SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 11, No. 1, February 2014, 121-131

UDC: 004.3`142:621.394.14 **DOI: 10.2298/SJEE140114011S**

Hardware Modules of the RSA Algorithm

Velibor Škobić[1], Branko Dokić[1], Željko Ivanović[1]

Abstract: This paper describes basic principles of data protection using the RSA algorithm, as well as algorithms for its calculation. The RSA algorithm is implemented on FPGA integrated circuit EP4CE115F29C7, family Cyclone IV, Altera. Four modules of Montgomery algorithm are designed using VHDL. Synthesis and simulation are done using Quartus II software and ModelSim. The modules are analyzed for different key lengths (16 to 1024) in terms of the number of logic elements, the maximum frequency and speed.

Keywords: RSA algorithm, FPGA, Encryption.

1 Introduction

Protection from unauthorized access to data and information is notable challenge within data transmission process. Encryption provides such a data protection. Transmitter encrypts data and sends it to the receiver which reconstructs the original data, using decryption. Eavesdropper may catch the data, but is not able to decrypt it, without knowledge about decryption method [1, 2].

Secure data transfer is very important aspect of bank transactions, online shopping, telephone communication, e-mail etc. Data transfer in these applications is provided by communication networks [2]. These ways of transfer are not secure and there is a possibility of unauthorized access to the data being transferred. There are several data encryption methods. Classical methods are based on secrecy of encryption and decryption algorithms. In modern cryptography, keys are being used for data encryption. Modern cryptography is based on the idea that encryption algorithms are public, while the keys are private. Algorithms are mostly based on mathematical problems that are difficult to compute.

One of the best known public key encryption algorithms is the RSA (*Rivest*, *Shamir*, *Adleman*) algorithm [3], which is based on the principles of number theory. This algorithm is implemented in operating systems, secure phones, and in many protocols for secure internet communications [4 – 6]. In the RSA algorithm, the methods of encryption and decryption are the same, but with

[1]University of Banja Luka, Faculty of Electrical Engineering, Patre 5, 78 000 Banja Luka, Bosnia and Herzegovina; E-mails: velibor.skobic@etfbl.net, bdokic@etfbl.net, zeljko.ivanovic@etfbl.net

application of different keys. Security of these algorithms strongly depends on the key length.

The structure of this paper is as follows. In the second section the RSA algorithm is described. The third section describes computational methods. Simulation results of implemented modules are presented in the fourth section. Conclusions are given in the fifth section.

2 Basics of the RSA Algorithm

Operation of the RSA algorithm is performed in three phases [3]: key generation, encryption and decryption. The key generation is done in the following way. Two primes p and q are generated, and then number m is obtained by multiplication of the primes:

$$m = pq\,. \tag{1}$$

The next step is computing of the *Euler function* φ of the number m. While p and q are primes, the value of φ is given by the formula:

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

After that, it is necessary to determine a number e having value greater than 1 and less than $\varphi(m)$. Another condition is that number 1 is the greatest common divisor of numbers e and $\varphi(m)$:

$$GCD(e, \varphi(m)) = 1\,. \tag{3}$$

Obtained numbers m and e represent a public key that is used for encryption. For decryption, besides number m, a secret key d is needed as well. The value of d is defined by the following equation:

$$d = \frac{k\varphi(m)+1}{e}, \quad k \in N\,, \tag{4}$$

under the constraint:

$$(de) \bmod \varphi(m) = 1\,. \tag{5}$$

Encryption/decryption is performed by exponentiation of the message by the value of key and the result of the exponentiation is divided modulo m. Complexity of this computation depends on the key length. Encrypted data correspond exactly to the input data if the input message P is shorter than number m. Encryption of the message P is done by the following:

$$C = P^e \bmod m\,. \tag{6}$$

while decryption of the message C is done by the following:

$$P = C^d \bmod m\,. \tag{7}$$

From the equations (6) and (7), it can be seen that encryption and decryption methods are identical. Application of a correct secret key, within process of decryption, provides recovery of the original message P.

