DESIGN AND IMPLEMENTATION OF AN EFFICIENT RSA CRYPTOPROCESSOR

By

DHINESH.R

Reg. No. 1020106004

of

KUMARAGURU COLLEGE OF TECHNOLOGY

(An Autonomous Institution affiliated to Anna University, Coimbatore)

COIMBATORE - 641006

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements
for the award of the degree
of

MASTER OF ENGINEERING
IN
APPLIED ELECTRONICS
MAY 2012

BONAFIDE CERTIFICATE

Certified that this project report entitled "**DESIGN AND IMPLEMENTATION OF AN EFFICIENT RSA CRPTOPROCESSOR**" is the bonafide work of **Dhinesh.R [Reg. no. 1020106004]** who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Project Guide	Head of the Department
Ms.M.Alagumeenaakshi.	Dr. Rajeswari Mariappan

The candidate with university **Register no.1020106004** is examined by us in the project viva-voce examination held on …………………….

Internal Examiner	External Examiner

ACKNOWLEDGEMENT

First I would like to express my praise and gratitude to the Lord, who has showered his grace and blessing enabling me to complete this project in an excellent manner. He has made all things beautiful in this time.

I express my profound gratitude to our chairman **Padmabhusan Arutselvar Dr.N.Mahalingam B.Sc.,F.I.E.** for giving this opportunity to pursue this course.

I express my sincere thanks to our beloved Director **Dr.J.Shanmugam**, Kumaraguru College of Technology, I thank for his kind support and for providing necessary facilities to carry out the work.

At this pleasing moment of having successfully completed the project work, I wish to acknowledge my sincere gratitude and heartfelt thanks to our beloved principal **Prof.Ramachandran,** for having given me the adequate support and opportunity for completing this project work successfully.

I express my sincere thanks to **Dr.Rajeswari Mariappan Ph.D.,** the ever active, Head of the Department of Electronics and Communication Engineering, who is rendering us help all the time throughout this project

In particular, I wish to thank with everlasting gratitude to the project coordinator **Ms.R.HemaLatha Asst.Prof(SRG),**Department of Electronics and Communication Engineering for her expert counseling and guidance to make this project to a great deal of success. Her careful supervision ensured me in attaining perfection of work.

I extend my heartfelt thanks to my internal guide **Ms.M.Alagumeenaakshi, Asst Prof(SRG).**for her ideas and suggestion, which have been very helpful for the completion of this project work.

Last, but not the least, I would like to express my gratitude to my family members, friends and to all my staff members of Electronics and Communication Engineering department for their encouragement and support throughout the course of this project.

ABSTRACT

In this project a hardware implementation of the RSA algorithm for public-key cryptography has been presented. Basically, the RSA algorithm entails a modular exponentiation operation on large integers, which is considerably time consuming to implement. Two types of architecture based on Montgomery's technique were proposed and efficient comparison is made with respect to speed and area. To this end, a trade-off between processing speed and space has been studied in the implementation of modular exponentiation using Montgomery's technique. Comparatively smaller amount of space is occupied in the FPGA due to its reusability. The architecture is synthesized and simulated using VHDL code and implemented using Xilinx.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

RSA	**-------**	**Rivest Shamir and Adleman**
VHDL	**-------**	**Very High Speed Integrated Circuit Hardware Description Language**
FPGA	**-------**	**Field-Programmable Gate Arrays**
ADC	**-------**	**Analog to Digital Converter**
ASIC	**-------**	**Application Specific Integrated Circuits**
SOC	**-------**	**System-On-Chip**

CHAPTER 1

INTRODUCTION

The network security problem for an increasing traffic of Internet transactions has been emerged and there have been lots of researches in cryptographic algorithms. Cryptography plays an important role in the information security in the modern communication and computer networks. As the most widely used public-key cryptography algorithm, RSA algorithm was proposed by R. L. Rivest, A. Shamir, L.Adleman in 1978. Its safety depends on the difficulty of big integer factorization. The RSA operation is the modular exponentiation with long number in essence, which can be calculated by the iteration of modular multiplication. In 1980, the Montgomery algorithm was proposed to enable people to see the possibility to realize RSA algorithm on the hardware. RSA system is one among them and widely used today. The system requires fast modular multiplication of integer numbers containing several hundred bits. The algorithm discussed here is that of providing hardware to perform modular multiplication using the algorithm proposed by P.L.Montgomery and further developed by the S.E.Eldridge and Kaliski. As cryptography becomes more widespread, fast cryptographic implementations are becoming important.

Experience with hardware tools has shown that speed often cannot fully be realized unless all cryptographic methods of interest are implemented in hardware. VHDL language provides good design tools for constructing systolic arrays as well as graphical ones. Even though RSA system is a perfect solution, it provide reasonable security feature over the Internet with hundred bits of encryption standard. Fast processing of the modular multiplication is one of the most important key issues on the RSA technology. As data size demanded over the network grows larger and larger every year, faster processors as well as sophisticated algorithm which is more effective for the hardware and software are always in high demand. One of the main objectives of cryptography consists of providing *confidentiality*, which is a service used to keep secret publicly available information from all but those authorized to access it. There exist many ways to providing secrecy. They range from physical protection to mathematical solutions, which render the data unintelligible.

Although the RSA algorithm has high security, due to the complication of its essential modular exponentiation of large numbers, the speed of calculation turns out to be the biggest bottleneck and technical problem. In recent years, the improvements in terms of algorithm and the architecture of the crypto-processor have diminished the time for calculation to a certain extent. However, it is still unsuitable to treat with the long data.

1.1. PROJECT GOAL

The essential arithmetic operation carried out in RSA algorithm is modular multiplication, which is used to calculate modular exponentiation. Modular exponentiation on numbers of hundreds of bits makes it difficult for RSA algorithm to attain output. The goal of this project is to implement two architectures based on Montgomery's Modular Multiplication algorithm and efficient comparison is made with respect to speed and area.

1.2 OVERVIEW

Montgomery modular multiplication algorithm is first implemented which can avoid range comparison which is a critical operation in traditional division in modular multiplication. Basically modular exponentiation with a large modulus is usually accomplished by performing repeated modular multiplication which is considerably time consuming. As a result the throughput rate of RSA cryptosystem will be entirely dependent on speed of modular multiplication and number of performed modular multiplications. To speed up the process the above implemented Montgomery modular multiplication is employed in exponentiation algorithm thus increasing performance of RSA cryptosystem.

1.3 SOFTWARE USED

- Model Sim 6.3f
- Xilinx ISE 8.1i

1.4 ORGANIZATION OF THE REPORT

- **Chapter 2** discusses about the general RSA algorithm overview.
- **Chapter 3** discusses about modular arithmetic carried in RSA algorithm.
- **Chapter 4** discusses the montgomery modular multiplication algorithm.
- **Chapter 5** discusses the hardware implementation of two architectures algorithm.
- **Chapter 6** discusses the simulation results.
- **Chapter 7** shows the conclusion of the project.

CHAPTER 2
RSA ALGORITHM

2.1 Introduction

This algorithm is based on the difficulty of factorizing large numbers that have 2 and only 2 factors (Prime numbers). The system works on a public and private key system. The public key is made available to everyone. With this key a user can encrypt data but cannot decrypt it, the only person who can decrypt it is the one who possesses the private key. It is theoretically possible but extremely difficult to generate the private key from the public key, this makes the RSA algorithm a very popular choice in data encryption.

2.2 Algorithm

- Choose two large primes p and q.
- Compute n = p×q.
- Calculate φ(n) = (p-1) ×(q-1).
- Select the public exponent e Є {1,2, . . . , φ(n)−1}.
- Plain text is converted in to cipher text by $C = M^e \ mod\ n$
- Compute the private key d such that d×e ≡ mod φ(n).
- Plain text is recovered by $M = C^d\ mod\ n$

First of all, two large distinct prime numbers p and q must be generated. The product of these, we call n is a component of the public key. It must be large enough such that the numbers p and q cannot be extracted from it 512 bits at least i.e. numbers greater than 10. We then generate the encryption key e which must be co-prime to the number φ(n) = (p - 1)(q - 1).We then create the decryption key d such that d*e mod φ(n) = 1. We now have both the public and private keys.

2.3 Encryption

We let y = E(x) be the encryption function where x is an integer and y is the encrypted form of x , $y = x^e$ mod n

2.4 Decryption

We let X = D(y) be the decryption function where y is an encrypted integer and X is the decrypted form of y, $X = y^d$ mod n

CHAPTER 3
MODULAR ARITHMETIC IN RSA ALGORITHM

3.1 Introduction

Two numbers are the modular inverse of each other if their product equals 1.

$$\text{If (AB) mod N} = 1 \text{ mod N, then } B = A^{-1}$$

The advantage of modular arithmetic is that it is no need to manipulate any huge numbers. The size of the number is depends on the modular base.

$$\text{AB mod N} = ((\text{A mod N}) (\text{B mod N})) \text{ mod N},$$

$$\text{ABC mod N} = ((\text{AB mod N}) (\text{C mod N})) \text{ mod N},$$

$$\text{ABCD mod N} = ((\text{AB mod N}) (\text{CD mod N})) \text{ mod N, and etc..}$$

This algorithm never looks at any product larger than $(N-1)^2$.

Above multiplication algorithm can be directly applied to compute

$$C^m \text{ mod N, where C=A=B=D=E…}$$

If C^m mod N = 1 mod N, then C^{m+1} mod N = C.

With a number C and some power *d* modulo N,

$$A = C^d \text{ mod N}.$$

By taking the result of the above exponentiation, A, it is possible to raise it to some other power *e*,

$$A^e \text{ mod N} = (C^d)^e \text{ mod N} = C^{de} \text{ mod N}.$$

3.2 Fermat's Little Theorem.

Leonhard Euler described φ (*n*), phi-function, which is the "exponential period" modulo *n* for numbers relatively prime that means it shares only one common factor, namely 1 with *n*.

For example, φ (6) = 2 because only 1 and 5 are relatively prime with 6. φ (7) = 6 because any number less than 7 can share with 7 only 1 as a common factor and they (1,2,3,4,5 and 6) are all relatively prime with 7. Clearly this result will extend to all prime numbers. Namely, if p is prime, φ (*p*) = *p*-1. For example, if *n* = 5, a prime number, then φ (*n*) = 4. Set *a* be 3.

$$a^{\varphi(n)} \text{ mod } n = 3^4 \text{ mod } 5$$

$$3^2 \text{ mod } 5 = 4$$

$$3^3 \text{ mod } 5 = 4*3 \text{ mod } 5 = 2$$

$$3^4 \text{ mod } 5 = 2*3 \text{ mod } 5 = 1$$

Fermat's little theorem is a statement about powers in modular arithmetic in the special case where the modular base is a prime number. Pohlig and Hellman study a scheme related to RSA, where exponentiation is done modulo a prime number.

$$a^{(p-1)} = 1 \text{ mod } p$$

unless *a* is a multiple of *p* which must be a $^{\text{prime}}$ number.

Rivest, Shamir, and Adelman designed a fascinating encryption algorithm with Fermat's little theorem. Decryption is only possible for the chosen few who have extra information.

For given *d, p* and computed $A = C^d$ mod *p*, to find *e* such that A^e mod *p* = 1, we can simply find *e* such that *d*e* = φ (*p*) which equals to *p*-1. Because then

$$A^e \text{ mod } p = C^{de} \text{ mod } p = 1 \text{ mod } p.$$

For given *d, p* and computed $A = C_d$ mod *p*, to find *e* such that A^e mod *p* = C, we need to find *e* such that *de* = φ (*p*) + 1. Because then

$$A^e \text{ mod } p = C^{de} \text{ mod } p = C \text{ mod } p.$$

From the fact that

$$(\varphi(p) + 1) \text{ mod } \varphi(p) = (\varphi(p) + 1) - \varphi(p) = 1 \text{ mod } \varphi(p),$$

Therefore, finding *e* such that *de* = φ (*p*) + 1 is equivalent to finding *e* such that *de* = 1 mod φ (*p*),which is known as the modular inverse. There is a method known as extended Euclidean algorithm for computing the modular inverse.

After picking a public exponent *d* and by finding a prime *p*, make those two values public. Using the extended Euclidean algorithm, determine *e*, the inverse of the public exponent modulo φ (*p*) = *p*-1. When people want to send someone a message C, they can encrypt and produce cipher text A by computing $A = C_d$ mod *p*. To recover the plain text message, compute $C = A^e$ mod *p*. But the private key *e* is the inverse of *d* modulo *p*-1. Since *p* is public, anyone can compute *p*-1 and therefore determine *e*.

RSA algorithm solves the above problem by using an Euler's multiplicative phi-function. If *p* and *q* are relative prime, then φ (*pq*) = φ (*p*) φ (*q*). Hence, for primes *p* and *q* and *n* = *pq*,

$$\varphi(n) = (p-1)(q-1).$$

The problem is finding *e* that satisfies

$$d*e = 1 \bmod (p-1)(q-1)$$

where the pair (*n,d*) is the public key and *e* is the private key. The prime p and q must be kept secret or destroyed. To compute cipher text A from a plain text message C, find $A = C^d \bmod n$. To recover original message, compute $C = A^e \bmod n$. Only the entity that knows *e* can decrypt. Because of the relationship between *d* and *e*, the algorithm correctly recovers the original message C, since

$$A^e \bmod n = C^{de} \bmod n = C1 \bmod n = C \bmod n.$$

To know φ (*n*) one must know *p* and *q*. In other words, they must factor *n*. Multiplying big prime numbers can be a one-way function. Factoring takes a certain number of steps, and the number of steps increases sub-exponentially as the size of the number increases. Extended Euclidean algorithm can be used to find private key *e*.

Using this fact, it is natural to build the private key using two primes and the public key using their product. There is one more condition, the public exponent *d*, must be relatively prime with (*p*-1)(*q*-1) to exist a modular inverse *e*. In practice, one would generally pick *d*, the public exponent first, then find the primes *p* and *q* such that *d* is relatively prime with *(p-1)(q-1)*. There is no mathematical requirement to do so, it simply makes key pair generation a little easier. In fact, the two most popular *d*'s in use today are $F_0 = 3$ and $F_4 = 65{,}537$. The F in F_0 and F_4 stands for Pierre de Fermat.

CHAPTER 4
MONTGOMERY MODULAR ALGORITHM

4.1 Introduction

The Montgomery algorithm for modular multiplication is considered to be the fastest algorithm to compute x*y mod n in computers when the values of x, y, and n are large. In this the Montgomery algorithm for modular multiplication is described. In order to compute x*y mod n choose a positive integer, r, greater than n and relative prime to n. The value of r is usually 2^m for some positive integer m. This is because multiplication, division and modulo by r can be done by shifting or logical operations in computers.

