An Efficient VLSI Architecture for Rivest-Shamir-Adleman Public-key Cryptosystem

Jen-Shiun Chiang*, Cheng-Chih Chien, Jian-Kao Chen and Hsin-Guo Chou

Department of Electrical Engineering
Tamkang University
Tamsui, Taiwan 251, R.O.C.
E-mail: chiang@ee.tku.edu.tw

Abstract

In this paper, a new efficient VLSI architecture to compute modular exponentiation and modular multiplication for Rivest-Shamir-Adleman (RSA) public-key cryptosystem is proposed. We modify the conventional H-algorithm to find the modular exponentiation. By this modified H-algorithm, the modular multiplication steps for n-bit numbers are reduced by $5n/18$ times. For the modular multiplication a modified L-algorithm (LSB first) is used. In the architecture of the modified modular multiplication the iteration times are only half of Montgomery's algorithm and the H-algorithm. The proposed architecture for the RSA public-key crypto-system has a data rate of 146 k*b*/s for 512-*b* words with a 200-MHz clock rate.

Key Words: Data Security, H-algorithm, L-algorithm, Modular Exponentiation, Modular Multiplication, Montgomery's Algorithm, Public-key Cryptosystem, RSA, VLSI

1. Introduction

In open network and communication systems, the security problems of electronic communication are severe. Traditionally, the common algorithm (common key) is used to improve the security problems. In the common algorithm approach, the transmitter encrypts codes by a secret key; the receiver uses the same (common) key to decrypt the received data. However, a problem is concerned: how to send the secret key between transmitter and receiver. In 1978, Rivest, Shamir, and Adleman (RSA) [1] proposed a public key cryptosystem to improve the communication security problem. In the public key cryptosystem, people use a public key to encrypt the code and transmit the data to the receiver. The receiver uses the private key to decrypt the received data. The public key can be retrieved by anybody, but the private key is held by the receiver only. By this arrangement, people do not need to worry about the key transmission problem, and thus can improve the communication security significantly. The public key cryptosystem uses a mathematical theory to map data in one way direction, $f(X)$: $X \rightarrow Y$, and makes the inverse transform $f^{-1}(Y)$; $Y \rightarrow X$ to be very difficult. The RSA cryptosystem [1] uses Euler and Fermat theorem [15]; its security is based on the decomposition of a number, N, that is the multiplication product of two distinct prime numbers. It is known that a large number is very difficult to decompose. For applications, the RSA cryptosystem cannot only be applied to electronic data communication, but also for electronic signature [1].

The safety of the RSA cryptosystem depends on the length of the key, usually the longer the key the more safety the data. Generally we need at least a 512-bit key. The processing of the key is composed of many modulo multiplication, modulo addition, and modulo exponentiation

*Corresponding author

operations. The RSA cryptosystem is briefly described as follows:

Let p and q be two distinct large random primes and a number N; denote

$$N = p \times q$$

Let us choose a large random number $d > 1$ such that

$$\gcd[(p-1)(q-1), d] = 1,$$

and compute the number e, $1 < e < (p-1)(q-1)$,

$$e \times d \equiv 1 \ (\mathrm{mod}(p-1)(q-1),$$

The numbers N, e, and d are called modulus, encryption, and decryption exponent respectively. The numbers N and e constitute the public encryption key, and p, q, $(p-1)(q-1)$ and d form the secret trapdoor. To encrypt and decrypt, the input text is first encoded to a number and is divided into blocks of suitable size. The blocks are then processed separately as follows:

$$C = M^e \ (\mathrm{mod}\ N) \tag{1}$$
$$M = C^d \ (\mathrm{mod}\ N) \tag{2}$$

C and M are referred to as ciphertext and plaintext blocks, respectively.

Equations (1) and (2) are in modular exponentiation operation and are the most critical operation in RSA. Therefore, how to increase the speed of the modular exponentiation is the main task for the RSA public-key cryptosystem. Basically the modular exponentiation needs modular multiplication. The modular multiplication is accomplished by addition and shift operations. Since the numbers that we deal with are large numbers ($\geq$ 512 bits), it is much different from the traditional number multiplication. In RSA we need modular multiplication and modular exponentiation, therefore, after the multiplication the modulus adjustment has to be operated, and that makes the calculation even more difficult than the calculation of normal numbers. The modulus adjustment is usually accomplished by range comparison. For real time operation, we have to use special methods to calculate the modular multiplication and exponentiation [16]. To reduce the time complexity for comparison, a modular multiplication algorithm based on Montgomery's modular arithmetic [17] was proposed by Eldridge [7]. The Montgomery's algorithm is very suitable for systolic array architecture [4,5,8,18]. Although the systolic array has the characteristic of regularity in the VLSI layout, the hardware cost is high [11,12,13]. Another approaches are H-algorithm (MSB first) [2,5,10] and L-algorithm (LSB first) [10]. These two algorithms are the basic algorithms for modular multiplication. Although the speed is slower than Montgomery's algorithm, the hardware cost is lower than the Montgomery's algorithm. In this paper, we modify the L-algorithm to calculate the modular multiplication [19]. The modified L-algorithm can increase the calculation speed twice faster than the conventional L-algorithm, and the hardware cost is almost the same. Montgomery's algorithm, H-algorithm, and L-algorithm are usually applied to calculate the modular exponentiation. Like the modular multiplication, Montgomery's algorithm for calculating modular exponentiation takes more hardware. Here we use a modified H-algorithm [19] to calculate the modular exponentiation. For an n-bit number, this approach can reduce $5n/18$ iteration times.

