FLEXIBLE HARDWARE DESIGN FOR ELLIPTIC CURVE METHOD OF INTEGER FACTORIZATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUSTAFA HAKAN SOLMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

JULY 2023

Approval of the thesis:

FLEXIBLE HARDWARE DESIGN FOR ELLIPTIC CURVE METHOD OF INTEGER FACTORIZATION

submitted by **MUSTAFA HAKAN SOLMAZ** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Cryptography Department, Middle East Technical University** by,

Prof. Dr. Ayşe Sevtap Selçuk-Kestel
Dean, Graduate School of **Applied Mathematics** ______________

Assoc. Prof. Dr. Oğuz Yayla
Head of Department, **Cryptography** ______________

Prof. Dr. Ersan Akyıldız
Supervisor, **Cryptograpy, METU** ______________

Examining Committee Members:

Prof. Dr. Ferruh Özbudak
Engineering and Natural Sciences, Sabancı University ______________

Prof. Dr. Ersan Akyıldız
Cryptography, METU ______________

Prof. Dr. Cüneyt Bazlamaççı
Computer Engineering, İzmir Institute of Technology ______________

Prof. Dr. Zülfükar Saygı
Mathematics, TOBB ETU ______________

Assoc. Prof. Dr. Oğuz Yayla
Cryptography, METU ______________

Date: ______________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MUSTAFA HAKAN SOLMAZ

Signature :

ABSTRACT

FLEXIBLE HARDWARE DESIGN FOR ELLIPTIC CURVE METHOD OF INTEGER FACTORIZATION

Solmaz, Mustafa Hakan
Ph.D., Department of Cryptography
Supervisor : Prof. Dr. Ersan Akyıldız

July 2023, 87 pages

In most of the electronic communication devices that surround us, advanced cryptographic algorithm needs are implemented on special hardware. These specialized hardware are divided into application-specific integrated circuit (ASIC) and field programmable gate arrays (FPGA). In this thesis we have designed and implemented all arithmetic primitives used in elliptic curve method (ECM) for integer factorization in FPGA platform. These primitives include point addition, point doubling and scalar multiplication of a point on elliptic curve. The curves used for this purpose are defined on prime fields. In the lowest layer there exists modular arithmetic, modular addition, subtraction and multiplications. As the most crucial and time-consuming operation modular multiplication is further studied. A memory and hard multiplier based Montgomery multiplier is designed. These low-level primitives are controlled by a novel micro-instruction controller to obtain scalar point multiplication results. ECM is a factorization method that can be implemented in parallel. To use this fact multiple instances of the whole coprocessor are instantiated in a Zynq based processing subsystem. By this way the ECM cores were easily accessible by an application. We achieved higher synthesis frequencies than similar studies in the literature. By the obtained scalable design it is possible to run the ECM in different FPGAs and obtain as much throughput as the FPGA resources permit.

Keywords: elliptic curve method, prime factorization, Montgomery modular multiplier, modular exponentiation FPGA, elliptic curve co-processor

ÖZ

ELİPTİK EĞRİ YÖNTEMİ İLE ÇARPANLARA AYIRMA İÇİN ESNEK DONANIM TASARIMI

Solmaz, Mustafa Hakan
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Prof. Dr. Ersan Akyıldız

Temmuz 2023, 87 sayfa

Çevremizi saran elektronik haberleşme cihazlarının çoğunda, gelişmiş kriptografik algoritma ihtiyaçları özel donanımlar üzerinde uygulanmaktadır. Bu özel donanımlar, uygulamaya özel tümleşik devre (ASIC) ve sahada programlanabilir kapı dizileri (FPGA) olmak üzere ikiye ayrılmıştır. Bu tez çalışmasında tamsayı çarpanlara ayırma için eliptik eğri yönteminde(ECM) kullanılan tüm aritmetik ilkelleri FPGA platformunda tasarladık ve uyguladık. Bu ilkel öğeler, eliptik eğri üzerindeki bir noktanın nokta toplaması, nokta ikiye katlama ve skaler çarpımını içerir. Bu amaçla kullanılan eğriler asal alanlar üzerinde tanımlanır. En alt katmanda modüler aritmetik, modüler toplama, çıkarma ve çarpma işlemleri bulunur. En önemli ve zaman alıcı işlem olan modüler çarpma daha fazla incelenmiştir. Bir bellek ve sabit çarpan tabanlı Montgomery çarpıcısı tasarlanmıştır. Bu düşük seviyeli ilkel öğeler, skaler nokta çarpma sonuçları elde etmek için yeni bir mikro komut denetleyicisi tarafından kontrol edilmiştir. ECM, paralel olarak uygulanabilen bir çarpanlara ayırma yöntemidir. Tüm yardımcı işlemcinin birden çok örneği, Zynq tabanlı bir işleme alt sisteminde çağrılmıştır. Bu sayede ECM çekirdeklerine bir uygulama ile kolayca erişilebilmektedir. Literatürdeki benzer çalışmalardan daha yüksek sentez frekansları elde edilmiştir. Elde edilen ölçeklenebilir tasarım sayesinde, ECM'yi farklı FPGA'lerda çalıştırmak ve FPGA kaynaklarının izin verdiği ölçüde verim elde etmek mümkündür.

Anahtar Kelimeler: Eliptik eğri yöntemi, asal çarpanlara ayırma, Montgomery modüler çarpıcı, modüler üs alma, FPGA, eliptik eğri yardımcı işlemcisi

To My Family

ACKNOWLEDGMENTS

First and foremost I would like to express my very great appreciation to my thesis supervisor Prof. Dr. Ersan Akyıldız for his patient guidance and continuous encouragement during my PhD study. I also would like to thank to my thesis jury, Prof. Dr. Ferruh Özbudak, Prof. Dr. Cüneyt Bazlamaçcı, Prof. Dr. Zülfükar Saygı and Assoc. Prof. Dr. Oğuz Yayla for their valuable time.

Prof. Dr. Cüneyt Bazlamaçcı was more than a co-advisor during the whole time. His insightful and rigorous perspective always took me further. In last year of the thesis Assoc. Prof. Dr. Oğuz Yayla picked me up and now I am able to complete the thesis with his calm guidance.

No one grows up in a vacuum, neither did I. I would like to thank all my teachers, elders and friends who have guided me in the past, in my old jobs, in my schools and in my new working environment. I will always remember and appreciate their help.

Montgomery modular multiplication, Montgomery curve and ladder which are invented by the brilliant mathematician and computer scientist Peter L. Montgomery are all around in this thesis. As I studied I felt a personal admiration and love to his also humble personality. I am very sorry that he passed away in 2020, let him rest in peace.

Finally I would like thank my father, mother, sisters, brother and nephews for their unlimited support and unconditional love throughout my life. I have four wonderful children Müjde, Yiğit, Orhan and Azra. I thank them all for their existence and for bearing my absence in all the busy years. Last but not least I wish to thank my beloved wife, Berin Nüha, who has stood by me through all my travails, my absences, through all my life. I am so glad to have you, furthermore, I want to rephrase the lyrics by Sagopa and Ceza: "What do I have other than you?"

TABLE OF CONTENTS

ABSTRACT vii

ÖZ ix

ACKNOWLEDGMENTS xiii

TABLE OF CONTENTS xv

LIST OF TABLES xix

LIST OF FIGURES xx

LIST OF ABBREVIATIONS xxii

CHAPTERS

1 INTRODUCTION 1

  1.1 Summary of Thesis Contributions 2

  1.2 Thesis Structure 4

2 MATHEMATICAL BACKGROUND 5

  2.1 Cryptography Basis 5

  2.2 Algebraic Structures and Number Theory Basics 9

  2.3 Square-and-Multiply Modular Exponentiation 14

  2.4 Elliptic Curves 15

3 TECHNICAL BACKGROUND 17

3.1 Development Flow of FPGA and ASIC 19

3.2 Basics of Digital Logic Design 21

3.2.1 Combinational logic circuits 21

3.2.2 Synchronous logic circuits 24

4 FLEXIBLE MONTGOMERY MULTIPLIER WITH BLOCK RAMS 29

4.1 Background and Related Work 30

4.2 Flexible Montgomery Multiplier 31

4.3 Modular Exponentiation 33

4.4 Testing of the Exponentiation Core 37

4.5 Implementation Results and Comparisons 40

4.6 Conclusion and Future Work 42

5 FLEXIBLE COPROCESSOR FOR ELLIPTIC CURVE CRYPTOGRAPHY 43

5.1 Related Work 44

5.2 Flexible Microcontroller Based Arithmetic Architecture . . . 45

5.2.1 Microcontroller Design Details 45

5.2.2 Scalar Point Multiplication 48

5.3 Implementation Results and Comparisons 52

5.4 Conclusion and Future Work 53

6 ELLIPTIC CURVE METHOD AS A PERIPHERAL TO EMBEDDED FPGA PROCESORS 55

6.1 Integer Factorization with Elliptic Curve Method 56
    6.1.1 Pollard's (p-1) Method of Factoring 56
    6.1.2 Elliptic Curve Method of Factoring 59
6.2 Use of ECM peripheral in a Processing Subsystem 63
6.3 Parallel Instantiation and Managing ECM Cores 68
6.4 Implementation Result and Comparisons 69
7 CONCLUSION AND FUTURE WORK 71
REFERENCES 73

APPENDICES

A ELLIPTIC CURVE METHOD APPENDICES 79
A.1 192 Bit factorization with GMP-ECM with $\sigma = 7$ 79
A.2 192 Bit factorization with GMP-ECM with $\sigma = 10$ 82
CURRICULUM VITAE 87

LIST OF TABLES

TABLES

Table 2.1 Cayley table of GF(5) 14

Table 3.1 Inverter truth table and circuit symbol 22

Table 3.2 Important two-input binary circuit *(logic gates)* truth tables and circuit symbols 22

Table 3.3 Multiplexer truth table 23

Table 4.1 Performance of proposed Montgomery multiplier alternatives (Throughput computed for 1024 bits) 41

Table 5.1 some examples of JMP instruction 47

Table 5.2 `ram_m` memory map 51

Table 5.3 `ram_ab` memory map 51

Table 6.1 Summary of timing performance of flexible ECM core 69

Table 6.2 Comparison of ECM Phase 1 implementations for composite N of 192 bits with smoothness bound $B_1 = 960$ (Result in (*) are taken from [22]) 70

LIST OF FIGURES

FIGURES

Figure 2.1 Communication model for secret-key cryptosystems 6

Figure 2.2 Communication model for public-key cryptosystems 7

Figure 3.1 FPGA and ASIC design flow 19

Figure 3.2 Multiplexer circuit (left) and logic symbol (right) 23

Figure 3.3 Combinational signals rtl schematics 24

Figure 3.4 Combinational signals simulation output 24

Figure 3.5 Sequential gates: D-latch, D-flip-flops 25

Figure 3.6 Synchronous signals rtl schematics 26

Figure 3.7 Synchronous signals simulation output 27

Figure 4.1 Montgomery modular multiplication algorithm data path 31

Figure 4.2 Montgomery modular multiplier with ram blocks 32

Figure 4.3 Memory occupation before execution 33

Figure 4.4 Memory occupation after execution 34

Figure 4.5 Exponentiation control block 35

Figure 4.6 Exponentiation control state machine 36

Figure 4.7 Test interface of the exponentiation core 37

Figure 4.8 Communication interface with the test module 38

Figure 4.9 Qt test program 192 bits modular multiplication running example . 38

Figure 4.10 Qt test program 512 bits modular multiplication running example . 39

Figure 4.11 Elapsed time counter design for each arithmetic operation 40

Figure 4.12 Utilization results of Montgomery multiplier with 3 DSP, 3 block rams 40

Figure 5.1 Microcontroller based ECC Architecture Block Diagram 46

Figure 5.2 Micro-instruction formats 47

Figure 5.3 Flexible elliptic curve coprocessor RTL schematics 51

Figure 5.4 Utilization results of ECC scalar multiplication with microcontroller 52

Figure 5.5 Qt test program 192-bits scalar point multiplication running example 53

Figure 6.1 LCM upto smoothness bound computation with Pari-GP 57

Figure 6.2 Online elliptic curve demonstration tool [6] 60

Figure 6.3 Zynq SOC(System on Chip) with two instances of ECM coprocessors 63

Figure 6.4 Zynq 7000 SOC(System on Chip) Overview [51] 64

Figure 6.5 ECM peripheral compile-time parameters 67

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
CMOS	Complementary Metal Oxide Semiconductor
COTS	Commercially of the Shelf
CPLD	Complex Programmable Logic Device
DES	Data Encryption Standard
DLP	Discrete Logarithm Problem
DPRAM	Dual-port Random Access Memory
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECM	Elliptic Curve Method
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
GCD	Greatest Common Divisor
GF	Galois Field
GMP	GNU Multiple Precision
GNU	GNU's Not Unix
GUI	Graphical User Interface
HDL	Hardware Description Languages
IMM	Interleaved Modular Multiplication
LUT	Lookup Table
MMM	Montgomery Modular Multiplication
MPIR	Multiple Precision Integers and Rationals
NCAP	New Car Assessment Programme

NIST	National Institute of Standards and Technology
PAL	Programmable Array Logic
PLD	Programmable Logic Device
PL	Programmable Logic
PS	Processing System
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RSA	Rivest-Shamir-Adleman encryption
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large-Scale Integration

CHAPTER 1

INTRODUCTION

With the advancement of technology in many fields and especially in semiconductor manufacturing area, new communication channels became widespread. It is estimated that 7.33 billion people have mobile phones, 6.92 of which are smartphones with touch screen, mobile applications and internet access [8]. This respectively constitutes 91.53% and 86.41% of total number of human being in the World. Along with this environment the need for securing personal data and communication became an evident fact. In the case of mobile communication, the embedded microprocessors need to consume low power, have low price, hence will have limited computing power. While this is the case, they need to comply with the cryptographic protocols which demand computationally expensive execution costs. In the meantime these cryptographic protocols need to evolve such that they do not consume data bandwidth, hence have low latency. All these constraints created an immense research area for mathematicians, electrical and computer engineers. Special purpose hardware implementations of cryptographic algorithms came into play as a solution.

Custom hardware implementations of cryptographic algorithms can be designed more favorable with respect to microprocessor-based implementations. There are two types of special purpose implementations. These are *Field Programmable Gate Array* (FPGA) and *Application Specific Integrated Circuit* (ASIC). Actually the name, Field Programmable Gate Array, defines the technology in a very accurate way. On the one end there are programmable microcontrollers where there is a fixed hardware on which different programs are “programmed” to solve different problems. On the other hand there are custom-built application specific integrated circuits where there is a circuit which is built to solve a dedicated problem. This circuit consists of thousands to billions of “gate arrays” to accomplish the desired functionality. FPGAs combine these two desired properties: programmability and gate arrays. By the help of hardware description languages, one can define a circuit for a desired functionality. This definition process is almost the same for ASIC and FPGA technologies. However, ASIC chips are very expensive to produce, and their production cycle is very long, from weeks to months. In FPGA technology the designed circuit is programmed and becomes ready to work in a few minutes.

While cryptography is aiming to satisfy security needs, cryptanalysis on the other hand is known as the art of revealing secret information. Although cryptanalysis seems to be destructive effort it is an essential concept for maintaining the effectiveness of cryptographic algorithms. It may be considered as an NCAP test of cryptographic algorithms, where they are evaluated under stress.

This thesis focuses on hardware implementations of both cryptographic and cryptanalysis applications. For the case of cryptographic application a flexible coprocessor is implemented for elliptic curve cryptography. The implementation is suited for different FPGA platforms and adjustable for different FPGA resources. It is also suitable for different elliptic curve equations and also prime field arithmetic.

For the case of cryptanalysis application elliptic curve method (ECM) is implemented on FPGA platform for integer factorization. ECM, invented by H.W. Lenstra, is a prime factorization method which can be made parallel to solve the problem. Advancement in the semiconductor industry resulted in new opportunities for these parallel execution alternatives. FPGA technology is one such alternative that makes it possible for an end-user to run his or her application in a very high speed and high throughput fashion.

1.1 Summary of Thesis Contributions

We studied hardware design for cryptanalysis applications during the course of this PhD thesis. Two applications have come to the forefront in the last two decades when we discuss cryptanalysis: brute force attacking the DES algorithm (data encryption standard) and prime factorization of large composite numbers. In the first case the whole key space is exhaustively searched for the correct key when a plain text and corresponding ciphertext is given. For the second case, prime factors of a large composite number are "searched" through different mathematical algorithms. In both cases dedicated computers, where sometimes we can define them as dedicated electronic circuits, are used to accomplish the result. The common point in two cases is that the solution algorithms are stated so that parallel execution of smaller parts of the algorithm is possible.

In this thesis we focused on the solution of the second problem, integer factorization, on FPGA platforms. As mentioned above, FPGAs are very convenient platforms for parallel execution of different instances of a task. Because of this an algorithm which permits parallelization is chosen, namely the elliptic curve method. Implementation of integer factorization based on the elliptic curve method (ECM) has been studied in the literature before but the most recent work dates back to the 2010s [19, 54]. In the meantime FPGA technology has drastically advanced. One contribution of this study will be to obtain benchmarks on new FPGA architectures. As part of the FPGA technology new arithmetic accelerators are introduced inside FPGAs for signal processing and other applications. These accelerators also found application areas in prime factorization architectures. Another contribution of the thesis will be to show the affect of these primitives in the whole architecture.

As the size, type and vendors of FPGAs grow their design methodologies also have expanded especially in the last five years. Hardware and software co-design methodologies are introduced by different FPGA vendors. In this study these tools and methods are used to obtain working solutions.

As part of the above studies we have made a full implementation of elliptic curve arithmetic primitives using generic FPGA resources. The lowest layer of arithmetic primitives, modular addition, subtraction and multiplication are implemented on FPGA resources with hard-wired control logic. The above arithmetic layer which includes point addition, point doubling and scalar multiplication are implemented with an indigenous micro-instruction architecture. This architecture has a basic set of instructions for register transfers and flow control of elliptic curve point operations. By the use of this micro-instruction architecture different curve equations were realizable on hardware with minimal effort.

For cipher breaking and massive parallel implementations the processing cores need to be instantiated many times. In recent years specially designed platforms like Copacabana [26] or RIVYERA X-32G1 [7] are introduced as examples of these platforms. As new FPGAs which contain embedded processors inside them, the same solutions can be applied with commercially of the shelf (COTS) FPGA platforms. As a demonstration of this idea we have developed a brute-force attacking implementation with Kintex-6 family of FPGAs [38]. In this thesis we will also use COTS FPGA platforms and propose hardware-software architectures for integer factorization by elliptic curve method.

1.2 Thesis Structure

The thesis is structured as follows: Chapter 2 gives basic definitions and facts about mathematical background. Chapter 3 introduces FPGA architectures and technologies that are used in the implementations. In Chapter 4 we investigate the details of Montgomery modular multiplication and propose an architecture which reads and writes operands and modulus from block memories. By this way a compact solution which can do any practical size of modular multiplication is obtained. An exponentiation controller is also included in the design to demonstrate the Montgomery modular multiplication in a real application. In Chapter 5 the previous architecture is enhanced to enable computations of elliptic curve field operations. A flexible novel micro-instruction controller is designed for scheduling point addition, point doubling and scalar point multiplication. The design space of FPGA resources are explored in this section. Different versions of scalar multiplier core are obtained, their performances are explored. Chapter 6 explores the employment of scalar multiplication circuit as a peripheral to embedded processors in FPGAs. Multiple instances of ECM cores are instantiated as peripherals and their performance results are measured. Finally, Chapter 7 concludes the thesis and gives an outlook to future work.

CHAPTER 2

MATHEMATICAL BACKGROUND

In this chapter we present mathematical background that is used in the rest of the thesis. The chapter begins with an introduction of cryptography basis. Relation between private and public key cryptosystems are mentioned. An overview of group theory and number theory is presented. Later mathematical background of RSA and Elliptic Curve Cryptography are described. Definitions, descriptions of arithmetic operations are stated.

2.1 Cryptography Basis

Cryptography deals with design of mathematical techniques that enable some form of secure communication between two parties A and B (Alice and Bob) in the presence of malicious opponent (Oscar) who wants to eavesdrop the communication channel. The communication channel may be detailed for different types of scenarios in which one or more of the major security concerns are addressed.

- *Confidentiality:* Keeping the data secret from all but those authorized to have it.
- *Data Integrity:* Preventing other parties from data manipulation. If the data sent by Alice is modified by Oscar, Bob should be able to detect this modification
- *Authentication:* Making sure that either Alice or Bob identify each other in the case of communicating over the insecure channel.

- *Non-repudiation:* Preventing from either Alice or Bob from denying previous commitments or actions.

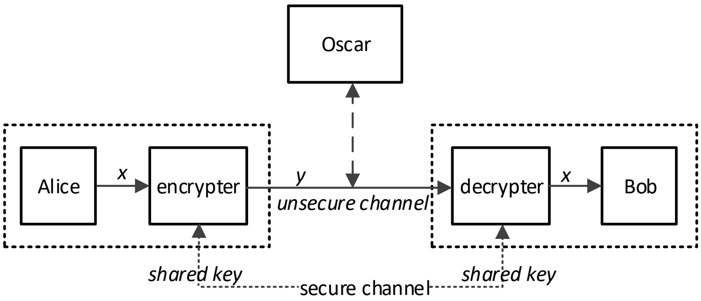


Figure 2.1: Communication model for secret-key cryptosystems

While the other security concerns are also important in today's wide range of cryptographic applications, confidentiality can be defined as the most known and historic purpose of cryptography. To enable confidentiality Alice encrypts *plaintext* message x by using a *secret key* k. By this conversion *ciphertext* y is obtained. Oscar who can observe ciphertext, should not obtain any meaningful information related to x. Bob receives the ciphertext and decrypts the message with the help of the same secret key k. Since the both parties needs the same key information this family of algorithms are called *symmetric key cryptography*.

As an adversary model it is assumed that Oscar has all the information about the encryption and decryption functions. It is also assumed that Oscar can read and record all data transmitted over the channel, change or modify this data. The only exception to Oscar's knowledge is the key information k. Due to this fact, this family of communication is also called *secret key cryptography*. The security of these types of communication is based on the secrecy of the key. In fact this concept is stated as a principle by Dutch linguistic and cryptographer Auguste Kerckhoff as “a cryptographic system should be designed to be secure, even if all its details, except for the key, are publicly known” [2].

Very efficient algorithms exist for symmetric key cryptography like DES, AES etc. These algorithms satisfy the above principle, they remain resistive against all the computation power of our age. As indicated in Figure 2.1 a secure channel is needed for key sharing or *key distribution*.

A secure channel can be established offline with a key transfer device, also called *fillgun*, which has been widely used in military communication [3]. A set of secret keys are loaded into these devices at a facility called key management center. Then a courier physically transfers the device to locations of Alice and Bob. Key transfer devices are tamper-proof devices, if one tries to open or tamper the device, an electronic or mechanical prevention system destroys the keys.

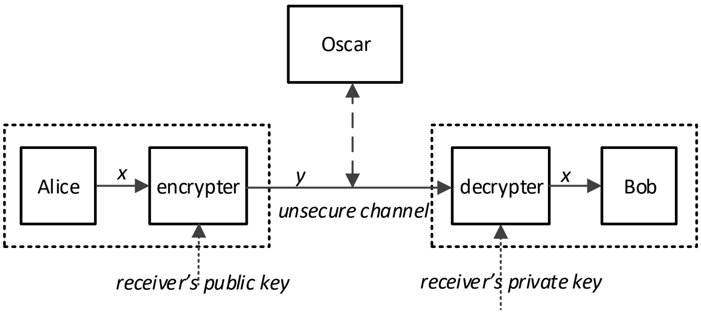


Figure 2.2: Communication model for public-key cryptosystems

Fortunately there is another method for key exchange without the establishment of a secure channel. It is called *public key cryptography* or *asymmetric key cryptography*, where there are two different keys for encryption and decryption, see Figure 2.2. Given only the encryption key (public key), it is hard to obtain the decryption key (private key). By the use of these algorithms, key exchange problem of symmetric key cryptosystems are solved. Public key crypto systems computationally cost more than secret key cryptosystems. Any modern cryptosystem uses both of the public and private key algorithms in their communication protocols.

There are three main types of public-key cryptosystems whose underlying computational problems are as follows.

- **Integer Factorization based algorithms:** Public-key schemes based on the fact that factoring large integers is difficult. The most known member of this family is the RSA algorithm.
- **Discrete Logarithm based algorithms:** In real numbers $\log_a b$ is a number x such that $a^x = b$ i.e. $\log_2 8 = 3$ In any group G powers of a, $b = a^k$, can be defined for all group elements. Discrete logarithm problem (DLP) is finding k only when b and a are given. Diffie-Hellman key exchange and ElGamal

encryption algorithms are DLP based public-key schemes.

- **Elliptic Curve Discrete Logarithm based algorithms:** Elliptic curve cryptosystems form an additive group with point addition operation. For $Q = k \cdot P$ it is difficult to find k when only P and Q are given. This problem is known as elliptic curve discrete logarithm problem (ECDLP). Algorithms in this family are Elliptic Curve Diffie-Hellman key exchange (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA).

The most prominent public-key cryptosystem is RSA algorithm invented by three professors from MIT, Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. It uses a pair of public and private keys to encrypt and decrypt the data. The algorithm consist of the following steps:

- Public and private keys are generated.
- Public keys are published or sent to participants.
- Participants encrypt plaintext messages using the public key and send this ciphertext over unsecure channel.
- After receiving the message, only the receiver can decrypt the ciphertext and obtain the plaintext.

Modular exponentiation is the main operation in RSA cryptosystems as will be detailed in the following subsections. The other popular public-key cryptosystem is *Elliptic Curve Cryptography*, where scalar point multiplication primitive exists analogous to modular exponentiation.

In the following subsections we present basic information about public key cryptosystems and cryptographic primitives used in these algorithms. But before that we will give definitions and theorems from Algebra and Number Theory as they will be the underlying nomenclature in the thesis.

2.2 Algebraic Structures and Number Theory Basics

Addition and multiplication defined over integers are two well-known operations. Given two integers $a, b \in \mathbb{Z}$, $c = a+b$ and $d = a \cdot b$ are all $\in \mathbb{Z}$, which is said as *closure property* of the operation. Given an arbitrary set G a *binary operation* is a mapping from $G \times G$ into G where $G \times G$ consits all the ordered pairs (g, h) such that $g, h \in G$. An *algebraic structure* is called a set together with one more operations defined on the set. Groups are algebraic structures with only one associative operation.

