A HIGH-SPEED, LOW LATENCY RSA DECRYPTION SILICON CORE

Ciaran McIvor, Máire McLoone, John V McCanny

DSiP Laboratories, School of Electrical and Electronic Engineering,
The Queen's University of Belfast, Northern Ireland.
E-Mail: c.mcivor@ee.qub.ac.uk, maire.mcloone@ee.qub.ac.uk, j.mccanny@ee.qub.ac.uk

Abstract

This paper introduces a novel and generic approach to the hardware implementation of the RSA decryption function, which may be used to create digital signatures in an RSA based signature scheme. The algorithm used for modular multiplication is Montgomery's multiplication algorithm. The design is speed optimised and as such employs the R-L binary method as a means for modular exponentiation. An RSA decryption can be performed in only $(k/2 + 3)^2$ clock cycles, where k is the size of the modulus, by employing carry save adders in order to achieve fast parallel addition and the Chinese Remainder Theorem to speed up exponentiation. To the authors' knowledge, this is the lowest number of clock cycles required for any radix 2 based RSA decryption system reported in the literature. As such the design can achieve a data throughput rate of 234.47 kb/s for a 512-bit modulus and a rate of 90.58 kb/s for a 1024-bit modulus when implemented onto a Xilinx Virtex2 XC2V8000 chip.

1. Introduction

The increasing use of e-commerce and the increase in demand for secure communications over the Internet, for example, has led to an ever-greater demand for reliable high-speed security products, which can carry out data encryption in real time. In particular, public-key cryptosystems provide a mechanism for secure symmetric key exchange, as well as being a useful method for creating and verifying digital signatures.

The RSA system [1] is one of the most commonly used public-key cryptosystems and is the subject of this paper. An RSA operation is essentially a modular exponentiation, which in turn requires modular multiplication. Also, in order for an RSA cryptosystem to be considered secure, it is widely recognised that key sizes should be in excess of 1024-bits [2], [3] which, in many cases, can only be achieved through direct hardware implementation. Thus, it follows that RSA cryptography is extremely computationally intensive.

This paper therefore introduces a novel hardware design aimed at tackling this problem, with a view to achieving a high RSA exponentiation throughput rate when implemented in silicon. The focus of the paper is on RSA decryption and all results are given in relation to the decryption function. The RSA private exponent d may be as large as $[(p-1)(q-1)-1]$, where p and q are two large prime numbers and the k-bit modulus n is equal to $p*q$. Thus, it is assumed that the private exponent d is a k-bit number. This is in contrast to the RSA public exponent e, which can be chosen arbitrarily (usually 3, 17 or $2^{16}+1$) as long as it satisfies the conditions stated in section 2 of this paper. In general, therefore, the RSA decryption function is more computationally intensive than the RSA encryption function. Also, RSA decryption is usually the most common operation carried out in applications such as smart cards. However, as both functions essentially perform identical arithmetic operations, the approach described in this paper can also be utilised to carry out encryption.

Previous RSA designs have mainly focused on two particular strategies. Firstly, the Montgomery multiplication algorithm [4], [5], [6] has been used alongside a redundant number representation [7], [8] in order to avoid long carry propagation. Secondly, a number of 2-D systolic array designs have been proposed [9], [10] for use alongside the Montgomery multiplication algorithm and typically consist of a k×k matrix of 1-bit processing elements. However, these arrays have proved costly to implement due to the amount of resources required since RSA cryptosystems typically need a modulus of around 1024-bits in length.

The approach described here utilises the Montgomery multiplication algorithm [4], [5], [6] for carrying out the modular multiplication operation. The corresponding modular exponentiation algorithm is the well-known R-L binary method [11], [12]. The design also makes extensive use of carry save adders as a means of fast, parallel addition. By combining these along with the Chinese Remainder Theorem (CRT) [13], an architecture has been developed, which has a latency of only $(k/2+3)^2$ clock cycles per decryption. To the authors' knowledge this is the lowest number of clock cycles required for any radix 2 based RSA decryption system reported in the literature. Carry propagation is avoided by using carry save adders and as such the corresponding combinational logic between clock cycles means that a good clock speed can still be achieved. Thus, since an RSA decryption requires a very low number of clock cycles, a high data throughput rate is achievable.