In the hardware design of this RSA cryptosystem, adders are massively used. To avoid unnecessary carry propagation, addition can be accomplished by the redundant binary adders [2] or carry save adders [3–6]. However, the adder cell of the redundant binary adder is very complicated, and we use carry save adder for this design. In the shift operation, there are two approaches, left shift (multiply by 2) [2,5,6] and right shift (divide by 2) [3,4]. In this paper, we use the left shift approach. The modular operation can be finished by comparators or by checking the overflow of the adder [5,6]. The former approach needs more hardware and the speed is slower. The latter approach needs less hardware and the speed is faster. Therefore, we use the latter approach to implement the modular operations. In order to reduce the hardware cost, the calculating data (message) of our design are divided into four segments, and each time only one segment is operated. The time to calculate a modular exponentiation is $2.65n^2$ clock cycles. By the proposed approaches we designed a RSA processor; the data rate is about 146 kb/s for 512-b words with 200-MHz clock frequency.

This paper is organized as follows. In Section 2, the modified modular exponentiation algorithm is described. The modified modular multiplication algorithm is described in Section 3. The modular operation is shown in Section 4. The hardware design of the RSA cryptosystem and the simulation results are explained in Section 5. Finally we give the conclusion in Section 6.

2. Modified Modular Exponentiation Algorithm

Repeating squaring and multiplying are the basic arithmetic operations for computing modular exponentiation. To compute $C = M^e \pmod N$, the conventional H-algorithm operates as follows [10]:

```
// The H-algorithm (MSB first)
//    M^e (mod N);
      P_0 = 1;
      for (i = n – 1; i > = 0; i – -){
         M_{n–i} = P^2_{n–i–1} (mod N)
         if (e_i == 1)
         P_{n–i} = M_{n–i} × M (mod N);
         else P_{n–i} = M_{n–i}  }
```

where $e = [e_{n-1}, e_{n-2}, \ldots e_1, e_0]_2$ is the encryption key, and P_i is the partial product. In the modular operation, '1' needs two iteration steps in e[]. In the worst case, we need $2n$ steps to compute the exponentiation. In order to reduce the iteration times, we partition the encryption key e[] into several segments, and each segment consists of four bits; $e[i]$ denotes the *ith* segment of e[]. Observing the bit patterns of the 4-bit segment, we find some rule to reduce the iteration times. For example, when $e[i]$ = 0000, the computation of $M^e \pmod N$ in the H-algorithm needs squaring four times of M, and $e[i]$ = 0001, the computation of $M^e \pmod N$ in the H-algorithm needs squaring three times and the 1 may be combined with next segment. Generally there need five iteration times at most in each segment. Whereas $e[i]$ = 0111, the operation needs seven iteration times with the traditional H-algorithm. By bit patterns of this 4-bit segment, the computation sequences of $C = M^e \pmod N$ within this 4-bit segment can be summarized in Table 1.

Let us describe the notation and operation of Table 1. Suppose X denotes the partial exponentiation of $M^e \pmod N$ in the H-algorithm of the modular exponentiation. In Table 1, 010 means $X^2 \pmod N$; 001 means $X \times M \pmod N$; 011 means $X \times M^3 \pmod N$; 101 means $X \times M^5 \pmod N$; 111 means $X \times M^7 \pmod N$. In the hardware implementation, we can pre-calculate $M_1 = M \pmod N$, $M_3 = M^3 \pmod N$, $M_5 = M^5 \pmod N$, and $M_7 = M^7 \pmod N$, and store these three n-bit numbers to tables.

Let us take an example to describe the rules. Suppose three 24-bit numbers N, M, and e are given as follows:

N = 6012707 = $5bbf23_{16}$ = $(0101\ 1011\ 1011\ 1111\ 0010\ 0011_2)$;
M = 5234673 = $4fdff1_{16}$ = $(0110\ 1111\ 1101\ 1111\ 1111\ 0001_2)$;
e = 3674911 = $38131f_{16}$ = $(0011\ 1000\ 0001\ 0011\ 0001\ 1111_2)$.

By our modified H-algorithm, e is partitioned into six 4-bit segments, and they are $e[5]$ = 0011, $e[4]$ = 1000, $e[3]$ = 0001, $e[2]$ = 0011, $e[1]$ = 0001, and $e[0]$ = 1111 respectively. $e[5]$ = 0011, therefore initially $P_5 = M^3 \pmod N$. Then we proceed to next segment, $e[4]$. Since $e[4]$ = 1000, from Table 1, we can find the sequences of operation are square, multiply by M, square, square, and square, i.e.,

$$P_4 = [[[[P_5^2 \pmod N]\cdot M \pmod N]^2 \pmod N]^2 \pmod N]^2 \pmod N.$$

Next the procedure proceeds to $e[3]$. Since $e[3]$ = 0001. Table 1 shows that the sequences are square, square, and square respectively. The LSB of $e[3]$ is combined with next segment. Here we find the partial exponentiation as follows:

$$P_3 = [[P_4^2 \pmod N]^2 \pmod N]^2 \pmod N.$$

Since the LSB of $e[3]$ is combined with $e[2]$; the sequences of $e[2]$ are square, multiply by M, square, square, square, square, and multiply by M^3 respectively. The partial exponentiation is as follows:

$$P_2 = [[[[P_3^2 \pmod N]^2 \pmod N]^2 \pmod N]^2 \pmod N) \cdot M^3 \pmod N.$$

Table 1. The encryption key table

0000	010 010 010 010
0001	010 010 010
0010	010 010
0011	010 010 010 010 011
0100	010 010 001 010 010
0101	010 010 010 010 101
0110	010 010 010 011 010
0111	010 010 010 010 111
1000	010 001 010 010 010
1001	010 001 010 010
1010	010 010 010 101 010
1011	010 010 010 101
1100	010 010 011 010 010
1101	010 010 011 010
1110	010 010 010 111 010
1111	010 010 010 111

