ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

DESIGN AND IMPLEMENTATION OF RSA CRYPTOSYSTEM USING PARTIALLY INTERLEAVED MODULAR KARATSUBA-OFMAN MULTIPLIER

M.Sc. THESIS

Ahmet ARIŞ

Department of Computer Engineering

Computer Engineering Programme

JUNE 2012

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

DESIGN AND IMPLEMENTATION OF RSA CRYPTOSYSTEM USING PARTIALLY INTERLEAVED MODULAR KARATSUBA-OFMAN MULTIPLIER

M.Sc. THESIS

Ahmet ARIŞ
(504091501)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. S. Berna ÖRS YALÇIN

JUNE 2012

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

İKİ PARÇALI ÖRGÜ MODÜLER KARATSUBA-OFMAN ÇARPICISI KULLANARAK RSA KRİPTOSİSTEMİ TASARIMI VE GERÇEKLEMESİ

YÜKSEK LİSANS TEZİ

Ahmet ARIŞ
(504091501)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Doç. Dr. S. Berna ÖRS YALÇIN

HAZİRAN 2012

Ahmet ARIŞ, a M.Sc. student of ITU Graduate School of Science 504091501 successfully defended the thesis entitled **"DESIGN AND IMPLEMENTATION OF RSA CRYPTOSYSTEM USING PARTIALLY INTERLEAVED MODULAR KARATSUBA-OFMAN MULTIPLIER"**, which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. S. Berna ÖRS YALÇIN** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Ece Olcay GÜNEŞ** Istanbul Technical University	.....
	Asst. Prof. Dr. D. Turgay ALTILAR Istanbul Technical University	.....

Date of Submission : **4 May 2012**
Date of Defense : **6 June 2012**

To the gorgeous people in my life,

FOREWORD

Without the help and support of some gorgeous people in my life, I do not think I would able to achieve anything.

During this thesis and my masters degree, Assoc. Prof. Dr. S. Berna Örs Yalçın has been an ideal advisor for me. Her support, ideas and comments always ignited me to work better. My other advisor, who accepted me to his career project and gave me chance to implement his original algorithm is Assis. Prof. Dr. Gökay Saldamlı. He has been an excellent advisor too. He was always going in front of me, coming up with new implementation ideas. His knowledge and experience served me as a model.

I would like to thank to my family(my dad, mum and brothers Osman and Aziz) and my girlfriend Mairi. They were always with me although we were mostly hundreds of kms far from each other physically. I always felt their support in troublesome situations and always celebrated my successes with them. In addition to my biological family and Mairi, there is also another family, namely Çaçaron family, who have been helping and supporting me since the English prep year of high school till now. They have always accepted me as a member of their family and I have always seen them as my family. I am wishing our relation to last for many more years. I would love to thank to all of them especially to Uğur Çaçaron.

I would also like to thank to my teachers from primary school and high school, namely Gürzat Kışlak, Yücel Erdoğan, Ozan Özel. Also my very close friends Şaban and Adem never left me alone for more than 15 years. Although primary school and high school years have already passed, they have always been great friends.

Lastly, I am grateful to my friends working in GSTL laboratory and my flatmates for their company and İnce Minare Cami foundation for their support.

June 2012

Ahmet ARIŞ
Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

AES	**:** Advanced Encryption Standard
ASIC	**:** Application Specific Integrated Circuit
BMM	**:** Bipartite Modular Multiplication
CLA	**:** Carry Lookahead Adder
CPLD	**:** Complex Field Programmable Logic Device
CSA	**:** Carry Save Adder
FA	**:** Full Adder
FPGA	**:** Field Programmable Gate Array
HA	**:** Half Adder
IC	**:** Integrated Circuit
KO	**:** Karatsuba-Ofman
LSB	**:** Least Significant Bit
MSB	**:** Most Significant Bit
PKC	**:** Public Key Cryptosystem
RSA	**:** Rivest-Shamir-Adleman
RTL	**:** Register Transfer Level
TSMC	**:** Taiwan Semiconductor Manufacturing Company
VHDL	**:** VHSIC Hardware Description Language

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF SYMBOLS

gcd	:	Greatest Common Divisor
$\mathscr{H}(E)$	:	Hamming Weight of E: number of 1's in the binary expansion of E
$\phi(N)$	:	Euler's Totient Function of N

DESIGN AND IMPLEMENTATION OF RSA CRYPTOSYSTEM USING PARTIALLY INTERLEAVED MODULAR KARATSUBA-OFMAN MULTIPLIER

SUMMARY

Importance of cryptography is becoming more and more important day by day. Secure communication will always be a crucial need independent from the technology in use. Applications of cryptography can be seen in online banking, purchases of goods and services by means of credit cards, ID cards, remote lock and start systems for cars, telecommunication, and in many more places. Different cryptosystems are employed according to different needs and different security levels.

Rivest-Shamir-Adleman(RSA) Algorithm is one of the most popular cryptosystem that is used in many sectors like banking. It takes its strength from factorization of very large integers. Briefly, it consists of modular exponentiations with large integers which are 1024-bit or 2048-bit numbers. As modular exponentiation operations are fundamentally composed of modular multiplications, designing a fast RSA cryptosystem becomes possible only with a fast modular multiplier implementation. Due to security and speed issues, modular exponentiation in RSA is implemented in software and modular multiplications are carried out in modular multipliers which are implemented in hardware.

Many methods exist in the literature in order to perform modular exponentiation. These methods try to reduce the total number of multiplications that are required for a modular exponetiation operation. In addition to modular exponentiation algorithms, there are several techniques to do modular multiplication as well. Modular multiplication may be carried out either by multiplying first and performing reduction later on, or by interleaving multiplication and reduction stages. In order to reach compact hardware designs the latter approach is utilized. According to the direction of reduction operation, modular multiplication algorithms may be classified into two groups, which are algorithms reducing from left-to-right and from right-to-left. The most well known examples of left-to-right approach are Blakley and Barrett algorithms, whereas Montgomery multiplication is the only member of the right-to-left modular multiplication. Although fast multiplication algorithms, such as Karatsuba-Ofman(KO), Schönhage-Strassen exist, these multiplication methods can not be interleaved with reduction algorithms. This is caused from the reduction approaches not allowing parallel processing. However, Bipartite Modular Multiplication(BMM) method introduced by Kaihara and Takagi proposes a partial solution to the parallel reduction problem. This reduction method modifies a feature of Montgomery algorithm. This modification separates the multiplier bits into two halves so that a product can be reduced from left and right simultaneously without a dependency issue.

The first method which combines Karatsuba-Ofman multiplication with bipartite reduction was proposed by Gökay Saldamlı. His algoritm, namely Partially Interleaved Modular Karatsuba-Ofman Multiplication, interleaves KO multiplier with the bipartite reduction on the uppermost layer of KO's recursion. Implementation of this new method requires Montgomery multiplier, Blakley multiplier and standard integer multipliers. However, for this approach, both Montgomery and Blakley multipliers are needed to be designed in a way that, they compute not only the modular multiplication result, but also quotient values, what is somewhat different than existing implementations.

In this thesis, two hardware implementations of Partially Interleaved Modular KO Multiplication and an RSA implementation utilizing the designed multiplier are proposed. The first design is Radix-2 implementation on FPGA technology. The second hardware implementation explores the design methodologies in order to reach a faster modular multiplier. These design methodologies include employing high radices, optimization of critical path according to the effect of control signals and analyzing parameter dependencies in Partially Interleaved Modular KO Multiplier and scheduling jobs effectively. Improved design was implemented on ASIC technology using 90 nm TSMC standard cell libraries in Design Compiler.

Hardware implementations proposed in this thesis are the first hardware implementations of Partially Interleaved Modular KO Multiplication method. Montgomery and Blakley multiplication algorithms were modified in order to produce desired results. Maple libraries which emulate the operation of hardware building blocks were written and both hardware implementations were firstly implemented in Maple. Tests were run with random input vectors and when correct operation of Maple implementations were verified, hardwares were described in VHDL and implemented using FPGA and ASIC design tools respectively. The second implementation of Partially Interleaved Modular KO multiplier was used as a modular multiplier for RSA cryptosystem and very promising results which are comparable with commercial RSA chips were achieved.

İKİ PARÇALI ÖRGÜ MODÜLER KARATSUBA-OFMAN ÇARPICISI KULLANARAK RSA KRİPTOSİSTEMİ TASARIMI VE GERÇEKLEMESİ

ÖZET

Kriptografinin, yani şifreleme biliminin önemi gün geçtikçe artmaktadır. Kullanılan teknoloji ne olursa olsun güvenli iletişim her zaman en başta gelen ihtiyaçlardan birisi olacaktır. Günümüzde şifreleme, sistemlere kullanıcı hesabıyla giriş yapılıyorken, internetten herhangi bir hizmet ya da ürün satın alınıyorken, iletişim araçları kullanılıyorken, araçların kapıları uzaktan kilitlenip açılıyorken ve daha birçok yerde kullanılmaktadır. Kullanım alanına ve güvenlik ihtiyacının niteliğine göre değişik şifreleme algoritmaları farklı teknolojilerle karşımıza çıkmaktadır. Bu algoritmaların bir tanesi de Rivest-Shamir-Adleman(RSA) şifreleme sistemidir. RSA bankacılık başta olmak üzere birçok sektörde şifreleme ve sayısal imza işlemleri için sıklıkla kullanılmaktadır.

RSA algoritması gücünü çok büyük sayıların asal çarpanlarına ayrılmalarındaki zorluktan almaktadır. Özetle çok büyük sayılarla yapılan modüler üs alma işlemlerinden oluşmaktadır. Modüler üs alma işlemleri de özünde modüler çarpma işlemlerinden ibaret olduğu için hızlı bir RSA gerçeklemesi ancak hızlı modüler çarpma işlemi yapan bir tasarımla mümkün olmaktadır. Güvenlik ve hız gibi sebeplerden ötürü RSA kriptosistemi genellikle modüler üs alma işleminin yazılımda gerçeklenmesi ve modüler çarpma işlemlerinin de donanımda tasarlanan özel bloklarla yapılması yoluyla gerçeklenir.

Modüler üs alma işlemi için birçok yöntem mevcuttur. Bu yöntemler modüler üs alma işlemi esnasında yapılan modüler çarpma işlemi sayısını değişik yöntemlerle en aza indirmeye çalışırlar. Modüler çarpma işlemi için de bilim dünyasında epeyi çalışmalar yapılmıştır. Modüler çarpma, önce çarpıp sonra indirgeme yapma ya da çarpma ve indirgeme işlemlerini iç-içe yapma gibi iki yöntemle mümkündür. Alan kısıtlamaları sebebiyle çoğunlukla ikinci metot tercih edilmektedir. İndirgeme işleminin uygulanma yönüne göre modüler çarpma algoritmaları soldan sağa doğru işleyenler ve sağdan sola doğru işleyenler olmak üzere ikiye ayrılır. Soldan sağa doğru işleyen yöntemlerin en bilindik örnekleri Blakley ve Barrett algoritmalarıdır. Sağdan sola doğru indirgeme yapan sınıfa ise tek örnek Montgomery yöntemidir. Hızlı çarpma yapan Karatsuba-Ofman, Schönhage-Strassen gibi yöntemler olsa da bu hızlı çarpıcılar indirgeme algoritmalarıyla birleştirilememektedirler. Bu sorun paralel çalışmaya izin vermeyen indirgeme yöntemlerinden kaynaklanmaktadır. Ancak Kaihara ve Takagi tarafından bilim dünyasına sunulan İki Parçalı Modüler çarpma yöntemi bu soruna bir nebze de olsa çözüm bulmaktadır. Bu indirgeme metodu Montgomery algoritmasının sahip olduğu bir özelliği kullanarak modüler çarpmadaki çarpanı ikiye ayırmakta ve böylelikle Blakley ve Montgomery paralel olarak çalışabilmektedir.

Hızlı çarpma algoritmalarından birisi olan Karatsuba-Ofman yöntemiyle İki Parçalı indirgemeyi birleştiren ilk çalışma Gökay Saldamlı tarafından ortaya atılmıştır. İki Parçalı Örgü Karatsuba-Ofman çarpıcısı iki parçalı indirgemeyi Karatsuba-Ofman rekürsif çarpma yönteminin en üst katmanında birleştirmektedir. Bu yeni yöntemin gerçeklenmesinde Montgomery çarpıcısına, Blakley modüler çarpıcısına ve standart çarpma yapan bloklara ihtiyaç vardır. Ancak bu algoritma Montgomery ve Blakley modüllerinde literatürdeki daha önceki gerçeklemelerde bulunmayan değerlerin de hesaplanmasına gereksinim duymaktadır.

Bu tezde İki Parçalı Modüler Örgü Karatsuba-Ofman çarpıcısının donanımda gerçeklenmesine ait iki çalışma ve bu tasarımlardan birisi kullanılarak gerçeklenen RSA kriptosistemi anlatılmaktadır. Bu çalışmalardan ilki çarpanı birer bit işleyen gerçeklemedir. FPGA teknolojisinde gerçeklenmiştir. İkinci gerçekleme ise ilk tasarımdaki eksikleri kapatan ve daha hızlı bir modüler çarpma için kodlama yöntemleri, daha fazla sayıda bit işleme, donanımın çalışma frekansını arttırmak için en büyük gecikmeye sahip yolu kontrol sinyallerini kullanarak optimize etmeye çalışan bir ASIC gerçeklemesidir.

Bu tezdeki donanım tasarımları İki Parçalı Örgü Karatsuba-Ofman çarpma yönteminin ilk gerçeklemeleridirler. Montgomery ve Blakley algoritmaları bu yeni yöntem için yeniden düzenlenmiştir. Tasarımların ikisi için de Maple'da kütüphaneler oluşturulmuş ve iki tasarım da Maple ortamında donanımla aynı yapıda gerçeklenmiş, gerekli testler yapılmış ve simülatörlerden gelen sonuçlarla yazılım gerçeklemesinden gelen sonuçlar karşılaştırılarak tasarımların doğru çalıştığı kanıtlanmıştır. İkinci modüler çarpma gerçeklemesi RSA kriptosistemi içinde modüler çarpıcı olarak kullanılmış ve piyasadaki RSA kırmıklarıyla karşılaştırılabilir sonuçlara ulaşılmıştır.

1. INTRODUCTION

The importance of cryptography dates back to the time of Julius Caesar. Since then, the communication medium has been changing from letter to radio, radio to telephone, personal computers and Internet, and currently smart phones. But the need for communicating securely remained crucial, independent from the underlying technology.

Cryptosystems may be classified into two categories: Symmetric Key Cryptosystems and Public Key Cryptosystems(PKC). In symmetric key cryptosystems, same secret key is used both for encryption and decryption operations. AES [1] is the most famous algorithm used in this class. A simple diagram can be seen in the Figure 1.1

In PKC, there are key pairs for each communication endpoints, namely public key and private key. As shown in the Figure 1.2, when Alice wants to send a message to Bob, Alice encrypts her message(plain text) with the public key of Bob which is public to everybody. She sends the encrypted message(cipher text) over an insecure channel. Underlying mathematical properties of PKC allow only Bob to decrypt it with his private key, which is only known by him.

RSA [2] is the most popular PKC. It gets its strength from the difficulty of large integer factorization. In exchange for providing a robust security, it has the drawback of modular exponentiation operation with very large integers(i.e. 1024, 2048-bit). Due to

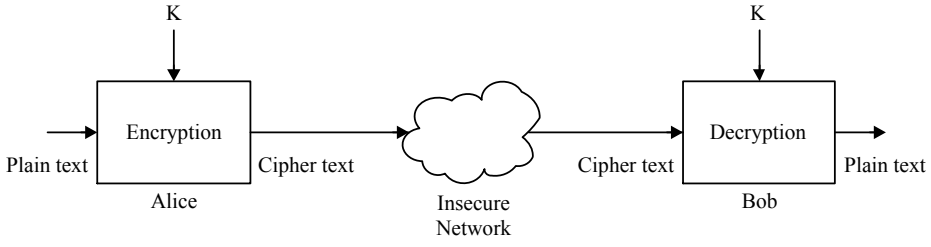

Figure 1.1: Symmetric Key Cryptography.

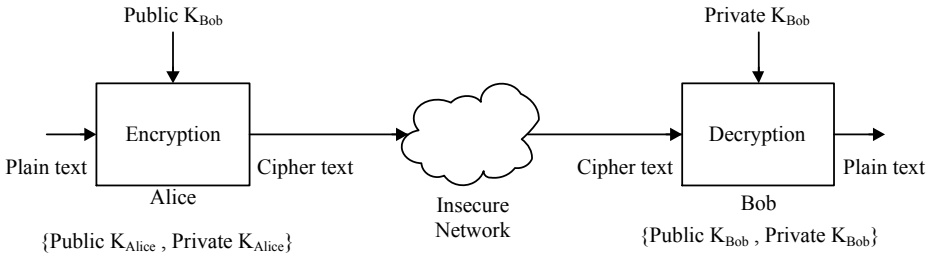

Figure 1.2: Public Key Cryptography.

the timing and security requirements, RSA cyrptosystem has been widely implemented on dedicated hardware, rather than software, either on FPGA or ASIC chips.

Modular exponentiation with large modulus in RSA consists of modular multiplication operations. Favorably, modular exponentiation is implemented in software, using modular multipliers that are implemented in hardware. In order to perform fast encryption or decryption, one has to decrease the total number of modular multiplications within a modular exponentiation by means of scanning techniques like m-ary method.

The key point in reaching a fast RSA cryptosystem is implementing a fast modular multiplier and using it with a modular exponentiation method which allows RSA to be performed in the least number of modular multiplications in the shortest amount of time.

Modular multiplication operation can be performed in two ways: either multiplying first and then reducing the product later or, interleaving the multiplication and reduction stages. In order to reach compact hardware designs, second approach is generally preferred by designers. There are various methods of applying reduction. There are also many ways to perform fast multiplication. The logical way to design a fast modular multiplier would be selecting best candidates from each set. But because of the dependency issue of the reduction algorithms which does not allow parallel reduction, fast multipliers such as Karatsuba-Ofman (KO) [3] could not be interleaved with these methods. However, couple of recent studies [4], [5] demonstrated a partial method of interleaving KO multiplier with the bipartite reduction on the uppermost layer of KO's recursion. Bipartite Modular Multiplication (BMM) method [6], [7]

which was originally introduced by Kaihara and Takagi, outlines a global method of parallel reduction in the way that, a product can be reduced from left and right simultaneously without a dependency issue.

In this thesis, two hardware implementations and an RSA cryptosystem employing the designed fast multiplier will be presented. The first multiplier design implements Partially Interleaved Modular KO Multiplier on FPGA platform. The second one explores the optimizations for the algorithm and for the previous design and implements the high radix version of the multiplier on ASIC, in order to reach a very fast RSA with the help of selected modular exponentiation methods. This work also aims to propose a new design methodology to the cryptographic hardware developers that has never been paid attention to. Up until now, the effect of control path to the critical path of a design has always been neglected. But, when a critical path of a design is analyzed, the impact of control path shows itself with extra logic gates inserted into the critical path, which was not an issue that designer counts. This study emphasizes control-path-aware digital design which will hopefully help designers to achieve faster cryptographic hardware designs.

This thesis is organized in a way that, firstly preliminaries will be given in Chapter 2. Then in Chapter 3, implementation environment and design tools will be presented. Hardware implementations of Partially Interleaved Modular KO Multiplication method will be explained in detail in Chapter 4. Chapter 5 will conclude the thesis.

2. THEORETICAL BACKGROUND

In this chapter, preliminaries will be given about RSA cryptosystem, modular exponentiation and modular multiplication operations, Partially Interleaved Modular Karatsuba-Ofman Multiplication algorithm, standard multiplication algorithm and Booth Encoding.

2.1 The RSA Cryptosystem

The RSA is one of the most famous public key cryptosystem. It was invented by Rivest, Shamir and Adleman [2]. It is used both to exchange private messages and sign digital documents. In RSA every communication endpoint has a key pair, particularly public key and private key. These keys are calculated as follows [8]:

Let p and q be two distinct large random primes. The modulus n is the product of these two primes: $n = pq$. Euler's totient function of n is given by

$$\phi(n) = (p-1)(q-1) \tag{2.1}$$

Now, select a number $1 < e < \phi(n)$ such that

$$\gcd(e, \phi(n)) = 1$$

and compute d with

$$d \equiv e^{-1} \bmod \phi(n)$$

using the extended Euclidean algorithm [9]. Here, e is the public exponent and d is the private exponent. Usually one selects a small public exponent, e.g., $e = 2^{16} + 1$. The modulus n and the public exponent e are published. The value of d and the prime numbers p and q are kept secret.

With public exponent e and modulus n and private exponent d, encryption and decryption operations of RSA are performed as follows:

Let plain text and cipher text are denoted by M and C respectively. Then, RSA encryption is the operation of

$$C = M^e (\bmod n) \tag{2.2}$$

where $0 \leq M < n$. Retrieval of the original message, which is simply RSA decryption is carried out as

$$M = C^d (\bmod n) \tag{2.3}$$

The correctness of RSA algorithm follows from Euler's theorem : Let n and a be positive, relatively prime integers. Then,

$$a^{\phi(n)} = 1 (\bmod n). \tag{2.4}$$

Since we have $ed = 1 \bmod \phi(n)$, i.e., $ed = 1 + K\phi(n)$ for some integer K, we can write

$$\begin{aligned} C^d &= (M^e)^d (\bmod n) \\ &= M^{ed} (\bmod n) \\ &= M^{1+K\phi(n)} (\bmod n) \\ &= M \cdot (M^{\phi(n)})^K (\bmod n) \\ &= M \cdot 1 (\bmod n) \end{aligned}$$

provided that $\gcd(M,n) = 1$.

