FAST, COMPACT AND SECURE IMPLEMENTATION OF RSA ON DEDICATED HARDWARE

by

ERSIN ÖKSÜZOĞLU

Submitted to the Graduate School of Engineering and
Natural Sciences in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University

June 2008

FAST, COMPACT AND SECURE IMPLEMENTATION OF

RSA ON DEDICATED HARDWARE

APPROVED BY:

Associate Prof. Dr. Erkay Savaş: .....

(Thesis Advisor)

Associate Prof. Dr. Albert Levi: .....

Assistant Prof. Dr. İlker Hamzaoğlu: .....

Assistant Prof. Dr. Ayhan Bozkurt: .....

Post Doc. Thomas Pedersen: .....

DATE OF APPROVAL: …….

ABSTRACT

RSA is the most popular Public Key Cryptosystem (PKC) and is heavily used today. PKC comes into play, when two parties, who have previously never met, want to create a secure channel between them. The core operation in RSA is modular multiplication, which requires lots of computational power especially when the operands are longer than 1024-bits. Although today's powerful PC's can easily handle one RSA operation in a fraction of a second, small devices such as PDA's, cell phones, smart cards, etc. have limited computational power, thus there is a need for dedicated hardware which is specially designed to meet the demand of this heavy calculation. Additionally, web servers, which thousands of users can access at the same time, need to perform many PKC operations in a very short time and this can create a performance bottleneck. Special algorithms implemented on dedicated hardware can take advantage of true massive parallelism and high utilization of the data path resulting in high efficiency in terms of both power and execution time while keeping the chip cost low. We will use the "Montgomery Modular Multiplication" algorithm in our implementation, which is considered one of the most efficient multiplication schemes, and has many applications in PKC.

In the first part of the thesis, our "2048-bit Radix-4 based Modular Multiplier" design is introduced and compared with the conventional radix-2 modular multipliers of previous works. Our implementation for 2048-bit modular multiplication features up to 82% shorter execution time with 33% increase in the area over the conventional radix-2 designs and can achieve 132 MHz on a Xilinx xc2v6000 FPGA. The proposed multiplier has one of the fastest execution times in terms of *latency* and performs better than (37% better) our reference radix-2 design in terms of time-area product. The results are similar in the ASIC case where we implement our design for UMC 0.18 μm technology.

In the second part, a fast, efficient, and parameterized modular multiplier and a secure exponentiation circuit intended for inexpensive FPGAs are presented. The design utilizes hardwired block multipliers as the main functional unit and Block-RAM as storage unit for the operands. The adopted design methodology allows adjusting the number of multipliers, the radix used in the multipliers, and number of words to meet the system requirements such as available resources, precision and timing constraints.

The deployed method is based on the Montgomery modular multiplication algorithm and the architecture utilizes a pipelining technique that allows concurrent operation of hardwired multipliers. Our design completes 1020-bit and 2040-bit modular multiplications* in *7.62 μs* and *27.0 μs* respectively with approximately the same device usage on Xilinx Spartan-3E 500. The multiplier uses a moderate amount of system resources while achieving the best area-time product in literature. 2040-bit modular exponentiation engine easily fits into Xilinx Spartan-3E 500; moreover the exponentiation circuit withstands known side channel attacks with an insignificant overhead in area and execution time. The upper limit on the operand precision is dictated only by the available Block-RAM to accommodate the operands within the FPGA. This design is also compared to the first one, considering the relative advantages and disadvantages of each circuit.

* With a multiplication engine that utilizes half of the device (i.e. which use 10 multipliers)

ÖZET

RSA, günümüzde en sık kullanılan Açık Anahtarlı Şifreleme (AAŞ) türüdür. Daha önce hiç karşılaşmamış iki tarafın birbirleri arasında güvenli bir iletişim kanalı oluşturabilmesi için AAŞ sistemleri kullanılır. RSA'de en temel işlem modüler çarpım işlemidir ve özellikle kullanılan anahtar 1024 bitten uzunsa, bu işlem çok yoğun bir hesaplama gücü gerektirir. Günümüzün kişisel bilgisayarları birkaç RSA işlemini bir saniyeden kısa bir zamanda bitirebilirken, avuçiçi bilgisayarları, cep telefonları ve smart kart gibi az işlem gücüne sahip ortamlarda, bu yüksek hesap gücü gerektiren işlem için kullanılacak ilave donanıma ihtiyaç vardır. Buna ek olarak binlerce kişinin aynı anda erişim isteyebileceği web servis sağlayıcılarında, bu işlem, bir performans darboğazı olarak görünebilir. Donanım için geliştirilen bazı özel algoritmalar sayesinde çok büyük ölçekte paralel hesaplamalar yapılabilir ve böylece donanımın kullanım oranı artırılarak hem enerji harcaması düşürülür, hem de toplam işlem zamanı kısaltılır. Bu amaçla, en verimli modüler çarpım işlemlerinden biri olarak bilinen ve birçok AAŞ alanında kullanılan "Montgomery Modüler Çarpım" algoritmasını kullanacağız.

İlk olarak "2048-bitlik ve 4 tabanında çalışan Modüler Çarpım" dizaynını anlatacağız ve bunu daha önceki çalışmalarda sıkça kullanılan 2 tabanındaki modüler çarpım devreleriyle karşılaştıracağız. Bizim devrenin, diğer devreleri simule etmek için yaptığımız referans devreye göre çalışma hızının % 82 arttığını ve bunu sadece %33'lük bir alan artışıyla gerçekleştirdiğini gördük. Ayrıca, Xilinx xc2v6000 FPGA'inde 132 MHz'de çalışan bu devre, referans dizayna göre %37'lere varan oranlarda, zaman alan çarpımını azalttı. Benzer kazanımları, UMC 0,18 µm teknolojisi için sentezlenen devre ile de elde ettik.

İkinci bölümde ise nispeten ucuz FPGA'lere uygun, hızlı, parametrik ve yan kanal ataklarına karşı dayanıklı bir modüler çarpım devresini ve bir üs alma devresini sunuyoruz. Bu dizayn, FPGA üzerinde bulunan çarpım birimlerini ve blok RAM'i kullanacak şekilde geliştirildi. Dizaynımızda çarpım işlemi için kullanılan bileşenlerin tabanı (radixi), çarpım ünitelerinin sayısı ve toplam *word* sayısı parametrik olarak istenen özelliklere göre ayarlanabilir. Mimari yapıda *pipelining* tekniğini kullandık ve bu bize yüksek frekanslarda, aynı anda birçok çarpım işlemini yapma özelliği kazandırdı. Dizaynımız 1020-bitlik ve 2040-bitlik modüler çarpım işlemlerini Xilinx Spartan-3E 500 FPGA'i üzerinde sırasıyla *7,62 µs* ve *27,0 µs*'de bitirmektedir ve bu

ölçümler FPGA'de bulunan çarpım birimlerinin sadece yarısı kullanılarak elde edilmiştir. Dizanımız daha önceki devrelerle karşılaştırıldığında en düşük alan zaman çarpımını elde etti. Ayrıca 2040-bitlik üs alma devresinin Xilinx Spartan-3E 500 çipine kolaylıkla sığabileceğini gördük. Kullandığımız üs alma devresi, bilinen tüm yan kanal ataklarına karşı korumalı bir şekilde dizayn edildi ve bu koruma çok ufak bir ek donanım getirilerek başarıldı. Üs alma devresi, işlemde kullanılan sayılar blok RAM'e sığdığı sürece her büyüklükteki sayı için kullanılabilir. Bu dizanımız ayrıca ilk dizaynla da avantaj ve dezavantajları açısından karşılaştırıldı.

Dedicated to my family…

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Associate Prof. Dr. Erkay Savaş, for his never-ending support and patience. His trust on my success encouraged me through the hard times during this project, helped me finish my thesis in time. We have spent long hours together to solve some problems, when I almost gave up hope. His energy has given me enough power to handle the difficulties I encountered in my master years. I really appreciate his advices and guidance concerning this thesis.

Throughout my master years, I am proud to be supported by TUBITAK, so that I can have more time to work on my project. The financial support provided by TUBITAK helped me very much.

Finally and most importantly, I want to thank my family, who always supported and encouraged me from the very beginning. With their high motivation and positive attitude to my academic life, I have succeeded in finishing my Master of Science degree on such a great university.

Table of Contents

List of Terms and Symbols xi

List of Figures xiii

List of Tables xiv

1 Background Information 1

1.1 Asymmetrical Cryptography 1

1.1.1 Euler's Totient Function 2

1.1.2 Fermat's Little Theorem 2

1.1.3 RSA Algorithm Basics 3

1.1.4 The Visualization of RSA 3

1.1.5 RSA Setup 4

1.1.6 An RSA Example 5

1.1.7 Modular Exponentiation 5

1.1.8 Modular Multiplication 6

1.2 Side Channel Attacks and Countermeasures 7

1.2.1 Simple Power Analysis Attack (SPA) 8

1.2.2 Timing Attacks 8

1.2.3 Fault Injection Attacks 8

1.2.4 Differential Power Analysis Attack (DPA) 9

1.3 Dedicated Hardware Basics 11

1.3.1 ASICs 11

1.3.2 FPGA 12

2 Radix-4 Implementation of 2048 bit Modular Multiplication on ASIC 14

2.1 Algorithm 14

2.1.1 Booth Recoding 14

2.2 Architecture 16

2.3 Theoretical Analysis of Performance 20
2.4 Synthesis 21
2.5 Conclusion 24
3 Parametric, Secure and Compact Implementation of RSA on FGPA 25
3.1 CIOS Method 26
3.2 The Multiplication Engine 27
3.2.1 Design Criteria 28
3.2.2 Implementation Details 29
3.2.3 Parametric Design 30
3.3 Simulation Results 31
3.4 Synthesis Results 32
3.4.1 Setup and Synthesis Configuration 32
3.4.2 Synthesis Results 33
3.5 Performance Analysis 34
3.6 Compatibility Problems 38
3.7 Conclusion and Future Work 39
4 Summary of Contributions 41
Appendix 43
References 44

List of Terms and Symbols

- **µs:** micro seconds: 10^{-6} seconds.
- **ASIC:** Application Specific Integrated Circuit: Type of hardware that is manufactured to realize specific calculations (It cannot be reprogrammed).
- **Block RAM:** The type of storage in an FPGA, that has write, read and address ports and write enable input. Only one "data word" can be requested and/or can be written in one clock cycle.
- **C:** Ciphertext: Encrypted message.
- **CIOS:** Coarsely Integrated Operand Scanning: A multi-precision multiplication algorithm
- **CPA:** Carry Propagate Adder.
- **CRA:** Carry Ripple Adder
- **CRT:** Chinese Remainder Theorem: A technique that can be used to speed up modular exponentiation.
- **CSA:** Carry Save Adder.
- **d:** Private key.
- **DPA:** Differential Power Analysis.
- **DSP48:** Data path element present in advanced FPGAs, that is capable of fast multiplication and addition. (Digital Signal Processor)
- **DSS:** Digital Signature Standard
- **e :** Public key.
- **ECC**: Elliptic Curve Cryptography
- **FPGA:** Field Programmable Gate Array: Type of hardware that has some memory and LUT's that can be re-programmed by a computer.
- **Gate:** Basic building blocks of ASIC's.
- **LUT:** Look up Tables: Logic functions in a circuit are mapped to FPGA's SRAM based programmable storage units.
- **m:** Plaintext: Numerical representation of the message to be encrypted.

