İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

POWER EFFICIENT FPGA IMPLEMENTATION OF RSA ALGORITHM

M.Sc. Thesis by
Dilek BAYHAN GÜMÜŞ

Department : Computer Engineering

Programme : Computer Engineering

SEPTEMBER 2010

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

POWER EFFICIENT FPGA IMPLEMENTATION OF RSA ALGORITHM

M.Sc. Thesis by
Dilek BAYHAN GÜMÜŞ
(504061543)

Date of submission : 13 September 2010
Date of defence examination: 21 September 2010

Supervisor (Chairman) : Assis. Prof. Dr. S. Berna ÖRS YALÇIN (ITU)
Members of the Examining Committee : Prof. Dr. Emre HARMANCI (ITU)
Assis. Prof. Dr. Gökay SALDAMLI (BU)

SEPTEMBER 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

RSA ALGORİTMASININ DÜŞÜK GÜÇ TÜKETİMLİ FPGA TASARIMI

YÜKSEK LİSANS TEZİ

Dilek BAYHAN GÜMÜŞ
(504061543)

Tezin Enstitüye Verildiği Tarih : 13 Eylül 2010
Tezin Savunulduğu Tarih : 21 Eylül 2010

Tez Danışmanı : Yrd. Doç. Dr. S. Berna ÖRS YALÇIN (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. Emre HARMANCI (İTÜ)
Yrd. Doç. Dr. Gökay SALDAMLI (BÜ)

EYLÜL 2010

FOREWORD

First, I would like to thank my supervisor Assistant Prof. Dr. Berna Örs Yalçın for her guidance and support during my thesis work.

I am also grateful to my family for their encouragement and support.

I also would like to thank my husband, for his patient and endless support.

September 2010

Dilek BAYHAN GÜMÜŞ

Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS ..... **ix**
LIST OF SYMBOLS ..... **xi**
LIST OF TABLES ..... **xiii**
LIST OF FIGURES ..... **xv**
LIST OF ALGORITHMS ..... **xvii**
SUMMARY ..... **xix**
ÖZET ..... **xxi**
1. INTRODUCTION **1**
2. CRYPTOGRAPHIC SYSTEMS **3**
2.1 Symmetric Key Cryptosystems 3
2.2 Public Key Cryptosystems 4
3. THE RSA CRYPTOSYSTEM **7**
3.1 Mathematical Background 7
3.2 RSA Algorithm 10
3.2.1 Modular exponentiation methods 10
3.2.2 Modular multiplication methods 15
4. POWER OPTIMIZATION TECHNIQUES **21**
4.1 Platform Dependent Power Optimization Techniques 21
4.2 Platform Independent Power Optimization Techniques 22
4.2.1 Glitching 22
4.2.2 Clock gating 24
4.2.3 Operand isolation 25
4.2.4 Re-timing 26
5. LOW POWER IMPLEMENTATION OF MONTGOMERY ALGORITHM **29**
5.1 Background 29
5.1.1 Montgomery's algorithm 29
5.1.2 Previous work 31
5.2 Varients of Multiplier Architectures 33
5.2.1 Parallel architecture of modular montgomery multiplier 33
5.2.2 Sequential architecture of modular montgomery multiplier 35
5.2.3 Systolic architecture of modular montgomery multiplier 37
5.3 Implementation Results 37
5.4 Discussion on Implementation Results 40
6. LOW POWER IMPLEMENTATION OF RSA ALGORITHM **41**
6.1 Binary Modular Exponentiation 41
6.1.1 Background on binary algorithm and hardware architecture 41
6.1.2 Verification of RSA implementation 45
6.1.3 Implementation results and comparison with previous works 47
6.2 The Sliding Window Techniques 51
6.2.1 Background on sliding window techniques and hardware architecture 51

6.2.2 Implementation results and comparison with previous works 58
7. CONCLUSION 60
REFERENCES ..... Error! Bookmark not defined.
CURRICULUM VITA 65

ABBREVIATIONS

FPGA	**:** Field Programmable Gate Array
RSA	**:** Rivest, Shamir, Adleman
MMM	**:** Montgomery Modular Multiplication
MSB	**:** Most Significant Bit
CLNW	**:** Constant Length Nonzero Window
VLNW	**:** Variable Length Nonzero Window
ZW	**:** Zero Window
NW	**:** Nonzero Window
CMOS	**:** Complementary Metal-Oxide Semiconductor
SRAM	**:** Static Read Access Memory
HDL	**:** Hardware Design Language
PE	**:** Processing Element

LIST OF SYMBOLS

M	**:** Message, plaintext
N	**:** Modulus, public-key
E	**:** Exponent, public-key
D	**:** Private-key
Φ(N)	**:** Euler's totient function of N
gcd()	**:** Great common divisor function

LIST OF TABLES

Page

Table 3.1: Steps of binary modular exponentiation operation 11
Table 3.2: The multiplications required by the binary method 11
Table 3.3: Quaternary method.....12
Table 3.4: Steps of *m*-ary modular exponentiation operation 13
Table 3.5: The average multiplications required by the *m*-ary method 13
Table 5.1: Performance comparison of modular multiplication.....32
Table 5.2: Example execution of parallel architecture 35
Table 5.3: Example execution of sequential architecture 36
Table 5.4: Implementation results of three architectures.....39
Table 6.1: Performance comparison of binary modular exponentiation.....51
Table 6.2: Performance comparison of RSA sliding window method 59

LIST OF FIGURES

Page

Figure 1.1 : RSA system based on montgomery algorithm. 1
Figure 2.1 : Symmetric key cryptosystem communication channel.....3
Figure 2.2 : Flow of information in public key system. 4
Figure 3.1 : The state diagram of Algorithm 3.4.....15
Figure 4.1 : Glitch caused by hazard. 22
Figure 4.2 : Glitch example.....22
Figure 4.3 : Reducing glitches by adding register blocks.....23
Figure 4.4 : The circuit that has unbalanced routing delays. 23
Figure 4.5 : The circuit that has balanced routing delays. 24
Figure 4.6 : Enable register with multiplexer.....24
Figure 4.7 : Clock gated register. 25
Figure 4.8 : Design without operand isolation. 25
Figure 4.9 : Design with operand isolation. 26
Figure 4.10 : Design without re-timing.....27
Figure 4.11 : Design with re-timing. 27
Figure 5.1 : Parallel architecture of modular montgomery multiplier.....33
Figure 5.2 : Basic PE architecture. 33
Figure 5.3 : Right border PE, cell0.....34
Figure 5.4 : Sequential architecture of modular montgomery multiplier. 35
Figure 5.5 : Systolic architecture of modular montgomery multiplier. 37
Figure 6.1 : State diagram of binary modular exponentiation RSA algorithm step 1. 43
Figure 6.2 : State diagram of binary modular exponentiation RSA algorithm step 2. 44
Figure 6.3 : Final architecture of the exponentiator. 45
Figure 6.4 : Verification of RSA implementation. 46
Figure 6.5 : Verification of RSA implementation, second step. 47
Figure 6.6: Encryption result of RSA implementation. 49
Figure 6.7: Decryption result of RSA implementation.....50
Figure 6.8 : State diagram of sliding window method RSA algorithm step 1. 55
Figure 6.9 : State diagram of sliding window method RSA algorithm step 2. 56
Figure 6.10 : State diagram of sliding window method RSA algorithm step 3. 57
Figure 6.11 : Architecture of sliding window based exponentiator. 58

LIST OF ALGORITHMS

Page

Algorithm 3.1: Key generation for RSA public-key encryption.....8
Algorithm 3.2: The binary method - left to right 10
Algorithm 3.3: The *m*-ary method.....12
Algorithm 3.4: The sliding window technique 14
Algorithm 3.5: Multiple-precision multiplication 16
Algorithm 3.6: Multiple-precision division 16
Algorithm 3.7: Classical modular multiplication 17
Algorithm 3.8: Montgomery modular multiplication 18
Algorithm 5.1: Montgomery modular multiplication 30
Algorithm 5.2: Modified Montgomery modular multiplication 31
Algorithm 6.1: Binary modular exponentiation.....41
Algorithm 6.2: Key generation for RSA public-key encryption 47
Algorithm 6.3: The sliding window method 52
Algorithm 6.4: Constant length nonzero window 53
Algorithm 6.5: Variable length nonzero window 53

POWER EFFICIENT FPGA IMPLEMENTATION OF RSA ALGORTIHM

SUMMARY

Cryptographic algorithm applications have been becoming widespread. In early, cryptographic circuit designs focused on high speed and high throughput. As its applications growing up on power and area resource-limited platforms, power and area efficient implementations are becoming more important. This is why there has been lots of research on low power design which advise different methods for reducing power consumption of implementations.

In this study, dynamic power consumptions of Field Programmable Gate Array (FPGA) implementations of the Rivest, Shamir, Adleman (RSA) has been reduced by using low power design methods. RSA is one of the most popular public key cryptographic algorithms. It is used in public signatures applications and generally in secure transactions. It offers good cryptographic security but due to its demanding mathematical calculation complexity, it lacks in speed when compared to symmetric key algorithms. That fact leads to a well founded need for speeding up the calculations for the RSA cryptosystem.

The mathematics behind RSA algorithm, are summarized in two operations, modular multiplication and modular exponentiation. In the RSA cryptosystem, the arithmetic operation M^E mod N is used, where N is a prime product of two relative prime numbers, M is the message and E is the public key. In order to create an efficient implementation of RSA, one has to design efficiently the multiplication of two modular numbers. So this mathematical background provides a good understanding that Modular Multiplication block dissipates the most of the power, dissipated in RSA. For comparison of power dissipations, different methods are used to implement Modular Multiplication block. Then RSA implemented by using Sequential Binary Modular Exponentiation which has widespread applications. Computer simulations have been used to show that the implementations of the algorithm generate correct outputs against test vectors.

Low power design techniques are examined and power consumption of implemented architectures of RSA algorithm is reduced by using these techniques. Because of much power dissipation, implemented Modular Multiplication block by using different methods are improved so that their power dissipation is reduced. In this study, three types of architecture are implemented for Modular Multiplication operation: Parallel Architecture of Modular Montgomery Multiplier, Sequential Architecture of Modular Montgomery Multiplier and Systolic Architecture of Modular Montgomery Multiplier. These low power techniques on Modular Multiplication architectures are compared according to the power dissipations and area requirements.

RSA ALGORİTMASININ DÜŞÜK GÜÇ TÜKETİMLİ FPGA TASARIMI

ÖZET

Kriptografik algoritmalar gün geçtikçe daha yaygın kullanım alanları bulmaktadır. İlk zamanlarda yüksek hız ve yüksek işlem gücü kapasitesine sahip devreler tasarlanmaya çalışılırken, enerji ve alan kısıtına sahip ortamlarda kullanım alanlarının artmasıyla güç ve alan tasaruflu gerçeklemeler büyük önem kazanmıştr. Bu konu üzerine artarak devam eden araştırmalarda düşük güç tasarrufu için farklı yöntemler önerilmekte ve gerçeklemelerin güç harcamaları azaltılmaya çalışılmaktadır.

Bu çalışmada Rivest, Shamir, Adleman (RSA) algoritması sahada programlanabilir kapı dizisi (FPGA: Field Programmable Gate Array) üzerinde gerçeklenmekte ve güç tasarruf yöntemlerinden yararlanılarak dinamik güç harcamaları azaltılmaktadır. RSA algoritması en yaygın kullanıma sahip açık anahtarlı şifreleme algoritmalarından biridir. RSA algoritması açık imza uygulamalarında ve genellikle güvenlik gerektiren işlemlerde kullanılır. RSA algoritması iyi derecede kriptografik güvenlik sağlamaktadır; ancak bu özelliğine karşılık karmaşık matematiksel hesaplamalar içermesinden dolayı simetrik anahtarlı algoritmalara kıyasla hızı düşüktür. Bu durum, RSA kriptosistemi içerisinde yer alan hesapların hızlandırılması ihtiyacını doğurmaktadır.

RSA algoritmasını oluşturan matematiksel temel işlemleri iki ana başlıkta toplamak mümkündür: moduler çarpma işlemi ve moduler üs alma, exponent işlemi. RSA algoritmasında kullanılan aritmetik işlem M^E mod N işlemidir. Bu işlemdeki N sayısı aralarında asal iki sayının çarpımından oluşan modulo değeri, M mesaj ya da düz metin dediğimiz bilgi, E ise açık anahtar olarak bilinen değerdir. İyi bir RSA gerçeklemesi oluşturmak istenirse; yapılması gereken en önemli şey, iyi bir modular çarpma devresi oluşturmaktır. Bu matematiksel açıklamalardan yola çıkararak anlamalıyız ki; bir RSA gerçeklemesinde en çok güç tüketen blok modular çarpma devresidir. Bu nedenle güç tüketimlerinin karşılaştırılması açısından modular çarpma devresine farklı teknikler uygulanmıştır. Daha sonra çok yaygın bir kullanıma sahip olan ardışıl ikili modular üs alma (Sequential Binary Modular Exponentiation) tekniği ile RSA algoritması gerçeklenmiştir. Bilgisayar benzetim programı ile yapıların test vektörü girişlerine karşılık doğru sonuçlar verdiği gösterilmiştir.

Literatürde yer alan güç tasarruf yöntemleri incelenmiş ve bu yöntemler kullanılarak farklı yapılarda gerçeklenen RSA algoritmaları üzerinde güç tasarrufu sağlanmıştır. Güç harcamasının fazla olması nedeniyle Modular Çarpma devresi üzerine yoğunlaşılmış ve farklı yöntemlerle gerçeklenen Modular Çarpma devresi üzerinde iyileştirmeler yapılmıştır. Bu çalışmada modular çarpma işlemi için üç farklı yapıda devre tasarımı gerçeklenmiştir: Modular Montgomery Çarpma devresinin paralel yapıda tasarımı, Modular Montgomery Çarpma devresinin ardışıl yapıda tasarımı ve Modular Montgomery Çarpma devresinin sistolik yapıda tasarımı. Daha sonra bu devreler üzerinde dinamik güç harcamaları karşılaştırılmış, ayrıca devrelere ait alan bilgileri de incelenmiştir.

1. INTRODUCTION

With the widespread popularity of electronic communication and commerce, data security issues have become of increasing concern. Among the various encryption algorithms used for security purposes, the public key cryptosystems are the most popular, due to both confidentiality and authentication facilities. The Rivest-Shamir-Adleman (RSA) cryptosystem [1] is one of the best-known public key cryptosystems based on the difficulty of the factorization of large integers [2]. The security of the RSA algorithm is based on the difficulty of solving the integer factorization problem. The main operation of the algorithm is modular exponentiation given as $C=M^E$ mod N for encryption and $M=C^D$ mod N for decryption, where M is the plain text, C is the cipher text, E is the public key, D is the private key. N is the modulus of the operation and equals to product of two large primes, p,q [3].

The large bit over 1024-bit modular operation makes the RSA system difficult to implement. For solving this problem, Montgomery modular reduction algorithm is usually adopted for modular multiplication. Figure 1.1 shows the RSA system based on Montgomery algorithm [4].

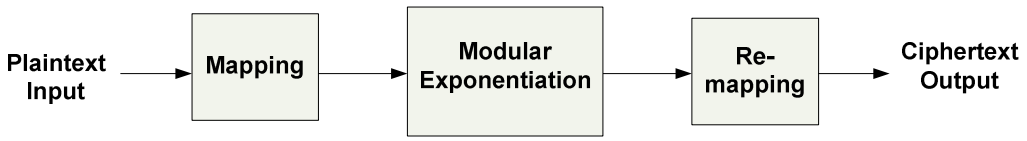

Figure 1.1 : RSA system based on montgomery algorithm.

In this study, a hardware architecture of the RSA cryptosystem has been proposed and implemented on Xilinx FPGA families. Low power design techniques are examined and power consumption of implemented architectures of RSA algorithm is reduced by using these techniques. Because of much power dissipation, implemented Modular Multiplication block by using different methods are improved so that their power dissipation is reduced. In this study, three types of architecture are implemented for Modular Multiplication operation: Parallel Architecture of Modular Montgomery Multiplier, Sequential Architecture of Modular Montgomery Multiplier and Systolic Architecture of Modular Montgomery Multiplier. Then RSA implemented by using Sequential Binary Modular Exponentiation, which has widespread applications.

This thesis presents a power efficient FPGA implementation of the RSA cryptosystem.

Chapter 2 presents the basics of cryptographic systems and explains about the main types of cryptosystems.

Chapter 3 explains the mathematical background behind the RSA cryptosystem and the fundamentals of RSA architecture both algorithmic and hardware based.

Chapter 4 explains the power optimization techniques: both platform dependent and platform independent power optimization techniques.

Chapter 5 explains the implementation done within this study: Montgomery Algorithm, Modular Multiplication Methods.

Chapter 6 explains the implementation done within this study: RSA Algorithm, Binary Exponentiation Method.

Chapter 7 is a review of the thesis and the conclusion is given.

2. CRYPTOGRAPHIC SYSTEMS

We stand today on the brink of a revolution in cryptography. The development of cheap digital hardware has freed it from the design limitations of mechanical computing and brought the cost of high grade cryptographic devices down to where they can be used in such commercial applications. In turn, such applications create a need for new types of cryptographic systems which minimize the necessity of secure key distribution channels and supply the equivalent of a written signature. At the same time, theoretical developments in information theory and computer science show promise of providing provably secure cryptosystems, changing this ancient art into a science [5].

There are two types of cryptosystems: symmetric key cryptosystems and public key cryptosystems.

