

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

EFFICIENT MODULAR MULTIPLICATION TECHNIQUES
FOR LARGE INTEGERS ON FPGAS

ERDEM ÖZCAN
A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
DEPARTMENT OF ELECTRONICS ENGINEERING

GEBZE
2019

T.R.
GEBZE TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

**EFFICIENT MODULAR
MULTIPLICATION TECHNIQUES FOR
LARGE INTEGERS ON FPGAS**

ERDEM ÖZCAN
**A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY**
DEPARTMENT OF ELECTRONICS ENGINEERING

THESIS SUPERVISOR
ASSOC. PROF. DR. SERDAR SÜER ERDEM

GEBZE
2019

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

FPGA ÜZERİNDE GENİŞ TAM SAYILAR
İÇİN VERİMLİ ÇARPMA TEKNİKLERİ

ERDEM ÖZCAN
DOKTORA TEZİ
ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMANI
DOÇ. DR. SERDAR SÜER ERDEM

GEBZE
2019

GTÜ Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 06/03/2019 tarih ve 2019/13 sayılı kararıyla oluşturulan jüri tarafından 13/03/2019 tarihinde tez savunma sınavı yapılan Erdem Özcan'ın tez çalışması Elektronik Mühendisliği Anabilim Dalında DOKTORA tezi olarak kabul edilmiştir.

JÜRİ

ÜYE

(TEZ DANIŞMANI)

: Doç. Dr. Serdar Süer Erdem



ÜYE

: Dr. Öğretim Üyesi Önder Şuvak



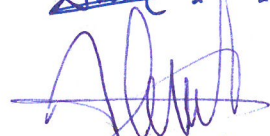
ÜYE

: Dr. Öğretim Üyesi Anıl Çelebi



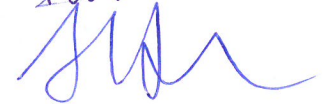
ÜYE

: Prof. Dr. Ali Tangel



ÜYE

: Dr. Öğretim Üyesi Köksal Hocaoglu



ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun

...../...../..... tarih ve/..... sayılı kararı.

SUMMARY

Public-key cryptography (PKC) introduced by Diffie and Hellman [1], is widely used in today's communication systems. RSA and elliptic curve based public-key cryptosystems heavily depend on modular multiplication. Modular multiplication operation with large integers is the main computation part in these systems. Conventional modular multiplication is an expensive operation because it requires division. In order to overcome this difficulty, effective modular multiplication algorithms are proposed in the literature. Most common ones are Montgomery and Barrett algorithms [5], [27] which replace division with multiplication and shift operations.

Therefore, in this thesis, we first designed a fast digit based Montgomery multiplier using DSP resources of FPGAs. The proposed hardware is implemented Verilog HDL and it takes 0.43 us to compute one 528 bit Montgomery modular multiplication and 1.09 us to compute one 1056 bit Montgomery modular multiplication in Virtex-7 xc7vx330tffg1157-3. We then, proposed full-word Barrett multiplier using DSP resources. It takes 0.49 us to compute one 528 bit Barrett modular multiplication and 1.88 us to compute one 1056 bit Barrett modular multiplication in Virtex-7 device xc7vx330tffg1157-3.

We, then designed ECPM hardware based on the proposed Montgomery multiplier. It takes 4.06 ms to compute one 528 bit ECPM in any prime field and 2.79 ms to compute NIST p-521 ECPM in Virtex-7 xc7vx330tffg1157-3.

Finally, we implemented Montgomery and Barrett algorithms using Vivado HLS tool. HLS implementation of Montgomery and Barrett algorithm takes 1.34 us and 2.57 us to compute one 528 bit modular multiplication in Virtex-7 device xc7vx330tffg1157-3 respectively.

Keywords: Montgomery, Barrett, Elliptic Curve, DSP, FPGA, HLS.

ÖZET

Diffie ve Hellman [1] tarafından ortaya konulan Açık Anahtarlı Şifreleme günümüz haberleşme sistemlerinde yaygınca kullanılmaktadır. RSA ve eliptik eğri tabanlı kriptosistemleri modüler çarpma işlemine dayanmaktadır. Geniş sayılarla modüler çarpma işlemi bu sistemlerdeki ana hesap kısmını oluşturmaktadır. Geleneksel modüler çarpma, bölme işlemi içerdiği için pahalı bir operasyondur. Bölme işleminden kurtulmak için, literatürde Montgomery ve Barrett algoritmaları [5], [27] öne sürülmüştür. Bu algoritmalar bölme işlemi yerine, çarpma ve kaydırma işlemini kullanmaktadır.

Bu tezde, ilk olarak, hane tabanlı ve FPGA'nın DSP kaynaklarını kullanan bir Montgomery çarpıcısı tasarlanmıştır. Tasarlanan donanım, Verilog diliyle yazılmıştır ve bir 528 bit Montgomery çarpma işlemini Virtex-7 xc7vx330tffg1157-3 çipinde 0.43us'de bir 1056 bit Montgomery çarpma işlemini 1.09 us'de hesaplamaktadır. Sonra, DSP kaynaklarını kullanan bir tam kelime Barrett çarpıcısı öne sürülmüştür. Öne sürülen donanım, bir 528 bit Barrett çarpma işlemini Virtex-7 xc7vx330tffg1157-3 çipinde 0.49 us'de bir 1056 bit Barrett çarpma işlemini 1.88 us'de hesaplamaktadır.

Daha sonra, öne sürülen Montgomery çarpıcısına dayanan eliptik eğri nokta çarpıcısı (ECPM) tasarlanmıştır. Tasarlanan ECPM donanımı Virtex-7 xc7vx330tffg1157-3 çipinde, bir adet 528 bit ECPM işlemini herhangi bir asal eğri için 4.06 ms'de, NIST p-521 eğrisi için 2.79 ms'de hesaplamaktadır.

Son olarak, Montgomery ve Barrett algoritmaları Vivado HLS aracıyla gerçekleştirilmiştir. Montgomery ve Barrett algoritmalarının HLS gerçeklemeleri bir 528 bit modüler çarpma işlemini, Virtex-7 xc7vx330tffg1157-3 çipinde, sırasıyla 1.34 us ve 2.57 us'de hesaplamaktadır.

Anahtar Kelimeler: Montgomery, Barrett, Eliptik Eğri, DSP, FPGA, HLS.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor Dr. Serdar Süer Erdem for his invaluable guidance and support throughout my study. I appreciate very much for his suggestions, detailed reviews, and invaluable advices.

I am sincerely grateful to my thesis committee members Dr. Önder Şuvak and Dr. Anıl Çelebi for their invaluable feedback.

I would like to thank Prof. Dr. Ali Tangel and Dr. Köksal Hocaoglu for joining my thesis defense.

I would also like to express my deepest gratitude for my family and my beloved wife Emel Özcan who always believed in me, and always tried their best to make things easier for me.

TABLE of CONTENTS

	<u>Sayfa</u>
SUMMARY	v
ÖZET	vi
ACKNOWLEDMENTS	vii
TABLE of CONTENTS	viii
LIST of ABBREVIATIONS and ACRONYMS	x
LIST of FIGURES	xi
LIST of TABLES	xii
1. INTRODUCTION	1
1.1. Motivation	1
1.2. Thesis Contributions	1
1.3. Thesis Organization	2
2. DIGIT BASED MONTGOMERY MULTIPLIER	3
2.1. Introduction	3
2.2. Montgomery Technique	5
2.2.1. Montgomery Domain	6
2.2.2. Montgomery Reduction Method	6
2.2.3. Montgomery Multiplication	8
2.2.4. Interleaved Montgomery Multiplication	9
2.2.5. Multiprecision Implementation	10
2.3. Proposed Montgomery Multiplier	12
2.3.1. DSP48E based 48x48 bit Multiplier Unit	12
2.3.2. General Structure	14
2.3.3. DSP48E based Carry Select Adder/Subtractor	17
2.3.4. Latency of Montgomery Multiplier	18
2.4. Performance Comparisons	20
2.5. Conclusion	21
3. FULL-WORD BARRETT MULTIPLIER	23
3.1. Introduction	23

3.2. Barrett Reduction	24
3.3. Proposed Barrett Multiplier	27
3.3.1. DSP48E based 48x48 bit Multiplier Unit	29
3.3.2. General Structure	29
3.3.3. DSP48E based Carry Select Adder/Subtractor	29
3.3.4. Latency of Barrett Multiplier	30
3.4. Performance Comparisons	31
3.5. Conclusion	34
4. ELLIPTIC CURVE POINT MULTIPLICATION	35
4.1. Introduction	35
4.2. Background Information	37
4.2.1. Elliptic Curves	37
4.2.2. Montgomery Multiplication	39
4.3. Proposed Elliptic Curve Point Multiplication Hardware	39
4.3.1. Point Doubling and Point Addition Scheduling	39
4.3.2. Point Multiplication	42
4.4. Implementation Results	44
4.4.1. Hardware Implementation	44
4.4.2. Performance Comparisons	46
4.5. Conclusions	47
5. HLS BASED IMPLEMENTATIONS	49
5.1. Introduction	49
5.2. Proposed HLS Implementations	50
5.2.1. Overall Structure	50
5.2.2. Performance Results	51
5.3. Conclusion	55
6. CONCLUSION	56
REFERENCES	57
BIOGRAPHY	61
APPENDICES	62

LIST of ABBREVIATIONS and ACRONYMS

<u>Abbreviations</u>	<u>Explanations</u>
-----------------------------	----------------------------

<u>And Acronyms</u>

CLB	: Configurable Logic Block
CSLA	: Carry Select Adder
DSP	: Digital Signal Processing
ECC	: Elliptic Curve Cryptography
ECPM	: Elliptic Curve Point Multiplication
FPGA	: Field Programmable Gate Array
FSM	: Finite State Machine
HLS	: High Level Synthesis
LUT	: Look-up Table
NIST	: National Institute of Standards and Technology
PA	: Point Addition
PD	: Point Doubling
PKC	: Public key Cryptography
RAM	: Random Access Memory

LIST of FIGURES

<u>Figure No:</u>	<u>Page</u>
2.1: 48x48 bit multiplication.	13
2.2: 48x48 bit multiplier unit.	13
2.3: The proposed Montgomery multiplier.	15
2.4: DSP48E based carry select adder/subtractor.	18
2.5: Time scheduling of 528 bit Montgomery multiplication.	19
2.6: Time scheduling of 1056 bit Montgomery multiplication.	19
3.1: Proposed 528 bit Barrett multiplier.	30
4.1: Point Addition and Point Doubling of elliptic curve points.	39
4.2: Overall elliptic curve point multiplication architecture.	43
5.1: Proposed HLS implementation.	50
5.2: Latency of Montgomery HLS implementation.	54

LIST of TABLES

<u>Tablo No:</u>	<u>Page</u>
2.1: Montgomery Multiplication algorithm.	7
2.2: Interleaved Montgomery multiplication algorithm.	8
2.3: Multiprecision interleaved Montgomery multiplication.	11
2.4: Area and time performances of FPGA implementations.	20
3.1: Full-Word Barrett multiplication.	28
3.2: Latency for 528 bit Barrett multiplication.	32
3.3: Area and time performances of FPGA implementations.	33
4.1: Point Doubling and Point Addition with jacobian coordinates.	40
4.2: Point Doubling and Point Addition with mixed jacobian/affine coordinates.	41
4.3: ECPM using window method ($w=3$).	44
4.4: Clock cycles for 521 bit ECPM in jacobian coordinates.	45
4.5: Clock cycles for 521 bit ECPM in mixed jacobian-affine coordinates.	45
4.6: Clock cycles for NIST P-521 ECPM in mixed jacobian-affine coordinates.	45
4.7: Fast reduction with P-521.	46
4.8: Area and time performance of point multiplication.	47
5.1: HLS Implementation results for Montgomery multiplication.	52
5.2: HLS Implementation results for Barrett multiplication.	53
5.3: Area and time performance of FPGA implementations.	53

1. INTRODUCTION

1.1. Motivation

Public-key cryptography (PKC) introduced by Diffie and Hellman [1], is widely used in today's communication systems. RSA [2] and elliptic curve [3], [4] based public-key cryptosystems heavily depend on modular multiplication. Modular multiplication operation with large integers is the main computation part in these systems. Conventional modular multiplication is an expensive operation because it requires division. In order to overcome this difficulty, effective modular multiplication algorithms are proposed in the literature. Most common ones are Montgomery and Barrett algorithms [5], [27] which replace division with multiplication and shift operations. These algorithms have computation cost and there is a demanding need for hardware acceleration of these algorithms. Therefore, efficient and high speed hardware accelerators implementing these algorithms reduce the burden of computation cost in cryptosystems. In this thesis, efficient modular multiplication hardware will be designed and accelerated in FPGA and will be compared with the state of the art designs.

1.2. Thesis Contributions

In this thesis, we first designed a fast digit based Montgomery multiplier using DSP resources of FPGAs. Digit size, $\delta = 48$ is selected for interleaved Montgomery multiplication. The proposed multiplier is developed for 528 bit Montgomery multiplication but it can be extended to perform 528ℓ bit Montgomery multiplication by combining ℓ different 528 bit Montgomery multipliers. The proposed design takes 0.43 us to compute one 528 bit Montgomery multiplication in a Virtex-7 FPGA. The proposed design is compared with the state of the art designs in literature and substantial logic savings are obtained.

We then proposed a full-word Barrett multiplier using DSP resources of FPGAs, and compared the performance results with the Montgomery multiplier. The proposed design takes 0.49 us to compute one 528 bit Barrett multiplication in a

Virtex-7 FPGA. Proposed Barrett multiplier can be a strong alternative to Montgomery multipliers in the literature.

After that, we designed an Elliptic Curve Point Multiplication (ECPM) hardware based on the proposed Montgomery multiplier. Proposed ECPM hardware supports both general and mersenne primes. It takes 4.06 ms to compute one 528 bit ECPM in any prime field and it takes 2.79 ms to compute NIST p-521 ECPM in a Virtex-7 FPGA. The proposed design is one of the fastest designs in literature and it has very modest resource utilization.

We then investigated High Level Synthesis (HLS) implementations of the Montgomery and Barrett multiplication. These algorithms are coded using C++ language and implemented in Vivado HLS tool. Promising results are obtained using specific directives. HLS implementation of Montgomery algorithm takes 1.34 us to compute one 528 bit Montgomery multiplication and HLS implementation of Barrett algorithm takes 2.57 us to compute one 528 bit Barrett multiplication.

1.3. Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, first, introduces Montgomery multiplication in detail. Then, it describes the proposed digit based Montgomery multiplier and presents the implementation results.

Chapter 3, first, introduces Barrett multiplication in detail. Then, it explains the full-word Barrett multiplier and presents the implementation results.

Chapter 4, first, introduces the Elliptic Curve point multiplication in detail. Then, it describes the proposed ECPM hardware and presents the implementation results.

Chapter 5 explains HLS implementations of Montgomery and Barrett multiplications and presents the implementation results.

Chapter 6 presents the conclusion.

2. DIGIT BASED MONTGOMERY MULTIPLIER

2.1. Introduction

The Public key cryptosystems Diffie Hellman [1], RSA [2], and elliptic curve cryptography [3, 4] are widely used in today's communication systems. Modular multiplication with large integers is the main computation in these cryptosystems. Straightforward modular multiplication is a costly operation since it requires division. The Montgomery algorithm [5] is an efficient modular multiplication technique trading the costly division for a multiplication with a precomputed value depending on the divisor. Modular multiplication with Montgomery algorithm is briefly called Montgomery multiplication. Montgomery multiplication with a divisor θ needs the precomputation

$$\Psi = -\theta^{-1} \bmod 2^\delta \quad (2.1)$$

where δ is the digit size of the multiplicands. In Montgomery multiplication, one multiplicand is multiplied with the digits of the other multiplicand, one digit at a time. After each multiplication, the partial product is accumulated and reduced by using the precomputation Ψ .

Then, at least n/δ clock cycles are needed to complete a Montgomery multiplication where n is the multiplicand bit length. Obviously, the multiplier latency decreases as the digit size δ gets larger. However, the multiplier area increases too. This area increase is very significant in cryptographic applications where computations with several hundred or even thousand bit numbers are common.

Bit serial Montgomery multipliers (digit size $\delta = 1$) are abundant in the literature [6]-[15]. These bit serial architectures process the multiplier bits, $\delta = 1$ bit at a time. In cryptographic applications, the divisor θ is typically an odd integer. Thus, the precomputation $\Psi = -\theta^{-1} \bmod 2^\delta = 1$ always when $\delta = 1$. Though the bit serial multipliers have low area complexity, their latencies are high, they need at least n/δ clocks cycles to perform a single modular multiplication.

Digit based Montgomery multipliers (digit size $\delta > 1$, precomputation $\Psi = -\theta^{-1} \bmod 2^\delta$) are also proposed in the literature to obtain speed improvement at the expense of area [13], [16] - [19]. A multiplier with $\delta = 2$ is presented in [13] and another one with $\delta = 3$ is presented in [16]. Also, the works in [17], [18] propose multiplier designs which can be implemented for arbitrary digit sizes δ . However, the work in [17] puts a serious restriction on the divisor θ such that the precomputation $\Psi = -\theta^{-1} \bmod 2^\delta = \pm 1 \bmod 2^\delta$. Also, the Montgomery multiplier in [19] uses a method based on canonic signed digit recoding and converts one of the operands into a sparse integer representation to speed up the multiplication.

The FPGAs are important implementation platforms. They can be used to implement any hardware design by their programmable logic blocks and routing. Moreover, modern FPGAs have a significant number of DSP blocks, which have advanced arithmetic capabilities. These DSP blocks commonly have 18×18 or 25×18 or 36×36 bit multipliers. Thus, one or more DSP blocks can be configured to implement a $\delta \times \delta$ bit multiplier. Such a multiplier is usually used to multiply the δ bit digits of the large operands in the FPGA implementations. Two n bit operands can be multiplied in $(n/\delta)^2$ clock cycles by using a single δ bit digit multiplier.