An illustration of data encryption/decryption using the RSA algorithm is given by the following example. Let p and q be the primes with values:

$$p = 17, \quad q = 19. \tag{8}$$

Compute m i $\varphi(m)$:

$$\begin{aligned} m &= pq = 323, \\ \varphi(m) &= (p-1)(q-1) = 288. \end{aligned} \tag{9}$$

The next step is determination of the number e according to the condition (3), e.g. $e = 11$. Then compute d according to the equation (5). Taking $k = 5$ in the relation (5) we get secret key $d = 131$. Numbers m and e make the public key, while numbers m and d represent the secret key. Let the input data be $P = 15$, then encrypted data is

$$C = 15^{11} \bmod 323 = 60, \tag{10}$$

while decryption yields:

$$P = 60^{131} \bmod 323 = 15, \tag{11}$$

i.e. the original data $P = 15$.

There are many methods to brake RSA encryption. In fact, they are based on the weakness of the whole data protection process, and not on weakness of encryption itself. Efficient way to break RSA encryption is not discovered until now. In order to break RSA encryption, it is necessary to find the factorization of number m, i.e. to determine the prime numbers p and q. Knowing p and q, it is possible to determine a secret key. Factorization of large numbers is a very complex and time consuming process. Considering large key lengths (1024 or 2048), even with application of the fastest modern computers and the best algorithms for decryption, it would take many years to finish the process of factorization. We emphasize, it is not mathematically proven that factorization of number m is needed in order to recover a message P from the message C [1].

3 Computation of the RSA Algorithm

Either software or hardware implementation of the RSA algorithm is possible. Software implementation means a program which operates on the digital processor. Data processing time depends on a frequency of processor and the key length. Increase of the key length increases algorithm security, as well as the data processing time. Systems that process large amount of data require some assistance to processor operation. Remarkable solution is hardware implementation of the RSA algorithm. In that case, data processing is mostly

done in parallel with processor operation, thus yields shorter time for encryption/decryption. There are several papers on this topic, e.g. [7 – 11].

From the equations (6) and (7) it is seen that encryption is done by exponentiation of the message P by e. Decryption means exponentiation of the encrypted message C by d. Then computation modulo m needs to be done. So, the basic algorithm relies on sequential multiplication of the message P (C for decryption) e (d) times, and then application of modulo m operator:

$$C = \left(\prod_{i=1}^{e} P_i \right) \bmod m . \tag{12}$$

The number of bits needed to store intermediate results during message exponentiation is given by the equaton:

$$Q_{bita} = \log_2 (P^e) \approx 2^k \cdot k , \tag{13}$$

where k is number of bits of the key and the message. Taking $k = 256$, according to the relation (10), to store that data we need $C_{bits} \approx 10^{80}$ bits, which is a huge value impossible to implement.

Using the following relationship:

$$(A \cdot B) \bmod m = (A \cdot (B \bmod m)) \bmod m \tag{14}$$

number of the bits to be stored can be reduced. The maximum number of bits, needed to store the data according to this method, is $2k$, while number of iterations is $e - 1$. For large values of e computation time is too long.

These examples illustrate the computing complexity of encryption/ decryption. These methods are appropriate neither for hardware nor software implementation, because of a great number of bits needed to store intermediate results, as well as the great number of iterations. Reduction of the number of iterations can be done by conversion of the number e to its binary form:

$$e = (e_{k-1}, ..., e_1, e_0) = \sum_{i=0}^{k-1} e_i 2^i , \quad e_i \in \{0,1\} . \tag{15}$$

In this case, the computing is performed in k iterations including two ways of computing, *left-to-right* and *right-to-left*. Following pseudo-code describes both algortihms [7]:

right-to-left

result $C = P^e \bmod m$

1. $Y = 1, \quad Z = P$
2. $i = 0$ to $i = k - 1$
   a. if $e_i = 1$ then $Y = Y \cdot Z \bmod m$
   b. $Z = Z \cdot Z \bmod m$
3. output $C = Y$

left-to-right

result $C = P^e \bmod m$

1. $Y = 1$
2. $i = k-1$ to $i = 0$
   a. $Y = Y \cdot Y \bmod m$
   b. if $e_i = 1$ then $Y = Y \cdot P \bmod m$
3. output $C = Y$

The first algorithm has two variables *Z* and *Y*, which means one register more than for the second algorithm, which has only one variable, *Y*. In respect to speed, second algorithm requires two consecutive modular multiplications, within iteration, while the first one requires just one modular multiplication per iteration.