The next procedure proceeds to $e[1]$. The LSB of $e[1]$ will be combined with $e[0]$, and the rest of the sequences of $e[1]$ are square, square, and square respectively. The partial exponentiation is as follows:

$$P_1 = [[P_2^2 \text{ (mod } N)]^2 \text{ (mod } N)]^2 \text{ (mod } N).$$

Then the next procedure is $e[0]$. The first two MSB's of $e[0]$ are combined with the LSB of $e[1]$, and the sequences are square, square, square, and multiply by M_7. The partial exponentiation is as follows:

$$P_0 = [[P_1^2 \text{ (mod } N)]^2 \text{ (mod } N)]^2 \text{ (mod } N) \cdot M^7 \text{ (mod } N)$$

The sequences of the final procedural are square, square, and multiply by M^3. The final exponentiation is:

$$P_f = [P_0^2 \text{ (mod } N)]^2 \text{ (mod } N) \cdot M^3 \text{ (mod } N)$$

By the above description, the sequences of the exponentiation are:

$$3\ 2\ 1\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 1\ 2\ 2\ 2\ 2\ 3\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 7\ 2\ 2\ 3_7.$$

Where '1' means multiplying *M1* to the partial exponentiation; '2' means square of partial exponentiation; '3' means multiplying *M3* to the partial exponentiation; '7' means multiplying *M7* to the partial exponentiation. The sequences are shown in Table 2, and the results are shown in Table 3.

By the above arrangement, [111] can reduce 2 iteration times, and [011] and [101] can reduce 1 iteration time. In the worst case, the iteration times of the modular multiplication of this approach are $4n/3$, and the multiplication average times are $11n/9$. Compared to the conventional H-algorithm (worst case = $2n$, average = $3n/2$), our approach reduces the multiplication times significantly.

Table 2. Sequences of the partial exponentiation

	e[5]	e[4]	e[3]	e[2]	e[1]	e[0]
P_5	0011	1000	0001	0011	0001	1111_2 3
P_4	0011	1000	0001	0011	0001	1111_2 2 1 2 2 2
P_3	0011	1000	0001	0011	0001	1111_2 2 2 2
P_2	0011	1000	0001	0011	0001	1111_2 2 1 2 2 2 2 3
P_1	0011	1000	0001	0011	0001	1111_2 2 2 2
P_0	0011	1000	0001	0011	0001	1111_2 2 2 2 7
P_f	0011	1000	0001	0011	0001	1111_2 2 2 3

Table 3. Results of $C = M^e \text{ (mod } N)$, with $N = 5bbf23_{16}$, $M = 4fdff1_{16}$, and $e = 38131f_{16}$

M1	$17ccd2_{16}$
M3	$37660c_{16}$
M5	$5d682_{16}$
M7	$2a1d8c_{16}$

3. Modified Modular Multiplication Algorithm

In the modular multiplication, the Montgomery's algorithm [3,4] or the H-algorithm [2,5] is applied widely. However, they have their drawbacks in the proposed architectures [11,12,13]. Here we would like to use a modified L-algorithm (LSB first) to find the modular multiplication. The conventional L-algorithm [10] is described as follows.

```
// L–algorithm (LSB first)
//     A × B (mod N);
       P0 = 0, M0 = A;
       for (i = 0; i < = n – 1; i + +){
          if (bi == 1)
           Pi+1 = Pi + Mi (mod N);
          else
           Pi+1 = Pi;
           Mi+1 = Mi << 1 (mod N); }
```

Where P_i is the partial product, and $B = [b_{n-1}, b_{n-2}, \ldots, b_1, b_0]_2$. The L-algorithm calculates the modular multiplication by checking the multiplier, B, from the LSB bit by bit toward the MSB. By this approach, an n–bit number needs n iteration times. We modify the L-algorithm by scanning two bits a time instead of one bit from LSB toward MSB of the multiplier B. Therefore; the iteration times can be reduced to only half of the traditional L-algorithm (the upper bound is $\lfloor n/2 \rfloor$). The modified L-algorithm is described as follows.

```
// Modified L-algorithm
//     A × B (mod N);
       P0 = 0, M0 = A, s = 0, c = 0;
          for (i = 0; i < = ⌊n/2⌋; i + +){
            s = b2i+1 × 2 + b2i + c, c = ⌊s/4⌋;
       switch(s[1:0]){
          case 3:
            c = c + 1;
            Pi+1 = (Pi + (–Mi)) (mod N);
            Mi+1 = (Mi × 4)(mod N);
          case 2:
            Mi+1 = (Mi × 2) (mod N);
```

Table 4. Clock cycles needed for modular multiplication

Algorithm	Each addition	Each multiplication
Montgomery	2	$3n$ *
H-algorithm	3	$4n$ *
L-algorithm	3	$3n$
Ours	4	$2n$

*Include the addition for next multiplication.

$P_{i+1} = (P_i + M_{i+1}) \pmod N$;
$M_{i+1} = (M_{i+1} \times 2) \pmod N$;
case 1:
$P_{i+1} = (P_i + M_i) \pmod N$;
$M_{i+1} = (M_i \times 4) \pmod N$;
case 0:
$P_{i+1} = P_i$;
$M_{i+1} = (M_i \times 4) \pmod N$;
} }

Table 4 lists the clock cycles that are needed to perform the modular multiplication of n-bit numbers with different algorithms.

