IMPLEMENTATION DESIGN FOR FASTER ENCRYPTION IN RSA

Suhrid Barthakur, Partha Pritam Paul, Jasbir Singh Birdi, Anirban Chatterjee
Department of Computer Science and Engineering, NIT Silchar

***Abstract*—The paper aims to propose an implementation design/procedure for faster encryption process mainly through optimizing the modular exponentiation process for large key of sizes 1024 bits or more.The design provides a recoding scheme and makes use of montgomery multiplication for reducing the total no. of operations and faster modular multiplication respectively.The design is based on parallel processing by two processor for achieving optimal performance.In addition to this a systolic array design is implemented to provide potentially higher throughput for a large number of computations.Experimental analysis and results are beyond the scope of this paper.**

Keywords: Parallel processing, Systolic arrays, Montgomery reduction.

I. Introduction [1].

Encryption is what guarantees the security of sensitive information. It provides the mechanisms in information confidentiality, functioned with digital signature, authentication, secret sub-keeping, system security and etc. Therefore, the purpose of adopting encryption techniques is to ensure the information confidentiality, integrity and certainty, prevent information from tampering, forgery and counterfeiting.

At present, the best known and most widely used public key system is RSA, which was first proposed in a paper named “A method for obtaining digital signatures and public-key cryptosystems” by RL Rivest et al. in 1978.

It is an asymmetric (public key) cryptosystem based on number theory, which is a block cipher system. Its security is based on the difficulty of the large number prime factorization, which is a well-known mathematical problem that has no effective solution. RSA public key cryptosystem is one of the most typical ways that most widely use for public key cryptography in encryption and digital signature standards.

This paper is designed to propose a design for faster encryption implementation which is mainly determined by the computation speed of performing the modular exponentiation

$$C = M^E modN$$

specially when E and N is very large which makes the public key pair (E,N).

For better security it has been found that the large size key are hard to factorize ,and the integrity of the RSA system depends solely on factorization of N which is the product of two large prime numbers. As it can seen from the RSA algorithm[4]:

1) Choose two large random primes p and q.
2) Generate N=p*q.
3) Determine the value of random large integer E such that GCD(E,(p-1)*(q-1))=1.
4) Determine D as the multiplicative inverse of E mod ((p-1)*(q-1)).
5) Generate public key (E,N) and publish it,and generate private key (D,N).

$$Ciphertext : C = M^E modN.$$

One of the well known algorithm for computing M^E is exponentiation by repeated squaring and multiplication and called the binary algorithm

II. Related Work

Our method of the faster encryption in RSA is based upon some of the previous works and experiments done on RSA to improve the basic encryption implementation techniques through faster modular exponentiation . Some of them are enlisted here.

D.E Knuth in his book “Semi numerical Algorithm”[2] suggested the binary algorithm which exponentiation by repeated squaring and multiplication. Computing M^E mod N using this algorithm requires 2 log E modulo multiplications in the worst case where N is the product of two very large integers and MN, EN. The total no. of modulo in the this algorithm is n+E1,where n is the number of bits in E and E1 is the number of 1’s in binary representation of E.The suggested algorithm makes it faster by recoding and reducing the value of E1

Chang N. Zhang and Herold L. Martin[3] suggested a parallel algorithmic design based on the binary algorithm in their paper for fast modular exponentiation which made use of parallel processing designs to make the binary algorithm faster. The suggested algorithm here, makes use of these techniques to further optimize the modulo multiplications with Montgomery reduction algorithm.

The Montgomery multiplication is used for faster modular exponentiation[5]. The well known algorithm for computing ME is exponentiation by repeated squaring and multiplication and called the binary algorithm. Computing M^E mod(N) by this algorithm requires 2—log E — modulo multiplications in the worst case where N is the product of two large prime numbers.

III. Proposed Work

The proposed implementation design aims to make the rsa cryptosystems faster through faster encryption and decryption process .The encryption is slow process when the key size is large like 1024 bits ,2048 bits or greater (for better security) ,due to computation of modular exponentiation C=M^Emod N, which becomes expensive for large public key (E,N). Thereby the design makes the modular exponentiation faster by making use of parallel processing and montgomery multiplication technique. Also a corresponding systolic array design is developed[2].

The montgomery multiplication is used for faster modular exponentiation. The well known algorithm for computing ME is exponentiation by repeated squaring and multiplication and called the binary algorithm. Computing M^E mod(N) by this algorithm requires 2—log E — modulo multiplications in the worst case where N is the product of two large prime numbers.

IV. Modular Exponentiation Algorithm

A. Recoding

The proposed technique in order to compute C= M^Emod N involves representing E in binary form i.e

$$E = \sum_{i=0}^{n-1} e_i 2^i$$

,where n is number of bits in E and $e_i \in \{0, 1\}$ similar to binary algorithm[6].

Now it known that the binary algorithm requires n + E1 total operations,where E1 is no of 1's representation in binary representation of E and n is the number of bits in E.Thus we need to reduce the value of E1 in order to make it faster.

To achieve this define a redundant expression of E say E'(E= E') which is given as:

$$E' = \sum_{i=0}^{n-1} e_i^* 2^i$$

where $e_i^* \in \{0, 1, -1\}$ and is a redundant representation of E in base of 0 ,1 ,-1.

Let us represent -1 in the redundant representation as $\overline{1}$.

For example E=11110=1000$\overline{1}$0=100000-10 Thus as in the above example used reduce the number of non zero digit with the help of the redundant expression. For any given integer E, E¡2^n, the time complexity of computing M^E(mod M) is $(n + |\frac{n}{2}| + 1)$ Tn by applying the recording algorithm and redundant binary algorithm.

