Hierarchical Optimizations for High Speed Implementation of Modular Exponentiation in ASIC

Xuemi Zhao, Zhiying Wang, Hongyi Lu and Kui Dai
Department of Computer Science and Technology National University of Defense Techonology
410073, Changsha, P. R. China
E-mail: zhaoxuemi@263.net, {zywang, hylu, kuidai}@nudt.edu.cn

KEYWORDS
Modular exponentiation ,Hierarchical optimization, Super-pipeline, Montgomery multiplication, RSA

ABSTRACT

This paper presents a new arithmetic architecture hierarchically optimized for implementing modular exponentiation in ASIC. We combine a new version of high radix Montgomery multiplication algorithm with a super-pipeline design. With this algorithm, modular exponentiation (ME) can be decomposed into a series of primitive operation (PO) matrixes. All the POs are scheduled on the pipeline by employing column-sharing strategy, and inside the PO the partial results are compressed first by Wallace tree to assure only one carry propagation in the critical path. We also investigates speed/area tradeoffs using a 0.18um library, and gets that the ratio of the increasing in speed to the extra area used decreases as radix length *k* steps up from 2 to 64. A ME coprocessor SEA-II with *k*=32 was fabricated, with 140 MHz clock, a decryption rate of 6353 Kb/s can be achieved for 1024-bit RSA.

1 Introduction

It is widely recognized that security issues will play a crucial role in many future computer and communication systems. Although public-key algorithm is the essential part because of its convenience in key management and distribution. Modular exponentiation (ME) of long integer is the primary operation of several widely used public-key cryptosystems and is often the bottleneck for implementation in e-commerce servers and certificate authority centers. So there has been great interest in finding efficient ways to perform long ME. Field-programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs) are the two approachs.

Some famous FPGA-based designs are listed as follows. (Shand and Vuillemin 1993) designed a RSA (Rivest et al. 1978) coprocessor on 16 XC3090s , and it achieved a decryption speed of 165 kbps for a 1024-bit key. This design had been the fastest RSA implementation until (Blum and Paar 2001) proposed a systolic architecture based on high radix Montgomery algorithm, which can finish a 1024-bit RSA decryption in 3.3 ms on XC2V3000-6 device when using Chinese Remainder Theory (CRT). Blum's architecture stores the multiples of long integers and performs multiplication through looking up these tables, the radix is limited to 2^4 by the device, and the pre-computations need extra cycles. (Tang et al. 2003) combined a semi-systolic architecture with a similar algorithm to (Blum and Paar 2001). This design extended the radix length to 17 by employing the 18×18 signed multipliers in XC2V3000-6, and decreased the decryption time to 0.66ms.

Although FPGA implementations need shorter time to market and lower development cost, their performance is constrained by device resources and the price is high for larger volumes. The ASIC approach can overcome these shortcomings inherently. Orup and Korerup presented their ASIC implementation in (Orup and Korerup 1991), which can finish a 512-bit RSA signature in 5.5ms. (Hong et al. 2000) used Montgomery algorithm and computed modular multiplication and modular square alternatively to get a speed of 4.1ms to decrypt a 512-bit RSA. (Wu et al. 2001) used CRT beyond (Hong et al. 2000) and decreased the time to 1.7ms. References (McIvor and McLoone 2003, Liu et al. 2004) presented their special accelerating techniques based on carry save method, and could decrypt a 1024-bit RSA in 1.32ms and 1.01ms respectively. These reported works show that ASIC implementations were slower than FPGA's. Except the gap in manufacturing technologies, the more important reason is that they did not employ the design flexibility of ASIC to dig the parallelism exhaustively.

We optimize the ASIC implementation of ME hierarchically, i.e. from algorithm, pipeline architecture to micro-architecture. The improved algorithm will be introduced in Section II, and architecture optimization techniques will be discussed in Section III and Section IV respectively. Section V presents the results and we draw some conclusions in Section VI.

2 Algorithms

A ME can be decomposed into a series of parallel modular multiplication (MM) and modular square (MS) (Knuth 1981) by a parallel multiplication and square algorithm. So, we will focus on implementation of MM. In this section, first we review the high radix Montgomery multiplication algorithm, and then we modify it to a radix-length based version for efficient hardware implementation.