Many Montgomery multiplier designs taking advantage of the FPGA DSP blocks are available in the literature. A single DSP block is used in [20]-[22] to calculate the digit products needed for the Montgomery multiplication. Though this is a quite modest use of the DSP resources, the resulting latency is very high in these works. On the other hand, the Montgomery multiplier designs in [23] - [25] use several DSP blocks in Xilinx FPGAs containing 25×18 bit multipliers to build two $\delta \times \delta$ bit digit multipliers. One of the multipliers is used to multiply the digits of the multiplicands. The other one is used in the multiplications needed for the reduction with the divisor θ . The work in [23] uses digit multipliers with the digit sizes $\delta = 32, 64$. The work in [24] uses digit multipliers with the digit sizes $\delta = 16, 32, 64$ and keep the multiplicands, divisor, and result in the internal RAM of the FPGA to use fewer logic blocks. The work in [25] uses 256×256 digit multipliers built from Xilinx IP cores.

This study presents a Montgomery multiplier designed to be implemented in Virtex-5 and higher Xilinx FPGAs, which have the DSP48E slices containing 25×18 bit multipliers. The proposed design is developed for 528 bit Montgomery

multiplication primarily. Yet, it can be extended to the higher bit lengths by combining the 528 bit Montgomery multipliers. The proposed design has several 48×48 bit multipliers built from the DSP48E slices.

The proposed design processes the digits of a multiplicand b iteratively. In each iteration, a single 48 bit digit of b is multiplied by the other multiplicand a . Then, the result is accumulated, and reduced by the Montgomery method. Each iteration needs twenty three 48×48 bit multiplications for a 528 bit a . These twenty three partial products are performed in eight cycles by three digit multipliers such that the partial products computed in each cycle do not overlap. Thus, the partial products can be accumulated separately.

Also, a fast adder is needed to accumulate the partial products without large carry propagation delays. This is achieved by a carry select adder built from the DSP48E slices performing 48 bit additions.

This chapter is organized as follows: Section 2.2 introduces Montgomery modular multiplication technique. Section 2.3 illustrates the proposed Montgomery multiplier. Section 2.4 presents Xilinx Virtex 7 implementation results for 528 and 1056 bit Montgomery multiplications and comparisons with other designs. Section 2.5 discusses the conclusion.

2.2. Montgomery Technique

Montgomery product of two integers a and b is defined for some integer Δ , as follows.

$$r = ab 2^{-\Delta} \bmod \theta \quad (2.2)$$

The computation above can be done without division by using the precomputation $-\theta^{-1} \bmod 2^\Delta$ if θ is odd.

2.2.1. Montgomery Domain

In the Montgomery product $r = ab 2^{-\Delta} \bmod \theta$, the integers a , b , and r are actually residual representations of some other integers A , B , and R .

$$\begin{aligned}
a &= A2^\Delta \bmod \theta \\
b &= B2^\Delta \bmod \theta \\
r &= R2^\Delta \bmod \theta = AB2^\Delta \bmod \theta
\end{aligned} \tag{2.3}$$

This special representation is called Montgomery domain representation. Here, $R = AB \bmod \theta$ is the actually intended modular multiplication. And, it is equivalent to the Montgomery product in Montgomery domain.

$$\begin{aligned}
ab2^{-\Delta} \bmod \theta &= (A2^\Delta)(B2^\Delta)2^{-\Delta} \bmod \theta \\
&= AB2^\Delta \bmod \theta
\end{aligned} \tag{2.4}$$

The Montgomery product $r = ab 2^{-\Delta} \bmod \theta$ can be computed much faster than $R = AB \bmod \theta$ by using the precomputation $-\theta^{-1} \bmod 2^\Delta$. Obviously, converting the integers A, B, R into a, b, r in Montgomery domain and converting them back require extra computation. Nevertheless, this is an affordable computational overhead when a large number of modular multiplications are performed in Montgomery domain.

2.2.2. Montgomery Reduction Method

Let $-\theta^{-1} \bmod 2^\Delta$ be precomputed. After computing $q = u\psi \bmod 2^\Delta$, the integer u can be reduced by Δ bits without division as follows.

$$r' = (u + \theta q)/2^\Delta \tag{2.5}$$

This is called Montgomery reduction and its result is actually equivalent to the modular reduction below.

$$r' = (u + \theta q)/2^\Delta \equiv r = u2^{-\Delta} \bmod \theta \tag{2.6}$$

The above equivalence relation can be proven as follows. Because the divisor θ is odd, $\gcd(\theta, 2^\Delta) = 1$ and there exist two integers $(2^{-\Delta} \bmod \theta)$ and $(\theta^{-1} \bmod 2^\Delta)$ such that

$$2^\Delta(2^{-\Delta} \bmod \theta) + \theta(\theta^{-1} \bmod 2^\Delta) = 1 \quad (2.7)$$

The precomputation $\psi = -\theta^{-1} \bmod 2^\Delta$. Thus,

$$2^{-\Delta} \bmod \theta = (1 + \theta\psi)/2^\Delta \quad (2.8)$$

and

$$\begin{aligned} r = u2^{-\Delta} \bmod \theta &\equiv (u + u\theta\psi)/2^\Delta \\ &\equiv (u + \theta(u\psi \bmod 2^\Delta))/2^\Delta \\ &\equiv (u + \theta q)/2^\Delta \end{aligned} \quad (2.9)$$

where $q = u\psi \bmod 2^\Delta$.

Table 2.1: Montgomery multiplication algorithm.

Input: integers a, b, θ such that $a < a_{sup}$, $b < 2^n$	
Output: $r = ab \cdot 2^{-n} \bmod \theta$ such that $r < a_{sup}$	
Precomputation: $\psi = -\theta^{-1} \bmod 2^n$	
function ModMult (a, b, θ, ψ)	
1.	$v \leftarrow ab$
2.	$u \leftarrow (v + \theta q) / 2^n$ for $q = v\psi \bmod 2^n$
3.	if $u \geq a_{sup}$
4.	$r \leftarrow u - \theta$
5.	else
6.	$r \leftarrow u$
7.	return r
end function	

Table 2.2: Interleaved Montgomery multiplication algorithm.

Input: integers a, b, θ such that $a < a_{sup}, b < 2^n$	
Output: $r = ab 2^{-n} \bmod \theta$ such that $r < a_{sup}$	
Precomputation: $\psi = -\theta^{-1} \bmod 2^\delta$	
function ModMult (a, b, θ, ψ)	
1.	$u \leftarrow 0$
2.	for $i = 0$ to $n/\delta - 1$
3.	$v \leftarrow (u + aB_i)$ for $B_i = \lfloor b/2^{i\delta} \rfloor \bmod 2^\delta$
4.	$u \leftarrow (v + \theta q_i)/2^\delta$ for $q_i = v\psi \bmod 2^\delta$
5.	end for
6.	if $u \geq a_{sup}$
7.	$r \leftarrow u - \theta$
8.	else
9.	$r \leftarrow u$
10.	return r
end function	

2.2.3. Montgomery Multiplication

Let $\psi = -\theta^{-1} \bmod 2^\delta$ be precomputed. After computing $q = ab\psi \bmod 2^\delta$, the product ab can be reduced by Δ bits without division as shown in (2.5).

$$r' = (u + \theta q)/2^\Delta \quad (2.10)$$

This is called Montgomery multiplication. As seen from (2.5), r' is actually equivalent to the Montgomery product.

$$r' = (ab + \theta q)/2^\Delta \equiv r = ab2^{-\Delta} \bmod \theta \quad (2.11)$$

Note that the Montgomery multiplication result satisfies

$$a < a_{sup}, \quad b < 2^\Delta \rightarrow r' = \frac{ab + \theta q}{2^\Delta} < a_{sup} + \theta \quad (2.12)$$

where $q = ab\psi \bmod 2^\Delta$ and $\psi = -\theta^{-1} \bmod 2^\Delta$. Then, a final subtraction of the divisor θ may be needed to reduce r' below the upper bound a_{sup} as follows.

$$r = ab2^{-\Delta} \bmod \theta \equiv r' = r' - \theta \quad \text{if } r' \geq \theta \quad (2.13)$$

Table 2.1 implements the Montgomery multiplication for $\Delta = n$. The operand a is multiplied by n bit b in Step 1. Then, n bit Montgomery reduction is performed in Step 2. A final subtraction may be performed in Step 4 to keep the result below a_{sup} . Here, Step 1 and Step 2 compute (2.11). Step 4 computes (2.12).

2.2.4. Interleaved Montgomery Multiplication

Table 2.2 interleaves the multiplication and reduction steps. Table 2.2 implements the Montgomery multiplication for $\Delta = \delta < n$. In Step 3, the operand a is multiplied by the δ bit digits of b and the result is accumulated. In Step 4, δ bit Montgomery reduction is performed. In the iteration $i = \{0, 1, \dots, n/\delta - 1\}$,

$$u = \frac{a\beta + \theta(a\beta(-\theta^{-1}) \bmod 2^{(i+1)\delta})}{2^{(i+1)\delta}} < a_{sup} + \theta \quad (2.14)$$

for the operands $a < a_{sup}$ and $b < 2^n$ where

$$\beta = b \bmod 2^{(i+1)\delta} = ((b_{(i+1)\delta-1}, \dots, b_0)_2) \quad (2.15)$$

That is $u = (a\beta + \theta q)/2^{(i+1)\delta}$ where

$$\begin{aligned} \beta &= b \bmod 2^\delta, & q &= a\beta(-\theta^{-1}) \bmod 2^\delta, & \text{for } i &= 0 \\ \beta &= b \bmod 2^{2\delta}, & q &= a\beta(-\theta^{-1}) \bmod 2^{2\delta}, & \text{for } i &= 1 \\ \beta &= b \bmod 2^{3\delta}, & q &= a\beta(-\theta^{-1}) \bmod 2^{3\delta}, & \text{for } i &= 2 \\ &\dots & & & & \\ \beta &= b \bmod 2^n, & q &= a\beta(-\theta^{-1}) \bmod 2^n, & \text{for } i &= n/\delta - 1 \end{aligned} \quad (2.16)$$

The proof of this fact is given in Theorem 1 of [18]. Note that Table 2.1 and Table 2.2 find the same result. After the last iteration of Table 2.2, $\beta = b \bmod 2^n = b$, $q = a\beta\psi \bmod 2^n = ab\psi \bmod 2^n$ for $\psi = -\theta^{-1} \bmod 2^n$. And $u = (ab + \theta q)/2^n$.

2.2.5 Multiprecision Implementation

In practice, the operands a , b and the divisor θ are represented by $n \geq \lceil \log_2 \theta \rceil$ bits. Also, the upper bound a_{sup} has not to be less than the divisor θ . That is

$$a, b, \theta < 2^n, \quad a < a_{sup} \in [0, 2^n] \quad (2.17)$$

Now, there are two practical cases here.

- $a_{sup} = \theta < 2^n$ (usual case).
- $\theta < a_{sup} = 2^n$ (incomplete arithmetic [26]).

In the usual case, $a_{sup} = \theta$ and the final result is guaranteed to be equal to the Montgomery product. Notice that the Montgomery multiplication in (2.11) and (2.12) becomes

$$\begin{aligned} r = ab2^{-n} \bmod \theta &\equiv r' = (ab + \theta q)/2^n < 2\theta \\ r = ab2^{-n} \bmod \theta &= r' = r' - \theta \quad \text{if } r' \geq \theta \end{aligned} \quad (2.18)$$

where $\Delta = n$. In incomplete arithmetic, $a_{sup} = 2^n$. Then, the Montgomery multiplication in (2.11) and (2.12) becomes

$$\begin{aligned} r = ab2^{-n} \bmod \theta &\equiv r' = (ab + \theta q)/2^n < 2^n + \theta \\ r = ab2^{-n} \bmod \theta &= r' = r' - \theta \quad \text{if } r' \geq 2^n \end{aligned} \quad (2.19)$$

where $\Delta = n$. Note that the final result $r' < 2^n$ and $r' < \theta$ does not necessarily hold. Thus, r' is only an n bit value equivalent to the Montgomery product in the second

Table 2.3: Multiprecision interleaved Montgomery multiplication.

Input: n bit integers a, b, θ	
Output: n bit integer $r = ab 2^{-n} \bmod \theta$	
Precomputation: δ bit integer $\psi = -\theta^{-1} \bmod 2^\delta$	
function ModMult (a, b, θ, ψ)	
1.	$u[n:0] \leftarrow 0$
2.	for $i = 0$ to $n/\delta - 1$
3.	$B_i \leftarrow b[\delta(i+1) - 1:\delta i], \quad v[n+\delta:0] \leftarrow aB_i + u[n:0]$
4.	$q_i = \psi v[\delta-1:0] \bmod 2^\delta, \quad u[n:0] \leftarrow (\theta q_i + v[n+\delta:0])/2^\delta$
5.	end for
6.	if $u[n:0] \geq 2^n$
7.	$r \leftarrow u[n:0] - \theta$
8.	else
9.	$r \leftarrow u[n-1:0]$
10.	return r
end function	

case. The purpose of the incomplete arithmetic is to keep the operands and the results in the range $[0, 2^n)$.

Table 2.3 gives a multiprecision implementation of the interleaved Montgomery multiplication. In the algorithm, the operands are n bit, the divisor is n bit, the digits are δ bit, and the upper bound $a_{sup} = 2^n$.

$$a, b, \theta < 2^n = a_{sup}, \quad B_i < 2^\delta \quad (2.20)$$

Also, $u < a_{sup} + \theta$ because of (2.13). As a result, the intermediate variables u and v are bounded as follows.

$$u < 2^n + \theta = 2^{n+1}, \quad v = aB_i + u < 2^{n+\delta+1} \quad (2.21)$$

$a_{sup} = 2^n$ is chosen instead of $a_{sup} = \theta$ in the algorithm (incomplete arithmetic). This choice simplifies the comparison in Step 6. Note that the

comparison $u \geq a_{sup} = 2^n$ requires only checking the n^{th} bit of u while the comparison $u \geq a_{sup} = \theta$ requires $(n+1)$ bit subtraction.

2.3. Proposed Montgomery Multiplier

The proposed design implements the interleaved Montgomery Multiplication in Table 2.3 by employing the DSP48E slices of Virtex-5 and higher Xilinx FPGAs.

The proposed design is developed for 528 bit Montgomery multiplication primarily. Nevertheless, it can be extended to perform 1056 bit Montgomery multiplication by combining two 528 bit Montgomery multipliers. The Montgomery multipliers of higher bit lengths are also possible by combining more 528 bit Montgomery multipliers.

The digit size $\delta = 48$ bits are chosen for the proposed design. The required 48×48 bit multiplier units are implemented from DSP48E slices. The proposed 528 bit Montgomery multiplier has three 48×48 bit multiplier units. These three units compute all the digit products required for the Montgomery multiplication. The computed digit products are properly shifted and accumulated by using a carry select adder built from DSP blocks.

2.3.1. DSP48E based 48×48 bit Multiplier Unit

Figure 2.1 and 2.2 illustrate how the designed 48×48 bit multiplier unit works. The 48 bit inputs X and Y of the multiplier unit are divided into smaller parts. The input X is divided into the 24 bit parts X_0 and X_1 . The input Y is divided into the 16 bit parts Y_0 , Y_1 , and Y_2 . Six DSP blocks compute the partial products X_0Y_0 , X_0Y_1 , X_0Y_2 , X_1Y_0 , X_1Y_1 , and X_1Y_2 . Then, these products are properly shifted and added together to find the 96 bit output $Z = XY$.

As seen in Figure 2.2, the partial products are stored in the buffers of the DSP blocks and added by the carry chain adders of the logic blocks. Also, the final output is stored in the flip flops of the logic blocks.

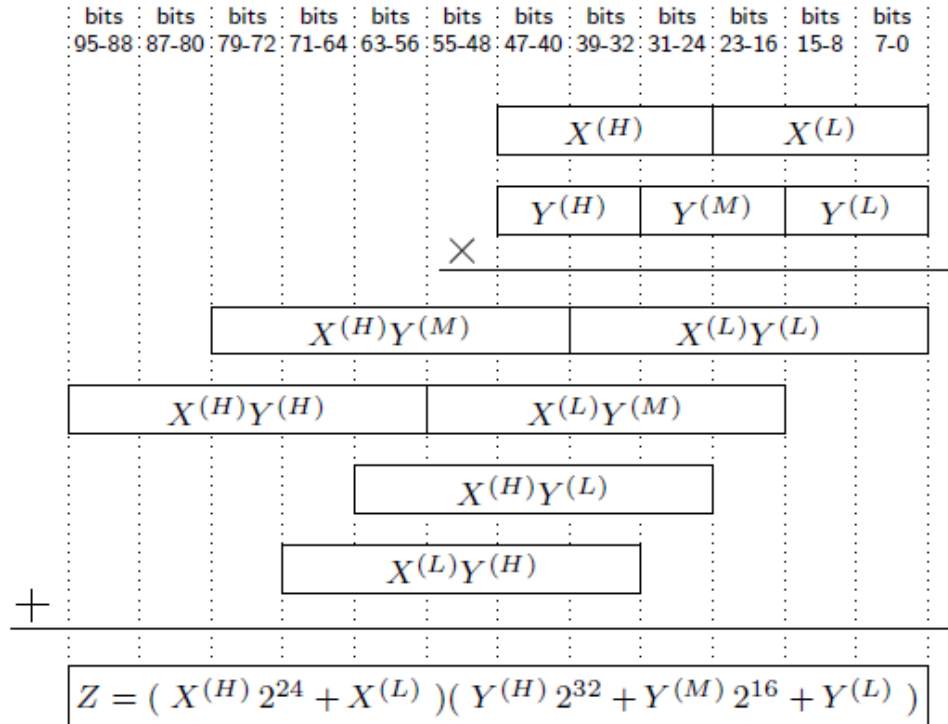


Figure 2.1: 48x48 bit multiplication.