Beside these, several other encryption/decryption algorithms are developed, such as *m* methods, adaptive *m* methods, *addition chains*, *factor method*, *power tree*, *Montgomery* etc. [12]. Most of these methods use modular multiplication, so implementation of an efficient modular multiplication algorithm is of high importance. One of the most frequently used algorithms for modular computing of P^e is the *Montgomery* algorithm. It is very efficient and simple for hardware implementation and it is given by the following expression:

$$Monpro(A,B,m) = A \cdot B \cdot 2^{-k} \bmod m . \tag{16}$$

As seen from (16), multiplication contains number 2^{-k}, so it is necessary to adapt the form $Y \cdot Y$ to the form $AB2^{-k}$. To achieve this, it is necessary to perform *Montgomery* modular multiplication by number 2^{2k}, on the initial values. The result should be modularly multiplied by number 1. Putting *Montgomery* modular multiplication in the algorithms of *left-to-right* and *right-to-left* multiplication, we get *Montgomery* modular computation of P^e.

Montgomery modular multiplication algorithm is given by the following pseudo code:

Result $S = Monpro(A,B,m)$

1. $S = 0$
2. $i = 0$ to $i = k-1$
   a. $S = S + A \cdot b_i$ (1^{st} adder)
   b. $S = (S + S(0) \cdot m)/2$ (2^{nd} adder)

This code defines algorithm with two adders (*Montgomery_2a*). For hardware realization of this algorithm, one shift register, register for storing the variable *S*, two adders, and multiplexers for signal routing are needed. The

Montgomery algorithm can be implemented with one adder. Algorithm with one adder is given by the following pseudo code (*Montgomery_1a*):

Result $S = Monpro(A, B, m)$

1. $S = 0,\ Am = A + m$
2. $i = 0$ to $i = k - 1$

a. $case(B(i)\ \&\ A(0)\ \&\ S(0))$

$when(001 \mid 011) L = m$

$when(100 \mid 111)\ L = A$

$when(101 \mid 110) L = A \cdot m$

when others L=0

$S = (S + L) / 2$ (adder)

For this realization, following components are needed: one adder, one shift register, two registers for storing S and $A \cdot m$, and multiplexer logic for routing signal L. Both algorithms take k+1 iterations for computing.

Complete Montgomery algorithm by method *right-to-left and left-to-right* is given by the pseudo-code:

right-to-left

result $C = P^e \bmod m$

1. $K = 2^{2n} \bmod m$
2. $Z = Monpro(1, K, M)$
3. $P = Monpro(P, K, m)$
4. $i = 0$ to $i = k - 1$

a. if $e_i = 1$ then $Z = Monpro(Z, P, m)$

b. $P = Monpto(P, P, m)$

5. $Z = Monpro(1, Z, m)$
6. $C = Z$

left-to-right

result $C = P^e \bmod m$

1. $K = 2^{2n} \bmod m$
2. $Z = Monpro(1, K, M)$
3. $P = Monpro(P, K, m)$
4. $i = k - 1$ to $i = 0$

a. $Z = Monpto(Z, Z, m)$

b. if $e_i = 1$ then $Z = Monpro(Z, P, m)$

5. $Z = Monpro(1, Z, m)$
6. $C = Z$

It takes $k+2$ iterations for computing. Each iteration includes two *Montgomery* modular multiplications. Algorithm *right-to-left* takes two *Montgomery* modular multipliers working in parallel, and *left-to-right* algorithm takes one *Montgomery* modular multiplier that works sequentially.

4 FPGA Implementation

In this paper, implementation of the RSA algorithm is made on FPGA integrated circuit EP4CE115F29C7, family Cyclone IV, Altera [13]. This component contains 266 embedded multipliers (18 x18 bits), 4 PLL blocks, 3888 Kbits of embedded memory, 528 I/O pins and 114480 logic elements. Preference for FPGA circuit relies on availability, easiness of system testing, flexibility, relatively good performance in terms of speed and power consumption.

Four modules for RSA encryption are implemented. Two of them implement the *Montgomery* algorithm *right-to-left* with one adder (*Montgomery_rl_1a*) and with two adders (*Montgomery_rl_2a*). Another two modules use the *Montgomery* algorithm *left-to-right* with one adder (*Montgomery_lr_1a*) and with two adders (*Montgomery_lr_2a*).

As mentioned before, the RSA algorithm is symmetric, so the same module may be used for encryption, as well as for decryption. The modules are designed using VHDL. Synthesis and simulation were done using Quartus II software and ModelSim. The RSA algorithm implementation using *Montgomery* modular multiplication is quite simple and suitable for hardware implementation, hence following key lengths (k) are achieved: 16, 32, 64, 128, 256, 512 and 1024. The analysis of implemented modules shows the number of needed resources, number of clocks for encryption, as well as maximum operating frequency of the modules.