In RSA cryptography system, it requires lots of modular multiplication. Montgomery proposed an algorithm that conducts the modular multiplication without trial division but produces some residue. This algorithm is suitable for hardware or software implementation. Let N be an integer (the modulus) and let R be an integer relatively prime to N. We represent the residue class A mod N as AR mod N and redefine modular multiplication as

$$\text{Mont\_Product}(A, B, N, R) = ABR^{-1} \bmod N.$$

It is not hard to verify that Montgomery multiplication in the new representation is isomorphic to modular multiplication in the ordinary one:

$$\text{Mont\_Product}(AR \bmod N, BR \bmod N, N, R) = (AB)R \bmod N.$$

Since AR mod N and BR mod N are both less than N, their product is less than NN that is less than RN, so it forms a legal input for Mont_Product. A drawback of the algorithm is the redundant factor of R for the desired result, (AB) mod N.

We can similarly redefine modular exponentiation as repeated Montgomery multiplication. This "Montgomery exponentiation" can be computed with all the usual modular exponentiation speedups.

4.2 Montgomery Modular Multiplication

Montgomery modular multiplication algorithm employs only simple additions, subtractions and shift operations to avoid trial division-a critical and time consuming operation in conventional modular multiplication. Let the modulus N be a k bit odd number

and the factor R is defined as 2^k mod N. The details of Montgomery modular multiplication algorithm employed to compute $A * B * R^{-1}$ mod N are stated in algorithm MM(A,B,N) in which A and B are integers smaller than N. In the algorithm the notation B[i] E {0,1} represent the ith bit of B. The value of k can be any number and when finding the value of S the loop is executed for k-1 iterations. So modular multiplications operations are carried within the loop execution and thus Montgomery algorithm can be implemented to exponentiation algorithm thus speeding up RSA process. The process that carried out in Montgomery modular multiplication algorithm is explained in stepwise procedure below.

```
   Algorithm MM (A, B, N)
//    montgomery's modular multiplication algorithm
//    Inputs: three k bit integers
        N (modulus) ,A (multiplicand),B(multiplier)
//    Outputs: S= A x B x R^-1  mod N

              Where R= 2^k mod N   and  0< S<N
              {
                   S=0;
                 for  i=0 to k-1    {
                 q=(S + A x B[ i]) mod 2;
                   S=(S + A x B[i] +q x N)/2;
                 }
                If (S> N) S= S-N;
                return  S;
              }
```

If the inputs of modular multiplication algorithm are given in binary representation, and the intermediate result is stored in carry save form, then a format conversion operation, i.e., a ripple carry addition is needed to convert the result back into binary representation. In this manner, the final result can be directly used in next modular multiplication as desired in modular exponentiation applications. All the intermediate variables are operated in carry save form to reduce critical path delay. The level of carry save addition tree can be reduced from three to two.

It can be applied to perform either square or multiplication operation in a modular exponentiation algorithm. From algorithm of modular exponentiation it is observed that the H algorithm transforms the computation of modular exponentiation into a sequence of squares and multiplications. Since the multiplicand of the multiplication operation is the value of the message in transformed domain, which is fixed within "for loop" of modular exponentiation algorithm, it can be converted into binary form right after the preprocessing step.

The modified algorithm with carry save representation and unified multiplication and square operations is stated in algorithm MM_UMS($A,B_C.B_S,N$).In the algorithm the modulus N, an odd number, has k+1 bits with a dummy bit N[K]=0.The (k+1) bit multiplicand A is less than 2N.The value of multiplier $B=B_C+B_S$ is also less than 2N with each variable of (k+1) bits. The extra factor R is redefined as 2^{k+3} mod N because one more iteration is needed.

```
Algorithm MM_U MS (A, B_C, B_S, N)

//Proposed modular multiplication algorithm
// Inputs   : four (k+1) bit variables
                N, A, B_C  and  B_S (with A and  B_C + B_S < 2N)
// Output   : S = (S_C, S_S)
                  = A x (B_C + B_S) x R^-1 mod N    // for multiplication or
                  = (B_C +B_S) x (B_C +B_S) x R^-1 mod N
                    //   for square.
                  for 0<S<2N and R=2^(K+3) mod N
{
      Ss = 0; Sc=0; carry = 0;
      B [k + 2] = B [k  + 1]=B [-1 ]=B [-2 ]=0;
       for i=0 to k +2 {
              (carry, B [i]) = B_C [i] +B_S [i] + carry;
                                                  // rebuild multiplier B [ i]
              If (mode = square) then {
                                                  // performing square
```

```
            PP = B [i] x $2^i$ + B [i -2:0] x B [i-1];
                                   //bit concatenation
             }
             else   {
                                   // performing multiplication
             PP = A[ k : 0] x B [i];
              }
            q = ($S_C$ +Ss +PP) mod 2;
            ($S_C$, $S_S$) = ($S_C$ + $S_S$+PP +q x N)/ 2;
                                    // a 4 to 2 CSA tree
              }
             Return ($S_C$, $S_S$);
}
```

In the Montgomery modular multiplication algorithm, inputs arrived are A, B_C, B_S and N and the condition applied is A, B_C + B_S should be less than 2N. The output is obtained in sum and carry form and since multiplication/ square module is carried out, the outputs for these two modules are carried out separately. For multiplication module the output is calculated as S=A x (B_C +B_S) x R^{-1} mod N and for the square module it is S=(B_C + B_S) x (B_C +B_S) x R^{-1} mod N. The condition is provided for $0<S< 2N$ and R =2^{k+3} mod N.

After bit concatenation operations and performing multiplication and square modules the output; a 4 to 2 CSA tree is arrived at the output stage of proposed modular multiplication algorithm. The control signal 'mode' used to select which operation is to be performed is not shown in the input list of algorithm. From the proposed algorithm it is concluded that the 4 to 2 CSA tree can be kept unchanged with very limited control overhead to support both multiplication and square operations and no format conversion is needed because related input variables and intermediate results are in carry save form.

A close examination of the algorithm execution may find a little delay and to remove this a modified version named as algorithm MM_UMS_P, which can be easily extended from algorithm MM_UMS by adding one more iteration step into new algorithm is introduced.

Here the control signal 'mode' is not used because delay occurring due to mode operation is eliminated in the MM_UMS_P algorithm. The algorithm for modified version is as follows.

```
Algorithm MM_UMS_P(A, $B_C$, $B_S$, N)

// The improved version of MM_UMS
// Inputs   : four (k+1) bit variables
              N, A, $B_C$ and $B_S$
// Output   : S = ($S_C$, $S_S$)
               = A x ($B_C$ + $B_S$) x $R^{-1}$ mod N    // for multiplication or
               = ($B_C$ +$B_S$) x ($B_C$ +$B_S$) x $R^{-1}$ mod N
                //   for square.
               For $0<S<2N$ and R=$2^{k+3}$ mod N
     {
     $S_S$ =0; $S_C$ =0; carry =0;
     B [k + 2]=B [k  + 1]=B [-1 ]=B [-2 ]=B [-3] = 0;
 //B [-3] =0 for pipelined operation
       for i=0 to k +3 {                  // bounded by k+3 instead of k+2
            (carry, B [i]) = $B_C$ [i] +$B_S$ [i] + carry;
                                       // rebuild multiplier B [ i]
            If (mode = square) then {
                                       // select multiplication
            PP = B [i-1] x $2^{i-1}$ + B [i -3:0] x B [i-2];
                                       // delayed bit concatenation           }
            else {                        // performing multiplication
            PP = A [k : 0] x B [i-1];
                                       //delayed partial product
              }
             q = ($S_C$ + $S_S$ +PP) mod 2;
             ($S_C$, $S_S$) = ($S_C$ + $S_S$+PP +q x N)/ 2;
                                       // a 4 to 2 CSA tree
```

```
              }
             Return ($S_C$, $S_S$);
   }
```

In improved version of MM_UMS an additional iteration step is present. Here the control signal mode is not present. Same as in MM_UMS output for multiplication and square module is obtained reducing the delay. This proposed Montgomery's modular multiplication algorithm is used to speed up the computation in proposed exponentiation algorithm.

4.3 Montgomery Modular Exponentiation

The modular exponentiation is usually accomplished by performing repeated modular multiplications. If operations are processed in Montgomery domain, then additional preprocessing and post processing steps are needed to convert operands into desired domain. First to bound an output range an extra bit of precision is added to intermediate results for precision consideration. When preprocessing steps are carried out unwanted factors are removed automatically. After the last iteration of modular multiplication operation, preprocessing is carried out. Therefore not only removing unwanted factor of the result but also make the result fall in the right range after post processing is done here.

Let M be a k bit message with its value less than the modulus N, and E denote the k_d bit exponent or key. The H algorithm of modular exponentiation is used to compute C=M^E mod N, which is summarized in algorithm ME(M,E,N) in which two factors W and R are computed in advance. The value of R is either 2^k or 2^{k+2} depending on the chosen modular multiplication algorithm.

In exponentiation algorithm in the preprocessing step the Montgomery function is called so that the computations are reduced, thus speeding up the exponentiation procedures. The value of S is computed taking the Montgomery domain but inputs are chosen such that W is calculated in advance algorithm execution is initiated where loop repeats for k_d-1 iterations. The value of k_d can be any number and depending on its value the loop execution depends on. In the final step output is computed and less time is required for computation because of

shorter critical path. Because of this algorithm induce less time to perform RSA operation and so higher performance is attained.

This investigate how to design a unified computation unit that can be applied to efficiently fulfill the square and multiplication operations for operands in carry save form. In this way a cost effective solution is derived to design modular exponentiation based on the H algorithm with almost no performance degradation.

```
Algorithm ME (M, E, N)
// H algorithm of modular exponentiation
// Inputs   : N and M (k bits), E ($k_d$ bits) and W=$R^2$ mod N
// Outputs: C=$M^E$ mod N, where $0<C<N$
             {
                M* =MM (M, W, N);           // preprocessing
                  S = MM (1, W ,N);
                 For i= $k_d$ -1 to 0{            // H algorithm
                  S= MM(S,S, N);              // Square
                  if ( E[i] =1){
                   S=MM (M*,S, N)              //Multiplication
                     }
               }
              C=MM(S,1, N);                      // post processing
                Return C;
}
```

In the preprocessing step, the Montgomery algorithm is called for with three inputs M, W and N and S is computed with Montgomery domain with inputs 1, W and N. The H algorithm is implemented inside for loop and square operations in the H algorithm of Modular exponentiation is indicated using MM with inputs S, S and N. The preprocessing M* is called inside the multiplication module which gives S whose inputs in Montgomery domain are M*, S and N. At post processing step output C is obtained which is MM(S, 1, N).

A square and multiply method has to be modified to transform the input value in to the Montgomery format, which include a R value and to transform the output from the Montgomery domain into plain format. When applying the developed algorithm MM_UMS_P to modular exponentiation only two format conversions are needed: one for preprocessing and other for post processing. The main part of modular exponentiation is free of format conversion. The following proposed modular exponentiation algorithm is denoted as algorithm ME_UMS(M,E,N).In algorithm ME_UMS, both modulus N and message M are (k + 1)bit input variables with dummy bits N[k] = M[k] = 0,the exponent E consists of k_d bits, and precomputed constants are $R=2^{k+3}$ mod N and W $=R^2$ mod N. The proposed exponentiation algorithm is as follows.

Algorithm ME_UMS(M,E,N)

```
  //  Inputs    : N and M (k + 1) bits, E ($k_d$ bits) and
                  W = $R^2$ mod N with R = $2^{k+3}$ mod N
  // Output     : C =$M^E$ mod N where 0<C< N
                {
                   (M'c,M's) =MM_UMS_P (M, 0, W, N);
                                                        // Preprocessing
                    M' =M'c + M's;                      // format conversion
                    ($S_C$, $S_S$) =MM_UMS_P (1, 0, W,N )
                     for i = $k_d$ -1 to 0 {
                           ($S_C$, $S_S$)  = MM_UMS_P (0, $S_C$, $S_S$, N);
                                                         // square
                     if (E[i] =1){
                        ($S_C$, $S_S$) = MM_UMS_P (M', $S_C$, $S_S$, N);
                                                         // multiplication
                                   }
                }

   ($S_C$, $S_S$) = MM_UMS_P (1, $S_C$, $S_S$, N);
```

```
                                                // post processing
    C= $S_C$ + $S_S$;                               // format conversion
     return C ;
   }
```

Thus output is obtained with several preprocessing and post processing steps and C = M^E mod N and with variations of M, E and N value outputs are obtained. The proposed Montgomery multiplication algorithm is employed in H algorithm and both square and multiplication operations call for the algorithm and finally output is obtained.

4.4 Implementation of Montgomery Modular Technique

Algorithm that formalize the operation of modular multiplication generally consist of two steps: one generates the product P = A-B and the other reduces this product P modulo M. 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 n clock cycles, n is the size of the operands. 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. One of the widely used algorithms for efficient modular multiplication is the Montgomery algorithm. This algorithm computes the product of two integers modulo a third one without performing division by M. It yields the reduced product using a series of additions

Let A, B and M be the multiplicand and multiplier and the modulus respectively and let n be the number of digit in their binary representation, i.e. the radix is 2. So, we denote A, B and M as follows:

$$A=\sum_{i=0}^{n-1} a_i \times 2^i,\quad B=\sum_{i=0}^{n-1} b_i \times 2^i \text{ and } M=\sum_{i=0}^{n-1} m_i \times 2^i$$

The pre-conditions of the Montgomery algorithm are as follows:

- The modulus M needs to be relatively prime to the radix, i.e. there exists no common divisor for M and the radix;
- The multiplicand and the multiplicator need to be smaller than M.

The binary representation of the operands were used here, then the modulus M has to be odd to satisfy the first pre-condition. The Montgomery algorithm uses the least significant digit of the accumulating modular partial product to determine the multiple of M to subtract. The usual multiplication order is reversed by choosing multiplier digits from least to most significant and shifting down. If R is the current modular partial product, then q is chosen so that R+q-M is a multiple of the radix r, and this is right-shifted by r positions, i.e. divided by r for use in the next iteration. So, after n iterations, the result obtained is R =A*B*r^{-n} mod M . A modified version of Montgomery algorithm is given below.

```
algorithm Montgomery(A, B, M)
      int R = 0;
  •   1: for i= 0 to n-1
  •   2: R = R + $A_i$ - B;
  •   3: if $r_o$ = 0 then
  •   4: R = R div 2
  •   5: else
  •   6: R = (R + M) div 2;
      return R;
end Montgomery.
```

In order to yield the right result, an extra Montgomery modular multiplication by the constant 2^n mod M is needed. However as the main objective of the use of Montgomery modular multiplication algorithm is to compute exponentiations, it is preferable to Montgomery pre-multiply the operands by 2^{2n} and Montgomery post multiply the result by 1 to get rid of the 2^n factor. The implementation of the Montgomery multiplication algorithm has been concentrated.

As binary representation of numbers were used, the final result has been computed using the algorithm which is illustrated below

```
algorithm ModularMult(A, B, M, n)
  •  const C := $2^n$ mod M;
  •  int R := 0;
  •  R := Montgomery(A, B, M);
return Montgomery(R, C, M);
end ModularMult.
```

CHAPTER 5
HARDWARE IMPLEMENTATION

Hardware implementation using Montgomery technique is implemented in two different architectures namely iterative multiplier and systolic multiplier.

