Design and Implementation of a Coprocessor for Cryptography Applications

Ander Royo, Javier Morán, Juan Carlos López

Dpto. Ingeniería Electrónica, Universidad Politécnica de Madrid
Ciudad Universitaria s/n. Madrid 28040. Spain
e-mail: {ander, moran, lopez}@die.upm.es

Abstract

In this paper, an ASIC suitable for cryptography applications based on modular arithmetic techniques, is presented. These applications, such as for example digital signature (DSA) and public key encryption and decryption (RSA), use, as basic operation, the modular exponentiation. This ASIC works as a coprocessor with a special set of instructions specialized on dealing with high accuracy integers, as well as on the rapid evaluation of modular multiplications and exponentiations. The algorithm, the hardware architecture, the design methodology and the results are described in detail.

1. Introduction

Security has become a key issue in the world of electronic communication. Besides how fast data are transmitted, the security of these data through the communication channel arises as one of the most important problems. Though, the time overhead due to data encryption and decryption should not impose a bottleneck in the communication process.

Public key cryptography (RSA), as well as other ways of ciphering and digital authentication based on modular arithmetic, is widely used in the world of secret digital transmission. But these systems have an important drawback: the slowness of their basic operation, the modular exponentiation with long (more than 512 bits) integers. Since software implementations are too slow, even running in fast processors, the use of a specific hardware seems to be the only reasonable solution to obtain good performances.

The main purpose of designing an ASIC to accomplish these generic goals is to have a hardware platform for the fast computation of modular exponentiation. At the same time, some other usual operations in the field of modular arithmetic cryptography (division, primality test,...) should be supported. Thus, a complete set of operations to give support to a wide range of cryptography applications has been developed. The result is the coprocessor whose design and implementation is described here.

The outline of this paper is the following. First, the selected algorithm is studied discussing all the different design decisions that have been made. Next, the hardware architecture and the design methodology are described. Finally, the results obtained with the chip that has been implemented are discussed and some conclusions are drawn.

2. The Algorithm

The RSA encryption method [4] is based on two *keys*, being each of one a pair of positive integers. One of the keys is public, and different users employ it to encrypt the data to be sent to the key owner. The other key is secret, and the owner uses it to decrypt the received message.

Let (a, M) be the public key and (b, M) the secret key. The cipher process is based on a modular exponentiation 'E = C^a *mod* M', where 'C' is the text to be ciphered, and 'E' the codified message. On the other hand, the keys have to fulfill some special conditions [4] to allow the decryption process to be made with the same exponential operation, but using the secret key instead the public one:

$$E^b \textit{ mod } M = (C^a \textit{ mod } M)^b \textit{ mod } M = (C^a)^b \textit{ mod } M = C \textit{ mod } M$$

Note that the message to be encoded is divided in blocks which value is smaller than the modulus, so 'C *mod* M = C'. Also, a complete explanation of the previous formula can be found in [4].

The starting point for all the algorithms used in modular exponentiation (ME) is the modular multiplication (MM). Thus, the faster the MM is performed, the faster the encryption and decryption processes will be accomplished. Therefore, the design and implementation of MM has become the key factor in designing cryptographic systems.

The procedure to evaluate a modular exponentiation is to perform successive multiplications. There are many efficient methods to evaluate exponentiations [9], but their complexity precludes us to describe all of them. In this case, the Russian Peasant method [9] has been selected, due to its low hardware requirements and easy control characteristics.

There are many algorithms to deal with MM. Some of them either use look-up tables or perform successive multiplications and divisions [3]. Others compute conventional multiplications and truncations. All of these methods can have some advantages within some specific modular arithmetic environments, but in the case of the

ED&TC '97 on CD-ROM

evaluation of MEs, the Montgomery's algorithm [8] seems to achieve the best results.

The advantages of the Montgomery's method are less area requirements, high performances and a simpler and faster selection logic (which can be pipelined) [5]. Even though, it also has some drawbacks. First, the calculation of a constant is necessary, increasing the whole encryption/decryption time. Second, two extra MMs are needed (to enter into and exit from what is called the Montgomery residue space) to perform a single ME. However, the effect of these additional multiplications is not significative when considering the global computation time, because of the large size of the exponent in the applications that are being considered.

