A new serial/parallel architecture for a low power modular multiplier *

JOHANN GROßSCHÄDL
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, A–8010 Graz, Austira
Phone: +43 316 8735518, Email: Johann.Groszschaedl@iaik.at

Key words: Public key cryptography, RSA algorithm, smartcard crypto coprocessor, modular arithmetic, serial/parallel multiplier architecture, low power VLSI design, half bit–level pipelining, full custom design methodology

Abstract: A new architecture for a modular multiplier is introduced that allows the efficient implementation of a low–power crypto coprocessor for smartcards. The multiplier architecture is optimized for accelerating asymmetric cryptographic operations based on long integer modular arithmetic. A modular multiplication is performed in a serial/parallel manner, this means the multiplier is scheduled sequentially (bit by bit) and the multiplicand is scheduled fully parallel. The modular reduction operation is integrated within the multiplication by continued modulus subtraction. It is shown that the serial/parallel architecture can be realized very easily in a full custom design methodology due to its high degree of regularity. For a 1024 bit modular multiplication the proposed multiplier requires about 1600 clock cycles, which allows to compute a 1024 bit RSA exponentiation in approx. 250 ms if the clock frequency is 10 MHz.

1. INTRODUCTION

Smartcards are an important component in many public key infrastructures because they can provide tamper–resistant storage for private keys and ensure correct implementation of security–critical computations like asymmetric decryption or generation of digital signatures. Many popular

* This work has been funded by the Austrian Science Foundation (FWF) as part of the project No. 12596–INF "Hochgeschwindigkeits–Langzahlen–Multiplizierer–Chip".

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35515-3_53

S. Qing et al. (eds.), *Information Security for Global Information Infrastructures*

public key algorithms like RSA algorithm [RSA78], DSA [Nat94] or Diffie–Hellman [DH76] cause very high computational cost as they involve long integer modular exponentiation. But most of today's smartcards include only 8 or 16 bit microcontrollers, which are too slow for implementing RSA algorithm in software. Therefore, a hardware accelerator for the cryptographic operations is necessary to reach acceptable computation speed.

Important smartcard manufacturers such as SGS–Thomson, Philips or Siemens offer smartcard chips with integrated crypto coprocessors [Han98]. Most of these coprocessors allow RSA calculations with a modulus length of up to 1024 bit. The typical computation time for a 1024 bit RSA exponentiation is approximately 800 ms if the crypto coprocessor is clocked with 5 MHz (e.g. Siemens SLE66CX160S or Philips P83W858 [Phi99]), so it can be estimated that a single 1024 bit modular multiplication would require (at most) 2600 clock cycles.

For future developments of crypto coprocessors two design goals are significant: they have to become faster and should consume less power. Computation speed is very important for high security applications with modulus lengths up to 2048 bit, and for the implementation of complex protocols like the SET protocol [SET97]. On the other side, low power consumption is a major design goal for contactless smartcards and for smartcards used in mobile, battery powered devices (e.g. GSM phone for direct banking or secure wireless commerce).

Unfortunately high calculation performance is in contradiction to low power consumption. This is because the two most relevant options to reach higher computation speed, which are increasing the clock frequency and scaling up the degree of parallelism in the multiplier core, will also cause higher power consumption. For the development of new multiplier architectures it is therefore extremely important to reach the required computation performance with the lowest possible clock frequency and a small degree of parallelism in the multiplier core.

This paper presents a serial/parallel algorithm and multiplier architecture for modular arithmetic, which needs only 1600 clock cycles for a single 1024 bit modular multiplication. Even with a low clock frequency of 5 MHz a full 1024 bit RSA operation just needs 500 ms. The modular reduction is realized during multiplication by iterated modulus subtraction, thus the multiplier core has to implement only three simple operations: left–shift, addition and subtraction. Another significant advantage of the proposed serial/parallel architecture is that it results in a very regular layout and therefore it is well suited for implementation in a full custom design methodology. Beside the multiplier core, only four registers are necessary to store all operands and intermediate results needed for a modular exponentiation. During the execution of an exponentiation no data transfers from or to the smartcard memory are required.

