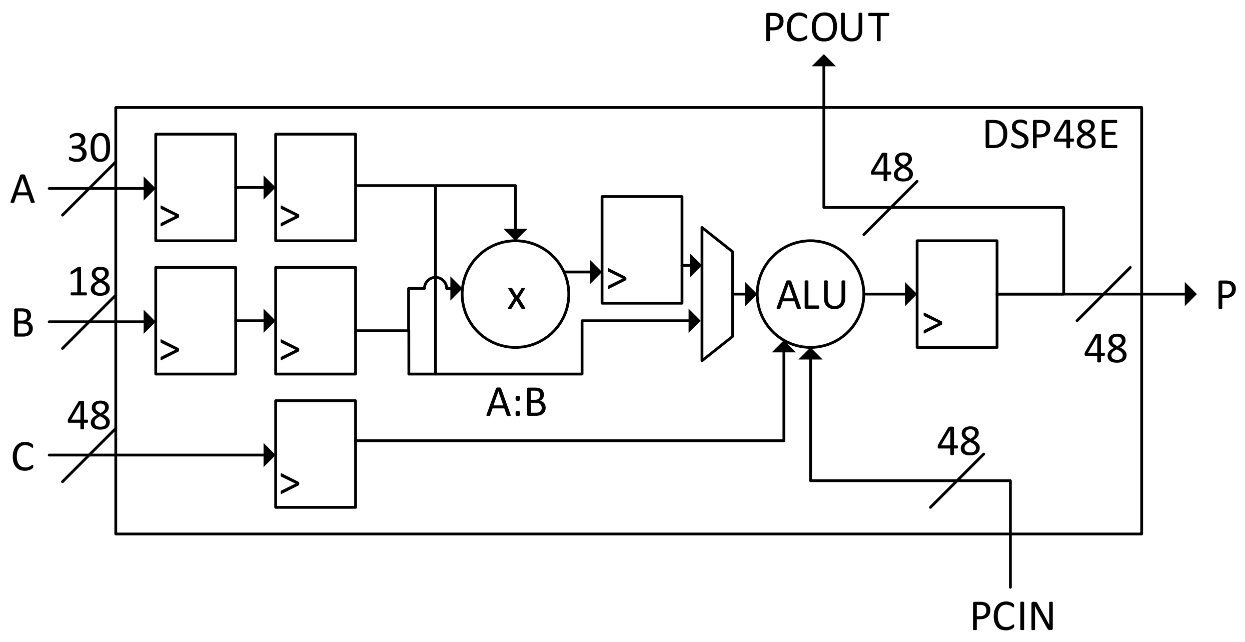


Figure 2.8: Simplified block diagram of the DSP48E slice.

rem. The theorem shows that $a^p \pmod p = a \pmod p$, so the following can be derived:

$$\begin{aligned} a^p \pmod p &= a \pmod p \\ a^{p-1} \pmod p &= 1 \pmod p \\ a \times a^{p-2} \pmod p &= 1 \pmod p \end{aligned} \tag{2.32}$$

Since $a \times a^{-1} \pmod p = 1 \pmod p$, then it follows that $a^{-1} \pmod p = a^{p-2} \pmod p$. Using this property, the inverse of an element in prime field can be evaluated using exponentiation by $p-2$, which can be evaluated by a series of multiplications and squaring operations.

2.4 Xilinx Virtex-5 FPGA and XtremeDSP Slices

In this thesis, the target platform selected for implementing the ECC processors is the Xilinx Virtex-5 family FPGA [65]. The Virtex-5 FPGA family features 6-input look-up tables (LUT), which are improved over the 4-input LUTs formerly used by older FPGAs. It also contains 36-Kbit block RAM (BRAM) and XtremeDSP slices, among other features. In this section, the XtremeDSP slices are discussed in particular, as they are used by the ECPs described in Chapter 4 and Chapter 5. The information in this section can be found in [66].

The XtremeDSP, or DSP48E, slice is a hardware arithmetic block resource that is available in Xilinx Virtex-5 FPGAs. The components of the DSP48E slice are fixed, but the slice can be configured to perform different operations. A simplified block diagram of the DSP48E slice is shown in Figure 2.8. The main components of the DSP48E slice are the 25×18-bit multiplier and a 48-bit arithmetic logic unit (ALU). The multiplier can be bypassed, in which case the inputs 'A' and 'B' are concatenated ('A:B' in Figure 2.8) to form one of the 48-bit operands of the ALU and port 'C' provides the other 48-bit operand. The 'PCIN' and 'PCOUT' ports are used to cascade DSP48E slices and can only be connected to the 'PCOUT' and 'PCIN' ports of an adjacent DSP48E slice.

The registers at the input ports 'A' and 'B' can be selected to have 0, 1 or 2 registers to facilitate pipeline balancing when using multiple cascaded DSP48E slices. The 'A' and 'B' ports may also use pipelining registers when cascading the DSP48E slices. The attributes AREG, BREG, ACASCREG, and BCASCREG select the number of register. Input port 'C' can be selected to have 0 or 1 registers using the CREG attribute. Since these are attribute settings, they can only be configured pre-synthesis and cannot be modified at runtime.

The ALU can be configured to perform addition, subtraction or other logic operations. The ALU may use 3 inputs that are selected by multiplexers. These are called X, Y and Z multiplexers in [66]. The OPMODE input to the DSP48E slice is 7 bits wide and can be changed at runtime to select different inputs for the ALU. As shown in Table 2.3, Table 2.4 and Table 2.5, OPMODE[1:0] controls the X multiplexer, OPMODE[3:2] controls the Y multiplexer and OPMODE[6:4] controls the Z multiplexer. In the tables, the selection of P feeds the output register back into the ALU without going outside of the DSP48E slice. In addition, in the Z multiplexer, the P or PCIN values can be right-shifted by 17 bits before inputting into the ALU. This is a built-in feature that does not require additional hardware resources and is used in the ECP design in Chapter 4.

In addition to the inputs, the operation of the ALU may also be selected during runtime through the 4-bit ALUMODE input. There are 2 types of operations that the ALU can perform: 3-operand addition/subtraction or 2-input logical operations. The operations for each input are shown in Table 2.6 and Table 2.7. When used for 3-input operations, the ALU uses the output of the X, Y and Z multiplexers as its inputs and performs different combi-

Table 2.3: OPMODE control signals for X Multiplexer

X OPMODE[1:0]	X Multiplexer Output	Notes
00	0	Default
01	M	OPMODE[3:2] must be 01
10	P	internal feedback from output register P
11	A:B	48-bit concatenation of ports 'A' and 'B'

Table 2.4: OPMODE control signals for Y Multiplexer

Y OPMODE[3:2]	Y Multiplexer Output	Notes
00	0	Default
01	M	OPMODE[1:0] must be 01
10	48'ffffffffffff	used for bitwise operation
11	C	

Table 2.5: OPMODE control signals for Z Multiplexer

Z OPMODE[6:4]	Z Multiplexer Output	Notes
000	0	Default
001	PCIN	
010	P	
011	C	
100	P	use for MACC extend only
101	17-bit shift PCIN	
110	17-bit shift P	
111	xx	Invalid

Table 2.6: Three-input ALU operations

ALUMODE[3:0]	Operation
0000	Z + X + Y + CARRYIN
0011	Z - (X + Y + CARRYIN)
0001	-Z + (X + Y + CARRYIN) - 1 = not(Z) + X + Y + CARRYIN
0010	not(Z + X + Y + CARRYIN) = -Z - X - Y - CARRYIN - 1

Table 2.7: Two-input ALU operations

OPMODE[3:2]	ALUMODE[3:0]	Operation
00	0100	X XOR Z
00	0101	X XNOR Z
00	0110	X XNOR Z
00	0111	X XOR Z
00	1100	X AND Z
00	1101	X AND (NOT Z)
00	1110	X NAND Z
00	1111	(NOT X) OR Z
10	0100	X XNOR Z
10	0101	X XOR Z
10	0110	X XOR Z
10	0111	X XNOR Z
10	1100	X OR Z
10	1101	X OR (NOT Z)
10	1110	X NOR Z
10	1111	(NOT X) AND Z

nations of integer additions and subtractions. When performing 2-input logical operations, only the X and Z multiplexer outputs are used and the Y multiplexer is set to either output 0 (OPMODE[3:2] = 00) or 48'ffffffffffff (OPMODE[3:2] = 10). The value of OPMODE[3:2] modifies the logical operation performed as shown in Table 2.7.

There is also a pattern detection feature built into the DSP48E block that is not shown in Figure 2.8. It compares the output ‘P’ to a pattern and outputs whether or not they match. The DSP48E slices can operate at up to 550 MHz by using all the pipeline stages, which is more efficient than implementing hardware multipliers using other FPGA logic (i.e. configurable logic blocks (CLB)). In addition, the ability of the ALU to switch between computing 48-bit addition and 48-bit XOR operation makes the DSP48E slice an optimal choice for implementing a unified ECP, since it is able to switch between prime and binary field operations on-the-fly.

Chapter 3

Scalable ECC Processors for Binary Curves

3.1 163-bit ECC Processor[1]

In this section, an ECC processor (ECP) that is specific to the 163-bit pseudo-random curve recommended by NIST [18] is discussed. This work is published in [50] and will be referred to as '163-bit ECP' in the remainder of this thesis. The architecture of the processor is based on the design presented by Zhang et al. [14]. The design goal of the ECP in [14] is to reduce the computation latency of ECPM by using 3 finite field arithmetic logic units (ALUs) in parallel. Furthermore, the design is optimized for $GF(2^{163})$, so it is not a scalable design. Nevertheless, this processor demonstrates some of the characteristics of an ECP.

Figure 3.1 shows the block diagram of the ECC processor presented by [14]. Figure 3.1 shows 3 finite field ALU cores that operate in parallel and the main controller that controls the operations of each core. Each ALU core, shown in Figure 3.2 [14], is made up of a finite field multiplier, an adder, a squarer (A^2) and a double squarer (A^4). The 163-bit multiplication is performed by taking 4 41×163-bit multiplications, shifting and adding, and reducing the result. The squarer is a hard-wired 163-bit finite field squarer, where the interleaving of bits in Equation (2.19) and the finite field reduction are combined into a single combinational logic block. Similarly, the double squarer is also a combinational logic block that interleaves more zero bits and integrates reduction. Overall, each core can simultaneously compute the results of $A \cdot B$, $(A + B)$ or $(A + B)^2$, and A^4. By doing so, the processor is able to take advantage of instruction level parallelism without sacrificing the clock frequency because the critical path of the processor lies on the multiplier.

[1]The work in this section is published in *ISED 2012* [50].

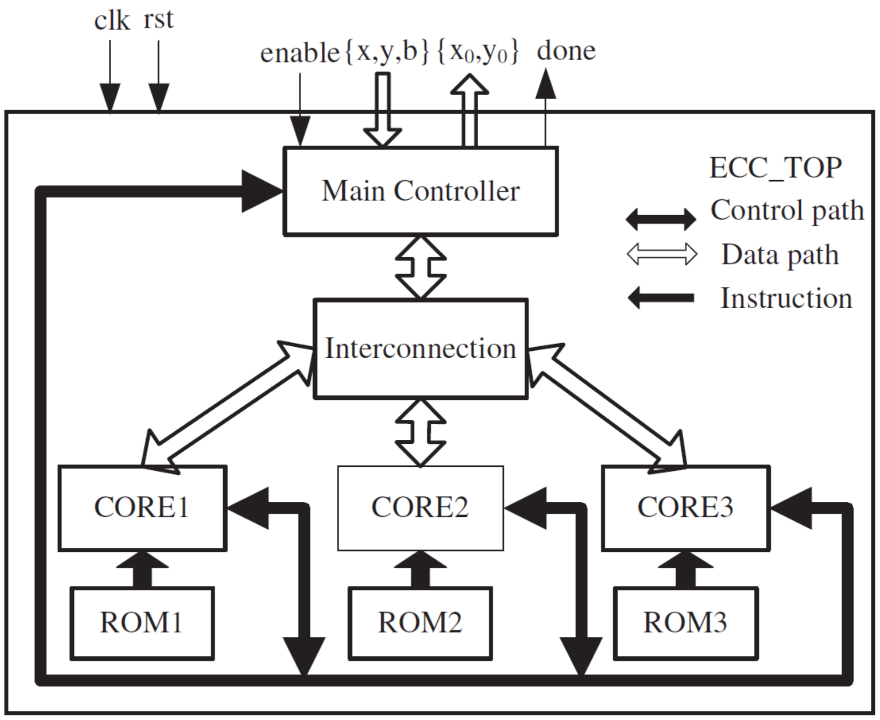


Figure 3.1: ECC processor from Zhang et al. [14]

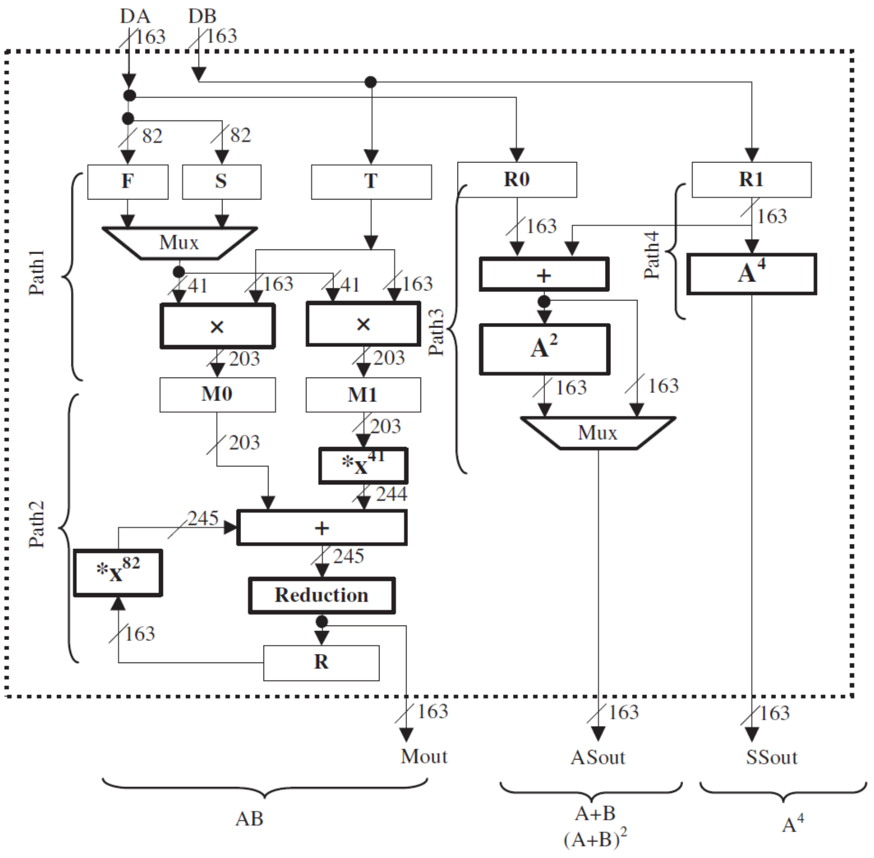


Figure 3.2: ALU of the ECC processor from Zhang et al. [14]

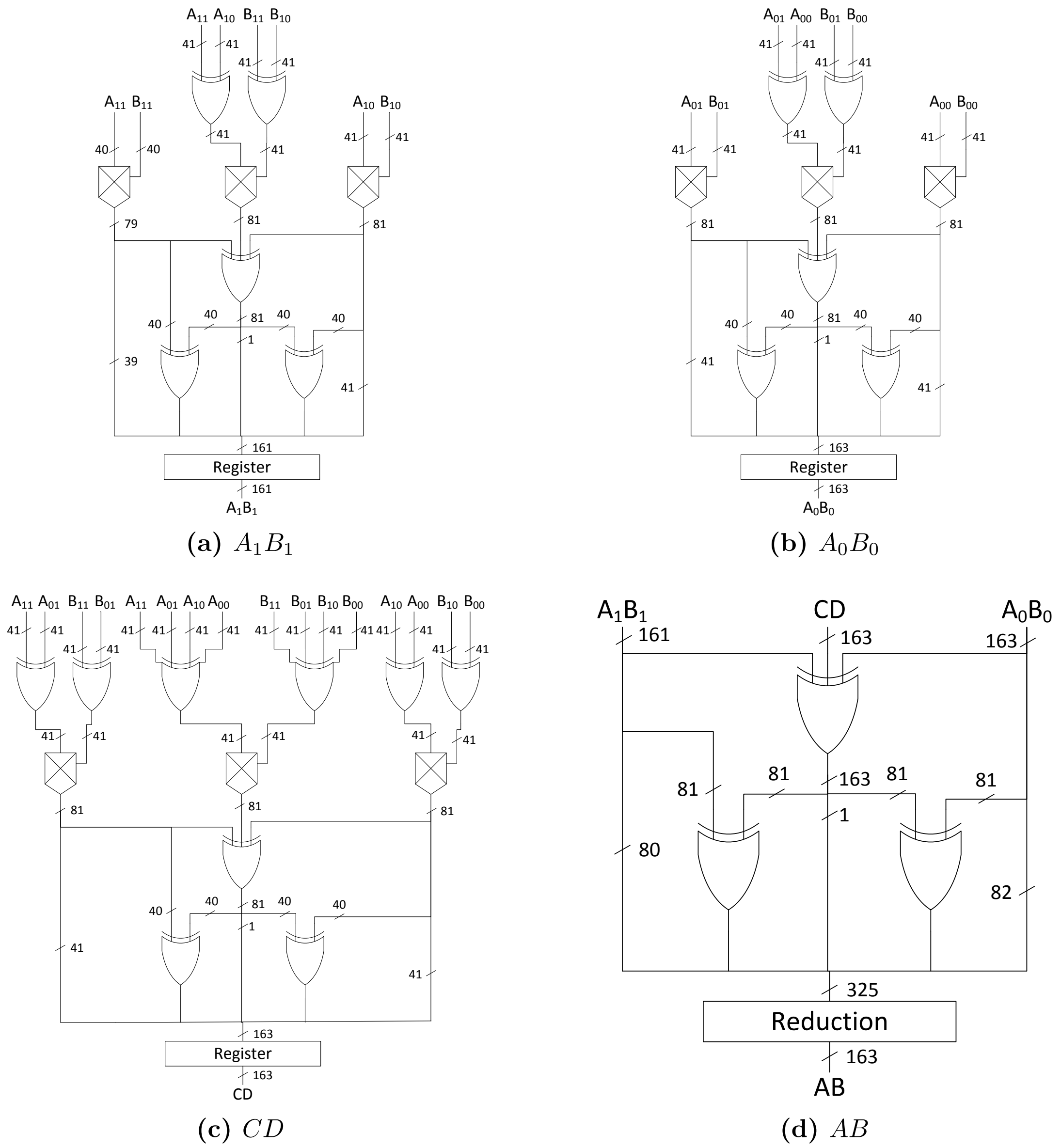
Table 3.1: Implementation Results and Comparison of the 163-bit ECP

Work	FPGA	Slices	Max. Freq. (MHz)	Latency (μs)
163-Bit ECP	Virtex-4 XC4VLX80	23,547	163	6.72
Zhang et al. [14]	Virtex-4 XC4VLX80	20,807	185	7.7

In the 163-bit ECP, the design of the multiplier has been improved. Instead of using two 41×163-bit multipliers as in [14], 2 levels of Karatsuba-Ofman multiplication are implemented. Figure 3.3 shows the architecture of the two-stage Karatsuba-Ofman multiplier, where 9 41×41-bit multipliers are used. By doing so, the number of clock cycles to compute each multiplication reduces from 3 to 2, which reduces the latency of the ECPM calculation. In addition, the 163-bit ECP also proposes a minor modification to use the 3 cores more efficiently when computing the finite field inversion for all 3 projective coordinates. As a result, the ECPM calculation reduces from 1428 clock cycles in [14] to 1088 clock cycles.

The design is implemented for a Virtex-4 FPGA and the results are shown in Table 3.1. Compared to the implementation results in [14], the number of slices utilized increases from 20,807 to 23,547 but the latency decreases from 7.7 μs to 6.72 μs. In the proposed implementation, the design goal is to increase the speed of the ECPM calculation for server-side applications. Thus, the increase in hardware resource utilization is an acceptable trade-off to decrease the latency.

In the above mentioned processor, one can notice that performance of the ECP depends highly on the implementation of the finite field arithmetic unit. By using the Itoh-Tsujii inversion algorithm, the most complex operation in the arithmetic unit is the multiplication, so the implementation of the finite field multiplier is crucial to the performance of the ECC processor. Thus, the implementation of scalable ECPs described in the subsequent sections focuses on managing the critical path of the multiplier while integrating the support for multiple curves into a single module.



(a) A_1B_1 **(b)** A_0B_0 **(c)** CD **(d)** AB

Figure 3.3: Architecture of 2-stage Karatsuba-Ofman multiplier

3.2 Scalable ECC Processor for Binary Curves

This section discusses architectures of scalable ECP implementation for curves over binary fields recommended by NIST [18]. The architectures discussed in this section can be found in [51, 52]. The work in [51] supports all 5 Koblitz curves, and is referred to as 'Koblitz ECP' in this thesis. The work in [52] supports all 5 pseudo-random curves recommended by NIST, and is referred to as 'Random ECP' in this thesis.

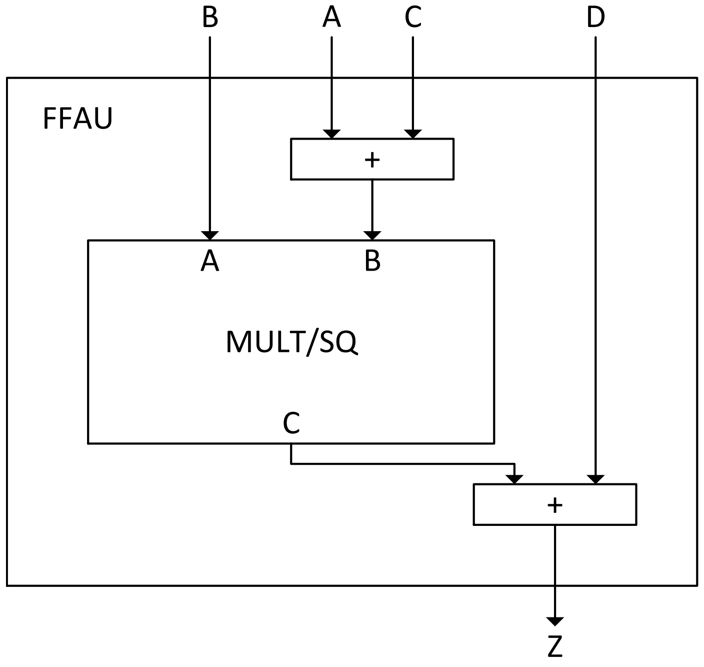
3.2.1 Scalable ECP over Koblitz Curves[2]

Recall from Algorithm 2.3 that using τNAF point multiplication, the point doubling (PDBL) operation is simplified to squaring of each coordinate using the Frobenius endomorphism (PFRB) operation. Thus, the point addition (PADD) operations requires more attention. In the Koblitz ECP, the finite field arithmetic unit (FFAU) is designed to either compute multiplication or squaring along with an addition at the input and at the output. Figure 3.4 shows the block diagram of the FFAU and the MULT/SQ unit. As shown in Figure 3.4(a), the FFAU computes two formats of operations, namely $Z = (C + A) \cdot B + D$ (MULT mode) and $Z = B^2 + D$ (SQ mode). In order to utilize the proposed FFAU architecture, the PADD operations in Equation (2.11) are rearranged as follows:

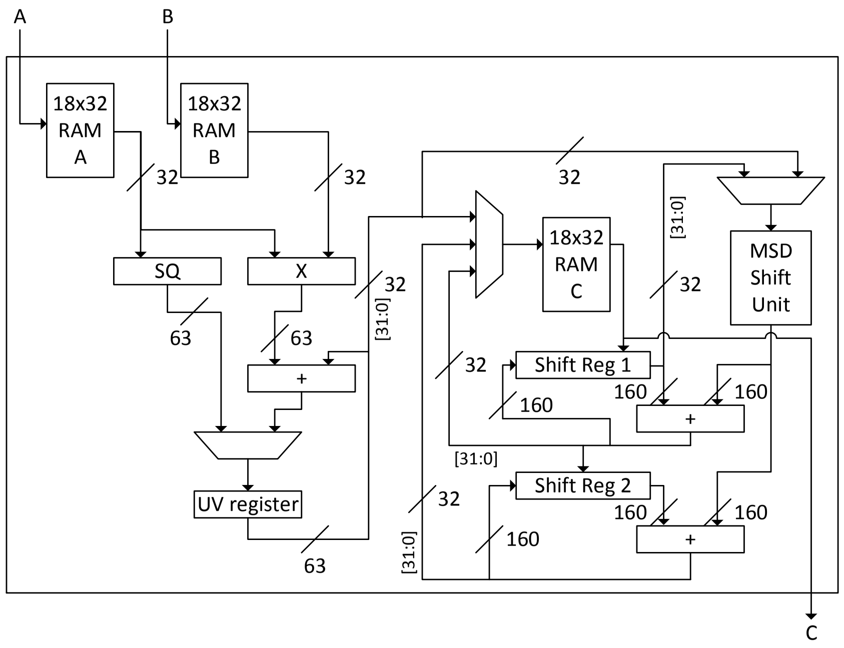
$$\begin{aligned}
T_1 &\leftarrow (0 + X_2) \cdot Z_3 + X_3 \\
X_3 &\leftarrow (0 + Z_3) \cdot T_1 + 0 \\
T_3 &\leftarrow Z_3^2 + 0 \\
Y_3 &\leftarrow (0 + Y_2) \cdot T_3 + Y_3 \\
Z_3 &\leftarrow X_3^2 + 0 \\
T_2 &\leftarrow (0 + Y_3) \cdot X_3 + 0 \\
T_1 &\leftarrow T_1^2 + 0 \\
X_3 &\leftarrow (aT_3 + X_3) \cdot T_1 + T_2 \\
X_3 &\leftarrow Y_3^2 + X_3 \\
T_1 &\leftarrow (0 + X_2) \cdot Z_3 + X_3 \\
Y_3 &\leftarrow Z_3^2 + 0 \\
Y_3 &\leftarrow (X_2 + Y_2) \cdot Y_3 + 0 \\
Y_3 &\leftarrow (T_2 + Z_3) \cdot T_1 + Y_3
\end{aligned} \tag{3.1}$$

Figure 3.4(b) shows a more detailed architecture of the MULT/SQ unit. The unit inputs the operands digit-wise, where the digit is selected to be 32-bits. The input values are stored in the 18×32-bit RAMs. For multiplication, the values in the RAMs are read out according to the indexes in the inner loops of the Comba algorithm in Algorithm 2.4 and passed to the multiplier block ('$\times$'). For squaring, the values in RAM A are read out sequentially and

[2]The work in this section is published in *Microprocessors and Microsystems* [51].



(a) Block diagram of the FFAU.



(b) Block diagram of the MULT/SQ unit.

Figure 3.4: Block diagram of the Finite Field Arithmetic Unit.

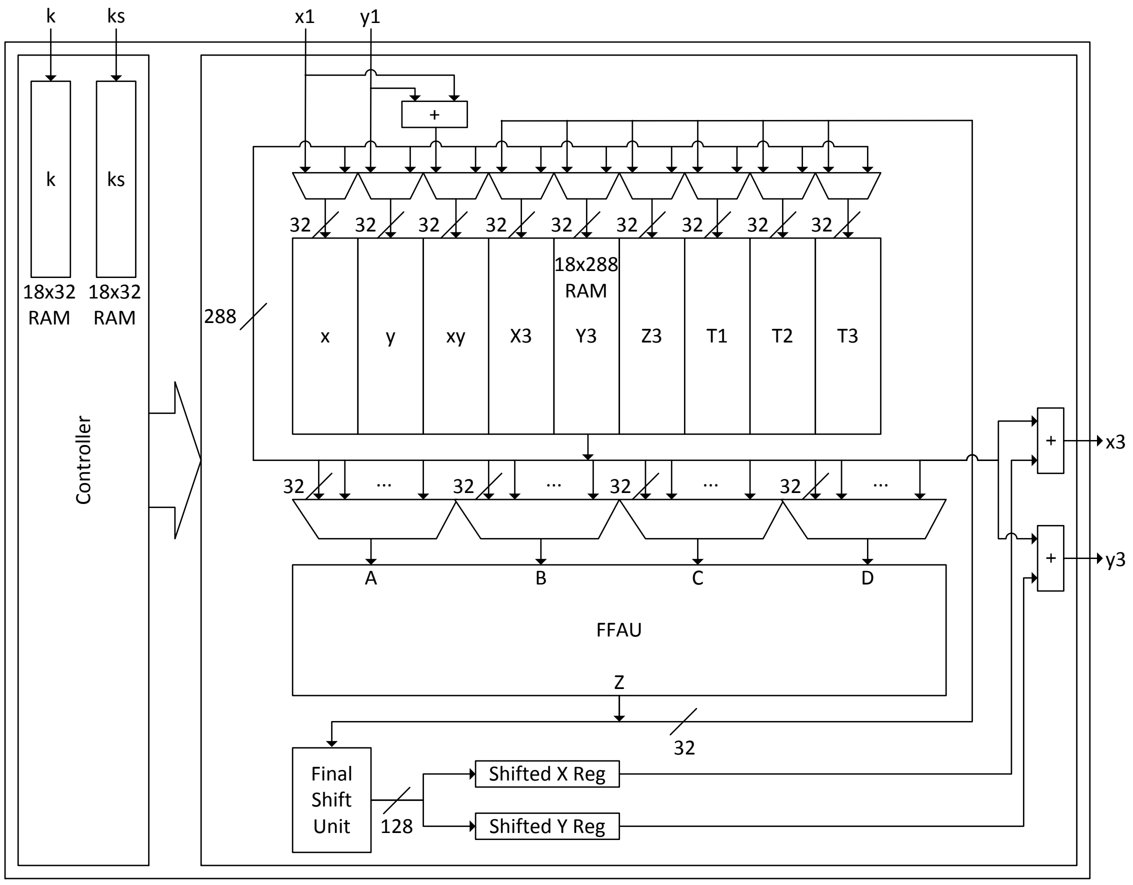


Figure 3.5: Block diagram of the Koblitz ECP.

passed to the 'SQ' block. The 32-bit multiplier ('$\times$') is a combinational Karatsuba-Ofman multiplier and the 'SQ' block interleaves zero bits with operand bits to evaluate Equation (2.19). The UV register accumulates the result of $(U, V) \leftarrow (U, V) + A_j \cdot B_{i-j}$ given in Algorithm 2.4. The right side of Figure 3.4(b) performs the reduction operation as shown in Algorithm A.1 – Algorithm A.5 with the exception of the final reduction step. In the Koblitz ECP, the output of the FFMULT and FFSQ is never completely reduced to improve efficiency. Since FFMULT and FFSQ are performed digit-wise, the reduction is only completed to the border of the digit. For example, in 233-bit mode, $s = \lceil 233/32 \rceil = 8$, so the reduction operation, reduces the digits $(Z_{15}, \ldots, Z_9, Z_8)$, but does not reduce the 23 most-significant bits (MSB) in Z_7, making all the intermediate results $8 \times 32 = 256$ bits instead of 233 bits. The final reduction step, which reduces the 23 MSBs of Z_7 occurs after all the calculations are performed and just before the result is output.