- **MM:** Montgomery Multiplier.
- **ms:** mili-seconds: 10^{-3} seconds.
- **n:** ($p.q$) Modulus for multiplications in RSA *or* bit length of the modulus
- **p , q:** Large primes (512 bit or longer)
- **PE:** Processing Element (The unit structure which the circuit is built upon) *or* total number of Processing Elements
- **PKC:** Public Key Cryptosystem.
- **Prime Numbers:** Positive integers that can be divisible by only itself and one without remainders.
- **Radix:** Base of a number.
- **Register:** Storage unit that can read or written synchronously both in ASIC's and FPGA's.
- **RSA:** A PKC algorithm found by Rivest, Shamir and Adleman in 1978.
- **s:** Total number of words (such as two 17-bit words)
- **Slice:** Basic building blocks of an FPGA, which consists of two LUT's, two flip-flops and two carry chains.
- **SPA:** Simple Power Analysis.
- **SRAM:** Static Random Access Memory: Type of storage that is made of four transistors and does not need a refresh signal.
- **w:** Bit length of a single word (e.g. 17-bit words)
- **Φ(n) :** Count of the numbers which are relatively prime to n. (Euler's Totient Function)

List of Figures

Figure 1. Visual Depiction of RSA 4
Figure 2. N-Residue Form 7
Figure 3. A three-bit CSA 17
Figure 4. Carry Propagate Adder 18
Figure 5. Radix-2 Montgomery Multiplier 18
Figure 6. Radix-4 Montgomery Multiplier 19
Figure 7. Execution Graph of The Parallelized CIOS algorithm 29
Figure 8. The Structure of a Processing Element 30
Figure 9. Waveform of One Stream 43

List of Tables

Table 1. Booth Recoding 15
Table 2. Time Complexities for One 2048-bit Modular Multiplication 20
Table 3. Xilinx Synthesis Tool (XST) Synthesis Results 21
Table 4. Overall Speed Comparison for 2048 bit modular multiplication 22
Table 5. The Time Area Products for FPGA designs 23
Table 6. The Time Area Products for ASIC designs 23
Table 7. Clock Cycles Required for One Multiplication (radix=2^{17}) 31
Table 8. Utilization Ratios (%) 32
Table 9. Synthesis Results for Multiplication Core 33
Table 10. Time Area Products 33
Table 11. Synthesis Results for 1020-bit exponentiation circuit 34
Table 12. Time Complexities 35
Table 13. Execution Times for 1024-bit modular multiplication 36
Table 14. Execution Engine Performance: 1024-bit Exponentiation Results 37
Table 15. Time-Area products normalized to proposed implementation 38
Table 16. The Required Clock Cycles for Compatible Versions 39

List of Algorithms

Algorithm 1: Binary Exponentiation (left to right) 5
Algorithm 2: Radix-2 Montgomery Multiplication 6
Algorithm 3: Montgomery Powering Ladder 9
Algorithm 4: Radix-4 Montgomery Multiplication 16
Algorithm 5: CIOS Montgomery Multiplication 27

1 Background Information

We can explain the term “cryptology” as the science of keeping data secret by preventing unauthorized access. Nowadays, as the Internet and the technology are evolving at a fascinating rate, cryptology plays a crucial role in many areas in our lives like accessing our bank account online, registering our courses, using credit cards and so on. It is so integrated that many of us are not even aware.

We can divide the cryptology into two parts: Symmetrical and asymmetrical. Former has only one key, which is used in both encryption and decryption. Its main application is ciphering large volumes of data, whereas the latter has two distinct keys, (one key for encryption, the other for decryption) and is used for key exchange and digital signature. Asymmetric Key Cryptosystems are also called Public Key Cryptosystems (PKC).

The functions used in the asymmetrical cryptology depend on the “Number Theory”, which is the branch of pure mathematics related with the properties of numbers in general, and integers in particular, and also other problems that occur from their study. The mathematical background of the algorithms will be explained while giving the details of the important functions.

At first, we will lay out the basic properties of asymmetrical cryptography and its main uses. In the following section, the side channel attacks (the attacks related not to the algorithm but to the implementation) and their countermeasures will be discussed. In the last part of this section, we will compare two dominant hardware platforms.

1.1 Asymmetrical Cryptography

The problem of common key delivery and management among communicating parties is the main use of asymmetrical cryptography (which is also called Public Key Cryptosystem [PKC]). Moreover, PKC’s can be used for digital signature. There are mainly six algorithms (or methods) that are commonly used in PKC:

- RSA [1]
- Diffie-Hellman [12]

- Elliptic Curve Cryptography (ECC) [13]
- El-Gamal [27]
- Digital Signature Standard (DSS) [28]
- Paillier [29]

In all PKC's, there are two keys, one *public key* (known by anybody) and one *private key* (only known by the owner). Public keys are used to encrypt messages and verifying the signatures; on the other hand, private keys are used to decrypt the ciphertext and sign messages.

Private and public keys are related to each other; however one cannot derive the private key by knowing only the public key in the practical computation limits for adequate key lengths. PKCs are similar to one way functions, where anyone can encrypt a message (or verify a signature), but no one can decrypt a message (or sign a message) without the related private key.

1.1.1 Euler's Totient Function

In Number Theory, the Euler's Totient of a positive integer n is defined to be the number of positive integers less than or equal to n that are co-prime to n. The function is calculated as follows:

$$\Phi(n) = n\prod_{p|n}\left(1-\frac{1}{p}\right)$$

For instance:

$$\Phi(45) = 45\times\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right) = 45\times\frac{2}{3}\times\frac{4}{5} = 24$$

1.1.2 Fermat's Little Theorem

Fermat's Little Theorem states that if p is a prime number, then for any integer a, $(a^p - a)$ will be evenly divisible by p.

$a^p \equiv a \pmod{p}$

We work in (mod n) for the base and (mod $\Phi(n)$) for the exponent and so we can generalize Fermat's Little Theorem using Euler's Totient Function:

$a^{\Phi(n)} \equiv 1 \pmod{n}$ (where n and a is relatively prime)

1.1.3 RSA Algorithm Basics

Rivest, Shamir and Adleman implemented a novel scheme for key exchange in 1978 [1]. It depends on the "Integer Factorization Problem" which is hard to solve if the bit length of the operands is adequately large[*]. We introduce three well-known characters in the cryptology world to describe RSA: Alice, Bob and Eve. Alice and Bob want to communicate through a common channel, which is wire-tapped by Eve. We will assume that Eve is a passive adversary who can see every message going through the channel; however she is unable to alter it. Alice and Bob have to share a secret key to encrypt the data they are sending via symmetrical cryptography; therefore, at first they have to exchange the common key using this channel.

1.1.4 The Visualization of RSA

- Alice has a safe and its key. She sends the safe unlocked to Bob, but keeps the key.
- Bob generates the common key (using a random number generator) to be used in the symmetrical cryptography and puts it into the safe and locks it.
- Bob sends the "locked safe" back to Alice.
- Alice (using the key of the safe) unlocks the safe and gets the hold of the common key (which is generated by Bob).
- As they have the same secret key, they can communicate by a symmetrical cipher using that key.

Figure 1 depicts each step of this communication.

[*] Today's most common use is 1024 bits.

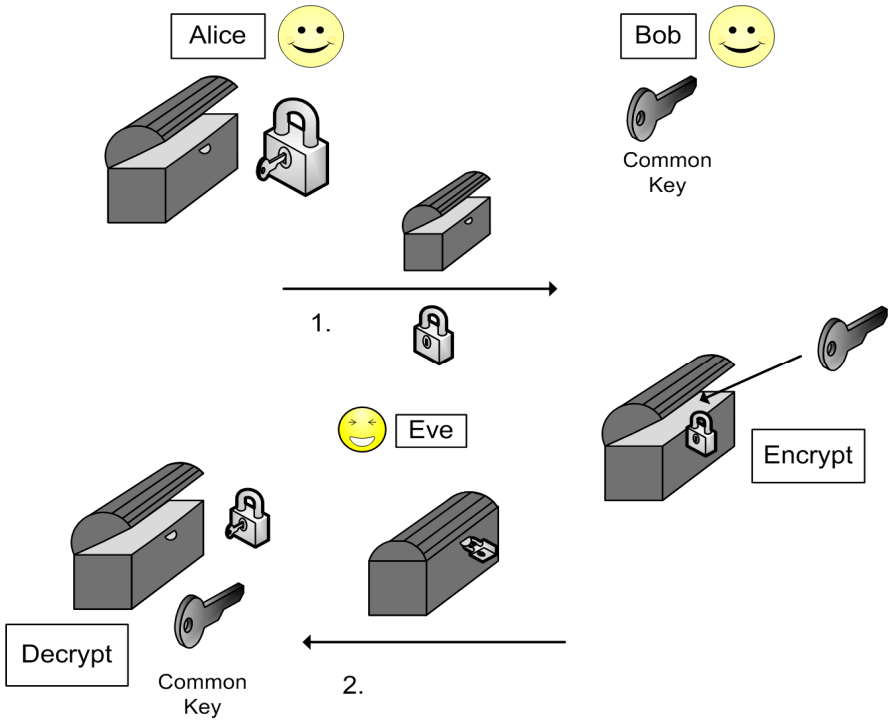


Figure 1. Visual Depiction of RSA

1.1.5 RSA Setup

For k bit security level, following operations must be carried out by Alice:

- Find two distinct (k/2) bit long prime numbers (p, q)
- Calculate $n=p.q$
- Calculate Euler's Totient Function $\Phi(n)= (p-1)(q-1)$
- Generate a random number e $(e<n)$, which is relatively prime to $\Phi(n)$; in other words, GCD* $(\Phi(n), e)$ must be 1.
    - e is the public key.
- The multiplicative inverse of e in mod $\Phi(n)$ gives out private key, d, which is calculated by "Extended Euclidian Algorithm":
    - $d \equiv e^{-1} \bmod \Phi(n)$

After these computations Alice sends $\{e, n\}$ to Bob and keeps d.

* Greatest Common Divisor

1.1.6 An RSA Example

Alice has to perform the following calculations:

- p=13 and q=11 (randomly choose two prime numbers)
- n=$p.q$=143
- $\Phi(n)=(p-1)(q-1)=12 \times 10=120$.
- e=7, (randomly chosen) check GCD(120,7)=1
- $e^{-1} \pmod{120} \equiv d \equiv 103$ (calculated by Extended Euclidian Algorithm)

Then Alice sends {e=7, n=143} to Bob.

Bob has a secret message m=111 ($m<n$). He must calculate $C=m^e \pmod{n}$

$C= 111^7 \equiv 45 \pmod{143}$.

Bob sends "C=45" to Alice. Alice decrypts the message using her private key d:

$45^d=45^{103} \equiv 111 \pmod{143}$

1.1.7 Modular Exponentiation

As the previous example indicates, RSA is based on modular exponentiation. However, if we try to perform an exponentiation operation of a large number by just successive multiplying, it can take years to calculate for large exponents. Instead, we can use "Binary Exponentiation" (or Square and Multiply algorithm):

Algorithm 1: Binary Exponentiation (left to right)

Inputs: m→base, e→exponent $(e_{k-1},e_{k-2},\ldots.,e_1,e_0)_2$

Output: result

1. result←1
2. for i=k-1 to 0 do
   a. result←result * result
   b. if (e_i==1) then result←result*m
3. return result.

The above algorithm is valid both in this normal form and in modular arithmetic form. However, Algorithm 1 has some weaknesses against side channel attacks and

timing attacks (see Section 1.2), therefore we must use a more secure algorithm. As the steps 2.a and 2.b in Algorithm 1 indicate, the core operation in modular exponentiation is modular multiplication.

1.1.8 Modular Multiplication

We can compute modular multiplication by the following operations:

- Ordinary multiplication (or Shift and Add Method)
- Trial division to find the remainder.

This technique is acceptable for one or two multiplications; however the loops in RSA will be iterated *thousands* of times; therefore, we need a better algorithm to solve this problem.

Modular multiplication is the most time consuming operation in RSA and entails prohibitively expensive multiplication and subsequent division operation; and thus is also very demanding on hardware resources. Montgomery Multiplication (MM) algorithm [2] enables these costly operations to be performed easily and efficiently both in hardware and software because it replaces the time-consuming division operation in the reduction phase with simple shift operations. The method consists of repeated additions and shifting; therefore it is well-suited for hardware implementations. The Montgomery Multiplication algorithm for radix-2 is given in Algorithm 2.

Algorithm 2: Radix-2 Montgomery Multiplication

INPUT: X = Multiplicand, $(X_{k-1}, X_{k-2}, \ldots, X_1, X_0)_2$, $X<N$
Y = Multiplier, $Y<N$
N = Odd modulus

OUTPUT: $MM(X,Y,N)=R = X.Y.2^{-k} \pmod N$ where
k = bit length of N.

1. $R \leftarrow 0$;

2. for i from 0 to k-1:
  a. if $X_i=1$, then $R \leftarrow R+Y$;
  b. if R is odd, then $R \leftarrow R+N$;
  c. $R \leftarrow R/2$

3. if $(R>N)$ then $R \leftarrow R-N$;

4. return R

Montgomery multiplication is not efficient when we perform only a few multiplications, because MM operates in *N-residue* class. The normal form and the N-residue form are mapped to each other by a one-to-one function (Figure 2).

