Substituting for the residues $x_1 = m_1 - 1$ and for $x_3 = 1$ satisfying the odd parity condition and simplifying

$$\begin{aligned} 2n\beta &< -m_1 m_3 + 2 - m_1 \\ n\beta &< -m_1(n+1) + 1 \\ \beta &< -m_1\left(1 + \frac{1}{n}\right) + \frac{1}{n} \\ \beta &< -m_1 - 2 + \frac{2}{n} \\ &< -m_3 + \frac{2}{n} \\ &< -m_2. \end{aligned}$$

Therefore,

$$\beta \leq -m_3.$$

Proof of C4: For the remaining range $(m_3 > \beta > -m_3)$, the correction $m_1 m_3/2$ yields the correct result.

ACKNOWLEDGMENT

The authors wish to express their gratitude to the anonymous reviewers who have taken time to go through this brief, and are thankful for their comments, which have considerably enhanced the quality of this brief.

REFERENCES

[1] A. B. Premkumar, "An RNS to binary converter in $2n+1$, $2n$, $2n-1$ moduli set," *IEEE Trans. Circuits Syst. II*, vol. 39, pp. 480–482, July 1992.
[2] K. M. Ibrahim and S. N. Saloum, "An efficient residue to binary converter design," *IEEE Trans. Circuits Syst.*, vol. 35, pp. 1156–1158, Sept. 1988.
[3] S. Andraos and H. Ahmad, "A new efficient memoryless residue to binary converter," *IEEE Trans. Circuits Syst.*, vol. 35, pp. 1441–1444, Nov. 1988.
[4] P. Bernardson, "Fast memoryless over 64 bits, residue to binary converter," *IEEE Trans. Circuits Syst.*, vol. CAS-32, pp. 298–300, Mar. 1985.
[5] B. Guan and E. V. Jones, "Fast conversion between binary and residue numbers," *Electron. Lett.*, vol. 24, no. 19, pp. 1195–1197, Sept. 1988.
[6] B. Vinnakota and V. V. B. Rao, "Fast conversion techniques for binary to RNS," *IEEE Trans. Circuits Syst. I*, vol. 41, pp. 927–929, Dec. 1994.
[7] S. J. Piestrak, "A high speed realization of residue to binary number system conversion," *IEEE Trans. Circuits Syst. II*, vol. 41, pp. 661–663, Dec. 1995.
[8] N. S. Szabo and R. I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology*. New York: McGraw-Hill, 1967.

A New RSA Cryptosystem Hardware Design Based on Montgomery's Algorithm

Ching-Chao Yang, Tian-Sheuan Chang, and Chein-Wei Jen

***Abstract*—In this paper, we propose a new algorithm based on Montgomery's algorithm to calculate modular multiplication that is the core arithmetic operation in an RSA cryptosystem. The modified algorithm eliminates over-large residue and has very short critical path delay that yields a very high-speed processing. The new architecture based on this modified algorithm takes about $1.5n^2$ clock cycles on the average to finish one n-bit RSA operation. We have implemented a 512-bit single-chip RSA processor based on the modified algorithm with Compass 0.6-μm SPDM CMOS cell library. The simulation results show that the processor can operate up to 125 MHz and deliver the baud rate of 164 Kbits/s on the average.**

I. INTRODUCTION

As the telecommunication network has grown explosively and the internet has become increasingly popular, security over the network is the main concern for further services like electronic commerce [1]. The fundamental security requirements include confidentiality, authentication, data integrity, and nonrepudiation. To provide such security services, most systems use public key cryptography. Among the various public key cryptography algorithms, the RSA cryptosystem[2] is the best known, most versatile, and widely used public key cryptosystem today. In public key cryptography algorithms, the essential arithmetic operation is modular multiplication, which is used to calculate modular exponentiation. However, modular exponentiation on numbers of hundreds of bits (512 bits or higher) makes it difficult for the RSA algorithm to attain high throughput.

