The Chinese Remainder Theorem and its Application in a High-Speed RSA Crypto Chip[*]

Johann Großschädl
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
Johann.Groszschaedl@iaik.at

Abstract

The performance of RSA hardware is primarily determined by an efficient implementation of the long integer modular arithmetic and the ability to utilize the Chinese Remainder Theorem (CRT) for the private key operations. This paper presents the multiplier architecture of the RSAγ crypto chip, a high-speed hardware accelerator for long integer modular arithmetic. The RSAγ multiplier datapath is reconfigurable to execute either one 1024 bit modular exponentiation or two 512 bit modular exponentiations in parallel. Another significant characteristic of the multiplier core is its high degree of parallelism. The actual RSAγ prototype contains a 1056∗16 bit word-serial multiplier which is optimized for modular multiplications according to Barret's modular reduction method. The multiplier core is dimensioned for a clock frequency of 200 MHz and requires 227 clock cycles for a single 1024 bit modular multiplication. Pipelining in the highly parallel long integer unit allows to achieve a decryption rate of 560 kbit/s for a 1024 bit exponent. In CRT-mode, the multiplier executes two 512 bit modular exponentiations in parallel, which increases the decryption rate by a factor of 3.5 to almost 2 Mbit/s.

1. Introduction

Many popular crypto-systems like the RSA encryption scheme [RSA78], the Diffie-Hellman (DH) key agreement scheme [DH76], or the Digital Signature Algorithm (DSA) [Nat94] are based on long integer modular exponentiation. A major difference between the RSA scheme and crypto-systems based on the discrete logarithm problem is the fact that the modulus used in the RSA encryption scheme is the product of two prime numbers. This allows to utilize the Chinese Remainder Theorem (CRT) in order to speed up the private key operations. From a mathematical point of view, the usage of the CRT for RSA decryption is well known. But for a hardware implementation, a special multiplier architecture is necessary to meet the requirements for efficient CRT-based decryption.

This paper presents the basic algorithmic and architectural concepts of the RSAγ crypto chip, and describes how they were combined to provide optimum performance. A special focus is directed towards the implementation of the CRT-based decryption. The major design goal with the RSAγ was the maximization of performance on several levels, including the implemented hardware algorithms, the multiplier architecture, and the VLSI circuit technique. The RSAγ crypto chip applies the *FastMM algorithm* [MPPS95] for modular multiplication. FastMM algorithm is based on Barret's modular reduction method [Bar87] and performs a modular multiplication by three simple multiplications and one addition. The division-free property of Barret's modular reduction method makes FastMM algorithm very attractive for hardware implementation.

The core of the RSAγ crypto chip is its $1056 * 16$ bit word-serial multiplier, which schedules the multiplicand fully parallel and the multiplier in 16 bit words. As a consequence of this direct processing of long integers, the computation time for an n bit modular exponentiation is dramatically reduced. Furthermore, *Booth recoding* [Mac61] is implemented to halve the number of partial-products. Thus the multiplication speed is almost doubled with low additional hardware effort. In addition, pipelining significantly increases the throughput in RSA en- and decryption. Due to the high degree of parallelism in the multiplier core, the RSAγ crypto chip executes a 1024 bit modular multiplication in 227 clock cycles. The multiplier core is dimensioned for a clock frequency of 200 MHz and designed in a full-

[*]The work described in this paper was supported by the Austrian Science Foundation (FWF) under grant number P12596–INF "Hochgeschwindigkeits-Langzahlen-Multiplizierer-Chip".

custom methodology, which results in a very small silicon area.

By taking advantage of the Chinese Remainder Theorem (CRT), the computational effort of the RSA decryption can be reduced significantly. If the two prime numbers P and Q of the modulus N are known, it is possible to calculate the modular exponentiation $M = C^D \bmod N$ separately $\mathrm{mod} P$ and $\mathrm{mod} Q$ with shorter exponents, as described in [QC82] and [SV93]. Since the length of the exponent is about $n/2$, approximately $3n/4$ modular multiplications are necessary for a single modular exponentiation. The RSAγ crypto chip is able to compute two exponentiations in parallel, as the 1024 bit multiplier core can be split into two 512 bit multipliers. Running the two 512 bit multipliers in parallel allows both CRT related exponentiations to be computed simultaneously. Compared to the non-CRT based RSA decryption performed on an n bit hardware, utilizing the CRT results in a speed-up factor of approximately 3.5.

The rest of the paper is structured as follows: In the next section, the implemented algorithms for exponentiation and modular multiplication are presented. Section 3 focusses on the Chinese Remainder Theorem and explains how it can be used to speed up the RSA decryption. Section 4 presents the architecture of the RSAγ multiplier core and describes the execution of a simple multiplication. In section 5, implementation problems like floorplanning and clock distribution are discussed. This topic is carried on in section 6, and it is detailed how the multiplier architecture was adapted to comply with the requirements for efficient CRT-based RSA decryption. The paper finishes with a summary of results and conclusions in section 7.