The concepts developed have been verified and demonstrated through an FPGA implementation, in this case using a Xilinx Virtex2 XC2V8000 chip. However, as will be discussed, the design is completely technology independent and can therefore just as easily be implemented onto ASIC technology.

The paper is arranged as follows. The RSA cryptosystem and modular exponentiation are summarised in section 2. The silicon implementation is discussed in section 3. Performance results are provided in section 4 and some concluding remarks are given in section 5.

2. Background Information

2.1 RSA Cryptography

The RSA cryptosystem as proposed by Rivest, Shamir and Adleman in 1978 [1] is summarised as follows:

0-7803-7761-3/03/$17.00 ©2003 IEEE

Step 1: Two prime numbers p and q are generated and their product is calculated, denoted by, $n = p*q$. n is called the modulus.
Step 2: A positive integer e is then determined, satisfying,
$3 \leq e < (p-1)(q-1)$;
e must also be relatively prime to $(p-1)(q-1)$.
Step 3: e is then used to determine another positive integer d for which, $ed = 1 \pmod{(p-1)(q-1)}$. i.e. d is the multiplicative inverse of e, modulo $(p-1)(q-1)$.
Step 4: Thus, the public key is represented as the pair (e, n) and the private key is represented as the pair (d, n).
Step 5: If M represents the plaintext and C represents the ciphertext, then encryption is carried out using the encryption function $C = M^e \pmod n$. Decryption is carried out in an almost identical manner using the function $M = C^d \pmod n$.

The underlying mathematics demonstrating the correctness of the RSA algorithm is given in [1]. It should also be noted that no public-key cryptosystem has ever been proven to be secure. The security of public-key systems is based on the presumed difficulty of a small set of number theory problems. The security of the RSA system lies in the fact that it would be computationally infeasible to calculate the value of d given only the public key pair (e, n), assuming that large enough prime numbers p and q are used in determining e, d and n. This would be equivalent to trying to factorise a 1024-bit number n, which theoretically has only two factors p and q. This will eventually be achieved but at present with current technology it is sufficient that n should be between 1024-bits and 2048-bits in length depending on the level of security required in the system.

2.2 Montgomery Multiplication

This section describes the Montgomery multiplication algorithm, which originated from Montgomery's paper published in 1985 [4]. In order for the Montgomery multiplication algorithm to operate correctly it is necessary for the radix to be relatively prime to n. This condition is satisfied here as n is clearly the product of two large prime numbers and the radix is equal to 2.

Now, given an integer $a < n$, α is said to be its n-residue with respect to r if, $\alpha = a*r \pmod n$, where $r = 2^k$. Likewise, given an integer $b < n$, β is said to be its n-residue with respect to r if, $\beta = b*r \pmod n$. The Montgomery product of α and β can then be defined as, $z = \alpha*\beta*r^{-1} \pmod n$, where r^{-1} is the inverse of r, modulo n. A variant of Montgomery's multiplication algorithm [5], which computes the Montgomery product of α and β, is presented in pseudo code below. All numbers are represented in binary form where n is the k-bit modulus, α and β are the k-bit operands and $\alpha, \beta < n$.

Algorithm 1: *Montgomery Multiplication* (α, β, n)
S[0] <= (others => '0');
for i in 0 **to** k-1 **loop**
    q_i <= $(S[i]_0 + \alpha_i * \beta_0)$ **mod** 2;
    S[i+1] <= $(S[i] + \alpha_i * \beta + q_i * n)$ **div** 2;
end loop;
return S;

2.3 The R-L Binary Method

The R-L binary method utilizes the Montgomery multiplication algorithm, which determines its intermediate results. Some on-chip pre-computation is required in order to convert the ordinary residue inputs to the chip into their n-residue form for use with algorithm 1, as described in section 2.2. In order to calculate the correct n-residues, it is necessary to pre-compute the value $2^{2k} \pmod n$ externally, which can then be stored alongside the relevant private or public keys. This value is relatively inexpensive to compute as once it is determined it need only be changed if the value of the modulus n changes. The value $2^{2k} \pmod n$ is used to convert (on-chip) the ordinary residue operands a and b into their n-residue values α and β respectively as follows:

$$\text{MontMult}(a, 2^{2k} \pmod n, n) = a*2^{2k}*r^{-1} \pmod n = a*r \pmod n = \alpha \quad \text{(i)}$$

b is converted to β in an identical manner. Also, it can be shown that the Montgomery product of two n-residues is indeed an n-residue itself as follows:

$$z = \text{MontMult}(\alpha, \beta, n) = a*r*b*r*r^{-1} \pmod n = a*b*r \pmod n \quad \text{(ii)}$$