An attractive method for faster implementations is based on Montgomery's modular multiplication algorithm [3], [4], in which the quotient only depends on the least significant digit of operands. Various algorithm modifications and hard-ware designs of Montgomery's algorithm can be found in [5]–[11]. To speed up processing, in [5] and [7], they used the high radix technique to reduce the required clock cycle number. In [6], they avoided the quotient determination by shifting the multiplicand 2 bit. This achieved a significant speed-up of modular multiplication available today. However, all the methods [5]–[9] suffer from the over-large residue. So an additional final reduction is required, which increases the hardware and time complexity. Though this problem is solved in [10] and [11], the iteration times are doubled. Also, in [5] and [7], the intermediate result is in carry-save form or redundant representation and the input operands are assumed to be in nonredundant binary form in the next modular multiplication. Therefore, additional cycles will be introduced at the start of the next iteration to convert the data format.

In this paper, we propose a modified Montgomery's algorithm to eliminate the aforementioned problems. The algorithm modifies the range of the partial product by separating the multiplication and modular reduction operation so that the output will fall in the right range after postprocessing. Therefore, we avoid the over-large residue problem and the additional subtraction procedure. The hardware

Manuscript received January 6, 1997; revised October 1, 1997. This work was supported by the National Science Council, R.O.C., under Grant NSC 84-2215-E009–057. This paper was recommended by Associate Editor K. K. Parhi.

The authors are with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

Publisher Item Identifier S 1057-7130(98)05048-4.

design based on the modified algorithm can start the next iteration immediately when the first digit of the previous iteration output is available. Hence, no additional cycles are required at the start of the next iteration. Furthermore, a fully pipelined design reduces the critical path and attains a high clock rate and high throughput.

This paper is organized as follows. In Section II, we will propose the modified Montgomery algorithm and apply it to the modular exponentiation. In Section III, the hardware design of a 512-bit RSA processor will be presented. Finally, we will conclude this paper in Section IV.

II. Modified Modular Exponentiation Algorithm

A. Modified Montgomery's Algorithm

Montgomery's modular multiplication algorithm [3] can avoid range comparison which is the critical operation in traditional division in modular multiplication. However, this algorithm requires some preprocessing and postprocessing to remove an extra factor and limit the range of intermediate output. To address these problems, Chen [10], [11] proposed a modified Montgomery's algorithm. Compared with the original Montgomery algorithm, Chen's algorithm has a simpler modular reduction step. The output will fall in the range smaller than N. However, the iteration step is increased from n to $2n$.

To eliminate such problems and keep the iteration step as low as in the original Montgomery algorithm, we propose a modified algorithm in C-like code as follows:

Algorithm 1 (The Modified Montgomery's algorithm)

```
(Inputs):
   Modulus:N (n-bit binary representation)
   Multiplier: A = (a_{n-1} a_{n-2} ... a_1 a_0)_2
   Multiplicand: B (n-bit number)
(Outputs):
   Result: R ≡ A * B * 2^{-n} (mod N),
           0 ≤ R ≤ 2^n + N < 2^{n+1}
   Quotient: Q = (q_{n-1} q_{n-2} ... q_1 q_0)_2
(Algorithm):
MM(A, B, N)
{
P1: A * B = C = C_1 * 2^n + C_0;
   (0 ≤ C_0, C_1 < 2^n, C_0 = (c_{n-1} c_{n-2} ... c_1 c_0)_2)
   P[0] = 0;
P2: for (i = 0; i < n; i++)
    {
            q_i = (P[i] + c_i) mod 2;
                // even or odd ?
       P[i + 1] = (P[i] + q_i * N + c_i) div 2;
          // right shift 1-bit
    }
    R = P[n] + C_1;
    Return R;
}
```

In this modified algorithm, similar to Chen's algorithm, the modular multiplication operation is split into multiplication procedure (P1) and Montgomery modular reduction procedure (P2). In P1, we only take the lower n bits (C_0) of the product to do the Montgomery reduction instead of whole $2n$ bits as in Chen's algorithm. The higher n bits (C_1) of the product has the same weight 2^n as $P[n]$. We sum C_1 with $P[n]$ to get the result (where $P[n]$ is the result of the C_0 after Montgomery reduction). Though the value of our result $(R = C_1 + P[n])$ is different from the result $(R[n])$ of Montgomery algorithm, they are equivalent in modulo N. This verification is shown in the Appendix which also shows the output range.