2.1 High Radix Montgomery Algorithm

Proceedings 20th European Conference on Modelling and Simulation
Wolfgang Borutzky, Alessandra Orsoni, Richard Zobel © ECMS, 2006
ISBN 0-9553018-0-7 / ISBN 0-9553018-1-5 (CD)

A MM can be unified as $R = A \times B \bmod M$. Montgomery proposed an efficient MM algorithm in (Montgomery 1985), it is a method for multiplying two integers modulo M, while avoiding division by M. However it still has difficulties in hardware implementation at two points: (I) the final subtraction induces irregularity of the hardware; (II) the operators' granularity is too large to be processed directly. (Eldridge and Walter 1993) proposed an algorithm suitable for hardware implementation by splitting the multiplier A in bits and avoided the final subtraction by extending the input condition from $0<A, B<M$ to $0<A, B<2M$, but the quotient determination limited the splitting granularity to higher radix. The quotient determination problem was solved in (Orup 1995), and the high radix method can accelerate the computation practically. The high radix version of Montgomery's algorithm is described as Algorithm 1.

Algorithm 1: High Radix Montgomery algorithm (HRM)

Inputs: $M=\sum_{i=0}^{m-1}(2^k)^i m_i, m_i \in \{0,1,2...2^k-1\}$;

$A=\sum_{i=0}^{m+2}(2^k)^i a_i, a_i \in \{0,1,2...2^k-1\}$, $a_{m+2}=0$;

$B=\sum_{i=0}^{m+1}(2^k)^i b_i$, $b_i \in \{0,1,2...2^k-1\}$;

$Ms'=-M^{-1} \bmod 2^k$, $\tilde{M}=M'\times M=\sum_{i=0}^{m}(2^k)^i \tilde{m}_i$, $\tilde{m}_i \in \{0,1,2...2^k-1\}$

Conditions: $A,B<2\tilde{M}$, $4\tilde{M}<2^{k(m+2)}$。

Output: $R = A\times B\times 2^{-k(m+2)} \bmod \tilde{M}$

1. $R_{-1} = 0$;
2. For (i=0;$i < m$+3; i=i+1)
   2.1 $q_i = R_{i-1} \bmod 2^k$;
   2.2 $R_i = (R_{i-1} + q_i \times \tilde{M})/2^k + a_i \times B$;
3. Output $R=R_{m+2}$.

When applying Algorithm 1 to ME, pre-computations and post-computations are required. These expenses can be negligible compared to the main part of ME, so we do not discuss these computations in this paper.

2.2 *Radix-length Based HRM Algorithm*

The complexity of Algorithm 1 lies in two $k\times(n+k)$ multiplications and one accumulation of three (n+2k)-bit addends, recall that $128<n<2048$ is of great interests in RSA. As the propagation of n carries is too slow and an equivalent carry look-ahead logic requires too many resources, two different approaches have been adopted: (I) Redundant representation: intermediate results are kept in a redundant form, resolution into binary representation is only done at the end. Carry-save method is a redundant representation (McIvor and McLoone 2003, Liu et al. 2004). (II) Pipelining: splitting multiplicand B to little words and pipelining. The widely used systolic array (Blum and Paar 2001, Tang et al. 2003) belongs to this category.

Two disadvantages exist when applying carry-save method to implement ME. Firstly because there is data dependency between adjacent iterations in ME algorithm [10], carry-save method needs extra carry propagation time, but pipelining method can avoid it through forwarding; secondly with the growth of the radix, the process unit in carry-save method will be too large to be optimized for current EDA tools, otherwise the process unit in pipelining method is much smaller for the same radix. Thus the pipelining method is better, but the splitting granularity is still a problem in the air.

Proposition 1: if employing pipelining method to implement Algorithm 1, we can get the highest performance when splitting granularity $L = k$.

Proof: a) As L decreases, the carry propagation path will decrease, and higher frequency will be get; b) if $L<k$, the data dependency of determining q will make the growth of the frequency nonsense, and the increased pipeline stages consume more resources to register the intermediate results, more register set-up time is needed, and the frequency will descend.

