An RSA Co-processor Architecture Suitable for a User-Parameterized FPGA Implementation

Joseph R Laracy
Seton Hall University
400 South Orange Ave
South Orange, New Jersey 07079
USA
joseph.laracy@shu.edu

ABSTRACT: *This paper describes an original and straightforward architecture for a logic circuit implementation of the RSA algorithms. The architecture is ideal for teaching advanced undergraduate or graduate students topics associated with public-key cryptography and digital system design. The system is designed with VHDL for execution on a FPGA. Software implementations of RSA running on standard PCs are relatively slow as standard microprocessors are not optimized for the operations that RSA must carry out. A key aspect of this approach is the use of Montgomery Multiplication, a method for performing fast modular multiplication.*

Keywords: Public-Key Cryptography, RSA, Digital Systems, Computer Engineering, Logic Design, Education

Received: Received: 12 December 2019, Revised March 5, 2020, Accepted March 12, 2020

DOI: 10.6025/jisr/2020/11/2/46-53

1. Introduction

In an important article in *Electronic Engineering Times*, John Fry and Martin Langhammer clearly describe the downsides of software based RSA systems as well as ASSP (application specific standard products) implementations. Software systems are too slow, while ASSPs are inflexible and expensive. They suggest that a programmable logic implementation would be ideal. Specifically, a VHDL (Very High-Speed Integrated Circuit Hardware Description Language) solution could offer a high performance, user-parameterized, synthesizable core. A RSA coprocessor could be integrated into a larger information system to provide strong RSA encryption for email communication. An FPGA implementation could easily connect to a high speed PC I/O port and provide all the encryption and decryption operations for secure email. In this concise paper, we describe our implementation of Fry and Langhammer's proposal, one that could ideally be presented to advanced undergraduate or beginning graduate students interested in computer engineering and cryptography [1].

2. Fundamentals of RSA

RSA is a public-key cryptographic algorithm developed by Ronald **R**ivest, Adi **S**hamir, and Leonard **A**dleman (hence RSA). The goal of their research was to develop a cryptosystem whereby communicating partners could send and

receive secure messages without having to worry about key integrity. Their solution was to develop a public key that could be published in a directory. This would allow anyone to encrypt a message for a particular recipient. The public key would be generated with a one-way (non-reversible) function. To decrypt a message, the recipient uses his or her private key. If two partners want to communicate without a third party listening in, the pair no longer needs to securely exchange their respective keys [2].

The RSA algorithm is rigorously defined as follows [3]:

1. Let $p \neq q$ be large primes, and define $n = pq$ and $\varphi = (p - 1)(q - 1)$.
2. Choose random encryption exponent e, $1 < e < \varphi$, so that $(e, \varphi) = 1$.
   (The notation $(e, \varphi) = 1$ denotes the greatest common divisor. Therefore e and φ are relatively prime.)
3. Use the Euclidean algorithm to find the decryption exponent d, $1 < d < \varphi$, so that $ed \equiv 1 \bmod \varphi$.
4. Define f: $\mathbb{Z}_n \rightarrow \mathbb{Z}_n$ by $f(m) = m^e \bmod n$.

Very interestingly, Roland Schmitz has shown that the security of RSA cryptographic systems, when interpreted as dynamical systems, relies on the discrete chaoticity of their orbits. His research demonstrates that the algebraic requirements for the RSA parameters may be translated into statements about the period lengths of the orbits [4]. Schmitz writes,

> The cipher $c = m^e \bmod n$, where m is the message, e is the public exponent and n is the public module, can be viewed as the e-th member of the orbit generated by x_0=1 for the discrete iteration $x_{k+1} = F_m(x_k)$, $F_m(x) = m \cdot x \bmod n$, $x_0 = 1$. The problem of finding m, when only c is known, can be stated as "*Given the e-th point of an orbit with known initial value and parameterized map, find the value of the parameter*." The receiver of the cipher, however, can use her private key d to recover m through the identity $m = c^d \bmod n = m^{ed} \bmod n$. The additional knowledge provided by d can thus be stated in dynamical systems' terms as "*The orbit generated by x_0=1 and the map $F_m(x) = m \cdot x \bmod n$ had period ed*." Since factorization of the public module n yields the private key d, the factorization problem can also be viewed as directly related to period finding in discretely chaotic orbits [5].

The following simple example clearly illustrates the steps involved in RSA protected communication. Suppose the fictional characters, Eric and Matt, want to communicate without Joe listening in to their conversation. Matt will encrypt an ASCII character using RSA.