Definition 2.2.1. A *group* $(G, *)$ is a set G together with a binary operation $*$ which has the following three properties:

- Identity element: For all $g \in G$ there exist an $e \in G$, such that $g * e = e * g = g$
- Inverse element: For each $g \in G$ there exist an inverse $g^{-1} \in G$, such that $g * g^{-1} = g^{-1} * g = e$
- Associativity: For all $g, h, k \in G$, $(g * h) * k = g * (h * k)$

With this definition we may give examples and counter examples of groups. $(\mathbb{Z}, +)$ is an additive group of Integers, while $(\mathbb{Z}, \times)$ does not form a group since multiplicative inverse of integers other than 0 and 1 are not integers. However, $(\mathbb{Q}, \times)$ forms a group under rational numbers. Thinking of addition and multiplication associativity seems to be a natural part of operation. If the binary operation is defined as $a * b = a^b$ then the operation would not be associative, i.e. $(2^3)^4 \neq 2^{(3^4)}$, hence it does not have a group structure.

The associative property enables us to write $a_1 * a_2 * \cdots a_n$ with $a_i \in G$ in any order, the expression represents the same element. If we use a multiplicative notation for $*$, $a_1 a_2 \cdots a_n = a^n$. If an additive notation is used in place of $*$ we write $a_1 + a_2 + \cdots + a_n = n \cdot a$. The inverse elements are also denoted $1/a$ and $-a$ in these cases. When every other element of the group can be written as power of a fixed element we reach to the below definition

Definition 2.2.2. A group $(G, *)$ is called as *cyclic* if there is an element $g \in G$ such that $G = \{g^n : n \in \mathbb{Z}\}$. Such an element g is called a *generator* of the group. We write $G = \langle g \rangle$ and read as "group G is generated by g."

Although the term *cyclic* inherently has a periodicity meaning, in this context there can be cyclic groups which has infinite elements. For example the group $(\mathbb{Z}, +)$ can be written as $(\mathbb{Z}, +) = \langle 1 \rangle = \langle -1 \rangle$. So let's write the definition related to group size.

Definition 2.2.3. A group $(G, *)$ is called as *finite* if G is a set with exactly n distict elements. In that case the number of elements in the group is called *order* of the group and written as $|G| = n$. Otherwise, the group is called *infinite*.

We skip to *congruence* definition before giving more group examples:

Definition 2.2.4. For $a, b \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$, positive integer, we say a is $congruent$ to b modulo n (or just mod n) if $a - b$ is divisible by n. In this case we write $a \equiv b \mod n$. The same condition is also expressed as $a = b + kn$ for some integer k.

Congruence modulo n is an *equivalance relation* on $\mathbb{Z}$ since it is

reflexive	: $a \equiv a \mod n$,
symmetric	: If $a \equiv b \mod n$, then $b \equiv a \mod n$
transitive	: $a \equiv b \mod n$ and $b \equiv c \mod n$ then $a \equiv c \mod n$

The equivalence classes of congruence mod n are denoted as $[0], [1], \cdots [n-1]$. It can be shown that the equivalence classes form a group with the operation $+$ such that $[a] + [b] = [a+b]$.

Definition 2.2.5. The set $\{[0], [1], \cdots [n-1]\}$ of equivalence classes modulo n with the operation $[a] + [b] = [a+b]$ is called *additive group of integers modulo n* and denoted by $(\mathbb{Z}_n, +)$ or $(\mathbb{Z}/n\mathbb{Z}, +)$.

In Example 2.2.1, this group is presented in the *Cayley table* where rows and columns form the ordered pairs (g, h) of $G \times G$, the elements in the intersection show the value of $g * h$, modular addition result in the example. The identity element 0, the inverse elements can be traced from the table. Further inspection of the table reveals that the subsets $\{[0], [2], [4], [6], [8]\}$ and $\{[0], [5]\}$ form their own groups under the same operation. This opens the door to *subgroups* which we will not cover.

Example 2.2.1. Cayley table for the group $(\mathbb{Z}_{10}, +)$. The elements of the group $[i]$ are denoted as i for the sake of simplicity.

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

The *commutative* property of group can be extracted from the symmetry of the table. The property is not a group requirement but due to its importance it will be given as a definition.

Definition 2.2.6. Given a group $(G, *)$, $g, h \in G$, if for all $g * h = h * g$ the group is called *commutative* or *Abelian* after the Norwegian mathematician Niels Henrik Abel who has died at the age of 26 from tuberculosis [5].

While addition is nice and smooth, let's try to observe modular multiplication operation for the same set in Example 2.2.2. We observe that 0 vanishes the first row and column. So we may remove 0 from the set. Ignoring the first row and column there are still other problems, for example in $5 \times 2 = 0$ the operation violates closure property. These elements are called *zero-divisors* and they prevent from defining the group. Carefully inspecting these elements and removing them also from the input set we obtain the table in Example 2.2.3.

Example 2.2.2. Cayley table for the group $(\mathbb{Z}_{10}, \times)$. The elements of the group $[i]$ are denoted as i for the sake of simplicity.

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

Example 2.2.3. Cayley table for the group $(\mathbb{Z}_{10}^*, \times)$.

×	1	3	7	9
1	1	3	7	9
3	3	9	1	7
7	7	1	9	3
9	9	7	3	1

Let's give the definition of this group:

Definition 2.2.7. $\mathbb{Z}_n^*$ is the subset of integers in $\mathbb{Z}_n$ between 1 and n that are relatively prime to n. The operation $\cdot$ is defined in this set as $[a] \cdot [b] = [a \cdot b]$. is called *multiplicative group of integers modulo n* and denoted by $(\mathbb{Z}_n^*, \cdot)$ or $(\mathbb{Z}/n\mathbb{Z}^*, \cdot)$.

Number of integers relatively prime to n is an important concept.

Definition 2.2.8. Number of integers in $\mathbb{Z}_n$ relatively prime to n is called as *Euler's phi function* and denoted as $\Phi(n)$.

The example we studied was a composite even number, $n = 10 = 2.5$, so we needed to eliminate more than half of the set. If n is chosen as a product of two very big primes p and q, then we would need to eliminate only a few elements from the set. If n is chosen as a prime number p all the integers up to $p - 1$ would be relatively prime with p, and we would not need to eliminate any number for multiplicative group construction. Actually this is the case when we obtain prime fields. An important theorem related to primality, factorization, and multiplicative groups is Fermat's Little Theorem:

Theorem 2.2.1. Let a be an integer and p be a prime. We have

$$a^p \equiv a \mod p$$

if a is relatively prime to p then $a^p - 1$ is an integer multiple of p,

$$a^p \equiv 1 \mod p$$

A general form of the theorem applicable to and modulus n not necessarily prime is known as Euler's theorem

Theorem 2.2.2. Let a and n be integers relatively prime, $\gcd(a, n) = 1$ then

$$a^{\Phi(n)} \equiv 1 \mod p$$

, where $\Phi(n)$ is the number of relatively prime numbers smaller than n.

In Example 2.2.3 we found that $\Phi(10) = 4$ by careful inspection.

So far we dealt with structures with a single operation. Sets binded with two operations constitute *Rings* and *Fields*. We directly go with field definition:

Definition 2.2.9. A field $(F, +, \cdot)$ is a set F on which two binary operations are defined such that $(F, +)$ is an Abelian group, with identity element 0. $(F^*, \cdot) \equiv (F\backslash\{0\}, \cdot)$ is an Abelian group under multiplication. The identity element in this group is denoted by 1. Distributive of $\cdot$ over $+$ is defined as $g, h, k \in F$ we have $(g + h) \cdot k = (g \cdot k) + (h \cdot k)$.

The last theorem is about the order or cardinality of fields.

Theorem 2.2.3. A field with order q only exists if $q = p^n$ where p is a prime number and n is a positive integer.

In this formula $n = 1$ forms a special class of field with the below definition.

Definition 2.2.10. The field of integers modulo p denoted as $GF(p)$ is defined as *prime field* or *Galois field* with a prime number of elements. All non-zero elements of the field has multiplicative inverse. The field has the modulo p arithmetic $(Z_p, +)$ and $(Z_p^*, \cdot)$

Galois field is named in honor of French mathematician Évariste Galois, who died in the age of 21 after a duel. He has laid the foundations of Galois theory and group theory, two major branches of abstract algebra [11].

As a last example we give the Cayley table of GF(5) in Table 2.1

Table 2.1: Cayley table of GF(5)

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\times$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

2.3 Square-and-Multiply Modular Exponentiation

Modular exponentiation operation is the main operation in RSA algorithm. It is achieved by the *square and multiply algorithm* as given below:

Algorithm 1 Left-to-right square and multiply algorithm

```
Require: $A, e \in Z_M$, $e = (e_{n-1}e_{n-2}e_{n-3} \dots e_1e_0)_2$
Ensure: $C = A^e \bmod M$
$S \leftarrow 1$
for $i = n - 1$ to 0 do
    $C \leftarrow C \dots C \bmod M$
    if $e_i = 1$ then
        $C \leftarrow C \dots A$
    end if
end for
return $C$
```

2.4 Elliptic Curves

Definition 2.4.1. An elliptic curve $E(\mathbb{F}_p)$ over the finite field $\mathbb{F}_p$ where p is a prime number can be defined as the set of solutions to the equation

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

This form of equation is called simplified Weierstrass Equation after the German mathematician who lived in 19^{th} century. A tuple (x_1, y_1) satisfying the equation is called a point P_1 on the curve. The set of all points satisfying the equation together with the identity element ∞ is denoted as $E(\mathbb{F}_p)$. Having two points P_1 and P_2 on the curve, it is possible to reach to a third point $P_3 = P_1 + P_2$ when $P_1 \neq P_2$ through *point addition* formula. When $P_1 = P_2$, $P_1 + P_1 = 2P_1$ is called *point doubling*, and it also has a formula. Having the points and addition operation, we have the *closed under addition* property. The identity element has the property $P_1 = P_1 + \infty = \infty + P_1$. Every point $P = (x, y)$ has a negative $-P = (x, -y)$.

$k \cdot P = P + P + P + \ldots + P$ is defined as the *scalar multiplication* of a point. Scalar multiplication is done by using point addition and point doubling repeatedly. This method is called *double-and-add algorithm* which is analogous to square-and-multiply method in Algorithm 1.

Definition 2.4.2. Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$. Assume P and Q are two points on E such that $Q = k \cdot P$. *The elliptic curve discrete logarithm problem (ECDLP)* is to find k given P and Q.

Further details of mathematical background will be given inside the chapters when implementations are described.

CHAPTER 3

TECHNICAL BACKGROUND

With advancements in semiconductor technology, ever more complex systems became possible. At the same time demands of society on the computing capacity is still increasing. While microprocessors are considered as central processing units as early as 1960s, reconfigurable logic became relevant as a data processing element in the late 1980s, namely by the companies Xilinx and Altera. The main difference between a processor and programmable logic device (PLD) is that the processor's flexibility is achieved through the use of software, while PLDs map processing functions at the hardware level. PLDs are made up of large matrix of logic blocks which can be flexilibly interconnected. While processors can be considered as state machines that process commands sequentially, PLDs exploit the parallelism of the structure. The best example for this case can be given as the implementation of a block-cipher algorithm, for example AES. In a microprocessor the software of the algorithm runs the operations of each block with a different instruction and with a number of cycles depending on the arithmetic logic unit (ALU) size, 8,16,32 or at most 64bits. In a PLD the algorithm can run 128-bit block within a single clock cycle due to the available parallelism. PLDs evolved in the form of PALs, CPLDs and FPGAs. Since FPGAs are the dominant member of the programmable logic devices, also called reconfigurable logic devices, we will be using term FPGA interchangeable with PLD.

Application specific integrated circuits (ASICs) also has the above parallelism. They even out-perform FPGAs in terms of better speed, lower power consumption and lower unit price. However, the initial costs for ASIC production are very high, so it makes economic sense for large quantities. The design-to-production time of ASIC chips is also high, in the order of months, compared with an FPGA. For FPGA the time to generate the design file and programming it into the FPGA takes in the order of minutes to hours depending on the size of the design. One thing FPGAs and ASICs have in common is that a fixed time limit can be determined for the processing of a task. By this way, the real-time requirements for the calculation time can be achieved. On the other hand, processor systems require software architectures such as operating systems which do not have a chronologically deterministic nature.

FPGAs stands between the two extremes, highly flexible processors that use software on the one hand, and the highly efficient but highly specialized ASIC implementations on the other hand. Modern FPGAs have been building a bridge between the processor side by installing processors inside FPGA unit. Xilinx started this era with PowerPC processor cores into Virtex-II, Virtex-4 and Virtex-5 Family of FPGAs. These were hard-processor cores, so that they are physically present in the FPGA silicon die even if you don't use them. Later Microsemi introduced Arm Cortex-M3 processors in their SmartFusion-2 family FPGAs and Intel introduced Arm Cortex-A9 processors in Cyclone-V family FPGAs. There are also various soft-processor options from these FPGA vendors. Soft-processors do not exist physically in the FPGA die, if you want to use them they are compiled and placed into *FPGA fabric*. Last but not least, there are Risc-V processor options which have been widely spreading in the industry in the last years.

By the use of processor systems FPGA resources are used as accelerator for time-critic and performance demanding parts of the problem. Processor subsystems are used for high-level management of these accelerators.

In this chapter, firstly, FPGA and ASIC design flow will be presented. After that digital logic design basics used in both ASIC and FPGA implementations will be summarized. When applicable the differences between ASIC and FPGA are also going to be mentioned. FPGA specific constructs such as math blocks, memory blocks are going to be presented. Processor-based solutions and terminology will also be introduced in this chapter.

3.1 Development Flow of FPGA and ASIC

In digital logic design four levels of structural abstractions are considered after [21]. These are, transistor level, gate level, register level and processor level. While these are the structural levels of digital systems, a corresponding behavioral resolution is also given. Differential equations, boolean equations, register transfer operations and algorithms are the levels of abstractions in the behavioral domain of digital systems.

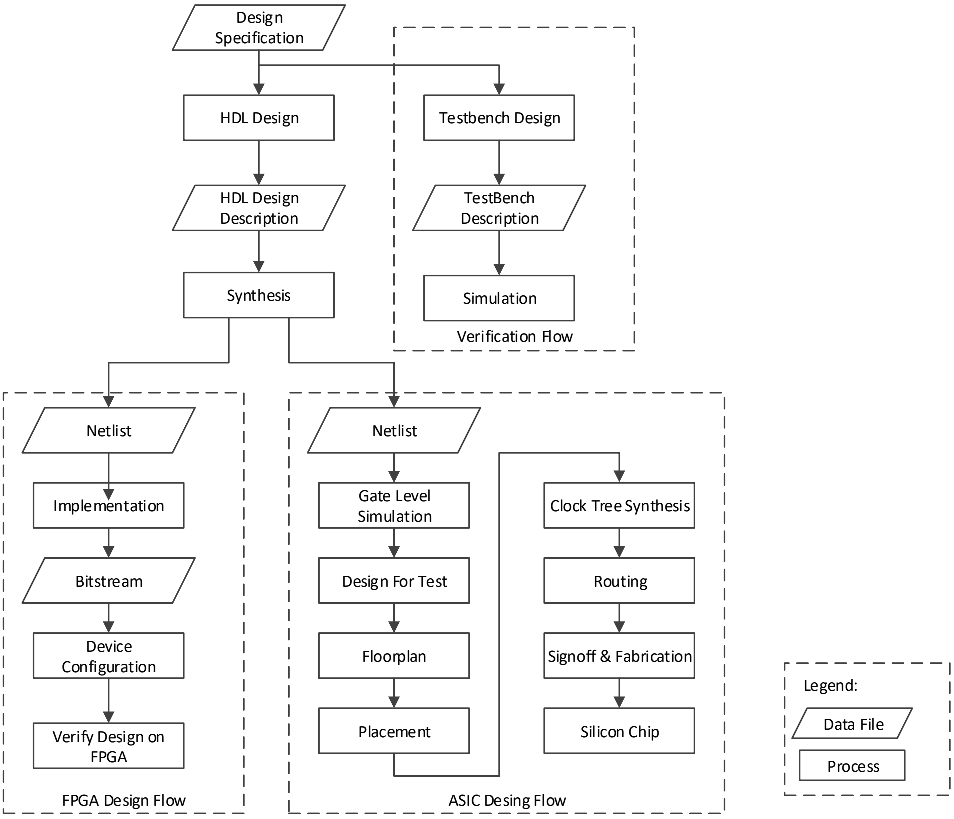


Figure 3.1: FPGA and ASIC design flow

As transistors in a chip reaches hundreds of millions, it is impossible to process the data directly on the lowest level. FPGA and ASIC design begins in an abstraction of register level. Translation of the information into lower layers is done by electronic design automation (EDA) software. A combined flow of FPGA and ASIC design is given in Figure 3.1. Design entry can be considered common for ASIC and FPGA flow. The flows differ after synthesis.

FPGAs are physical parts that have a finite number of carefully gathered resources. There is no physical implementation in the FPGA design flow. FPGA implementation phase is placing the synthesis output into FPGA resources, and then connecting (routing) the resources to enable desired functionality. This place and route information is called as configuration information or bitstream. When this configuration is loaded into FPGA the flow is finished after testing the design on real hardware.

On the other hand ASIC flow includes tedious steps necessary to generate a successful chip fabrication. Design for test is the step which is a very important aspect in ASIC design. The purpose of this step is to detect defects in the fabrication process. Built-in self-test and scanning control circuits are added into the design for testing after fabrication. In ASIC design there is no infrastructure like FPGA, so everything is built from ground. For a detailed coverage of ASIC design flow we refer to CMOS VLSI Design book by Neil Weste and David Harris [50].

Design Entry: In the old times of digital design, schematic entry of the design had been used. As the design sizes rapidly increased, *hardware description languages* (HDL) are introduced for design entry. There exist two main languages Verilog and VHDL for hardware description. In both the language has two main class of instructions, synthesizable and non-synthesizable. Synthesizable part of the language describes the part of the design which will go into silicon (ASIC), or FPGA. Non-synthesizable instructions describe verification of the design.

Verification: Verification is the activity of checking whether a design meets the requirements. Verification has two aspects in the context of digital design *functional* and *performance*. Functional verification checks whether the designed system generates the intended outputs. Performance of a digital design is that the design produces the output in accordance with a time constraint. Timing performance is the process of checking whether the design produces the output within given time limits. Verification can be done in different phases of the design flow. Functional verification is the first stage of verification. It does not consider the timing-delays in the circuit. It behaves as if the logic elements respond with zero-delay and there is no other delay in the design.

Simulation: The most commonly used method of verification is *simulation* which is emulating real design behavior in a software environment. A model of the real design is constructed and test patterns representing the inputs of the design are applied to the model. The response of the design is evaluated according to the expected references which are computed again at the simulation environment. Simulation can be applied in different levels of abstraction and different level of design cycle. Functional simulation only checks for the correctness of the output. Post-synthesis and post-layout simulations also considers the placement and routing effects of design cycles. The environment which provides test-inputs and checks the outputs according to a reference is called as *testbench*.

3.2 Basics of Digital Logic Design

Logic circuits deal with digital signals and realized as electronic circuits where the signal values are bounded to a few discrete values. In binary logic circuits these values are 0 and 1. The amplitude of the electrical signal may be in a range between $[0 - 3.3V]$ for example. All the elements in the circuit quantize this signal to 1 if it's above a threshold, to 0 if it is below a threshold. This class of circuits are also called *digital circuit*. In constract there are *analog circuits* where the electrical signals may take values on a continuous range and every other element process that continuous signal.

3.2.1 Combinational logic circuits

A logic circuit may have one or more binary logic inputs and one or more binary logic outputs. The simplest logic circuits of interest has one input and one output, called as inverter and its truth table together with its symbol is shown in Table 3.1. Two-input binary logic circuits are the main building blocks of combinational circuits. The output of the function is a combination of inputs but nothing else. They don't have internal state or memory. Important two-input binary logic elements also called *logic gates* are seen in Table 3.2. In the first row of the table circuit symbols are also presented. Inspecting the table we see that the latter three functions are binary opposite of the first three functions. This case is indicated with a bubble in the symbol, with an overline in the output expression and binary opposite values in the truth table.

We will give one more example of combinational circuit as in Table 3.3 Inspecting

Table 3.1: Inverter truth table and circuit symbol

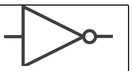
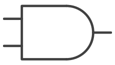
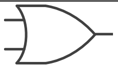
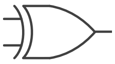
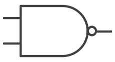
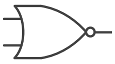
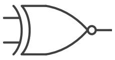
x_1	 $\overline{x_1}$
0	1
1	0
	NOT

Table 3.2: Important two-input binary circuit *(logic gates)* truth tables and circuit symbols

x_1	x_2	 $x_1 \cdot x_2$	 $x_1 + x_2$	 $x_1 \oplus x_2$	 $\overline{x_1 \cdot x_2}$	 $\overline{x_1 + x_2}$	 $\overline{x_1 \oplus x_2}$
0	0	0	0	0	1	1	1
0	1	0	1	1	1	0	0
1	0	0	1	1	1	0	0
1	1	1	1	0	0	0	1
		AND	OR	XOR	NAND	NOR	XNOR

the table it is seen that the select signal s selects x_1 when $s = 0$ and selects x_2 when $s = 1$. It *multiplexes* one of the inputs to the output depending on the value of the select signal s. The logic circuit and symbol of multiplexer is given in Figure 3.2. In the left of the figure, equivalent circuit in terms of two-input logic building blocks are seen. The circuit reveals the functionality when signal s is traced. When $s = 1$ the output of the upper AND gate will be equal to x_2, the lower AND gate will produce zero since inverse of s, $\overline{s}$ will be connected to below AND gate. So x_2 will appear at the output of OR gate. For $s = 0$ the reverse path will be active and x_1 will appear at the output. In the right of the figure the corresponding logic symbol of the multiplexer(mux) is given.

The set of AND, OR and NOT gates can be used in cascade to represent any other combinational circuit. In this respect NAND gate and NOR gates are by themselves enough to describe any other logic circuit, so they are called *universal gates*. In ASIC technology *standart-cell library* is a collection of well-defined logic gates that can be used to implement a digital design.

Table 3.3: Multiplexer truth table

s	x_1	x_2	$f(s, x_1, x_2)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

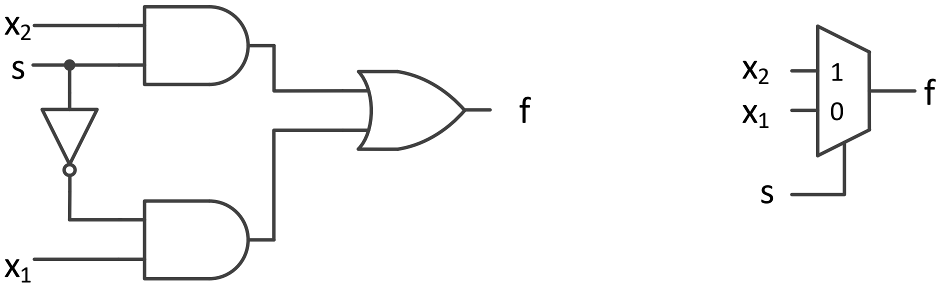


Figure 3.2: Multiplexer circuit (left) and logic symbol (right)

An example synthesis output of combinational gates is given in Figure 3.3. The example generates all the outputs in Table 3.2 and output of mux in Table 3.3 in response to inputs x_1, x_2 and s. Observing *rtl schematic* the basic logic gates and multiplexer symbols are seen in the Figure 3.3. The synthesis software generates the optimum combinational circuit by its internal algorithms. In the schematics it is seen that there is not an explicit NAND or NOR gate, although it was defined in the HDL Definition Instead of these inverters are concatenated after AND and OR gates. Logic synthesis software are capable to optimize far more complex circuit descriptions.

The circuit in Figure 3.3 is excited with a series of signals by a testbench. The behavior of combinational gates was defined by truth tables in Table 3.2. The testbench in this example applies the values in the truth tables for different time durations. The changes of signal values in time can be presented in graphical form known as *timing diagram*. The timing diagram in Figure 3.4 applies all combinations of (x_1, x_2, s) for 6 times (periods). In the first two periods, changes are scheduled at every 100 ns. In later periods the changes were scheduled for 7 ns. Observing the timing-diagram it is seen that the output signals are generated instantaneously in response to input sig-

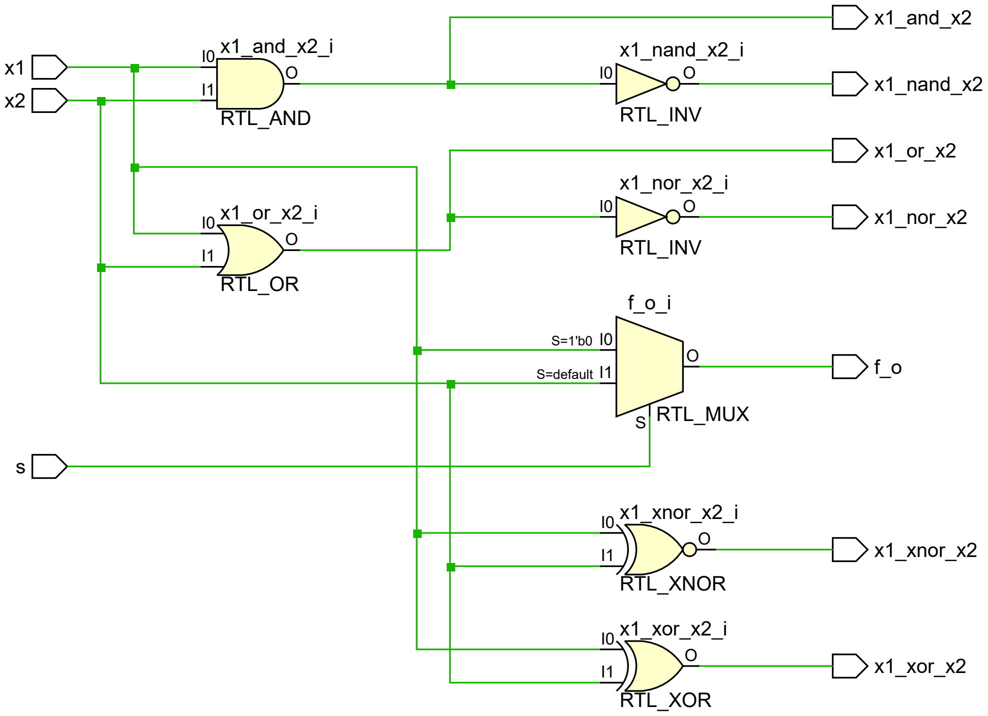


Figure 3.3: Combinational signals rtl schematics

nals. This type of response or simulation is called *functional behavior* or *functional simulation*.

