FPGA Implementation of RSA algorithm and to develop a crypto based security system

A Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology
In
Electronics and Communication Engineering

Submitted by:

Ranjeet Behera (109ec0215)

&

Pradhan Abhisek (109ec0337)

Under the supervision of

Prof. Ayas Kanta Swain

Department of Electronics and Communication Engineering,

National Institute of Technology, Rourkela

2012-2013

FPGA Implementation of RSA algorithm and to develop a crypto based security system

A Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology
In
Electronics and Communication Engineering

Submitted by:

Ranjeet Behera (109ec0215)

&

Pradhan Abhisek (109ec0337)

Under the supervision of

Prof. Ayas Kanta Swain

Department of Electronics and Communication Engineering,

National Institute of Technology, Rourkela

2012-2013

National Institute of Technology, Rourkela

C E R T I F I C A T E

This is to certify that the Thesis entitled, '**FPGA Implementation of RSA algorithm and to develop a crypto based security system**' submitted by **Ranjeet Behera & Pradhan Abhisek** in partial fulfillment of the requirements for the award of **Bachelor of Technology Degree** in **Electronics an Communication Engineering** at the **National Institute of Technology, Rourkela** is a bona fide work carried out by them under my supervision. To the best of my knowledge and belief the matter embodied in the Thesis has not been submitted by them to any other University/Institute for the award of any Degree/Diploma.

Prof. Ayas Kanta Swain

Department of Electronics and Communication Engineering,
National Institute of Technology Rourkela

A C K N O W L E D G M E N T

We have taken efforts in making this project. However, it would not have been possible without the kind support and help of many individuals and organizations. We would like to extend our sincere thanks to all of them.

We take this opportunity to express our profound gratitude and deep regards to our guide Prof. Ayas Kant Swain for his monitoring, exemplary guidance and constant encouragement throughout the course of this thesis. The help, guidance and blessing given by him time to time shall carry us a long way in the journey of life on which we are about to embark.

We are obliged to members of VLSI Lab, the PhD personnel and especially Jagannath Sir for the valuable information provided by them and their support. We are grateful for their cooperation during the period of our assignment.

Lastly, we thank our parents, brother, sisters and friends for their constant encouragement without which this assignment would not be possible.

A B S T R A C T

This project is aimed to implement RSA algorithm on FPGA and to use the cryptography algorithm (RSA) to develop a crypto based security system.

The control and data path of RSA algorithm (decryption only) is implemented on FPGA to behave as an independent password checker for the security system. The encryption part of the algorithm is done by the system itself. The system has the different public key for encryption for different users and the corresponding private key of the user is saved in the FPGA. The system generates a random 16-bit number and encrypts it using the encryption algorithm of RSA and sends the encrypted message to the FPGA using a USB to serial cable and the FPGA decrypts it using the decryption algorithm of FPGA and sends back the decrypted message to the system. The system checks the random message it generated before with the decrypted message send by FPGA for the particular user. If both the data matches then the system welcomes the user and if it doesn't matches then it will give two more chances for entering the correct user name and connecting the correct FPGA.

C O N T E N T S

LIST OF FIGURES

LIST OF TABLES

C H A P T E R - 1

I N T R O D U C T I O N

1. Introduction

1.1 Motivation

The rising growth of data communication technique and electronic transactions over the web has made system security to become the most important issue over the network. To provide modern security features, public-private key cryptosystems are used. One of such cryptosystem is RSA algorithm. Though computation in RSA takes more time by if the message to be encrypted is generated randomly then RSA will prove to be good cryptography algorithm for system security.

For the better working of RSA based cryptosystem the system has the public key for decryption and the user will have the device containing the private key assigned to the user. And instead of entering the password the user will just have to insert the device to the system and the system will do the cross checking of the password for that particular user and allow access accordingly. As the user will not know the password as well as the password length so he can't give the password to any other person and the person will be solely responsible for any wrong doing in the system.

1.2 Problem statement

The primary objective of this project is to implement the decryption part of RSA algorithm in FPGA and the encryption part in the system with a GUI (in visual basic .NET) to connect to the FPGA with a USB to serial cable.

1.3 Organization of thesis

The Thesis has been divided into five chapters including this one. The first chapter provides an introduction and gives an overview of the project. The second chapter explains the basic concepts regarding RSA algorithm and various components used in implementation of RSA algorithm. The third chapter deals with the implementation of decryption part of RSA algorithm on FPGA. It also includes the VHDL coding for various component used for implementation of the algorithm. Chapter four concerns with the graphical user interface (GUI) developed in visual basic for the encryption part of RSA algorithm. This chapter also deals with the connection made with FPGA to act as a 'single system -single user' crypto system. The last and final chapter provides the future aspect of this project along with inference drawn from this project.

C H A P T E R – 2

B A C K G R O U N D

I N F O R M A T I O N

2. Background Information

2.1 Basics of RSA algorithm

RSA algorithm is a cryptographic algorithm introduced in the year 1978 by Ron Rivest, Adi Shamir and Leonard Adleman. RSA implemented following two important ideas:

1. Public-key encryption. In RSA, encryption keys are made public while the decryption keys are kept private, so only the person with the correct decryption key can decipher an encrypted message. Everyone has their own encryption and corresponding decryption keys. The keys are made in such a way that the decryption key cannot be easily deduced from the public encryption key.

2. Digital signatures. The receiver may need to verify that a transmitted message is actually originated from the sender, and didn't just come from authentication. This is done with the help of the sender's decryption key, and later the signature can be verified by anyone, using the corresponding public encryption key. Signatures therefore cannot be copied. Also, no signer can later deny having signed the message.

The various steps involved in RSA algorithm are :

2.1.1 Finding large prime numbers

Finding ‘n’ is the first step of the algorithm, where ‘n’ is the product of two prime numbers ‘p’ & ‘q’. The number ‘n’ will be revealed in the encryption and decryption keys, but the numbers ‘p’ and ‘q’ will not be explicitly shown. The prime numbers ‘p’ and ‘q’ should be large such that it will be very difficult to derive from ‘n’.

$$n = p * q \qquad (1)$$

2.1.2 Finding the public key (e)

Choose a number ‘e’ such that ‘e’ is co-prime to φ (n), where φ (n) is the Euler’s totient function that counts the number of positive integers less than or equal to ‘n’ that are relatively prime to ‘n’ *i.e.*

$$\varphi\,(n) = (p-1)(q-1) \tag{2}$$

&

$$Gcd\big(e, \varphi(n)\big) = 1 \tag{3}$$

Where, $1 < e < \varphi(n)$

2.1.3 Determine the private key (d)

Determine the private key ‘d’ such that‘d’ is the multiplicative inverse of the public key ‘e’ i.e.

$$d^{-1} = e\ (mod\big(\varphi(n)\big)) \tag{4}$$

2.1.4 Encryption

Let ‘m’ be the message (integer type) that is to be encrypted using public key ‘e’ to give the encrypted message as ‘c’ where ‘c’ is calculated as

$$c = m^e\ (mod(n)) \tag{5}$$

2.1.5 Decryption

The decrypted message ‘m’ is found out using the private key‘d’ and is calculated as:

$$m = c^d \ (mod(n)) \tag{6}$$

2.2 Existing Architectures for Modular Multiplication

2.2.1 Carry Save Adders and Redundant Representation

Addition is the core operation of most algorithms for modular multiplication .Different methods for addition in hardware: carry ripple addition, carry select addition, and carry look ahead addition and others. The primary disadvantage of these methods is the carry propagation, as it is directly proportional to the length of the operands. It may not be a big problem for operands of size 32 or 64 bits but the typical operand size in cryptographic applications range from 160 to 2048 bits. So the resulting delay has a significant influence on the time complexity of these adders.