2.1 Symmetric Key Cryptosystems

The fundamental objective of symmetric key cryptography is to enable two people, usually referred to as Alice and Bob, to communicate over an insecure channel. An opponent Oscar can not understand what is being said [6].

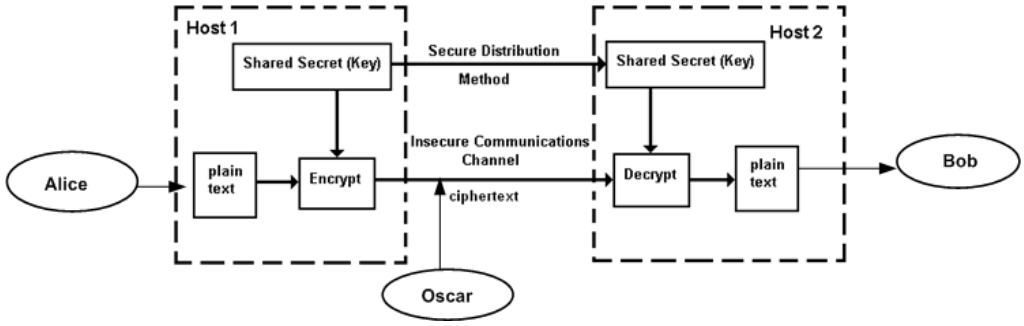

Figure 2.1 : Symmetric key cryptosystem communication channel.

Alice encrypts the plaintext and sends the resulting cipher text over the channel. Oscar, upon seeing the cipher text in the channel by eavesdropping, can not

determine what the plaintext was. However Bob, who knows the encryption key, can decrypt the cipher text and reconstruct the plaintext [6].

One of the major issues with symmetric-key systems is to find an efficient method to agree upon and exchange keys securely. This problem is referred to as the *key distribution problem* [7]. Symmetric-key system requires the prior communication of the key K, between Alice and Bob, using a secure channel, before any cipher text is transmitted. In practice, this may be very difficult to achieve [6]. The second problem is that digital signature is not available in secret key cryptosystems. Since both Alice and Bob share the same secret key, it will be ambiguous who has signed the plaintext [6].

To overcome these problems**,** Diffie and Hellman proposed in 1976 the concept of *public key cryptography*, which has triggered the revolution of cryptography [8].

2.2 Public Key Cryptosystems

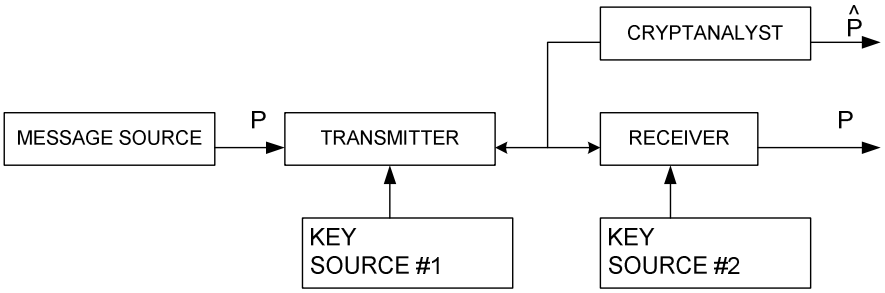

Figure 2.2 : Flow of information in public key system.

Diffie and Hellman proposed that it was possible to develop systems of the type shown in Fig. 2.2, in which two parties communicating solely over a public channel and using only publicly known techniques can create a secure connection. They had examined two approaches to this problem, called public key cryptosystems and public key distribution systems, respectively [5].

As proposed by Diffie and Hellman in [5], a *public key cryptosystem* is a pair of families $\{E_K\}_{K\in\{K\}}$ and $\{D_K\}_{K\in\{K\}}$ of algorithms representing invertible transformations,

$$E_k : \{M\} \rightarrow \{M\} \tag{2.1}$$

$$D_k : \{M\} \rightarrow \{M\} \tag{2.2}$$

on a finite message space $\{M\}$, such that

a. for every $K \in \{K\}$, E_K is the inverse of D_K,

b. for every $K \in \{K\}$ and $M \in \{M\}$, the algorithms E_K and D_K are easy to compute,

c. for almost every $K \in \{K\}$, each easily computed algorithm equivalent to $_K D$ is computationally infeasible to derive from $_K E$,

d. for every $K \in \{K\}$, it is feasible to compute inverse pairs E_K and D_K from K [11].

A function E satisfying (a)-(c) is a "trap-door one-way function" with D_K being the trapdoor information necessary to compute the inverse function and hence allow decryption. This is unlike symmetric-key ciphers where E_K and D_K are essentially the same [7]. If E function also satisfies (d) it is a "trap-door one-way permutation". Diffie and Hellman introduced the concept of trap-door one-way functions in [5], but did not present any examples. These functions are called "one-way" because they are easy to compute in one direction but (apparently) very difficult to compute in the other direction. They are called "trapdoor" functions since the inverse functions are in fact easy to compute once certain private "trap-door" information is known. A trap-door one-way function which also satisfies (d) must be a permutation: every message is the ciphertext for some other message and every ciphertext is itself a permissible message. (The mapping is "one-to-one" and "onto"). Property (d) is needed only to implement "signatures" [1].

In 1978, Rivest, Shamir, and Adleman proposed the RSA public-key cryptosystem, which meets the criteria defined by Diffie and Hellman, has become the most widely used public-key cryptosystem due to the fact that it can be used for both data encryption and authentication.

3. THE RSA CRYPTOSYSTEM

RSA is one of the most popular public key cryptographic algorithms. It is used in public signatures applications and generally in secure transactions. It offers good cryptographic security but due to its demanding mathematical calculation complexity, it lacks in speed when compared to symmetric key algorithms. That fact leads to a well-founded need for speeding up the calculations for the RSA cryptosystem [9].

The mathematics behind RSA algorithm, are summarized in two operations, modular multiplication and modular exponentiation. In the RSA cryptosystem, the arithmetic operation A^C mod N is used, where N is a prime product of two relative prime numbers, A is the message and C is the public key. In order to create an efficient implementation of RSA one has to design efficiently the multiplication of two modular numbers [9].

Before describing how RSA Cryptosystem works, we need to get familiar with some concepts that will be used widely in the rest of the thesis. Eq. (3.1) shows the encryption algorithm, where *M* is the message (plaintext), (*E*, *N*) are the public key pair, and *C* is the cipher text. Eq. (3.2) shows the decryption algorithm where *D* is the private key [10].

$$C = M^{E} \pmod{N} \qquad \textbf{(3.2)}$$

$$M = C^{D} \pmod{N} \qquad \textbf{(3.2)}$$

The detailed mathematical background of RSA and the details of RSA algorithm is given in the following sections of this chapter.

3.1 Mathematical Background

Each entity creates an RSA public key and a corresponding private key. Each entity A should do the following instructions, which are represented as Algorithm 3.1 [7]:

Algorithm 3.1: Key generation for RSA public-key encryption

1. Generate two large random (and distinct) primitives p and q, each roughly the same size.
2. Compute $n = pq$ and $\Phi = (p-1)(q-1)$.
3. Select a random integer e, $1 < e < \Phi$, such that $gcd(e, \Phi) = 1$.
4. Use the extended Euclidean algorithm to compute the unique integer d, $1 < d < \Phi$, such that $ed \equiv 1 \pmod{\Phi}$.
5. A's public key is (n,e); A's private key is d.

One obvious attack on the RSA cryptosystem is for a cryptanalyst to attempt to factor n. If this can be done, it is a simple manner to compute $\Phi(N) = (p-1) \times (q-1)$ and then compute the decryption exponent d from e exactly Bob did [6].

If the RSA Cryptosystem is to be secure, it is certainly necessary that $n = p.q$ must be large enough that factoring it will be computationally infeasible [6]. So, let p and q be two large random (and distinct) numbers, to create encryption and decryption keys, whose products makes up the k-bit N modulus [1].

$$N = pq, p \neq q, 2^{k-1} < N < 2^k \tag{3.3}$$

It is very easy to choose a number E, is relatively prime to $\Phi(N)$. For example, any prime number greater than $max(p,q)$ will do this [1]. E is the public exponent such that the greatest common divisor of E and $\Phi(N)$ is 1 and E is smaller than N [6],

$$\gcd(E, \Phi(N)) = 1, E \in \{1, \ldots, N-1\} \tag{3.4}$$

where $\Phi(N)$ is Euler's totient function of N given by

$$\Phi(N) = (p-1) \cdot (q-1) \tag{3.5}$$

Afterwards we compute the private key D with

$$D = E^{-1} \bmod (\Phi(N)) \tag{3.6}$$

To compute D, it is used Euclid's algorithm for computing the greatest common divisor of $\Phi(N)$ and E [1]. Euclid's algorithm is based on the following observation [10]:

$$\gcd(\Phi(N), E) = \gcd(E, \Phi(N) \bmod E) \tag{3.7}$$

Calculate $gcd(\Phi(N), E))$ by computing a series x_0; x_1; x_2; : : :, where [1]

$$\begin{aligned} x_0 &= \Phi(N) \\ x_1 &= E \\ &. \\ &. \\ &. \\ x_{i+1} &= x_{i-1} (\bmod x_i) \end{aligned} \tag{3.8}$$

until an x_k equal to 0 is found. Then

$$\gcd(x_0, x_1) = x_{k-1}. \tag{3.9}$$

RSA encryption is performed by a modular exponentiation operation as shown by Eq. (3.10) [1]

$$C = M^E \bmod N, \quad C, M, E \in \{0,1,\ldots,N-1\} \tag{3.10}$$

And RSA decryption is realized through the same function as RSA encryption as shown by Eq.(3.11)

$$M = C^D \bmod N, \quad C, M, D \in \{0,1,\ldots,N-1\} \tag{3.11}$$

where M is the plain text, C is the cipher text, N and E are the public keys, and D is the private key.

3.2 RSA Algorithm

The mathematics behind RSA algorithm, are summarized in two operations, modular multiplication and modular exponentiation. In order to create an efficient implementation of RSA one has to design efficiently the multiplication of two modular numbers. However, modular multiplication has a very big drawback; trial division has to be employed to obtain the necessary remainder value [9]. Many attempts have been made to overcome the trial division obstacle [7]. The most popular solution is the Montgomery Modular Multiplication algorithm (MMM), first proposed by P. Montgomery in [11].

The native answer of how to calculate the modular exponentiation operation $C = M^E \bmod N$, is to start with $C := M \bmod N$ initial value and keep on multiplying the result with M continuously for $E-1$ times [10]. This is obviously the most time consuming and infeasible way to do the exponentiation. This explanation gives the answer of question why we need different methods for modular exponentiation operation and why it has been worked on different methods for modular exponentiation operation.

3.2.1 Modular exponentiation methods

There are three types of modular exponentiation methods: the binary method, the *m*-ary method and the sliding window method.

3.2.1.1 The binary method

The "binary method", which is also called the "square and multiply method", scans the bits of exponent E one by one [10]. This scanning can be performed either from left to right or vice a versa. Let E be a k -bit number. The binary method algorithm is given in Algorithm 3.2.

Algorithm 3.2: The binary method – left to right

Inputs: $N = (n_{k-1} \ldots n_1 n_0)$, $E = (e_{k-1} \ldots e_1 e_0)$, $M = (m_{k-1} \ldots m_1 m_0)$.

Output: $C = M^E \bmod N$

1. if $e_{k-1} = 1$ then $C := M$ else $C := 1$
2. for $i = k - 2$ down to 0 do

3. $C := C \cdot C \bmod N$

4. if $e_i = 1$ then $C := C \cdot M \bmod N$

5. return C

For example, $E = 250 = (11111010)$, thus $k = 8$. Initially, $C = M$ since $E_{k-1} = E_7 = 1$.

Table 3.1: Steps of binary modular exponentiation operation

i	e_i	Step 2a	Step 2b
7	1	M	M
6	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
5	1	$(M^3)^2 = M^6$	$M^6 \cdot M = M^7$
4	1	$(M^7)^2 = M^{14}$	$M^{14} \cdot M = M^{15}$
3	1	$(M^{15})^2 = M^{30}$	$M^{30} \cdot M = M^{31}$
2	0	$(M^{31})^2 = M^{62}$	M^{62}
1	1	$(M^{62})^2 = M^{124}$	$M^{124} \cdot M = M^{125}$
0	0	$(M^{125})^2 = M^{250}$	M^{250}

The number of multiplications in this example is 7+5 = 12 [10].

The *binary method* requires (k−1) squaring operations (Step 2a) and multiplications which is equal to the number of 1's in the binary expansion of E, excluding the MSB (Step 2b). The total number of multiplications is illustrated in table 3.2 [10]:

Table 3.2: The multiplications required by the binary method

The Binary Method	Multiplications
Maximum	$2(k-1)$
Minimum	$k-1$
Average	$\frac{3}{2}(k-1)$

3.2.1.2 The *m*-ary method

The *m*-ary method reduces the number of multiplications processed in an exponentiation [12]. The exponent E is scanned r -bits at a time, where $m = 2^r$ and $sr = k$, where k is the bit length of E. Preprocessing is necessary for the exponentiation process, in which the powers of M mod N from 2 to $m-1$ are calculated [13]. The *m*-ary method is given in Algorithm 3.3.

Algorithm 3.3: The *m*-ary method

Inputs: $N = (n_{k-1} \ldots n_1 n_0)$, $E = (e_{k-1} \ldots e_1 e_0)$, $M = (m_{k-1} \ldots m_1 m_0)$.

Output: $C = M^E \bmod N$

1. Compute and store $M^w \bmod N$ for $w = 2,3,4, \ldots, m-1$
2. Decompose E into r -bits words F_i for $i = 0,1,2, \ldots, s-1$, $sr = k$
3. $C := M^{F_{s-1}} \bmod N$
4. for $i = s-2$ down to 0 do
5. $\quad C := C \cdot C^{2^r} \bmod N$
6. $\quad$ if $F_i \neq 0$ then $C := C \cdot M^{F_i} \bmod N$
7. return C

This method is more specifically called the "quaternary method" when $m = 2$ and the "octal method" when $m = 3$ [10].

Table 3.3: Quaternary method

bits	j	M^j
00	0	1
01	1	M
10	2	$M \cdot M = M^2$
11	3	$M^2 \cdot M = M^3$

For example, $e = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$.

Table 3.4: Steps of *m*-ary modular exponentiation operation

Bits	j	M^j
11	M^3	M^3
11	$(M^3)^4 = M^{12}$	$M^{12} \cdot M^3 = M^{15}$
10	$(M^{15})^4 = M^{60}$	$M^{60} \cdot M^2 = M^{62}$
10	$(M^{62})^4 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

The number of multiplications is 2+6+3 = 11.

The Table 3.5 shows the average number of multiplications (including squarings) required by the *m*-ary method [10].

Table 3.5: The average multiplications required by the *m*-ary method

The *m*-ary Method	Average Multiplications
Preprocessing	$2^r - 2$
Squarings	$k - r$
Multiplications	$\left(\frac{k}{r} - 1\right)\left(1 - 2^{-r}\right)$
Total	$2^r - 2 + k - r + \left(\frac{k}{r} - 1\right)\left(1 - 2^{-r}\right)$

3.2.1.3 The sliding window technique

In the *m*-ary method, a zero word makes us skip the multiplication. In order to increase the number of skipped operations and reduce the number of total operations executed, the sliding window technique has been suggested by Bos and Coster and Knuth in [12, 14].

A sliding window exponentiation algorithm decomposes E into zero and nonzero words, which are called windows. In this technique, nonzero words cannot end with 0. Therefore the multiplications in the preprocessing step are only done to evaluate the odd numbers: 3, 5, 7, . . ., $m-1$. The preprocessing multiplications are almost halved [13].

Two algorithms using this technique are "Constant Length Nonzero Window" (CLNW) proposed by Knuth [12], and "Variable Length Nonzero Window" (VLNW) by Bos and Coster [14]. Both algorithms scan the exponent bits from right to left.

In CLNW, the algorithm checks the first bit of the window, if it is a 0, then it becomes a zero window (ZW) and keeps that way until a 1 comes. A 1 starts a nonzero window (NW) and keeps that way for a constant length of *d*-bits.

Algorithm 3.4 shows how the right-to-left VLNW method produces zero and nonzero windows. It is necessary to introduce some notations before describing Algorithm 3.4:

- *d:* the maximum length of the nonzero window;
- *q:* the minimum number of zeros that ends the current nonzero window;
- *k* and *r:* integers satisfying $d = 1 + kq + r$ where $1 \leq r < q$.

Let *WS,* whose domain is $(S_0, S_1, \ldots, S_{k+2}\}$, be the *state* variable associated with the number of scanned bits in the current nonzero window. *The current bit* is the rightmost bit among the bits, which are not scanned (read), and *current j bits are* the rightmost *j* bits among the bits, which are not scanned. *The last 1 bit* is the leftmost 1 bit among the scanned bits. Algorithm 3.4 starts at the rightmost bit of *E* and iterates until the leftmost bit of *E* is reached. Algorithm 3.4 does one of its four operations according to *WS* in each iteration. At the beginning, the current bit is the rightmost bit of *E* and $WS = So$ [15].