Figure 3.5 shows the top level block diagram of the Koblitz ECP. The affine coordinates of the point, (x_1, y_1), enter the module as 32-bit digits, along with the τNAF converted value of k, with the magnitude entering in k and the sign bits in ks. The controller contains the finite state machine (FSM) that dictates the control flow of the processor. The temporary values are stored in the RAM and the 'Final Shift Unit' performs the final reduction step in Algorithm A.1 – Algorithm A.5. Finally, the resultant affine coordinates are output through x_3 and y_3.

During the projective to affine coordinate conversion, a finite field inversion is required. The ECP is able to implement inversion using the same FFAU described above by using the Itoh-Tsujii algorithm described in Section 2.3.1. As previously mentioned, the Itoh-Tsujii algorithm converts inversion into a series of multiplication and squaring operations. Thus, the FFAU is able to support inversion without any modifications and a separate inversion unit is not required.

In order for the architecture to be scalable, the RAM must be able to support $\lceil 571/32 \rceil =$ 18 digits. Furthermore, since the FFAU inputs operands digit-by-digit, the architecture supports multiple field sizes without the need to reconfigure the hardware making the proposed ECP scalable.

The main contribution of the architecture of the Koblitz ECP is the compactness of the design, while providing the ability of evaluating the ECPM of all 5 NIST recommended Koblitz curves without the need to reconfigure the hardware. Furthermore, the finite field addition operations are integrated with the multiplication and squaring without affecting the critical path, which lies inside the MULT/SQ unit.

Table 3.2 shows the implementation results of the above mentioned Koblitz ECP on FPGA platforms. The register, look-up table (LUT), slice, BRAM and maximum frequency values are obtained post-place and route from the Xilinx ISE Software. The latency value is the time to compute the ECPM operation obtained by multiplying the number of clock cycles needed to evaluate the ECPM by the minimum clock period of the design (minimum period = 1 / maximum frequency):

$$\begin{aligned} \text{Latency} &= (\text{Number of clock cycles per ECPM}) \times (\text{Minimum clock period}) \\ &= \frac{\text{Number of clock cycles per ECPM}}{\text{Maximum clock frequency}} \end{aligned} \tag{3.2}$$

Table 3.2: Implementation Results and Comparison of the Koblitz ECP

Work	FPGA	Registers	LUT	Slices	BRAM	Max. Freq. (MHz)	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
Koblitz ECP	Spartan-3 XC3S400	1,232	3,850	2,220	8	93.08	163	0.456	0.988
							233	1.008	0.447
							283	1.227	0.367
							409	3.215	0.140
							571	7.236	0.062
Hassan and Benaissa [39]	Spartan-3 XC3S200	913	2,028	1,278	4	90	163	15.5	0.050
							283	45.1	0.017
							571	121.4	0.006

To more easily compare the performance between different hardware implementation, the efficiency metric is used throughout this thesis and is defined as:

$$\begin{aligned}\text{Efficiency} &= \frac{\text{Number of ECPM per second}}{\text{Number of slices}} \\ &= \frac{1}{(\text{Latency}) \times (\text{Number of slices})}\end{aligned} \tag{3.3}$$

The efficiency metric is computed as throughput divided by the number of slices, such that a higher efficiency represents better performance. The Koblitz ECP is implemented on a Spartan-3 FPGA to compare its performance with the work in [39], where the authors propose a scalable ECP for Koblitz curves, but only supports 163, 283 and 571 bit lengths. The Koblitz ECP has a 73.7% increase in number of slices used, but decreases the latency by a factor of 16.7 to 34 times. As a result, the efficiency metric shows that the Koblitz ECP outperforms the work in [39]. It is important to note that the same hardware is used to evaluate the ECPM for the 5 Koblitz curves and the selection of the curve can be performed on-the-fly with a change of the input value to the ECP. As previously mentioned, this feature is referred to as scalability in this thesis.

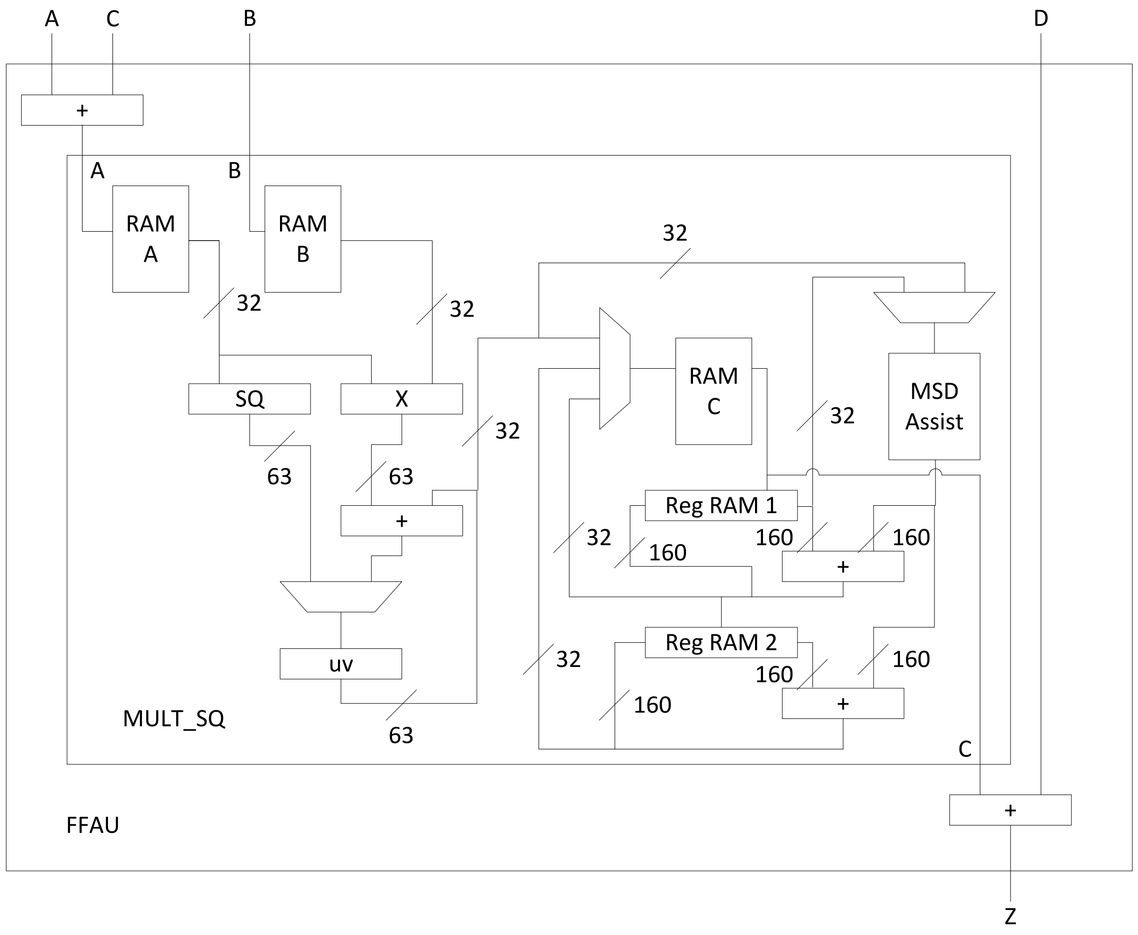


Figure 3.6: Block diagram of the FFAU of the Random ECP.

3.2.2 Scalable ECP over Pseudo-Random Curves[3]

The architecture of the Random ECP is very similar to the one of the Koblitz ECP, except the processor is designed for pseudo-random curves recommended by NIST instead of Koblitz curves. Since the pseudo-random curves use the same field sizes and irreducible polynomials as the Koblitz curve equivalent, the Random ECP uses similar finite field arithmetic blocks as the Koblitz ECP. The block diagram of the FFAU used in the Random ECP is shown in Figure 3.6.

The main difference in the FFAU is that in the Random ECP, it has the ability to perform $Z = (C+A)^2+D$ instead of $Z = B^2+D$ in SQ mode. The extra addition allows for the ECPM operations to be more optimized. In the Random ECP, the Lopez-Dahab (LD) algorithm as

[3]The work in this section is published in *ISCAS 2013* [52].

shown in Algorithm 2.2 is used in the evaluation of the ECPM. In order for the operations to work with the proposed finite field arithmetic blocks, the operations in the main loop in Algorithm 2.2 are modified to the following:

$$\begin{aligned}
T_1 &\leftarrow (0+X_1)*Z_2+0 \\
T_2 &\leftarrow (0+X_2)*Z_1+0 \\
Z_2|Z_1 &\leftarrow (T_1+T_2)^2+0 \\
T_3 &\leftarrow (0+x)*Z_2|Z_1+0 \\
X_2|X_1 &\leftarrow (0+T_1)*T_2+T_3 \\
T_1 &\leftarrow (0+X_1|X_2)^2+0 \\
T_2 &\leftarrow (0+T_1)^2+0 \\
T_3 &\leftarrow (0+Z_1|Z_2)^2+0 \\
R &\leftarrow (0+T_3)^2+0 \\
X_1|X_2 &\leftarrow (0+R)*b+T_2 \\
Z_1|Z_2 &\leftarrow (0+T_1)*T_3+0
\end{aligned} \tag{3.4}$$

Furthermore, the operations in the projective to affine coordinate conversion, Mxy in Algorithm 2.2, also need to be modified to the following to take advantage of the finite field arithmetic blocks:

$$\begin{aligned}
T_1 &\leftarrow (0+x)*Z_1+X_1 \\
T_2 &\leftarrow (0+x)*Z_2+X_2 \\
T_2 &\leftarrow (0+T_1)*T_2+0 \\
T_3 &\leftarrow (0+x)^2+y \\
T_3 &\leftarrow (0+T_3)*Z_2+0 \\
T_3 &\leftarrow (0+T_3)*Z_1+T_2 \\
T_3 &\leftarrow (0+T_1)*T_3+0 \\
T_2 &\leftarrow (0+Z_1)^2+0 \\
T_2 &\leftarrow (0+x)*T_2+0 \\
T_2 &\leftarrow (0+Z_2)*T_2+0 \\
T_1 &\leftarrow (0+T_2)*y+T_3
\end{aligned} \tag{3.5}$$

The above sequence of operations are followed by a finite field inversion on T_2, which stores $xZ_1^2Z_2$ and a multiplication with T_1, which stores the numerator portion of y_0. Subsequently,

Table 3.3: Implementation Results and Comparison of the Random ECP

Work	FPGA	Registers	LUT	Slices	BRAM	Max. Freq. (MHz)	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
Random ECP	Spartan-3 XC3S400	1,337	4,261	2,418	8	79.64	163	0.864	0.479
							233	1.957	0.211
							283	2.514	0.164
							409	6.911	0.060
							571	16.48	0.025
Random ECP	Virtex-4 XC4VFX12	1,241	4,231	2,648	8	142.53	163	0.483	0.782
							233	1.093	0.346
							283	1.404	0.269
							409	3.861	0.098
							571	9.208	0.041
Hassan and Benaissa [17]	Spartan-3 XC3S200	650	2,205	1,127	4	68.26	163	38	0.023
							233	73.4	0.012
							283	104	0.008
							409	251	0.004
							571	287.4	0.003

the inversion of Z_1 is evaluated and its result is multiplied by X_1 to produce x_0.

Table 3.3 shows the Spartan-3 and Virtex-4 implementation results of the Random ECP. The values in the table are obtained in a similar fashion as the values in Table 3.2. However, the implementation results in Table 3.3 cannot be used to compare to the values in Table 3.2 because they evaluate the ECPM for a different set of curves and use different algorithms. Nevertheless, one observation can be made in comparison to the ECP described in Section 3.1. The Virtex-4 implementation of the 163-bit ECP requires 23,547 slices and the latency of the ECPM is 6.72 μs. These results yield an efficiency value of 6.320, which is higher than 0.782 shown in Table 3.3. However, one must remember that the Random ECP is a scalable ECP, which supports all 5 NIST recommended pseudo-random curves without the need to reconfigure the hardware. If the architecture of the 163-bit ECP was to be expanded to support larger key sizes, the hardware resource utilization of the ECP would increase dramatically. The trade-off of efficiency for scalability is expected and is also observed in other architectures described in this thesis. Since the goal of the ECPs in this thesis is to allow for server-side applications to support a wide variety of security requirements, the performance degradation is tolerable.

A comparable design is shown in [17] and its implementation result is also shown in Table 3.3. The design in [17] is very similar to the design in [39], except it handles pseudo-random curves instead of Koblitz curves. The Spartan-3 implementation result of the Random ECP shows that it outperforms the design in [17]. The improved performance is mainly due to the reduced number of clock cycles of the ECPM operation as a result of the use of the FFAU.

3.3 Parallelization of Scalable ECC Processor for Binary Curves[4]

One of the disadvantages of the architecture of the Koblitz ECP and Random ECP described in Section 3.2 is the long latency of each multiplication and squaring operation. Furthermore, since only 1 FFAU is used, the aforementioned ECPs do not take advantage of the potential for parallelism of the ECPM operations. The ECPs implementation described in this section improves on the architecture of the Koblitz ECP and Random ECP by resolving some of the shortcomings. The work presented in this section has been submitted to a journal for peer review in [67].

3.3.1 Parallelization of Scalable ECP over Koblitz Curves

Figure 3.7 shows the revised finite field arithmetic blocks to improve the performance of the previous ECPs. Notice that in the revised design, multiplication and squaring are now separated into 2 blocks and can operate simultaneously. The architecture of the MULT block, as shown in Figure 3.7(a), is very similar to the multiplication operation in the FFAU in Figure 3.4(b), except that 2 32-bit multiplier blocks are used, squaring has been removed, and reduction is not performed in the MULT block. Alternatively, reduction is performed in the SA block shown in Figure 3.7(b). The SA block can also perform addition and repeated squaring operations. Each reduction module in the SA block (i.e. 'R163', 'R233', 'R283', 'R409', 'R571') is based on the reduction matrix according to Equation (2.24). The operands

[4]The work in this section has been submitted to *IEEE Transactions on Computers* for peer review [67].

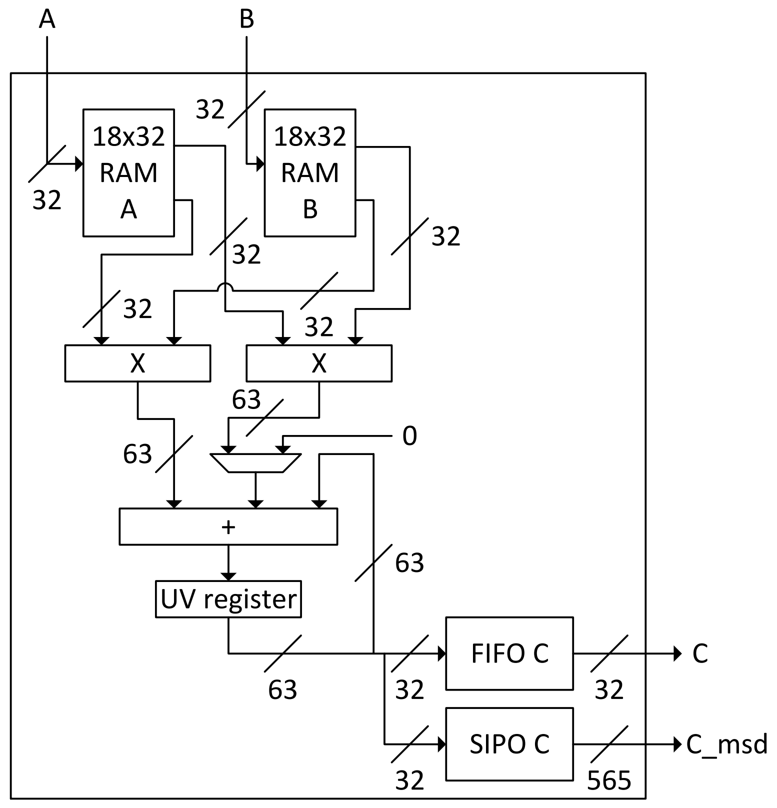
in both the MULT and SA blocks are 32-bit digits.

To demonstrate the operation of the dual '$\times$' block in the MULT block, consider the case of the 163-bit operating mode. The addresses read from the RAMs are shown in Figure 3.8. The contents in RAMs A and B are $(A_5, A_4, \cdots, A_0)$ and $(B_5, B_4, \cdots, B_0)$, where A_i and B_i are 32-bit digits. The numbers shown represent the index, i, of the 32-bit value to be read from the RAM. Each column shows the index read from each RAM port during a specific clock cycle. The vertical bar represents the completion of an inner loop in the Comba algorihtm (Algorithm 2.4). Thus, as shown in Figure 3.8, during the first clock cycle, A_0 and B_0 are read and multiplied in multiplier 1 ('$\times$' block on the left in Figure 3.7(a)), which completes the first inner loop. In the next clock cycle, A_0 and B_1 are multiplied in multiplier 1, and simultaneously, A_1 and B_0 are multiplied in multiplier 2 ('$\times$' block on the right in Figure 3.7(a)), which completes the second inner loop. Notice in Figure 3.8 that the last column of every other section, the RAM access for multiplier 2 is not needed. Thus, Figure 3.7(a) shows a multiplexer to input '0' into the adder when necessary.

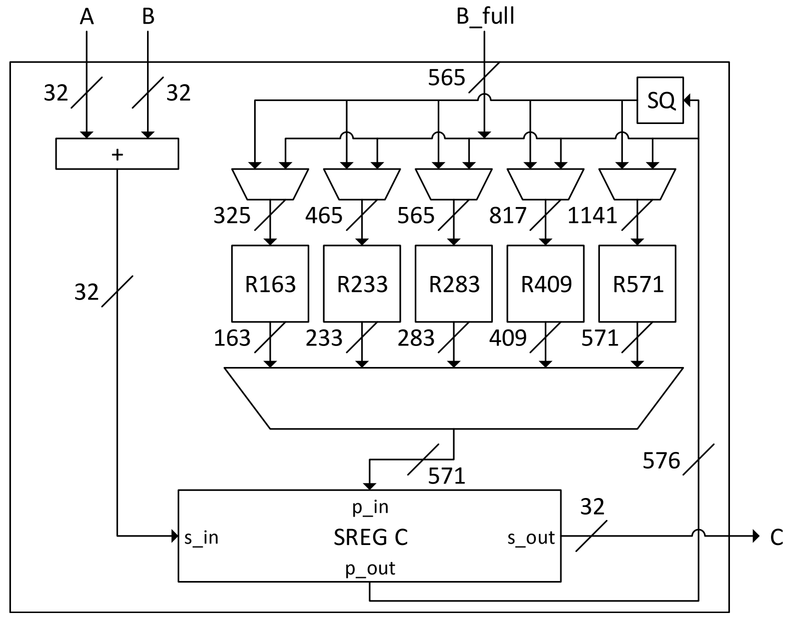
The output of the '$\times$' blocks are accumulated in the 63-bit 'UV register'. The addition operation is performed using XOR operations. Once the inner loop is completed, which corresponds to the vertical bars in Figure 3.8, the least-significant 32 bits of 'UV register' are sent to the 'FIFO C' or 'SIPO C' for storage and the register is right-shifted by 32 bits to prepare for the next inner loop calculation.

The 'FIFO C' and 'SIPO C' blocks are both connected to the output port. The 'FIFO C' block is a first-in first-out register that collects the least significant s 32-bit digits of the product, where $s = \lceil m/32 \rceil$. The 'SIPO C' block is a serial-in parallel-out shift register that collects the most significant s 32-bit digits of the product. When multiplication is complete, the product of $2s$ 32-bit digits, are output from the MULT block through ports 'C' and 'C_msd'. Port 'C' outputs the least significant s digits on a 32-bit bus and 'C_msd' outputs the most significant digits of the product in a bus of up to 565 bits wide. The 'C_msd' port is connected directly to the SA block, where it is concatenated with the least significant digits after the addition operation. By doing so, the SA block requires only s clock cycles to load input values, even when the input is a $2s$-digit product from the MULT block.

One special characteristic of the SA block is its ability to perform repeated squaring with



(**a**) Block diagram of the multiplier (MULT) block.



(**b**) Block diagram of the square-add (SA) block.

Figure 3.7: Block diagram of Parallelized Finite Field Arithmetic Blocks.

Multiplier 1											
Address A	0	0	0 1	0 1	0 1 2	0 1 2	1 2 3	2 3	3 4	4	5
Address B	0	1	2 1	3 2	4 3 2	5 4 3	5 4 3	5 4	5 4	5	5
Multiplier 2											
Address A		1	2	3 2	4 3	5 4 3	5 4	5 4	5	5	
Address B		0	0	0 1	0 1	0 1 2	1 2	2 3	3	4	

Figure 3.8: The RAM addresses read in the MULT block for 163-bit operation.

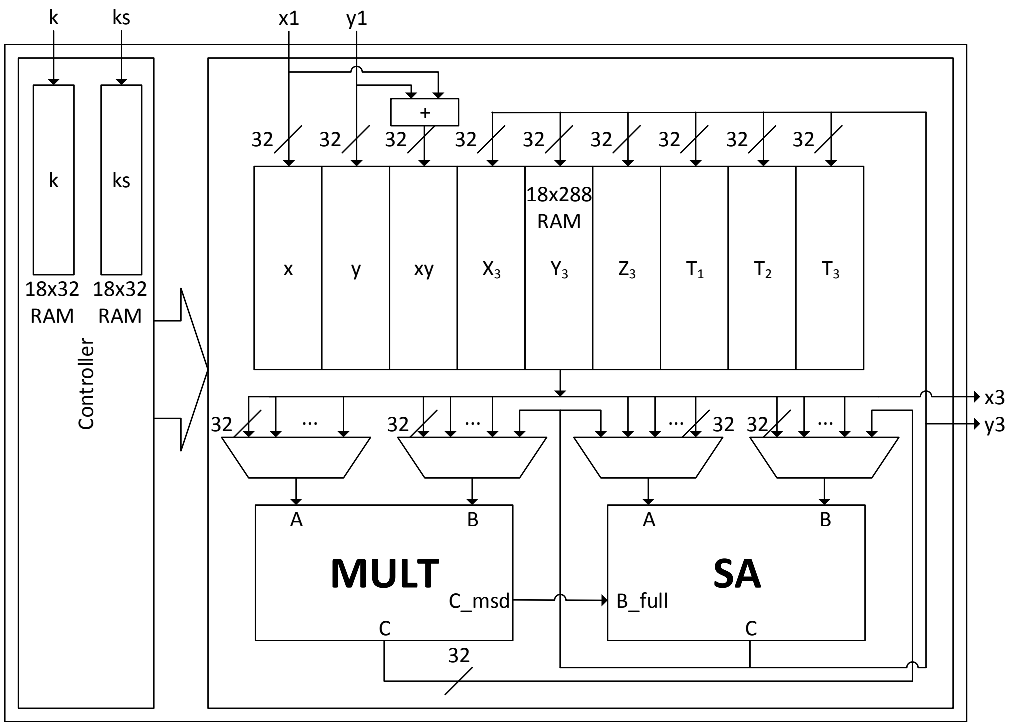


Figure 3.9: Block diagram of the 1-MULT Koblitz ECP.

only 1 addition clock cycle per squaring. The value to be squared is input in the SA block through port ‘A’ as 32-bit digits and collected in ‘SREG C’. The complete operand is then squared through the ‘SQ’ block and reduced through the reduction blocks in 1 clock cycle. This feature is useful in reducing the latency of the Itoh-Tsujii algorithm for inversion, where multiple successive squaring operations are required.

Figure 3.9 shows the top level block diagram of the parallelized scalable ECP for Koblitz curves, also referred to as ‘1-MULT Koblitz ECP’ in this thesis. Notice in Figure 3.9 that the result of the MULT block does not get stored back into the RAM. Instead, the product

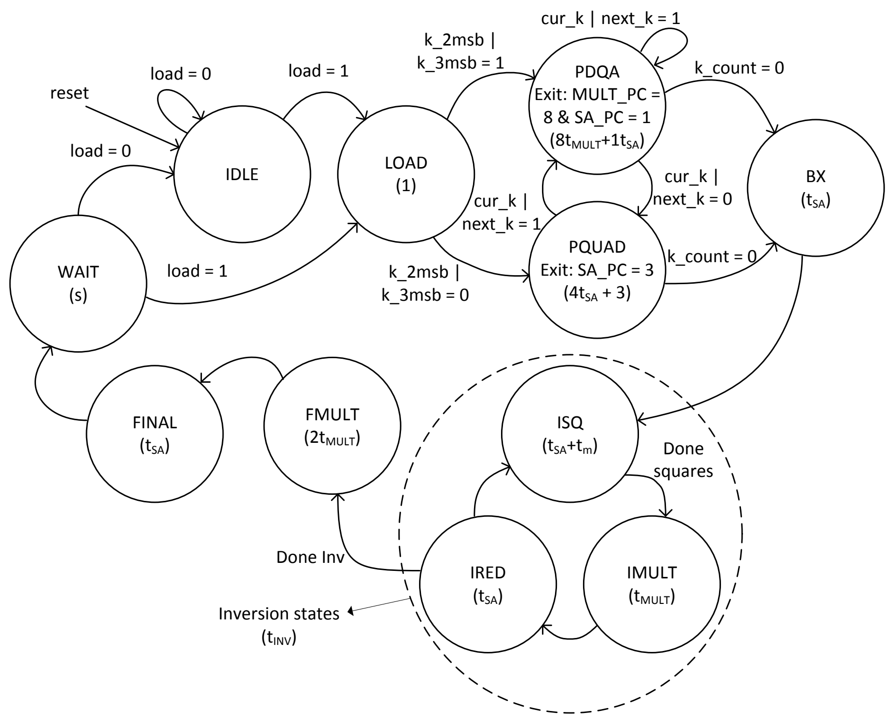


Figure 3.10: FSM of the 1-MULT Koblitz ECP.

enters the SA block through the 'B_full' input and through the multiplexer into the 'B' input. Furthermore, the reduction step of the multiplication is not computed until the next multiplication has started and it is computed in the SA block. By doing so, there is no need to store the result of the multiplication before reduction and the latency is reduced since the reduction steps are masked by the latency of the next multiplication, which is computed in parallel.

Due to the ability of the SA block to perform repeated squaring efficiently, the τNAF point multiplication algorithm in Algorithm 2.3 has been modified such that the PFRB and PADD operations are combined forming the PDQA state in the finite state machine (FSM) of the ECP, which is shown in Figure 3.10. The PQUAD state performs a series of PFRB operations to either square or double-square each coordinate. Furthermore, Table 3.4 shows that the latency in number of clock cycles of the MULT block is higher than the latency of the SA blocks. The lowest ratio of t_{MULT}:t_{SA} occurs in the 163-bit case, where the ratio is 4.29. Thus, up to 4 SA block operations may be executed during the execution of a single MULT block operation.