In section 2 the basic algorithmic concepts for the modular multiplication and redundant number estimations are explained. Section 3 covers the multiplier architecture and provides information how a modular multiplication is executed. Additionally, the multiplication performance is estimated in this section and also some concrete implementation aspects like pipelining are discussed. The paper finishes with conclusions in section 4.

2. MODULAR MULTIPLICATION ALGORITHM

When applying the *binary exponentiation method* (also known as the *square and multiply algorithm*), a modular exponentiation is performed by successive modular multiplications [Knu69]. Modular reduction after each multiplication step avoids the exponential growth in size of intermediate results. The binary exponentiation method needs $3n/2$ modulo multiplications for an n bit exponent, assuming the exponent is composed of half "0" and half "1". The efficient implementation of the modular multiplication is therefore the key to high performance. Beside the popular *Montgomery multiplication* [Mon85] and *Barret modular reduction method* [Bar87], for hardware implementation also a serial/parallel algorithm for modular multiplication can be very efficient.

2.1 Shift and add multiplication with integrated modulus subtraction

The usual way of multiplying two numbers A and B is done by scanning the multiplier B from most significant bit (MSB) to least significant bit (LSB) and adding the partial product $A \cdot B[i]$. The notation $X[i]$ always denotes the i–th bit of a number X, $X[0]$ is the LSB, and $X[n-1]$ the MSB. The partial product $A \cdot B[i]$ is either 0 (if $B[i]$ is 0) or the multiplicand A (if $B[i]$ is 1). After each partial product addition the intermediate result must be shifted one position to the left to align it for the next partial product, which is equal to an arithmetic multiplication by 2. This method is known as *shift and add multiplication*.

Figure 1 shows how the simple shift and add multiplication can be expanded to perform a modular multiplication. The modular reduction of the intermediate result is realized by subtraction of the product $q \cdot M$, whereby q is the quotient of R and M:

$$q \quad \text{with} \quad q \quad \{0,1,2 \tag{1}$$

```
Input:
Modulus M, 2^n-1 <= M < 2^n
Multiplicand A, 0 <= A < M
Multiplier B, 0 <= B < M

Output:
Result R, R = A*B mod M

Algorithm:
i = n - 1
R = 0
WHILE i >= 0 DO
  R = 2*R + A*B[i]
  IF R >= 2*M THEN
    R = R - 2*M
  ELSE IF R >= M THEN
    R = R - M
  ENDIF
  i = i - 1
ENDWHILE
```

Figure 1: Simple shift and add multiplication with integrated modular reduction.

The quotient q can be at most 2 since the term $2 \cdot R + A \cdot B[i]$ is always smaller than three times the modulus M. For hardware implementation, the subtraction of M or $2 \cdot M$ can be realized very easily by two's complement addition. But this simple algorithm has also two serious disadvantages for implementation in hardware:

1. Addition of long integers causes significant carry propagation from LSB to MSB.
2. The comparison of the intermediate result with the modulus to decide whether the quotient q is 0, 1 or 2 is also very difficult to realize in hardware.

The carry propagation is easily eliminated by the implementation of *carry save adders* (CSA). A n bit CSA consists of n fulladders, and solves the carry propagation problem by using a redundant representation for the result. The basic principle of the carry save addition is to reduce the sum of three binary numbers X, Y, A to the sum of two binary numbers S, C without carry propagation according to the following formulas:

$$S[i] \quad X[i] \quad Y[i] \quad A[i] \tag{2}$$

$$C[i \quad 1] \quad X[i] \; Y[i] \quad X[i] \; A[i] \quad Y[i] \; A[i] \text{ with } C[0] \quad 0 \tag{3}$$

Note that the operators in the above formulas are logic operators and not arithmetic operators. When using CSAs, the intermediate result R is not a single binary number, it is given in a *redundant representation* as $(Rs+Rc)$ instead, whereby Rs denotes the sum part of the result, and Rc the carry part respectively. This is because a CSA has always two outputs: S and C.