3.2.2 Synchronous logic circuits

We have covered combinational circuits which have no memory. A *sequential circuit* can be defined as a digital circuit which has internal state or memory. As a special

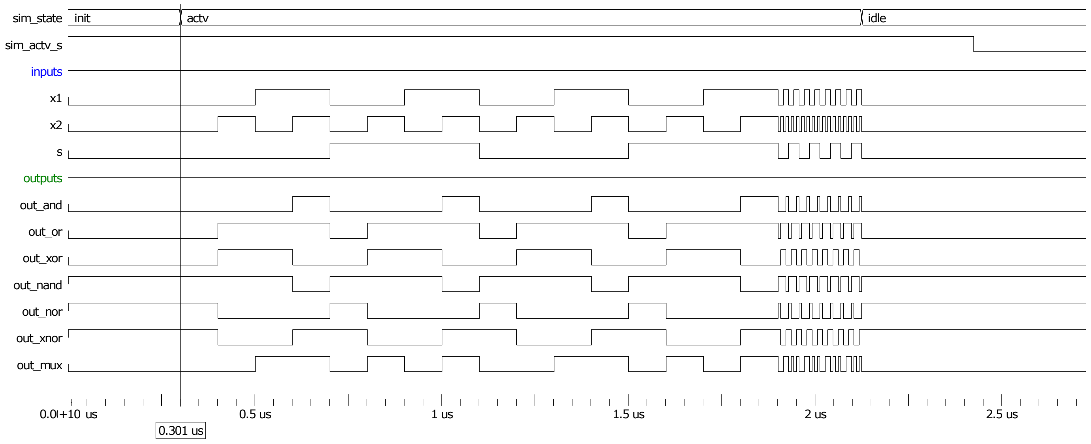
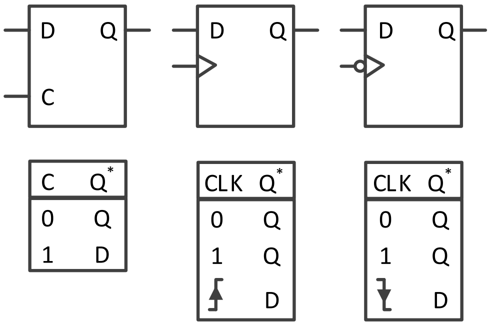


Figure 3.4: Combinational signals simulation output

case of sequential circuit, *synchronous sequential circuit* is circuit elements which have internal state or memory and all the updates of these elements are controlled by a synchronizing signal. This synchronizing signal is known as *clock signal* and mostly abbreviated with `clk`. Circuit symbols and truth tables of important synchronous gates are given in Figure 3.5. In the truth tables the next value of Q is denoted with Q^*. Inspecting the *D-latch* we can say that it is a level triggered sequential element. When $C = 1$ the output will follow the input. If C goes to zero, the last state of the Q will remain at the output. Although latches exist as a sequential element, they need to be avoided in digital designs. *D-type flip-flop (D-FF)* is an edge-triggered sequential element. It only changes state at the rising-edge (middle primitive) or falling edge of the clock signal. In all the other times it presents the internal value Q to the output. A D-type flip-flop can only store a single bit. A *register* is a collection of D-flip-flops which share the same clock signal. The design methodology based on synchronous gates is called as *synchronous digital design* or *sequential circuit design*. This methodology is by far the most important and widely used electronic design technique.



C	Q*
0	Q
1	D

CLK	Q*
0	Q
1	Q
↑	D

CLK	Q*
0	Q
1	Q
↓	D

Figure 3.5: Sequential gates: D-latch, D-flip-flops

The combinational examples given in Figure 3.3 is adapted to synchronous design technique in Figure 3.6. The same testbench is applied to this circuit, the timing diagram is given in Figure 3.7. In synchronous design technique all combinational circuit elements are placed between two D-flip flop. When these flip-flops are driven by the same clock signal (as marked with line in Figure 3.6) a *clock-domain* is obtained. In this case the synthesis and timing analysis of the circuit is possible through extensive *electronic design automation* (EDA) tools.

In Figure 3.7 the signals `x1,x2` and `s` are grouped as asynchronous inputs, i.e. they

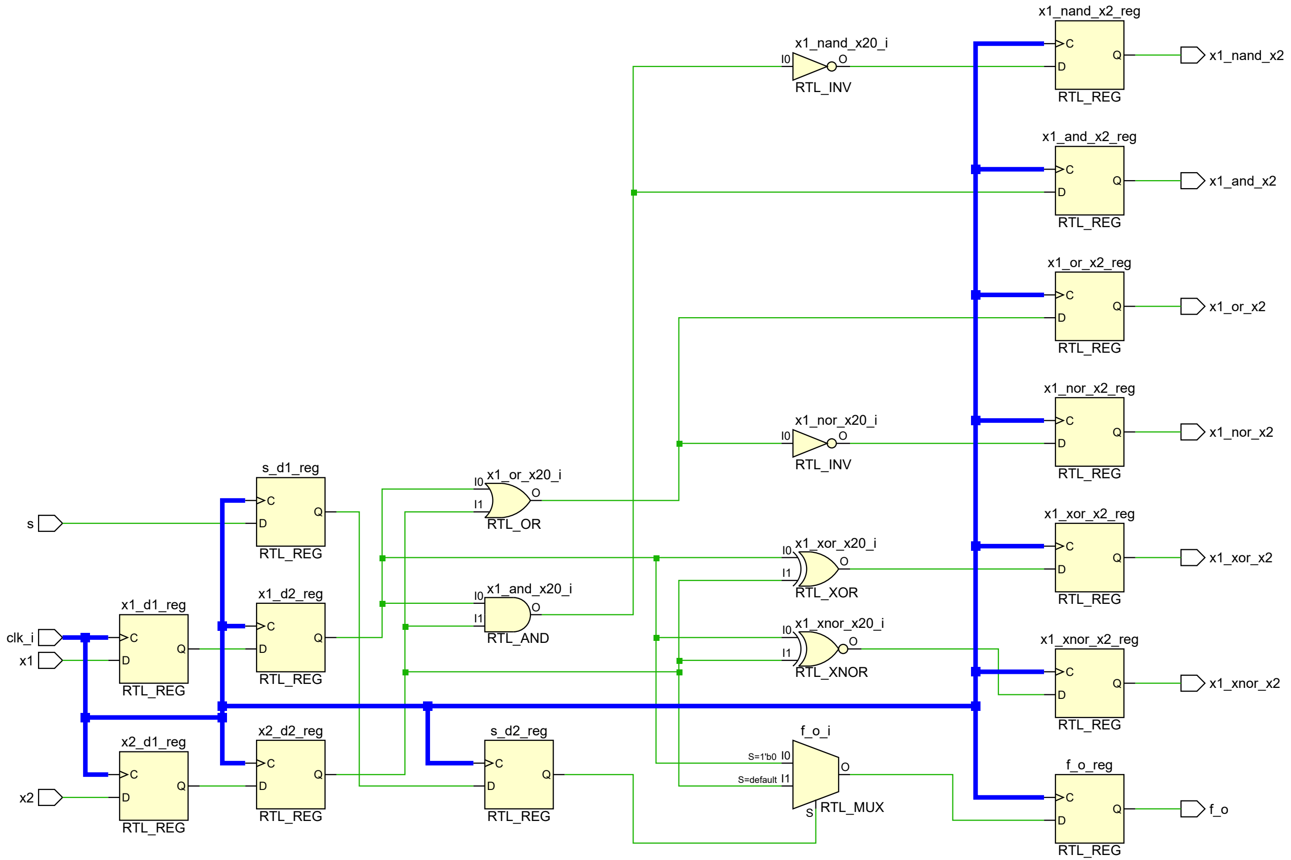


Figure 3.6: Synchronous signals rtl schematics

are not changing together with a clock. Such signals must not exist inside a synchronous design. So they are *synchronized with a double flip-flop* before being processed. Inspecting the signals with postfixed with `_d2` it is seen that they are delayed by two clocks and synchronous with the rising edge of the clock signal. When these signals are applied to the circuit the same functional outputs are obtained. Again carefully inspecting the output signals it can be observed that the outputs are *delayed by one clock cycle* with respect to the inputs. This is due to the output registers placed on the right side of the schematics.

Special case of synchronization can be observed during the fast-changing region of the timing diagram. In `x2_dd` signal some periods of the input signal is lost during synchronization. Also, the duty-cycle, (duration of high-low regions) of `x1_dd` signal is modified after synchronization. The missing signals are due-to the fact that clock period in this example is 10 nanoseconds(ns), whereas in the fast changing region the signals are changing with a period of 7 ns. There are such alignments in the timing that either a low or high part of the input signal is not *sampled* or *registered* with the rising-edge of the clock.

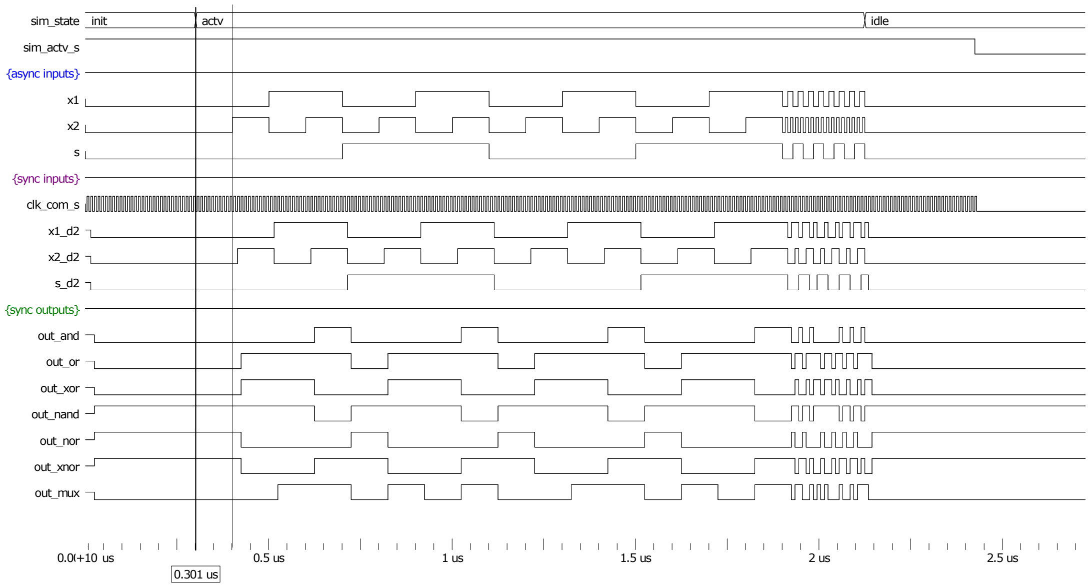


Figure 3.7: Synchronous signals simulation output

Further digital design and FPGA building blocks are going to be detailed in the later chapters during design descriptions.

CHAPTER 4

FLEXIBLE MONTGOMERY MULTIPLIER WITH BLOCK RAMS

Modular multiplication $A \cdot B \mod M$ can be considered as the most crucial primitive in RSA and elliptic curve cryptography. Modular multiplication methods can be classified into three main categories. The first one is the standard method in which a division by M is followed after obtaining the product $A \cdot B$. The second method, Interleaved Modular Multiplication (IMM), [14] propose that reduction is done while doing the multiplication. Intermediate results are obtained, added together to reach the final result. The algorithm targets to obtain directly the remainder of the division, the quotient of standard division algorithm is not generated as a result. The third and most prominent method of modular multiplication is Montgomery Modular Multiplication (MMM) which is discovered by Peter L. Montgomery [35]. The method became widely known and both software and hardware implementations are extensively studied [15, 25].

In this chapter, a flexible digit based Montgomery multiplier which can work with any modulus and any bit length is described. The multiplier is designed to work as a coprocessor with an independent clock frequency from the rest of the FPGA design. Operands of the multiplier are stored in block-rams and hard multiplier macros, DSP blocks are used for high-radix multipliers. A hard-wired control block is added to this multiplier to obtain a modular exponentiation circuit to be used in RSA cryptosystem. Any length of practical exponentiation can be done with the obtained arithmetic core by run-time configuration.

The rest of the chapter is organized as follows: In Section 4.1 we mention related works and necessary algorithms. In Section 4.2 proposed multiplier architecture is described. Exponentiation design 4.3, test and verification of arithmetic core 4.4, implementation results 4.5 followed next. The chapter is finalized with conclusion and future works. 4.6

4.1 Background and Related Work

Modular multiplication, $A \cdot B \ mod \ M$, requires multiplication and division operations. Division is an expensive operation in computer arithmetic. Montgomery Modular Multiplication without division is discovered by Peter Montgomery [35], which is described in the following algorithm.

Algorithm 2 Calculate ABR^{-1} mod M

```
Require: $A = (a_{n-1}, \dots, a_0)$, $B = (b_{n-1}, \dots, b_0)$, $M = (m_{n-1}, \dots, m_0)$ where
    $0 \leq A, B < 2.M, M < R = \beta^n, \beta = 2^{ws}, gcd(M, R) = 1, m' = -{m_0}^{-1}$ mod $\beta$
Ensure: $S = ABR^{-1}$ mod $M$
$S \leftarrow 0$
for $0 \leq i < n$ do
    $q_i \leftarrow (s_0 + b_0 * a_i)m'$ mod $\beta$
    $S \leftarrow (S + B.a_i + M.q_i)/\beta$
end for
return $C$
```

In the formulation w is *word size* of the number representation. If for example, $w = 4$ than a_i, b_i and m_i are i^{th} digits of the big numbers, *operands* A, B and M, respectively. In this case each digit is in base $\beta = 2^4 = 16$. The digits in base-16 is well-known to be as *hexadecimal* numbers. In Algorithm 2, there are two big multiplication operations $a_i \cdot B$ and $q_i \cdot M$. These multiplications require that another loop with j index access all digits b_j and m_j, i.e. $a_i \cdot B = \sum_j a_i.b_j.\beta^j$ The operands which consist of many digits are denoted by capital letters, their digits are denoted by small cases as a convention. Furthermore, the subscripts i and j denote outer and inner loops of the algorithm. If digits of operands are accessed one digit at a time, the algorithm will require number of steps in the order of n^2, where n is the number of digits of operands in base-β.

4.2 Flexible Montgomery Multiplier

The data path of Algorithm 2 is seen in Figure 4.1. By examining the graph, we observe that there are three multipliers in the loop. It is also seen that there are two two-input adders. In the data path figure the only item which has i subscripts are a_i and q_i. This represents the fact that for each digit of operand A, $\frac{1}{n}$ of the result is obtained. When all the digits of A is consumed the final result S is reached. The implementation of the current `mon_mul` module is done by using multiplier macros. By inspecting the Algorithm 2 that the only modulus related parameters are m' and n, number of digits for number representation. If these two parameters can be dynamically configurable for the modular arithmetic core, than the core would be capable of doing computation on any modular multiplication. This parametrization of the core will be detailed later in this chapter.

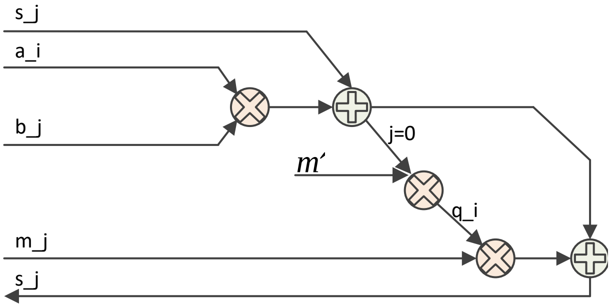


Figure 4.1: Montgomery modular multiplication algorithm data path

The Montgomery modular multiplier is shown with the block rams of FPGA as in Figure 4.2. There are four input and one output operands to the algorithm. When implemented with dual-port rams, there needs to be at least 3 dual-port RAMs for continuous data feeding and data writing. Although not shown in the figure, there are address ports together with data ports. These ports are not denoted in the figure for the sake of simplicity. 5 of 6 ports of the block rams are driven by `mon_mul` module. The 6^{th} port of the block memory is driven by external circuit which wants to make use of the desired functionality.

In practical FPGA applications there are designs with more than one clock frequencies, where different regions of the design work with different clocks. This is also the case when there exist soft or hard processors in the design. To be used in such a case

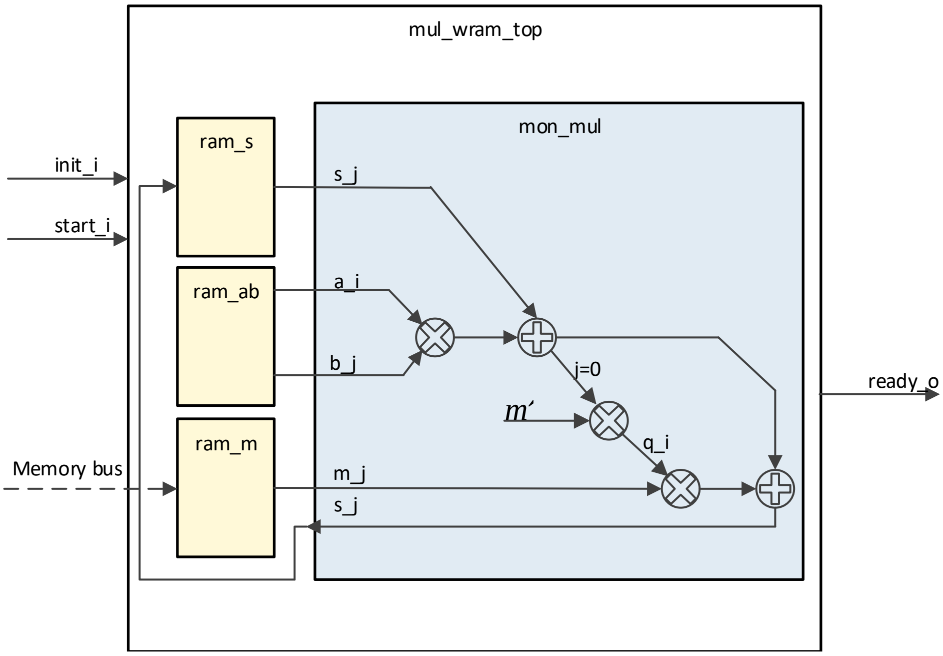


Figure 4.2: Montgomery modular multiplier with ram blocks

we designed the arithmetic circuit and controlling circuit in two different clocks. The dashed input line in Figure 4.2 represents the clock other than the arithmetic clock. Throughout the design interface circuit remains in a slower clock domain, for example 100 MHz, while the arithmetic circuit is maintained in a faster clock frequency, for example 200 MHz. The *clock domain crossing* of signals are accomplished by the use an independent dual-port ram and pulse synchronization circuits between clock domains.

The design described above comes with a cost. Since all parameters and operands are written from slow clock domain, they need to be moved to other block rams before starting the computation. The memory map after initialization from the slow clock is depicted in Figure 4.3. It is seen that the modulus M is located into `op_1` location. A and B are placed into locations `op_2` and `op_3`. As will be explained later, these locations are not fixed. They change according to n where n is the number of digits required to represent the numbers in $\mathbb{Z}_M$.

The memory map after the computation is shown in Figure 4.4. During the computation, 5 ports of dual-port rams need to be used concurrently. For this reason A and B are moved to `ram_ab`. On the other hand, `ram_s` is used to save the outer loop result S and to read them in the next loop. It is seen that only a small portion of `ram_s`

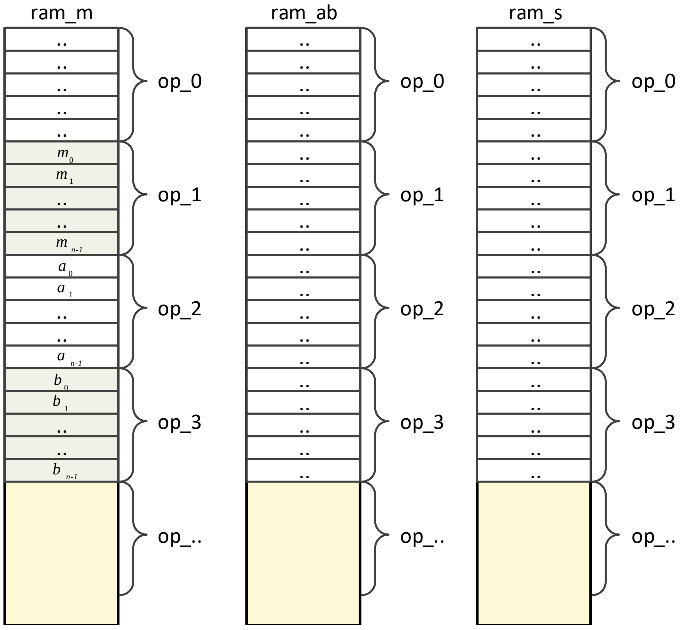


Figure 4.3: Memory occupation before execution

is used. The result is copied back to `ram_m` for the slow clock domain to read for the next time.

4.3 Modular Exponentiation

In the previous section, a block-memory based Montgomery modular multiplier core is described. Block memories are organized in different sizes in different FPGA brands and FPGA part numbers. For Artix7 family of FPGAs one block memory is 18Kbits. When organized as a 16-bits wide memory, each block memory can hold up to 1024 digits. Since three operands are necessary for single modular operation $A \cdot B\ mod\ M$, with a single block memory it is possible to make computations upto $1024/3 = 340$ digits, which is $340 * 16 = 5440$ bits. The memory blocks in the core are inferred from VHDL code, by this way the same core can be easily extended to support the next practical length of 8192-bits operations.

Montgomery modular multiplication is efficient if repeated computations are done in the Montgomery domain. One major application of repeated computation is the modular exponentiation operation

$$A^e \mod M,$$

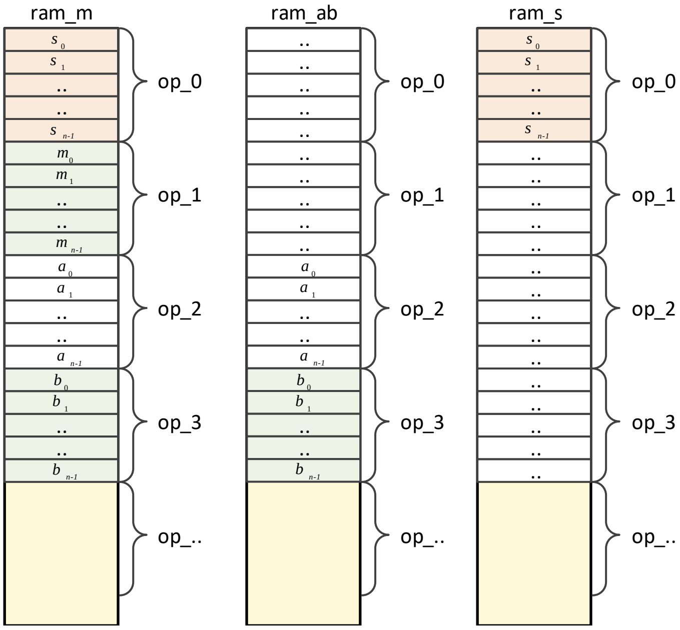


Figure 4.4: Memory occupation after execution

where A and e are integers in $\mathbb{Z}_M$. The modular exponentiation can be computed with the algorithm known as square and multiply method given in Algorithm 1.

For controlling the consecutive calls of Montgomery multiplier block, a hard-wired control module called `mon_manager` is designed. The block diagram of the exponentiation `mon_manager` , `mon_mul` and block RAMs are seen in Figure 4.5. As stated before dashed lines correspond to other clock domains of FPGA which can not be controlled. The solid lines show the `clock_math` domain. Other than the memory interface, there are trig signals which initiates multiplication and exponentiation operations. There are three type of interface to this core. These are initialization, memory copy and start arithmetic operation.

One novelty of our multiplier design is its run-time configurable property for different modulus numbers, M, and for different digit lengths. This design property is also shown in Figure 4.5 with green rectangles. Referring to 2 there is a pre-computation value m' which is related to m. The Algorithm needs multiples of digit lengths, which are saved in registers indicated with green rectangles. When a configuration data is written to interface RAM, `ram_m`, init command is triggered to the core. After this trig, the new modulus, and operand size namely number of digits `n_digs` is loaded into different registers. After the configuration of the core, operands are loaded into

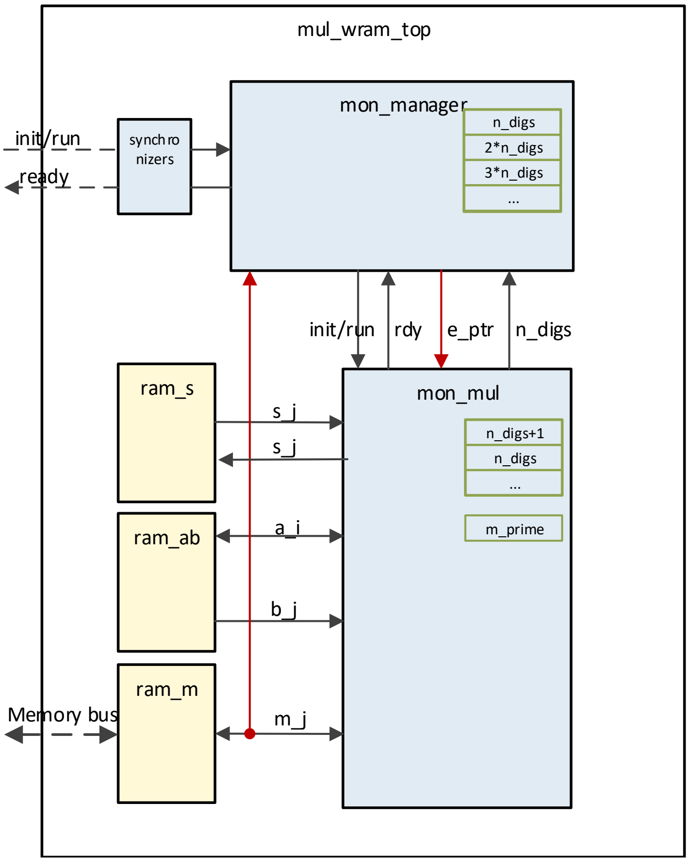


Figure 4.5: Exponentiation control block

interface ram with memory copy operation. As the last step start arithmetic operation is triggered after which the core either does a single modular multiplication or a series of squaring and multiplications which correspond to exponentiation operation.