Algorithm 3.4: The sliding window technique

- *CASE 1.* $WS = S_0$. Read the current bit. If the current bit is 0, set *WS* as S_0; otherwise, start a nonzero window at the current bit and set $WS = S_1$.
- *CASE 2.* $WS = S_i$ $(1 \leq i \leq k)$. Read current *q* bits. If current *q* bits are all 0's, end the nonzero window at the last 1 bit and set *WS* as S_0; otherwise, set *WS* as S_{i+1}.
- *CASE 3.* $WS = S_{k+1}$. Read current *r* bits. If current *r* bits are all 0's, end the nonzero window at the last 1 bit and set *WS* as S_0; otherwise, set *WS* as S_{k+2}.

- *CASE 4*. $WS = S_{k+2}$. End the nonzero window at the last 1 bit and read the current bit. If the current bit is 0, set WS as S_0; otherwise, start a new nonzero window at the current bit and set $WS = S_1$.

The Figure 3.1 shows state diagram of the Algorithm 3.4;

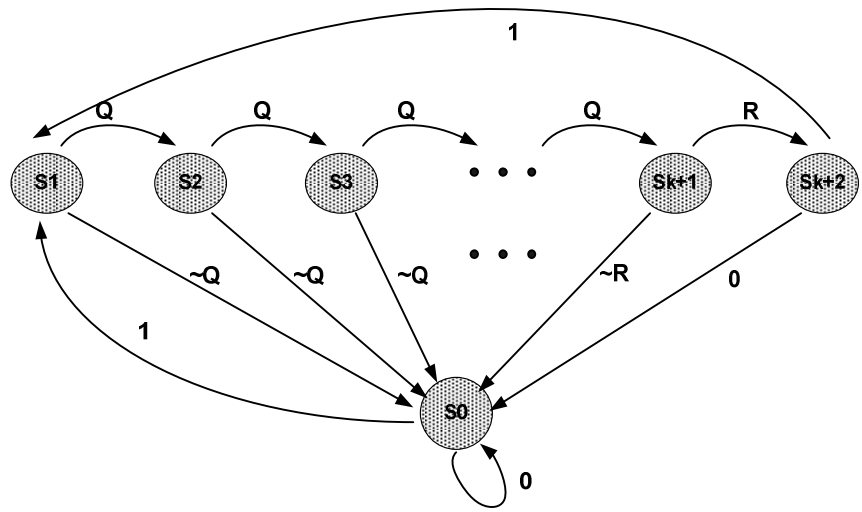

Figure 3.1 : The state diagram of Algorithm 3.4.

Transitions with 0 (1) occur when the scanned bit is 0 (1). Transitions with *Q (R)* occur when *q (r)* scanned bits are not all zeros. Transitions with ~*Q (~ R)* occur when *q (r)* scanned bits are all zeros.

The following example is the output of Algorithm 3.4 when $d = 10$ and $q = 4$ ($k = 2$ and $r = 1$). Nonzero windows are underlined:

E = <u>1011011</u> 0000 <u>1</u> 0000 <u>1111110101</u> 00 <u>11110111</u> 0000 <u>11011</u>.

3.2.2 Modular multiplication methods

There are may types of modular multiplication methods, however it will be introduced first the classical modular multiplication method and then Montgomery modular multiplication method.

3.2.2.1 Classical modular Multiplication

The most straightforward method for performing modular reduction is to compute the remainder on division by m, using a multiple-precision division algorithm such as Algorithm 3.6; this is commonly referred to as the *classical algorithm* for performing modular multiplication.

Classical Algoritm is given in Algorithm 3.7 and the algorithms which are used in classical algorithm are given in Algorithm 3.5 and 3.6 respectively [7].

Algorithm 3.5: Multiple-precision multiplication

Inputs: positive integers x and y having n+1 and t+1 base b digits, respectively.

Output: the product $x \cdot y = (w_{n+t+1} \ldots w_1 w_0)_b$ in radix b representation.

1. For i from 0 to $(n + t + 1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to t do the following:
   (1) $c \leftarrow 0$.
   (2) For j from 0 to t do the following:
   Compute $(uv)_b = w_{i+j} + x_j \cdot y_i + c$, and set $w_{i+j} \leftarrow v$, $c \leftarrow u$.
   (3) $w_{i+n+1} \leftarrow u$.
3. Return($(w_{n+t+1} \ldots w_1 w_0)$).

Algorithm 3.6: Multiple-precision division

Inputs: positive integers $x = (x_n \ldots x_1 x_0)_b$, $y = (y_t \ldots y_1 y_0)_b$ *with* $n \geq t \geq 1$, $yt \neq 0$.

Output: the quotient $q = (q_{n-t} \ldots q_1 q_0)_b$ and remainder $r = (r_t \ldots r_1 r_0)_b$ such that $x = qy + r$, $0 \leq r < y$.

1. For j from 0 to $(\mathrm{n} - t)$ do: $q_j \leftarrow 0$.
2. While $(x \geq ybn{-}t)$ do the following: $q_{n-t} \leftarrow q_{n-t}+1$, $x \leftarrow x - yb^{n-t}$.
3. For i from n down to $(t + 1)$ do the following:
   (1) If $x_i = y_t$ then set $q_{i-t-1} \leftarrow b-1$; otherwise set $q_{i-t-1} \leftarrow \lfloor (x_i b + x_{i-1}) / y_t \rfloor$.
   (2) While $(q_{i-t-1}(y_t b + y_{t-1}) > x_i b^2 + x_{i-1} b + x_{i-2})$ do: $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
   (3) $x \leftarrow x - q_{i-t-1} y b^{i-t-1}$.
   (4) If $x < 0$ then set $x \leftarrow x + yb^{i-t-1}$ and $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
4. $r \leftarrow x$.
5. Return(q,r).

Algorithm 3.7: Classical modular multiplication

Inputs: two positive integers x, y and a modulus m, all in radix b representation.

Output: $x \cdot y \bmod m$

1. Compute $x \cdot y$ (using Algorithm 3.5).
2. Compute the remainder r when $x \cdot y$ is divided by m (using Algorithm 3.6).
3. Return(r).

3.2.2.2 Montgomery modular multiplication

In 1985 Montgomery introduced a new method for modular multiplication [11]. Montgomery reduction is a technique which allows efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step [7]. The approach of Montgomery avoids the time consuming trial division that is the common bottleneck of other algorithms. His method is proven to be very efficient and is the basis of many implementations of modular multiplication in hardware as well as software [16].

The modular exponentiation in RSA obviously requires repeated modular multiplications. In 1985, Montgomery introduced an algorithm for computing $R = ab \bmod N$, which is in total, more efficient than first multiplying and afterwards finding the N residue, which would have required k times k-bit additions for the multiplication, and k times k-bit subtractions and comparisons for the division [11].

The Montgomery algorithm computes the result by replacing the division operation with k times division by a power of 2, where a, b and N are k-bit binary numbers [13].

Given an integer $a<N$; where N is the k-bit modulus, A is said to be its N-residue with respect to r if:

$$A = a \cdot r \bmod N \tag{3.12}$$

where $r = 2^k$.

Likewise, given an integer $b<N$; B is said to be its N-residue with respect to r if:

$$B = b \cdot r \bmod N \tag{3.13}$$

The Montgomery product of A and B can then be defined as:

$$R' = A \cdot B \cdot r^{-1} \mod N \qquad \textbf{(3.14)}$$

where r^{-1} is the inverse of r; modulo N.

When Eq.(3.12), (3.13) and (3.14) are combined, we get

$$R' = a \cdot r \cdot b \cdot r \cdot r^{-1} \mod N = a \cdot b \cdot r \mod N \qquad \textbf{(3.15)}$$

Eq.(3.12), Eq.(3.13) is the preprocessing of Montgomery Multiplication. As R' is not the final result of the multiplication, we need a post-processing, where R' and 1 are the multiplicands of the Montgomery Multiplication, shown in Eq.(3.16)[17].

$$R = (a \cdot b \cdot r) \cdot 1 \cdot r^{-1} \mod N = a \cdot b \mod N \qquad \textbf{(3.16)}$$

The modulus N must also be an odd number, a condition always satisfied in an RSA cryptosystem. Because a conversion to and from N-residue format is required when using Montgomery multiplication, its use only really becomes attractive in applications requiring many repeated modular multiplications. [18]. Thus, Montgomery Multiplication is suitable for RSA.

The radix-2 version of Montgomery's multiplication algorithm [19], which calculates the Montgomery product of A and B, is summarized in the pseudo-code below [18].

The detail of Montgomery multiplication and the details how it is implemented in our implementation will be explained in Chapter 5.

Algorithm 3.8: Montgomery modular multiplication

Inputs: $N = (n_{k-1} \ldots n_1\ n_0)_2$, $A = (a_{k-1} \ldots a_1\ a_0)_2$, $B = (b_{k-1} \ldots b_1\ b_0)_2$, $r = 2^k \bmod N$, $n_0 = 1$

Output: $Montgomery(A, B, N) = A.B.r^{-1} \bmod N$

int $R := 0$;

6. for $i := 0$ to $k-1$ do

7. $q_i := (r_0 + a_i \times b_0) \bmod 2$;

8. $R := (R + a_i \times B + q_i \times N) div 2$;

9. return *R;*

end.

4. POWER OPTIMIZATION TECHNIQUES

Platform dependent power optimization techniques which are explained in part 4.1 are not used during the implementation of the thesis; these techniques are stated in order to provide a background for future work on this subject.

4.1 Platform Dependent Power Optimization Techniques

Platform dependent power optimization techniques are implemented by using the opportunites which are provided by the implementation platform. One of the power optimization techniques are sleep mode operation which is called as power gating. The static power consumption of a CMOS circuit is caused by the leakage currents of transistors and pn junctions [20]. Especially SRAM based FPGA platform causes the circuit consumes a huge amount of static power caused by the leakage currents when the circuit is off. Power gating method prevents the power consumption by using sleep mode for the states that the circuit is off [17].

There are memory blocks in the FPGA's which cause the dynamic power consumption. If there are input ports to read or write on these memory blocks, these inputs are allowed to disable for the memory blocks which are not used at that time. In this way, dynamic power consumption of unused memory blocks is prevented. [17].

The clock signal in the FPGA has to reach for every single sequential block; so it has a long routing line. These long routing lines causes power dissipation by charging and discharging the nodes capacitance, which is therefore also referred to as the capacitive power dissipation. It is also obvious that clock signal has a high frequency of logic level change; this is why its dynamic power consumption is high. So there is a way of power optimization by preventing of clock routing to the blocks which are unused. This feature is available on some of FPGA platforms [17].

4.2 Platform Independent Power Optimization Techniques

4.2.1 Glitching

Glitches are unwanted transitions of a signal after an input change until the final output value is reached. This behavior is due to different arrival times of signals to a gate, called logic hazards. Figure 4.1 shows the circuit for the logic equation $Q = AB + BC$ which exhibits a static-1 hazard. When the inputs A and C are logic 1 any change on B will cause a transition on Q. There are two paths for B to the output Q where one path contains an inverter. This causes a slightly longer delay, resulting in a glitch in the output Q [20]. More complex circuits e.g. ripple carry adders, amplify this problem. In typical combinational circuits glitching accounts for between 10% and 40% of the dynamic power consumption. Hazards and therefore glitches can be avoided at the cost of more circuitry [21].

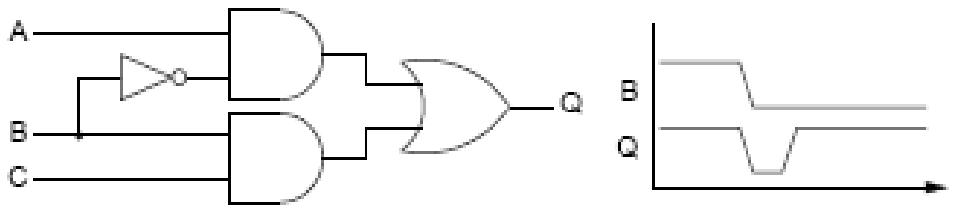

Figure 4.1 : Glitch caused by hazard.

Figure 4.2 gives another example for glitching. Input A is inverted in Δt_1 time period. Because of this delay, on AA' output instead of a constant continues 0 logic value, we see logic 1 for a small time periods.

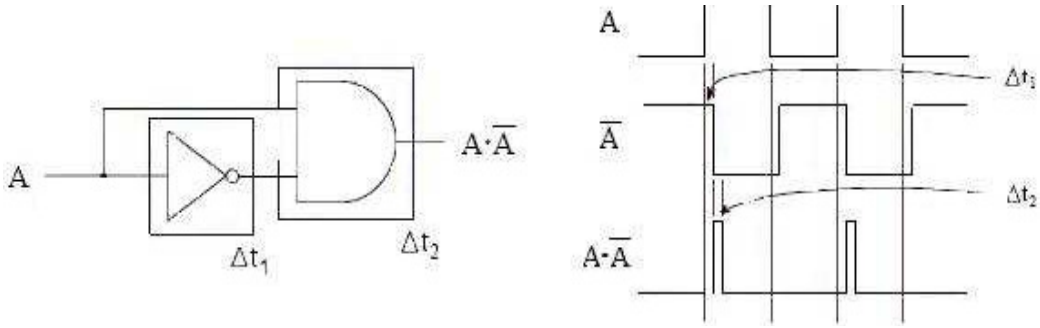

Figure 4.2 : Glitch example.

There are two types of ways in order to solve this glitching problem in the circuit: The first method, which is widely used in our implementation, is to place register blocks between large combinational circuits. These register blocks not only decreases the logic deepness in the circuit, but also increases the clock frequency in the circuit. However, to place these register blocks increases the data processing time. This method is shown in Figure 4.3.

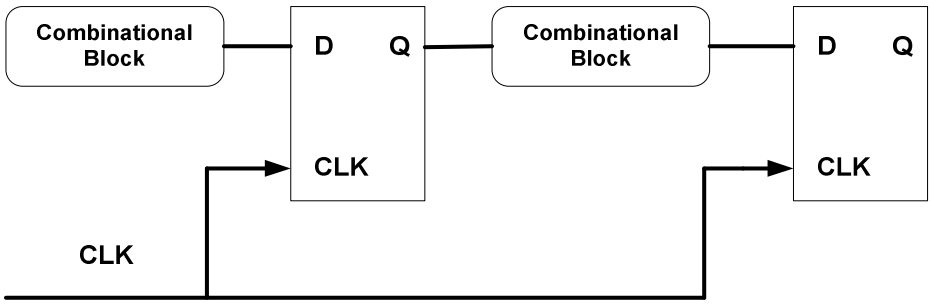

Figure 4.3 : Reducing glitches by adding register blocks.

Second method is to solve glitch problem by reducing the logic deepness of the circuit. This solution is applied to the circuit during the HDL code implementation by using some coding hints. For example the circuit in Figure 4.4 can be converted into a circuit as Figure 4.5 by doing some changes in the HDL where *if, elsif and else* blocks are stated. In this way, both the logic deepness of the circuit and the amount of the glitches are reduced [22].

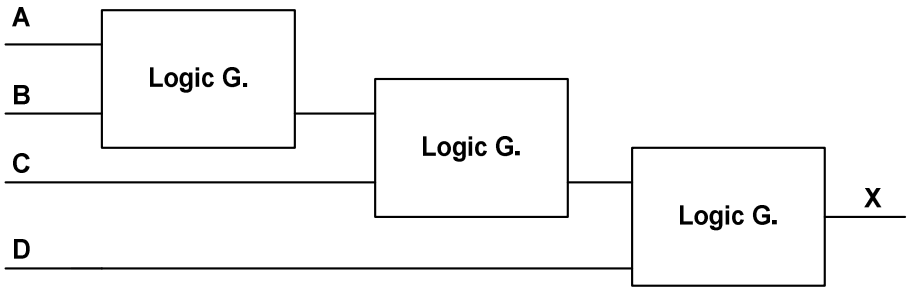

Figure 4.4 : The circuit that has unbalanced routing delays.

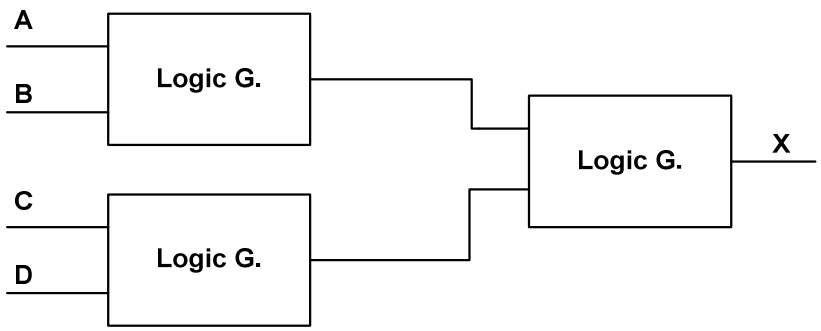

Figure 4.5 : The circuit that has balanced routing delays.

4.2.2 Clock gating

Figure 4.6 shows a typical implementation of a synchronous register with enable. We assume that a register is multiple bits wide and consists of one flip-flop per bit. The register is disabled when the enable signal is at logic 0. Its output is fed back to its input through the multiplexer. When the enable signal is at logic 1 the register can load new values from data in. In this design each flip-flop of the register requires a multiplexer at its data input [20].

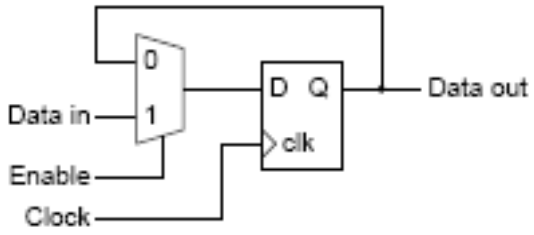

Figure 4.6 : Enable register with multiplexer.