The carry save adder seems to be the most cost effective adder for our application. Carry save addition is a method for an addition without carry propagation. It is simply a parallel ensemble of n full-adders without any horizontal connection. Its function is to add three integers (n-bit) X, Y, and Z to produce two integers C and S as results such that

C + S = X + Y + Z,

Where C represents the carry and S the sum.

The i^{th} bit s_i of the sum S and the $(i + 1)^{st}$ bit c_i+1 of carry C are calculated using the boolean equations

$$s_i = x_i \text{ xor } y_i \text{ xor } z_i$$

$$c_i+1 = x_iy_i \text{ and } x_iz_i \text{ and } y_iz_i$$

$$c_0 = 0$$

When carry save adders are used in an algorithm one uses a notation of the form

$$(S, C) = X + Y + Z$$

This indicates that two results are produced by the addition.

The results are now represented in two binary words, an n-bit word S and an (n+1) bit word C. Of course, this representation is redundant in the sense that we can represent one value in several different ways. This redundant representation has the advantage that the arithmetic operations are fast, because there is no carry propagation. On the other hand, it brings to the fore one basic disadvantage of the carry save adder:

It does not solve our problem of adding two integers to produce a single result. Rather, it adds three integers and produces two such that the sum of these two is equal to that of the three inputs. This method may not be suitable for applications which only require the normal addition.

2.2.2 Circuit Description

A Carry-Save Adder is just a set of one-bit full-adders, without having the usual carry-chaining. Therefore, an CSA of n-bit receives three n-bit operands, namely A (n-i)...A (0), B (n-i)...B (0), & CIN (n-i)...CIN (0), and generates two n-bit result values, SUM (n-i)...SUM (0) and COUT (n-i)...COUT (0).

The most important application of a carry-save adder is to calculate the partial products in integer multiplication. Doing this allows architectures, where a tree of carry-save adders (a so called Wallace tree) is used to calculate the partial products very fast. One 'normal' adder is then used to add the last set of carry bits to the last partial products, which gives the final multiplication result. Generally, a very fast carry-look ahead or carry-select adder is used for the last stage, so as to obtain the optimal performance

Complexity Model

For comparison of different algorithms we need a complexity model that allows for a realistic evaluation of time and area requirements of the considered methods. The delay of a full adder (1 time unit) is taken as a reference for the time requirement and quantifies the delay of an access to a lookup table with the same time delay of a single time unit. Now the area estimation is based on empirical studies in full- custom and semi-custom layouts for adders and storage elements: The area for 1 bit in a lookup table corresponds to 1 area unit. A register cell requires 4 area units per bit and a full adder requires 8 area units. These values provide a powerful and realistic model for evaluation of area and time for most algorithms for modular multiplication.

2.2.3 Montgomery Multiplication Algorithm

The Montgomery algorithm [Algorithm 1] computes $P = (X*Y* (2n)^{-1}) \bmod M$. The idea of Montgomery is to keep the lengths of the intermediate results smaller than n+1 bit. This is achieved by interleaving the computations and additions of new partial products with divisions by 2; each of them reduces the bit- length of the intermediate result by one.

The key concepts of the Montgomery algorithm [Algorithm 1(2.2.4)] are the following:

- Adding a multiple of M to the intermediate result does not change the value of the final result; because the result is computed modulo M. M is an odd number.
- After each addition in the inner loop the least significant bit (LSB) of the intermediate result is checked. If it is 1, that means intermediate result is an odd no., so we add M to make it even. This even number can be divided by 2 without remainder. This division by 2 reduces the intermediate result to n+1 bit again.
- After successive n steps these divisions all add up to (one division by 2^n).

The Montgomery algorithm is very easy to implement since it operates least significant bit first and does not require any comparisons. A modification of [Algorithm 1 (2.2.4)] with carry save adders is given in [Algorithm 2 (2.2.5)]:

2.2.4 Algorithm 1: Montgomery multiplication

- Inputs: X,Y, M with 0<=X, Y<M
- Output: $P = (X*Y(2^n)^{-1}) \bmod M$
- n: number of bits in X;
- x_i : i^{th} bit of X;
- s_0 : LSB of S;

1) S := 0 ; C := 0 ;
2) For(i=0;i<n;i++){
3) (S,C) := S + C + x_i *Y ;
4) (S,C) := S + C + p_0 *M;
5) S := S DIV 2; C := C DIV 2 ; }
6) P := S + C;
7) If (P > M) then P := P – M;

2.2.5 Algorithm 2: Fast Montgomery multiplication

- Inputs: X,Y, M with 0<=X, Y<M
- Output: P = $(X*Y(2^n)^{-1})$ mod M
- n: number of bits in X;
- x_i : i^{th} bit of X;
- s_0 : LSB of S;

1) S := 0 ; C := 0 ;
2) For(i=0;i<n;i++){
3) (S,C) := S + C + x_i *Y ;
4) (S,C) := S + C + p_0 *M;
5) S := S DIV 2; C := C DIV 2 ; }
6) P := S + C;

   If (P > M) then P: = P – M;

In this algorithm the delay of one pass through the loop is reduced from O (n) to O (1). This remarkable improvement of the propagation delay inside the loop of [Algorithm 2 (2.2.5)] is due to the use of carry save adders to implement step (3) and (4) in [Algorithm 1a (2.2.4)].

Step (3) and (4) in [Algorithm 2(2.2.5)] represent carry save adders. S and C denote the sum and carry of the three input operands respectively.

Of course, the additions in step (6) and (7) are conventional additions. But since they are performed only once while the additions in the loop are performed n times which is subdominant with respect to the time complexity.

Figure 1 shows the architecture for the implementation of the loop of [Algorithm 2 (2.2.5)]. The layout comprises of two carry save adders (CSA) and registers for storing the intermediate results of the sum and carry. The carry save adders are the dominant occupiers of area in hardware especially for very large values of n (e.g. n > 1024).

Next we shall see the changes that were made in [Fig. 2] to reduce the number of carry save adders in Figure1 from 2 to 1, thereby saving considerable hardware space. However, these changes also brought about other area consuming blocks such as lookup tables for storing pre-computed values before the start of the loop.

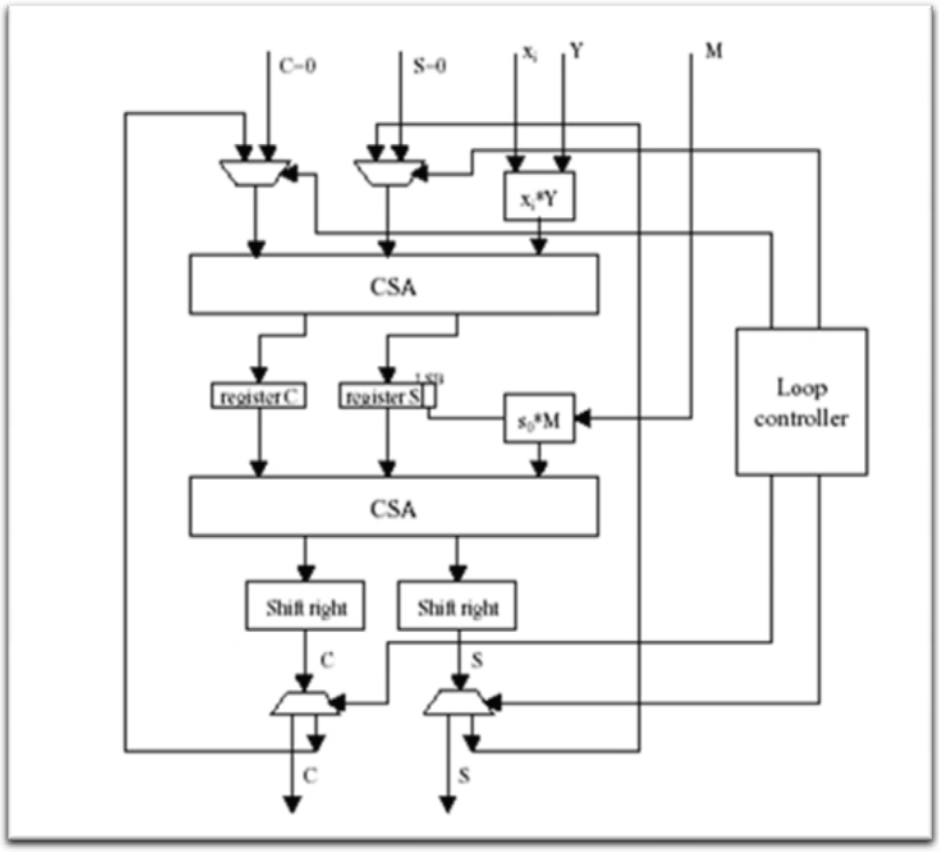

Fig 2.1: Architecture for the loop of algorithm 2

There are various modifications to the Montgomery algorithm in [5], [6] and [7]. All these algorithms aimed at decreasing the operating time for faster system performance and reducing the chip area for practical hardware implementation.