1. Eric must pick two large prime numbers, p and q. (This example uses very small numbers for simplicity.)
   $p = 17$, $q = 11$.
2. Eric then computes $N = pq = 187$.
3. Eric picks a number $e = 7$; e should be relatively prime to $((p-1)(q-1)) = \varphi$.
4. Eric can now post the pair (N, e) in a public directory.
   Everyone in the directory may share the same e, but have different N's.
5. Eric also computes d, where $e \times d \equiv 1 \pmod{\varphi}$.
   $d = 23$. d is Eric's private key.
6. Matt wishes to tell Eric he can attend the secret meeting so he sends the character Y (for "Yes").
   The message in this case, M = "Y" (for "Yes") = 89.
7. Matt encrypts his message using the formula: $C = M^e \pmod N = C = 89^7 \pmod{187} = 166$, and sends 166 down the channel.
8. Eric decrypts Matt's message using the formula: $M = C^d \pmod N = 166^{23} \pmod{187} = 89$.
   89 = "Y" in ASCII. (Meaning "Yes")
   Eric knows Matt can participate in the meeting.

Because exponentiation in modular arithmetic is a one-way function, Joe cannot determine M from C. Joe's one hope is to factor N into its respective primes. However, in real systems where N is a 2048-bit number, this is highly computationally expensive and therefore infeasible.

Menezes, van Oorschot, and Vanstone provide an elegant proof of decryption that we faithfully present here [6].

Since $ed \equiv 1 \pmod{\varphi}$, there exists an integer k such that $ed = 1 + k\varphi$. Now, if $\gcd(m,p) = 1$ then by Fermat's theorem,

$$m^{p-1} \equiv 1 \pmod{p}.$$

Raising both sides of this congruence to the power $k(q-1)$ and then multiplying both sides by m yields

$$m^{1+k(p-1)(q-1)} \equiv m \pmod{p}.$$

On the other hand, if $\gcd(m,p) = p$, then this last congruence is again valid since each side is congruent to 0 modulo p.

Hence, in all cases

$$m^{ed} \equiv m \pmod{p}.$$

By the same argument,

$$m^{ed} \equiv m \pmod{q}.$$

Finally, since p and q are distinct primes, it follows that

$$m^{ed} \equiv m \pmod{n}$$

and, hence,

$$c^d \equiv (m^e)^d \equiv m \pmod{n}.$$

Therefore, decryption works!

3. Montgomery Reduction

In 1985, Peter L. Montgomery developed a method for modular reduction that did not require explicit division. His method allows multiplications modulo an odd number to be replaced with multiplications modulo an even number, 2^k [7]. Recall that division by a power of 2 is accomplished in a binary digital system simply by right shifts [8]. RSA cryptography requires modular exponentiation of very large numbers [9]. Traditional mathematical methods generate partial results in excess of 2048-bits and require division by a similarly large number.

Montgomery defines n and r to be relatively prime integers where r^{-1} is the multiplicative inverse of $r \bmod n$ and n^{-1} is the multiplicative inverse of $n \bmod r$. He also defines $n' = -n^{-1} \bmod r$ and $m = tn' \bmod r$ where t is an integer defined by

$$(t + mn)/r \equiv tr^{-1} \bmod n$$

The right side of this congruence is much more computationally expensive than the left. On the left side of the congruence, we take congruences mod r and division by r. Because $t + mn \equiv 0 \bmod r$, the division has no remainder. By choosing r as some 2^s, where $2^{s-1} \le n < 2^s$, $x \bmod r$ can be computed by shifting x s-bits to the right. To apply this principle, and perform Montgomery Reduction, we shift our computation modulo n into a complete residue system.

$$R = R(r, n) = \{ir \bmod n \mid 0 \le i < n\}$$

One can then define a Montgomery Product "x" of a and b in R:

$$a \times b = abr^{-1} \bmod n$$

Suppose $0 \le a, y < n$. Define $a' = aR \bmod n$, and $y' = yR \bmod n$. Therefore, the Montgomery Reduction of $a'b'$ is $a'b'R^{-1} \bmod n = abR \bmod n$.