Furthermore the clock network has to drive each flipflop. Clock gating provides a way to disable the clock signals for a register, therefore eliminating the need for separate multiplexers for each input bit. Figure 4.7 shows such a design. The enable signal is usually the output of some combinatorial logic and may contain glitches. The latch prevents glitches from the enable signal to propagate to the clock input of the register. The AND gate performs the actual gating. Clock gating replaces the multiplexers with a single clock gating cell and isolates the register clock from the global clock. The clock gating cell, containing a latch and an AND gate, consumes more power than a single bit multiplexer. However, when this technique is applied to multiple bit registers it can conserve both, static and dynamic power. We observed savings even at registers that were only 8-bits wide [20].

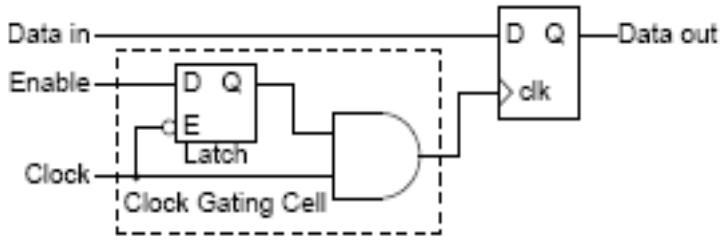

Figure 4.7 : Clock gated register.

4.2.3 Operand isolation

Operand Isolation is a method to selectively stop data from entering a block of complex combinatorial logic, causing many transitions and therefore dynamic power consumption, when the output is discarded by either an unselected multiplexer or a currently disabled register. Figure 4.8 shows an example where changes to the input A consume power even when the output A' is not used.

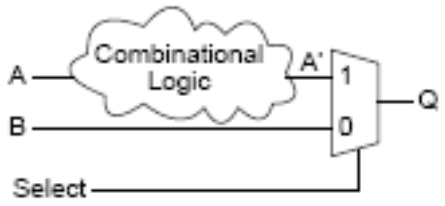

Figure 4.8 : Design without operand isolation.

To prevent this unnecessary power consumption isolation logic can be added at the input to the complex combinatorial logic. It prevents changes to input A from propagating through the combinatorial logic. The isolation logic usually consists of either AND or OR gates depending on the specific application. The example in Figure 4.9 uses an AND gate for operand isolation. The combinatorial logic only receives the input A when its output A' is selected by the multiplexer. Otherwise its input is 0. In this way, it is prevented unnecessary power consumption when control signal Select is not logic 1, which means the output of combinatorial logic is not used [20].

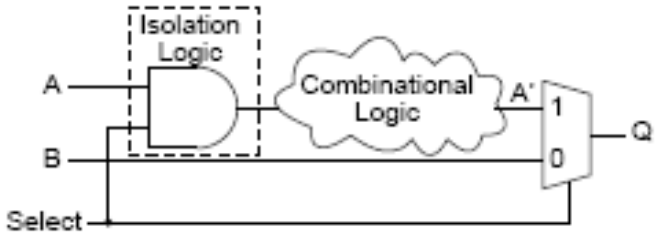

Figure 4.9 : Design with operand isolation.

4.2.4 Re-timing

Retiming for low-power is the process of positioning new or moving existing flip-flops so that they separate parts of the circuit that cause glitching from parts that have high input capacitance. As glitches do not get propagated through flip-flops this technique significantly reduces the switching activity of the high input capacitance part of the circuit and hence reduces the dynamic power consumption [20].

The critical path in Figure 4.10 is decreased by changing the places of registers. The circuit in Figure 4.10 is redrawn in Figure 4.11 after being applied this retiming method [22].

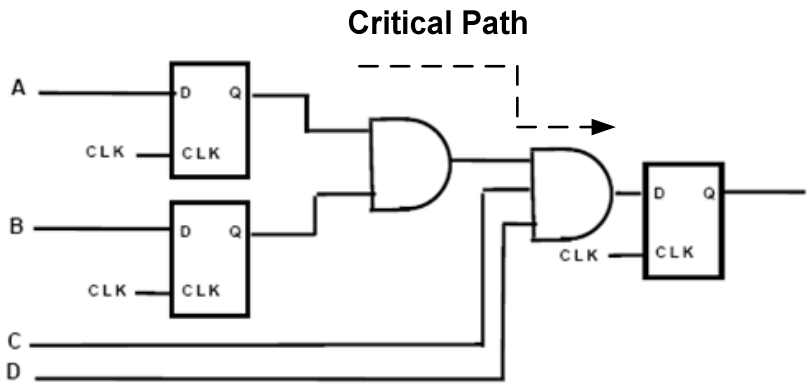

Figure 4.10 : Design without re-timing.

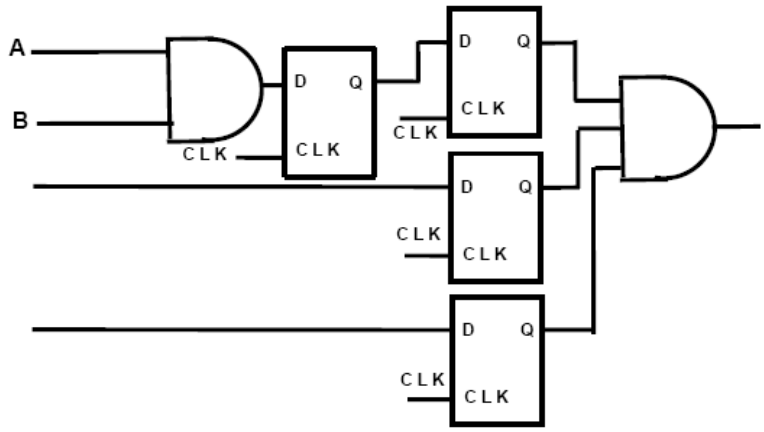

Figure 4.11 : Design with re-timing.

5. LOW POWER IMPLEMENTATION OF MONTGOMERY ALGORITHM

In this study, dynamic power consumptions of FPGA implementations of the RSA have been reduced by using low power design methods. For this purpose, first of all, it is identified that Modular Multiplication block dissipates the most of the power, dissipated in RSA. For comparison of power dissipations, different methods are used to implement Modular Multiplication block.

5.1 Background

Montgomery multiplication is a technique that allows efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step [7] [11]. The method avoids the time-consuming trial division that is the common bottleneck for division algorithms. In fact, this feature makes the Montgomery multiplication "the most popular" approach [16] in hardware and software implementations of cryptographic systems such as RSA, Diffie-Hellman, ECC and many others.

5.1.1 Montgomery's algorithm

Algorithms that formalize the operation of modular multiplication generally consist of two steps: the first generates the product $P = A \ x \ B$ and the second reduces this product P *modulo* N [23].

The straightforward way to implement a multiplication is based on an iterative adder-accumulator for the generated partial products. However, this solution is quite slow as the final result is only available after k clock cycles, where k is the size of the operands [24].

A faster version of the iterative multiplier should add several partial products at once. This could be achieved by unfolding the iterative multiplier and yielding a combinatorial circuit that consists of several partial product generators together with several adders that operate in parallel [24].

One of the widely used algorithms for efficient modular multiplication is Montgomery's algorithm. This algorithm computes the product of two integers a third one without performing division by *N*. It yields the reduced product using a series of additions. It yields the reduced product using a series of additions [23].

Let A, B, and N be the multiplicand, the multiplier and the modulus, respectively, and let *k* be the number of digits in their binary representations. So, we denote A, B and N as follows:

$$A = \sum_{i=0}^{k-1} a_i \times 2^i, \quad B = \sum_{i=0}^{k-1} b_i \times 2^i, \quad N = \sum_{i=0}^{k-1} n_i \times 2^i . \qquad \textbf{(5.1)}$$

The preconditions of the Montgomery algorithm are as follows:

1. The modulus *N* needs to be relatively prime to the radix, i.e., there exists no common divisor for and the radix;
2. The multiplicand and the multiplier need to be smaller than *N*.

As we use the binary representation of the operands, then the modulus *N* needs to be odd to satisfy the first precondition.

The Montgomery Algorithm is that of Algorithm 5.1 which is mentioned first in Chapter 3.

Algorithm 5.1: Montgomery modular multiplication

Inputs: $N = (n_{k-1} \ldots n_1 \; n_0)_2$, $A = (a_{k-1} \ldots a_1 \; a_0)_2$, $B = (b_{k-1} \ldots b_1 \; b_0)_2$, $r = 2^k \bmod N$, $n_0 = 1$

Output: $Montgomery(A, B, N) = A.B.r^{-1} \bmod N$

int $R := 0$;

1. for $i := 0$ to $k-1$ do
2. $q_i := (r_0 + a_i \times b_0) \bmod 2$;
3. $R := (R + a_i \times B + q_i \times N) div 2$;
4. return R;

end.

A bit-wise version of the Algorithm 5.1, which is at the basis of our implementation, is described in Algorithm 5.2. All algorithms, i.e., those of Algorithm 5.1 and Algorithm 5.2 are equivalent. They yield the same result.

Algorithm 5.2: Modified montgomery modular multiplication

Inputs: $N = (n_{k-1} \ldots n_1\ n_0)_2$, $A = (a_{k-1} \ldots a_1\ a_0)_2$, $B = (b_{k-1} \ldots b_1\ b_0)_2$, $NB = (nb_{k-1} \ldots nb_1\ nb_0)_2$, $r = 2^{k+1} \bmod N$, $n_0 = 1$

Output: $Montgomery(A, B, N) = A.B.r^{-1} \bmod N$

int $R := 0$; bit $carry := 0, x$;

1. for $i := 0$ to k do
2. $\quad q_i := r_0^{(i)} \oplus a_i \cdot b_0$;
3. $\quad$ for $j := 0$ to k do
4. $\quad\quad$ switch a_i, q_i
5. $\quad\quad\quad$ 1,1 : x := nb_i;
6. $\quad\quad\quad$ 1,0 : x := b_i;
7. $\quad\quad\quad$ 0,1 : x := n_i;
8. $\quad\quad\quad$ 0,0 : x := 0;
9. $\quad\quad\quad$ $r_j^{(i+1)} := r_{j+1}^{(i)} \cdot x_i \cdot carry$
10. $\quad\quad\quad$ $carry := r_{j+1}^{(i)} \cdot x_i + r_{j+1}^{(i)} \cdot carry + x_i \cdot carry$
11. return R;

end.

In Algorithm 5.2, NB represents the result of $N+B$, which has at most $k+1$ bits.

5.1.2 Previous work

In this thesis, we have studied three different architectures for Montgomery's modular multiplication implemented on the same FPGA technology, thus we provide

a fair comparison platform in order to understand the strengths of the target architectures. These architectures are parallel, systolic and serial.

Eventually, serial architecture reduces area usage and average power consumption while it increases total multiplication time. This explanation gives idea about the starting point of the implementation on serial architecture; however we will be presenting the implementation results and comparison of three architectures by looking on timing, area and power analysis information in section 5.3 .

In Table 5.1 we give a comparison between our work and previous works in the literature. Note that these selected studies all use the Montgomery algorithm for their hardware modular multiplication architectures. For a fair comparison we adjust some of these figures to 1024-bit operation time.

Table 5.1: Performance comparison of modular multiplication

Work	**k**	**Platform**	**Area (Slice)**	**Freq. (MHz)**	**MTime**
Parallel	1024	Virtex-5 XC5VLX50	5360	4.25	240.8 us
Systolic	1024	Virtex-5 XC5VLX50	5822	166	18.44us
Serial	1024	Virtex-5 C5VLX50	3966	416.6	2.516 ms
Systolic [25]	1024	Virtex-5	11346	92.649	1.85 us
Parallel [25]	1024	Virtex-5	3044	95.913	3.71 us
[26]	1024	Xilinx 3090	--	63.7	7.77 µs
[27]	1024	CMOS 0.5 μm	--	80	43 us
[18]	1024	XC2V3000	11520	111.32	8.79 us
[28]	1024	XC2V3000	12284	90.415	5.58 us
[29]	1024	Virtex-II-6	5158	254.55	5.05 us

5.2 Varients of Multiplier Architectures

5.2.1 Parallel architecture of modular montgomery multiplier

Assuming the Algorithm 5.2 as basis, the main processing element (PE) of the parallel architecture of the Montgomery modular multiplier computes a bit r_j of residue R. This represents the computation of line 9. The right-border PE of the line performs the same computation but beside that, it has to compute bit q_i as well. This is related to the computation of line 2.

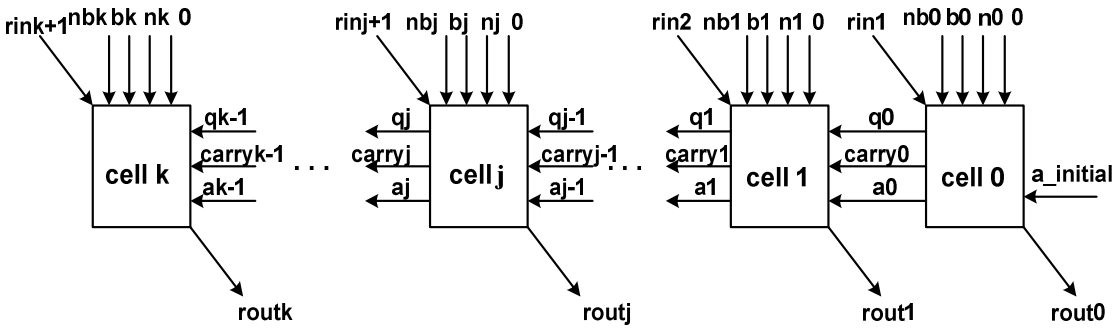

Figure 5.1 : Parallel architecture of modular montgomery multiplier.

The architecture of the basic PE, i.e., $cell_j$, $1 \leq j \leq k-1$ is shown in Figure 5.2. It implements the instructions of lines 3–10 in Modified Montgomery Algorithm 5.2.

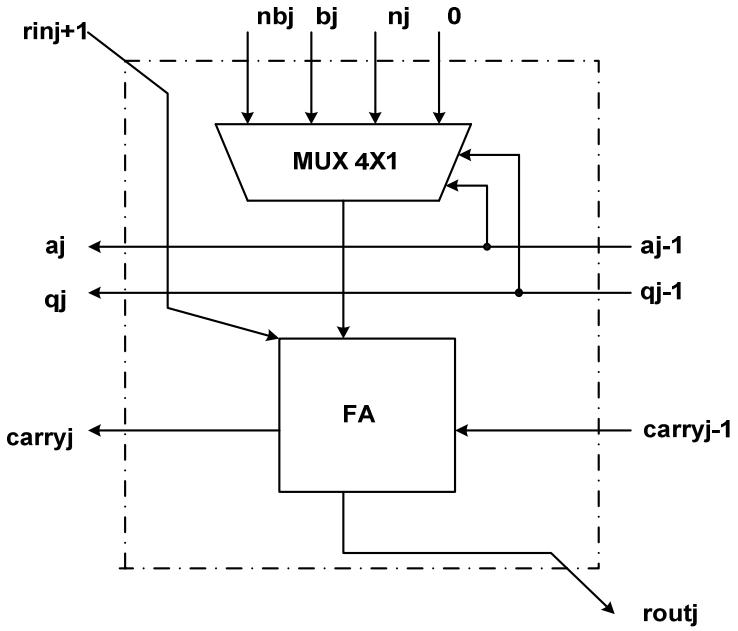

Figure 5.2 : Basic PE architecture.

The architecture of the right border PE, is given in Figure 5.3. Besides the computation of lines 3-10, it implements the computation indicated in line 2, which means it computes q0. Moreover, the full-adder is substituted by a half-adder as the carry in signal is zero for this PE.

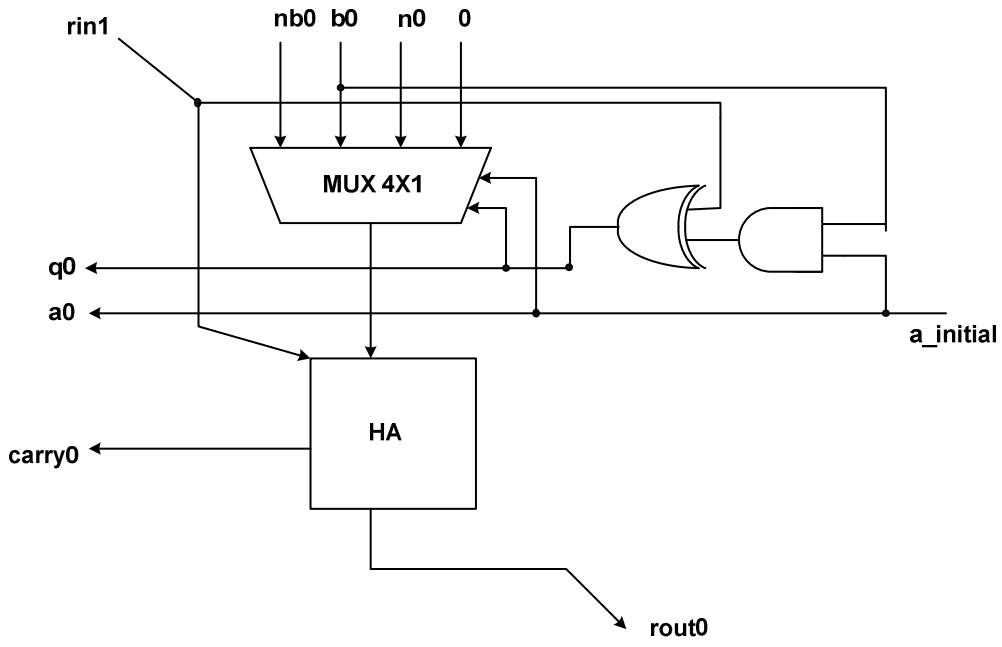

Figure 5.3 : Right border PE, cell0.