The algorithm takes n clock cycles as Montgomery's algorithm in [6] but shorter than Chen's algorithm that needs $2n$ cycles. Besides, the hardware cost for P2 is only one level of n-bit adders, which is as low as Chen's algorithm and half of Montgomery's algorithm in [6] because of only two n-bits (P[i] and qi * N) and one carry instead of three n-bit additions. So the critical path of the modified algorithm is the same as Chen's algorithm and is shorter than that in [6] in hardware implementation. Furthermore, the final modification of the result in [6] is no longer required in the modified algorithm, which means less delay and hardware.

B. Modified Modular Exponentiation Algorithm

The modified Montgomery algorithm cannot directly apply to modular exponentiation due to the extra factor 2^{-n} and the residue. We add some preprocessing and postprocessing steps to solve these problems. First, to bound the output range, we add one extra bit of precision to intermediate results A and B for precision consideration. This will increase the iteration steps from n to $n+2$ steps, and the extra factor will be $2^{-(n+2)}$. The extra factor is not removed explicitly. Instead, we first **pre-process** M by taking M and $(2^{2(n+2)} \bmod N)$ to compute $M' \equiv M2^{(n+2)} (\bmod N)$, so the unwanted factor will be removed automatically.

After the last iteration of modular multiplication operation, we **post-process R** by taking the result and 1 as input operands to remove the extra factor, i.e., R = MM(R * 2^{n+2}, 1). We can observe that if the input operand is 1, the higher n bit of product (C_1) will be zero, so the output result of postprocessing will be less than the modulus N. Therefore, we not only remove the unwanted factor 2^{n+2} of the result but also make the result fall in the right range after postprocessing. The new modified algorithm for computing modular exponentiation is described below:

Algorithm 2 (The modified modular exponentiation algorithm)

```
(Inputs):
   Modulus: N (n-bit number)
   Exponent: E = (1 e_{k-2} e_{k-3} ... e_1 e_0)_2
   Message: M (n-bit number)
   Constant: C = 2^{2(n+2)} mod N
(Function):
   MM(A, B, N) = A * B * 2^{-(n+2)} (mod N)
(Outputs):
   Result: R[k-1] = M^E mod N, 0 ≤ R < N
(algorithm):
   MM_E(M, E, N, C)
   {
       M' = MM(M, C, N); // pre-processing
       R[0] = M';
       for (i = 0; i < k - 1; i++)
       {
         R[i + 1] = MM(R[i], R[i], N);
         if (e_{k-i-2} == 1)
            R[i + 1] = MM(R[i + 1], M', N);
         else
            R[i + 1] = R[i + 1];
       }
       R[k - 1] = MM(R[k - 1], 1, N); //
       post-processing return R[k - 1];
   }
```

This algorithm takes about the same cycles as Montogomery's algorithm [3], [6] applied to modular exponentiation but needs less time because of a shorter critical path. The number of modular

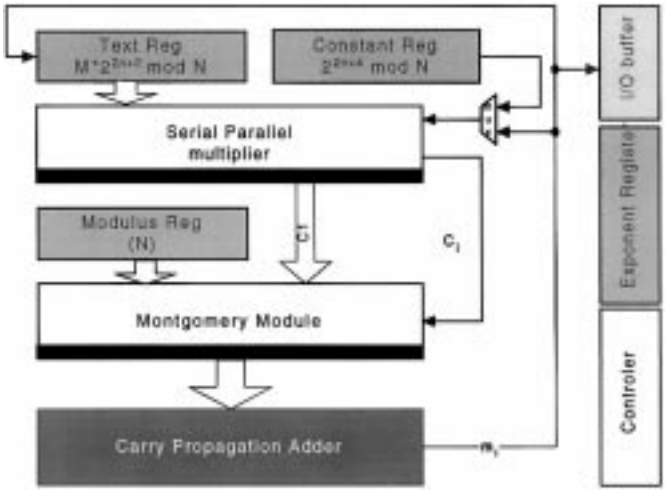

Fig. 1. Architecture of the 512-bit RSA processor.