Montgomery Exponentiation is the next logical step in this sequence. For example, to compute $a^5 \bmod n$ for integer a, $1 \leq a < n$, first compute $a' = aR \bmod m$. After that, determine the Montgomery Reduction of $a'a'$, $X = a'^2R^{-1} \bmod n$. Then compute the Montgomery Reduction of X^2, $X^2R^{-1} \bmod n = a'^4R^{-3} \bmod n$. The last Montgomery Reduction is applied to $(X^2R^{-1} \bmod n)\ a' = (X^2R^{-1})\ a'R^{-1} \bmod n = a'^5R^{-4} \bmod n = a^5R \bmod n$. The result must then be converted back into the integer domain by multiplying it by $R^{-1} \bmod n$ and reducing modulo n, yielding $a^5 \bmod n$ [10].

4. Algorithms

An algorithm used to accomplish Montgomery Multiplication (MM) is defined in pseudo code below. It interleaves the multiplication and reduction steps, thereby preventing the partial sum from creating an overflow condition [11].

INPUT: integers $m = (m_{n-1} \ldots m_1m_0)_b$, $x = (x_{n-1} \ldots x_1x_0)_b$, $y = (y_{n-1} \ldots y_1y_0)_b$ with $0 \leq x; y < m$, $R = b^n$ with gcd(m, b) = 1, and $m' = -m^{-1} \bmod b$.

1. $H \leftarrow 0$. (Notation: $H = (h_nh_{n-1} \ldots h_1h_0)_b$.)
2. For i from 0 to ($n-1$) do the following:
   2.1 $u_i \leftarrow (h_0 + x_iy_0)m' \bmod b$.
   2.2 $H \leftarrow (H + x_iy + u_im) / b$.
3. If $H \geq m$ then $H \leftarrow H - m$.
4. Return(H).

OUTPUT: $xyR^{-1} \bmod m$.

MM is not the perfect solution for all applications. It requires substantial work to set up the system and convert the result back to the standard integer domain. As a result, traditional algorithms are faster by a factor of two in the computation of one modular multiplication. However, a substantial speedup is realized over classical systems when one number is multiplied by itself under mod N a predefined number of times.

Montgomery Exponentiation (ME) is a special case of Montgomery Multiplication. It utilizes the multiplication function to accomplish its computation. Exponentiation is accomplished by a sequence of squaring operations followed by a multiplication. Assuming l-bit inputs to the system. The loop body of the algorithm requires $2l(l+1)$ single-precision multiplications and no explicit division. Traditional algorithms would require the same number of multiplications, but also l divisions. ME eliminates on the order of 1000 divisions in real systems. This improvement greatly overshadows the small cost of pre- and post-computation [12].

INPUT: $m = (m_{l-1} \ldots m_0)_b$, $R = b^l$, $m' = -m^{-1} \bmod b$, $e = (e_t \ldots e_0)_2$ with $e_t = 1$, and an integer x, $1 \leq x < m$, $s = R^2 \bmod m$, $A = R \bmod m$

1. $x' \leftarrow$ MontMul(x, s).
2. For i from t down to 0 do the following:
   2.1 $A \leftarrow$MontMul(A, A).
   2.2 If $e_i = 1$ then $A \leftarrow$MontMul(A, x').
3. $A \leftarrow$MontMul(A, 1).
4. Return(A).

OUTPUT: $x^e \bmod m$.

The pseudo code shows that all the steps of the algorithms can be implemented in logic circuits very easily. The main loop of MM requires only AND & OR gates, as well as adders and shifters. A few comparators and control logic are all this is necessary to realize the algorithm in hardware.