4. Modular Operation

In order to increase the speed of the modular operation, carry save adders are used in this RSA processor. In the hardware point of view, if the sum of the addition overflows, the modulus adjustment has to be proceeded. Otherwise, no modulus adjustment needs to be done. There are four cases of sums may cause overflow, and they are 2^n, 2×2^n, 3×2^n and 4×2^n respectively. These four numbers can be precalculated and let us denote these four numbers as k_1, k_2, k_3 and k_4 respectively, and they are:

$$k1 \equiv 2^n \pmod N),$$
$$k2 \equiv 2^{n+1} \pmod N),$$
$$k3 \equiv 3 \times 2^n \pmod N),$$
$$k4 \equiv 2^{n+2} \pmod N).$$

Table 5. Overflow vs. k value

$C(S)_{n+1}$	C_n	S_n	k
0	0	0	0
0	0	1	$k1 \equiv 2^n \pmod N$
0	1	0	$k1 \equiv 2^n \pmod N$
0	1	1	$k2 \equiv 2^{n+1} \pmod N$
1	0	0	$k2 \equiv 2^{n+1} \pmod N$
1	0	1	$k3 \equiv 3 \times 2^n \pmod N$
1	1	0	$k3 \equiv 3 \times 2^n \pmod N$
1	1	1	$k4 \equiv 2^{n+2} \pmod N$

The overflow can be determined by checking $C(S)_{n+1}$, C_n, and S_n of the carry save adder and the values are shown in Table 5. If overflow occurs, k_1, k_2, k_3 or k_4 has to be added to the sum to finish the modulus compensation.

By the overflow checking method, it is very easy to implement modulo operations, and the speed can be increased.

In the modular exponentiation and multiplication, $(-M_i) \pmod N$ is needed in our modified L-algorithm. For simplicity we do not deal with negative numbers in the modular operation. Mathematically the value of $(-M_i) \pmod N$ can be calculated by adding a number of multiples of N to $-M_i$ and make it to be positive. The range of M_i is as follows:

$$0 < M_i = C_{Mi} + S_{Mi} < 2 \times 2^n + 2^n = 3 \times 2^n$$

When there is an overflow, i.e. $c = 1$, we have to add a number $k6$ in the range of $3 \times 2^n < k6 < 4 \times 2^n$, that is multiple of N. Therefore, the range of $(k6 - M_i)$ can be found as follows:

$$\begin{aligned} &2^n \le M_i < 3 \times 2^n \\ &-3 \times 2^n < -M_i \le -2^n \\ &0 < k6 - M_i < 3 \times 2^n \end{aligned} \tag{3}$$

On the other hand, if there is no overflow, i.e. $c = 0$, we can add number k5 in the range of $2 \times 2^n < k5 < 3 \times 2^n$, which is multiple of N to the sum, and the range of $(k5 - M_{i-1})$ can be found as follows:

$$\begin{aligned} &0 < M_i < 2 \times 2^n \\ &-2^{n+1} < -M_i < 0 \\ &0 < k5 - M_i < 3 \times 2^n \end{aligned} \tag{4}$$

Equations (3) and (4) are within the range of the carry save adder, and therefore these two numbers, $(k6 - M_i)$ and $(k5 - M_i)$, can be used in next step and no modular adjustment is needed. Since N is known, the values of $k5$ and $k6$ can be precalculated.

5. Hardware Design

The main operation of the modular multiplier is addition, and carry save adders are commonly used to avoid unnecessary carry propagation delays [3–6]. In this modular multiplier, there are five units. The first unit is "Partial Product Adder" to find the partial product; the second unit is "Summand Generator" to generate the summand for the partial product; the third unit is "Shift Register"; the fourth unit is "Table", and the last unit is "Controller". In order to reduce the hardware cost, the message is partitioned into four segments in the RSA processor. For an n-bit message there are $n/4$ bits in each stage, and we need four clocks to finish each iteration of the modular multiplication. In the modified L-algorithm as mentioned above, two bits are scanned each time, and this 2-bit number can decide 0, A, $2A$ or $-A$ that will be added to the partial product. These four cases are summarized in Table 6. The details of the hardware units are illustrated in the following subsections.

5.1. Summand Generator (SG)

The block diagram of the Summand Generator (SG) is shown in Figure 1. The inputs are $k1 \sim k6$, Carryin, Sumin, and the control signal flag; the outputs are A(Carry), A(Sum), $-A$(Carry), $-A$(Sum), $2A$(Carry), and $2A$ (Sum). These outputs are applied to find the partial product. There are three carry save adders in the SG, and they are pipelined in four stages. According to Figure 1,

CSA2 tries to finish Carry2 + Sum2 = $\lfloor$ (Carry1 + Sum1)/2 $\rfloor$(mod N);
CSA4 tries to finish Carry4 + Sum4 = $\lfloor$ (Carry3 + Sum3)/2 $\rfloor$(mod N);
CSAB tries to finish Carryb + Sumb = $\lfloor$ (−Carry1 + −Sum1)/2 $\rfloor$(mod N).

By this arrangement, the cycle time of the SG is only one delay of the FA.

