ISSN: 2277-9655
Impact Factor: 1.852

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Reducing Runtime of RSA Processors Based On High-Radix Montgomery Multipliers

Prabha N.
nprabha1234@gmail.com

Abstract

Depends on various requirements the paper presents & optimized Rivest–Shamir–Adleman (RSA) processor which satisfies circuit area, operating time. we also introduces 3 multiplier based data path using different intermediate data forms: 1) single form, 2) semicarry-save form, and 3) carry-save form, and combined them witha wide variety of arithmetic components. A total of 242 datapaths for 1024-bitRSA processors were obtained for each radix. We can reduce the RSA runtime up to 0.24ms. As a result, the fastest design can perform the RSA operation in less than 1.0 ms.

Keywords: ASIC implementation, high-radix Montgomery multiplication, RSA.

Introduction

The mathematical details of the algorithm used in obtaining the public and private keys are available at the RSA Web site. Briefly, the algorithm involves multiplying two large prime numbers (a prime number is a number divisible only by that number and 1) and through additional operations deriving a set of two numbers that constitutes the public key and another set that is the private key. Once the keys have been developed, the original prime numbers are no longer important and can be discarded. Both the public and the private keys are needed for encryption /decryption but only the owner of a private key ever needs to know it. Using the RSA system, the private key never needs to be sent across the Internet. The private key is used to decrypt text that has been encrypted with the public key. Thus, if I send you a message, I can find out your public key (but not your private key) from a central administrator and encrypt a message to you using your public key. When you receive it, you decrypt it with your private key. In addition to encrypting messages (which ensures privacy), you can authenticate yourself to me (so I know that it is really you who sent the message) by using your private key to encrypt a digital certificate. When I receive it, I can use your public key to decrypt it. A table might help us remember this.

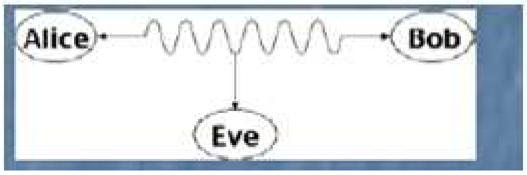


Fig 1.0 RSA Block Diagram

RSA is an Internet encryption and authentication system that uses an algorithm developed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. The RSA algorithm is the most commonly used encryption and authentication algorithm and is included as part of the Web browsers from Microsoft and Netscape. It's also part of Lotus Notes, Intuit's Quicken, and many other products. The encryption system is owned by RSA Security. The company licenses the algorithm technologies and also sells development kits. The technologies are part of existing or proposed Web, Internet, and computing standards.A single application of the Montgomery algorithm (henceforth referred to as a "Montgomery step") is faster than a "naive" modular multiplication:

$$c = a \times b \pmod{n}.$$

Because numbers have to be converted to and from a particular form suitable for performing the Montgomery step, a single modular multiplication performed using a Montgomery step is actually slightly less efficient than a "naive" one. However, modular exponentiation can be implemented as a sequence of Montgomery steps, with conversion only required once at the start and once at the end of the sequence. In this case the greater speed of the Montgomery steps far outweighs the need for the extra conversions. Working with *n*-digit numbers to base *d*, a Montgomery step calculates $a \times b \div d^n \pmod{r}$. The base *d* is typically 2 for microelectronic applications or 2^{32} or 2^{64} for software applications. For the purpose of exposition, we shall illustrate with $d = 10$ and $n = 4$.

To calculate 0472 × a ÷ 10000:

1. Zero the accumulator.

2. Starting from the last digit; add 2*a* to the accumulator.
3. Shift the accumulator one place to the right (thus dividing by 10).
4. Add 7*a* to the accumulator.
5. Shift the accumulator one place to the right.
6. Add 4*a* to the accumulator.
7. Shift the accumulator one place to the right.
8. Add 0*a* to the accumulator.
9. Shift the accumulator one place to the right.