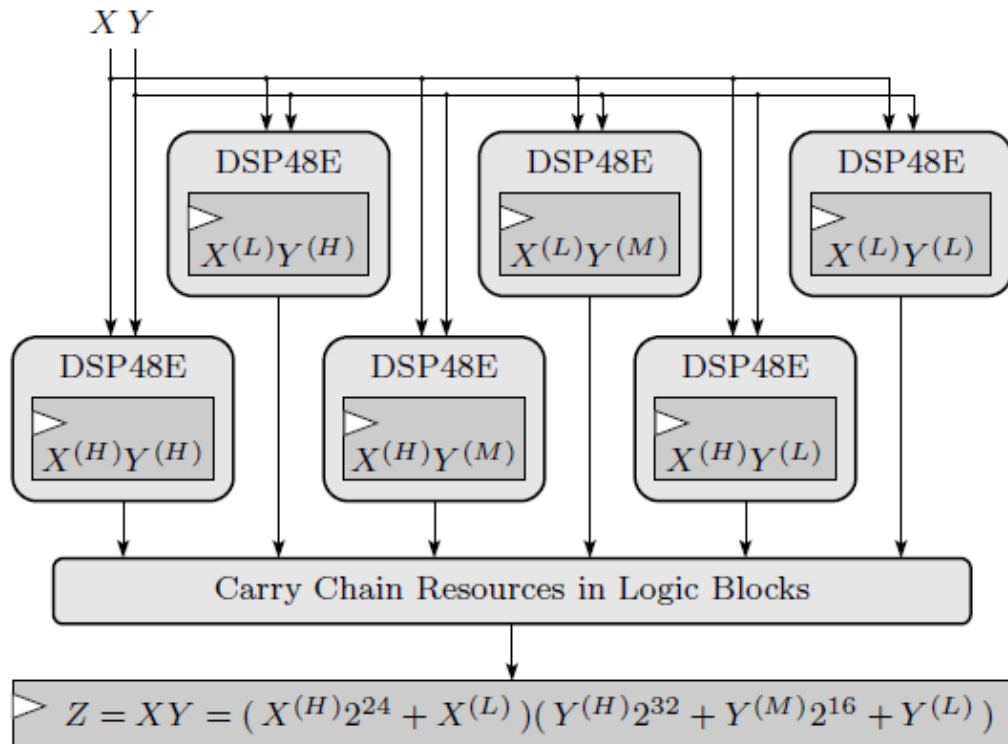


Figure 2.2: 48x48 bit multiplier unit.

2.3.2. General Structure

Figure 2.3 depicts the proposed design. Since the digit size $\delta = 48$ bits, the operands a , b and divisor θ are divided into 48 bit digits A_i , B_i and θ_i . Then, they are divided into 11 digits for $528 = 11 \times 48$ bit Montgomery multiplication.

$$\begin{aligned} A_{10}, A_9, \dots, A_2, A_1, A_0 \\ B_{10}, B_9, \dots, B_2, B_1, B_0 \\ \theta_{10}, \theta_9, \dots, \theta_2, \theta_1, \theta_0 \end{aligned} \quad (2.22)$$

And, they are divided into 22 digits for $1056 = 22 \times 48$ bit Montgomery multiplication.

$$\begin{aligned} A_{21}, A_{20}, \dots, A_2, A_1, A_0 \\ B_{21}, B_{20}, \dots, B_2, B_1, B_0 \\ \theta_{21}, \theta_{20}, \dots, \theta_2, \theta_1, \theta_0 \end{aligned} \quad (2.23)$$

The products AB_i , and θq_i , in the i^{th} iteration of Table 2.3 are computed from the 48×48 bit multiplications

$$A_j B_i, q_i = \psi v \bmod 2^{48}, \theta_j q_i \text{ for } 0 \leq j < J \quad (2.24)$$

where $J = 11$ for 528 bit Montgomery multiplication and $J = 22$ for 1056 bit Montgomery multiplication. The proposed design has three 48×48 bit multiplier units. In 8 cycles, three units compute the digit products

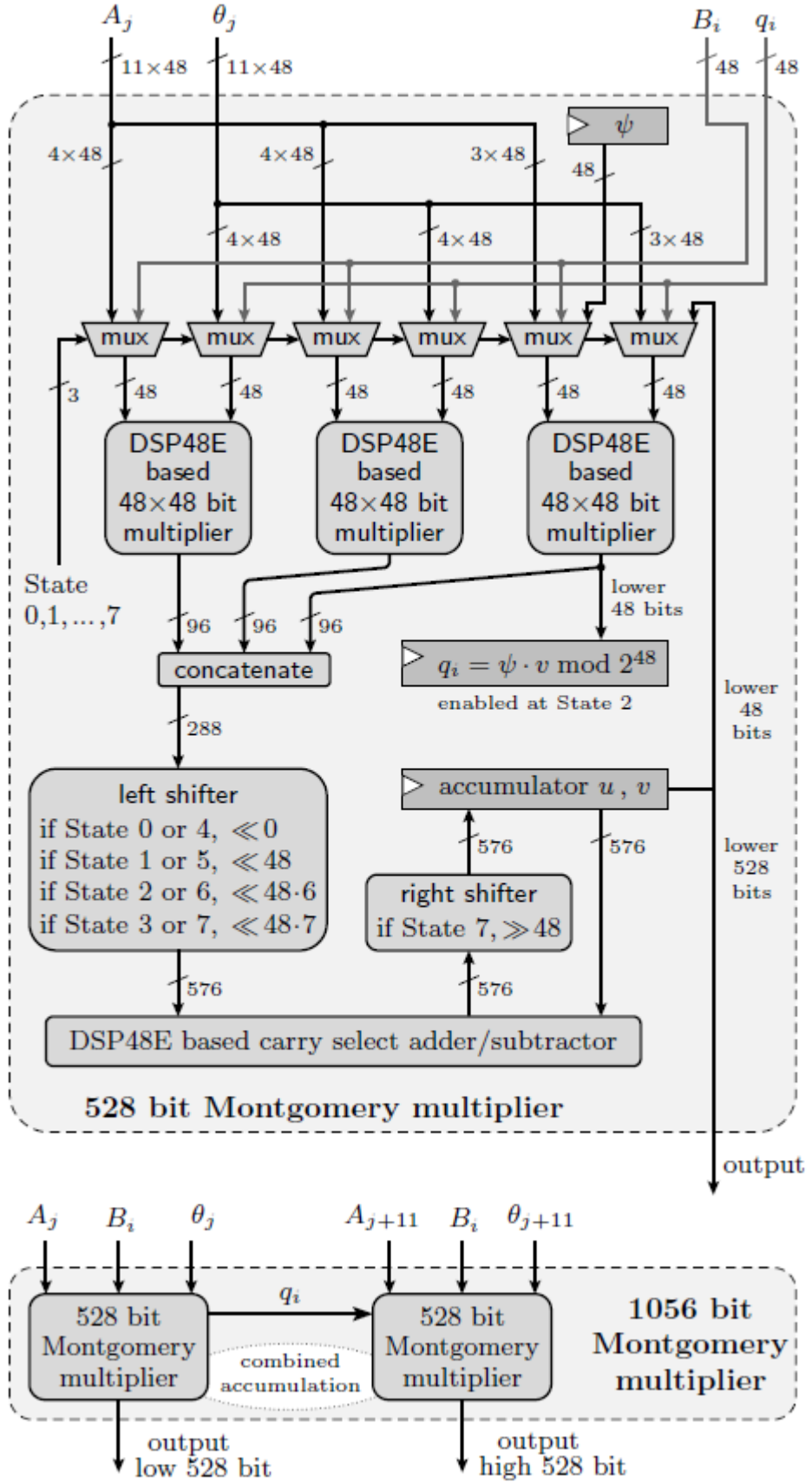


Figure 2.3: The proposed Montgomery multiplier.

Cycle 0:	A_0B_i, A_2B_i, A_4B_i	
Cycle 1:	A_1B_i, A_3B_i, A_5B_i	
Cycle 2:	$A_7B_i, A_9B_i, q_i = \psi v \bmod 2^{48}, \theta_j q_i$	
Cycle 3:	$A_6B_i, A_8B_i, A_{10}B_i$	
Cycle 4:	$\theta_0 q_i, \theta_2 q_i, \theta_4 q_i$	(2.25)
Cycle 5:	$\theta_1 q_i, \theta_3 q_i, \theta_5 q_i$	
Cycle 6:	$\theta_7 q_i, \theta_9 q_i, 0$	
Cycle 7:	$\theta_6 q_i, \theta_8 q_i, \theta_{10} q_i$	

These are all the digit products required for 528 bit Montgomer multiplication. The digit products computed in a cycle are concatenated, properly shifted, and accumulated. Concatenation is sufficient to combine the digit products computed in a cycle thanks to the clever strategy adopted by the proposed design. As seen from (2.23), if $A_j B_i$ is computed in one cycle, $A_{j+1} B_i$ is computed in another cycle to prevent overlapping. The same is true for $\theta_j q_i$. Thus, the digit products computed in each cycle are added to the different parts of the accumulated sum.

After the cycles 0, 1, 2, 3, the accumulator and the register q_i respectively hold the values

$$v = u + aB_i, \quad q_i = v\psi \bmod 2^{48} \quad (2.26)$$

where u is the initial accumulator value at the beginning of the i^{th} iteration. After the cycles 4, 5, 6, 7 the initial accumulator value u has been updated as follows.

$$u = (v + \theta q_i)/2^{48} = (u + aB_i + \theta q_i)/2^{48} \quad (2.27)$$

Note that this is Step 2 of Table 2.3.

DSP48E based carry select adder enables the fast accumulation of the partial products in each cycle. Also, this unit can perform subtraction. Thus, it is used to perform the final subtraction in Step 3 of Table 2.3 too.

For 1056 bit Montgomery multiplication, two 528 bit Montgomery multiplier are used. One of them handles the lower 11 digits of a and θ . The other one handles

the upper 11 digits of them. Moreover, The partial product accumulations in both multiplier are combined. That is a single $1056 + 48 = 1104$ bit accumulator is used instead of two separate $528 + 48 = 576$ bit accumulators. The additions and shifts are not 576 bit but 1104 bit.

Also, because $q_i = \psi v \bmod 2^{48}$ is calculated from the lower digits of the accumulated value v , the 528 bit Montgomery multiplier handling the lower digits computes q_i in each iteration and gives the q_i value as input to the multiplier handling the upper digits. As a result, both multipliers use the same q_i value.

2.3.3. DSP48E based Carry Select Adder/Subtractor

Applications usually require large Montgomery multiplications. Thus, Montgomery multipliers have to perform large additions to add the partial products. Naturally, carry propagation delay is a big problem in Montgomery multipliers. To prevent the performance degradation due to carry propagation, the proposed design uses carry select addition (CSLA), which is a fast addition scheme.

Figure. 2.3 shows the carry select adder implemented with DSP blocks for the proposed design. A carry select adder splits a large addition into smaller additions. DSP48E slices can perform very fast 48 bit additions. Thus, the proposed design splits large additions into 48 bit additions and computes these 48 bit additions with DSP blocks.

In carry select addition, each small addition is calculated for two cases: one for 0 carry input and the other one for 1 carry input. The correct carry input and the corresponding correct sum are selected by the multiplexers as shown in Figure 2.3. Though this scheme doubles the DSP blocks and needs multiplexers, it breaks large carry chains into smaller ones, and thus enables very fast additions.

The designed carry select adder can perform subtraction too. Thus, it is also used for the final subtraction in Step 7 of Table 2.3. The DSP48E slices perform addition or subtraction depending on the four bit ALUMODE control input. They perform addition when ALUMODE is 0 and subtraction when ALUMODE is 1.

As discussed in the previous section, a 528 bit Montgomery multiplier requires a 576 bit adder while a 1056 bit Montgomery multiplier requires a 1104 bit adder. A

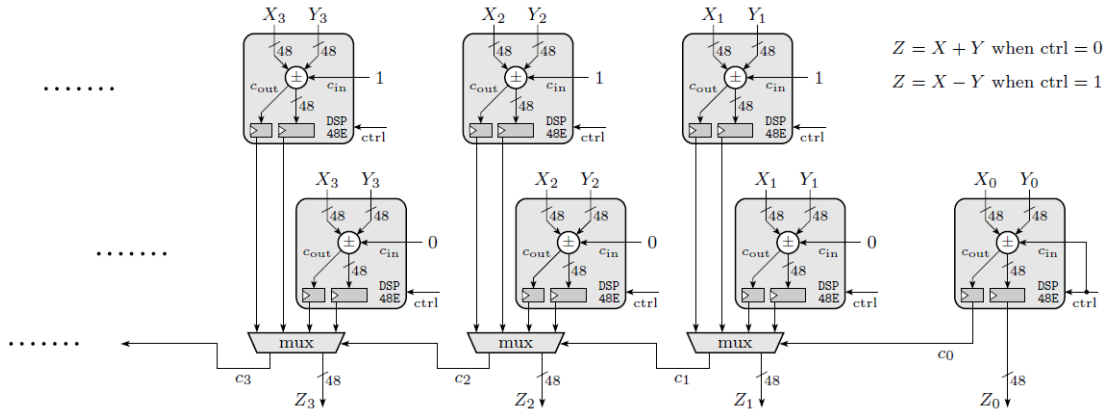


Figure 2.4: DSP48E based carry select adder/subtractor.

576 = 12x48 bit adder can be implemented by 12 DSP blocks while a 1104 = 23x48 bit adder can be implemented by 23 DSP blocks.

However, carry select addition doubles the required DSP blocks except the initial one. Thus, 576 bit carry select adder requires 12+11=23 DSP blocks while 1056 bit carry select adder requires 23+22=45 DSP blocks.

2.3.4. Latency of Montgomery Multiplier

The latency of the proposed design are 91 cycles for the 528 bit Montgomery Multiplier and 179 cycles for the 1056 bit Montgomery Multiplier.

Now, we explain the delays contributing this latency. There are 11 digits B_i for 528 bit Montgomery multiplication as seen from (2.20) and there are 22 digits B_i for 1056 bit Montgomery multiplication as seen from (2.21). Then, processing the digits B_i requires 11 iterations for 528 bit Montgomery multiplication and 22 iterations for 1056 bit Montgomery multiplication.

Also, each iteration is completed in eight cycles as seen from (2.23). Thus, all iterations and the final subtraction respectively require $8 \times 11 + 1 = 89$ and $8 \times 22 + 1 = 177$ cycles for 528 bit and 1056 bit Montgomery multiplications. The final subtraction is the last step of the Montgomery multiplication in Table 2.3 and requires a single cycle.

Moreover, there is a two cycle delay due to the 48x48 bit digit multipliers as can be seen from the time scheduling of the design in Figure 2.5 and Figure 2.6. Even though the digit multipliers calculate a product in each cycle (throughput is one

Divisor	bits 527-480	bits 479-432					bits 143-96	bits 95-48	bits 47-0
	θ_{10}	θ_9	• • • • •				θ_2	θ_1	θ_0
Operand	A_{10}	A_9	• • • • •				A_2	A_1	A_0

528 bit Operand Case	DSP blocks compute and store the partial products of 3 digit multiplications.	Carry chain adders combine the partial products to find the 3 digit multiplications.	3 digit multiplications are concatenated, appropriately shifted, and accumulated.
cycle $n-1$	$\theta_{10}q_{i-1}, \theta_8 q_{i-1}, \theta_6 q_{i-1}$	$\theta_9 q_{i-1}, \theta_7 q_{i-1}, 0$	$S = S + \{ \theta_5 q_{i-1}, \theta_3 q_{i-1}, \theta_1 q_{i-1} \} \ll 48$
cycle n	$A_4 B_i, A_2 B_i, A_0 B_i$	$\theta_{10}q_{i-1}, \theta_8 q_{i-1}, \theta_6 q_{i-1}$	$S = S + \{ \theta_9 q_{i-1}, \theta_7 q_{i-1} \} \ll (48 \cdot 7)$
cycle $n+1$	$A_5 B_i, A_3 B_i, A_1 B_i$	$A_4 B_i, A_2 B_i, A_0 B_i$	$S = S + \{ \theta_{10}q_{i-1}, \theta_8 q_{i-1}, \theta_6 q_{i-1} \} \ll (48 \cdot 6)$
cycle $n+2$	$A_9 B_i, A_7 B_i, S\psi \bmod 2^{48}$	$A_5 B_i, A_3 B_i, A_1 B_i$	$S = S + \{ A_4 B_i, A_2 B_i, A_0 B_i \}$
cycle $n+3$	$A_{10}B_i, A_8 B_i, A_6 B_i$	$A_9 B_i, A_7 B_i, q_i = S\psi \bmod 2^{48}$	$S = S + \{ A_5 B_i, A_3 B_i, A_1 B_i \} \ll 48$
cycle $n+4$	$\theta_4 q_i, \theta_2 q_i, \theta_0 q_i$	$A_{10}B_i, A_8 B_i, A_6 B_i$	$S = S + \{ A_9 B_i, A_7 B_i \} \ll (48 \cdot 7)$
cycle $n+5$	$\theta_5 q_i, \theta_3 q_i, \theta_1 q_i$	$\theta_4 q_i, \theta_2 q_i, \theta_0 q_i$	$S = S + \{ A_{10}B_i, A_8 B_i, A_6 B_i \} \ll (48 \cdot 6)$
cycle $n+6$	$\theta_9 q_i, \theta_7 q_i, 0$	$\theta_5 q_i, \theta_3 q_i, \theta_1 q_i$	$S = S + \{ \theta_4 q_i, \theta_2 q_i, \theta_0 q_i \}$
cycle $n+7$	$\theta_{10}q_i, \theta_8 q_i, \theta_6 q_i$	$\theta_9 q_i, \theta_7 q_i, 0$	$S = S + \{ \theta_5 q_i, \theta_3 q_i, \theta_1 q_i \} \ll 48$
cycle $n+8$	$A_4 B_{i+1}, A_2 B_{i+1}, A_0 B_{i+1}$	$\theta_{10}q_i, \theta_8 q_i, \theta_6 q_i$	$S = S + \{ \theta_9 q_i, \theta_7 q_i \} \ll (48 \cdot 7)$
cycle $n+9$	$A_5 B_{i+1}, A_3 B_{i+1}, A_1 B_{i+1}$	$A_4 B_{i+1}, A_2 B_{i+1}, A_0 B_{i+1}$	$S = S + \{ \theta_{10}q_i, \theta_8 q_i, \theta_6 q_i \} \ll (48 \cdot 6)$
cycle $n+10$	$A_9 B_{i+1}, A_7 B_{i+1}, S\psi \bmod 2^{48}$	$A_5 B_{i+1}, A_3 B_{i+1}, A_1 B_{i+1}$	$S = S + \{ A_4 B_{i+1}, A_2 B_{i+1}, A_0 B_{i+1} \}$

Figure 2.5: Time scheduling of 528 bit Montgomery multiplication.