Therefore, as modular exponentiation here is simply a series of Montgomery multiplications, then the result from the loop statement of algorithm 2 below will also be in n-residue form (i.e. $R_k = M*r \pmod n$). Thus, a further Montgomery multiplication is required in order to convert this result back into its ordinary residue form as follows:

$$M = R_k*r^{-1} \pmod n = 1*R_k*r^{-1} \pmod n = \text{MontMult}(1, R_k, n) \quad \text{(iii)}$$

Now, if the computation to be carried out is $M = C^d \pmod n$, where d is the k-bit private exponent (as d may be as large as $[(p-1)(q-1)-1]$), then the following modular exponentiation algorithm may be used.

Algorithm 2: *Modular Exponentiation (C, d, n)*
$K = 2^{2k} \pmod n$; (computed externally)
P_0 <= MontMult (K, C, n);
R_0 <= MontMult (K, 1, n);
for i in 0 to k-1 **loop**
if $d_i = 1$ **then** R_{i+1} <= MontMult (R_i, P_i, n) **end if**;
    P_{i+1} <= MontMult (P_i, P_i, n);
end loop;
M <= MontMult (1, R_k, n);
return M;

2.4 The Chinese Remainder Theorem

The basic application of the CRT to achieve faster RSA decryption is outlined in this section. A more detailed explanation of the background is given in [13]. As outlined in section 2, the RSA decryption function is given by $M = C^d \pmod n$, which in turn can be written as, $M = C^d \pmod{p*q}$, since $n = p*q$. Fundamentally, the CRT allows this computation to be broken down into two smaller, less complex computations given as follows,

$$M_p = C_p^{Dp} \pmod p$$
$$M_q = C_q^{Dq} \pmod q$$

where

$$C_p = C \pmod p,\ D_p = d \pmod{(p-1)}$$
$$C_q = C \pmod q,\ D_q = d \pmod{(q-1)}$$

The final value M is then obtained using the equation,

$$M = [(M_q - M_p)(p^{-1} \pmod q) \pmod q]p + M_p \qquad \textbf{(iv)}$$

3. Novel Hardware Implementation

The approach used to implement the RSA system in hardware is based on algorithm 2 in section 2.3. This incorporates algorithm 1 in section 2.2. The CRT is also applied to significantly speed up the RSA decryption computation. A modulus size of 512-bits has been chosen here for illustrative purposes. However, by capturing this generically (in VHDL) the modulus size is completely variable and is easily changed by altering two parameters.

3.1 Montgomery Exponentiation Overview

Figure 3.1 provides an overview of the Modular exponentiation design approach used. Even though the modulus size is 512-bits, all major inputs to the exponentiation core are only 256-bits due to the use of the CRT. This clearly illustrates the speed advantage that CRT based designs hold over alternative structures. The signals p and q represent the two large prime numbers that form the modulus $n = p*q$. The signals C_p and C_q correspond to the ciphertext, which is to be decrypted. The symbols D_p and D_q refer to the corresponding values outlined in section 2.4. The signal *p_constant* is the externally computed value $2^{2(k/2)} \pmod p$ and corresponds to the value $2^{2k} \pmod n$ as discussed in section 2.3. Likewise, the signal *q_constant* signifies the value $2^{2(k/2)} \pmod q$. When the *ready* signal is set to high this indicates that the correct values of the decrypted ciphertext M_p and M_q should now be registered. The component *COUNTER* essentially implements the iterative loop statement of algorithm 2 in section 2.3.