The second problem is the calculation of the quotient q, which is the same as to decide whether the actual intermediate result is smaller than M ($q = 0$), or bigger than M ($q = 1$), or bigger than $2 \cdot M$ ($q = 2$). This decision is difficult for long integers, because a n bit modulus M can vary between its minimum value M_{min} of 2^{n-1} and its maximum value M_{max} of $2^n - 1$:

$$2^{n \; 1} \delta M \quad 2^n \qquad M_{min} \quad 2^{n \; 1}, M_{max} \quad 2^n \quad 1 \tag{4}$$

```
Input:
Modulus M, 2^n-1 <= M < 2^n
Multiplicand A, 0 <= A < M
Multiplier B, 0 <= B < M

Output:
Result Rs+Rc (redundant representation),
(Rs+Rc) = (A*B mod M)+k*M with k {0,1,2}

Algorithm:
i = n - 1
(Rs+Rc) = 0
WHILE i >= 0 DO
  (Rs+Rc) = 2*(Rs+Rc) + A*B[i]
  WHILE (Rs+Rc) >= 2*2^n DO
    (Rs+Rc) = (Rs+Rc) - 2*M
  ENDWHILE
  WHILE NOT (Rs+Rc) < 3*2^n-1 DO
    (Rs+Rc) = (Rs+Rc) - M
  ENDWHILE
  i = i - 1
ENDWHILE
```

Figure 2: Modified version of the shift and add multiplication with integrated modular reduction.

Additionally, the redundant representation of the intermediate result does not make this task easier. An efficient solution for this problem is to compare the redundant intermediate result to $3 \cdot M_{min}$ and $2 \cdot M_{max}$ instead of the exact value of M and $2 \cdot M$, since this comparison is easy to implement in hardware, as will be demonstrated in section 2.2.

Figure 2 shows a modified version of the shift and add multiplication, which is optimized for hardware implementation. The intermediate result is written in redundant representation (Rs+Rc) to symbolize that the additions and subtractions are especially suited to be performed by carry save adders.

Since in the modified algorithm the intermediate result (Rs+Rc) is compared to $2 \cdot 2^n$ ($2 \cdot M_{max}$) and $3 \cdot 2^{n-1}$ ($3 \cdot M_{min}$) instead of $2 \cdot M$ and M, it may happen that the intermediate result is not always fully reduced. But the algorithm guarantees that before the next multiplier bit $B[i]$ is processed, the intermediate result is always smaller than three times the modulus. This is true for every modulus M which satisfies Equation (4), even for $M = M_{min}$.

Another interesting detail of the modified algorithm is the fact that the modular reduction is not carried out "at once", but it is splitted into continued subtractions of $2 \cdot M$ and M, until the intermediate result is smaller than $3 \cdot 2^{n-1}$ ($3 \cdot M_{min}$). Of course this raises the question how many subtractions of $2 \cdot M$ and M will be (at most) necessary. Because it is guaranteed that the intermediate result (Rs+Rc) is smaller than $3 \cdot M$ before $2 \cdot (Rs+Rc) + A \cdot B[i]$ is calculated, the product $2 \cdot (Rs+Rc)$ is smaller than $6 \cdot M$, and since the partial product $A \cdot B[i]$ is always smaller than M it is proven that the intermediate result is smaller than $7 \cdot M$ before beginning the modular reduction. Thus at most two subtractions of $2 \cdot M$ or M are necessary until (Rs+Rc) is smaller than $3 \cdot M$. After the modular multiplication is finished, a final reduction may be necessary to get the exact result of $A \cdot B$ mod M. This final reduction is simply done by subtracting once or twice the modulus.

2.2 Redundant number estimations

The algorithm in Figure 2 applies redundant number estimations to decide whether or not a subtraction of $2 \cdot M$ and/or M has to be performed. For hardware implementation this is a significant improvement of the first algorithm because it avoids the necessity for an exact comparison between the intermediate result and the modulus.

$Rs[n]$	$Rc[n]$	$Rs[n-1]$	$Rc[n-1]$	$Rs+Rc$	Estimation
0	0	0	0	$Rs+Rc < 2^{n-1}+2^{n-1}$	$Rs+Rc < 3 \cdot 2^{n-1}$
0	0	0	1	$Rs+Rc < 2^{n-1}+2^{n}$	
0	0	1	0	$Rs+Rc < 2^{n}+2^{n-1}$	
0	0	1	1	$Rs+Rc \tau\ 2^{n-1}+2^{n-1}$	$Rs+Rc \tau\ 2^{n}$
0	1	X	X	$Rs+Rc \tau\ 2^{n}$	
1	0	X	X	$Rs+Rc \tau\ 2^{n}$	
1	1	X	X	$Rs+Rc \tau\ 2^{n}+2^{n}$	$Rs+Rc \tau\ 2 \cdot 2^{n}$

Figure 3: Redundant number estimations. Rs and Rc are both $n+1$ digit numbers, $Rs[n]$ is the MSB of Rs, $Rc[n]$ is the MSB of Rc.