We take $L = k$ to split multiplicand B and modify Algorithm 1 as follows. At first we transform operation 2.2 in Algorithm 1 to $R_i = (R_{i-1}+q_i\times\tilde{M} + a_i\times(B<<k))>>k$. After substituting ($B<<k$) with B', we get $R_i = (R_{i-1}+q_i\times\tilde{M} + a_i\times B')>>k$. Then we split R_i, B' and $\tilde{M}$, that is

$R_i = \sum_{j=0}^{m+2} r_{ij}(2^k)^j, r_{ij} \in \{0,1,...,2^k-1\}, r_{i(m+2)} = 0$;

$b_0' = 0, b_j' = b_{j-1}, B' = \sum_{j=0}^{m+2} b_j'(2^k)^j$;

$\tilde{M} = \sum_{j=0}^{m+2} \tilde{m}_j(2^k)^j, \tilde{m}_{m+1} = \tilde{m}_{m+2} = 0$.

At last the long integer multiplication and addition of Algorithm 1 can be broken into multiplications and additions of k-bit digits, as Algorithm 2 describe.

Algorithm 2: Radix-length Based HRM Algorithm

Inputs, conditions and output are as same as Algorithm 1.

1. $R_{-1} = 0$; $B' = B<<k$;
2. For ($i = 0; i \le m+2; i = i+1$) // Loop i
   2.1 $q_i = r_{(i-1)0}$; $q\tilde{M}C_{i(-1)}=0$; $aB'S_{i(-1)}=0$; $C_{i(-1)}=0$;
   2.2 For ($j = 0; j \le m+2; j = j+1$) // Loop j
   2.2.1 $q\tilde{M}C_{ij} = (q_i \times \tilde{m}_j + q\tilde{M}C_{i(j-1)}) \mathrm{div} 2^k$;
   $q\tilde{M}S_{ij} = (q_i \times \tilde{m}_j + q\tilde{M}C_{i(j-1)}) \bmod 2^k$;
   2.2.2 $aB'C_{ij} = (a_i \times b_j' + aB'C_{i(j-1)}) \mathrm{div} 2^k$;
   $aB'S_{ij} = (a_i \times b_j' + aB'S_{i(j-1)}) \bmod 2^k$;
   2.2.3 $C_{ij} = (r_{(i-1)j} + q\tilde{M}S_{ij} + aB'S_{ij} + C_{i(j-1)}) div 2^k$;
   $r_{ij} = (r_{(i-1)j} + q\tilde{M}S_{ij} + aB'S_{ij} + C_{i(j-1)}) \bmod 2^k$;
   2.2.4 $r_{i(j-1)}=r_{ij}$; // k-bit right shift
3. Output $R=R_{m+2}$.

3 Column-sharing Super-pipelined Architecture

In this section, a super-pipelined architecture is proposed to minimize computing cycles.

3.1 Parallelism Analysis

Definition 1: A primitive operation (PO) is a oeperation set which includes all the calculations in the same

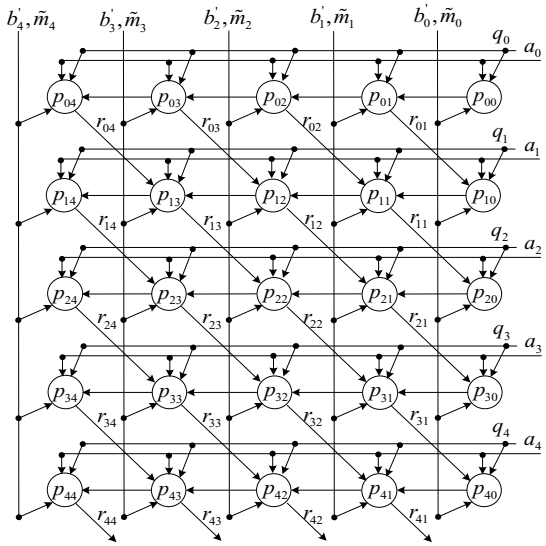

Figure 1: Data dependency analysis of PO matrix for m=2

iteration of loop j in Algorithm 2, the PO must be implemented in single cycle and can be presented as P_{ij}.