Suppose E is the integer randomly distributed in the range of 2^{n-1} and $2^n - 1$. It is easy to figure out that the average number of 1's for the integers between 2^{n-1} and $2^n - 1$ is $(\frac{n}{2} + 1)$.

The average number[11] of non-zero's after applying the recoding algorithm for the integers distributive in the range of 2^{n-1} and $2^n - 1$ can be calculated by the following equation:

$$n_2 = \frac{\sum_{i=0}^{2^{n-1}-1} f(i)}{2^{n-1}}$$

Where f(i) is the number of non-zero's after execution the recoding algorithm to the integer $2^{n-1} + i(i = 0...2^{n-1} - 1)$. Table 1 shows the comparison of the number of non-zero elements between the original integer and the recoded one.

Table I
Comparison of Number of Non Zero's Between Binary Representation and Redundant Representation.

n number of bits	no. of 1's before recoding n_1	no. of non-zeroes after recoding n_2	$\frac{n_1 - n_2}{n_1}$
10	6	4.44	26.0
15	8.5	6.11	28.1
20	11	7.77	29.4
25	13.5	9.44	30.8

Formally it can be proved that the ratio of $\frac{(n_1 - n_2)}{n_2}$ is greater than $\frac{1}{3}$ when $n \geq 25$. Notice that in practice, E is very large, and one may need to compute a series of computations of $M_i^E(modN)$, i=1, 2...m. Therefore, the total saving is $m * (n_1 - n_2)$ on average[9].

B. Montgomery Multiplication

Now in order to perform a modular exponentiation we use the montgomery multiplication .Montgomery modular multiplication[8], more commonly referred to as Montgomery multiplication, is a method for performing fast modular multiplication.For computations using montgomery multiplication , first perform modular exponentiation in montgomery form ,and then convert back the obtained result out of montgomery form to obtain the required result i.e

$$C = M^E modN$$

For montgomery multiplication it is defined as, given an integer $a < N$, where N is the k-bit modulus, A is said to be its N-residue with respect to R if ,

$$A = a * R(modN), whereR = 2^k$$

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

$$B = b * R(modN)$$

Ordinary Domain	$\Longleftrightarrow$	Montgomery Domain
X	$\leftrightarrow$	$X' = X \cdot 2^n \pmod{M}$
Y	$\leftrightarrow$	$Y' = Y \cdot 2^n \pmod{M}$
XY	$\leftrightarrow$	$(X \cdot Y)' = X \cdot Y \cdot 2^n \pmod{M}$

Figure 1. Mapping to montgomery form.

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

$$Z = A * B * R'(modN),$$

where R' is the inverse of R modulo N. i.e $RR' \equiv 1(modN)$.

Since $GCD(N, R') = 1$,extended Euclidean algorithm can efficiently determine integers R' and N' that satisfy Bézout's identity:

$$0 < R' < N, 0 < N' < RandRR' - NN' = 1.$$

The montgomery reduction algorithm[10] is based on the fact that the computation of $A * B * R' modN$ can be done very efficiently by the montgomery reduction algorithm REDUCE as shown below .

```
1:  function REDUCE(X)
2:      begin
3:          $q \Leftarrow (XmodR)N'modR$;
4:          $a \Leftarrow (X + qN)/R$;
5:          if $a > N$ then
6:              $a \Leftarrow a - N$
7:          end if
8:          return a;
9:      end
10: end function
```

This reduction algorithm[5] is used for converting in and out of montgomery form and calculating the montgomery product.And hence can be used for modular exponentiation algorithm.

C. The Proposed Algorithm

For computing $C = M^E modN$,repeat the following steps of the algorithm.It is assumed that R', N' and R are predetermined as required for montgomery reduction[2].

1) Input M as integer , E as E' i.e redundant representation of binary expression for E and N as integer value
2) Take a variable result to store the temporary result for each iteration .Initialize result as the montgomery form of 1.Thus we initialize

$$result = R(modN), where\, gcd(R, N) = 1\, and\, R > N.$$

3) If k is the no of bits in E then starting from most significant bit in 'E' i.e k-1 th bit to the least significant bit,and for each bit i from k-1 to 0 do the following:
   a) result=REDUCE((result.R mod N)*(result.R mod N))
   result= (result.R') mod N
   b) **If the ith bit is 1 then set**
   result=REDUCE((result.R mod N)*(M.R mod N))
   result=result.R' mod N

   OR

   If the ith bit is $\bar{1}$ then set
   result=REDUCE((result.R mod N)*(M^{-1}.R mod N))
   result=result.R' mod N

   OR

   else if the ith bit is 0 then set result=result
4) After completion of all iterations return result, which is in montgomery form and we convert it out of montgomery form to obtain the final result of encryption

$$C = REDUCE(result.R\ modN) = M^E modN.$$

The algorithm computes modular exponentiation in O((T(n))(n+E1)) where T(n) is time complexity of Montgomery reduction algorithm,n is the no. of bits in redundant representation of E and E1 is no of non zero elements in redundant representation.

This is much faster in comparison to the binary algorithm which has complexity of O(M(n)(n +Z)),n being the number of bits in E, Z being the no of 1's in the binary form of E. M(n) is time complexity for performing modular multiplication. Since $M(n) > T(n)$ and $E1 < Z$.

V. Adding Parallelism and Systolic Array Design

A. Parallel Processing

The Step 3 in the mentioned algorithm can be made faster by using two parallel processor say A,B for faster computations[14].