Figure 3 shows a simplified truth table to decide the two inequations $(Rs+Rc) < 3 \cdot 2^{n-1}$ and $(Rs+Rc) \tau\ 2 \cdot 2^{n}$. For decision of the first inequation only the two most significant bits of Rs and Rc need to be scanned, and for the second inequation only the MSB of Rs and Rc. Note that Rs and Rc are both $n+1$ bit numbers, therefore the MSB of Rs is $Rs[n]$ and the MSB of Rc is $Rc[n]$. The hardware to decide the multiple of the modulus to subtract can be realized very easily by the following two simple boolean functions:

$$sub1 = Rs[n]+Rc[n]+(Rs[n-1] \cdot Rc[n-1]) \quad (5)$$

$$sub2 = Rs[n] \cdot Rc[n] \quad (6)$$

If $sub1 = 0$ then the intermediate result is smaller than $3 \cdot 2^{n-1}$ and thus smaller than $3 \cdot M$. This estimation is correct for every value of M according to Equation (4), even for $M = M_{min}$. On the other side, if $sub1 = 1$, the intermediate result is bigger than 2^{n}, and so it can be estimated to be also bigger than M. Therefore at least one subtraction of M is necessary, even if $M = M_{max}$. For every n bit modulus M with $2^{n-1} \delta\ M < 2^{n}$ the redundant number estimations of the algorithm can be summarized as follows:

$$sub1 = 0 \text{ and } sub2 = 0 \qquad Rs+Rc < 3 \cdot M \quad (7)$$

$$sub1 = 1 \text{ and } sub2 = 0 \qquad Rs+Rc > M \quad (8)$$

$$sub2 = 1 \qquad Rs+Rc > 2 \cdot M \quad (9)$$

3. MULTIPLIER ARCHITECTURE

The architecture of the multiplier core is in principle a direct mapping of the algorithm. Therefore, the multiplier core has to perform only three simple tasks: addition of the partial product $A \cdot B[i]$ to the last intermediate result, subtraction(s) of $2 \cdot M$ and/or M, and shifting left the actual intermediate result to align it for the next partial product.

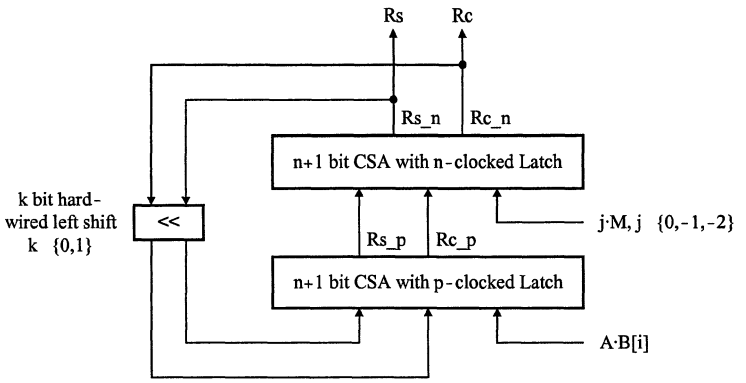

Figure 4: Block diagram of the multiplier core.

Figure 4 illustrates the block diagram of the multiplier core. Its main components are two n+1 bit carry save adders. The first CSA at the bottom performs the addition of the partial product to the actual intermediate result. The sum output *Rs_p* and the carry output *Rc_p* of the first CSA are used to estimate the multiple of the modulus to become subtracted in the second (upper) CSA. This estimation is performed as described in section 2.2, and only the two highest order bits of *Rs_p* and *Rc_p* are needed to implement the logic functions of Equation (5) and (6). Therefore, the hardware to decide whether to subtract $0 \cdot M$, $1 \cdot M$ or $2 \cdot M$ can be implemented very efficiently and will not cause a long critical path in the multiplier core.