2 Implemented algorithms

In order to develop high-speed RSA hardware, we not only need good and efficient algorithms for modular arithmetic, but also a multiplier architecture which has to be optimized for those algorithms. This section presents hardware algorithms for exponentiation, modular reduction and modular multiplication.

2.1 Binary exponentiation method

When applying the binary exponentiation method (also known as *square and multiply algorithm*), a modular exponentiation $C^E \bmod N$ is performed by successive modular multiplications [Knu69]. The MSB to LSB version of the algorithm is frequently preferred against the LSB to MSB one, because the latter requires storage of an additional intermediate result. Modular reduction after each multiplication step avoids the exponential growth in size of the intermediate results. The square and multiply algorithm needs $3n/2$ modular multiplications for an n bit exponent E, assuming the exponent contains roughly 50% ones. Therefore, the efficient implementation of the modular multiplication is the key to high performance.

2.2 Barret's modular reduction method

In 1987, Paul Barret introduced an algorithm for the modulo reduction operation which he used to implement RSA encryption on a digital signal processor [Bar87]. At a first glance, a modular reduction is simply the computation of the remainder of an integer division:

$$Z \bmod N = Z - \left\lfloor \frac{Z}{N} \right\rfloor N = Z - qN \text{ with } q = \left\lfloor \frac{Z}{N} \right\rfloor \quad (1)$$

But, compared to other operations, even to the multiplication, a division is very costly to implement in hardware. Barret's basic idea was to replace the division with a multiplication by a precomputed constant which approximates the inverse of the modulus. Thus the calculation of the exact quotient $q = \left\lfloor \frac{Z}{N} \right\rfloor$ is avoided by computing the quotient $\widetilde{q}$ instead:

$$\widetilde{q} = \left\lfloor \frac{\left\lfloor \frac{Z}{2^{n-1}} \right\rfloor \left\lfloor \frac{2^{2n}}{N} \right\rfloor}{2^{n+1}} \right\rfloor \quad (2)$$

Although equation (2) may look complicated, it can be calculated very efficiently, because the divisions by 2^{n-1} or 2^{n+1}, respectively, are simply performed by truncating the least significant $n-1$ or $n+1$ bits of the operands. The expression $\lfloor 2^{2n}/N \rfloor$ depends on the modulus N only and is constant as long as the modulus does not change. This constant can be precomputed, whereby the modular reduction operation is reduced to two simple multiplications and some operand truncations.

When calculating a modular reduction according to Barret's method, the result may not be fully reduced, but it is always in the range 0 to $3N-1$. Therefore, one or two subtractions of N could be necessary to get the exact result.

2.3 FastMM algorithm

A closer look at Barret's algorithm shows that truncation of operands at $n-1$ or $n+1$ bit borders are necessary. For reasons of regularity, it would be advantageous to apply truncations only at multiples of the word-size w of the multiplier hardware (usually 16 or 32 bit) rather than at the original bit positions. Therefore, we modified Barret's algorithm in order to apply the truncations only at multiples of w, whereby these truncations can be performed by successive w bit right-shift operations. In the modified Barret reduction, denoted as *FastMM algorithm* [MPPS95], the quo-

$$\left.\begin{array}{rl} Z & = A[n+2w-1\,..\,0]\cdot B[n+2w-1\,..\,0] \\ Q & = Z[2n+w-1\,..\,n-w]\cdot N1[n+2w-1\,..\,0] \\ NegR & = Q[2n+4w-1\,..\,n+2w]\cdot NegN[n+2w-1\,..\,0] \\ X & = Z[n+2w-1\,..\,0]+NegR[n+2w-1\,..\,0] \end{array}\right\} X = A\cdot B \bmod N + eN \text{ with } e\in\{0,1\}$$

Figure 1. Modular multiplication according to the FastMM algorithm.

tient $\widetilde{q}$ is calculated as follows:

$$\widetilde{q} \;=\; \left\lfloor \frac{\left\lfloor \frac{Z}{2^{n-w}} \right\rfloor \left\lfloor \frac{2^{2n+w}}{N} \right\rfloor}{2^{n+2w}} \right\rfloor \tag{3}$$

The FastMM algorithm combines multiplication and modified Barret reduction to implement the modular multiplication by three multiplications and one addition according to the formulas shown in figure 1. The values in the squared brackets indicate the bit positions of the operands. All three multiplications and the addition are performed with $n+2w$ significant bits. The FastMM algorithm uses two constants, $N1$ and $NegN$, to calculate the (possibly not fully reduced) result of the modular multiplication. Since these two constants depend on the modulus N only, they can be precomputed according to the following equations:

$$N1 \;=\; \left\lfloor \frac{2^{2n+w}}{N} \right\rfloor \tag{4}$$

$$NegN \;=\; 2^{n+2w} - N \tag{5}$$

Precomputation of $N1$ and $NegN$ has no significant influence on the overall performance if the modulus N changes rarely compared to the data.