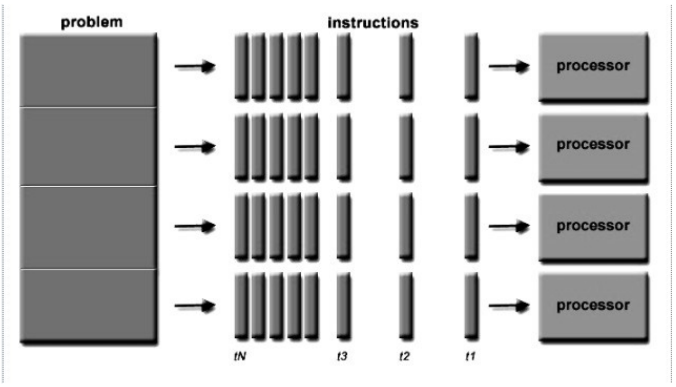


Figure 2. Parallel processing model to solve a problem.

By using two parallel processor the computations can be made twice as fast than usual.

The following is the modification to the step 3(a) and 3(b)

Step 3(a):
[Processor A]
result=REDUCE((result.R mod N)*(result.R mod N))
[Processor A]
result= (result.R') mod N

AND

At time same time the processor B will simultaneously compute...

Step 3(b):
If the ith bit is 1 then set
[Processor B]
result=REDUCE((result.R mod N)*(M.R mod N))
[Processor B]
result=result.R' mod N

OR

If the ith bit is $\bar{1}$ then set

[Processor B]
result=REDUCE((result.R mod N)*(M^{-1}.R mod N))
[Processor B]
result=result.R' mod N

OR

else if the i^th bit is 0 then set
[Processor B]
result=result

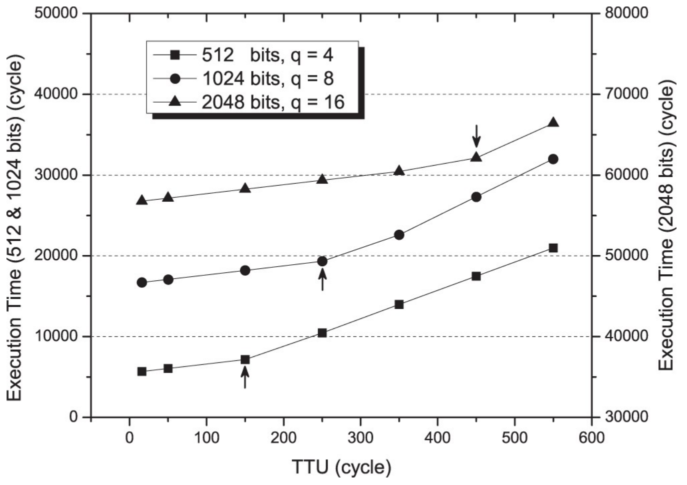


Figure 3. Execution time of Montgomery multiplication on multi core system. q:no. of cores.

Here instead of computing step 3(a) and 3(b) sequentially one after the other we parallelly compute their value using two different processors thereby saving time and overload involved when using a single processor[8].

Thus it makes the time of computation get reduced by a factor of 2.

B. Systolic Array Design

It is often in RSA cryptosystems that the values of private key and public key are kept same for encryption and decryption of multiple messages say[3]

$$M_j^E modN forj = 1, 2, ...m.$$

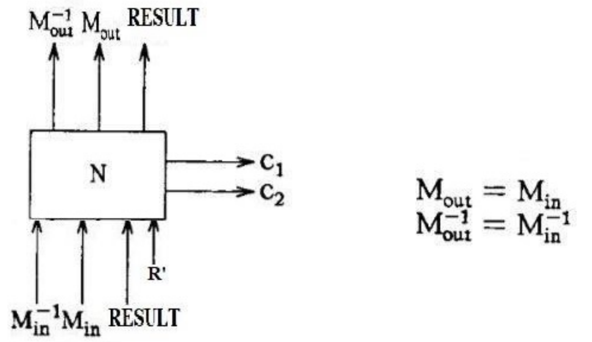
In this section[9] a systolic array design is proposed for the corresponding algorithm. Proposed design is a double linear systolic architecture for implementing the algorithm .This structure consists of 2n processing elements,n is the no. of bits in E and takes advantage of using the pipeline structure to reduce average computation time.The proposed double linear systolic array design can compute $M^E modN$ in asymptotically constant time on average.

Each of the processing elements has two control inputs to perform three different operation depending on whether the current bit in E is 1,0 or $\overline{1}$. Here $M^{-1}.RmodN$ and $M.RmodN$ is precomputed .The processing unit is as shown in figure below .C_1 and C_2 are the control inputs which are set to either 10,01 or 00 depending on the i^th bit in E that is currently being processed[14].

$$C_1C_2 = 01\ if\ e_i^* = 1$$

$$C_1C_2 = 10\ if\ e_i^* = 1$$

$$C_1C_2 = 00\ if\ e_i^* = 0$$



M_{out}^{-1} M_{out} RESULT
N
C_1
C_2
R'
M_{in}^{-1} M_{in} RESULT
$M_{out} = M_{in}$
$M_{out}^{-1} = M_{in}^{-1}$

Figure 4. Basic Function of Processing Element.

The processing elements perform the montgomery reduction and multiplication depending on the control inputs[13]:

.

If the i^th bit is $\overline{1}$ i.e C_1C_2=10 then the processing element perform:
result=REDUCE((result.R mod N)*(M^{-1}.R mod N))
result=result.R' mod N

Or if the i^th bit is 1 i.e C_1C_2=01 then,
result=REDUCE((result.R mod N)*(M.R mod N))
result=result.R' mod N

Otherwise, if the i^th bit is 0 i.e C_1C_2=00 then,
DO NOTHING

The figure below shows the double systolic array design having a pipeline network of processing elements to compute $M_j^E modN$ in montgomery form.