Table 6. Summand factor

S	Summand
00	0
01	A
10	2A
11	−A

5.2. Partial Product Adder (PPA)

The block diagram of the Partial Product Adder (PPA) is shown in Figure 2. There are three steps to find the partial product, and they are pipelined in four stages. We use three $n/4$ - bit carry save adders in this unit. Carry save adder CSAP1 finishes $P_{i+1} = P_i$ + SCarry; carry save adder CSAP2 finishes $P_{i+1} = P_i$ + SCarry + SSum; carry save adder CSAP3 finishes $P_{i+1} = P_{i+1}$ (mod N).

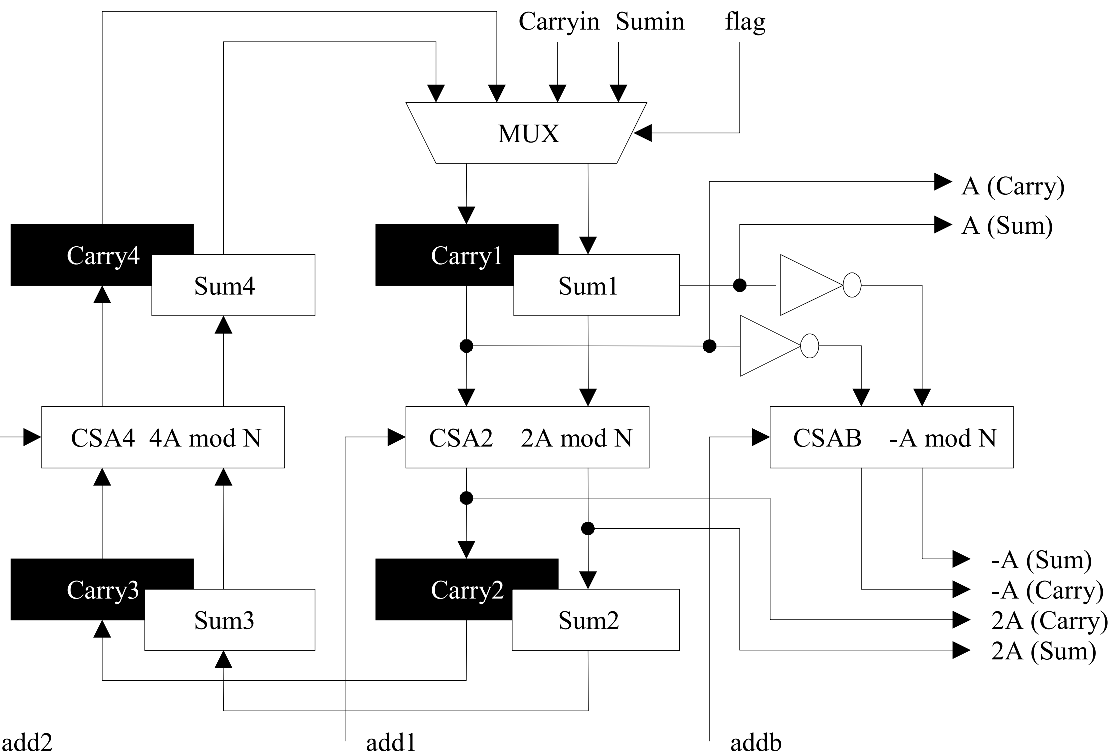

Figure 1. Summand generator.

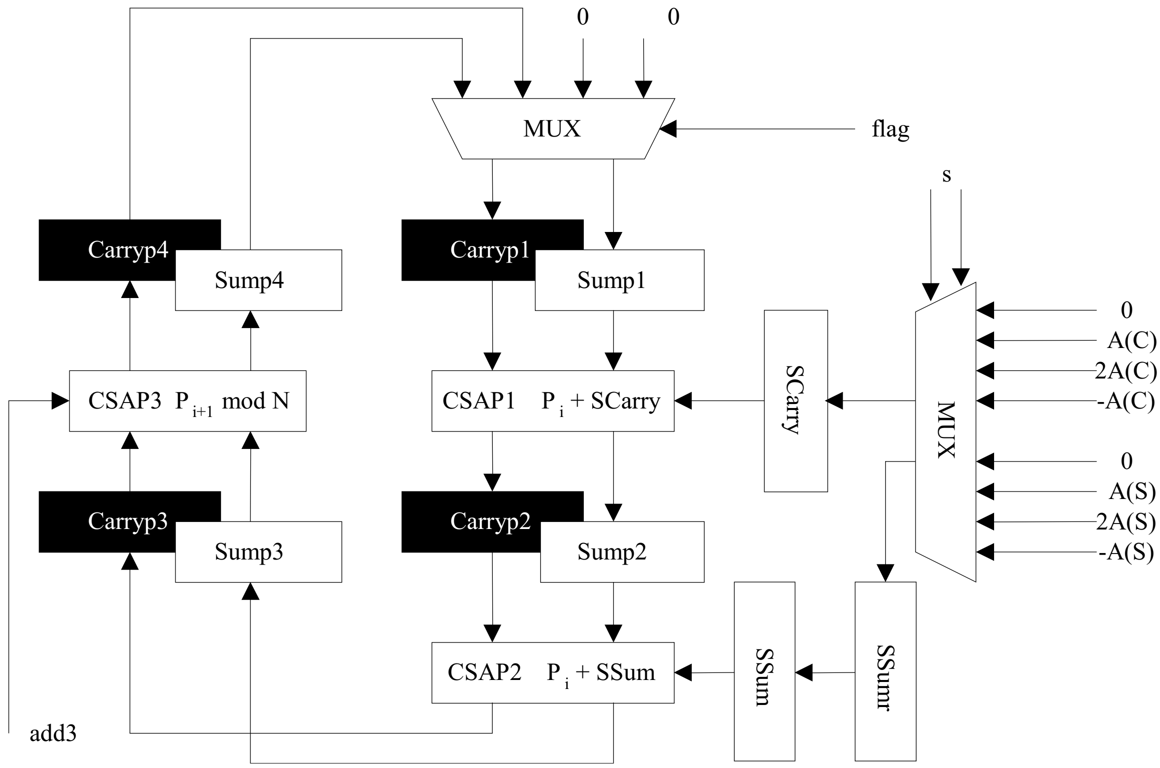

Figure 2. Partial product adder.