The constant $N1$ approximates the exact value of $\frac{2^{2n+w}}{N}$ with limited accuracy, therefore some error eN is introduced when calculating the result X. An exact analysis of the FastMM algorithm according to [Dhe98] shows that the result of the modular multiplication is given as $AB \bmod N + eN$ with $e \in \{0, 1\}$. This means that X might not be fully reduced, but is in the range 0 to $2N-1$, thus the error is at most once the modulus.

When applying the square and multiply algorithm together with FastMM algorithm to calculate a modular exponentiation, no correction of the intermediate results is necessary. Although the intermediate results are not always fully reduced, continuing the exponentiation with an incomplete reduced intermediate result does not cause an error bigger than once the modulus. An exact proof with detailed error estimation can be found in [Dhe98] or [PP89]. If a final modular reduction is necessary after the modular exponentiation has finished, it can be performed by adding $NegN$ to the result. Thus the final reduction requires no additional hardware effort or precomputed constants.

3 Chinese Remainder Theorem

The complexity of the RSA decryption $M = C^D \bmod N$ depends directly on the size of D and N. The decryption exponent D specifies the numbers of modular multiplications necessary to perform the exponentiation, and the modulus N determines the size of the intermediate results. A way of reducing the size of both D and N is to take advantage of properties stated by the Chinese Remainder Theorem (CRT) and Fermat's Little Theorem.

3.1 Mathematical background

In the following, some basic facts and conclusions of the CRT are summarized. This mathematical background knowledge is of elementary importance for the efficient realization of RSA decryption.

Theorem 1 (Chinese Remainder Theorem) *Let the numbers $n_1, n_2, \ldots, n_k$ be positive integers which are relatively prime in pair, i.e. $\gcd(n_i, n_j) = 1$ when $i \neq j$. Furthermore, let $n = n_1 n_2 \ldots n_k$ and let $x_1, x_2, \ldots, x_k$ be integers. Then the system of congruences*

$$\begin{array}{rcl} x & \equiv & x_1 \bmod n_1 \\ x & \equiv & x_2 \bmod n_2 \\ & \vdots & \\ x & \equiv & x_k \bmod n_k \end{array}$$

has a simultaneous solution x to all of the congruences, and any two solutions are congruent to one another modulo n. Furthermore there exists exactly one solution x between 0 and $n-1$.

A constructive proof of the Chinese Remainder Theorem is given in [Kob94]. The unique solution x of the simultaneous congruences satisfying $0 \leq x < n$ can be calculated as

$$\begin{aligned} x &= \left(\sum_{i=1}^{k} x_i r_i s_i \right) \bmod n \\ &= (x_1 r_1 s_1 + x_2 r_2 s_2 + \ldots + x_k r_k s_k) \bmod n \end{aligned} \tag{6}$$

where $r_i = \frac{n}{n_i}$ and $s_i = {r_i}^{-1} \bmod n_i$ for $i = 1, 2, \ldots, k$. In [MVV97], this method is termed as *Gauss's algorithm.*

Corollary 1.1 *If the integers $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \ldots n_k$, then for all integers a, b it is always valid that $a \equiv b \bmod n$ if and only if $a \equiv b \bmod n_i$ for each $i = 1, 2, \ldots, k$.*

As a consequence of the CRT, any positive integer $a < n$ can be uniquely represented as a k-tuple $[\,a_1, a_2, \ldots, a_k\,]$ and vice versa, whereby a_i denotes the residue $a \bmod n_i$ for each $i = 1, 2, \ldots, k$. The conversion of a into the residue system defined by $n_1, n_2, \ldots, n_k$ is simply done by modular reductions $a \bmod n_i$. Conversion back from residue representation to "standard notation" is somewhat more difficult as it requires the calculation stated in equation (6).

Theorem 2 (Fermat's Little Theorem) *Let p be a prime. Any integer a not divisible by p satisfies $a^{p-1} \equiv 1 \bmod p$.*

For a proof refer to [Kob94]. Fermat's little theorem is very useful for calculating the multiplicative inverse of an integer a because $a^{p-2} \equiv a^{-1} \bmod p$.

Corollary 2.1 *If an integer a is not divisible by p and if $n \equiv m \bmod p - 1$, then $a^n \equiv a^m \bmod p$.*

Collorally (2.1) states that when working modulo a prime p, the exponents can be reduced $\mathrm{mod}(p-1)$. This allows to perform the RSA decryption with significantly shorter exponents.