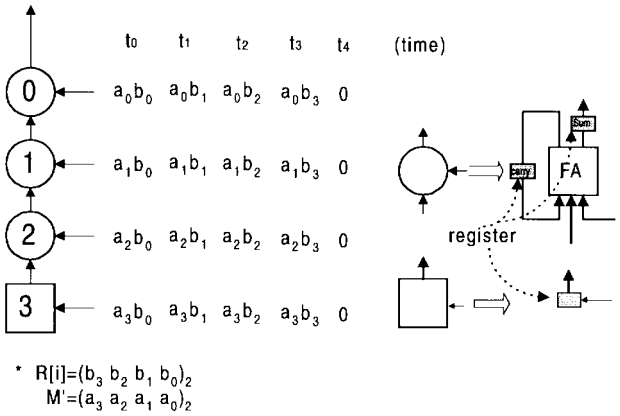

Fig. 2. Architecture and timing of linear array multiplier.

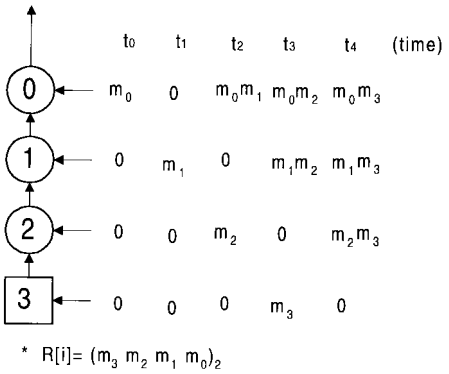

Fig. 3. The input data sequence for the squaring operation.

multiplications is $\lceil \log_2 E \rceil + v(E) - 2$, where $v(E)$ is the number of nonzero bits in E. So, for n-bit RSA modular exponentiation with equal probability for 0 and 1, the number of modular multiplication is $(2n+2)$ for the worst case and $(1.5n+2)$ for the average case. Algorithm 2 takes $(2n+2) \times n$ clock cycles which is shorter than that in [10] and [11] which need $(2n+2) \times 2n$ cycles to complete a modular exponentiation in the worst case. Since cycle time is equal to that in [10] and [11], our algorithm takes less time to complete RSA operations and has higher throughput.

III. Hardware Design and Realization

A. Hardware Design

Fig. 1 shows the architecture of a 512-bit RSA processor based on the modified algorithm. We use four 512-bit linear shift registers to store operands needed in computing 512-bit RSA operation $(\mathrm{M}^{\mathrm{E}} \bmod \mathrm{N})$. The operations of the RSA processor are described below. In the initial stage, RSA operands are loaded into shift registers serially through an 8-bit input buffer. While loading message M into the text register, we shift the exponent register until the first nonzero is the most significant bit and count the number of bits of exponent, $\lceil \log_2 E \rceil$. After the initial stages, we start the multiplier. Once the first output bit of the multiplier is ready, we start the Montgomery module immediately. So the execution time of CPA, multiplier, and Montgomery module is almost overlapped. Therefore, the function units of our design are fully utilized during computation.

1) Carry-Propagation Adder and Serial Parallel Multiplier: The carry-propagation adder converts the carry-save form of the output from the Montgomery module to nonredundant binary form. It generates one bit output per cycle to the serial-parallel multiplier for the next iteration. The serial-parallel multiplier shown in Fig. 2 is to realize the multiplication and square of two $n+1$ bit numbers. It first generates the $n+2$ lower bits of a product serially to the Montgomery module, then it stops and holds the n higher bit of the product. The n higher bits of the product will be added with the output of the Montgomery module to get the modular multiplication result.

The multiplier itself is a linear array type of multiplier with a special input circuit. The linear array shown in Fig. 2 (neglecting the AND array) is a direct systolic implementation. When the multiplier is generating a product of two numbers, the parallel input M' is ready in the text register and another operand $R[i]$ can arrive in serial. However, if we want to square one number, a serial input of the operand will make the multiplier fail. We solved this problem by scheduling the serial input operands $R[i]$ and insert some zeros to ensure the square operation from failure, as shown in Fig. 3.