The internal state machines inside `mon_manager` module is shown in Figure 4.6. The first machine on top-left corresponds to parameter initialization which runs after init command. The machine on top-right corresponds to main states of `mon_manager` for each of three interface commands. The below machine starts with and idle state, then goes to trailing zero state where it searches for the first non-zero bit of the exponent. After the first 1, the state machine does squaring at each state. it also does multiplication if a 1 occurs in exponent. There is one other novelty of this module which is indicated with red lines in Figure 4.5. The exponent control machine needs to get exponent parameter from RAM. Since our design is RAM-based, the whole operand remains in RAM. Exponent control circuit updates an `e_ptr` value which points to a specific 16-bit portion of exponent. This portion of the exponent is read into `mon_manager` module from the RAM port. By this way the exponent control

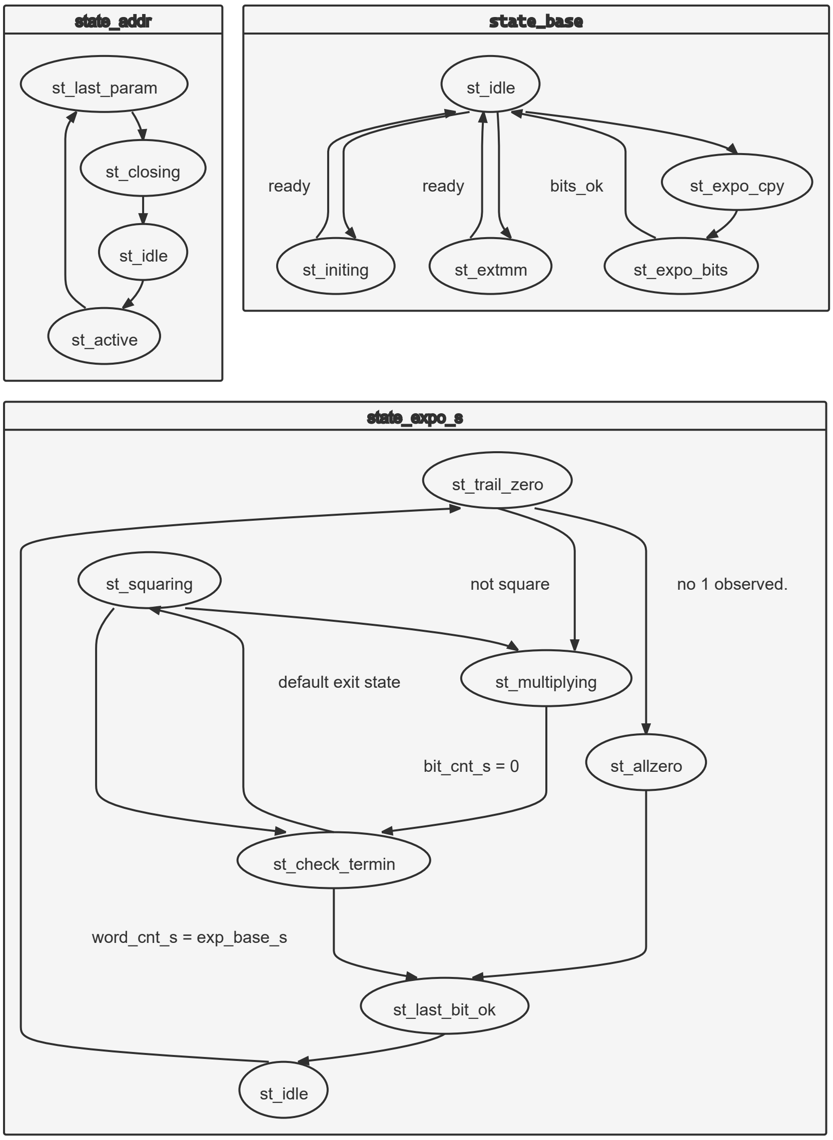


Figure 4.6: Exponentiation control state machine

circuit can work on any length of exponent without affecting logic resource usage. From implementation point of view, this functionality can be realized easily with a shift-register. However using a shift-register will violate our length-independent design criteria. By placing this parameter into block memory, any-length of operand can be represented with a change of pointer.

4.4 Testing of the Exponentiation Core

Testing and verification of an FPGA design may be very hard in case of simulation mismatches, timing violations or long implementation times. Design implementation of a big FPGA may take a long time so that one can only try a few designs in a day. Due to these reasons a test environment which talks with the exponentiation core is designed as shown in Figure 4.7. The test environment targets to verify functionality of the core in a more controlled small environment.

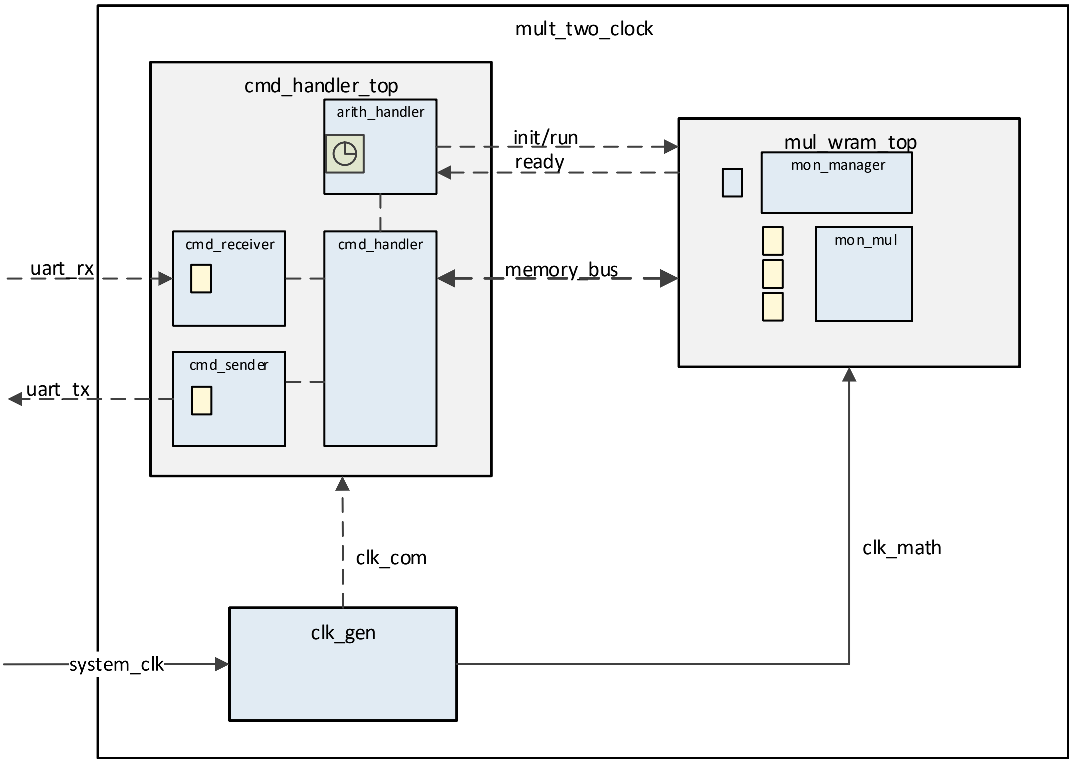
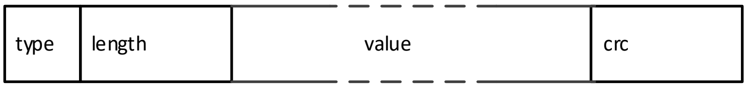


Figure 4.7: Test interface of the exponentiation core

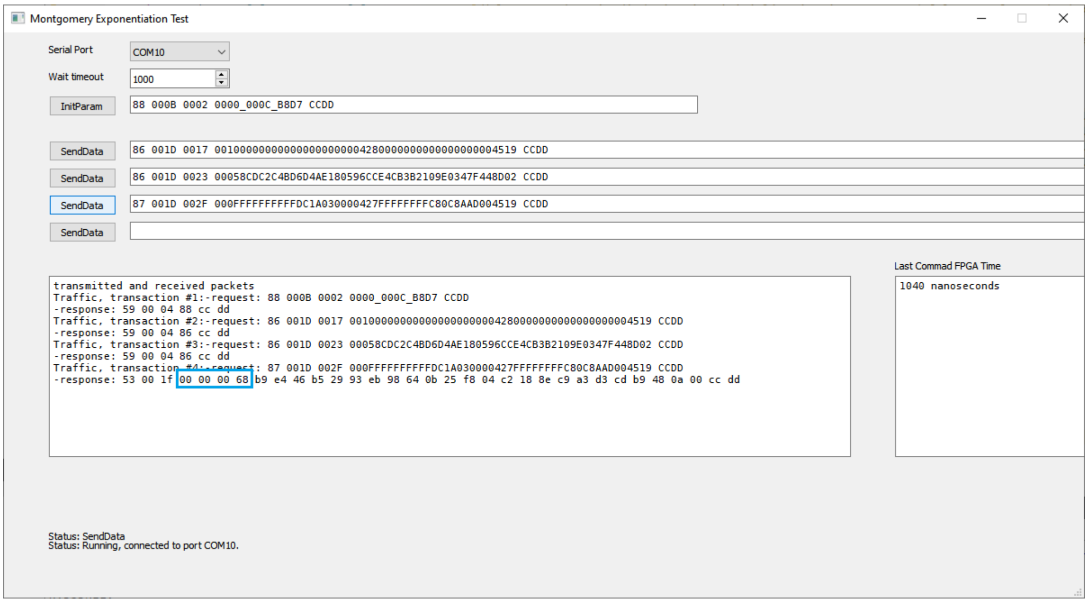
In Figure 4.7, details of `mul_wram_top` module are blurred since they are already explained. cmd_handler_top module consists of four main sub-modules. Two of them are cmd_receiver and cmd_sender which communicate with a personal computer through serial ports. The command packets are saved in FIFO blocks in receiver and sender. `Cmd_handler` module moves arithmetic payload from these buffers to blockrams of `mul_wram_top` module. `Arithmetic_handler` module generates and receives control signals between `cmd_handler` and `cmd_wram_top` modules. Furthermore, there is a timer inside the module which counts number of clocks between start and ready signals of any operation which is requested from multiplier core.

For communication with the test module a TLV (type, length, value) [10] based packet structure is defined. One byte is reserved for type field, two bytes for length field, and variable length is used for value or payload field. Two bytes are reserved for cyclic redundancy check. The details of packet structure is seen in Figure 4.8. Packets starting with 0x8 are transferred to test module, packets starting with 0x5 are transferred from test module.



type	Description
0x86	Memory write
0x87	Memory write and run multiplication
0x88	Memory write and init parameters
0x89	Memory write and run exponantiation
0x83	Run multiplication
0x53	Acknowledge
0x59	Arithmetic result

Figure 4.8: Communication interface with the test module



Serial Port COM10

Wait timeout 1000

```
88 000B 0002 0000_000C_B8D7 CCDD
86 001D 0017 001000000000000000428000000000000000000000004519 CCDD
86 001D 0023 00058CDC2C4BD6D4AE180596CCE4CB3B2109E0347F448D02 CCDD
87 001D 002F 000FFFFFFFFFFDC1A030000427FFFFFFFFC80C8AAD004519 CCDD
```

```
transmitted and received packets
Traffic, transaction #1:-request: 88 000B 0002 0000_000C_B8D7 CCDD
-response: 59 00 04 88 cc dd
Traffic, transaction #2:-request: 86 001D 0017 001000000000000000428000000000000000000000004519 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #3:-request: 86 001D 0023 00058CDC2C4BD6D4AE180596CCE4CB3B2109E0347F448D02 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #4:-request: 87 001D 002F 000FFFFFFFFFFDC1A030000427FFFFFFFFC80C8AAD004519 CCDD
-response: 53 00 1f 00 00 00 68 b9 e4 46 b5 29 93 eb 98 64 0b 25 f8 04 c2 18 8e c9 a3 d3 cd b9 48 0a 00 cc dd
```

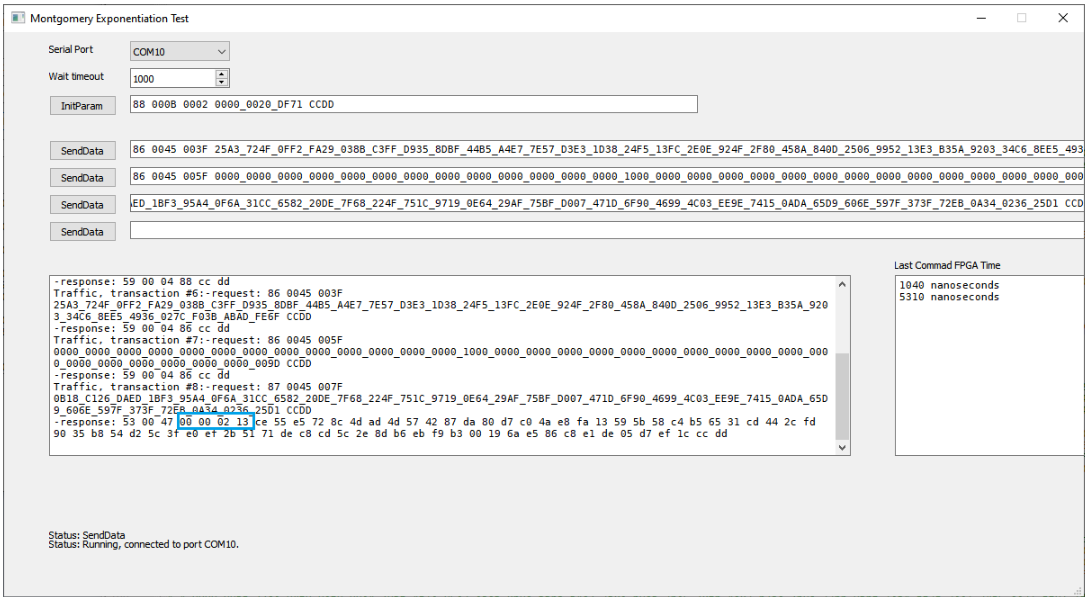
Last Commad FPGA Time

```
1040 nanoseconds
```

Status: SendData
Status: Running, connected to port COM10.

Figure 4.9: Qt test program 192 bits modular multiplication running example

A simple Qt based graphical user interface(GUI) is designed to test the implemented design through serial port. A screen capture of the test program is seen Figure 4.9 and 4.10. The program opens with a predefined input parameters. It can further be



```
Serial Port      COM10
Wait timeout     1000
InitParam        88 000B 0002 0000_0020_DF71 CCDD
SendData         86 0045 003F 25A3_724F_0FF2_FA29_038B_C3FF_D935_8DBF_44B5_A4E7_7E57_D3E3_1D38_24F5_13FC_2E0E_924F_2F80_458A_840D_2506_9952_13E3_B35A_9203_34C6_8EE5_493
SendData         86 0045 005F 0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_1000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_000
SendData         ED_1BF3_95A4_0F6A_31CC_6582_20DE_7F68_224F_751C_9719_0E64_29AF_75BF_D007_471D_6F90_4699_4C03_EE9E_7415_0ADA_65D9_606E_597F_373F_72EB_0A34_0236_25D1 CCD
SendData
```

```
-response: 59 00 04 88 cc dd
Traffic, transaction #6:-request: 86 0045 003F
25A3_724F_0FF2_FA29_038B_C3FF_D935_8DBF_44B5_A4E7_7E57_D3E3_1D38_24F5_13FC_2E0E_924F_2F80_458A_840D_2506_9952_13E3_B35A_920
3_34C6_8EE5_4936_027C_F03B_ABAD_FE6F CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #7:-request: 86 0045 005F
0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_1000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_000
0_0000_0000_0000_0000_0000_0000_009D CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #8:-request: 87 0045 007F
0B18_C126_DAED_1BF3_95A4_0F6A_31CC_6582_20DE_7F68_224F_751C_9719_0E64_29AF_75BF_D007_471D_6F90_4699_4C03_EE9E_7415_0ADA_65D
9_606E_597F_373F_72EB_0A34_0236_25D1 CCDD
-response: 53 00 47 00 00 02 13 ce 55 e5 72 8c 4d ad 4d 57 42 87 da 80 d7 c0 4a e8 fa 13 59 5b 58 c4 b5 65 31 cd 44 2c fd
90 35 b8 54 d2 5c 3f e0 ef 2b 51 71 de c8 cd 5c 2e 8d b6 eb f9 b3 00 19 6a e5 86 c8 e1 de 05 d7 ef 1c cc dd
```

```
1040 nanoseconds
5310 nanoseconds
```

```
Status: SendData
Status: Running, connected to port COM10.
```

Figure 4.10: Qt test program 512 bits modular multiplication running example

programmed according to the command structure in Figure 4.8. In the first example a 192 bits multiplication is seen. In the second example the core is configured for 512 bits modulus and a known test vector is applied through the GUI. The test circuit which manages the multiplier also measures the elapsed time between request and reply of the multiplier. Notice the serial communication or other computer-related delay is not included into time computation. The elapsed time counter is transmitted to the host computer inside communication packets. This section of the packets is encircled in the examples. The hexadecimal cycle counts, `00000068` and `00000213` are later translated to time in terms of nanoseconds. The resolution of the timer is 10 ns, working frequency of the command handler module.

Each command's duration is measured by this way and transmitted to GUI for analysis. Timing diagram of this time-measurement related part is depicted in Figure 4.11. When a start signal is captured from the command handler, the timer starts counting until ready signal is generated. A 32-bit register is reserved to timer which can measure quite long period of time.

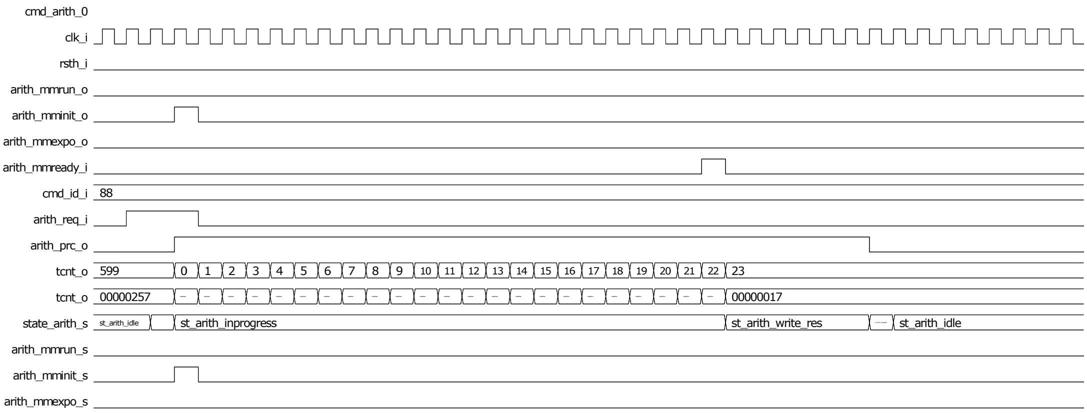
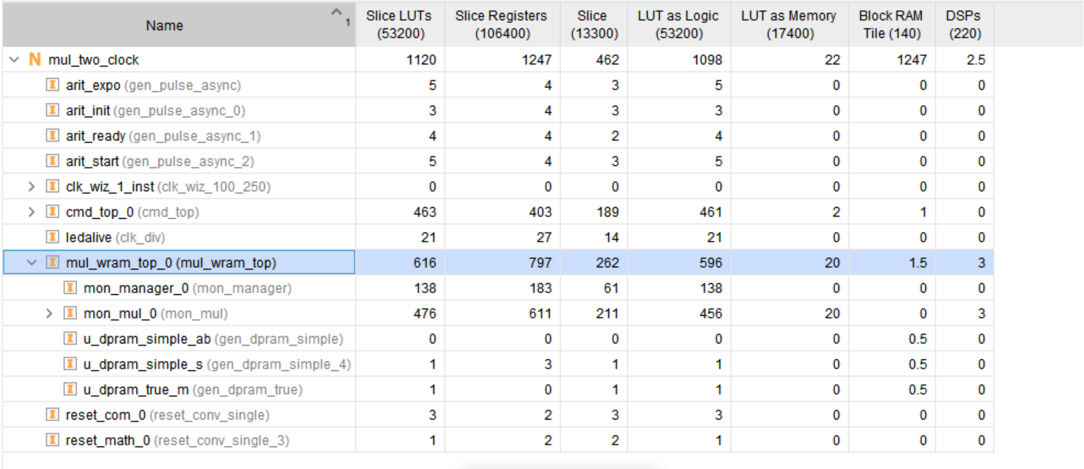


Figure 4.11: Elapsed time counter design for each arithmetic operation

4.5 Implementation Results and Comparisons

The multiplier with exponentiation support is implemented in Xilinx Zynq7020 FPGA. The hierarchical resource utilization of the design together with the test circuit is given in Figure 4.12. The test circuit which consumes 180 slices of logic (448 LUTs+ 403 registers) is not related to multiplier so, they will not be counted in multiplier cost. The multiplier consumes total of 198 slices, 3 DSP and 3 memory blocks. In this family of FPGAs the memory blocks are placed in dual-tiles, each block of 18Kbits memory is counted as 0.5 tile. Together with the exponentiation module (mon_manager), total logic cost reaches to 257 slices. The synthesis frequency obtained for this circuit is 225 MHz. In this version 3 DSP multipliers and 3 block memories are used.



Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)
mul_two_clock	1120	1247	462	1098	22	1247	2.5
arit_expo (gen_pulse_async)	5	4	3	5	0	0	0
arit_init (gen_pulse_async_0)	3	4	3	3	0	0	0
arit_ready (gen_pulse_async_1)	4	4	2	4	0	0	0
arit_start (gen_pulse_async_2)	5	4	3	5	0	0	0
clk_wiz_1_inst (clk_wiz_100_250)	0	0	0	0	0	0	0
cmd_top_0 (cmd_top)	463	403	189	461	2	1	0
ledalive (clk_div)	21	27	14	21	0	0	0
mul_wram_top_0 (mul_wram_top)	616	797	262	596	20	1.5	3
mon_manager_0 (mon_manager)	138	183	61	138	0	0	0
mon_mul_0 (mon_mul)	476	611	211	456	20	0	3
u_dpram_simple_ab (gen_dpram_simple)	0	0	0	0	0	0.5	0
u_dpram_simple_s (gen_dpram_simple_4)	1	3	1	1	0	0.5	0
u_dpram_true_m (gen_dpram_true)	1	0	1	1	0	0.5	0
reset_com_0 (reset_conv_single)	3	2	3	3	0	0	0
reset_math_0 (reset_conv_single_3)	1	2	2	1	0	0	0

Figure 4.12: Utilization results of Montgomery multiplier with 3 DSP, 3 block rams

FPGAs are platforms which have predefined resources of configurable logic, block rams, DSP block and other special hardware resources. These resources are either utilized for some function or remain unused if they are not configured. Logic resources are the common elements which can be utilized as memory or arithmetic function. To observe the effect of this conversion three different variants of the core is realized. This study can be considered as the design space exploration of FPGA resources. The first variant is using logic resources instead of DSP multipliers. The second variant is using logic resources instead of block memories. Total of four implementations are evaluated in Table 4.1. V0 to V3 are these versions. For block memory- distributed memory conversion we only converted one of the block memories into distributed memory. In this case one block memory slightly increased logic slice consumption. The same synthesis frequency is obtained between V0 and V1. When DSP multipliers are moved to logic resources, there is a considerable increase in the number of logic resource usage. Furthermore, synthesis frequency is decreased to $\frac{1}{3}$ of the DSP version. This directly affected the throughput of the multiplier. We confirmed the efficiency of DSP multipliers by this comparison.

Table 4.1: Performance of proposed Montgomery multiplier alternatives (Throughput computed for 1024 bits)

.	width	Option		Time		Area			Macros		Th.put
	bits	DSP	MEM	FREQ	PERIOD	SLICE	LUT	REG	BRAM	DSP	Mbps
V0	16	true	block	225	4.44	262	616	797	3	3	53.05
V1	16	true	disrib	225	4.44	266	667	772	2	3	53.05
V2	16	false	block	75	13.33	500	1453	934	3	0	17.65
V3	16	false	disrib	75	13.33	506	1504	941	2	0	17.65
V4	32	true	block	130	7.69	574	1648	1031	6	11	130.72
[39]	34	true	block	119	8.4	1553	-	-	4	10	133.8
[37]	64	true	block	-	-	-	704	337	8	33	242.66
[37]	32	true	block	-	-	-	425	177	4	11	60.23

Two typical previous work is compared with our design. The work in [39] used a Spartan3 FPGA. They used a systolic-like architecture. With two processing elements their implementation is equivalent to processing 34-bits in one clock cycle. As a consequence their DSP multiplier consumption is higher than ours.For the V4 version where we also used a 32-bit data-path our solution is almost the same as [39]. In [37], different multiplier widths are studied. As the DSP multiplier width is increased, the throughput of the multiplier increased proportionally.

4.6 Conclusion and Future Work

In this section a block-memory and DSP-multiplier based Montgomery multiplier is proposed. As a demonstration of the multiplier modular exponentiation support is also designed as an additional unit. The designed multiplier can be defined as *compact* comparing with the other implementations in the literature. The multiplier is run-time configurable for different modulus values and modulus lengths. The implementation is tested on real FPGA with a test circuit. The test circuit is controlled from a host computer. A graphical user interface is designed to analyze the response time of the circuit.

Our design is structured around the DSP-multiplier which has an internal 25x18 hard multiplier. Since wanted to use the multiplier with the closest standard width, we designed the core as a 16-bit module. We further implemented a 32-bit version of the multiplier. Since the data-width of the memories are doubled, block memory usage is increased by a factor of two. For obtaining 32-bit multipliers we used FPGA tool's generator. This generator produced synthesis result of 130 Mhz by using 4 DSP blocks for a latency of 3 clock cycles. One of the multipliers in the solution requires 2 cycles of latency which resulted in 3 DSP block.

CHAPTER 5

FLEXIBLE COPROCESSOR FOR ELLIPTIC CURVE CRYPTOGRAPHY

Elliptic Curve Cryptography (ECC) [24, 33] became the predominant public-key scheme in the last four decades due to its smaller key sizes compared with other alternatives. It is reported in [17] Table 3 that elliptic curve over 160-bit prime field provides equivalent security of 1024-bit RSA Algorithm. Today there are also mature ECC standards from different organizations [17], [13] [12] and [18]. Being the dominating standard for public-key cryptography, there is a vast amount of research on the implementation of ECC on different software and hardware platforms including application specific integrated circuits (ASIC) and field programmable gate arrays (FPGA).

Together with the improvements in the semiconductor technology and cryptology there is always room for research. While some studies focus on high performance, they miss efficient use of resources. Studies which focus on efficient use of resources put some limitations on their implementation. Fair comparison of architectures and implementations is also an important issue which cannot be satisfied due to evolving technologies in FPGA industry. In this work, while doing a balanced design between area and performance, we will be focusing on the flexibility of the architecture.