2.1. Montgomery's Algorithm for Modular Multiplication

Though a deep study of the Montgomery's algorithm is out of the scope of this paper, a brief description is made for the sake of completeness. This will allow a better discussion on the different compromise factors that led us to the final design. Relevant references for a better understanding of the Montgomery's algorithm are [8], [2] and [3].

The basic form of the Montgomery's algorithm is:

```
R := 0;
FOR i := 0 TO (n - 1) DO
BEGIN
        Q_i := ((R_0 + A_i * B_0 ) * (r - M_0)^-1) mod r;
        R := (R + A_i * B + Q_i * M) div r;
END
IF (R > M) R := R - M;
```

where ‘R’ is a iteration register and the place where the final operation result is stored; ’M’ is the modulus, ‘A’ and ‘B’ the words that have to be multiplied, ‘r‘ the radix used in the algorithm, ‘n’ the number of digits, radix ‘r’, of the modulus, and ‘Q’ the quotient of the operation. The variables with subscripts show which digit, radix ‘r’, is being processed in every iteration. In this way, ‘A_i’ is the digit ‘i’, radix ‘r’, of the number ‘A’. ‘X^{-1} *mod* r‘ is the multiplicative inverse of ‘X’ ($X\ X^{-1}$ *mod* $r \equiv 1$). For a good understanding of the different hardware implementations is important to point out that the first operation in every iteration is made with words of few bits (it is a truncation), and the second one involves the addition of very long words (>512 bits) and a shift of the result.

2.2. System Design Issues

As shown before, the MM is the key operation in cryptography applications. In this section, different techniques to improve the implementation of the MM are considered and the corresponding tradeoffs are analyzed. As a result, important improvements in the MM algorithm can be obtained. Finally, a modified Montgomery’s algorithm that incorporates some of these methods is presented.

Operands pre-scaling [2] [6] can be used to improve the algorithm behavior. If the multiplicand is scaled with the square of the radix, its lowest two digits become zero, simplifying the computation of the quotient digit, ‘Q’. This is of special importance when using a radix greater or equal than four because this is the critical path which limits the system clock. The cost of this approach is that two more algorithm iterations are needed.

Therefore, the Montgomery’s algorithm can be modified as follows:

```
R := 0;
Q_0 := 0;
B := B * 16;
FOR i := 0 TO (n + 1) DO
BEGIN
        R := (R + A_i * B + Q_i * M) div 4;
        Q_i+1 :=(R_0 * (4 - M_0)^-1) mod 4;
END
IF (R > M) R := R - M;
```

Most of the specific high performance hardware for MM use ALUs as wide as the word to be processed. Due to the length needed in the applications we are considering (more than 512 bits), every time an addition is performed the carry propagation introduces a huge delay in the final computation time. The use of a Carry Save Representation (CSR) reduces this delay dramatically, speeding up the multiplication process. An extended CSR [6] can also be considered, but it imposes a larger area, with a more complex conversion to conventional binary representation.

These techniques have a drawback: the need of conversion to binary representation at the end of the multiplication process. This involves again a long carry propagation time. Nevertheless, the mean value of the carry propagation chain, given by the expression ‘$\log_2 n$’, where ‘*n*’ is the operands word length, is 10 for n = 1024 (the longest propagation carry chain is 1024). As it will be shown later, this allows the use of some specific techniques for the carry propagation control that make the whole conversion process faster. Note that the conversion might also be performed at the end of the exponentiation operation, but this increases the hardware complexity.

The choice of the radix value is very important to obtain good performance while keeping an acceptable area. The higher the radix is, the faster the algorithm runs (less iterations). But this does not necessary mean that the computation is faster, since the clock frequency decreases due to the larger multiplier size.

Finally, the exponentiation process can evaluate first either the MSB or the LSB of the exponent. Starting with the former [5], a register can be eliminated, but it involves a more complex logic control.

After this analysis, and considering the different area-speed tradeoffs, the following design decisions have been made:

- Radix 4 (r=4).
- Two´s complement representation for B and M.
- Signed-digit representation for A and Q.
- Multiplicand (B) scaling.
- CSR for R.
- Conversion from redundant CSR to non redundant binary representation every MM.

Under these conditions, if 'M *mod* 4 ≡ 3', $(4\text{-}M_0)^{-1}$ can be replaced by '1'. Otherwise, it can be replaced by '-1'.