The operations executed in each state of the FSM is shown in Table 3.5. In Table 3.5,

Table 3.4: Latency in number of clock cycles of the MULT and SA blocks.

m	t_{MULT}	t_{SA}	Ratio (t_{MULT}:t_{SA})
163	30	7	4.29
233	47	9	5.22
283	57	10	5.70
409	107	14	7.64
571	192	19	10.1

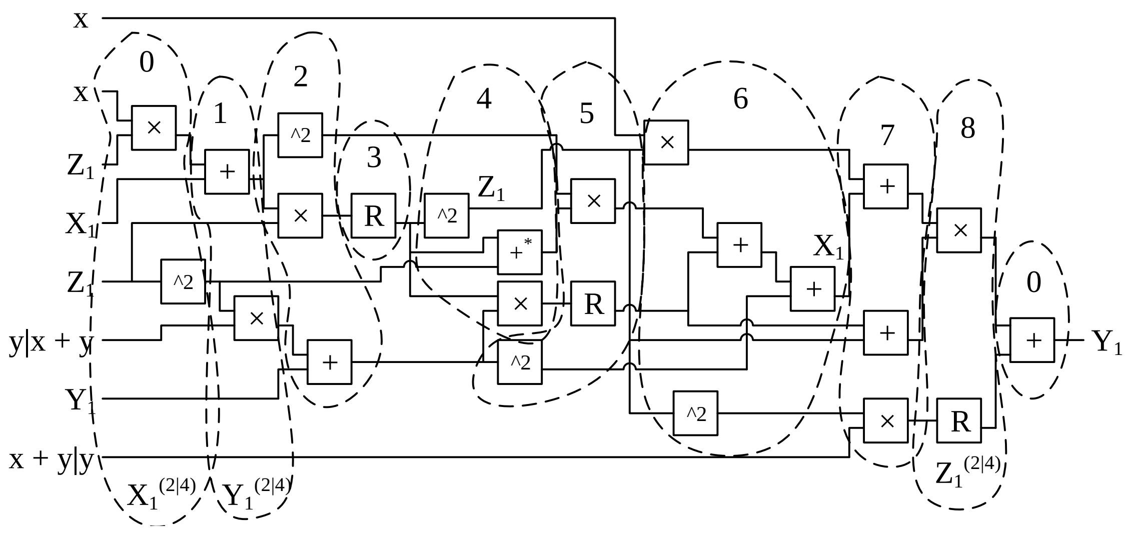


Figure 3.11: Data dependency graph of PDQA for 1-MULT Koblitz ECP.

MULT_PC and SA_PC refer to the program counter for the MULT block and the SA block, respectively. The symbol '|' signifies that the operand or operation is selected based on certain conditions set by the controller. As previously mentioned, every MULT block operation must be followed by a reduction in the SA block. Thus, when SA_PC is 0, the operand M is used to input the product of the MULT block.

In order to take advantage of the ECP architecture shown in Figure 3.9, the data dependency of the Koblitz curve operations in the PDQA state are analyzed to optimize the usage of the MULT and SA blocks and shown in Figure 3.11. The numbers shown inside the dashed-line loops in Figure 3.11 correspond to the MULT_PC values in Table 3.5. Notice that 0 appears at the left and at the right because the final operation to obtain Y_1 is not executed until the beginning of the next iteration. By doing so, its latency is masked by the

Table 3.5: Instructions executed by the 1-MULT Koblitz ECP

MULT_PC	MULT	SA_PC	SA
	PDQA State		
0	$x \times (Z_1\|R)$	0	$Y_1 = T_1 + (Y_1\|M)$
		1	$T_2 = Z_1^2$
		2	$X_1 = X_1^{(2\|4)}$
1	$T_2 \times (y\|xy)$	0	$T_1 = X_1 + M$
		1	$Y_1 = Y_1^{(2\|4)}$
2	$T_1 \times Z_1$	0	$X_1 = Y_1 + M$
		1	$T_1 = T_1^2$
3		0	$Z_1 = 0 + M$
4	$X_1 \times R$	0	$Y_1 = Z_1 + a \cdot T_2$
		1	$Z_1 = Z_1^2$
5	$T_1 \times Y_1$	0	$T_2 = 0 + M$
		1	$X_1 = X_1^2$
6	$x \times Z_1$	0	$Y_1 = T_2 + M$
		1	$T_3 = Z_1^2$
		2	$X_1 = X_1 + Y_1$
7	$T_3 \times (xy\|y)$	0	$T_3 = X_1 + M$
		1	$T_2 = Z_1 + T_2$
8	$T_2 \times T_3$	0	$T_1 = 0 + M$
		1	$Z_1 = Z_1^{(2\|4)}$
	PQUAD State		
0		0	$Y_1 = T_1 + (Y_1\|M)$
		1	$Y_1 = R_1^{(4\|2)}$
		2	$X_1 = X_1^{(4\|2)}$
		3	$Z_1 = Z_1^{(4\|2)}$
	BX State		
0		0	$Y_1 = T_1 + Y_1\|R$
	ISQ State		
0		0	$R = (Z_1\|R)^{2^r}$
	IMULT State		
0	$(Z_1\|T_3) \times R$	0	
	IRED State		
0		0	$T_3 = 0 + M$
	FMULT State		
0	$X_1 \times R$	0	$T_3 = R^2$
1	$Y_1 \times T_3$	0	$T_1(x_3) = 0 + M$
	FINAL State		
0		0	$T_2(y_3) = 0 + M$

MULT block operation. The $X_1^{(2|4)}$, $Y_1^{(2|4)}$, and $Z_1^{(2|4)}$ operations shown in loops indexed 0, 1 and 8 in Figure 3.11 are PFBR operations for each coordinate that is executed in the PDQA state. The 'R' blocks correspond to reduction operations that are performed by adding $0 + M$ as shown in Table 3.5. The '+*' block corresponds to $Y_1 = Z_1 + a \cdot T_2$ when MULT_PC is 4 and SA_PC is 0. Since a is either 0 or 1, the multiplication is replaced with a conditional addition of T_2.

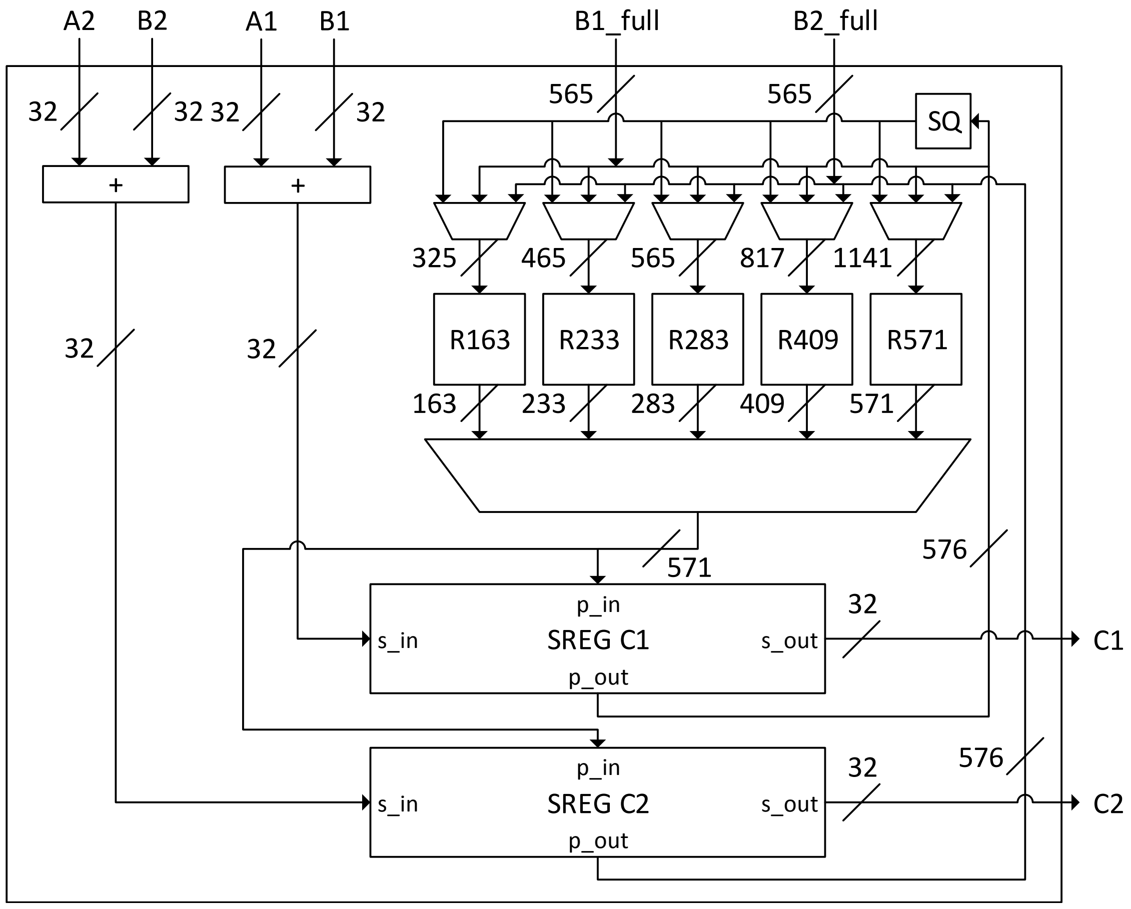


Figure 3.12: Block diagram of the revised SA block to support 2 MULT blocks.

To further reduce the latency of the described design, another ECP has been designed, where a second MULT block is instantiated to further parallelize the point operations. This ECP is referred to as '2-MULT Koblitz ECP' in this thesis. The design requires the SA block to replicate the 'SREG C' block to accommodate 2 MULT block results to be reduced simultaneously. The revised design of the SA block, shown in Figure 3.12, also uses an extra adder to allow for the extra MULT block result to be added to another argument before it is reduced. The revised SA block also has 2 outputs, 'C1' and 'C2' to output 2 reduced products simultaneously.

The block diagram of the 2-MULT Koblitz ECP is shown in Figure 3.13. The 2-MULT Koblitz ECP replicates the MULT block and uses the revised SA block in Figure 3.12 for reduction and addition. The 'C1' and 'C2' output ports of the SA port are connected back to the RAM and the byte-write feature of the Xilinx BRAM block allows for both reduced

Table 3.6: Instructions executed by the 2-MULT Koblitz ECP

MULT_PC	MULT_1	MULT_2	SA_PC	SA_1	SA_2
		PDQA State			
0	$M_1 = x \times (Z_1 \mid R_1)$		0	$Y_1 = (Y_1 \mid M_1) + (0 \mid T_1)$	
			1	$T_2 = Z_1^2$	
			2	$X_1 = X_1^{(2\mid 4)}$	
1			0	$R_1 = M_1 + X_1$	
2	$M_1 = R_1 \times Z_1$	$M_2 = T_2 \times (y \mid xy)$	0	$T_1 = R_1^2$	
			1	$Y_1 = Y_1^{(2\mid 4)}$	
3			0	$T_3 = M_1 + 0$	$X_1 = M_2 + Y_1$
			1	$Y_1 = R_1 + T_2 \cdot a$	
			2	$Z_1 = T_3^2$	
4	$M_1 = T_1 \times Y_1$	$M_2 = X_1 \times T_3$	0	$X_1 = X_1^2$	
			1	$T_3 = Z_1^2$	
5	$M_1 = x \times Z_1$	$M_2 = T_3 \times (xy \mid y)$	0	$R_1 = M_1 + 0$	$T_2 = M_2 + 0$
			1	$R_1 = R_1 + R_2$	
			2	$X_1 = R_1 + X_1$	
			3	$T_2 = Z_1 + T_2$	
6			0	$R_1 = M_1 + X_1$	$T_1 = M_2 + 0$
7	$M_1 = T_2 \times R_1$		0	$Z_1 = Z_1^{(2\mid 4)}$	
		PQUAD State			
0			0	$R_1 = (Y_1 \mid M_1) + (0 \mid T_1)$	
			1	$Y_1 = R_1^{(4\mid 2)}$	
			2	$X_1 = X_1^{(4\mid 2)}$	
			3	$Z_1 = Z_1^{(4\mid 2)}$	
		BX State			
0			0	$Y_1 = (Y_1 \mid M_1) + (0 \mid T_1)$	
		ISQ State			
0			0	$R_1 = (Z_1 \mid R_1)^{2^r}$	
		IMULT State			
0	$M_1 = R_1 \times (Z_1 \mid T_3)$		0		
		IRED State			
0			0	$T_3 = M_1 + 0$	
		FMULT State			
0			0	$R_1 = R_1^2$	
1	$M_1 = R_1 \times Y_1$	$M_2 = X_1 \times T_3$	0		
		FINAL State			
0			0	$T_2(y_3) = M_1 + 0$	$T_1(x_3) = M_2 + 0$

values to be written to the RAM simultaneously.

Using the data dependency graph of Figure 3.11, the grouping of the MULT and SA block operations can be modified to use 2 MULT blocks simultaneously. The sequence of instructions executed for each state of the 2-MULT Koblitz ECP can be found in Table 3.6.

One can notice that the use of 2 MULT blocks only reduces the number of groups of operations from 9 to 8. However, in Table 3.5, when MULT_PC is 3 the MULT block is not used, so the latency of the PDQA state is $7t_{\text{MULT}} + t_{\text{SA}}$. In comparison, in Table 3.6, when

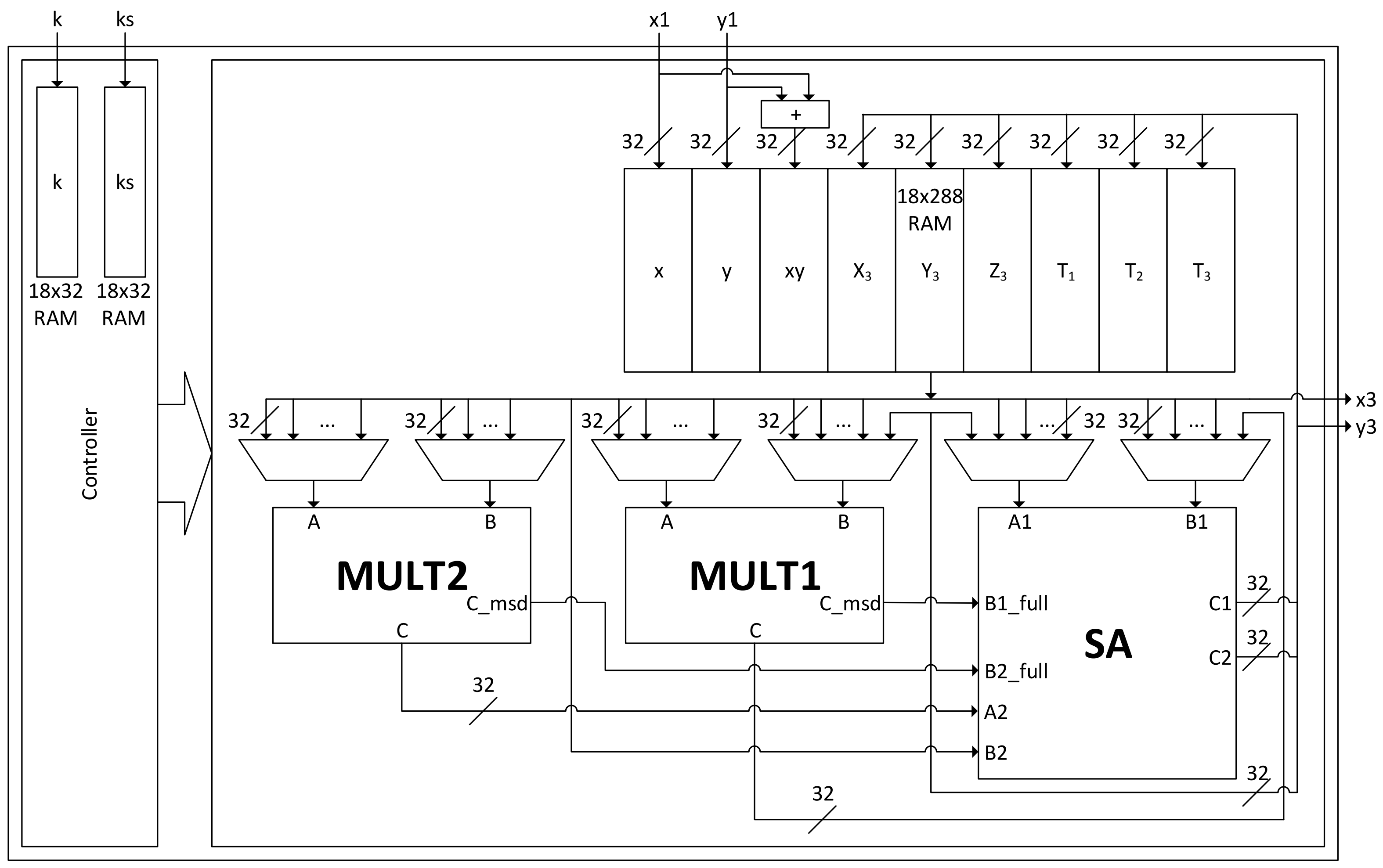


Figure 3.13: Block diagram of the 2-MULT Koblitz ECP.

Table 3.7: Implementation Results and Comparison of the Parallelized Scalable ECPs for Koblitz Curves

Work	FPGA	Registers	LUT	Slices	BRAM	Max. Freq. (MHz)	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
1-MULT Koblitz ECP	Virtex-5 XC5LX110T	1,704	7,073	2,199	5	223.46	163	0.068	6.669
							233	0.149	3.053
							283	0.215	2.112
							409	0.566	0.803
							571	1.391	0.327
2-MULT Koblitz ECP	Virtex-5 XC5LX110T	3,134	8,609	2,708	5	222.67	163	0.055	6.760
							233	0.114	3.228
							283	0.163	2.267
							409	0.409	0.903
							571	0.973	0.380
Koblitz ECP	Virtex-5 XC5LX110T	1,401	3,003	1,246	8	206.27	163	0.206	3.903
							233	0.455	1.764
							283	0.554	1.449
							409	1.451	0.553
							571	3.266	0.246

MULT_PC is 1, 3 or 6, neither MULT block is in use, so the latency is $5t_{\text{MULT}} + 5t_{\text{SA}}$, which is lower than the 1-MULT case.

Table 3.7 shows the implementation results of the above mentioned parallelized Koblitz curve scalable ECPs on the Virtex-5 FPGA. The table also shows the results of the Koblitz ECP described in Section 3.2.1 implemented on the same Virtex-5 FPGA. From Table 3.7, one can see that the 1-MULT Koblitz ECP has a much lower latency than the Koblitz ECP due to the use of a separate SA block to perform reduction. Even though the hardware resource utilization of the 1-MULT Koblitz ECP is higher, the efficiency metric shows that it is more efficient than the Koblitz ECP. Table 3.7 also shows the comparison between the 1-MULT Koblitz ECP and the 2-MULT Koblitz ECP. As expected, the 2-MULT Koblitz ECP requires more registers (3,134 compared to 1,704) and slices (2,708 compared to 2,199), but it is able to further reduce the latency of the ECPM. Using the efficiency metric, one can see that the 2-MULT Koblitz ECP slightly outperforms the 1-MULT counterpart.

3.3.2 Parallelization of Scalable ECP over Pseudo-Random Curves

Similar to the parallelized Koblitz ECP implementations described in Section 3.3.1, the parallelized pseudo-random ECPs use 1 and 2 MULT blocks, respectively. These are referred

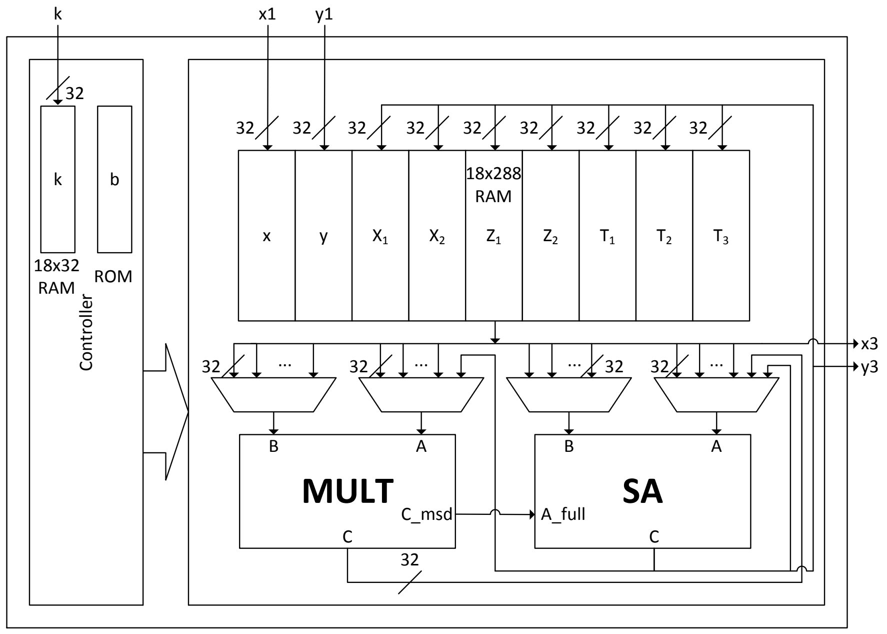


Figure 3.14: Block diagram of the 1-MULT Random ECP.

to as '1-MULT Random ECP' and '2-MULT Random ECP' in this thesis. The difference between the Koblitz curve implementation and the pseudo-random curve implementation is in the design of the finite state machine (FSM) in the controller and the instructions that are operated.

The block diagram of the 1-MULT Random ECP is shown in Figure 3.14. It highly resembles the block diagram shown in Figure 3.9, with the exception that the controller does not require ks, since the value of k is represented in binary and the temporary values in the RAM are different.

The FSM and the operations executed by the 1-MULT Random ECP are shown in Figure 3.15 and Table 3.8, respectively. From Figure 3.15, it can be seen that the FSM is very similar to its Koblitz counterpart shown in Figure 3.10. The differences are in the states that evaluate the main loop of the ECPM. The IDLE, LOAD, FINAL, WAIT and inversion states are the same as in the Koblitz ECPs. As in the Koblitz ECP implementations, the data dependency graph is used to optimize the ECPM operations. The data dependency graph of the LOOP state is shown in Figure 3.16. The LOOP state executes the *Madd* and *Mdouble* operations shown in Equations (2.13) and (2.14). As in Figure 3.11, the numbers in the dashed-line loops correspond to the MULT_PC values in Table 3.8. Notice that in

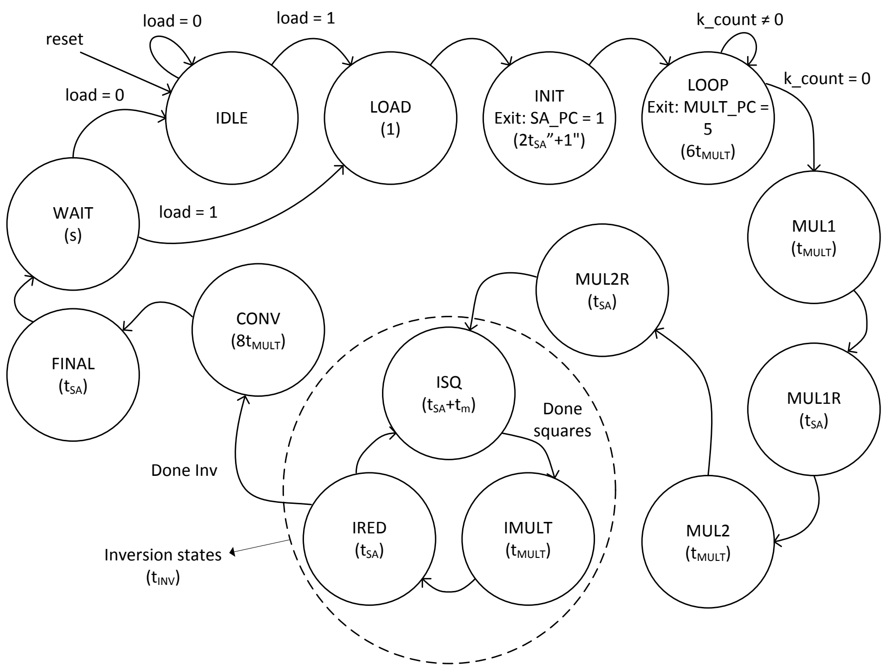


Figure 3.15: FSM of the 1-MULT Random ECP.

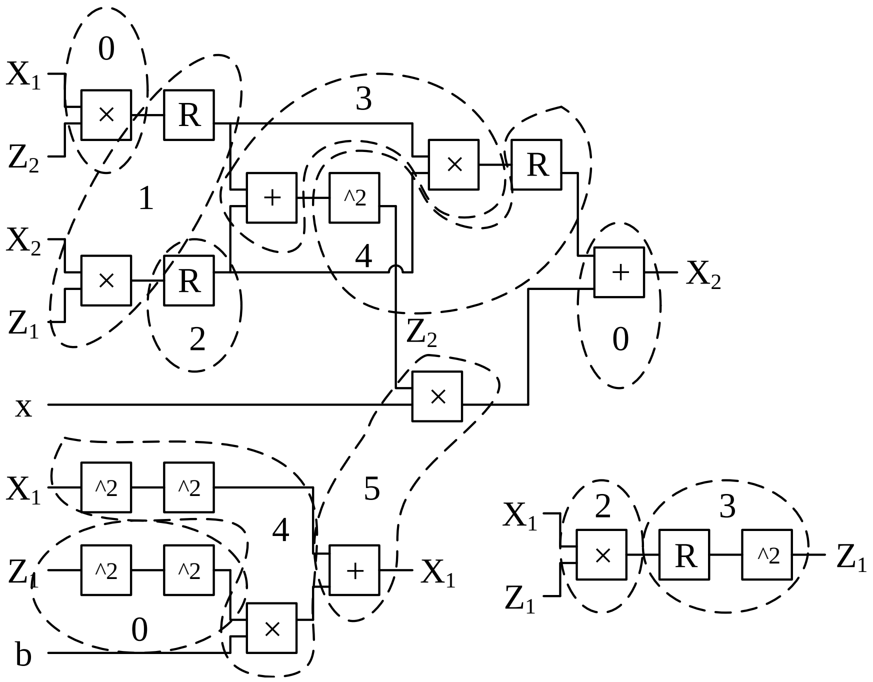


Figure 3.16: Data dependency graph of LOOP for the 1-MULT Random ECP.

Table 3.8 some of the operations select between different X_1 and X_2 or Z_1 and Z_2. These are conditionally selected based on the bit of k to perform the **if** statement in the Lopez-Dahab algorithm in Algorithm 2.2.

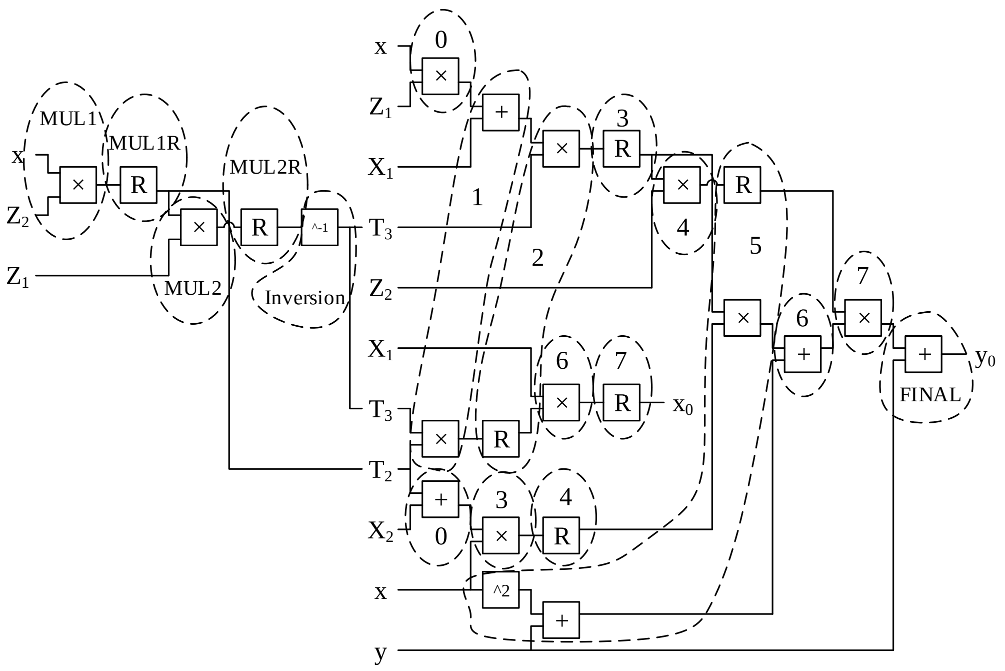


Figure 3.17: Data dependency graph of coordinate conversion for the 1-MULT Random ECP.

One of the improvements made on the 1-MULT Random ECP over the Random ECP discussed in Section 3.2.2 is the projective to affine coordinate conversion. The improvement modifies the *Mxy* operation shown in Algorithm 2.2 into the following:

$$\begin{aligned} x_0 &\leftarrow \tfrac{xZ_2X_1}{xZ_1Z_2} \\ y_0 &\leftarrow (\tfrac{Z_2(xZ_1+X_1)}{xZ_1Z_2})(\tfrac{x(xZ_1+X_1)(xZ_2+X_2)}{xZ_1Z_2} + x^2 + y) + y \end{aligned} \tag{3.6}$$

By doing so, only 1 finite field inversion is needed, for xZ_1Z_2, to perform the conversion as opposed to 3 individual inversions for x, Z_1 and Z_2. Since the latency of the inversion is much higher than that of multiplication, the conversion algorithm in Equation (3.6) requires fewer clock cycles than in Algorithm 2.2. Using the data dependency graph shown in Figure 3.17 the operations of the coordinate conversion are optimized for the MULT and SA block.