5.1 Iterative multiplier

In this section, the architecture of the Montgomery modular multiplier is explained. It expects the operands A, B and M and it computes R = (A×B×2^{-n}) mod M. Detailed architecture of the Montgomery modular multiplier is given in Figure 5.1.

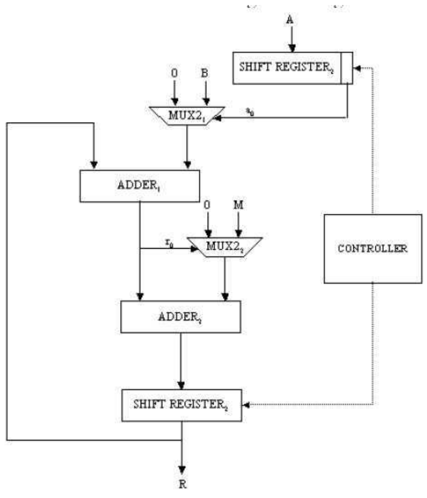


Figure 5.1:Montgomery modular architecture

It uses two multiplexers, two adders, two shift registers, three registers and a controller. The first multiplexer of the proposed architecture, i.e. MUX2_1 passes 0 or the content of register B depending on whether bit a_0 indicates 0 or 1 respectively. The second multiplexer, i.e. MUX2_2 passes 0 or the content of register M depending on whether bit r_0 indicates 0 or 1respectively. The first adder, i.e. ADDER1, delivers the sum R + A_i × B and the second adder, i.e. ADDER2, yields the sum R + M. The shift register SHIFT REGISTER1 provides the bit a_i. In each iteration i of the multiplier, this shift register is right-shifted once so that a_0 contains a_i.

The role of the controller consists of synchronizing the shifting and loading operations of the SHIFT REGISTER1 and SHIFT REGISTER2. It also controls the number of iterations that have to be performed by the multiplier. For this end, the controller uses a simple down counter. The counter is inherent to the controller.

In order to synchronize the work of the components of the architecture, the controller consists of a state machine, which has 6 states defined as follows:

- S0: Initialize of the state machine;
  Go to S1;
- S1: Load multiplicand and modulus into the corresponding registers;
  Load multiplier into shift register1;
  Go to S2;
- S2: Wait for ADDER1;
  Wait for ADDER2;
  Load multiplier into shift register2;
  Increment counter;
  Go to S3;
- S3: Enable shift register2;
  Enable shift register1;
- S4: Check the counter;
  If 0 then go to S5 else go to S2;
- S5: Halt;

Modular Multiplier Architecture

The modular multiplier yields the actual value of A×B mod M. It first computes R = A×B×2^{-n} mod M using the Montgomery modular multiplier. Then, it computes R × C mod M, where C = 2n mod M.

The modular multiplier uses a 4-to-1 multiplexer MUX4 and a register REGISTER.

- Step 0: Multiplexer MUX4 passes 0 or B. MUX2 passes A. It yields R1 = A×B×2-n mod M. The register denoted by REGISTER contains 0.
- Step 1: Multiplexer MUX4 passes 0 or R. MUX2 passes C. It yields R = R1×C mod M. The register denoted by REGISTER contains the result of the first step computation, i.e. R = A×B×2^{-n} mod M. The modular multiplier architecture is given in Figure 5.2.

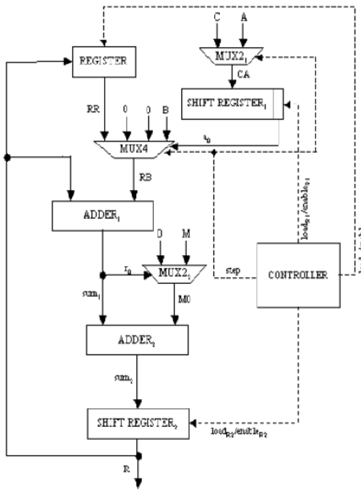


Figure 5.2:Montgomery multiplier architecture

In order to synchronize the actions of the components of the modular multiplier, the architecture uses a controller, which consists of a state machine of 10 states. The interface of CONTROLLER is shown in Figure 5.3.The modular multiplier controller does all the control that the Montgomery modular multiplier needs as described in the previous section. Furthermore, it controls the changing from step 0 to step 1, the loading of the register denoted by REGISTER.

- S0: Initialize of the state machine;
  Set step to 0; Go to S1;
- S1: Load multiplicand and modulus;
  Load multiplier into SHIFT REGISTER1; Go to S2;
- S2: Wait for adder1; Wait for ADDER2;
  Load partial result into SHIFT REGISTER2;
  Increment counter; Go to S3;
- S3: Enable SHIFT REGISTER2;
  Enable SHIFT REGISTER1; Go to S4;
- S4: Load the partial result of step 0 into REGISTER;
  Check the counter;
  If 0 then go to S5 else go to S2;
- S5: Load constant into SHIFT REGISTER1;
  Reset REGISTER;
  Set step to 1; Go to S6;
- S6: Wait for ADDER1; Wait for ADDER2;
  Load partial result into SHIFT REGISTER2;
  Increment counter; Go to S7;
- S7: Enable SHIFT REGISTER2;
  Enable SHIFT REGISTER1; Go to S8;
- S8: Check the counter;
  If 0 then go to S9 else go to S6;

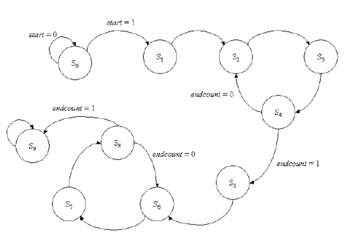


Figure 5.3: State Machine of the Controller

5.2 Systolic multiplier

The concept of systolic architecture combines a highly parallel array of identical Processing Elements or data-paths with local connections, which take external inputs and process them in a predetermined manner and in a pipelined fashion. The proposed systolic architecture is directly based on the arithmetic operations of the Montgomery Algorithm, which are performed in a numerical base 2*k*, in which the large input operands are processed in a multi-precision context containing *m* words of *k* bits.

The architecture is composed of *m* Processing Elements distributed in a one-dimensional array, where each Processing Element is responsible for the calculus involving *k* bits words of the input operands with the same index of the Processing Element. Between the Processing Elements, there is a propagation of carry signals which are the most significant bits of the arithmetic processes in each PE. The carry signals are processed as input parameters by the Processing Elements that receive them. The algorithm for systolic Montgomery multiplier which is implemented is shown below

```
**algorithm** SystolicMontgomery(A,B,M,MB)
**int** R := 0;
**bit** carry := 0, x;
0: **for** i: = 0 **to** n
1: $q_i$: = $r_0^{(i)} \oplus a_i.b_0$;
2: **for** j: = 0 **to** n
3: **switch** $a_i$, $q_i$
4: 1,1: x := $mb_i$;
5: 1,0: x := $b_i$;
6: 0,1: x := $m_i$;
7: 0,0: x := 0;
8: $r_j^{(i+1)} := r_{j+1}^{(i)} \oplus x_i \oplus$ carry;
9: carry:= $r_{j+1}^{(i)}. x_i + r_{j+1}^{(i)}$.carry +$x_i$. carry;
return R;
**end** SystolicMontgomery.
```

In the systolic architecture, the Processing Elements are designed by finite state machines. The control block communicates with the first Processing Element (PE) and with the block responsible for the quotient calculation q_i. The overall architecture for systolic multiplier is given in figure 5.4. which consists of four different cells namely

- Right Most Top Most PE.
- Basic Processing Element.
- Left Border PEs $cell_{0,j}$.
- Right Border PEs $cell_{i,0}$.

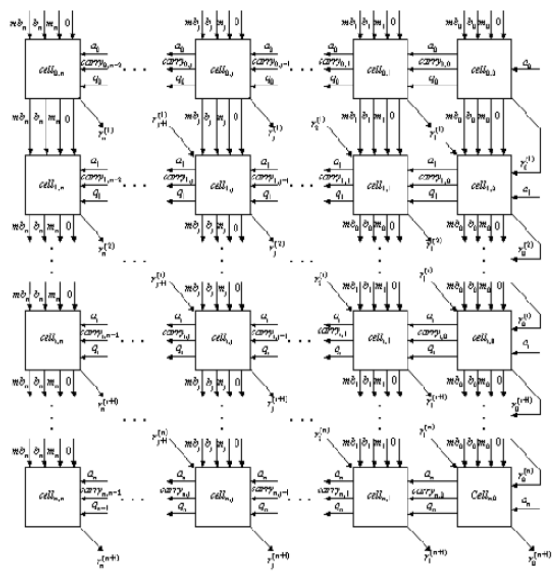


Figure 5.4: Systolic Multiplier.

5.2.1. Right Most Top Most PE.

The first Processing Element (PE) establishes communication with the control block and receives a_i and q_i words at each Montgomery Algorithm iteration. This PE differs from the other Processing Elements because it does not receive any carry signal as input and it discards the first word of the *Sum,* therefore full adder is replaced by the half adder in order to save the area. right most top most PE i.e $cell_{0,0}$ is shown in figure 5.5.

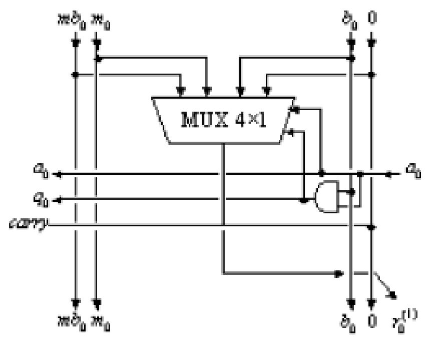


Figure 5.5: Right most top most PE i.e $cell_{0,0}$.

5.2.2. Basic Processing Element.

The other Processing Elements are different from basic PE because they have a word from the S result as output and they also transmit and receive carry signals of the multi-precision multiplications and additions. Each Processing Element is activated by the previous Processing Element when the latter finishes its calculation and sends out its carry signals, which means that the architecture works with a pipeline behavior. Figure 5.6 presents the architecture of the basic Processing Elements.

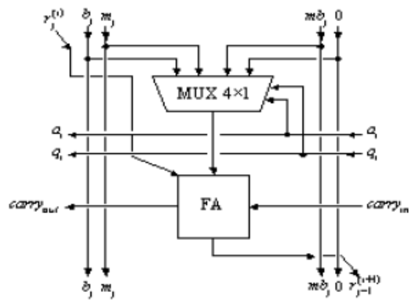


Figure 5.6: Basic PE i.e cell$_{i,j}$.

5.2.3. Left Border PEs cell$_{0,j}$.

In left border PEs sum is calculated using half adder, since previous sum to this cell (i.e initial sum) will be zero.

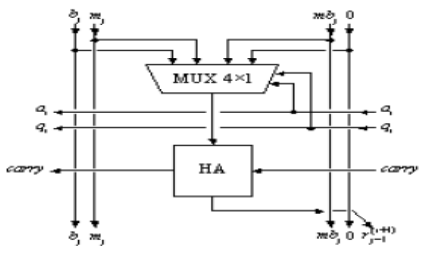


Figure 5.7: Left Border PEs cell$_{0,j}$.

Here carry of the previous cell will be summed up with one of the output from 4X1 MUX, whereas select signal for this MUX will be a_i and q_i. Figure 5.7 presents the architecture of the Left Border PEs cell$_{0,j}$.

5.2.4. Right Border PEs cell$_{i,0}$

In right border PEs initial carry will be zero, therefore sum can be calculated by using half adder instead of full adder. Here sum of the previous cell will be summed up with one of the output from 4X1 MUX, whereas select signal for 4 input MUX will be a_i and b_0 (i.e $r_0^{(i)} \oplus (a_i b_i)$) Figure 5.8 presents the architecture of the Right Border PEs cell$_{i,0}$.

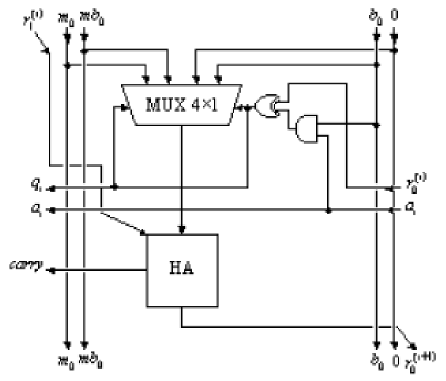


Figure 5.8: Right Border PEs cell$_{i,0}$.

Hardware implementation for two different architectures namely Iterative multiplier and Systolic multiplier is simulated using Model Sim and implemented by Xilinx. The speed of execution and the occupation of the speed is compared for both the architectures. Simulation result for those architecture is discussed in chapter 6.

CHAPTER 6
RESULTS AND DISCUSSIONS

The simulation of this project has been done using MODELSIM XE111 6.2g and XILINX ISE 8.1i.

Modelsim is a simulation tool for programming {VLSI} {ASIC}s, {FPGA}s, {CPLD}s, and {SoC}s. Modelsim provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC.

Xilinx was founded in 1984 by two semiconductor engineers, Ross Freeman and Bernard Vonderschmitt, who were both working for integrated circuit and solid-state device manufacturer Zilog Corp. The Virtex-II Pro, Virtex-4, Virtex-5, and Virtex-6 FPGA families are particularly focused on system-on-chip (SOC) designers because they include up to two embedded IBM PowerPC cores The ISE Design Suite is the central electronic design automation (EDA) product family sold by Xilinx. The ISE Design Suite features include design entry and synthesis supporting Verilog or VHDL, place-and-route (PAR), completed verification and debug using Chip Scope Pro tools, and creation of the bit files that are used to configure the chip.

The simulation results for the implementation of two architectures namely Iterative and Systolic Multiplier using Montgomery modular multiplication algorithm were shown in this section.

6.1 Simulation Result for Iterative Multiplier

6.1.1 Iterative Multiplier First Multiplication

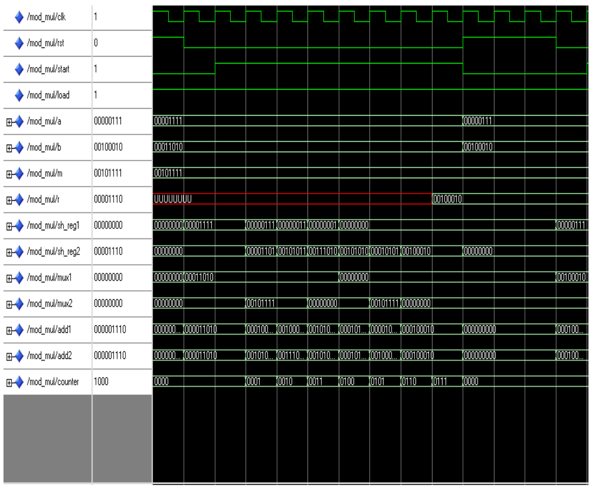


Figure 6.1.1 Simulation result for iterative multiplier behavior during the first multiplication (i)

The simulation of iterative multiplier using Montgomery algorithm with the operands a=15, b=26, and m=47 has been carried out and the behavior of the multiplier during the first modular multiplication is shown in figure 6.1.1 (here signal *step* is not set).