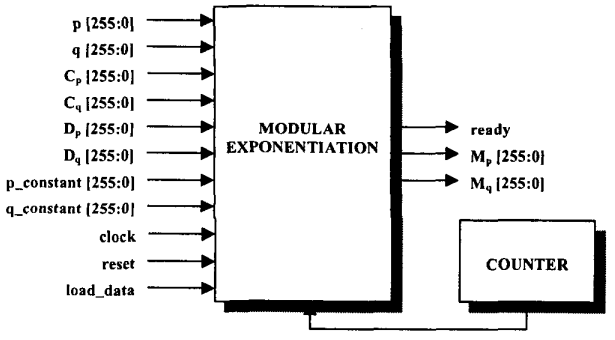

Figure 3.1: Overview of Modular Exponentiation Core

3.2 Montgomery Multiplication Overview

Figure 3.2 gives an outline of the Montgomery multiplier architecture developed. Three rounds of carry save logic are utilised to construct a 5-input carry save adder, which is used to calculate the intermediate results S from algorithm 1 in section 2.2. These intermediate results are kept in redundant form until the loop statement of algorithm 2 in section 2.3 has terminated, at which point the redundant representation of the output from the exponentiation algorithm is added together using a 1-D array of full adders. The 5-input carry save adder requires only one clock cycle to produce its output and so each iteration of the loop statement of algorithm 1 requires only one clock cycle to compute.

The inputs *a1*, *a2*, *b1* and *b2* correspond to the α and β inputs of algorithm 1. Again the iterative 'for loop' is represented by the component *COUNTER* in figure 3.2. The function of the array of *FA* (full adder) cells are to perform a summation of the carry save representation of the α input to algorithm 1. This is to ensure that the correct values of the α_i are used at each iteration of algorithm 1. These values are registered in the *FA REGISTER* but are readily available when they are required at each iteration of the loop in algorithm 1.

The main purpose of the *Q_ADDER* component is to determine the q_i values of algorithm 1. These, in turn, are fed into a multiplexer to determine whether or not the modulus should be fed into the *CSA* component along with other relevant input values needed to calculate the intermediate values S at each iteration of the loop statement in algorithm 1.

When algorithm 2 has terminated the correct outputs from the *CSA REGISTER* are then added together to form the correct output value M_p as shown in figure 3.1. The output value M_q is computed using a separate Montgomery multiplier that operates in parallel with the M_p calculations.

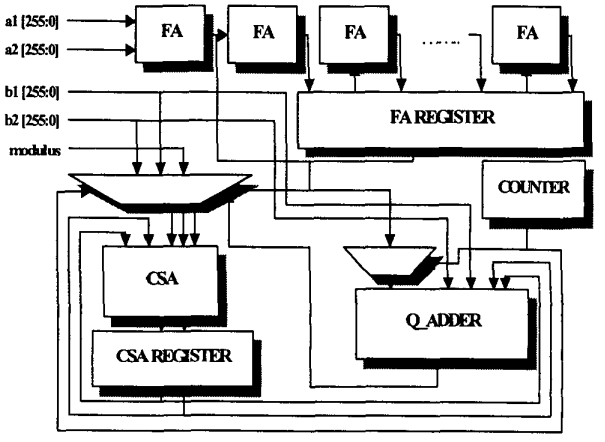

Figure 3.2: Overview of Montgomery Multiplication Component

4. Performance Results

The system described has been captured generically using VHDL and used to create a silicon demonstrator using a Xilinx Virtex2 XC2V8000BF957-5 device. Table 1 provides performance and area values obtained for varying bit lengths k.

The performance results can be compared with those recently reported in the literature. Daly and Marnane [11] recently reported a rate of 115.5 kb/s for a 480-bit modulus and 45.8 kb/s for a 1080-bit modulus, which are only half the speed of the equivalent throughput values reported in Table 1. Their design was targeted for a Xilinx Virtex V1000FG680-6 device.

Bit Length (k)	No. of CLB Slices	Data Throughput Rate	Time For One RSA Decryption (ms)
128	3110	1.51 Mb/s	0.08
256	6147	537.63 kb/s	0.48
512	12041	234.47 kb/s	2.18
1024	24174	90.58 kb/s	11.30