3.2 RSA decryption using the CRT

Since P and Q are primes, any message $M < N = PQ$ is uniquely represented by the tuple $[\,M_P, M_Q\,]$, where $M_P = M \bmod P$ and $M_Q = M \bmod Q$. Therefore, it is also possible to obtain M by computation of M_P, M_Q and recombining them according to equation (6), rather than the usual computation of $M = C^D \bmod N$. By using the corollary (2.1), the size of the exponent can be scaled down:

$$\begin{aligned} M_P &= M \bmod P = \left(C^D \bmod N\right) \bmod P \\ &= C^D \bmod P \ \ (\text{since } N = PQ) \\ &= C^{D \bmod (P-1)} \bmod P \\ &= C^{D_P} \bmod P \ \text{ with } D_P = D \bmod (P-1) \end{aligned} \tag{7}$$

Furthermore, it is easily observed that the ciphertext C can be reduced modulo P before computing M_P, so the lengths of all operands are scaled down by half. With the quantities $C_P = C \bmod P$ and $C_Q = C \bmod Q$, as well as $D_P = D \bmod (P-1)$ and $D_Q = D \bmod (Q-1)$, we get the following equations for M_P and M_Q:

$$M_P = {C_P}^{D_P} \bmod P \text{ and } M_Q = {C_Q}^{D_Q} \bmod Q \tag{8}$$

The recombination of M_P and M_Q to get M can be done according to equation (6). For the special case of $k = 2$, $n_1 = P$, $n_2 = Q$ and $n = N = PQ$, we get $r_1 = N/P = Q$ and $r_2 = N/Q = P$. Moreover, equation (6) can be simplified by using Fermat's little theorem:

$$\begin{aligned} M &= \left(M_P Q \left(Q^{-1} \bmod P\right) + M_Q P \left(P^{-1} \bmod Q\right)\right) \bmod N \\ &= \left(M_P Q \left(Q^{P-2} \bmod P\right) + M_Q P \left(P^{Q-2} \bmod Q\right)\right) \bmod N \\ &= \left(M_P \left(Q^{P-1} \bmod N\right) + M_Q \left(P^{Q-1} \bmod N\right)\right) \bmod N \end{aligned} \tag{9}$$

The last equality in equation (9) comes from the fact that $a\,(b \bmod c) = (ab) \bmod (ac)$ for any nonnegative integers a, b, c. Note that the coefficients $Q^{P-1} \bmod N$ and $P^{Q-1} \bmod N$ are constant and can be precomputed, thereby the effort for the recombination of M_P and M_Q is reduced to two multiplications, one addition and one reduction modulo N.
When assuming that the exponents $D_P = D \bmod (P-1)$ and $D_Q = D \bmod (Q-1)$, as well as the constants which are needed for the recombination $R_P = Q^{P-1} \bmod N$ and $R_Q = P^{Q-1} \bmod N$ have been precomputed, the CRT-based RSA decryption can take place according to the following steps [SV93]:

1. Calculate $C_P = C \bmod P$ and $C_Q = C \bmod Q$.
2. Calculate the exponentiations $M_P = {C_P}^{D_P} \bmod P$ and $M_Q = {C_Q}^{D_Q} \bmod Q$.
3. Calculate the coefficients $S_P = M_P R_P \bmod N$ and $S_Q = M_Q R_Q \bmod N$.
4. Calculate $M = S_P + S_Q$. If $M \geq N$ then calculate $M = M - N$.

It is obvious to see that the initial reductions of the ciphertext C (step 1) and the Chinese recombination (steps 3 and 4) do not cause significant computational cost compared to the calculations in step 2. The two exponentiations (step 2) can be computed independent from each other and in parallel. Compared to the non-CRT based decryption performed on an n bit hardware, one of the CRT-based exponentiations is about 4 times faster if an $n/2$ bit hardware is used. This dramatic speed improvement is due to the 50% length reduction of both, the exponent and the modulus. Performing the two exponentiations within step 2 in parallel requires two $n/2$ bit modular multipliers, yielding a speed up factor of $4 - \epsilon$, with ϵ accounting for the steps 1, 2 and 4. See section 6 for a suggested reconfigurable modular multiplier which is able to execute either one n bit exponentiation or two $n/2$ bit exponentiations in parallel.

4 Multiplier architecture

In section 2 it was explained how a modular exponentiation can be calculated by continued modular multiplica-