3.1 Execution of a single modular multiplication

The subtraction of M or $2 \cdot M$ is realized by addition of the two's complement of M or $2 \cdot M$ to the output of the first CSA, which takes place in the upper CSA. But one subtraction of $2 \cdot M$ or M might not be enough to guarantee that the intermediate result is smaller than three times the modulus. Therefore Equation (5) is applied again to decide whether an extra subtraction of M or $2 \cdot M$ is necessary or not. If an extra subtraction is required, the outputs *Rs_n* and *Rc_n* of the upper CSA are lead back to the first CSA without left–shift. For an extra subtraction, the multiplier bit $B[i]$ must be masked

out, so that no partial product (i.e. zero) is added in the first CSA. After that, the extra subtraction of M or $2 \cdot M$ takes place again in the upper CSA.

If no extra subtraction is required, the processing of the multiplier bit is finished. The outputs of the upper CSA are lead back to the inputs of the bottom CSA with an one bit hardwired left–shift. *Rs_p* and *Rc_p* are now correctly aligned for addition of the next partial product and the same procedure starts again. After the last multiplier bit has been processed, the sum and carry of the upper CSA represent the redundant result (*Rs+Rc*) of the modular multiplication.

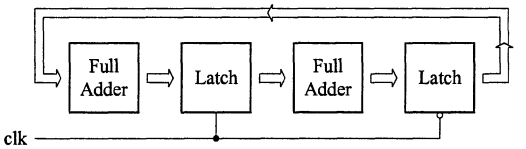

Figure 5: Half bit–level pipelining.

For concrete implementation of the multiplier core, it would be advantageous to separate the two carry save adders by storage elements (flip flops or latches) in order to decrease the critical path length. This will result in a pipelined datapath structure as illustrated in Figure 5. Using latches as storage elements has also a positive effect on the overall computation time, because the addition of the partial product and the first subtraction of M or $2 \cdot M$ are performed within one clock cycle. An extra modulus subtraction requires an additional clock cycle, so that the processing of a single multiplier bit needs at most two clock cycles.

3.2 Execution of a modular exponentiation

Figure 6 presents the complete serial/parallel modular multiplier, consisting of the multiplier core, four n+1 bit registers and pipelined w–bit carry look ahead adder (CLA), whereby w denotes the word–width of the registers. The *Data register* is loaded at the beginning of the exponentiation with the block of data to be de/encrypted. The *Modulus register* is needed to store the modulus. *Multiplicand* and *Multiplier register* are used to store the actual operands of the modular multiplication. Both registers can carry out w–bit shift operations in LSB direction, register Multiplier can additionally perform one bit shifts in MSB direction (1 bit left–shifts). All four registers are connected by a n+1 bit bus and are able to perform parallel register transfers.

At the beginning of a modular exponentiation, the contents of register Data is loaded into register Multiplicand and register Multiplier (the first operation of the binary exponentiation method is a square step).

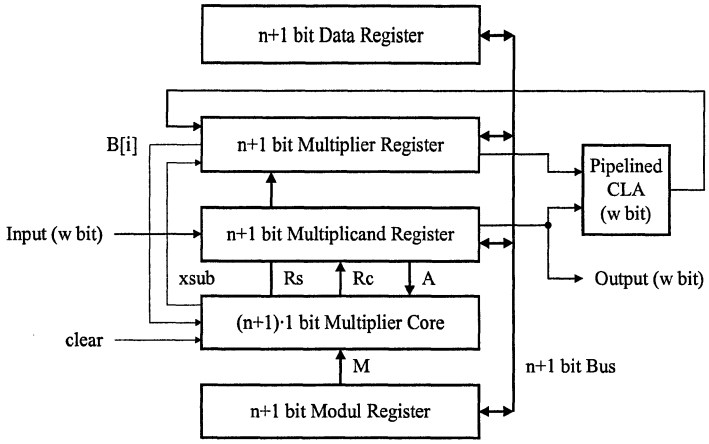

Figure 6: Serial/parallel modular multiplier.