The network consists of 2n processing elements ,the left side computes:
result=REDUCE((result.R mod N)*(result.R mod N))
result= (result.R') mod N

which corresponds to processor A in the proposed algorithm and is synchronized to the right side using a common clock. The control inputs are always kept 10 for left as they need to perform a fixed operation ,while for the right side corresponding to Processor B , the control varies depend on e_i bit in E under consideration[16].

The starting input to the left side is always R mod N and R' while the right side is fetched a sequence of M's (multiple messages) in their montgomery form and also M^{-1} in its corresponding montgomery form. The final RESULT is then return by the top right processing element (Processor B) which

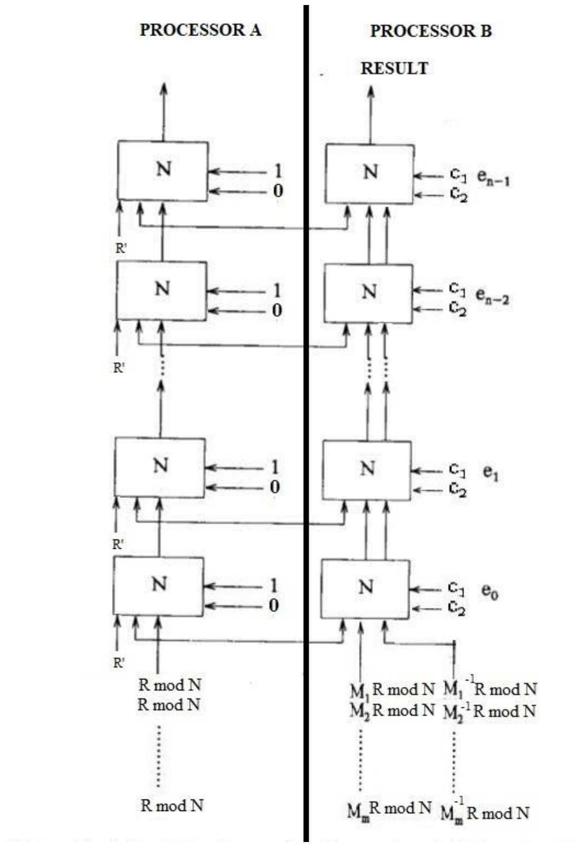


Figure 5. Systolic Array Implementation for computing $M_j^E modN$ in montgomery form.

is in montgomery form and from this result we obtain the final encrypted text by reverting out of the montgomery form.

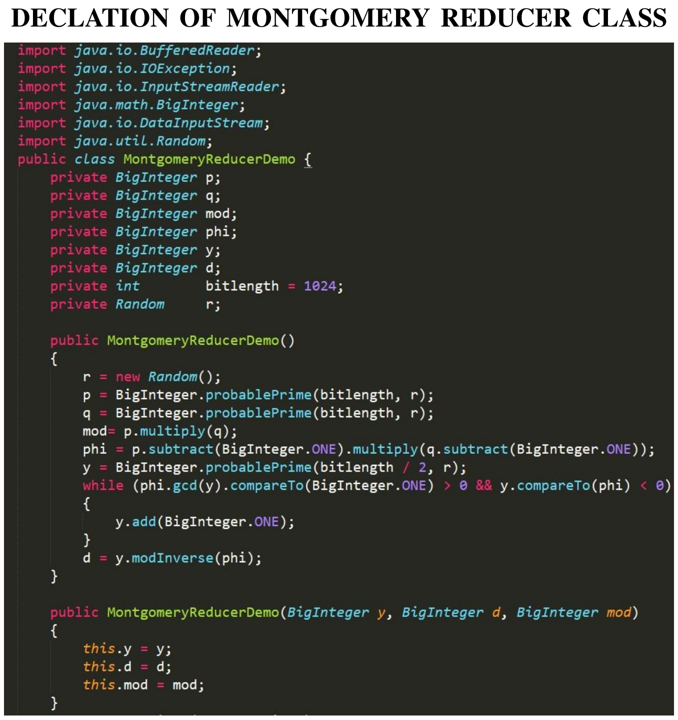
$$C = M^E modN = REDUCE(RESULT.RmodN)$$

It takes n time units delay to get the first result, from the time of the first message sent in and then the following results will be obtained in subsequent each time unit delay.The average time delay per message,$T(n) = O(n(m+n-1)/m)$

Now if m=1 then $T(n) = O(n^2)$ which is the case of implementing by two processing elements[3].

If m=n-1 then T(n)=O(2n) and if $m >> n$ then naturally T(n)=O(1), and therefore this network achieves constant time delay to compute each $M_j^E modN$ (j=1,2,....m) on average if m (no. of messages) is big enough.