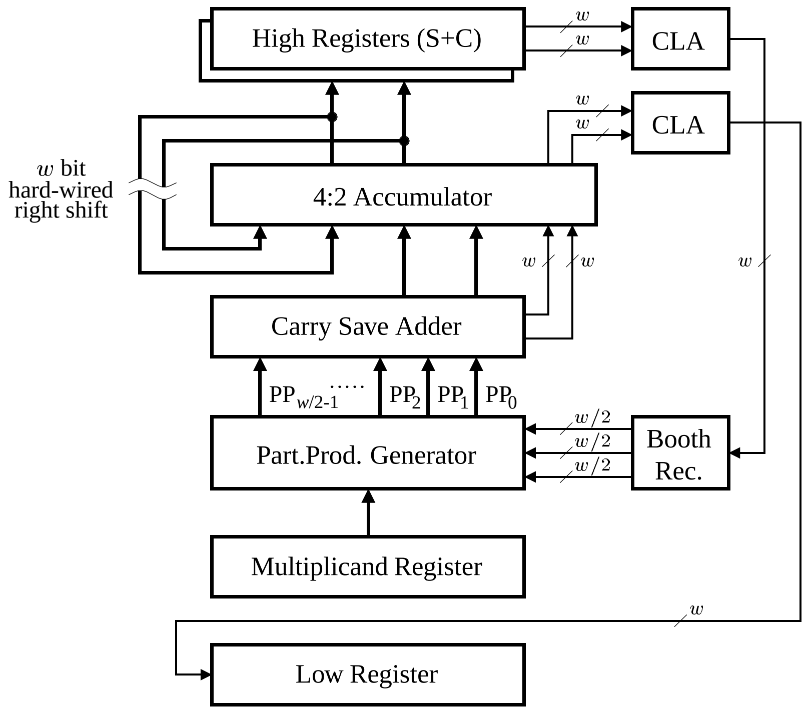

Figure 2. Basic architecture of the partial parallel multiplier.

tions, and how three simple multiplications and one addition result in a modular multiplication. The multiplier hardware which will be introduced in this section is optimized for executing long integer multiplications according to the FastMM algorithm.

4.1 Partial parallel multiplier

Figure 2 illustrates the architecture of the high-speed word-serial multiplier of the RSAγ, in the following denoted as *partial parallel multiplier* (PPM). According to the word-size w, the PPM processes $w/2$ partial-products of n bits at once, whereby n denotes the length of the modulus. In order to reach a high degree of parallelism, a word-size of $w = 16$ was chosen for the PPM. The actual RSAγ prototype is optimized for a modulus length of $n = 1024$, thus the PPM has a dimension of $(n+2w)*w = 1056*16$ bits. The PPM could be implemented in an array type architecture [Rab96] or a Wallace tree architecture [Wal64]. For RSAγ the array architecture is the better choice since it results in a more regular layout and less routing effort, especially when Booth recoding is applied.

Booth recoding [Mac61] is implemented to halve the number of partial-products, which almost doubles the multiplication speed with low additional hardware effort. According to the word-size w, the PPM processes $w/2$ partial-products of n bits at once. Since Booth recoding incorporates a radix 4 encoding of the multiplier, it requires a more complex *partial-product generator* (PPG) and a *Booth recoder circuit* (BR). The Booth recoder circuit is needed to generate the appropriate control signals for the PPG. Assuming $w = 16$ and $n = 1024$, there are eight PPGs required to calculate the eight partial-products, whereby each PPG consists of 1024 Booth multiplexers.

In order to reduce these $w/2$ partial-products to a single redundant number, the array multiplier needs $w/2-2$ *carry save adders* (CSA), assuming that the first three partial-products are processed by one CSA. Each CSA in the multiplier core consists of n half-cycle full adders presented in [Sch96], which introduce only low latching overhead and allow the maximum clock frequency to be kept high.

The output of the CSA is accumulated to the current intermediate sum in a carry save manner, too. This implies a carry save *accumulator* circuit performing a 4:2 reduction. The accumulator circuit also consists of half cycle full adders, thus the 4:2 reduction is finished after one clock cycle. Aligning the intermediate sum to the next CSA output is done by a w bit hard-wired right shift operation. Beside the carry save adders, also two *carry lookahead adders* (CLA) are required to perform a redundant to binary conversion of 16 bit words. Redundant to binary conversion is a 2:1 reduction, therefore a pipelined version of the CLA proposed in [BK82] is used to overcome the carry delay.

Additionally, the PPM also consists of seven *registers*, each of them is n bit wide: *HighSum* and *HighCarry, Low, Multiplicand, Data, N1* and *NegN.* Note that the registers *Data, N1* and *NegN* are not shown in figure 2. The registers *HighSum* and *HighCarry* store the upper part of the product. Two registers are necessary since the upper part is only available in redundant representation. Register *Low* receives the lower part of the result and register *Multiplicand* contains the actual multiplicand. The register *Data* is commonly used for storing the n bit block of ciphertext/plaintext to become de/encrypted. The registers *N1* and *NegN* contain the two precomputed constants for the FastMM algorithm. All seven registers and the accumulator are connected by an n bit bus to enable parallel register transfers.

4.2 Execution of a simple multiplication

In order to explain how the PPM executes a single multiplication, let us assume a modulus length of $n = 1024$ bit and a word-size of $w = 16$. This means that each shift operation of the registers results in a 16 bit right-shift of the stored value. At the beginning of a multiplication, the multiplicand resides within the register *Multiplicand,* and the multiplier (which is assumed to be available in redundant representation) resides within the registers *HighSum* and *HighCarry.* A single multiplication takes place in the following way:

1. Within each step, a 16 bit word of the (redundant) multiplier is shifted out of the registers *HighSum* and *HighCarry,* starting with the least significant word. These 16 bit words are converted from redundant into binary representation by a CLA, which requires three clock cycles.