At the end of the algorithm, the redundant codification of 'R' and 'A_i' is assumed. The signed-digit representation of 'A_i' and 'Q_i' (both are codified in a range between {-1, 2}) makes much simpler the logic that performs the multiplication by 'B' and 'M', respectively. A CSR for 'R' avoids the carry propagation in every iteration. The conversion from redundant representation to two´s complement is performed by means of an adder with asynchronous carry control. This method improves the carry propagation time.

More details about the algorithm transformation can be found in [8], [2] and [3].

3. The Hardware Architecture

The system has been designed to work as a coprocessor of a generic CPU. Two registers, the status register and the command register, perform the communication between the CPU and the chip. The data to be processed are stored in an intermediate memory. Both, the CPU and the internal control block can access this memory. The data transactions are not a bottleneck in the ASIC functionality. Therefore, no DMA support is necessary. Next, the main blocks the system is composed of are described as well as the complete set of instructions the ASIC supports.

3.1. The Datapath

The datapath width is 768 bits, but can work with data of smaller word length. The *working* datapath width is controlled by a specific instruction that sets the operands length. The format of data is two's complemented.

The user can access three registers of the datapath. The width of each register is also 768 bits.

Internally, the datapath has been designed with slices of 64 bits. There are three kinds of slices: one for the most significant bits, another for the least significant bits, and the last one for the intermediate bits. Each block has its own peculiarities. The slice for most significant bit has to control the sign extension, the overflow, etc. The slice for least significant bit performs the quotient calculation. In fact, both of these blocks are modifications of the intermediate slice. The datapath width can be easily expanded in blocks of 64 bits inserting an adequate number of intermediate blocks. Figure 2 shows the datapath slice interconnection.

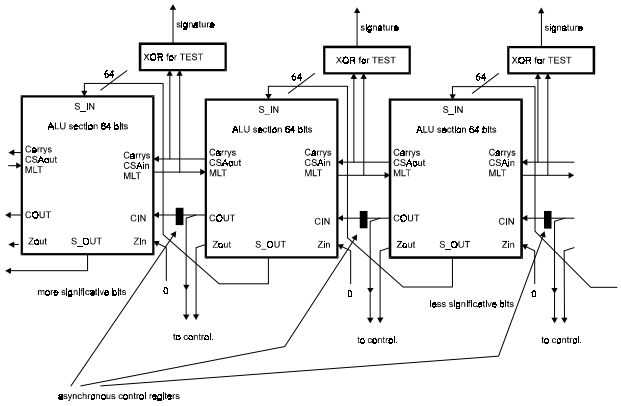

Figure 2

Every section has an input and an output data bus interface and is connected to the others by means of a 64 bit shift register. There are some additional connections between blocks to keep the carry chains of the internal adders.

The core of the datapath is the Montgomery multiplier. The internal structure of this multiplier is shown in the Figure 3

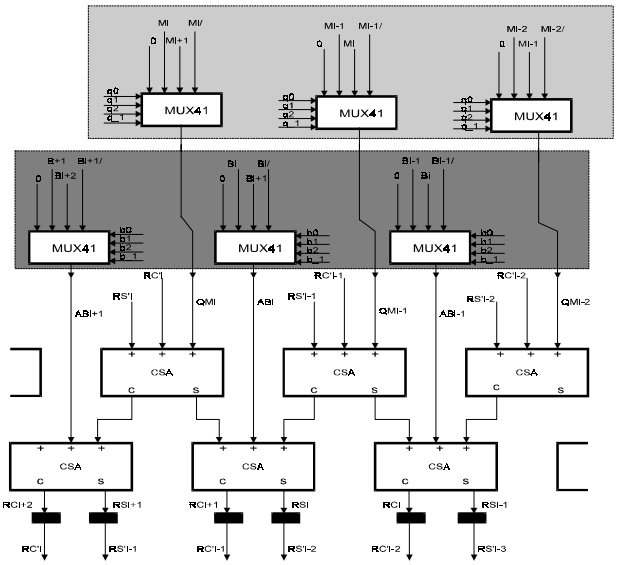

Figure 3

The components of the dark gray zone implement the multiplier of the product a_i*B. In the light gray zone are the components of the product q_i*M. Below these multipliers there are two stages of Carry Save Adders, whose results are stored in the iteration register ´R´. These bits are shifted and fed back to the Montgomery multiplier.