Public-key directory, as shown in the Table 2.1, contains the pairs $(e,n)$for each user. The pair n_a and e_a are the modulus and public exponent for Alice respectively. If Alice wants to send a private message to Bob, then she executes the following steps [8]:

Table 2.1: Public-key Directory, adapted from (Koç, 1994).

User	Public Keys
Alice	(e_a, n_a)
Bob	(e_b, n_b)
...	...

1. Alice obtains Bob's public-key exponent and modulus (e_b, n_b) from the directory.
2. Alice computes $C = M^{e_b} (\bmod n_b)$.
3. Alice sends C to Bob over the network.
4. Bob receives C and uses his private-key exponent and modulus and computes $M = C^{d_b} (\bmod n_b)$ in order to obtain M.

RSA provides robust security by means of its large distinct primes p and q. Large modulus n and d makes integer factorization infeasible using the current technology. As processing capabilities of computers increase, security parameters of RSA change in parallel.

In exchange for ensuring a strong security, RSA has a drawback of modular exponentiation operations with very large integers(i.e., 1024-bit). Due to the timing and security requirements RSA cyrptosystem is widely implemented on dedicated hardware(either on FPGA or ASIC chips), rather than software.

2.2 Modular Exponentiation

As Koç states in his report [8], the first rule of modular exponentiation is not to compute

$$C = M^e (\bmod n)$$

by first exponentiating

$$M^e$$

and then performing a division to obtain the remainder

$$C = M^e (\%n).$$

Due to the space requirements, temporary results must be reduced modulo n at each step of the exponentiation. This means modular exponentiation operation has to consist of series of modular multiplications.

Accompanying different methods for modular exponentiation yields different performances. In order to speed up the modular exponentiation operation, a scanning method for the exponent must be chosen which allows modular exponentiation to be performed in the least number of modular multiplications. The simplest way to compute $C = M^e (\bmod n)$ would be starting with $C = M (\bmod n)$ and then performing $C = C \cdot M (\bmod n)$ for $e-1$ times. For example, assume $e = 20$, then using this approach, one modular exponentiation would be completed in 19 multiplications:

$$M \rightarrow M^2 \rightarrow M^3 \rightarrow M^4 \rightarrow M^5 \rightarrow \cdots \rightarrow M^{20}$$

However, not all powers of M are needed to be computed in order to obtain M^{20}. For example a faster method to compute M^{20} is shown below:

$$M \rightarrow M^2 \rightarrow M^4 \rightarrow M^5 \rightarrow M^{10} \rightarrow M^{20}$$

As it can be seen clearly, only 5 multiplications are needed to calculate M^{20}. This approach is known as *binary method* or *square and multiply method* and it can be used to compute M^e for any e [8].

2.2.1 The binary method

The binary method scans the exponent bit by bit either from left to right(MSB to LSB) or right to left. At it can be understood from the name, a squaring is performed at each step and then according to the scanned exponent bit, either a multiplication is done or not. Let k denote the number of bits in the exponent, then the binary method can be seen in the Algorithm 1.

For example, let $e = 116 = (1110100)$. Here $k = 7$. Since $e_{k-1} = e_6 = 1$ initially $C = M$. The binary method proceeds as shown in the Table 2.2.

Algorithm 1 The Binary Method.

Require: M, e, n
Ensure: $C = M^e \pmod n$
1: **if** $e_{k-1} = 1$ **then**
2: $C := M$;
3: **else**
4: $C := 1$;
5: **end if**
6: **for** $i = k-2$ **downto** 0 **do**
7: $C := C \cdot C \pmod n$
8: **if** $e_i = 1$ **then**
9: $C := C \cdot M \pmod n$
10: **end if**
11: **end for**
12: **return** C

Table 2.2: Binary Method Steps, adapted from (Koç, 1994).

i	e_i	Step 7	Step 9
5	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
4	1	$(M^3)^2 = M^6$	$M^6 \cdot M = M^7$
3	0	$(M^7)^2 = M^{14}$	M^{14}
2	1	$(M^{14})^2 = M^{28}$	$M^{28} \cdot M = M^{29}$
1	0	$(M^{29})^2 = M^{58}$	M^{58}
0	0	$(M^{58})^2 = M^{116}$	M^{116}

As shown in the Table 2.2, in order to compute M^{116} $6+3=9$ multiplications are needed. For an arbitrary k-bit number e with $e_{k-1} = 1$, the binary method requires [8]:

- Squarings(Step 7): $k-1$
- Multiplications(Step 9): $H(e)-1$ where $H(e)$ is the Hamming Weight of e.

$0 \leq H(e)-1 \leq k-1$, assuming $e > 0$. Total number of multiplications is found as shown in the Table 2.3 below:

Table 2.3: Total Number of Multiplication for Binary Method, adapted from(Koç, 1994).

Case	Number of Multiplications
Maximum	$(k-1)+(k-1) = 2(k-1)$
Minimum	$(k-1)+0 = (k-1)$
Average	$(k-1)+\frac{1}{2}(k-1) = \frac{3}{2}(k-1)$

2.3 Modular Multiplication

In the literature, there are various proposals and enhancements for carrying out the modular multiplication operations in RSA. These proposals can be divided into two simple categories with respect to their reduction approach: namely, algorithms reducing from left-to-right and from right-to-left [4]. In fact, for left-to-right approach, several proposals exist [10], [11]. Algorithms including the standard division can be put into this category. Whereas Montgomery multiplication [12] is the only example of right-to-left reduction.

2.3.1 Classic modular multiplication

Algorithms including the standard division can be put into this category [4], [5]. Assume that A, B and N are positive numbers; division algorithm states that there exist positive integers Q and T (namely quotient and remainder) such that

$$A \cdot B = Q \cdot N + T \text{ for } 0 \leq T < N \tag{2.5}$$

Blakley [10] describes one of the simplest algorithm in this class. As presented in the following paragraphs, the computation of AB mod N is achieved by interleaving the shift-add steps of the standard multiplication and the shiftsubtract steps of the usual reduction.

Let a_i and b_i represent the bits of the k-bit numbers A and B, respectively. Then, the product, which is a $2k$-bit number can be written as [8]:

$$A \cdot B = (\sum_{i=0}^{k-1} a_i 2^i) \cdot B = \sum_{i=0}^{k-1} (a_i \cdot B) 2^i \tag{2.6}$$

Blakley's algorithm is based on the above formulation of the product; it interleaves the shift-add steps of the standard multiplication and the shift-subtract steps of the usual reduction to make sure that the remainder is less than the modulus N.

In the Algorithm 2, reduction operation is performed between the steps 4 and 9. These steps of the algorithm can also be denoted as $R := R \bmod N$ or :

$$q_i := \lfloor \tfrac{R}{N} \rfloor$$

Algorithm 2 Blakley's Algorithm.

Require: Integers A, B and N
Ensure: $P = AB(\bmod N)$
1: $R := 0$
2: **for** $i = k-1$ **downto** 0 **do**
3: $R := 2R + a_i \cdot B$
4: **if** $R \geq N$ **then**
5: $R := R - N$
6: **end if**
7: **if** $R \geq N$ **then**
8: $R := R - N$
9: **end if**
10: **end for**
11: **return** R

$$R := R - q_i \cdot N$$

Note that, if the last representation is used for the reduction step of the Blakley algorithm, then q_is can be accumulated to constitute the Q value, which is the quotient in the standard division algorithm.

Let A, B, N and Q are represented with $k = 2h$ bits for some positive integer h. They can be partitioned into their least and most significant h bits as:

$$A = A_1 \cdot 2^h + A_0 = A_1 \cdot r + A_0$$
$$B = B_1 \cdot 2^h + B_0 = B_1 \cdot r + B_0$$
$$N = N_1 \cdot 2^h + N_0 = N_1 \cdot r + N_0$$
$$Q = Q_1 \cdot 2^h + Q_0 = Q_1 \cdot r + Q_0$$

where $r = 2^h$. Operation of Blakley, namely left-to-right modular multiplication , can be illustrated as shown in the Figure 2.1 [4].

The interpretation of the Figure 2.1 can be made as classic modular multiplication consists of multiplication and reduction stages and in order to compute the result AB mod N, QN which is $Q_1N2^h + Q_0N$, is subtracted from the product of AB. During the first h cycles of the algorithm Q_1N2^h is subtracted from AB and the least significant h bits of AB, that is filled with dashed pattern in the figure is not manipulated.

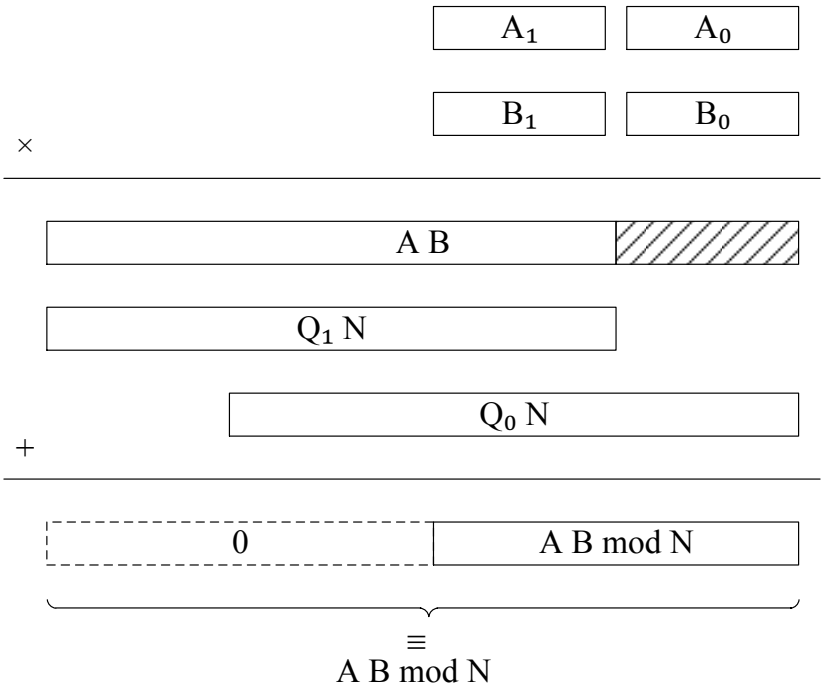

Figure 2.1: Left-to-Right Modular Multiplication, adapted from (Saldamlı, 2011).

2.3.2 Montgomery Multiplication

Montgomery multiplication, which was proposed by Peter L. Montgomery [12] is the only example of right-to-left reduction and the most preferable method of modular multiplication in comparison with left-to-right methods. It replaces costly compare and reduce step of the classic modular multiplication with simple add and shift operations.

Let A, B, and N are k-bit binary numbers. The Montgomery reduction computes $R = AB \bmod N$ without performing a division by the modulus N. Assuming that modulus N is a k bit number, i.e. $2^{k-1} \leq N < 2^k$, let R be 2^k [8]. Montgomery reduction algorithm requires R and N to be relatively prime, i.e. $\gcd(R,N) = \gcd(2^k,N) = 1$. This is satisfied when N is odd.

N-residue of an integer A ($0 \leq A < N$) with respect to R is defined as:

$$\bar{A} = A \cdot R \bmod N.$$

Montgomery reduction of A modulo N with respect to R is defined as:

$$A \cdot R^{-1} \bmod N.$$

Then, Montgomery reduction of two N-residues $\bar{A}$, and $\bar{B}$ is $\bar{A}\bar{B}R^{-1} \bmod N = ABR \bmod N$. This observation is used for modular exponentiation [9]. For example, in order to compute $A^5 \bmod N$, firstly N-residue of A, $\bar{A} = A \cdot R \bmod N$, is computed. Then the Montgomery reduction of $\bar{A}\bar{A}$, which is $X = \bar{A}^2R^{-1} \bmod N$, is computed. The Montgomery reduction of X^2 is $X^2R^{-1} \bmod N = \bar{A}^4R^{-3} \bmod N$. Finally, the Montgomery reduction of $(X^2R^{-1} \bmod N)\bar{A}$ is $(X^2R^{-1})\bar{A}R^{-1} \bmod N = \bar{A}^5R^{-4} \bmod N = A^5R \bmod N$. Multiplying this value by $R^{-1} \bmod N$ and reducing modulo N gives $A^5 \bmod N$.

If N is represented as a base b integer of length k, then a typical choice for R is b^k [9]. In order to describe the Montgomery reduction algorithm, an additional quantity, N', is needed, which is an integer with the property of [8]

$$R \cdot R^{-1} - N \cdot N' = 1. \tag{2.7}$$

The integers R^{-1} and N' can both be computed by the extended Euclidian algorithm [9]. The Montgomery reduction algorithm which computes $AR^{-1} \bmod N$ is given in the Algorithm 3.

Algorithm 3 Montgomery Reduction Algorithm.

Require: Integers $A = (a_{2k-1}a_{2k-2}\cdots a_0)_b < NR$, $N = (n_{k-1}n_{k-2}\cdots n_0)_b$ with $\gcd(N,b) = 1$, $R = b^k$, $N' = -N^{-1} \bmod b$.
Ensure: $AR^{-1} (\bmod N)$

```
T := A
for i = 0 to (k − 1) do
    q_i := t_i N' mod b
    T := T + q_i N b^i
end for
T := T / b^k
if T ≥ N then
    T := T − N
end if
return T
```

Montgomery reduction of the product of two integers, namely Montgomery multiplication is outlined in Algorithm 4. Montgomery multiplication algorithm pieces Montgomery reduction and ordinary multiplication up together. In its reduction steps, firstly, a multiple of the modulus is determined by the least significant digit of the

partial sum and partial product. This multiple is then added to the partial sum making the least significant digit 0 which means a trivial right shift is applicable for reduction.

Algorithm 4 Montgomery Multiplication Algorithm.

Require: Integers $X = (x_{k-1}x_{k-2}\cdots x_0)_b$, $Y = (y_{k-1}y_{k-2}\cdots y_0)_b$, $N = (n_{k-1}n_{k-2}\cdots n_0)_b$ with $0 \leq X,Y < N$, $R = b^k$ with $\gcd(N,b) = 1$ and $N' = -N^{-1} \bmod b$.
Ensure: $XYR^{-1} \pmod N$

```
A := 0 (Notation: A := (a_k a_{k-1} ··· a_0)_b.)
for i = 0 to (k − 1) do
    q_i := (a_0 + x_i y_0) N' mod b
    A := (A + x_i Y + q_i N)/b
end for
if A ≥ N then
    A := A − N
end if
return A
```

Let q_i values are accumulated throughout the Montgomery multiplication and stored to constitute the Q' value. As mentioned in the previous section, let A, B, N and Q' are represented with $k = 2h$ bits for some positive integer h. They can be partitioned into their least and most significant h bits as:

$$A = A_1 \cdot 2^h + A_0 = A_1 \cdot r + A_0$$
$$B = B_1 \cdot 2^h + B_0 = B_1 \cdot r + B_0$$
$$N = N_1 \cdot 2^h + N_0 = N_1 \cdot r + N_0$$
$$Q' = Q'_1 \cdot 2^h + Q'_0 = Q'_1 \cdot r + Q'_0$$

where $r = 2^h$. Operation of Montgomery multiplication, namely left-to-right modular multiplication, can be illustrated as shown in the Figure 2.2 [4].

As illustrated in the Figure 2.2, Montgomery multiplication combines multiplication and reduction stages. In order to get the result $ABr^{-2} \bmod N$, $Q'N = Q'_0N + Q'_1Nr$ is added to the product of AB. Note that, during the first h cycles of the algorithm Q'_0N is added to the product of AB and the most significant h bits of AB, that is shaded in the figure is not modified.

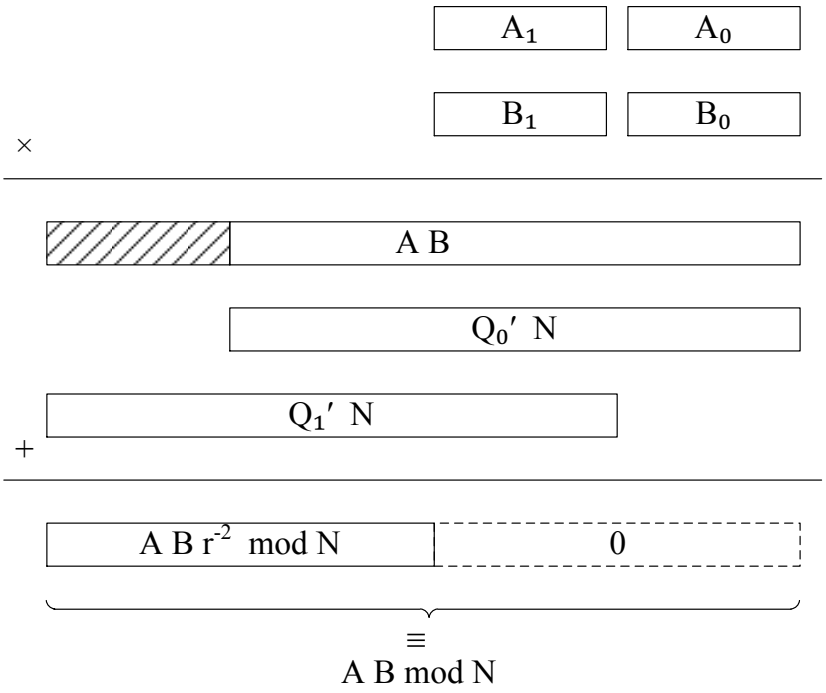

Figure 2.2: Right-to-Left Modular Multiplication, adapted from (Saldamlı, 2011).

2.3.3 Bipartite Modular Multiplication

Bipartite Modular Multiplication(BMM) method introduced by Kaihara and Takagi in [6] and [7], presents a semi parallel reduction based on an observation that a product could simultaneously be reduced from left and right without a dependency issue. Although, the dependency exists within each direction, BMM algorithm outlines a global method of parallel reduction [4], [5].

In the previous sections left-to-right and right-to-left reduction approaches were explained. In order to boost the speed up, BMM method links these two approaches by setting the R parameter in Montgomery algorithm less than the modulus N. This condition enables the multiplier to be split into two parts, which can be processed separately in parallel [7].

Let $X = (x_{k-1}x_{k-2}\cdots x_0)_b$ and $Y = (y_{k-1}y_{k-2}\cdots y_0)_b$ be k-digit N-residue integers with $N = (n_{k-1}n_{k-2}\cdots n_0)_b$, $0 \leq X,Y < N$, $\gcd(N,b) = 1$, $R = b^t$ and $0 < t < k$. Now consider the multiplier B in Montgomery multiplication of $ABR^{-1} \bmod N$ to be split into two parts B_H and B_L , so that $B = B_H \cdot b^t + B_L$. Then, Montgomery multiplication of the N-residue integers A and B with respect to $R = b^t$ can be computed as follows:

$$
\begin{aligned}
ABR^{-1} \bmod N &= A(B_H \cdot R + B_L)R^{-1} \bmod N \\
&= (AB_H RR^{-1} + AB_L R^{-1}) \bmod N \\
&= (AB_H \bmod N + AB_L R^{-1} \bmod N) \bmod N
\end{aligned}
$$

The left term inside the last parentheses, $AB \bmod N$, can be calculated using the classical modular multiplication algorithm that processes the upper part of the split multiplier B_H. The second term, $AB_L R^{-1} \bmod N$, can be calculated using the Montgomery algorithm that processes the lower part of the split multiplier B_L [7]. Since left-to-right and right-to-left reductions do not have any dependency during the first half of their reduction steps as stated in Blakley and Montgomery sections, they could be combined as seen in Figure 2.3. In fact, the figure gives a sketch of the bipartite reduction [5].

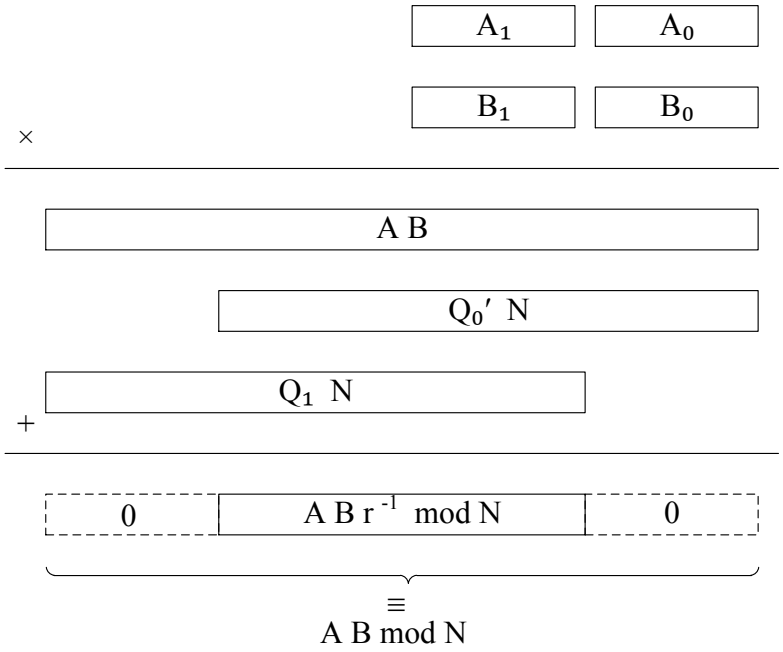

Figure 2.3: Bipartite Modular Multiplication, adapted from (Saldamlı, 2011).