2. When the 16 bit binary multiplier word reaches the Booth recoder circuit, it generates the control signals for the PPG. The PPG calculates a set of eight partial-products, which is propagated to the CSA. Booth recoding and the distribution of the control signals requires two clock cycles.

3. Within three clock cycles, the CSA reduces the set of eight partial-products to a single redundant number. However, as the CSA is a pipelined circuit, one set of eight partial-products can be processed each clock cycle.

4. The output of the CSA is then accumulated to the current intermediate sum within one cycle. Now, the least significant 16 bit of the intermediate sum already represent a word of the lower part of the product.

5. A CLA is used again to convert the redundant 16 bit words of the lower part into binary representation. Subsequently, the binary words are shifted into the register *Low,* where they finally represent the complete lower part of the product.

6. After the last set of eight partial-products has been processed, the upper part of the result resides within the accumulator after three cycles and can be loaded into the registers *HighSum* and *HighCarry.*

Whenever the upper part of the product is needed as operand for the next multiplication, it can be used as the multiplier which is allowed to be redundant. The lower part of the product always appears in binary representation; therefore it might be used as multiplicand or as multiplier. The following table summaries the operand requirements and appearance of the product.

operand	representation	schedule
multiplier	bin. or red.	sequentially, w bit words
multiplicand	binary	parallel, begin of multiplication
lower part of product	binary	sequentially, w bit words
upper part of product	redundant	parallel, end of multiplication

The steps needed for a single multiplication depend on the length of the modulus n and the word-size w on which the multiplier operates. The actual RSAγ prototype has a word-size of $w = 16$ and needs exactly 80 clock cycles for a single 1024 bit multiplication.

4.3 Execution of a modular multiplication

When applying the square and multiply algorithm, a modular exponentiation is performed by successive square and multiply steps. For a square step, the result of the previous modular multiplication, which resides within the register *Low,* acts as both, multiplicand as well as multiplier. For a multiply step, the n bit block of data to become de/encrypted is the multiplicand, and the result of the previous modular multiplication is the multiplier.

According to the FastMM algorithm presented in section 2, a modular multiplication takes place in the following way:

1. For multiplication 1 of the FastMM algorithm, the (redundant) registers *High* are loaded from register *Low* and register *Multiplicand* is either loaded from register *Low* (square step) or from register *Data* (multiply step). After the multiplication has been performed as

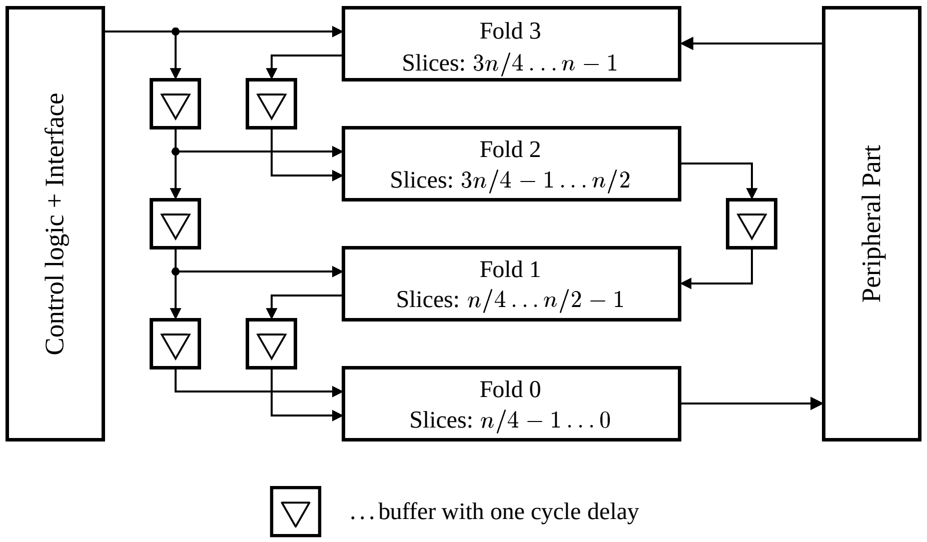
Control logic + Interface
Fold 3
Slices: $3n/4 \ldots n-1$
Fold 2
Slices: $3n/4-1 \ldots n/2$
Fold 1
Slices: $n/4 \ldots n/2-1$
Fold 0
Slices: $n/4-1 \ldots 0$
Peripheral Part
…buffer with one cycle delay

Figure 3. The clocked folding principle.

described in section 4.2, the lower part of the result resides within the register *Low* and the upper part resides within the accumulator.

2. For multiplication 2 of the FastMM algorithm, the (redundant) registers *High* are loaded from the accumulator and register *Multiplicand* is loaded from register *N1*. The multiplication is performed as described in section 4.2, but without shifting the words of the lower part result into register *Low*. Note that only the upper part of the result from multiplication 2 is needed for the next multiplication. Register *Low* still contains the lower part result of multiplication 1.