2.2.6 Newer Architectures for Modular Multiplication

Herein, a summary of these algorithms and architectures is given. These have been designed so as to meet the core requirements of most modern devices: small chip area and low power consumption.

2.2.7 Faster Montgomery Algorithm

In Figure 1, the layout for the implementation of the loop of (Algorithm 2) consists of two carry save adders. For large word sizes (e.g. n = 1024 or higher), this would require considerable hardware resources to implement the architecture of (Algorithm 2). The reason for this optimized algorithm is that of reducing the chip area for practical hardware implementation of (Algorithm 2). This is possible if we can precompute the four possible values to be added to the intermediate result within the loop of Algorithm 2, thus reducing the number of carry save adders used from 2 to 1.

There are four possible scenarios:

- If the sum of the values (odd) of S and C is an even number, and if the actual bit x_j of X is 0, then we add 0 before we perform the reduction of S and C by division by 2.
- If the sum of the old values of S and C is an odd number, and if the actual bit x_j of X is 0, then we must add M to make the intermediate result even. Afterwards, we divide S and C by 2.

- if the sum of the old values of S and C is an even number, and if the actual bit x_i of X is 1, but the increment x_i *Y is even, too, then we do not need to add M to make the intermediate result even. Thus, in the loop we add Y before we perform the reduction of S and C by division by 2. The same action is necessary if the sum of S and C is odd, and if the actual bit x_i of X is 1 and Y is odd as well. In this case, S+C+Y is an even number, too.
- If the sum of the old values of S and C is odd, the actual bit x: of X is 1, but the increment x: *Y is even, then we must add Y and M to make the intermediate result even. Thus, in the loop we add Y+M before we perform the reduction of S and C by division by 2.

  The same action is necessary if the sum of S and C is even, and the actual bit x_i of X is 1, and Y is odd. In this case, S+C+Y+M is also an even number. The computation of Y+M can be done prior to the loop. This saves one of the two additions which are replaced by the choice of the right operand to be added to the old values of S and C. Now [Algorithm 3(2.2.8)] is a modification of Montgomery's method which takes advantage of this idea.

  The advantage of [Algorithm 3(2.2.8)] in comparison to [Algorithm 1(2.2.5)] can be seen in the implementation of the loop of [Algorithm 3(2.2.8)] in Figure 2. The possible values of I are stored in a lookup-table, where the actual values of x, y_0, s_0 and c_0. Address the lookup table. The operations in the loop are now reduced to one table lookup and one carry save addition. Doing this allows both of these activities to be performed concurrently. Point to be noted is that the shift right operations that implement the division by 2 can be done by routing.

2.2.8 Algorithm 3: The Faster Montgomery multiplication

- Inputs: X,Y, M with 0<=X, Y<M
- Output: P = $(X*Y(2^n)^{-1})$ mod M
- n: number of bits in X;
- x_i : i^{th} bit of X;
- s_0 : LSB of S ; c_0 : LSB of C ; y_0 : LSB of Y;
- R: pre-computed value of Y + M;

1) S := 0 ; C := 0 ;
2) For(i=0;i<n;i++){
3) if (($s_0 = c_0$) and not x_i) then I := 0 ;
4) if (($s_0 \neq c_0$) and not x_i) then I := M ;
5) if((not (s_0 xor c_0 xor y_0) and x_i) then I := Y ;
6) if(((s_0 xor c_0 xor y_0) and x_i) then I := R ;
7) (S,C) : = S + C +I;
8) S := S DIV 2; C := C DIV 2 ; }
9) P := S + C;
10) If (P > M) then P := P – M;

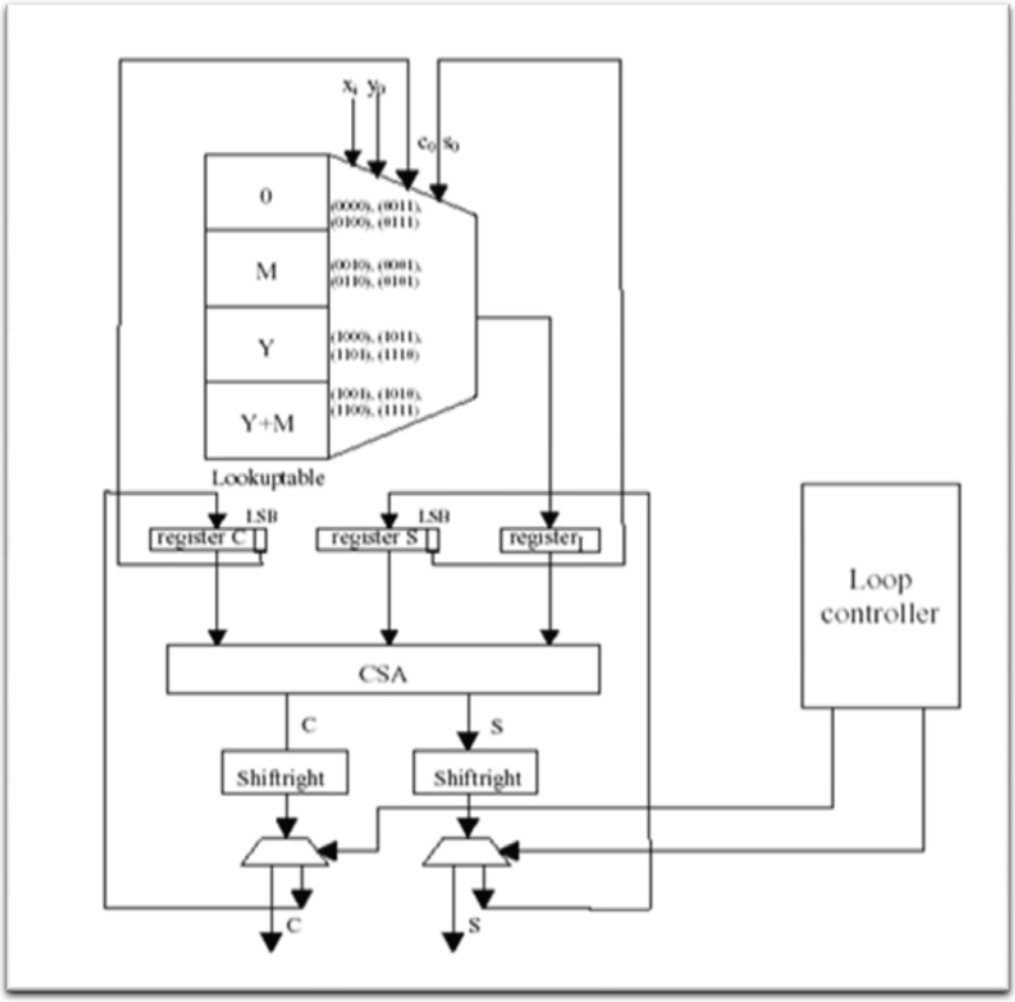

Fig. 2.2: Architecture for Algorithm 3

C H A P T E R – 3

R S A A L G O R I T H M:

F P G A I M P L E M E N T A T I O N

3. RSA Algorithm: Hardware Implementation

3.1 Modeling Technique

The design of the architectures was done using Very High Speed Integrated Circuit Hardware Description Language (VHDL) and the complete source codes for 32 to 1024 bit implementations of Fast Montgomery, Faster Montgomery and Optimized Interleaved multipliers are available in electronic form.