In this chapter, we propose an ECC architecture over prime fields which is flexible in various aspects. First flexibility is about the curve equation, we do not restrict the architecture to a specific curve like NIST P-224 or P-256. A novel, small footprint microcontroller is designed and employed for point addition, doubling operations and scalar multiplication operations. Second flexibility is about the length of prime field

GF(p). The architecture that we propose is run-time configurable to different and any prime field sizes, including 192, 384 and 521 which are the generally used bit lengths in the context of elliptic curves over GF(p). Although we utilize vendor dependent hard-multiplier macros, called DSP blocks, it is easy to modify the design to other vendors FPGAs since all of them has some version of multiplier blocks.

The rest of the chapter is organized as follows: In Section 5.1 we mention previously published related architectures. We describe mathematical background of ECC and give information about algorithms that are subject to this chapter. In Section 5.2 we discuss the main architectural ideas that we propose. Later we discuss implementation results and compare with other works Section 5.3 after which we conclude with Section 5.4.

5.1 Related Work

A survey of hardware implementations of elliptic curve cryptosystems can be found in [43]. High-radix Montgomery modular multiplications were considered for ASIC world in [45] where implementations independent of operand length are introduced. Early microcode based architectures were introduced in [30], which was based on Xilinx Virtex XCV300. In [32] they implemented a systolic array implementation of Montgomery modular multiplication in the finite field $GF(2^m)$ by using Virtex XCV800. Embedded multipliers in Virtex2 family FPGAs are used in [31]. Later in [23] they used Virtex-4 FPGAs' resources generously for NIST-224 and NIST-256 curves. In the same years, the paper [55] made use of the same FPGA resources, Virtex-4 SX35 in COPACOBANA platform for prime factorization with Elliptic Curve Method. This work is related to [20] in which that used Spartan-3 FPGAs without DSP resources. More compact, microcode based recent design include [47] where they used Virtex2 resources in different scales. In [46] Virtex2 and Microsemi SmartFusion FPGAs were used to include different levels of HW multipliers for Virtex2 and pure logic for SmartFusion. Their implementation were confined to NIST curves since they used fast reduction algorithm which only works for special form of prime fields. In [42] the same design is enhanced to include RSA algorithm and side-channel protection countermeasures on a Virtex-5 FPGA. In [52] they targeted for lightweight ASIC implementation which consist of no platform dependent hardware multipliers. While our DPRAM architecture is similar to [46], our micro-instruction architecture used the ideas in [52]. In [46] they also used block memories for all operand storing. They explicitly stated this as a design principle. Variable length instruction architecture, is similar to [52].

5.2 Flexible Microcontroller Based Arithmetic Architecture

The block diagram of the architecture that we propose is given in Figure 5.1. `mon_mul` and `mon_manager` modules were detailed in Chapter 4. They remain the same as an interface but, their functionalities have differences. Compared with the architecture in Figure 4.5 a micro-instruction controller is designed to control elliptic curve field operations up to scalar point multiplication. In the previous architecture there were state machines to control multiplication and exponentiation operation. In this architecture desired functionality is loaded into program memory. Initialization operation again exists to configure modulus and prime field size. After that phase the interface memory is loaded with the operands, then a `host_start` signal is triggered from external design. When the execution of the internal program finishes, it raises an acknowledge signal `host_ack` to the external design. Receiving this signal the external design may read the result from the external memory interface.

5.2.1 Microcontroller Design Details

The details of the microcontroller will be described over the instruction set summary. The micro-instructions are 16-bits wide which have one of the three structures as shown in Figure 5.2. In the first set, ARITH_OP, there are three instruction, mmm, msub and madd which are Montgomery modular multiplication, modular addition and modular subtraction operations, respectively. RES_PTR is the index of result location in `ram_ab`, whereas OP1_PTR and OP2_PTR are index locations of operands. The details of arithmetic instructions are as below. Notice that modulus M and R^{-1} are inherent in the instructions.

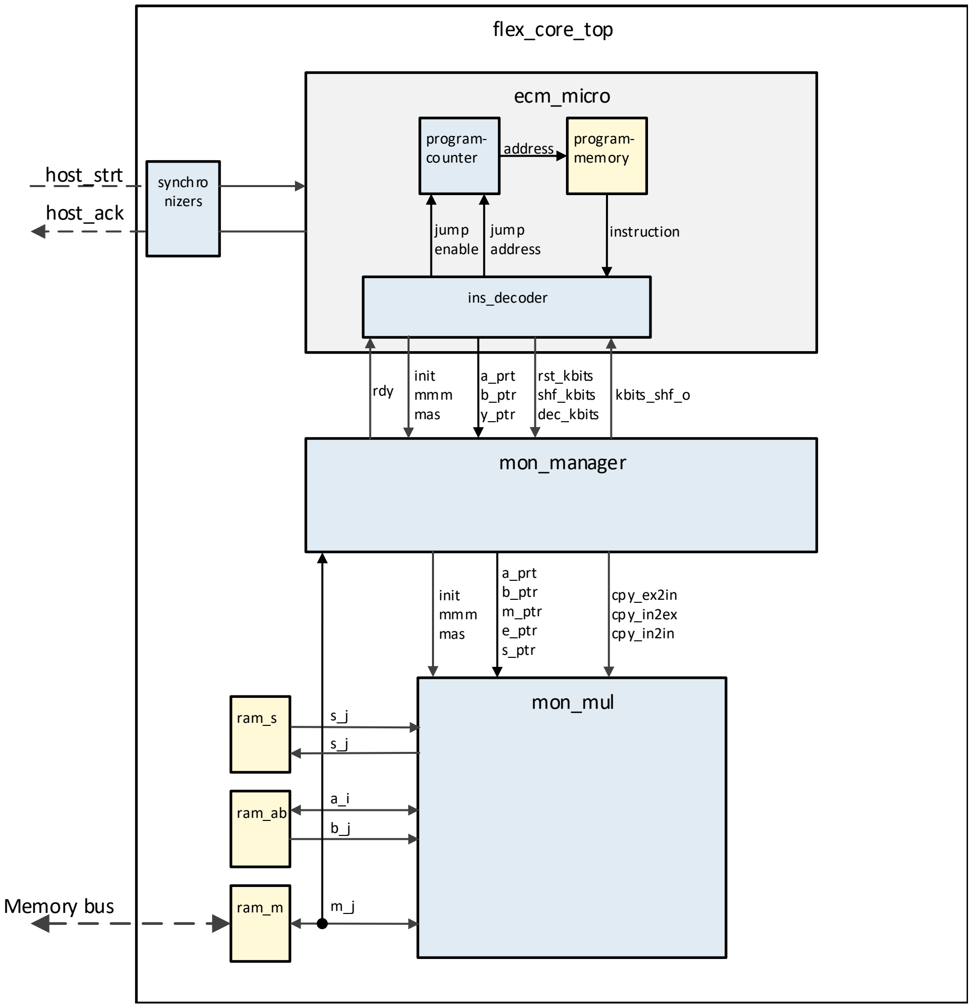


Figure 5.1: Microcontroller based ECC Architecture Block Diagram

- **mmm : Montgomery modular multiplication**

  Syntax example : **`mmm`** `ram_xp ram_t1 ram_t0`

  Function : $x_p \leftarrow mmm(t_1, t_0)$

  Arithmetic : $x_p = t_1 \cdot t_0 \cdot R^{-1} \mod M$

- **madd : Modular Addition**

  Syntax example: **`madd`** `ram_t1 ram_xp ram_zp`

  Function : $t_1 \leftarrow madd(x_p, z_p)$

  Arithmetic : $t_1 = x_p + z_p \mod M$

- **msub : Modular Subtraction**

  Syntax example: **`msub`** `ram_t2 ram_xp ram_zp`

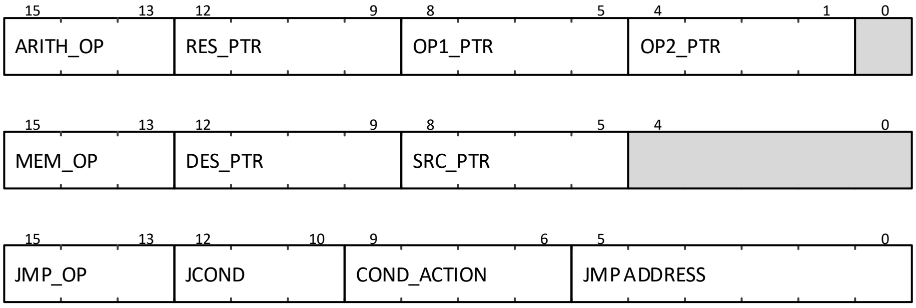


Figure 5.2: Micro-instruction formats

$$\text{Function} : t_2 \leftarrow msub(x_p, z_p)$$

$$\text{Arithmetic} : t_2 = x_p - z_p \mod M$$

The second instruction group consists of memory copy operations. Due to our dual-clock design, arithmetic operands are written from an independent clock domain to `ram_m`. These operands need to be copied into `ram_ab` before arithmetic operations begin. cpei, cpie, and cpii are copy external-to-internal, copy internal-to-external and copy internal-to-internal instructions move operands from `ram_m` to `ram_ab` etc. These instructions have two operands, the first operand is destination index, the second operand is the source index.

The third instruction group actually consists of a single instruction and it is the jump instruction. The instruction executes the following statement:
if JCOND is true then **do** COND_ACTION and **goto** JMP_ADDRESS
With versatile jump conditions and conditional actions, this instruction is used to represent NOP, CALL, RETURN, SET or RESET internal flags and counters of the microcontroller.

Table 5.1: some examples of JMP instruction

#	Instruction	Meaning
1.	`jmp alwy none next`	jump always, do nothing, goto next address (NOP)
2.	`jmp alwy cj_set next`	jump always, set counter j, goto next address (SET)
3.	`jmp alwy cj_dec next`	jump always, decrement counter j, goto next address
4.	`jmp zcntj sr_dec skip_nxt`	jump if cntj is zero , decrement sr, goto address +2
5.	`jmp alwy push ptmlt`	jump always, save return addr in stack, goto ptmlt (CALL)
6.	`jmp alwy pop ret_addr`	jump always, get return addr from stack, goto return addr (RET)

The microcontroller waits for `host_strt` signal after reset. When it receives start signal, it begins execution of instructions by instruction decoder, `ins_decoder`. Each instruction is run in three phases, fetch, decode and execute in a pipelined manner. In fecth phase, the instruction is read from program memory. In decode phase, the instruction is decoded in instruction decoder and necessary control signals are generated. In execution phase, the decoded functionality is realized. Only JMP instructions have constant latency of 3 clock cycles. For the other group of instructions the microcontroller generates run signals, and then wait for the executing module, `mon_mul` to generate ready signals. For this reason they have variable clock cycle latencies depending on the instruction and number of digits in an operand.

The microcontroller is able to support any length of elliptic curve arithmetic. However, the micro-instruction structure is independent of the length of operands: the same assembly code is used for any field length. The length and real addresses of operands are managed in `mon_manager` module. Real addresses of operands are translated in `mon_manager`.

Although its name remained as `mon_mul`, the arithmetic module in this chapter also supports modular addition and subtraction. Subtraction is obtained on the same adder circuit by using 2's complement notation. The algorithm is given by Algorithm 3. Similar memory access operations are handled for modular adder and subtractor circuits.

5.2.2 Scalar Point Multiplication

An efficient algorithm known as Montgomery Ladder for scalar point multiplication was proposed in [36]. The algorithm makes all the computations with only x and z-coordinates. Binary bits of scalar k is scanned from left-to-right. In the algorithm we want to emphasize that trailing zeros before the first 1 is not relevant to the output. The algorithm is analogous to square and multiply algorithm of modular exponentiation in the sense of operand scanning. Unlike from square and multiply, this algorithm is constant-time. For equal-length k's, the latency of the algorithm is independent of the number of 0's or 1's in k. Observing Algorithm 4 it is seen that for each bit the same arithmetic is realized: $P + Q$ and $2Q$ or $2P$.

Algorithm 3 Calculate $A \pm B \mod M$

```
Require: Modulus $M = (m_{n-1}, \ldots, m_0)$, Integer $A, B \in \mathbb{Z}_M$ op $= \pm$,
Ensure: $S = A \pm B \mod M$
$S \leftarrow 0$
if op $= +$ then
    $S \leftarrow A + B - M$
else if op $= -$ then
    $S \leftarrow A - B$
end if
if $S \geq 0$ then
    return $S$
else
    return $S + M$
end if
```

Algorithm 4 Left-to-right Montgomery Ladder Algorithm to obtain $Q = k \cdot P_0$

```
Require: $P_0 = (x_0 :: z_0)$ , $k = (00 \ldots 01k_{n-2} \ldots k_0)$
Ensure: $Q = k \cdot P_0$
$Q \leftarrow P_0,\ P \leftarrow 2 \cdot P_0$
for $i = n - 2$ downto $0$ do
    if $k_i = 0$ then
        $Q \leftarrow 2Q,\ P \leftarrow P + Q$
    else
        $Q \leftarrow P + Q,\ P \leftarrow 2P$
    end if
end for
return Q
```

The elliptic curve group operation $P+Q$, and $2P$ are given in (5.1) and (5.2), respectively. In the addition formulas, for the coordinates of $(x_{P+Q} : : z_{P+Q})$ one needs to know the coordinates of $(x_P : : z_P)$, $(x_Q : : z_Q)$ and $(x_{P-Q} : : z_{P-Q})$. Analyzing the equations it is seen that point addition requires 6 modular multiplications whereas point doubling requires 5 modular multiplications. It can also be observed that if the initial coordinate of z_0 in $(x_0 : : z_0)$ is chosen as 1, one multiplication can be saved in point addition.

$$\begin{aligned} x_{P+Q} &= z_{P-Q}\left[(x_P - z_P)(x_Q + z_Q) + (x_P + z_P)(x_Q - z_Q)\right]^2 \\ z_{P+Q} &= x_{P-Q}\left[(x_P - z_P)(x_Q + z_Q) - (x_P + z_P)(x_Q - z_Q)\right]^2 \end{aligned} \tag{5.1}$$

$$\begin{aligned} 4x_P z_P &= (x_P + z_P)^2 - (x_P - z_P)^2 \\ x_{2P} &= (x_P + z_P)^2 (x_P - z_P)^2 \\ z_{2P} &= 4x_P z_P\left[(x_P - z_P)^2 + 4x_P z_P (A+2)/4\right] \end{aligned} \tag{5.2}$$

The algorithm is traced for an example of $29 \cdot P_0$ in Example 5.2.1. If a 1 is observed in k, the summation is done on point Q, else the summation is done on point P. The algorithm is defined in such a way that the difference $P - Q$ is always constant and equal to initial point P_0. Due to this construction, the coordinates z_{P-Q} and x_{P-Q} can be replaced by x_0 and y_0 in (5.1).

Example 5.2.1. Computation of $29 \cdot P_0$, where $k = (11101)_2$ by Algorithm 4

i	k_i	Q	P	$P-Q$
4	1	P_0	$2P_0$	P_0
3	1	$3P_0$	$4P_0$	P_0
2	1	$7P_0$	$8P_0$	P_0
1	0	$14P_0$	$15P_0$	P_0
0	1	$29P_0$	$30P_0$	P_0

Point Multiplication in microcontroller

The computation of scalar multiplication requires the mapping of Montgomery Ladder Algorithm into the microcontroller assembly. Before the computation begins all input operands are loaded into their respective locations in `ram_m` which is illustrated in Table 5.3. A host_start signal is triggered to start scalar-point multiplication. The micro instruction code moves operands into internal ram as they are necessary. Although `ram_m` is writable from microcontroller side, it is only accessed for reading the modulus M and scalar k during arithmetic operations. Due to this fact it is named as `rom` (read only memory) in the context of microcontroller assembly. For assembly to hex conversion a batch find and replace script is used.

Table 5.2: `ram_m` memory map

index	assembly symbol	arith. value
0	–	–
1	`rom_M`	M
2	`rom_r2`	$r^2 \mod M$
3	`rom_x0`	x_0
4	`rom_z0`	z_0
5	`rom_a2`	$(A+2)/4$
6	`rom_1`	1
7	`rom_k`	k
8	`rom_xq`	x_q
9	`rom_zq`	z_q
10	–	–
11	–	–
12	–	–
13	–	–
14	–	–
15	–	–

Table 5.3: `ram_ab` memory map

index	assembly symbol	arith. value
0	–	–
1	`ram_x0`	x_0
2	`ram_z0`	z_0
3	`ram_a2`	$(A+2)/4$
4	`ram_t0`	t_0
5	`ram_t1`	t_1
6	`ram_t2`	t_2
7	`ram_t3`	t_3
8	`ram_t4`	t_4
9	`ram_t5`	t_5
10	`ram_xq`	x_q
11	`ram_zq`	z_q
12	`ram_xp`	x_p
13	`ram_zp`	z_p
14	`ram_x1`	x_1
15	`ram_z1`	z_1

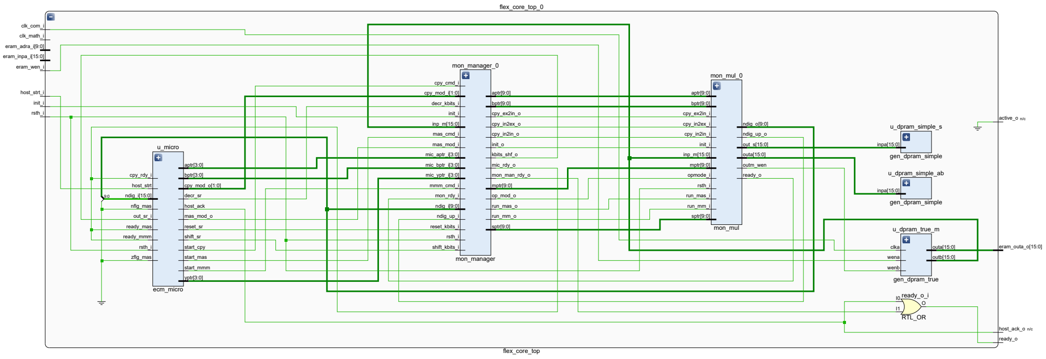
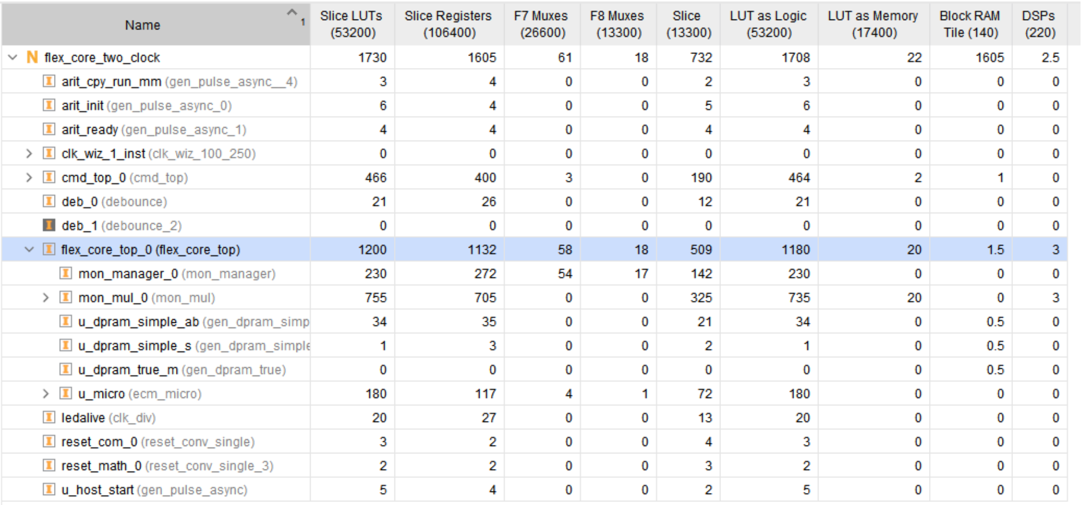


Figure 5.3: Flexible elliptic curve coprocessor RTL schematics

A schematic output of the flexible coprocessor is shown in Figure 5.3. Not all the memory interfaces are shown in the schematics to make the figure more traceable. In the figure it is seen that `mon_manager` stands between the microcontroller and multiplier. All the memory ports except one port of `ram_m` is connected to multiplier.

5.3 Implementation Results and Comparisons

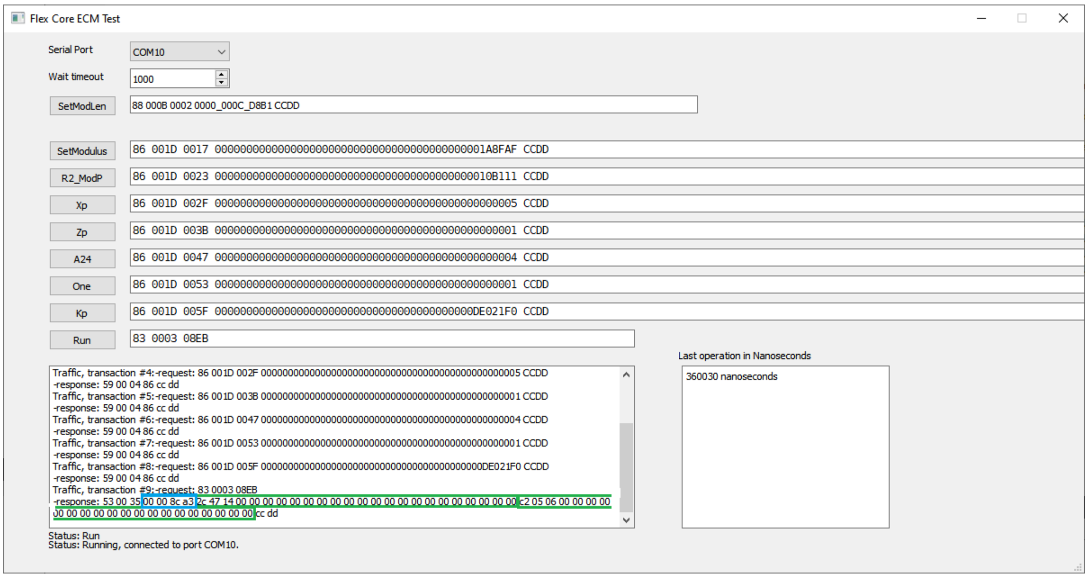
The flexible coprocessor which can compute elliptic curve scalar multiplication is implemented in Xilinx Zynq7020 FPGA. The hierarchical resource utilization of the design together with the test circuit is given in Figure 5.4. The test circuit which consumes 190 slices of logic (466 LUTs+ 400 registers) is not related to coprocessor so, they will not be counted in cost. The multiplier consumes total of 325 slices, 3 DSP and 3 memory blocks. In this family of FPGAs the memory blocks are placed in dual-tiles, each block of 18Kbits memory is counted as 0.5 tile. Together with the address translation module (mon_manager), and microcontroller module (ecm_micro) total logic cost reaches to 509 slices. The synthesis frequency obtained for this circuit is 225 MHz. In this version 3 DSP multipliers and 3 block memories are used.



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)
flex_core_two_clock	1730	1605	61	18	732	1708	22	1605	2.5
arit_cpy_run_mm (gen_pulse_async__4)	3	4	0	0	2	3	0	0	0
arit_init (gen_pulse_async_0)	6	4	0	0	5	6	0	0	0
arit_ready (gen_pulse_async_1)	4	4	0	0	4	4	0	0	0
clk_wiz_1_inst (clk_wiz_100_250)	0	0	0	0	0	0	0	0	0
cmd_top_0 (cmd_top)	466	400	3	0	190	464	2	1	0
deb_0 (debounce)	21	26	0	0	12	21	0	0	0
deb_1 (debounce_2)	0	0	0	0	0	0	0	0	0
flex_core_top_0 (flex_core_top)	1200	1132	58	18	509	1180	20	1.5	3
mon_manager_0 (mon_manager)	230	272	54	17	142	230	0	0	0
mon_mul_0 (mon_mul)	755	705	0	0	325	735	20	0	3
u_dpram_simple_ab (gen_dpram_simp	34	35	0	0	21	34	0	0.5	0
u_dpram_simple_s (gen_dpram_simple	1	3	0	0	2	1	0	0.5	0
u_dpram_true_m (gen_dpram_true)	0	0	0	0	0	0	0	0.5	0
u_micro (ecm_micro)	180	117	4	1	72	180	0	0	0
ledalive (clk_div)	20	27	0	0	13	20	0	0	0
reset_com_0 (reset_conv_single)	3	2	0	0	4	3	0	0	0
reset_math_0 (reset_conv_single_3)	2	2	0	0	3	2	0	0	0
u_host_start (gen_pulse_async)	5	4	0	0	2	5	0	0	0

Figure 5.4: Utilization results of ECC scalar multiplication with microcontroller

The same test capability used in Chapter 4 is also employed here. A sample run of the ECM scalar point multiplication is given in Figure 5.5. The memory map is filled according to Table 5.2, then a start signal is sent to the ECM core. The result of the scalar point multiplication is sent back to the GUI together with an elapsed time counter. This is indicated with blue rectangle, the point coordinates of x_p and z_p are indicated in green rectangles in Figure 5.5.



```
Serial Port   COM10
Wait timeout  1000
SetModLen     88 000B 0002 0000_000C_D8B1 CCDD

SetModulus    86 001D 0017 0000000000000000000000000000000000000000001A8FAF CCDD
R2_ModP       86 001D 0023 00000000000000000000000000000000000000000010B111 CCDD
Xp            86 001D 002F 000000000000000000000000000000000000000000000005 CCDD
Zp            86 001D 003B 000000000000000000000000000000000000000000000001 CCDD
A24           86 001D 0047 000000000000000000000000000000000000000000000004 CCDD
One           86 001D 0053 000000000000000000000000000000000000000000000001 CCDD
Kp            86 001D 005F 0000000000000000000000000000000000000000000DE021F0 CCDD
Run           83 0003 08EB
```

```
Traffic, transaction #4:-request: 86 001D 002F 000000000000000000000000000000000000000000000005 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #5:-request: 86 001D 003B 000000000000000000000000000000000000000000000001 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #6:-request: 86 001D 0047 000000000000000000000000000000000000000000000004 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #7:-request: 86 001D 0053 000000000000000000000000000000000000000000000001 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #8:-request: 86 001D 005F 0000000000000000000000000000000000000000000DE021F0 CCDD
-response: 59 00 04 86 cc dd
Traffic, transaction #9:-request: 83 0003 08EB
-response: 53 00 35 00 00 8c a3 2c 47 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 c2 05 06 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 cc dd
```

```
360030 nanoseconds
```

```
Status: Run
Status: Running, connected to port COM10.
```

Figure 5.5: Qt test program 192-bits scalar point multiplication running example

5.4 Conclusion and Future Work

In this chapter we presented a microcontroller based design to perform arithmetic operations in prime fields with an operand-length independent Montgomery modular multiplication engine. Since we want to use the coprocessor for integer factorization with elliptic curve method, we configured the core with a scalar point multiplication code. The code is written with a special assembly language. The assembly is translated into binary and saved in a ROM near microcontroller. A curve-independent and prime-size independent design is obtained with a similar area performance of [46].