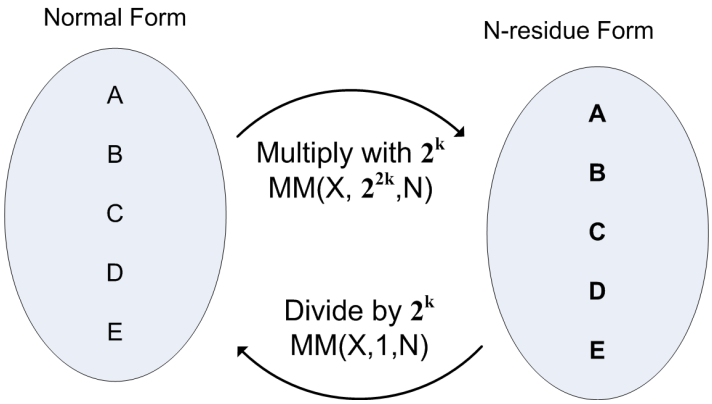


Figure 2. N-Residue Form

To convert the operands to N-residue form, we have to calculate MM(X, 2^{2k}, N) = X.2^k = **X** at first. Now, we can use **X** and its multiples in Montgomery loop; because MM(**X**, **X**, N)= $\mathbf{X^2}$. After the last iteration, we have to use MM one more time to convert the result from N-residue form to normal form by multiplying with one, i.e. MM(**R**, 1, N)=R. There are two extra multiplications in one exponentiation, and as the number of iterations in RSA loop is very large (>512), these additional multiplications will be negligible for practical purposes.

1.2 Side Channel Attacks and Countermeasures

Although the RSA algorithm with certain key sizes is considered safe in the mathematical sense, straightforward implementations of modular exponentiation on hardware (and also software) may have vulnerabilities that can lead the attackers to recover the secret key easily. All implementations (on both hardware and software) have some unintentional, yet observable outputs (through side-channels), which may compromise the desired security level substantially. Therefore, we have to mask the side-channel information such as the variations in the power consumption of the device and execution time of the algorithm that depend on the secret key. We also have to prevent the faulty outputs or so-called safe-errors that can be easily induced by spikes in

the input voltage or any other facile means. Below are the known attacks and their countermeasures:

1.2.1 Simple Power Analysis Attack (SPA)

In the binary exponentiation algorithm (Algorithm 1), there are two operations (square and multiply) with different power characteristics. In most hardware implementations, squaring consumes less power than multiplying; therefore, by analyzing the instantaneous power consumption of the device, an attacker can deduce the secret key even in a single run. Therefore, we have to choose an algorithm that has a more regular execution pattern than the ordinary square and multiply method.

1.2.2 Timing Attacks

We can make Algorithm 1, resistant against SPA attacks by squaring and multiplying in each step regardless of the exponent (using different variables) and in the next iteration, we have to select the correct variable according to the exponent, therefore in case of having 0-bit in the exponent, we have performed one dummy multiplication (we will use the outcome of the squaring operation only). This method is called "Square-and-multiply-always" algorithm. However, when Chinese Remainder Theorem (CRT) is used for faster execution times, there will be compare and subtract steps (which is comparing message m with secret primes "p and q") at the beginning of the algorithm. By finding three distinct m values, ($m_1 < p < m_2 < q < m_3$) (The execution time will be largest for m_3); the attacker can find the ranges of the secret primes (i.e. p and q). A brute force search in these ranges can be used to factorize n.

1.2.3 Fault Injection Attacks

Square-and-multiply-always algorithm simply masks the Hamming weight of the exponent. However, this method cannot resist against the so-called "C and M safe-error" attacks depicted in [20], which are based on faults inflicted on dummy multiplications and used memory locations respectively, which do not change the final outcome. This allows the attacker to distinguish the dummy operations and consequently obtain the secret key bits.

Algorithm 3: Montgomery Powering Ladder

Inputs: m = input message,
d = $(d_{t-1},...,d_0)$ exponent.
Output: $C=m^d$

1. $R_0 \leftarrow 1$
2. $R_1 \leftarrow m$
3. for i=t-1 to 0
a. if (d_i==1)
$R_0 \leftarrow R_0R_1$ $R_1 \leftarrow (R_1)^2$ //in parallel
b. else
$R_1 \leftarrow R_1R_0$ $R_0 \leftarrow (R_0)^2$ //in parallel
4. return R_0

An efficient countermeasure to these attacks is to use the "Montgomery Powering Ladder" algorithm (Algorithm 3) proposed in [20]. This algorithm additionally provides parallelism for hardware implementations and is highly regular. A fault induced in any step of the algorithm escalates to the end of the execution which always produces an incorrect result; therefore the attacker cannot find any relation to the secret exponent.

Many techniques can be employed to prevent outputting faulty results. As the errors induced by C and M safe-error attacks are completely at random, running the algorithm twice and checking the equality of the two results can easily prevent these attacks. Another method is after the calculation of $m^d \equiv C \pmod{n}$, checking whether $C^e \equiv m \pmod{n}$, where e and d are public and private keys respectively; however both methods are costly. In case a countermeasure is needed, the extra check proposed in [23] can be incorporated to the data path with *virtually* no cost.

1.2.4 Differential Power Analysis Attack (DPA)

If a cipher algorithm is deterministic, an attacker uses the implementation as an encryption oracle and encrypts as many messages as possible. Statistical analysis of the differences in the power traces sampled for different input messages may reveal secret exponent "*d"*. In "the doubling attack" which is explained by *Yen et al* [21], the implementation can be broken in only two runs (one with input message *m*, other one with m^2).

A well-accepted countermeasure is to embed randomness into the algorithm so that the power traces of each run will be different, even if the same input values are used. There are three kinds of randomization methods that can be used in modular exponentiation $m^d \equiv C \pmod{n}$:

(1) Message Blinding: We choose a random variable $r < n$ and calculate $r^e \pmod{n}$ and multiply this with C^*:

$$(C \times r^e)^d \pmod{n} \equiv C^d \times r^{ed} \equiv m \times r \pmod{n}.$$

At the end of exponentiation, we can get the ciphertext C back by multiplying it with $r^{(-1)} \pmod{n}$. The multiplicative inverse of the random value r has to be computed on the fly, which is naturally costly and undesirable from the implementation point of view.

(2) Modulus Blinding: The modulus n is multiplied by some random variable r (here r around 2^{16} is enough for practical purposes) and all exponentiation operation is carried out in mod $(n \times r)$ [23]. We need to reduce the result to (mod n) at the end of the calculation. In this method, we need to compute additional variables[†], which makes this method time costly.

(3) Exponent Blinding: We can add random multiples of $\Phi(n)$ to the exponent before the main computation. At the end of the exponentiation, there is no need for correction since $m^{d+r.\Phi(n)} \equiv m^d \pmod{n}$; therefore, this method of blinding costs considerably less than the other methods. For all practical purposes a 17-bit random number (the word size used in this implementation) r is sufficient resulting in $(1/s \times 100)$ percent overhead, where s is the number of words in the exponent d.

There is another type of attack where all precautions to prevent side channel attacks, except for modulus blinding, fail. When the input message is selected as $m = (n-1)$, there will be two intermediate results (see [21]) depending on the related bit

[*] ciphertext

[†] For $2^{N-1} < n < 2^N$, we need to calculate both $2^{2N} \pmod{n}$ and $-(n_0)^{-1} \bmod 2^N$, and $2^{2N+32} \pmod{n}$ and $-(n_0)^{-1} \bmod 2^{N+16}$ where r is a 16-bit integer for the Montgomery multiplication.

of the exponent: 1[*] and $(n\text{-}1)$[†]. The instantaneous power graph of a single run will immediately shows two distinct characteristics for these two possible outputs; therefore, as a simple countermeasure, we propose to halt the calculation if the input message is selected as $(n\text{-}1)$, which is easy to check.

1.3 Dedicated Hardware Basics

There are two dominant types of dedicated hardware in the market: ASICs and FPGAs. The main difference between these two is re-programmability. ASICs functionality, once manufactured, cannot be changed at all; however FPGAs can be re-programmed many times. On the other hand, ASICs take the advantage of being fine grained; therefore have a higher computation performance and lower power consumption when compared to FPGAs.

Both hardware platforms have the technology metric, "the feature size[‡]" (like 90 nm or 65 nm). As the feature size shrinks, maximum possible frequency increases, the power consumption and the cost of the chip decrease.

1.3.1 ASICs

ASIC is an integrated circuit (IC) customized for a particular use, rather than general-purpose use. A typical ASIC generally may have the following components:

- **32-bit CPU** (maybe 16-bit or 64-bit) → Central Processing Unit: The unit which carries out sequential operations those are not "dense".
- **ROM** → Read Only Memory: Non-volatile memory which is generally used to store constants and conversation tables.
- **RAM** → Random Access Memory: Volatile memory which holds the data required by current operations.
- **Flash Memory** → Non-volatile memory, which can be written and read.

[*] For even exponents

[†] For odd exponents

[‡] The minimum feature size is the width of the smallest line or gap that appears in the design.

- **DSP** → Digital Signal Processor: A unit that is specifically designed for performing frequently used operations (like MAC* operations)

The complexity of an ASIC is usually measured with the number of logic gates it has (e.g. 10 k gates or 2 M gates). According to the number of gates, the ASICs can be divided into groups like: VLSI (Very Large Scale Integration) or ULSI (Ultra Large Scale Integration).

ASICs have two types of development procedures: Full-Custom versus Standard Cell Library based designs. In full-custom designs, special acceleration and PAR (Place and Route) methods can be applied on smaller scale (which can be layout level) for better performance. In "Standard Cell Library" based designs, the functions in HDL† code are directly mapped to predefined cells like AND2, OR4, NAND3, FF, etc. by CAD tools.

1.3.2 FPGA

The basic building block of FPGAs is called slice. Each slice has two flip-flops as storage units and two 4-input SRAM based LUTs‡ which can be programmed as combinational functions, two independent fast carry chains, and MUXs among them [18]. Each slice can assume the role of a LUT, shift register or distributed RAM. The term "CLB§" can be used for describing two or four slices. According to their manufacturing purposes, FPGAs can have the following structures:

- **Embedded processors** (soft or hard): Some expensive operations can be performed in hardware (by LUTs), while less complex and sequential operations can be carried out by a general purpose CPU. The balance between hardware and software will provide the best time area product according to the design specifications.
- **DSP units:** Multiply and Accumulate (MAC) operation is frequently used in DSP, therefore some FPGAs have dedicated blocks (e.g. DSP48 in Xilinx Virtex 4 Series) to perform multiplication and addition operations efficiently.

* Multiply and Accumulate

† Hardware Description Languages, such as Verilog and VHDL.

‡ Look Up Tables.

§ Configurable Logic Block

- **Multipliers:** While the adders can be implemented efficiently with LUTs which utilize fast carry chains, the multiplication operation implemented by LUTs performs poorly. Therefore many FPGAs have dedicated multiplication units that support up to specified bit length (e.g. 17 bit unsigned operands).
- **Dual-port Block RAM (BRAM):** As the distributed memory consumes slices, which are one of the most important resources in FGPA designs; large portions of data must be stored in Block-RAMs. Block-RAMs have synchronous write and read ports which provide fast access times.

1.3.3 The Differences between ASICs and FPGA

The advantages of FPGA platform can be summarized as follows:

- **Field re-programmability**: The implementation can be upgraded or changed at any time without any cost. The user just needs to upload the new bit stream.
- **Shorter time-to-market**: There is no need to deal with layouts, masks or other manufacturing steps in FPGA designs.
- **No upfront NRE**[*]: The FGPA design flow is cost effective when small volume of chips is needed.
- **Simpler design cycle**: Automated software takes care of all design steps. (from synthesis to PAR stage)

On the other hand, ASIC has beneficial properties like:

- **Full custom capability:** ASICs are custom built circuits; therefore the designers have the opportunity to optimize the implementation in terms of both area and speed.
- **Higher raw internal clock speeds**: ASIC implementation allows higher frequencies, even if their feature size is the same with the FPGAs.
- **Lower unit costs:** For very high volume productions, the cost per chip in ASICs is lower than that of FPGAs.
- **Smaller form factor**: The area utilization is higher in ASICs when compared to FPGAs, because some sources have to be wasted in an FPGA, when the circuit cannot exactly fit the board.

[*] non recurring expenses

2 Radix-4 Implementation of 2048 bit Modular Multiplication on ASIC

Previous studies with pipelined approach* for modular multiplication ([6], [8], [9], [11]) generally suffer from high latency because of the data dependency among processing elements (PE) in ASIC implementations. Although their performance can be adjusted by various parameters such as bit length of the words and number of PEs, they have an upper speed bound on which adding more PEs have no beneficial effect. Likewise, conventional radix-2 based non-pipelined organizations ([7], [10]) cannot be optimized further than N clock cycles (bit length of the modulus); for instance, the fastest implementation [10] needs $N+1$ clock cycles. As speed being our primary concern in this ASIC implementation, we focus on decreasing the total clock cycles with radix-4 scheme, with minimal area. We lay out the synthesis results for both UMC ASIC library and FPGA.

2.1 Algorithm

The MM can be used with different radices. Popular choices are radix-2, radix-4 and radix-8 [24]. In [25], it is shown that radix-4 would be a wise choice in terms of speed and area. (Radix-2 is slow, and radix-8 is under-utilized). Higher radices than 8 need much more space and necessitate additional calculations, which reduce overall efficiency and frequency.