2) Montgomery Module: The Montgomery module shown in Fig. 4 performs the modular reduction by repeating the following procedure: $a_i = (P[i]+c_i) \bmod 2$, $P[i+1] = (P[i]+a_i*N+c_i)/2$. $C_0 = (c_{n+1}\ c_n\ c_{n-1}\ \cdots\ c_1\ c_0)_2$ is the $n+2$ lower bit of the product from the multiplier. C_0 entered the Montgomery module one bit per cycle from the lower bit to the higher bit in series. The reduction step is a shift-and-add step that is very similar to the basic step of a multiplication. The quotient determination is a parity decision on the summation of the intermediate result and the carry. This can be done simply by an exclusive-OR gate with inputs of c_i and the LSB of the intermediate result in the previous iteration. After $n+2$ iterations, the Montgomery module will add $P[n+2]$ and the n higher bits of the product from the multiplier together. The result is then sent to the carry-propagation adder for the next modular multiplier iteration.

3) Hardware Cost and Performance: The total clock cycle of one 512-bit RSA operation in the processor is

$$4 * 512 + 519 * 2 + (\lceil \log_2 E \rceil + \nu(E)) * 519$$

where $\nu(E)$ is the number of 1 bit in the exponent. It takes about 0.39 M clock cycles for the average case (equal to 0 or 1 probability) or 0.54 M clock cycles for the worst case. The hardware cost analysis of the design (excluding control and I/O buffer) is listed in Table. I. The size of the controller part that uses 11 states for the counter-based finite state machine is quite small compared with the other part, as the final layout shows.

4) A 512-Bit Single-Chip RSA Processor: Fig. 5 shows the layout of the 512-bit RSA chip. We partition the 512 bit into eight main

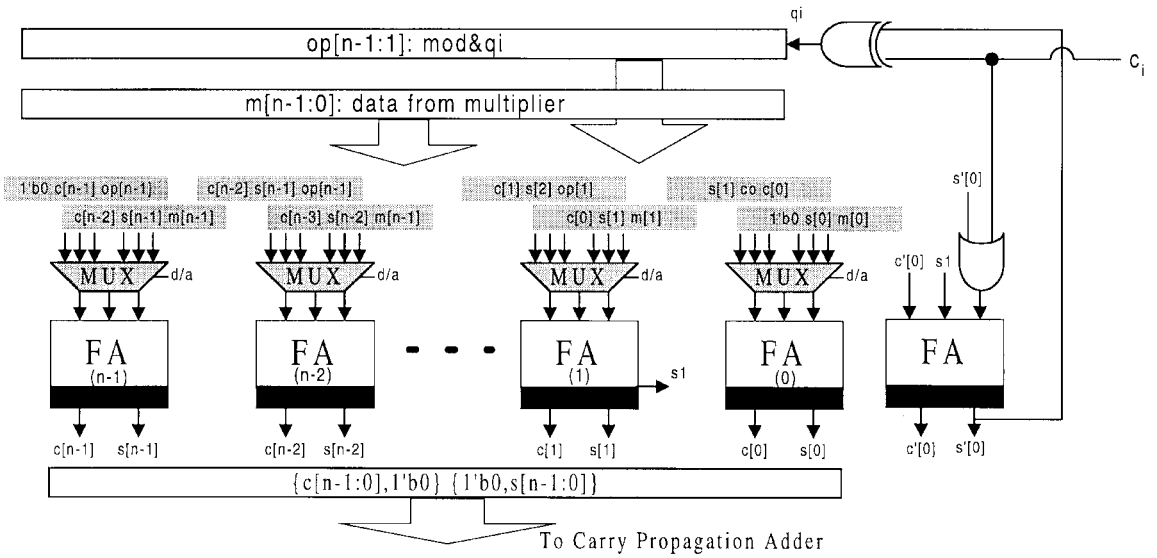

Fig. 4. The circuit of the Montgomery module.

TABLE I
HARDWARE COST ANALYSIS OF OUR DESIGN FOR THE n-BIT RSA ALGORITHM

	Register	Full Adder	AND	MUX
Modulus Reg	1n	0	0	0
Constant Reg	1n	0	0	1
Exponent Reg	1n	0	0	1
Text Reg	1n	0	0	1
Mont. Module	2n	n	1n	3n
CPA	2n	1	0	2n
Multiplier	4n	n	2n	3n
Total	12n	2n	3n	8n

blocks that were described in previous sections. The interconnections of data signals are all locally connected to neighboring blocks. This design has tried to minimize the effects of the global signal lines. We use a buffered tree structure to drive all global signals including control signals and clock lines. The tree structure resembles the H-tree structure. So the gate delay penalty caused by propagating the global signals is minimized to 1.5 ns. To include the effect of the wire loading, we use a Compass ISM (input slope model) delay model to estimate the critical path delay 6.06 ns that includes the gate delay and wire-loading delay. For a more conservative estimation, we add an extra 30% delay to the ISM results, so the total delay is $6.06 \times 1.3 = 7.8$ ns, roughly equals to 8 ns. So the chip can operate up to 125-MHz clock. (The 30% extra delay is according to our empirical experiences on the ISM model and physical measurement results.) The main technical characteristics of the chip are listed in Table II.