In the next chapter the use of this coprocessor in an embedded FPGA processor will be deployed.

CHAPTER 6

ELLIPTIC CURVE METHOD AS A PERIPHERAL TO EMBEDDED FPGA PROCESORS

The design and implementations in the previous chapters of this thesis constructed a base for different numerical algorithms in prime fields, $\mathbb{F}_p$, and especially on elliptic curves, $E(\mathbb{F}_p)$. In this chapter we will propose the use of this infrastructure for elliptic curve method of factoring large integers.

The FPGA core which is designed as a peripheral to an embedded processor is employed for elliptic curve method. A hardware-software partitioning of the factoring algorithm is proposed. Multiple instances of the core is instantiated in the processing sub-system. The whole FPGA solution is constructed to work with minimal communication need to a remote main controller computer.

The outline of the Chapter is as follows: In Section 6.1 we outline the ECM algorithm and give numerical examples from software implementation. A closely related factoring algorithm, Pollard's (p-1)-method is also mentioned in this section. In Section 6.2 the use of ECM peripheral will be described in a processing subsystem. The main time-consuming part of the algorithm is the scalar point multiplication. The ECM peripheral is responsible for scalar point multiplication. However, pre-computations and post-computations of the algorithm exist. Implementation alternatives for these operations are also mentioned in this section. In Section 6.3, multiple instances of the peripheral is discussed and management of these units in software is proposed. In Section 6.4, the results of resource usages and time latencies are summarized.

6.1 Integer Factorization with Elliptic Curve Method

Lenstra published the Elliptic Curve Method (ECM) paper in 1984. The paper describes the powerful algorithm for factoring large integers [29]. Before ECM, a closely related algorithm, Pollard(p-1) method, is described [41].

6.1.1 Pollard's (p-1) Method of Factoring

ECM method is inspired by Pollard's (p-1) method which we describe in this subsection. For describing the method some number theory definitions are stated.

Theorem 6.1.1 (Fundamental theorem of arithmetic)**.** Every integer larger than 1 can be written uniquely as a product of primes up to the order of factors.

For example $1200 = 2^4 \cdot 3^1 \cdot 5^2$. The theorem states that however the factors are written, there will be four 2s one 3 and two fives. Since primes are required according to the theorem, $20 \cdot 30 \cdot 2$ is not a valid factorization.

Definition 6.1.1. Let n be a positive integer, $n = \prod {p_i}^{e_i} n =$, then n is B-powersmooth if ${p_i}^{e_i} \leq B$ for all i.

For example 1200 is 25-powersmooth where as $640 = 2^7 \cdot 5$ is 128-powersmooth.

The Pollard's (p-1) algorithm works as follows. Let N be a number that we want to factorize. We suppose that N has a prime divisor p. Due to Fermat's Little Theorem, it is known that

$$a^{p-1} \equiv 1 \mod p.$$

Any multiple of $p-1$ also satisfies the relation:

$$a^{(p-1)m} \equiv 1 \mod p$$

Converting these equivalence relations to divisibility expressions we obtain

$$p \mid (a^{p-1} - 1).$$

Any multiple of $p-1$ also satisfies the relation:

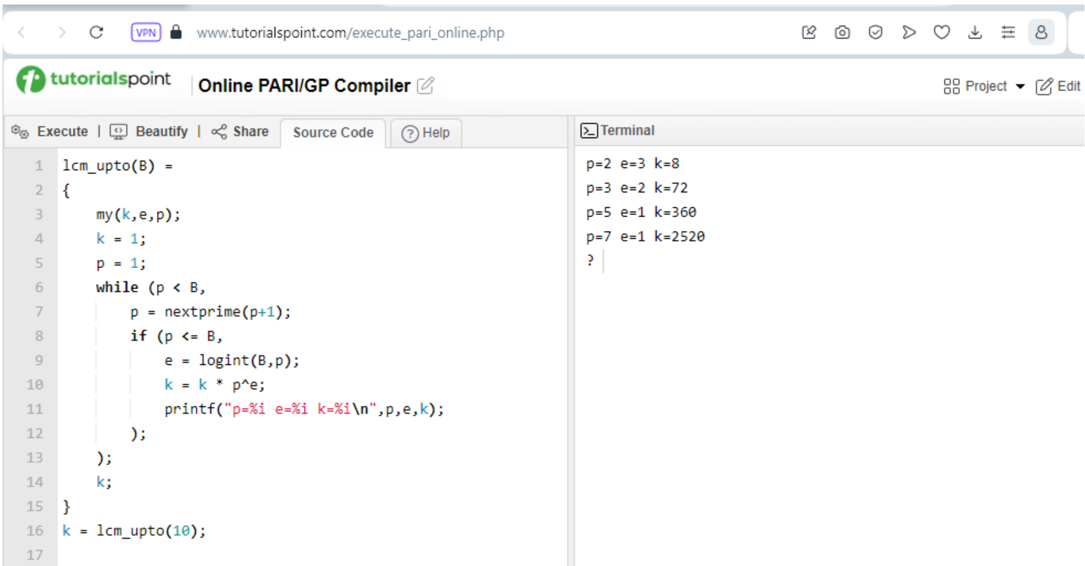
$$p \mid (a^{(p-1)m} - 1)$$

Therefore,

$$p \mid gcd(a^{(p-1)m} - 1, N).$$

If we could calculate $gcd(a^{(p-1)m} - 1, N)$ we could find p or some multiple of it. However, p is what we are looking for. For some primes this difficulty can be overcome and $gcd(a^{(p-1)m} - 1, N)$ can be computed without knowing p. Suppose that $p - 1$ is the product of small primes to small powers, in other words $p - 1$ is B-powersmooth. We can obtain a number k that is the product of many small primes to small powers. By this way we may reach to a number such that $k = (p-1)m$ for some m. If $gcd(a^k - 1, N) < N$ is found then a nontrivial factor of N is reached. If a gcd of 1 is reached some higher smoothness bound should be chosen and tried again.

The number k in the above paragraph is called least common multiples of numbers up to B where it is called the smoothness bound. For example lcm_upto$(10) = 2^3 \cdot 3^2 \cdot 5 \cdot 7 = 2520$. For numerical examples we have employed Pari-GP interpreter which is an open-source computer algebra system. It can be installed with a small binary or can be used through different online versions. An example online Pari-GP interpreter is shown in Figure 6.1. Here



```
lcm_upto(B) =
{
    my(k,e,p);
    k = 1;
    p = 1;
    while (p < B,
        p = nextprime(p+1);
        if (p <= B,
            e = logint(B,p);
            k = k * p^e;
            printf("p=%i e=%i k=%i\n",p,e,k);
        );
    );
    k;
}
k = lcm_upto(10);
```

```
p=2 e=3 k=8
p=3 e=2 k=72
p=5 e=1 k=360
p=7 e=1 k=2520
?
```

Figure 6.1: LCM upto smoothness bound computation with Pari-GP

The algorithmic steps of Pollard's (p-1) method for factoring N can be summarized as follows.

- Compute lcm : Choose a smoothness bound B and compute k

- Set $a = 2$, compute $x = a^k \mod N$ and $g = gcd(x, N)$

- if $q \neq 1$ or $q \neq N$ a factor of N is obtained

- if $q = 1$ or $q = N$ try with a different a and larger B

The Pari-GP function can be seen in the Listing 6.1 together with the lcm computation of prime powers up to smoothness bound B. This and some other example codes of this thesis will be available at [44].

```
lcm_upto(B) =
{
    my(k,e,p);
    k = 1;
    p = 1;
    while (p < m,
        p = nextprime(p+1);
        if (p <= m,
            e = logint(m,p);
            k = k * p^e;
        );
    );
    \\ printf("k= %i = 0x%X\n",k,k);
    k;
}
pollard(n,a,b)=
{
    k = lcm_upto(b);
    a= Mod(a,n);
    g=gcd(a^k-1,n);
    g;
}
```

Listing 6.1: Listing Pollard's (p-1) Method

As an example if Pollard's method is run for $N = 38057, a = 2, B = 8$ no factor is found. But if the smoothness bound B is set to 9, the factor 19 is found. Inspecting the example we see that $38057 = 19 \cdot 2003$, where $p - 1$ values are 18 and 2002. 18 of powersmooth of 9. Lcm up to 8 is 840, where lcm upto 9 is 2520. While 2520 contains $(p - 1) = 18$ inside itself, 840 does not contain 18. Because of this $B = 8$ is not enough for this example.

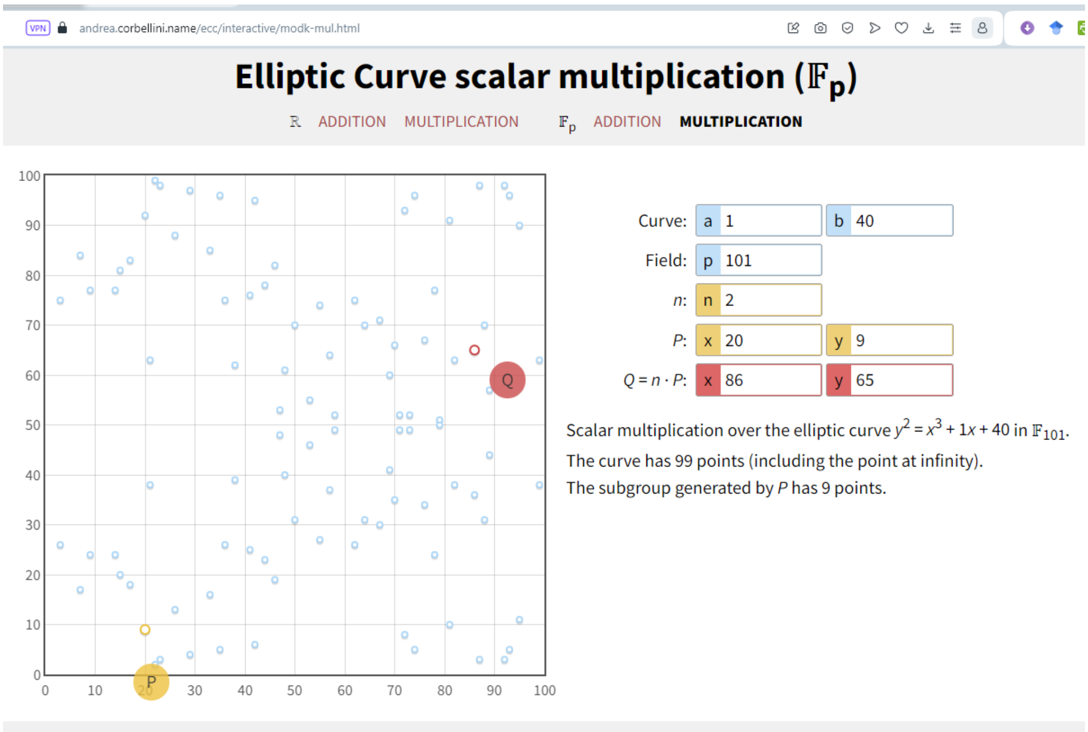
6.1.2 Elliptic Curve Method of Factoring

Pollard's (P-1) method relies on the possibility of smoothness $p-1$ and $q-1$ of $N = p.q$. If the numbers $p-1$ and $q-1$ are not smooth, the method fails for the defined bound B. As a numerical example take $N = 22523 = 101 \cdot 223$ where $p-1$ and $q-1$ are 100 and 222. These numbers are decomposed as $100 = 2^2 \cdot 5^2$, and $222 = 2 \cdot 3 \cdot 37$. We observe that they are 25-powersmooth, 37-powersmooth respectively. Pollard's (P-1) does not work with a smoothness bound of, let's say, $B = 20$ for this case. However, $p-2 = 99$ and $q-2 = 221$ are 11-powersmooth and 17-powersmooth respectively. If there was a way to formulate the Fermat's Theorem with these numbers, we would be able reach a factor with the same smoothness bound $B = 20$.

Elliptic Curve Method replaces the multiplicative field of $(\mathbb{F}_p^*)$, which has group order $p-1$ by the group of points on elliptic curve E over $(\mathbb{F}_p)$. According to the Hasse's theorem the number of points on a curve, i.e the group order of the curve, is bounded in the range $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. For the above examples of $p = 101$ and $q = 223$ we can have curves of orders in the range $[82, 122]$ and $[195, 253]$. Again following the previous argument it is possible to obtain elliptic curves of group orders of 99 and 221 since they are inside the Hasse bound. As an example, curve $y^2 = x^3 + x + 40$ in $(\mathbb{F}_{101})$ of order 99 is depicted in Figure 6.2.

The algorithmic steps of Elliptic Curve Method of factoring N can be summarized as follows.

- Compute lcm: Choose a smoothness bound B and compute k
- Choose random elliptic curve : Set $a \in \mathbb{Z}_N$, such that $4a^3 + 27 \in \mathbb{Z}_N^*$, then $P = (0, 1)$ is a point on the elliptic curve $y^2 = x^3 + ax + 1$ over $\mathbb{Z}_N$
- Scalar point multiplication: Compute $k \cdot P$. If at some point sum of points is not possible because modular inverse with respect to N does not exists, we compute the greatest common divisor of this number with N and return it as a nontrivial factor of N
- If $k \cdot P$ can be computed without any problem, repeat the steps with other



andrea.corbellini.name/ecc/interactive/modk-mul.html

Elliptic Curve scalar multiplication ($\mathbb{F}_p$)

ℝ ADDITION MULTIPLICATION $\mathbb{F}_p$ ADDITION MULTIPLICATION
0
10
20
30
40
50
60
70
80
90
100
P
Q

Curve: a 1 b 40

Field: p 101

n: n 2

P: x 20 y 9

$Q = n \cdot P$: x 86 y 65

Scalar multiplication over the elliptic curve $y^2 = x^3 + 1x + 40$ in $\mathbb{F}_{101}$.

The curve has 99 points (including the point at infinity).

The subgroup generated by P has 9 points.

Figure 6.2: Online elliptic curve demonstration tool [6]

curves.

A second phase of the algorithm is later proposed by Brent [16] and Montgomery [36]. If a factor with the smoothness bound $B = B_1$ is not found, a second larger bound B_2 is chosen. The last point obtained in the first phase is multiplied with the primes between B_1 and B_2. The same gcd computation is done when an inversion computation fails.

The curve of the form

$$y^2 = x^3 + ax + 1$$

is called the Weierstrass form and the original algorithm is defined with this form. Later in [36] Montgomery suggested the below form of curve, later called as Montgomery's form:

$$by^2 = x^3 + ax^2 + x$$

where $b \neq 0$ and $a^2 \neq 4$ with all operations are implicitly implied as modulo N. The curves are also defined with the assumption that N is a prime. Weierstrass to Montgomery's form conversion can be done with the change of variables

$$x \rightarrow (3x + a)/(3b),$$

$$y \to y/b,$$

$$a \to (3 - a^2)/(3b^2),$$

$$b \to (2a^3/9 - a)/(3b^3).$$

Furthermore, homogeneous form of the curve is used in computer implementations.

$$by^2z = x^3 + ax^2z + xz^2.$$

In this case the triple $(x : y : z)$ projective coordinates corresponds to the affine coordinates $(x/z : y/z)$, if it is not the point of infinity. The curve parameters are determined by Suyama's Parametrization method. The method starts with a $\sigma > 5$ value obtains a, b, x_0 and y_0 values

$$\begin{aligned} u = \sigma^2 - 5, v = 4\sigma \\ x_0 = u^3, z_0 = v^3 \\ a = \frac{(v-u)^3(3u+v)}{4u^3v} - 2 \end{aligned}$$

This method is widely used, and it is possible to reproduce factorization between different platforms. The parameters of b and y are not needed for ecm algorithm. All the operations include x and z only. We only use x and z coordinates and write $P = (x :: z)$ as was discussed in Section 5.2.2.

As a reference software implementation we employed GMP-ECM by Paul Zimmermann et al. [53]. The implementation is done in C language to support various processor platforms. As a typical computing platform we ported the implementation to a 64 bit Intel 7 processor laptop. As a compiler we used Visual Studio 2019. The underlying arithmetic library was GMP[9] for GMP-ECM where the name implies. For better Visual Studio and Microsoft compatibility we used MPIR library [4] instead of the GMP. MPIR is a fork from GMP which is aimed to natively support Microsoft C compilers.

We further modified the source code to output debug information through different stages of implementation.

Example 6.1.1. Consider $N = 137975715611134168913629730197855098971975874567487892 3 = 0xE67C42800000000000000000000000000000001B028FCB$ for factoring with GMP-ECM. Two different results are going to be displayed for analysis.

```
ecm -v -v -inp b180 -param 0 -sigma 7 960 50000
ecm -v -v -inp b180 -param 0 -sigma 10 960 50000
```

In this example `b180` is a text file which has the above N value which is around 180 bits. The algorithm rounds up to 192 bits for computations. The results for $\sigma = 7$ and $\sigma = 10$ are given in Listings A.1 and A.2. For $\sigma = 7$, Phase 1 (Stage 1) of the algorithm does not reach to a solution with the given bounds. For $\sigma = 10$, the selected curve reaches to solution in Phase 1.

Examining the reduced execution logs in Listings A.1 and A.2, we see that point multiplication algorithm is independently called for each prime. Another property of the software implementation is that point multiplication for primes larger than 3 are computed with an adder chain method, PRAC [34], instead of Montgomery ladder method. This method saves the number of multiplications in the expense of more calls to the scalar multiplication function. In software implementation increased number of calls does not affect the performance since the function is called through memory pointers and there is a single memory between the arithmetic logic unit and the computer memory.

In FPGA implementation there are two different memory regions. One is connected to the embedded processor, the other is inside the coprocessor. Since we will run the ECM scalar multiplication in parallel, each coprocessor will have its independent memory region. Moving curve parameters from embedded processor memory to coprocessor memory will be done by embedded software. Because of this architecture difference we want to decrease the calls to the coprocessor. In the above particular example the lcm up to $B_1 = 960$ is a huge number which is 1374 bits as seen in Listing A.1, `big_r`, starting with 2810, ending with f600. This big number is far beyond the field size, 192 bits, in this example. For reaching the $k.P$ result, we divide number k into smaller parts that can fit into field size. The details will be described in the next section.

6.2 Use of ECM peripheral in a Processing Subsystem

The coprocessor in the previous chapter was designed as a general scalar point multiplication design specialized for elliptic curve cryptography. In this chapter this coprocessor will be employed as a peripheral to embedded processors inside FPGAs. These processors are widespread among FPGA vendors. As an easily accessible member of this class, we used a Zynq based processing system as development environment. Other processing subsystems can also be considered for evaluation.

In this chapter we verified the ECM peripheral first with two instances connected to the Zynq processor as in Figure 6.3. Later the performance will be measured with more cores. The red blocks in Figure 6.3 are two instances of ECM cores which has 16-bit data-path width. The peripherals are connected to processor through AXI bus interface [1]. Each peripheral has two axi connection, one of which is used for register interface, the other is used as block memory interface. Each core also has two clocks. One of the clock is a AXI interface clock the other is the processing clock of arithmetic operations as was discussed in previous chapters. The core have different outputs for diagnosis purposes. The critical output of the core is the `ready_o` output which is used as interrupt input to the processor IRQ (interrupt request) input.

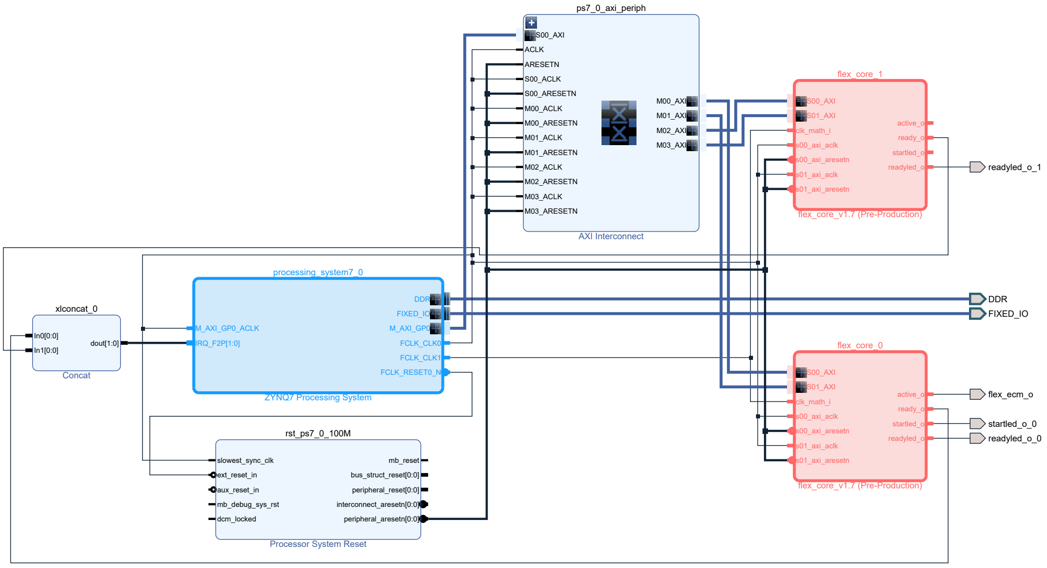


Figure 6.3: Zynq SOC(System on Chip) with two instances of ECM coprocessors

The more blue block in Figure 6.3 corresponds to Zynq processing subsystem. Since we are using only limited capability of the subsystem, it has a few connections. A detailed picture of the Zynq processing subsystem is seen in Figure 6.4, wherre the Processing System (PS) is detailed. It has a dual Arm Cortex-A9 Application Processing Units, several I/O peripherals, memory interface controllers and several interconnect blocks are present in the PS side. In the lower part of the figure, Programmable Logic (PL) part of the FPGA is seen. Since the figure is focusing on the PS side, most of the figure's area is allocated by PS. In real world implementations, PL part dominates most of the area of FPGA.

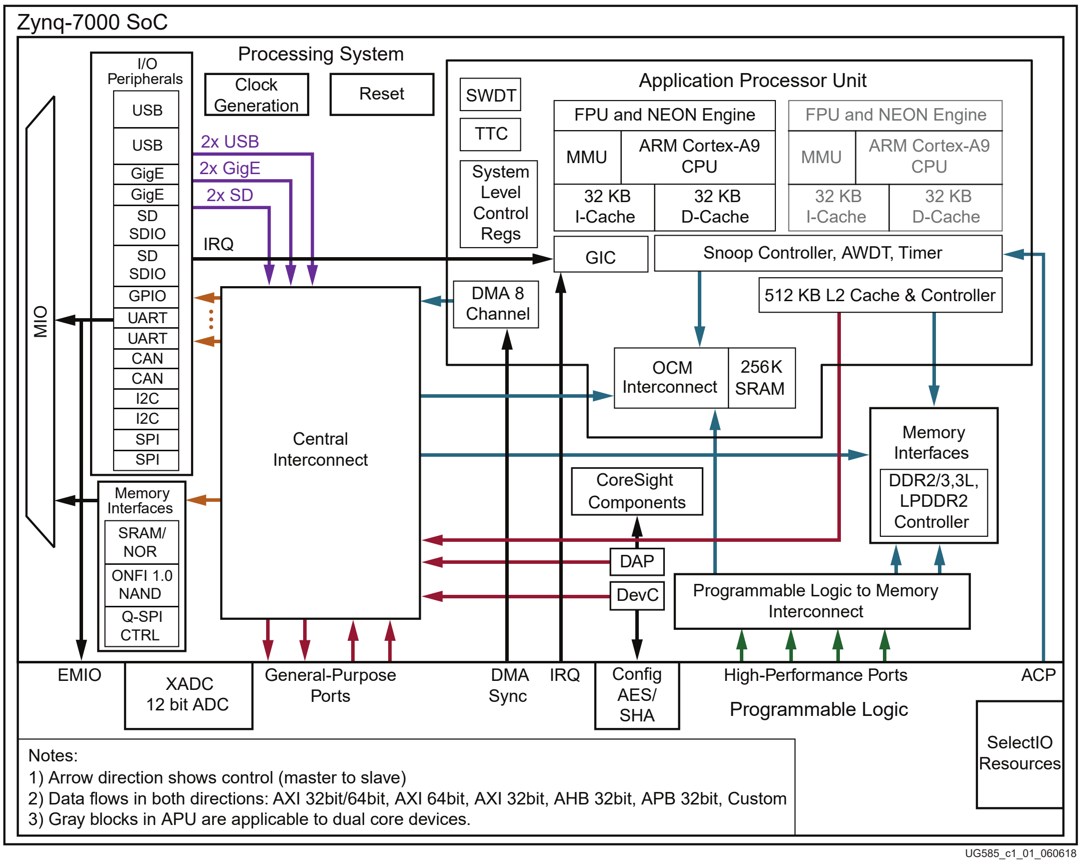


Figure 6.4: Zynq 7000 SOC(System on Chip) Overview [51]

Before feeding scalar point multiplication inputs, the ECM peripheral core needs to be initialized with the modulus, and number of digits. The modulus related parameter m' is written to register-space of the peripheral. Once this initialization is done, memory map of the peripheral is filled with data according to the Table 5.3. Then the run trigger of the peripheral is set through register space. The peripheral will generate the results and output an interrupt when the result is ready. By the interrupt mechanism the software on the embedded processor will not need to poll the output of multiple cores. Once the result is ready the processor will either read the result or will generate another point multiplication request. For a k value of around 1374 bits as in Example 6.1.1, the k value will be divided into smaller parts like $k = k_0 \cdot k_1 \cdots k_7$. The memory map of the peripheral is organized so that only new k value is written for multiple calls. For $B = 960$ an example set of initializations of a 192-bits core is seen as below

```
char ram_mod[] = "000E67C428000000000000000000000000000000001B028FCB";
char ram_r2mp[] = "00008191D00000000000000000003840000000000000F2F166";
char ram_x0[] = "00002B018A8AD278E8DCEBD930835BCD3F70B3201832C839";
char ram_z0[] = "000000000000000000000000000000000000000000000001";
char ram_a24[] = "0001DB5E4E84D2789F5D56943586474641FFB9A8E9A04C9F";
char ram_one[] = "000000000000000000000000000000000000000000000001";

char kparams[numof_k][48] = {
"000048560dd908d23f1bf9510081e21e5a7dd5d5207d5600",
"000060946cd4ae17f452d0e3542a24b11c4dee78abc24131",
"000b91f95bd45d3da8850514c26f81cf6caab0407400584b",
"00003f3c80ae2354787ad750798f36af240cee2b94164915",
"0000173b8f66ed0a1b971dd2f6b29321912ae7665d2e3367",
"0003f7970e696052440a31d5fe43a81615d35bade4a1fe35",
"000025be4dbbef0c97aa21dbb4bb1324cf799cbcb7eb002d",
"000000000009ad638c152bd2e71f865bb2887d73170f7141"};
char xref[] = "000B14718F87E6D4D6CF4ABD33475D34401080253A13398D";
char zref[] = "0005707D48825FDBD2C2A0A0E224B6AAFADEBDC9BB041924";
```

For verification purposes these values are generated in GMP-ECM implementation and the outputs `xref` and `zref` are checked at the end of 8 successive calls of the core in this case. In this example a point $P = (x_0, z_0)$ is multiplied with a k value which is composed of 192-bits values is in `kparams`.