Divisor	bits 1055-1008	bits 1007-960					bits 143-96	bits 95-48	bits 47-0
	θ_{21}	θ_{20}	• • • • •				θ_2	θ_1	θ_0
Operand	A_{21}	A_{20}	• • • • •				A_2	A_1	A_0

1056 bit Operand Case	DSP blocks compute and store the partial products of 6 digit multiplications.	Carry chain adders combine the partial products to find the 6 digit multiplications.	6 digit multiplications are concatenated, appropriately shifted, and accumulated.
cycle $n-1$	$\theta_{10}q_{i-1}, \theta_8 q_{i-1}, \theta_6 q_{i-1}, \theta_{21}q_{i-1}, \theta_{19}q_{i-1}, \theta_{17}q_{i-1}$	$\theta_9 q_{i-1}, \theta_7 q_{i-1}, 0, \theta_{20}q_{i-1}, \theta_{18}q_{i-1}, 0$	$S = S + \{ \theta_5 q_{i-1}, \theta_3 q_{i-1}, \theta_1 q_{i-1} \} \ll 48$ $+ \{ \theta_{16}q_{i-1}, \theta_{14}q_{i-1}, \theta_{12}q_{i-1} \} \ll (48 \cdot 12)$
cycle n	$A_4 B_i, A_2 B_i, A_0 B_i, A_{15}B_i, A_{13}B_i, A_{11}B_i$	$\theta_{10}q_{i-1}, \theta_8 q_{i-1}, \theta_6 q_{i-1}, \theta_{21}q_{i-1}, \theta_{19}q_{i-1}, \theta_{17}q_{i-1}$	$S = S + \{ \theta_9 q_{i-1}, \theta_7 q_{i-1} \} \ll (48 \cdot 7)$ $+ \{ \theta_{20}q_{i-1}, \theta_{18}q_{i-1} \} \ll (48 \cdot 18)$
cycle $n+1$	$A_5 B_i, A_3 B_i, A_1 B_i, A_{16}B_i, A_{14}B_i, A_{12}B_i$	$A_4 B_i, A_2 B_i, A_0 B_i, A_{15}B_i, A_{13}B_i, A_{11}B_i$	$S = S + \{ \theta_{10}q_{i-1}, \theta_8 q_{i-1}, \theta_6 q_{i-1} \} \ll (48 \cdot 6)$ $+ \{ \theta_{21}q_{i-1}, \theta_{19}q_{i-1}, \theta_{17}q_{i-1} \} \ll (48 \cdot 17)$
cycle $n+2$	$A_9 B_i, A_7 B_i, S\psi \bmod 2^{48}, A_{20}B_i, A_{18}B_i, 0$	$A_5 B_i, A_3 B_i, A_1 B_i, A_{16}B_i, A_{14}B_i, A_{12}B_i$	$S = S + \{ A_4 B_i, A_2 B_i, A_0 B_i \}$ $+ \{ A_{15}B_i, A_{13}B_i, A_{11}B_i \} \ll (48 \cdot 11)$
cycle $n+3$	$A_{10}B_i, A_8 B_i, A_6 B_i, A_{21}B_i, A_{19}B_i, A_{17}B_i$	$A_9 B_i, A_7 B_i, q_i = S\psi \bmod 2^{48}, A_{20}B_i, A_{18}B_i, 0$	$S = S + \{ A_5 B_i, A_3 B_i, A_1 B_i \} \ll 48$ $+ \{ A_{16}B_i, A_{14}B_i, A_{12}B_i \} \ll (48 \cdot 12)$
cycle $n+4$	$\theta_4 q_i, \theta_2 q_i, \theta_0 q_i, \theta_{15}q_i, \theta_{13}q_i, \theta_{11}q_i$	$A_{10}B_i, A_8 B_i, A_6 B_i, A_{21}B_i, A_{19}B_i, A_{17}B_i$	$S = S + \{ A_9 B_i, A_7 B_i \} \ll (48 \cdot 7)$ $+ \{ A_{20}B_i, A_{18}B_i \} \ll (48 \cdot 18)$
cycle $n+5$	$\theta_5 q_i, \theta_3 q_i, \theta_1 q_i, \theta_{16}q_i, \theta_{14}q_i, \theta_{12}q_i$	$\theta_4 q_i, \theta_2 q_i, \theta_0 q_i, \theta_{15}q_i, \theta_{13}q_i, \theta_{11}q_i$	$S = S + \{ A_{10}B_i, A_8 B_i, A_6 B_i \} \ll (48 \cdot 6)$ $+ \{ A_{21}B_i, A_{19}B_i, A_{17}B_i \} \ll (48 \cdot 17)$
cycle $n+6$	$\theta_9 q_i, \theta_7 q_i, 0, \theta_{20}q_i, \theta_{18}q_i, 0$	$\theta_5 q_i, \theta_3 q_i, \theta_1 q_i, \theta_{16}q_i, \theta_{14}q_i, \theta_{12}q_i$	$S = S + \{ \theta_4 q_i, \theta_2 q_i, \theta_0 q_i \}$ $+ \{ \theta_{15}q_i, \theta_{13}q_i, \theta_{11}q_i \} \ll (48 \cdot 11)$
cycle $n+7$	$\theta_{10}q_i, \theta_8 q_i, \theta_6 q_i, \theta_{21}q_i, \theta_{19}q_i, \theta_{17}q_i$	$\theta_9 q_i, \theta_7 q_i, 0, \theta_{20}q_i, \theta_{18}q_i, 0$	$S = S + \{ \theta_5 q_i, \theta_3 q_i, \theta_1 q_i \} \ll 48$ $+ \{ \theta_{16}q_i, \theta_{14}q_i, \theta_{12}q_i \} \ll (48 \cdot 12)$
cycle $n+8$	$A_4 B_{i+1}, A_2 B_{i+1}, A_0 B_{i+1}, A_{15}B_{i+1}, A_{13}B_{i+1}, A_{11}B_{i+1}$	$\theta_{10}q_i, \theta_8 q_i, \theta_6 q_i, \theta_{21}q_i, \theta_{19}q_i, \theta_{17}q_i$	$S = S + \{ \theta_9 q_i, \theta_7 q_i \} \ll (48 \cdot 7)$ $+ \{ \theta_{20}q_i, \theta_{18}q_i \} \ll (48 \cdot 18)$
cycle $n+9$	$A_5 B_{i+1}, A_3 B_{i+1}, A_1 B_{i+1}, A_{16}B_{i+1}, A_{14}B_{i+1}, A_{12}B_{i+1}$	$A_4 B_{i+1}, A_2 B_{i+1}, A_0 B_{i+1}, A_{15}B_{i+1}, A_{13}B_{i+1}, A_{11}B_{i+1}$	$S = S + \{ \theta_{10}q_i, \theta_8 q_i, \theta_6 q_i \} \ll (48 \cdot 6)$ $+ \{ \theta_{21}q_i, \theta_{19}q_i, \theta_{17}q_i \} \ll (48 \cdot 17)$
cycle $n+10$	$A_9 B_{i+1}, A_7 B_{i+1}, S\psi \bmod 2^{48}, A_{20}B_{i+1}, A_{18}B_{i+1}, 0$	$A_5 B_{i+1}, A_3 B_{i+1}, A_1 B_{i+1}, A_{16}B_{i+1}, A_{14}B_{i+1}, A_{12}B_{i+1}$	$S = S + \{ A_4 B_{i+1}, A_2 B_{i+1}, A_0 B_{i+1} \}$ $+ \{ A_{15}B_{i+1}, A_{13}B_{i+1}, A_{11}B_{i+1} \} \ll (48 \cdot 11)$

Figure 2.6: Time scheduling of 1056 bit Montgomery multiplication.

product per cycle), they have a two cycle delay due to their internal storage as can be seen from Figure 1. In the 1st cycle, DSP blocks perform 24×16 bit multiplications and store the results in their internal registers. These are the partial products of the 48×48 bit multiplications. In the 2nd cycle, the logic blocks add the partial products

Table 2.4: Area and time performances of FPGA implementations.

Mult	Op. size	Area			Cycles	Period ns	extra hardware	
		slice	LUT	FF			DSP48E	BRAM
Ours	528	1285	4039	2547	91	4.80	41	
	1056	2265	6217	4968	179	6.10	81	
[24]	512	208	N/A	N/A	1056	3.33		4
	1024	2636	N/A	N/A	272	8.67		8
	512	10	N/A	N/A	1056	N/A	4	4
	1024	219	N/A	N/A	272	N/A	33	8
[9]	512	1513	4650	4914	513	2.04		
	1024	2981	9297	9564	1025	2.26		
[11]	512	1503	4649	6863	516	1.68		
	1024	3078	9277	13570	1028	2.04		
[18]	512	1335	4608	3739	266	2.19		
		3074	11541	3903	134	3.08		
		5655	20795	3931	67	6.27		
	1024	2504	9304	7492	530	2.23		
		5836	22625	7533	264	3.83		
		9677	34904	7427	132	6.88		

with their fast carry chains and store the result of the 48×48 bit multiplication in their flip flops.

2.4. Performance Comparisons

The proposed Montgomery multiplier is coded in Verilog HDL. The design has been synthesized and implemented on Virtex 7 xc7vx330t with -3 speed grade. Xilinx Vivado 2016.4 tool with default optimization strategy is used for development. The correctness of the Verilog implementation is tested by comparing the results of the Vivado simulator with the results of a software implementation of the Montgomery multiplication.

Table 2.4 gives the FPGA implementation results of the proposed Montgomery multiplier and some other Montgomery multipliers in the literature. These designs are compared in terms of FPGA logic slices (area) and computation time. Additional hardware resources used by the designs such as DSP blocks (DSP48E) and block RAMs (BRAM) are also given in the table. Moreover, the works in [9], [11] implemented their Montgomery multipliers on Virtex 2. For a fair comparison, Table

2.4 gives the Virtex 7 reimplementation results of these Montgomery multipliers found in [18].

As seen, the shortest computation times belong to the proposed design and [18]. Though the design in [18] has slightly better timings for the digit sizes $\delta = 4$ and $\delta = 8$, the timings are very close. Actually, the proposed design saves a large number of logic slices at the expense of using some number of DSP slices. In the case of 512 bit Montgomery multiplication, the proposed design uses 1789 fewer logic slices than [18] for $\delta = 4$ and 4370 fewer logic slices than [18] for $\delta = 8$ at the expense of 41 DSP48E slices. In the case of 1024 bit Montgomery multiplication, the proposed design uses 3571 fewer logic slices than [18] for $\delta = 4$ and 7412 fewer logic slices than [18] for $\delta = 8$ at the expense of 81 DSP48E slices.

Also, the proposed work has the smallest area-time product in the case that block RAMs (BRAMs) are not used as additional resources. On the other hand, keeping the input and output data on the BRAMs greatly reduces the logic slice usage, and thus the area-time product as seen from the performance of [24] in Table 2.4.

The Montgomery multiplier in [24] places the two multiplicands, the divisor θ , and the accumulated sum in the BRAMs of the FPGA to decrease the logic slice usage. Unfortunately, using BRAMS has some drawbacks. The port width of a BRAM can not exceed 36 bits in Xilinx FPGAs. However, all the operands have to be fed into the digit multipliers δ bit at a time. Thus, each operand must be placed in a different BRAM when the digit size $\delta = 32$ and two different BRAMs when the digit size $\delta = 64$.

2.5. Conclusion

This paper presents a fast and efficient Montgomery multiplier developed for FPGAs with DSP resources. The proposed multiplier is developed for 528 bit Montgomery multiplication but it can be extended to perform 528ℓ bit Montgomery multiplication by combining ℓ different 528 bit Montgomery multipliers.

3ℓ different digit multipliers are used for 528ℓ bit Montgomery multiplication. Digit multipliers in the proposed design perform 48×48 bit multiplications. These digit multipliers can easily be built with the DSP slices found in modern FPGAs. In

this work, they are built with the DSP48E slices found in Virtex-5 and higher Xilinx FPGAs.

Also, a $528\ell + 48$ bit carry select adder is used to accumulate the outputs of the digit multipliers in each clock cycle. In this work, carry select adder is built from the DSP48E slices performing 48 bit additions.

In our design, one multiplicand is multiplied by the digits of the other multiplicand in digit serial fashion. Thus, in each iteration, one multiplicand is multiplied with a single digit. Then, the result is accumulated and reduced by Montgomery technique. 3ℓ digit multipliers compute all the products needed in an iteration in eight cycles.

Performance results obtained from Xilinx Vivado suite for Virtex 7 show that the proposed Montgomery multiplier is one of the fastest designs in the literature. Moreover, the proposed design leads to a considerable reduction in the required FPGA logic resources.

3. FULL-WORD BARRETT MULTIPLIER

3.1. Introduction

The Public key cryptosystems such as Diffie-Hellman [1], RSA [2], and elliptic curve cryptography [3] are widely used in communication systems. Modular multiplication with large integers increases the computational complexity in these cryptosystems. Straightforward modular multiplication requires division, which is a very costly operation. There are two widely known algorithms in the literature to eliminate the division operation. Montgomery algorithm [5] and Barrett algorithm [27] trade the division operation for a multiplication with a precomputed value which depends on the modulus.

Modular multiplication using Montgomery algorithm is called as Montgomery multiplication. Montgomery multipliers are abundant in literature with varying digit sizes (δ). Bit-serial architectures [12] - [15] process the operand bits, $\delta = 1$ bit at a time. Although these architectures have low area complexity, they require larger clock cycles to perform one single modular multiplication. Digit based ($\delta > 1$) Montgomery multipliers are also proposed in the literature to enhance performance at the expense of increased area [17] - [19]. Recent Montgomery multipliers take the advantage of DSP slices of FPGAs. These dedicated hardware blocks are abundant in FPGAs and can perform advanced arithmetic operations [24], [28], [29].

Montgomery multipliers are more widely used in cryptosystems since their efficient implementation and high performance in hardware platforms. Modular multipliers using Barrett algorithm can also show equal performance and efficiency. Also, Barrett algorithm works with real numbers and does not require any domain transformation. There are some Barrett modular multipliers available in the literature [17], [30]. Barrett multiplier with digit size, $\delta = 32$ is presented in [17]. Specific modulus sets are used to get rid of some multiplication steps and enhance performance. The design is implemented for ASIC platforms. In [30], Barrett multiplier is proposed for at most 128 bit operands and the multiplier is designed using only LUTs.

This chapter presents a full-word Barrett multiplier which shows comparable performance with the latest Montgomery multipliers. The proposed multiplier is

designed to be implemented in Virtex-5 and higher Xilinx FPGA series, which include the DSP48E slices. These slices contain 25x18 bit dedicated multipliers. The proposed design has several 48x48 bit multiplier blocks built from the DSP48E slices. The proposed design is developed for 528 bit Barrett multiplication primarily. It can be extended easily to the higher bits by combining 528 bit Barrett multipliers.

The proposed design works iteratively. In each iteration, 48 bit of multiplicand b , is multiplied by the other multiplicand a . For a 528 bit of a , eleven 48x48 bit multiplications are needed to complete one iteration. To compute one 528x528 multiplication requires eleven iterations, making a total of 121 partial products. These partial products can be computed using six 48x48 bit multipliers in twenty two clock cycles such that partial products computed in each cycle do not overlap. Thus, they can be accumulated separately. Then the final product is reduced using Barrett method.

Also, carry select adder is built from the DSP48E slices to accumulate the partial products without having large carry chain delays.

This paper is organized as follows: Section 3.2 introduces Barrett reduction technique. Section 3.3 presents the proposed Barrett multiplier. Section 3.4 gives the implementation results for 528 and 1056 bit Barrett multiplications and comparisons with other designs in the literature. Section 3.5 discusses the conclusion.

3.2. Barrett Reduction

Let $\theta > 1$ be an integer. The modular reduction $r = u \bmod \theta$ of an integer u can be computed as follows:

$$q = \lfloor u/\theta \rfloor \quad r = u - q\theta \quad (3.1)$$

Choose two different integers k, l such that $2^k > 1$, $2^l > \theta$

$$q = \left\lfloor \frac{u}{\theta} \right\rfloor = \left\lfloor \frac{\left(\frac{u}{2^k}\right) \left(\frac{2^l}{\theta}\right)}{2^{l-k}} \right\rfloor = \left\lfloor \frac{v\psi}{2^{l-k}} \right\rfloor \quad (3.2)$$

where $v = (u/2^k)$ and $\psi = (2^l/\theta)$. If ψ is precomputed, it is easier to obtain q , because costly division operation is omitted and the computation only involves multiplication by a precomputation and shifting.