3. For multiplication 3 of the FastMM algorithm, the (redundant) registers *High* are loaded from the accumulator and register *Multiplicand* is loaded from register *NegN*. The multiplication is performed as described in section 4.2, but the accumulator is initialized with the lower part result of multiplication 1. Thus, multiplication 3 is performed together with the addition of the FastMM algorithm. The result of the modular multiplication resides within register *Low* after multiplication 3.

A modular multiplication for 1024 bit operands requires 227 cycles when it is performed on a PPM with a word-size of $w = 16$.

5 Floorplanning and clocked folding

From the viewpoint of floorplanning, the RSAγ datapath shown in figure 2 can be divided into two parts: a regular part which includes all n bit wide circuits (registers, carry save adders, partial-product generators, and the accumulator), and a peripheral part which consists of all the other circuits (CLA, booth recoder, and control logic). Since the regular part is about 80% of the total chip area, its layout is subject of detailed optimization. In order to exploit the regularity of the datapath structure, the regular part is built of $n + 2w$ identical copies of a one bit *slice*. This slice consists of seven 1-bit register cells, eight 1-bit adder cells and eight 1-bit partial-product generator cells, including a uniform inter-cell routing.
A slice-based layout for the regular part of the RSAγ crypto chip has two significant advantages:

1. The place–and–route procedure needs to be solved only for a single slice.

2. As all bit positions have a uniform layout, the verification process (parasitics extraction, timing simulation, back annotation) is simplified. If a particular timing is verified within a single slice, it is also verified in all other slices.

The slices are supposed to connect by abutment, as also the routing between the slices is uniform. Since the wire length of control signals and inter-slice routing grows with the width of a single slice, narrow slices reduce the total area demand. Based on a 0.6μ CMOS process, the slice width is limited to about 35μm with a corresponding slice length of about 1 mm, which results in a datapath layout size of about $35*1$ mm. Such an aspect ratio would be unacceptable, because chip packages are most frequently considered for square shapes. Not only packaging requires a fairly square shaped chip layout, also the distribution of global signals

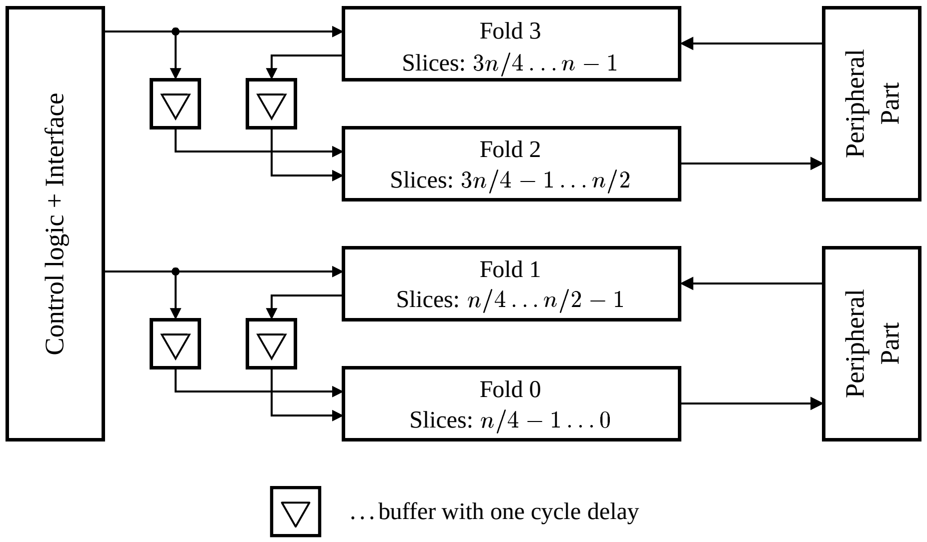
Control logic + Interface
Fold 3
Slices: $3n/4 \ldots n-1$
Fold 2
Slices: $3n/4-1 \ldots n/2$
Fold 1
Slices: $n/4 \ldots n/2-1$
Fold 0
Slices: $n/4-1 \ldots 0$
Peripheral Part
Peripheral Part
…buffer with one cycle delay

Figure 4. Reconfigured multiplier datapath for CRT-based RSA decryption.

(e.g. control signals, output signals of the booth recoder) is much easier when the layout has an aspect ratio close to 1, since the corresponding wires are significantly shorter. Also minimizing the clock skew is very important. Again, delays due to interconnection must be minimized.

In order to fulfill the requirement for a square shaped layout, a special floorplanning technique termed *folding* is applied to the regular part of the RSAγ datapath. The 1056 bit wide regular part is divided into four folds, whereby each fold consists of 264 slices, as illustrated in figure 3. Note that folding can only be applied if the direction of data signal flow is restricted from more to less significant bit positions. It is not difficult to see that the components shown in figure 2 can be arranged in a way to meet this restriction.