Some RSA chips presented so far are listed in Table III, which also gives cost and performance comparisons with our design. All the data are in the worst-case scenario. This table does not include data of [5], [6], [8], and [9] since no detailed chip information is available. The highest baud rate has been achieved in [7] by incorporating the Chinese Remainder Theorem (CRT) to gain the $4\times$ speedup. Our design can also incorporate the CRT to gain such speedup. However, the CRT is only suitable for the users who know the factorization of modulus N. In [12], the number of clock cycles needed in a 512-bit RSA operation has been greatly reduced by using the radix-32 technique. However, the critical path will also increase, so the clock rate will be lower. In our design, a higher clock rate can be applied because the critical path delay is shorter than others. Therefore, our design is the fastest chip, excluding the design in [7].

TABLE II
FEATURES OF THE RSA PROCESSOR

Technology	**CMOS 0.6μm SPDM**
Package	**80 CQFP**
Gate Counts(2 input NAND)	**74493**
Chip Size	**7996.8μm x 6993.9μm.**
Baud Rate (512-bit)	**164Kbits/s with 125Mhz clock (average case)**
I/O	**8-bit parallel, asynchronous**
Control	**on-chip**
number of cloks (512-bit)	**0.39M (average case)**

IV. CONCLUSION

In this paper, we propose a scheme based on Montgomery's algorithm to implement the most widely used RSA cryptosystem. The modified Montgomery's algorithm efficiently reduces the critical

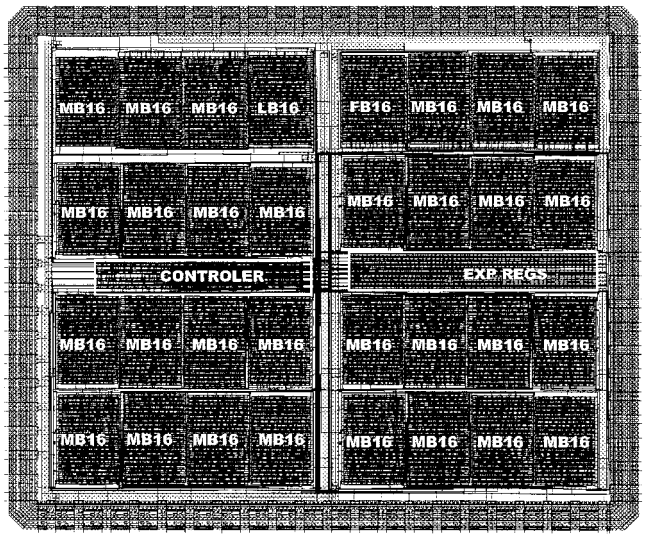

Fig. 5. Layout of the RSA processor.

TABLE III
SOME RSA CHIPS PRESENTED SO FAR

	Year	Gate Counts	bits per chip	# of clocks (512bits)	Technology	clock (Hz)	Baud Rate (bits/s)	CRT
M. Shand[7]	1993	100k	512	N/A	PAM*	40M	600k	Yes
Victor[12]	1994	75k	512	0.125M	1μm	25M	100k	No
NTT[13]	1994	105k	1024	1M	0.5μm gate array	40M	20k	No
Chen[10][11]	1995/96	77k	512	1.05M	0.8μm	50M	24.3k	No
Our design	1996	74k	512	0.54M	0.6μm	125M	118k	No

*Programmable active memory chip

path and operation cycles to increase the throughput rate. This scheme can avoid the final adjustment of residue and save the data format conversion time. A hardware design of a 512-bit RSA processor was also proposed and implemented based on this algorithm. The processor has high utilization and is regular to be cascaded for higher bits. The shortcoming of this design is that the global signals span 512 cells. A conservative analysis has included wire loading to estimate their effects. The processor takes about 0.54-M clock cycles to finish a 512-bit RSA encryption (decryption) and delivers a baud rate of 118 kbit/s at 125 MHz in the worst case. The variable encoding and decoding time for 512-bit RSA operation may not be desirable in some system designs but can be easily coped in asynchronous system designs. Besides, since we do not operate with all 512 bit, this design can save power consumption.