According to Definition 1, the MM can be decomposed into a (m+3) x (m+3) matrix of POs, and the ME can be taken as a series of PO matrixes. Fig.1 illustrates the data relationship in the PO matrix of MM for m=2 and we get the data sharing and data dependency as follow:

- Data sharing: (I) The POs of row i share a_i and q_i; (II) the POs of column j share b_j' and $\tilde{m}_j$.

- Data dependency: (I) The P_{ij} depends on the result of $P_{(i-1)(j+1)}$ and the carries of $P_{i(j-1)}$; (II) the q_i which is shared by the POs of row i depends on the result of $P_{(i-1)1}$.

In Fig.2, we simplify the graph and do a transformation to make the parallelism apparently. Because the MM and MS of the same iteration can be executed simultaneously [10], we analyze them together, and the principle is that POs in the same column have no data dependency. The upper half of Fig.2 shows the parallelism of the MM and MS separately, the lower half shows the scheduled result with column-sharing strategy. The concept of column-sharing is that POs in the same column of Fig.1 share one processing unit (PU). PU realizes the PO physically. In the lower half of Fig.2, every PU orderly performs POs in the same circle. We can get that the most parallelism is 5 for m=2, and m+3 for common cases. So to get the highest speed, the number of PU should be m+3.

3.2 Super-pipelining Architecture

Fig.2 demonstrates the idea of column-sharing super-pipelining and we generate the pipeline as follow:

- The m+3 PUs are arranged from right to left with j ascending.

- The architecture of a PU is generated. A register is generated for each source operator of PO, and let the combinational logic array (CLA) to

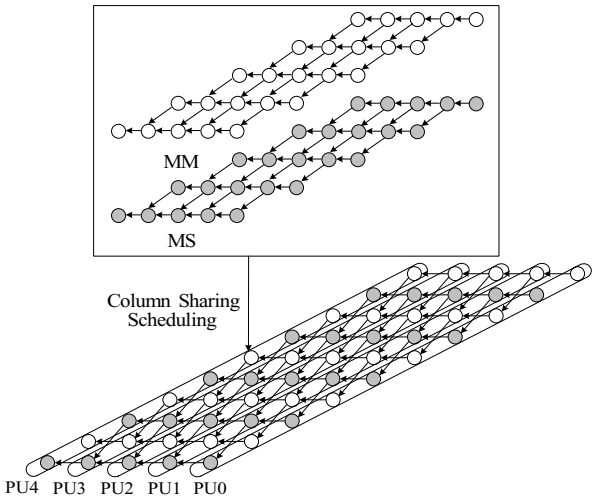

Figure 2. Column-sharing scheduling of MM and MS PO matrixes

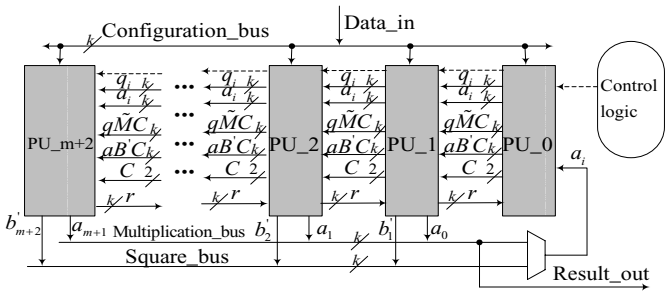

Figure 3. Architecture of Column-sharing Super-pipeline.

perform the computations of PO. Then seven k-bit registers are generated for *a_i*, *b_i*, *aBC_i*, *q_i*, *m_i*, *qMC_i*, and *r_i*, and one 2-bit register is required for *C_i*. Micro architecture of CLA will be described in Section IV.

- Interconnection between adjacent PUs is generated according to the data sharing and dependency relationship. Signals passed from right to left are q_i, a_i, carries and control signals; the signal feed-backed is r. M, A, B are all distributed stored in the PUs. The initial value of q_i is the result of PU_1. Configuration bus is used for initialization. Multiplication bus and square bus are added to read a_i. Results are accessed through multiplication bus.

- The control block generates control signals to the pipeline and the buses. The final pipeline architecture is shown in Fig.3.