The boundaries of integer u and modulus θ are as follows:

$$0 \leq u < 2^{n+\delta}, \quad 2^{n-1} < \theta < 2^n \quad (3.3)$$

v and ψ values are bounded as follows:

$$v = \frac{u}{2^k} < 2^{n+\delta-k}, \quad \psi = \frac{2^l}{\theta} < 2^{l-n+1} \quad (3.4)$$

The integer and fractional parts of v and ψ are denoted as follows:

$$\begin{aligned} v^{(I)} &= \lfloor u/2^k \rfloor, & v^{(F)} &< 1 \\ \psi^{(I)} &= \lfloor 2^l/\theta \rfloor, & \psi^{(F)} &< 1 \end{aligned} \quad (3.5)$$

Then q can be computed as follows:

$$q = \left\lfloor \frac{v\psi}{2^{l-k}} \right\rfloor = \left\lfloor \frac{v^{(I)}\psi^{(I)}}{2^{l-k}} + \frac{v^{(I)}\psi^{(F)}}{2^{l-k}} + \frac{v^{(F)}\psi}{2^{l-k}} \right\rfloor \quad (3.6)$$

Computation with fractional parts are difficult to handle. Thus only integer parts of v and ψ are included in Barrett reduction. q is estimated as follows:

$$q^{est} = \left\lfloor \frac{\lfloor u/2^k \rfloor \lfloor 2^l/\theta \rfloor}{2^{l-k}} \right\rfloor = \left\lfloor \frac{v^{(I)}\psi^{(I)}}{2^{l-k}} \right\rfloor \quad (3.7)$$

Let $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ be the estimation errors where,

$$\begin{aligned}
\mathcal{E}_1 &= \frac{v^{(I)}\psi^{(F)}}{2^{l-k}} < 2^{n+\delta-l} \\
\mathcal{E}_2 &= \frac{v^{(F)}\psi}{2^{l-k}} < 2^{k-n+1}, \\
\mathcal{E}_3 &= \frac{v^{(I)}\psi^{(I)}}{2^{l-k}} - \left\lfloor \frac{v^{(I)}\psi^{(I)}}{2^{l-k}} \right\rfloor < 1
\end{aligned} \tag{3.8}$$

and q can be represented as follows:

$$\begin{aligned}
q &= \left\lfloor \left\lfloor \frac{v^{(I)}\psi^{(I)}}{2^{l-k}} \right\rfloor + \mathcal{E}_1 + \mathcal{E}_2 + \mathcal{E}_3 \right\rfloor \\
q &= q^{est} + \lfloor \mathcal{E}_1 + \mathcal{E}_2 + \mathcal{E}_3 \rfloor
\end{aligned} \tag{3.9}$$

Then total error is bounded as follows:

$$\begin{aligned}
q - q^{est} &= \lfloor \mathcal{E}_1 + \mathcal{E}_2 + \mathcal{E}_3 \rfloor \\
q - q^{est} &< 2^{n+\delta-l} + 2^{k-n+1} + 1
\end{aligned} \tag{3.10}$$

When $k = n - 2$ and $l = n + \delta + 1$ is chosen, it is guaranteed that the estimation error will be $q - q^{est} \leq 1$ and the integer parts will be as follows:

$$\begin{aligned}
v^{(I)} &= \lfloor u/2^k \rfloor = \lfloor u/2^{n-2} \rfloor < 2^{\delta+2}, \\
\psi^{(I)} &= \lfloor 2^l/\theta \rfloor = \lfloor 2^{n+\delta+1}/\theta \rfloor < 2^{\delta+2}
\end{aligned} \tag{3.11}$$

Also, the quotient estimation in (3.7) will be,

$$q^{est} = \left\lfloor \frac{v^{(I)}\psi^{(I)}}{2^{l-k}} \right\rfloor = \left\lfloor \frac{v^{(I)}\psi^{(I)}}{2^{\delta+3}} \right\rfloor < 2^{\delta+1} \tag{3.12}$$

When $k < n - 2$ and $l > n + \delta + 1$ is chosen, it is still guaranteed that the estimation error will be $q - q^{est} \leq 1$ but the product required for the estimation,

$$v^{(l)}\psi^{(l)} = \lfloor u/2^k \rfloor \lfloor 2^l/\theta \rfloor \quad (3.13)$$

gets larger. Note that, $r = u \bmod \theta = u - q\theta$ can be estimated as $r^{est} = u - q^{est}\theta$ from q^{est} in (3.12). Because $q - q^{est} \leq 1$, a final subtraction may be needed to compute $r = u \bmod \theta$ as follows:

$$\begin{aligned} r &= r^{est}, & \text{if } r^{est} - \theta < 0 \\ r &= r^{est} - \theta, & \text{if } r^{est} - \theta \geq 0 \end{aligned} \quad (3.14)$$

Table 3.1 implements full-word Barrett multiplication where $k = n$, $l = 2n$ and $\delta = n$. Operand a is multiplied by the w bits of operand b iteratively and partial products are accumulated to obtain full-word product, Acc in Step 3. Then, overflowed n bits, q_1 in Step 5 is multiplied by the precomputed value ψ in Step 6. Upper n bits of this product, q_3 is multiplied by the modulus θ in Step 8. Resulting product, q_4 is subtracted from the full-word product, Acc to obtain the reduced value, r in Step 9. Final subtractions are needed because of the estimation error in (3.10). For Table 3.1, total error is bounded as follows:

$$\begin{aligned} q - q^{est} &< 2^{n+n-2n} + 2^{n-n+1} + 1 \\ q - q^{est} &< 4 \end{aligned} \quad (3.15)$$

That is, at most four subtractions are enough to keep the final result below modulus, θ for Table 3.1.

3.3. Proposed Barrett Multiplier

The proposed design implements the full-word Barrett multiplication described in Algorithm 1 by using the DSP48E slices of Virtex-5 and higher Xilinx FPGAs.

The proposed design is developed for 528 bit Barrett multiplication primarily. However, it can be extended easily to perform 1056 bit Barrett multiplication by combining two 528 bit Barrett multipliers.

Table 3.1: Full-word Barrett multiplication.

Input: a, b, θ where $a < 2^n$, $b < 2^n$, $2^{n-1} < \theta < 2^n$	
Output: $r = ab \bmod \theta$	
Precomputation: $\psi = \lfloor 2^{2n}/\theta \rfloor$	
function ModMult (a, b, θ, ψ)	
1.	$Acc \leftarrow 0$
2.	for $i \leftarrow 0$ to $n/w - 1$ do
3.	$Acc \leftarrow Acc + (ab_i) \ll (iw)$ for $b_i = (b_{iw+w-1}, \dots, b_{iw})_2$
4.	end for
5.	$q_1 \leftarrow \lfloor Acc \gg k \rfloor$
6.	$q_2 \leftarrow q_1 \psi$
7.	$q_3 \leftarrow \lfloor q_2 \gg (l - k) \rfloor$
8.	$q_4 \leftarrow q_3 \theta$
9.	$r \leftarrow Acc - q_4$
10.	while $r \geq \theta$ do
11.	$r \leftarrow r - \theta$
12.	end while
13.	return r
end function	

In each iteration of Step 3, 48 bit words of operand a , is multiplied by a single 48 bit word of operand b . Therefore, several 48x48 multiplier units are required for the proposed design. These multiplier units are implemented using DSP48E slices. The proposed 528 bit Barrett multiplier has six 48x48 multiplier units. These six multiplier units compute all the full-word products in Step 3, Step 6 and Step 8 of Table 3.1. Accumulation and subtraction operations are implemented using carry select adder built from DSP48E slices.

3.3.1 DSP48E based 48x48 bit Multiplier Unit

48x48 bit Multiplier Unit is described in Section 2.3.1. and Figure 2.1 and Figure 2.2.

3.3.2. General Structure

Figure 3.1 shows the proposed Barrett multiplier. Interconnect Unit consists of several multiplexers to parse the inputs in Table 3.1. For Step 3, 48 bit words of operand a , is multiplied by a single 48 bit word of operand b in each iteration. Each iteration takes two clock cycles. Six 48x48 bit multipliers compute one 528x528 bit multiplication in eleven iterations, making twenty two clock cycles. Partial products computed during this multiplication is properly shifted, concatenated using the shift and concatenate unit and accumulated using carry select adder. The same multiplication, shift and accumulate operations are also carried out for Step 6 and Step 8. Final subtractions in Step 11 are implemented using only carry select adder.

Concatenation is enough to combine partial products due to a clever strategy. In one cycle, odd numbered 48 bit words of operand a , are multiplied by a single 48 bit word of operand b . In the following cycle, even numbered words of a , are multiplied by the single word of b . This strategy prevents the overlapping of the partial products as shown in Table 1. The same procedure applies for the following multiplications in Step 6 and Step 8.

For 1056 bit Barrett multiplication, two 528 bit Barrett multipliers are used. One of the multipliers handles with the lower 528 bits and the other multiplier handles with the upper 528 bits of operands. The partial products for both multipliers are combined using a single 2112 bit carry select adder rather than two separate 1056 bit carry select adders. Shift and concatenate sizes are arranged for 1056 bit operands in this case.

3.3.3. DSP48E based Carry Select Adder/Subtractor

DSP48E based Carry Select Adder/Subtractor is described in Section 2.3.3. and Figure 2.4. Note that for 528 bit full-word Barrett multiplication, 1056 bit carry

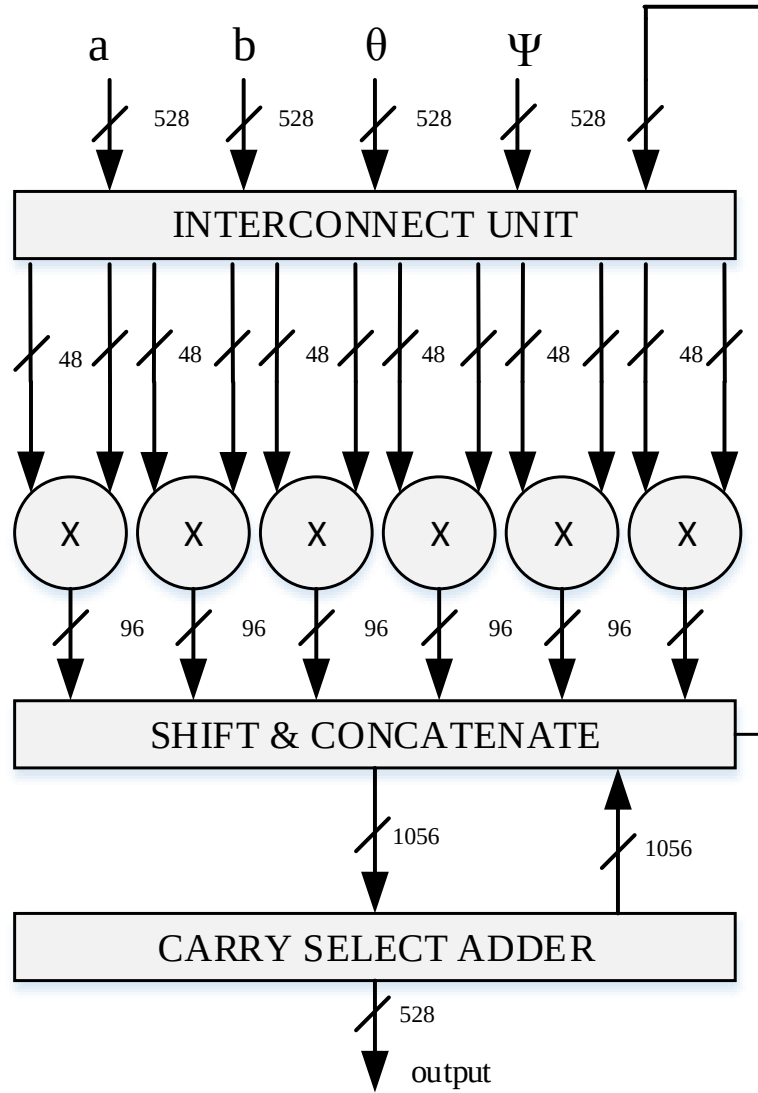


Figure 3.1: Proposed 528 bit Barrett Multiplier.

select adder is required, and for 1056 bit full-word Barrett multiplication, 2112 bit carry select adder is required.

3.3.4. Latency of Barrett Multiplier

The latency of the proposed multiplier is as follows: 79 cycles for the 528 bit Barrett multiplier and 145 cycles for the 1056 bit Barrett multiplier.

Table 3.2 shows the latency of the Barrett multiplier for 528 bit operands. Each iteration takes two clock cycles to complete one ab_i multiplication. For a 528x528 bit multiplication, eleven iterations are needed, which takes a total of twenty two

clock cycles. For 1056x1056 bit multiplication, twenty two iterations are required, which takes forty four clock cycles.

There is a two cycle delay because of the 48x48 bit multipliers as shown in Table 3.2. Although, throughput is one partial product per cycle, two cycles are needed to obtain the corresponding product due to the internal storage of the multipliers as shown in Figure 2.4. Thus, at the end of iterations, there are wait states consisting of two cycles for last partial products and one cycle for last accumulation of these products. Note that, carry select adder takes only one cycle to compute a 1056 bit addition or a 2112 bit addition for 528 bit or 1056 bit Barrett multiplication respectively.

The same iterations and wait states apply for the $q_1\psi$ and $q_3\theta$ multiplications. In the end, final subtractions are needed to keep the result below modulus. According to (3.15), at most four subtractions are enough for Table 3.1, which takes only four cycles.

3.4. Performance Comparisons

The proposed Barrett multiplier is implemented using Verilog HDL. The target device is selected as Virtex 7 xc7vx330t-3 with default optimization strategy. Proposed implementation is verified by comparing the Vivado simulator results with a software which implements the Barrett multiplication algorithm.

Proposed Barrett multiplier is compared with the recent Montgomery multipliers in the literature. The designs are compared in terms of logic slices and computation time as shown in Table 3.3. Additional hardware such as DSP48E slices and BRAM are also given.

Shortest computation times belong to the design in [18]. [18] implements the multiplier using only logic slices. The design in [28] consumes almost the same computation time with [18] and it saves a significant number of logic slices at the expense of some DSP slices. [24] uses lower amount of logic and DSP slices. However, higher number of clock cycles are required to complete one multiplication. The work in [29] uses a huge number of DSP slices. Karatsuba multipliers are used to implement full-word Montgomery multiplication.

Table 3.2: Latency for 528 bit Barrett multiplication.

Operands: $a = (a_{11}, \dots, a_1)_2$, $b = (b_{11}, \dots, b_1)_2$		
Modulus: $\theta = (\theta_{11}, \dots, \theta_1)_2$		
Precomputation: $\psi = (\psi_{11}, \dots, \psi_1)_2$		
State	Multiplication	Shift and Accumulation
for $i \leftarrow 1$ to 11 do		
st0 _i	$a_1 b_i, a_3 b_i, a_5 b_i$ $a_7 b_i, a_9 b_i, a_{11} b_i$	$acc + \leftarrow \{a_{11} b_{i-1}, a_9 b_{i-1}, a_7 b_{i-1}, a_5 b_{i-1}, a_3 b_{i-1}, a_1 b_{i-1}\}$ $\ll (i-1) \times 48$
st1 _i	$a_2 b_i, a_4 b_i, a_6 b_i$ $a_8 b_i, a_{10} b_i, 0$	$acc + \leftarrow \{a_{10} b_{i-1}, a_8 b_{i-1}, a_6 b_{i-1}, a_4 b_{i-1}, a_2 b_{i-1}\}$ $\ll (i) \times 48$
st0 _{i+1}	$a_1 b_{i+1}, a_3 b_{i+1}, a_5 b_{i+1}$ $a_7 b_{i+1}, a_9 b_{i+1}, a_{11} b_{i+1}$	$acc + \leftarrow \{a_{11} b_i, a_9 b_i, a_7 b_i, a_5 b_i, a_3 b_i, a_1 b_i\} \ll (i-1) \times 48$
st1 _{i+1}	$a_2 b_{i+1}, a_4 b_{i+1}, a_6 b_{i+1}$ $a_8 b_{i+1}, a_{10} b_{i+1}, 0$	$acc + \leftarrow \{a_{10} b_i, a_8 b_i, a_6 b_i, a_4 b_i, a_2 b_i\} \ll (i) \times 48$
end for		
wait 3 cc	2 cc last partial product results 1 cc for last accumulation	
	$q_1 \leftarrow acc \gg n$ where $q_1 = (q_{1,11}, \dots, q_{1,1})_2$	
for $i \leftarrow 1$ to 11 do		
st2 _i	$q_{1,1} \psi_i, q_{1,3} \psi_i, q_{1,5} \psi_i$ $q_{1,7} \psi_i, q_{1,9} \psi_i, q_{1,11} \psi_i$	$q_2 + \leftarrow \{q_{1,11} \psi_{i-1}, q_{1,9} \psi_{i-1}, q_{1,7} \psi_{i-1}, q_{1,5} \psi_{i-1}, q_{1,3} \psi_{i-1}, q_{1,1} \psi_{i-1}\}$ $\ll (i-1) \times 48$
st3 _i	$q_{1,2} \psi_i, q_{1,4} \psi_i, q_{1,6} \psi_i$ $q_{1,8} \psi_i, q_{1,10} \psi_i, 0$	$q_2 + \leftarrow \{q_{1,10} \psi_{i-1}, q_{1,8} \psi_{i-1}, q_{1,6} \psi_{i-1}, q_{1,4} \psi_{i-1}, q_{1,2} \psi_{i-1}\}$ $\ll (i) \times 48$
st2 _{i+1}	$q_{1,1} \psi_{i+1}, q_{1,3} \psi_{i+1}, q_{1,5} \psi_{i+1}$ $q_{1,7} \psi_{i+1}, q_{1,9} \psi_{i+1}, q_{1,11} \psi_{i+1}$	$q_2 + \leftarrow \{q_{1,11} \psi_i, q_{1,9} \psi_i, q_{1,7} \psi_i, q_{1,5} \psi_i, q_{1,3} \psi_i, q_{1,1} \psi_i\}$ $\ll (i-1) \times 48$
st3 _{i+1}	$q_{1,2} \psi_{i+1}, q_{1,4} \psi_{i+1}, q_{1,6} \psi_{i+1}$ $q_{1,8} \psi_{i+1}, q_{1,10} \psi_{i+1}, 0$	$q_2 + \leftarrow \{q_{1,10} \psi_i, q_{1,8} \psi_i, q_{1,6} \psi_i, q_{1,4} \psi_i, q_{1,2} \psi_i\} \ll (i) \times 48$
end for		
wait 4 cc	2 cc last partial product results 1 cc for last accumulation 1 cc addition for last bit of ψ .	
	$q_3 \leftarrow \lfloor q_2 \gg n \rfloor$ where $q_3 = (q_{3,11}, \dots, q_{3,1})_2$	
for $i \leftarrow 1$ to 11 do		
st4 _i	$q_{3,1} \theta_i, q_{3,3} \theta_i, q_{3,5} \theta_i$ $q_{3,7} \theta_i, q_{3,9} \theta_i, q_{3,11} \theta_i$	$q_4 + \leftarrow \{q_{3,11} \theta_{i-1}, q_{3,9} \theta_{i-1}, q_{3,7} \theta_{i-1}, q_{3,5} \theta_{i-1}, q_{3,3} \theta_{i-1}, q_{3,1} \theta_{i-1}\}$ $\ll (i-1) \times 48$
st5 _i	$q_{3,2} \theta_i, q_{3,4} \theta_i, q_{3,6} \theta_i$ $q_{3,8} \theta_i, q_{3,10} \theta_i, 0$	$q_4 + \leftarrow \{q_{3,10} \theta_{i-1}, q_{3,8} \theta_{i-1}, q_{3,6} \theta_{i-1}, q_{3,4} \theta_{i-1}, q_{3,2} \theta_{i-1}\}$ $\ll (i) \times 48$
st4 _{i+1}	$q_{3,1} \theta_{i+1}, q_{3,3} \theta_{i+1}, q_{3,5} \theta_{i+1}$ $q_{3,7} \theta_{i+1}, q_{3,9} \theta_{i+1}, q_{3,11} \theta_{i+1}$	$q_4 + \leftarrow \{q_{3,11} \theta_i, q_{3,9} \theta_i, q_{3,7} \theta_i, q_{3,5} \theta_i, q_{3,3} \theta_i, q_{3,1} \theta_i\}$ $\ll (i-1) \times 48$
st5 _{i+1}	$q_{3,2} \theta_{i+1}, q_{3,4} \theta_{i+1}, q_{3,6} \theta_{i+1}$ $q_{3,8} \theta_{i+1}, q_{3,10} \theta_{i+1}, 0$	$q_4 + \leftarrow \{q_{3,10} \theta_i, q_{3,8} \theta_i, q_{3,6} \theta_i, q_{3,4} \theta_i, q_{3,2} \theta_i\} \ll (i) \times 48$
end for		
wait 2 cc	2 cc last partial product results	
sub 1 cc	$r \leftarrow acc - q_4$	
sub 3 cc	$r \leftarrow r - \theta$	

Table 3.3: Area and time performances of FPGA implementations.