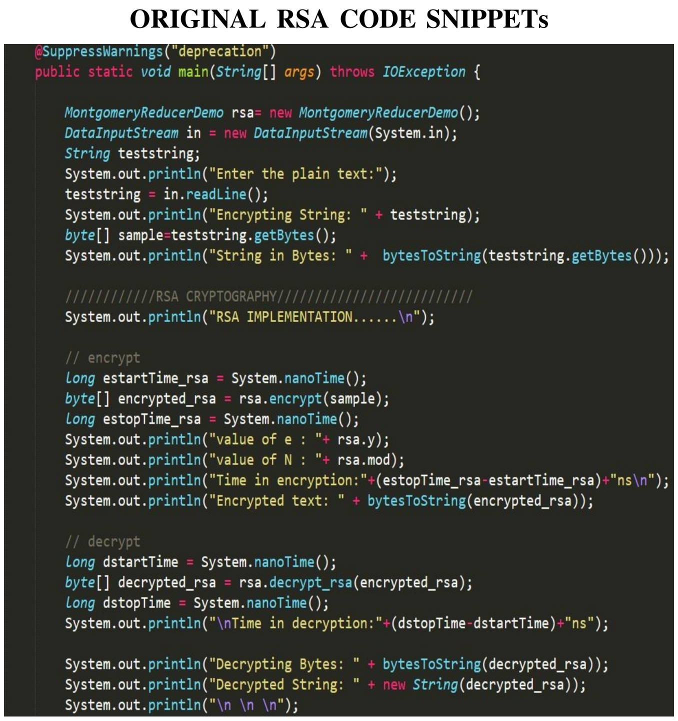
VI. Code Implementation and Results



DECLATION OF MONTGOMERY REDUCER CLASS

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.io.DataInputStream;
import java.util.Random;
public class MontgomeryReducerDemo {
    private BigInteger p;
    private BigInteger q;
    private BigInteger mod;
    private BigInteger phi;
    private BigInteger y;
    private BigInteger d;
    private int        bitlength = 1024;
    private Random     r;

    public MontgomeryReducerDemo()
    {
        r = new Random();
        p = BigInteger.probablePrime(bitlength, r);
        q = BigInteger.probablePrime(bitlength, r);
        mod= p.multiply(q);
        phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
        y = BigInteger.probablePrime(bitlength / 2, r);
        while (phi.gcd(y).compareTo(BigInteger.ONE) > 0 && y.compareTo(phi) < 0)
        {
            y.add(BigInteger.ONE);
        }
        d = y.modInverse(phi);
    }

    public MontgomeryReducerDemo(BigInteger y, BigInteger d, BigInteger mod)
    {
        this.y = y;
        this.d = d;
        this.mod = mod;
    }
```

Figure 6.

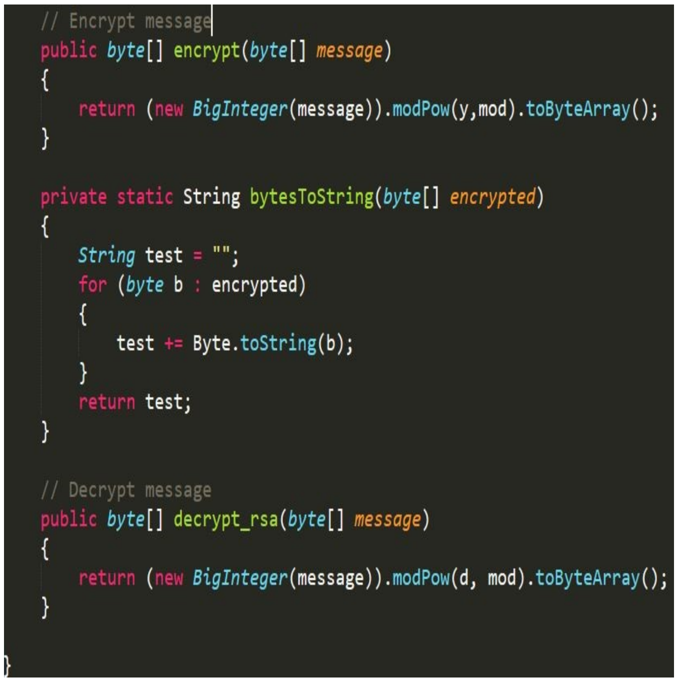


ORIGINAL RSA CODE SNIPPETs

```
@SuppressWarnings("deprecation")
public static void main(String[] args) throws IOException {

    MontgomeryReducerDemo rsa= new MontgomeryReducerDemo();
    DataInputStream in = new DataInputStream(System.in);
    String teststring;
    System.out.println("Enter the plain text:");
    teststring = in.readLine();
    System.out.println("Encrypting String: " + teststring);
    byte[] sample=teststring.getBytes();
    System.out.println("String in Bytes: " +  bytesToString(teststring.getBytes()));

    ///////////RSA CRYPTOGRAPHY////////////////////////
    System.out.println("RSA IMPLEMENTATION......\n");

    // encrypt
    long estartTime_rsa = System.nanoTime();
    byte[] encrypted_rsa = rsa.encrypt(sample);
    long estopTime_rsa = System.nanoTime();
    System.out.println("value of e : "+ rsa.y);
    System.out.println("value of N : "+ rsa.mod);
    System.out.println("Time in encryption:"+(estopTime_rsa-estartTime_rsa)+"ns\n");
    System.out.println("Encrypted text: " + bytesToString(encrypted_rsa));

    // decrypt
    long dstartTime = System.nanoTime();
    byte[] decrypted_rsa = rsa.decrypt_rsa(encrypted_rsa);
    long dstopTime = System.nanoTime();
    System.out.println("\nTime in decryption:"+(dstopTime-dstartTime)+"ns");

    System.out.println("Decrypting Bytes: " + bytesToString(decrypted_rsa));
    System.out.println("Decrypted String: " + new String(decrypted_rsa));
    System.out.println("\n \n \n");
```

Figure 7. Original RSA implementation in main function

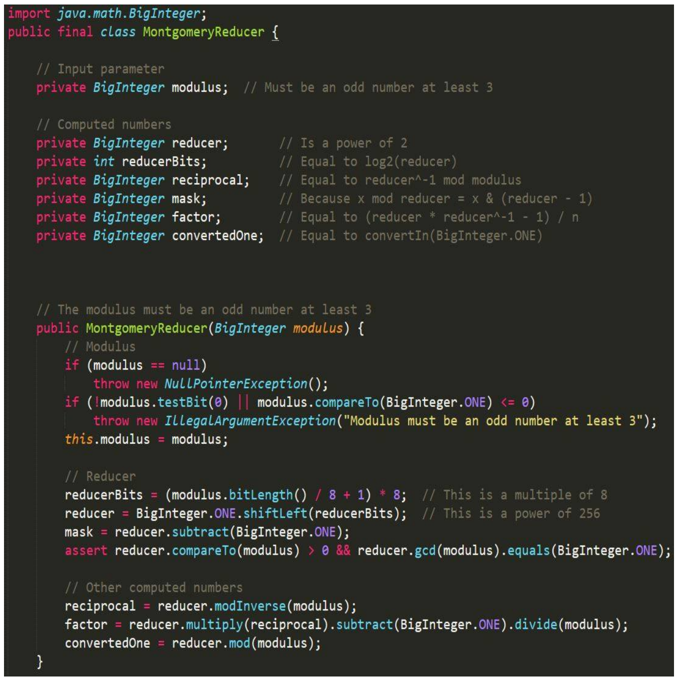
ENCRYPTION AND DECRYPTION FUNCTIONS