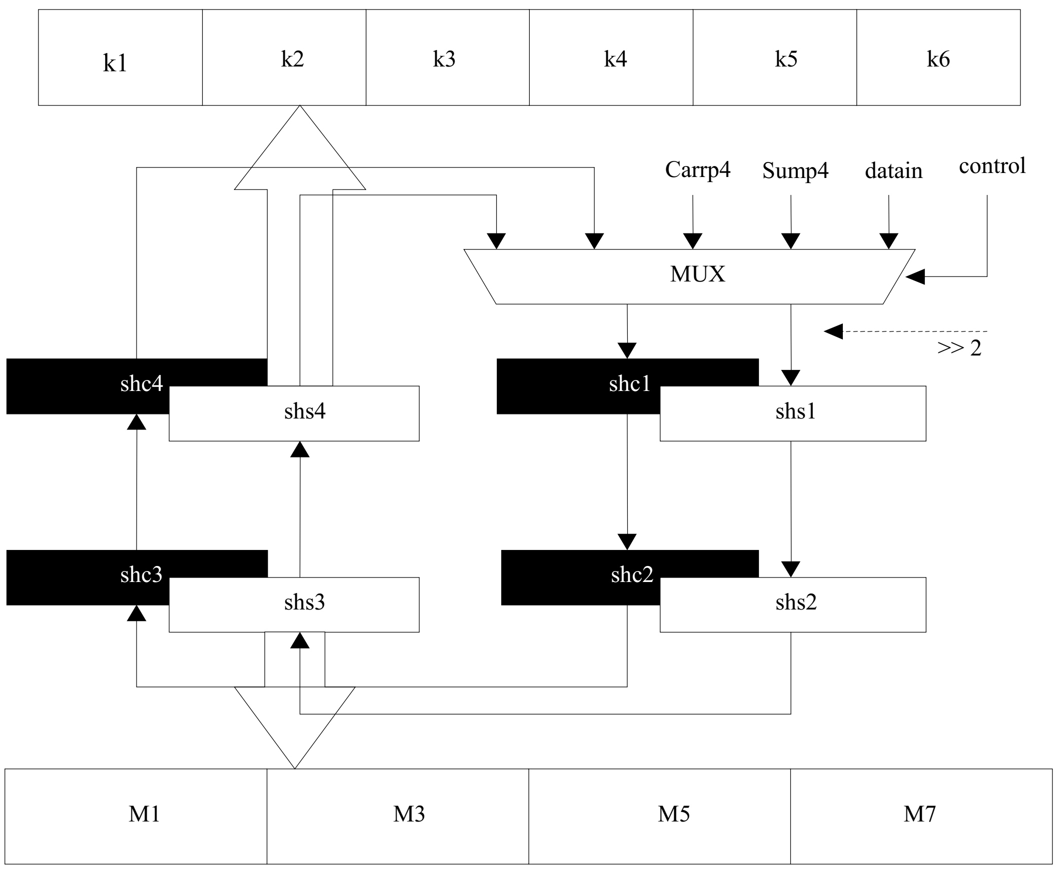

Figure 3. Tables and shift register.

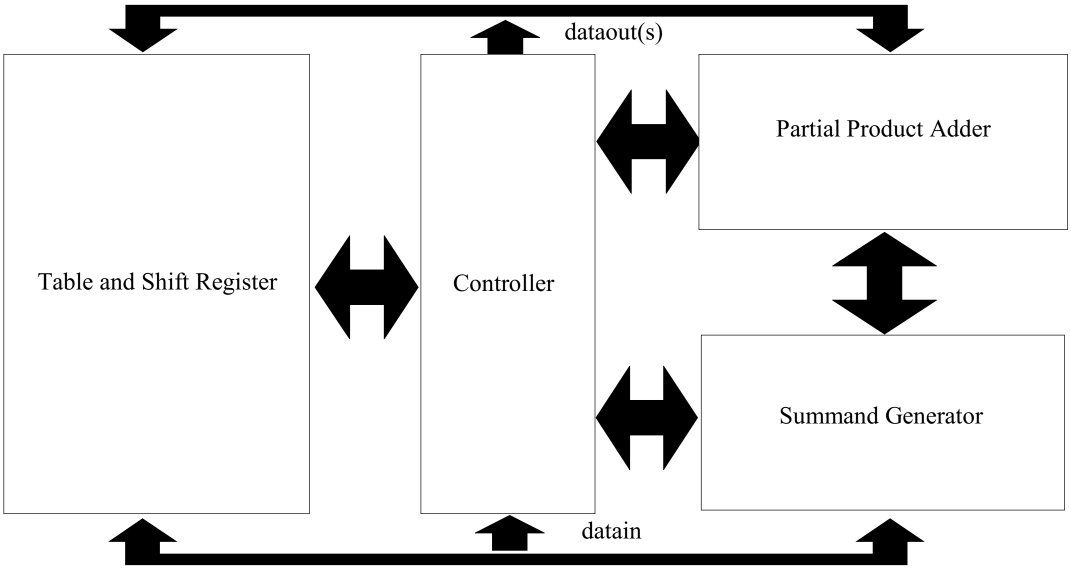

Figure 4. RSA processor.