In the Fig. 2.3, $Q'_0 N$ represents the reduction value that is to be added to the lower part of the product AB in Montgomery multiplication and $Q_1 N$ corresponds to the reduction

value which is subtracted from the upper part of the product in Blakley multiplication. They can be applied in parallel without any dependency issue.

2.4 Partially Interleaved Modular Karatsuba-Ofman Multiplication

Partially Interleaved Modular Karatsuba-Ofman(KO) Multiplication was proposed by Gökay Saldamlı. His algorithm [4] combines KO multiplier and bipartite modular reduction and presents an interleaved processing on the uppermost layer of KO's recursion. As stated in the introduction section, this thesis proposes two hardware implementations which are based on Saldamlı's algorithm and applicable improvements to the algorithm. In the following subsections, preliminaries will be given related to this algorithm.

2.4.1 Karatsuba-Ofman multiplication

Karatsuba-Ofman algorithm [3] presents a recursive method that requires asymptotically fewer bit operations than the standard multiplication. For a brief explanation, firstly, decompose A and B into two equal-size parts:

$$A := 2^h A_1 + A_0,$$
$$B := 2^h B_1 + B_0,$$

i.e., A_1 and A_0 represent the most and least significant h bits of A respectively, assuming k is even and $2h = k$ [4], [5]. The Karatsuba-Ofman multiplication algorithm breaks the multiplication of A and B into multiplication of the parts A_1, A_0, B_1 and B_0 [8]. Since,

$$\begin{aligned} T &:= A \cdot B \\ &:= (2^h A_1 + A_0)(2^h B_1 + B_0) \\ &:= 2^{2h}(A_1 B_1) + 2^h(A_1 B_0 + A_0 B_1) + A_0 B_0 \\ &:= 2^{2h} T_2 + 2^h T_1 + T_0 \ . \end{aligned}$$

standard multiplication of two $2h$-bit numbers seems to require the multiplication of four h-bit numbers, Karatsuba-Ofman algorithm is based on the observation that only three multiplications suffice to achieve the same purpose as seen in

$$
\begin{aligned}
T_0 &:= A_0 \cdot B_0 \\
T_2 &:= A_1 \cdot B_1 \\
T_1 &:= (A_0 + A_1)(B_0 + B_1) - T_0 - T_2 \\
&:= A_0 \cdot B_1 + A_1 \cdot B_0 \ .
\end{aligned}
$$

Karatsuba-Ofman recursive multiplication algorithm(KORMA) is shown below [8]:

---

function $KORMA(A,B)$
$T_0 := KORMA(A_0, B_0)$
$T_2 := KORMA(A_1, B_1)$
$U_0 := KORMA(A_0 + A_1, B_0 + B_1)$
$T_1 := U_0 - T_0 - T_2$
return $(2^{2h}T_2 + 2^h T_1 + T_0)$

Note that, one has the option of stopping at any point during the recursion [4]. For example, one level of recursion can be applied first and then required three multiplications can be computed using the standard nonrecursive multiplication algorithm. Operation of KO multiplication can be seen in the Figure 2.4.

2.4.2 Interleaving BMM and KO multiplication

As mentioned in the introduction section, because of the reduction algorithms not allowing parallel reduction due to data dependency issues, these methods can not be interleaved with fast multipliers. But, Saldamlı's algorithm [4] solves this problem to an extend by interleaving bipartite reduction with Karatsuba-Ofman multiplier on the uppermost layer of KO recursion [5].

Q, Q' and N in Bipartite reduction can be rewritten as:

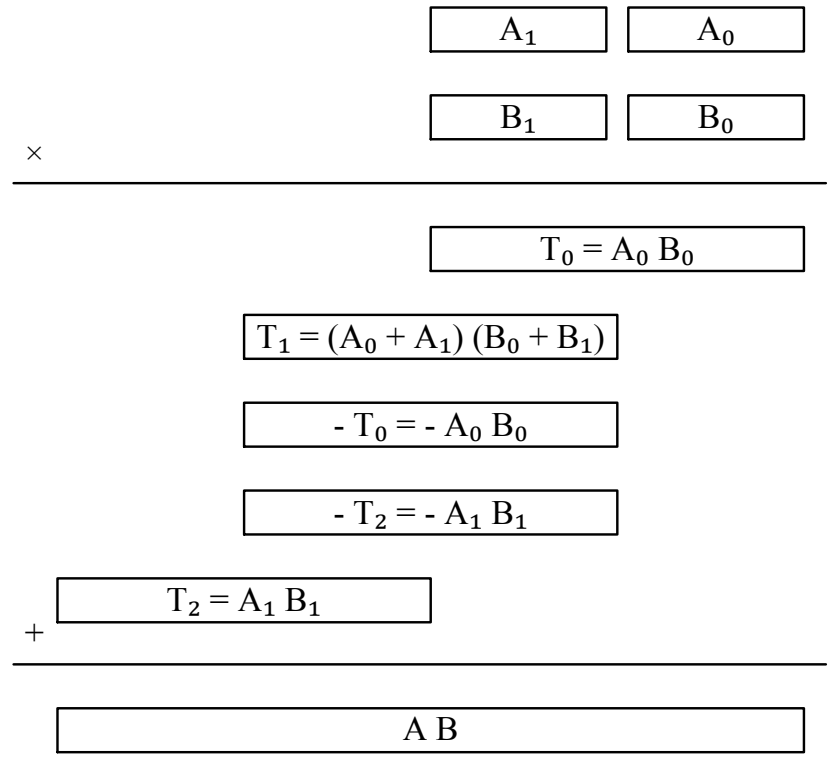

Figure 2.4: Karatsuba-Ofman Multiplication, adapted from (Saldamlı, 2011).

$$N := 2^h N_1 + N_0$$
$$Q := 2^h Q_1 + Q_0$$
$$Q' := 2^h Q'_1 + Q'_0$$

Following partial products are defined as:

$$T'_0 \cdot r = A_0 \cdot B_0 - Q'_0 \cdot N_0 = T_0 - Q'_0 \cdot N_0$$
$$T'_2 = A_1 \cdot B_1 - Q_1 \cdot N_1 = T_2 - Q_1 \cdot N_1$$

Using these values, T_1' can be calculated as follows [4];

$$\begin{aligned}
T_1' &= T_1 + T_0' + T_2' r - Q_0' N_1 - Q_1 N_0 \\
&= (A_0 + A_1)(B_0 + B_1) - T_0 - T_2 + T_0' + T_2' r - Q_0' N_1 - Q_1 N_0 \\
&= (A_0 + A_1)(B_0 + B_1) - (T_0' r + Q_0' N_0) - (T_2' + Q_1 N_1) + T_0' + T_2' r - Q_0' N_1 - Q_1 N_0 \\
&= (A_0 + A_1)(B_0 + B_1) - (Q_0' + Q_1)(N_0 + N_1) + T_0' - T_0' r - T_2' + T_2' r \; .
\end{aligned}$$

In fact, T_1' gives the desired modular reduction, $ABr^{-1} \bmod N$. For a simpler explanation of T_1' calculation, remember the Fig. 2.3 of Bipartite reduction. In order to compute the result $ABr^{-1} \bmod N$, Montgomery and Blakley reduction values $Q_0'N$, $Q_1 N$ had to be subtracted respectively from the product AB. As Karatsuba-Ofman calculates the product AB using the following equation:

$$\begin{aligned}
A \cdot B &:= 2^{2h} T_2 + 2^h T_1 + T_0 \\
&:= 2^{2h} A_1 B_1 + 2^h ((A_0 + A_1)(B_0 + B_1) - T_2 - T0) + A_0 B_0
\end{aligned}$$

Subtracting the Bipartite reduction values $Q_0'N$ and $Q_1 N$ from KO product AB gives the desired modular reduction of $ABr^{-1} \bmod N$. Partially Interleaved Modular KO multiplication is shown in the Figure 2.5.

As shown in the Fig. 2.5, firstly $Q_0'N_0$ and $Q_1 N_1$ are subtracted from $T_0 = A_0 B_0$ and $T_2 = A_1 B_1$ respectively. And $T_0' r$ and T_2' are computed as results. These operations are simply half-sized reductions of right-to-left(Montgomery) and left-to-right(Classic). Remember that Partially Interleaved Modular KO multiplier combines bipartite reduction with KO multiplication. As bipartite reduction values are $Q_0'N = Q_0'N_1 r + Q_0'N_0$ and $Q_1 N = Q_1 N_1 r + Q_1 N_0$, half of these reduction values are($Q_0'N_0$ and $Q_1 N_1$) used up here and $Q_0'N_1 r$ and $Q_1 N_0$ are left. T_1 and $-Q_0'N_1$ and $-Q_1 N_0$ are summed up with the calculated T_2 and T_0' values to give the modular reduction, $ABr^{-1} \bmod N$. These operations are rearranged in the second half of the figure. At it can be seen, the desired modular reduction can be computed as: $ABr^{-1} \bmod N = (T_2' r \& T_0') + (A_0 + A_1)(B_0 + B_1) - (Q_0' + Q_1)(N_0 + N_1) - (T_0' r \& T_2')$, where & is the concatenation operation.

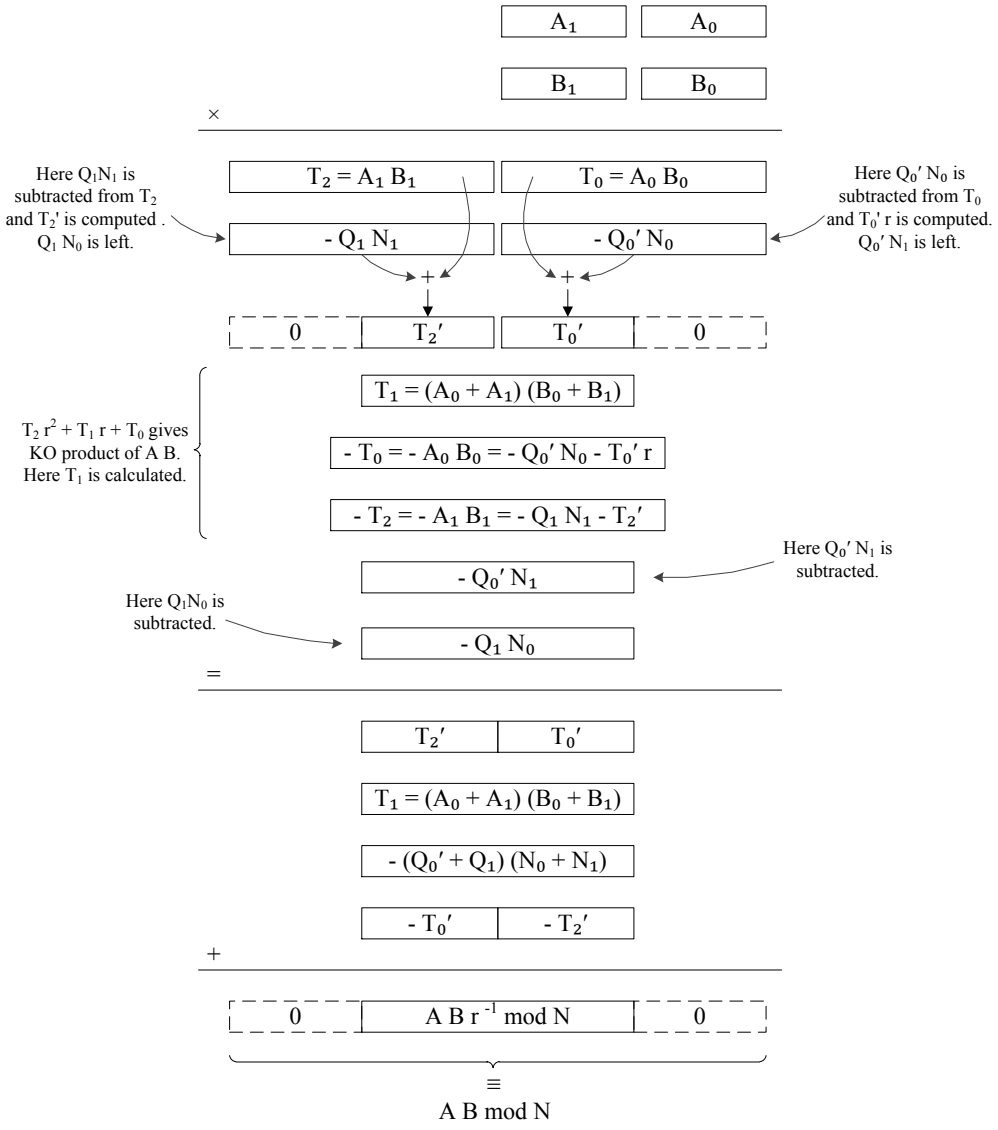

Figure 2.5: Partially Interleaved Modular Karatsuba-Ofman Multiplication, adapted from (Saldamlı, 2011).

Block diagram of the method can be seen in Figure 2.6. As shown in the figure, firstly using the half-sized left-to-right interleaved multiplication and right-to-left interleaved multiplication T_2', T_0', Q_1 and Q_0' are calculated. Then, standard multiplication operations $(A_0+A_1)(B_0+B_1)$ and $(Q_0'+Q_1)(N_0+N_1)$ are done. And finally, additions are made in order to get the modular multiplication result.

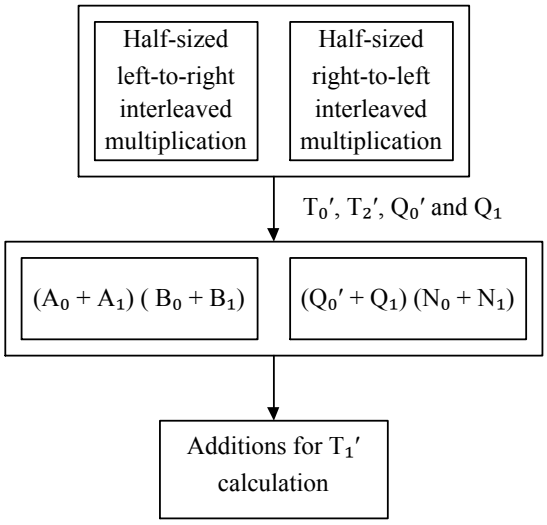

Figure 2.6: Block Diagram of the Method, adapted from (Saldamlı, 2011).

Neglecting the addition operations, implementation of this method can be done by using Classic modular multiplication for left-to-right reduction, Montgomery multiplication for right-to-left reduction and integer multiplication for the multiplication operations of $(A_0+A_1)(B_0+B_1)$ and $(Q_0'+Q_1)(N_0+N_1)$. In the next section, preliminaries about the multiplication algorithms will be presented.

2.5 Integer Multiplication

Various multiplication algorithms exist in the literature in order to perform fast multiplication. Examples of such methods are Karatsuba-Ofman [3], Schönhage-Strassen [13] and Fürer [14]. Among these, Karatsuba-Ofman multiplication was explained in Section 2.4.1. In order to use the similar hardware structures and balance the delays of hardware modules, basic shift-and-accumulate method was accomodated in this work. Shift-and-accumulate approach is explained in the following subsection.

2.5.1 Shift and accumulate method

Let multiplicand A, multiplier B be k-bit, and product P be $2k$-bit numbers as follows:

$$A := (a_{k-1}a_{k-2}\cdots a_0)$$
$$B := (b_{k-1}b_{k-2}\cdots b_0)$$
$$P := AB = (p_{2k-1}p_{2k-2}\cdots p_0)$$

Figure 2.7 shows the multiplication of two 4-bit unsigned binary numbers in dot notation [15]. Multiplicand A and multiplier B are shown at the top. Each of the following four rows of dots corresponds to the product of A and a single bit of B, where each dot represents the product of two bits. Since numbers are in binary form, each dot can either be 0 or 1 and each row can either be A or 0. Thus the binary multiplication problem reduces to adding a set of numbers, each of which is 0 or a shifted version of the multiplicand A.

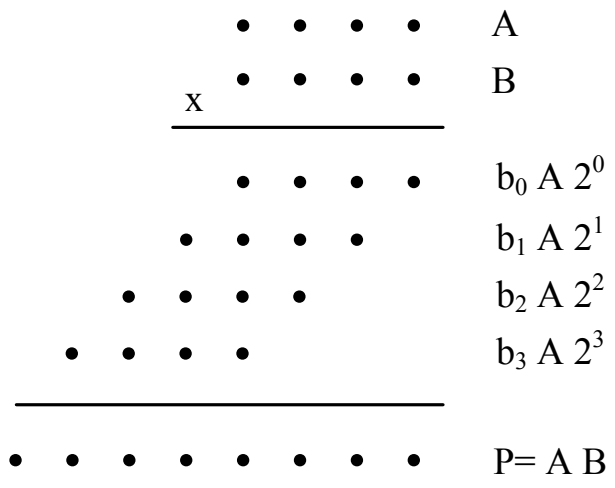

Figure 2.7: Shift and Accumulate Method, adapted from (Parhami, 2000).

Sequential or bit-at-a-time multiplication can be done in two directions, either from left-to-right, or right-to-left. In case of left-to-right, a cumulative partial product, which

is initialized to 0 is shifted left at each step and proper b_iA is added to it. Left-to-right operation can be shown as:

$$P = ((((P + b_3A) \cdot 2 + b_2A) \cdot 2 + b_1A) \cdot 2 + b_0A), (P = 0 \text{ initially})$$

In right-to-left operation, at each step, a cumulative partial product, again initialized to 0 is summed up with the proper b_iA and least significant bit of it is stored in another variable, namely right part of the product. This operation goes on until the end of the multiplication. At the end, right part of the product and cumulative partial product is concatenated to get the product.

Here, number of partial products is equal to the bit length of the multiplier. As multiplier is k-bit long, product P is computed via addition of k generated partial product one by one. This operation can be accelerated by grouping the bits of the multiplier into pairs, triples or quadruples and so on. The way of grouping the multiplier bits widens the partial product set and bring advantages and disadvantages. A very famous encoding method, namely Booth Encoding which groups the multiplier bits effectively will be presented in the following section.

2.6 Booth Encoding

A generator that creates a smaller number of partial products will allow the partial product summation to be faster and use less hardware [16]. A basic way of reducing the number of partial products could be grouping the multiplier bits into pairs. In this way, partial products would be selected from the set of $\{0, A, 2A, 3A\}$, where A is the multiplicand. This reduces the number of partial products by half but brings a problem: generation of $3A$. 0, A, $2A$ are very easy to generate. But, $3A$ generation needs either $2A + A$ precomputed and stored, or on the fly calculation. Instead of this, a method known as Modified Booth's Algorithm [17], [16] is used, which reduces the number of partial products by about a factor of two and does not require $3A$ neither to be precomputed, nor on the fly computation. The idea of Booth's algorithm is doing a little more work when decoding the multiplier such that the required multiples of multiplicand come from the set of $\{0, A, 2A, 4A + -A\}$. All of the elements of this set can be generated by simple shift operations. This method works by replacing any use of

Table 2.4: Modified Booth Encoding Scheme.

b_{i+1}	b_i	b_{i-1}	I		b_{i+1}	b_i	b_{i-1}	I
0	0	0	0		1	0	0	-2A
0	0	1	A		1	0	1	-A
0	1	0	A		1	1	0	-A
0	1	1	2A		1	1	1	0

3A either by $4A-A$. Depending on the adjacent multiplier bits either 4A is pushed into the next most significant group(becoming A because of the different arithmetic weight of the group) or $-A$ is pushed into the previous least significant group, becoming $-4A$ [16]. Figure 2.8 shows the dot diagram of 8x8 multiplication using the 2 bit version of the algorithm.

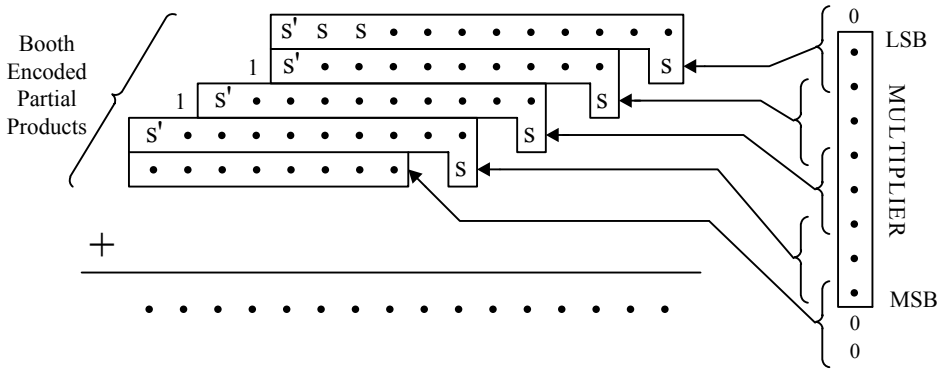

Figure 2.8: 8x8 Modified Booth Multiplication, adapted from (Bewick, 1994).