The architecture in Figure 5.1 is executed for k+1 cycle as it is stated in Algorithm 5.2. *a_initial* input is changing its value for every cycle. This new value of *a_initial* reaches to all PE's in the architecture at the same time, since there is no register between the combinational blocks. For each cycle, a new result R, which is a subtotal of the final multiplication, occurs. The following, shows each result of k+1 cycles:

1. cycle, i=0: $R_{0,k}\ R_{0,k-1}\ R_{0,k-2}$ …… $R_{0,2}\ R_{0,1} R_{0,0}$

2. cycle, i=1: $R_{1,k}\ R_{1,k-1}\ R_{1,k-2}$ …… $R_{1,2}\ R_{1,1} R_{1,0}$

……

……

k. cycle, $i=k$-1: $R_{k-1,k}\ R_{k-1,k-1}\ R_{k-1,k-2}$ …… $R_{k-1,2}\ R_{k-1,1}\ R_{k-1,0}$

(k+1). cycle, $i=k$: $R_{k,k}\ R_{k,k-1}\ R_{k,k-2}$ …… $R_{k,2}\ R_{k,1}\ R_{k,0}$

Table 5.2 is an example execution of parallel architecture for k is equal to 8. This table shows that for iteration i, the jth cell produces the result $R_{i,j}$.

Table 5.2: Example execution of parallel architecture

	Bitwise Result								
	j=8	j=7	j=6	j=5	j=4	j=3	j=2	j=1	j=0
i=0	$R_{0,8}$	$R_{0,7}$	$R_{0,6}$	$R_{0,5}$	$R_{0,4}$	$R_{0,3}$	$R_{0,2}$	$R_{0,1}$	$R_{0,0}$
i=1	$R_{1,8}$	$R_{1,7}$	$R_{1,6}$	$R_{1,5}$	$R_{1,4}$	$R_{1,3}$	$R_{1,2}$	$R_{1,1}$	$R_{1,0}$
i=2	$R_{2,8}$	$R_{2,7}$	$R_{2,6}$	$R_{2,5}$	$R_{2,4}$	$R_{2,3}$	$R_{2,2}$	$R_{2,1}$	$R_{2,0}$
i=3	$R_{3,8}$	$R_{3,7}$	$R_{3,6}$	$R_{3,5}$	$R_{3,4}$	$R_{3,3}$	$R_{3,2}$	$R_{3,1}$	$R_{3,0}$
i=4	$R_{4,8}$	$R_{4,7}$	$R_{4,6}$	$R_{4,5}$	$R_{4,4}$	$R_{4,3}$	$R_{4,2}$	$R_{4,1}$	$R_{4,0}$
i=5	$R_{5,8}$	$R_{5,7}$	$R_{5,6}$	$R_{5,5}$	$R_{5,4}$	$R_{5,3}$	$R_{5,2}$	$R_{5,1}$	$R_{5,0}$
i=6	$R_{6,8}$	$R_{6,7}$	$R_{6,6}$	$R_{6,5}$	$R_{6,4}$	$R_{6,3}$	$R_{6,2}$	$R_{6,1}$	$R_{6,0}$
i=7	$R_{7,8}$	$R_{7,7}$	$R_{7,6}$	$R_{7,5}$	$R_{7,4}$	$R_{7,3}$	$R_{7,2}$	$R_{7,1}$	$R_{7,0}$
i=8	$R_{8,8}$	$R_{8,7}$	$R_{8,6}$	$R_{8,5}$	$R_{8,4}$	$R_{8,3}$	$R_{8,2}$	$R_{8,1}$	$R_{8,0}$

5.2.2 Systolic architecture of modular montgomery multiplier

Systolic Architecture of Modular Montgomery Multiplier is given in Figure 5.4. It is same as parallel architecture in Figure 5.1, except it has registers between PE blocks. In Figure 5.4, there should be 3 different D flip-flops after each cell. D flip-flop is used in Figure 5.4 is just a representation; it is used like this in order not to repeat the same shape for three times.

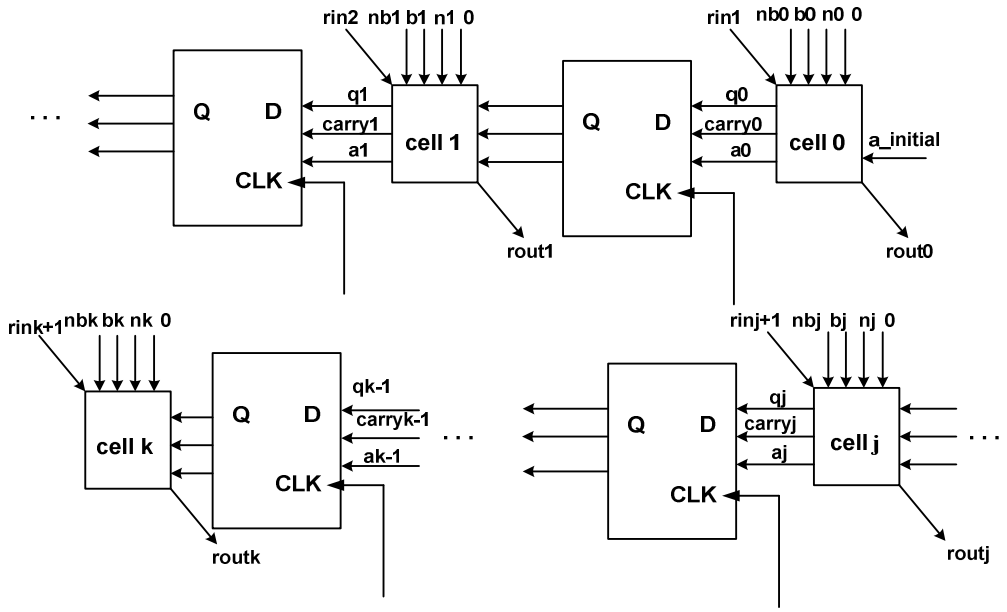

Figure 5.4 : Systolic architecture of modular montgomery multiplier.

In order to get the final result with this architecture we need *2 x (k+1) + k* cycle execution time in total. *a_initial* input is changing its value for every "*2*" cycle. This new value of *a_initial* reaches to next PE in the architecture after "*1*" clock cycle, since there are register blocks between the combinational cells.

Putting registers between combination blocks is a technique to reduce the power consumption. This technique is explained in Chapter 4, in part 4.1.2 Glitching. This method provides to reduce dynamic power consumption, by decreasing the amount of glitches.

To understand how this architecture works is a bit complex by comparing to the parallel architecture. So instead of investigating this architecture for a general *k*, it is better to see how it works on a small example. Table 5.3 gives a good example, which is for, *k* is equal to 8.

Table 5.3: Example execution of systolic architecture

	Bitwise Result								
	j=8	j=7	j=6	j=5	j=4	j=3	j=2	j=1	j=0
Cycle 1	--	--	--	--	--	--	--	--	$R_{0,0}$
Cycle 2	--	--	--	--	--	--	--	$R_{0,1}$	$R_{0,0}$
Cycle 3	--	--	--	--	--	--	$R_{0,2}$	$R_{0,1}$	$R_{1,0}$
Cycle 4	--	--	--	--	--	$R_{0,3}$	$R_{0,2}$	$R_{1,1}$	$R_{1,0}$
Cycle 5	--	--	--	--	$R_{0,4}$	$R_{0,3}$	$R_{1,2}$	$R_{1,1}$	$R_{2,0}$
Cycle 6	--	--	--	$R_{0,5}$	$R_{0,4}$	$R_{1,3}$	$R_{1,2}$	$R_{2,1}$	$R_{2,0}$
Cycle 7	--	--	$R_{0,6}$	$R_{0,5}$	$R_{1,4}$	$R_{1,3}$	$R_{2,2}$	$R_{2,1}$	$R_{3,0}$
Cycle 8	--	$R_{0,7}$	$R_{0,6}$	$R_{1,5}$	$R_{1,4}$	$R_{2,3}$	$R_{2,2}$	$R_{3,1}$	$R_{3,0}$
Cycle 9	$R_{0,8}$	$R_{0,7}$	$R_{1,6}$	$R_{1,5}$	$R_{2,4}$	$R_{2,3}$	$R_{3,2}$	$R_{3,1}$	$R_{4,0}$
Cycle 10	$R_{0,8}$	$R_{1,7}$	$R_{1,6}$	$R_{2,5}$	$R_{2,4}$	$R_{3,3}$	$R_{3,2}$	$R_{4,1}$	$R_{4,0}$
Cycle 11	$R_{1,8}$	$R_{1,7}$	$R_{2,6}$	$R_{2,5}$	$R_{3,4}$	$R_{3,3}$	$R_{4,2}$	$R_{4,1}$	$R_{5,0}$
Cycle 12	$R_{1,8}$	$R_{2,7}$	$R_{2,6}$	$R_{3,5}$	$R_{3,4}$	$R_{4,3}$	$R_{4,2}$	$R_{5,1}$	$R_{5,0}$
Cycle 13	$R_{2,8}$	$R_{2,7}$	$R_{3,6}$	$R_{3,5}$	$R_{4,4}$	$R_{4,3}$	$R_{5,2}$	$R_{5,1}$	$R_{6,0}$
Cycle 14	$R_{2,8}$	$R_{3,7}$	$R_{3,6}$	$R_{4,5}$	$R_{4,4}$	$R_{5,3}$	$R_{5,2}$	$R_{6,1}$	$R_{6,0}$
Cycle 15	$R_{3,8}$	$R_{3,7}$	$R_{4,6}$	$R_{4,5}$	$R_{5,4}$	$R_{5,3}$	$R_{6,2}$	$R_{6,1}$	$R_{7,0}$
Cycle 16	$R_{3,8}$	$R_{4,7}$	$R_{4,6}$	$R_{5,5}$	$R_{5,4}$	$R_{6,3}$	$R_{6,2}$	$R_{7,1}$	$R_{7,0}$
Cycle 17	$R_{4,8}$	$R_{4,7}$	$R_{5,6}$	$R_{5,5}$	$R_{6,4}$	$R_{6,3}$	$R_{7,2}$	$R_{7,1}$	$R_{8,0}$
Cycle 18	$R_{4,8}$	$R_{5,7}$	$R_{5,6}$	$R_{6,5}$	$R_{6,4}$	$R_{7,3}$	$R_{7,2}$	$R_{8,1}$	$R_{8,0}$
Cycle 19	$R_{5,8}$	$R_{5,7}$	$R_{6,6}$	$R_{6,5}$	$R_{7,4}$	$R_{7,3}$	$R_{8,2}$	$R_{8,1}$	x
Cycle 20	$R_{5,8}$	$R_{6,7}$	$R_{6,6}$	$R_{7,5}$	$R_{7,4}$	$R_{8,3}$	$R_{8,2}$	x	x
Cycle 21	$R_{6,8}$	$R_{6,7}$	$R_{7,6}$	$R_{7,5}$	$R_{8,4}$	$R_{8,3}$	x	x	x
Cycle 22	$R_{6,8}$	$R_{7,7}$	$R_{7,6}$	$R_{8,5}$	$R_{8,4}$	x	x	x	x
Cycle 23	$R_{7,8}$	$R_{7,7}$	$R_{8,6}$	$R_{8,5}$	x	x	x	x	x
Cycle 24	$R_{7,8}$	$R_{8,7}$	$R_{8,6}$	x	x	x	x	x	x
Cycle 25	$R_{8,8}$	$R_{8,7}$	x	x	x	x	x	x	x
Cycle 26	$R_{8,8}$	x	x	x	x	X	x	x	x

In Table 5.3, "--" symbol in first 8 cycles means that the result of that cell is not produced yet. "x" symbol in last 8 cycles means that this cell does not produce a new result, instead it keeps its last value.

5.2.3 Serial architecture of modular montgomery multiplier

Serial Architecture of Modular Montgomery Multiplier is given in Figure 5.5. The main difference in this architecture is that it contains only one cell. The aim of this implementation is to see the reduction of power consumption by using less area.

Each time the cell produces one bit of the subtotal of multiplication operation. Therefore, in order to get the final multiplication result, we need *(k+1) x (k+1)* cycles in total.

As it is seen in Figure 5.5, the register A produces a parallel output for every *(k+1)* cycle that gives the subtotal results of the final multiplication.

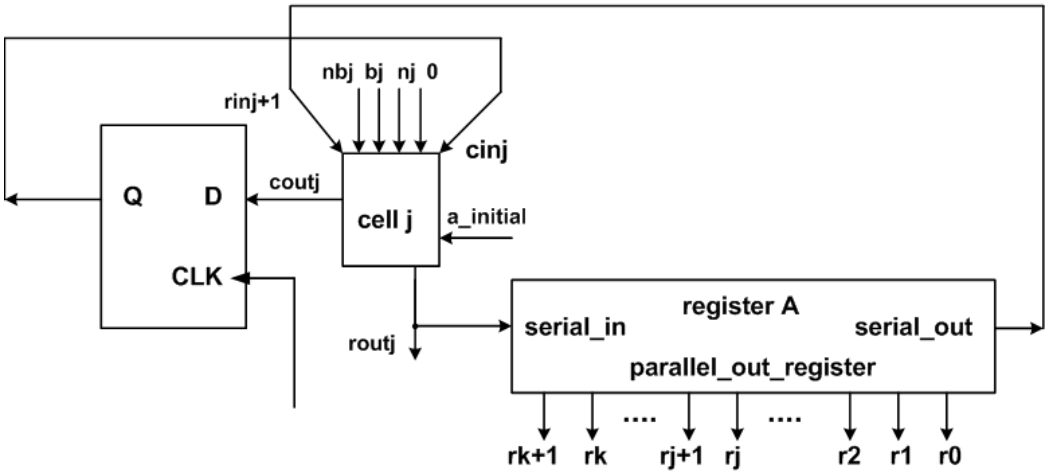

Figure 5.5 : Serial architecture of modular montgomery multiplier.

5.3 Implementation Results

In fact, various architectures have been proposed in the literature for modular multiplication design. Although some of the proposed designs have detailed analysis for their deployment, most studies seem not appropriate for a fair comparison as they rely on specific technologies.

For a fair comparison purpose, we used Virtex-5 XC5VLX50 FPGA family in all our implementations. The main reason for choosing this family is mostly because of their density since typical 1024 bit Montgomery multiplier block needs a fairly big area on the FPGA. To be more specific, roughly 30K slices would provide the necessary logic units and input/output ports for all three kinds of Montgomery multiplier implementations.

In this thesis, VHDL hardware design language is used in code design and Xilinx ISE 10.1 is used in design for the stages of synthesize and implementation (translate, map and place&route). For simulation stage of complete flow, ModelSim XE 6.1e tool is used and for power analysis XPower analyzer, which is embedded into the Xilinx ISE 10.1, is used.

One of most important key point in low power design is to choose the technology of the FPGA. Among three most popular FPGA technologies (i.e. Antifuse, SRAM and FLASH) we decided to use the SRAM based FPGAs since in these FPGAs the static power consumption does not entirely depend on the used gated arrays. In fact, there is a fixed static power consumption independent of the design and the portion that is used on FPGA. The amount of this fixed value depends on the type of the FPGA. In our measurements this fixed static power consumption prevents us to get precise information about the actual leakage power of our designs. Therefore, in our analysis, we decide to use the dynamic power consumption measurements in comparing the Montgomery multiplier implementations.

Table 5.4 reveals the time, area and power consumption measurements of the different types of Montgomery multiplications for a popular key size k =1024 bits. Note that the values L, S and SR stand for number of slice LUTs, number of occupied Slices and number of Slice Registers respectively. Moreover, by “throughput rate” we mean the amount of bits that are processed in one second.

Table 5.4: Implementation results of three architectures

	Architecture Type		
	Parallel Arch.	**Serial Arch.**	**Systolic Arch.**
Area:	11581 L 5360 S 13624 SR	5032 L 3966 S 13689 SR	11905 L 5822 S 18810 SR
Clock Frequency (MHz):	4.25	416.6	166
Throughput Rate (Mb/s):	4.252	0.406	55.53
Total Dynamic Power at Max. freq. (mW):	55	695	571
Total Dynamic Power at 4.25 MHz (mW):	55	47	52
Energy 1 (uJoule):	13.25	1748.62	10.529
Energy 2 (uJoule):	13.25	11596	37.58

Since it is not correct to comment on the total dynamic power consumption at the maximum frequency of the design and not fair to compare different designs at different frequencies, we report dynamic power consumption in two rows. The first row gives the power consumption at maximum frequency where the second one presents it at a fixed frequency for all structures. First energy row in Table 5.4 is calculated by multiplying total dynamic power consumption with total multiplication time from Table 5.1 and this gives energy measurement when all the designs work at their maximum frequency. However, second row gives energy measurements when all realizations work at the same frequency (4.25 MHz).

5.4 Discussion on Implementation Results

We studied the power consumption of parallel, systolic and serial architectures for Montgomery multiplication algorithms. We apply low power design techniques to our architectures. Correspondently, low power techniques on modular multiplication architectures are compared according to the power dissipations and area requirements.

First of all, our serial design is the very first reported realization of Montgomery multiplier with a single PE. As we predict the total dynamic power of this serial design is the best choice for such requirements.