At the beginning of implementing ME with the super-pipeline, P and Z are mapped to A and B respectively. A and B are initiated through the configuration bus. Then MM ($P = P$ x Z) and MS ($Z = Z^2$) in the same iteration in ME will be processed in the pipeline as Fig.2 shows. Every PU computes the POs of MM and MS alternatively, but the result of MM is written to A only when corresponding bit of exponent is not zero.

Because data dependency only exists between adjacent iterations, the ME can be pipelined by forwarding. But this would induce that the critical path

is composed of the computation path and the forwarding. This would expand the cycle time heavily. So here we write the result of one MM or MS to the PUs first, then read it by multiplication_bus or square_bus, and the critical path is broken into computation path and transition path. Consequently one pipeline stall must be inserted. Additional m+2 cycles are required to flush the pipeline, so the whole cycles consumed by the main part of one ME are:

$$K = l \times [2 \times (m+3)+1] + m + 2. \quad (1)$$

4 Micro Architecture

In this section, we optimize micro architecture of CLA to reduce the clock time. As shown in previous section, CLA is the kernel of a processing unit, which implements the PO. From Fig.4(a) we can get that the critical path of CLA is composed of one multiply-accumulation (MAC) and two additions, if we employ parallel MACs and adders directly, there will be carry propagation additions (CPAs) for three times. Recall that carry propagation is time-consuming. Our target is to reduce CPAs, and the idea is that all the partial results are compressed first by Wallace tree to assure only one CPA in the critical path. We extract the CPAs from parallel MACs and merge it with the two later additions, and then the 6 addends will be compressed, finally only one CPA is left.

Fig.4(b) shows the optimized micro architecture. AND arrays perform bit multiplications to generate partial products, a and b as same as q and m generate k partial products. Combined with carries $aB'C\_i$ or $q\bar{M}C\_i$ of long multiplications, these partial products are compressed to 2k-bit (C, S) pair by Wallace tree, which is called 'Wallace tree 0'. The higher significant half are used to generate the current PU's carry $aB'C\_o$ or $q\bar{M}C\_o$ of long multiplications. Carry from the lower half of (C, S) pair is extracted by using carry generator (CG). Carry select adder (CSA) is employed to implement CPA. Carries of long multiplications are calculated as follow: two results are first calculated by CSAs which speculate their input carry with 0 and 1 separately, the lower half's carry extracted by CG selects the result by multiplexer (MUX) .

The lower half of (C, S) are compressed with r_i and C_i by another Wallace tree, i.e. 'Wallce tree 1', and we get a new (k+3)-bit (C, S) pair. Lower k-bit of the new (C, S) are added to generate r_o. The left 3-bit with carry generated by this adder and the carries from CGs construct C_o through a carry amender (CA).

To generate the bit-wise partial products, we need k^2 AND gates. And the Wallace tree 0 and 1 require k^2-2k+4and 3k+3 full adders (FAs) respectively. For CSA, $2 \cdot 3\log_2 k - 1$ FAs and $\sum_{i-2}^{\log_2 k} 2^{i-1} \cdot 3^{\log_2 k - i}$ MUXes are required. CG is also implemented with carry select

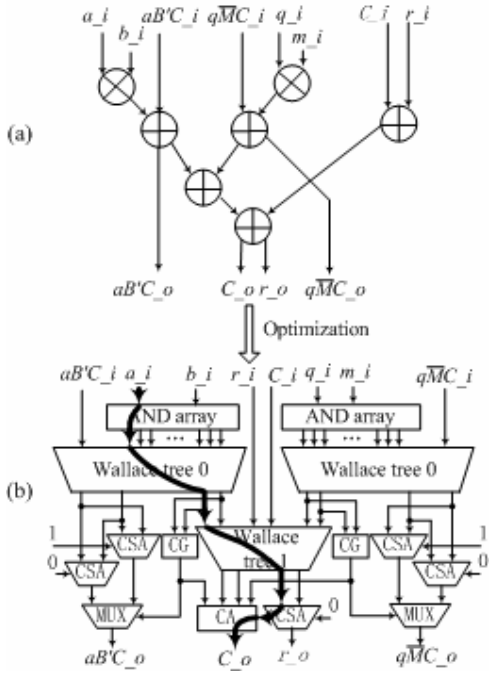

Figure 4. (a) Operations involved in CLA (b) Optimized micro architecture

method, but needs less k/2 MUXes. CA is very small and can be ignored. So gate count of the CLA can be calculated as:

$$G_{CLA} = 2k^2 \bullet G_{AND} + (14 \bullet 3^{\log_2 k - 1} + 2k^2 - k + 11) \bullet G_{FA} + (7 \bullet \sum_{i=2}^{\log_2 k} 2^{i-1} \bullet 3^{\log_2 k - i} + k) \bullet G_{MUX} \quad (2)$$

The critical path of the optimized architecture is marked with bold lines in Fig.4(b), and it consists of one AND gate, one Wallace tree with (k+1) partial products (i.e. Wallace tree 0), one Wallace tree with 6 addends (i.e. Wallace tree 1), one k-bit carry select adder and part of CA. The delay of Wallace tree 0 and CSA can be calculated as:

$$D_{Wallacetree0} = [\log_{1.5}(k+1)] \bullet D_{FA} \quad (3)$$

$$D_{CSA} = D_{FA} + [\log_2 k - 1] \bullet D_{MUX} \quad (4)$$

Delay of Wallace tree 1 is a constant, which equals $3 \bullet D_{FA}$. Time consumed by part of CA in the critical path is $1 \bullet D_{MUX}$. With equation (3) and (4), we get the CLA's delay:

$$D_{CLA} = D_{AND} + [4 + \log_{1.5}(k+1)] \bullet D_{FA} + [\log_2 k] \bullet D_{MUX} \quad (5)$$

5 Results

This section presents the experiment results. First we reviewed how the radix length k takes effects on the area and performance for our architecture. Then a ME coprocessor with k=32 was fabricated in a multi project wafer (MPW), and results of applying the prototype chip to RSA were compared with previous works.

5.1 Reviewing Radix Length

To review radix length k conveniently, a super-pipeline generator and a PU generator were implemented in Python. Input arguments are radix

length k and maximum modulus length n, the output are source codes in Verilog hardware description language. These designs were synthesized by using a 0.18um CMOS stand cell library.

Table 1. Relationship between k and area, performance

k	Num. of PU	Scale (K NANDs)	Delay (ns)	Time(us)
2	515	162	1.67	1764.0
4	259	214	2.01	1068.7
8	131	318	2.83	762.5
16	67	433	4.52	625.1
32	35	786	6.98	507.8
64	19	1592	10.92	436.3

When modulus length n=1024 and exponent length l = 1024, the results are shown in Table I. When doing

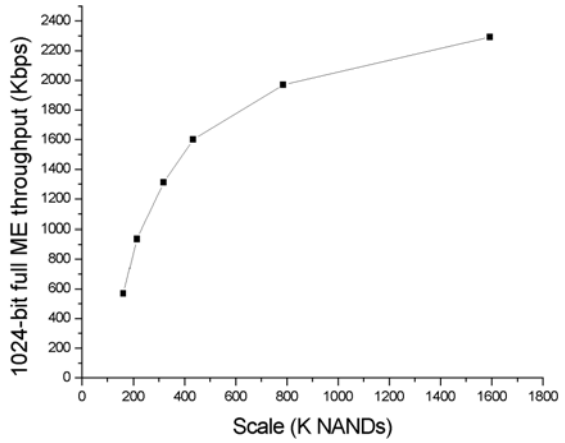

Figure 5. Tradeoffs between scale and throughput for 1024-bit ME

floor plan, we set the utilization to 70% to leave enough area for clock tree and in placement optimization. We got the computing cycles with equation (1), and the critical path delay, i.e. the clock cycle time, was achieved by static time analysis and signal integrity was fully taken into account.

For the super-pipeline, the number of PU is m+3, i.e. [n/k]+3. Additional with equation (2), and considering the sequential part, we can get the coprocessor's scale in theory: $G_{coprocessor} = ([n/k]+3)\times(G_{CLA}+(7k+2)G_{FF})$. The actual scale result is from 30 to 38 percent higher than the theoretical value. The area complexity increases with O(k), when k steps up.