As shown in the Fig. 2.8, multiplier is partitioned into overlapping groups of 3 bits. Each group starting from LSB is decoded to select a single partial product(I) according to the selection Table 2.4. Corresponding multiple of the multiplicand is determined according to the addition of $(-2\cdot b_{i+1}+b_i+b_{i-1})$, where b_{-1}, b_k and b_{k+1} bits are padded with 0. Booth encoded partial products are showns as horizontal row of dots in the figure. Each partial product is shifted 2 bit positions with respect to its neighbours. Number of partial products are reduced to 5, instead of 8. In general, the number of partial products is $\lfloor\frac{k+2}{2}\rfloor$, where k is the length of the multiplier [16].

All partial products shown in the Table 2.4 can be generated by simple operations like shift and complement. Negative partial products can be easily generated by bit-by-bit complementing the corresponding positive product and adding 1 to the least significant

position of the partial product, which is shown as s at the LSB position of partial products.

In the Fig. 2.8, it can be seen that some partial products' MSB positions are padded with s, s' or 1. This padding is called as *sign extension*. As Booth encoding scheme generates both positive and negative multiples of the multiplicand, it is difficult to control the sign and guarantee the product to have the desired sign. Incorporating sign extension solves this problem by padding s, s' and 1 to corresponding partial products' MSB positions. The logic of sign extension is providing a general form, which works in any combination of positive and negative multiples.

The partial products for the 8x8 multiplication example, assuming that all partial products are positive, are shown in the Figure 2.9. Each partial product, except for the bottom one, is 9 bits long, since numbers as large as 2 times the multiplicand must be dealt with [16]. The bottom partial product is 8 bits long, because multiplier is padded with two zeroes in order to guarantee the positive result.

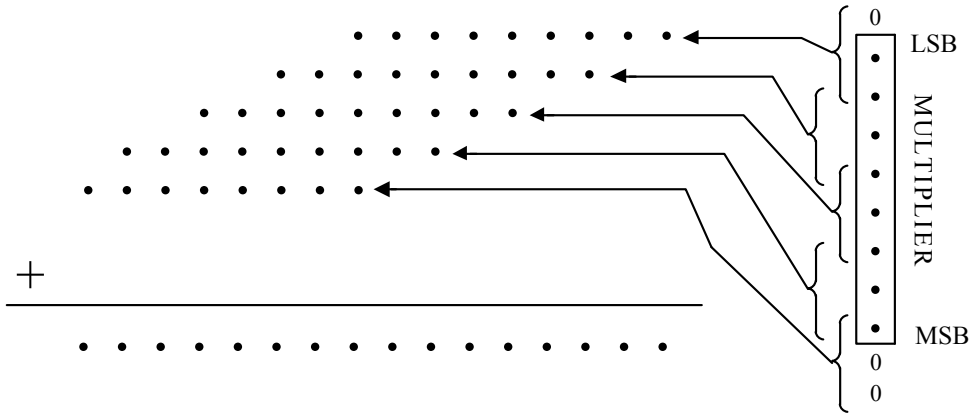

Figure 2.9: 8x8 Booth Multiplication with Positive Partial Products, adapted from (Bewick, 1994).

Figure 2.10 shows the partial products if they all happen to be generated negative. Using 2's complement, every bit of the negated partial products is complemented, including any leading zeroes and 1 is added at the LSB. The bottom partial product is never negated, because zero padding assures that it is always positive [16]. Triangle of 1's on the left handside can be summed to produce the Figure 2.11, which is exactly equivalent to the situation shown in Fig. 2.10.

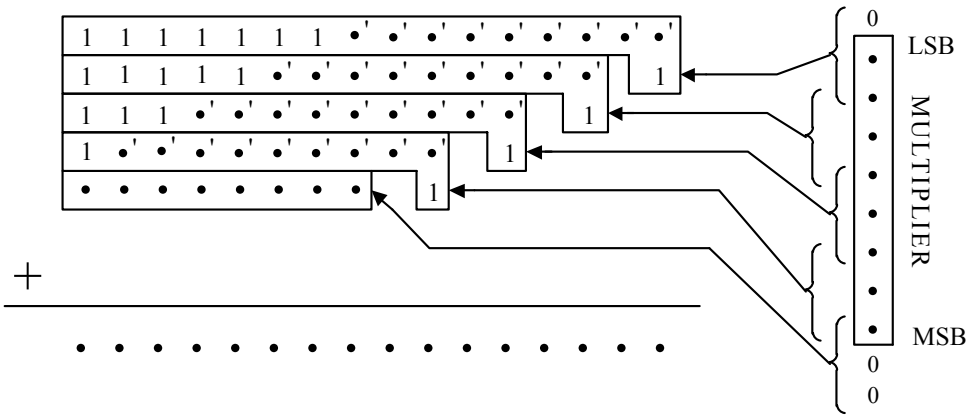

Figure 2.10: 8x8 Booth Multiplication with Negative Partial Products, adapted from (Bewick, 1994).

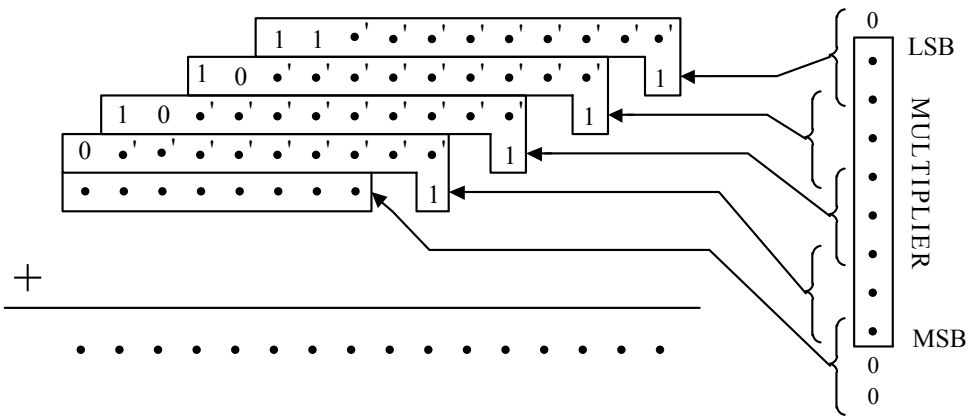

Figure 2.11: Negative Partial Products with Summed Sign Extension, adapted from (Bewick, 1994).

Now, suppose that a particular partial product turns out not be negative. The leading string of 1's in that particular partial product can be converted back to a leading of zeroes, by adding a single 1 at the least significant bit of the string. In addition to this, a 1 is added into the least significant bit of a partial product, only if it is negative. Figure 2.8 illustrates this configuration. The s' bits represent the 1's that are needed to clear the sign extension bits for positive partial products, and the s bits represent the 1's that are added at the least significant bit of each partial product if it is negative [16].

3. IMPLEMENTATION ENVIRONMENT

In this section, implementation environment will be described. Implementation environment includes device technologies(FPGA and ASIC), and design & verification tools which were used throughout the hardware implementations.

Two hardware designs are proposed in this thesis. The first one is FPGA implementation of Partially Interleaved Modular KO Multiplier [5]. The second one is ASIC implementation of high radix and optimized version of the first design and incorporation of it in RSA cryptosystem. VHDL was used as a hardware description language. Before coding of the design elements in VHDL, Maple library of building blocks and primitives, which emulate the hardware components, were prepared. Then, same hardware designs were implemented in Maple using these building blocks. Maple implementation of the design is tested and verified. In the next step, design was coded in VHDL. According to the implementation technology, Xilinx(for FPGA implementation), or Synopsys tools(for ASIC implementation) were used for synthesis, mapping, placing and routing operations. Behavioral and gate level simulations were performed using Modelsim and VCS simulators. Verification of correct operation was done according to the results computed in Maple and results produced in simulations. In the following subsections these device technologies and design tools will be briefly described.

3.1 Device Technologies

There are varieties of device technologies which can be chosen to build a custom digital system. Designer has to consider the trade-offs among various factors, including chip area, speed, power consumption and cost.

Device technologies can be classified according to the customization method. The customization of a circuit can be performed "in the field", by downloading a connection pattern, also known as *programming file* to the device's internal memory.

These devices are *Field Programmable Gate Arrays(FPGA)* and *Complex Field Programmable Logic Devices(CPLD)*. On the other hand, some device technologies need one or more patterned layers(silicon, polysilicon, metal, etc.) to be fabricated. This process is expensive and complex and can only be done in a fabrication plant(known as a foundry or a fab). Device technologies requiring a fab to do customization is called as *Application Specific Integrated Circuit(ASIC)* [18].

3.1.1 FPGA

Field Programmable Gate Array devices(FPGA) were introduced by Xilinx in mid 1980s [19]. Designers benefit from its features of flexibility, low cost, high performance and short configuration time. The basic architecture of an FPGA is shown in Figure 3.1 [20].

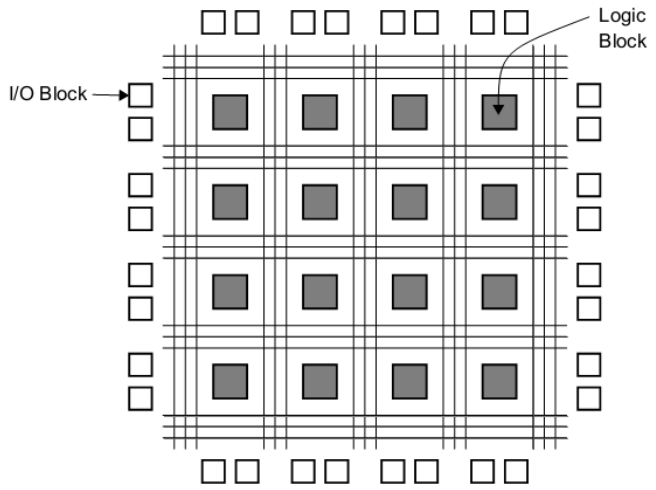

Figure 3.1: FPGA Architecture, adapted from (Brown, 1996).

As illustrated in Fig. 3.1, FPGA architecture consists of matrix of configurable logic blocks(CLB). These blocks are connected to each other via vertical and horizontal routing channels which are shown as a mesh network in the figure. CLBs and routing channels are enclosed with programmable input/output blocks. CLBs usually accomodate several logic gates, lookup tables, multiplexers, adders and flip-flops. An FPGA may also have different CLBs which may be memory blocks, or multiplier blocks, etc.

3.1.2 ASIC

Application Specific Integrated Circuits(ASIC) in an integrated circuit(IC) customized for a particular use, rather than intended for general-purpose use [21]. As it is customized for a special application, it provides efficient use of chip area, low power dissipation and usually the smallest propagation delay and best speed.

ASICs can be *full-custom*, *gate-array* or *standard-cell* based. In this work Taiwan Semiconductor Manufacturing Company(TSMC) 90 nm standard-cell library is used. With this reason, standard-cell ASIC will be explained.

In standard-cell ASIC technology, a circuit is constructed by using a set of predefined logic components, known as *standard cells*. These cells are predesigned and their layouts are validated and tested. Standard-cell ASIC technology allows a designer to work at the gate level rather than at the transistor level, thus greatly simplifies the design process. The device manufacturer provides a library of standard cells as the basic building blocks. This library consists of basic logic gates, combinational components and memory elements [18]. Standard cell libraries may target different design challenges. For instance, a standard cell library might have been specially designed for low-power consumption and another library may be full of cells which can operate in very high frequencies.

3.2 Maple

Maple is a mathematical software which can be used for differentiation, integration, finding limits, solving equations, working with matrices and polynomials. All of these functionalities of it can be integrated with modules and libraries coded in Maple.

As stated in Sec. 2.1, RSA algorithm requires modular multiplications with very large integers, i.e. 1024-bit. When hardware is designed, verification of correct operation becomes a very difficult process as operands in the design are such long integers. For example, finding the error which causes i'th bit of a 512-bit operand can be such a big problem if there is not a way to compare partial results bit-by-bit in every layer of the design. The reasons of using Maple in this work are:

- Discovering possible design problems before description of hardware,

- Having a functionally correct working software emulation of the same hardware in hand,
- Using Maple result for debugging the hardware simulations,
- Identifying the upper limits of operands in hardware and using this information to describe a more efficient hardware.

Incorporating Maple allows a designer to manage all of the advantages above. In fact, any programming languages and any software development environment can provide the similar functionality. But, Maple makes it simpler and provides extra functions in its libraries which can be used with polynomials, matrices, equations, etc.

Maple procedures are described as:

```
procedure_name := proc( list_of_parameters )
  variable declarations
  functionality
  ...
end proc:
```

Procedures are Maple modules. They can be written with any aim. For example a procedure may describe a simple *print* operation, or a relatively more complex encryption operation. Maple implementation of *AND* and *XOR* gates and a *half adder* are shown below:

```
#AND gate
AND:=proc(x,y)
  return min(x,y);
end proc:

#XOR gate
XOR:=proc(x,y)
  return (max(x,y)-min(x,y));
```

```
end proc:

#Half adder
HA:=proc(x,y)
  local s,c;
  s:=XOR(x,y);
  c:=AND(x,y);
  return(c,s);
end proc:
```

There are four Maple procedures defined above. As AND gate in hardware produces logic 1 when its inputs are $(1,1)$, and logic 0 for the rest, Maple procedure *AND* emulates this behavior by means of Maple function *min*, which gives the smaller one of the operands x and y there. This procedure gives 1 as output only when both x and y are 1 and 0 to the rest of inputs, which means there is at least one 0 among operands which is always the minimum value.

In *HA* procedure, previously defined modules(XOR and AND) are used to produce sum s and carry c values. This is simply the operation of a half adder, accepting two binary inputs and producing sum and carry as outputs. Example instantiation of *HA* module is given below, where d stores the sum and carry values of $a+b$ and e stores sum and carry values of $1+0$.

```
a:=1;
b:=1;
d:=HA(a,b);
e:=HA(1,0);
print("d=a+b",d);
```

```
                                    a := 1

                                    b := 1
```

```
d := 1, 0

e := 0, 1

"d=a+b", 1, 0
```

3.3 FPGA Design Tools

Partially Interleaved Modular KO multiplication was implemented on Xilinx Virtex 5 XCVFX130T FPGA using Xilinx ISE Design Suite. ISE software provides tools for HDL coding of the design, synthesis of a design description, implementation of the synthesized design specific to the selected FPGA device, programming the FPGA. In addition to these features, ISE can generate various reports related to timing, area and power dissipation analyses which specify the performance of the design. Note that Xilinx ISE can be used only for Xilinx FPGA devices. Other FPGA manufacturers provide similar tools to carry out the same operations.

The first step of the FPGA design is the *Design Entry* step, where design is described in an HDL. FPGA family, device, package and speed parameters are determined in this step. HDL source files are created and checked against any syntax errors. This is called as *Register Transfer Level(RTL)* description. Before the *Design Synthesis* step, design can be functionally verified by behavioral simulation. Behavioral simulation, which can be performed with simulators like ISim, Modelsim, VCS verifies the functionality of the design without taking delays into account.

When a design's functionality is verified, then the next step is the *Design Synthesis* step. In this step, ISE software allows a designer to use either Xilinx's synthesis tool of *XST* or different synthesis tools like Design Compiler of Synopsys. Various parameters can be set here related to optimization goal and effort, hiearchical options, utilization of FPGA resources, power reduction, etc. When synthesis tool is run, it performs RTL-level synthesis. The synthesizer converts HDL (VHDL/Verilog) code into a gate-level netlist (represented in the terms of the UNISIM component library, a

Xilinx library containing basic primitives) [22]. After the synthesis step, post-synthesis simulations can be made via simulation tools.

Design Implementation step consists of translate, map, place&route operations. During the translate phase previously synthesized gate-level netlist is converted to another netlish which is represented in the terms of the SIMPRIM component library of Xilinx. In the mapping phase, SIMPRIM primitives of the netlist are mapped on specific device resources, like lookup tables, flip-flops, etc. Place&route tool defines how device resources are located and interconnected inside an FPGA [22]. Again, several parameters can be set here about placing and routing efforts, optimization, etc. Post-translate, post-map and post-place&route simulations are performed after each step. Here, the most important one is the post-place&route simulation, where simulation is performed with the true delay information of the design.

After the implementation step, next step is generating a programming file and programming the FPGA. Note that, timing and placement constraints can be given to the implementation tool in order to direct the implementation process to give the desired results.

3.4 ASIC Design Tools

High radix and optimized version of Partially Interleaved Modular KO multiplier was implemented on ASIC 90nm TSMC technology library using Synopsys Design Vision. Design Vision is in fact a graphical user interface of Synopsys Design Compiler program. As all synthesis and implementation operations can be performed via scripts running on a terminal emulator, same operations can be done using Design Vision graphical user interface. Synopsys VCS simulator was used for behavioral, and post synthesis simulations.

Firstly VHDL source files of the design are created using an editor, i.e. a simple text editor. Then configuration files are prepared for Design Vision which point out the location of working director, standard cell libraries, logical library mappings, etc. In addition to this, a script file is created which will give constraint directives to Design

Compiler. Again several constraints can be given here. For example 'CLK_PERIOD' constraint, which sets the *clock*'s period to 0.9 nano second can be given as:

```
set CLK_PORT [get_ports clock]
set CLK_PERIOD 0.9
```

Before synthesis operations VCS simulator can be used to perform behavioral simulation. After the behavioral simulation, synthesis process starts by Desing Vision, reading and analyzing HDL files. In synthesis phase, Design Compiler uses technology libraries and other libraries(synthetic, symbol, etc.) to translate the HDL description to components extracted from the generic technology library which is independent from technology [23].

After translating the HDL description to gates, Design Compiler optimizes and maps the design to target library. This process is contraint driven. The result of the logic synthesis process is an optimized-gate level netlist, which is a list of circuit elements and their interconnections and ready for the place&route tools. [23].

4. HARDWARE IMPLEMENTATION

This section will propose two hardware designs which are based on the same modular multiplication algorithm, namely Partially Interleaved Modular KO Multiplication which was described in Sec. 2.4 and a very fast RSA implementation which utilizes the implemented modular multiplier. The first design, that was implemented on Xilinx Virtex 5 XCVFX130T FPGA, includes Radix-2 Montgomery and Radix-2 Classic Modular multipliers and Radix-4 Integer Multipliers. The second design implements a high radix and optimized version of the same algorithm on 90 nm ASIC technology. In includes Radix-4 Booth Encoded Montgomery Multiplier, Radix-4 Classic Modular Multiplier and Radix-8 Integer Multipliers. RSA was implemented using the high radix modular multiplier.

Although both of the implementations differ in many ways, they have some common properties. These properties are implementation organizations and a few hardware modules having the same structures. The first property is dot diagrams. Dot diagrams were created for every multiplier module. These diagrams, which will be explained in the next sections, provide a view to the designer. Designer can plan how to design the data path of the hardware easily by analyzing the dot diagram of the design. Another property that hardware implementations share is, adder structures. In both designs, *Carry Save Adder(CSA)* and *Carry Lookahead Adder(CLA)* were used. Before HDL coding and synthesis processes, designs were implemented and verified in Maple.

In the next sections, dot diagrams, adder structures and hardware implementations will be explained in detail.

4.1 Dot Diagrams

Figure 4.1 shows the dot digram for a simple 8-bit multiplication. Each dot in the diagram represents a single bit which can be zero or one. Partial products are represented by a horizontal row of dots. Each partial product is determined according

to the corresponding multiplier bit in that arithmetic weight [16]. If multiplier bit is 1, then partial product is equal to the multiplicand, and 0 if multiplier bit is 0. The partial products are shifted to account for the differing arithmetic weight of the bits in the multiplier, aligning dots of the same arithmetic weight vertically [16]. The product is shown below which is 16 bits long.

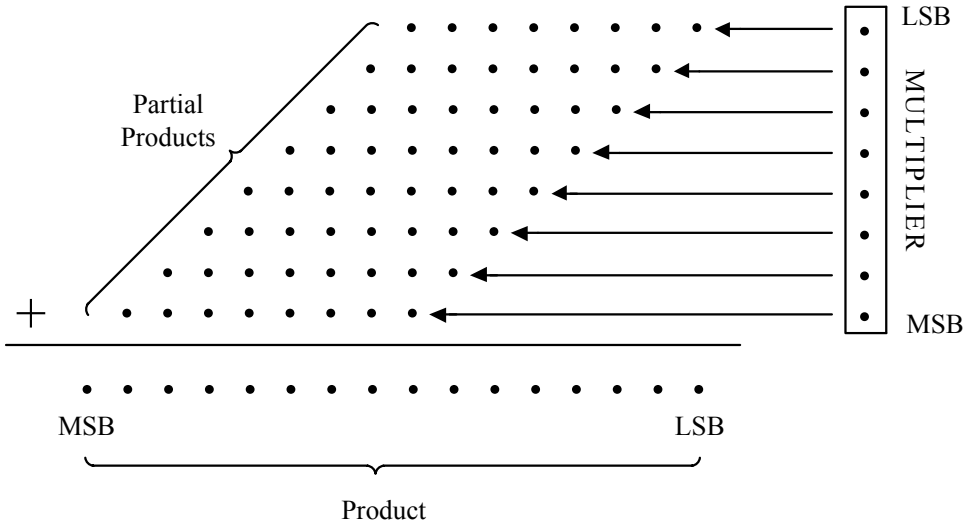

Figure 4.1: 8-bit Multiplication Dot Diagram, adapted from (Bewick, 1994).