2.1.1 Booth Recoding

We have to express the multiplier, X, in MM using a different representation, known as Booth recoding [26], where the digits are {-2,-1, 0, 1, 2} for efficient radix-4 implementation (the multiple "3" is not used because it cannot be calculated easily by shift and/or invert operations). The booth converter reads three consecutive bits of the

* The calculation is divided into simple stages that can be overlapped to speed up the operation.

multiplier, *X*, to decide which multiple of the multiplicand, *Y*, is going to be added. The conversion ($-2X_i + X_{(i-1)} + X_{(i-2)}$) is shown in Table 1.

For instance, with Booth recoding technique, $27 = (011011)_2$ can expressed as: (beginning from $i = 1$, and adding 2 to i in each iteration since base is 4):

- For i = 1: 110 = -1 (empty bit is assumed as 0 i.e. $i_{(-1)} = 0$)
- For i = 3: 101 = -1
- For i = 5: 011 = 2

$27 = (2,-1,-1)_{Booth}$

X_i	$X_{(i-1)}$	$X_{(i-2)}$	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table 1. Booth Recoding

We can check the recoding as follows:

$-1 \times 4^0 = -1$

$-1 \times 4^1 = -4$

$2 \times 4^2 = 32$

The result is 32-4-1 = 27.

2.1.2 Radix-4 Implementation

In radix-2 MM, we add the modulus *N* to the partial result *R*, if *R* is odd, and shift right afterwards (Algorithm 2). In radix-4 MM, we must add multiples of *N* to make the result divisible by four, because we need to shift *R* to the right twice. We have four possible multiples (0, 1, 2, 3) of *N* and as these values are in radix-4 modular arithmetic, we can rewrite these possibilities as (0, *N*, 2*N*, -*N*). Therefore, all the multiples of *N* we need, are easy to handle in hardware by just shifting and/or bitwise

inverting. For signed calculations, “two’s complement form” is used, so we must add one (using carry-in of the adders) for negative *Y* (-*N* does not need this correction since it is an odd integer). The radix-4 MM algorithm, which is similar to the radix-4 multiplier in [25] is given in Algorithm 4.

As the numbers used in Algorithm 4 are signed, they have to be represented with one bit more than *N*, i.e. *k+1*. At the end of calculation, negative results must be converted by one last addition of *N*.

Algorithm 4: Radix-4 Montgomery Multiplication

INPUTS: N is odd modulus
X is multiplicand, $X=(0,0,X_{k-1},X_{k-2},\ldots,X_1,X_0)_2 < N$, $X_{-1}=0$
Y is multiplier, Y<N
{} is used for bitwise concatenation.

OUTPUT: $MMr4(X,Y,N)=R = X.Y.2^{-(k+2)} \pmod N$ (k=2048)

1. R ← 0;

2. for i from 1 to k+1 step 2:
  a. R←R + Booth{X_i, $X_{(i-1)}$,$X_{(i-2)}$}*Y
  b. if ({R_1,R_0}+ {N_1,N_0})%4==0) then R←R+N;
  c. else if ({R_1,R_0}+{N_0,0})%4==0) then R←R+2N;
  d. else if ({R_1,R_0}-{N_1,N_0})%4==0) then R←R-N;
  e. R←R/4
3. return R

2.2 Architecture

2.2.1 Carry-Save Adder (CSA)

We need a very fast yet small adder for the Montgomery modular multiplication algorithm, as the addition will be the core operation that will be repeated millions of times. Carry ripple adder can be used for smaller operands; whereas carry propagation for larger numbers (like 1024 bit) will be a significant limitation on the whole circuit. As we are adding to the same number *R* in each step of the algorithm, we can use carry save adders as shown in Figure 3, whose longest combinational path is independent from bit length of the operands. Overall critical path is just serially connected XOR gates in the full adder. CSA takes three operands and reduces them to two and the result is in redundant format. (i.e. sum and carry are calculated and stored separately):

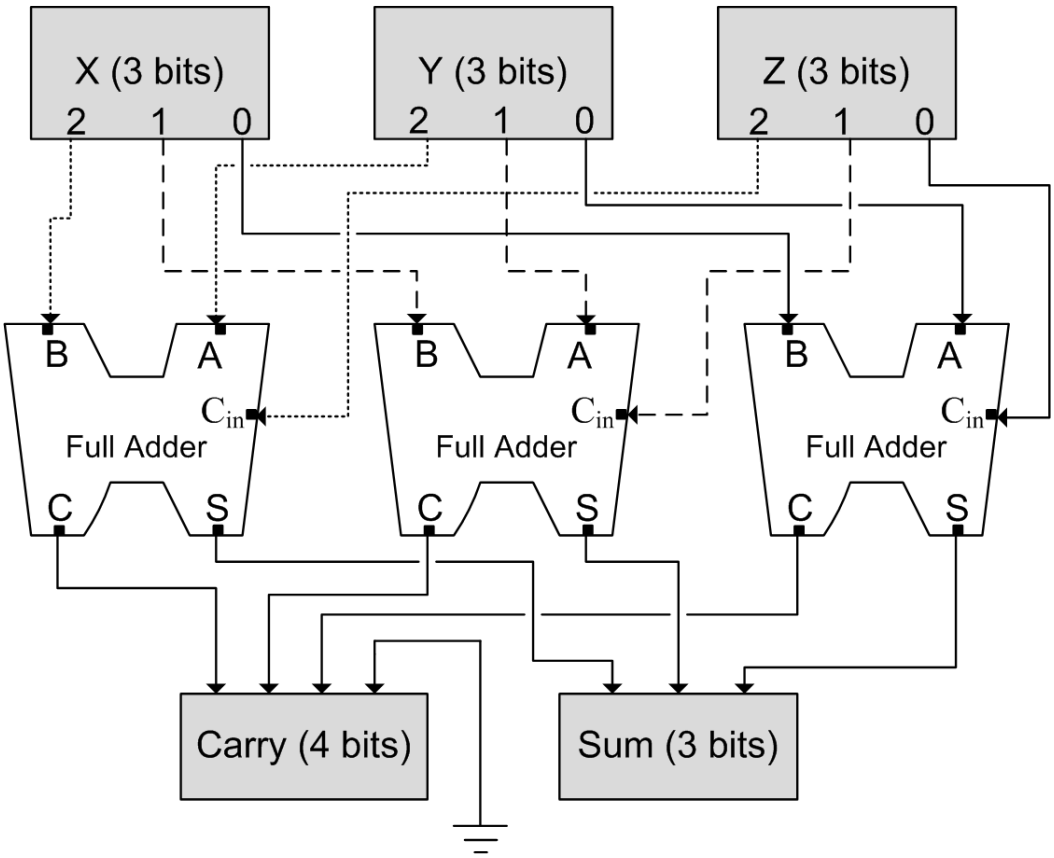


Figure 3. A three-bit CSA

2.2.2 Carry Propagate Adder (CPA)

At the end of each multiplication, we have to convert the result from redundant form to normal form. As we cannot directly add these two 2048-bit integers (i.e. *sum* and *carry*) in one clock cycle (due to hardware limitations), we have to divide this process into sub additions. We used a 32-bit carry ripple adder (CRA) for 64 clock cycles as shown in Figure 4.

The CPA has two 32-bit 64-to-1 MUX and one 2048-bit shift register. Two 2048-bit input registers (*sum* and *carry*) is part of the MM logic, not CPA; therefore we only need one 2048 bit register for CPA.

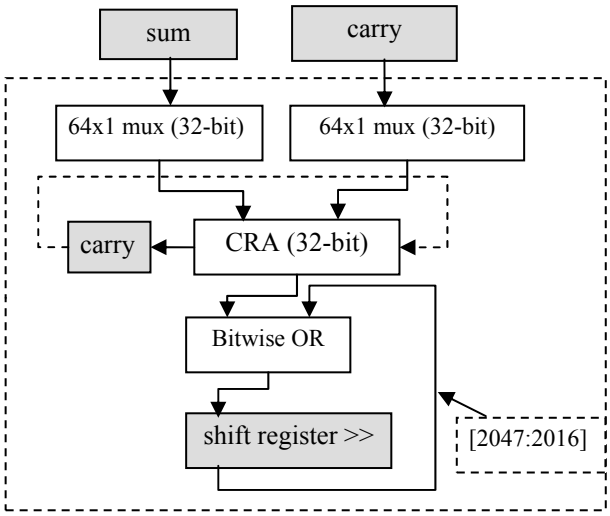


Figure 4. Carry Propagate Adder*

In radix-2 implementation, we can add four possible values to partial product in each iteration and these values are (0, *N*, *Y*, *Y*+*N*). One could pre-compute Y+*N* at first

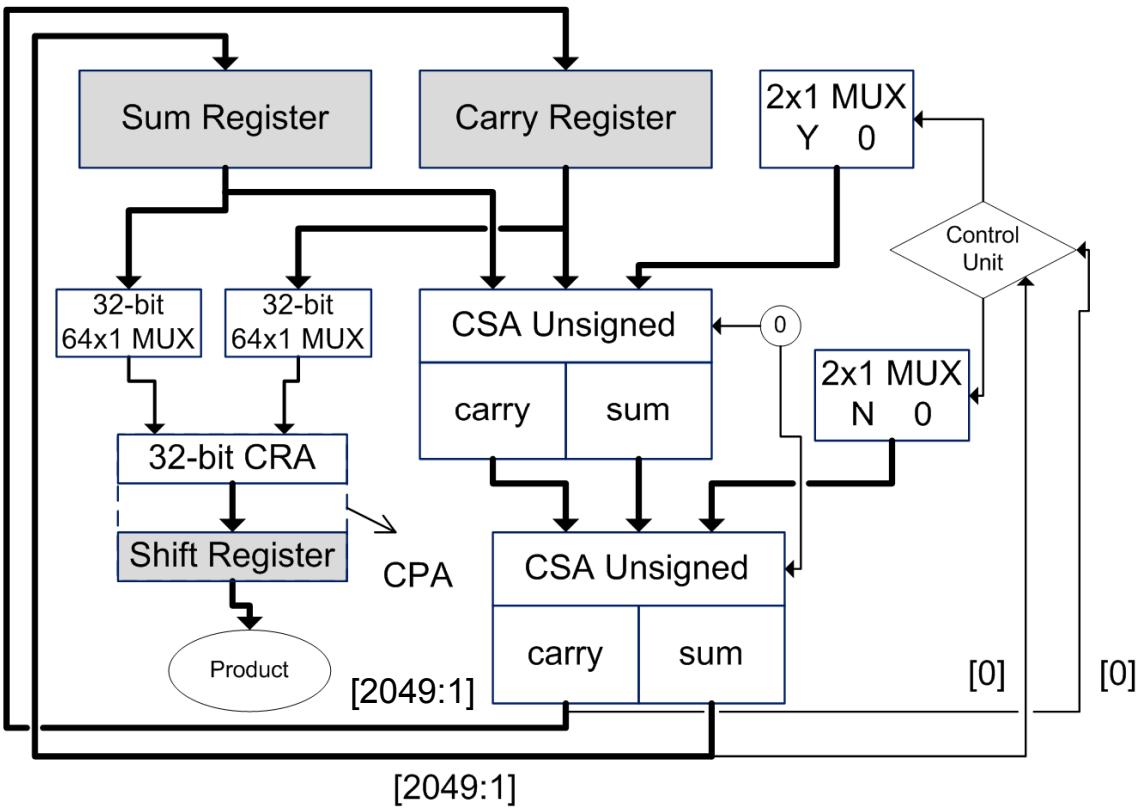


Figure 5. Radix-2 Montgomery Multiplier

* Grey boxes are registers and [ ] is used for bitwise concatenation.

and then store it in a register, so using only one CSA array of 2048 bit would suffice. Another approach (instead of one CSA array and one extra register of 2048 bit) is using two CSA arrays and add *N* and *Y* on-the-fly, which turns out to be more efficient; as depicted in Figure 5.

In radix-4 implementation, we need 12 pre-calculated numbers so storing all of them will be redundant. (Four possibilities are from booth recoding of *X* (i.e. *Y*, 2*Y*, -*Y*, -2*Y*) and three possibilities are from *N* (i.e. *N*, 2*N*, -*N*). Therefore we need to add these numbers using two CSA's like in the radix-2 case as shown in Figure 6.