From the user point of view, the datapath can be depicted as shown in figure 4.

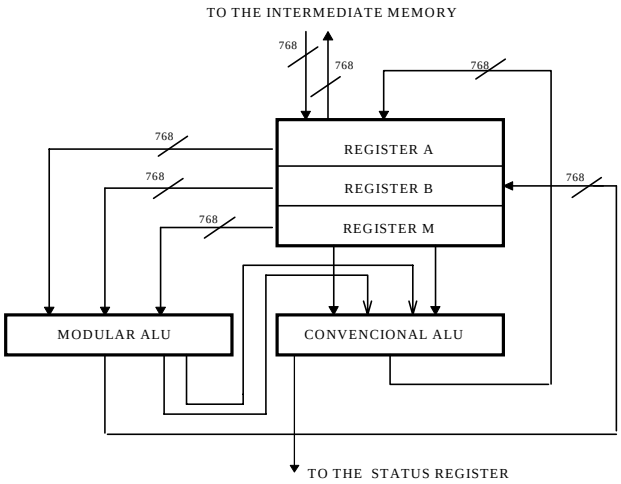

Figure 4

The datapath supports two types of operations: modular operations (MO) (multiplication and exponentiation), and conventional operations (CO) (addition, subtraction, shift, etc.). In a MO, each register has its specific role. They are non-interchangeable. In a CO, all of them are equivalents. They can be either operand source or store register.

Comparatively, the COs are slower than the MOs. Neither specific hardware nor a specific codification is used to speed COs up. Nevertheless, they are always executed faster than in a conventional CPU.

3.2. The Memory

The memory is single-port and is able to store 512 words of 16 bits. It can be accessed from either the CPU or the control block, sending or loading data to or from the datapath registers. This memory acts as a buffer between the CPU and the datapath, and is the only way to access the internal registers. The control avoids the collisions in accessing the memory. If there is a data contention in the memory access, the datapath has priority over the CPU. The memory can store up to eight datapath registers.

3.3. The Control Block

The control is implemented as an one-hot encoded Finite State Machine. All the outputs of this block are registered except the datapath selection signals that are decoded to avoid collisions during test.

It must be observed that the sequence of operations needed to perform a complex cryptographic operation (RSA cryptography, primality test, keycode generation, etc.) has to be ordered by the master CPU, since no internal sequencing capability has been implemented. Note that the CPU can always make some decisions regarding the program execution just reading the available status register.

3.4. The Input-Output Interface

The ASIC uses an standard input-output interface, with a 16 bit bi-directional data bus, a 9 bit address bus, CS_ , WR_ and RD_ low level active signals, data/command selection signal, low level active asynchronous reset, busy flag that notifies the CPU when the ASIC is running.

4. The Set of Instructions

This coprocessor supports the necessary set of instructions to perform complex cryptography applications, such as DSA and RSA encryption/decryption, primality test, keycode generation, etc. Next, these operations are briefly described:

MOVTOREG, MOVTOMEM: They load the datapath registers with data from the intermediate memory and store data from the registers into the memory.
ADD, SUB: These instructions add or subtract two datapath registers. The effective operands length is 768 bits.
MOV: It moves data directly from one register to another.
CMP: It compares two registers, modifying the status register.
SHITL, SHITR: They shift the datapath register left or right. The instruction includes a bit that is inserted and the most/lest significant bit is stored in the status register.
MULM: It performs a Montgomery multiplication. The maximum word length in MOs is 762 bits.
EXPM: A Montgomery's modular exponentiation is evaluated. It takes the base from a register and a 64-bit word from the memory is used as the exponent.
EXPS: It is used to evaluate Montgomery's modular exponentiations with exponents larger than 64 bits.
SETL: It sets the modulus word length.
CONF: This instruction configures the chip (number of memory wait states, number of wait states of a datapath slice, etc.). Configuration data are closely related with the working clock rate.