Roughly speaking, the number of dots in the partial product section of the dot diagram is proportional to the amount of hardware required to sum the partial products and form the final product [16]. Dot diagram of Radix-2 Montgomery multiplication is shown in Figure 4.2 where multiplicand, multiplier and modulus N operands are 8 bits. Here, partial result is kept in sum and carry vectors. In every step of Montgomery algorithm a new partial product is generated. Then a specific value is determined according to the least significant bits of partial result and partial product, which is denoted as $Q' * N$ vector in the figure. Aim of $Q' * N$ is making the least significant bit of partial result 0, so that a right shift can be applied without any data loss. Here, dot diagram shows general operation of Montgomery. At each step, these vectors can be summed by utilizing a carry save adder(CSA) and a full adder(FA). At the end of the multiplication

steps, a ripple carry adder(RCA) can be used to determine the result of Montgomery multiplication.

Using dot diagrams, designer can understand the overall design process. These diagrams help designers to create parametric designs, to map parts of a design to hardware modules efficiently.

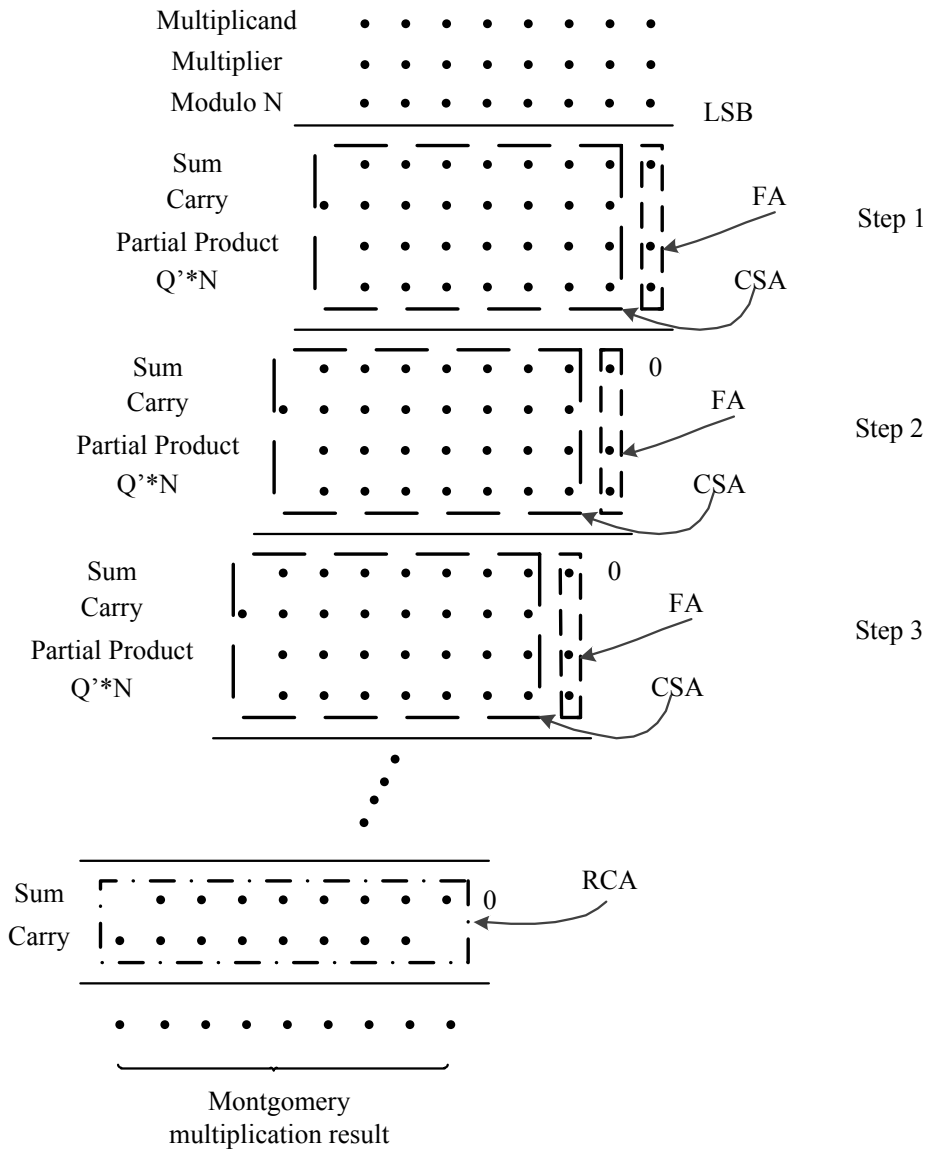

Figure 4.2: Radix-2 Montgomery Multiplication Dot Diagram.

4.2 Adder Structures

Addition operations constitute the body of multiplication operations. Multiplication is nothing more than series of partial product summations. In order to design a fast multiplier, one has to consider employing fast adders. In this section adders will be explained starting from basic addition blocks to fast adders.

Addition operations in this work are performed by adder modules which use basic adder circuits iteratively. Adder modules accept binary input vectors and produce binary output vectors. Same addition operation is applied to each bit position.

A straightforward implementation of an adder module to sum two binary vectors $X = (x_{k-1} \cdots x_2 x_1 x_0)$ and $Y = (y_{k-1} \cdots y_2 y_1 y_0)$ is achieved through the use of k basic units, called *full adders*(FA) [24]. A full adder accepts two operand bits x_i and y_i and an incoming carry bit c_i and then produces the corresponding sum bit s_i and an outgoing carry bit c_{i+1}. This outgoing carry bit is used as the incoming carry bit in the next FA, which accepts x_{i+1} and y_{i+1} as inputs. Boolean functions for the FA are as follows:

$$s_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i)$$

where $\oplus$ is the exclusive-or(XOR) operation, $\wedge$ is the AND operation and $\vee$ is the OR operation. A full adder and a ripple carry adder which sums two 4-bit binary vectors X and Y are shown in Figure 4.3.

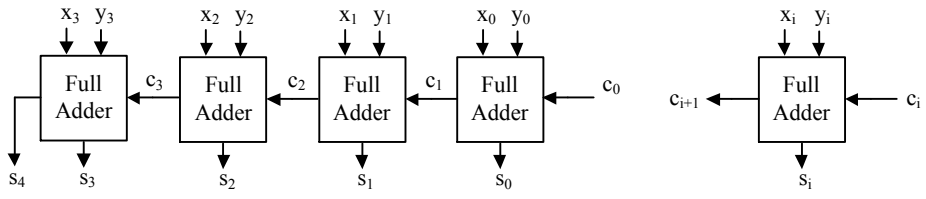

Figure 4.3: Full Adder and A 4-bit Ripple Carry Adder, adapted from (Koren, 2002).

As shown in the Fig. 4.3, in order to get correct result from ripple carry adder, carry produced in a FA has to propagate until the FA in the MSB position. In other words, one has to wait until the carries *ripple* through all FAs before claiming the output

correct. With this reason it is called as *ripple carry adder*. Note that in the ripple carry adder, first incoming carry, c_0 is 0 at the beginning of addition operation. This means that FA in LSB position can be changed with a simpler adder circuit, which adds two bits and produce two bits results, sum and outgoing carry. This can be accomplished by using an *half adder(HA)*, which accepts two input bits x_i and y_i and produces sum s_i and outgoing carry bit c_{i+1}. Boolean equations of an HA is:

$$s_i = x_i \oplus y_i$$

$$c_{i+1} = (x_i \wedge y_i)$$

The ripple carry adder, although simple in concept, has a long circuit delay due to the many gates in the carry path from LBS to MSB [25]. This delay may reach unacceptable values when size of operands increase and affect the performance of multiplier negatively. Eliminating this bottleneck is possible to a certain extend. Two approaches can be employed to achieve this:

- Partitioning carry propagation path into blocks and using a faster adder iteratively to perform additions in these smaller blocks,
- If producing one binary vector from summation of two or more vectors is unnecessary for a while, then keeping produced sum and outgoing carries in two separated vectors, namely *carry-save form*.

Both of these approaches were used in this work. For cyclic summation of binary vectors(more than 2 vectors) a special class of adders, *Carry Save Adders(CSA)* were used and partial result, which is represented by vectors of produced sum and outgoing carries, is kept in carry-save form as long as possible. When there is no more binary vectors left for addition, sum and carry vectors were partitioned into groups and result is produced by repetitive use of *Carry Lookahead Adders(CLA)* which are faster than basic ripple carry adders. CLA and CSA will be explained in the next sections.

4.2.1 Carry Lookahead Adder

The most commonly used scheme for accelerating carry propagation is the *carry-lookahead* scheme [24]. It is a practical design with reduced delay at the price of more complex hardware [25]. CLA reduces the delay by computing each carry bit independent from each other, which means in order to get the correct result no carry propagation between FAs are needed.

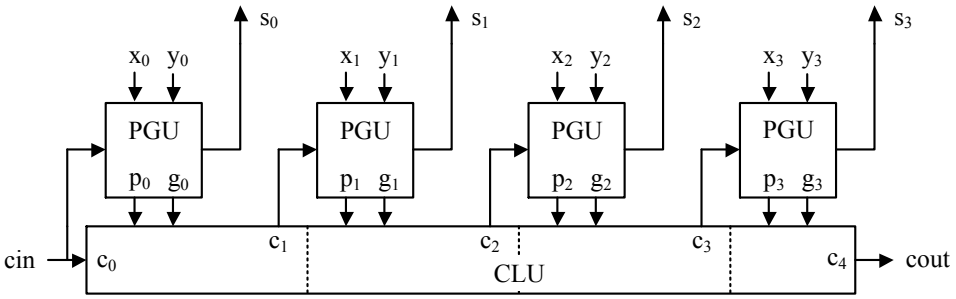

Figure 4.4: 4-bit Carry Lookahead Adder, adapted from (Pedroni, 2004).

A 4-bit CLA is shown in the Fig. 4.4. Implementation of CLA is based on the *generate* and *propagate* concept [19]. The *generate(g)* and *propagate(p)* signals for two input vectors $X = (x_{k-1} \cdots x_1 x_0)$ and $Y = (y_{k-1} \cdots y_1 y_0)$ are defined as:

$$g_i = x_i \wedge y_i$$

$$p_i = x_i \oplus y_i$$

Since these signals do not have any dependency with carry bits, they can be computed in advance. Now, consider the carry vector $C = (c_{k-1} \cdots c_1 c_0)$. Each carry bit can be computed from propagate and generate signals as:

$$c_0 = cin$$

$$c_1 = c_0 p_0 \vee g_0$$

$$c_2 = c_0 p_0 p_1 \vee g_0 p_1 \vee g_1$$

$$c_3 = c_0 p_0 p_1 p_2 \vee g_0 p_1 p_2 \vee g_1 p_2 \vee g_2, etc.$$

As it can be seen from equations above, each carry bit is computed independent from each other. No wait for carry propagation is required. This is in fact the reason of CLA, being faster than ripple carry adder. On the other hand, the hardware complexity grows very fast, limiting this approach to just a few bits, typically 4 [19]. These 4-bit units can be combined to implement larger adders.

As shown in the Fig. 4.4, the $PGU(Propagate-Generate\ Unit)$ computes p, g and sum bit s, and the $CLAU(Carry-Lookahead\ Unit)$ computes the carry bits.

4.2.2 Carry Save Adder

According to the mathematical operation, sometimes three or more vectors have to be added simultaneously. For example, consider the modular multiplication operation. At every step, in addition to the partial product, a reduction vector has to be added to the partial sum and this has to be repeated several times, until all multiplier bits are processed. Employing a ripple carry adder causes time-consuming carry propagation several times. The technique which is most-commonly used to lower the carry propagation is *carry save* addition [24]. Carry-save addition allows carry propagation only in the last step. In all other steps partial sum and generated carries are kept in two vectors separately. The basic carry-save adder(CSA) accepts three k-bit operands and produces k-bit sum and $k+1$-bit carry, which means a CSA can reduce the number of operands from 3 to 2 without waiting for carries.

The simplest method to implement a CSA is using full adders, which accepts three input bits and generate two output bits. CSAs are also called as counters. Because, the outputs of CSAs are the weighted binary representation of the number of 1s in the inputs [24]. A $(3,2)$ CSA consists of k FAs operating in parallel with no carry links between them is shown in Figure 4.5.

Another type of CSA, which accepts four binary vectors is $(4,2)$ CSA. A $(4,2)$ CSA consists of $(5;3)$ *compressors*, which is made up of two FAs connected to each other. A $(5;3)$ compressor accepts four operands and an incoming carry, and produces sum, carry and outgoing carry bits. A $(5;3)$ compressor and a $(4,2)$ CSA are shown in the Figure 4.6.

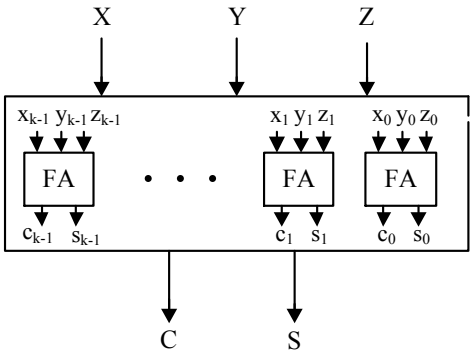

Figure 4.5: A (3,2) Carry Save Adder, adapted from (Koren, 2002).

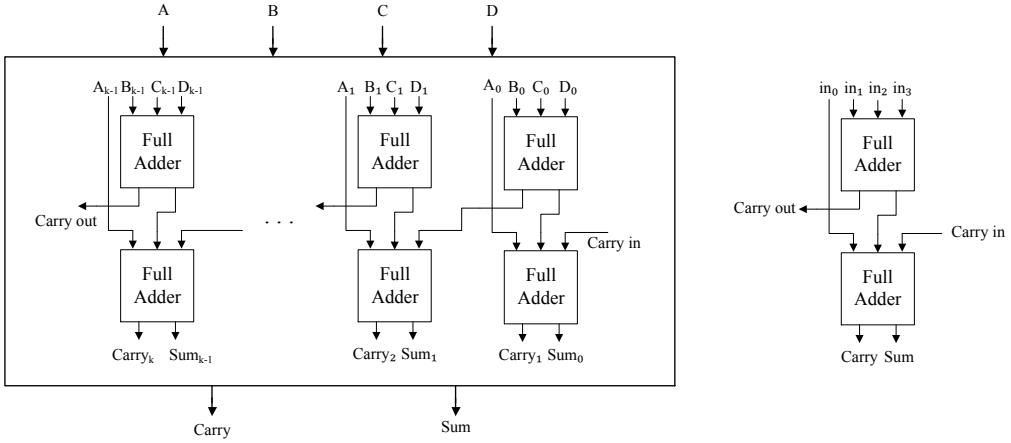

Figure 4.6: A (4,2) Carry Save Adder and (5;3) Compressor.

In this work, both $(3,2)$ CSA and $(4,2)$ CSA were used in modular multipliers and integer multipliers in order to eliminate the delay of carry propagation until the very last addition. In the next section, hardware implementations will be explained.

4.3 Partially Interleaved Modular KO Multiplier

Hardware implementation of Partially Interleaved Modular KO Multiplication was done using separate multiplier modules. To be more specific, classic(Blakley) modular multiplier, Montgomery multiplier and integer multipliers were utilized for left-to-right, right-to-left and integer multiplications respectively [5]. Block diagram of the method was shown in Fig. 2.6. Hardware implementation of Partially Interleaved Modular KO Multiplier can be seen in Fig. 4.7.

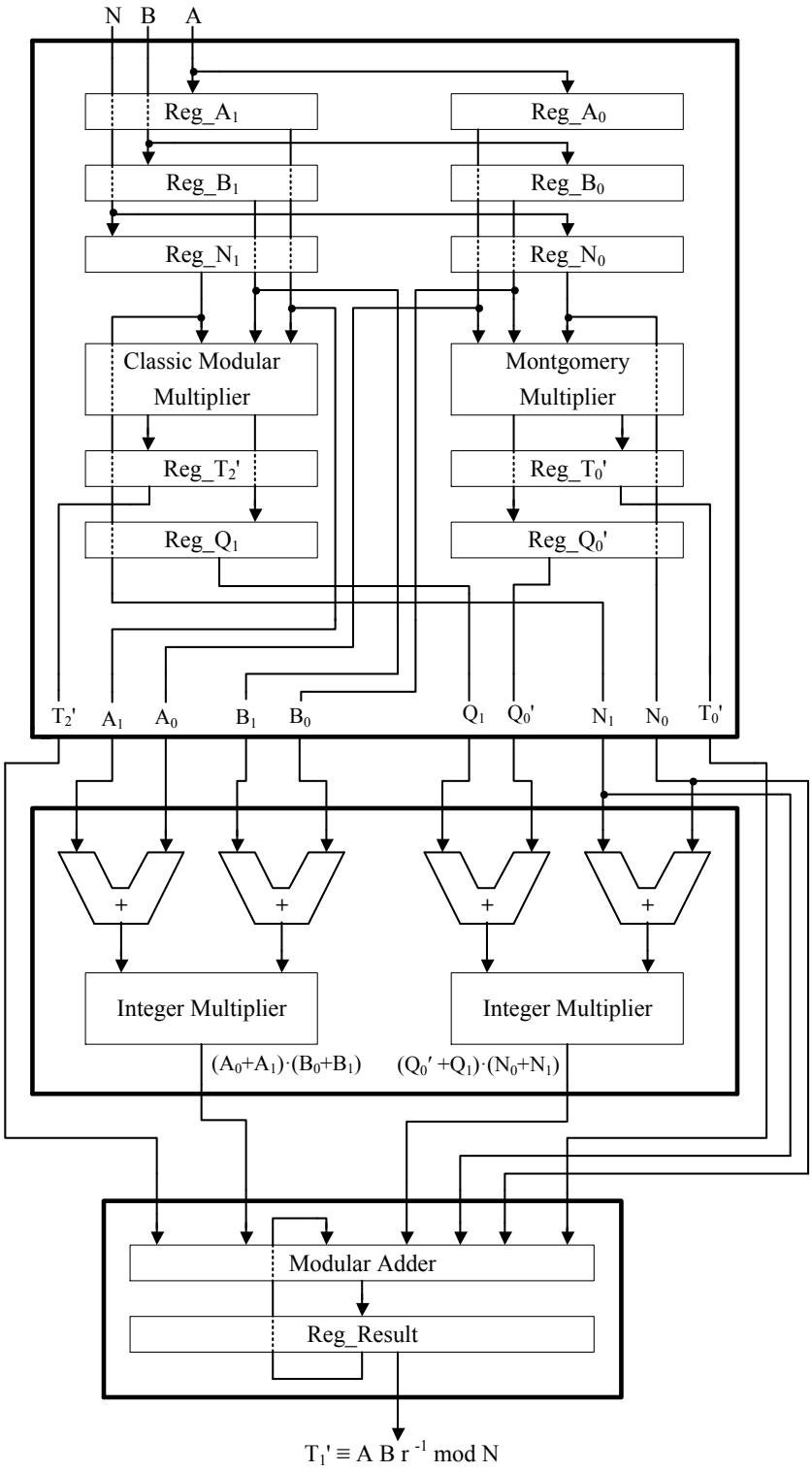

Figure 4.7: Block diagram of Partially Interleaved Modular KO Multiplier.

As shown in the Figure 4.7, the most significant half of multiplier A, multiplicand B and modulus N are processed in classic modular multiplier, while the least significant half of them are processed in Montgomery multiplier. Left-to-right reduction and right-to-left reduction are performed here. T_2' which is the modular multiplication result and Q_1 are produced in classic modular multiplier. Montgomery multiplier computes T_0' and Q_0' values. These partial results are stored in registers. In the second part of the design, multiplication operations of $(A_0+A_1)(B_0+B_1)$ and $(Q_0'+Q_1)(N_0+N_1)$ are carried out by means of integer multipliers. At the last step of the design, final additions/subtractions are performed and modular multiplication result $ABr^{-1} \bmod N$ is computed, where $r = 2^h$. Note that all multiplier modules implemented here are half-sized modules. Three types of adder modules are employed: $(3,2)$ CSA, $(4,2)$ CSA and iteratively used 64-bit CLA which is made up of 16 4-bit CLA adder blocks serially connected. Details of multiplier blocks will be given in the following sections.

4.3.1 Radix-2 Classic Modular Multiplier

Classic modular multiplication accomodates several subtractions at each step of reduction as previously shown in Algorithm 2. As stated in Bunimov et al. [26], classical approach has some drawbacks that, it requires three additions with carry propagation and also two full-bits-lengthed comparisons in the worst case. An estimation logic using the most significant two bits of intermediate result and modulus was proposed in their work in order to reduce the subtractions to a single subtraction. This method also brings the advantage that, instead of two full-bits-lengthed comparisons with modulus, only a comparison to $t \cdot 2^k (t = 0,1,\cdots,6)$ is performed at each step, which can be done in constant time, as values of $t \cdot 2^k \bmod N$ are precomputed before the execution of the loop.

In order to get rid of carry propagation in additions, $(3,2)$ CSAs were employed in their work. They also proposed an optimized version of the algorithm, where only one carry save addition is performed at each step with a few changes made in lookup table. It has the overhead of lookup table and calculation of table elements before the modular multiplication steps. Bunimov's algorithm is shown below:

Algorithm 5 Classic Modular Multiplication Algorithm of Bunimov.