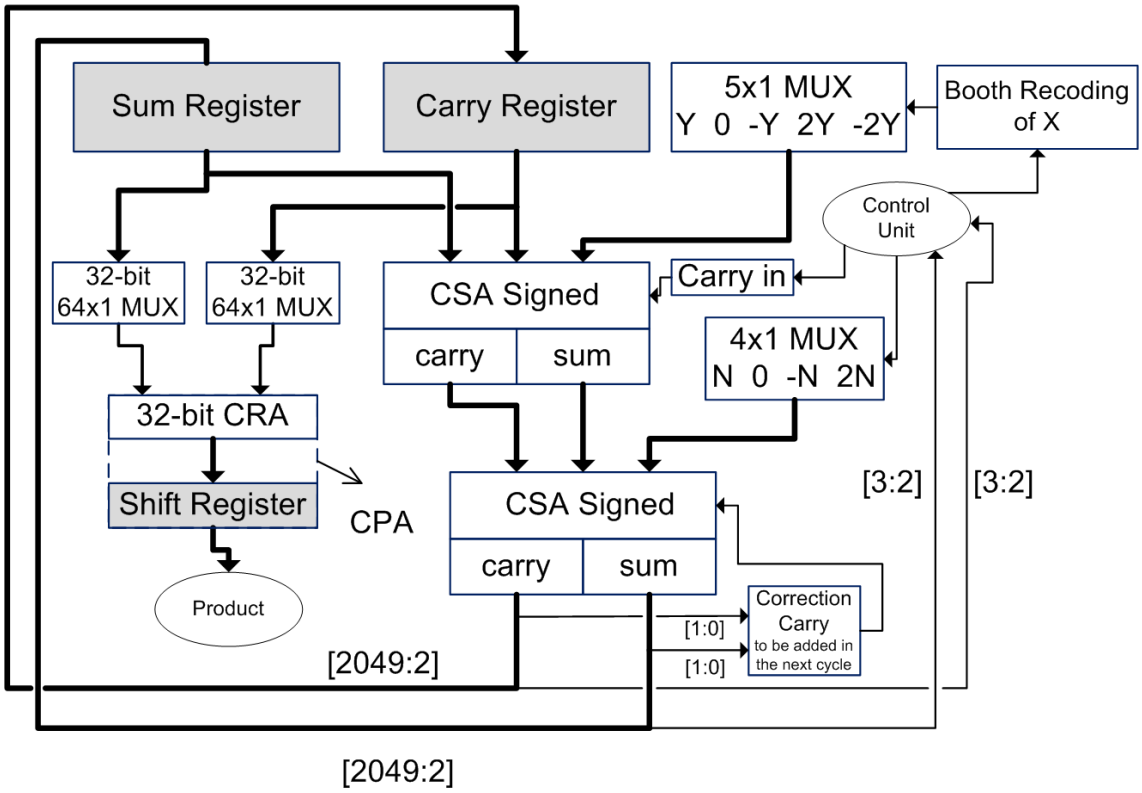


Figure 6. Radix-4 Montgomery Multiplier

In Figure 6, the CSA arrays are signed as the inputs can be negative. The carry-in of the upper CSA is used for two's complement of *Y*. Although the number from the output of the second CSA is divisible by 4, we have to check two least significant bits (LSB) of *carry* and *sum* registers in order not to lose information, because the output is in redundant format. The possible values of the two LSB of the *sum* and *carry* registers are ({00, 00}, {01, 11}, {11, 01}, {10, 10}). Except for the first one (00, 00), the remaining cases need to be corrected by a carry that is going to be added in the next clock cycle via the carry in of the lower CSA array.

2.3 Theoretical Analysis of Performance

We examine various designs synthesized for different technologies to compare with ours. Each design offers a distinct solution to the same problem and has its own advantages and disadvantages. We focus on modular multiplication performance and Table 2 shows the number of clock cycles required for one 2048-bit modular multiplication with different Montgomery multipliers.

Design	Implementation	Time Complexity	Clock Cycles
Proposed	Radix-4	N/2+(N/32)*1,5	**1120**
[14]	Radix-2, pipelined	N	2048
[10]	Radix-2	N+1	2049
[11]	Radix-2, pipelined	N+3	2051
[7]	Radix-2	N+2+N/32	2114
[8]	Radix-2, pipelined*	(N/(w_1*p)). Max (p , N/w_2)	2124
[6]	Radix-4, pipelined	$\left\lceil \frac{N}{2NS} \right\rceil \left(\left\lceil \frac{N}{w} \right\rceil + 1 \right) + 2NS \, , if \left\lceil \frac{N}{w} \right\rceil > 2NS$ $\left\lceil \frac{N}{2NS} \right\rceil (2NS + 1) + \left\lceil \frac{N}{w} \right\rceil + 1 \, , otherwise$	~2128 (w=128, NS=8)
[9]	Radix-2, pipelined	2N +N/w-1 if N/w+1 < (2p)	~4128
[15]	Radix-2, pipelined	3*N	6144

Table 2. Time Complexities† for One 2048-bit Modular Multiplication

For the design in [8], it is stated that 2048-bit modular multiplication takes 26,55 μs with a 80 MHz clock, therefore we can deduce that the multiplier uses around 2124 cycles to compute one modular multiplication. In this architecture, the word-length of the operands, X and Y, can be chosen separately (w_1 and w_2).

Our radix-4 implementation and [7] uses CPA to convert numbers from redundant format to non-redundant format. The radix-2 multiplier in [7] needs only one addition with CPA; therefore the overhead is N/32. The architecture described in [10]

* For pipelined design, we try to find optimum values.

† w is bit length of a word and p is the number of processing elements or pipeline stages (PE or NS)

does not need this conversion, but uses more registers to compensate. Our radix-4 design first converts the number (costing *N/32* cycles) and checks whether the result is negative. If, for instance, the result is negative, it adds *N* to result to bring it to the desired range which is from *0* to (*N-1)*. The result can be 50% percent negative on average so the overhead will be (*N*/32)*1.5.

2.4 Synthesis

We implemented the design using "Verilog" HDL and verified with "ModelSim" [30] simulation tool. We synthesized to both UMC 0.18 μm Standard Cell Library and Xilinx xc2v6000-6bf957 FPGA, which has following properties:

- Number of Slices: 33792
- Number of Slice Flip Flops: 67584
- Number of 4 input LUTs: 67584

To compare our radix-4 design with [7] and similar designs, we also implemented a reference radix-2 core whose schematic is depicted in Figure 5. Both radix-2 and radix-4 designs have been synthesized with area and speed priority to consider time x area criterion (using execution time and slices) in FPGAs.

2048-bit (MM)	Slices	Freq	Ex. time (μs)	Time x area
Radix-4 (speed)	16657	132.4	8.47	140 905
Radix-4 (area)	16549	90.9	12.34	203 904
Radix-2 (speed)	12564	137.2	15.41	193 588
Radix-2 (area)	10920	107.1	19.74	215 545

Table 3. Xilinx Synthesis Tool (XST) Synthesis Results*

We try to optimize both designs for a fair comparison. The synthesis results for FPGA, illustrated in Table 3, show that our radix-4 core in comparison with radix-2 design has following features:

* As speed optimized cores have smaller time area product they are going to be used for comparison with other designs

- The frequency decreases a negligible amount (3.6%), because we use inverters along the critical path in radix-4 design.
- Execution time is shortened dramatically (by 82%), since the cycle count is approximately halved, while keeping the frequency nearly the same.
- Time area product also improves by 37%, which is one of our main goals.

Design	Technology	Freq (MHz)	Area	Ex.Time (µs)
Proposed (radix-4)	Xilinx xc2v6000	132.4	16657 slices	**8.47**
Reference (radix-4)	UMC 0.18 µm	80	158k gates	**14.00**
[6]*	AMI05_fast	147 (T=6.8;w=128)	~66k gates	14.90
Reference (radix-2)	Xilinx xc2v6000	137.2	12564 slices	**15.41**
[14]†	Xilinx Virtex 2‡ (?)	129.1	(7222) slices	15.87
[9]	0.5µm CMOS	166 (T=6.0; w=64)	85k gates	24.87
Reference (radix-2)	UMC 0.18 µm	80	118k gates	**26.43**
[8]	N/A	80	N/A	26.55
[10]	Xilinx xc2v6000	70.6	23060 slices	29.02
[11]	Xilinx xcv1000	~52	(11k) slices	39.44
[7] (REDC)	0.65 µm SOG	50	(120k) gates	42.28
[15]	Xilinx V812E-BG-	~96	(10960)	~64

Table 4. Overall Speed Comparison for 2048 bit modular multiplication

We compare performances of the multipliers in terms of execution time in Table 4. The area values in parenthesis are calculated with interpolation and are overestimated, because only the area of the datapath is subject to change in case of doubling the operands' bit length, not the area of the control unit. As one cannot estimate the area coverage of the datapath and control unit separately for each reference design, we have to assume that the datapath dominates the whole circuit area for comparison purposes.

Our radix-4 design has the lowest execution time in Table 4 and outperforms the other designs by a great margin, although it does not have the highest frequency. Our

* The areas of [6,9] are calculated with the parameters given in Table 2.

† Excluding pre- computation unit: This design uses pre-computed values, but the pre-computation unit is not included in the multiplier (it is a part of the exponentiation circuitry)

‡ [14] does not mention the model of the FGPA, therefore we assumed that Xilinx Vertex 2 series was used.

UMC 0.18 μm implementation is also faster than other ASIC based designs. The authors in [10] claim to have the fastest MM whose execution time is more than triple of ours for the same technology.

Although Table 4 shows a general comparison among a wide range of designs, we have to compare time area products of FGPA designs and ASIC designs separately. Not all designs are synthesized to the same technology, therefore we cannot directly use execution times, instead we can use "total clock cycles for one multiplication" as the performance indicator.

Design	Area (slices)	Clock Cycles	Time x Area*
Reference (radix-2)	12564	2114	1.424
[10]	23060	2049	2.533
[11]	11000	2051	1.209
[14]	7222	2048	0.793
[15]	10960	6144	3.609
Purposed (Radix-4)	16657	1120	**1.000**

Table 5. The Time Area Products for FPGA designs

In Table 5 and Table 6, the proposed design and the other multipliers are compared in terms of (area × time) metric. The design in [14] seems to have the smallest time-area product in Table 5 ; but, as stated before, the pre-computation unit which will consume considerable amount of hardware space is not included in the area. Therefore, the proposed radix-4 design has one of the best (area × time) metric among FPGA designs.

Design	Technology	Area (k gates)	Clock Cycles	Time x Area
Proposed	UMC 0.18 μm	158	1120	**1.000**
[6]	AMI05_fast	66	2128	0.794
[9]	0.5μm CMOS	85	4128	1.983
Reference (radix-2)	UMC 0.18 μm	118	2114	1.410
[7] (REDC)	0.65 μm SOG	120	2114	1.434

Table 6. The Time Area Products for ASIC designs

* Time area product is normalized to proposed design.

In Table 6, only [6] has a better time area product than ours. Our radix-4 implementation offers 41% decrease with respect to its radix-2 reference, which is a similar case shown in Table 3.

2.5 Conclusion

We modified the well known radix-2 Montgomery multiplication (Algorithm 2) to radix-4 scheme to decrease (time × area) product and obtained a significant reduction (37% in FPGA and 41% in ASIC) in comparison to our radix-2 reference core. Our radix-4 implementation for 2048 bit operands is the fastest among the other designs given in the literature (both in execution time and total clock cycles). The main advantage of the radix-4 scheme over radix-2 schemes is providing twice the performance at the expense of placing bitwise inverters in the critical path. Using larger MUXs and the necessity to convert negative numbers to positive at the end (3% increase in clock cycles on average) are among the main disadvantages of the proposed architecture, which are greatly compensated by the reduction in execution time and (time × area) metric.

3 Parametric, Secure and Compact Implementation of RSA on FGPA

One encryption or decryption operation in RSA, the first and most widely deployed public key cryptosystem, requires the execution of thousands of modular multiplications. The challenge is usually designing fast hardware multipliers to meet the timing requirements of cryptographic applications, which cannot be attained with software realizations on general-purpose processors. The endeavors toward designing *the* fastest hardware multipliers are meaningful especially when the throughput is of a concern (e.g. in server applications where thousands of cryptographic operations are performed). In literature therefore, there is a plethora of reports of very fast implementations of modular multipliers in hardware, which utilize considerable amount of resources.

ASIC's and FPGA's are two commonly used hardware platforms for cryptographic implementations where the latter becomes more and more popular recently since it is reconfigurable and relatively easy to access from economical and usability point of view. Therefore, some of the previous works utilize resource rich, but relatively expensive FPGA devices to design fast multipliers. There is, however, a paucity of interests in the implementation of multipliers on the smallest and the most economically accessible FPGA devices such as Xilinx Spartan 3 series [18]. As our dependency on public key operations is increasing at an impressive rate even on the simplest devices such as car keys and identity cards, there is a great initiative to design fast multipliers on the cheapest possible way; therefore Xilinx Spartan 3 FPGAs make the products financially viable and shortens the time-to-market period.