For the implementation of the multipliers, a very structured approach was used which shows the hierarchical decomposition of the multipliers into sub modules The basic units of the architectures which comprises carry save adders, shift registers and registers were modeled as components which are independently functional. These components are then wired together by means of signals to construct the structure of the multiplier as shown in [Figure 3] for the Faster Montgomery architecture [Figure 2].

3.2 Structural Elements of Multipliers

Every VHDL design consists of at least an Entity and Architecture pair. Entity describes the interface of the system from the perspective point of its input and output, while Architecture describes the behavior or the functionality of the digital system itself. In the next sub-sections, the pair of Entity and Architecture, the structural elements in the Faster Montgomery architecture presented in Figure 3 is described.

3.3 Architecture for Faster Montgomery Architecture

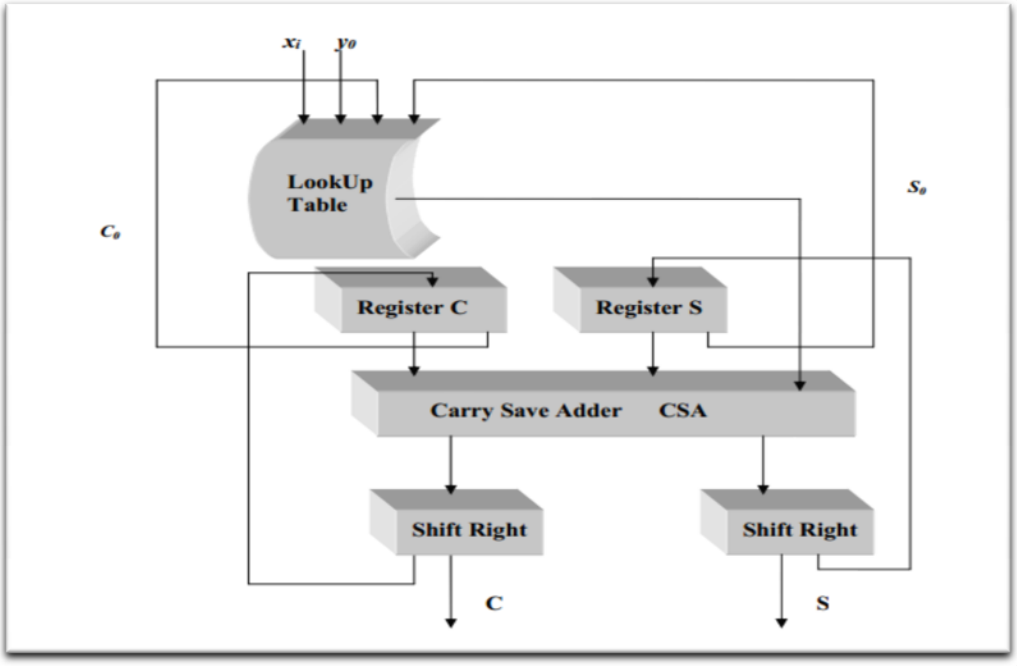

Fig. 3.1: Block diagram showing components that were implemented for the Faster Montgomery Architecture

3.3.1 Carry save adder

The carry save adder is simply a parallel ensemble of n full-adders without having any horizontal connection. The main purpose is to add three n-bit integers X, Y and Z so as to produce two integers C and S such that

$$C + S = X + Y + Z$$

Where C and S represent the usual carry and sum respectively.

```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

--  EXTERNAL PORTS
ENTITY c_s_adder IS
GENERIC(n: integer := 32);
PORT( x:         IN    std_logic_vector(n DOWNTO 0);
      y:         IN    std_logic_vector(n DOWNTO 0);
      carry_in:    IN      std_logic_vector(n DOWNTO 0);
      sum:         OUT    std_logic_vector(n+1 DOWNTO 0);
      carry:   OUT    std_logic_vector(n+1 DOWNTO 0));
END c_s_adder;

--  INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF c_s_adder IS
BEGIN
sum <= '0' & ((x XOR y) XOR carry_in);
carry <= (((x AND y) OR (x AND carry_in)) OR (y AND carry_in))& '0' ;
END behavioral;
```

3.3.2 Lookup Table

The lookup table is one of the most important units inside the new optimized architectures in [Fig. 3]. It is used to store the values of pre-computations that are performed prior to the execution of the loop. This eliminates time consuming operations that are performed inside the loop, thus improving the speed of computation.

```
--EXTERNAL PORTS
ENTITY table_LU IS
GENERIC ( n      : integer := 32);
PORT (
address : IN  std_logic_vector(3 DOWNTO 0); -- 4-bit Address
M,Y,MY    :   IN std_logic_vector(n DOWNTO 0); -- input
result   : OUT std_logic_vector(n DOWNTO 0)); -- output
END table_LU;
-- INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF table_LU IS

BEGIN

pro_LU: PROCESS(address)
BEGIN
CASE address IS
WHEN "0000" =>
result <= "00000000000000000000000000000000";
WHEN "0001" =>
result <= M;
WHEN "0010" =>
result <= M;
WHEN "0011" =>
result <= "00000000000000000000000000000000";

WHEN "0100" =>
result <= "00000000000000000000000000000000";

WHEN "0101" =>
result <= M ;

WHEN "0110" =>
result <=  M ;
```

```
WHEN "0111" =>
result <= "00000000000000000000000000000000";

WHEN "1000" =>
result <= Y ;

WHEN "1001" =>
result <= MY ;

WHEN "1010" =>
result <= MY ;

WHEN "1011" =>
result <= Y ;

WHEN "1100" =>
result <= MY ;

WHEN "1101" =>
result <=  Y ;

WHEN "1110" =>
result <= Y ;

WHEN "1111" =>
result <= MY ;
WHEN OTHERS =>
result <= "00000000000000000000000000000000";
END CASE;
END PROCESS pro_LU;
END behavioral;
```

3.3.3 Register

The purpose of the Registers (i.e. Register C and Register S) in Figure 2 and Figure 10 are to hold the intermediate values of the carry and sum respectively during the execution of the loop. So the registers must have memory and be able to save their state over a given amount of time. Such a behavior can be obtained by the following rules must be observed during the implementation:

- The sensitivity list of a process should not include all the signals that are being read with the process.
- If-Then-Else should be incompletely used.

```
--  EXTERNAL PORTS
ENTITY h_register IS
GENERIC( n: IN   integer := 32);
PORT(
   rst:  IN  std_logic;
   clk:  IN  std_logic;
   d:  IN  std_logic_vector(n DOWNTO 0);
   q:  OUT  std_logic_vector(n DOWNTO 0);
   q_lsb: OUT  std_logic);
END h_register;

--  INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF h_register IS

SIGNAL q_temp: std_logic_vector(n DOWNTO 0);
BEGIN
   PROCESS(rst, clk)
   BEGIN
      IF rst = '1' THEN
      q_temp <= (q_temp'RANGE => '0');
      ELSIF clk'EVENT AND clk = '1' THEN
      q_temp <= d;
      END IF;
   END PROCESS;
q <= q_temp;
q_lsb <= q_temp(0);
END behavioral;
```