The states MUL1, MUL1R, MUL2 and MUL2R are used to prepare the value to be inverted, which is xZ_1Z_2. The product of $x \times Z_2$ is evaluated first and stored in the temporary variable T_2 to be used during the CONV state. The inversion of xZ_1Z_2 is evaluated during the inversion states and the operations of the CONV state are grouped by the MULT_PC values shown in Figure 3.17.

Table 3.8: Instructions executed by the 1-MULT Random ECP

MULT_PC	MULT	SA_PC	SA
		INIT State	
0		0	$Z_2 = x^2$
		1	$R = R^2$
		LOOP State	
0	$(X_1\|X_2) \times (Z_2\|Z_1)$	0	$(X_2\|X_1) = (M\|R) + (T_2\|b)$
		1	$T_3 = (Z_1\|Z_2)^4$
1	$(X_2\|X_1) \times (Z_1\|Z_2)$	0	$T_2 = M + 0$
2	$(X_1\|X_2) \times (Z_1\|Z_2)$	0	$T_1 = M + 0$
3	$T_2 \times T_1$	0	$R = M + 0$
		1	$(Z_1\|Z_2) = R^2$
		2	$T_1 = T_1 + T_2$
4	$b \times T_3$	0	$T_2 = M + 0$
		1	$(Z_2\|Z_1) = T_1^2$
		2	$T_3 = (X_1\|X_2)^4$
5	$x \times (Z_2\|Z_1)$	0	$(X_1\|X_2) = M + T3$
		MUL1 State	
0	$x \times Z_2$	0	$(X_2\|X_1) = M + T_2$
		MUL1R State	
0		0	$T_2 = M + 0$
		MUL2 State	
0	$R \times Z_1$	0	
		MUL2R State	
0		0	$T_1 = M + 0$
		ISQ State	
0		0	$R = R^{2^r}$
		IMULT State	
0	$R \times (T_1\|T_3)$	0	
		IRED State	
0		0	$T_3 = M + 0$
		CONV State	
0	$x \times Z_1$	0	$T_3 = R^2$
		1	$T_1 = X_2 + T_2$
1	$T_2 \times T_3$	0	$T_2 = M + X_1$
2	$T_2 \times T_3$	0	$Z_1 = M + 0$
3	$x \times T_1$	0	$T_1 = M + 0$
4	$T_1 \times Z_2$	0	$T_3 = M + 0$
5	$T_1 \times T_3$	0	$T_1 = M + 0$
		1	$R = x^2$
		2	$T_3 = R + y$
6	$X_1 \times Z_1$	0	$T_2 = M + T_3$
7	$T_2 \times T_1$	0	$T_1(x_3) = M + 0$
		FINAL State	
0		0	$R(y_3) = M + y$

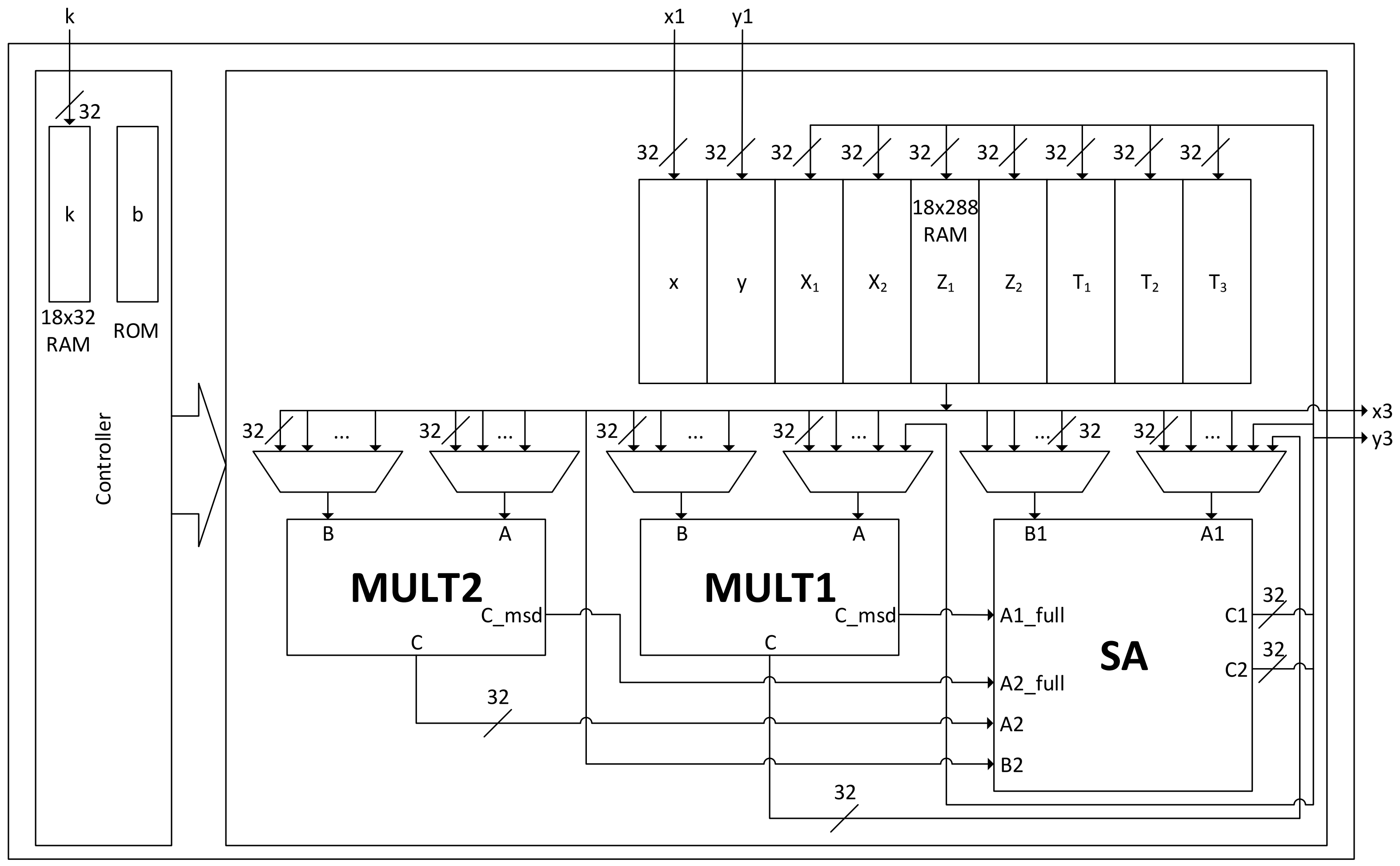


Figure 3.18: Block diagram of the 2-MULT Random ECP.

Table 3.9: Instructions executed by the 2-MULT Random ECP

MULT_PC	MULT_1	MULT_2	SA_PC	SA_1	SA_2
		INIT State			
0			0	$Z_2 = (x+0)^2$	
			1	$R_1 = (R_1+0)^2$	
			2	$X_2 = R_1 + b$	
		LOOP State			
0	$M_1 = (X_1\|R_1) \times Z_2$	$M_2 = (X_2\|R_1) \times Z_1$	0	$T_3 = (Z_1\|Z_2 + 0)^4$	
1	$M_1 = (X_1\|X_2) \times (Z_1\|Z_2)$	$M_2 = T_3 \times b$	0	$T_1 = M_1 + 0$	$T_2 = M_2 + 0$
			1	$(Z_2\|Z_1) = (R_1 + R_2)^2$	
			2	$T_3 = (0 + (X_1\|X_2))^4$	
2	$M_1 = x \times (Z_2\|Z_1)$	$M_2 = T_2 \times T_1$	0	$R_1 = M_1 + 0$	$(X_1\|X_2) = M_2 + T_3$
			1	$(Z_1\|Z_2) = (R_1+0)^2$	
		MUL1 State			
0	$M_1 = x \times Z_2$		0	$R_1 = M_1 + 0$	$R_2 = M_2 + 0$
			1	$(X_2\|X_1) = R_1 + R_2$	
		MUL1R State			
0	$M_1 = x \times Z_1$		0	$T_2 = M_1 + 0$	
		MUL2 State			
0	$M_1 = T_2 \times Z_1$		0	$Z_1 = M_1 + X_1$	
			1	$X_2 = T_2 + X_2$	
		MUL2R State			
0			0	$T_1 = M_1 + 0$	
		ISQ State			
0			0	$(R_1\|T_3) = (R_1+0)^{2^r}$	
		IMULT State			
0	$M_1 = R_1 \times (T_1\|T_3)$		0		
		IRED State			
0			0	$T_3 = M_1 + 0$	
		CONV State			
0	$M_1 = x \times X_2$	$M_2 = R_1 \times Z_1$	0		
1	$M_1 = T_2 \times T_3$		0	$T_3 = M_1 + 0$	$T_2 = M_2 + 0$
			1	$T_1 = (x+0)^2$	
2	$M_1 = T_2 \times Z_2$	$M_2 = T_2 \times T_3$	0	$T_2 = M_1 + 0$	
			1	$T_3 = T_1 + y$	
3			0	$R_1 = M_1 + 0$	$R_2 = M_2 + T_3$
4	$M_1 = R_1 \times R_2$	$M_2 = T_2 \times X_1$	0		
		FINAL State			
0			0	$T_1(y_3) = M_1 + y$	$T_2(x_3) = M_2 + 0$

The 2-MULT Random ECP is also designed in a similar fashion as its Koblitz curve counterpart. The block diagram of the 2-MULT Random ECP is shown in Figure 3.18. Similar to the 2-MULT Koblitz ECP, the 2-MULT Random ECP also uses the revised SA block shown in Figure 3.12 for reduction and addition. Due to the data dependency in the Lopez-Dahab algorithm used in pseudo-random curves, the 6 MULT block operations in the

Table 3.10: Implementation Results and Comparison of the Parallelized Scalable ECPs for Pseudo-Random Curves

Work	FPGA	Registers	LUT	Slices	BRAM	Max. Freq. (MHz)	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
1-MULT Random ECP	Virtex-5 XC5LX110T	1,650	7,128	2,290	5	224.84	163	0.135	3.246
							233	0.299	1.460
							283	0.440	0.993
							409	1.186	0.368
							571	2.965	0.147
2-MULT Random ECP	Virtex-5 XC5LX110T	3,118	8,784	2,708	5	223.26	163	0.080	4.626
							233	0.172	2.148
							283	0.250	1.479
							409	0.652	0.567
							571	1.593	0.232
Random ECP	Virtex-5 XC5LX110T	1,225	3,191	1,150	5	181.19	163	0.380	2.290
							233	0.860	1.011
							283	1.105	0.787
							409	3.037	0.286
							571	7.243	0.120

LOOP state can all be parallelized, so in the 2-MULT Random ECP, the LOOP state only requires $3t_{\text{MULT}}$ instead of $6t_{\text{MULT}}$. This modification reduces the LOOP state latency by 50% with only a slight increase in hardware utilization. The operations executed in the 2-MULT Random ECP are shown in Table 3.9.

The implementation results of the above mentioned ECPs are shown in Table 3.10. The Virtex-5 implementation result of the Random ECP described in Section 3.2.2 is also included for comparison purposes. As expected, Table 3.10 shows that the 1-MULT Random ECP is more efficient than the Random ECP, as it parallelizes the multiplication and reduction operations. Table 3.10 also shows the improvement of the 2-MULT Random ECP over the 1-MULT counterpart. Comparing the results in Table 3.7 and in Table 3.10, one can see that the parallelization to 2 MULT blocks is more efficient for pseudo-random curves than for Koblitz curves. From Table 3.10, it can be seen that the efficiency improvement of the 2-MULT Random ECP is between 43% and 57%, whereas the improvement of the 2-MULT Koblitz ECP is only between 1.4% to 16%.

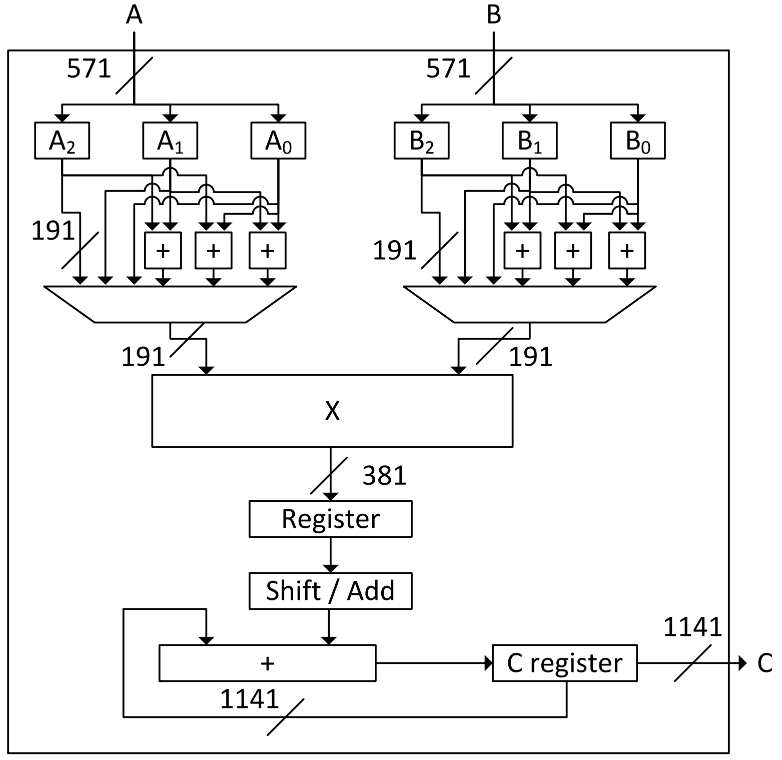
3.4 Low Latency Scalable ECC Processor for Binary Curves[5]

Despite the lower latency of the ECPs described in Section 3.3 compared to the ones in Section 3.2, the former ECPs have one drawback. The implementation of the Comba algorithm for multiplication in the MULT block results in a latency in the order of $O(s^2)$, where s is the number of 32-bit digits. Thus, for large key sizes, the latency increases quadratically with respect to the number of 32-bit digits. The ECPs presented in this section redesigns the MULT and SA blocks to improve the latency of the ECPM operation on high bit lengths. The work described in this section is published in [53] and are referred to as 'Low-Latency Koblitz ECP' and 'Low-Latency Random ECP' in this thesis. Collectively, the 2 ECPs are referred to as 'Low-Latency ECPs'.

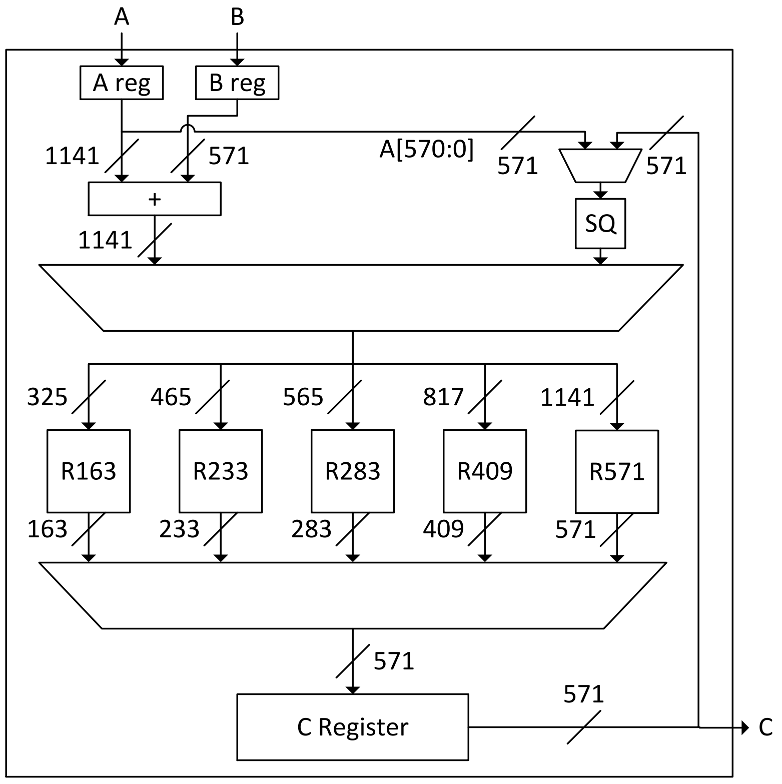
The MULT block of the Low-Latency ECPs is shown in Figure 3.19(a). It uses the Karatsuba-Ofman multiplication algorithm described in Equation (2.22). However, instead of implementing a 571-bit parallel multiplier, the MULT block of the Low-Latency ECPs divides the 571-bit operand into 3 191-bit parts and evaluates each 191-bit multiplication on every clock cycle. The 191-bit multiplier is in turn a combinational Karatsuba-Ofman multiplier applied recursively, with 2 levels of pipelining built into it. Thus, each 571-bit multiplication requires only 9 clock cycles. Since operands with fewer bits can be evaluated with the same hardware by setting the MSBs to 0, the multiplier can support all multiplications up to 571 bits. The result of each partial multiplication is accumulated in the 'C register' and output as a single wide bus from the MULT block through port 'C'.

The SA block, as shown in Figure 3.19(b), has been modified from the version in Figure 3.7(b) to use a full length 1141-bit port 'A' for the result of the MULT block and a 571-bit 'B' port for other operands. The reduction blocks in Figure 3.19(b) are the same as the ones used in Figure 3.7(b). Since the complete operand is available at the input of the SA block, the latency of the SA block is 2 clock cycles regardless of the key size. Since the t_{MULT}:t_{SA} ratio is 9:2 = 4.5, up to 4 SA block operations may execute in parallel with a

[5]The work in this section has been publihsed in *ISCAS 2014* [53].



(a) Block diagram of the MULT block.



(b) Block diagram of the SA block.

Figure 3.19: Block diagram of the Finite Field Arithmetic Blocks in the Low-Latency ECPs.

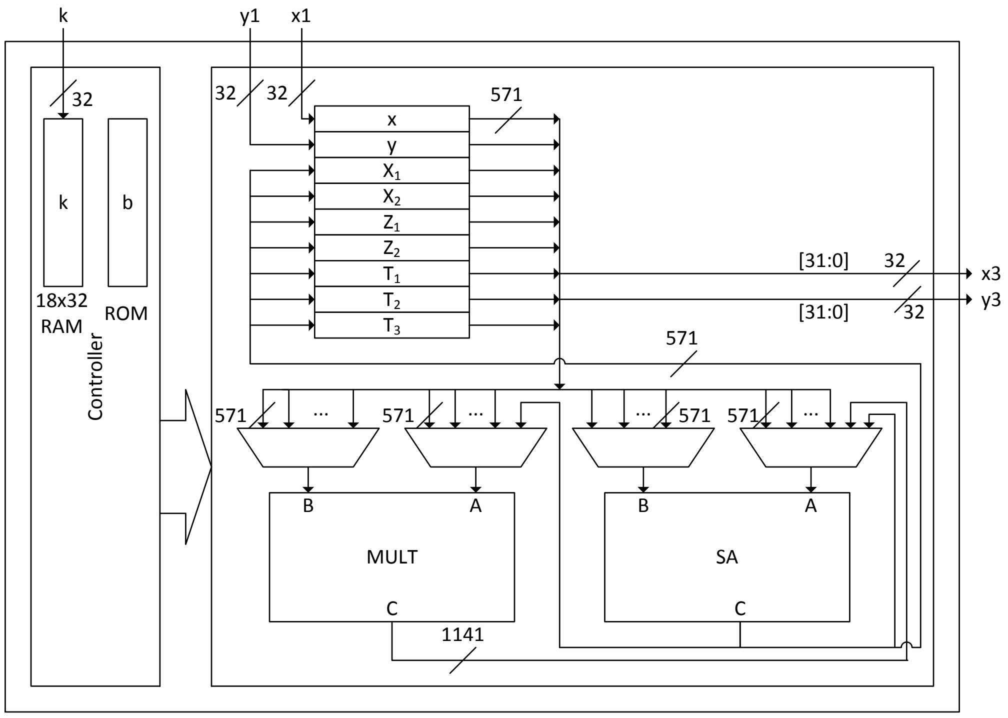


Figure 3.20: Block diagram of the Low-Latency Random ECP.

MULT block operation. Thus, the algorithms shown in Table 3.5 and Table 3.8 can be used without further modifications.

The block diagram of the Low-Latency Random ECP is shown in Figure 3.20. The biggest difference between Figure 3.20 and the ECPs discussed in Section 3.3 is that 571-bit registers are used in Figure 3.20 instead of a RAM. This is due to output of the SA block being a 571-bit port instead of 32-bit digits. The block diagram of the Low-Latency Koblitz ECP is similar with the exception that it has an additional input for the sign bit of the scalar multiplier k of the ECPM. This input, 'ks', is connected to the Controller shown in Figure 3.20. In addition, the ROM for the constant b is not necessary in the Low-Latency Koblitz ECP.

The implementation results of the Low-Latency ECPs are shown in Table 3.11. The implementation of these ECPs requires a much higher number of hardware resources since it implements much wider multipliers. However, it reduces the latency of the ECPM due to the reduced latency of the MULT block using the Karatsuba-Ofman algorithm for multiplication. Using the efficiency metric and comparing the results in Table 3.7 and Table 3.10 with Table 3.11, one can see that the Low Latency ECPs are less efficient than the 2-MULT ECPs

Table 3.11: Implementation Results of the Low-Latency ECPs

Work	FPGA	Registers	LUT	Slices	BRAM	Max. Freq. (MHz)	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
Low-Latency Koblitz ECP	Virtex-5 XC5LX110T	13,076	26,111	7,427	0	162.07	163	0.029	4.599
							233	0.042	3.213
							283	0.050	2.667
							409	0.073	1.855
							571	0.101	1.331
Low-Latency Random ECP	Virtex-5 XC5LX110T	12,983	24,974	7,978	0	154.35	163	0.059	2.119
							233	0.084	1.489
							283	0.102	1.228
							409	0.147	0.852
							571	0.205	0.611

for lower bit lengths (i.e. 163 and 233 in Koblitz; 163, 233 and 283 in pseudo-random). For higher bit lengths, the Low-Latency ECPs have a higher efficiency. This result is expected since the latency of the implemented Karatsuba-Ofman multiplier is constant instead of increasing quadratically with the number of 32-bit digits. The lower efficiency at lower bit lengths is a results of a slower maximum clock frequency due to the increased complexity of the 191-bit multiplier. Thus, the advantage of the reduced latency is more apparent in higher bit lengths than in lower bit lengths.

Unlike the ECPs described in Section 3.3, the parallelization of the MULT block in the Low-Latency ECPs would not be feasible because the hardware resource utilization of the MULT block is relatively high compared to the 32-bit MULT block shown in Figure 3.7(a). Thus, the implementation of a 2-MULT ECP using Figure 3.19(a) would reduce the latency, but increase the number of slices such that the efficiency of the overall ECP would decrease.

Chapter 4

Scalable ECC Processor for Prime Curves[1]

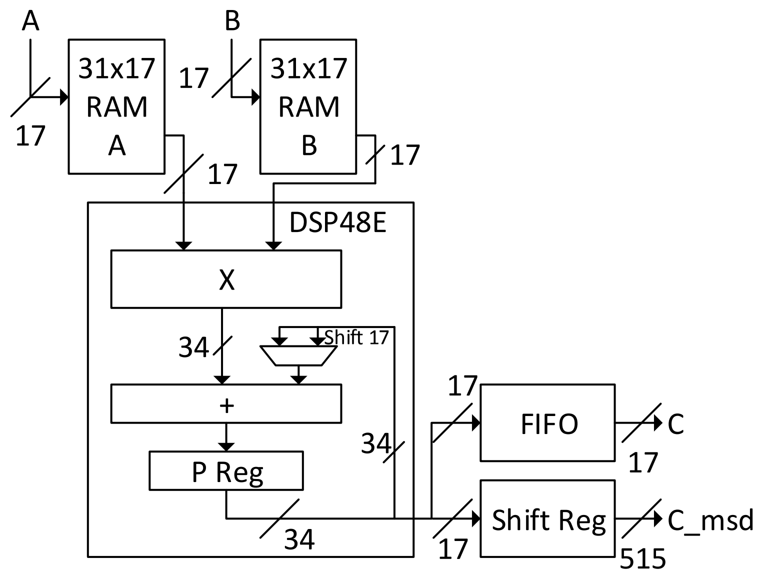
In the previous chapter, only the binary field curves recommended by NIST have been discussed. In this chapter, the design and implementation of a scalable ECP for the 5 NIST recommended prime curves are discussed. The work described in this chapter is available in [54] and is referred to as 'Prime ECP' in this thesis. As mentioned in Section 2.3, the main difference between prime field and binary field arithmetic is the carry propagation of addition, which influences all other arithmetic operations. This chapter is divided into 2 sections: Section 4.1 presents the design and architecture of the Prime ECP; Section 4.2 shows the implementation results and the comparison with another scalable prime ECP in the current literature.

4.1 Design and Architecture

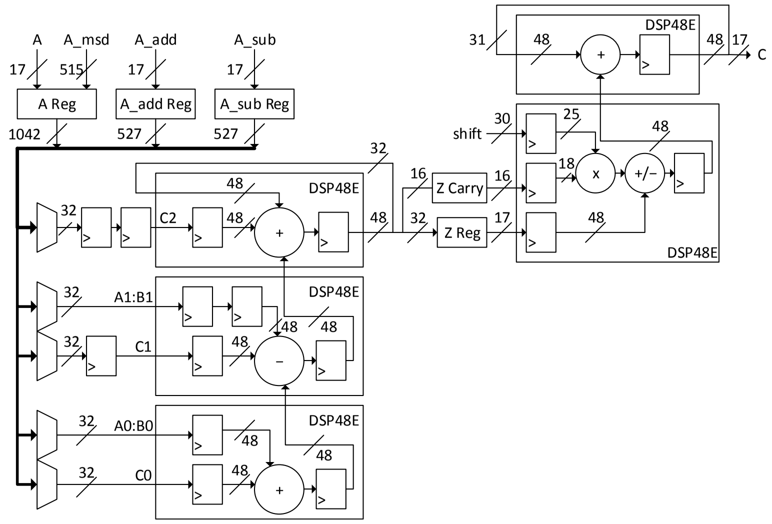
From the lessons learned from the binary ECPs, the Prime ECP also parallelizes the multiplication and the reduction steps. Since the squaring operation cannot be simplified to interleaving zeros as in the binary case, squaring is performed as a multiplication of identical operands. Thus, the finite field arithmetic blocks of the Prime ECP consist of the MULT and the addition/subtraction/reduction (AR) block.

The block diagram of the MULT and AR blocks are shown in Figure 4.1. Notice that the architecture of both the MUTL and AR blocks are built with the DSP48E slices described in Section 2.4 because they have built-in 25×18-bit hardware multipliers that can be used instead of designing multipliers using FPGA fabric that generally have lower performance.

[1]The work in this chapter has been accepted to *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* [54].



(a) Block diagram of the MULT block.



(b) Block diagram of the AR block.

Figure 4.1: Block diagram of the Finite Field Arithmetic Blocks in the Prime ECP.

The MULT block implements the Comba algorithm (Algorithm 2.4) using 17-bit digits. The reason for choosing 17-bit digits instead of 32-bit digits is to take advantage of the 17-bit shifted feedback input available in the DSP48E slices. As can be seen in Figure 4.1(a), the MULT block inputs the operands 'A' and 'B' as 17-bit digits and stores the values in 31×17-bit RAMs. The RAM values are read into the multiplier of the DSP48E slice according to the indexes in the Comba algorithm. Since the multiplier in the DSP48E slices is a 25×18 bits, the 17-bit operands are zero padded. Inside the DSP48E slice, the product of the multiplier is accumulated in the internal 'P' register. The 17-bit shifted value is chosen when an inner loop of the Comba algorithm is completed. Simultaneously, the lower 17 bits of the product are shifted into the FIFO or the shift register. The FIFO and the shift register operate in a similar fashion as the ones in Figure 3.7(a).

As discussed in Section 2.3.2, due to the choice of the prime numbers, the reduction operation can be simplified to a series of modular additions and subtractions. Thus, the addition and subtraction operations are built into the architecture of the reduction operation forming the AR block shown in Figure 4.1(b). Since the reduction algorithms for the NIST recommended prime numbers access the operand in 32-bit digits, the AR block collects the input values as 17-bit operands and stores them in registers. 'A Reg' is a 1042-bit register to accommodate the product of 2 521-bit values, and 'A_add Reg' and 'A_sub Reg' are 527-bit ($\lceil 521/17 \rceil \times 17 = 527$) registers that store the operands to be added and subtracted. Thus, the AR block performs the operation 'A Reg' + 'A_add Reg' − 'A_sub Reg' $\pmod{p}$.

The AR block uses 3 DSP48E slices to perform the first stage of addition, subtraction and reduction. These DSP48E slices are cascaded such that they compute the operation: [A0:B0] + C0 + C2 − ([A1:B1] + C1), where the operands correspond to the input ports of the DSP48E slice. Each operand is a 32-bit digit extracted from the input registers and zero-padded to 48 bits. Notice that the multiplier of the DSP48E slice is bypassed, so the DSP48E slice is used as a 48-bit arithmetic block. The multiplexers on the left of Figure 4.1(b) are used to select these 32-bit digits according to the reduction algorithm of the specific prime number. For example, consider the case of the 192-bit prime number. The sequence of digits selected for each input is shown in Table 4.1.