Table 1 presents results of the analysis in the means of logic resources needed for implementation of the *Montgomery rigth-to-left* algorithm.

Table 1

Number of logic elements for the Montgomery right-to-left algorithm.

k	*Montgomery_rl_1a*	*Montgomery_rl_2a*
16	395	386
32	746	822
64	1451	1575
128	2738	3117
256	5427	6189
512	10808	11602
1024	22776	24132

Table 2 gives results of the analysis with respect to the logic resources needed for implementation of the *Montgomery left-to-right* algorithm.

Table 2
Number of logic elements for the Montgomery left-to-right algorithm.

k	*Montgomery_lr_1a*	*Montgomery_lr_2a*
16	329	322
32	621	683
64	1195	1257
128	2231	2483
256	4408	4916
512	9356	10317
1024	18701	20584

From the results given in the **Table 1** and **Table 2**, *Montgomery right-to-left* implementation occupies more logic resources than *left-to-right*. This is due to the fact that implementation of *right-to-left* requires two *Montgomery* modular multipliers, while implementation of *left-to-right* requires one *Montgomery* modular multiplier. Implementation of *Montgomery* modular multiplication with one adder requires less resource then implementation with two adders. For addition, arithmetic operation defined in the package *ieee.numeric_std* was used. With this implementation of adders, the realization takes logical elements connected in series, which works in arithmetic mode. One *k* bit adder takes *k* logical elements. Reduction of number of *k* bits adders saves the resources. For key length of 1024 bits, the least resources requires *Montgomery_ld_1a* implementation, with 18701 logic elements.

Maximum operating frequency analysis was performed by using *TimeQuest Timing Analyzer* included in the *Quartus II* software. The results for the *Montgomery right-to-left* algorithm are presented in the **Table 3**, and for the *Montgomery left-to-right* algorithm in the **Table 4**.

Table 3
Maximum operating frequency of Montgomery right-to-left implementation [μs].

k	*Montgomery_rl_1a*	*Montgomery_rl_2a*
16	250	250
32	121.2	111.73
64	91.74	83.41
128	64.86	63.29
256	41.09	37.42
512	24.19	22.82
1024	13.19	12.73

Table 4
Maximum operating frequency of Montgomery left-to-right implementation [μs].

K	*Montgomery_lr_1a*	*Montgomery_lr_2a*
16	250	250
32	115.55	108.23
64	96.32	88.64
128	64.91	62.45
256	41.44	41.45
512	24.31	23.68
1024	13.31	12.86

The greatest maximum operating frequency has *Montgomery_ld_1a* implementation. This is caused by the fact that it requires less resources, shorter routing links, which results in shorter propagation time. The lowest maximum operating frequency has *Montgomery_dl_2a*. This is due to the fact that it requires the most resources, longer routhing links, thereby greater propagation time. For key length of 1024 bits, *Montgomery_ld_1a* implementation has the highest operating frequency, i.e. 13.31 MHz.

To encrypt one data in *Montgomery right-to-left* implementation, it takes $(k+3)(k+2)$ cycles, where each of $k+3$ of modular P^e computation cycles requires $k + 2$ cycles for modular multiplying. *Montgomery left-to-right* implementation requires $2(k+3)(k+2)$ cycles, where each of $2(k+3)$ of modular P^e computation cycles requires $k + 2$ cycles for modular multiplying. *Left-to-right* implementation requires twice more cycles than *right-to-left* implementation. This is due to the fact that *left-to-right* implementation requires one *Montgomery* modular multiplier that works sequentially, and *right-to-left* implementation requires two *Montgomery* modular multipliers that works in parallel.

Combination of the results for maximum operating frequency (**Table 3** and **Table 4**), number of cycles for encryption and key length yields maximum data encryption speed in bits per second, as a function of the key length (*maxfreq*·*k*/*cycles*). In the **Table 5** the results for *right-to-left* implementation are presented, while the **Table 6** gives the results for left-to-*right* implementation. From the analyzis of the results given in the **Table 5** and in the **Table 6**, it is obvious that *Montgomery_dl_1a* implementation has maximum speed of encryption, because in this implementation *Montgomery* modular multipliers works in parallel (less cycles for compytation), and *Montgomery* modular multiplier use one adder (less logic elements, less delay). An increase of the key length, yields reduction of encryption speed. reduces. For key length of 1024 bits, maximum encryption speed is 12.81 kb/s. If implementation with less resources is used, maximum encryption speed is achieved by *Montgomery_ld_1a* implementation, with 6.46 kb/s.