Require: Integers X, Y, N with $0 \leq X, Y \leq N$.
Ensure: $P \equiv XY \bmod N$
    k :number of bits in X
    x_i :i'th bit of X.
1: $S := 0;\ C := 0;\ A := 0;$
2: **for** $i = k-1$ **downto** 0 **do**
3: $S := S \bmod 2^k;$
4: $C := C \bmod 2^k;$
5: $S := 2S;$
6: $C := 2C;$
7: $A := 2A;$
8: $I := x_i \cdot Y;$
9: $(S,C) := CSA(S,C,I);$
10: $(S,C) := CSA(S,C,A);$
11: $A := (2 \cdot s_{k+1} + s_k + 2 \cdot c_{k+1} + c_k) \cdot 2^k \bmod N$
12: **end for**
13: $P := (S+C) \bmod N$
14: **return** P

In the algorithm shown above, $(2 \cdot s_{k+1} + s_k + 2 \cdot c_{k+1} + c_k)$ determines the value of t. As all possible values of $t \cdot 2^k \bmod N$ were precomputed, value of A is evaluated in constant time. Here, sum S and carry C values can never exceed $k+2$ bits, so operations in (3) and (4) are simple operations as making the most significant two bits of S and C 0.

Classic modular multiplication with quotient calculation is outlined in Algorithm 6. Note that, this algorithm is different than ordinary classic modular multiplication algorithm in the way that an additional value Q is calculated.

At each cycle of the algorithm, $t \cdot 2^k \bmod N$ value, which is denoted by A, is added to the intermediate results where the most significant two bits of sum S and carry C determine the value of t. Operations of determining the regarding $t \cdot 2^k \bmod N$ value according to the most significant two bits of S and C, then adding it to the $\bmod 2^k$ of S and C mean, reducing the most significant two bits of S and C with respect to $\bmod N$. Here, in order to compute the quotient, value coming from the quotient lookup table according to t is added to the partial quotient values Q_{1S} and Q_{1C} at each cycle. At the end of modular multiplication cycles Q_{1S} and Q_{1C} are summed up to constitute Q_1. To

better understand the process of quotient calculation, it would be useful to have a look at the calculation of lookup table for $t \cdot 2^k \bmod N (t = 0, 1, \cdots, 6)$ [5].

To determine each lookup value ($t \cdot 2^k \bmod N$ for $t = 0, 1, \cdots, 6$), modulus N is subtracted from the value of $t \cdot 2^k$ before the modular multiplication steps. Number of subtractions for each lookup value is stored in a small lookup table that is used for quotient calculation. This new lookup table has 7 elements and each element is a number between 0 and 15, signifying that whenever a reduction of most significant two bits of S and C is done with $A = t * 2^k \bmod N$, Q_{1New} times N is being subtracted from it, which constitutes the quotient Q_1 at the end of the classical modular multiplication [5].

Algorithm 6 Classical Modular Multiplication Algorithm with Quotient Calculation.

Require: Integers X, Y, N with $0 \leq X, Y \leq N$.
Ensure: $P \equiv XY \bmod N$,Q_1
k :number of bits in X.
x_i :i'th bit of X.
Q_1 : Quotient value shown in Figs. 2.1 and 2.5 .
ALookUpTable : Lookup table for all values of $t \cdot 2^k \bmod N (t = 0, 1, \cdots, 6)$
QLookUpTable : Lookup table storing the values of $\lfloor \frac{t \cdot 2^k}{N} \rfloor$.

$S := 0$; $C := 0$;
$A := 0$; $Q_1 := 0$; $t := 0$;
$Q_{1S} := 0$; $Q_{1C} := 0$ $Q_{1New} := 0$;
for $i = k - 1$ **downto** 0 **do**
  $S := S \bmod 2^k$;
  $C := C \bmod 2^k$;
  $S := 2S$;
  $C := 2C$;
  $A := 2A$;
  $Q_{1S} := 2Q_{1S}$;
  $Q_{1C} := 2Q_{1C}$;
  $Q_{1New} := 2Q_{1New}$;
  $I := x_i \cdot Y$;
  $(S, C) := CSA_{(4,2)}(S, C, I, A)$;
  $(Q_{1S}, Q_{1C}) := CSA_{(3,2)}(Q_{1S}, Q_{1C}, Q_{1New})$;
  $t := (2 \cdot s_{k+1} + s_k + 2 \cdot c_{k+1} + c_k)$;
  $A := ALookUpTable(t)$;
  $Q_{1New} := QLookUpTable(t)$;
end for
$P := (S + C) \bmod N$
$Q_1 := Q_{1S} + Q_{1C} + \lfloor \frac{S+C}{N} \rfloor$;
return P, Q_1

To summarize, at each step of the classical modular multiplication, partial product I, sum S, carry C and remainder A of the reduction (i.e. the most significant two bits of S and C from the previous cycle) are added using the $(4,2)$ CSA block without carry propagation. According the value of t, a quotient value is accumulated and at the end of the multiplication, T_2' and Q_1 values (shown in Fig. 2.5) are calculated.

Block diagram of Radix-2 Classic modular multiplier is shown in Figure 4.8.

Radix-2 Classic modular multiplier performs modular multiplication with k-bit operands vectors X, Y and N and computes the k-bit modular multiplication result P and $k+1$-bit quotient value Q_1.

4.3.2 Radix-2 Montgomery Multiplier

Implementation of Montgomery multiplier module is similar to classical one apart from the estimation logic. Using the CSA at each step, a partial product I, some multiple of modulus N and A are added to the sum S and carry C values. The value of A is determined by using the least significant bits of partial product and sum at each step. Here number of modulo N's that are added during the multiplication steps and subtracted at the end of the algorithm determines the value of Q_0' shown in the Fig. 2.5. Montgomery multiplication algorithm in CSA form with Q_0' calculation is outlined in the Algorithm 7.

Note that, Q_0' value computed in the Algorithm 7 may be a negative number in the case of when no final subtraction is performed. Block diagram of the Radix-2 Montgomery multiplier with Q_0' calculation is shown in Figure 4.9. Montgomery multiplier accepts three k-bit vectors X, Y and modulus N and computes k-bit modular multiplication result $P = XY2^{-k} \bmod N$ and $k+3$ bits signed Q_0' value.

4.3.3 Radix-4 Integer Multiplier

For the integer multiplication operations, a Radix-4 integer multiplier module was implemented according to Algorithm 8. At each integer multiplication cycle, two bits of the multiplier are processed from right to left. Two partial products I_1 and I_2 are added to the intermediate result, which consists of sum S and carry C using

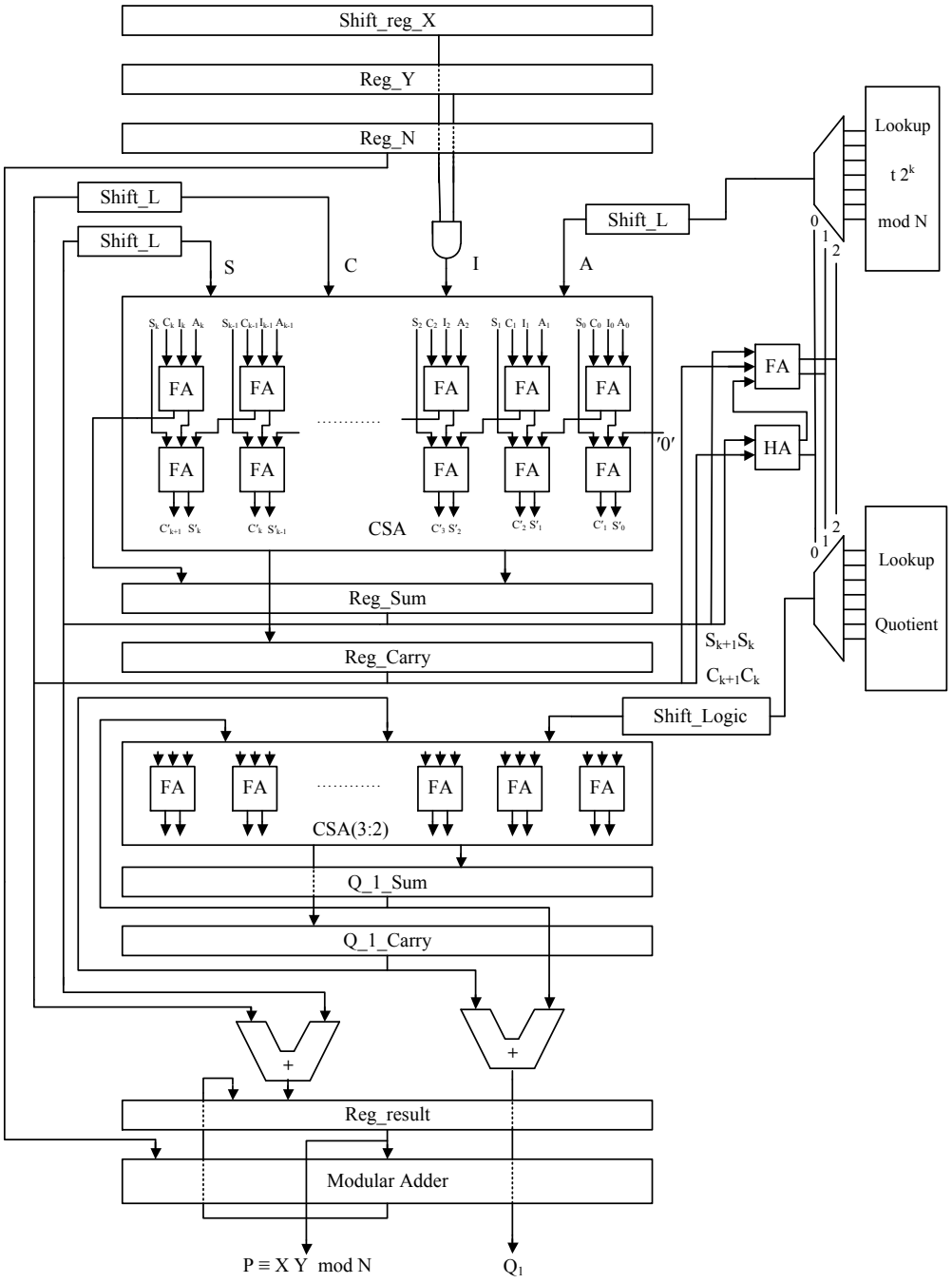

Figure 4.8: Block diagram of Radix-2 Classic Modular Multiplier with Quotient Calculation.

Algorithm 7 Radix-2 Montgomery Multiplication with Q'_0 Calculation.

Require: N: $2^{k-1} \leq N \leq 2^k$, $\gcd(N,2) = 1$;
X,Y: $0 \leq X,Y \leq N$.
Ensure: $P \equiv XY2^{-k} \bmod N$,Q'_0
k :number of bits in X.
x_i :i'th bit of X.
Q'_0 : value shown in Fig. 2.5.

1: $S := 0;\ C := 0;\ A := 0;$
2: $Q'_0 := 0;\ Q_{in} := 0;\ Q_{out} := 0;$
3: **for** $i = 0$ **to** $k-1$ **do**
4: $I := x_i \cdot Y;$
5: $u := (s_0 + i_0) \bmod 2$;
6: $A := u \cdot N;$
7: $(S,C) := CSA(S,C,I,A);$
8: $Q_{in}(i) := u;$
9: $S := \frac{S}{2};\ C := \frac{C}{2};$
10: **end for**
11: $P := (S+C);$
12: **if** $P \geq N$ **then**
13: $P := P - N;$
14: $Q_{out}(k) := 1;$
15: **end if**
16: **if** $P \geq N$ **then**
17: $P := P - N;$
18: $Q_{out}(k) := 0;$
19: $Q_{out}(k+1) := 1;$
20: **end if**
21: $Q'_0 := Q_{out} - Q_{in};$
22: **return** P, Q'_0

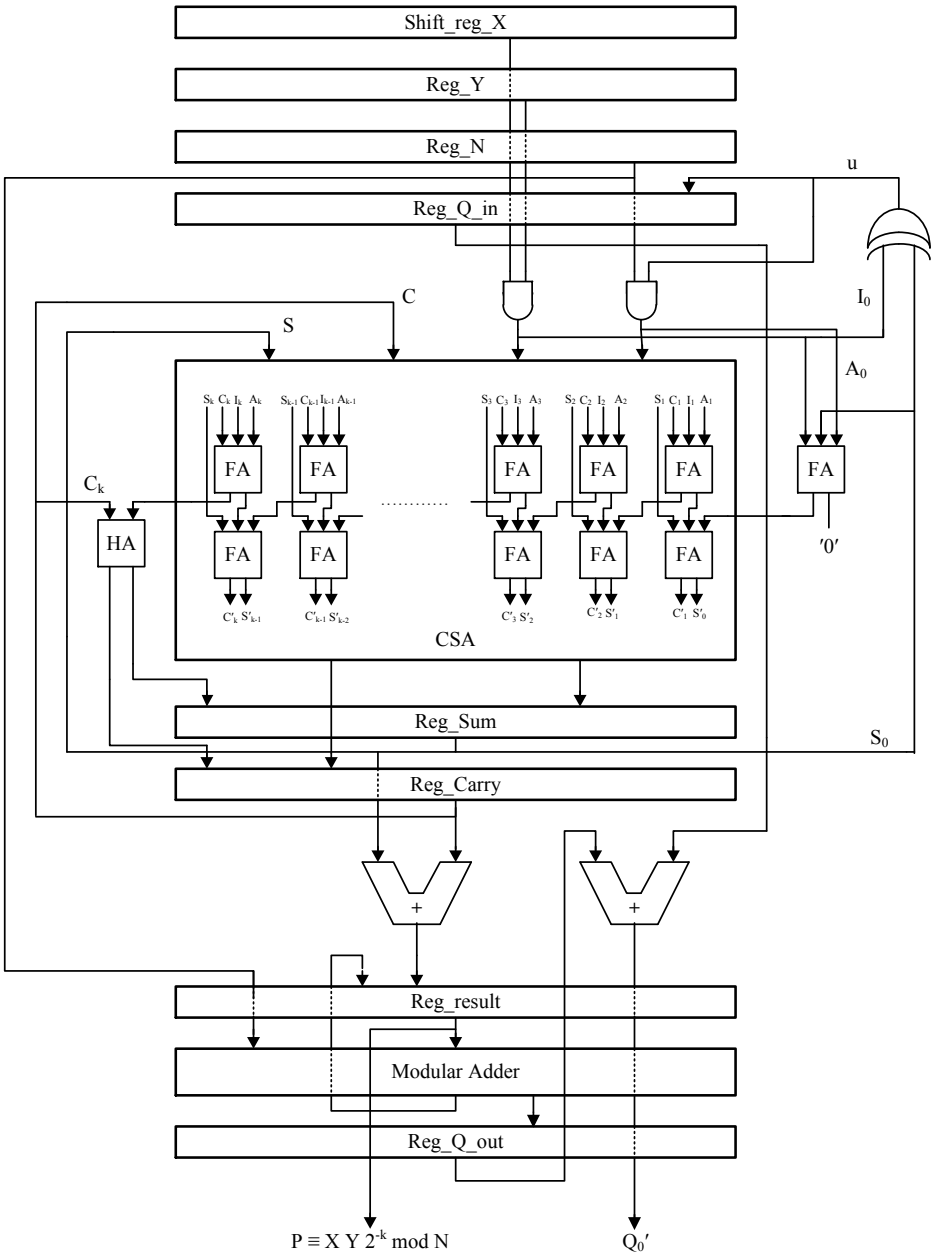

Figure 4.9: Block diagram of Radix-2 Montgomery Multiplier with Q_0' Calculation.

the CSA. Product's least significant k-bits(P_R) are accumulated at each cycle and combined with the most significant k-bit of the product(P_L) at the end of the algorithm. The idea behind designing such an integer multiplier was using the same hardware skeleton which is used not only in Classic modular multiplier but also in Montgomery multiplier.

Algorithm 8 Radix-4 Integer Multiplication Algorithm.

Require: X, Y
Ensure: $P := XY$

1: $S := 0;\ C := 0;\ C_{in} := 0;$
2: **for** $i = 0$ **to** $k-2$ **by** 2 **do**
3: $\quad I_1 := x_i \cdot Y;$
4: $\quad I_2 := x_{i+1} \cdot Y;$
5: $\quad (S_{HA}, C_{in}) := HA(S_1, C_1);$
6: $\quad P_R(i) := S_1;$
7: $\quad P_R(i+1) := S_{HA};$
8: $\quad (S, C) := CSA(S, C, I_1, I_2, C_{in});$
9: **end for**
10: $P_L := (S + C);$
11: **return** $P := (P_L \& P_R);$

In order to use Radix-4 Integer Multiplier for multiplication with a negative number, negative number's absolute value is multiplied by positive number. And then product is converted back to 2's complement notation. These conversions are perfomed in the top module, which is Partiall Interleaved Modular KO multiplier and Radix-4 Integer multiplier performes only multiplication of two positive numbers.

4.3.4 Implementation Results

Partially Interleaved Modular KO multiplier was described in VHDL and implemented on Xilinx Virtex 5 XCVFX130T FPGA. Verification of correct operation of the design was done according to the modular multiplication values coming from the software implementation of the same design in Maple.

For the k-bit operands X, Y and modulus N, Radix-2 Montgomery multiplier implementation performs one modular multiplication operation in $(35k/64) + 18$ clock cycles. Classic modular multiplier performs modular multiplication in $(79k/128) + 49$ clock cycles. Integer multiplier module performs multiplication in $(65k/256) + 6$ clock

Table 4.1: FPGA Implementation Results.

Multiplier Size(k)	512	1024
Minimum Period (ns)	10	10
Maximum Frequency (MHz)	100	100
Total # of Clock Cycles	631	1161
Total Computation Time (μs)	6.310	11.610
Area (slice)	28810	55702
Area (LUT)	26202	49509
Area (Flip-Flop)	35276	67007

cycles. As Montgomery multiplier and Classic modular multiplier work in parallel, first part of the design performs its operation in $(79k/128)+49$ cycles. Two Integer Multiplier modules work in parallel in the second part of the design and complete the integer multiplication in $(65k/256)+6$ clock cycles. Based on these values, our multiplier architecture performs modular multiplication in $((263*k)/236)+98$ clock cycles.

Total number of clock cycles, minimum period, maximum frequency, total computation time and area results of 512, and 1024 bits implementation are shown in the Table 4.1.

Comparison of the FPGA implementation results with previously proposed designs is not easy in terms of architectural differences. First of all, BMM is an ASIC implementation and authors of BMM presented implementation results of their optimized algorithm, instead of a BMM implementation with plain Blakley and Montgomery multipliers. Harris [27] implemented a scable Radix-2 Montgomery multiplier on Xilinx Virtex Pro using processing elements. As FPGA implementation of Partially Interleaved Modular KO multiplier does not employ processing elements and does not target a scalable design, it can not be directly compared with Harris's work neither. Neverthless, in order to give an idea to the reader, a plain comparison of these studies are given in Table 4.2.

Although implementation technologies differ, results show that FPGA implementation of Partially Interleaved Modular KO multiplier could not reach promising results. In spite of using a newer FPGA technology, it could not provide a faster modular multiplication. These results pointed out possible problems with the design that has to

Table 4.2: Comparison of 1024-bit Implementations.

	BMM [7]	Harris [27]	This work
FPGA/ASIC	0.35μm	Virtex Pro	Virtex 5 XCVFX130T
Area(Slices/Gates)	109851 Gates	5598 Slices+5n bits RAM	55702 Slices
Frequency	76.27MHz	144MHz	100 MHz
Total Comp. Time	3.42μs	8.05μs	11.60μs

be taken into account in order to take advantage of Partially Interleaved Modular KO multiplication method and reach faster implementations.

4.4 High Radix Partially Interleaved Modular KO Multiplier

The first hardware implementation of Partially Interleaved Modular KO multiplier was a naive implementation. Implementation results which were not very promising, revealed the need for a better implementation to reach a better performance.

In order to achieve a better implementation, the following design methodology was thought to be used:

- Paying extra attention to control signals in all hiearchies of the design. Reducing the number of control signals as much as possible, so that effect of control signals on the operating frequency of design can be minimized.
- Reanalyzing the Partially Interleaved Modular KO multiplication algorithm and clarifying the dependencies of modules and parameters. This may be useful to better schedule the additions/subtractions performed at the end of the modular multiplication.
- Employing high radices in Montgomery, Classic and Integer multiplier modules. High radix means processing more bits at every clock period, which reduces the number of clock cycles required for modular multiplication.
- Changing implementation technology from FPGA to ASIC in order to work in high frequencies.

4.4.1 Control Signals

Control signals are the signals that control the operation of modules and registers. These signals are like flags, indicating corresponding module must start processing, or register to load incoming bits or clear its stored bits to all zeroes. Critical path of a design, namely the longest path includes not only the data path signals going trough the components like gates, but also control signals. Every control signal is inserted into the critical path by means of new gates. Therefore, in order to design a fast circuit, one has to pay attention to both data path and control path. With this reason, reducing the number of control signals plays a crucial role. Critical path of a design, highlighted in red from one register to another, is shown in the Fig. 4.10. Note that control signals and their paths are highlighted in blue. Although it can not be seen in the figure, control signals are inserted into the critical path via an AND gate and an OR gate.