The operation of the AR block is separated into digits and passes in Table 4.1. The digit

Table 4.1: Operation sequence for the addition/subtraction/reduction (AR) block for p_{192}

Digit	0		1		2		3		4		5	
Pass	0	1	0	1	0	1	0	1	0	1	0	1
C0	add_0	0	add_0	0	add_0	a_6	add_0	a_6	add_0	a_4	add_0	a_4
A1:B1	sub_0	0	sub_0	0	sub_0	0	sub_0	0	sub_0	0	sub_0	0
C1	0	0	0	0	0	0	0	0	0	0	0	0
A2:B2	a_0	a_{10}	a_0	a_{10}	a_0	a_8	a_0	a_8	a_0	a_6	a_0	a_6
C2	a_6	0	a_6	0	a_4	0	a_4	0	0	0	0	0

column indicates the index of the output digit that is being computed. The pass column indicates the sequence of values input to the 3 DSP48E slices. In Table 4.1, during the 0th pass of the 0th digit, the 0th digit of 'A_add Reg' (add_0) is selected for C0, the 0th digit of 'A_sub Reg' (sub_0) is selected for [A1:B1], 0 is input into C1, the 0th digit of 'A Reg' (a_0) for [A2:B2] and the 6th digit (a_6) for C2. Once the 0th and 1st passes are complete for digit 0, the input registers are shifted by 32 bits, such that add_1 becomes add_0, sub_1 becomes sub_0, a_1 becomes a_0, etc. By doing so, when evaluating the result of digit 1, the multiplexer selects add_0 and sub_0 again for C0 and [A1:B1], respectively. Using this method, the size of the multiplexers is reduced.

The resultant digits of the reduction are accumulated in 'Z Reg' to be used by the second stage of the AR block, which performs the final $\pmod p$ of the reduction algorithm and converts the result back to 17-bit digits. 'Z Reg' is a 544-bit ($\lceil 521/32 \rceil \times 32 = 544$) register. At the completion of the series of additions and subtractions at the first half of the AR block, at most 1 extra 17-bit digit can be produced (stored in 'Z Carry'). Thus, the second stage of the AR block performs the reduction of the 1 extra digit. Since 'Z Reg' outputs 17-bit digits, the 'shift' input is a 1-hot value (i.e. only 1 bit is asserted in the number) that is multiplied to 'Z Carry' to shift the extra digit accordingly for reduction. The top-right DSP48E unit in Figure 4.1(b) is used to handle carryout bits from each digit to be carried to the next digit. The result of the AR block is output through port 'C' as 17-bit digits.

The block digram of the Prime ECP is shown in Figure 4.2. The architecture of the ECP

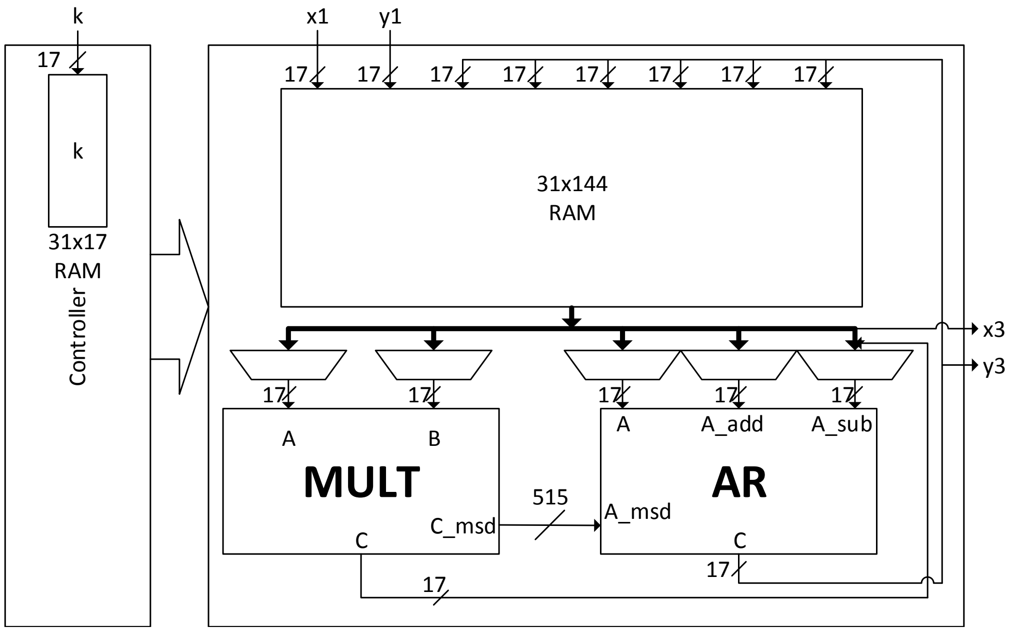


Figure 4.2: Block diagram of the Prime ECP.

Table 4.2: Latency in number of clock cycles of the MULT and AR blocks in the Prime ECP.

m	t_{MULT}	t_{SA}	Ratio (t_{MULT}:t_{SA})
163	145	32	4.53
233	197	36	5.47
283	257	48	5.35
409	530	67	7.91
571	962	56	17.2

is very similar to the ones presented in Section 3.3, except the RAM handles 17-bit digits instead of 32-bit digits. The RAM is $8 \times 18 = 144$ bits wide instead of $8 \times 17 = 136$ bits because of the use of the byte-writing capability on the Xilinx Block RAMs (BRAMs).

The latency in clock cycles of the MULT and AR blocks is shown in Table 4.2. As in the binary case, the ratio of t_{MULT}:t_{SA} shows that up to 4 AR operations may be executed simultaneously for each MULT operation. The double-and-add ECPM algorithm (Algorithm 2.1) and Jacobian projective coordinates are used in the implementation of the prime ECP. Thus, the data dependency graph is drawn for the PDBL and PADD operations shown in Equation (2.16) and (2.15), respectively, to optimize the usage of the MULT and AR blocks. The data

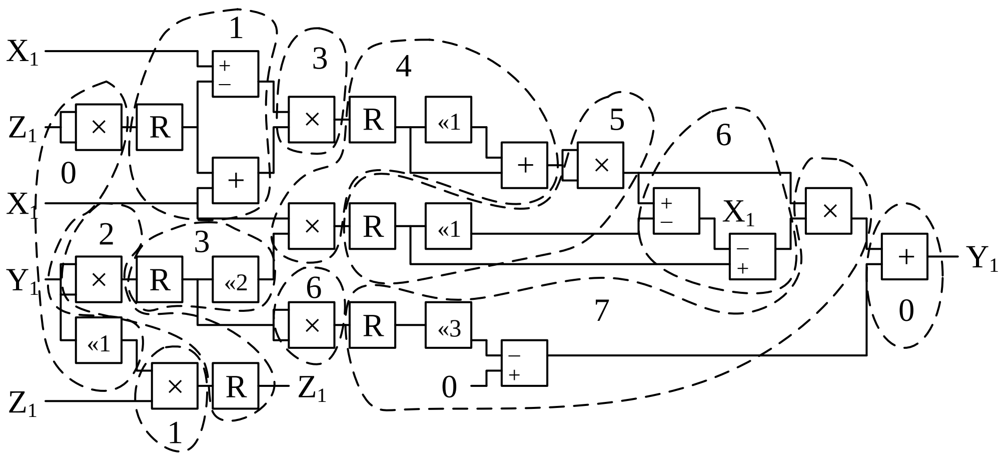


Figure 4.3: Data dependency graph of PDBL of the Prime ECP.

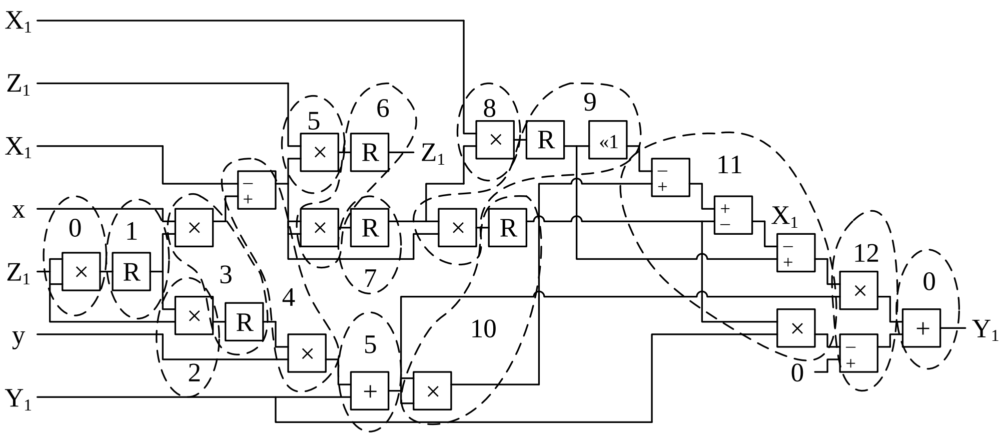


Figure 4.4: Data dependency graph of PADD of the Prime ECP.

dependency graph of the PDBL and PADD states are shown in Figure 4.3 and Figure 4.4, respectively.

In the data dependency graphs, '$\ll c$' signifies the modular left shift by c, or $\times 2^c \pmod{p}$. This operation can be very easily integrated into the AR block by using the 'A_msd' input for the shifted MSBs. Using the data dependency graph, the FSM and the instructions for each state are shown in Figure 4.5 and Table 4.3.

Notice in Table 4.3 that the evaluation of the Y_1 in each iteration is performed in the beginning of the next iteration. This is the same technique used in the 1-MULT Koblitz ECP described in Section 3.3.1. Furthermore, in the PDBL state when MULT_PC is 6 and AR_PC is 2, the operation $T_3 = P_1 \ll 1 + 0 - 1$ is evaluated but does not appear in the data dependency graph in Figure 4.3. The same expression appears in the PADD state when

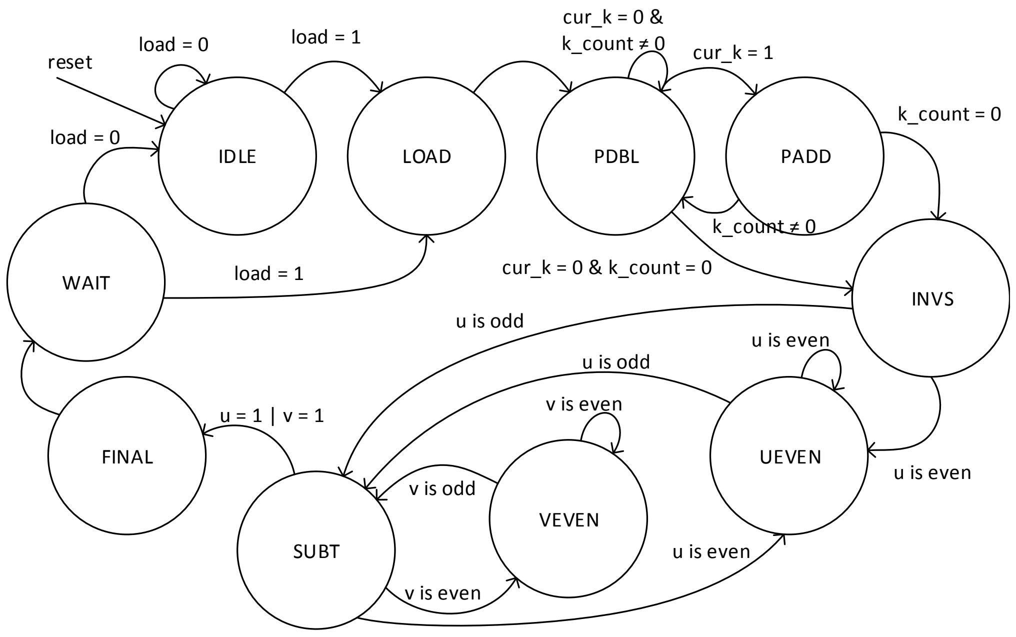


Figure 4.5: FSM of the Prime ECP.

MULT_PC is 12 and AR_PC is 2. This operation is used to set up for the inversion used in the projective to affine coordinate conversion.

The instructions executed for inversion are shown in Table 4.4. In the Prime ECP, inversion is performed using the binary inversion algorithm (Algorithm 2.6). In Table 4.4, the INVS state completes the evaluation of Y_1 for the final iteration. There are 3 states used for inversion. The Z1EVEN state corresponds to the **while** loop where u is even in Algorithm 2.6. The T3EVEN state corresponds to the **while** loop where v is even, and the SUBT state corresponds to the **if** statement for $u \geq v$. The naming of the states is based on the names of the registers used in the ECP. In other words, the value of u in Algorithm 2.6 is stored in Z_1 during inversion and v is stored in T_3.

In Table 4.4, '$\gg 1$' signifies a right-shift by 1 bit or a modular division by 2. The right-shift operation can be very easily handled by the AR block because the division by 2 of an even number does not require the reduction step. During the Z1EVEN and T3EVEN states, the operation is slightly modified from the expressions shown in Algorithm 2.6. Consider the Z1EVEN state, the **if** statement in the **while** loop evaluates $x_1 \leftarrow x_1/2$ if x_1 is even, and $x_1 \leftarrow (x_1 + p)/2$, otherwise. In Table 4.4, T_1 corresponds to x_1 in Algorithm 2.6. In the inversion implemented in the prime ECP, the expression $(x_1 + p)/2$ is modified to

Table 4.3: Instructions executed during PDBL and PADD states by the Prime ECP

MULT_PC	MULT	AR_PC	AR
		PDBL State	
0	$Z_1 \times Z_1$	0	$Y_1 = M\|0 + Y_1 - 0$
		1	$T_3 = R \ll 1 + 0 - 0$
1	$T_3 \times Z_1$	0	$T_1 = M + 0 - 0$
		1	$T_2 = 0 + X_1 - R$
		2	$T_1 = T_1 + X_1 - 0$
2	$Y_1 \times Y_1$	0	$Z_1 = M + 0 - 0$
3	$T_1 \times T_2$	0	$T_1 = M + 0 - 0$
		1	$T_3 = R \ll 2 + 0 - 0$
4	$T_3 \times X_1$	0	$R = M + 0 - 0$
		1	$T_2 = R \ll 1 + R - 0$
5	$T_2 \times T_2$	0	$T_3 = M + 0 - 0$
		1	$X1 = R \ll 1 + 0 - 0$
6	$T_1 \times T_1$	0	$X_1 = M + 0 - X_1$
		1	$T_1 = 0 + T_3 - R$
		2	$T_3 = P_1 \ll 1 + 0 - 1$
7	$T_1 \times T_2$	0	$R = M + 0 - 0$
		1	$R = R \ll 3 + 0 - 0$
		2	$T_1 = 0 + 0 - R$
		PADD State	
0	$Z_1 \times Z_1$	0	$Y_1 = M + Y_1 - 0$
1		1	$T_1 = M + 0 - 0$
2	$Z_1 \times R$	0	
3	$T_1 \times x$	0	$T_2 = M + 0 - 0$
4	$T_2 \times y$	0	$T_1 = M + 0 - X1$
5	$T_1 \times Z_1$	0	$T_2 = M + 0 - Y_1$
6	$T_1 \times T_1$	0	$Z_1 = M + 0 - 0$
7		0	$T_3 = M + 0 - 0$
8	$X_1 \times R$	0	
9	$T_3 \times T_1$	0	$T_3 = M + 0 - 0$
		1	$T1 = R \ll 1 + 0 - 0$
10	$T_2 \times T_2$	0	$X_1 = M + 0 - 0$
11	$X_1 \times Y_1$	0	$R = M + 0 - T_1$
		1	$X_1 = 0 + R - X_1$
		2	$T_3 = 0 + T_3 - R$
12	$T_3 \times T_2$	0	$R = M + 0 - 0$
		1	$Y_1 = 0 + 0 - R$
		2	$T_3 = P_1 \ll 1 + 0 - 1$

$x_1/2 + (p+1)/2$, which is possible because both p and x_1 are odd. Thus, instead of storing p in the ECP and evaluating $(p+1)/2$ at every iteration of the binary inversion algorithm, $(p+1)/2$ is stored in P_1 and added to $x_1/2$ when AR_PC is 2 in Z1EVEN. Recall the operation $T_3 = P_1 \ll 1 + 0 - 1$ in PDBL and PADD states discussed earlier. This operation evaluates

Table 4.4: Instructions executed during inversion and FINAL states by the Prime ECP

MULT_PC	MULT	AR_PC	AR
		INVS State	
0		0	$Y_1 = M + Y_1 - 0$
		Z1EVEN State	
0		0	$Z_1 = 0 + Z_1 - 0 \gg 1$
		1	$R = 0 + T_1 - 0 \gg 1$
		2	$T_1 = P_1\|0 + R - 0$
		T3EVEN State	
0		0	$T_3 = 0 + T_3 - 0 \gg 1$
		1	$R = 0 + T_2 - 0 \gg 1$
		2	$T_2 = P_1\|0 + R - 0$
		SUBT State	
0		0	$Z_1\|T_3 = 0 + Z_1\|T_3 - T_3\|Z_1$
		1	$T_1\|T_2 = 0 + T_1\|T_2 - T_2\|T_1$
		FINAL State	
0	$T_1\|T_2 \times T_1\|T_2$	0	
1		0	$T_2\|T_1 = M + 0 - 0$
2	$X_1 \times R$	0	
3	$T_1 \times T_2$	0	$T_1 = M + 0 - 0$
4		0	$R = M + 0 - 0$
5	$Y_1 \times R$	0	
6		0	$T_2 = M + 0 - 0$

$T_3 = P_1 \times 2 - 1 = (p+1)/2 \times 2 - 1 = p$, which initializes T_3 to p ($v \leftarrow p$ in Algorithm 2.6) as required by the binary inversion algorithm.

4.2 Implementation Results

The implementation results on the Virtex-5 FPGA of the Prime ECP is shown in Table 4.5. Notice that the latency values are higher compared to the binary counterpart (e.g. 2-MULT Koblitz ECP or 2-MULT pseudo-random ECP), and hence the efficiencies are much lower. One of the reasons for the lower performance is the more complex addition in prime fields compared to binary fields. Another reason is the long latency due to the use of 17-bit digits in the MULT block instead of 32-bit blocks. As previously mentioned, since the latency of the Comba algorithm is in the order of $O(s^2)$, using fewer bits per digit results in a higher number of digits, which increases the latency.

Table 4.5 also shows the Prime ECP uses 7 DSP48E slices. Out of these 7, 1 is used in

Table 4.5: Implementation Results and Comparison of the Prime ECP

Work	FPGA	Registers	LUT	Slices	BRAM	DSP	Max. Freq. (MHz)	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
Prime ECP	Virtex-4 XC4VFX100	3,545	12,435	7,020	4	8	181.95	192	2.361	0.060
								224	3.663	0.039
								256	5.457	0.026
								384	16.31	0.009
								521	38.73	0.004
Prime ECP	Virtex-5 XC5LX110T	3,567	6,115	1,980	2	7	251.32	192	1.709	0.295
								224	2.652	0.190
								256	3.951	0.128
								384	11.81	0.043
								521	28.04	0.018
Ananyi et al. [42]	Virtex-4 XC4VFX100	n/a	31,946	20,793	1	32	60	192	4.8	0.010
								224	5.8	0.008
								256	6.9	0.007
								384	19.9	0.002
								521	45.6	0.001

the MULT block and 5 are used in the AR block as shown in Figure 4.1. The final DSP48E slice is used in the controller to compare the Z_1 and T_3 values during the inversion states to determine whether or not $u \geq v$. Furthermore, it can be observed that with the use of the DSP48E slices results in the use of fewer FPGA slices.

The Virtex-4 implementation result of the Prime ECP is also shown in Table 4.5. The Virtex-4 implementation requires an extra DSP48E slices compared to the Virtex-5 because the Virtex-4 version of the DSP48E slice does not support the Z − (X + Y + CARRYIN) operation shown in Table 2.6. Thus, an extra DSP48E slice is used to perform the addition of the ports [A1:B1] and C1.

The Virtex-4 implementation compared to the work in [42], which resembles the Prime ECP since it also supports all 5 NIST recommended prime curves without the need to reconfigure the hardware. As can be seen in Table 4.5, the design in [42] requires a higher number of slices (20,793 compared to 7,020) and DSP48E slices (32 compared to 8). Furthermore, the latency of the Prime ECP is also between 15% to 50% lower compared to [42]. The higher performance of the Prime ECP is due to the lower datapath with of the finite field arithmetic blocks. In [42], the authors implement a 265-bit adder/subtractor, an integer multiplier with a 521-bit reductor and a separate module to perform inversion. These result in a much higher hardware resource utilization and lower maximum clock frequency.

Chapter 5
Scalable and Unified ECC Processor

Based on the design of the scalable ECPs discussed in the previous chapters, an ECP that supports all 15 NIST recommended elliptic curves [18] is implemented and described in this chapter. In Section 5.1, the design and architecture of the 'Scalable and Unified ECP' is described with the details of its operation. This work has been submitted to a journal for peer review in [68]. In Section 5.2, the interaction of the ECP described in Section 5.1 with the Microblaze soft-core processor is discussed.

5.1 Design and Architecture[1]

Recall that this thesis defines a unified ECP as one that supports both binary and prime fields on-the-fly. Throughout the development of the previously described ECPs for Koblitz, pseudo-random and prime curves, the architectures of the ECPs have been kept as consistent as possible in order to facilitate the integration into a scalable and unified ECP. Thus, one can notice many similarities among the aforementioned ECP architectures. This section describes an ECP that is able to support all 15 curves recommended by NIST [18] without the need to reconfigure the hardware.

Recall from Section 2.4 that the DSP48E slices have the capability to perform both integer addition and binary field addition, through the built-in configurable ALU. Thus, DSP48E slices are used as building blocks for the finite field arithmetic units, similar to the prime ECP described in Chapter 4.

The Scalable and Unified ECP uses the same structure as the previous ECPs by adopting

[1]The work in this section has been submitted to *IEEE Transactions on Industrial Electronics* [68] for peer review.

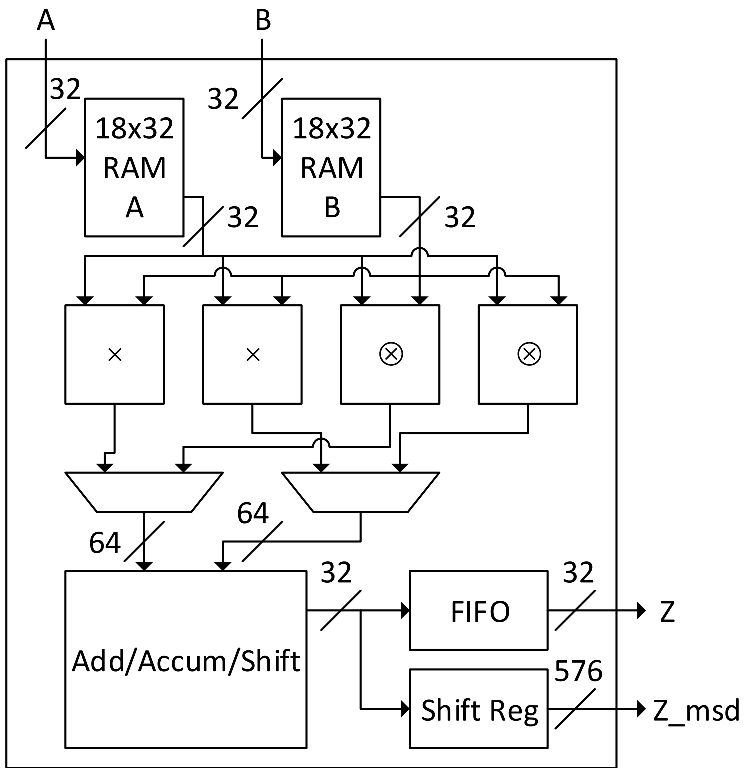


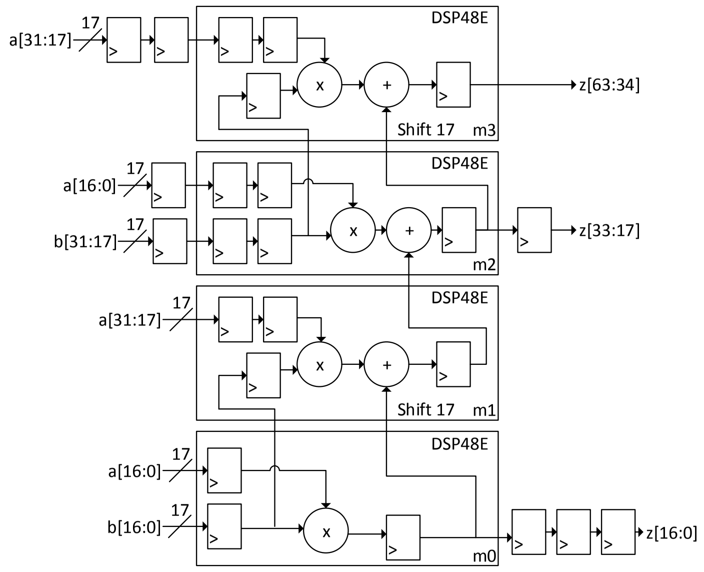
Figure 5.1: Block diagram of the MULT block for the Scalable and Unified ECP.

the use of parallel MULT and AR blocks. The block diagram of the MULT block is shown in Figure 5.1. Since the DSP48E slices do not support binary field multiplication, separate multiplier blocks are used for prime and binary field multiplication. The MULT block adopts the dual multiplier block architecture used in Figure 3.7(a).

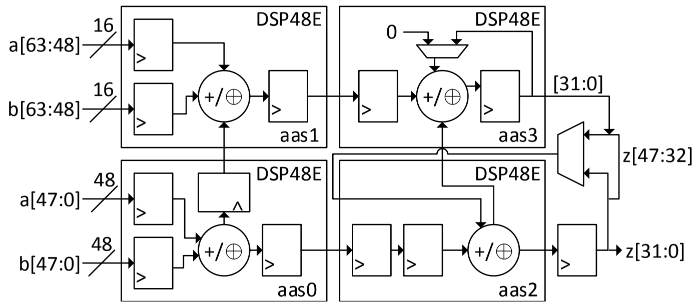
The '$\times$' block performs integer multiplication for prime fields and the '$\otimes$' block performs polynomial multiplication for binary fields. The '$\otimes$' block is the same as the multiplier block in Figure 3.7(a), which is a pipelined 32×32-bit Karatsuba-Ofman multiplier.

Since the DSP48E slices only have 25×18-bit multipliers, in order to implement a 32×32-bit integer multiplier, multiple DSP48E slices are needed. The DSP48E user guide [66] describes the architecture of a 42×35-bit multiplier, which is used to implement the multiplier shown in Figure 5.2(a). The '$\times$' block uses 4 DSP48E blocks in cascade and takes 6 clock cycles to complete the first multiplication, but requires only 1 additional clock cycle for every subsequent execution.

The 64-bit product from the multipliers is passed to the 'add/accum/shift' block that performs addition, accumulation and shifting as per the Comba algorithm. Since the MULT



(a) 32×32-bit integer multiplier ('$\times$' block).



(b) Add/Accum/Shift

Figure 5.2: Block diagram blocks used in the MULT block of the Scalable and Unified ECP.

block uses 32-bit digits, it cannot take advantage of the 17-bit shift built into the DSP48E slices. Thus, the architecture shown in Figure 5.2(b) is used. Its architecture is based on the 96-bit adder/subtractor and 96-bit accumulator shown in [66]. The blocks labeled 'aas0' and 'aas1' perform a 64-bit addition and the blocks labeled 'aas2' and 'aas3' perform a 64-bit accumulation. The feedback of the accumulator is placed outside the DSP48E slice in order for a multiplexer to be added to perform 32-bit right-shifting of the Comba algorithm. For binary field operation, the 'add/accum/shift' block is configured to use the XOR gates instead of integer addition. The carry signals (i.e. connection from 'aas0' to 'aas1' and from 'aas2' to 'aas3') are ignored by the DSP48E slice. The output of the 'add/accum/shift' block is connected to a FIFO and a shift register. These blocks perform the same function as the ones in Figure 3.7(a).

The block diagram of the AR block is shown in Figure 5.3. It performs the operation 'A Reg' + 'B Reg' − 'C Reg' $\pmod{p}$ in prime fields and 'A Reg' + 'B Reg' + 'C Reg' $\pmod{P(t)}$ in binary fields. Its architecture is similar to the AR block of the Prime ECP shown in Figure 4.1(b). However, the AR block in Figure 5.3 implements a tree architecture to eliminate the need for multiple passes for digit used in the AR block for the prime ECP.

For prime fields, upon examining Algorithm B.1 to Algorithm B.5 in Appendix B, one can notice that at most 8 digits are added and at most 4 digits are subtracted in the reduction algorithms. Thus, the DSP48E slices in Figure 5.3 labeled 'a0', 'a1', 'a2' and 'a3' are used for addition and 's0' and 's1' are used for subtraction. The 'as0' block is used to input the digits of 'B Reg' and 'C Reg'. Since the DSP48E slices have the ability to input a 3$^{\text{rd}}$ operand through the cascaded input 'PCIN', the adder tree is built to take advantage of this feature to reduce the number of DSP48E slices required. At the top of the adder tree, the digits are collected at the 'Z FIFO' block similar to 'Z Reg' in Figure 4.1(b). The second stage of the AR block shown at the bottom-right corner of Figure 5.3 ('f0' and 'f1' blocks) performs the final $\pmod{p}$ operation of the reduction algorithms. Finally, the AR block outputs the result as 32-bit digits.