To compute the partial product in the correct range, the summands of the final modular multiplication are set to zero.

5.3. Table and Shift Register

Figure 3 shows the block diagram of Table and Shift Register. The Table block stores precalculated values of $M1$, $M3$, $M5$, $M7$, and $k1 \sim k6$. While Shift Register is used to store B and put $M1$, $M3$, $M5$, $M7$, and $k1 \sim k6$ to the Table. Where B is the initial output of the partial product in each modular multiplication.

Table 7. Features of our RSA chip

Design Tool	Verilog-XL
Synthesis Tool	Synopsys
Technology Cell Library	Compass Standard Cell Library
Process Technology	TSMC 0.6 μm 1P3M Process
Power Supply	5 V
Gate Counts (2 input NAND)	80550
Die Size	5304.0 μm × 5356.8 μm
I/O	20-bit parallel, synchronous
Baud Rate (512-bit)	146 *k*bits/s with 200 MHz (worst case)
Voltage	5 V
Power consumption	N/A

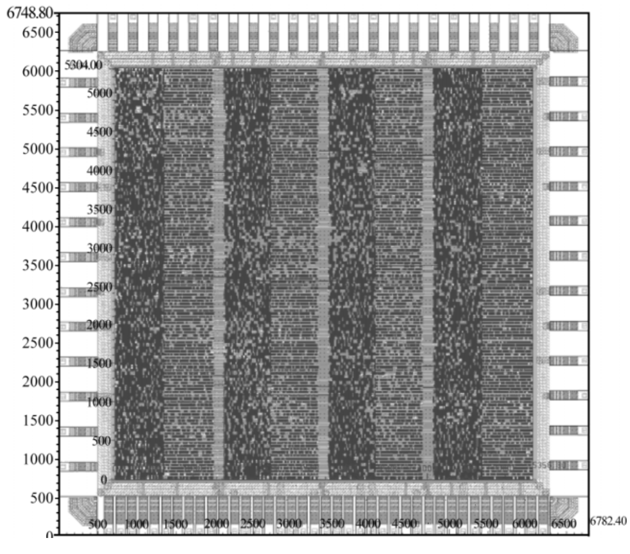

Figure 5. VLSI Layout of the 512-bit RSA chip.

Table 8. Features of four RSA chips

	Victor [20]	NTT [21]	Chen [22]	This Chip
Clock Speed	25 MHz	40 MHz	50 MHz	200 MHz
Baud rate Per 512 bits	100 K	20 K	24.3 K	146 K
Clock Cycles Per512Bits Encryption	0.125 M	1 M	1.05 M	0.7 M
Technology	1 μm	0.5 μm	0.8 μm	0.6 μm
Bits Per Chip	512	1024	512	512
Gate Counts	75 K	105 K	78 K	80 K

Table 9. Hardware requirement for various architectures

Authors		Hardware Requirement			
		Fas	REGs	MUXs	RAM
[3]	Wang I	$2n$	$13n$	$10n$	0
[3]	Wang II	$4n$	$26n$	$20n$	0
[6]	Sheu I	$3.18n$	$10.24n$	$9n$	0
[6]	Sheu II	$3.38n$	$13.88n$	$19.1n$	0
[7]	Eldridge	$3n + A1$*	$16n$	$9n$	0
[8]	Wu	$2n + A2$*	$12n$	$6n$	512×512
[9]	Juang	$2n$	$14n$	$10n$	10×512
	Ours	$1.5n$	$6.75n$	$4n$	10×512

*$A1 = n$ and $A2 = \log_2(n + 1)$ represent the number of used half adders.

Table 10. Time complexity for various architectures

Authors		Time Complexity		
		Addition	Comparison	Cycle time
[3]	Wang I	$1.5n^2$	No	FA
[3]	Wang II	n^2	No	FA
[6]	Sheu I	$2.4n^2$	Simple	2FA
[6]	Sheu II	$2.5n^2$	Simple	FA
[7]	Eldridge	$4n^2$	Simple	2FA
[8]	Wu	$2n^2$	No	FA
[9]	Juang	$6n^2$	Simple	FA
	Ours	$2.67n^2$	Simple	FA

5.4. RSA Processor

Figure 4 shows the architecture of the RSA processor.

We use Compass standard cell library (TSMC 0.6*um* process) to design a 512-bit RSA processor. The design is simulated by Compass ISM (input slope model) delay model. The simulation results show that the critical path delay is only 5*ns*, and the chip can operate up to 200-MHz clock. The processor delivers a baud rate of 146 kbits/s in the worst case. The features of our RSA processor are shown in Table 7, and the features of four RSA chips are shown in Table 8. Otherwise, Figure 5 shows the layout of the RSA chip.

The comparisons of hardware requirement and time complexity of the mentioned algorithms are listed in Tables 9 and 10 respectively. From the comparisons, the hardware of our architecture is small; the speed is reasonable, and the area–time product is very good.