The Multiplier register is shifted bit by bit in MSB direction to deliver the multiplier bits $B[n-1]$ to $B[0]$ to the multiplier core. A modular multiplication now works as described in section 3.1. The control signal *xsub* is generated from the two higher order bits of *Rs_n* and *Rc_n* according to Formula (5). Whenever *xsub* is 1, the multiplier core has to perform an extra subtraction and register Multiplier must stop the left–shift until $xsub = 0$.

After the least significant multiplier bit $B[0]$ has been processed, the redundant result ($Rs+Rc$) is loaded into register Multiplier and Multiplicand. Note that the old multiplier and multiplicand are not needed any more. Now the redundant result must be converted into binary representation. This is done by the pipelined CLA and requires n/w clock cycles plus the delay of the CLA (usually $\mathrm{ld}(w)$ clock cycles). The output of the CLA is lead back to the Multiplier register. Since the binary result may not be fully reduced, at most two additional subtractions of the modulus and redundant to binary conversions may be necessary. After the modular multiplication has finished, the result resides within register Multiplier, where it might act as multiplier for the next modular multiplication. Register Multiplicand has to be loaded either from register Multiplier (square) or register Data (multiply).

3.3 Computation performance

When performing a modular multiplication, each multiplier bit requires at least one clock cycle and at most two clock cycles to be processed. So it can be estimated an average number of 1.5 clock cycles per multiplier bit $B[i]$. Additionally, between one and at most three redundant to binary conversions are required for one modular multiplication, each needs $n/w + \mathrm{ld}(w)$ clock cycles. For a $(n+1)\cdot 1$ bit multiplier core and a w bit CLA, the total number of clock cycles for a single modular multiplication can be estimated as follows:

$$c \mid 1.5 \cdot n + 2 \cdot (n/w + \mathrm{ld}(w)) \qquad (10)$$

For $n = 1024$ and $w = 32$, a single modular multiplication requires an average number of approximately 1600 clock cycles. Assuming that a 1024 bit modular exponentiation is calculated by 1536 modular multiplications, a 1024 bit exponentiation needs approximately $2.5 \cdot 10^6$ clock cycles.

4. CONCLUSIONS

This paper presented a new serial/parallel algorithm and architecture for a modular multiplier. The modular reduction is realized by continued subtractions of multiples of the modulus. The algorithm is fast and easy to implement, because neither precomputed constants (as in Barret's algorithm) nor correction of the result (as in Montgomery's algorithm) are necessary. Compared to other serial/parallel architectures, the proposed architecture profits from fast and efficient redundant number estimations to decide the multiple of the modulus M to be subtracted, and from easy subtrahend generation, since the subtrahend is either 0, M, or $2 \cdot M$. Furthermore, the presented serial/parallel architecture is very regular, and therefore well suited for implementation in a full custom design methodology.

5. REFERENCES

[Bar87] P. Barrett. *Implementing the RSA public–key encryption algorithm on a standard digital signal processor.* Advances in Cryptology: CRYPTO'86 (A.M. Odlyzko ed.) LNCS 263, Springer Verlag, 1987.

[DH76] W. Diffie and M.E. Hellman. *New Directions in Cryptography.* IEEE Transactions on Information Theory, IT22(6), Nov. 1976.

[Han98] H. Handschuh and P. Paillier. *Smart Card Cryptocoprocessors for Public Key Cryptography.* CARDIS'98, LNCS, Springer Verlag, 1998.

[Knu69] D.E. Knuth. *Seminumerical Algorithms*, Vol. 2 of The Art of Computer Programming, Addison–Wesley, 1969.

[Mon85] P. Montgomery. *Modular multiplication without trial division.* Mathematics of Computation, vol44(170), 1985.

[Nat94] National Institute of Standards and Technology (NIST) FIPS Publication 186: *Digital Signature Standard.* May 1994.

[Phi99] Philips Semiconductors. *"Controllers for High Security, Crypto and Dual Interface Smart Cards"*, 1999. Available for download at http://www.semiconductors.com.

[RSA78] R.L. Rivest, A.Shamir and L. Adleman. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems.* Communications of the ACM, 21(2), February 1978.

[SET97] *SET Secure Electronic Transaction Specification*, May 1997. Available for download at http://www.setco.org/download.html.