3.3.4 M register

This register stores the modulus value. So it just needs an output, and cannot be changed once the hardware is made.

```
--  EXTERNAL PORTS
ENTITY m_register IS
GENERIC( n: IN   integer := 32);
PORT(
   q:  OUT  std_logic_vector(n DOWNTO 0)
   );
END m_register;

--  INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF m_register IS

SIGNAL q_temp: std_logic_vector(n DOWNTO 0);
BEGIN
q_temp <= (others => '0');
q <= q_temp;
END behavioral;
```

3.3.5 One-Bit Shifter

The one-bit shifters inside Figure 2 and Figure 10 are used to perform 1-bit, right- shift operations. The behavioral description of this unit is as shown in Figure below. Here, the least significant bit of the input is discarded at the output, thereby reducing the bit length output by 1.

```
--   EXTERNAL PORTS
ENTITY shift_right IS
GENERIC( n: integer := 32);
PORT(
x_in: IN  std_logic_vector(n+1 DOWNTO 0);
x_out: OUT  std_logic_vector(n DOWNTO 0));
END shift_right;
--   INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF shift_right IS
SIGNAL temp:  std_logic_vector(n DOWNTO 0);
BEGIN
PROCESS(x_in)
BEGIN
FOR i IN n+1 DOWNTO 1 LOOP
temp(i - 1) <= x_in(i);
END LOOP;
END PROCESS;
x_out <= temp;
END behavioral;
```

3.3.6 Data path Result

All the work done in the multiplication part was in 32 bit format. But actual value is 16 bit, so this structure convers the 32 bit value to 16 bit by removing extra bits.

```
--   EXTERNAL PORTS
ENTITY result IS
GENERIC( n: integer := 32);
PORT(
x_in: IN  std_logic_vector(n-1 DOWNTO 0);
x_out: OUT  std_logic_vector(n/2 -1 DOWNTO 0));
END result;
--   INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF result IS
SIGNAL temp:  std_logic_vector(n/2 -1 DOWNTO 0);
BEGIN
PROCESS(x_in)
BEGIN
FOR i IN n-1 DOWNTO n/2 LOOP
temp (i-n/2) <= x_in(i);
END LOOP;
END PROCESS;
x_out <= temp;
END behavioral;
```

3.3.7 MUX

Multiplexers are used before the registers which can get value from 2 different locations

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2_1_32 is
GENERIC(n: integer := 32);
port (
      i0 : in std_logic_vector(n-1 DOWNTO 0);
      i1 : in std_logic_vector(n-1 DOWNTO 0);
     sel : in std_logic;
     bitout : out std_logic_vector(n-1 DOWNTO 0)
     );
end mux2_1_32;

architecture Behavioral of mux2_1_32 is
begin

process(i0,i1,sel)
begin
case sel is
  when '0' => bitout <= i0;
  when '1' => bitout <= i1;
  when others => bitout <= "00000000000000000000000000000000";
end case;
end process;