It is easy to see that the result is 0.0472 × *a*, as required. To turn this into a modular operation with a modulus *r*, add, immediately before each shift, whatever multiple of *r* is needed to make the value in the accumulator a multiple of 10. The result will be that the final value in the accumulator will be an integer (since only multiples of 10 have ever been divided by 10) and equivalent (modulo *r*) to 472 × *a* ÷ 10000. Finding the appropriate multiple of *r* is a simple operation of single-digit arithmetic. When working to base 2, it is trivial to calculate: if the value in the accumulator is even, the multiple is 0 (nothing needs to be added); if the value in the accumulator is odd, the multiple is 1 (*r* needs to be added). The Montgomery step is faster than the methods of "naive" modular arithmetic because the decision as to what multiple of *r* to add is taken purely on the basis of the least significant digit of the accumulator. This allows the use of carry-save adders, which are much faster than the conventional kind but are not immediately able to give accurate values for the more significant digits of the result. Working with *n*-digit numbers to base *d*, a Montgomery step calculates $a \times b \div d^n \pmod{r}$. The base *d* is typically 2 for microelectronic applications or 2^{32} or 2^{64} for software applications. For the purpose of exposition, we shall illustrate with *d* = 10 and *n* = 4.To turn this into a modular operation with a modulus *r*, add, immediately before each shift, whatever multiple of *r* is needed to make the value in the accumulator a multiple of 10. The result will be that the final value in the accumulator will be an integer (since only multiples of 10 have ever been divided by 10) and equivalent (modulo *r*) to 472 × *a* ÷ 10000. Finding the appropriate multiple of *r* is a simple operation of single-digit arithmetic. When working to base 2, it is trivial to calculate: if the value in the accumulator is even, the multiple is 0 (nothing needs to be added); if the value in the accumulator is odd, the multiple is 1 (*r* needs to be added). The Montgomery step is faster than the methods of "naive" modular arithmetic because the decision as to what multiple of *r* to add is taken purely on the basis of the least significant digit of the accumulator. This allows the use of carry-save adders, which are much faster than the conventional kind but are not immediately able to give accurate values for the more significant digits of the result.The encryption/decryption process usually requires a large amount ofarithmetic operations with very large operands. In particular,Rivest–Shamir–Adleman (RSA) cryptosystem [1] usually performsmodular exponentiation using operands longer than 1000bits. Modular exponentiation is performed by repeating modularmultiplication and squaring operations, and thus optimization ofmodular multiplication is essential in order to achieve high-performance RSA cryptosystem designs. The Montgomery multiplication algorithm [2], which does not require trial division, is widely used for practical hardware and software implementations because of its high speed capability.Many omputational techniques and hardware architectureshave been proposed for Montgomery multiplication [3]–[11]. Among them, the radix-2 algorithms proposed in [3] and [4] areprimarily implemented with long -bit adders to scan the -bitoperand bit-by-bit in a straightforward manner. Hardware architectureshave large fan-out signals and large wire delays for longoperands. These drawbacks can be reduced by systolic array architectures[6], [7] with multiple operation units. However, these architectures are usually tailored for fixed-precision computations and cannot respond flexibly to changes in operand size. To deal with variable-length data, a radix-2 architecture was proposed [8]–[10] in which a -bit operand is divided into -bit word blocks, and -bit addition is performed by repeating –bit addition times. These radix-2 architectures are quite simple, but have difficulty in improving the performances of circuit area and efficiency. A high-radix architecture using a 64-bit 64-bitmultiplier was proposed in [11] to achieve higher circuit efficiency.

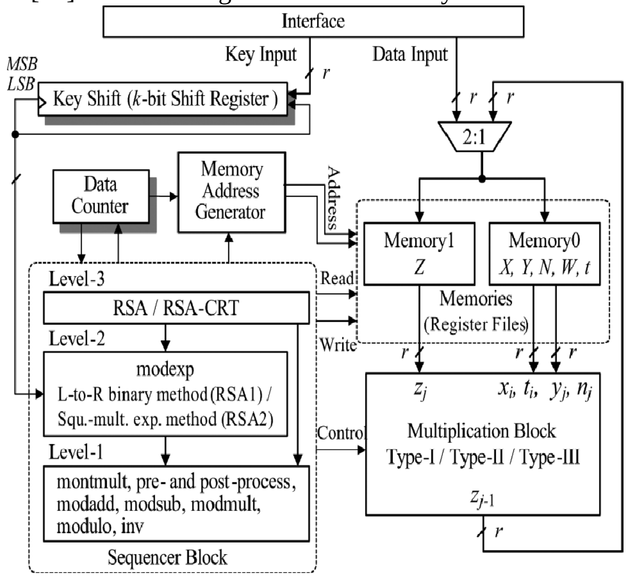

Interface
Key Input
Data Input
MSB
LSB
Key Shift (k-bit Shift Register)
r
2:1
Data Counter
Memory Address Generator
Address
Memory1 Z
Memory0 X, Y, N, W, t
Memories (Register Files)
Read
Write
Control
Level-3
RSA / RSA-CRT
Level-2
modexp
L-to-R binary method (RSA1) / Squ.-mult. exp. method (RSA2)
Level-1
montmult, pre- and post-process, modadd, modsub, modmult, modulo, inv
Sequencer Block
z_j
x_i, t_i, y_j, n_j
Multiplication Block Type-I / Type-II / Type-III
z_j-1