Our second conclusion is that in terms of energy, parallel design gives the best result. In fact this was also expected as it takes less time to multiply means less FPGA running and energy consumption time.

Lastly, we conclude that systolic architectures could be a good trade-off in between, as it gives reasonable power consumption for a fair operation speed.

6. LOW POWER IMPLEMENTATION OF RSA ALGORITHM

In this study, RSA implemented by using Sequential Binary Modular Exponentiation, which has widespread applications. Computer simulations have been used to show that the implementations of the algorithm generate correct outputs against test vectors.

6.1 Binary Modular Exponentiation

6.1.1 Background on binary algorithm and hardware architecture

The sequential straightforward binary modular exponentiation is given in Algorithm 6.1.

Algorithm 6.1: Binary modular exponentiation

Inputs: $N = (n_{k-1} \ldots n_1\ n_0)_2$, $E = (e_{k-1} \ldots e_1\ e_0)_2$, $M = (m_{k-1} \ldots m_1\ m_0)_2$
Output: $M^E \bmod N$

1. $Const := 2^{2(k+1)} \bmod N$
2. $M' := Montgomery(M, Const)$
3. if $e_{k-1} = 1$ then $R' := M'$ else $R' := 1$
4. $Start := 0$
5. for $i = k\text{-}2$ down to 0 do
6. if $Start = 1$ then
7. $R' := Montgomery(R', R')$
8. if $e_i = 1$ then $R' := Montgomery(R', M')$
9. else if $e_i = 1$ then $Start := 1$
10. $R := Montgomery(R', 1)$
11. return R;

The exponentiation is realized by squaring and multiplications, while the bits of the exponent E are scanned. The number E either can be k bits or can be less than k bits. Therefore the multiplications do not start until the actual most significant bit of E where the first '1' is seen. Afterwards a squaring is done for every bit of E, and a multiplication is done if the scanned bit is '1' [30].

Figure 6.1 and 6.2 shows the state diagram of binary modular exponentiation, which is also called square and multiply algorithm. The Montgomery algorithm (Algorithm 5.2) yields the result $2^{-(k+1)} \; x \; A \; x \; B \; mod \; N$. To compute the right result, we need to further Montgomery multiply the result by the constant $2^{(k+1)}$. However, as we are interested rather in the exponentiation result than a simple product, we only need to pre-Montgomery multiply the operands by $2^{2(k+1)} \; mod \; N$ and post-Montgomery multiply the obtained result by "*1*" to get rid of the factor $2^{(k+1)}$ that is carried by every partial result.

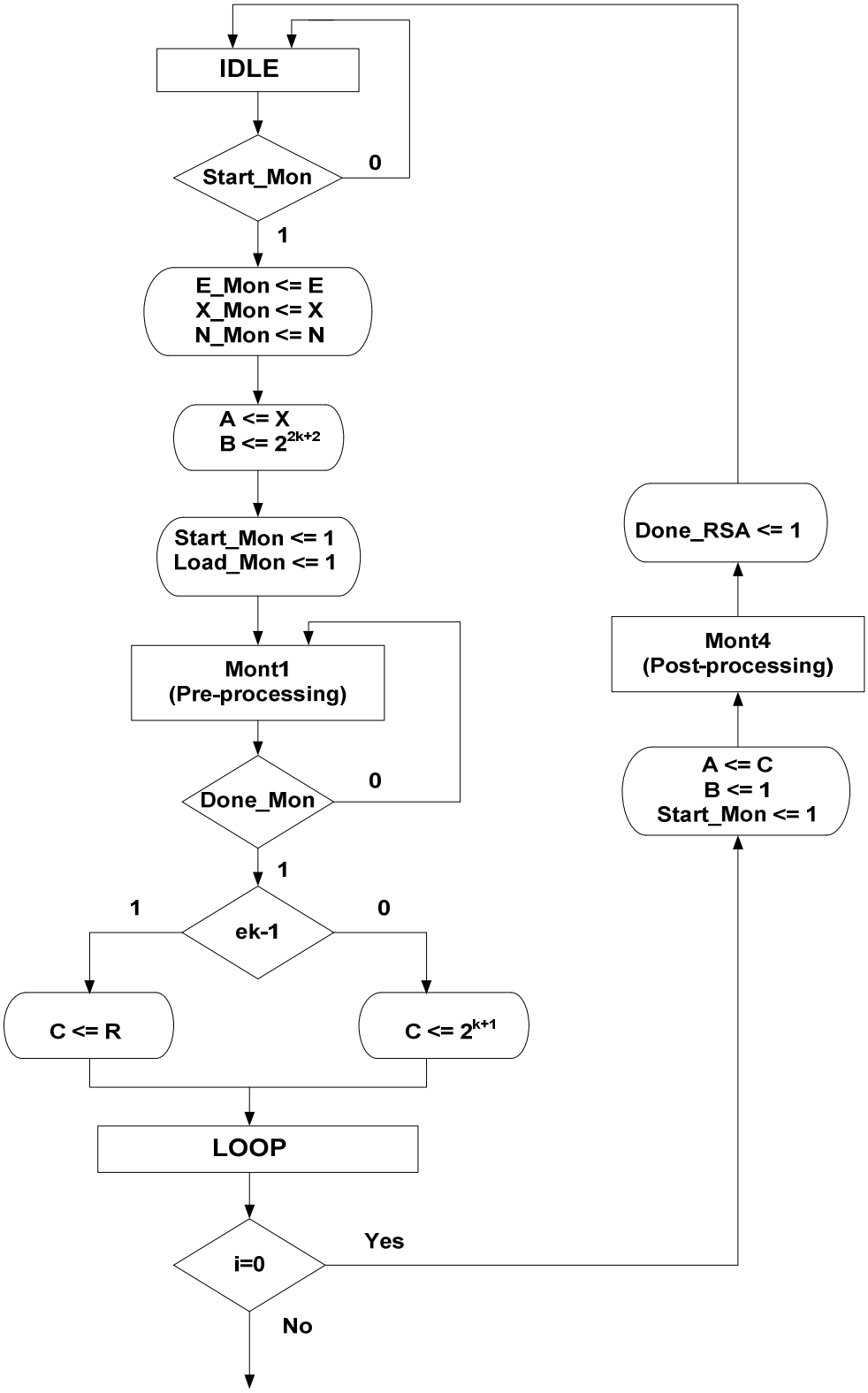

Figure 6.1 : State diagram of binary modular exponentiation RSA algorithm step 1.

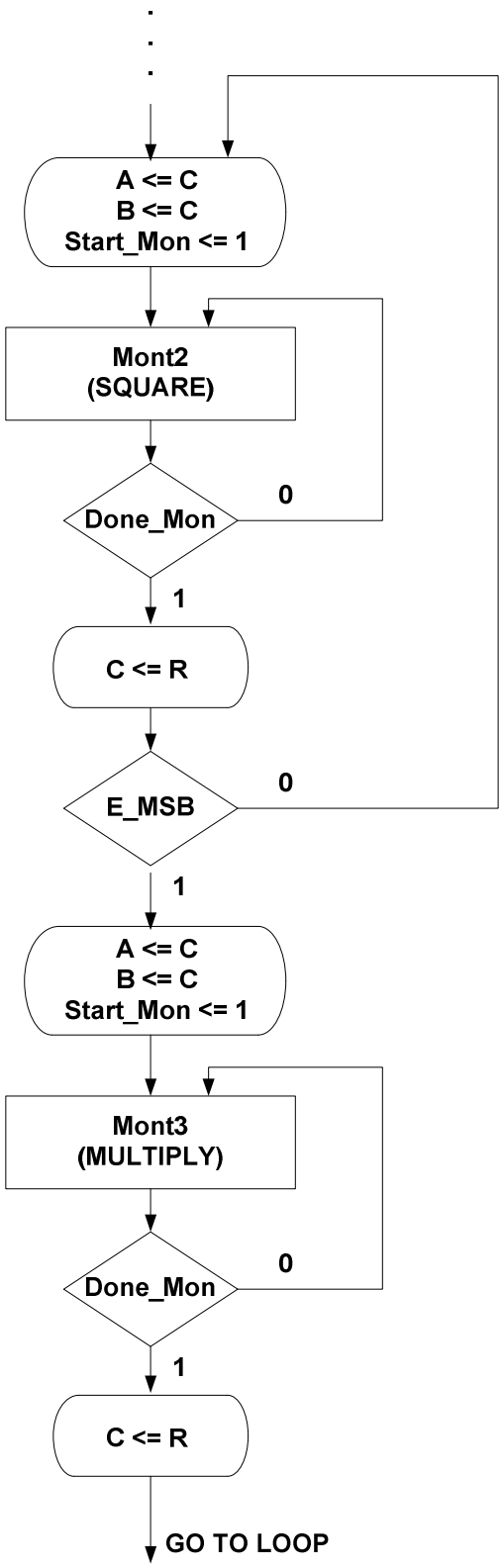

Figure 6.2 : State diagram of binary modular exponentiation RSA algorithm step 2.

In Figure 6.1, Mont1 is the pre-Montgomery operation and Mont4 is the post-Montgomery operation. In Figure 6.2, Mont2 is the square operation, which corresponds to the operations in line 7 in Algorithm 6.1, and Mont3 is the multiplication operation, which corresponds to the line 8 in Algorithm 6.1.

The final hardware architecture of exponentiator is that of Figure 6.3 augmented with two extra Montgomery Modular Multiplier(MMM) PEs.

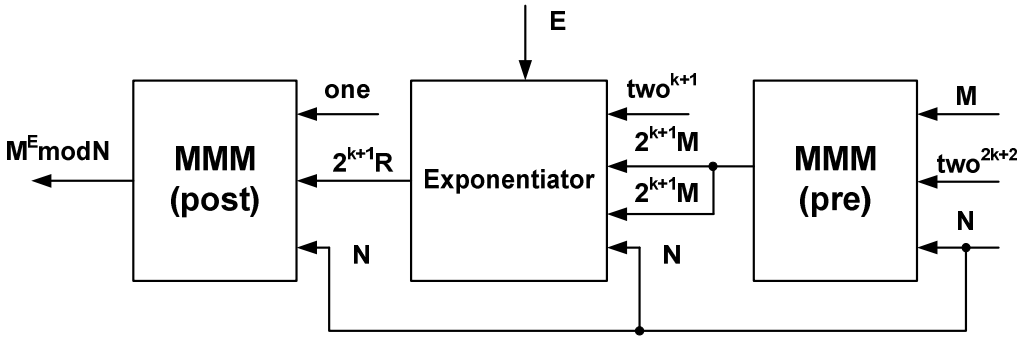

Figure 6.3 : Final architecture of the exponentiator.

In Figure 6.3, M is the message (plain text), two^{2k+2} is the constant and N is the modulus, which are the inputs of MMM-pre block. The output of MMM-pre block is the N-residue transformed form of the message M. The output of the exponentiator is the result of the exponentiation in N-residue domain. So, post-Montgomery multiply the obtained result by "*1*" to get rid of the factor $2^{(k+1)}$ and we get the final result as the output of MMM-post block $M^E \bmod N$.

6.1.2 Verification of RSA implementation

There are two steps of verifying the implementation. First, the plaintext is given to the hardware and obtained the cipher text. The hardware model has been checked by decrypting the encrypted data and comparing the plaintext with the decrypted text. If the decrypted text is equal to the input of the encryption step which is called plaintext, then this means the first step of verification is completed. The illustration of the first step is given in Figure 6.4. As it is seen in Figure 6.4 this step is also partitioned to sub-steps.

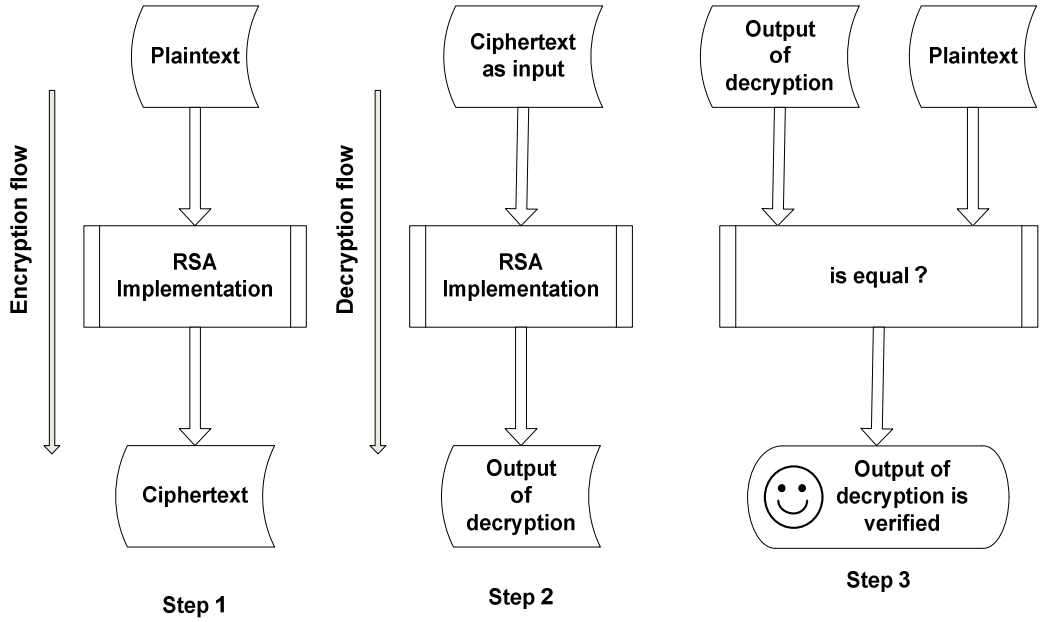

Figure 6.4 : Verification of RSA implementation.

The second step is to verify the experimental results with the calculated, mathematical results. Once it is seen that these results are equal, this means that RSA implementation is working fine. Figure 6.5 illustrates this second step of the verification.

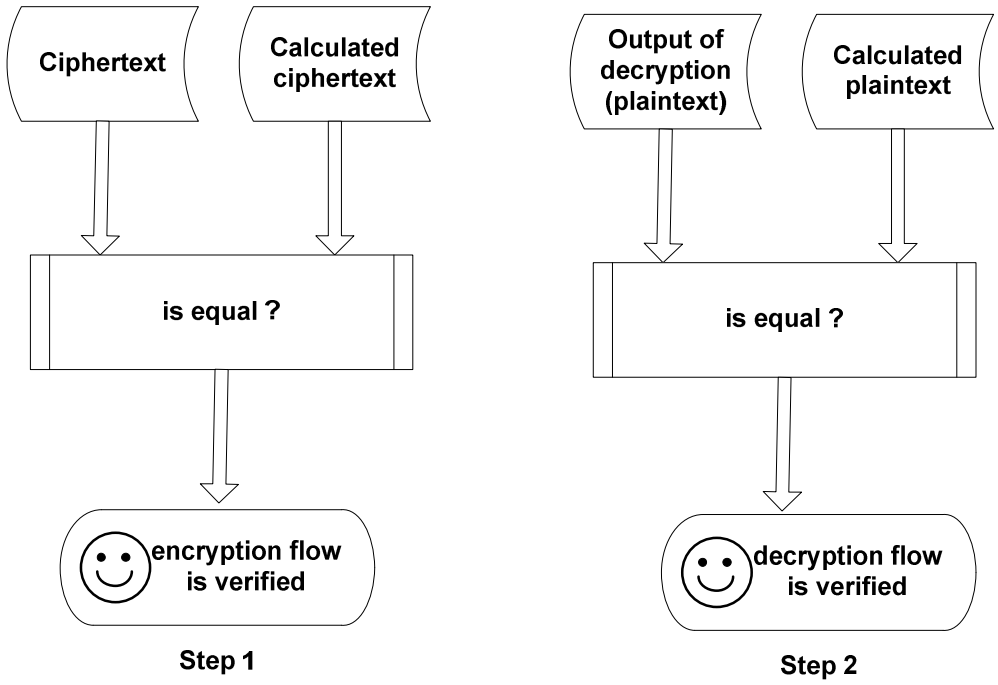

Figure 6.5 : Verification of RSA implementation, second step.

6.1.3 Implementation results and comparison with previous works

The verification method is explained in section 6.2. So it is better to give an example to prove that our implementation provides this verification. We need to follow the following algoritm, this is the same algoritm as described in section 3.1:

Algorithm 6.2: Key generation for RSA public-key encryption

1. Generate two large random (and distinct) primitives p and q, each roughly the same size.
2. Compute $n = pq$ and $\Phi = (p-1)(q-1)$.
3. Select a random integer e, $1 < e < \Phi$, such that $gcd(e, \Phi) = 1$.
4. Use the extended Euclidean algorithm to compute the unique integer d, $1 < d < \Phi$, such that $ed \equiv 1 \pmod{\Phi}$.
5. A's public key is (n,e); A's private key is d.

Example:

1. Lets choose two large random primitives p and q:

p = 7

q = 19

2. Compute N and $\Phi = (p-1)(q-1)$.

N = 7 x 19 = 133

$\Phi = (p-1)(q-1)$ = 6 x 18 = 108

3. Lets select a random small integer co-prime to Φ.

E = 5 (gcd(E, Φ)=1)

4. Find D, such that ed mod $\Phi \equiv 1$

D = 65

5. A's public key is (n,e); A's private key is d.

Public Key	**Secret Key**
N = 133 E = 5	N = 133 E = 65

Encryption:

For this example, lets use the message "6".