end Behavioral;
```

3.3.8 Datapath

The total representation of data path which contains the total flow of data, starting from the data it gets from buffer to actual output or result.

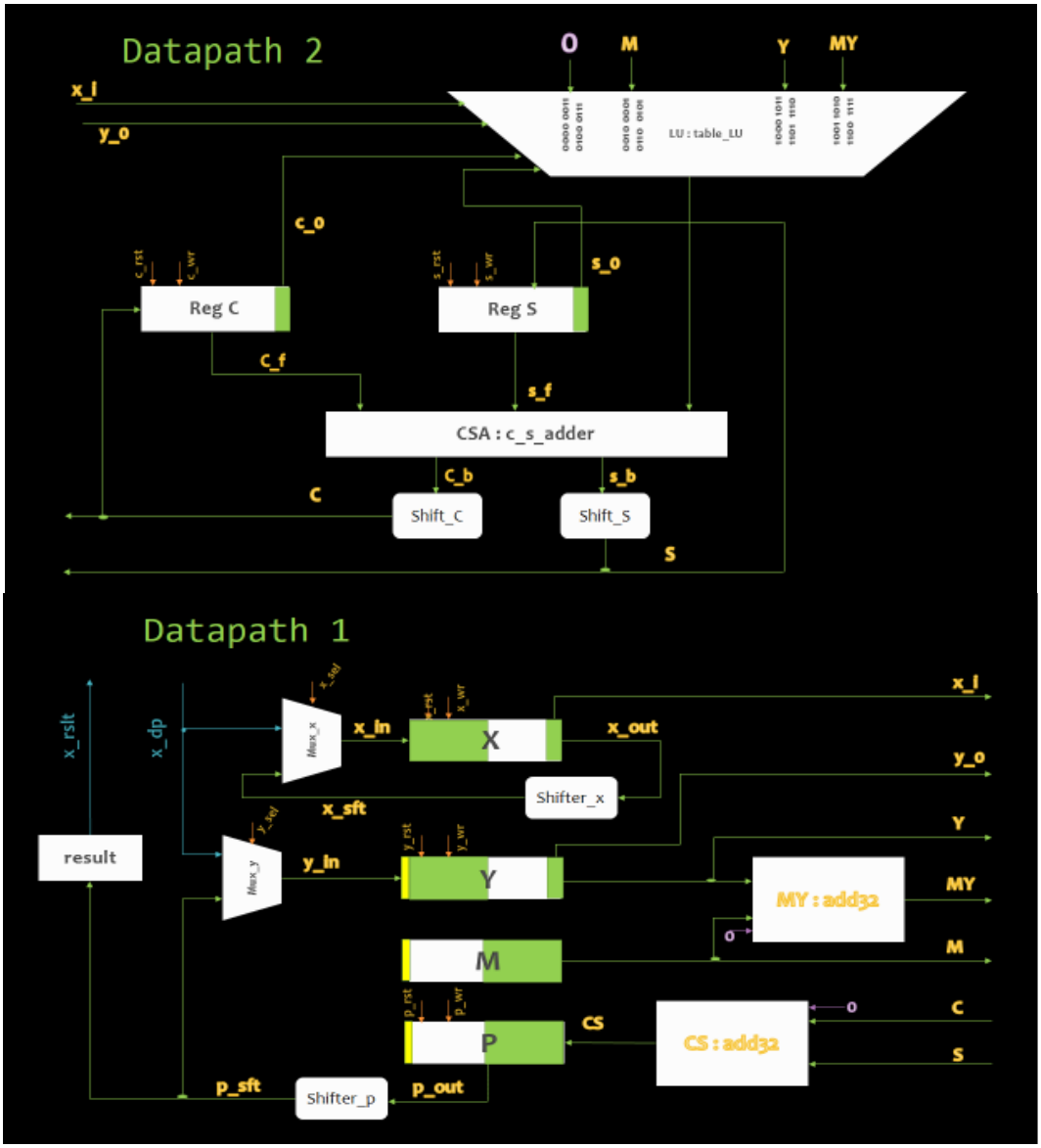

Fig. 3.2: Data path 1 & 2

3.4 Medium

This acts as the buffer for the data that has to be transmitted and received . Data comes from the serial port to MUart and stored in a temporary register a_1 and a_2 for the tho higher and lower bytes of data respectively. Then it is transferred to a 32 bit register A because directly this register will be accessed .While returning the output (to be transmitted by UART) the 16 bit result is broken into 2 bytes and through MUX eac byte of data is passed to Uart.

Fig. 3.3: Medium- Buffer to MUart

3.5 MUart

A lighter version of Uart is implemented for use with requirement of Rx Tx and Ground pins to be connected,so hardware area required decreases. A soft IP core is taken and is modified for specific use.

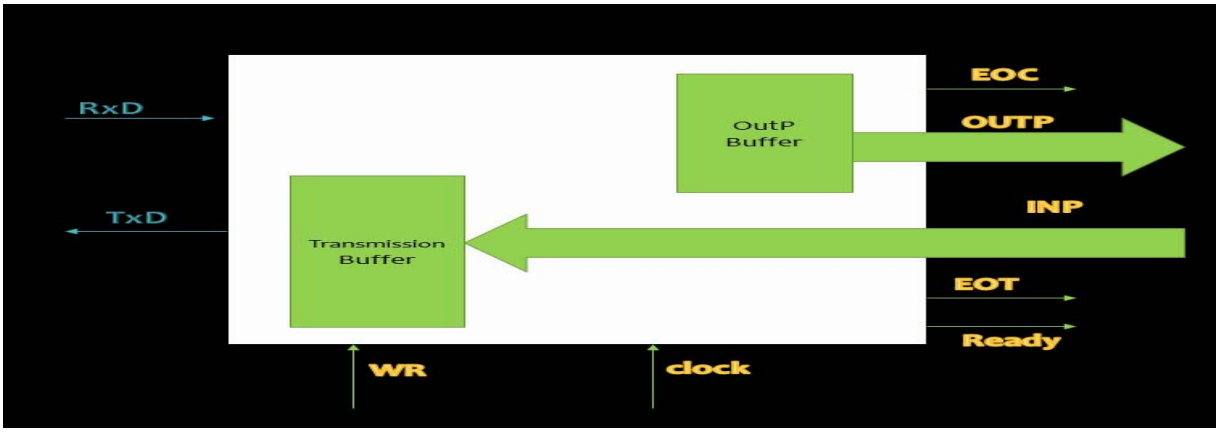

Fig. 3.4: MUart

3.6 Controller

This has got FSM states for controlling the whole of Decryption process. Data first flows from Uart to Medium to Datapath in a sequence do comtrol is easy .The control in datapath follows the algorithm so different states are mentioned in the table. Then data moves to medium and through Uart and serial cable back to PC.

Table 3.1: Data path States

Array (8 downto 0) of STD_LOGIC_VECTOR (11 downto 0);

dp_st	x_sel	x_wr	x_rst	y_sel	y_wr	y_rst	p_wr	p_rst	c_wr	c_rst	s_wr	s_rst	condition
0	0	0	1	0	0	1	0	1	0	1	0	1	Reset
1	0	0	0	0	0	0	0	0	0	0	0	0	Base State
2	0	0	0	0	1	0	0	0	0	0	0	0	MY:=M+Y adder
3	0	1	0	1	0	0	0	0	0	0	0	0	Another X e times multiply
4	1	0	0	1	0	0	0	0	0	0	0	0	X_shift selected
5	1	0	0	1	0	0	0	0	1	0	1	0	C,S_reg updated
6	1	1	0	1	0	0	0	0	0	0	0	0	X rotated
7	1	0	0	1	0	0	1	0	0	0	0	0	C,S added
8	1	0	0	1	1	0	0	0	0	0	0	0	Y updated

Table 3.2: Medium States

array(6 downto 0) of sts_logic_vector(5 downto 0)

med_st	rst	a1_clk	a2_clk	a_clk	b_clk	bb_clk	sel	condition
0	1	0	0	0	0	0	0	Reset
1	0	0	0	0	0	0	0	Base State
2	0	1	0	0	0	0	0	Receives 2 bytes of data and stored in 2 registers a1, a2
3	0	0	1	0	0	0	0	
4	0	0	0	1	0	0	0	Data is stored in a 32 bit register for further use
5	0	0	0	0	1	0	0	Data from result is stored in a 16 bit register
6	0	0	0	0	0	1	0	Data now stored in 2 separate registers and data from 1 is transmitted
7	0	0	0	0	0	0	1	Data from the 2nd register is now transmitted