Fig 1.1 RSA Processor architecture

The performance of such a multiplier-based architecture depends heavily on the datapath structure, and varies with the structure of the arithmetic components, but previous papers have focused on designing their own architectures. These architectures are optimized for some design parameters, such assize and speed, while the most suitable design point in practical use varies depending on the application and the user quirements.Therefore, in order to provide the best design which satisfies these requirements, a systematic study considering the entire process of design from the datapath architecture level to the arithmetic-component level is indispensable from a practicalstandpoint.On the other hand, cryptanalysis based on side-channel information is a major concern for hardware designers. When a cryptographic module performs encryption or decryption, secret parameters related to the intermediate data being processed can leak as side-channel information in the form of power dissipation, electromagnetic radiation, or operating time. Among them,two of the best known attacks are simple power analysis (SPA)and differential power analysis (DPA).Many important cryptosystems such as RSA and DSA are based on arithmetic operations, such as multiplications, modulo a large number. The classical method of calculating a modular product involves first multiplying the numbers as if they were and then taking the modulo of the result. However, modular reduction is very expensive computationally—equivalent to dividing two numbers. The situation is even worse when the algorithm requires modular exponentiation. However, the performance of RSA processors with such countermeasures has not been fully evaluated in previous work. This paper proposes a systematic design of RSA processors combining various datapath architectures and exponentiation algorithms (i.e., sequences) for performance and resistance against side-channel attacks, respectively. This systematic approach is divided into four design stages: 1) algorithm design; 2)radix design; 3) architecture design; and 4) arithmetic-component design. We first select a modular exponentiation algorithm considering the tradeoff between the RSA computation time and tamper resistance. We then select the radix to determine the basic characteristics of the processor, such as circuit area and operation frequency (i.e., critical path). Finally, we adopt the datapath architecture and the arithmetic components to optimize the circuit performance.

High-Radix Montgomery Multiplier

A. Montgomery multiplication algorithm

Given two large integers X and Y, the Montgomery multiplication algorithm performs the following operation:

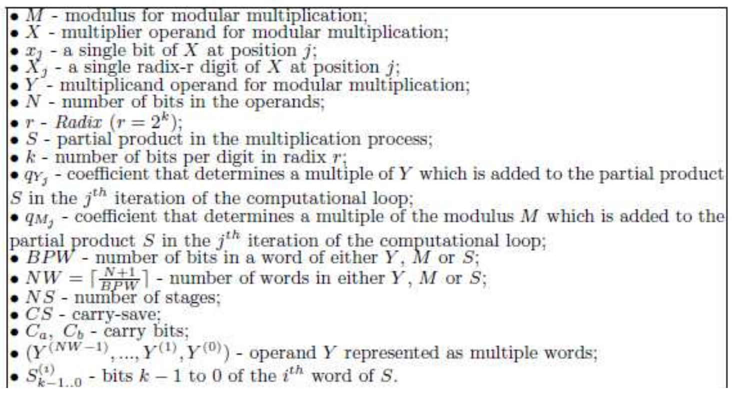
$$Z = XYR^{-1} \mod N, \quad (1)$$

where $R = 2k$ and the modulus N is an integer in the range $2n¡1 < N < 2n$ such that gcd(R,N) = 1.
For cryptographic applications, N is usually a prime number or a product of primes, and thus satisfies the condition easily. In addition, the k-bit integers X, Y, R, and N satisfy the following condition: $0 \cdot X,Y < N < 2k = R$. (2)

ALGORITHM 1 shows the original Montgomery multiplication algorithm [1], which replaces a modular division by-N with a k-bit right shift operation. Equation (1) can This paper describes an algorithm and architecture based on an extension of a scalable radix-2 architecture proposed in a previous work. The algorithm is proven to be correct and the hardware design is discussed in detail. Experimental results are shown to compare a radix-8 implementation with a radix-2 design. The scalable Montgomery multiplier is adjustable to constrained areas yet being able to work on any given precision of the operands. Similar to some systolic implementations, this design avoid the high load on signals that broadcast to several components, making the delay independent of operand's precision.

b) High-radix Word-based Montgomery Algorithm

The notation used throughout this text is shown in Table 1. Figure 1 shows the Multiple-word High-Radix (2k) Montgomery Multiplication algorithm (MWR2kMM), a generalization of the MM algorithm presented in.A full-precision High-Radix Montgomery algorithm has been presented BNand proven to be correct in [8]. To prove correctness of the algorithm in Figure 1 we show that it is equivalent to the one presented in [8].