6.1.2 Iterative Multiplier Second Multiplication

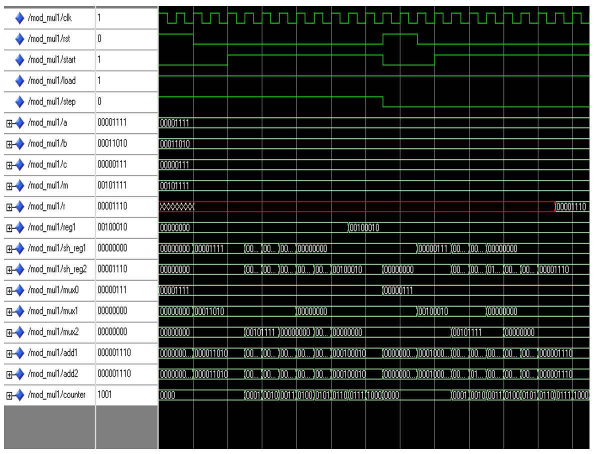


Figure 6.1.2 Simulation result for iterative multiplier behavior during the second multiplication (i)

Figure 6.1.2 shows the results of the Iterative multiplication during the second level of multiplication which finally returns r = 14 (signal *step* is set).

6.1.3 Iterative Multiplier First Multiplication

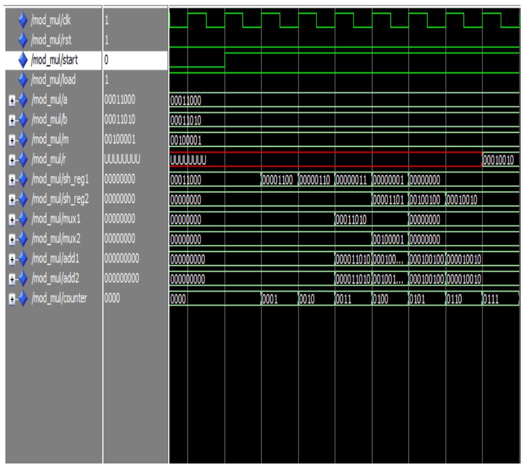


Figure 6.1.3 Simulation result for iterative multiplier behavior during the first multiplication (ii)

The simulation of iterative multiplier using Montgomery algorithm with the operands a=24, b=26, and m=33 has been carried out and the behavior of the multiplier during the first modular multiplication is shown in figure 6.1.3 (here signal *step* is not set).

6.1.4 Iterative Multiplier Second Multiplication

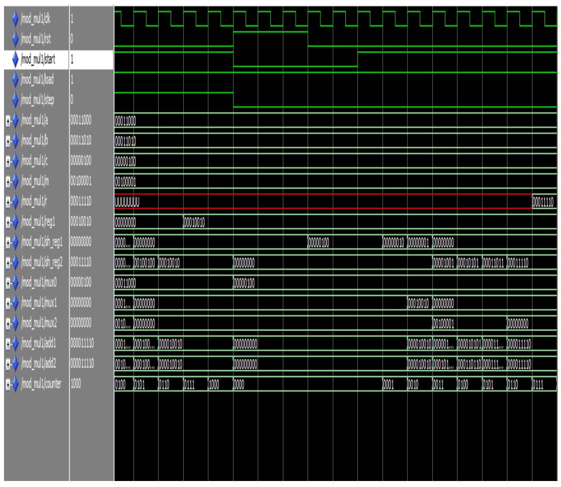


Figure 6.1.4 Simulation result for iterative multiplier behavior during the second multiplication (ii)

Figure 6.1.4 shows the results of the Iterative multiplication during the second level of multiplication which finally returns r = 14 (signal *step* is set).

6.2 Simulation Result for Systolic Multiplier

6.2.1 Systolic Multiplier Simulation Result for Operands (23,27,33)

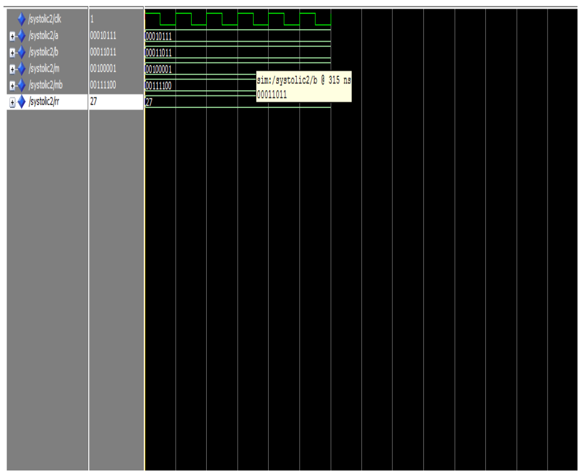


Figure 6.2.1: Simulation waveform for systolic multiplier (i)

The simulated output waveform for systolic Montgomery multiplier with the operands a=23, b=27, m=33 is shown in figure 6.2.1 which also returns r = 27.

6.2.1 Systolic Multiplier Simulation Result for Operands (53,57,61)

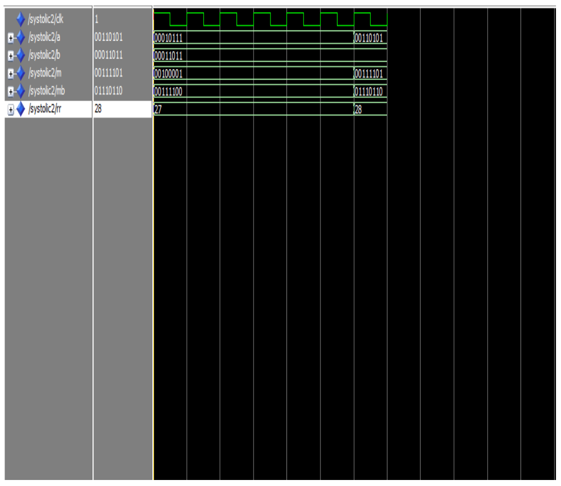


Figure 6.2.2: Simulation waveform for systolic multiplier (ii)

The simulated output waveform for systolic Montgomery multiplier with the operands a=53, b=57, m=61 is shown in figure 6.2.2 which also returns r = 28.

6.3 Design summary:

6.3.1 Iterative multiplier:

Timing Summary:

Speed Grade: -5

Minimum period: 11.222ns

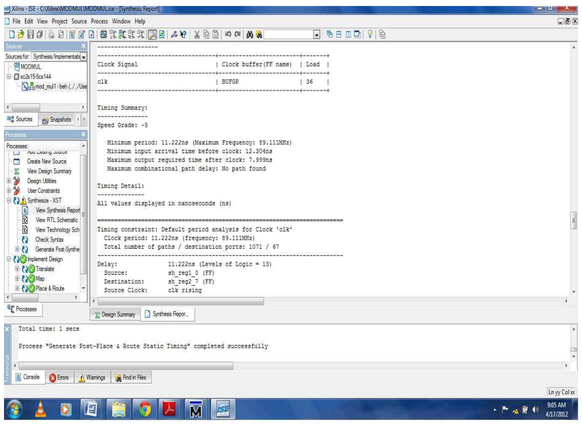
Maximum Frequency: 89.111MHz

Minimum input arrival time before clock: 12.304ns

Maximum output required time after clock: 7.999ns

Total equivalent gate count for design: 793

Additional JTAG gate count for IOBs: 2,160



```
-----------------
---------------------------------+----------------------+-------+
Clock Signal                     | Clock buffer(FF name) | Load  |
---------------------------------+----------------------+-------+
clk                              | BUFGP                 | 36    |
---------------------------------+----------------------+-------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 11.222ns (Maximum Frequency: 89.111MHz)
   Minimum input arrival time before clock: 12.304ns
   Maximum output required time after clock: 7.999ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 11.222ns (frequency: 89.111MHz)
  Total number of paths / destination ports: 1071 / 67
-------------------------------------------------------------------------
Delay:               11.222ns (Levels of Logic = 13)
  Source:            sh_reg1_0 (FF)
  Destination:       sh_reg2_7 (FF)
  Source Clock:      clk rising
```

```
Total time: 1 secs

Process "Generate Post-Place & Route Static Timing" completed successfully
```

Figure 6.3.1: Synthesis report for Iterative Multiplier

6.3.2 Systolic Multiplier:

Timing Summary:

Speed Grade: -5

Minimum period: 4.791ns

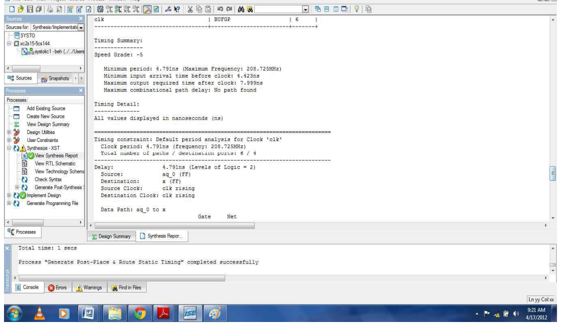
Maximum Frequency: 208.725MHz

Minimum input arrival time before clock: 4.423ns

Maximum output required time after clock: 7.999ns

Total equivalent gate count for design: 1,341

Additional JTAG gate count for IOBs: 1,920

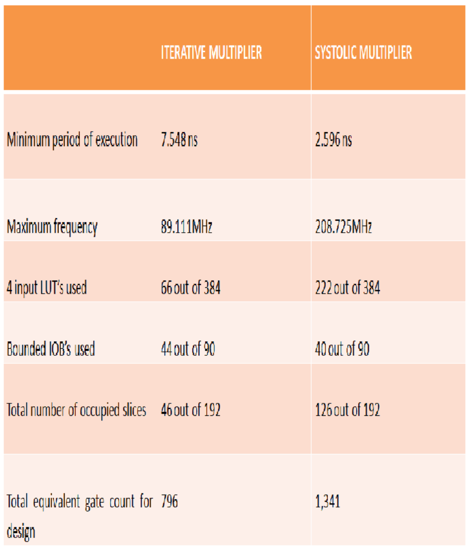


```
clk                              | BUFGP                 | 6     |
---------------------------------+----------------------+-------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 4.791ns (Maximum Frequency: 208.725MHz)
   Minimum input arrival time before clock: 4.423ns
   Maximum output required time after clock: 7.999ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 4.791ns (frequency: 208.725MHz)
  Total number of paths / destination ports: 6 / 4
-------------------------------------------------------------------------
Delay:               4.791ns (Levels of Logic = 2)
  Source:            aq_0 (FF)
  Destination:       x (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: aq_0 to x
                                Gate     Net
```

```
Total time: 1 secs

Process "Generate Post-Place & Route Static Timing" completed successfully
```

Figure 6.3.2: Synthesis report for Systolic Multiplier

6.4. Comparison of Iterative multiplier vs Systolic multiplier:



	ITERATIVE MULTIPLIER	SYSTOLIC MULTIPLIER
Minimum period of execution	7.548 ns	2.596 ns
Maximum frequency	89.111MHz	208.725MHz
4 input LUT's used	66 out of 384	222 out of 384
Bounded IOB's used	44 out of 90	40 out of 90
Total number of occupied slices	46 out of 192	126 out of 192
Total equivalent gate count for design	796	1,341

Table 6.1 Comparison of Iterative multiplier vs Systolic multiplier:

From the synthesis estimate, the minimum clock period for iterative multiplier was found to be **7.548ns**, for which the maximum clock frequency is **89.111Mhz** and for systolic multiplier the minimum clock period was found to be **2.596ns** for which maximum clock frequency is **208.725Mhz**.Execution time for Systolic multiplier is less than Iterative multiplier whereas area occupied will be more .

6.5. Graphical Representation of Speed and Area Comparison

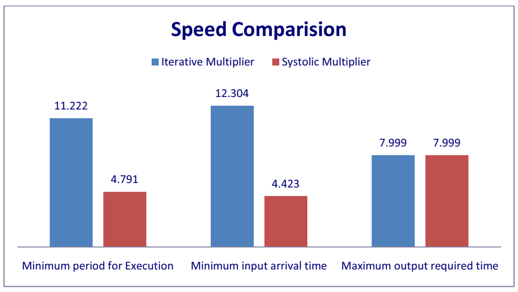


Figure 6.5.1: Speed Comparison

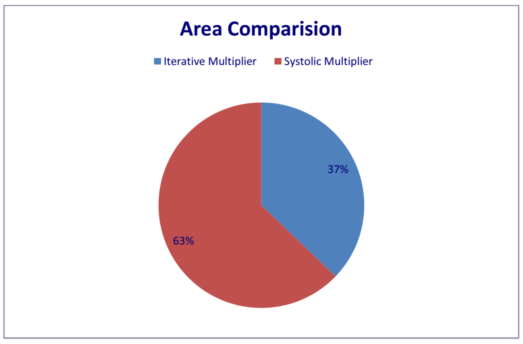


Figure 6.5.2: Area Comparison

CHAPTER 7
CONCLUSION AND FUTURE SCOPE

In RSA algorithm great consequence factor which affects its performance is modular multiplication. Here efficient implementation of modular multiplication is implemented using Montgomery technique. Each module is coded with VHDL. The VHDL code of modular multiplication is developed and synthesized using Xilinx ISE and verified. From the synthesis estimate, the minimum clock period for iterative multiplier was found to be **7.548ns**, for which the maximum clock frequency is **89.111MHz** and for systolic multiplier the minimum clock period was found to be **2.596ns** for which maximum clock frequency is **208.725MHz**. Execution time for Systolic Multiplier is less than that of Iterative Multiplier and area occupied will be more in Systolic than Iterative Multiplier so depending on the application requirement the choice of the architecture can be made for high speed or less space occupation.

FUTURE SCOPE

Future scope of this project is to develop the various efficient architectures for Montgomery modular technique which supports RSA algorithm

BIBLIOGRAPHY

[1] Jizhong Liu, Jinming Dong,"Design and implementation of an efficient RSA crypto-processor",IEEE2010

[2] Christof Paar • Jan Pelzl, "Understanding Cryptography" Springer,2010.

[3] Parth Mehta, Dhanashri Gawali, "Conventional versus Vedic Mathematical Method for Hardware Implementation of a Multiplier",2009 International Conference on Advances in Computing, Control,and Telecommunication Technologies, Trivandrum, Kerala, India,December 2009, pp. 640-642, 2009.

[4] N. Ncdjah, M. M. L. De, "Three hardware architectures for the binary modular exponentiation: sequential, parallel, and systolic," IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 2006, 53(3), pp.627-633.

[5] R.V.Kamala, M.B.Srinivas," High-Throughput Montgomery Modular Multiplication"IEEE 2006.

[6] William Stallings, "Cryptography and Network Security: Principles and Practices." 3rd edition, 2003.

[7] Navabi, Z., *VHDL - Analysis and modeling of digital systems*, McGraw Hill, Second Edition,1998.

[8] C. D. Walter, "Systolic modular multiplication," IEEE Transactions on Computers, 1993,42(3), pp.376-378.

[9] Peter J Ashenden & Jim Lewis, "The Designer's Guide to VHDL",Morgan Kaufmann Publishers.