The peripheral core generates a hardware controlled timer value together with the interrupt. By this way the latency of the computation is measured with a precision of 10 nanoseconds, one of the clock periods of Zynq embedded processor. The software log of the above vectors are as shown below:

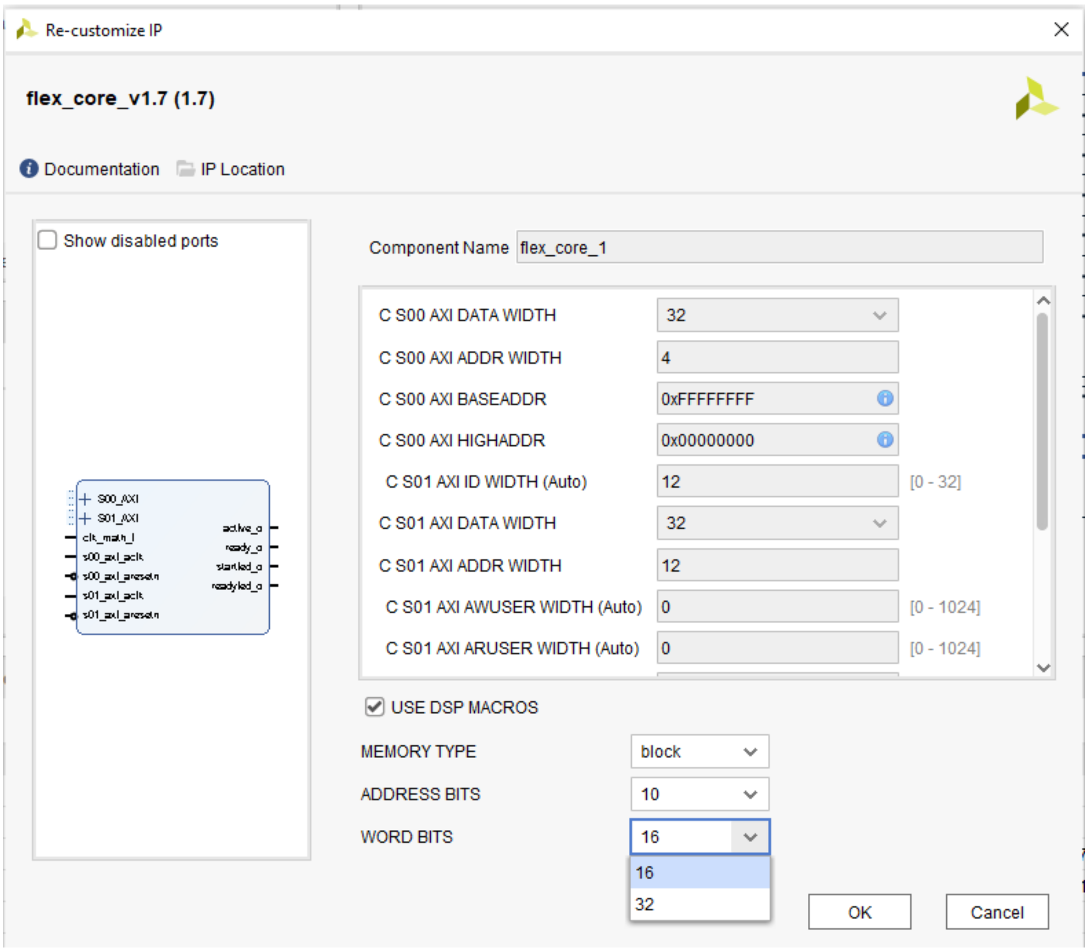
```
::::::  ECM 0 ref ::::::
k = 0:  smul time = 2442.7000 us pnding_core[0] 0
k = 1:  smul time = 2440.2300 us pnding_core[0] 0
k = 2:  smul time = 2508.1500 us pnding_core[0] 0
k = 3:  smul time = 2423.3600 us pnding_core[0] 0
k = 4:  smul time = 2401.1300 us pnding_core[0] 0
k = 5:  smul time = 2473.1000 us pnding_core[0] 0
k = 6:  smul time = 2410.3200 us pnding_core[0] 0
k = 7:  smul time = 2057.6800 us pnding_core[0] 0
xp strings match
zp strings match
::::::  ECM 1 ref ::::::

k = 0:  smul time = 2442.7000 us pnding_core[1] 0
k = 1:  smul time = 2440.2400 us pnding_core[1] 0
k = 2:  smul time = 2508.1500 us pnding_core[1] 0
k = 3:  smul time = 2423.3500 us pnding_core[1] 0
k = 4:  smul time = 2401.1300 us pnding_core[1] 0
k = 5:  smul time = 2473.1000 us pnding_core[1] 0
k = 6:  smul time = 2410.3200 us pnding_core[1] 0
k = 7:  smul time = 2057.6800 us pnding_core[1] 0
xp strings match
zp strings match
```

In results section we will summarize the results. However, we want to emphasize the precision of the timers. The first and second groups correspond to different peripherals. Since the last k parameters in this example is smaller than the others, it returned faster than the other k values. The configuration in this case corresponds to 192-bit modulus, 16-bits data-width, 200 Mhz arithmetic processor clock.

Pre-computations and post-computations: The number to be factorized must be given definitely as an input to FPGA design. The smoothness bound is also a design parameter. It is not practical to transfer the k number which is computed as described above. For k computation and other pre-computations we need to do big-integer arithmetic on the embedded processor inside FPGA. For this purpose we have chosen to use the big-integer library described in the book Cryptography in C and C++[48]. The library was available at [49].

ECM-Peripheral Variants: The modulus and number of digits are run-time parameters that can be adjusted without compiling the source code. As discussed in Chapter 5, use of multipliers, use of block memories are compile-time parameters which can be decided when the core is being instantiated. In this chapter we also incorporated the data-width as a compile-time parameter. It is possible to instantiate the core in either 16-bits or 32-bits mode. The parametrization can be done with a drop-down menu in FPGA design tool as shown in Figure.



☐ Show disabled ports

Component Name flex_core_1

C S00 AXI DATA WIDTH	32	
C S00 AXI ADDR WIDTH	4	
C S00 AXI BASEADDR	0xFFFFFFFF	
C S00 AXI HIGHADDR	0x00000000	
C S01 AXI ID WIDTH (Auto)	12	[0 - 32]
C S01 AXI DATA WIDTH	32	
C S01 AXI ADDR WIDTH	12	
C S01 AXI AWUSER WIDTH (Auto)	0	[0 - 1024]
C S01 AXI ARUSER WIDTH (Auto)	0	[0 - 1024]

☑ USE DSP MACROS

MEMORY TYPE	block
ADDRESS BITS	10
WORD BITS	16

Figure 6.5: ECM peripheral compile-time parameters

6.3 Parallel Instantiation and Managing ECM Cores

We have already verified two cores in design shown in Figure 6.3. To see the performance of the solution with more peripherals in the processing subsystem 8 cores are instantiated similar to Figure 6.3. All these cores are continuously updated with fresh parameters for one second with a function `feedCore` as in Listing 6.2. Each core's running counts are printed at the end of one second. It is observed that 8 cores can work without having any degradation. In other words, when all the cores are run, each core's count is equal to the number when only a single core is active.

```
// 1 second measurement loop /////////////
XTime_GetTime(&bef);
int cont = 1;
xil_printf("Entering 1 second Loop\n\r");

while (cont == 1) {

  //usleep(1);
  feedCore(0, ec_x0, ec_z0, xref, zref);
  feedCore(1, ec_x0, ec_z0, xref, zref);
  feedCore(2, ec_x0, ec_z0, xref, zref);
  feedCore(3, ec_x0, ec_z0, xref, zref);
  feedCore(4, ec_x0, ec_z0, xref, zref);
  feedCore(5, ec_x0, ec_z0, xref, zref);
  feedCore(6, ec_x0, ec_z0, xref, zref);
  feedCore(7, ec_x0, ec_z0, xref, zref);

  XTime_GetTime(&aft);
  durat = aft - bef;
  tcost = (double) durat / (COUNTS_PER_USECOND);
  if (durat > 325000000) //counter measured for 1 second
    cont = 0 ;
}

xil_printf("Exiting 1 Second Loop\n\r");

for (core_i = 0; core_i < NUMOF_CORES; core_i++) {
  printf("count_%i : %i \n",core_i,  core_run_count[core_i]);
}
cleanup_platform();
return 0;
```

Listing 6.2: Running ECM cores in parallel

6.4 Implementation Result and Comparisons

To the best of our knowledge our ECM core is the only design which can do any (practical) length of scalar multiplication without FPGA reconfiguration. All the previous results concentrate only for 200-bit long composite numbers. Our solution can run other lengths with a register configuration. Once the core's register is loaded for a particular modulus and for a particular number of digits, it is able to run for that setting. Other than this dynamic configuration we also provide a design-time parameter of multiplier width. Either a 16-bits or a 32-bits version of the core is possible to be instantiated as shown in Figure 6.5. All the ECM Phase 1 timing results of widely used prime field sizes, 192, 256, 384 and 512, are measured on two different FPGA implementations in Table 6.1. One major difference between 16-bits and 32-bits implementations is that the synthesis frequencies of the designs are 200 Mhz and 130 Mhz respectively. Because of this the 32-bit design did not yield an order of 4 enhancement. There is one more special case which appears in 32-bits version is that, the latency of our Montgomery modular multiplication is 12 cycles. When 16-bits multiplier is used in 192-bits field, $192/16 = 12$ cycles are needed for a whole number to be processed. At this time the result is just ready and the pipeline delay does not affect latency. In the case of 32-bits multiplier, the whole number is processed in $192/32 = 6$ cycles. In these cases the 32-bit multiplier needs to wait for the result to appear at the end of pipeline. As a result of this case the advantage of 32-bits processing is fully-benefited for field sizes of 384-bits. This result is also seen in Table 6.1. For 384 and 512-bits the 32-bit core performed exactly two times better than 16-bit version. For 192-bits field, the benefit is almost 10%.

Table 6.1: Summary of timing performance of flexible ECM core

Multiplier Width	Composite length	Point add double time	Phase 1 time	#Phase 1 per s.
16	$N = 192$	14.7 μs	19.254 ms	50
	$N = 256$	21.7 μs	29.147 ms	34
	$N = 384$	41.6 μs	56.136 ms	18
	$N = 512$	68.2 μs	92.768 ms	11
32	$N = 192$	13.5 μs	17.716 ms	55
	$N = 256$	16.6 μs	22.047 ms	44
	$N = 384$	22.3 μs	29.448 ms	33
	$N = 512$	33.5 μs	44.602 ms	22

The utilization results of our design together with previous works is given in Table

6.2. These are the three major implementations in the literature. Our design is similar to [40] in the elliptic field layer: The modular multiplier is serially used. By this way point addition and doubling time is about 10 times of multiplication time. 796-8200 vs 192-2925. This order is smaller in [19] 216-1212. This is achieved by using two Montgomery multipliers in elliptic layer. The study in [56] does not report utilization results, only timing performance is reported. We obtained the BRAM and DSP usage from the thesis report of the same people in [22]. Our results are better than the first two studies, comparable with the third study since we are using smaller amounts of DSP blocks, see Table 6.2. As discussed in the above paragraph, the performance of 32-bit version is not apparent for 192-bit field. It is two times faster for large field sizes.

Table 6.2: Comparison of ECM Phase 1 implementations for composite N of 192 bits with smoothness bound $B_1 = 960$ (Result in (*) are taken from [22])

	Simka et.al. [40]	KrisGaj [19]	Zimmermann [56]	this work 16-bits	this work 32-bits
	Timing Performance Summary				
Montgomery Mult.cycles Montgomery Mult.time	796 20.7 μs	216 4.1 μs	201 1.005 μs	192 0.960 μs	108 0.830 μs
Point add + doub.cycles Point add + doub.time	8200 213.2 μs	1212 23.0 μs	N/A N/A	2925 14.62 μs	1676 12.892 μs
Phase 1 cycles Phase 1 time	11,266,800 292.9 ms	1,713,576 31.7 ms	1,473,596 7.37 ms	4,021,875 20.10 ms	2,304,500 17.726 ms
	Utilization Summary per One ECM unit				
FPGA Model Name	Virtex-2000	Virtex-2000	Virtex4	Zynq7020	Zynq7020
CLB Slices	N/A	3102	N/A	564	692
Slice LUTs	1754	4933	N/A	1476	1798
Slice registers	506	3129	N/A	1354	1839
BRAMs	44	2	9 (*)	3	6
DSP	0	0	22 (*)	3	11

CHAPTER 7

CONCLUSION AND FUTURE WORK

FPGAs are reconfigurable devices which are located in between ASIC and general purpose processors. In this thesis we have implemented the elliptic curve method of integer factorization on reconfigurable hardware. For this implementation we started with a modular multiplier implementation which is based on hard multipliers, called DSP blocks, which are widely available in modern FPGAs. Later this multiplier core is controlled with a micro-instruction controller to obtain field operations of elliptic curve, point addition and point doubling. The processor had the capability to make two level nested function calls. With this property a special for-loop implementation is used to obtain scalar point multiplication result by using Montgomery Ladder algorithm. The ECM coprocessor is later connected to embedded processors which are widely available in modern FPGAs. By this way the ECM cores were easily accessible by an application. The obtained design can be scaled by instantiating it as many as FPGA resources permits.

The main novelty of the ECM coprocessor is its run-time and compile-time flexibility. As a run-time flexibility the ECM core can process any practical length of prime field, including the range of practical RSA sizes of 2048 and 4096. To obtain this flexibility, operands are saved in block memories including the scalar multiplier coefficient k. Dual-port property of the block memory is used to isolate the operating frequency of the arithmetic modules from the rest of the design. As a compile-time flexibility the data-path of the ECM core can be chosen as either 16-bit or 32-bit. All the control logic and memory instantiations are designed to embrace this change. As a result a generic-parameter of VHDL 16 or 32 was enough to obtain either versions. Although we are able to synthesize the 16-bit version of the core with a frequency of 200 Mhz, 32-bit version was synthesized with a frequency of 130 Mhz. If it were possible to have the same frequency we would gain a factor of 4 since the latency cost is proportional to n^2 where n is the number of digits in an operand. Nevertheless, a gain of two factors is obtained with this parametrization.

The final aim of this design is to be used as an accelerator for integer factorization. As the most mature software implementation we inspected the GMP-ECM library and made our FPGA design to work in coherency with the software implementation. To make this possible a big-integer library is ported into embedded processor on FPGA. Pre-computations and post-computations of the ECM can be done with this library.

Studying on an area related to semiconductor technology is prone to Moore's Law, *Number of transistors in an integrated circuit doubles about every two years*. As part of this law, we need to update software versions and FPGA models through the course of the thesis. However, there are much more advancements in high performance computing than we can catch.

FPGA based cloud services have been widely available in the recent years [28]. These platforms can be used as accelerator of integer factorization.

Another recent topic in the FPGA arena is the high-level synthesis technologies. This technology directly converts a high-level description of an algorithm into FPGA configuration. By this way design abstraction is raised one level higher. The solutions are becoming widely available and surprisingly productive [27]. HLS can be another direction for further studies.

Embedded processors were limited to vendor provided models when we started this thesis. RISC-V is an open standard instruction set architecture based on the principles of reduced instruction set computer(RISC). There are many companies providing hardware and operating system solutions to RISC-V. The embedded processor solution of this study can be switched to a RISC-V architecture as a further study.

REFERENCES

[1] AMBA Bus Specifications, `https://www.arm.com/architecture/system-architectures/amba/amba-specifications`, accessed: 2023-09-01.

[2] Kerckhoffs's principle , `http://www.crypto-it.net/eng/theory/kerckhoffs.html`, accessed: 2023-06-01.

[3] Key Transfer Devices, `https://www.cryptomuseum.com/crypto/fill.htm`, accessed: 2023-06-01.

[4] Multiple Precision Integers and Rationals, `https://en.wikipedia.org/wiki/MPIR_(mathematics_software)`, accessed: 2023-09-18.

[5] Niels Henrik Abel, `https://en.wikipedia.org/wiki/Niels_Henrik_Abel`, accessed: 2023-06-01.

[6] Online Elliptic Curve Drawing Tool , `https://andrea.corbellini.name/ecc/interactive/modk-mul.html`, accessed: 2023-09-01.

[7] rivyera high performance computing platform , `https://www.sciengines.com/technology-platform/sciengines-hardware/`, accessed: 2023-06-01.

[8] smartphone forecast, `https://www.bankmycell.com/blog/how-many-phones-are-in-the-world`, accessed: 2023-06-01.

[9] The GNU MP Bignum Library, `https://gmplib.org`, accessed: 2020-12-18.

[10] Type-Length-Value Protocols, `https://en.wikipedia.org/wiki/Type-length-value`, accessed: 2023-06-01.

[11] Évariste Galois, `https://en.wikipedia.org/wiki/Evariste_Galois`, accessed: 2023-06-01.

[12] A. ANSI, X9. 62: 2005: Public key cryptography for the financial services industry, The elliptic curve digital signature algorithm (ECDSA), 2005.

[13] S. Blake-Wilson and M. Qu, Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters, Certicom Research, Oct, 1999.

[14] G. R. Blakely, A computer algorithm for calculating the product AB modulo M, IEEE Transactions on Computers, 100(5), pp. 497–500, 1983.

[15] J. W. Bos and A. K. Lenstra, *Topics in Computational Number Theory Inspired by Peter L. Montgomery*, Cambridge University Press, 2017.

[16] R. P. Brent, Some integer factorization algorithms using elliptic curves, arXiv preprint arXiv:1004.3366, 2010.

[17] D. Brown, Standards for efficient cryptography, SEC 1: elliptic curve cryptography, Released Standard Version, 1, 2009.

[18] P. FIPS, 186-4: Federal information processing standards publication. Digital Signature Standard (DSS), Information Technology Laboratory, National Institute of Standards and Technology (NIST), Gaithersburg, MD, pp. 20899–8900, 2013.

[19] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi, Implementing the elliptic curve method of factoring in reconfigurable hardware, in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 119–133, Springer, 2006.

[20] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi, Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware, in L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pp. 119–133, Springer, 2006.

[21] Gajski and Kuhn, Guest editors' introduction: New VLSI tools, Computer, 16(12), pp. 11–14, 1983.

[22] T. Guneysu, *Cryptography and Cryptanalysis on Reconfigurable Devices*, Phd thesis, Ruhr-University Bochum, Bochum, February 2009, available at `https://informatik.rub.de/wp-content/uploads/2021/11/phd_gueneysu.pdf`.

[23] T. Güneysu and C. Paar, Ultra High Performance ECC over NIST Primes on Commercial FPGAs, in E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pp. 62–78, Springer, 2008.

[24] N. Koblitz, Elliptic curve cryptosystems, Mathematics of computation, 48(177), pp. 203–209, 1987.

[25] C. K. Koc, T. Acar, and B. S. Kaliski, Analyzing and comparing Montgomery multiplication algorithms, IEEE micro, 16(3), pp. 26–33, 1996.

[26] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, Breaking ciphers with COPACOBANA–a cost-optimized parallel code breaker, in *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*, pp. 101–118, Springer, 2006.

[27] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, Are We There Yet? A Study on the State of High-Level Synthesis, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(5), pp. 898–911, 2019.

[28] M. Leeser, S. Handagala, and M. Zink, FPGAs in the Cloud, Computing in Science Engineering, 23(6), pp. 72–76, 2021.

[29] H. W. Lenstra Jr, Factoring integers with elliptic curves, Annals of mathematics, pp. 649–673, 1987.

[30] P. H. W. Leong and I. K. H. Leung, A microcoded elliptic curve processor using FPGA technology, IEEE Trans. VLSI Syst., 10(5), pp. 550–559, 2002.

[31] C. McIvor, M. McLoone, and J. V. McCanny, Hardware Elliptic Curve Cryptographic Processor Over $\rm GF(p)$, IEEE Trans. on Circuits and Systems, 53-I(9), pp. 1946–1957, 2006.

[32] N. Mentens, S. B. Örs, B. Preneel, and J. Vandewalle, An FPGA Implementation of a Montgomery Multiplier Over GF(2^m), Computers and Artificial Intelligence, 23(5), pp. 487–499, 2004.

[33] V. S. Miller, Use of elliptic curves in cryptography, in *Conference on the theory and application of cryptographic techniques*, pp. 417–426, Springer, 1985.

[34] P. L. Montgomery, Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$, `https://cr.yp.to/bib/1992/montgomery-lucas.pdf`, accessed: 2023-06-01.

[35] P. L. Montgomery, Modular multiplication without trial division, Mathematics of computation, 44(170), pp. 519–521, 1985.

[36] P. L. Montgomery, Speeding the Pollard and elliptic curve methods of factorization, Mathematics of computation, 48(177), pp. 243–264, 1987.

[37] M. Morales-Sandoval and A. Diaz-Perez, Scalable GF (p) Montgomery multiplier based on a digit–digit computation approach, IET Computers & Digital Techniques, 10(3), pp. 102–109, 2016.

[38] H. E. Mustafa Hakan Solmaz and R. Aykac, A Scalable Platform for Cryptanalysis of Computationally Intensive Algorithms, in *International Information Security and Cryptology Conference, ISCTURKEY*, pp. 195–200, 2012.

[39] E. Oksuzoglu and E. Savas, Parametric, secure and compact implementation of RSA on FPGA, in *2008 International Conference on Reconfigurable Computing and FPGAs*, pp. 391–396, IEEE, 2008.

[40] J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovskỳ, V. Fischer, and C. Paar, Area–time efficient hardware architecture for factoring integers with the elliptic curve method, IEE Proceedings-Information Security, 152(1), pp. 67–78, 2005.

[41] J. M. Pollard, Theorems on factorization and primality testing, in *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 76, pp. 521–528, Cambridge University Press, 1974.

[42] C. Pöpper, O. Mischke, and T. Güneysu, MicroACP - A Fast and Secure Reconfigurable Asymmetric Crypto-Processor - -Overhead Evaluation of Side-Channel Countermeasures-, in D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, editors, *Reconfigurable Computing: Architectures, Tools, and Applications - 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings*, volume 8405 of *Lecture Notes in Computer Science*, pp. 240–247, Springer, 2014.

[43] B. Rashidi, A Survey on Hardware Implementations of Elliptic Curve Cryptosystems, CoRR, abs/1710.08336, 2017.

[44] M. H. Solmaz, Elliptic Curve Method(ECM) of Integer Factorization related material , `https://github.com/mhsolmaz/ecm_thesis`, 2023.

[45] A. F. Tenca, G. Todorov, and Ç. K. Koç, High-Radix Design of a Scalable Modular Multiplier, in Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pp. 185–201, Springer, 2001.

[46] M. Varchola, T. Güneysu, and O. Mischke, MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor, in P. M. Athanas, J. Becker, and R. Cumplido, editors, *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, Cancun, Mexico, November 30 - December 2, 2011*, pp. 204–210, IEEE Computer Society, 2011.

[47] J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi, and I. Verbauwhede, A compact FPGA-based architecture for elliptic curve cryptography over prime fields, in F. Charot, F. Hannig, J. Teich, and C. Wolinski, editors, *21st IEEE International Conference on Application-specific Systems Architectures and Processors, ASAP 2010, Rennes, France, 7-9 July 2010*, pp. 313–316, IEEE Computer Society, 2010.

[48] M. Welschenbach, *Cryptography in C and C++*, Apress, 2001.

[49] M. Welschenbach, Source code for 'Cryptography in C and C++', `https://github.com/apress/cryptography-in-c-cpp`, 2005.

[50] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*, Pearson Education India, 2015.

[51] I. Xilinx, Zynq-7000 Technical Reference Manual, UG585, 2014.

[52] T. Yalcin, Compact ECDSA engine for IoT applications, Electronics Letters, 52(15), pp. 1310–1312, 2016.

[53] P. Zimmermann and B. Dodson, 20 years of ECM, in *International Algorithmic Number Theory Symposium*, pp. 525–542, Springer, 2006.

[54] R. Zimmermann, T. Güneysu, and C. Paar, High-performance integer factoring with reconfigurable devices, in *2010 International Conference on Field Programmable Logic and Applications*, pp. 83–88, IEEE, 2010.

[55] R. Zimmermann, T. Güneysu, and C. Paar, High-Performance Integer Factoring with Reconfigurable Devices, in *International Conference on Field Programmable Logic and Applications, FPL 2010, August 31 2010 - September 2, 2010, Milano, Italy*, pp. 83–88, IEEE Computer Society, 2010.

[56] R. Zimmermann, T. Güneysu, and C. Paar, High-Performance Integer Factoring with Reconfigurable Devices, in *International Conference on Field Programmable Logic and Applications, FPL 2010, August 31 2010 - September 2, 2010, Milano, Italy*, pp. 83–88, IEEE Computer Society, 2010.