Above shown is the different states the sub structural units can be. Following the concepts of Algorithm 3 the total controller processor can be written down. With obvious addition of hardware reset and small changes.

3.7 Simulation Result

MATLAB Simulation for RSA algorithm, both in hard crude method of exponentiation and then modulus to using Montgomery multiplication as a base was done. Both worked fine. Then, the steps that are followed in the hardware were tested again using Mat LAB. Results were error free.

Then simulation using ModelSim was done, [Fig. 4] below shows simulation of one of the structural components.

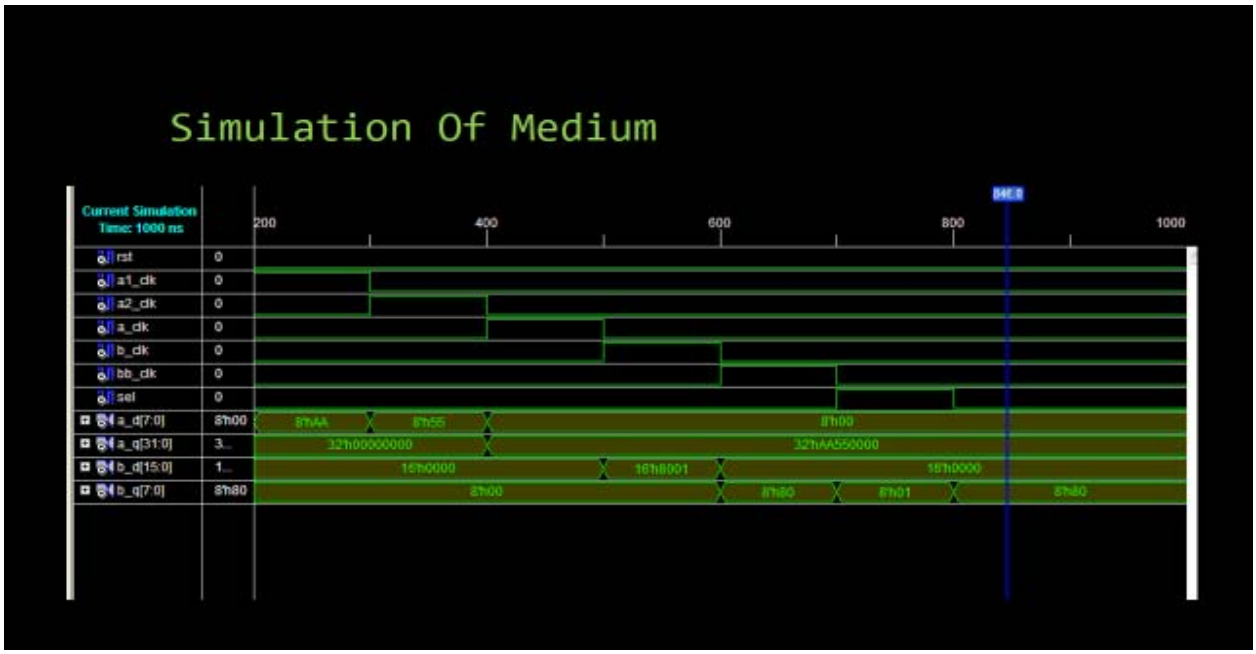

Fig. 3.5: Simulation of Medium

C H A P T E R – 4

G R A P H I C A L U S E R

I N T E R F A C E (G U I)

4. Graphical User Interface (GUI)

4.1 Approach

Graphical User Interface (GUI) for implementing the encryption part of RSA algorithm was done using visual studio 2010 (.NET). Visual basic was chosen to make the interface due to special provisions available in it for serial port interface. The interface is made such that whenever the user name is entered correctly it opens the COM port connected to the FPGA for communication, checks for any exception such as if FPGA is connected or wrongly connected and generates a 16-bit message randomly, encrypts it using the public key assigned to that particular user name and sends the 16-bit encrypted message through the COM port in two clock cycles with the higher sent first and then the lower byte. A USB to serial driver was installed so that the USB port available in the system will send and receive data serially. Also a USB to serial connecter was used to connect the system with the FPGA.

4.2 Single User – Single System: Working

'Single user – single system' means a particular can only be accessed by using a single user name only that is assigned to it. The sequence of windows that appears for authentication process can be given by the following steps:

4.2.1 Enter Correct User Name

The first window that shows up asks for the correct user name. Enter the correct user name and press OK. In this case we have assigned “ranjeet289” as the correct user name.

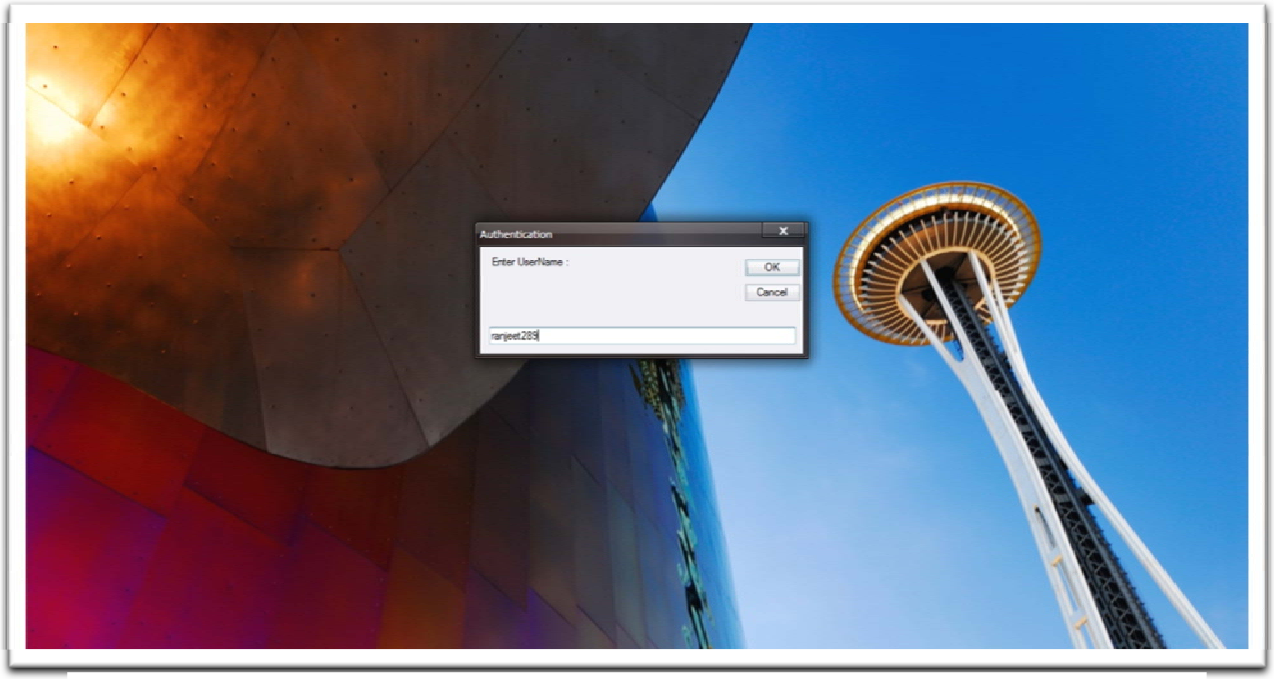

Fig. 4.1: window asking to enter correct user name

If the user name is incorrect then it will show “Invalid User Name” and will give two more chance for entering the correct user name.

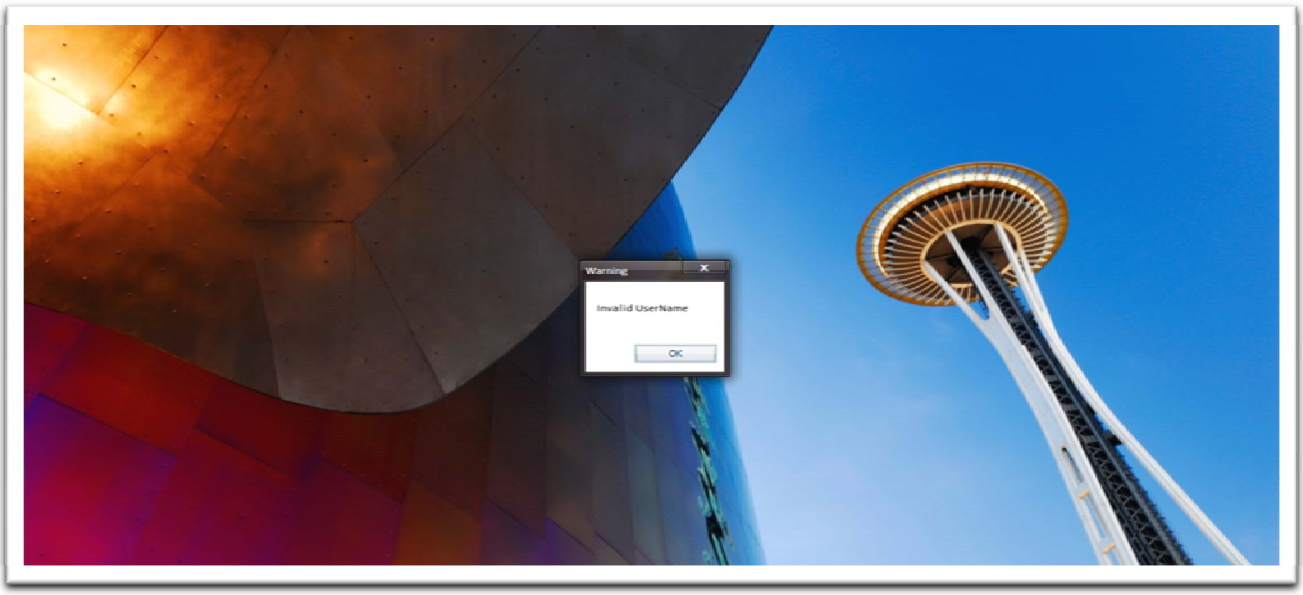

Fig. 4. 2: Pop Up window after entering wrong user name