Mult	Op. size	Area			Cycles	Period ns	extra hardware	
		slice	LUT	FF			DSP48E	BRAM
Ours	528	3530	12134	4942	79	6.30	79	
	1056	9120	24589	9661	145	13.0	153	
[28]	528	1285	4039	2547	91	4.80	41	
	1056	2265	6217	4968	179	6.10	81	
[24]	512	208	N/A	N/A	1056	3.33		4
	1024	2636	N/A	N/A	272	8.67		8
	512	10	N/A	N/A	1056	N/A	4	4
	1024	219	N/A	N/A	272	N/A	33	8
[29]	512	N/A	23511	11307	29	16.3	560	
		N/A	26519	13249	38	13.8	492	
[18]	512	1335	4608	3739	266	2.19		
		3074	11541	3903	134	3.08		
		5655	20795	3931	67	6.27		
	1024	2504	9304	7492	530	2.23		
		5836	22625	7533	264	3.83		
		9677	34904	7427	132	6.88		

Montgomery multiplication has a big architectural advantage over Barrett multiplication. Montgomery algorithm uses lower bits of ab product for subsequent multiplications with the precomputation. Thus, it allows us to start these multiplication without waiting the completion of ab product. However, Barrett algorithm uses overflowed bits of ab product and it is necessary to wait the completion of this product to start subsequent multiplications with the precomputation. This difference leads to the reduction in computation time for Montgomery multiplication.

The works in [18], [24], [28] are digit based Montgomery multipliers where n bit operand is multiplied by δ bit of the other operand and reduced in each iteration. This also reduces the area complexity and clock frequency. However, digit based design is not very efficient for Barrett multiplication because of the possible wait cycles before each reduction step. To eliminate these wait cycles and increase performance, full-word multiplication is more convenient for Barrett multiplication. However, this method needs larger additions. Thus, it leads to higher area and lower clock frequency as opposed to digit based designs. Nevertheless, proposed design also shows competitive performance for 528 bit modular multiplication when it is compared to digit based multipliers in [18], [24], [28].

The work in [29] implements full-word Montgomery multiplier. Proposed design has almost the same latency and less DSP and logic resources than [29].

3.5. Conclusion

This chapter presents a high performance Barrett multiplier, which takes the advantage of DSP48E slices in modern Xilinx FPGAs. These DSP slices perform fast arithmetic operations required for the multiplication and the addition with large integers. The proposed multiplier is developed for 528 bit Barrett multiplication, but it can be extended to perform 1056 bit Barrett multiplication by combining two different 528 bit Barrett multipliers.

In our design, 48x48 bit multiplier is designed first. Then these multipliers are used to multiply full-word operands. In order to accumulate the partial products during these multiplications, a fast 1056 and 2112 bit carry select adder is used for 528 and 1056 bit operands respectively.

Performance results obtained for Virtex 7 show that the proposed Barrett multiplier can be a strong alternative to the fastest Montgomery multipliers in the literature. Thus, the proposed multiplier can be used in RSA and elliptic curve cryptosystems which require high performance.

4. ELLIPTIC CURVE POINT MULTIPLICATION

4.1. Introduction

Elliptic curve cryptography (ECC) proposed by Miller [4] and Koblitz [3] is widely used in public-key cryptography [1]. Over the years, ECC has gained popularity and shown to be more efficient than RSA [2] based crypto-systems. Compared with RSA, ECC uses shorter keys to achieve the same security level. Therefore ECC hardware architectures are preferred for high-speed, low complexity and low power designs.

Elliptic curve point multiplication (ECPM) operation is the main computation part in ECC hardware architecture. ECPM can be examined in three distinct layers [31]: the modular arithmetic, elliptic curve point addition (PA) and point doubling (PD) and the scalar multiplication. Modular arithmetic, consisting of modular multiplications, modular additions/subtractions and modular inversions, forms the basis of all others. Time-consuming modular inversion can be mostly avoided using projective coordinates such as standard, Jacobian and Chudnovsky projective points. Therefore the most critical operation becomes the modular multiplication. The most common algorithm chosen to be suitable for modular multiplication is Montgomery algorithm [5].

Hardware implementations of Montgomery algorithm are abundant in literature. Most of them use classical techniques, such as systolic array multiplication, bit-serial multiplication, digit-serial multiplication. Although bit-serial multipliers [6] - [15] (digit size $\delta = 1$) have low area complexity, they require huge amount of clock cycles to finish one multiplication. Digit-serial multipliers [13], [16] - [19] (digit size $\delta > 1$) are used to obtain speed improvements at the expense of increased area. Although these implementations fit well into CLBs of FPGAs, hardware complexity grows quickly with larger sizes of digit and modulus. Modern FPGAs have a significant amount of DSP slices, which perform arithmetic operations. Each DSP slice is capable of at most 36x36 bit multiplication. Therefore these DSP slices can be utilized to build larger modular multipliers. Montgomery multiplier designs using DSP slices are also available in literature. In [20] - [22], a single DSP slice is used to calculate intermediate products. In [23] - [25], several

DSP slices are used to build digit multipliers. In [24], digit multipliers with digit sizes $\delta = 32, 64$ are proposed. In [24], BRAMs of FPGA are utilized to keep intermediate results and multipliers with digit sizes $\delta = 16, 32, 64$ are proposed. In [25] 256x256 digit multipliers are built using Xilinx IP Core generator.

ECC hardware implementations over prime fields are available in the literature [23], [32] - [36]. In [32], an ultra-high performance ECC hardware on Virtex-4 FPGA is proposed for NIST [37] primes P-224 and P-256. The proposed design makes an intensive use of DSP slices. The work in [33] proposes an area-efficient design for P-224 and P-256 without any use of DSP slice. In [34], data is transferred into atomic blocks to resist side-channel attacks. It is proposed that atomic blocks make Point Addition and Point Doubling operations indistinguishable. In [35], two 31x17 BRAMs are utilized to keep to data. However, it takes many clock cycles to fetch operands. They adjust DSP slices in multiply-and-accumulate fashion in order to avoid using additional logic resources. The work in [36], uses NAF scalar multiplication. The modular inversion is implemented separately. Thus inversion module consumes too much area. In [23], a flexible ECC processor is proposed for arbitrary prime fields using Jacobian projective coordinates. Digit multipliers are built from DSP slices with digit sizes $\delta = 32, 64$. Data is stored in distributed RAMs.

This chapter presents a high performance Montgomery multiplier based ECPM hardware to be implemented in Xilinx Virtex-5 and higher series, which have DSP48E slices containing 25x18 multipliers. First of all, a fast digit-based Montgomery multiplier using DSP48E slices is designed with digit size $\delta = 48$. The proposed multiplier is capable of maximum 528x528 bit multiplication. Also, a fast adder circuit is needed to sum large partial products. In order to get rid of long carry chains, carry select adder is built using DSP48E slices. Each DSP48E slice achieves 48 bit addition or subtraction.

Point Addition (PA) and Point Doubling (PD) operations are implemented using finite state machines (FSM). In order to avoid time-consuming modular inverse operation in affine coordinates, Jacobian projective coordinates are selected. To store large partial values in PA and PD steps, true dual port RAMB36E slices are used. Storing these partial values in CLBs increases the interconnect delays in FPGA fabric. In order to improve the speed of the design and reduce the critical timing paths, BRAMs are utilized.

Scalar multiplication uses PA and PD state machines. PA takes almost two times more time than PD. To obtain speed enhancement, windowing method is utilized to decrease the total amount of PA operation. Selecting window size $w = 3$, total amount of PA operation is reduced to $1/3$ at the expense of some pre-computation values stored in BRAMs.

This paper is organized as follows: Section 4.2 gives some background information on ECC and Montgomery multiplication. Section 4.3 presents the designed hardware modules for ECPM architecture. This consists of Montgomery modular multiplication circuit, carry-select adder, PA and PD state machines, point multiplication, windowing method and top-level architecture. Section 4.4 proposes Xilinx Virtex-7 implementation results for ECPM over both general primes and NIST (p-521) curve and comparison with other state-of-the-art designs. Section 4.5 discusses the conclusion.

4.2. Background Information

In this section, a brief background information about elliptic curve over prime fields and modular arithmetic is mentioned.

4.2.1. Elliptic Curves

An elliptic curve over prime field $GF(p)$ can be described as the set of solution which satisfy the following short Weierstrass equation:

$$y^2 = x^3 + ax + b \mod p \quad \text{where } p > 3 \text{ and } 4a^3 + 27b^2 \neq 0 \quad (4.1)$$

a , b , and p coefficients specify the elliptic curve. Although there are many elliptic curves in short Weierstrass form, the most widely used curves are proposed by NIST, bit-lengths of operands ranging from 192 to 521. Security levels increase with the higher bit-lengths. This chapter focuses on both $GF(p)$ curves with general primes and the p-521 curve proposed by NIST.

There are two point operations that can be calculated on a curve. PD and PA are defined in (4.2) and (4.3) respectively.

$$\begin{aligned}
P_3(x_3, y_3) &= 2 \cdot P_1(x_1, y_1) \\
x_3 &= \lambda^2 - 2x_1 \pmod{p} \\
y_3 &= \lambda(x_1 - x_3) - y_1 \pmod{p} \\
\text{where } \lambda &= \frac{3x_1^2 + a}{2y_1} \pmod{p}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
P_3(x_3, y_3) &= P_1(x_1, y_1) + P_2(x_2, y_2) \\
x_3 &= \lambda^2 - x_1 - x_2 \pmod{p} \\
y_3 &= \lambda(x_1 - x_3) - y_1 \pmod{p} \\
\text{where } \lambda &= \frac{y_2 - y_1}{x_2 - x_1} \pmod{p}
\end{aligned} \tag{4.3}$$

PA and PD operations can be seen geometrically in Figure 4.1 [31].

Both PD and PA operations in affine coordinates require modular inversion which is an expensive operation. In order to avoid this inversion operation, affine coordinates are transformed into projective coordinates. Some common projective coordinates are as follows. *Standard projective* points (x, y, z) correspond to affine points (xz^{-1}, yz^{-1}) where $z \neq 0$. *Jacobian projective* points (x, y, z) , correspond to affine points (xz^{-2}, yz^{-3}) where $z \neq 0$. *Chudnovsky projective* points (x, y, z, u, v) correspond to *Jacobian* points (x, y, z) with two additional coordinates $(u=z^2, v=z^3)$. Converting projective to affine points require one modular inversion and some modular multiplications. However, the cost of this inversion is affordable when a huge number of PA and PD operations are carried out in projective coordinates without any inversion. Our work uses both *Jacobian projective* and *mixed Jacobian-affine* coordinates points.

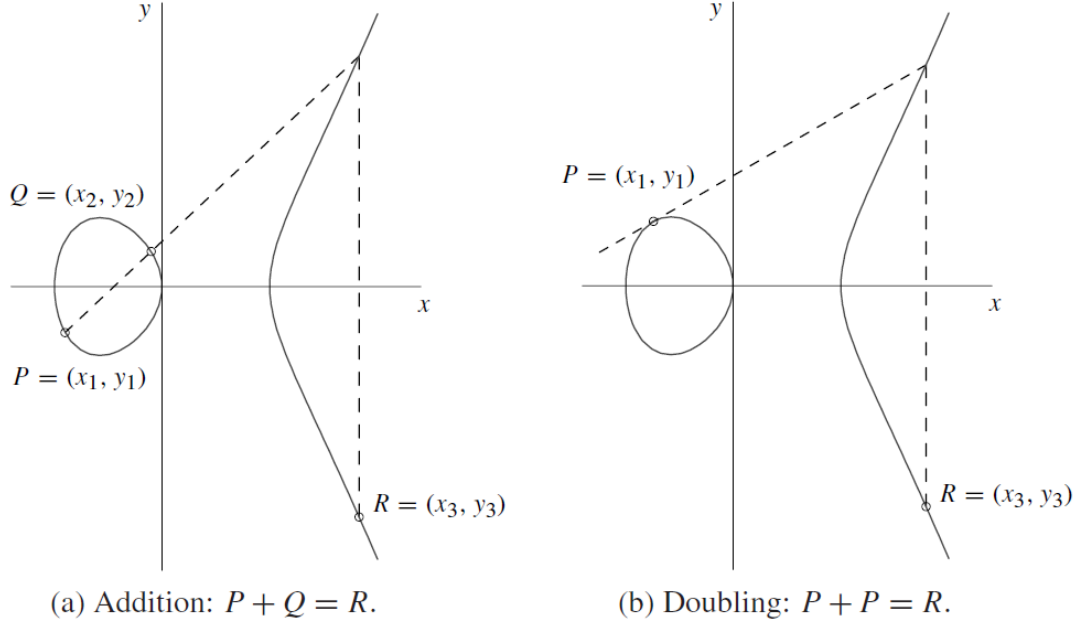


Figure 4.1: Point Addition and Point Doubling of elliptic curve points.

4.2.2. Montgomery Multiplication

Montgomery multiplication algorithm and implementation details are given in Chapter 2. Proposed elliptic curve hardware is based on the Montgomery multiplication implemented in Chapter 2.

4.3. Proposed Elliptic Curve Point Multiplication Hardware

The proposed design implements the Elliptic Curve Point Multiplication (ECPM) algorithm employing DSP48E and RAMB36E slices of Virtex-5 and higher Xilinx FPGAs.

4.3.1. Point Doubling and Point Addition Scheduling

The layer of Point Doubling and Point Addition operations is implemented using first Jacobian projective coordinates. Table 4.1 shows the each step of PD and PA in Jacobian coordinates. In Jacobian coordinates, PD requires 10 modular multiplication, whereas PA requires 16 modular multiplication without any inversion. When $a = -3$ is selected in short Weierstrass equation, 2 modular

Table 4.1: Point Doubling and Point Addition with jacobian coordinates.

Point Doubling(a = -3)	Point Addition
Input : X, Y, Z	Input : X ₁ , Y ₁ , Z ₁ , X ₂ , Y ₂ , Z ₂
Output : X ₃ , Y ₃ , Z ₃ = 2 · (X, Y, Z)	Output : X ₃ , Y ₃ , Z ₃ = (X ₁ , Y ₁ , Z ₁) + (X ₂ , Y ₂ , Z ₂)
function PD(X, Y, Z, X ₃ , Y ₃ , Z ₃)	funcction PA(X ₁ , Y ₁ , Z ₁ , X ₂ , Y ₂ , Z ₂ , X ₃ , Y ₃ , Z ₃)
1. T ₅ ← Y · Y	1. T ₁ ← Z ₁ · Z ₁
2. T ₁ ← X · T ₅	2. T ₂ ← T ₁ · Z ₁
3. T ₁ ← T ₁ + T ₁	3. T ₃ ← Z ₂ · Z ₂
4. T ₁ ← T ₁ + T ₁	4. T ₄ ← T ₃ · Z ₂
5. T ₂ ← Z · Z	5. T ₃ ← X ₁ · T ₃
6. T ₃ ← X + T ₂	6. T ₁ ← X ₂ · T ₁
7. T ₄ ← X - T ₂	7. T ₄ ← Y ₁ · T ₄
8. T ₂ ← T ₃ · T ₄	8. T ₂ ← Y ₂ · T ₂
9. T ₃ ← T ₂ + T ₂	9. T ₁ ← T ₁ - T ₃
10. T ₂ ← T ₂ + T ₃	10. T ₂ ← T ₂ - T ₄
11. T ₃ ← T ₂ · T ₂	11. T ₅ ← T ₁ · T ₁
12. T ₄ ← T ₁ + T ₁	12. T ₆ ← T ₅ · T ₁
13. X ₃ ← T ₃ - T ₄	13. T ₇ ← T ₂ · T ₂
14. T ₃ ← T ₁ - X ₃	14. T ₅ ← T ₃ · T ₅
15. T ₃ ← T ₂ · T ₃	15. T ₃ ← T ₅ + T ₅
16. T ₄ ← T ₅ · T ₅	16. X ₃ ← T ₇ - T ₆
17. T ₄ ← T ₄ + T ₄	17. X ₃ ← X ₃ - T ₃
18. T ₄ ← T ₄ + T ₄	18. T ₅ ← T ₅ - X ₃
19. T ₄ ← T ₄ + T ₄	19. T ₅ ← T ₅ · T ₂
20. Y ₃ ← T ₃ - T ₄	20. T ₆ ← T ₆ · T ₄
21. T ₃ ← Y · Z	21. Y ₃ ← T ₅ - T ₆
22. Z ₃ ← T ₃ + T ₃	22. Z ₃ ← Z ₁ · Z ₂
23. return X ₃ , Y ₃ , Z ₃	23. Z ₃ ← Z ₃ · T ₁
end function	24. return X ₃ , Y ₃ , Z ₃
	end function

Table 4.2: Point Doubling and Point Addition with mixed jacobian/affine coordinates.