APPENDIX A

ELLIPTIC CURVE METHOD APPENDICES

A.1 192 Bit factorization with GMP-ECM with $\sigma = 7$

```
D:\devsoft\ecm\bin\x64\Debug>ecm  -v -v -inp b180 -param 0 -sigma 7
↪  960 50000
GMP-ECM 7.0.6-dev [configured with MPIR 3.0.0, --enable-openmp] [ECM]
Tuned for x86_64/corei7/params.h
Input number is
↪  1379757156111341689136297301978550989719758745674878923 (55
↪  digits)
Using MODMULN [mulredc:0, sqrredc:0]
ecm:after mpmod_init_MODMULN
modulus->orig    = E67C428000000000000000000000000000001B028FCB
modulus->bits    = 192
modulus->repr    = 3
modulus->R2      = 8191D00000000000000000038400000000000F2F166
modulus->R3      = 4CACA1FFFFFFFF968800000000000000000008FC3AFC
modulus->tmp1    = 1000000000000000000000000000000000000000000000000
modulus->multiple= 10008485EF00000000000000000000000000001E00F87B202
modulus->tmp2    = EB1977672D36F62F50E0E8B8150DEE8AFC6109FC17BE621D
modulus->Nprim   = dcdaf0
modulus->Nprim   = FC6109FC17BE621D
last A  = 2ED4A7BF6A86E90E97111020F8D41EA9F8B6BD53D2CB9
Using B1=960, B2=50000, polynomial x^1, sigma=0:7
dF=32, k=6, d=240, d2=7, i0=-2
modulus       : x0=E67C428000000000000000000000000000001B028FCB
R^2           : x0=8191D00000000000000000038400000000000F2F166
R^2           : x0=8191D00000000000000000038400000000000F2F166
R^3           : x0=4CACA1FFFFFFFF968800000000000000000008FC3AFC
Nprime        : x0=14473968
*Nprime       : x0=FC6109FC17BE621D
Bits          : x0=192
BefA->=2ED4A7BF6A86E90E97111020F8D41EA9F8B6BD53D2CB9
BefA. =2ED4A7BF6A86E90E97111020F8D41EA9F8B6BD53D2CB9
A=76D793A1349E27D755A50D6191D1907FEE6A3A681327A
```

```
Bef_x0. =1D5F089C5C8C509B3DF290CB023D337FA00300F6235AB
starting point: x0=2B018A8AD278E8DCEBD930835BCD3F70B3201832C839
Bef z=1.R mod n =0
Aft z=1.R mod n =61f6537fffffffffffffffffffffffffffe200b7addc9
Bef A  =2ed4a7bf6a86e90e97111020f8d41ea9f8b6bd53d2cb9
A+2  =c450c3f6a86e90e97111020f8d41ea9f87abd1305880
b=(A+2)*/4  =311430fdaa1ba43a5c444083e3507aa7e1eaf44c1620
ecm_stage1:Montgomery Curve :
 ↪  1db5e4e84d2789f5d56943586474641ffb9a8e9a04c9f.Y^2= X^3 +
 ↪  76d793a1349e27d755a50d6191d1907fee6a3a681327a.X^2 + X
default x0,z0  x=1d5f089c5c8c509b3df290cb023d337fa00300f6235ab
 ↪  z=61f6537fffffffffffffffffffffffffffe200b7addc9 go=1
normal  x0,z0  x=2b018a8ad278e8dcebd930835bcd3f70b3201832c839 z=1
 ↪  go=1
montgo  x0,z0  x=1d5f089c5c8c509b3df290cb023d337fa00300f6235ab
 ↪  z=61f6537fffffffffffffffffffffffffffe200b7addc9 go=1
ecm_stage1:x2   x=1461619737E78F7A4AE1D80FD217DB01429E63A137B6D
 ↪  z=AD3017728F8FDDCD78CAAC735B346E8D7613AF1501EF1 p=2 big_r=2
ecm_stage1:x2   x=89203D122C4816EA002AFAEC850B859159AB56DCEB643
 ↪  z=3403A28C8FBC2D72BB76250EBCAA109A842451BD3CB2B p=2 big_r=4
ecm_stage1:x2   x=A948280C40EB9EF40705F872406CC7CAE0689F8066BE9
 ↪  z=1F3455EF9A48D8702DA726AC374129C0E44A27E3B186C p=2 big_r=8
ecm_stage1:x2   x=E49A5A16668F1A0445C33C3203FB61FD0ABE9315AE15B
 ↪  z=80E8AB8DB649F599B6638455D77A39B4B3D09C8B2515E p=2 big_r=10
ecm_stage1:x2   x=A4271ED85297E13E4A504E85444BBE69994917B118203
 ↪  z=92D32E506841D6898958D4B19894823020010A075755A p=2 big_r=20
ecm_stage1:x2   x=55DD70EC15660C048AA27C2CB79B9228DABF2C76705A1
 ↪  z=AFB3CB00B2129D38D4EBA9820CD985A0255BFF0B6E7C2 p=2 big_r=40
ecm_stage1:x2   x=5C0033BE99BA118F12A312749D9CDFFF04F12D3E43EBF
 ↪  z=BE271EE98627286DD2D208A26016DC775D4C26FC6141A p=2 big_r=80
ecm_stage1:x2   x=7052F1A0EEE92DE769171DEB151A93268C4F6269271AA
 ↪  z=97B4A8A6DBDF24BC0324D1AA8CF1D286F0404DA10FC0D p=2 big_r=100
ecm_stage1:x2   x=12152BF8E7ED45E26B58327650070729416E5B0928F1E
 ↪  z=5B390F08D6CCE287E463E09389917CE8BD7F676B4438F p=2 big_r=200
ecm_stage1:x3   x=AC4B7206F1427F6E94F55E6ECF5E34042B1960316732B
 ↪  z=C12405AA83444952758F74223585964F285AD90168098 p=3 big_r=600
ecm_stage1:x3   x=AA929D5D0363F716703ED794FD68F0B6C92C42D048052
 ↪  z=54962066AF840EA9D4869658C976446DAA1AF193AA48A p=3 big_r=1200
ecm_stage1:x3   x=CD860D9CC798EB88C7DC56265FE562E12B801D097FB91
 ↪  z=49791A1F06638D87DF2F1DEB6219886F2386125BB24E1 p=3 big_r=3600
ecm_stage1:x3   x=DF972BDBB15D5137A39FD65C994CE2B15438274AA49B6
 ↪  z=119685A3CD4349FE700F02A5FCE2BEDBF5E359AAA06C5 p=3 big_r=A200
ecm_stage1:x3   x=CBA42D9154A711B31BF4E9AB41E73132F2470FAA71A8B
 ↪  z=A914E569F2D70BE3C63A3C67DDE8D308860E27AC45B p=3 big_r=1E600
ecm_stage1:x3   x=1CC19B52D0F969B90DFC5B1964451A803DE442F33C44C
 ↪  z=150177EAFB4F0AFD6377CEC0D44C82A7A06E16392BAAB p=3 big_r=5B200
ecm_stage1:prac x=7C3FAAA13BDA1FEEF9A20DEBB787C5FA0556B6AF30F9A
 ↪  z=70119A93380FC8E3459D4E40D1F844A9690A0C6939F23 p=5 r=5
ecm_stage1:prac x=7F68582AF536C2FAD59D8FF4DD5A14226E0A680E56916
 ↪  z=1D1F4BE702F2E6CD79AB9DC498DE973303DA413E6B5C2 p=5 r=25
```

```
ecm_stage1:prac x=1A3E8257D5300DA2E8BDF645AE4A6DA3A2E62505CA464
 ↪  z=C20EAB4C15527A8D904EB2231D14ABC625BA8361E4163 p=5 r=125
ecm_stage1:prac x=D090382B61560BB730370779C20D5B518EFB72C343BFF
 ↪  z=3981D00D376FF5D33DE795280F26FA31F6BCC383A7E6C p=5 r=625
ecm_stage1:prac x=60AEEA6A63193F1AEE82D0B6E619F08A9731758F2A8E7
 ↪  z=B449695AC130448667C1A0A620AB265C6AA3C4ECF06D2 p=7 r=7
ecm_stage1:prac x=336546C38D1FDB11FB09CBA48F3367A333090EDCC8B7F
 ↪  z=719FA51B1DCF7F1811244E38598534ED6C8929CF80539 p=7 r=49
ecm_stage1:prac x=D572629471E737A5BAA2175EE4EEFC4C5AF5AAB170204
 ↪  z=DF03E0D116F84C44AB6E127210E9FA929AE93D8595CC8 p=7 r=343
...
...
ecm_stage1:prac x=DC7921FB8ADAAA90C40814CAEAD7DE4E4169BD49DDE2B
 ↪  z=41109E62A16F19B0E27D3A04CAC575D786CDA7C2EE169 p=877 r=877
ecm_stage1:prac x=E4FB55B32079C5C659762AB77FA3F85EC2D394FD9ED86
 ↪  z=B819976F1C2A9D9488551AB866DDA035ED0B542D01529 p=881 r=881
ecm_stage1:prac x=323CF80AFF9EA908D6101F1B8768DABFA0827AB533D5E
 ↪  z=10BB2744493354F952D9705C18A78D3BD4B79B2C900B3 p=883 r=883
ecm_stage1:prac x=1B101600B5F63A370AF3A2C6B13CD77F4482C67EDDFB
 ↪  z=22D7E3DBC200725C6B618B8F2E576A42B586B272C5512 p=887 r=887
ecm_stage1:prac x=21AD93F5C207DF19C2A0B5ADB6FA5AD76B2DA0EC19842
 ↪  z=13FE3B6E6853C14D30711E3FEFA0802F2E94466E2A1F2 p=907 r=907
ecm_stage1:prac x=824CCB69E4AD4702A59F0C57AD8B376D95FF8AA8B416A
 ↪  z=22563A12AE401D76C706941CC2318E9DE7CD3170B3548 p=911 r=911
ecm_stage1:prac x=D6007E273CB68C6FC953E56ABB6D0CC75796A95DE9988
 ↪  z=32963F51969560098FBD610FF2E4AB41442150E2B670B p=919 r=919
ecm_stage1:prac x=C6DC8D2275966EFAFE37607095088948AAE4E78D93B6D
 ↪  z=96BF85CD0049ADA1B20B24AF28D06913248245AA01993 p=929 r=929
ecm_stage1:prac x=73394CFCEF610796C96067210BDA67AEA4D98AA52292A
 ↪  z=55FF7E4E252D49CF5DA0747453568E58B51687835B757 p=937 r=937
ecm_stage1:prac x=52DA16E3772A59D02206EDAE4AECFD654E1087D534951
 ↪  z=7FF28390CFED6DD0F091BEDDCFA91C8F97FA4B8C4C07E p=941 r=941
ecm_stage1:prac x=A72102CD8D1D0A680995D3B65EEBFA6B981B20B2E2AB4
 ↪  z=C4168D1843F77167E5C3EB32ED39F93E096FEA85901CA p=947 r=947
ecm_stage1:prac x=C1B201EC35AA85BC116FA073FECF9E7884E4E91DA182C
 ↪  z=BD32A1176121C768DFAD6BD8A9E5CB80337B1119AD2D5 p=953 r=953
ecm:ecm_stage1:ecm_mul result:
 ↪  x=afbd13db4798059de86be952465d19cc8eef928e9793b
 ↪  z=bed0782c1b39b8f58d218795a5ef1af9ae9453a3e8e80
Prac result of ecm stage1   =========================>>>>>>>>x/z in
 ↪  norm form: c35a0ca1152a657b5e84b6b6ac8b95034d6d5db2dc28e
ecm_stage1:big_r
 ↪  281091188610b1a7d57f82e293e9aefe96cde6a215fd76eb40f25393070f92
64a49b76730d8c265aa850558dfeda6e831629c8cfe5c610562e03311728173b32
c61c69e9573cf2438fe26bc7a3a09d85751ba39fd5085a3713e7fee9db6c2a9e0d
e0eefeb7523e3b3dd9cd8edb073c475c476470e17d9f00345fbc988a3306b548ae
1b5e64e768cfb47408cb73bacd1c68ae0d56ca6430f631a3dfe3870d5bc2125fca
2f5a139c079c32f600
x=116943630372914346472713101932410020702471508353704001 4
After switch to Weierstrass form,
 ↪  P=(cec8aef85e008ffbe656481589ccd8928437a312d63b9,
 ↪  71eb541db9f480c69958063889aa7821b7bd62e3fa43)
```

```
on curve Y^2 = X^3 + 2a00285ec8c54990488458c5824a1fd23a143a062fb09 *
↪  X + b
********** Factor found in step 2: 30210181
Found probable prime factor of 8 digits: 30210181
Probable prime cofactor
↪  45671926166590716193865151022383844364247891983 has 47 digits
Peak memory usage: 5MB

D:\devsoft\ecm\bin\x64\Debug>
```

Listing A.1: GMP-ECM reduced output for $\sigma = 7, B1 = 960, B2 = 50000$

A.2 192 Bit factorization with GMP-ECM with $\sigma = 10$

```
D:\devsoft\ecm\bin\x64\Debug>ecm  -v -v -inp b180 -param 0 -sigma 10
↪  960 50000
GMP-ECM 7.0.6-dev [configured with MPIR 3.0.0, --enable-openmp] [ECM]
Tuned for x86_64/corei7/params.h
Input number is
↪  1379757156111341689136297301978550989719758745674878923 (55
↪  digits)
Using MODMULN [mulredc:0, sqrredc:0]
ecm:after mpmod_init_MODMULN
modulus->orig    = E67C428000000000000000000000000000000001B028FCB
modulus->bits    = 192
modulus->repr    = 3
modulus->R2      = 8191D000000000000000000384000000000000F2F166
modulus->R3      = 4CACA1FFFFFFFF968800000000000000000000008FC3AFC
modulus->tmp1    = 1000000000000000000000000000000000000000000000000
modulus->multiple= 10008485EF00000000000000000000000000001E00F87B202
modulus->tmp2    = EB1977672D36F62F50E0E8B8150DEE8AFC6109FC17BE621D
modulus->Nprim   = cce0c0
modulus->Nprim   = FC6109FC17BE621D
last A  = 3262CD21F20A86703190B3BFA79E5CC9193FF385D9825
Using B1=960, B2=50000, polynomial x^1, sigma=0:10
dF=32, k=6, d=240, d2=7, i0=-2
modulus       : x0=E67C428000000000000000000000000000000001B028FCB
R^2           : x0=8191D000000000000000000384000000000000F2F166
R^2           : x0=8191D000000000000000000384000000000000F2F166
R^3           : x0=4CACA1FFFFFFFF968800000000000000000000008FC3AFC
Nprime        : x0=13426880
*Nprime       : x0=FC6109FC17BE621D
Bits          : x0=192
BefA->=3262CD21F20A86703190B3BFA79E5CC9193FF385D9825
```

```
BefA. =3262CD21F20A86703190B3BFA79E5CC9193FF385D9825
A=537F39F9BA5AD840322993FC90F6A1C2447367E289B27
Bef_x0. =457052FFFFFFFFFFFFFFFFFFFFFFFFFFFE6E1B82329BA
starting point: x0=72CAE31EC00000000000000000000000000000D73C6AB
Bef z=1.R mod n =0
Aft z=1.R mod n =61f6537fffffffffffffffffffffffffffe200b7addc9
Bef A  =3262cd21f20a86703190b3bfa79e5cc9193ff385d9825
A+2  =fd331a1f20a86703190b3bfa79e5cc91903f3450c3ec
b=(A+2)*/4  =3f4cc687c82a19c0c642cefe9e797324640fcd1430fb
ecm_stage1:Montgomery Curve :
↪  4e7edf1e6e96b6100c8a64ff243da870911cda64acabd.Y^2= X^3 +
↪  537f39f9ba5ad840322993fc90f6a1c2447367e289b27.X^2 + X
default x0,z0  x=457052fffffffffffffffffffffffffffe6e1b82329ba
↪  z=61f6537fffffffffffffffffffffffffffe200b7addc9 go=1
normal  x0,z0  x=72cae31ec00000000000000000000000000000d73c6ab z=1
↪  go=1
montgo  x0,z0  x=457052fffffffffffffffffffffffffffe6e1b82329ba
↪  z=61f6537fffffffffffffffffffffffffffe200b7addc9 go=1
ecm_stage1:x2   x=A5BFD1FFFFFFFFFFFFFFFFFFFFFFFFF16B92D1F0D08DE
↪  z=D976567FFFFFFFFFFFFFFFFFFFFFFFFC5C27AE55F0E23 p=2 big_r=2
ecm_stage1:x2   x=5980751D3AF60869F4328FD9D35B585ECBFB3657CDFF6
↪  z=6532A27B3EA28AC3D347874B5D775E0645F6D3195DD76 p=2 big_r=4
ecm_stage1:x2   x=5E7ADDCB22A7B6CE0C13DCC26F76272B38B0770BFB8AD
↪  z=B52D84641FC6FDA2BC5B1355786C7ACB1424A6A4E3FF1 p=2 big_r=8
ecm_stage1:x2   x=4440A83BCCE663EF95642D46E8311F1C0C809EB3304E1
↪  z=60B2E9839BE010941C9DCACFDC4E70DCA58DBC707EBD1 p=2 big_r=10
ecm_stage1:x2   x=BE362FF2300BA8F7652B7C85E4F440FE4BBD3A46AECFE
↪  z=DD2BD4956B6EA4681799071829588C543BFD3FB3F1D53 p=2 big_r=20
ecm_stage1:x2   x=1C33AE8F2B3D9D10E9D279F9B5D26FE5DBE8B909383A9
↪  z=1EED1B190C533DD0BE121705C32B3BD93919C5ABAF173 p=2 big_r=40
ecm_stage1:x2   x=AF5F03E2DCBD962D8E5710F9424346E94BED8487C6B17
↪  z=D233B287EBFB32BD8EF41AA827E001A141006FD0D4366 p=2 big_r=80
ecm_stage1:x2   x=2E546874F151ACBBD82E8CD6BE8B78AB09DF01EAAB556
↪  z=82DA9905EA778904C1CD1AD6A88312B340387AA823B38 p=2 big_r=100
ecm_stage1:x2   x=23DD0B3EFCE5AF676A76F45E3179321798B54DDB981DB
↪  z=B8178735E55A05F63CFD630A65B2BDBCA0482D6C71BC2 p=2 big_r=200
ecm_stage1:x3   x=E2D304769505A47A47A6454E352114D4B0D0EDD3A5B20
↪  z=C5CFC365BC0E345BE8657D0AFE0C9818EE4ACBA4B920D p=3 big_r=600
ecm_stage1:x3   x=D90AECDFEA786C87BB764D9AAC93896E22E2B6B422A13
↪  z=363EF78B0224B55905030CBBC8E72F411B1B0F6169CC7 p=3 big_r=1200
ecm_stage1:x3   x=E12056429B89C8EAAD9344EE4F440E624918B2CCE35B0
↪  z=54D18DCB52195CC2B99A3CA494F2EA02AE0E8992FAA87 p=3 big_r=3600
ecm_stage1:x3   x=7CA5D1011B554FBD7F5F690E53670C40DDFFE759E684
↪  z=46F59AAC5E0473D12A777DF2098522170B28CF16C7108 p=3 big_r=A200
ecm_stage1:x3   x=69A5F8F5091BC2A624EE6FD4DA306AC70253217282DBD
↪  z=58221F556C6570F465ADE38223ED180FB7CC399464850 p=3 big_r=1E600
ecm_stage1:x3   x=A6C882C26CC4866D12EB85B9D821EAB5E52F3492C5598
↪  z=38F6374288691B958F8A232D21679BE4E98F7CD0E6F93 p=3 big_r=5B200
ecm_stage1:prac x=7309AD5BDF71F468CA2B2EA7D82E69E64B83786CFE483
↪  z=77A20DC4A70D4A3F9928A903BF81AAABD80CF3BDA3BC2 p=5 r=5
```

```
ecm_stage1:prac x=B06B8860FCFA465F963A9DAAEA7701D553B2E65BE0B18
↪  z=AFF7E244514D97F13F06FF2A939E7A2BF2CC975FA14E2 p=5 r=25
ecm_stage1:prac x=C6C219CFB56CBE07217DED3F453AF1F8DF07399DBF750
↪  z=96723C29EE896FE30DC38A4445B67CB8B80FAF7D7D660 p=5 r=125
ecm_stage1:prac x=437E9BAE034406E158C640773C02588360D3C3AA27308
↪  z=E14F8A054405DA7D7DF10B0621087FF3AFFEFC1B73759 p=5 r=625
ecm_stage1:prac x=601C8172675D737EAC351F527A50D9D6C06DABA192E4A
↪  z=8FC322B6B61CE7DA1D6611420F4F22F8B5E583786DA66 p=7 r=7
ecm_stage1:prac x=698297CC247CA77FD796B671B06F93B23BC108F093400
↪  z=C6B25A91BDC58CEDD8B3FE613AB398DC21EEE0C99DA7B p=7 r=49
ecm_stage1:prac x=96499030E19D555807541FED9C7B43608F74DA0AD6885
↪  z=257223D5BAFA480734FAC341EF7FF37BBA3F46161BD06 p=7 r=343
...
...
ecm_stage1:prac x=AFA653617D1F497999C82F9027FFCE93B855D32E7B581
↪  z=4B7793800600E4CDB06B60EC95A66A4667CE28815479 p=877 r=877
ecm_stage1:prac x=30ACACFFFC653FA97940C9A84EC97E472E96AC238C710
↪  z=C987029B13EA9E92A1CCCAD0506EF915EF5E02B990E2F p=881 r=881
ecm_stage1:prac x=755F487DBB933797B680A75A65496293D0E1CF797F5F6
↪  z=A0930E560FB23CFF0B0F6006B2BE09234648BB8A1C7AB p=883 r=883
ecm_stage1:prac x=D9A1BCA7CD6EE18AF2C3FCE66657E612BFE909D31116D
↪  z=D0058DFB575EB696D359BBB6CEC63F264DFB1B94BF5D6 p=887 r=887
ecm_stage1:prac x=D78F26E792E4F2D8BED73D971D726277AE0FD7998D01F
↪  z=2728C517D57A2CB3F2263894FD665B7BF77F50DD7A3AD p=907 r=907
ecm_stage1:prac x=731C3F493E367A0352C8AF5B8180F54D151CA11BA6CEF
↪  z=6032DC2363A2EFFDF27760D038021478AE4139F2A4C8F p=911 r=911
ecm_stage1:prac x=A67EE90B988A0119B866DB9D92E7C496E4B56EEE9AE46
↪  z=7AF587BC6E88AE812DBA61ABA842DA2799970BFFD8C0F p=919 r=919
ecm_stage1:prac x=47959946EF946E2ADD5D2C1D95A3C699F38F240E36969
↪  z=525728DD80184134654BFBB189B4EE7B0E23DB2ED2C45 p=929 r=929
ecm_stage1:prac x=86E0D20A1F48DFD7501E94A5AAFC99DFD811A257F799B
↪  z=A7F288D1084AE121414E79B39FE9812A79D22D43F131C p=937 r=937
ecm_stage1:prac x=6FCF5CD72A0665DD15E4677BE8273378AEC28A384ACED
↪  z=D4E0113E8F088D5B72950ADA7FE2E6400F80D56FF63A0 p=941 r=941
ecm_stage1:prac x=B0187D7A9154F1B43887036F44429793403F882C4F2DA
↪  z=CBDF3162C1C66ECCCA084BCBBFEC0C88C5900A44928D4 p=947 r=947
ecm_stage1:prac x=84F4247064AC524A678CD34BA8D869615CDC301DDD937
↪  z=D2E5F642F83F2AFB85645750A5FBD3CFE6E0E3FF4C5A6 p=953 r=953
ecm:ecm_stage1:ecm_mul result:
↪  x=8b9744e5e22c899de630b52971eae817d7a5f400bae72
↪  z=650d4c930662f2ae42aecd3e68c9ec9e3b63362a5f636
Prac result of ecm stage1   =========================>>>>>>>>x/z in
↪  norm form: 633294cb7d5c9b91960839af924a29621c3925dc8129e
ecm_stage1:big_r
↪  281091188610b1a7d57f82e293e9aefe96cde6a215fd76eb40f25393070f92
64a49b76730d8c265aa850558dfeda6e831629c8cfe5c610562e03311728173b32
c61c69e9573cf2438fe26bc7a3a09d85751ba39fd5085a3713e7fee9db6c2a9e0d
e0eefeb7523e3b3dd9cd8edb073c475c476470e17d9f00345fbc988a3306b548ae
1b5e64e768cfb47408cb73bacd1c68ae0d56ca6430f631a3dfe3870d5bc2125fca
2f5a139c079c32f600
```

```
********** Factor found in step 1: 30210181
Found probable prime factor of 8 digits: 30210181
Probable prime cofactor
 ↪  45671926166590716193865151022383844364247891983 has 47 digits
Peak memory usage: 5MB

D:\devsoft\ecm\bin\x64\Debug>
```

Listing A.2: GMP-ECM reduced output for $\sigma = 10, B1 = 960, B2 = 50000$

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Solmaz, Mustafa Hakan
Nationality: Turkish

EDUCATION

Degree	Institution	Year of Graduation
M.S. Electronics Eng.	Graduate School of Engineering, Bilkent	1998
B.S. Electronics Eng.	Electrical Faculty of Engineering, Bilkent	1996
High School	Konya Anadolu Lisesi	1992

PUBLICATIONS

International Conference Publications

Mustafa Hakan Solmaz, H. Erdoğan, and R. Aykaç, A scalable platform for cryptanalysis of computationally intensive algorithms, in *International Information Security and Cryptology Conference, ISCTURKEY,* pp. 195–200, 2012.

International Journal Publications

Mustafa Hakan Solmaz and Oğuz Yayla, Flexible FPGA Design for Elliptic Curve Method, preprint

TEZ İZİN FORMU / THESIS PERMISSION FORM

ENSTİTÜ / INSTITUTE

Fen Bilimleri Enstitüsü / Graduate School of Natural and Applied Sciences ☐

Sosyal Bilimler Enstitüsü / Graduate School of Social Sciences ☐

Uygulamalı Matematik Enstitüsü / Graduate School of Applied Mathematics ☒

Enformatik Enstitüsü / Graduate School of Informatics ☐

Deniz Bilimleri Enstitüsü / Graduate School of Marine Sciences ☐

YAZARIN / AUTHOR

Soyadı / Surname : **SOLMAZ**
Adı / Name : **Mustafa Hakan**
Bölümü / Department : **Kriptografi**

TEZİN ADI / TITLE OF THE THESIS (**İngilizce** / English) : **Flexible hardware design for elliptic curve method of integer factorization**

TEZİN TÜRÜ / DEGREE: **Yüksek Lisans** / Master ☐ **Doktora** / PhD ☒

1. **Tezin tamamı dünya çapında erişime açılacaktır. /** Release the entire work immediately for access worldwide. ☒
2. **Tez iki yıl süreyle erişime kapalı olacaktır.** / Secure the entire work for patent and/or proprietary purposes for a period of **two year**. * ☐
3. **Tez altı ay süreyle erişime kapalı olacaktır.** / Secure the entire work for period of **six months**. * ☐

***** *Enstitü Yönetim Kurulu Kararının basılı kopyası tezle birlikte kütüphaneye teslim edilecektir.*
A copy of the Decision of the Institute Administrative Committee will be delivered to the library together with the printed thesis.

Yazarın imzası / Signature .... **Tarih** / Date ....