5. Circuit Design

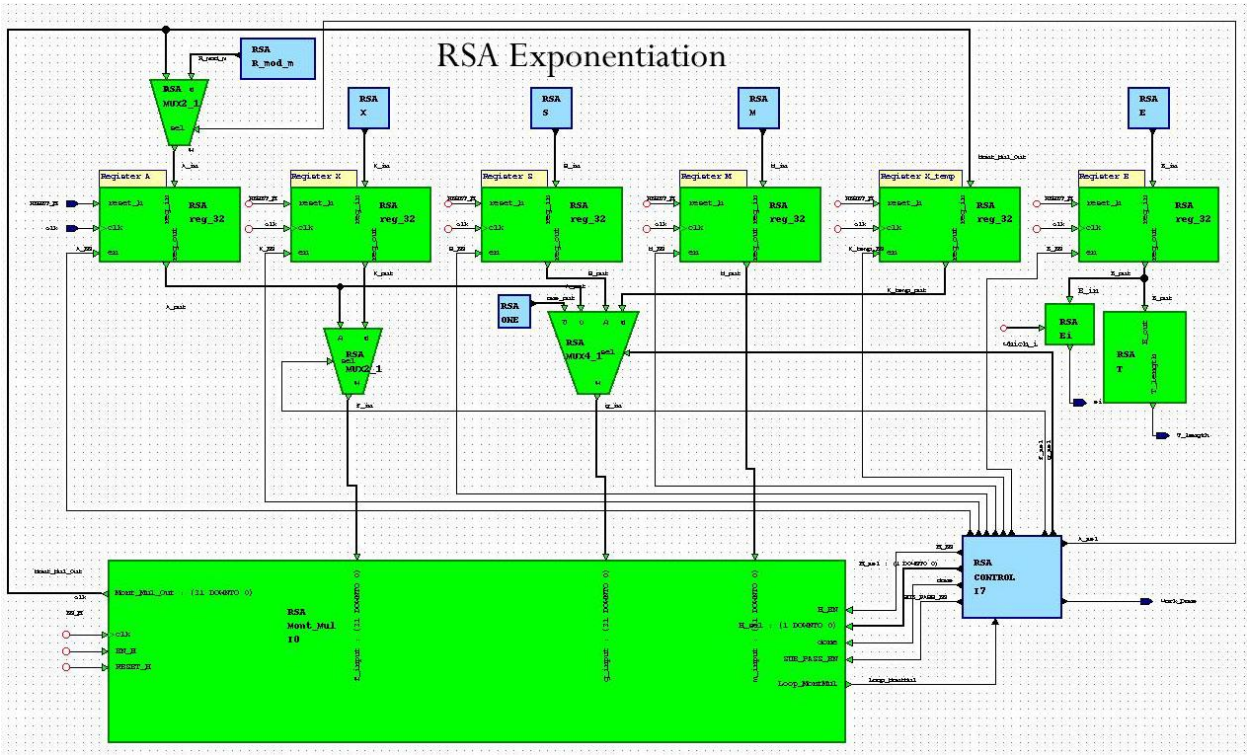

Figure 1. RSA High Level Circuit Design

The multiplier circuit contains one register, *H*, which stores the result of the multiplication and passes this value to the exponentiation circuit. A multiplexor selects between the two possible inputs to *H*, the result of step 2.2, or the final result after step three. Bit rippers parse the 32-bit words entering the multiplier. The least significant bit (LSB) is always required from register H and the multiplier. The bit selected from the multiplicand is dependent on the loop iteration.

Step 2.1 requires that the LSB of *H* is added to the logical AND of the LSB of the multiplier and current iteration bit of the multiplicand. The result of this add can be a two-bit number, so a two-bit adder is used. $M' = -m^{-1} \bmod 2 = 1$ in all cases in this system, so no additional work must be done to compute it. To finish the computation of u_i, the previously computed sum must undergo the mod 2 operation. A serial shifter shifts the LSB out of the sum and routes u_i to a parallel array of 32 AND gates. This component simultaneously computes the result of $u_i m$. Another instance of this component ANDs the current iteration bit of the multiplicand with all the bits of the multiplier. The one-bit outputs of the ANDER components are then zero-extended and add together. Their sum is then added to the present value of register *H*.

After thirty-two iterations of step two, the contents of register *H* and *M* are passed into a 32-bit comparator. If $H \geq M$, the comparator output asserts, and the SUB_PASS component will subtract M from H. However, if $H < M$, the SUB_PASS component will simply pass H through to the multiplier circuit output port.

The higher level circuit responsible for exponentiation uses the multiplier circuit for four different purposes. Initially, it is used to compute the temporary value *X*'. Next it is used to either square the contents of register *A* or multiply the contents of register *A* and register *X*'. Finally, the Montgomery Multiplier is used to convert back to the integer domain in step three.

Register *A* is initialized to *R* mod *m*, but is overwritten during each iteration of step two. The loop represented by step 2 is executed *t*+1 times where *t* is the bit position in register *E* where the most significant one is found. A combinational circuit determines this value and the finite state machine providing system control uses this value. At each instance of step 2.2, another component rips the current iteration bit from *E* and checks if it is one. If so, it will execute step 2.2.

Two multiplexors select from the possible inputs to the Montgomery multiplier. The multiplicand accepts either register *A* or *X*. The multiplier accepts either register *A*, *S*, or *X*'.