DESIGN AND IMPLEMENTATION OF AN EFFICIENT RSA CRYPTOPROCESSOR

By

DHINESH.R

Reg. No. 1020106004

of

KUMARAGURU COLLEGE OF TECHNOLOGY

(An Autonomous Institution affiliated to Anna University, Coimbatore)

COIMBATORE - 641006

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements
for the award of the degree
of

MASTER OF ENGINEERING
IN
APPLIED ELECTRONICS
MAY 2012

BONAFIDE CERTIFICATE

Certified that this project report entitled "**DESIGN AND IMPLEMENTATION OF AN EFFICIENT RSA CRPTOPROCESSOR**" is the bonafide work of **Dhinesh.R [Reg. no. 1020106004]** who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Project Guide	Head of the Department
Ms.M.Alagumeenaakshi.	Dr. Rajeswari Mariappan

The candidate with university **Register no.1020106004** is examined by us in the project viva-voce examination held on …………………….

Internal Examiner	External Examiner

ACKNOWLEDGEMENT

First I would like to express my praise and gratitude to the Lord, who has showered his grace and blessing enabling me to complete this project in an excellent manner. He has made all things beautiful in this time.

I express my profound gratitude to our chairman **Padmabhusan Arutselvar Dr.N.Mahalingam B.Sc.,F.I.E.** for giving this opportunity to pursue this course.

I express my sincere thanks to our beloved Director **Dr.J.Shanmugam**, Kumaraguru College of Technology, I thank for his kind support and for providing necessary facilities to carry out the work.

At this pleasing moment of having successfully completed the project work, I wish to acknowledge my sincere gratitude and heartfelt thanks to our beloved principal **Prof.Ramachandran,** for having given me the adequate support and opportunity for completing this project work successfully.

I express my sincere thanks to **Dr.Rajeswari Mariappan Ph.D.,** the ever active, Head of the Department of Electronics and Communication Engineering, who is rendering us help all the time throughout this project

In particular, I wish to thank with everlasting gratitude to the project coordinator **Ms.R.HemaLatha Asst.Prof(SRG),**Department of Electronics and Communication Engineering for her expert counseling and guidance to make this project to a great deal of success. Her careful supervision ensured me in attaining perfection of work.

I extend my heartfelt thanks to my internal guide **Ms.M.Alagumeenaakshi, Asst Prof(SRG).**for her ideas and suggestion, which have been very helpful for the completion of this project work.

Last, but not the least, I would like to express my gratitude to my family members, friends and to all my staff members of Electronics and Communication Engineering department for their encouragement and support throughout the course of this project.

ABSTRACT

In this project a hardware implementation of the RSA algorithm for public-key cryptography has been presented. Basically, the RSA algorithm entails a modular exponentiation operation on large integers, which is considerably time consuming to implement. Two types of architecture based on Montgomery's technique were proposed and efficient comparison is made with respect to speed and area. To this end, a trade-off between processing speed and space has been studied in the implementation of modular exponentiation using Montgomery's technique. Comparatively smaller amount of space is occupied in the FPGA due to its reusability. The architecture is synthesized and simulated using VHDL code and implemented using Xilinx.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

RSA	-------	**Rivest Shamir and Adleman**
VHDL	-------	**Very High Speed Integrated Circuit Hardware Description Language**
FPGA	-------	**Field-Programmable Gate Arrays**
ADC	-------	**Analog to Digital Converter**
ASIC	-------	**Application Specific Integrated Circuits**
SOC	-------	**System-On-Chip**

CHAPTER 1

INTRODUCTION

The network security problem for an increasing traffic of Internet transactions has been emerged and there have been lots of researches in cryptographic algorithms. Cryptography plays an important role in the information security in the modern communication and computer networks. As the most widely used public-key cryptography algorithm, RSA algorithm was proposed by R. L. Rivest, A. Shamir, L.Adleman in 1978. Its safety depends on the difficulty of big integer factorization. The RSA operation is the modular exponentiation with long number in essence, which can be calculated by the iteration of modular multiplication. In 1980, the Montgomery algorithm was proposed to enable people to see the possibility to realize RSA algorithm on the hardware. RSA system is one among them and widely used today. The system requires fast modular multiplication of integer numbers containing several hundred bits. The algorithm discussed here is that of providing hardware to perform modular multiplication using the algorithm proposed by P.L.Montgomery and further developed by the S.E.Eldridge and Kaliski. As cryptography becomes more widespread, fast cryptographic implementations are becoming important.

Experience with hardware tools has shown that speed often cannot fully be realized unless all cryptographic methods of interest are implemented in hardware. VHDL language provides good design tools for constructing systolic arrays as well as graphical ones. Even though RSA system is a perfect solution, it provide reasonable security feature over the Internet with hundred bits of encryption standard. Fast processing of the modular multiplication is one of the most important key issues on the RSA technology. As data size demanded over the network grows larger and larger every year, faster processors as well as sophisticated algorithm which is more effective for the hardware and software are always in high demand. One of the main objectives of cryptography consists of providing *confidentiality*, which is a service used to keep secret publicly available information from all but those authorized to access it. There exist many ways to providing secrecy. They range from physical protection to mathematical solutions, which render the data unintelligible.

Although the RSA algorithm has high security, due to the complication of its essential modular exponentiation of large numbers, the speed of calculation turns out to be the biggest bottleneck and technical problem. In recent years, the improvements in terms of algorithm and the architecture of the crypto-processor have diminished the time for calculation to a certain extent. However, it is still unsuitable to treat with the long data.

1.1. PROJECT GOAL

The essential arithmetic operation carried out in RSA algorithm is modular multiplication, which is used to calculate modular exponentiation. Modular exponentiation on numbers of hundreds of bits makes it difficult for RSA algorithm to attain output. The goal of this project is to implement two architectures based on Montgomery's Modular Multiplication algorithm and efficient comparison is made with respect to speed and area.

1.2 OVERVIEW

Montgomery modular multiplication algorithm is first implemented which can avoid range comparison which is a critical operation in traditional division in modular multiplication. Basically modular exponentiation with a large modulus is usually accomplished by performing repeated modular multiplication which is considerably time consuming. As a result the throughput rate of RSA cryptosystem will be entirely dependent on speed of modular multiplication and number of performed modular multiplications. To speed up the process the above implemented Montgomery modular multiplication is employed in exponentiation algorithm thus increasing performance of RSA cryptosystem.

1.3 SOFTWARE USED

- Model Sim 6.3f
- Xilinx ISE 8.1i

1.4 ORGANIZATION OF THE REPORT

- **Chapter 2** discusses about the general RSA algorithm overview.
- **Chapter 3** discusses about modular arithmetic carried in RSA algorithm.
- **Chapter 4** discusses the montgomery modular multiplication algorithm.
- **Chapter 5** discusses the hardware implementation of two architectures algorithm.
- **Chapter 6** discusses the simulation results.
- **Chapter 7** shows the conclusion of the project.

CHAPTER 2

RSA ALGORITHM

2.1 Introduction

This algorithm is based on the difficulty of factorizing large numbers that have 2 and only 2 factors (Prime numbers). The system works on a public and private key system. The public key is made available to everyone. With this key a user can encrypt data but cannot decrypt it, the only person who can decrypt it is the one who possesses the private key. It is theoretically possible but extremely difficult to generate the private key from the public key, this makes the RSA algorithm a very popular choice in data encryption.

2.2 Algorithm

- Choose two large primes p and q.
- Compute n = p×q.
- Calculate φ(n) = (p-1) ×(q-1).
- Select the public exponent e € {1,2, . . . , φ(n)−1}.
- Plain text is converted in to cipher text by C = M^e *mod n*
- Compute the private key d such that d×e ≡ mod φ(n).
- Plain text is recovered by M =C^d *mod n*

First of all, two large distinct prime numbers p and q must be generated. The product of these, we call n is a component of the public key. It must be large enough such that the numbers p and q cannot be extracted from it 512 bits at least i.e. numbers greater than 10. We then generate the encryption key e which must be co-prime to the number φ(n) = (p - 1)(q - 1).We then create the decryption key d such that d*e mod φ(n) = 1. We now have both the public and private keys.

2.3 Encryption

We let y = E(x) be the encryption function where x is an integer and y is the encrypted form of x , y = x^e mod n

2.4 Decryption

We let X = D(y) be the decryption function where y is an encrypted integer and X is the decrypted form of y, X = y^d mod n

CHAPTER 3

MODULAR ARITHMETIC IN RSA ALGORITHM

3.1 Introduction

Two numbers are the modular inverse of each other if their product equals 1.

$$\text{If (AB) mod N} = 1 \text{ mod N, then B} = A^{-1}$$

The advantage of modular arithmetic is that it is no need to manipulate any huge numbers. The size of the number is depends on the modular base.

$$\text{AB mod N} = \text{((A mod N) (B mod N)) mod N,}$$

$$\text{ABC mod N} = \text{((AB mod N) (C mod N)) mod N,}$$

$$\text{ABCD mod N} = \text{((AB mod N) (CD mod N)) mod N, and etc..}$$

This algorithm never looks at any product larger than $(N\text{-}1)^2$.

Above multiplication algorithm can be directly applied to compute

$$C^m \text{ mod N, where C=A=B=D=E…}$$

If C^m mod N = 1 mod N, then C^{m}+1 mod N = C.

With a number C and some power *d* modulo N,

$$A = C^d \text{ mod N.}$$

By taking the result of the above exponentiation, A, it is possible to raise it to some other power *e*,

$$A^e \text{ mod N} = (C^d)^e \text{ mod N} = C^{de} \text{ mod N.}$$

3.2 Fermat's Little Theorem.

Leonhard Euler described φ (*n*), phi-function, which is the "exponential period" modulo *n* for numbers relatively prime that means it shares only one common factor, namely 1 with *n*.

For example, φ (6) = 2 because only 1 and 5 are relatively prime with 6. φ (7) = 6 because any number less than 7 can share with 7 only 1 as a common factor and they (1,2,3,4,5 and 6) are all relatively prime with 7. Clearly this result will extend to all prime numbers. Namely, if p is prime, φ (*p*) = *p*-1. For example, if *n* = 5, a prime number, then φ (*n*) = 4. Set *a* be 3.

$$a^{\varphi(n)} \text{ mod } n = 3^4 \text{ mod } 5$$

$$3^2 \text{ mod } 5 = 4$$

$$3^3 \text{ mod } 5 = 4*3 \text{ mod } 5 = 2$$

$$3^4 \text{ mod } 5 = 2*3 \text{ mod } 5 = 1$$

Fermat's little theorem is a statement about powers in modular arithmetic in the special case where the modular base is a prime number. Pohlig and Hellman study a scheme related to RSA, where exponentiation is done modulo a prime number.

$$a^{(p-1)} = 1 \text{ mod } p$$

unless *a* is a multiple of *p* which must be a $^{\text{prime}}$ number.

Rivest, Shamir, and Adelman designed a fascinating encryption algorithm with Fermat's little theorem. Decryption is only possible for the chosen few who have extra information.

For given *d, p* and computed A = C^d mod *p*, to find *e* such that A^e mod *p* = 1, we can simply find *e* such that *d***e* = φ (*p*) which equals to *p*-1. Because then

$$A^e \text{ mod } p = C^{de} \text{ mod } p = 1 \text{ mod } p.$$

For given *d, p* and computed A = C_d mod *p*, to find *e* such that A^e mod *p* = C, we need to find *e* such that *de* = φ (*p*) + 1. Because then

$$A^e \text{ mod } p = C^{de} \text{ mod } p = C \text{ mod } p.$$

From the fact that

$$(\varphi(p) + 1) \text{ mod } \varphi(p) = (\varphi(p) + 1) - \varphi(p) = 1 \text{ mod } \varphi(p),$$

Therefore, finding *e* such that *de* = φ (*p*) + 1 is equivalent to finding *e* such that *de* = 1 mod φ (*p*),which is known as the modular inverse. There is a method known as extended Euclidean algorithm for computing the modular inverse.

After picking a public exponent *d* and by finding a prime *p*, make those two values public. Using the extended Euclidean algorithm, determine *e*, the inverse of the public exponent modulo φ (*p*) = *p*-1. When people want to send someone a message C, they can encrypt and produce cipher text A by computing A = C_d mod *p*. To recover the plain text message, compute C = A^e mod *p*. But the private key *e* is the inverse of *d* modulo *p*-1. Since *p* is public, anyone can compute *p*-1 and therefore determine *e*.

RSA algorithm solves the above problem by using an Euler's multiplicative phi-function. If *p* and *q* are relative prime, then φ (*pq*) = φ (*p*) φ (*q*). Hence, for primes *p* and *q* and *n* = *pq*,

$$\varphi(n) = (p\text{-}1)(q\text{-}1).$$

The problem is finding *e* that satisfies

$$d{*}e = 1 \bmod (p\text{-}1)(q\text{-}1)$$

where the pair (*n,d*) is the public key and *e* is the private key. The prime p and q must be kept secret or destroyed. To compute cipher text A from a plain text message C, find A = C^d mod *n*. To recover original message, compute C = A^e mod *n* . Only the entity that knows *e* can decrypt. Because of the relationship between *d* and *e*, the algorithm correctly recovers the original message C, since

$$A^e \bmod n = C^{de} \bmod n = C_1 \bmod n = C \bmod n.$$

To know φ (*n*) one must know *p* and *q*. In other words, they must factor *n*. Multiplying big prime numbers can be a one-way function. Factoring takes a certain number of steps, and the number of steps increases sub-exponentially as the size of the number increases. Extended Euclidean algorithm can be used to find private key *e*.

Using this fact, it is natural to build the private key using two primes and the public key using their product. There is one more condition, the public exponent *d*, must be relatively prime with (*p*-1)(*q*-1) to exist a modular inverse *e*. In practice, one would generally pick *d*, the public exponent first, then find the primes *p* and *q* such that *d* is relatively prime with *(p-1)(q-1).* There is no mathematical requirement to do so, it simply makes key pair generation a little easier. In fact, the two most popular *d*'s in use today are F_0 = 3 and F_4 = 65,537. The F in F_0 and F_4 stands for Pierre de Fermat.

CHAPTER 4
MONTGOMERY MODULAR ALGORITHM

4.1 Introduction

The Montgomery algorithm for modular multiplication is considered to be the fastest algorithm to compute x*y mod n in computers when the values of x, y, and n are large. In this the Montgomery algorithm for modular multiplication is described. In order to compute x*y mod n choose a positive integer, r, greater than n and relative prime to n. The value of r is usually 2^m for some positive integer m. This is because multiplication, division and modulo by r can be done by shifting or logical operations in computers.

In RSA cryptography system, it requires lots of modular multiplication. Montgomery proposed an algorithm that conducts the modular multiplication without trial division but produces some residue. This algorithm is suitable for hardware or software implementation. Let N be an integer (the modulus) and let R be an integer relatively prime to N. We represent the residue class A mod N as AR mod N and redefine modular multiplication as

$$\text{Mont\_Product}(A, B, N, R) = ABR^{-1} \bmod N.$$

It is not hard to verify that Montgomery multiplication in the new representation is isomorphic to modular multiplication in the ordinary one:

$$\text{Mont\_Product}(AR \bmod N, BR \bmod N, N, R) = (AB)R \bmod N.$$

Since AR mod N and BR mod N are both less than N, their product is less than NN that is less than RN, so it forms a legal input for Mont_Product. A drawback of the algorithm is the redundant factor of R for the desired result, (AB) mod N.