Table 5
Maximum speed of encryption for Montgomery right-to-left implementation [kb/s].

k	*Montgomery_dl_1a*	*Montgomery_dl_2a*
16	11695.91	11695.91
32	3259.16	3004.5
64	1327.76	1207.2
128	487.49	475.69
256	157.41	143.35
512	46.78	44.13
1024	12.81	12.37

Table 6
Maximum speed of encryption for Montgomery left-to-right implementation [kb/s].

k	*Montgomery_ld_1a*	*Montgomery_ld_2a*
16	5847.95	5847.95
32	1553.61	1455.19
64	697.02	641.44
128	243.93	234.69
256	79.37	79.39
512	23.51	22.9
1024	6.46	6.24

5 Conclusion

Four FPGA modules, which implement the RSA encryption algorithm, are made on Altera's EP4CE115F29C7 circuit. Synthesis and simulation has been performed using Quartus II and ModelSim software. For exponentiation, the binary algorithm has been used, while for modular multiplications, the *Montgomery* algorithm has been used. Selected FPGA device allows key lengths of 16, 32, 64, 128, 256, 512 and 1024 bits. Number of required logic elements increases with the key length. *Right-to-left* implementation occupies more resources than *left-to-right* implementation. Also, *Montgomery* modular multiplication with one adder occupies fewer resources than implementation with two adders. The least resources take *Montgomery_ld_1a* implementation. For key length of 1024 bits, *Montgomery_ld_1a* takes 18701 logic elements. *Right-to-left* implementation has greater encryption speed than *left-to-right* implementation. Maximum encryption speed can be achieved using *Montgomery_dl_1a* implementation. For key length of 1024 bits, *Montgomery_dl_1a* has encryption speed of 12.81 kb/s.

6 References

[1] A.S. Tanenbaum: Computer Networks, Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[2] B. Schneier: Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley and Sons, NY, USA, 1996.

[3] R.L. Rivest, A. Shamir, L. Adleman: A Method for Obtaining Digital Signatures and Public-key Crypto Systems, Communications of the ACM, Vol. 21, No. 2, Feb. 1978, pp. 120 – 126.

[4] A. Karaca, O. Cetin: A Robust Real-time Secure Communication approach Over Public Switched Telephone Network, Journal of Naval Science and Engineering, Vol. 7, No. 1, April 2011, pp. 37 – 47.

[5] K. Chakravarthy, M. Srinivas: Speech Encoding and Encryption in VLSI, Asia and South Pacific Design Automation Conference, Kitakyushu, Japan, 21 – 24 Jan. 2003, pp. 569 – 570.

[6] M.I. Ibrahimy, M.B.I. Reaz, K. Asaduzzaman, S. Hussain: FPGA Implementation of RSA Encryption Engine with Flexible Key Size, International Journal of Comunication, Vol. 1, No. 3, 2007, pp. 107 – 113.

[7] C.K. Koc: High-speed RSA Implementation, RSA Laboratories, Redwood City, CA, USA, Nov. 1994.

[8] S.K. Sahu, M. Pradhan: FPGA Implementation of RSA Encryption System, International Journal of Computer Applications, Vol. 19, No. 9, Apr. 2011, pp. 10 – 12.

[9] R. Ghayoula, E. Hajlaoui, T. Korkobi, M. Traii, H. Trabelsi: FPGA Implementation of RSA Cryptosystem, International Journal of Social, Human Science and Engineering – World Academy of Science, Engineering and Technology, Vol. 2, No. 8, 2008, pp. 848 – 852.

[10] J. Fry, M. Langhammer: RSA and Public Key Cryptography in FPGAs, Technical Report TR CF-032305-1.0, Altera Corporation, 2005.

[11] A. Anand, P. Praveen: Implementation of RSA Algorithm on FPGA, International Journal of Engineering Research and Technology, Vol. 1, No. 5, July 2012.

[12] P.L. Montgomery: Modular Multiplication without Trial Division, Mathematics of Computation, Vol. 44, No. 170, April 1985, pp. 519 – 521.

[13] Cyclone IV EP4CE115F29C7 Data Sheets. http://www.altera.com.