APPENDIX

The following is to verify that our modified Montgomery algorithm is modulo equivalent to the original Montgomery algorithm:

Verification:

In the for loop, by induction:

$$P[i+1] \star 2^{i+1} = \sum_{j=0}^{j=i} (c_j * 2^j) + N \star \sum_{j=0}^{j=i} (q_j * 2^j)$$

so, $0 \le P[i] < N + 1$

$$P[n] \star 2^n = \sum_{j=0}^{j=n-1} (c_j * 2^j) + N \star Q$$
$$= C_0 + N \star Q$$

Since

$$R = P[n] + C_1$$

Then

$$R \star 2^n = P[n] \star 2^n + C_1 \star 2^n$$
$$= C_1 \star 2^n + C_0 + N \star Q$$
$$= A \star B + N \star Q$$

Hence,

$$R \equiv A \star B \star 2^{-n} (\mathrm{mod}\, N)$$
$$0 \le R \le 2^n + N < 2^{n+1}$$

REFERENCES

[1] A. Bhimani, "Securing the commercial internet," *Commun. ACM*, vol. 39, no. 6, pp. 29–35, June 1996.
[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signature and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.
[3] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, pp. 519–521, Apr. 1985.
[4] C. K. Koc, T. Acar, and B. S. Kaliski, Jr., "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro. Chip, Systems, Software and Applications*, pp. 26–33, June 1996.
[5] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proc. 12th Symp. on Computer Arithmetic*, July 1995, pp. 193–199.
[6] S. E. Eldridge and C. D. Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," *IEEE Trans. Comput.*, vol. 42, pp. 693–699, June 1993.
[7] N. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," in *Proc. 11th Symp. on Computer Arithmetic*, 1993, pp. 252–259.

[8] C. D. Walter, "Systolic modular multiplication," *IEEE Trans. Comput.*, vol. 42, pp. 376–378, Mar. 1993.
[9] ____, "Still faster modular multiplication," *Electron. Lett.*, vol. 31, pp. 263–264, Feb. 1995.
[10] P. S. Chen, "VLSI implementation for a systolic RSA public key cryptosystem," Master's thesis, National Tsing Hua University, Taiwan, June 1995.
[11] P. S. Chen, S. A. Hwang, and C. W. Wu, "A systolic RSA public key cryptosystem," in *Proc. ISCAS*, 1996, vol. 4, pp. 408–411.
[12] H. Orup, "A 100 Kbits/s single chip modular exponentiation processor," in *HOT Chips VI, Symp. Rec.*, pp. 53–59.
[13] S. Ishii, K. Ohyama, and K. Yamanaka, "A single-chip RSA processor implemented in a 0.5 μm rule gate array," in *Proc. 7th Annu. IEEE Int. ASIC Conf. Exhibit*, 1994, pp. 433–436.

Analog Implementation of Fast Min/Max Filtering

S. Siskos, S. Vlassis, and I. Pitas

***Abstract*— An analog implementation of running min/max filters based on current-mode techniques is presented in this brief. Switched-current delay cells and current/voltage two inputs min/max selectors are used either for current or voltage inputs respectively. The voltage two input Min/Max circuit is designed using current conveyors and a modified structure of this is used to implement the running Min/Max filter for window size $n = 8$. Simulation results demonstrate the feasibility of the proposed implementation, which can be extended to a higher window size.**