For binary fields, the same architecture is used for reduction. The digits selected by the DSP48E slices are based on the reduction matrix developed for Equation (2.24). However, when using the DSP48E slices for logical operations (i.e. XOR), they do not support the 3$^{\text{rd}}$

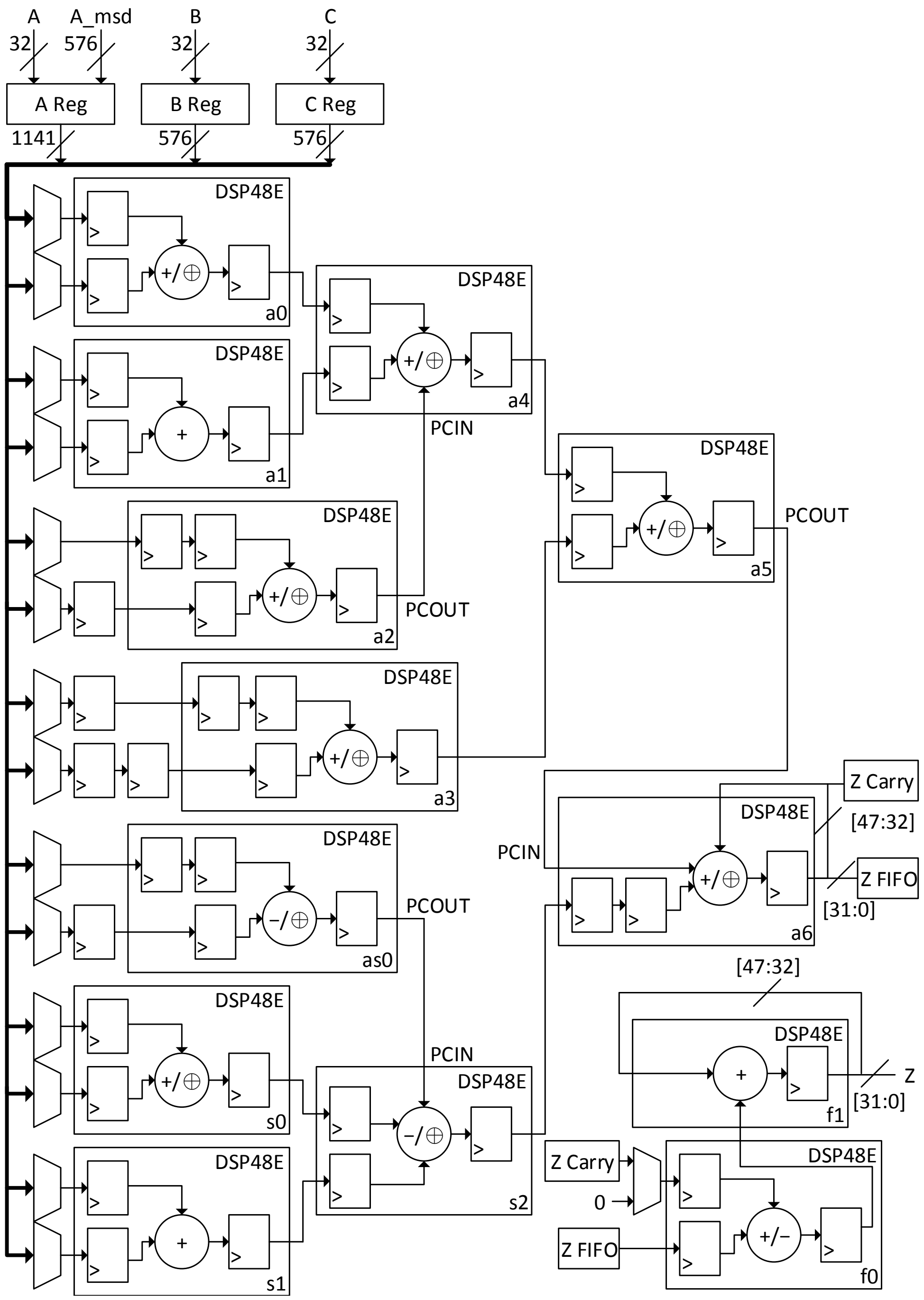


Figure 5.3: Block diagram of the AR block for the Scalable and Unified ECP.

operand as in the integer addition and subtraction case. Thus, the slices 'a1' and 's1' are disabled and the remainder of the DSP48E slices are configured to perform XOR operations. Since binary field operations do not generate carry bits, the 'f0' and 'f1' blocks do not modify the value in the 'Z FIFO', yet they are still used to preserve the latency between the prime

Table 5.1: Latencies in clock cycles of the MULT and AR blocks in the Scalable and Unified ECP.

m	t_{MULT}	t_{AR}	Ratio (t_{MULT}:t_{AR})
Binary			
163	34	23	1.48
233	51	27	1.89
283	61	29	2.10
409	111	37	3.00
571	196	47	4.17
Prime			
192	34	23	1.48
224	42	25	1.68
256	51	27	1.89
384	97	35	2.77
521	177	45	3.93

field and binary field operations.

Other than addition, subtraction and reduction, the AR block also performs squaring in binary fields because squaring is simply interleaving zero bits with operand bits. When the AR block is used for squaring, the operand to be squared is stored in 'A Reg' with zero bits interleaved and the reduction operation is performed. Thus, the latency of squaring using the AR block is the same as addition and reduction. This feature cannot be applied to prime fields, so squaring in prime field is performed by the MULT block.

Table 5.1 shows the latencies of the MULT and AR blocks for binary and prime fields. As shown by the t_{MULT}:t_{AR} ratio column, unlike in previous ECPs, in some cases only 1 AR block operation can completely finish its execution during the execution of a MULT block operation. Thus, the execution of the MULT and AR blocks are modified slightly from the previous ECPs. In previous ECPs, the MULT block operation initiates at the same time as the AR block operations. Once all the AR operations complete, the AR block becomes idle and waits for the MULT block to complete its operation. In the Scalable and Unified ECP, the MULT and AR blocks begin their operations, depending on the sequence of instructions, the block that completes its operations first becomes idle and waits for the other to complete. By doing so, the ECP can still take advantage of parallel executions of the MULT and AR

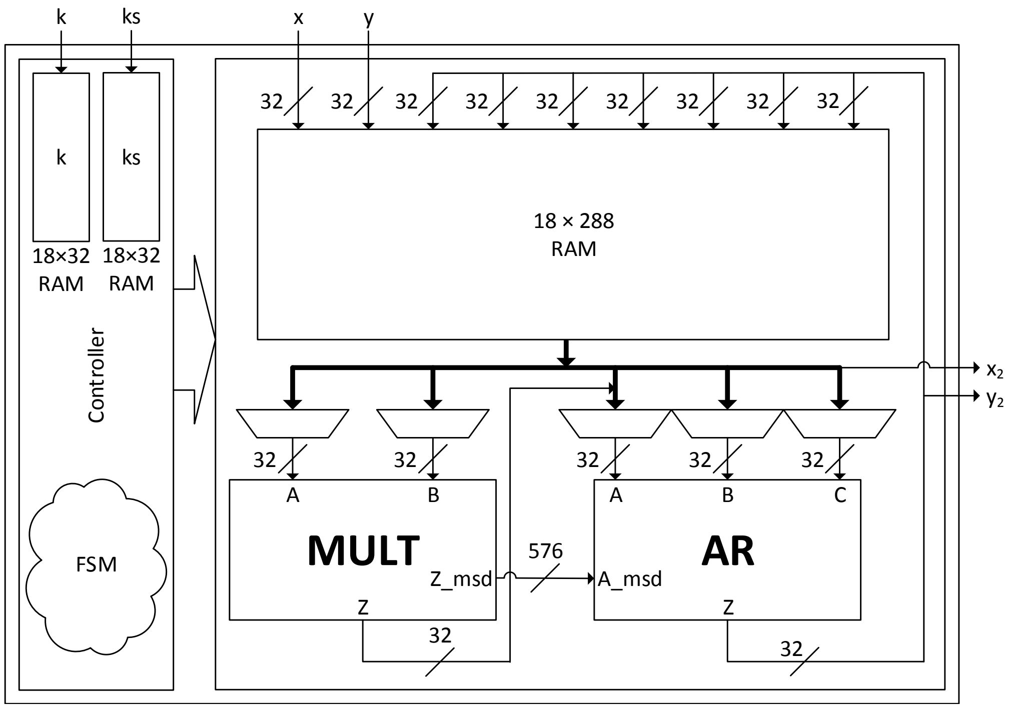


Figure 5.4: Block diagram of the Scalable and Unified ECP.

blocks.

The block digram of the Scalable and Unified ECP is shown in Figure 5.4. This architecture is similar to the one used in the ECPs described in the previous chapters. The 'k' and 'ks' ports input the value of k for the ECPM. When the ECP is set to operate on pseudo-random or prime curves, the 'ks' port is not used. The RAM is 18×288 bits to store the x and y coordinates of the point to be multiplied and 7 other temporary variables ($9 \times 32 = 288$).

The top level FSM of the Scalable and Unified ECP is shown in Figure 5.5. The circular shapes are individual states and the cloud shapes are collections of states. This FSM combines the FSMs of the 1-MULT Koblitz ECP, 1-MULT Random ECP and the Prime ECP. The IDLE and LOAD states are common to all FSMs. From the LOAD state, depending on the curve selected, the FSM moves into one of the 3 collections of states (one for each type of curve) to perform the main loop of the ECPM and the inversion setup state. Subsequently, the FSM performs finite field inversion. The choice of the inversion algorithm will be discussed later in this section. From the inversion states, the FSM moves to the FINAL states to complete the coordinate conversion and move to the common WAIT state to complete the ECPM operation.

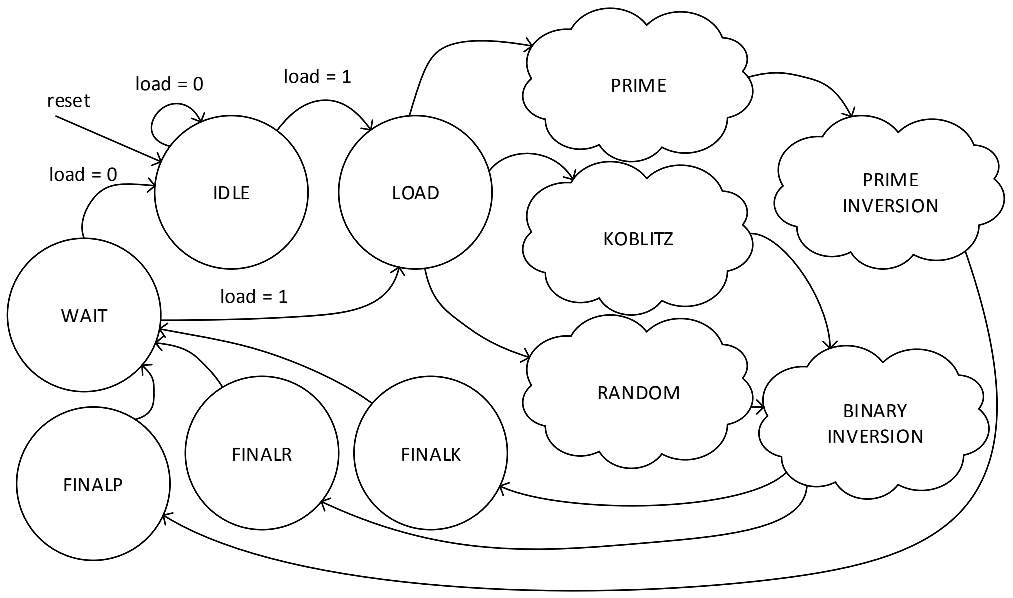
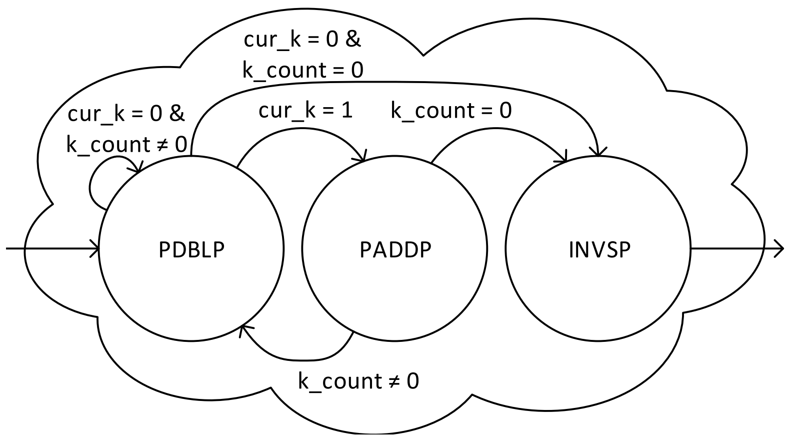


Figure 5.5: FSM of the Scalable and Unified ECP.

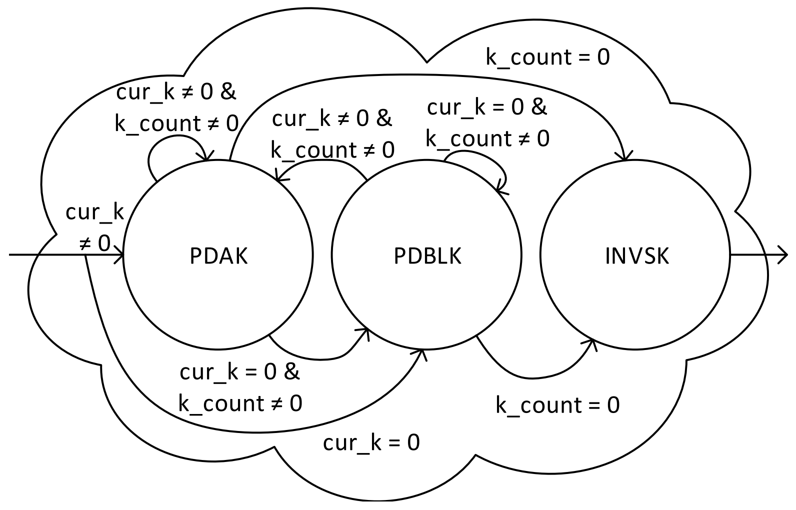
The sub-FSMs of the PRIME, KOBLITZ and RANDOM clouds in Figure 5.5 are shown in Figure 5.6 and the instructions executed are shown in Table 5.2, Table 5.3 and Table 5.4, respectively. Notice that the instructions of the scalable and unified ECP presented in this section are slightly different from the instructions shown in previous sections. The modifications minimize the number of AR block operations that are executed for each MULT block operation. By doing so, the latency is reduced when the t_{MULT}:t_{AR} ratio is low for lower bit lengths.

In order to select the algorithm to use for inversion in the Scalable and Unified ECP, the inversion algorithms described in Section 2.3 are analyzed using the architecture of the finite field arithmetic blocks described above. The algorithms that are taken into consideration are the binary inversion algorithm for both binary and prime fields, the Itoh-Tsujii algorithm for binary fields, and using Fermat's Little Theorem for prime fields.

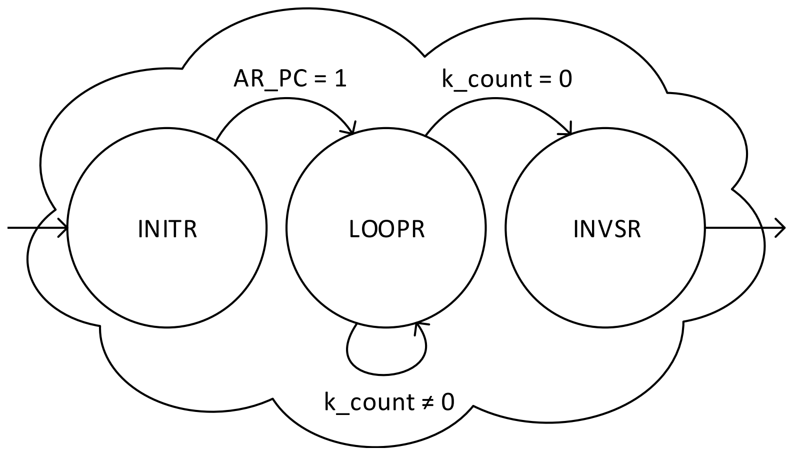
The binary inversion algorithm has been previously used for the prime ECP in Chapter 4. By carefully analyzing Algorithm 2.6, one can notice that after both inner **while** loops exit, both u and v are odd numbers. Thus, after the subtraction of $u \leftarrow u - v$ or $v \leftarrow v - u$, only one of u or v is an even number. Therefore, only one of the inner **while** loops is entered at each iteration of the outer **while** loop. Furthermore, during each iteration of the inner **while** loop, u or v is divided by 2, reducing its bit length by 1. Since the outer **while** loop exits



(a) Sub-FSM for Prime curves.



(b) Sub-FSM for Koblitz curves



(c) Sub-FSM for Pseudo-random curves

Figure 5.6: Sub-FSM for the main loop of the Scalable and Unified ECP.

Table 5.2: Instructions executed in the PRIME states of the Scalable and Unified ECP

MULT_PC	MULT	AR_PC	AR
		PDBLP State	
0	$Z_1 \times Z_1$	0	$Y_1 = M\|0 + Y_1\|0 - Y_1\|R$
		1	$T_3 = R \ll 1 + 0 - 0$
1	$T_3\|R \times Z_1$	0	$T_1 = M + 0 - 0$
		1	$T_2 = 0 + X_1 - R$
2	$Y_1 \times Y_1$	0	$Z_1 = M + 0 - 0$
		1	$T_1 = T_1 + X_1 - 0$
3	$T_1\|R \times T_2$	0	$T_1 = M + 0 - 0$
		1	$T_3 = R \ll 2 + 0 - 0$
4	$T_3\|R \times X_1$	0	$R = M + 0 - 0$
		1	$T_2 = R \ll 1 + R - 0$
5	$T_2\|R \times T_2\|R$	0	$T_3 = M + 0 - 0$
		1	$X1 = R \ll 1 + 0 - 0$
6	$T_1 \times T_1$	0	$X_1 = M + 0 - X_1\|R$
		1	$T_1 = 0 + T_3 - R$
7	$T_1\|R \times T_2$	0	$R = M + 0 - 0$
		1	$Y_1 = R \ll 3 + 0 - 0$
		PADDP State	
0	$Z_1 \times Z_1$	0	$Y_1 = M + 0 - Y_1\|R$
1		0	$T_1 = M + 0 - 0$
2	$Z_1 \times R$	0	
3	$T_1 \times x$	0	$T_2 = M + 0 - 0$
4	$T_2 \times y$	0	$T_1 = M + 0 - X1$
5	$T_1 \times Z_1$	0	$T_2 = M + 0 - Y_1$
6	$T_1 \times T_1$	0	$Z_1 = M + 0 - 0$
7		0	$T_3 = M + 0 - 0$
8	$X_1 \times R$	0	
9	$T_3 \times T_1$	0	$T_3 = M + 0 - 0$
		1	$T1 = R \ll 1 + 0 - 0$
10	$T_2 \times T_2$	0	$X_1 = M + 0 - 0$
		1	$T_1 = T_1 + R - 0$
11	$X_1 \times Y_1$	0	$X_1 = M + 0 - T_1\|R$
		1	$T_3 = 0 + T_3 - R$
12	$T_3\|R \times T_2$	0	$Y_1 = M + 0 - 0$
		INVSP State	
0		0	$Y_1 = M + 0 - Y_1\|R$

when $u = 1$ or $v = 1$, the outer **while** loop executes at most $2m$ iterations, where m is the bit length. Using the instructions provided in Table 4.4, the binary inversion algorithm requires $t_{\text{BININV}} = 2m(2t_{\text{SHIFT}} + t_{\text{AR}} + 2t_{\text{AR}})$, where $t_{\text{SHIFT}} = s + 1$ is the latency of the right-shift by 1 operation and s is the number of 32-bit digits. Using the binary inversion algorithm for binary fields, the same analysis can be performed and yields the same expression for latency.

The Itoh-Tsujii algorithm for inversion in binary fields has been previously discussed in Section 2.3.1 and it is used by the binary ECPs described in Section 3.2 and Section 3.3.

Table 5.3: Instructions executed in the KOBLITZ states of the Scalable and Unified ECP

MULT_PC	MULT	AR_PC	AR
		PDAK State	
0	$x \times Z_1 \vert R$	0	$Y_1 = M \vert 0 + Y_1 + 0$
		1	$T_2 = Z_1^2 + 0 + 0$
		2	$X_1 = X_1^2 + 0 + 0$
1	$T_2 \times y \vert xy$	0	$T_1 = M + X_1 \vert R + 0$
		1	$Y_1 = Y_1^2 + 0 + 0$
2	$T_1 \times Z_1$	0	$X_1 = M + Y_1 \vert R + 0$
		1	$T_1 = T_1^2 + 0 + 0$
3		0	$Z_1 = M + 0 + 0$
4	$X_1 \times R$	0	$Y_1 = 0 + R + T_2 \vert 0$
		1	$Z_1 = Z_1^2 + 0 + 0$
5	$Y_1 \times T_1$	0	$T_2 = M + 0 + 0$
		1	$X_1 = X_1^2 + 0 + 0$
6	$x \times Z_1$	0	$X_1 = M + X_1 \vert R + T_2$
		1	$T_1 = Z_1^2 + 0 + 0$
7	$T_1 \vert R \times xy \vert y$	0	$T_3 = M + X_1 + 0$
		1	$T_2 = 0 + T_2 + Z_1$
8	$T_3 \times T_2 \vert R$	0	$Y_1 = M + 0 + 0$
		1	$Z_1 = Z_1^2 + 0 + 0$
		PDBLK State	
0		0	$Y_1 = M \vert 0 + Y_1 + 0$
		1	$X_1 = X_1^2 + 0 + 0$
		2	$Y_1 = Y_1^2 + 0 + 0$
		3	$Z_1 = Z_1^2 + 0 + 0$
		INVSK State	
0		0	$Y_1 = M \vert 0 + Y_1 + 0$

Analyzing the example shown in Equation (2.27), one can see that the latency of the Itoh-Tsujii algorithm requires $(m-1)$ squaring operations and $\lfloor \log_2(m-1) \rfloor + H(m-1) - 1$ multiplications, where $H(x)$ is the Hamming weight of the value x. Using the MULT and AR blocks described in this section, the squaring operation is performed by the AR block and each multiplication requires a MULT block operation for integer multiplication and an AR block operation for reduction. Thus, the latency of the Itoh-Tsujii algorithm is $t_{\mathrm{ITAINV}} = (m-1)t_{\mathrm{AR}} + (\lfloor \log_2(m-1) \rfloor + H(m-1) - 1)(t_{\mathrm{MULT}} + t_{\mathrm{AR}})$.

Using Fermat's Little Theorem for prime field inversion transforms the inversion operation into $a^{-1} \pmod{p} = a^{p-2} \pmod{p}$. One method of evaluating exponentiation is by using the square-and-multiply algorithm, similar to the double-and-add algorithm for multiplication. By doing so, the number of squaring and multiplications required depends on the binary representation of the value $p-2$, and can be evaluated with $(m-1)$ squarings and $H(p-$

Table 5.4: Instructions executed in the RANDOM states of the Scalable and Unified ECP

MULT_PC	MULT	AR_PC	AR
		INITR State	
0		0	$Z_2 = x^2 + 0 + 0$
		1	$X_2 = R^2 + 0 + b$
		LOOPR State	
0	$X_1\|X_2 \times Z_2\|Z_1$	0	$X_2\|X_1 = M\|R + T_2\|0 + 0$
		1	$R = (Z_1\|Z_2)^2 + 0 + 0$
		2	$T_3 = R^2 + 0 + 0$
1	$X_2\|X_1 \times Z_1\|Z_2$	0	$T_2 = M + 0 + 0$
2	$X_1\|X_2 \times Z_1\|Z_2$	0	$T_1 = M + 0 + 0$
		1	$Z_2\|Z_1 = 0 + R + T_2$
3	$T_2 \times T_1$	0	$R = M + 0 + 0$
		1	$Z_1\|Z_2 = R^2 + 0 + 0$
		2	$Z_2\|Z_1 = (Z_2\|Z_1)^2 + 0 + 0$
4	$b \times T_3$	0	$T_2 = M + 0 + 0$
		1	$R = (X_1\|X_2)^2 + 0 + 0$
		2	$T_3 = R^2 + 0 + 0$
5	$x \times Z_2\|Z_1$	0	$X_1\|X_2 = M + T_3\|R + 0$
		INVSR State	
0	$x \times Z_1$	0	$X_2\|X_1 = M + T_2 + 0$
1	$x \times Z_2$	0	$Z_1 = M + 0 + 0$
		1	$T_1 = 0 + R + X_1$
2	$Z_1 \times Z_2$	0	$T_2 = M + 0 + 0$
		1	$X_2 = 0 + R + X_2$
3	$X_1 \times T_2$	0	$Z_1 = M + 0 + 0$
4	$X_2 \times T_1$	0	$X_2 = M + 0 + 0$
5	$T_1 \times Z_2$	0	$X_1 = M + 0 + 0$
6	$x \times X_1$	0	$Z_2 = M + 0 + 0$
7		0	$X_1 = M + 0 + 0$

2) $-$ 1 multiplications. Using the MULT and AR blocks in prime field, both squaring and multiplication are evaluated using the MULT block followed by an AR block operation for reduction. Thus, the latency of inversion using Fermat's Little Theorem is $t_{\text{FERMAT}} = (m - 1)(t_{\text{MULT}} + t_{\text{AR}}) + (H(p-2) - 1)(t_{\text{MULT}} + t_{\text{AR}})$.

Table 5.5 shows the latencies in number of clock cycles of the inversion algorithms discussed. The ratio column provides the ratio of the binary inversion algorithm to the Itoh-Tsujii algorithm in binary fields and Fermat's Little Theorem in prime fields. From Table 5.5, one can obverse that the binary inversion algorithm is worse than its counterpart for every value of m except for 521-bit prime field. Furthermore, the latency of the Itoh-Tsujii algorithm is over a factor of 6 less than the binary inversion algorithm. The main reason that the difference is not as significant in prime fields is because squaring is performed using the

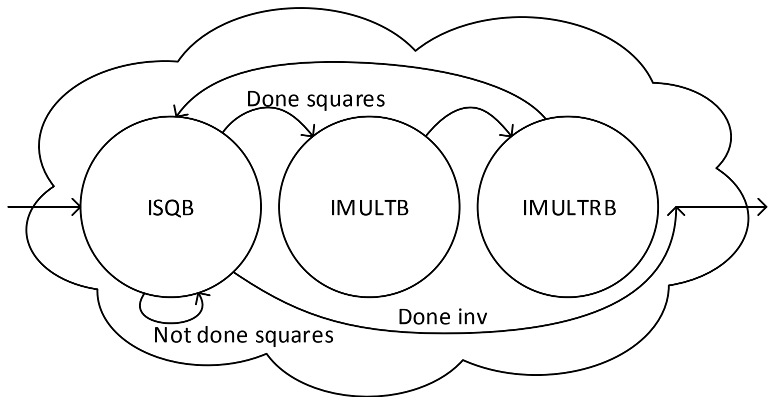
Table 5.5: Latency comparison of the inversion algorithms

m	t_{ITAINV}	t_{FERMAT}	t_{BININV}	Ratio
Prime Fields				
192	n/a	21660	31872	1.47
224	n/a	29815	40768	1.37
256	n/a	29796	50688	1.70
384	n/a	92400	100608	1.09
521	n/a	230658	178182	0.77
Binary Fields				
163	4239	n/a	27058	6.38
233	7044	n/a	46134	6.55
283	9168	n/a	60562	6.61
409	16724	n/a	113702	6.80
571	29949	n/a	204418	6.83

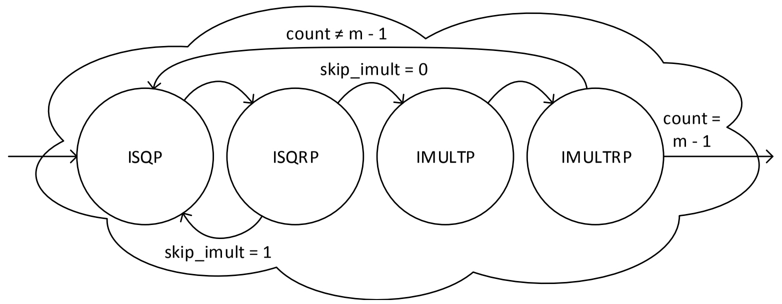
MULT block in prime fields and requires a reduction step.

From these observations, the Scalable and Unified ECP chooses to use the Itoh-Tsujii algorithm for inversion in binary fields and Fermat's Little Theorem for inversion in prime fields. The sub-FSM of the inversion states are shown in Figure 5.7 and the instructions executed are shown in Table 5.6 and Table 5.7. The sub-FSM of the Itoh-Tsujii algorithm shown in Figure 5.7(a) resembles the inversion states in the ECPs described in Section 3.3. The ISQB state executes squaring and the IMULTB and IMULTRB states execute the multiplication. The sub-FSM for inversion based on Fermat's Little Theorem shown in Figure 5.7(b) is very similar to the one for Itoh-Tsujii, except it requires an extra state to perform reduction for squaring. Finally, the instructions executed for the FINAL states for each of the 3 types of curves are shown in Table 5.8. These states complete the coordinate conversion.