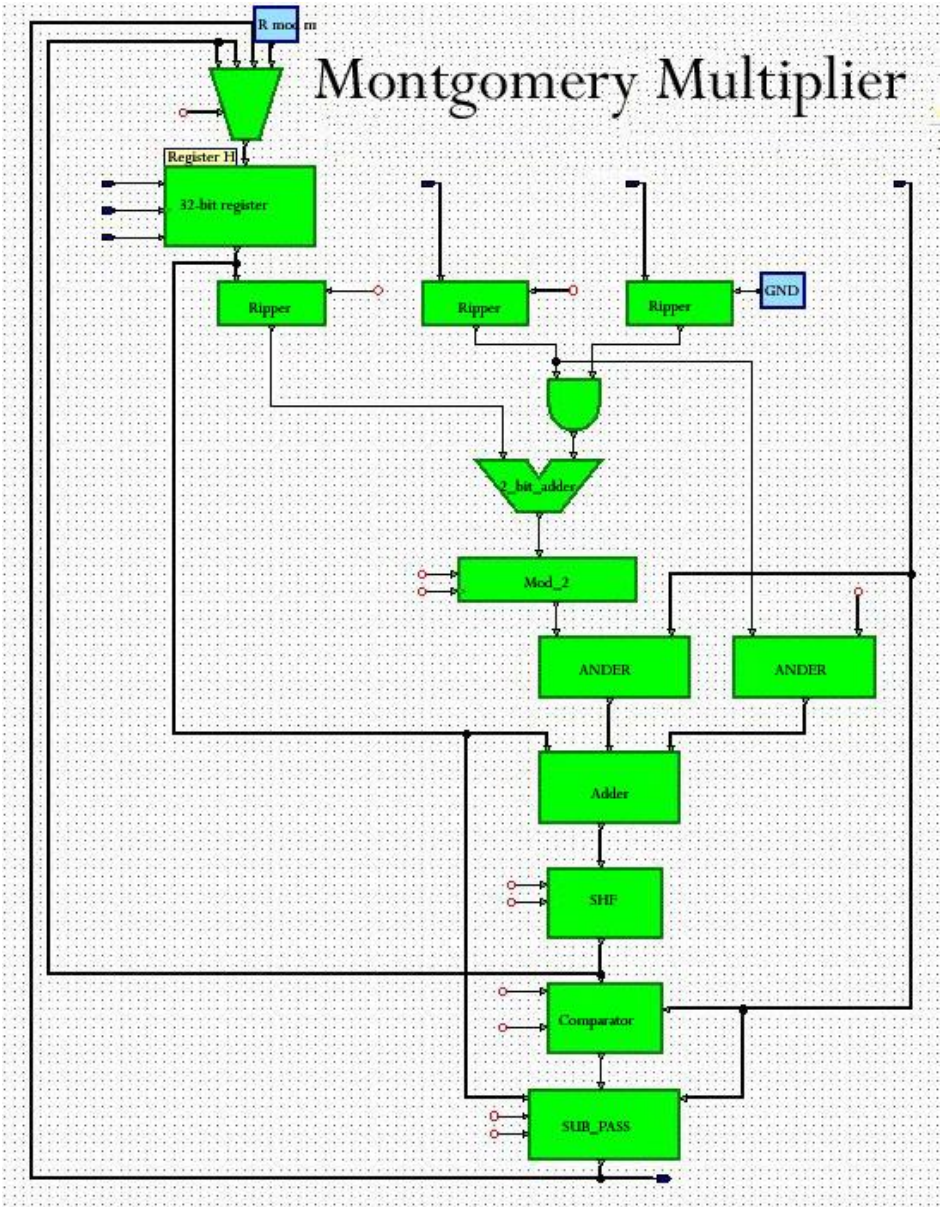

Figure 2. Montgomery Multiplier Circuit

6. First-Order Performance Analysis and Practical Considerations for Deployed Systems

Assuming a 1ns delay for each logic gate, the Montgomery Multiplier circuit takes 592ns to execute. In a worst case scenario where register E is all ones, the total execution time for the system is 2368ns. This means the coprocessor can run at 422kHz. 32-bit words contain four ASCII characters, so 1.69 million characters can be processed in one second.

The longest operation in Montgomery Multiplication is addition. The logic and shift operations are effectively instantaneous. The loop represented by step 2 must be executed n times where n is the number of bits that can be processed. Within each iteration of the loop, step 2.2 requires an n-bit addition. Carry look-ahead adders must be exploited to perform the addition, because ripple-carry adders are infeasible with such large data words. The total system execution time for exponentiation scales according to 2 T_{mm} [multiplication time] + (L_e [length of E] $\cdot$ 2 $\cdot$ T_{mm}). The system built for the purposes of this research utilized 32-bit data words. State of the art commercial systems must require more rigorous constraints on the input variables. Per Menezes et al., we suggest the following:

- **Recommended size of modulus *N***: N should be 2048-bits or higher in order to beat the quadratic sieve and number field sieve factoring algorithms.