We can similarly redefine modular exponentiation as repeated Montgomery multiplication. This "Montgomery exponentiation" can be computed with all the usual modular exponentiation speedups.

4.2 Montgomery Modular Multiplication

Montgomery modular multiplication algorithm employs only simple additions, subtractions and shift operations to avoid trial division-a critical and time consuming operation in conventional modular multiplication. Let the modulus N be a k bit odd number

and the factor R is defined as 2^k mod N. The details of Montgomery modular multiplication algorithm employed to compute A * B * R^{-1} mod N are stated in algorithm MM(A,B,N) in which A and B are integers smaller than N. In the algorithm the notation B[i] E {0,1} represent the i^{th} bit of B. The value of k can be any number and when finding the value of S the loop is executed for k-1 iterations. So modular multiplications operations are carried within the loop execution and thus Montgomery algorithm can be implemented to exponentiation algorithm thus speeding up RSA process. The process that carried out in Montgomery modular multiplication algorithm is explained in stepwise procedure below.

```
   Algorithm MM (A, B, N)
//   montgomery's modular multiplication algorithm
//   Inputs: three k bit integers
       N (modulus) ,A (multiplicand),B(multiplier)
//   Outputs: S= A x B x $R^{-1}$  mod N
             Where R= $2^k$ mod N   and  0< S<N
             {
                 S=0;
               for  i=0 to k-1    {
               q=(S + A x B[ i]) mod 2;
                 S=(S + A x B[i] +q x N)/2;
               }
              If (S> N) S= S-N;
              return  S;
           }
```

If the inputs of modular multiplication algorithm are given in binary representation, and the intermediate result is stored in carry save form, then a format conversion operation, i.e., a ripple carry addition is needed to convert the result back into binary representation. In this manner, the final result can be directly used in next modular multiplication as desired in modular exponentiation applications. All the intermediate variables are operated in carry save form to reduce critical path delay. The level of carry save addition tree can be reduced from three to two.

It can be applied to perform either square or multiplication operation in a modular exponentiation algorithm. From algorithm of modular exponentiation it is observed that the H algorithm transforms the computation of modular exponentiation into a sequence of squares and multiplications. Since the multiplicand of the multiplication operation is the value of the message in transformed domain, which is fixed within "for loop" of modular exponentiation algorithm, it can be converted into binary form right after the preprocessing step.

The modified algorithm with carry save representation and unified multiplication and square operations is stated in algorithm MM_UMS(A,B_C.B_S,N).In the algorithm the modulus N, an odd number, has k+1 bits with a dummy bit N[K]=0.The (k+1) bit multiplicand A is less than 2N.The value of multiplier B=B_C+B_S is also less than 2N with each variable of (k+1) bits. The extra factor R is redefined as 2^{k+3} mod N because one more iteration is needed.

```
Algorithm MM_U MS (A, $B_C$, $B_S$, N)

//Proposed modular multiplication algorithm
// Inputs   : four (k+1) bit variables
               N, A, $B_C$  and  $B_S$ (with A and  $B_C$ + $B_S$ < 2N)
// Output   : S = ($S_C$, $S_S$)
                 = A x ($B_C$ + $B_S$) x $R^{-1}$ mod N    // for multiplication or
                 = ($B_C$ +$B_S$) x ($B_C$ +$B_S$) x $R^{-1}$ mod N
                  //   for square.
                 for 0<S<2N and R=$2^{K+3}$ mod N
{
      Ss = 0; Sc=0; carry = 0;
      B [k + 2] = B [k  + 1]=B [-1 ]=B [-2 ]=0;
       for i=0 to k +2 {
              (carry, B [i]) = $B_C$ [i] +$B_S$ [i] + carry;
                                               // rebuild multiplier B [ i]
              If (mode = square) then {
                                               // performing square
```

```
            $PP = B[i] \times 2^{i} + B[i-2:0] \times B[i-1]$;
                                          //bit concatenation
             }
             else   {
                                          // performing multiplication
             $PP = A[k:0] \times B[i]$;
              }
            $q = (S_C + S_s + PP) \bmod 2$;
            $(S_C, S_S) = (S_C + S_S + PP + q \times N)/2$;
                                          // a 4 to 2 CSA tree
              }
             Return $(S_C, S_S)$;
}
```

In the Montgomery modular multiplication algorithm, inputs arrived are A, B_C, B_S and N and the condition applied is A, $B_C + B_S$ should be less than 2N. The output is obtained in sum and carry form and since multiplication/ square module is carried out, the outputs for these two modules are carried out separately. For multiplication module the output is calculated as $S=A \times (B_C + B_S) \times R^{-1} \bmod N$ and for the square module it is $S=(B_C + B_S) \times (B_C + B_S) \times R^{-1} \bmod N$. The condition is provided for $0<S< 2N$ and $R = 2^{k+3} \bmod N$.

After bit concatenation operations and performing multiplication and square modules the output; a 4 to 2 CSA tree is arrived at the output stage of proposed modular multiplication algorithm. The control signal 'mode' used to select which operation is to be performed is not shown in the input list of algorithm. From the proposed algorithm it is concluded that the 4 to 2 CSA tree can be kept unchanged with very limited control overhead to support both multiplication and square operations and no format conversion is needed because related input variables and intermediate results are in carry save form.

A close examination of the algorithm execution may find a little delay and to remove this a modified version named as algorithm MM_UMS_P, which can be easily extended from algorithm MM_UMS by adding one more iteration step into new algorithm is introduced.

Here the control signal 'mode' is not used because delay occurring due to mode operation is eliminated in the MM_UMS_P algorithm. The algorithm for modified version is as follows.

```
Algorithm MM_UMS_P(A, $B_C$, $B_S$, N)

// The improved version of MM_UMS
// Inputs   : four (k+1) bit variables
              N, A, $B_C$ and $B_S$
// Output   : $S = (S_C, S_S)$
              $= A \times (B_C + B_S) \times R^{-1} \bmod N$    // for multiplication or
              $= (B_C + B_S) \times (B_C + B_S) \times R^{-1} \bmod N$
               //   for square.
              For $0<S<2N$ and $R=2^{k+3} \bmod N$
     {
     $S_S = 0$; $S_C = 0$; carry =0;
     $B[k+2]=B[k+1]=B[-1]=B[-2]=B[-3]=0$;
 //$B[-3]=0$ for pipelined operation
       for i=0 to k +3 {                    // bounded by k+3 instead of k+2
           (carry, $B[i]$) = $B_C[i] + B_S[i]$ + carry;
                                            // rebuild multiplier B [ i]
           If (mode = square) then {
                                            // select multiplication
           $PP = B[i-1] \times 2^{i-1} + B[i-3:0] \times B[i-2]$;
                                            // delayed bit concatenation          }
           else {                          // performing multiplication
           $PP = A[k:0] \times B[i-1]$;
                                            //delayed partial product
             }
            $q = (S_C + S_S + PP) \bmod 2$;
            $(S_C, S_S) = (S_C + S_S + PP + q \times N)/2$;
                                            // a 4 to 2 CSA tree
```

```
              }
             Return $(S_C, S_S)$;
   }
```

In improved version of MM_UMS an additional iteration step is present. Here the control signal mode is not present. Same as in MM_UMS output for multiplication and square module is obtained reducing the delay. This proposed Montgomery's modular multiplication algorithm is used to speed up the computation in proposed exponentiation algorithm.

4.3 Montgomery Modular Exponentiation

The modular exponentiation is usually accomplished by performing repeated modular multiplications. If operations are processed in Montgomery domain, then additional preprocessing and post processing steps are needed to convert operands into desired domain. First to bound an output range an extra bit of precision is added to intermediate results for precision consideration. When preprocessing steps are carried out unwanted factors are removed automatically. After the last iteration of modular multiplication operation, preprocessing is carried out. Therefore not only removing unwanted factor of the result but also make the result fall in the right range after post processing is done here.

Let M be a k bit message with its value less than the modulus N, and E denote the k_d bit exponent or key. The H algorithm of modular exponentiation is used to compute $C=M^E \bmod N$, which is summarized in algorithm ME(M,E,N) in which two factors W and R are computed in advance. The value of R is either 2^k or 2^{k+2} depending on the chosen modular multiplication algorithm.

In exponentiation algorithm in the preprocessing step the Montgomery function is called so that the computations are reduced, thus speeding up the exponentiation procedures. The value of S is computed taking the Montgomery domain but inputs are chosen such that W is calculated in advance algorithm execution is initiated where loop repeats for k_d-1 iterations. The value of k_d can be any number and depending on its value the loop execution depends on. In the final step output is computed and less time is required for computation because of

shorter critical path. Because of this algorithm induce less time to perform RSA operation and so higher performance is attained.

This investigate how to design a unified computation unit that can be applied to efficiently fulfill the square and multiplication operations for operands in carry save form. In this way a cost effective solution is derived to design modular exponentiation based on the H algorithm with almost no performance degradation.

```
Algorithm ME (M, E, N)
// H algorithm of modular exponentiation
// Inputs   : N and M (k bits), E ($k_d$ bits) and $W=R^2 \bmod N$
// Outputs: $C=M^E \bmod N$, where $0<C<N$
          {
             M* =MM (M, W, N);          // preprocessing
               S = MM (1, W ,N);
              For i= $k_d$ -1 to 0{          // H algorithm
               S= MM(S,S, N);             // Square
               if ( E[i] =1){
                S=MM (M*,S, N)            //Multiplication
                  }
             }
            C=MM(S,1, N);                   // post processing
              Return C;
}
```

In the preprocessing step, the Montgomery algorithm is called for with three inputs M, W and N and S is computed with Montgomery domain with inputs 1, W and N. The H algorithm is implemented inside for loop and square operations in the H algorithm of Modular exponentiation is indicated using MM with inputs S, S and N. The preprocessing M* is called inside the multiplication module which gives S whose inputs in Montgomery domain are M*, S and N. At post processing step output C is obtained which is MM(S, 1, N).

A square and multiply method has to be modified to transform the input value in to the Montgomery format, which include a R value and to transform the output from the Montgomery domain into plain format. When applying the developed algorithm MM_UMS_P to modular exponentiation only two format conversions are needed: one for preprocessing and other for post processing. The main part of modular exponentiation is free of format conversion. The following proposed modular exponentiation algorithm is denoted as algorithm ME_UMS(M,E,N).In algorithm ME_UMS, both modulus N and message M are (k + 1)bit input variables with dummy bits N[k] = M[k] = 0,the exponent E consists of k_d bits, and precomputed constants are $R=2^{k+3}$ mod N and $W = R^2$ mod N. The proposed exponentiation algorithm is as follows.

```
Algorithm ME_UMS(M,E,N)

  //  Inputs   : N and M (k + 1) bits, E ($k_d$ bits) and
                 W = $R^2$ mod N with R = $2^{k+3}$ mod N
  // Output    : C =$M^E$ mod N where 0<C< N
                 {
                    (M'c,M's) =MM_UMS_P (M, 0, W, N);
                                                      // Preprocessing
                    M' =M'c + M's;                    // format conversion
                    ($S_C$, $S_S$) =MM_UMS_P (1, 0, W,N )
                     for i = $k_d$ -1 to 0 {
                           ($S_C$, $S_S$)  = MM_UMS_P (0, $S_C$, $S_S$, N);
                                                      // square
                     if (E[i] =1){
                        ($S_C$, $S_S$) = MM_UMS_P (M', $S_C$, $S_S$, N);
                                                      // multiplication
                                 }
                 }

    ($S_C$, $S_S$) = MM_UMS_P (1, $S_C$, $S_S$, N);
```

```
                                                      // post processing
    C= $S_C$ + $S_S$;                                 // format conversion
     return C ;
   }
```

Thus output is obtained with several preprocessing and post processing steps and C = M^E mod N and with variations of M, E and N value outputs are obtained. The proposed Montgomery multiplication algorithm is employed in H algorithm and both square and multiplication operations call for the algorithm and finally output is obtained.

4.4 Implementation of Montgomery Modular Technique

Algorithm that formalize the operation of modular multiplication generally consist of two steps: one generates the product P = A-B and the other reduces this product P modulo M. 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 n clock cycles, n is the size of the operands. 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. One of the widely used algorithms for efficient modular multiplication is the Montgomery algorithm. This algorithm computes the product of two integers modulo a third one without performing division by M. It yields the reduced product using a series of additions

Let A, B and M be the multiplicand and multiplier and the modulus respectively and let n be the number of digit in their binary representation, i.e. the radix is 2. So, we denote A, B and M as follows:

$$A=\sum_{i=0}^{n-1} a_i \times 2^i ,\quad B=\sum_{i=0}^{n-1} b_i \times 2^i \text{ and } M=\sum_{i=0}^{n-1} m_i \times 2^i$$

The pre-conditions of the Montgomery algorithm are as follows:

- The modulus M needs to be relatively prime to the radix, i.e. there exists no common divisor for M and the radix;
- The multiplicand and the multiplicator need to be smaller than M.

The binary representation of the operands were used here, then the modulus M has to be odd to satisfy the first pre-condition. The Montgomery algorithm uses the least significant digit of the accumulating modular partial product to determine the multiple of M to subtract. The usual multiplication order is reversed by choosing multiplier digits from least to most significant and shifting down. If R is the current modular partial product, then q is chosen so that R+q-M is a multiple of the radix r, and this is right-shifted by r positions, i.e. divided by r for use in the next iteration. So, after n iterations, the result obtained is R =A*B*r^{-n} mod M . A modified version of Montgomery algorithm is given below.

```
algorithm Montgomery(A, B, M)
        int R = 0;
  •     1: for i= 0 to n-1
  •     2: R = R + $A_i$ - B;
  •     3: if $r_o$ = 0 then
  •     4: R = R div 2
  •     5: else
  •     6: R = (R + M) div 2;
        return R;
end Montgomery.
```

In order to yield the right result, an extra Montgomery modular multiplication by the constant 2^n mod M is needed. However as the main objective of the use of Montgomery modular multiplication algorithm is to compute exponentiations, it is preferable to Montgomery pre-multiply the operands by 2^{2n} and Montgomery post multiply the result by 1 to get rid of the 2^n factor. The implementation of the Montgomery multiplication algorithm has been concentrated.

As binary representation of numbers were used, the final result has been computed using the algorithm which is illustrated below

```
algorithm ModularMult(A, B, M, n)
  •  const C := $2^n$ mod M;
  •  int R := 0;
  •  R := Montgomery(A, B, M);
return Montgomery(R, C, M);
end ModularMult.
```

CHAPTER 5
HARDWARE IMPLEMENTATION

Hardware implementation using Montgomery technique is implemented in two different architectures namely iterative multiplier and systolic multiplier.

5.1 Iterative multiplier

In this section, the architecture of the Montgomery modular multiplier is explained. It expects the operands A, B and M and it computes R = (A×B×2^{-n}) mod M. Detailed architecture of the Montgomery modular multiplier is given in Figure 5.1.