```java
    // Encrypt message
    public byte[] encrypt(byte[] message)
    {
        return (new BigInteger(message)).modPow(y,mod).toByteArray();
    }

    private static String bytesToString(byte[] encrypted)
    {
        String test = "";
        for (byte b : encrypted)
        {
            test += Byte.toString(b);
        }
        return test;
    }

    // Decrypt message
    public byte[] decrypt_rsa(byte[] message)
    {
        return (new BigInteger(message)).modPow(d, mod).toByteArray();
    }

}
```

Figure 8. Encryption and Decryption function in original RSA

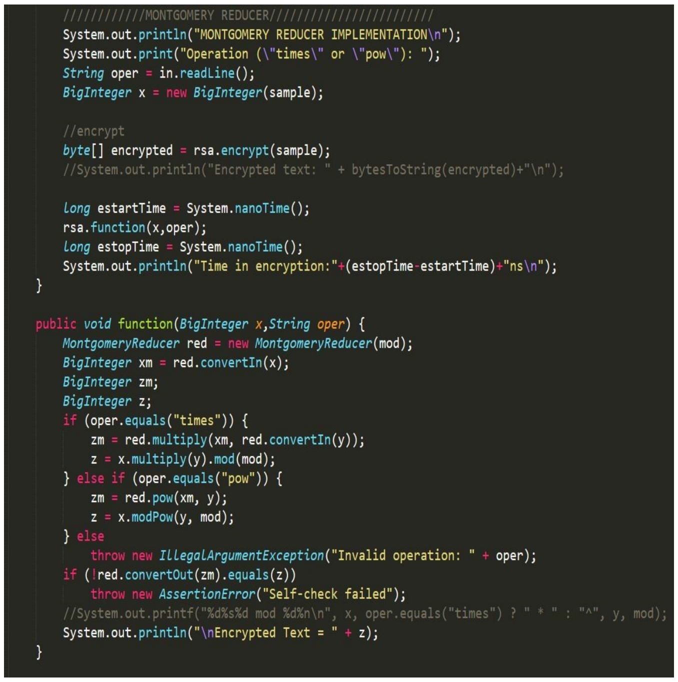
MONTGOMERY MULTIPLICATION



```java
import java.math.BigInteger;
public final class MontgomeryReducer {

    // Input parameter
    private BigInteger modulus;  // Must be an odd number at least 3

    // Computed numbers
    private BigInteger reducer;       // Is a power of 2
    private int reducerBits;          // Equal to log2(reducer)
    private BigInteger reciprocal;    // Equal to reducer^-1 mod modulus
    private BigInteger mask;          // Because x mod reducer = x & (reducer - 1)
    private BigInteger factor;        // Equal to (reducer * reducer^-1 - 1) / n
    private BigInteger convertedOne;  // Equal to convertIn(BigInteger.ONE)



    // The modulus must be an odd number at least 3
    public MontgomeryReducer(BigInteger modulus) {
        // Modulus
        if (modulus == null)
            throw new NullPointerException();
        if (!modulus.testBit(0) || modulus.compareTo(BigInteger.ONE) <= 0)
            throw new IllegalArgumentException("Modulus must be an odd number at least 3");
        this.modulus = modulus;

        // Reducer
        reducerBits = (modulus.bitLength() / 8 + 1) * 8;  // This is a multiple of 8
        reducer = BigInteger.ONE.shiftLeft(reducerBits);  // This is a power of 256
        mask = reducer.subtract(BigInteger.ONE);
        assert reducer.compareTo(modulus) > 0 && reducer.gcd(modulus).equals(BigInteger.ONE);

        // Other computed numbers
        reciprocal = reducer.modInverse(modulus);
        factor = reducer.multiply(reciprocal).subtract(BigInteger.ONE).divide(modulus);
        convertedOne = reducer.mod(modulus);
    }
```

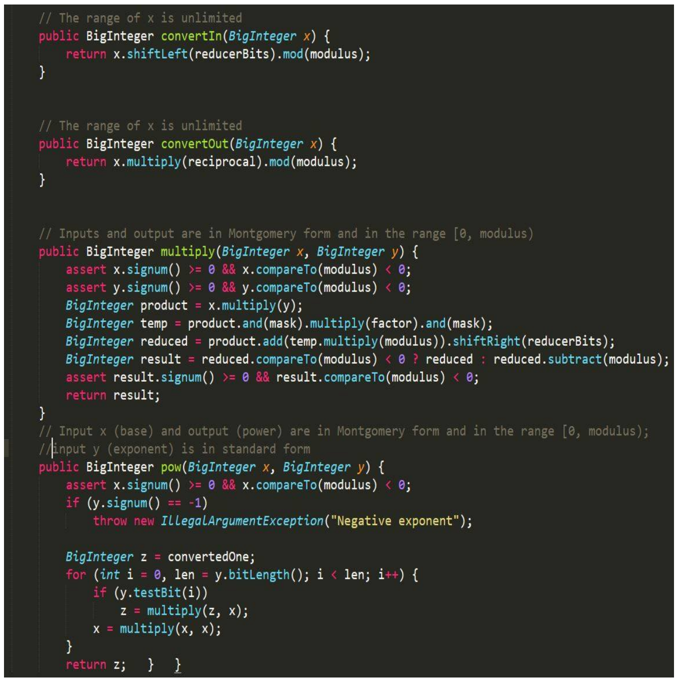
Figure 10. Montgomery Reduction code

MONTGOMERY CODE