- **Recommended criteria for selection of *p* & *q***: Both primes should be about 1024-bits and close to the same length in order to beat the elliptic curve factoring algorithm. Also, $p - q$ should be large. If the difference is very small, then $p \approx q$ and therefore $p \approx \text{sqrt}(N)$. Finally, p and q should be strong primes:
    - $p - 1$ has a large prime factor, r
    - $p + 1$ has a large prime factor
    - $r - 1$ has a large prime factor

  in order to beat Pollard's $p-1$ factoring algorithm.

- **Recommends size of exponent *e*:** The exponent does not need to be a large number. In fact, e =3 or 65537 are common exponents. However, ($p - 1$) nor ($q - 1$) should not be divisible by e [13].

7. Conclusion and Future Work

In this brief paper we have described a straightforward architecture for a logic circuit implementation of the RSA cryptosystem. The architecture and implementation suggestions are ideal for educational purposes, e.g., advanced undergraduate or beginning graduate students. An effort is made to communicate salient, fundamental topics associated with public-key cryptography and digital system design in a clear and straightforward way.

Future work could entail developing a high throughput pipelined implementation for use in a network interface card, whether wired or wireless. Through effective load balancing, the total execution time to process a long string of input words would decrease by a constant factor likely in the range of two to five. There is also work to be done interfacing the FPGA implementation of the circuit with a PC. The most appropriate bus must be determined and a protocol developed to maximize throughput. Finally, a program should be written to interface with email clients, e.g., Microsoft Outlook, to support seamless RSA encryption in email.

Acknowledgments

The author is very grateful for the guidance and insights of Dr. Julian Palmore and Dr. Sanjay Patel during this research.

8. References

[1] John Fry and Martin Langhammer, "FPGAs Lower Costs for RSA Cryptography," *Electronic Engineering Times*, September 26, 2003, accessed April 18, 2019, https://www.eetimes.com/document.asp?doc_id=1271927.

[2] Ronald L. Rivest, Adi Shamir, and Leonard Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM* 21, no. 2 (1978): 120–126.

[3] D. C. Hankerson et al., *Coding Theory and Cryptography*, 2nd ed. (New York: Marcel Dekker, 2000), 284.

[4] See Neal Koblitz, *Algebraic Aspects of Cryptography*, Algorithms and Computation in Mathematics 3 (New York: Springer, 2004), 1–16. For a detailed overview of salient number-theoretic issues involved in RSA cryptography, see Valery V. Yaschenko, *Cryptography: An Introduction*, Student Mathematical Library 18 (Providence, RI: American Mathematical Society, 2002), chap. 4.

[5] Roland Schmitz, "Public Key Cryptography: A Dynamical Systems Perspective," in *Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies* (Cap Esterel, France, 2008), 211.

[6] Alfred J. Menezes et al., *Handbook of Applied Cryptography* (Boca Raton, FL: CRC Press, 2001), 286–287.

[7] The early modular multiplication and exponentiation algorithms based on the Montgomery reduction technique required that the modulus be an odd integer. Koç work shows that with the help of the Chinese Remainder Theorem, the Montgomery reduction algorithm can be used to efficiently execute these modular arithmetic operations with respect to an even modulus. See Ç. K. Koç, "Montgomery Reduction with Even Modulus," *IEE Proceedings - Computers and Digital Techniques* 141, no. 5 (September 1994): 314–316.

[8] Peter L. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation* 44, no. 170 (April 1985): 519–521.

[9] The efficient computation of the modular exponentiations is very important for public-key cryptosystems. For example, see Chia-Long Wu, Der-Chyuan Lou, and Te-Jen Chang, "An Efficient Montgomery Exponentiation Algorithm for Cryptographic Applications," *Informatica* 16, no. 3 (2005): 449–468 as well as Corinne McIvor, Maire McLoone, and John V. McCanny, "Modified Montgomery Modular Multiplication and RSA Exponentiation Techniques," *IEE Proceedings - Computers and Digital Techniques* 151, no. 6 (November 2004): 402–408.

[10] Michael Welschenbach, *Cryptography in C and C++*, 2nd ed. (New York: Apress, 2005), 106–109.

[11] Menezes et al., *Handbook of Applied Cryptography*, 602.

[12] Ibid., 619–620.

[13] Ibid., 290–291.