As the security level provided by public key cryptosystems is directly related to the bit length of the key and 1024-bit RSA is thought to be not providing adequate level of security any more, RSA with 2048-bit (and longer) keys will be more and more popular to meet the security challenge of the future. Therefore, there is a strong need to

implement multipliers with the longest key possible on the cheapest device without sacrificing speed.

Koc et al. [5] proposed several algorithms to implement the Montgomery multiplication operation in software. These algorithms also prove to be useful for hardware implementations when fast block multipliers are available as in the case of many FPGAs. Moreover, these multipliers can work in a pipelined fashion to take the advantage massive parallelism, despite the fact that these software algorithms are originally designed for a single multiplier that is available in general-purpose processors.

Previous studies employing conventional radix-2 based non-pipelined organizations [7, 10, 19] and pipelined approaches [6, 8, 9, 11, 15, 17] generally avoid using multipliers which consume a considerable amount of chip space, and have a long combinational delay. Instead, they perform multiplication by repeated addition through carry-save adders (CSA). Although the repeated addition approach seems to be a reasonable solution for ASIC realizations, the FPGA's have a different inner structure that allows us to implement alternative circuits. For instance, a recent work by *Suzuki* [3] successfully utilizes powerful DSP macro cells available on an expensive FPGA device to achieve the best time performance for multiplication and exponentiation operations.

At first, we present the CIOS method [5] for Montgomery multiplication on which we base our design in Section 3.1. In the following sections, we introduce our architecture and explain the details of its inner workings and comment on the simulation results. In Section 3.4, the results of synthesis are presented. In the next section, we compare our circuit with previous realizations. Finally, we summarize the achievements and contributions followed by a research plan for future in Section 3.6.

3.1 CIOS Method

While all of the multi-precision Montgomery multiplication algorithms in [5] require the same number of word-level multiplications, the number of additions and memory requirements slightly differ. The CIOS method seems to be the best choice for hardware implementation since it has a regular execution pattern and needs only $s+3$ words (the least among the others) memory space where s is the number of words in one operand. Likewise, *McIvor et al.* [4] also conclude that the CIOS method, which is given in Algorithm 5, provides the fastest timing results for FPGA implementations.

The operands and the modulus in Algorithm 5 are represented as arrays of words, e.g. $a = (a_{s-1}, a_{s-2}, \ldots, a_1, a_0)$.

Algorithm 5: CIOS Montgomery Multiplication

Inputs: a_j, b_j: Operand words (w bits each)
n_j: Words of the modulus (w bits each)
s: Number of words in the operands
2^w:=radix, C: carry, S: sum
n_0^{-1}:= multiplicative inverse* of n_0
{}→ used for concatenation

Output: t[i]:= intermediate and final result, all words of t are assigned to 0 at first.

for i=0 to s-1

1. C←0
2. for j=0 to s-1
   a. $\{C,S\} \leftarrow t_j + a_j \times b_i + C$
   b. $t_j \leftarrow S$
3. $\{C,S\} \leftarrow t_s + C$
4. $t_s \leftarrow S$
5. $t_{s+1} \leftarrow C$
6. $C \leftarrow 0$
7. $m \leftarrow t_0 \times (-n_0^{-1}) \bmod 2^w$
8. $\{C, S\} \leftarrow t_0 + n_0 \times m$
9. for j=1 to s-1
   a. $\{C, S\} \leftarrow t_j + n_j \times m + C$
   b. $t_{j-1} \leftarrow S$
10. $\{C, S\} \leftarrow t_s + C$
11. $t_{s-1} \leftarrow S$
12. $t_s \leftarrow t_{s+1} + C$

3.2 The Multiplication Engine

In this section, we outline our design criteria used in the implementation and explain the implementation details.

* "Least significant word of inverse n" in mod 2^r, where $2^{r-1} < n < 2^r$

3.2.1 Design Criteria

It is essential to lay out the design criteria to meet the challenges and requirements of the application. These criteria are enumerated as follows:

(1) The design must be flexible to fit in both small and large FPGA's efficiently with adjustable number of processing elements.

(2) The bit-length of the words must be parametric so that the full performance of multipliers is utilized.

(3) The design must be scalable to work with operands of virtually any length (e.g. 2048 bit, 4096 bit, etc.)

(4) 2048-bit exponentiation engine must easily fit into even a smallest FPGA with a good timing performance.

(5) The implementation must resist against all side channel attacks with minimal overhead.

(6) All hardwired multipliers must work at maximum possible frequency (They should be instantiated as registered multipliers).

(7) All variables for operands must be kept in Block-RAM's to ensure minimum area consumption.

(8) The connection network must be simple yet effective.

As Algorithm 5 is specifically designed for software implementations, we need to modify it for efficient computation in hardware by taking advantage of parallelization through hardwired multipliers. The execution graph of Algorithm 5 modified for pipelined computation is depicted in Figure 7. The circuit essentially consists of processing elements (PEs, shown in Figure 8) which are responsible for executing a single iteration* of the loops in Steps 2 and 9 of Algorithm 5. Once PE_0 generates the first word of the intermediate result (i.e. the least significant word), the next processing unit PE_1 concurrently starts the computation for the second iteration of the loop with the values it obtains from PE_0. When a PE finishes the computing one iteration, it is immediately assigned to the next available iteration. The results of last PEs are kept in dual port Block-RAM.

* Steps 2 and 9 of Algorithm 5 are performed together within the same PE.

3.2.2 Implementation Details

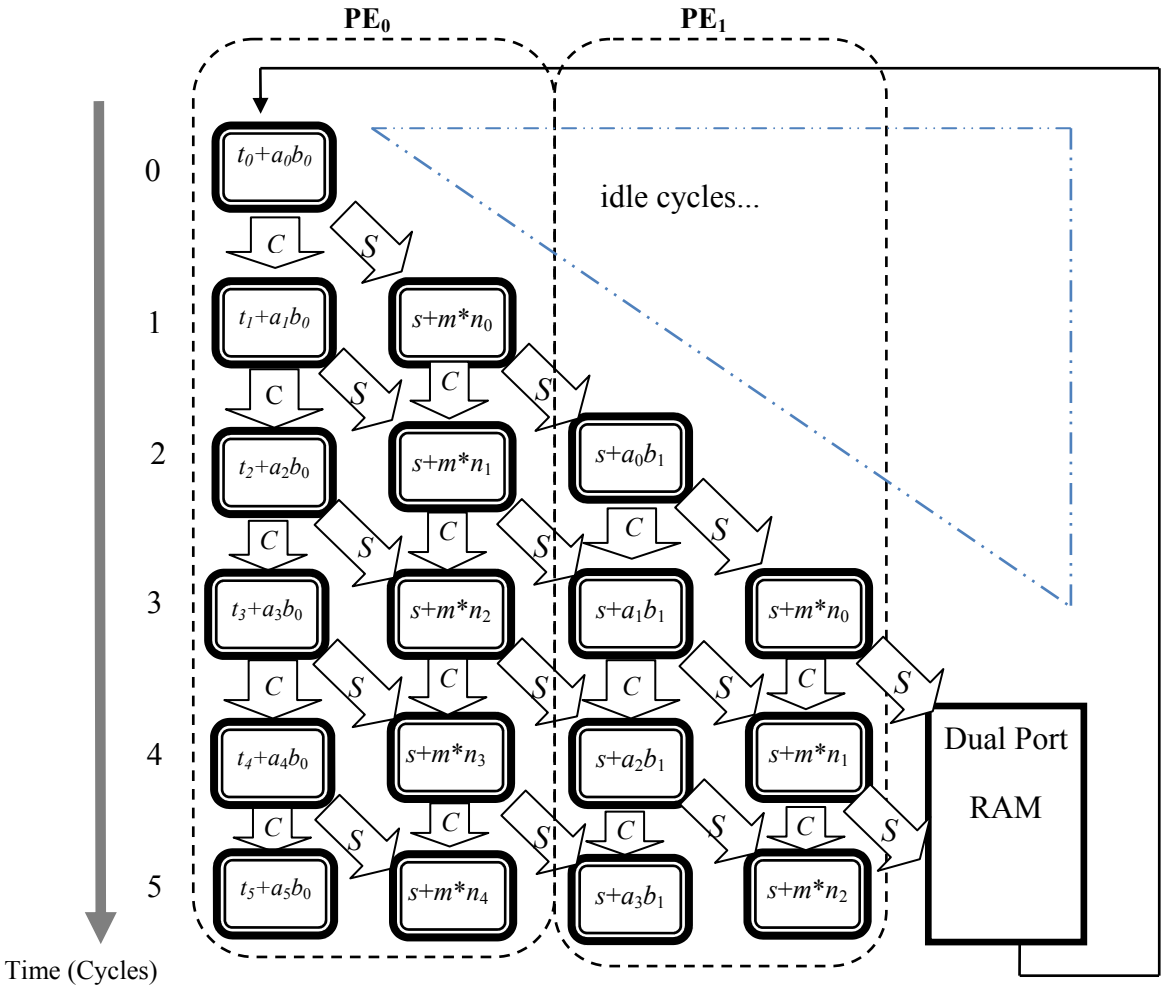


Figure 7. Execution Graph of The Parallelized CIOS algorithm

Before the execution of each iteration of the loop (at each increment of the loop counter "*i*"), the value "*m*" must be calculated as shown in Step 7 in Algorithm 5. (The value of *"n_0^{-1}"* is calculated offline (only one word) and fixed as long as the modulus does not change). However, in the meanwhile, other PEs are still performing multiplication operation, therefore to maintain a continuous data flow, we need to insert FIFO buffers among the PEs and compensate for the time lost by the pre-calculation step. After *"m"* is ready, there are two important steps remaining for execution: Steps 2.a (multiplication) and 9.a (reduction). These steps account for all computation burden since they are word multiplications; the remaining steps are only initializations. Once the value t_i is calculated in Step 2.a, it can immediately be used in Step 9.a.

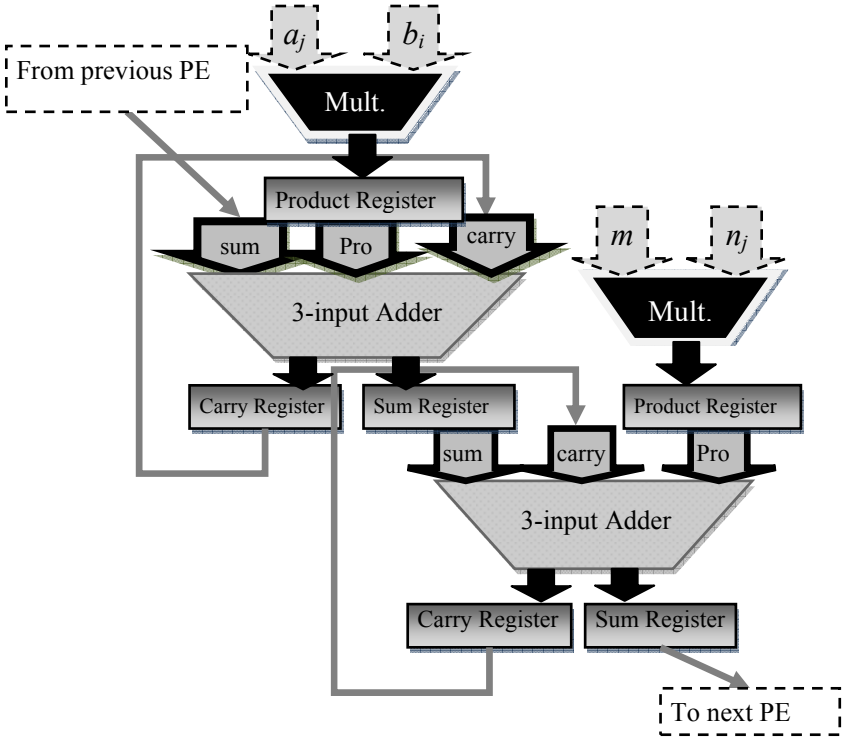


Figure 8. The Structure of a Processing Element

As only one word per cycle can be requested from each Block-RAM, only the first PE directly receives data from Block-RAMs, and only the last PE writes words t_i to the Block-RAM. All PEs forward "used input variables" (a_j and n_j) and the *sum* to the next PE to exploit data reuse and simplify connection network. Figure 8 shows the inner structure of a processing element, which mainly consists of two multipliers, two adders and six registers.

3.2.3 Parametric Design

We can adjust the multiplier to meet the application requirements or to utilize a given FPGA device efficiently by changing the following three parameters at the compile time:

(1) Number of PEs (*PE*): Total number of PEs is the main area vs. performance trade-off metric. The proposed design must have at least two PEs, because the first and last processing elements are hardwired to RAM. In other words, total number of block multipliers must be at least four. The upper bound for PEs is determined with the amount of hardwired multipliers of the target FPGA, which is 10 in our case (i.e. 20 block multipliers in total).