```java
        ////////////MONTGOMERY REDUCER//////////////////////////
        System.out.println("MONTGOMERY REDUCER IMPLEMENTATION\n");
        System.out.print("Operation (\"times\" or \"pow\"): ");
        String oper = in.readLine();
        BigInteger x = new BigInteger(sample);

        //encrypt
        byte[] encrypted = rsa.encrypt(sample);
        //System.out.println("Encrypted text: " + bytesToString(encrypted)+"\n");

        long estartTime = System.nanoTime();
        rsa.function(x,oper);
        long estopTime = System.nanoTime();
        System.out.println("Time in encryption:"+(estopTime-estartTime)+"ns\n");
    }

    public void function(BigInteger x,String oper) {
        MontgomeryReducer red = new MontgomeryReducer(mod);
        BigInteger xm = red.convertIn(x);
        BigInteger zm;
        BigInteger z;
        if (oper.equals("times")) {
            zm = red.multiply(xm, red.convertIn(y));
            z = x.multiply(y).mod(mod);
        } else if (oper.equals("pow")) {
            zm = red.pow(xm, y);
            z = x.modPow(y, mod);
        } else
            throw new IllegalArgumentException("Invalid operation: " + oper);
        if (!red.convertOut(zm).equals(z))
            throw new AssertionError("Self-check failed");
        //System.out.printf("%d%s%d mod %d%n\n", x, oper.equals("times") ? " * " : "^", y, mod);
        System.out.println("\nEncrypted Text = " + z);
    }
```

Figure 9.

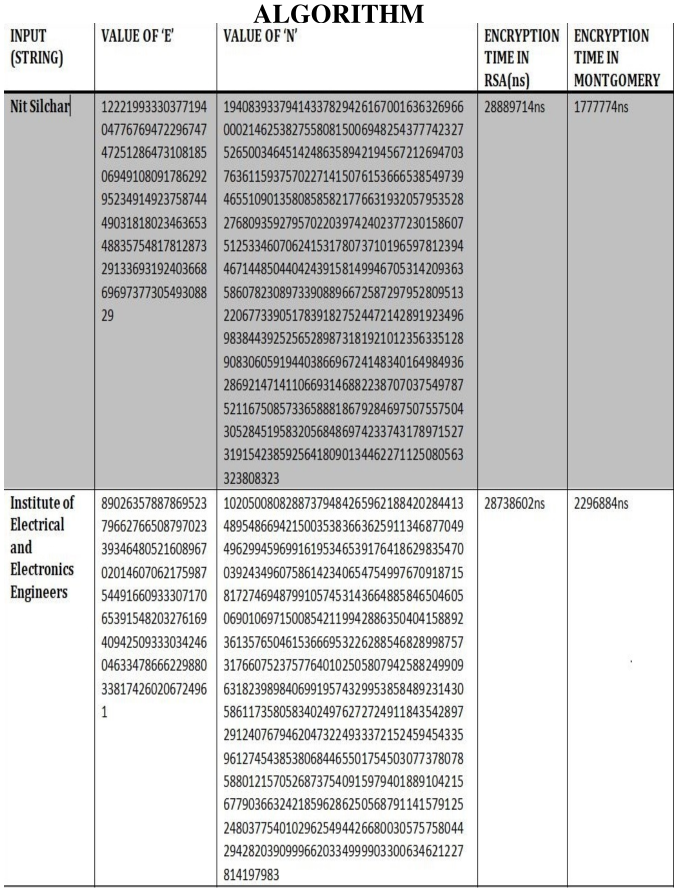


```java
    // The range of x is unlimited
    public BigInteger convertIn(BigInteger x) {
        return x.shiftLeft(reducerBits).mod(modulus);
    }


    // The range of x is unlimited
    public BigInteger convertOut(BigInteger x) {
        return x.multiply(reciprocal).mod(modulus);
    }


    // Inputs and output are in Montgomery form and in the range [0, modulus)
    public BigInteger multiply(BigInteger x, BigInteger y) {
        assert x.signum() >= 0 && x.compareTo(modulus) < 0;
        assert y.signum() >= 0 && y.compareTo(modulus) < 0;
        BigInteger product = x.multiply(y);
        BigInteger temp = product.and(mask).multiply(factor).and(mask);
        BigInteger reduced = product.add(temp.multiply(modulus)).shiftRight(reducerBits);
        BigInteger result = reduced.compareTo(modulus) < 0 ? reduced : reduced.subtract(modulus);
        assert result.signum() >= 0 && result.compareTo(modulus) < 0;
        return result;
    }
    // Input x (base) and output (power) are in Montgomery form and in the range [0, modulus);
    //input y (exponent) is in standard form
    public BigInteger pow(BigInteger x, BigInteger y) {
        assert x.signum() >= 0 && x.compareTo(modulus) < 0;
        if (y.signum() == -1)
            throw new IllegalArgumentException("Negative exponent");