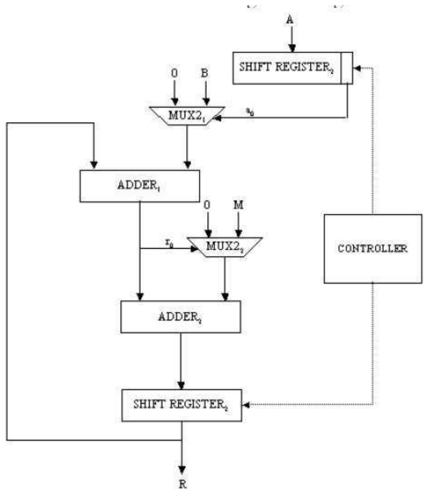


Figure 5.1:Montgomery modular architecture

It uses two multiplexers, two adders, two shift registers, three registers and a controller. The first multiplexer of the proposed architecture, i.e. $MUX2_1$ passes 0 or the content of register B depending on whether bit a_0 indicates 0 or 1 respectively. The second multiplexer, i.e. $MUX2_2$ passes 0 or the content of register M depending on whether bit r_0 indicates 0 or 1respectively. The first adder, i.e. ADDER1, delivers the sum R + A_i × B and the second adder, i.e. ADDER2, yields the sum R + M. The shift register SHIFT REGISTER1 provides the bit a_i. In each iteration i of the multiplier, this shift register is right-shifted once so that a_0 contains a_i.

The role of the controller consists of synchronizing the shifting and loading operations of the SHIFT REGISTER1 and SHIFT REGISTER2. It also controls the number of iterations that have to be performed by the multiplier. For this end, the controller uses a simple down counter. The counter is inherent to the controller.

In order to synchronize the work of the components of the architecture, the controller consists of a state machine, which has 6 states defined as follows:

• S0: Initialize of the state machine;
Go to S1;
• S1: Load multiplicand and modulus into the corresponding registers;
Load multiplier into shift register1;
Go to S2;
• S2: Wait for ADDER1;
Wait for ADDER2;
Load multiplier into shift register2;
Increment counter;
Go to S3;
• S3: Enable shift register2;
Enable shift register1;
• S4: Check the counter;
If 0 then go to S5 else go to S2;
• S5: Halt;

Modular Multiplier Architecture

The modular multiplier yields the actual value of A×B mod M. It first computes R = A×B×2^{-n} mod M using the Montgomery modular multiplier. Then, it computes R × C mod M, where C = 2n mod M.

The modular multiplier uses a 4-to-1 multiplexer MUX4 and a register REGISTER.

• Step 0: Multiplexer MUX4 passes 0 or B. MUX2 passes A. It yields R1 = A×B×2-n mod M. The register denoted by REGISTER contains 0.

• Step 1: Multiplexer MUX4 passes 0 or R. MUX2 passes C. It yields R = R1×C mod M. The register denoted by REGISTER contains the result of the first step computation, i.e. R = A×B×2^{-n} mod M. The modular multiplier architecture is given in Figure 5.2.

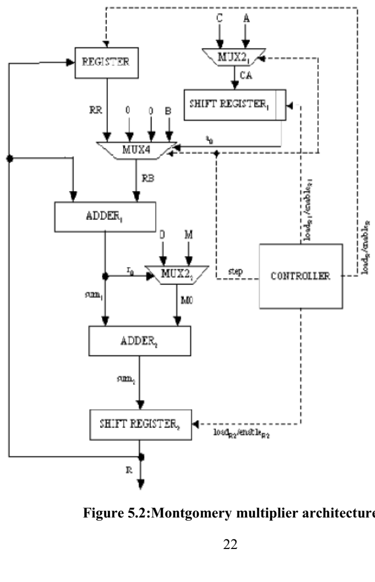


Figure 5.2:Montgomery multiplier architecture

In order to synchronize the actions of the components of the modular multiplier, the architecture uses a controller, which consists of a state machine of 10 states. The interface of CONTROLLER is shown in Figure 5.3.The modular multiplier controller does all the control that the Montgomery modular multiplier needs as described in the previous section. Furthermore, it controls the changing from step 0 to step 1, the loading of the register denoted by REGISTER.

• S0: Initialize of the state machine;
Set step to 0; Go to S1;
• S1: Load multiplicand and modulus;
Load multiplier into SHIFT REGISTER1; Go to S2;
• S2: Wait for adder1; Wait for ADDER2;
Load partial result into SHIFT REGISTER2;
Increment counter; Go to S3;
• S3: Enable SHIFT REGISTER2;
Enable SHIFT REGISTER1; Go to S4;
• S4: Load the partial result of step 0 into REGISTER;
Check the counter;
If 0 then go to S5 else go to S2;
• S5: Load constant into SHIFT REGISTER1;
Reset REGISTER;
Set step to 1; Go to S6;
• S6: Wait for ADDER1; Wait for ADDER2;
Load partial result into SHIFT REGISTER2;
Increment counter; Go to S7;
• S7: Enable SHIFT REGISTER2;
Enable SHIFT REGISTER1; Go to S8;
• S8: Check the counter;
If 0 then go to S9 else go to S6;

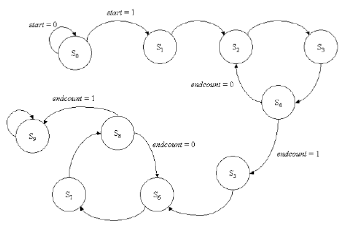


Figure 5.3: State Machine of the Controller

5.2 Systolic multiplier

The concept of systolic architecture combines a highly parallel array of identical Processing Elements or data-paths with local connections, which take external inputs and process them in a predetermined manner and in a pipelined fashion. The proposed systolic architecture is directly based on the arithmetic operations of the Montgomery Algorithm, which are performed in a numerical base 2*k*, in which the large input operands are processed in a multi-precision context containing *m* words of *k* bits.

The architecture is composed of *m* Processing Elements distributed in a one-dimensional array, where each Processing Element is responsible for the calculus involving *k* bits words of the input operands with the same index of the Processing Element. Between the Processing Elements, there is a propagation of carry signals which are the most significant bits of the arithmetic processes in each PE. The carry signals are processed as input parameters by the Processing Elements that receive them. The algorithm for systolic Montgomery multiplier which is implemented is shown below

```
algorithm SystolicMontgomery(A,B,M,MB)
int R := 0;
bit carry := 0, x;
0: for i: = 0 to n
```

```
1: q_i: = r_0^(i) ⊕ a_i.b_0;
2: for j: = 0 to n
3: switch a_i, q_i
4: 1,1: x := mb_j;
5: 1,0: x := b_j;
6: 0,1: x := m_j;
7: 0,0: x := 0;
8: r_j^(i+1) := r_{j+1}^(i) ⊕ x_i⊕ carry;
9: carry:= r_{j+1}^(i). x_i + r_{j+1}^(i).carry +x_i. carry;
return R;
end SystolicMontgomery.
```

In the systolic architecture, the Processing Elements are designed by finite state machines. The control block communicates with the first Processing Element (PE) and with the block responsible for the quotient calculation q_i. The overall architecture for systolic multiplier is given in figure 5.4. which consists of four different cells namely

- Right Most Top Most PE.
- Basic Processing Element.
- Left Border PEs $cell_{0,j}$.
- Right Border PEs $cell_{i,0}$.

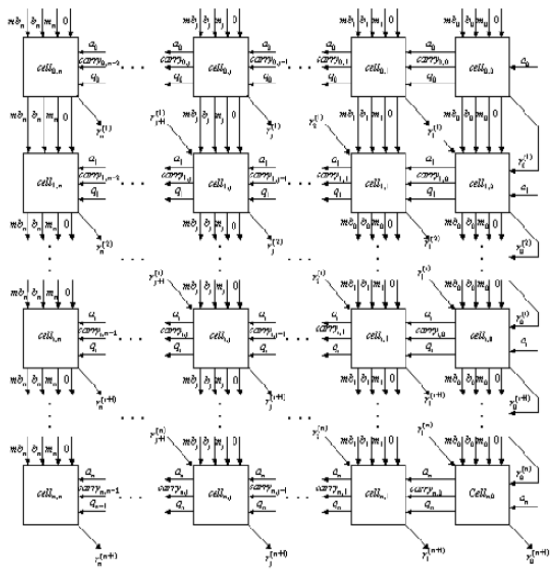


Figure 5.4: Systolic Multiplier.

5.2.1. Right Most Top Most PE.

The first Processing Element (PE) establishes communication with the control block and receives a_i and q_i words at each Montgomery Algorithm iteration. This PE differs from the other Processing Elements because it does not receive any carry signal as input and it discards the first word of the *Sum,* therefore full adder is replaced by the half adder in order to save the area. right most top most PE i.e $cell_{0,0}$ is shown in figure 5.5.

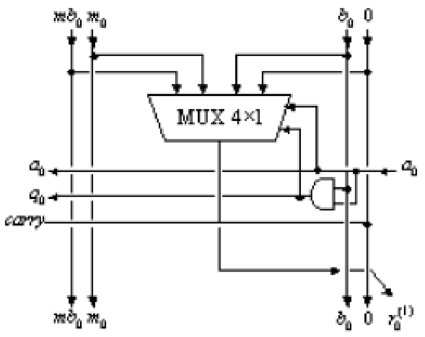


Figure 5.5: Right most top most PE i.e $cell_{0,0}$.

5.2.2. Basic Processing Element.

The other Processing Elements are different from basic PE because they have a word from the S result as output and they also transmit and receive carry signals of the multi-precision multiplications and additions. Each Processing Element is activated by the previous Processing Element when the latter finishes its calculation and sends out its carry signals, which means that the architecture works with a pipeline behavior. Figure 5.6 presents the architecture of the basic Processing Elements.

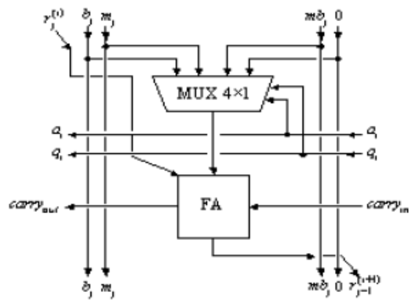


Figure 5.6: Basic PE i.e cell$_{i,j}$.

5.2.3. Left Border PEs cell$_{0,j}$.

In left border PEs sum is calculated using half adder, since previous sum to this cell (i.e initial sum) will be zero.

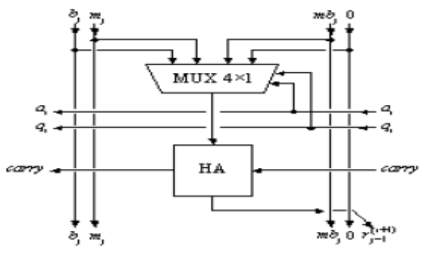


Figure 5.7: Left Border PEs cell$_{0,j}$.

Here carry of the previous cell will be summed up with one of the output from 4X1 MUX, whereas select signal for this MUX will be a_i and q_i. Figure 5.7 presents the architecture of the Left Border PEs cell$_{0,j}$.

5.2.4. Right Border PEs cell$_{i,0}$

In right border PEs initial carry will be zero, therefore sum can be calculated by using half adder instead of full adder. Here sum of the previous cell will be summed up with one of the output from 4X1 MUX, whereas select signal for 4 input MUX will be a_i and b_0.(i.e $r_0^{(i)} \oplus (a_i b_i)$) Figure 5.8 presents the architecture of the Right Border PEs cell$_{i,0}$.

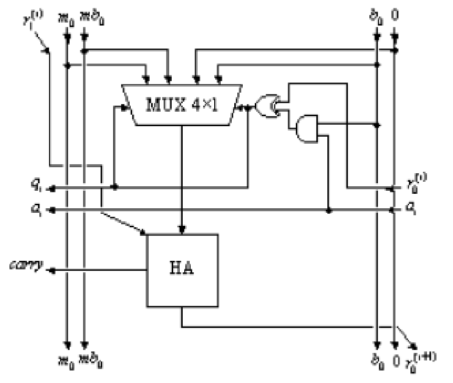


Figure 5.8: Right Border PEs cell$_{i,0}$.

Hardware implementation for two different architectures namely Iterative multiplier and Systolic multiplier is simulated using Model Sim and implemented by Xilinx. The speed of execution and the occupation of the speed is compared for both the architectures. Simulation result for those architecture is discussed in chapter 6.

CHAPTER 6
RESULTS AND DISCUSSIONS

The simulation of this project has been done using MODELSIM XE111 6.2g and XILINX ISE 8.1i.

Modelsim is a simulation tool for programming {VLSI} {ASIC}s, {FPGA}s, {CPLD}s, and {SoC}s. Modelsim provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC.

Xilinx was founded in 1984 by two semiconductor engineers, Ross Freeman and Bernard Vonderschmitt, who were both working for integrated circuit and solid-state device manufacturer Zilog Corp. The Virtex-II Pro, Virtex-4, Virtex-5, and Virtex-6 FPGA families are particularly focused on system-on-chip (SOC) designers because they include up to two embedded IBM PowerPC cores The ISE Design Suite is the central electronic design automation (EDA) product family sold by Xilinx. The ISE Design Suite features include design entry and synthesis supporting Verilog or VHDL, place-and-route (PAR), completed verification and debug using Chip Scope Pro tools, and creation of the bit files that are used to configure the chip.

The simulation results for the implementation of two architectures namely Iterative and Systolic Multiplier using Montgomery modular multiplication algorithm were shown in this section.

6.1 Simulation Result for Iterative Multiplier

6.1.1 Iterative Multiplier First Multiplication

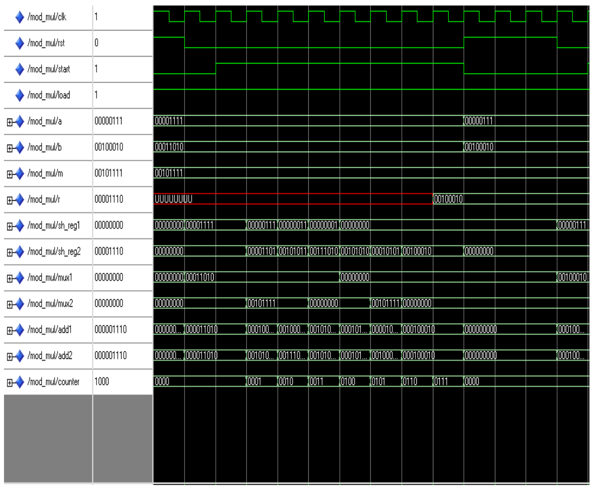


Figure 6.1.1 Simulation result for iterative multiplier behavior during the first multiplication (i)

The simulation of iterative multiplier using Montgomery algorithm with the operands a=15, b=26, and m=47 has been carried out and the behavior of the multiplier during the first modular multiplication is shown in figure 6.1.1 (here signal *step* is not set).