(2) Radix (*R*): This parameter determines the bit length of the hardwired multipliers and adders shown in Figure 8. As the radix closely relates to the maximum combinational path delay in the adder design, it has a direct effect on the frequency. This parameter must be adjustable to take full advantage of the block multipliers in a given device to achieve the best timing performance.

(3) Number of Words (*s*): The radix and the number of words in each operand together determine the bit-length of the operands; for instance, for 2048-bit operands and radix=16, the number of words is 128. The number of words determines also the depth of the Block-RAM.

3.3 Simulation Results

The clock cycles required for one multiplication heavily depends on the number of PEs. More PEs result in faster designs as expected. However, the multiplier utilization decreases when the number of PEs increases. Similarly, using longer words also has a negative effect on the frequency due to longer carry chains in adders used in PEs.

Table 7 shows the exact cycle count for one modular multiplication including data load time from the Block-RAM. The multiplication circuit has the following timings (see Figure 9 in Appendix for waveform):

- After start signal is asserted, it takes 9 cycles for the first PE to yield the first word of the result.
- The number of clock cycles spent between the appearances of the first word of the results in consecutive PEs is 9.

	The Number of PEs					
Bitlength-#words	**2**	**4**	**5**	**6**	**8**	**10**
4080* **(240 words)**	30256	15154	12139	10132	7630	6136
2040 (120 words)	7936	3994	3211	2692	2050	1672
1020 (60 words)	2176	1114	907	772	638	630
510 (30 words)	646	352	330	326	322	318

Table 7. Clock Cycles Required for One Multiplication (radix=2^{17})

*The key length must be a multiple of 17 bits (because our multipliers are 17 bits long) and (s/PE) ratio must be an integer to take the full advantage of block multipliers.

The overall cycle count can be approximated (with error margin less than 5%) using the following formula:

$$CC = \max((14+s), (12+PE*9)) * (s/PE) \approx \frac{s\times(14+s)}{PE} \text{ (for large s)}$$

where *CC*, *PE*, and *s* stand for the total clock cycles, the number of processing elements, and the number of words, respectively.

As indicated in [5], the CIOS method requires $2s^2+s$ word multiplications. If there were no data dependencies, the required clock cycles would be $(2s^2+s)/(2*PE)$. The PE utilization is over 85% for 2040-bit or larger operands which can be seen in Table 8.

	The Number of PEs					
Bitlength-words	**2**	**4**	**5**	**6**	**8**	**10**
4080 (240 words)	95.4	95.2	95.1	94.9	94.6	94.1
2040 (120 words)	91.1	90.5	90.1	89.5	88.2	86.5
1020 (60 words)	83.4	81.5	80.0	78.4	71.1	57.6
510 (30 words)	70.8	65.0	55.5	46.8	35.5	28.8

Table 8. Utilization Ratios (%)

3.4 Synthesis Results

In this section, we provide the synthesis results summarizing the resource usage and timing performance of the multiplier and exponentiation circuit for the target device. We implemented our design using Verilog and simulated with ModelSim [30] tool.

3.4.1 Setup and Synthesis Configuration

We use XST (Xilinx Synthesis Tool) from Xilinx ISE v9.1 package with following optimizations:

(1) Register Balancing

(2) Equivalent register removal

(3) Optimization Effort: High

(4) Optimization Priority: Speed

(5) Maximum Fan-out: 17

The target device is Xilinx 3s500e-4FG320C whose properties are given in [18].

3.4.2 Synthesis Results

Table 9 shows the resource usage for different number of processing elements from 2 to 10. As can be observed in the table, the resource usage is modest even for the maximum configuration with the largest number of processing elements.

	PE=2	**PE=4**	**PE=5**	**PE=6**	**PE=8**	**PE=10**	**Total**
Slices	679	1260	1553	1838	2435	3028	**4656**
FF	809	1505	1854	2199	2901	3602	**9312**
LUT	1180	2232	2760	3292	4353	5426	**9312**
Block RAM	4	4	4	4	4	4	**20**
Multiplier	4	8	10	12	16	20	**20**

Table 9. Synthesis Results for Multiplication Core*

For 1020-bit or longer operands, a multiplication engine with 4, 5 and 6 PEs offer the lowest time-area product (Table 10). The 510-bit key is obsolete; however, we included it for efficiency comparison. With 5 PEs per multiplication core, we can fit two cores into the same FPGA, which takes full advantage of the parallelism in Algorithm 3.

	The Number of PEs					
Bitlength-words	**2**	**4**	**5**	**6**	**8**	**10**
4080 (240 words)	1.1058	1.0277	1.0147	1.0023	**1.0000**	**1.0000**
2040 (120 words)	1.0891	1.0171	1.0078	**1.0000**	1.0089	1.0232
1020 (60 words)	1.0526	**1.0000**	1.0035	1.0109	1.1068	1.3591
510 (30 words)	**1.0000**	1.0111	1.1684	1.366	1.7875	2.1952

Table 10. Time Area Products: The values are normalized to the smallest in the same row.

* radix $=2^{17}$, s=120 (2040 bit)

	SPA protected*	SPA+DPA Protected†
Slices	3799 (81 %)	3899 (83 %)
FF	4416 (47 %)	4493 (48 %)
LUT	6750 (72 %)	6931 (74 %)
Block Ram	14 (70 %)	16 (80 %)
Multipliers	20 (100 %)	20 (100 %)
Frequency	119 MHz	119 MHz
Clock Cycles (max)	929 519	946 127
Max Ex Time	7.81 ms	7.95 ms

Table 11. Synthesis Results for 1020-bit exponentiation circuit (radix = 2^{17} and s = 60)

The exponentiation circuit (5 PE x 2) with and without DPA countermeasure are synthesized with speed optimization and the results are illustrated in Table 11. The area consumption stays approximately the same for larger bit-lengths and so does the frequency. Second circuit has a $(1/s \times 100)$ percent cycle overhead due to DPA protection.

3.5 Performance Analysis

3.5.1 Clock Cycle Comparison

In this section, we provide a comparative analysis of the proposed design with respect to other designs synthesized for different FPGA technologies in literature. Table 12 shows the number of clock cycles required for one 1024-bit modular multiplication via different Montgomery multipliers.

* Montgomery Powering Ladder (Algorithm 3) is used as SPA protection.

† Exponent blinding is used for DPA protection.

Design	Implementation	Time Complexity	Clock Cycles
[17]	(R-2), pipelined*	2× (# *multipliers*+5)	134 (*mult*=62)
[19]	Radix-4	$N/2+(N/32)\text{x}1.5$	560
Prop.	R-2^{17}, pipelined	$\max((14+s), (12+9PE))\times(s/PE)$	**907 (*PE*=5)**
[14]	R-2, pipelined	N	1024
[10]	Radix-2	$N+1$	1025
[11]	R-2, pipelined	$N+3$	1027
[7],[19]	Radix-2	$N+2+N/32$	1058
[8]	R-2, pipelined	$(N/(w_1 \times PE)).\ max\ (PE\ ,\ N/w_2)$	1062
[6]	R-4, pipelined	$\left\lceil \frac{N}{2NS} \right\rceil \left(\left\lceil \frac{N}{w} \right\rceil + 1 \right) + 2NS\ ,\quad if \left\lceil \frac{N}{w} \right\rceil > 2NS$ $\left\lceil \frac{N}{2NS} \right\rceil (2NS+1) + \left\lceil \frac{N}{w} \right\rceil + 1\ ,\quad otherwise$	~1104 (w=64, NS=8)
[9]	R-2, pipelined	$2N+N/w-1\ ,$ if $N/w+1 < 2.PE$ $(N/PE)(N/w+1)+2(PE-1),$ otherwise	~2080 (w=32, p = 16)
[15]	R-2, pipelined	$3\times N$	3072

Table 12. Time Complexities: N is the modulus bit-length and R stands for the radix.

In [19], we have two circuits, one is based on conventional radix-2 implementation which is designed to simulate [7] on the same FPGA device, the other circuit is based on radix-4. Both designs use distributed RAM as the main storage element and are non-pipelined. Although the design in [17] is the fastest in Table 12, it cannot fit in our target FPGA, Xilinx Spartan 3E-500, in that configuration due to its excessive use of multipliers.

3.5.2 Execution Time Comparison

Table 13 summarizes the resource usage and performance of various FPGA designs and the proposed one. Although the proposed design is not the fastest circuit, its execution speed outperforms many others; moreover, it performs the best in terms of time area product.

* For pipelined designs, we select optimum (both for area and speed) values for w and p. (w is bit length of a word and PE (or NS) is the number of processing elements or pipeline stages)

Design	Technology	Freq (MHz)	Area	Ex. Time (µs)
[17]	Xilinx xc2v3000-6	90.11	N/A	1.49
[19] radix-4	Xilinx xc2v6000	132.4	8328 slices	4.23
Proposed (1020 bit)	**Xilinx xc3s500e-4FG320C**	**119**	**1553 slices + 10 multipliers**	**7.62**
[14]*	FPGA (?)	129.1	3611 slices	7.93
[19] radix-2	Xilinx xc2v6000	137.2	6282 slices	8.21
[10]	Xilinx xc2v3000	75.23	11617 slices	13.45
[11]	Xilinx xcv1000	~55	5058 slices	18.67
[15]	Xilinx V812E-BG-560	~96	5706 slices	32.12

Table 13. Execution Times for 1024-bit modular multiplication

We do not have entire performance and area details concerning the multiplication units in designs [3, 16, 17]; however, the exponentiation timings and areas are available. Our exponentiation engine has DPA and SPA protection, which the other designs lack and our execution time is fixed for a given bit-length.

* The authors in [14] use pre-computed values, but the pre-computation unit is not included in the multiplier (it is a part of the exponentiation circuitry)

Design	Technology	Frequency	Area	Ex. Time (ms)
[3]	Xilinx xc4vfx-10sf363	~200/400*	3937 slices + 17 DSP48	1.71 (max)
[17]	Xilinx xc2v3000-6	90.11	14334 slices + 62 multipliers	2.33 (avg.)
Proposed (1020 bit)	**Xilinx xc3s500e**	**119**	**3899 slices + 20 multipliers**	**7.95 (max)**
[19] radix-4	Xilinx xc2v6000	132.4	8328 slices	8.66 (max)
[22]	Xilinx xc3s4000	66	18247 slices+ 66 multipliers	11.1 (?)
[16]	Xilinx xc40250xv	45.66	6633 slices	11.95 (max)
[19] radix-2	Xilinx xc2v6000	137.2	6282 slices	16.8 (max)

Table 14. Execution Engine Performance: 1024-bit Exponentiation Results

Our foremost design goal is not achieving the best timing but the best time-area product on an inexpensive FPGA. This gap in performances can be attributed to the following factors favoring the designs in [3] and [17]:

(1) More advanced (and expensive) FPGA,
(2) More resource usage,
(3) Higher clock frequency (favoring only [3]),
(4) Powerful DSP cells (favoring only [3]),
(5) Special acceleration techniques† used for exponentiation.

Considering that the proposed circuit is intended for a low-end device, the achieved exponentiation speed, which is so far the *record* for a very low-price FPGA device to best of our knowledge, and is satisfactory for many applications. In Table 13 and Table 14, the designs are mapped onto FPGA's with different speed grades and features; e.g., the multiplier in [3] uses built-in DSP cells, which are available neither in our target device nor in many other FPGA devices. In this work, we try to use the maximum potential available on one of the smallest FPGAs; therefore, the time-area product is the vital criterion for us.

* The control unit is running at 200 MHz, while DSP48 cells (data path) are running at 400 MHz.

† [3] uses sliding window mechanism.

We cannot directly use execution times for comparison purposes (because of the technological differences), instead we can use "total clock cycles required for one modular multiplication" as the performance indicator. Table 15 shows that the proposed design achieves the best {time×area} metric, which is an indication of good design and high utilization of the target device.

Design	Area (slices)	Clock Cycles	Time × Area
[15]	5706	3072	12.607
[10]	11617	1025	8.564
[19] radix-2	6282	1058	4.780
[11]	5058	1027	3.736
[19] radix-4	8330	560	3.354
[14]	3611	1024	2.659
Proposed	**1553*(3453)**	**907**	**1.000 (2.223)**

Table 15. Time-Area products normalized to proposed implementation†

3.6 Compatibility Problems

As we use 17 bit x 17 bit multipliers in the design to take the full advantage of given features of the FPGA chip, the implemented bit lengths are smaller than the widely employed ones that are the powers of 2 (e.g. 512-bit, 1024-bit, 2048-bit, etc). The security level provided by a 1020-bit implementation is approximately the same with 1024-bit implementation, however there can be compatibility problems between 1024 and 1020 bit circuits in practical world (same case with 2048 bit and 2040-bit implementations). Therefore, we also include a table (Table 16) showing the required time for compatible versions of our implementations at the expense of some clock cycles. The number of words in each compatible version is one more than the previously mentioned designs; however, there will be no change in the frequency and the area at all. The average slowdown ratio is 3.6 %.