5. The Design Methodology

The design methodology that has been used has proved to be useful in many aspects. It is worthwhile mentioning the following points: a) The evaluation of the different algorithms and verification of the results was made with *Mathematica* [7], (Wolfram Research Inc) that can operate with numbers with an arbitrary precision. This program generated the files that were used for automatic simulation and verification along the whole design process.

b) The functional simulation was used to analyze the different problems that appear when working with redundant representations. Also, the impact of other design decisions

	Year	Tech. (m)	Modulus width (bits)	Clock (MHz)	Area (mm^2)	Baudrate (Kbits/sec)
Software Implementation (Sparc10 & lib. Big Num)	1996		512			0.0017
British Telecom	1988	2.5	256	10		10.2
Sandia	1989	2.0	512	8		10.0
Philips	1989	1.2	512	16		2.0
National Tsing Hua University [1]	1995	0.8	512	50	76	24.3
Our design	1996	0.7	768	50	77	72.5

Table 1

could be evaluated. In this stage a first area estimation was made. A redesign of the internal datapath structure could be considered at this point. All the tests were made with a model of 192 bits to avoid huge simulation times. The characteristics of the design made it possible to derive the final chip area and performance (with a word length of 768 bits).

c) To generate the datapath, the '*ES2 Datapath-Compiler'* was used. The interconnections between the different slices were laid out very carefully to achieve a very compact design.

d) The rest of the design was specified with '*Verilog'* and synthesized with Synergy, a tool included in the '*Design-Framework II'* from Cadence.

6. Test

The test has been implemented following two strategies. On one hand, the automatic synthesized blocks (control, auxiliary registers, etc.) incorporate a scan path chain. On the other hand, the blocks that have been generated with the ES2 Datapath Compiler could not be tested with a scan path. Thus, an '*xor tree*' has been inserted between adjacent slices. The algorithm propagates all the possible errors from the MSB to the LSB very fast. The test has been carried out interleaving functional test and scan path chains.

7. Results

The chip has been manufactured with ES2 0.7 μm CMOS technology. Its area is 77 mm^2 and works at 50 MHz (measured clock frequency). A Montgomery's modular multiplication with a 762-bit modulus takes 400 clock cycles (8 microseconds aprox.). A Montgomery's modular exponentiation with an 768-bit exponent (assuming equal number of 0's than 1's) takes 500,000 clock cycles (10 milliseconds).

It is very difficult to carry out a comparative study with other similar designs, since usually there is not enough available information about how the different tests have been performed (i.e., the exact key length, how many 1's the key has, ...). Table 1 shows some comparative results, though.

The chip that has been presented here is faster than others that are also based on standard cells, and presents some additional features, such modular multiplication, addition, etc. with a very low extra area requirements.

8. Conclusions

An ASIC with a powerful set of instructions to perform complex cryptographic operations has been presented. A variety of design issues have been thoroughly studied, analyzing the different compromise factors. A final implementation based on a careful selection among the possible design choices has been discussed. The hardware architecture has been described and the resulting design features have been finally highlighted.

9. Acknowledgments.

This work has been funded by PENTA 3, SA through the ESPRIT project GAME.

We would also like to thank Carlos Santos for his valuable collaboration and Manuel Serna for his patient simulation work.

10. References

[1] P. S. Chen, S. A. Hwang, and C. W. Wu, *"A systolic RSA Public Key Cryptosystem"*, Proc. of ISCAS, 1995.

[2] S. E. Eldridge, C. D. Walter, *Hardware Implementation of Montgomery´s Modular Multiplication Algorithm.* IEEE Transactions on Computers, vol. 42, no. 6, pp. 693-699, Jun. 1993.

[3] P. Kornerup, *High-Radix modular Multiplication for Cryptosystems*. Proceedings 11th Symposium on Computer Arithmetic, pp. 277-283, Jun.1993.

[4] R. L. Rivest, A. Shamir, and L. Adleman, *"A method for obtaining digital signatures and public-key cryptosystems"*, Communications of the ACM, vol. 21, no. 2, pp. 120-126, Feb. 1978

[5] M. Shand, J. Vuillemin, *Fast Implementations of RSA Cryptography*. Proceedings 11th Symposium on Computer Arithmetic, pp. 252-259, Jun. 1993.

[6] C. D. Walter, *Still faster modular multiplication*. Electronics Letters, vol. 31, no. 4, pp. 263-264, Feb. 1995.

[7] S. Wolfram, *Mathematica, a System for doing Mathematics by Computer*, Addison-Wesley 1991.

[8] P. L. Montgomery. *Modular Multiplication Without Trial Division*. Mathematics of computation, 44 (170): 519: 521, April 1985.

[9] D. E. Knuth, *'The art of computer programming'*, vol. 2: Seminumerical algorithms. Addison-Wesley, 1981.