6. Conclusion

We propose two methods to speed up the operation for modular exponentiations and modular multiplication respectively. The modified H-algorithm for modular exponentiation reduces the number of modular multiplication to $4n/3$. The modified L-algorithm for the modular multiplication reduces the operation times to half of the original L-algorithm and Montgomery's algorithm. In order to reduce the hardware requirement, only $n/4$ bits are executed in each stage of the proposed RSA processor. For the reduction of modular operation, we use the idea of replacing the overflow sum with the equivalent values that are precomputed, and thus no comparison with the modulus (N) is needed. Based on the algorithm, this RSA processor can achieve high performance. The simulation results show that the critical path delay is only 5*ns*. In the worst case, the architecture takes 0.7 *M* clock cycles to finish the modular exponentiation (512-bit modulus and 512-bit exponent). The processor delivers a baud rate of 146 *k*bits/s with 200-MHz clock frequency in the worst case.

References

[1] Rivest, R. L., Shamir, A. and Adleman, L., "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Com. of ACM*, Vol. 21, pp. 120–126 (1978).

[2] Takagi, N. and Yajima, S., "Modular Multiplication Hardware Algorithms with a Redundant Representation and Their Application to Rsa Cryptosystem", *IEEE Trans. on Computers*, Vol. 41, pp. 887–891 (1992).

[3] Wang, P. A., Tsai, W.-C. and Shung, C. B., "New VLSI Architecture of RSA Public-key Cryptosystem," *IEEE Int. Symp. on Circuits and Systems,* Hong Kong, Vol. 3, pp. 2040–2043 (1997).

[4] Chen, P.-S., Hwang, S.-A. and Wu, C.-W., "A Systolic RSA Public Key Cryptosystem," *IEEE Int. Symp. on Circuits and Systems*, Atlanta, GA, U.S.A., Vol. 4, pp. 408–411 (1996).

[5] Jeong, Y.-J. and Burleson, W. P., "VLSI Array Algorithm and Architectures for Rsa Modular Multiplication," *IEEE Trans. on Very Large Scale Integration* (*VLSI*) *Systems,* Vol. 5, pp. 211–217 (1997).

[6] Sheu, J.-L., Shieh, M.-D., Wu, C.-H. and Sheu, M.-H., "A Pipelined Architecture of Fast Modular Multiplication for Rsa Cryptography," *IEEE Int. Symp. on Circuits and Systems*, Monterey, CA, U.S.A., Vol. 2, pp. 121–124 (1998).

[7] Eldridge, S. E., "A Faster Modular Multiplication Algorithm," *Int. J. Computer Math*, Vol. 40, pp. 63–68

(1991).

[8] Su, C.-Y. and Wu, C.-W., "A Practical VLSI Architecture for Rsa Public-key Cryptosystem," *Proc. Sixth VLSI Design/CAD Symp.*, Nan–Tou, Taiwan, pp. 273–276, (1995).

[9] Juang, Y.-J., Lee, E.-H. and Chen, C.-H., "A New Architecture for Fast Modular Multiplication," *Int. Symp. on VLSI Technology, System, and Application,* pp. 357–360 (1989).

[10] Knuth, D. E., *Seminumerical Algorithms*, *the Art of Computer Programming*, Vol. 2, 2nd Ed. Reading, MA: Addison-Wesley, (1981).

[11] Yang, C., "IC Design of a High Speed RSA Processor," *Master Thesis, Institute of Electronic, National Chiao Tung University*, Hsin-Chu, Taiwan, June (1996).

[12] Takagi, N., "A Radix-4 Modular Multiplication Hardware Algorithm Efficient for Iterative Modular Multiplcations," *10th IEEE Symp. on Computer Arithmetic*, Grenobal, France, pp. 35–42, (1991).

[13] Shand, M. and Vuillemin, J., "Fast Implementation of RAS Cryptograph," *11th IEEE Symp. on Computer Arithmetic*, Windsor, Ont., Canada, pp. 252–259, (1993).

[14] Kahn, D., *The Codebreakers.* NY, U.S.A., Macmillan, (1967).

[15] Niven, I., Zuckerman, H. S. and Montgomery, H. L., *An Introduction to the Theory of Numbers*, Wiley, NY, U.S.A., (1991).

[16] Orup, H., "Exponentiation, Modular Multiplication and VLSI implementation of High-Speed RSA Cryptosystem," PhD dissertation, Dept. Comput. Sci., Univ. Aarhus, Arahus, Denmark (1995).

[17] Montgomery, P. L., "Modular Multiplication without trial Division," *Math. Comput.*, Vol. 44, pp. 519–521 (1985).

[18] Su, C.-Y., Hwang, S.-A., Chen, P.-S. and Wu, C.-W., "An improved Montgomery's Algorithm for High-Speed RSA Public-Key Cryptosystem," *IEEE Trans. on Very Large Scale Integration* (*VLSI*) *System,* Vol. 7, pp. 280–284 (1999).

[19] Chiang, J.-S. and Chen, J.-K., "An Efficient VLSI Architecture for RSA Public-Key Cryptosystem," *IEEE Int. Symp. on Circuits and Systems*, Orlando, FL, U.S.A., Vol. 2, pp. 121–124 (1999).

[20] Ishii, S., Ohyama, K. and Yamanaka, K., "A Signal-chip RSA Processor Implemented in a 0.5 μm Rule Gate Array" *IEEE International ASIC conference and Exhibit*, pp. 433–436 (1994).

[21] Holger, Orup "A 100 K bits/s Signal Chip Modular Exponentation Processor," *In HOT Chips VI, Symposium Record*, pp. 53–59 (1994).

[22] Chen, P.-S., Hwang S.-A. and Wu, C.-W., "A Systolic RSA Public Key Cryptosystem," *IEEE International Symposium Circuits and Systems*, Vol. 4, pp. 408–411 (1996).

Manuscript Received: Mar. 19, 2004
Accepted: Jun. 30, 2004