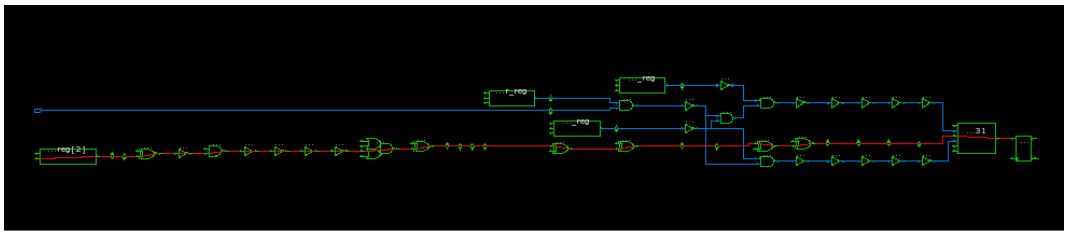

Figure 4.10: Critical Path of a Design.

4.4.2 High Radix Choice

Utilizing high radices in multiplier modules reduces the number of partial products and total number of clock cycles required to complete a multiplication. Because, more multiplier bits are processed at every step of multiplication. As many radices exist, such as 4, 8, 16, and more, a suitable radix must be chosen in order to achieve desired results. Different radices have different requirements which affect the speed or area of the implementation. As the perfomance criteria for this work is speed, then a suitable radix that is faster than the rest of them must be selected. About radix selection, this work took Bewick's PHD thesis [16] as a reference. In his thesis, Bewick implemented fast multipliers with different radices and compared implementation results according to the area, power and delay. Comparison tables show that, according to the delay,

Radix-4(Booth 2) is the best. Therefore, 4 is chosen as radix for modular multipliers, which means at every multiplication step, 2 bits of multiplier are processed.

4.4.3 Radix-4 Classic Modular Multiplier

In the first hardware implementation, an estimation logic using the most significant two bits of intermediate result and modulus, proposed by Bunimov [26] was utilized in order to reduce the subtractions in Blakley to a single subtraction. This method was also bringing the advantage that, instead of two full-bits-lengthed comparisons with modulus, only a comparison to $t \cdot 2^k (t = 0, 1, \cdots, 6)$ was performed at each step, which could be done in constant time, as values of $t \cdot 2^k \bmod N$ were precomputed and stored in a lookup table before the execution of the loop.

Classic modular multiplication algorithm in the first implementation was for Radix-2, whereas for Radix 4 implementation, it needs to be modified. In the case of Radix-4 multiplication, it is an issue to clarify the number of most significant bits of intermediate result to be used in the estimation logic. In addition to this, size of the lookup table storing the precomputed $t \cdot 2^k \bmod N$ values is an issue too. Moreover, as multiplicand Y is multiplied by two multiplier bits, a_i and a_{i-1} in Radix-4, generated partial product I has a value set of $\{0, \mathrm{Y}, 2\mathrm{Y}, 3\mathrm{Y}\}$. Y and $2Y$ are easy multiples, whereas generation of $3Y$ is another problem.

After analysis, it was understood that the optimal choice in Radix-4 Classical modular multiplication is to use the most significant three bits for the estimation logic. Using most significant 3 bits of the intermediate result, which is kept in carry-save form in the multiplier brought a drawback of t to be in the set of $\{0, 1, ..., 14\}$, which means doubling the size of the lookup table.

In order to overcome $3Y$ generation, calculation and storage of $3Y$ before modular multiplication steps was decided to be used. Because, analysis showed that employing Booth encoding causes more lookup table values to be stored. This is due to the sign extension bits that are added to the MSB positions of partial products in Booth encoding.

At every multiplication cycle, partial Q_1 value(Q_{1New}) is determined according to the most significant three bits of S and C. This partial Q_1 value is added to the shifted sum (Q_{1S}) and carry(Q_{1C}) values of Q_1 using $(3,2)$ CSA. Here, operations of determining the regarding $t \cdot 2^k \bmod N$ value according to the most significant three bits of S and C, then adding it to the $\bmod 2^k$ of S and C mean, reducing the most significant three bits of S and C with respect to modulus N.

Algorithm 9 Radix-4 Classic Modular Multiplication Algorithm with Quotient Calculation.

Require: Integers X, Y, N with $0 \leq X, Y \leq N$.
Ensure: $P \equiv XY \bmod N$,Q_1
k :Number of bits in X.
x_i :i'th bit of X.
Q_1 : Quotient value.
ALookUpTable : Lookup table storing all values of $t \cdot 2^k \bmod N (t = 0, 1, \cdots, 14)$
QLookUpTable : Lookup table storing the values of $\lfloor \frac{t \cdot 2^k}{N} \rfloor$.
1: $S := 0$; $C := 0$; $A := 0$; $t := 0$;
2: $Q_{1S} := 0$; $Q_{1C} := 0$ $Q_{1New} := 0$;
3: **for** $i = k-1$ **downto** 0 **by** -2 **do**
4: $S := S \bmod 2^k$;
5: $C := C \bmod 2^k$;
6: $S := 4S$;
7: $C := 4C$;
8: $A := 4A$;
9: $Q_{1S} := 4Q_{1S}$;
10: $Q_{1C} := 4Q_{1C}$;
11: $I := (2x_i + x_{i-1}) \cdot Y$;
12: $(S, C) := CSA_{(4,2)}(S, C, I, A)$;
13: $(Q_{1S}, Q_{1C}) := CSA_{(3,2)}(Q_{1S}, Q_{1C}, Q_{1New})$;
14: $t := (4 \cdot s_{k+2} + 2 \cdot s_{k+1} + s_k + 4 \cdot c_{k+2} + 2 \cdot c_{k+1} + c_k)$;
15: $A := ALookUpTable(t)$;
16: $Q_{1New} := QLookUpTable(t)$;
17: **end for**
18: $P := (S + C) \bmod N$
19: $Q_1 := Q_{1S} + Q_{1C} + \lfloor \frac{S+C}{N} \rfloor$;
20: **return** P, Q_1

As this work proposes an ASIC implementation, both lookup tables *ALookUpTable* and *QLookUpTable*, were implemented as ROM's storing the precomputed $t \cdot 2^k \bmod N (t = 0, 1, \cdots, 14)$ and $\lfloor \frac{t \cdot 2^k}{N} \rfloor$ values respectively. Determination of each lookup element is done in the same way as explained in Sec. 4.3.1. Block diagram of Radix-4 Classic Modular Multiplier is shown in Figure 4.11.

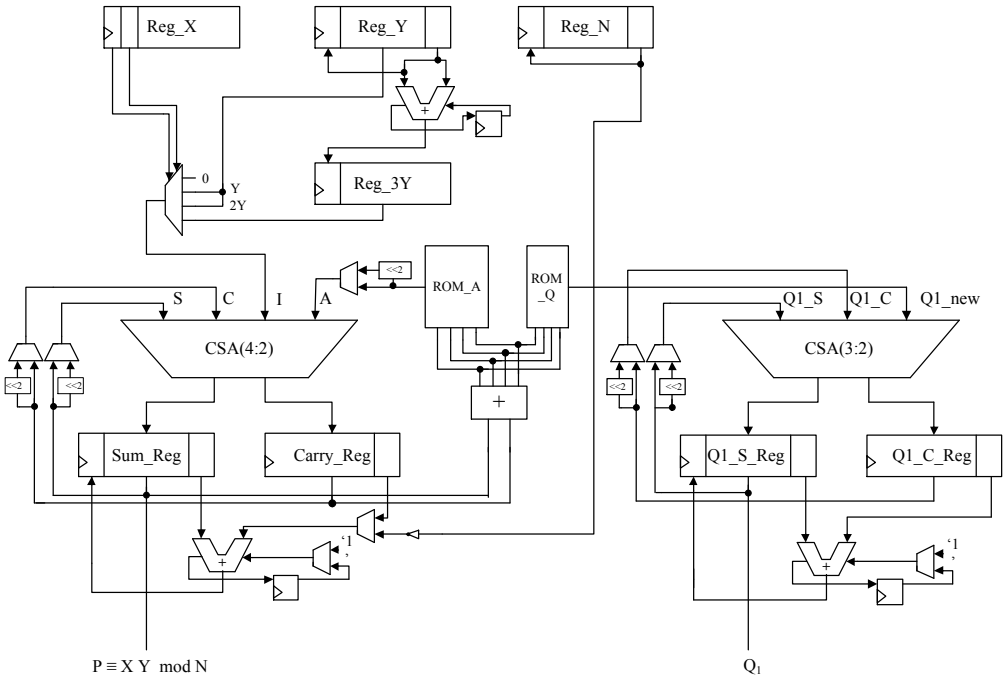

Figure 4.11: Block Diagram of the Radix-4 Classic Modular Multiplier with Quotient Calculation.

In order to understand the block diagram assume that there is a separation between modular multiplication side and quotient calculation side.

In modular multiplication side at every cycle, partial product I and value coming from ROM_A are added to partial sum S and carry C vectors respectively. Produced sum and carry vectors are stored in registers Sum_Reg and Carry_Reg. Then in the next cycle, the most significant 3 bits of S and C are summed up in order to produce new A value by means of ROM_A. At the same time, new S and C vectors are reformed in the way that firstly the most significant 3 bits of them are shifted out and these vectors are multiplied by 4(shifted 2 bits to the left). New partial product I is determined and I, A, S and C are summed up again.

In quotient calculation side, quotient value $Q1\_new$ is determined according to most significant 3 bits of S and C at every cycle. ROM_Q stores precomputed quotient values, which are 5-bit unsigned numbers. This new quotient value is added to 2-bit left shifted $Q1\_S$ and $Q1\_C$, which are sum and carry vectors of quotient.

For the k-bit operands X, Y and modulus N, addition of S, C, I and A continues for $k/2$ cycles. When all multiplier bits are processed 32-bit CLAs are used iteratively in order

to sum up S and C for modular multiplication result and $Q1\_S$ and $Q1\_C$ for quotient. Addition results are stored in registers Sum_Reg and Q1_S_Reg. Final reduction is performed here which is shown as Step 18 in the Alg. 9. In this step, modulus N is subtracted from $S+C$ iteratively using the same CLA. Number of subtractions are added to the quoetient value.

In this design, Sum_Reg and Q1_S_Reg registers are designed in a way that they can accept binary vectors; at the end of the multiplication they can shift the least significant 32 bits out and accept 32 bits to the MSB position of them when CLAs are used iteratively for additions.

$3Y$ calculation logic is also shown in Figure 4.11. Here, $3Y$ is calculated by iterative use of a 32-bit CLA. In the summation of $2Y$ and Y, same shifted Y bits are used. Assume shifted Y bits are denoted as Y_s. Then Y_s and $2*Y_s$ are sent to the CLA as inputs. Results are stored in Reg_3Y register. At the same time, shifted bits are restored by Reg_Y again.

Radix-4 Classical modular multiplier performs one modular multiplication in $(21k/32)+17$ clock cycles where k is the bit length of X, Y and modulus N. k-bit modular multiplication result and $k+1$ bit quotient are computed.

4.4.4 Radix-4 Booth Encoded Montgomery Multiplier

In hardware implementation of Radix-4 Montgomery multiplier, 3 methods were applied in order to improve the performance.

First of all, as two bits of multiplier is processed at every step, partial product has a value set of $\{0, Y, 2Y, 3Y\}$. Among the value set, 0, Y and $2Y$ are easy products which can be generated by simple shift operations. But $3Y$ product needs extra operations such as precomputing and storing or, calculating $2Y+Y$ during the multiplication. In order to figure out this burden, Booth encoding(see Sec. 2.6) is used. Utilizing Booth encoding converts the value set of I from $0, Y, 2Y, 3Y$ to $\{0, Y, 2Y, -Y, -2Y\}$ in which every element of this set can be computed easily by shift and/or complement operations. Booth encoding scheme takes a bit stream of the multiplier $(x_{i+1}, x_i, x_{i-1})_2$

Table 4.3: Montgomery Encoding Scheme.

sp_1	sp_0	n_1	q_{i+1}	q_i	QN	Sign
0	0	0	0	0	0	
0	0	1	0	0	0	
0	1	0	0	1	-N	1
0	1	1	0	1	N	0
1	0	0	1	0	$2N$	0
1	0	1	1	0	$2N$	0
1	1	0	0	1	N	0
1	1	1	0	1	-N	1

as input and generates an encoded partial product I according to the Table 2.4 where x_{-1} is defined to be 0 [28].

Secondly, in Radix-4 Montgomery multiplication, QN value has to be computed to make the least significant two bits of the partial result(sum and carry) 0. In normal operation of the Montgomery algorithm, Q is calculated according to the least significant two bits of partial result and partial product. As Booth encoding is incorporated in this design, determination of Q became more and more difficult. A very simple solution to this problem was proposed in [29] , [30], which is, utilizing $4Y$(4*Multiplicand) instead of original Y. This modification makes the least significant two bits of partial product I always 00. Thus, calculation of Q becomes independent from I, which means I and Q can be computed in parallel.

Lastly, in Radix-4 Montgomery multiplication, because Q is two bits now, QN has a value set of $\{0, N, 2N, 3N\}$. Another encoding scheme, namely *Montgomery encoding* is used to figure out the $3N$ problem. Let $(sp_1, sp_0)_2$ and $(n_1, n_0)_2$ be the least significant two bits of the partial result($sum + carry$) and N respectively. Remember the input condition of N, being odd. Taking this into account, Montgomery encoding scheme takes a bit stream of $(sp_1, sp_0, n_1)_2$ and generates a recoded QN according to the Table 4.3, where q_i and q_{i+1} are the recoded quotient bits for QN at the i-th iteration [28].

Booth encoded Radix-4 Montgomery multiplication algorithm with Q' calculation is outlined in Algorithm 10. As Y^* is used instead of Y, Alg. 10 iterates one more cycle than ordinary Montgomery in order to get the correct result $P \equiv XY2^{-k} \bmod N$.

Readers should note that the reason behind the Q_+ and Q_- variables are the recoded q_i values having different signs as shown in the Table 4.3. Final Q'_0 is the Q value in the Montgomery's equation: $P \equiv (XY + QM)2^{-k} \bmod N$. Block diagram of Radix-4 Montgomery multiplier is shown in Fig. 4.12.

Algorithm 10 Radix-4 Montgomery Multiplication Algorithm with Q' Calculation.

Require: N: $2^{k-1} \leq N \leq 2^k$, $\gcd(N,2) = 1$;
X, Y: $0 \leq X, Y \leq N$. $Y^* := 4Y$;
Ensure: $P \equiv XY2^{-k} \bmod N$,Q'
k :number of bits in X.
x_i : i'th bit of X.
$S := 0;\ C := 0;\ QN := 0;\ Q' := 0;\ Q_+ := 0;\ Q_- := 0;$
for $i = 0$ **to** k **by** 2 **do**
  $I := BoothEncoder(x_{i+1}, x_i, x_{i-1}, Y^*)$;
  $(sp_1, sp_0) := (2s_1 + s_0 + 2c_1 + c_0)$;
  $(Sign, q_{i+1}, q_i, QN) := MontgomeryEncoder(sp_1, sp_0, n_1)$;
  $(S,C) := CSA_{(4,2)}(S, C, I, QN)$;
  $S := \frac{S}{4}$;
  $C := \frac{C}{4}$;
  if $Sign = 0$ **then**
    $Q_-(i) := q_i$;
    $Q_-(i+1) := q_{i+1}$;
  else
    $Q_+(i) := q_i$;
    $Q_+(i+1) := q_{i+1}$;
  end if
end for
$P := (S + C)$;
$Q' := (Q_+ - Q_-)$;
if $P \geq N$ **then**
  $P := P - N$;
  $Q' := Q' + 2^{k+2}$;
else if $P \leq N$ **then**
  $P := P + N$;
  $Q' := Q' - 2^{k+2}$;
end if
$Q' := \frac{Q'}{4}$;
return P, Q'

As shown in Fig. 4.12, at every multiplication cycle Booth encoder calculates the partial product I according to the shifted multiplier bits. Montgomery encoder computes QN which makes the least significant two bits of S and C 0. Partial result vectors S and C are summed up with I and QN by means of $(4,2)$ CSA. Readers may

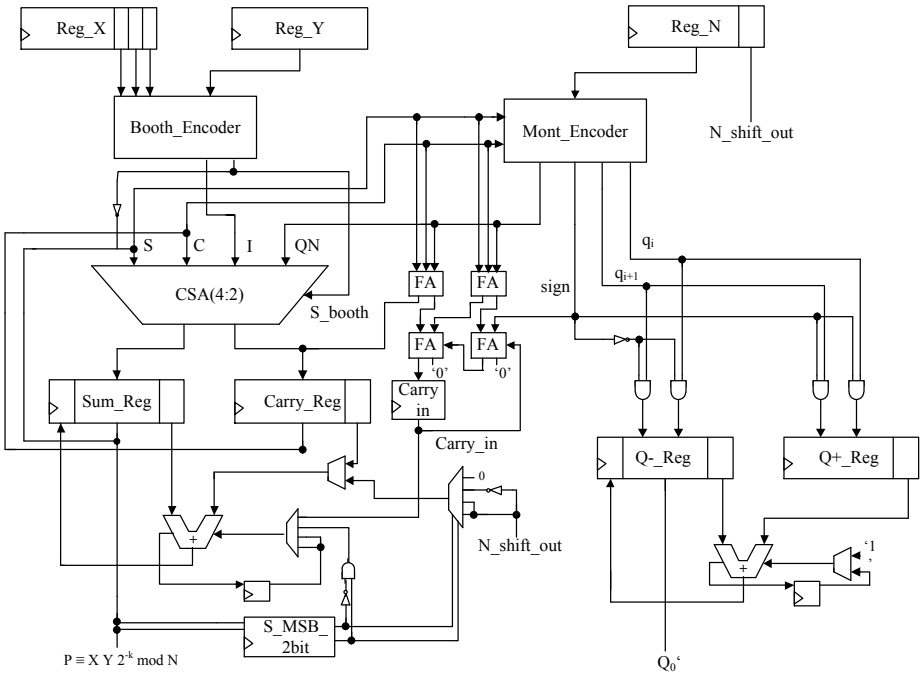

Figure 4.12: Block Diagram of the Radix-4 Montgomery Multiplier with Q'_0 Calculation.

notice four full adders on the right handside of the CSA. As the least significant two bits of I are zeroes, I is not included in the addition of these two bits of binary vectors. These FAs are utilized to sum up the least significant two bits of S, C and QN. Four bits are produced here, where two bits in the LSB position are zeroes(due to Montgomery reduction) and the other two bits in the same arithmetic weight; one of them is joined to the LSB position of C and the other one is stored as Carry_in signal for the next multiplication cycle.

In addition to computing QN, Montgomery encoder module also computes q_i and q_{i+1} signals. These values are stored in Q registers according to the *sign* bit generated by Montgomery encoder. At the end of multiplication cycles, S and C vectors and Q_- and Q_+ vectors are summed up via 32-bit CLA iteratively. There is a register denoted as *S_MSB_2bit*. This register stores the most significant two bits of sum. When $S+C$ is performed and result is stored in Sum_Reg, the most significant two bits of this result is registered by *S_MSB_2bit*. These stored two bits are used in order to determine the sign of modulus N to be added for the final reduction.

For the k-bit operands X, Y and modulus N, Radix-4 Montgomery multiplier performs one modular multiplication in $(9k/16)+8$ clock cycles.

4.4.5 Radix-8 Integer Multiplier

Radix-8 Integer multiplier scans 3 bits of multiplier at every cycle. As it uses the same carry save adder structure ((4,2) CSA), in addition to the sum S, and carry C values, two partial products, namely I_1 and I_0 are used.

Multiplexers, selecting I_1 and I_0 according to the 3 multiplier bits x_{i+2}, x_{i+1} and x_i were designed in the way that, each partial product has a value set with 4 values and when I_1 and I_0 are summed up, original value set for 3 multiplier bits is reached (8 values,$\{0,Y,2Y,3Y,4Y,5Y,6Y,7Y\}$) as shown in the Multiplexing table (Table 4.4 below.

Table 4.4: Multiplexing Scheme for Partial Products.

Multiplier Bits			Selected Partial Products	
x_{i+2}	x_{i+1}	x_i	I_1	I_0
0	0	0	0	0
0	0	1	0	Y
0	1	0	Y	Y
0	1	1	$2Y$	Y
1	0	0	$3Y$	Y
1	0	1	$2Y$	$3Y$
1	1	0	$3Y$	$3Y$
1	1	1	$3Y$	$4Y$

Radix-8 Integer multiplication algorithm is outlined in Algorithm 11.

Product's least significant k-bits(R_PART) are accumulated at each cycle and concatenated with the most significant k-bit of the product(L_PART) at the end of the algorithm.

Note that, for the multiplication operations including negative numbers, firstly number is converted to its absolute value. Multiplication is performed between two positive numbers and then result is converted back to negative form.

Algorithm 11 Integer Multiplication Algorithm.

Require: X, Y.
Ensure: $P := XY$
1: $S := 0$; $C := 0$; $C_{in} := 0$;
2: **for** $i = 0$ **to** $k-3$ **by** 3 **do**
3: $I_1 := MUX(x_{i+2}, x_{i+1}, x_i, Y, 2Y, 3Y, 0)$;
4: $I_0 := MUX(x_{i+2}, x_{i+1}, x_i, Y, 3Y, 4Y, Y)$;
5: $(S\_HA, C\_HA) := HA(S_1, C_1)$;
6: $(S\_FA, C_{in}) := FA(S_2, C_2, C\_HA)$;
7: $(S, C) := CSA(S, C, I_0, I_1, C_{in})$;
8: $R\_PART(i) := S_0$; $R\_PART(i+1) := S\_HA$; $R\_PART(i+2) := S\_FA$;
9: **end for**
10: $L\_PART := (S + C)$;
11: **return** $P := (L\_PART \& R\_PART)$;

For the k-bit operands X, Radix-8 Integer multiplier performs one multiplication in $(11k/24) + 13$ clock cycles.