- M - modulus for modular multiplication;
- X - multiplier operand for modular multiplication;
- x_j - a single bit of X at position j;
- X_j - a single radix-r digit of X at position j;
- Y - multiplicand operand for modular multiplication;
- N - number of bits in the operands;
- r - *Radix* ($r = 2^k$);
- S - partial product in the multiplication process;
- k - number of bits per digit in radix r;
- q_{Y_j} - coefficient that determines a multiple of Y which is added to the partial product S in the j^{th} iteration of the computational loop;
- q_{M_j} - coefficient that determines a multiple of the modulus M which is added to the partial product S in the j^{th} iteration of the computational loop;
- BPW - number of bits in a word of either Y, M or S;
- $NW = \lceil \frac{N+1}{BPW} \rceil$ - number of words in either Y, M or S;
- NS - number of stages;
- CS - carry-save;
- C_a, C_b - carry bits;
- $(Y^{(NW-1)}, ..., Y^{(1)}, Y^{(0)})$ - operand Y represented as multiple words;
- $S^{(i)}_{k-1..0}$ - bits $k-1$ to 0 of the i^{th} word of S.

c) High-radix Montgomery Multiplier - System level

For high-precision computation it is beneficial to divide the multiplicand Y , the modulus M and the result S into words [18]. The approach keeps the gates and the wire delays inside reasonable boundaries. With operands' precision of thousands of bits, a conventional design to multiply all the bits at once would have a high number of pins, increased fan-in for the gates, high gate loads, and gate outputs driving long wires. The multiplications (qY ∗ Y)(∗) and (qM ∗ M)(∗) shown in the MWR2Kmm algorithm can be implemented by multiplexers (MUXes) and adders. The shifting operation in Step 10 is simple in hardware. Additions can be done using Carry- Save Adders (CSA), and keeping S in redundant form. With this approach the carries generated during addition are not propagated but rather stored in a separate bit-vector along with a bit-vector for the sum bits. The most complex operations of finding the coefficients qY and qM (steps 3 and 5) can be executed by table look-up. qY is pre-computed before the computational cycle begins since it depends only on the least significant k bits of X. This observation leaves the computation of qM in the most critical part of the algorithm as it is also pointed out by other authors. The architecture of a Montgomery multiplier implementing the MWR2kMMalgorithm is shown in Fig. 3. There are two main functional blocks: Kernel and IO. Only the data path is shown. The Kernel's data path is where the computation takes place according to the algorithm. A control block (not shown) supplies the signals to synchronize the system.

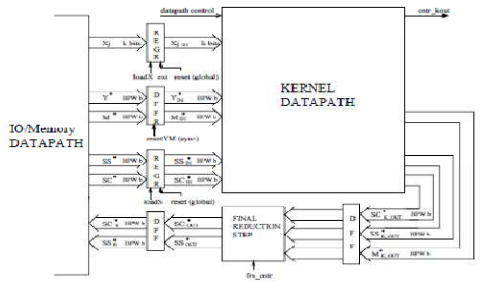
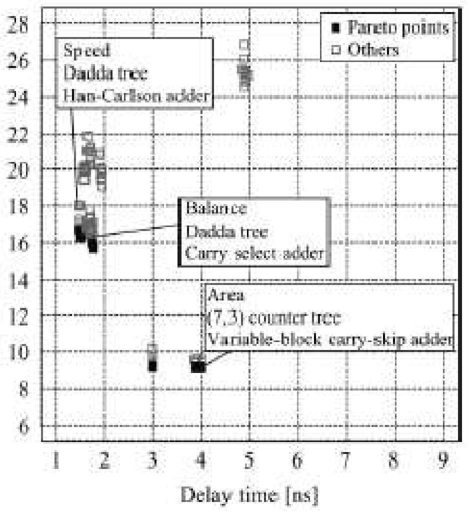
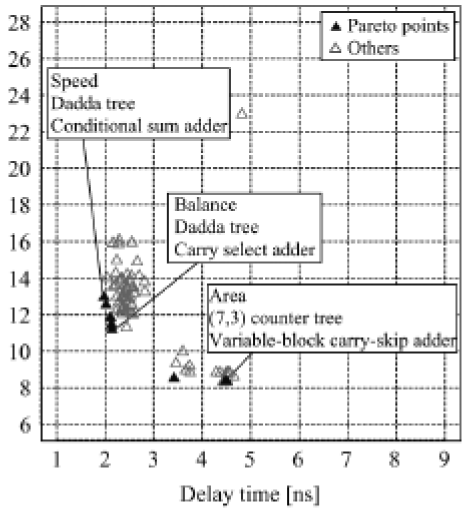