Table 1: Performance and Area Results Obtained

Blum and Paar [2] presented an FPGA systolic array implementation and were able to process a 1024-bit RSA decryption in 10.18ms, 12.41ms and 12.52ms dependent on the size of the processing element u=4, u=8 and u=16 respectively. These times are comparable to the 1024-bit decryption time reported in Table 1. Their design was implemented onto the Xilinx XC4000 series of FPGAs. It should be noted that these implementations may yield better performance results if implemented onto the latest Xilinx FPGAs such as the Virtex2 family of devices. However, a major drawback of their implementations is that they are technology dependent and targeted to exploit certain features of specific Xilinx devices. For example, Daly and Marnane [11] exploit the fast carry chains of their chosen Xilinx Virtex device. Similarly, Blum and Paar [2] rely on the dedicated hardware within their target device, which accelerates the carry path of adders and counters. This is in contrast to the design presented, which is completely technology independent. Thus, even higher data rates should be achievable when the design is migrated to modern ASIC technology. Previous experience would indicate that a further doubling in performance is possible if the design is implemented in a 0.18μm CMOS. However, the FPGA implementation presented still compares very favourably with recent VLSI designs. Kwon *et al.* [14] report an RSA architecture, which takes 22ms to complete a 1024-bit RSA decryption. This is approximately only half the speed achieved by the design presented in this paper.

5. Conclusion

A novel approach to the silicon design and implementation of RSA decryption cores has been presented. This uses a combination of a 5-input carry save adder and redundant representation of the intermediate results of algorithm 1 and algorithm 2 to achieve a latency of $(k/2 + 3)^2$ clock cycles, where k is the length of the modulus. We believe that this latency is the smallest reported to date. This combined with a shallow logic depth, achieved using the fast, parallel carry save approach, produces a circuit with very impressive throughput rates. This has been demonstrated through a generic implementation on a Xilinx Virtex2 XC2V8000 FPGA. The design is also completely technology independent and can be migrated to ASIC or other FPGA / PLD technologies.

6. Acknowledgements

This research has been funded by Amphion Semiconductor Ltd. and by a Northern Ireland Department of Learning postgraduate studentship in the form of a CAST award.

7. References

[1] R.L. Rivest, A. Shamir, L. Adleman: "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". Communications of the ACM, 21(2): 120–126, February 1978.

[2] T. Blum, C. Paar: "Montgomery Modular Exponentiation on Reconfigurable Hardware". Proceedings 14th Symposium on Computer Arithmetic, pp. 70–77, 1999.

[3] A. J. Elbirt, C. Paar: "Towards an FPGA Architecture Optimized for Public-Key Algorithms". SPIE Symposium on Voice, Video and Communications, Sept 1999.

[4] P.L. Montgomery: "Modular Multiplication without Trial Division". Math. Computation, Vol. 44, 1985, pp. 519–521.

[5] C.D. Walter: "Montgomery Exponentiation Needs No Final Subtractions". Electronics Letters, 35(21): 1831–1832, October 1999.

[6] C.K. Koc, Tolga Acar, Burton S. Kaliski Jr.: "Analyzing and Comparing Montgomery Multiplication Algorithms". IEEE Micro, 16(3): 26–33, June 1996.

[7] C.D. Walter: "Fast Modular Multiplication using 2-Power Radix". International Journal of Computer Mathematics, Vol. 3, pp. 21–28, 1991.

[8] S.E. Eldridge, C.D. Walter: "Hardware Implementation of Montgomery's Modular Multiplication Algorithm". IEEE Transactions on Computers, Vol. 42, pp. 693–699, July 1993.

[9] A. Tiountchik: "Systolic Modular Exponentiation via Montgomery Algorithm". Electronics Letters, Vol. 34, pp. 874–875, April 1998.

[10] C.D. Walter: "Systolic Modular Multiplication". IEEE Transactions on Computers, Vol. 42, pp. 376–378, Mar 93.

[11] A. Daly, W. Marnane: "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". FPGA 2002, February 2002.

[12] C.K. Koc: "High-Speed RSA Implementation". Technical Report, RSA Laboratories, RSA Data Security, Inc., Redwood City, CA, 1994.

[13] J. Quisquater, C. Couvreur: "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem". Electronics Letters, Vol. 18, pp. 905–907, October 1982.

[14] Taek-Won Kwon, Chang-Seok You, Won-Seok Heo, Yong-Kyu Kang, Jun-Rim Choi: "Two Implementation Methods of a 1024-bit RSA Cryptoprocessor Based on Modified Montgomery Algorithm". IEEE International Symposium on Circuits and Systems, pp. 650–653, May 2001.