If one has used up all the attempts to enter the correct user name then a window will pop up showing “You Have Exceeded the Number of Attempts” followed by the message “System Is Shutting Down”.

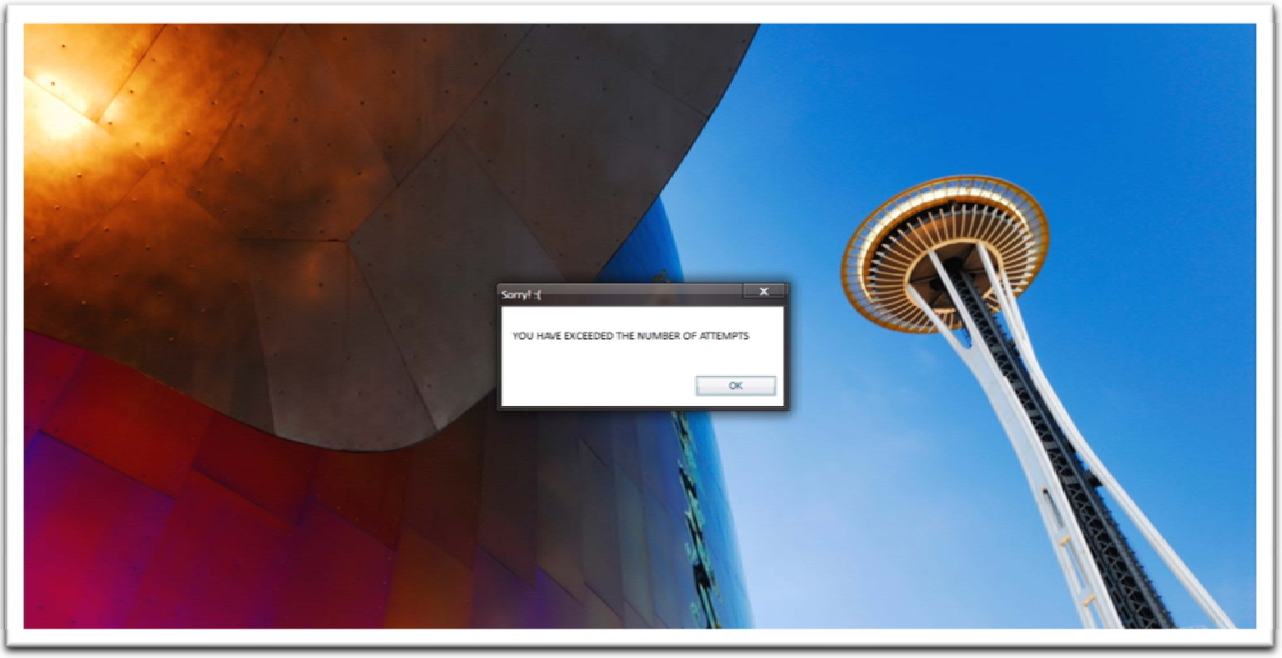

Fig. 4.3: pop up window after exceeding no. of attempts

Fig. 4.4: Final window when one has exceeded the no. of attempts to enter correct username

4.2.2 Verifying Password

After entering the correct user name, the system will randomly generate a 16-bit message, encrypts it using public key and will send the data to FPGA as mentioned in the previous section. Then it will wait for the FPGA to send the decrypted message back to the system. Now the system will compare the decrypted message sent by FPGA with the original message it generated. If both the passwords are same then it will show a pop up window with the message “Password Accepted” followed by welcoming message to the user.

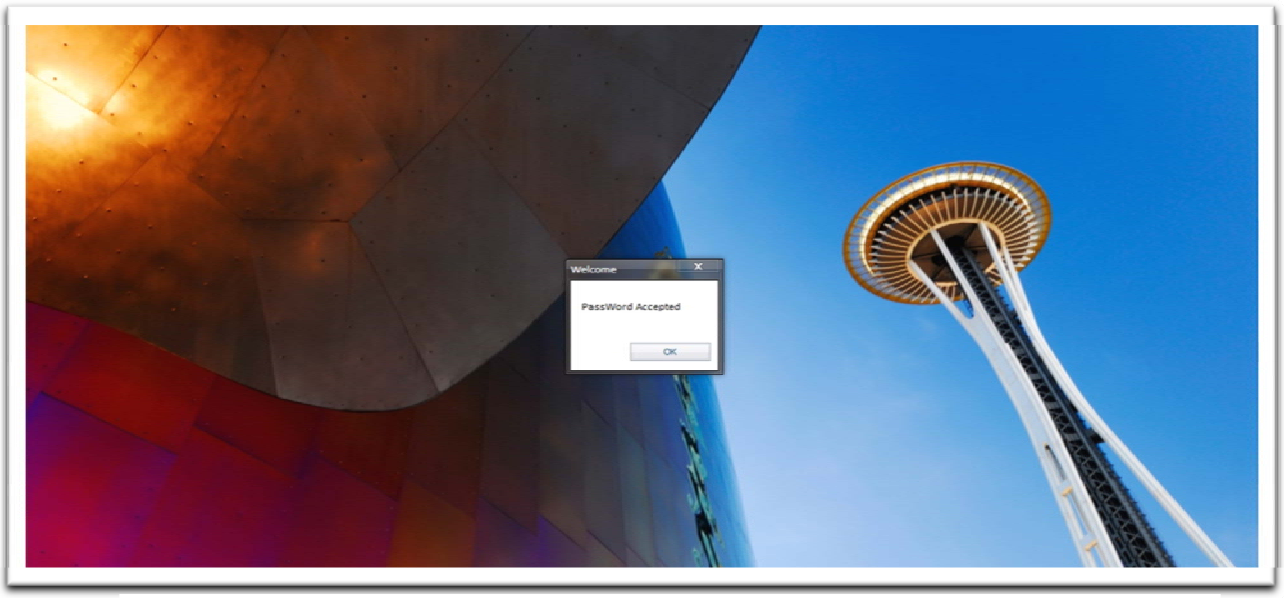

Fig 4.5: pop up window when password matches

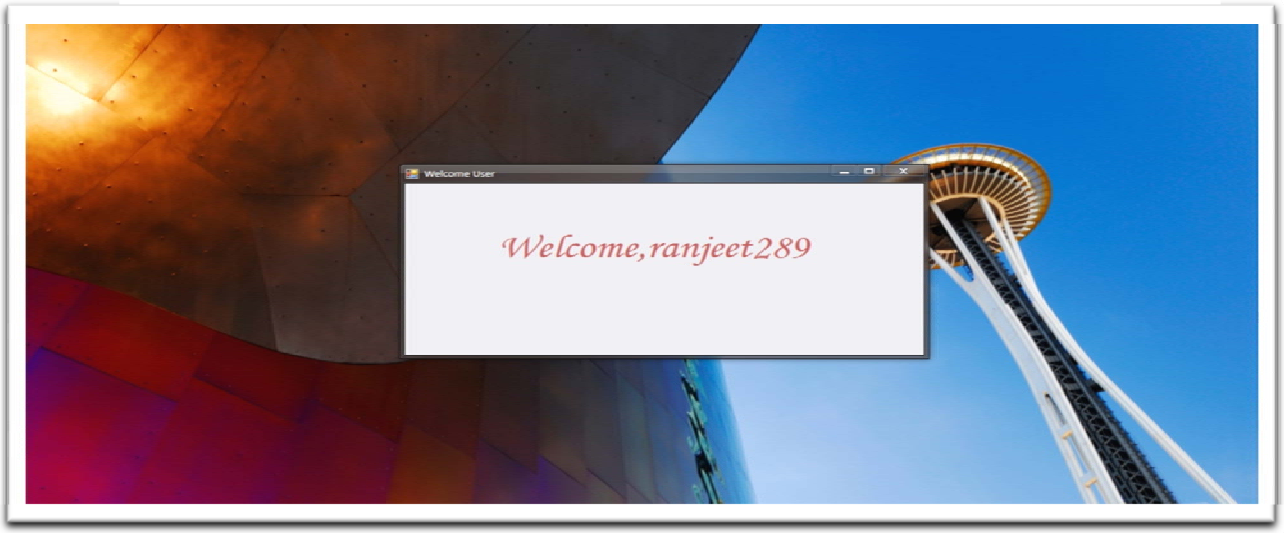

Fig 4.6: Final window welcoming the user

CHAPTER 5

CONCLUSION AND FUTUREWORK

5. Conclusion and Future Work

A crypto based security system was developed using RSA algorithm as the cryptographic algorithm. Each components used for the implementation on FPGA was optimized upto certain level. Implementation of 16-bit encryption and decryption was done since 32-bit implementation needed much more components that are available on Spartan 3E FPGA board. Since FPGA doesn't have USB port but only have serial port, a USB to serial driver was installed on the system to send and receive data from the FPGA serially.

The **future work** that can be done in this regard includes

1) Optimization and IC design.
2) Implementation of better cryptographic algorithm for encryption and decryption
3) Develop driver for serial to USB communication.

R E F E R E N C E

[1] William Stallings, Cryptography and Network Security Principals and Practices, 4th Edition, Pearson Education, Inc., 2006, ISBN 81-7758-774-9.

[2] Ridha Ghayoula, ElAmjed Hajlaoui, Talel Korkobi, Mbarek Traii, Hichem Trabelsi, "*FPGA Implementation of RSA Cryptosystem*", International Journal of Engineering and Applied Sciences 2:3 2006.

[3] M. D. Shieh, J. H. Chen, H. S. Wu, and W. C. Lin, "A new modular exponentiation Architecture for efficient design of RSA cryptosystem," IEEE Trans. Very Large Scale Integration. (VLSI) Syst., vol. 16, no. 9, pp. 1151-1161,Sep. 2008.

[4] opencores.org