Fig 2.0 System level diagram of modulator multiplier

Conclusion

Modular exponentiation architecture was derived that combines a high radix version of Montgomery's algorithm with novel systolic array architecture. The design was optimized for modern FPGAs. For an optimal speed area trade–off a radix of 16 was chosen. We showed that it is possible to implement 1024–bit modular exponentiation on a single commercially available FPGA. 1024–bit RSA is performed in 3.1 ms using a clock rate of 45.6 MHz and an area of 6826 CLB's on a Xilinx XC40250XV, speed grade -09. These performances are better than all previously reported implementations presented in technical literature. As a result A total of 242 datapaths for 1024-bitRSA processors were obtained for each radix. We can reduce the RSA runtime up to 0.24ms. As a result, the fastest design can perform the RSA operation in less than 1.0 ms.





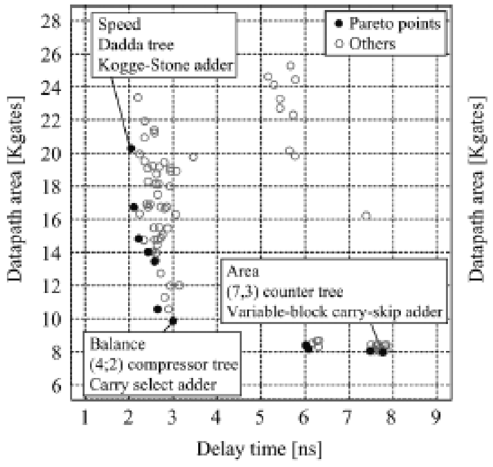


TABLE IV
PPA AND CPA ALGORITHMS

PPA algorithms	CPA algorithms
Array	Ripple carry adder
Wallace tree	Carry look-ahead adder
Balanced delay tree	Ripple-block CLA
Overturned-stairs tree	Block CLA
Dadda tree	Kogge-Stone adder
(4;2) compressor tree	Brent-Kung adder
(7,3) counter tree	Han-Carlson adder
	Ladner-Fischer adder
	Conditional sum adder
	Carry select adder
	Fixed-block-size carry-skip adder
	Variable-block-size carry-skip adder

References

[1] P. Montgomery, “Modular multiplication without trial division,” Mathematics of Computation, vol. 44, pp. 519–21, April 1985.

[2] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, “Programmable active memories: Reconfigurable systems come of age,” IEEE Transactions on VLSI Systems, vol. 4, pp. 56–69, Mar 1996.

[3] M. Shand and J. Vuillemin, “Fast implementations of RSA cryptography,” in Proceedings 11th IEEE Symposium on Computer Arithmetic, pp. 252–259, 1993.12

[4] S. E. Eldridge and C. D. Walter, “Hardware implementation of Montgomery’s modular multiplication algorithm,” IEEE Transactions on Computers, vol. 42, pp. 693–699, July 1993.

[5] H.Orup, “Simplifying quotient determination in high-radix modular multiplication,” in Proceedings 12th Symposium on Computer Arithmetic, pp. 193–9, 1995.

[6] P. Kornerup, “A systolic, linear-array multiplier for a class of right-shift algorithms,” IEEE Transactions on Computers, vol. 43, pp. 892–8, August 1994.

[7] C. K. Koc, T. Acar, and B. Kaliski, “Analyzing and comparing Montgomery multiplication algorithms,” IEEE Micro, vol. 16, pp. 26–33, June 1996.

[8] T. Blum and C. Paar, “Montgomery modular exponentiation on reconfigurable hardware,” in Proceedings 14th Symposium on Computer Arithmetic, pp. 70–7, 1999.

[9] Xilinx, Inc., San Jose, CA, The Programmable Logic Data Book, 1996.

[10] T. Blum, “Modular exponentiation on reconfigurable hardware,” Master’s thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1999.

[11] P. Alfke, “Xilinx M1 Timing Parameters.” Electronic Mail Personal Correspondence, December 1999.
[12] R. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public key cryptosystems,” Communications of the ACM, vol. 21, pp. 120–6, Feb. 1978.
[13] D. Knuth, The Art of Computer Programming. Volume 2: Seminumerical Algorithms. Reading, Massachusetts: Addison-Wesley, 2nd ed., 1981.
[14] J. Quisquater and C. Couvreur, “Fast decipherment algorithm for RSA public–key cryptosystem,” Electronics Letters, vol. 18, pp. 905–7, October 1982.
[15] E. D. Win, S. Mister, B. Preneel, and M. Wiener, “On the performance of signature schemes based on elliptic