From equation (1) and (5), we get that time complexity of coprocessor to implement ME is $O(\log_2 k/k)$. But the actual time decreases much slower than it, when k steps up. The reason is that optimization ability of back-end EDA tools descends heavily while single PU's scale increases with O(k^2).

The result of reviewing tradeoffs between scale (i.e. area) and performance is shown in Fig.5. Ratio of the delta in speed to the corresponding change in area decreases as k increases. The ratio is greater than 1 when 2<k<16, and less than 1 when 16<k<64.

5.2 Result of Fabrication

Because 5mm x 5mm was the basic block in MPW that our design took part in, considering fee, we took k=32 to finish the full design of a long integer ME coprocessor: SEA-II. SEA-II has 38 PUs, it support either one full 1024-bit ME or two parallel full 512-bit MEs. Besides this kernel PU array, it has a pre-process unit, a post-process unit, a 256 x 32 register file, and an interface to general purpose processors. In addition of IO pads and power ring, the whole die area is 4.7mm x 4.7mm, and the scale equals 923K NANDs. Now the prototype of SEA-II has been fabricated and tested.

5.3 Test Results for RSA

The test system is a board which was connected with a personal computer through a PCI bus. Hardware of the computer was Pentium4 2.4G + 512MB memories, and the operation system was Redhat 9.0. Except for SEA-II, there were three other chips in the test board: a 16-bit Digital Signal Processor (DSP), a PCI slave chip, and an 8-MB SDRAM, the DSP controlled the whole board and measured performance, and the SDRAM stored the raw data and results. Table II shows the test results.

Table 2. Test results of SEA-II for RSA

Key length	Operation	SEA-II operation	Time with SEA-II (us)
768	Signature	(384 x 384) x 2	95.9
768	Authentication	(768 x 17) x 1	11.1
1024	Signature	(512 x 512) x 2	157.4
1024	Authentication	(1024 x 17) x 1	14.3
2048	Signature	(1024 x 1024) x 2	1127.9

Column 'SEA-II operation' presents the operations performed by SEA-II, e.g. a 1024-bit RSA signature needs two MEs which are 512-bit in both modulus length and exponent length. Exclude time consumed by SEA-II, the whole time includes data I/O and assemble time to finish CRT in signature. When I/O voltage is 3.3V and the core voltage is 1.8V, the top working frequency of SEA-II is 140 MHz. For key length = 1024, a RSA decryption (i.e. signature) rate of 6353 Kbps can be achieved with SEA-II as a coprocessor, and the actual measured average power is 1.619W.

5.4 Comparison to Previously Reported Works

Table III lists comparisons to previously reported implementations since 2001, they all perform 1024-bit RSA signature.

Table 3. Comparison with previously reported works

	Year	Device/ Technology	Scale (K NANDs)	Clock (MHz)	Rate (Kbps)
[3]	2001	XC40250XV-09	-	46	322
[4]	2003	XC2V4000-6	-	90	1515
[8]	2003	0.18um	109	150	294
[9]	2004	0.13um	198	556	986
SEA-II	2006	0.18um	923	140	6353

The fastest implementation that we found is in the reference [4], which is based on XC2V4000-6 FPGA

device. This work employed a semi-systolic architecture and 18×18 signed multipliers embedded in the device. The multiplier resources constrained the algorithm's radix length to 17, and its micro-architecture of the processing unit used the embedded multiplier directly, and three CPAs are required. However, by using our optimization technique, the PU of SEA-II only needs one CPA and has higher clock. By extending radix length to 32, for the same ME, SEA-II needs much less cycles, and SEA-II is 4.2 times as fast as [4] as a whole. Compared to the reported ASIC implementations, SEA-II is 7.2 times faster than [8] at additional area cost of 3.9 times in the same technology. Although the feature size which [9] targeted is smaller, SEA-II is 5.7 times faster at larger scale of 3.7 times. The carry-save method that [8, 9] used has difficulties in extending radix to higher length because single processing unit will be too large to be optimized for current EDA tools. As Table I shows that our architecture has the similar scale as [9] when k=2, we can get a throughput of 2251 Kbps when using CRT, and this speed is still better.