M = 6

$$\begin{aligned} C &= M^e \bmod N \\ &= 6^5 \bmod 133 \\ &= 7776 \bmod 133 \\ &= 62 \end{aligned}$$

Let's see the simulation results for this encryption example:

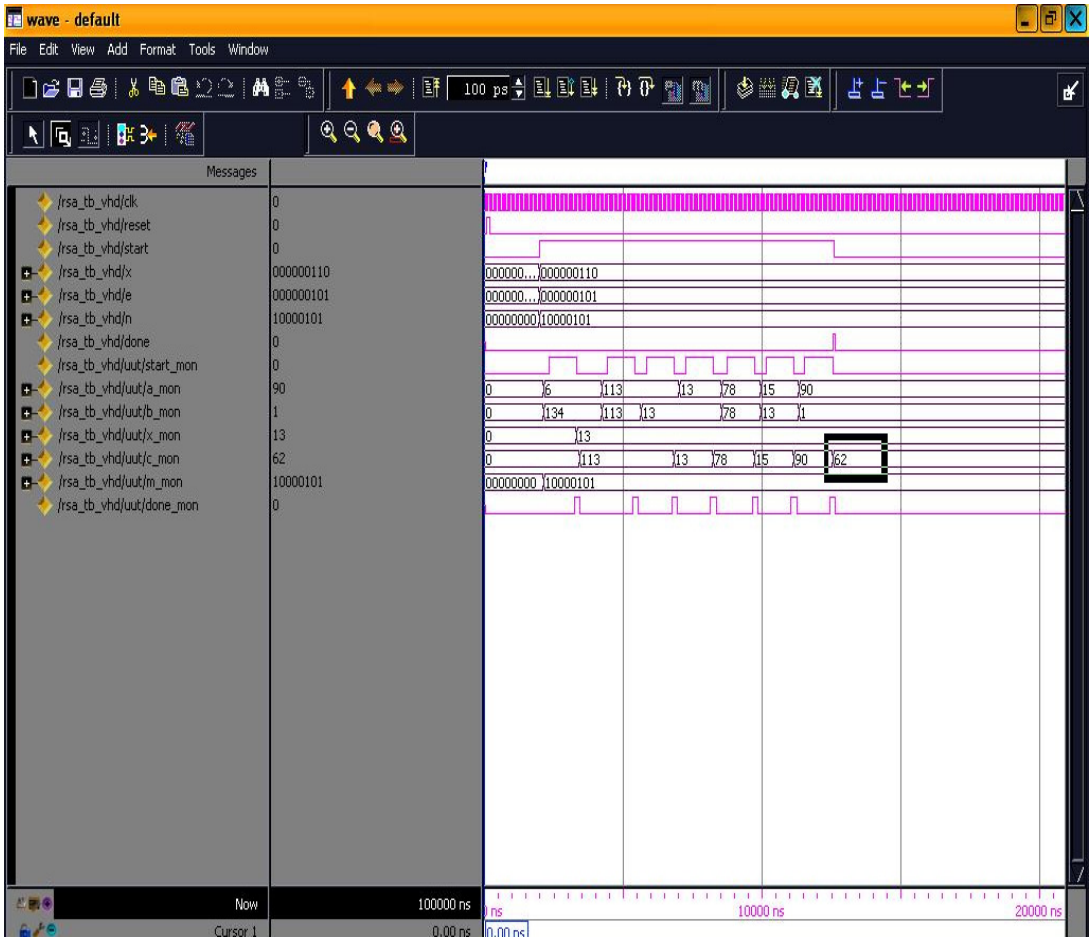

Figure 6.6: Encryption result of RSA implementation.

Decryption:

This works very much like encryption, but involves a larger exponentiation, which is broken down into several steps.

$$
\begin{aligned}
M &= C^d \bmod N \\
&= 62^{65} \bmod 133 \\
&= 62 * 62^{64} \bmod 133 \\
&= 62 * (62^2)^{32} \bmod 133 \\
&= 62 * 3844^{32} \bmod 133 \\
&= 62 * (3844 \bmod 133)^{32} \bmod 133 \\
&= 62 * 120^{32} \bmod 133
\end{aligned}
$$

We now repeat the sequence of operations that reduced 62^{65} to 120^{32} to reduce the exponent down to 1.

$= 62 * 36^{16} \bmod 133$
$= 62 * 99^{8} \bmod 133$
$= 62 * 92^{4} \bmod 133$
$= 62 * 85^{2} \bmod 133$
$= 62 * 43 \bmod 133$
$= 2666 \bmod 133$
$= 6$

Let's see the simulation results for this decryption example:

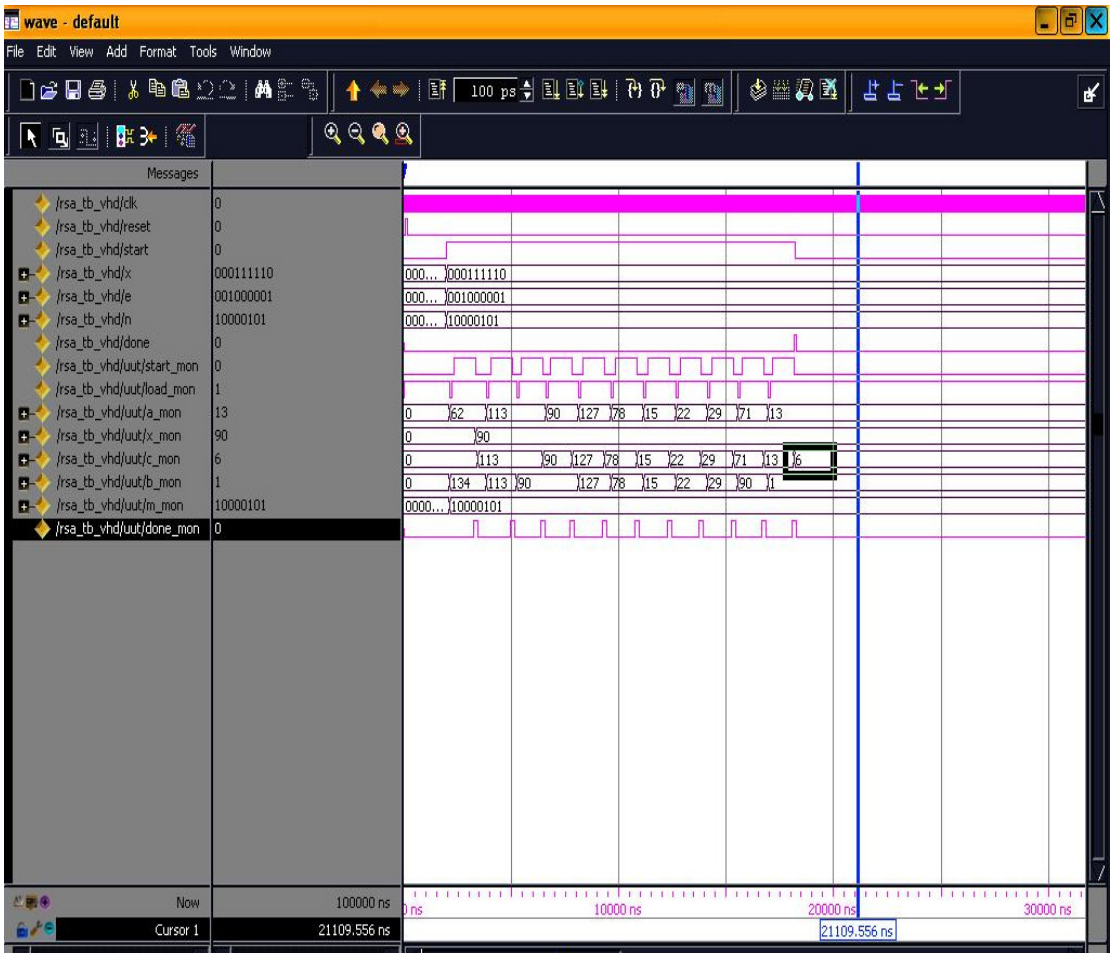

Figure 6.7: Decryption result of RSA implementation.

The data are shown in table 6.1 compared with similar design. This means we compare our implementation results with similar designs, which use binary modular exponentiation. It is better to remind that we have used parallel Montgomery structure in this implementation.

Table 6.1: Performance comparison of binary modular exponentiation

Work	k	Platform	Area (Slice)	Total Dynamic Power (mW)	Freq. (MHz)	Baud Rate (Mbps)	Total Time
Our work	128	Virtex-5 XC5VLX50	151	32	304.913	1.482	86.318 us
Our work	256	Virtex-5 XC5VLX50	246	45	303.377	0.768	332.97 us
Our work	512	Virtex-5 XC5VLX50	427	76	300.481	0.390	1.31 ms
Our work	1024	Virtex-5 XC5VLX50	1117	156	300.481	0.195	5.24 ms
[31]	1024	Virtex-5	--	--	142	0.123	--
[26]	1024	Xilinx XC4000	--	--	52	0.025	--
[32]	1024	Virtex V1000FG680-6	--	--	49.63	0.045	--
[18]	1024	XC2V6000	11520	--	95.90	4.79	0.21 ms
[28]	1024	XC2V3000	12284	--	90.415	--	5.58 us
[29]	1024	Virtex-II-6	5158	--	254.55	--	5.05 us

As it is discussed in section 5.4, we decided that systolic architecture could be a good trade-off in between, as it gives reasonable power consumption for a fair operation speed. And as we know from mathematical background in section 3.2, Sliding Window Method has less multiplication operation for RSA modular exponentiation. So as a result, we concluded that an efficient RSA implementation would be using Sliding Window Exponentiation Method with systolic Montgomery structure. In section 6.2 we have implemented such design and have concluded our work with the implementation results and comparison with previous works.

6.2 The Sliding Window Techniques

6.2.1 Background on sliding window techniques and hardware architecture

The *m*-ary method decomposes the bits of the exponent into d-bit words. The probability of a word of length d being zero is equal to 2^{-d}, assuming that the 0 and 1

bits are produced with equal probability. In *m*-ary method, we skip a multiplication whenever the current word is equal to zero. Thus, as d grows larger, the probability that we have to perform a multiplication operation becomes larger. However, the total number of multiplications increases as *d* decreases. The sliding window algorithms provide a compromise by allowing zero and nonzero words of variable length; this strategy aims to increase the average number of zero words, while using relatively large values of *d* [10].

A sliding window exponentiation algorithm first decomposes *E* into zero and nonzero windows *Fi* of length *L*(*Fi*). The number of windows *p* may not be equal to *k/d*. We take *d* to be the length of the longest window, i.e., *d* = max (*L*(*Fi*)) for *i* = 0; 1; : : : ; *k* - 1. Furthermore, if *Fi* is a nonzero window, then the least significant bit of *Fi* must be equal to 1. This is because we partition the exponent starting from the least significant bit, and there is no point in starting a nonzero window with a zero bit. Consequently, the number of preprocessing multiplications in step 1 of Algorithm 6.3 are nearly halved, since M^w are computed for odd *w* only [10].

Algorithm 6.3: The sliding window method

Inputs: $N = (n_{k-1} \ldots n_1\ n_0)_2$, $E = (e_{k-1} \ldots e_1\ e_0)_2$, $M = (m_{k-1} \ldots m_1\ m_0)_2$
Output: $C = M^E \bmod N$

1. Compute and store $M^w (\bmod n)$ for all $w = 3, 5, 7, \ldots, 2^d - 1$.
2. Decompose *E* into zero and nonzero windows F_i of length $L(F_i)$ for $i = 0, 1, 2, \ldots, p\text{-}1$
3. $C := M^{Fk-1} \bmod n$
4. for $i = p\text{-}2$ down to 0 do
5. $\quad C := C^{power} \bmod n$, $power = 2^{L(Fi)}$
6. $\quad$ if $Fi = 1$ then $C := C \times M^{Fi} \bmod n$
7. return *C;*

Two sliding window partitioning strategies have been explained in section 3.2. These methods differ in whether the length of a nonzero window must be a constant (=*d*) or can be variable (however, <=*d*). In the following section, we give algorithmic descriptions of these two partitioning strategies.

6.2.1.1 Constant Length Nonzero Windows

The constant length nonzero window (CLNW) partitioning algorithm scans the bits of the exponent from the least significant to the most significant. At any step, the algorithm is either forming a zero window (ZW) or a nonzero window (NW).

The algoritm is explained in Algorithm 6.4 as follows:

Algorithm 6.4: Constant length nonzero window

Inputs: $E = (e_{k-1} \ldots e_1 e_0)_2$, d (constant integer)

ZW: Check the incoming single bit: if it is a 0 then stay in ZW; else go to NW.

NW: Stay in NW until all d bits are collected. Then check the incoming single bit: if it is a 0 then go to ZW; else go to NW.

Notice that while in NW, we distinguish between staying in NW and going to NW. The former means that we continue to form the same nonzero window, while the latter implies the beginning of a new nonzero window. The CLNW partitioning strategy produces zero windows of arbitrary length, and nonzero windows of length d.

6.2.1.2 Variable length nonzero windows

The CLNW partitioning strategy starts a nonzero window when a 1 is encountered. Although the incoming d-1 bits may all be zero, the algorithm continues to append them to the current nonzero window. The strategy that allows a variable length for non-zero partitions proceeds as described in the Algorithm 6.5. The VLNW partitioning strategy has two integer parameters:

d : maximum nonzero window length,

q : minimum number of zeros required to switch to ZW.

Algorithm 6.5: Variable length nonzero window

Inputs: $E = (e_{k-1} \ldots e_1 e_0)_2$, d (constant integer), q (constant integer)

ZP: check the incoming sigle bit;

if it is 0 **then** stay in ZP, **else** go to NP;

NP: Check the incoming q bits:

if these are all zero **then** go to ZP, **else** stay in NP;

Let $d = lq + r + 1$, where $1 < r <= q$;

Stay in NP **until** $lq + 1$ bits are received;

At the last step, the number of incoming bits will be equal to r.

if these r bits are all zero **then** go to ZP, **else** stay in NP.

After d bits collected, check the single incoming bit;

if it is zero **then** go to ZP, **else** go to NP.

The VLNW partitioning produces nonzero windows which start with a 1 and end with a 1. Two nonzero windows may be adjacent; however, the one in the least signi_cant position will necessarily have d bits. Two zero windows will not be adjacent since they will be concatenated. For example, let $d = 5$ and $q = 2$, then 5 = 1+1x2+2, thus l = 1 and r = 2.

The following illustrates the partitioning of a long exponent according to these parameters:

<u>101</u> 0 <u>11101</u> 00 <u>101</u> <u>10111</u> 000000 <u>1</u> 00 <u>111</u> 000 <u>1011</u>

In our implementation we have used Variable Length Nonzero Window method for partitioning process of E exponent. The architecture of this design will be explained later. However, Figure 6.3, 6.4 and 6.5 shows the state diagram of sliding window method for modular exponentiation.

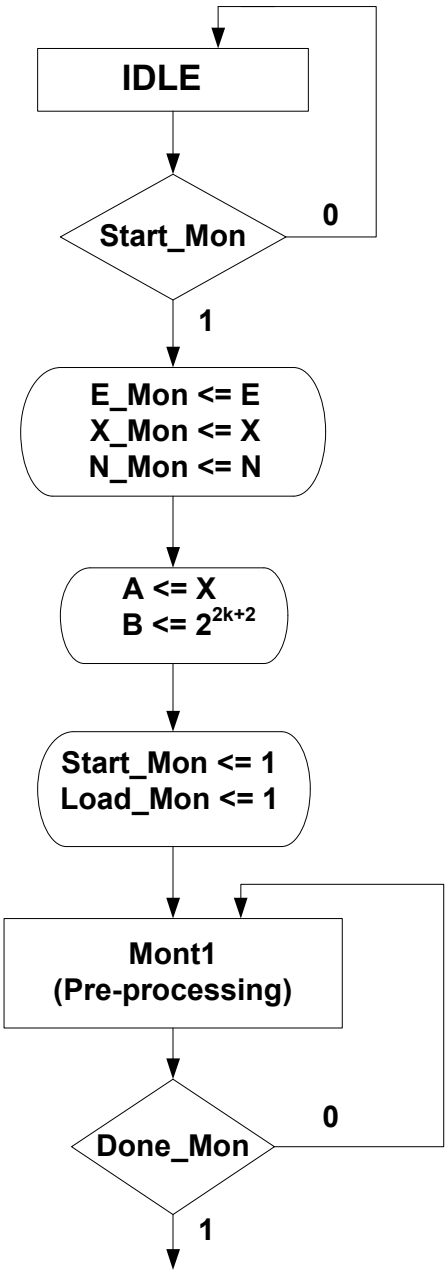

Figure 6.8 : State diagram of sliding window method RSA algorithm step 1.