***Index Terms*—Min/Max filters, mixed analog–digital integrated circuits, nonlinear filters, running filters.**

I. Introduction

In the recent years the use of nonlinear filters has exhibited a strong growth due to their capabilities to cope with system nonlinearities, non-Gaussian noise environments and sensor and perceptual system nonlinearities [1]. One of the most frequently used classes of nonlinear filters is based on order statistics [2]. Let us suppose that the input samples in the filter window are denoted by $x_1, x_2, \cdots, x_n$. If we order them according to their magnitude, we get their order statistics: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$. The minimal input sample is $x_{(1)}$ and the maximal input sample is $x_{(n)}$. The ith-order sample is denoted by $x_{(i)}, 1 \leq i \leq n$. The median of the input samples is $x_{(v)}$, where $n = 2v + 1$. Max/min filtering as well as median filtering are very frequently used in digital signal and image processing. In particular, maximum and minimum filtering are directly linked to the gray scale mathematical morphology operations *dilation* and *erosion* respectively [3]. Dilation/erosion is essentially a maximum or minimum operation respectively on the samples within the filter window. Both dilation and erosion have numerous applications, particularly in digital image filtering, edge detection, region segmentation and shape analysis. In the following, we shall concentrate our efforts in proposing digital filter architectures that are suitable to max/min filtering and that are easily implemented in a hybrid (analog/digital) way. The motivation to built these architectures is to construct extremely fast, simple, and affordable filters that operate directly on the analog signal and can be easily incorporated to smart sensors as well as into smart cameras. The proposed architectures are essentially suited to one dimensional signal filtering (e.g., sound, ECG/EEGS, measurements). However, due to the separability property of the max/min filtering, the same architectures and their implementation can be used for two-dimensional signal (image) processing, by applying them along image rows and columns independently.

Manuscript received February 7, 1997; revised August 8, 1997. Paper recommended by Associate Editor L. A. Akers.

S. Siskos and S. Vlassis are with the Laboratory of Electronics, Department of Physics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece.
I. Pitas is with the Department of Informatics Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece.

Publisher Item Identifier S 1057-7130(98)05049-6.

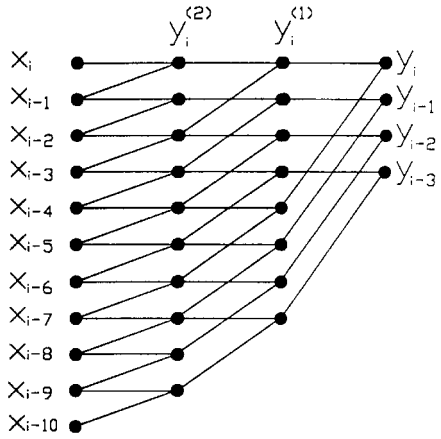

Fig. 1. The $\max$ calculation flow diagram for $n = 8$.

II. Fast Structures for Running Max/Min Filtering

The problem of running max/min filtering can be formulated as follows. Let $x_i, i \in Z$ by an one-dimensional signal. The output of a $\max$ or $\min$ filter $y_i, i \in Z$ is given by

$$y_i = T(x_i, \cdots x_{i-n+1}) \tag{1}$$

where n is the filter length (window size) and T is the$\max$ or $\min$ operator, respectively. Equation (1) is called "running" $\max$ or $\min$ filtering because after each output calculation, the filter window is shifted one position to the right (i.e., it "runs").

The computational complexity, measured in number of comparisons per output point, is $C(n) = n - 1$. It is desirable to construct filter structures that have a smaller number of comparisons per output point in order to speed the filtering process. This is accomplished by employing the "divide-and-conquer" strategy.

Let as suppose that the filter window size n is a power of two: $n = 2^k$. It is easily seen that $\max$ or $\min$ calculation of n numbers can be split into the $\max$ or $\min$ calculation of two subsequences of length $n/2$ each:

$$\begin{aligned} y_i &= T(x_i, \cdots, x_{i-n+1}) \\ &= T[T(x_i, \cdots, x_{i-(n/2)+1}), T(x_{i-(n/2)}, \cdots, x_{i-n+1})]. \end{aligned} \tag{2}$$

This procedure can be repeated recursively until we reach subsequences of length 2 [4]. In this case, the $\max$ or $\min$ calculation of two numbers is done by one comparison only. The corresponding flow diagram is shown in Fig. 1 for $n = 8$. Each dot corresponds to one comparison $T[\cdot, \cdot]$. The flow diagram has $\log_2 n$ stages. Only one extra comparison per output point is needed at each stage. Therefore, the computational complexity of this structure is reduced to $C(n) = \log_2 n$, which is much less than the complexity $n - 1$