6.1.2 Iterative Multiplier Second Multiplication

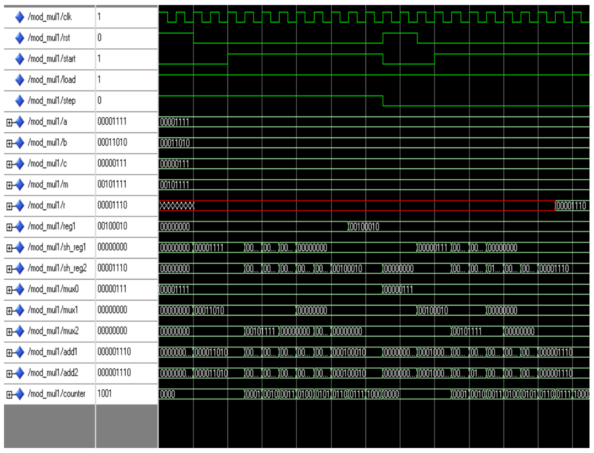


Figure 6.1.2 Simulation result for iterative multiplier behavior during the second multiplication (i)

Figure 6.1.2 shows the results of the Iterative multiplication during the second level of multiplication which finally returns r = 14 (signal *step* is set).

6.1.3 Iterative Multiplier First Multiplication

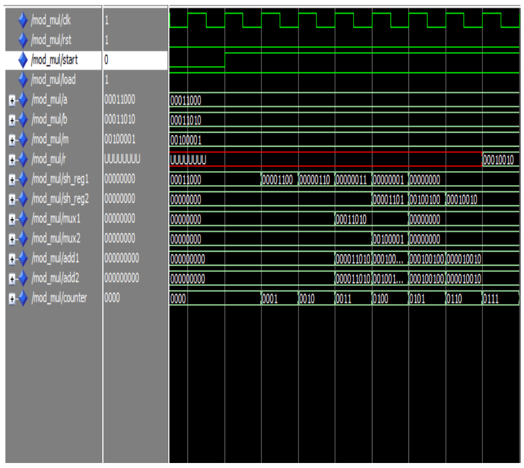


Figure 6.1.3 Simulation result for iterative multiplier behavior during the first multiplication (ii)

The simulation of iterative multiplier using Montgomery algorithm with the operands a=24, b=26, and m=33 has been carried out and the behavior of the multiplier during the first modular multiplication is shown in figure 6.1.3 (here signal *step* is not set).

6.1.4 Iterative Multiplier Second Multiplication

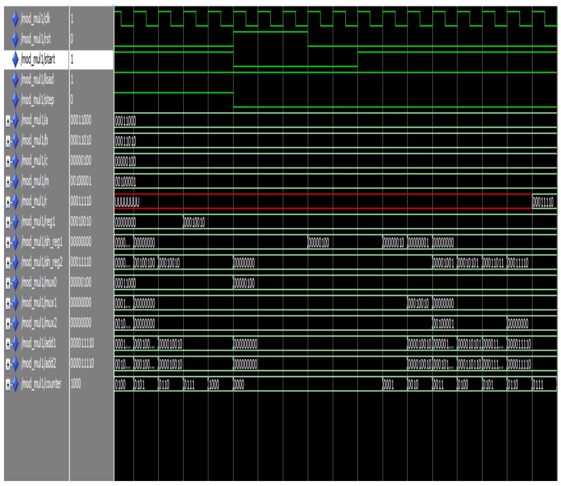


Figure 6.1.4 Simulation result for iterative multiplier behavior during the second multiplication (ii)

Figure 6.1.4 shows the results of the Iterative multiplication during the second level of multiplication which finally returns r = 14 (signal *step* is set).

6.2 Simulation Result for Systolic Multiplier

6.2.1 Systolic Multiplier Simulation Result for Operands (23,27,33)

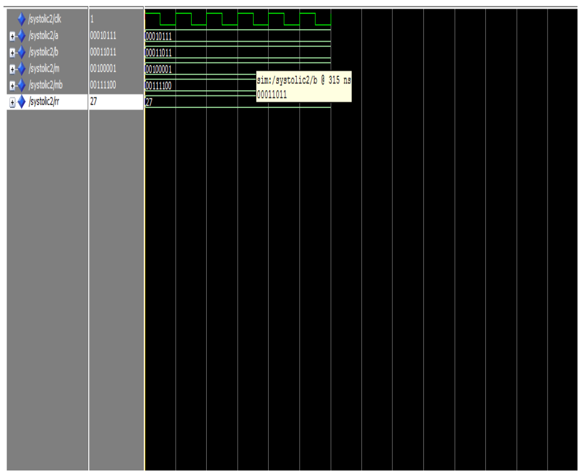

/systolic2/clk 1
/systolic2/a 00010111 00010111
/systolic2/b 00011011 00011011
/systolic2/m 00100001 00100001
/systolic2/mb 00111100 00111100
/systolic2/rr 27 27
sim:/systolic2/b @ 315 ns
00011011

Figure 6.2.1: Simulation waveform for systolic multiplier (i)

The simulated output waveform for systolic Montgomery multiplier with the operands a=23, b=27, m=33 is shown in figure 6.2.1 which also returns r = 27.

6.2.1 Systolic Multiplier Simulation Result for Operands (53,57,61)

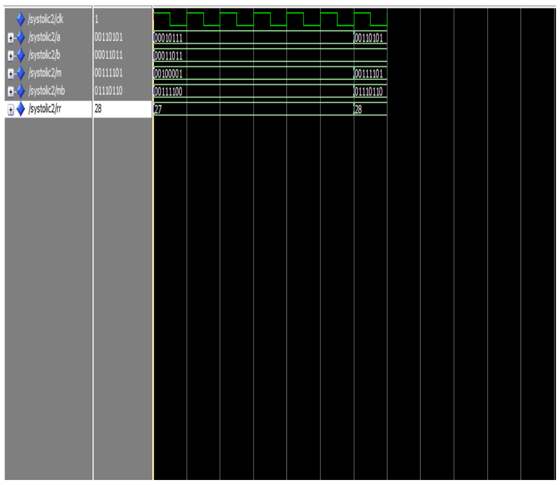

/systolic2/clk 1
/systolic2/a 00110101 00010111 00110101
/systolic2/b 00011011 00011011
/systolic2/m 00111101 00100001 00111101
/systolic2/mb 01110110 00111100 01110110
/systolic2/rr 28 27 28

Figure 6.2.2: Simulation waveform for systolic multiplier (ii)

The simulated output waveform for systolic Montgomery multiplier with the operands a=53, b=57, m=61 is shown in figure 6.2.2 which also returns r = 28.

6.3 Design summary:

6.3.1 Iterative multiplier:

Timing Summary:

Speed Grade: -5

Minimum period: 11.222ns

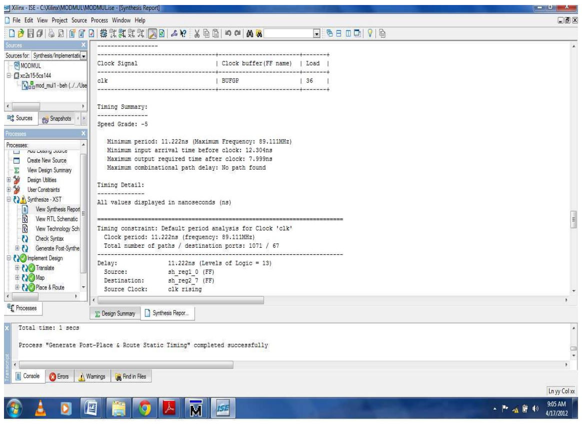
Maximum Frequency: 89.111MHz

Minimum input arrival time before clock: 12.304ns

Maximum output required time after clock: 7.999ns

Total equivalent gate count for design: 793

Additional JTAG gate count for IOBs: 2,160



```
-----------------
-----------------------------------+------------------------+--------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+--------+
clk                                | BUFGP                  | 36    |
-----------------------------------+------------------------+--------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 11.222ns (Maximum Frequency: 89.111MHz)
   Minimum input arrival time before clock: 12.304ns
   Maximum output required time after clock: 7.999ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 11.222ns (frequency: 89.111MHz)
  Total number of paths / destination ports: 1071 / 67
-------------------------------------------------------------------------
Delay:               11.222ns (Levels of Logic = 13)
  Source:            sh_reg1_0 (FF)
  Destination:       sh_reg2_7 (FF)
  Source Clock:      clk rising
```

```
Total time: 1 secs

Process "Generate Post-Place & Route Static Timing" completed successfully
```

Figure 6.3.1: Synthesis report for Iterative Multiplier

6.3.2 Systolic Multiplier:

Timing Summary:

Speed Grade: -5

Minimum period: 4.791ns

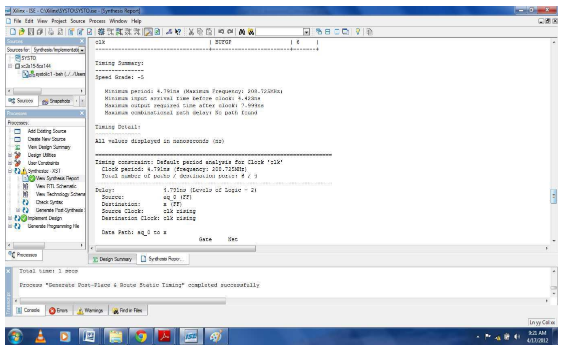
Maximum Frequency: 208.725MHz

Minimum input arrival time before clock: 4.423ns

Maximum output required time after clock: 7.999ns

Total equivalent gate count for design: 1,341

Additional JTAG gate count for IOBs: 1,920

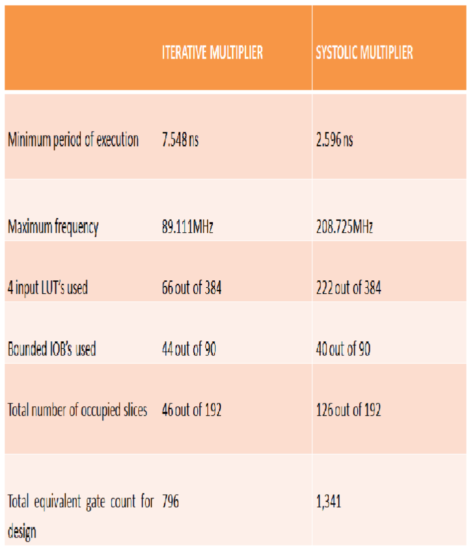


```
clk                                | BUFGP                  | 6     |
-----------------------------------+------------------------+--------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 4.791ns (Maximum Frequency: 208.725MHz)
   Minimum input arrival time before clock: 4.423ns
   Maximum output required time after clock: 7.999ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 4.791ns (frequency: 208.725MHz)
  Total number of paths / destination ports: 6 / 4
-------------------------------------------------------------------------
Delay:               4.791ns (Levels of Logic = 2)
  Source:            aq_0 (FF)
  Destination:       x (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: aq_0 to x
                                Gate     Net
```

```
Total time: 1 secs

Process "Generate Post-Place & Route Static Timing" completed successfully
```

Figure 6.3.2: Synthesis report for Systolic Multiplier

6.4. Comparison of Iterative multiplier vs Systolic multiplier:



	ITERATIVE MULTIPLIER	SYSTOLIC MULTIPLIER
Minimum period of execution	7.548 ns	2.596 ns
Maximum frequency	89.111MHz	208.725MHz
4 input LUT's used	66 out of 384	222 out of 384
Bounded IOB's used	44 out of 90	40 out of 90
Total number of occupied slices	46 out of 192	126 out of 192
Total equivalent gate count for design	796	1,341

Table 6.1 Comparison of Iterative multiplier vs Systolic multiplier:

From the synthesis estimate, the minimum clock period for iterative multiplier was found to be **7.548ns**, for which the maximum clock frequency is **89.111Mhz** and for systolic multiplier the minimum clock period was found to be **2.596ns** for which maximum clock frequency is **208.725Mhz**.Execution time for Systolic multiplier is less than Iterative multiplier whereas area occupied will be more .

6.5. Graphical Representation of Speed and Area Comparison

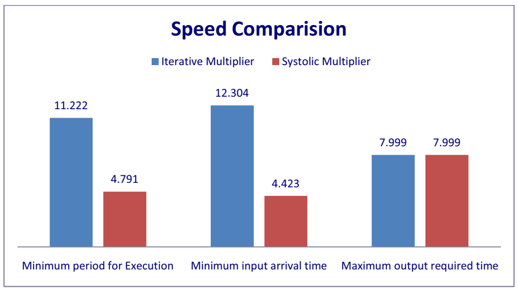


Figure 6.5.1: Speed Comparison

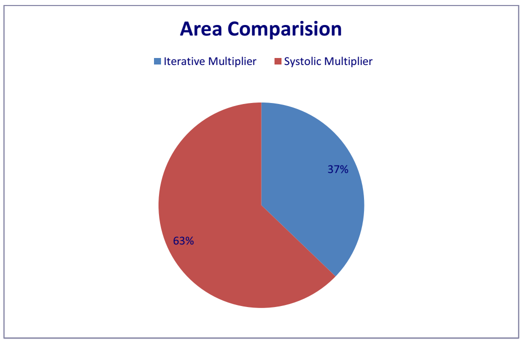


Figure 6.5.2: Area Comparison

CHAPTER 7
CONCLUSION AND FUTURE SCOPE

In RSA algorithm great consequence factor which affects its performance is modular multiplication. Here efficient implementation of modular multiplication is implemented using Montgomery technique. Each module is coded with VHDL. The VHDL code of modular multiplication is developed and synthesized using Xilinx ISE and verified. From the synthesis estimate, the minimum clock period for iterative multiplier was found to be **7.548ns**, for which the maximum clock frequency is **89.111MHz** and for systolic multiplier the minimum clock period was found to be **2.596ns** for which maximum clock frequency is **208.725MHz**. Execution time for Systolic Multiplier is less than that of Iterative Multiplier and area occupied will be more in Systolic than Iterative Multiplier so depending on the application requirement the choice of the architecture can be made for high speed or less space occupation.

FUTURE SCOPE

Future scope of this project is to develop the various efficient architectures for Montgomery modular technique which supports RSA algorithm

BIBLIOGRAPHY

[1] Jizhong Liu, Jinming Dong,"Design and implementation of an efficient RSA crypto-processor",IEEE2010

[2] Christof Paar • Jan Pelzl, "Understanding Cryptography" Springer,2010.

[3] Parth Mehta, Dhanashri Gawali, "Conventional versus Vedic Mathematical Method for Hardware Implementation of a Multiplier",2009 International Conference on Advances in Computing, Control,and Telecommunication Technologies, Trivandrum, Kerala, India,December 2009, pp. 640-642, 2009.

[4] N. Ncdjah, M. M. L. De, "Three hardware architectures for the binary modular exponentiation: sequential, parallel, and systolic," IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 2006, 53(3), pp.627-633.

[5] R.V.Kamala, M.B.Srinivas," High-Throughput Montgomery Modular Multiplication"IEEE 2006.

[6] William Stallings, "Cryptography and Network Security: Principles and Practices." 3rd edition, 2003.

[7] Navabi, Z., *VHDL - Analysis and modeling of digital systems*, McGraw Hill, Second Edition,1998.

[8] C. D. Walter, "Systolic modular multiplication," IEEE Transactions on Computers, 1993,42(3), pp.376-378.

[9] Peter J Ashenden & Jim Lewis, "The Designer's Guide to VHDL",Morgan Kaufmann Publishers.