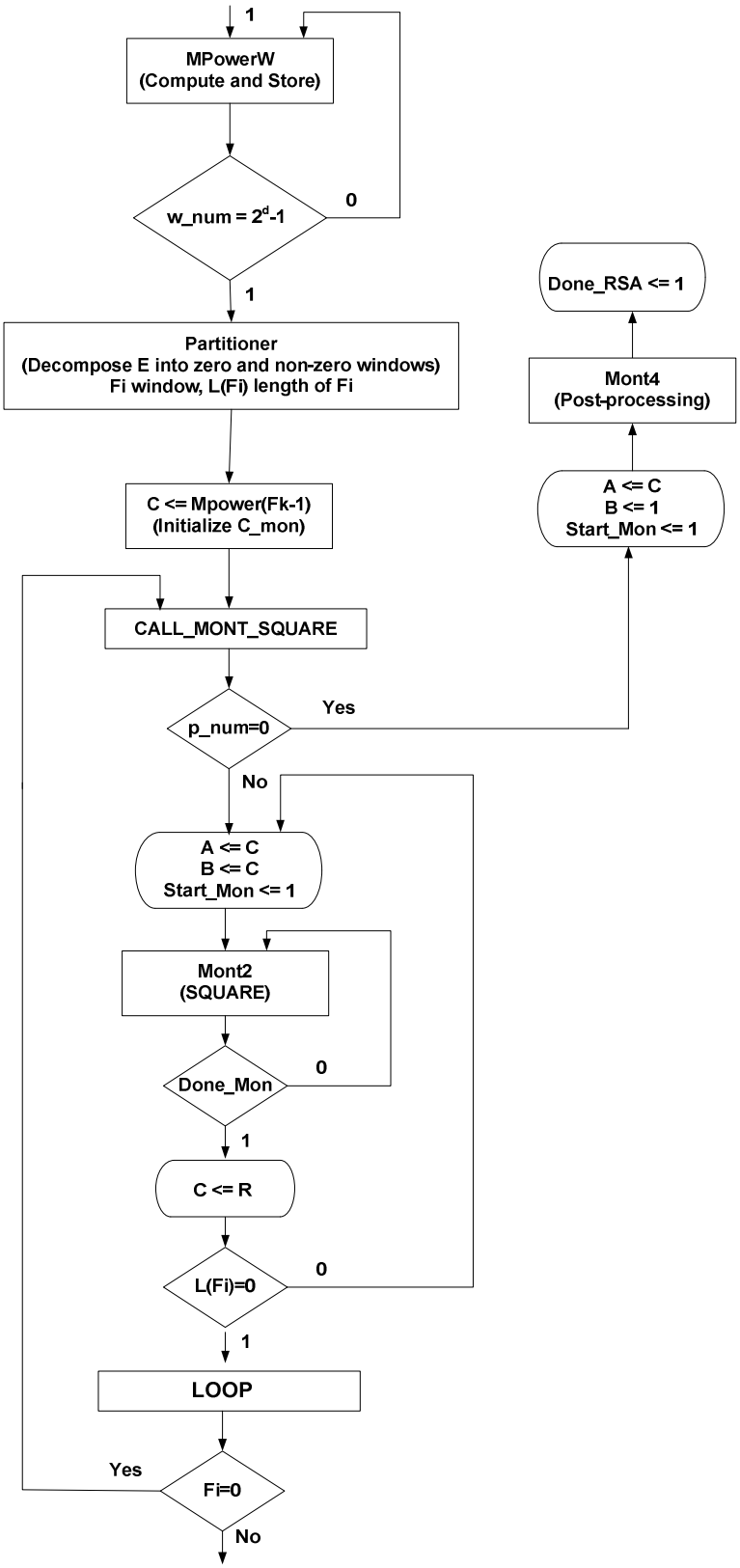

Figure 6.9 : State diagram of sliding window method RSA algorithm step 2.

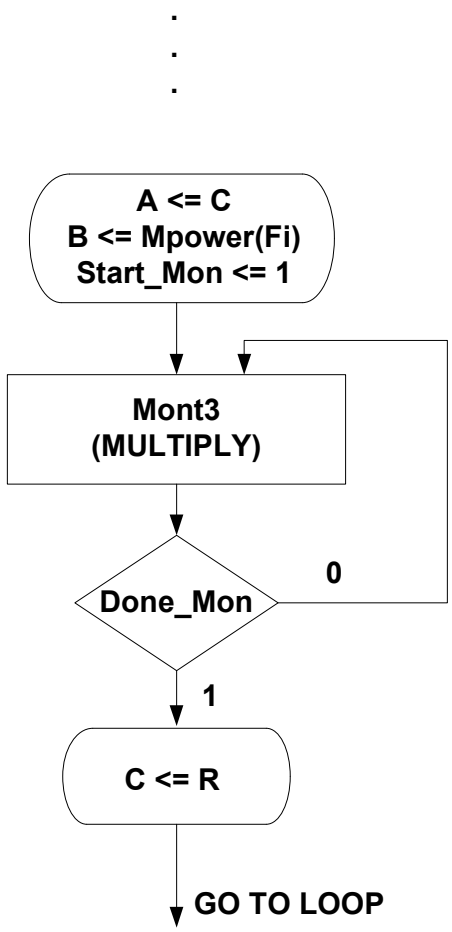

Figure 6.10 : State diagram of sliding window method RSA algorithm step 3.

The architecture of the hardware for the modular exponentiator is depicted in Fig. 6.11. It uses *modular multiplier* that implements the modular multiplication using Montgomery's algorithm. The exponentiator uses *partitioner* that takes care of the partitioning process as described in Algorithm 6.5. The output of this process is available in a memory that is used by exponentiator. Besides, the exponentiator computes all the possible modular powers of M for odd exponents, considering the maximum length of a non-zero partition d. As explained in section 6.1, we need to pre-Montgomery multiply the operands by $2^{2(k+1)}$ *mod* N and post-Montgomery multiply the obtained result by "*1*" to get rid of the factor $2^{(k+1)}$ that is carried by every partial result.

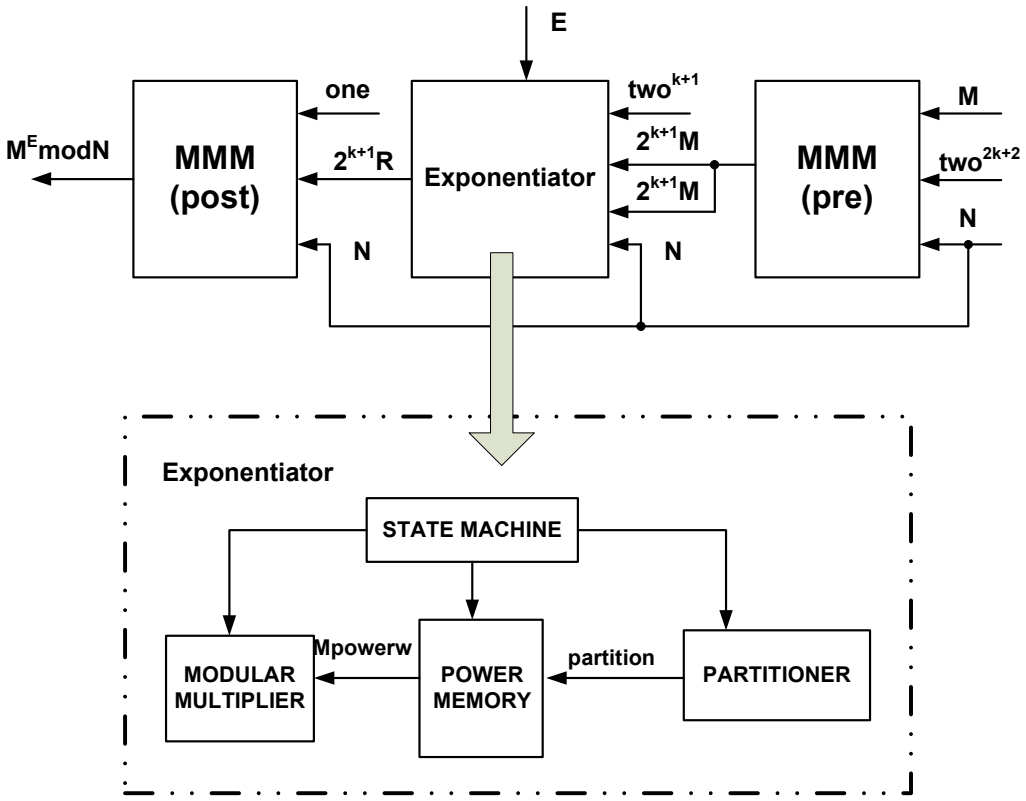

Figure 6.11 : Architecture of sliding window based exponentiator.

6.2.2 Implementation results and comparison with previous works

As it is indicated in section 6.1.2, the verification of RSA Sliding Window Algorithm is provided by getting encryption and decryption results. The data in table 6.2 shows the comparison of our implementation on RSA Sliding Window Algorithm and similar designs.

Table 6.2: Performance comparison of RSA sliding window method

Work	k	d	Platform	Area (Slice)	Total Dynamic Power (mW)	Baud Rate (Mbps)	Total Time
Our work	32	5	Virtex-5 XC5VLX50	665	24	1.345	23.791 us
Our work	64	5	Virtex-5 XC5VLX50	4250	28	0.777	82.35 us
[33]	64	3	FPGA Spartan Family	567 (in CLB)	--	--	12.3 ns
[33]	64	4	FPGA Spartan Family	678 (in CLB)	--	--	10.4 ns
[33]	128	3	FPGA Spartan Family	899 (in CLB)	--	--	13.7 ns
[33]	128	4	FPGA Spartan Family	992 (in CLB)	--	--	10.1 ns

7. CONCLUSION

In this study, dynamic power consumptions of Field Programmable Gate Array (FPGA) implementations of the Rivest, Shamir, Adleman (RSA) has been reduced by using low power design methods.

Modular Multiplication block dissipates the most of the power, dissipated in RSA. For comparison of power dissipations, different methods are used to implement Modular Multiplication block. Then RSA implemented by using Sequential Binary Modular Exponentiation which has widespread applications. Computer simulations have been used to show that the implementations of the algorithm generate correct outputs against test vectors.

Low power design techniques are examined and power consumption of implemented architectures of RSA algorithm is reduced by using these techniques. Because of much power dissipation, implemented Modular Multiplication block by using different methods are improved so that their power dissipation is reduced. In this study, three types of architecture are implemented for Modular Multiplication operation: Parallel Architecture of Modular Montgomery Multiplier, Sequential Architecture of Modular Montgomery Multiplier and Systolic Architecture of Modular Montgomery Multiplier. These low power techniques on Modular Multiplication architectures are compared according to the power dissipations and area requirements. We concluded that, the total dynamic power of our serial design is the best choice for such requirements. Our second conclusion is that in terms of energy, parallel design gives the best result. Lastly, we conclude that systolic architectures could be a good trade-off in between, as it gives reasonable power consumption for a fair operation speed

First we have implemented RSA binary algorithm by using parallel Montgomery Multipler Architecture, since this binary algorithm is the basic RSA algorithm and it is simple. However, we know that RSA Sliding Window Method is more efficient since it reduces total multiplication operations. As a result, we concluded that an efficient RSA implementation would be Sliding Window Exponentiation Method

with systolic Montgomery Architecture. So we have implemented such design and have concluded our work with the implementation results.

REFERENCES

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

[2] **Yeşil, S., İsmailoğlu, A. N., Tekmen, Y. C., Aşkar, M.,** 2004. Two Fast RSA Implementations Using High-Redix Montgomery Algorithm, *IEEE International Symposium on Circuits And Systems*, Vancouver, Canada.

[3] **Güdü, T.,** 2007. A New Scalable Hardware Architecture for RSA Algorithm, *IEEE Field Programmable Logic and Applications*, Aug, pp. 670-674.

[4] **Kwon, T., You, C., Heo, W., Kang, Y., Choi, J.,** 2001. Two implementation methods of a 1024-bit RSA cryptoprocessor based on modified Montgomery algorithm, *IEEE International Symposium on Circuits and Systems (ISCAS 2001),* May, **4**, pp. 650-653.

[5] **Diffie, W. and Hellman, M.E.,** 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory,* Vol. 22, pp. 644-654.

[6] **Stinson, D.R., 2002.** Cryptography Theory and Practice, Chapman & Hall/CRC, Waterloo, Ontario.

[7] **Menezes, A.J., Van Oorschot, P.C., and Vanstone, S.A.,** 1996. Handbook of Applied Cryptography, CRC Press.

[8] **Guo, J., Wang C., Hu, H.,** 1999. Design and implementation of an RSA public-key cryptosystem, *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems,* June, **1**, pp. 504-507.

[9] **Fournaris, A.P., Koufopavlou, O.** 2005. A new RSA encryption architecture and hardware implementation based on optimized Montgomery multiplication, *IEEE International Symposium on Circuits and Systems (ISCAS 2005),* Kobe, Japan, May, **5**, pp. 4645-4648.

[10] **Koç, Ç.K.,** 1994. High-Speed RSA Implementation, *RSA Laboratories Technical Report*, Redwood City, California, USA.

[11] **Montgomery, P.L.,** 1985. Modular Multiplication without Trial Division, *Mathematics of Computation*, **44**, pp. 519-521.

[12] **Knuth, D.E.,** 1981. The Art of Computer Programming: Seminumerical Algorithms, Addison-Wesley, Reading.

[13] **Bayam, K. A., Örs S. B.,** 2010.Differential Power Analysis Resistant Hardware Implementation of the RSA Cryptosystem, *The Turkish Journal of Electrical Engineering & Computer Sciences,* **18**, No:1.

[14] **Bos, J. and Coster, M.** 1989. Addition Chain Heuristics, *Advances Cryptology - CRYPTO 89,* Lecture Notes in Computer Science, Santa Barbara,

California, USA, 435, pp. 400-407, Ed. Brassard, G., Springer-Verlag.

[15] **Park, H., Park, K., Cho Y.,** 1999. Analysis of the variable length nonzero window method for exponentiations, *Science Direct*, *Computers & Mathematics with Applications,* **37**, Iss. 7, pp. 21.

[16] **Batina, L., Örs, S.B., Preneel, B., and Vandewalle, J.,** 2003. Hardware Architectures for Public Key Cryptography. *The VLSI Journal Integration*, **34**, pp. 1-64, Elsevier Science Publishers B. V.

[17] **A.P. Chandrakasan, S.S. and Brodersen, R.,** 1992. Low-power CMOS Digital design, *IEEE Journal of Solid-State Circuits*, Apr., **27(4)**, pp. 473-484.

[18] **McIvor, C., McLoone, M., and McCanny, J.V.,** 2004. Modified Montgomery modular multiplication and RSA exponentiation techniques*, Proceedings of Computers and Digital Techniques,* **151**, pp. 402-408.

[19] **Walter, C.D.,** 1999. Montgomery Exponentiation Needs No Final Subtraction, *Electronic Letters*, **35**, pp. 1831-1832.

[20] **Kaps, J.P.,** 2006. Cryptography for Ultra-Low Power Devices, *Ph.D. thesis*, Worcester Polytechnic Institue.

[21] **Ghosh, A., Devadas, S., Keutzer, K. and White, J.,** 1992. Estimation of average switching activity in combinational and sequential circuits, *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pp. 253-259.

[22] **Doğan, A.Y.,** 2008. AES Algoritmasının FPGA Üzerinde Düşük Güçlü Tasarımı, *M.Sc. Thesis*, Istanbul Technical University, Istanbul.

[23] **N. Nedjah, L.M. Mourelle,** 2006. Three Hardware Architectures for the Binary Modular Exponentiation: Sequential, Parallel, and Systolic, *IEEE Transactions on Circuits and Systems*, March, **53**, No:3.

[24] **Rabaey, J.,** 1995. Digital Integrated Circuits: A Design Perspective, Englewood Cliffs, NJ:Prentice-Hall.

[25] **P. Guilherme, D.G. Mesquita, F.L Herrmann., J.B. Martins.,** 2010. Montgomery Modular Multiplication on Reconfigurable Hardware Fully Systolic Array vs Parallel Implementation, in Proceedings of VI Southern Programmable Logic Conference, 2010, Recife, p. 61-66.

[26] **T. Blum, and C. Paar,** Montgomery modular exponentiation on reconfigurable hardware, in Proc. 14th IEEE Symp. On Computer Arithmetic, 1999, pp. 70 - 77.

[27] **A. Tenca and C.K. Koc,** A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm, IEEE Transactions on Computers, 2003.

[28] **R.V. Kamala and M.B. Srinivas,** "High-Throughput Montgomery Multiplication", IFIP International Conference on Very Large Scale Integration (IEEE/ACM SIGDA Conference), VLSI SOC-2006, Nice, France, 16th Oct - 18th Oct 2006.

[29] **M.D. Shieh, W.C. Lin,** 2010. Word-Based Montgomery Modular Word-Based Montgomery Modular Multiplication Algorithm for Low-Latency Scalable Architectures, IEEE Transactions on Computers, **59**, pp. 1151-1151.

[30] **K.A. Bayam,** 2007. Differential Power Analysis Resistant Hardware Implementation of the RSA Cryptosytem, *M.Sc. Thesis*, Istanbul Technical University, Istanbul.

[31] **X. Hong, H. Wenhao, Y. Jiangyu,** 2009. Design and Implementation of High-performance Modular Exponentiation Arithmetic Unit, Information Science and Engineering (ICISE), p.1694-1694.

[32] **A. Daly, W. Marnane**, 2002. Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. Proceedings of Tenth ACM International Symposium on Field Programmable Gate Arrays (FPGA'02), pp. 44 - 49.

[33] **N. Nedjah, L.M. Mourelle,** 2008. Efficient Hardware for Modular Exponentiation Using the Sliding Window Method. Proceedings of the 9th International Conference for Young Coumputer Scientiests, pp. 1985-1985.

CURRICULUM VITA

Candidate's full name:	Dilek BAYHAN GÜMÜŞ
Place and date of birth:	Istanbul, 13.07.1983
Permanent Address:	Prof. Hıfzı Özcan cd., Gül sk., Alize Ap., No:5, Daire:7, Ataşehir / ISTANBUL
Universities and Colleges attended:	2001-2005: BSc. Program in Electronics Engineering, Istanbul Technical University
	2002-2006: BSc. Program in Computer Engineering, Istanbul Technical University