Point Doubling(a = -3)	Point Addition with Mixed Addition
Input : X, Y, Z	Input : X ₁ , Y ₁ , Z ₁ , X ₂ , Y ₂ , Z ₂
Output : X ₃ , Y ₃ , Z ₃ = 2 · (X, Y, Z)	Output : X ₃ , Y ₃ , Z ₃ = (X ₁ , Y ₁ , Z ₁) + (X ₂ , Y ₂ , Z ₂)
function PD(X, Y, Z, X ₃ , Y ₃ , Z ₃)	function PA(X ₁ , Y ₁ , Z ₁ , X ₂ , Y ₂ , Z ₂ , X ₃ , Y ₃ , Z ₃)
1. T ₅ ← Y · Y	1. T ₁ ← Z ₁ · Z ₁
2. T ₁ ← X · T ₅	2. T ₂ ← T ₁ · Z ₁
3. T ₁ ← T ₁ + T ₁	3. T ₁ ← X ₂ · T ₁
4. T ₁ ← T ₁ + T ₁	4. T ₂ ← Y ₂ · T ₂
5. T ₂ ← Z · Z	5. T ₁ ← T ₁ - X ₁
6. T ₃ ← X + T ₂	6. T ₂ ← T ₂ - Y ₁
7. T ₄ ← X - T ₂	7. T ₅ ← T ₁ · T ₁
8. T ₂ ← T ₃ · T ₄	8. T ₆ ← T ₅ · T ₁
9. T ₃ ← T ₂ + T ₂	9. T ₇ ← T ₂ · T ₂
10. T ₂ ← T ₂ + T ₃	10. T ₅ ← X ₁ · T ₅
11. T ₃ ← T ₂ · T ₂	11. T ₃ ← T ₅ + T ₅
12. T ₄ ← T ₁ + T ₁	12. X ₃ ← T ₇ - T ₆
13. X ₃ ← T ₃ - T ₄	13. X ₃ ← X ₃ - T ₃
14. T ₃ ← T ₁ - X ₃	14. T ₅ ← T ₅ - X ₃
15. T ₃ ← T ₂ · T ₃	15. T ₅ ← T ₅ · T ₂
16. T ₄ ← T ₅ · T ₅	16. T ₆ ← T ₆ · Y ₁
17. T ₄ ← T ₄ + T ₄	17. Y ₃ ← T ₅ - T ₆
18. T ₄ ← T ₄ + T ₄	18. Z ₃ ← Z ₁ · T ₁
19. T ₄ ← T ₄ + T ₄	19. return X ₃ , Y ₃ , Z ₃
20. Y ₃ ← T ₃ - T ₄	end function
21. T ₃ ← Y · Z	
22. Z ₃ ← T ₃ + T ₃	
23. return X ₃ , Y ₃ , Z ₃	
end function	

multiplications are saved in PD steps. Thus, PD requires 8 modular multiplication as shown in Table 4.1. Note that, modular addition/subtraction operations can be realized using fast CSLA circuit explained in Section 2.3.3 and take only 1 clock cycle to obtain result. Thus, modular multiplication becomes the most time-consuming operation.

When *mixed Jacobian-affine* coordinate is chosen as shown in Table 4.2, 5 modular multiplications are saved for Point Addition when it is compared to *Jacobian* coordinates in Table 4.1.

T_i parameters represent the temporary values computed in each step of PD and PA operations. In order to store these temporary values, true dual port BRAMs are used. BRAMs are special blocks implemented in FPGA fabric to store data. Storing T_i values in CLBs requires a large number of slice register usage. Although modern FPGAs have thousands of slice registers, routing delays increase with the high register usage.

BRAMs are customized using Xilinx IP Generator tool provided by Vivado Design Suite 2016.4. Since each RAMB36E slice stores 36 bit data, multiples of RAMB36E slices are concatenated to each other to hold large input operands and temporary T_i values. Generated BRAM core consumes 15 RAMB36E slices. To increase performance two generated BRAM cores are instantiated in the proposed design. First operand in Table 4.2 comes from first BRAM core and the second operand in Table 4.2 comes from second BRAM core. This allows the operands to be fetched from two generated BRAM cores at the same time.

4.3.2. Point Multiplication

The layer of point multiplication or scalar multiplication is implemented using PA and PD state machines described in Section 4.3.1. There are many methods for point multiplication proposed in literature. Some of them are as follows: *double-and-add method*, *Montgomery ladder method* and *window method*.

Although *double-and-add method* is the simplest one and requires less hardware usage, its operation time depends on the number of 1 bits in scalar. Thus, this method is vulnerable to timing-based side channel attacks. *Montgomery ladder method* computes point multiplication in constant time. Therefore it is better than

double-and-add method. However, Montgomery ladder method, needs to compute PA operation for each bit in scalar. PA operation takes approximately two times longer than PD.

In order to decrease total number of PA counts in point multiplication, *window method* is utilized in the design at the expense of some pre-computation as shown in Table 4.3. This pre-computation is affordable when a large scalar is taken into consideration.

Window size is selected as $w=3$ as shown in Table 4.3. Six pre-computed values described by *buffer* in Table 4.3, have to be stored for further usage. This pre-computed values are stored in two generated BRAM cores. By choosing $w=3$, total PA counts decreases to $1/3$ of total PA counts in *Montgomery ladder method*. Although choosing larger window sizes results in further decrease in PA counts, the pre-computation time increases and larger storage area is also required. Thus $w=3$ is suitable for the proposed design.

The overall architecture implementing ECPM algorithm using three distinct layers is shown in Figure 4.2. k register holds the scalar input. One data and one address ports of each true dual port BRAMs are exported to top module for external read/write purposes. Finite state machines (FSM) of layer 1, 2, 3 in Figure 4.2 are responsible for the modular multiplication/addition; PA, PD, and inversion; window

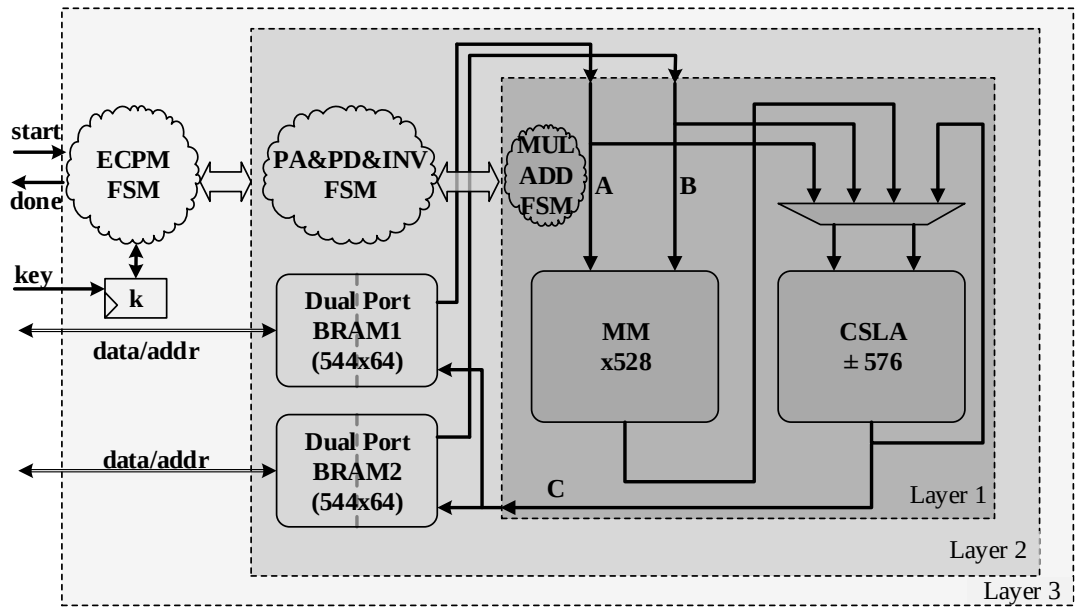


Figure 4.2: Overall elliptic curve point multiplication architecture.

Table 4.3: ECPM using window method ($w=3$).

Input: $(P_x, P_y, P_z), k = k_{u-1, \dots, 0}$
Output: $(T_x, T_y, T_z) = k \cdot (P_x, P_y, P_z)$
function windowed ($k, (P_x, P_y, P_z), (T_x, T_y, T_z)$)
1. for $i \leftarrow 1$ to 2^w-1 do
2. $\text{buffer}[i] \leftarrow i \cdot (P_x, P_y, P_z)$
3. end for
4. $(T_x, T_y, T_z) \leftarrow \text{buffer}[k_{u-1, u-2, u-3}]$
5. for $j \leftarrow u/w-2$ downto 0 do
6. $k \leftarrow k \gg w$
7. $(T_x, T_y, T_z) \leftarrow \text{PD}((T_x, T_y, T_z))$
8. $(T_x, T_y, T_z) \leftarrow \text{PD}((T_x, T_y, T_z))$
9. $(T_x, T_y, T_z) \leftarrow \text{PD}((T_x, T_y, T_z))$
10. $(T_x, T_y, T_z) \leftarrow \text{PA}((T_x, T_y, T_z), \text{buffer}[k_{u-1, u-2, u-3}])$
11. end for
12. return (T_x, T_y, T_z)
end function

technique based ECPM and pre-computation respectively. Each FSM is connected to each other by control signals.

4.4. Implementation Results

4.4.1. Hardware Implementation

The proposed ECPM architecture is implemented in Verilog HDL. The implementation is verified with Xilinx Vivado 2016.4 RTL simulations. RTL simulation results matched the results of a software model of ECPM algorithm implemented in Python language and verified with test vectors given by NIST. The proposed hardware is implemented for Virtex-7 device xc7vx330tffg1157-3 with default optimization strategy.

Table 4.4: Clock cycles for 521 bit ECPM in jacobian coordinates.

Operation	Clock cycles
528 bit Montgomery multiplication	91
Carry-Select addition/subtraction	1
Point Doubling	876
Point Addition	1603
Pre-computation	7437
Inversion (Fermat's Theorem)	101658
Point Multiplication (w=3)	845331

Table 4.5: Clock cycles for 521 bit ECPM in mixed jacobian-affine coordinates.

Operation	Clock cycles
528 bit Montgomery multiplication	91
Carry-Select addition/subtraction	1
Point Doubling	876
Point Addition	1117
Inversion (Fermat's Theorem)	101658
Point Multiplication (w=3)	753468

Table 4.6: Clock cycles for NIST P-521 ECPM in mixed jacobian-affine coordinates.

Operation	Clock cycles
Modular multiplication with fast reduction	58
Carry-Select addition/subtraction	1
Point Doubling	611
Point Addition	754
Inversion (Fermat's Theorem)	67074
Point Multiplication (w=3)	517914

Table 4.7: Fast reduction with P-521.

Input: $c = (c_{1041}, \dots, c_2, c_1, c_0)_2$
Output: $: c \bmod p_{521}$
function moduloReduction
1. $t_1 = (c_{1041}, \dots, c_{523}, c_{522}, c_{521})_2$
2. $t_2 = (c_{520}, \dots, c_2, c_1, c_0)_2$
3. return $(t_1 + t_2) \bmod_{521}$
end function

512 bit key length is chosen for Elliptic curve prime fields. The number of clock cycles for *Jacobian* coordinate operations in three distinct layers are shown in Table 4.4. The number of clock cycles for mixed Jacobian/affine coordinates are given in Table 4.5. Note that Point Addition is 43%, and Point multiplication is 12% faster than the case in Table 4.4.

For fast modular reduction schemes, special primes called mersenne primes are chosen by NIST. Fast modulo reduction for p-521 prime is given in Table 4.7. This reduction only needs addition and subtraction. No additional multiplication with the precomputation is required.

Table 4.6 gives clock cycles for NIST P-521 curve. Note that, Point multiplication with NIST implementation in Table 4.6 is 45% faster than the case in Table 4.5.

For window size $w=3$, 173 PA and 519 PD operations are required to complete one point multiplication. Pre-computation phase computes 3 PD + 3 PA operations for further use in point multiplication. Inversion is implemented using Fermat's Theorem using the same Montgomery multiplication and carry-select addition blocks in the design without no additional hardware. Note that the proposed design computes the point multiplication operation in constant time. Thus, it is resistant to timing based side channel attacks.

4.4.2. Performance Comparisons

The proposed ECPM architecture is compared with the state-of-the-art designs in literature implementing ECPM for 512 bit. Table 4.8 gives the FPGA

Table 4.8. Area and time performance of Point Multiplication.

Impl.	Curve	Area					Period (ns)	Latency (ms)
		slice	LUT	FF	DSP	RAM		
Ours	Any	1951	5929	4264	41	31	5.40	4.06
	NIST	2178	6255	4203	41	31	5.40	2.79
[34]	NIST	11200	32900	N/A	289	128	10.0	1.60
[35]	NIST	1980	6115	3567	7	2	3.98	28.04
[36]	NIST	20793	31946	N/A	32	1	16.66	39.90
[23]	Any	N/A	6816	N/A	20	0	4.40	9.70
		N/A	8273	N/A	64	0	6.21	5.02

implementation results and the latency of the proposed ECPM design and some other ECPM designs in literature.

As seen the fastest design belongs to [34]. The design in [34] uses approximately 5 times more slices, LUTs and DSPs and 4 times more BRAM slices than our design. The design in [35] uses two BRAMs which have 17 bits read/write data width and 31 depth. This increases latency to fetch large operands from BRAMs. The design in [36] uses a high number of LUTs but it has very low performance. The design in [23] does not use BRAM slices, instead data is stored in distributed RAMs. Double-and-add algorithm is used. Therefore, it does not need to compute any pre-computation at the expense of larger latency.

4.5. Conclusions

This chapter presents a high performance ECPM architecture for both general primes and NIST prime fields. The proposed design takes the advantage of DSP48E slices in Xilinx FPGAs to compute arithmetic operations. The proposed design also utilizes RAMB36E slices to store large operands. Firstly, Montgomery modular multiplication and carry-select adder circuits are designed. Then, PA and PD operations are implemented using both *Jacobian* projective and mixed *Jacobian-affine* coordinates. Finally, *window method* is used for point multiplication to enhance performance.

The Virtex-7 implementation runs at 5.4 ns clock period and computes ECPM in 2.79 ms for p-521 and 4.06 for 521 bit general prime. The proposed architecture is one of the fastest designs in the literature with modest area complexity.

5. HLS BASED IMPLEMENTATIONS

5.1. Introduction

Recently, HLS tools are started to be used in hardware development for a couple of reasons. Firstly, higher level languages such as C or C++ provide higher abstraction for circuit description than the hardware description languages Verilog or VHDL. Secondly, hardware tools involve more steps to test and verify the design. It also requires more tedious work to change the architecture in case of any need. This flow is faster with the HLS tools.

Modular multipliers using Montgomery and Barrett reductions are abundant in literature with varying digit sizes (δ). Bit-serial architectures [12] - [15] process the operand bits, $\delta = 1$ bit at a time. Although these architectures have low area complexity, they require larger clock cycles to perform one single modular multiplication. Digit based ($\delta > 1$) multipliers are also proposed in the literature to enhance performance at the expense of increased area [17] - [19]. Recent multipliers take the advantage of DSP slices of FPGAs. These dedicated hardware blocks are abundant in FPGAs and can perform advanced arithmetic operations [24], [28], [29].

Xilinx HLS based Elliptic curve scalar multiplication processor is proposed in [38]. Software development methods, such as Incremental and Iterative Design (IID) are used to obtain better performance results. Proposed design uses 256 bit Montgomery multiplier. The architecture is optimized iteratively to decrease latency and area usage.

In this chapter, we propose the first implementations of 528 and 1056 bit Montgomery and Barrett multipliers using Xilinx Vivado HLS tool. Interleaved multiplication is used in Montgomery multiplier with digit size $\delta = 48$. Full-word implementation is used in Barrett multiplier.

The rest of the chapter is organized as follows: The proposed designs based on HLS implementations are given in Section 5.2. Section 5.3 discusses the conclusion.

5.2. Proposed HLS Implementations

The proposed designs for Montgomery and Barrett multiplications mainly consist of several 48x48 bit multiplier units and one carry-select adder. These are the building blocks of Montgomery and Barrett multiplications. Then these blocks are combined in a top block depending on the modular multiplication algorithm. All of the blocks are implemented in C++ code and given as an input to Xilinx Vivado HLS tool.

Vivado HLS provides a template class, *ap_[u] int*, which allows to declare arbitrary precision integer data types. Thus, software model shows the same behavior with the corresponding hardware model.

5.2.1. Overall Structure

Figure 5.1 shows the overall HLS implementations. Input operands are stored into input arrays. Then, they are fed into multiplier units in the correct order according to Table 2.2 and Table 3.1. Partial products are accumulated in the carry-select adder unit. For loops provide multiply and add iterations. After enough iterations modular multiplication output is stored in the output arrays.

Both Montgomery and Barrett multiplications can be implemented using the same blocks in Figure. 5.1. with different array indexing and loop iterations.

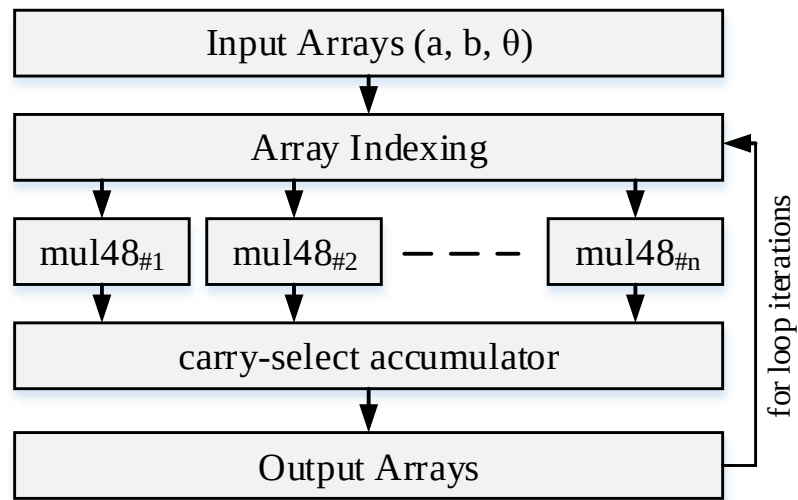


Figure 5.1: Proposed HLS Implementation.

5.2.2. Performance Results

C++ codes implementing the design in Figure 5.1. are given to Vivado HLS tool. Target device is Virtex-7 vx330tffg1157 with speed grade -3.

Vivado HLS tool allows to optimize the design using specific directives inserted into the code. First of all, the design is analyzed using no user directive. As it can be seen in Table I, HLS tool consumes balanced resource utilization. However, the latency is too high because the tool tries to use the same multiplier block in each iteration.

Since the code involves loops, some of them can be unrolled without any data dependency. Thus, UNROLL directive is applied to some loops in the code. Unrolling loops leads to multiple instantiations of multiplier unit in Figure 5.1. As seen in Table 5.1, this directive improved the latency approximately eight and thirteen times for 528 and 1056 bit Montgomery multiplications respectively. This directive also improved the latency approximately five and three times for 528 and 1056 bit Barrett multiplications as shown in Table 5.1.

In order to increase throughput PIPELINE directive is applied. Although, pipelining intermediate steps increases resource usage, latency is improved as shown in Table 5.1. and Table 5.2. Also fifo interface is selected for the top level ports to be able to package the HLS implementation.

Vivado HLS tool also automatically applies some optimizations. HLS inlines small functions and partitions large arrays into smaller ones for fast data access.