The FPGA implementation result of the Scalable and Unified ECP is shown in Table 5.9. It can be seen that 25 DSP48E slices are used, which includes 4 in each '$\times$' block of the MULT block, 4 in the add/accum/shift block and 13 in the AR block. The maximum clock frequency of the Scalable and Unified ECP is 155.35 MHz, which is lower than in the 1-MULT Koblitz ECP, 1-MULT Random ECP and Prime ECP. The lower clock frequency is expected since the hardware is more complex with the combination of binary and prime field



(a) Sub-FSM for Binary Inversion



(b) Sub-FSM for Prime Inversion

Figure 5.7: Sub-FSM for inversion of the Scalable and Unified ECP.

Table 5.6: Instructions executed in the BINARY INVERSION states of the Scalable and Unified ECP

MULT_PC	MULT	AR_PC	AR
	ISQB State		
0		0	$T_1\|R = (Z_1\|R)^2 + 0 + 0$
	IMULTB State		
0	$R \times Z_1\|T_1$	0	
	IMULTRB State		
0		0	$T_1 = M + 0 + 0$

Table 5.7: Instructions executed in the PRIME INVERSION states of the Scalable and Unified ECP

MULT_PC	MULT	AR_PC	AR
	ISQP State		
0	$Z_1\|R \times Z_1\|R$	0	
	ISQRP State		
0		0	$R = M + 0 + 0$
	IMULTP State		
0	$R \times Z_1$	0	
	IMULTRP State		
0		0	$T_1 = M + 0 + 0$

operations.

In general, it is expected that the Scalable and Unified ECP has lower performance than the previously described scalable ECPs that only support 1 of the 3 types of curves, since the underlying finite field arithmetic units must support both binary and prime fields, whereas the scalable ECPs only support one of the fields. Comparing the efficiency values of the Scalable and Unified ECP in Koblitz and pseudo-random mode with the results in Table 3.7 and Table 3.10 confirms this expectation. However, in prime curves mode, the efficiency of the Scalable and Unified ECP outperforms the Prime ECP described in Chapter 4. The reason for the improved performance of the Scalable and Unified ECP is due to the use of

Table 5.8: Instructions executed in FINAL states of the Scalable and Unified ECP

MULT_PC	MULT	AR_PC	AR
	FINALP State		
0	$R \times R$	0	
1		0	$T_2 = M + 0 - 0$
2	$X_1 \times R$	0	
3	$T_1 \times T_2$	0	$T_1 = M + 0 - 0$
4		0	$R = M + 0 - 0$
5	$Y_1 \times R$	0	
6		0	$T_2 = M + 0 - 0$
	FINALK State		
0	$X_1 \times R$	0	$T_3 = R^2 + 0 + 0$
1	$T_3 \times Y_1$	0	$T_1 = M + 0 + 0$
2		0	$T_2 = M + 0 + 0$
	FINALR State		
0	$R \times X_1$	0	$T_3 = x^2 + y + 0$
1	$T_1 \times Z_2$	0	$T_3 = M + T_3 + 0$
2	$X_2 \times T_1$	0	$T_1 = M + 0 + 0$
3	$T_1 \times T_3$	0	$T_1 = M + 0 + 0$
4		0	$T_2 = M + y + 0$

the dual 32-bit multiplier in the MULT block instead of a 17-bit multiplier, which reduces the latency of the MULT block operation considerably.

Table 5.9 also shows the implementation results of some designs in the current literature. As previously mentioned, there is no work in the current literature that implements all 15 NIST recommended curves in a single hardware device. Thus, the comparison with other designs is not entirely fair, but are included in for reference purposes. The designs in [30] and [29] show ECPs that are unified but not scalable.

In [30], the authors propose a unified ECP architecture using multiple word-based arithmetic units (AU) that consist of a unified multiplier and a unified adder. Since [30] uses an older FPGA (Virtex-II Pro vs Virtex-5), the comparison is not completely fair. However, Table 5.9 shows that the latency and the number of slices used in the Scalable and Unified ECP is lower than in [30]. The higher number of slices in [30] is due to the use of 4 AUs each with a multiplier and an adder, whereas the Scalable and Unifeid ECP only has 1 MULT

Table 5.9: Implementation Results and Comparison of the Scalable and Unified ECP

Work	FPGA	Reg.	LUT	Slices	BRAM	DSP	Max. Freq. (MHz)	Curve	m	Latency (ms)	Efficiency $\left(\frac{\text{ECPM}}{\text{s}\cdot\text{slice}}\right)$
Scalable and Unified ECP	Virtex-5	4,244	8,316	2,291	5	25	155.35	Prime	192	0.857	0.510
									224	1.127	0.387
									256	1.378	0.317
									384	3.922	0.111
									521	9.662	0.045
								Koblitz	163	0.239	1.825
									233	0.399	1.094
									283	0.533	0.818
									409	1.185	0.368
									571	2.643	0.165
								Pseudo-Random	163	0.365	1.195
									233	0.646	0.676
									283	0.870	0.502
									409	1.866	0.234
									571	4.523	0.097
Lai and Huang [30]	Virtex-II Pro	n/a	n/a	39,531	n/a	n/a	94.7	Prime	160	0.782	0.032
				40,219					192	1.25	0.020
				41,595					256	2.66	0.009
				39,531				Binary	160	0.574	0.044
Wang et al. [29]	Virtex-4	n/a	n/a	5,227 CLBs	n/a	n/a	150.5	Prime	192	0.542	0.353[a]
								Pseudo-Random	163	0.347	0.551[a]
Ananyi et al. [42]	Virtex-4	n/a	31,946	20,793	1	32	60	Prime	192	4.8	0.010
									224	5.8	0.008
									256	6.9	0.007
									384	19.9	0.002
									521	45.6	0.001
Hassan & Benaissa [39]	Spartan-3	913	2028	1278	4	0	90	Koblitz	163	15.5	0.050
									283	45.1	0.017
									571	121.4	0.006
Hassan & Benaissa [17]	Spartan-3	650	2025	1127	4	0	68	Pseudo-Random	163	38	0.023
									233	73.4	0.012
									283	104	0.009
									409	251	0.004
									571	287.4	0.003

[a] Assumes only 1 slice is used per CLB.

and 1 AR block.

The design in [29] is a unified ECP that can perform both RSA and ECC operations. The reported ECP is able to compute ECPM for a 192-bit prime curve, 163-bit pseudo-random curve and 1024-bit RSA. It implements a separate modular inversion unit to evaluate the binary inversion algorithm. Since only the configurable logic block (CLB) information is given in [29], the efficiency metric assumes that each CLB only uses 1 of the 4 slices, which is underestimated. Even though the ECP in [29] has a lower latency for P-192 and B-163, the results in Table 5.9 show that [29] has a lower efficiency compared to the scalable and unified ECP, and it does not support the other NIST recommended curves with larger bit lengths.

Other researchers presented works on scalable ECPs that are not unified. These works have been previously shown in Table 3.2, Table 3.3 and Table 4.5, but are shown again in Table 3.2 for comparison with the Scalable and Unified ECP. In [42], the authors present an ECC processor that supports all 5 NIST recommended prime curves. The design uses a wide datapath, which results in a slow maximum clock frequency of only 60 MHz and a large number of slices. Even though the Scalable and Unified ECP supports binary curves in addition to prime curves, it still outperforms the design in [42] in both timing performance and area. Furthermore, the Scalable and Unifeid ECP supports all 15 NIST recommended curves with 25 DSP slices, whereas the design in [42] uses 32 DSP slices to support only the 5 prime curves.

In [39] and [17], the authors have designed ECPs for Koblitz and pseudo-random curves for area-constrained environments. The Spartan-3 is an older FPGA, so the comparison is not completely fair. However, the low maximum frequency and the high number of clock cycles due to the use of Hardware/Software Co-design (HSC) result in very long latencies compared to the Scalable and Unified ECP. These comparisons demonstrate high performance of the Scalable and Unified ECP.

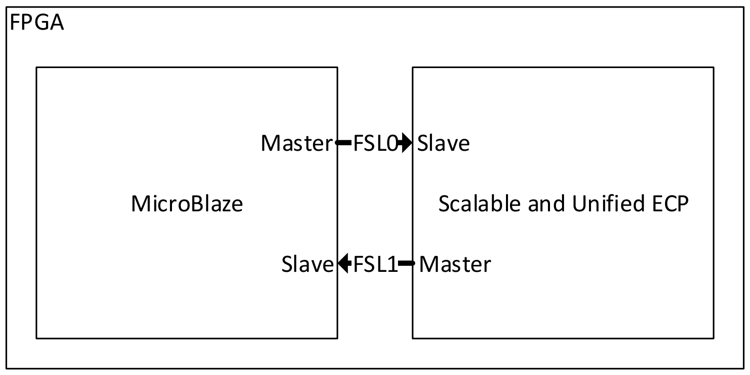


Figure 5.8: Block diagram of the interface between the Microblaze and the ECP.

5.2 Hardware/Software Co-design

This section describes the interfacing of the Scalable and Unified ECP described in Section 5.1 with software. In particular, this section describes the implementation of the ECP with the soft-core Microblaze microprocessor available in Xilinx FPGAs.

A soft-core processor is one that is implemented using FPGA fabric, so it is extremely flexible and can be configured to suit the user's needs. Xilinx provides software packages to very easily integrate the Microblaze into the FPGA and interface it with custom hardware designs. The goal of the design in this section is to set up a platform for future extensions in developing a protocol accelerator using the Scalable and Unified ECP developed in the previous section.

The top level block diagram of the interface between the Miroblaze soft-core processor and the Scalable and Unified ECP is shown in Figure 5.8. The connection between the two blocks is a Fast Simple Link described in [69]. The FSL is a one-directional communication link available in the Xilinx IP catalog to provide fast communication between modules. The architecture of the FSL is a FIFO with various control signals associated. The block digram of the FSL is shown in Figure 5.9 [69].

The FSL has a master and a slave side and data always flows from the master to the slave. The 'FSL_M_Clk' and 'FSL_S_Clk' ports are independent clock signals for the master and slave sides. In the current design, the same clock is used to operate the Microblaze and the ECP to simplify the design. The 'FSL_M_Data' and 'FSL_S_Data' ports are set to 32 bits by

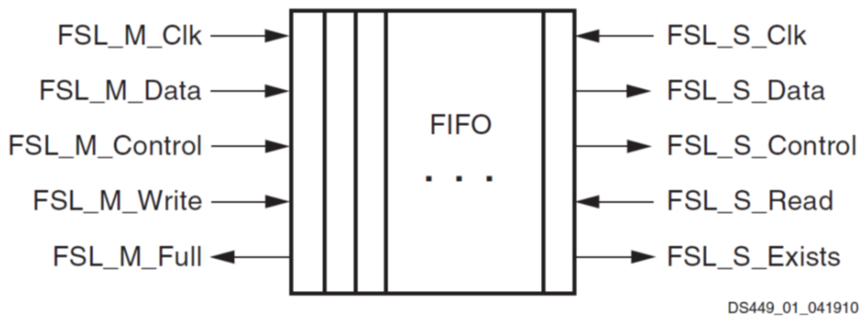


Figure 5.9: Block diagram of Xilinx Fast Simplex Link (FSL) [69].

the Microblaze by default. These are used to pass data through the FSL. Data is input from the master by asserting the 'FSL_M_Write' flag and stored in the FIFO. The slave side reads the data by asserting the 'FSL_S_Read' flag. The 'FSL_M_Control' and 'FSL_S_Control' ports are used to transmit a 1-bit signal from the master to the slave to indicate that the contents in the 'FSL_S_Data' port are used for control purposes. This feature allows the master to send 2 types of information to the slave (i.e. raw data or control signals) using the same 'FSL_M_Data' port.

Using 2 FSLs to connect the Microblaze to the ECP allows the ECP to act as a co-processor to the Microblaze. When an ECPM operation is required, the Microblaze calls a function to pass arguments to FSL0 to the ECP, where the Microblaze is the master and the ECP is the slave. This function must first send a control signal to configure the ECP to the appropriate mode (i.e. the type of curve and the bit length). Subsequently, the function transmits values required for ECPM operation (i.e. the x- and y-coordinates of the point, P, and the scalar multiplier, k) to the FSL as 32-bit digits. If Koblitz curves are selected, the value of k is expected to be τNAF converted and the input is separated into magnitude and sign. The ECP evalutes the ECPM and writes the resultant affine coordinates to FSL1, where the ECP is the master and the Microblaze is the slave. These values are also passed as 32-bit digits. When the Microblaze is ready, it may read the result from the FSL. During the time that the ECPM is evaluated by the ECP, the Microblaze is free to perform other functions.

Since the Scalable and Unified ECP shown in Figure 5.4 expects the inputs 'x', 'y', 'k' and 'ks' simultaneously, a wrapper module is used to collect the values from the FSL and input into the ECP. Similarly, the outputs 'x_2' and 'y_2' also require a wrapper module to package the values accordingly. Since both the ECP and the FSL operate in 32-bit digits, the wrapper module is simply a collection of FIFOs to collect the input and output values. Having the Scalable and Unified ECP connected to the MicroBlaze processor, the development of other protocol level algorithms, such as the ECDH or ECDSA, may be implemented in software while taking advantage of the high performance ECP.

Chapter 6
Conclusion

In today's society, network security is becoming growingly important with the increased reliance on the computers to exchange secret information. The increased amount of encrypted data transmitted across networks also demands more efficient implementations of security protocols to handle the higher network traffic while maintaining security. Elliptic Curve Cryptography (ECC) is regarded as an excellent successor to the Rivest-Shamir-Adleman (RSA) algorithm for public-key cryptography due to the level of security it can provide with much smaller key sizes. Reduced key sizes mean implementations of ECC can be much faster and use fewer resources than the respective RSA implementation to provide the same level of security. Hardware accelerators are commonly used to improve the performance of servers by offloading computationally intensive ECC operations to hardware. By doing so, the server's capabilities to respond to a high volume of requests increase and the server's processors can handle other operations more efficiently. The operations that are offloaded can range from simply moving the finite field operations to hardware or implementing a complete ECC scheme, such as ECDSA, in hardware.

This thesis focuses mainly on the elliptic curves recommended by NIST [18] as these are the most commonly used curves in other standards and protocols, such as SECG [19], and FIPS 186-3 [20]. NIST recommends a total of 15 curves and they are divided into 3 categories: binary pseudo-random curves, binary Koblitz curves and prime curves. Each of these categories can take advantage of different algorithms to compute the elliptic curve point multiplication (ECPM) operation, which is the most important computation in ECC.

In this thesis, the hardware implementation of various scalable elliptic curve processors (ECP) have been explored. Scalability refers to the design of an ECP that can compute the elliptic curve point multiplication (ECPM) in hardware and support multiple curves on-the-fly without the need to reconfigure the hardware. On server-side applications, highly scalable and high throughput implementations of ECPs are desired since servers must be able to handle a variety of security levels and must be able to respond to requests quickly. Thus, the designs in this thesis are more suitable for high-end server-side applications. This thesis also implements a scalable and unified ECP, where unified is defined as the ability to support both binary and prime fields using the same hardware.

As shown throughout this thesis, the efficiency and performance of a scalable ECP is highly dependent on the implementation of the underlying finite field arithmetic units. Thus, in this thesis, much of the attention is focused on designing efficient finite field arithmetic blocks. Chapter 3, Chapter 4 and Chapter 5 describe the architecture of these ECPs.

- The ECPs in Chapter 3 target the binary curves recommended by NIST [18].
- The design in Section 3.1 shows a ECP specific to the 163 pseudo-random curve recommended by NIST. The design has extremely low latency, but uses a large amount of hardware resources for only 1 type of curve. This work is published in [50]. This section shows that the ECP implementation of extremely low latencies is possible with a high usage of hardware resources. However, if the same architecture is used to implement scalable ECPs, the hardware resource utilization would be extremely high and would not fit in a single FPGA. The use of multiple FPGAs would increase the cost of the implementation.
- The designs in Section 3.2 are scalable ECPs for Koblitz and pseudo-random curves. The latencies are higher than the 163-bit ECP, but the hardware utilization is controlled by resource sharing among different bit lengths. Namely, the finite field arithmetic unit (FFAU) is designed to support all 5 binary fields recommended by NIST. These ECPs are published in [51] and [52]. The main drawback of these ECPs is that the reduction step of both multiplication and squaring is performed in every operation. In addition, only a single FFAU is used in the ECP. Thus, the latencies can be further reduced by exploring the parallelization of the multiplication and the ECPM operations.
- Section 3.3 presents binary field ECPs that parallelize the instructions of the ECPM by deploying a MULT and an AR block, and parallelize the multiplication by using two multipliers in the MULT block. These improvements result in binary ECPs with much

better efficiencies. The work related to these ECPs has been submitted for peer review in [67]. Despite the reduced latencies, longer latencies are observed for larger key sizes due to the use of the Comba algorithm, which has a latency in the order of $O(s^2)$, where s is the number of digits in the multiplication. Thus, the Karatsuba-Ofman algorithm is used in the Low Latency ECPs to improve the performance when using large key sizes.

- The architecture in Section 3.4 reduces the latencies of the ECP for higher bit lengths by using a wider datapath and the Karatsuba-Ofman algorithm. The ECPs in this section are published in [53]. These ECPs have lower latencies compared to the Parallelized ECPs, but require more slices. Nevertheless, they show to be more efficient than the Parallelized ECPs for larger key sizes.
- Using the experience from the binary ECPs, Chapter 4 describes the Prime ECP that uses a similar processor architecture as the binary ECPs but support the 5 prime curves recommended by NIST. This work is published in [54]. Taking advantage of the high performance Xilinx DSP48E slices, the Prime ECP increases the efficiency of the ECP compared to a reference design in the current literature. The improved performance can be attributed to the smaller datapath, which decreases the number of slices required and increases the maximum frequency of the design. Furthermore, despite the usage of Xilinx DSP48E slices in the implemented ECP, the architecture is portable and may be modified to utilize DSP slices in newer Xilinx FPGAs or FPGAs from other manufacturers, such as Altera.
- The Scalable and Unified ECP described in Section 5.1 combines the ECPs previously developed and has the ability to support all 15 curves recommended by NIST without the need to reconfigure the hardware. This work has been submitted for peer review in [68]. The implementation results of the Scalable and Unified ECP show an improvement over the Prime ECP due to the use of 32-bit datapath instead of 17-bit datapath in the MULT block. However, the Scalable and Unified ECP is not as efficient as the Parallelized ECPs in Section 3.3. This result is expected since the hardware implementation of binary field arithmetic is more efficient than prime fields in hardware. Thus, the Parallelized ECPs that only implement binary field arithmetic should be more effi-

cient than the Scalable and Unified ECP, since it must accommodate both prime and binary field arithmetic.

- Finally, the ECP is interfaced with the Microblaze soft-core processor as described in Section 5.2. The integration with software demonstrates the concept of offloading the computationally intensive ECPM operation to a hardware accelerator. The resultant designs also provide a platform for future development of implementing various protocols that can take advantage of the Scalable and Unified ECP.

The main contributions of this thesis are the implementations of the scalable and unified finite field arithmetic blocks that are carefully designed to support various prime and binary field operations. The architecture of the arithmetic blocks and the realization of the ECPM algorithms have been studied and various methods have been investigated to improve the performance of the ECPs. These methods include: the parallelization of the arithmetic and the ECPM algorithm; the use of different algorithms for multiplication, squaring, reduction and inversion; and the use of DSP slices on FPGAs. The improvement in the finite field arithmetic blocks contribute to the improved efficiency of the overall ECPs.

The implementations described in this thesis can be easily integrated into servers or act as a stand-alone system-on-chip in the development of secure network systems. It can support all 15 curves recommended by NIST without the need to reconfigure the hardware and the operations can be done at high speeds. Since the entire implementation of ECC operations and schemes is enclosed in a single FPGA, the design is highly portable and can be migrated to newer and more advanced FPGAs in the future. The techniques used to design the finite field arithmetic blocks can also be used in other application where finite field arithmetic is required, such as error correction codes.

Chapter 7
Future Work

This chapter describes some potential future work that extends from the current work described in this thesis. Section 7.1 discusses the τNAF conversion algorithms in more detail and the possible integration with the scalable and unified ECP. Section 7.2 discusses the topic of post-quantum cryptography. Section 7.3 discusses identity-based encryption (IBE).

7.1 Koblitz τNAF Conversion

In the Koblitz ECPs in Chapter 3 and Scalable and Unified ECP in Koblitz mode described in Section 5.1, the input for the value of k is assumed to have been converted from binary to τ-adic non-adjacent-form (τNAF). The reason for the exclusion of the evaluation of the conversion in the ECP is because the main goal of this thesis is in the design and implementation of the ECP to evaluate the ECPM operation. Furthermore, as explained in [70], for some cryptosystems, a random τNAF number can be generated for use in the ECPM. Nevertheless, as a potential future work to this thesis, the inclusion of the τNAF converter in either software or hardware can extend the capabilities of the ECP.

Solinas [58] describes an algorithm for τNAF conversion and is shown in Algorithm 7.1. In the algorithm, $\mu = (-1)^{1-a}$, where a is defined in Equation (2.4). Solinas [58] shows that if Algorithm 7.1 is used by setting $r_0 = k$ and $r_1 = 0$, the Hamming weight of the resultant τNAF converted k is $2m/3$, where m is the bit length of k. Using a simple double-and-add ECPM algorithm would require $m/2$ PADD. Thus, by using the τNAF converted k, the PDBL operations are replaced by PFRB operations, which are simply squarings, at the expense of the increase in PADD operations. This trade-off is undesirable, so [58] shows a method of modifying the value of k prior to applying Algorithm 7.1.

Algorithm 7.1 τNAF conversion algorithm

```
Input: integers $r_0$ and $r_1$
Output: $\tau$NAF$(r_0 + r_1\tau)$
  $c_0 \leftarrow r_0$, $c_1 \leftarrow r_1$, $S \leftarrow \langle\rangle$
  while $c_0 \neq 0$ or $c_1 \neq 0$ do
    if $c_0$ is odd then
      $u \leftarrow 2 - (c_0 - 2c_1 \bmod 4)$
      $c_0 \leftarrow c_0 - u$
    else
      $u \leftarrow 0$
    end if
    Prepend $u$ to $S$
    $(c_0, c_1) \leftarrow (c_1 + \mu c_0/2, -c_0/2)$
  end while
  return $S$
```

Solinas [58] shows that given $\delta = (\tau^m - 1)/(\tau - 1)$, if $\gamma = \rho \pmod{\delta}$, then $\gamma P = \rho P$. Therefore, in terms of the ECPM operation, γ and ρ are equivalent. Thus, the reduced τNAF conversion algorithm evaluates $\rho = k \bmod \delta$ followed by $S = \tau\text{NAF}(\rho)$. By doing so, Solinas shows that the Hamming weight of S is approximately $m/3$.

As pointed out in [70], one of the drawbacks of the reduced τNAF algorithm proposed by Solinas [58] is the complex multiplications and divisions required for the $\pmod{\delta}$ operation. In [58], it also shows that if $\gamma = \rho \pmod{\tau^m - 1}$, then $\gamma P = \rho P$. In [70], the authors explain that by taking $\pmod{\tau^m - 1}$ instead of $\pmod{\delta}$, the reduction algorithm can be greatly simplified by the following:

$$\begin{aligned} k &= (d_0 + d_1)\tau^m + b_0 + b_1\tau \\ &= (d_0 + d_1\tau)(\tau^m - 1) + b_0 + d_0 + (b_1 + d_1)\tau \end{aligned} \tag{7.1}$$

Thus, the reduction of $k \pmod{\tau^m - 1} = (b_0 + d_0) + (b_1 + d_1)\tau$ can be evaluated by repeatedly dividing k by τ and rearranging the coefficients of the quotient and remainder. Since the division by τ only involves additions and divisions by 2, its implementation is much simpler compared to the reduction proposed in [58]. The authors in [70] explain that using this reduction algorithm, named lazy reduction, followed by the τNAF conversion algorithm in Algorithm 7.1, the maximum length of the converted value is $m + 4$.

The work in [71] further improves on the performance of the work in [70] by developing the double lazy reduction and double τNAF algorithm that generates 2 τNAF converted digits of k at a time. By doing so, the hardware implementation of [71] can complete the conversion in almost half of the time as in [70].

The implementations provided in [70] and [71] are made for 1 specific curve at a time. By adopting these τNAF conversion techniques and the scalable finite field arithmetic described in this thesis, a scalable τNAF converter can be implemented to further improve the capabilities of the ECP.

7.2 Post-Quantum Cryptography

Post-quantum cryptography refers to a branch of cryptography that focuses on the cryptographic algorithms that are still secure with the birth of quantum computers. The first paragraph in [72] describes post-quantum cryptography very adequately by posing the questions:

> "Imagine that its fifteen years from now and someone announces the successful construction of a large quantum computer. The *New York Times* runs a front-page article reporting that all of the public-key algorithms used to protect the Internet have been broken. Users panic. What exactly will happen to cryptography?"

In [72], the authors explain that using certain quantum algorithms, such as Shor's algorithm or Grover's algorithm, many of the most popular public-key cryptography can be broken with the existence of a quantum computer. However, there are several classes of algorithms that current research has shown to be unaffected by these quantum algorithms:

- Hash-based cryptography
- Code-based cryptography
- Lattice-based cryptography
- Multivariate-quadratic equations cryptography
- Secret-key cryptography

One of the reasons these algorithms are not currently in use is the efficiency of the algorithm. In [72], the authors explain that in order to provide a similar level of security as a few-thousand-bit RSA, the McEliece algorithm (example of a code-based cryptography) requires close to a million bits. Thus, research in post-quantum cryptography involves improving the efficiency of currently known algorithms that are resistant against quantum algorithms.

In particular, code-based cryptography is based on binary linear codes that are commonly used in communications for error correction. Recent works in [73, 74, 75] have shown architectures implemented on FPGAs for the McEliece cryptography. Since binary linear codes are based on binary field operations, it may be possible to use some of the techniques and architectures for the finite field arithmetic blocks described in this thesis.

7.3 Identity-Based Encryption

Identity-based encryption (IBE) was originally proposed by Shamir in 1984 [76]. The main idea of the scheme is the ability to encrypt a message by using an arbitrary string. The original motivation was to be implemented on e-mail systems, where the intended recipient's email address is to be used as the public key. In order for the intended recipient to decrypt the message, he or she requests a private key from a trusted third party, called the Private Key Generator (PKG), and uses the private key to decrypt the received message.

IBE involves the use of 4 algorithms:

- **Setup:** The PKG runs this algorithm to set up the environment and generate the master key that is used to derive the users' private keys.
- **Extract:** This algorithm uses the master key and the arbitrary string used in the encryption to generate the private key for the user.
- **Encrypt:** This algorithm generates the ciphertext using the arbitrary string and the message to be encrypted.
- **Decrypt:** This algorithm returns the original message using the private key generated from the Extract algorithm and the ciphertext generated from the Encrypt algorithm.

In [77], the authors propose the use of Weil pairings on elliptic curves in the implementation of IBE. In [78], the authors also make use of pairings on elliptic curves to realize IBE.

Similar to the ECPM operation, the algorithms used for IBE are also defined over finite field arithmetic. These operations involve a similar set of instructions as the ones implemented by the scalable and unified finite field arithmetic blocks described in this thesis. Furthermore, IBE using pairing-based cryptography schemes operate on prime [79], binary [80] or ternary fields [81]. This thesis has shown an efficient implementation of combining prime and binary fields on the same device. The finite field arithmetic blocks may be further extended to ternary fields to support efficient implementations of IBE as the design in [82].

References

[1] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[2] V. Gupta, D. Stebila, S. Fung, S. Chang, N. Gura, and H. Eberle, "Speeding up secure web transactions using elliptic curve cryptography," in *Cryptography, The 11th Network and Systems Security Symposium*, 2004, pp. 231–239.

[3] W. Chou, "Inside SSL: accelerating secure transactions," *IT Professional*, vol. 4, no. 5, pp. 37–41, September/October 2002.

[4] Broadcom, "BCM800 Produc Brief: CrytoNetX$^{\text{TM}}$ SSL800 Accelerator Adapter," February 2004. [Online]. Available: http://www.broadcom.com/collateral/pb/SSL800-PB03-R.pdf

[5] ——, "BCM4000 Produc Brief: CrytoNetX$^{\text{TM}}$ SSL4000 Accelerator Adapter," February 2004. [Online]. Available: http://www.broadcom.com/collateral/pb/SSL4000-PB03-R.pdf

[6] Elliptic Technologies, "SPP-200 Product Brief: SSL/TLS/DTLS PDU Processor," 2012. [Online]. Available: https://elliptictech.com/images/stories/product_briefs/SPP-200_SSL_TLS_DTLS_PDU.pdf

[7] ——, "SPP-330 Product Brief: IPSec/TLS Multi-protocol PDU Processor," 2013. [Online]. Available: https://elliptictech.com/images/stories/product_briefs/SPP-330_ESP_AH_TLS_Multi-PDU_Processor_v1p0.pdf

[8] ——, "CLP-17 Product Brief: Binary Elliptic Curve Point Multiplier Core," 2011. [Online]. Available: https://elliptictech.com/images/stories/product_briefs/CLP-17_Binary_ECC_PMult.pdf

[9] ——, "CLP-300 Product Brief: Public Key Accelerator," 2011. [Online]. Available: https://elliptictech.com/images/stories/product_briefs/CLP-300_PKA.pdf

[10] V. Miller, "Use of elliptic curves in cryptography," in *CRYPTO85: Proceedings of the Advances in Cryptology.* Springer–Verlag, 1986, pp. 417–426.