        BigInteger z = convertedOne;
        for (int i = 0, len = y.bitLength(); i < len; i++) {
            if (y.testBit(i))
                z = multiply(z, x);
            x = multiply(x, x);
        }
        return z;   }   }
```

Figure 11. Proposed algorithm exponentiation code

TABLE 2

COMPARISON OF RSA WITH PROPOSED ALGORITHM



INPUT (STRING)	VALUE OF 'E'	VALUE OF 'N'	ENCRYPTION TIME IN RSA(ns)	ENCRYPTION TIME IN MONTGOMERY
Nit Silchar	12221993330377194047767694722967474725128647310818506949108091786292952349149237587444903181802346365348835754817812873291336931924036686969737730549308829	19408393379414337829426167001636326966000214625382755808150069482543777423275265003464514248635894219456721269470376361159375702271415076153666538549739465510901358085858217766319320579535282768093592795702203974240237723015860751253346070624153178073710196597812394467144850440424391581499467053142093635860782308973390889667258729795280951322067733905178391827524472142891923496983844392525652898731819210123563351289083060591944038669672414834016498493628692147141106693146882238707037549787521167508573365888186792846975075575043052845195832056848697423374317897152731915423859256418090134462271125080563323808323	28889714ns	1777774ns
Institute of Electrical and Electronics Engineers	8902635788786952379662766508797023393464805216089670201460706217598754491660933307170653915482032761694094250933303424604633478666229880338174260206724961	10205008082887379484265962188420284413489548669421500353836636259113468770494962994596991619534653917641862983547003924349607586142340654754997670918715817274694879910574531436648858465046050690106971500854211994288635040415889236135765046153666953226288546828998757317660752375776401025058079425882499096318239898406991957432995385848923143058611735805834024976272724911843542897291240767946204732249333721524594543359612745438538068446550175450307737807858801215705268737540915979401889104215677903663242185962862505687911415791252480377540102962549442668003057575804429428203909996620334999903300634621227814197983	28738602ns	2296884ns

Figure 12. Table showing encryption time comparision b/w original and proposed

VII. Conclusion and Future Work

Thus the proposed algorithm makes the encryption faster through recoding of E into a redundant representation and also makes the modular multiplications faster through montgomery reduction algorithm . On top of it the parallel processor and systolic array design further increases computation by factor of 2 and enables fast encryption of a sequence of messages for fixed public and private key.

Although the proposed algorithm reduces the total operations and optimizes each of the operation by the underlying algorithm remains the same hence the time complexity in close to one another and difference is insignificant for small sizes keys,however in case of very large keys this optimization makes a great effort in reducing the time for encryption.Experimental analysis of the algorithm is left as a future scope of study and enhancements and also side effects on the other supporting processes in the cryptosystems are kept as future scope of study and is beyond the scope of the paper.

References

[1] Ravi Shankar Dhakar,Amit Kumar Gupta, Prashant Sharma," Modified RSA Encryption Algorithm (MREA)",*pp.426 – 429, 7-8 Jan. 2012, Rohtak, Haryana, India*

[2] Martin Kochanski, "Montgomery Multiplication" a colloquial explanation.

[3] Chang N.Zhang , Herold L.Martin, "Parallel algorithms and Systolic Array Design For RSA Cryptosystems" Systolic Arrays, 1988., Proceedings of the *International Conference on 25-27 May 1988,San Diego, CA, USA, USA.*

[4] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM, vol. 21, no. 2, pp. 120–126, Feb. 19*

[5] Nitha Thampia, Meenu Elizabath Jose, "Montgomery Multiplier for Faster Cryptosystems" *Procedia Technology 25 (2016) 392 – 398 Available online at www.sciencedirect.com*

[6] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. "Handbook of Applied Cryptography". *CRC Press, 1996. ISBN 0-8493-8523-7, chapter 14.*

[7] H. T. Kung, C. E. Leiserson: Algorithms for VLSI processor arrays; in: C. Mead, L. Conway (eds.): Introduction to VLSI Systems; Addison-Wesley, 1979

[8] A Parallel Implementation of Montgomery Multiplication on Multicore Systems: *Algorithm, Analysis, and Prototype,Issue No. 12 - December (2011 vol. 60)* Zhimin Chen , Virginia Polytechnic Institute and State University, Blacksburg,Patrick Schaumont , Virginia Polytechnic Institute and State University, Blacksburg.

[9] S. Y. Kung: VLSI Array Processors; Prentice-Hall, Inc., 1988

[10] Chih-Yuang Su, Shih-Arn Hwang, Po-Song Chen and Cheng-Wen Wu,"An Improved Montgomery's Algorithm for High-Speed RSA Public-Key Cryptosystem", *IEEE Transactions on VLSI Systems, vol. 7, no. 2, pp. 280 - 284 June 1999*

[11] Hubbard, Steven S, "Radix – 4 Implementation of a Montgomery Multiplier for a RSA Cryptosystem." (2006).

[12] C. McIvor, M. McLoone and J.V. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques", *IEE Proceedings - Computers and Digital Techniques, Volume: 151, Issue: 6, pp. 402 - 408, 18 Nov. 2004*

[13] S. E. Eldridge, "A faster modular multiplication algorithm," *Int. J. Comput. Math., vol. 40, pp. 63–68, 1991.*

[14] N. Petkov: "Systolic Parallel Processing"; North Holland Publishing Co, 1992

[15] Sushanta Kumar Sahu and Manoranjan Pradhan,"Implementation of Modular multiplication for RSA Algorithm", *2011 International Conference on Communication Systems and Network Technologies.*

[16] Min-Sup Kang, Dong-Wook Kim, "Systolic array based on fast modular multiplication algorithm for RSA cryptosystem",*TENCON 99. Proceedings of the IEEE Region 10 Conference,pp. 305 - 308 vol.1, 15-17 Sept. 1999,South Korea*