6 Conclusions

In this paper, we presented a new arithmetic architecture hierarchically optimized for high-speed implementation of modular exponentiation in ASIC. Based on radix-length based high radix Montgomery algorithm, we took the long integer modular exponentiation as a series of primitive operation matrixes, then we proposed a column-sharing super-pipelining architecture to minimize the processing cycles and a novel micro architecture to reduce the carry propagation additions in the critical path of a primitive operation. Experiment results about relationship between radix-length k and the area/performance showed that area increased with O(k), and performance increased with much slower than O(k/log$k$), when k steps up. Result of reviewing tradeoffs between area and performance showed that, the ratio of the delta in speed to the corresponding change in area decreases as radix length k steps up from 2 to 64. Because of cost constraints, we took k=32 to implement a full coprocessor SEA-II, which had been taped out on .18um 1P6M CMOS logic technology. When applying the prototype chip to RSA, for key length equals 1024, we can achieve a RSA decryption (i.e. signature) rate of 6353 Kb/s. The performance is much better than the previous reported FPGA or ASIC implementations.

REFERENCES

Rivest R. L； A Shamir； L Adlema. 1978. A Method for Obtaining Digital Signature and Public-key Cryptosystems, Communications of ACM, 21(2): 120-126.

Shand M. and J. E. Vuillemin. 1993. Fast Implementations of RSA Cryptography, Proceedings 11th IEEE Symposium on Computer Arithmetic

Blum T.and C. paar. 2001. High-radix Montgomery Modular Exponentiation on Reconfigurable Hardware," IEEE transactions on Computers, 50(7):759-764

Tang S. H.; K. S. Tsui and P. H. W. Leong. 2003. Modular Exponentiation using Parallel Multipliers, Proceedings of the 2003 IEEE International Conference on FPGA.

Orup H. and P. Kornerup. 1991. A High-radix Hardware Algorithm for Calculating the Exponential m^e Modulo n, Proceedings of the 10th IEEE Symposium on Computer Arithmetic.

Hong J. H.; P. Y. Tsai; and C. W. Wu. 2000. Interleaving Schemes for a Systolic RSA Public-key Cryptosystem Based on an Improved Montgomery's Algorithm, Proceedings 11th VLSI Design/CAD Symposium.

Wu C. H.; J. H. Hong and C. W Wu. 2001. RSA Cryptosystem Design Based on the Chinese Remainder Theorem. Proceedings of ASP-DAC of Asia and South Pacific.

McIvor C. and M McLoone. 2003. Fast Montgomery modular multiplication and RSA cryptographic processor architecture, Proceedings of 37th Asilomar Conference on Signals, Systems and Computers.

Liu Q.; F. Ma; D. Tong and X. Cheng. 2004. A Regular Parallel RSA Processor, Proceedings of 47th Midwest Symposium on Circuits and Systems.

Knuth D. E.. 1981. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley.

Montgomery P. L.. 1985. Modular Multiplication without Trial Division, Mathematics of Computation, 44(170): 519-521.

Eldridge S. E. and C. D. Walter. 1993. Hardware Implementation of Montgomery's Modular Multiplication Algorithm, IEEE transactions on Computers, 42(6): 693-699, 1993.

Orup H. 1995. Simplifying Quotient Determination in High-radix Modular Multiplication, Proceedings of 12th Symposium on Computer Arithmetic.

AUTHOR BIOGRAPHIES

Xuemi Zhao, born in Shandong of China. He is a Ph. D. degree candidate in computer science from National University of Defense Technology, Changsha, China. His research interest is designing efficient architectures for DSP.

Zhiying Wang, born in Shanxi of China. He is a professor in National University of Defense Technology, Changsha, China. His research interest includes computer architecture, parallel computing, computer security etc.

Hongyi Lu, born in Guangxi of China. He is a lecturer in National University of Defense Technology, Changsha, China, and his research interest including high-performance embedded processor, VLSI design etc.

Kui Dai, born in Hubei of China. He is an associate processor in National University of Defense Technology, Changsha, China. His research interest include computer security, computer architecture etc