[11] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[12] R. 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, February 1978.

[13] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," RFC 4492 (Informational), Internet Engineering Task Force, May 2006, updated by RFC 5246. [Online]. Available: http://www.ietf.org/rfc/rfc4492.txt

[14] Y. Zhang, D. Chen, Y. Choi, L. Chen, and S. Ko, "A high performance ECC hardware implementation with instruction-level parallelism over GF(2^{163})," *Microprocessors and Microsystems*, vol. 34, no. 6, pp. 228–236, October 2010.

[15] C. Kim, S. Kwon, and C. Hong, "FPGA implementation of high performance elliptic curve cryptographic processor over GF(2^{163})," *Journal of Systems Architecture*, vol. 54, no. 10, pp. 893–900, 2008.

[16] K. Sakiyama, L. Batina, and B. Preneel, "High-performance public-key cryptoprocessor for wireless mobile applications," *Mobile Networks and Applications*, vol. 12, no. 4, pp. 245–258, 2010.

[17] M. Hassan and M. Benaissa, "A scalable hardware/software co-design for elliptic curve cryptography on PicoBlaze microcontroller," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, June 2010, pp. 2111–2114.

[18] National Institute of Standards and Technology, "Recommended Elliptic Curves for Federal Government Use," July 1999.

[19] Standards for Efficient Cryptography, "SEC 2: Recommended Elliptic Curve Domain Parameters," July 2000.

[20] Federal Information Processing Standard, "FIPS PUB 186-3: Digital Signature Standard (DSS)," June 2009.

[21] National Security Agency (NSA), "The Case for Elliptic Curve Cryptography," January 2009. [Online]. Available: http://www.nsa.gov/business/programs/elliptic_curve.shtml

[22] E. Savas, A. F. Tenca, and C. K. Koc, "A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop*, ser. Lecture Notes in Computer Science, vol. 1965. Springer, August 2000, pp. 277–292.

[23] E. Savas, A. Tenca, M. Ciftcibasi, and C. Koc, "Multiplier architectures for $GF(p)$ and $GF(2^n)$," *IEE Proceedings on Computers and Digital Techniques*, vol. 151, no. 2, pp. 147–160, March 2004.

[24] W. Chelton and M. Benaissa, "A scalable $GF(2^m)$ arithmetic unit for application in an ECC processor," in *IEEE Workshop on Signal Processing Systems*, October 2004, pp. 355–360.

[25] K. Tanimura, R. Nara, S. Kohara, K. Shimizu, Y. Shi, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "Scalable unified dual-radix architecture for Montgomery multiplication in $GF(P)$ and $GF(2^n)$," in *Asia and South Pacific Design Automation Conference*, March 2008, pp. 697–702.

[26] C. Chiou, C.-Y. Lee, and J.-M. Lin, "Unified dual-field multiplier in $GF(P)$ and $GF(2^k)$," *IET Information Security*, vol. 3, no. 2, pp. 45–52, June 2009.

[27] J. Goodman and A. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808–1820, November 2001.

[28] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 449–460, April 2003.

[29] Y. Wang, D. L. Maskell, and J. Leiwo, "A unified architecture for a public key cryptographic coprocessor," *Journal of Systems Architecture*, vol. 54, pp. 1004–1016, 2008.

[30] J.-Y. Lai and C.-T. Huang, "Elixir: High-Throughput Cost-Effective Dual-Field Processors and the Design Framework for Elliptic Curve Cryptography," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1567–1580, 2008.

[31] ——, "A Highly Efficient Cipher Processor for Dual-Field Elliptic Curve Cryptography," *IEEE Transactions on Circuits and Systems—Part II: Express Briefs*, vol. 56, no. 5, pp. 394–398, 2009.

[32] ——, "Energy-Adaptive Dual-Field Processor for High-Performance Elliptic Curve Cryptographic Applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1512–1517, 2011.

[33] J.-H. Chen, M.-D. Shieh, and W.-C. Lin, "A High-Performance Unified-Field Reconfigurable Cryptographic Processor," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1145–1158, August 2010.

[34] Y.-L. Chen, J.-W. Lee, P.-C. Liu, H.-C. Chang, and C.-Y. Lee, "A dual-field elliptic curve cryptographic processor with a radix-4 unified division unit," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2011, pp. 713–716.

[35] J.-W. Lee, S.-C. Chung, H.-C. Chang, and C.-Y. Lee, "Efficient Power-Analysis-Resistant Dual-Field Elliptic Curve Cryptographic Processor Using Heterogeneous Dual-Processing-Element Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 1, pp. 49–61, 2014.

[36] M. Benaissa and W. M. Lim, "Design of flexible $GF(2^m)$ elliptic curve cryptography processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 6, pp. 659–662, 2006.

[37] M. Hassan and M. Benaissa, "Low Area - Scalable Hardware/Software Co-design for Elliptic Curve Cryptography," in *3rd International Conference on New Technologies, Mobility and Security (NTMS)*, December 2009, pp. 1–5.

[38] M. N. Hassan and M. Benaissa, "Embedded Software Design of Scalable Low-Area Elliptic-Curve Cryptography," *IEEE Embedded Systems Letters*, vol. 1, no. 2, pp. 42–45, 2009.

[39] M. Hassan and M. Benaissa, "Flexible Hardware/Software Co-design for Scalable Elliptic Curve Cryptography for Low-Resource Applications," in *21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, July 2010, pp. 285–288.

[40] M. Morales-Sandoval, C. Feregrino-Uribe, R. Complido, and I. Algredo-Badillo, "A reconfigurable $GF(2^m)$ elliptic curve cryptographic coprocessor," in *2011 VII Southern Conference on Programmable Logic (SPL)*, April 2011, pp. 209–214.

[41] C. McIvor, M. McLoone, and J. McCanny, "Hardware Elliptic Curve Cryptographic Processor Over $GF(p)$," *IEEE Transactions on Circuits and Systems—Part I: Regular Papers*, vol. 53, no. 9, pp. 1946–1957, 2006.

[42] K. Ananyi, H. Alrimeih, and D. Rakhmatov, "Flexible Hardware Processor for Elliptic Curve Cryptography Over NIST Prime Fields," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 8, pp. 1099–1112, 2009.

[43] M. Varchola, T. Güneysu, and O. Mischke, "MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, Nov 2011, pp. 204–210.

[44] K. Järvinen and J. Skyttä, "Fast point multiplication on Koblitz curves: Parallelization method and implementations," *Microprocessors and Microsystems*, vol. 33, pp. 106–116, 2009.

[45] T. Güneysu and C. Paar, "Ultra High Performance ECC over NIST Primes on Commercial FPGAs," in *Cryptographic Hardware and Embedded Systems CHES 2008*, ser. Lecture Notes in Computer Science, E. Oswald and P. Rohatgi, Eds. Springer Berlin Heidelberg, 2008, vol. 5154, pp. 62–78.

[46] S. Ghosh, M. Alam, D. R. Chowdhury, and I. S. Gupta, "Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks," *Computers and Electrical Engineering*, vol. 35, pp. 329–338, 2008.

[47] D. M. Schinianakis, A. P. Fournaris, H. E. Michail, A. P. Kakarountas, and T. Stouraitis, "An RNS Implementation of an F_p Elliptic Curve Point Multiplier," *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, vol. 56, no. 6, pp. 1202–1213, 2009.

[48] M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, and K. Navi, "Efficient RNS Implementation of Elliptic Curve Point Multiplication Over GF(p)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 8, pp. 1545–1549, 2013.

[49] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Efficient Elliptic Curve Point Multiplication Using Digit-Serial Binary Field Operations," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 1, pp. 217–225, 2013.

[50] K. C. C. Loi and S.-B. Ko, "Improvements for High Performance Elliptic Curve Cryptosystem Processor over $GF(2^{163})$," in *International Symposium on Electronic System Design (ISED)*, December 2012, pp. 140–144.

[51] ——, "High Performance Scalable Elliptic Curve Cryptosystem Processor for Koblitz Curves," *Microprocessors and Microsystems*, vol. 37, no. 4–5, pp. 394–406, 2013.

[52] ——, "High performance scalable elliptic curve cryptosystem processor in $GF(2^m)$," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2013, pp. 2585–2588.

[53] K. C. C. Loi, S. An, and S.-B. Ko, "FPGA Implementation of Low Latency Scalable Elliptic Curve Cryptosystem Processor in $GF(2^m)$," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, June 2014, pp. 822–825.

[54] K. C. C. Loi and S.-B. Ko, "Scalable Elliptic Curve Cryptosystem FPGA Processor for NIST Prime Curves," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2014, accepted for future publication.

[55] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography.* New York, NY, USA: Springer-Verlag, 2004.

[56] W. Diffie and M. Hellman, "New direction in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[57] J. Lopez and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation," in *CHES99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems.* Springer-Verlag, 1999, pp. 316–327.

[58] J. Solinas, "Efficient Arithmetic on Koblitz Curves," *Designs, Codes and Cryptography*, vol. 19, pp. 195–249, 2000.

[59] G. N. Selimis, A. P. Fournaris, H. E. Michail, and O. Koufopavlou, "Improved throughput bit-serial multiplier for $GF(2^m)$ fields," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, January 1987.

[60] A. Reyhani-Masoleh and M. A. Hasan, "Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over $GF(2^m)$," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 945–959, August 2004.

[61] C. H. Kim, C. P. Hong, and S. Kwon, "A Digit-Serial Multiplier for Finite Field $GF(2^m)$," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 476–483, April 2005.

[62] J. Großschädl, S. Tillich, P. Ienne, L. Pozzi, and A. K. Verma, "When Instruction Set Extensions Change Algorithm Design: A Study in Elliptic Curve Cryptography," in *4th Workshop on Application Specific Processors*, 2005, pp. 2–9.

[63] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, 1988.

[64] V. Dimitrov and K. Järvinen, "Another Look at Inversions over Binary Fields," in *2013 IEEE 21st Symposium on Computer Arithmetic*, July 2013, pp. 211–218.

[65] Xilinx, "Virtex-5 Family Overview," February 2009. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[66] ——, "Virtex-5 FPGA XtremeDSP Design Considerations: User Guide," January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug193.pdf

[67] K. C. C. Loi and S.-B. Ko, "Parallelization of Scalable Elliptic Curve Cryptosystem Processors in $GF(2^m)$," *IEEE Transactions on Computers*, 2013, draft submitted for review.

[68] ——, "Efficient Scalable and Unified Elliptic Curve Cryptography Coprocessor using DSP Slices on FPGAs," *IEEE Transactions on Industrial Electronics*, 2015, draft submitted for review.

[69] Xilinx, "LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)," April 2010. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf

[70] B. B. Brumley and K. U. Järvinen, "Conversion Algorithms and Implementations for Koblitz Curve Cryptography," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 81–92, 2010.

[71] J. Adikari, V. S. Dimitrov, and K. U. Järvinen, "A Fast Hardware Architecture for Integer to τNAF Conversion for Koblitz Curves," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 732–737, 2012.

[72] D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds., *Post-Quantum Cryptogrpahy.* Berlin Heidelberg: Springer-Verlag, 2009.

[73] T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar, "Microeliece: Mceliece for embedded devices," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds. Springer Berlin Heidelberg, 2009, vol. 5747, pp. 49–64.

[74] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert, "A Novel Cryptoprocessor Architecture for the McEliece Public-Key Cryptosystem," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1533–1546, Nov 2010.

[75] S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede, "A Speed Area Optimized Embedded Co-processor for McEliece Cryptosystem," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July 2012, pp. 102–108.

[76] A. Shamir, "Identity-based cryptosystems and signature schemes," in *Advances in Cryptology*, ser. Lecture Notes in Computer Science, G. Blakley and D. Chaum, Eds. Springer Berlin Heidelberg, 1985, vol. 196, pp. 47–53.

[77] D. Boneh and M. Franklin, "Identity-Based Encryption from the Weil Pairing," in *Advances in Cryptology—CRYPTO 2001*, ser. Lecture Notes in Computer Science, J. Kilian, Ed. Springer Berlin Heidelberg, 2001, vol. 2139, pp. 213–229.

[78] R. Sakai and M. Kasahara, "ID based Cryptosystems with Pairing on Elliptic Curve." [Online]. Available: http://eprint.iacr.org/2003/054

[79] A. Barenghi, G. Bertoni, L. Breveglieri, and G. Pelosi, "A FPGA Coprocessor for the Cryptographic Tate Pairing over Fp," in *Fifth International Conference on Information Technology: New Generations (ITNG), 2008*, April 2008, pp. 112–119.

[80] R. Ronan, C. O. hEigeartaigh, C. Murphy, M. Scott, and T. Kerins, "FPGA acceleration of the tate pairing in characteristic 2," in *IEEE International Conference on Field Programmable Technology (FPT), 2006*, Dec 2006, pp. 213–220.

[81] L. Amaral, G. Araujo, and J. Lopez, "HW/SW co-design of Identity-Based Encryption using a custom instruction set," in *International Conference on Field-Programmable Technology (FPT), 2009*, Dec 2009, pp. 510–513.

[82] T. Vejda and J. Großschädl and D. Page, "A Unified Multiply/Accumulate Unit for Pairing-Based Cryptography over Prime, Binary and Ternary Fields," in *2011 14th Euromicro Conference on Digital System Design (DSD)*, Aug 2011, pp. 658–666.

Appendix A

Binary Fields Reduction Algorithms

The following algorithms are modified from [55] and are used for binary finite field reduction. The algorithms split the operand, Z, into 32-bit digits and it is modified to output the result least-significant-digit-first (LSD-first) instead of most-significant-digit-first (MSD-first). In the algorithms, $\oplus$ is modulo 2 addition, (A, B, C) represents a concatenation of A, B, and C, $>>$ and $<<$ represents right shift and left shift operations, & is a bit-wise logical AND operation and '0x' precedes a hexadecimal number.

Algorithm A.1 Reduction by $P(t) = t^{163} + t^7 + t^6 + t^3 + 1$

```
Input: $Z = (Z_{11}, \dots, Z_1, Z_0)$
Output: $Z \mod P(t) = (Z_5, \dots, Z_1, Z_0)$
  for $i$ from 6 to 11 do
    if $i \le 9$ then
      $(Z_{i-4}, Z_{i-5}, Z_{i-6}) = (Z_{i-4}, Z_{i-5}, Z_{i-6}) \oplus (Z_i << 29) \oplus (Z_i << 32) \oplus (Z_i << 35) \oplus (Z_i << 36)$
    else if $i = 10$ then
      $(T_0, Z_5, Z_4) = (0, Z_5, Z_4) \oplus (Z_i << 29) \oplus (Z_i << 32) \oplus (Z_i << 35) \oplus (Z_i << 36)$
      $(Z_2, Z_1, Z_0) = (Z_2, Z_1, Z_0) \oplus (T_0 << 29) \oplus (T_0 << 32) \oplus (T_0 << 35) \oplus (T_0 << 36)$
    else // $i = 11$
      $(T_1, T_0, Z_5) = (0, 0, Z_5) \oplus (Z_i << 29) \oplus (Z_i << 32) \oplus (Z_i << 35) \oplus (Z_i << 36)$
      $(Z_2, Z_1, Z_0) = (Z_2, Z_1, Z_0) \oplus (T_0 << 29) \oplus (T_0 << 32) \oplus (T_0 << 35) \oplus (T_0 << 36)$
      $(Z_3, Z_2, Z_1) = (Z_3, Z_2, Z_1) \oplus (T_1 << 29) \oplus (T_1 << 32) \oplus (T_1 << 35) \oplus (T_1 << 36)$
    end if
  end for
  // Final reduction
  $T = Z_5 >> 3$
  $Z_0 = Z_0 \oplus (T << 7) \oplus (T << 6) \oplus (T << 3) \oplus T$
  $Z_1 = Z_1 \oplus (T >> 25) \oplus (T >> 26)$
  $Z_5 = Z_5$ & 0x7
  return $Z = (Z_5, \dots, Z_1, Z_0)$
```

Algorithm A.2 Reduction by $P(t) = t^{233} + t^{74} + 1$

```
Input: $Z = (Z_{15}, \ldots, Z_1, Z_0)$
Output: $Z \mod P(t) = (Z_7, \ldots, Z_1, Z_0)$
  for $i$ from 8 to 15 do
    if $i \le 11$ then
      $(Z_{i-4}, Z_{i-5}, Z_{i-6}, Z_{i-7}, Z_{i-8}) = (Z_{i-4}, Z_{i-5}, Z_{i-6}, Z_{i-7}, Z_{i-8}) \oplus (Z_i << 23) \oplus (Z_i << 97)$
    else if $i = 12$ then
      $(T_0, Z_7, Z_6, Z_5, Z_4) = (0, Z_7, Z_6, Z_5, Z_4) \oplus (Z_i << 23) \oplus (Z_i << 97)$
      $(Z_4, Z_3, Z_2, Z_1, Z_0) = (Z_4, Z_3, 0, Z_1, Z_0) \oplus (T_0 << 23) \oplus (T_0 << 97)$
    else if $i = 13$ then
      $(T_1, T_0, Z_7, Z_6, Z_5) = (0, 0, Z_7, Z_6, Z_5) \oplus (Z_i << 23) \oplus (Z_i << 97)$
      $(Z_4, Z_3, Z_2, Z_1, Z_0) = (Z_4, Z_3, Z_2, Z_1, Z_0) \oplus (T_0 << 23) \oplus (T_0 << 97)$
      $(Z_5, Z_4, Z_3, Z_2, Z_1) = (Z_5, Z_4, Z_3, Z_2, Z_1) \oplus (T_1 << 23) \oplus (T_1 << 97)$
    else if $i = 14$ then
      $(T_2, T_1, T_0, Z_7, Z_6) = (0, 0, 0, Z_7, Z_6) \oplus (Z_i << 23) \oplus (Z_i << 97)$
      $(Z_4, Z_3, Z_2, Z_1, Z_0) = (Z_4, Z_3, Z_2, Z_1, Z_0) \oplus (T_0 << 23) \oplus (T_0 << 97)$
      $(Z_5, Z_4, Z_3, Z_2, Z_1) = (Z_5, Z_4, Z_3, Z_2, Z_1) \oplus (T_1 << 23) \oplus (T_1 << 97)$
      $(Z_6, Z_5, Z_4, Z_3, Z_2) = (Z_6, Z_5, Z_4, Z_3, Z_2) \oplus (T_2 << 23) \oplus (T_2 << 97)$
    else // $i = 15$
      $(T_3, T_2, T_1, T_0, Z_7) = (0, 0, 0, 0, Z_7) \oplus (Z_i << 23) \oplus (Z_i << 97)$
      $(Z_4, Z_3, Z_2, Z_1, Z_0) = (Z_4, Z_3, Z_2, Z_1, Z_0) \oplus (T_0 << 23) \oplus (T_0 << 97)$
      $(Z_5, Z_4, Z_3, Z_2, Z_1) = (Z_5, Z_4, Z_3, Z_2, Z_1) \oplus (T_1 << 23) \oplus (T_1 << 97)$
      $(Z_6, Z_5, Z_4, Z_3, Z_2) = (Z_6, Z_5, Z_4, Z_3, Z_2) \oplus (T_2 << 23) \oplus (T_2 << 97)$
      $(Z_7, Z_6, Z_5, Z_4, Z_3) = (Z_7, Z_6, Z_5, Z_4, Z_3) \oplus (T_3 << 23) \oplus (T_3 << 97)$
    end if
  end for
  // Final reduction
  $T = Z_7 >> 9$
  $Z_0 = Z_0 \oplus T$
  $Z_2 = Z_2 \oplus (T << 10)$
  $Z_3 = Z_3 \oplus (T >> 22)$
  $Z_7$ & 0x1FF
  return $Z = (Z_7, \ldots, Z_1, Z_0)$
```

Algorithm A.3 Reduction by $P(t) = t^{283} + t^{12} + t^7 + t^5 + 1$

```
Input: $Z = (Z_{17}, \ldots, Z_1, Z_0)$
Output: $Z \mod P(t) = (Z_8, \ldots, Z_1, Z_0)$
  for $i$ from 9 to 17 do
    if $i \leq 16$ then
      $(Z_{i-8}, Z_{i-9}) = (Z_{i-8}, Z_{i-9}) \oplus (Z_i << 5) \oplus (Z_i << 10) \oplus (Z_i << 12) \oplus (Z_i << 17)$
    else // $i = 17$
      $(T_0, Z_8) = (0, Z_8) \oplus (Z_i << 5) \oplus (Z_i << 10) \oplus (Z_i << 12) \oplus (Z_i << 17)$
      $(Z_1, Z_0) = (Z_1, Z_0) \oplus (T_0 << 5) \oplus (T_0 << 10) \oplus (T_0 << 12) \oplus (T_0 << 17)$
    end if
  end for
  // Final reduction
  $T = Z_8 >> 27$
  $Z_0 = Z_0 \oplus T \oplus (T << 5) \oplus (T << 7) \oplus (T << 12)$
  $Z_8 = Z_8$ & 0x7FFFFFF
  return $Z = (Z_8, \ldots, Z_1, Z_0)$
```

Algorithm A.4 Reduction by $P(t) = t^{409} + t^{87} + 1$

```
Input: $Z = (Z_{25}, \ldots, Z_1, Z_0)$
Output: $Z \mod P(t) = (Z_{12}, \ldots, Z_1, Z_0)$
  for $i$ from 13 to 25 do
    if $i \leq 22$ then
      $(Z_{i-10}, Z_{i-11}, Z_{i-12}, Z_{i-13}) = (Z_{i-10}, Z_{i-11}, Z_{i-12}, Z_{i-13}) \oplus (Z_i << 7) \oplus (Z_i << 94)$
    else if $i = 23$ then
      $(T_0, Z_{12}, Z_{11}, Z_{10}) = (0, Z_{12}, Z_{11}, Z_{10}) \oplus (Z_i << 7) \oplus (Z_i << 94)$
      $(Z_3, Z_2, Z_1, Z_0) = (Z_3, Z_2, Z_1, Z_0) \oplus (T_0 << 7) \oplus (T_0 << 94)$
    else if $i = 24$ then
      $(T_1, T_0, Z_{12}, Z_{11}) = (0, 0, Z_{12}, Z_{11}) \oplus (Z_i << 7) \oplus (Z_i << 94)$
      $(Z_3, Z_2, Z_1, Z_0) = (Z_3, Z_2, Z_1, Z_0) \oplus (T_0 << 7) \oplus (T_0 << 94)$
      $(Z_4, Z_3, Z_2, Z_1) = (Z_4, Z_3, Z_2, Z_1) \oplus (T_1 << 7) \oplus (T_1 << 94)$
    else // $i = 25$
      $(T_2, T_1, T_0, Z_{12}) = (0, 0, 0, Z_{12}) \oplus (Z_i << 7) \oplus (Z_i << 94)$
      $(Z_3, Z_2, Z_1, Z_0) = (Z_3, Z_2, Z_1, Z_0) \oplus (T_0 << 7) \oplus (T_0 << 94)$
      $(Z_4, Z_3, Z_2, Z_1) = (Z_4, Z_3, Z_2, Z_1) \oplus (T_1 << 7) \oplus (T_1 << 94)$
      $(Z_5, Z_4, Z_3, Z_2) = (Z_5, Z_4, Z_3, Z_2) \oplus (T_2 << 7) \oplus (T_2 << 94)$
    end if
  end for
  // Final reduction
  $T = C_{12} >> 25$
  $Z_0 = Z_0 \oplus T$
  $Z_2 = Z_2 \oplus (T << 23)$
  $Z_{12} = Z_{12}$ & 0x1FFFFFF
  return $Z = (Z_{12}, \ldots, Z_1, Z_0)$
```

Algorithm A.5 Reduction by $P(t) = t^{571} + t^{10} + t^5 + t^2 + 1$

```
Input: $Z = (Z_{35}, \ldots, Z_1, Z_0)$
Output: $Z \mod P(t) = (Z_{17}, \ldots, Z_1, Z_0)$
  for $i$ from 18 to 35 do
    if $i \leq 34$ then
      $(Z_{i-17}, Z_{i-18}) = (Z_{i-17}, Z_{i-18}) \oplus (Z_i << 5) \oplus (Z_i << 7) \oplus (Z_i << 10) \oplus (Z_i << 15)$
    else // $i = 35$
      $(T_0, Z_{17}) = (0, Z_{17}) \oplus (Z_i << 5) \oplus (Z_i << 7) \oplus (Z_i << 10) \oplus (Z_i << 15)$
      $(Z_1, Z_0) = (Z_1, Z_0) \oplus (T_0 << 5) \oplus (T_0 << 7) \oplus (T_0 << 10) \oplus (T_0 << 15)$
    end if
  end for
  // Final reduction
  $T = Z_{17} >> 27$
  $C_0 = C_0 \oplus T \oplus (T << 2) \oplus (T << 5) \oplus (T << 10)$
  $C_{17} = C_{17}$ & 0x7FFFFFF
  return $Z = (Z_{17}, \ldots, Z_1, Z_0)$
```

Appendix B

Prime Fields Reduction Algorithms

The following algorithms are taken from [55] and are used for prime field reduction. The algorithms are generated by using a similar analysis as the one shown in Section 2.3.2. In the algorithms, values enclosed in brackets represent that the binary values are concatenated. Furthermore, the input value c breaks down into different digit sizes. For example, if base 2^{64} is used, the digits, c_i, are 64-bit integers.

Algorithm B.1 Reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$

```
Input: $c = (c_5, c_4, \ldots, c_0)$ in base $2^{64}$ with $0 \le c < p_{192}^2$
Output: $c \mod p_{192}$
  Define 192-bit integers:
  $s_1 = (c_2, c_1, c_0)$, $s_2 = (0, c_3, c_3)$,
  $s_3 = (c_4, c_4, 0)$, $s_4 = (c_5, c_5, c_5)$
  Return $(s_1 + s_2 + s_3 + s_4 \mod p_{192})$
```

Algorithm B.2 Reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$

```
Input: $c = (c_{13}, c_{12}, \ldots, c_0)$ in base $2^{32}$ with $0 \le c < p_{224}^2$
Output: $c \mod p_{224}$
  Define 224-bit integers:
  $s_1 = (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$, $s_2 = (c_{10}, c_9, c_8, c_7, 0, 0, 0)$,
  $s_3 = (0, c_{13}, c_{12}, c_{11}, 0, 0, 0)$, $s_4 = (c_{13}, c_{12}, c_{11}, c_{10}, c_9, c_8, c_7)$,
  $s_5 = (0, 0, 0, 0, c_{13}, c_{12}, c_{11})$
  Return $(s_1 + s_2 + s_3 - s_4 - s_5 \mod p_{224})$
```

Algorithm B.3 Reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

```
Input: $c = (c_{15}, c_{14}, \ldots, c_0)$ in base $2^{32}$ with $0 \le c < p_{256}^2$
Output: $c \mod p_{256}$
  Define 256-bit integers:
  $s_1 = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$, $s_2 = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0)$,
  $s_3 = (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0)$, $s_4 = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$,
  $s_5 = (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$, $s_6 = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$,
  $s_7 = (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$, $s_8 = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13})$,
  $s_9 = (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$
  Return $(s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \mod p_{256})$
```

Algorithm B.4 Reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

```
Input: $c = (c_{23}, c_{22}, \ldots, c_0)$ in base $2^{32}$ with $0 \leq c < p_{384}^2$
Output: $c \mod p_{384}$
  Define 384-bit integers:
  $s_1 = (c_{11}, c_{10}, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,
  $s_2 = (0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, 0, 0)$,
  $s_3 = (c_{23}, c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12})$,
  $s_4 = (c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23}, c_{22}, c_{21})$,
  $s_5 = (c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{20}, 0, c_{23}, 0)$,
  $s_6 = (0, 0, 0, 0, c_{23}, c_{22}, c_{21}, c_{20}, 0, 0, 0, 0)$,
  $s_7 = (0, 0, 0, 0, 0, 0, c_{23}, c_{22}, c_{21}, 0, 0, c_{20})$,
  $s_8 = (c_{22}, c_{21}, c_{20}, c_{19}, c_{18}, c_{17}, c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{23})$
  $s_9 = (0, 0, 0, 0, 0, 0, 0, c_{23}, 0, c_{22}, c_{21}, c_{20}, 0)$,
  $s_{10} = (0, 0, 0, 0, 0, 0, 0, c_{23}, c_{23}, 0, 0, 0)$
  Return $(s_1 + 2s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10} \mod p_{384})$
```

Algorithm B.5 Reduction modulo $p_{521} = 2^{521} - 1$

```
Input: $c = (c_{1041}, c_{1040}, \ldots, c_0)$ in base 2 with $0 \leq c < p_{521}^2$
Output: $c \mod p_{521}$
  Define 521-bit integers:
  $s_1 = (c_{1041}, c_{1040}, \ldots, c_{521})$,
  $s_2 = (c_{520}, c_{519}, \ldots, c_0)$
  Return $(s_1 + s_2 \mod p_{521})$
```