* The hardwired multipliers (we use 10 multipliers here [only one multiplication engine]) are not included in this area value. The value in parenthesis is the total area including the multipliers.

† for ≈1024 bit multiplication

	The Number of PEs					
Bitlength -#words	**2**	**4**	**5**	**6**	**8**	**10**
4097 (241 words)	30620	15440	12404	10380	7850	6332
2057 (121 words)	8120	4130	3332	2800	2135	1736
1037 (61 words)	2270	1175	956	810	648	644
527 (31 words)	695	369	344	340	332	332

Table 16. The Required Clock Cycles for Compatible Versions

3.7 Conclusion and Future Work

We designed a fast, efficient and parameterized multiplier and a secure exponentiation circuit for simple FPGA devices in the price range of $2-9 US*. This price range is at least one order of magnitude less than other devices used in previous works, where the primary purpose is to achieve the fastest time in modular exponentiation. It is true that time performance is always of an important concern; however, the price of the device used for realization is also an issue in many applications and there is not much work in this direction. We intended to fill this gap with our design, which achieves the best time-area product to the best of our knowledge in this category.

Our target technology, Xilinx Spartan 3E-500, is a cost effective solution in many aspects, especially the use of the 90nm technology significantly reduces the die size, cost and the total power consumption, while increasing the frequency, and therefore it is one of the best choices for practical applications, where the manufacturing cost is the primary concern.

The proposed multiplier is parametric, and therefore can be used for virtually any bit-length, where the upper limit on precision is dictated only by the capacity of Block-RAM available on the device†. However, since the most popular public key cryptosystem nowadays is RSA, we focused on the designs with precisions of 1020-bit and 2040-bit; the latter precision will be favored over the former in the near future due to increased security concerns. Our design completes one 1020-bit and 2040-bit

* The prices are from the year 2006
http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/spartan3e_fpgas/index.htm

† The area and frequency of our circuit with longer operands stay approximately the same.

modular multiplications* in *7.62 μs* and *27.0 μs,* respectively with approximately the same device usage. The timing performance achieved for multiplication is either comparable or superior to most of the other designs in the literature despite the low resources available on the target device. In addition, our design has the lowest {time × area} product among the other multipliers.

We have also achieved to fit the exponentiation circuit (additional control unit) into the same device. Few designs in literature *can* outperform our design *only* by using more resources, better and expensive devices, and acceleration techniques for exponentiation. From practical point of view, our exponentiation circuit also resists against all known side-channel attacks (namely SPA, DPA, fault attacks and (*n-1*) attacks) with minimal overhead.

As future work, we plan to design an improved exponentiation circuit that utilizes acceleration techniques such as the sliding window mechanism and bit encoding schemes to reduce the total execution time of modular exponentiation. Moreover, we will consider implementing our design to alternative FPGA's such as Xilinx Spartan 3A and 3AN series that have DSP units.

* With a multiplication engine that utilizes half of the device (i.e. 5 PEs which use 10 multipliers)

4 Summary of Contributions

In this thesis, we have presented two designs that have different goals. In the first design, we tried to maximize the speed of 2048-bit modular multiplication for hardware platforms. We adapted the well-known radix-2 Montgomery algorithm and obtained following results with the use of radix-4:

- While the frequency of the circuit stayed approximately the same with respect to our reference radix-2 core, the execution time for 2048-bit modular multiplication improved significantly (82% reduction in FPGA implementation), because the cycle count is approximately halved with the use of radix-4 datapath. Moreover, our circuit outperformed previous works significantly in terms of execution time.
- A major improvement was also achieved in terms of time area product, which decreased by 37% and 41% for FPGA and ASIC designs respectively in comparison to the reference core.

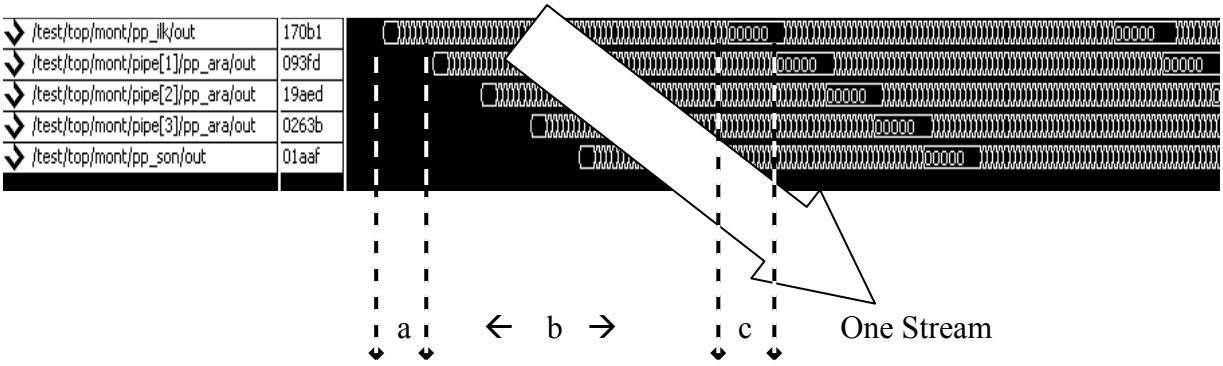
In the second design, we optimized our circuit according to the resources on our target FPGA, Xilinx Spartan 3E-500, which uses *90 nm* technology. This technology is advantageous over previous generations; because faster clock speeds can be obtained while being cost effective. Our contribution in this design can be summarized as follows:

- As the most popular PKC is RSA nowadays, we optimized our circuit for higher bit-lengths (for operands greater than 512 bits). Our implementation can be used in other PKC algorithms as is, except for ECC, (where a slight modification of the design is needed) because ECC utilizes much shorter key lengths.
- We designed a high-speed multiplier and exponentiation circuit on an inexpensive FPGA by utilizing its hardwired multipliers, which are becoming more and more common in reconfigurable devices and we showed that 2040-bit exponentiation circuit can easily fit on such FPGAs.

- We provided an efficient modification of one of the best software algorithms for Montgomery multiplication to take advantage of simultaneously operating multipliers.
- We showed that the {time × area} metric will shrink considerably by the use of the dedicated multipliers on FPGA.
- We arranged the multipliers in a pipelined fashion to increase the device utilization and clock frequency. This arrangement also rendered a parametric design that can be used to perform multiplications and exponentiations up to virtually any bit-length as long as the memory resources sufficed.
- We provided a parametric design for modular multiplier, which could be implemented on an FPGA that has as low as four block multipliers where the block size is also adjustable.
- By the use of dual-port Block RAMs, we could overlap the phases of the algorithm and therefore the execution was accelerated considerably, while maintaining low area consumption.
- We also implemented countermeasures to all known side-channel and fault-induction attacks and demonstrated that they were affordable on a very modest FPGA device.

Appendix

Figure 9. Waveform of One Stream



- a = Start to start time (9 cycles)
- a + b = Time required to finish one iteration of loop *i* in Algorithm 5. (*s* + 3 cycle)
- c = idle period of one multiplier

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signature and Public-key Cryptosystems*," Comm. ACM, v01.2 1, pp. 120-126*, 1978.

[2] P. L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation, vol. 44, pp.519-521,* 1985.

[3] D. Suzuki, "How to Maximize the Potential of FPGA resources for Modular Exponentiation", *CHES 2007, LNCS 4727, pp. 272-288*, 2007

[4] C. McIvor, M. McLoone, J.V. McCanny. "FPGA Montgomery Multiplier Architectures - a Comparison" in Field-Programmable Custom Computing Machines, 2004. *FCCM 2004. 12th Annual IEEE Symposium on Volume , Issue , Page(s): 279 – 282*. April 2004

[5] Ç. K. Koc, T. Acar, B.S. Kaliski: "Analyzing and Comparing Montgomery Multiplication Algorithms". *IEEE Micro, Vol. 16, No. 3, pp. 26-33,* June 1996.

[6] L. A. Tawalbeh, "Radix-4 ASIC Design of a Scalable Montgomery Modular Multiplier using Encoding Techniques", Master Thesis, Oragon State University, USA 2000.

[7] Y. S. Kim, W. S. Kang, and J. R. Choi, "Implementation of 1024-bit Modular Processor for RSA Cryptosystem*", The Second IEEE Asia Pacific Conference on ASICs*, 2000.

[8] L. Batina, G. Muurling, "Montgomery in Practice: How to do it more efficiently in hardware", *Cryptographers' Track RSA Conference 2002,* San Jose, USA

[9] A. F. Tenca, and C. K. Koc, "A Scalable Architecture for Modular Multiplication based on Montgomery's Algorithm", *IEEE Transaction on Computers, vol.52 no.9,* 2003

[10] C. McIvor, M. Mcloone, J. N. McCanny, A. Daly, W. Marnane, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures", *37th Annual Asilomar Conference on Signals, Systems and Computers,* California , 2003

[11] A. Daly and W. Marnane, "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *in proc. of 10th International symposium on FPGA's*, 2002

[12] W. Diffie and M. E. Hellman, "New Directions in Cryptography," IEEE Trans. Info. Theory, vol. IT-22, Nov. 1976, pp. 644–54.

[13] N. Koblitz, Elliptic curve cryptosystems, *in Mathematics of Computation 48*, pp. *203–209,* 1987.

[14] P. Fournaris, O. Koufopavlou, "A New RSA Encryption Architecture and Hardware Implementation based on Optimized Montgomery Multiplication*" in proceedings of 2005 IEEE International Symposium on Circuits and Systems (ISCAS 2005),* Kobe, May 23 -26, Japan, 2005.

[15] S. B. Ors, L. Batina, B. Preneel and J. Vandawalle, “Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array”, *International Parallel and Distributed processing symposium (IPDPS ’03),* 2003

[16] T. Blum, C. Paar, “High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware”, *IEEE Transaction on Computers 50(7), 759-764* (2001).

[17] S. H. Tang, K. S. Tsui, P. H. W. Leong, “Modular Exponentiation using Parallel Multipliers” , *Proc of the 2003 IEEE International Conference on Field Programmable Technology (FTP 2003), pp. 52-59* (2003)

[18] Xilinx, Inc.: http://www.xilinx.com, “Xilinx Spartan 3E-500 Data Sheets”.

[19] E. Oksuzoglu, E. Savas, “A Fast and Efficient Hardware Implementation of 2048-bit Radix-4 Modular Multiplication Circuit for Public Key Cryptosystems”, *submitted to JCSC*, 2007

[20] M. Joye, S.M. Yen, “The Montgomery Powering Ladder”, *Cryptographic Hardware and Embedded Systems – CHES 2002*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 291–302, Springer-Verlag, 2003

[21] S.M. Yen, W.C. Lien, S. Moon, and J. Ha, “Power Analysis by Exploiting Chosen Message and Internal Collisions – Vulnerability of Checking Mechanism for RSA-Decryption”, *Mycrypt 2005, LNCS 3715,* pp. 183–195, 2005

[22] N. Mentens, K. Sakiyama, L.Batina, I. Verbauwhede, B. Preneel, “FPGA-Oriented Secure Data Path Design: Implementation of a Public Key Coprocessor”, *16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, IEEE, pp. 133-138, 2006.

[23] C. Giraud, “An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis”, IEEE Trans. Computers, (55): 9, pages 1116-1120, 2006.

[24] A.F Tenca, G. Todorov and C.K. Koc, “High-radix Design of a Scalable Modular Multiplier,” in *Cryptographic Hardware and Embedded Systems* - CHESS 2001, C.K Koc and C. Paar, Eds.2001, Lecture Notes in Computer Science, No. 1717, pp. 186-206, Springer, Berlin, Germany

[25] L. A. Tawalbeh, A. F. Tenca and C. K. Koc, “A Radix-4 Design of a Scalable Modular Multiplier with Recoding Techniques”: islab.oregonstate.edu/papers/j66radix.pdf, 2002

[26] A.D.Booth, “A Signed Binary Multiplication Technique,” *Q.J.Mech. Appl. Math,* Vol.4, no.2, pp236-240, 1951

[27] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", *IEEE Transactions on Information Theory, v. IT-31, n. 4, 1985, pp469–472* or *CRYPTO 84, pp10–18*, Springer-Verlag

[28] NIST, “Digital Signature Stantard (DSS)”, *FIPS PUB186-2*, 2000.

[29] P. Paillier, “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”, *Eurocrypt 1999, pp223-238*.

[30] ModelSim Simulation Tool. *Mentor Graphics Corporation*, http://www.model.com/