HLS tool generates Verilog/VHDL files from C/C++ files. Generated Verilog files are synthesized and mapped to the Virtex-7 vx330tffg1157 device using Vivado synthesis tool. The design is verified using both C/C++ testbench in HLS and Vivado simulator.

Montgomery multiplication gives better timing and area results than Barrett multiplication for HLS implementation. Firstly, Montgomery algorithm uses the least significant bits for the quotient calculation in Table 2.2, whereas Barrett algorithm uses the most significant bits in Table 3.1. Thus, quotient calculation can start earlier in Montgomery multiplication. Secondly, Montgomery algorithm in Table 2.2 carries digit based calculations whereas Barrett multiplication in Table 3.1 carries full-word multiplications. Digit based design is not very efficient for Barrett multiplication because of the wait cycles for the most significant bits before each reduction step. To

eliminate these wait cycles and increase performance, full-word multiplication is more convenient for Barrett multiplication. However, this method needs larger additions. Thus, it leads to higher area and lower clock frequency as opposed to digit based designs.

Since Montgomery algorithm gives better results and majority of works in the literature implement Montgomery algorithm, our HLS implementation is compared with the recent Montgomery multipliers in Table 5.3. For comparison, the designs which have the lowest latency in Table 5.1 are selected.

The fastest design belongs to the work in [18]. Carry-save adders are used to break long carry chains and the design is only mapped to logic slices. No DSP or BRAM slices are used. In [28], both carry-select adders and multiplier blocks are implemented using DSP slices. The registers inside the DSP slices are used to decrease logic register count. The design in [24] uses very low logic resources, at the expense of high latency. The design in [29] achieves high performance using very high logic and DSP slices. Our HLS implementation is slower than the works in [18, 28, 29]. It is better than [24] for 528 bit operand size.

Table 5.1: HLS implementation results for Montgomery multiplication.

Opt.	Op. size	Area				Cycles	Period ns	Time us
		LUT	FF	DSP	BRAM			
No	528	2915	1981	18	12	4334	4.53	19.63
UNROLL	528	6272	8165	36	-	574	5.02	2.88
UNROLL +PIPE	528	6528	9631	42	-	161	8.38	1.34
No	1056	3092	1966	18	16	18343	4.41	80.89
UNROLL	1056	11867	16742	66	-	1278	5.02	6.41
UNROLL +PIPE	1056	12013	19416	72	34	533	8.38	4.46

Table 5.2: HLS implementation results for Barrett multiplication.

Opt.	Op. size	Area				Cycle	Period ns	Time us
		LUT	FF	DSP	BRAM			
No	528	5131	2583	18	23	12608	4.41	55.60
UNROLL	528	15213	14220	6	3	3087	5.00	10.16
UNROLL +PIPE	528	54313	81097	660	-	454	5.68	2.57
No	1056	5200	2654	18	23	48908	4.41	215.6
UNROLL	1056	30083	27840	6	3	11975	5.64	67.53
UNROLL +PIPE	1056	109465	160489	1240	-	1305	5.78	7.54

Table 5.3: Area and time performance of FPGA implementations.

Mult	Op. size	Area				Cycles	Period ns	Time us
		LUT	FF	DSP48	BRAM			
Our	528	6528	9631	42	-	161	8.38	1.34
HLS	1056	12013	19416	72	34	533	8.38	4.46
[28]	528	4039	2547	41	-	91	4.80	0.43
	1056	6217	4968	81	-	179	6.10	1.09
[24]	512	N/A	N/A	4	4	1056	N/A	8.67
	1024	N/A	N/A	33	8	272	N/A	4.21
[29]	512	23511	11307	560	-	29	16.3	0.47
		26519	13249	492	-	38	13.8	0.52
[18]	512	20795	3931	-	-	67	6.27	0.42
	1024	34904	7427	-	-	132	6.88	0.91

Latency flow for the proposed HLS implementation of 528 bit Montgomery multiplication is given in Figure 5.2. Mul48 function is mapped to DSP48 slices, where each 48 bit multiplication takes three cycles as shown in Figure 5.2. Digits of A are multiplied by the corresponding digit of B in the first two mul48 operations. This mul48 function is mapped to six DSP48 slices for 528 bit Montgomery

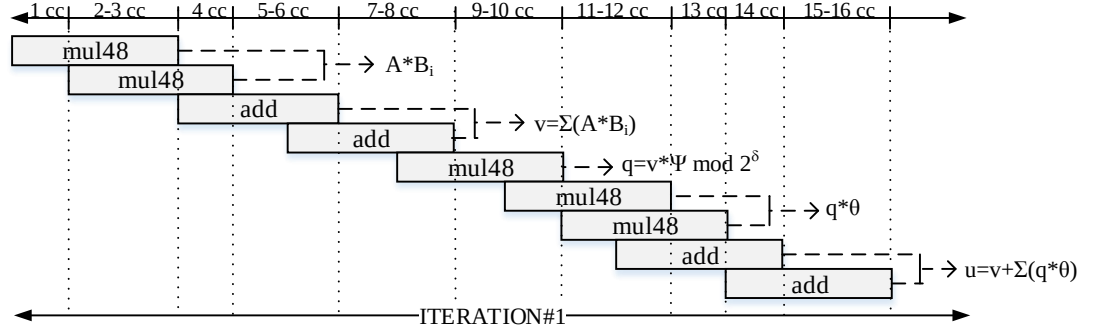


Figure 5.2: Latency of Montgomery HLS implementation.

multiplication and the number of DSP slices doubles for the 1056 bit multiplication. Note that second mul48 can start before the end of first mul48 because of the pipeline strategy.

Partial products computed in the mul48 operations are added to each other using carry-select addition method as shown in Figure 5.2. This addition is mapped to LUTs instead of DSP slices by the HLS tool. Proposed adder circuit in Figure 2.4. gives better results with logic slices than DSP slices. Addition operation also takes three cycles. Even numbered partial products are accumulated in the first add operation and odd numbered partial products are accumulated in the second add operation. The reason for this depends on the clever strategy adopted in the design. Odd and even numbered partial products can be concatenated and fed to the adder circuit. Thus one extra addition step for the partial products is eliminated.

Least significant digit of the accumulated value is multiplied by the precomputed value to obtain q according to the Table 2.2. The same multiplication and addition steps are carried out for q and θ according to Table 2.2, using the same multiplier and adder units.

The same calculations in Figure 5.2 are computed for each iteration in Table 2.2. The total iteration count depends on the operand length. Because of the pipeline directive inserted into the iteration loop, HLS tool achieves twelve cycles initiation interval in order to improve throughput. Thus, overall latency for 528 bit Montgomery multiplication becomes 138 cycles. However, fifo interface created for top level function parameters limits data accesses and overall latency becomes 161 cycles. This fifo interface may be omitted for custom designs, where module is instantiated directly into the Verilog/VHDL code. However, fifo interface is

extensively used in system level designs. Thus, latency results in Table 5.1, 5.2 and 5.3 are given with fifo interface latency.

Although proposed HLS implementation of Montgomery algorithm has competitive results when it is compared to the state-of-the art designs in the literature shown in Table III, the HLS implementation has some drawbacks. The most critical one is quotient estimation computed during the 8-10th cycles in Figure 5.2, can be computed just after the least significant digit is ready in the 4th cycle. Thus, it enables to reduction in latency of iterations. However HLS tool waits the completion of additions to start quotient computation in Figure 5.2.

Handwritten Verilog/VHDL codes allows to design more specific architectures. Thus, high performance designs can be obtained. However, HLS tools try to convert high level C/C++ languages into hardware description languages. Since this conversion may not be able to reflect all architectural details which a handwritten Verilog/VHDL design has, performance degradations occur in HLS implementation. However, HLS allows to change the design quickly and it is much easier to maintain and test the C/C++ code than Verilog/VHDL code. Thus, development time decreases very much with HLS implementation.

5.3. Conclusion

In this chapter, the first HLS implementations of 528/1056 bit Montgomery and Barrett multiplications are proposed. The proposed hardware is implemented on Xilinx Virtex 7 FPGA using Vivado HLS tool. Development time decreased significantly by using a HLS tool. The proposed Montgomery HLS implementations are compared to some recent Montgomery multipliers in literature.

As a future work, different design techniques for these algorithms will be examined to improve the performance of the HLS implementations. Based on the proposed Montgomery and Barrett HLS implementations, elliptic curve and RSA processors will be implemented.

6. CONCLUSION

In this thesis, we first implemented Montgomery and Barrett modular multipliers using DSP resources of modern FPGAs. These multipliers are the core components of RSA and elliptic curve based public-key cryptosystems. Proposed Montgomery and Barrett multipliers are implemented in Verilog HDL and take 0.43 us and 0.49 us to compute one 528 bit modular multiplication in a Virtex-7 FPGA respectively.

We then proposed a Montgomery multiplier based Elliptic Curve Point Multiplication (ECPM) hardware which support prime fields. The hardware can also be configured easily to support mersenne primes. The proposed design is implemented in Verilog and takes 4.06 ms to compute a point multiplication in any prime field and it takes 2.79 ms to compute a point multiplication in NIST p-521 curve in a Virtex-7 FPGA. The proposed hardware is one of the fastest designs in literature and has a very modest resource utilization.

We then, implemented these Montgomery and Barrett multipliers using HLS flow. Proposed designs are implemented in C++ and given in Vivado HLS tool. Using specific directives HLS results of these algorithms can be improved. Montgomery and Barrett multiplier take 1.34 us and 2.57 us to compute one 528 bit modular multiplication in a Virtex-7 FPGA respectively.

REFERENCES

- [1] Diffie W., Hellman M.E., (2006), “New directions in cryptography”, IEEE Trans. Inf. Theory 22 (6), 644–654.
- [2] Rivest R.L., Shamir A., Adleman L.M., (1978), “A method for obtaining digital signatures and public-key cryptosystems”, Commun. ACM 21 (2), 120–126.
- [3] Koblitz N., (1987), “Elliptic curve cryptosystems”, Math. Comput. 48 (177), 203–209.
- [4] Miller V.S., (1985), “Use of elliptic curves in cryptography”, in: H.C. Williams (Ed.), Advances in Cryptology - CRYPTO '85 proceedings, 218 Springer-Verlag, 417–426, Santa Barbara, CA, USA, 18-22 August.
- [5] Montgomery P.L., (1985), “Modular multiplication without trial division”, Math. Comput. 44 (170), 519–521.
- [6] Yang C.C., Chang T.S., Jen C.W. (1998), “A new RSA cryptosystem hardware design based on montgomery’s algorithm”, IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 45 (7), 908–913.
- [7] Tenca A.F., Koç Ç.K., (1999), “A scalable architecture for Montgomery multiplication, in: Ç.K. Koç, C. Paar (Eds.)”, CHES'99 First International Workshop on Cryptographic Hardware and Embedded Systems, 1717 Springer, 94–108, Worcester, MA, USA, 12-13 August.
- [8] Tenca A.F., Koç Ç.K. (2003), “A scalable architecture for modular multiplication based on montgomery’s algorithm”, IEEE Trans. Comput. 52 (9), 1215–1221.
- [9] McIvor C., McLoone M., McCanny J.V., (2004), “Modified montgomery modular multiplication and RSA exponentiation techniques”, IEE Proceedings Computers and Digital Techniques 151 (6), 402–408.
- [10] Harris D. M., Krishnamurthy R., Anders M., Mathew S., Hsu S. (2005), “An improved unified scalable radix-2 Montgomery multiplier”, in: P. Montuschi, E. Schwarz (Eds.), Proceedings 17th IEEE Symposium on Computer Arithmetic, IEEE, 172–178, Cape Cod, MA, USA, 27-29 June.
- [11] Shieh M.D., Chen J.H., Wu H.H., Lin W.C., (2008), “A new modular exponentiation architecture for efficient design of RSA cryptosystem”, IEEE Transactions on Very Large Scale Integration Systems 16 (9), 1151–1161.
- [12] Shieh M.D., Lin W.C., (2010), “Word-based montgomery modular multiplication algorithm for low-latency scalable architectures”, IEEE Trans. Comput. 59 (8), 1145–1151.

- [13] Huang M., Gaj K., El-Ghazawi T., (2011), “New hardware architectures for Montgomery modular multiplication algorithm”, *IEEE Trans. Comput.* 60 (7), 923–936.
- [14] Kuang S.R., Wang J.P., Chang K.C., Hsu H.W. (2013), “Energy-efficient high-throughput montgomery modular multipliers for RSA cryptosystems”, *IEEE Transactions on Very Large Scale Integration Systems* 21 (11), 1999–2009.
- [15] Kuang S.R., Wu K.Y., Lu R.Y., (2016), “Low-cost high-performance vlsi architecture for montgomery modular multiplication”, *IEEE Transactions on Very Large Scale Integration Systems* 24 (2), 434–443.
- [16] Tenca A.F., Todorov G., Koç Ç.K., (2001), “High-radix design of a scalable modular multiplier”, in: Ç.K. Koç, D. Naccache, C. Paar (Eds.), *Proceedings Third International Workshop Cryptographic Hardware and Embedded Systems (CHES’01)*, 2162 Springer-Verlag, 185–201, Paris, France, 13–16 May.
- [17] Knezevic M., Vercauteren F., Verbauwhede I., (2010), “Faster interleaved modular multiplication based on barrett and montgomery reduction methods”, *IEEE Trans. Comput.* 59 (12), 1715–1721.
- [18] Erdem S.S., Yanık T., Çelebi A., (2017), “A general digit-serial architecture for Montgomery modular multiplication”, *IEEE Transactions on Very Large Scale Integration Systems* 25 (5), 1658–1668.
- [19] Rezai A., Keshavarzi P., (2015), “High-throughput modular multiplication and exponentiation algorithms using multibit-scan-multibit-shift technique”, *IEEE Transactions on Very Large Scale Integration Systems* 23 (9), 1710–1719.
- [20] Alho T., Hamalainen P., Hannikainen M., Hamalainen T.D., (2007), “Compact modular exponentiation accelerator for modern FPGA devices”, *Comp. Elect. Eng.* 33 (5), 383–391.
- [21] Bo S., Kawakami K., Nakano K., Ito Y., (2010), “An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA”, *First International Conference on Networking and Computing (ICNC)*, IEEE, 140–147, Hiroshima, Japan, 17–19 November.
- [22] Bo S., Kawakami K., Nakano K., Ito Y., (2011), “An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA”, *Intern. J. Network. Comp.* 1 (2), 277–289.
- [23] Amiet D., Curiger A., Zbinden P., (2016), “Flexible FPGA-based architectures for curve point multiplication over $GF(p)$ ”, in: P. Kitsos (Ed.), *19th Euromicro Conference on Digital System Design (DSD)*, IEEE, 107–114, Limassol, Cyprus, 31 August - 2 September.
- [24] Morales-Sandoval M., Diaz-Perez A., (2016), “Scalable $GF(p)$ montgomery multiplier based on a digit-digit computation approach”, *IET Computers and Digital Techniques* 10 (3), 102–109.

- [25] Yang Y., Wu C., Li Z., Yang J., (2016), "Efficient FPGA implementation of modular multiplication based on montgomery algorithm", *Microprocessors & Microsystems* 47 (Part A), 209–215.
- [26] Yanık T., Savaş E., Koç Ç.K., (2002), "Incomplete reduction in modular arithmetic", *IEE Proceedings Computers and Digital Techniques* 149 (2), 46–52.
- [27] Barrett P., (1986), "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor", *Proc. Ann.Int'l Cryptology Conf. Advances in Cryptology (CRYPTO '86)*, 311-323, Santa Barbara, CA, USA, 11-15 August.
- [28] Özcan E., Erdem S. S., (2018), "A fast digit based Montgomery multiplier designed for FPGAs with DSP resources", *Microprocessors and Microsystems*, 62, 12–19.
- [29] Khan S., Javeed K., Shah Y., (2018), "High-speed FPGA implementation of full-word Montgomery multiplier for ECC applications", *Microprocessors and Microsystems*, 62, 91-101.
- [30] Wu T., Li S., Liu L., (2012), "Modular multiplier by folding Barrett modular reduction" *IEEE 11th International Conference on Solid-State and Integrated Circuit Technology*, 1-3, Xi'an, China, 29 October – 1 November.
- [31] Hankerson, D., Vanstone, S., Menezes, A.J. (2004), "Guide to Elliptic Curve Cryptography", Springer, New York.
- [32] Güneysu T., Paar C. (2008), "Ultra high performance ECC over NIST primes on commercial FPGAs". In: Oswald E., Rohatgi P. (eds), *Cryptographic Hardware and Embedded Systems – CHES 2008. Lecture Notes in Computer Science*, 5154. Springer, Washington, DC, USA, 10-13 August.
- [33] Hossain M. S., Kong Y., Saeedi E. Vayalil N. C., (2017), "High-performance elliptic curve cryptography processor over NIST prime fields", *IET Computers & Digital Techniques*, 11, 33-42.
- [34] Alrimeih H., Rakhmatov D., (2014), "Fast and flexible hardware support for ECC over multiple standard prime fields", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22 (12), 2661-2674.
- [35] Loi K. C. C., Ko S. B., (2015), "Scalable elliptic curve cryptosystem FPGA processor for NIST prime curves", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23 (11), 2753-2756.
- [36] Ananyi K., Alrimeih H. and Rakhmatov D., (2009), "Flexible hardware processor for elliptic curve cryptography over NIST prime fields", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17 (8), 1099-1112.

- [37] National Institute of Standards and Technology, FIPS 186-2, (2000), “Digital Signature Standard (DSS)”, Gaithersburg, MD, USA.
- [38] Pontie S. et al., (2016), “HLS-based methodology for fast iterative development applied to elliptic curve arithmetic”, 2016 Euromicro Conference on Digital System Design (DSD), 511-518, Limassol, Cyprus, 31 August - 2 September.

BIOGRAPHY

Erdem Özcan was born in İzmir in 1989. He obtained his BSc. and MSc. degrees in Electronics Engineering from Sabancı University in 2011 and 2013 respectively. After working as an Embedded Software Engineer in Vestel R&D, he started working as a Researcher at TUBITAK BILGEM in 2014. He started his PhD. study in Electronics Engineering at Gebze Technical University in 2015. His research area includes reconfigurable computing, FPGAs, hardware accelerator design for cryptography and embedded system security.

APPENDICES

Appendix A: Publications of the Thesis

Özcan, E., Erdem S.S., (2018), “A fast digit based Montgomery multiplier designed for FPGAs with DSP resources”, *Microprocessors and Microsystems*, 62, 12–19.