4.4.6 Integration of Multipliers

High radix implementation of Partially Interleaved Modular Multiplier is shown in Figure 4.13. As shown in the figure, T_0' and Q_0' are computed by Radix-4 Booth encoded Montgomery multiplier. T_2' and Q_1 are calculated by Radix-4 Classic modular multiplier. A_0 and A_1, B_0 and B_1, N_0 and N_1 are summed up by means of CLAs and results are stored in A_1, B_1 and N_1 registers respectively. $(A_0 + A_1)$ and $(B_0 + B_1)$ are multiplied by the first integer multiplier. The second integer multiplier multiplies $(N_0 + N_1)$ with $(Q_0' + Q_1)$. Integer multiplications' results are stored in A0A1B0B1_reg and Q0Q1N0N1_reg. Then final addition/subtraction and reduction operations are performed. Which are simply: $P \equiv \left[(T_2' \& T_0') + (A0 + A1)(B0 + B1) - (Q_0' + Q_1)(N0 + N1) - (T_0' \& T_2')\right] \bmod N$. Result is stored in register T2T0_reg.

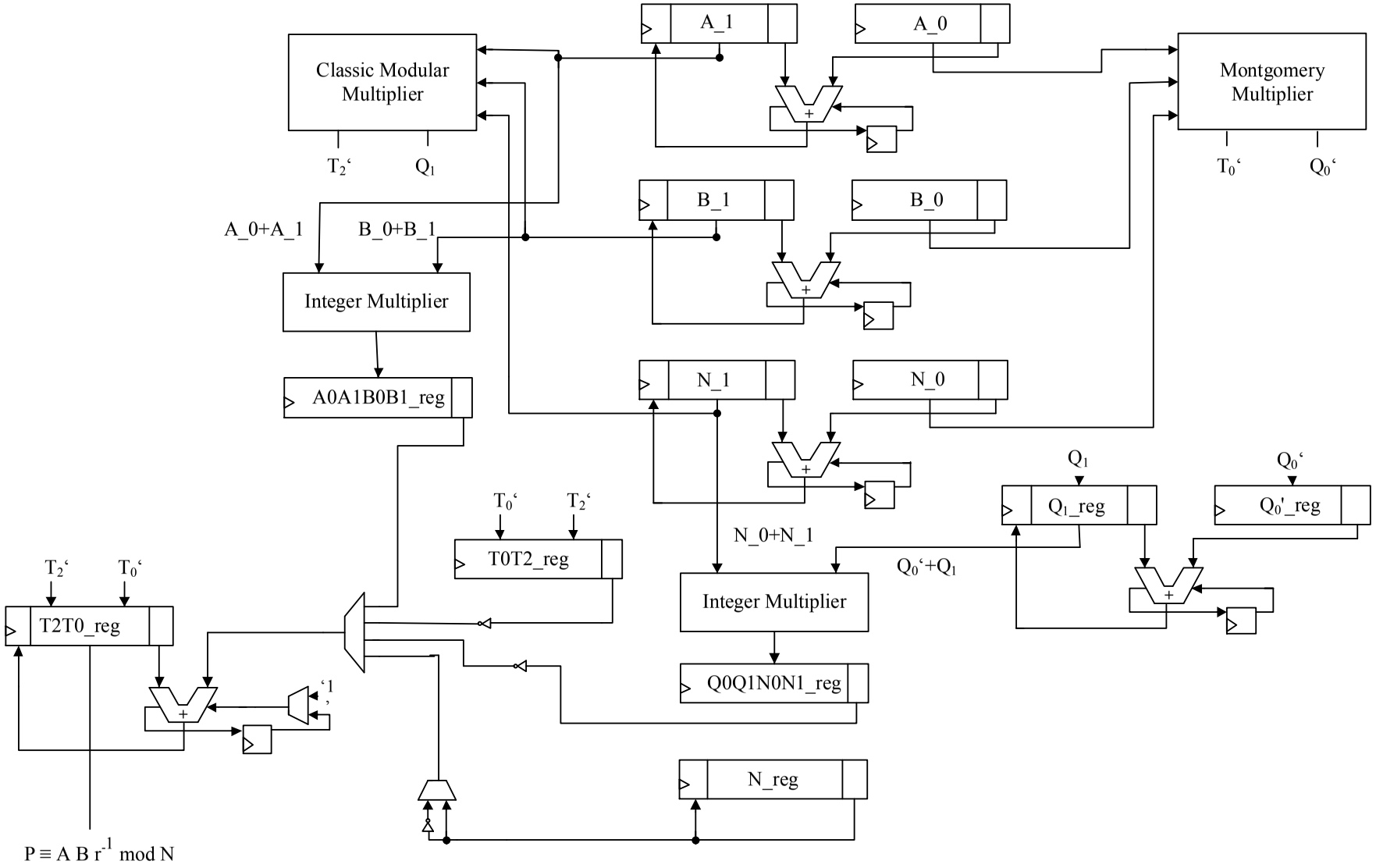

Figure 4.13: High Radix Partially Interleaved Modular KO Multiplier.

In order to effectively use modular multipliers and integer multipliers parameter dependencies were analyzed. As main operation is

$$P \equiv \left[(T_2'\&T_0') + (A0+A1)(B0+B1) - (Q_0' + Q_1)(N0+N1) - (T_0'\&T_2')\right] \bmod N$$

operations are needed to be scheduled to perform modular multiplication in the least number of clock cycles. To achieve this, a scheduling table shown in Figure 4.14 was prepared for 1024-bit implementation.

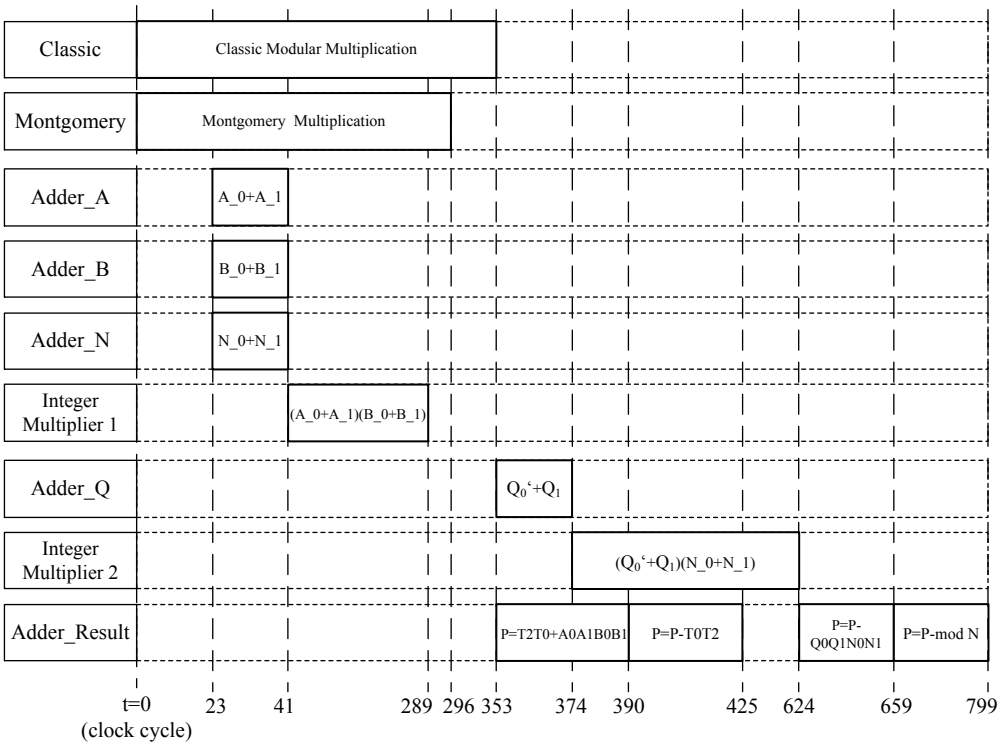

Figure 4.14: Job Scheduling Table.

Remember $A0$, $A1$, $B0$, $B1$, $N0$, $N1$ are input values to the High Radix Partially Interleaved Modular KO Multiplier. These binary vectors are always available to use. Montgomery multiplier and Classic modular multiplier need these values until they fill in their registers with these inputs. When they fill their registers, they start modular multiplication operations and they dont need them any more. With this reason, firstly Montgomery and Classic multipliers start processing. Their dependencies to these vectors finish 23 clock cycles later. Then, $A0+A1$, B_0+B1 and N_0+N1 operations are performed by Adder_A, Adder_B and Adder_N

respectively. At 41st cycle these addition results are ready, so that IntegerMultiplier_1 may start multiplication of $(A0+A1)(B0+B1)$. IntegerMultiplier_1 finishes its operation slightly earlier than Montgomery multiplier. Montgomery multiplier finishes multiplication at cycle 296 and Classic finishes at cycle 353. Now, T_2', Q_1, T_0', Q_0' and $(A0+A1)(B0+B1)$ are ready. Adder_Q starts summing up Q_0' and Q_1. Meanwhile $T2T0+A0A1B0B1$ operation starts at Adder_Result. Adder_Q finishes its operation at cycle 374 and IntegerMultiplier_2 starts multiplying $Q_0'+Q_1$ and $N0+N1$. Before IntegerMultiplier_2 finishes multiplication $P=T2T0+A0A1B0B1$ finishes and Adder_Result subtracts $T_0'T_2'$ between cycles of 390 and 425. When IntegerMultiplier_2 finishes multiplication at cycle 624, only two jobs are left in the schedule. Adder_Result subtracts multiplication result of $Q0Q1N0N1$ from P. And then three subtractions are performed at most in order to complete the final reduction. In total, for 1024-bit input vectors A, B and modulus N, one modular multiplication is completed in 799 clock cycles.

4.4.7 Implementation Results

For the k-bit operands X, Y and modulo N, High Radix Partially Interleaved Modular KO multiplier performs one modular multiplication in $((35k+24)/48)+52$ clock cycles. Design was described in VHDL and synthesized using 90nm TSMC standard cell libraries in Design Compiler with the following operating conditions:

```
Operating Condition Name : BCCOM
Library : tcbn90ghpbc
Temperature :   0.00
Voltage :   1.10
```

Note that *BCCOM* signifies the best corner conditions in which transistors(PMOS and NMOS) operate in fast mode in $0\,^{\circ}$C and with $1.1V$ V_{dd}.

Total number of clock cycles, minimum period, maximum frequency, total computation time and area results of 1024-bit implementation are shown in the Table 4.5.

Table 4.5: Implementation Results of Modular Multiplier.

Min. Period	Max. Frequency	Total # of Clock Cycles	Total computation time
0.8 *ns*	1.25 *GHz*	799	639 *ns*

Table 4.6: Comparison of 1024-bit Implementations.

	BMM [7]	Harris [27]	This work
FPGA/ASIC	0.35μm	Virtex Pro	90nm
Area(Slices/Gates)	109851 Gates	5598 Slices+5n bits RAM	253K Gates
Frequency	76.27MHz	144MHz	1.25 GHz
Total Comp. Time	3.42μs	8.05μs	639ns

Comparison of high radix implementation with previous modular multipliers are given in Table 4.6.

In Table 4.7, implementation results of RSA with binary exponentiation are given. Recall that $2^{16}+1$ is selected as the modular exponent for RSA encryption operations.

Table 4.7: Implementation Results of RSA encryption ($e = 2^{16}+1$).

RSA Comp. Time	Throughput	# of RSA Encryptions per sec.
10863 *ns*	92 *Mbps*	90K

Comparison of this work with commercial RSA chips and recently proposed RSA implementations is given in Table 4.8 where comparison is done according to RSA encryption operations with modular exponent $2^{16}+1$.

Comparison of 1024-bit RSA decryption operations with the previous works is given in Table 4.9. These works are the most recent high speed implementations of RSA cryptosystem. As shown in the table, the only work that is faster than the proposed RSA implementation among [34], [33], and [34] was introduced by Miyamoto et al. in [35], where Radix-128 Montgomery multiplier is implemented. As they process 7 bits at a time, their critical path delay is long and shows its effect on operating frequency of 421.94 MHz. Their design has a smaller area than this design. And there is only 0.09 ms difference between the RSA computation times of this design and their design.

Table 4.8: Comparison of 1024-bit RSA Encryptions.

	Encryptek [31]	Broadcom [32]	[33]	This work
FPGA/ASIC		0.13μm	XC2V6000	90nm
Area(Slices/Gates)				253K
Frequency			215.83 MHz	1.25 GHz
# of RSA Encryptions per sec.	90K	15K	11K	90K

Table 4.9: Comparison of 1024-bit RSA Decryptions.

	[35]	[34]	[33]	This work
ASIC tech.	90nm	0.18μm	0.18μm	90nm
Area(Gates)	153K	192K	184K	253K
Frequency	421.94 MHz	300 MHz	550 MHz	1.25 GHz
RSA comp. time(ms)	0.89	2.8	3.86	0.98

5. CONCLUSION

In this thesis two hardware implementations of Partially Interleaved Modular KO multiplication are proposed. In addition to this the second high performance implementation of modular multiplier is embedded into RSA cryptosystem and provided a high speed RSA implementation which is comparable with commercial RSA chips and one of the fastest works in the literature.

These hardware implementations are the first implementations of Saldamli's method of modular multiplication. His algorithm's requirements were successfully fulfilled with modified Blakley and Montgomery multipliers. The first implementation targeting FPGA platforms did not give promising results. These results and possible problems of the design pioneered the explorations for a better, and faster modular multiplier, which is the second implementation. In order to boost the speed up, high radix choice was made according to Bewick's thesis [16]. High radix bringing the advantage of reduced number of partial products to be processed brought disadvantages. These drawbacks were eliminated by incorporation of Booth Encoding and Montgomery Encoding schemes. What is more, by means of a very simple technique, inner operations of Montgomery multiplier became parallel. In addition to these approaches, extra attention was paid into control signals and effect of control signals on critical path was reduced as much as possible. Moreover, parameter dependencies of multiplier blocks were reanalyzed in Partially Interleaved Modular KO multiplication and a scheduling table was prepared which let the modular multiplication operation to be performed in least number of clock cycles.

As a future work, integer multiplier modules may be replaced with faster integer multipliers. These new faster multipliers may reduce total number of clock cycles of the whole design, so a faster modular multiplier may be designed. In addition to this, pipelining methodology may be incorporated into the multiplier designs. Effect of pipeling may be researched whether it helps the multiplier to boost the speed up

or not. Moreover, Montgomery and Classic modular multipliers may be redesigned to perform the integer multiplication operations and modular multiplication operations simultaneously. By means of this improvement, total number of clock cycles may be reduced significantly and probably the fastest RSA hardware implementation may be achieved.

REFERENCES

[1] **National Institute of Standards and Technology (NIST)**, (2001), FIPS 197: Advanced Encryption Standard.

[2] **Rivest, R.L.**, **Shamir, A. and Adleman, L.**, (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, **21(2)**, 120–126.

[3] **Karatsuba, A. and Ofman, Y.**, (1963). Multiplication of multidigit numbers by automata, *Soviet Physics-Doklady*, **7**, 595–596.

[4] **Saldamli, G.**, (2011). Partially Interleaved Modular Karatsuba-Ofman Multiplication, *preprint*.

[5] **Aris, A.**, **Ors, B. and Saldamli, G.**, (2011). Architectures for Fast Modular Multiplication, Digital System Design (DSD), 2011 14th Euromicro Conference on, IEEE Computer Society, Oulu, Finland, pp.434 –437.

[6] **Kaihara, M. and Takagi, N.**, (2005). Bipartite Modular Multiplication, Cryptographic Hardware and Embedded Systems – CHES 2005, volume3659 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp.201–210.

[7] **Kaihara, M. and Takagi, N.**, (2008). Bipartite Modular Multiplication Method, *IEEE Transactions on Computers*, **57(2)**, 157–164.

[8] **Çetin Kaya Koç**, (1994). High-Speed RSA Implementation, Technical ReportTR 201, RSA Laboratories, 73 pages.

[9] **Menezes, A.**, **van Oorschot, P. and Vanstone, S.**, (1997). Handbook of Applied Cryptography, CRC Press.

[10] **Blakley, G.R.**, (1983). A computer algorithm for the product AB modulo M, *IEEE Transactions on Computers*, **32(5)**, 497–500.

[11] **Barrett, P.**, (1987). Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, Proceedings on Advances in cryptology—CRYPTO '86, Springer-Verlag, Santa Barbara, California, United States, pp.311–323.

[12] **Montgomery, P.L.**, (1985). Modular multiplication without trial division, *Mathematics of Computation*, **44(170)**, 519–521.

[13] **Schönhage, A. and Strassen, V.**, (1971). Schnelle Multiplikation grafer Zahlen, *Computing*, **7**, 281–292.

[14] **Fürer, M.**, (2007). Faster integer multiplication, Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, STOC '07, ACM, San Diego, California, USA, pp.57–66.

[15] **Parhami, B.**, (2000). Computer Arithmetic Algorithms and Hardware Designs, Oxford University Press.

[16] **Bewick, G.W.**, (1994). Fast Multiplication: Algorithms and Implementation, Ph.D. thesis, Department of Electrical Engineering, Stanford University.

[17] **Booth, A.D.**, (1951). A Signed Binary Multiplication Technique, *Quarterly Journal of Mechanics and Applied Mathematics*, **4**, 236–240.

[18] **Chu, P.P.**, (2006). RTL Hardware Design Using VHDL, Wiley-Interscience.

[19] **Pedroni, V.A.**, (2004). Circuit Design with VHDL, MIT Press.

[20] **Brown, S. and Rose, J.**, (1996). FPGA and CPLD architectures: a tutorial, *Design Test of Computers, IEEE*, **13(2)**, 42 –57.

[21] **Wikipedia**, (2012), Application-specific integrated circuit — Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/w/index.php?title=Application-specific_integrated_circuit&oldid=488157454`, [Online; accessed 25-April-2012].

[22] **Central, F.**, (2012), FPGA synthesis and implementation(Xilinx design flow), `http://www.fpgacentral.com/docs/fpga-tutorial/fpga-synthesis-and-implementation-xilinx`, [Online; accessed 26-April-2012].

[23] **Synopsys**, (2010), Design Compiler User Guide.

[24] **Koren, I.**, (2002). Computer Arithmetic Algorithms, A K Peters Natick, Massachusetts.

[25] **Mano, M.M. and Kime, C.R.**, (2004). Logic and Computer Design Fundementals, Pearson Prentice Hall.

[26] **Bunimov, V. and Schimmler, M.**, (2003). Area and time efficient modular multiplication of large integers, Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on, IEEE Computer Society, Los Alamitos, CA, USA, pp.400 – 409.

[27] **Harris, D.**, **Krishnamurthy, R.**, **Anders, M.**, **Mathew, S. and Hsu, S.**, (2005). An improved unified scalable radix-2 Montgomery multiplier, Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on, IEEE Computer Society, Cape Cod,Massachusetts,USA, pp.172 – 178.

[28] **Son, H. and Oh, S.**, (2004). Design and implementation of scalable low-power Montgomery multiplier, Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on, IEEE Computer Society, San Jose, California, USA, pp.524 – 531.

[29] **Elridge, S.E. and Walter, C.D.**, (1993). Hardware Implementation of Montgomery's Modular Multiplication Algorithm, *IEEE Transactions on Computers*, **42(6)**, 693–699.

[30] **Liu, J. and Dong, J.**, (2010). Design and implementation of an efficient Montgomery modular multiplier with a new linear systolic array, Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on, IEEE, Beijing, China, pp.225–229.

[31] **Encryptek**, (2012), Radium PCIe Card, `http://www.encryptek.net/radium_pcie_card.html`, [Online; accessed 04-May-2012].

[32] **Broadcom**, (2006), BCM5825 Product Brief High Performance Security Processor.

[33] **Shieh, M.D.**, **Chen, J.H.**, **Lin, W.C. and Wu, H.H.**, (2009). A New Algorithm for High-Speed Modular Multiplication Design, *Circuits and Systems I: Regular Papers, IEEE Transactions on*, **56(9)**, 2009 –2019.

[34] **Koo, B.**, **Lee, D.**, **Ryu, G.**, **Chang, T. and Lee, S.**, (2006). High-Speed RSA Crypto-processor with Radix-4 Modular Multiplication and Chinese Remainder Theorem, Information Security and Cryptology – ICISC 2006, volume4296 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp.81–93.

[35] **Miyamoto, A.**, **Homma, N.**, **Aoki, T. and Satoh, A.**, (2011). Systematic Design of RSA Processors Based on High-Radix Montgomery Multipliers, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, **19(7)**, 1136 –1146.

CURRICULUM VITAE

Name Surname: AHMET ARIŞ

Place and Date of Birth: Fethiye 01.10.1986

E-Mail:arisahmet@gmail.com

B.Sc.: Bahçeşehir University, Computer Engineering, June 2009

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- **Aris A.**, Ors B., Saldamli G., 2011: Architectures for fast modular multiplication*14th Euromicro Conference on Digital System Design*, August 31-September 2, 2011 Oulu, Finland.