But folding has also a serious disadvantage: Transmitting data signals between folds requires long interconnection wires, therefore buffers have to be inserted to maintain the increased capacitive load. The additional delay caused by the buffers and the interconnection wires would compromise the overall performance. This pipeline bottleneck can be removed by inserting buffers with one cycle delay between the folds. Since the data signal flow is limited from more to less significant bit positions, the architecture allows one cycle delay between subsequent folds. When all data input signals for fold 2 are provided by fold 3, the control signals can also be delayed by one cycle, causing calculations taking place in fold 2 to be delayed by one cycle. Likewise calculations in fold 1 and fold 0 are delayed by two and three cycles. The additional delay of three cycles caused by this *clocked folding* does not compromise overall latency very much. But on the other hand, buffers with one cycle delay allow much higher clock frequencies than ordinary buffers.

6 Fast reconfigurable multiplier datapath

For efficient CRT-based RSA decryption, a number of hardware options to the architecture presented in section 4 are required. As a consequence of the folding technique, the regular part of the multiplier is already divided into four parts. So it is obvious to add a second irregular part to get two $n/2$ bit multipliers. Each of this $n/2$ bit multipliers consists of two folds and one irregular part. By adding multiplexers to switch between the folds and the irregular parts, the multiplier is reconfigurable to perform either one n bit exponentiation or two $n/2$ bit exponentiations in parallel. The block diagram in figure 3 represents the multiplier architecture if non-CRT based RSA decryptions is selected. In this case all four folds are needed to execute an n bit exponentiation. Alternatively, the architecture can be split into to two multipliers, each consisting of two folds and one irregular part, as illustrated in figure 4. Note that the multiplexers to switch between CRT-based and non CRT-based operating mode are not shown in figure 4. Each of the $n/2$ bit multipliers is connected to the same control signals, causing them running synchronously, except for the exponents D_P, D_Q controlling the square and multiply sequences (not shown in figure 4). Thus the clocked folding technique is also very advantageous for the realization of a reconfigurable multiplier datapath.

Beside a second irregular part and a number of multiplexers, also additional storage is required for CRT-based RSA decryption. The precomputed exponents, the precomputed coefficients for Chinese recombination and the prime decomposition P, Q have to be stored on–chip. Embedded SRAM provides a low area solution for that purpose. Note that there is no necessity for additional registers in the multiplier core, as in CRT mode the registers *NegN* and *N1* are also split into two parts, whereby one part stores the precomputed constants for $\mathrm{mod} P$ calculations, and the other part stores the constants for $\mathrm{mod} Q$ calculations.

7 Summary of results and conclusions

The subject of this paper was to present efficient algorithms for modular arithmetic and a multiplier architecture which is optimized for these algorithms. The prototype of the RSAγ crypto chip is designed for a modulus length of $n = 1024$ bits and a multiplier word-size of $w = 16$. Based on a 0.6μ standard CMOS process, the silicon area of the multiplier core is about 70 mm^2 and contains approx. 1,000,000 transistors, whereby most parts of the chip were implemented in full custom design methodology. The execution of a 1024 bit modular multiplication needs 227 clock cycles. When the multiplier core is clocked with 200 MHz, this results in a decryption rate of 560 kbit/s. In CRT mode, the decryption rate increases to 2 Mbit/s. This high performance confirms the efficiency of the implemented hardware algorithms and the multiplier architecture. Furthermore, the proposed design is highly scalable with respect to the multiplier word-size as well as the modulus length. The only limiting factor is the available silicon area. A state-of-the-art 0.25μ CMOS process would allow to increase the multiplier word-size to $w = 32$, which doubles the performance. The high regularity of the design makes it attractive for implementation in a full custom design methodology. Another significant advantage of the architecture is its reconfigurability to execute either one 1024 bit exponentiation or two 512 bit exponentiations in parallel. Thus the CRT-based RSA decryption increases the performance by a factor of 3.5, with only low additional hardware effort.

References

[Bar87] P. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in cryptology: CRYPTO '86: proceedings*, volume 263 of *Lecture Notes in Computer Science*, pp 311–323, Springer-Verlag, Berlin, Germany, 1987.

[BK82] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3), pp. 260-264, 1982.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6) pp. 644–654, November 1976.

[Dhe98] J. F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. Thesis (Ph.D.), Université catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.

[Knu69] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, 1969.

[Kob94] N. Koblitz. *A Course in Number Theory and Cryptography*, volume 114 of *Graduate texts in mathematics*. Springer-Verlag, Berlin, Germany, second edition, 1994.

[Mac61] O. L. MacSorley. High-Speed Arithmetic in Binary Computers. *Proceedings of the Institute of Radio Engineers*, 49:67–91, 1961.

[MPPS95] W. Mayerwieser, K. C. Posch, R. Posch, and V. Schindler. Testing a High-Speed Data Path: The Design of the RSAβ Crypto Chip. *J.UCS: Journal of Universal Computer Science*, 1(11) pp. 728–744, November 1995.

[MVV97] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, Boca Raton, FL, USA, 1997.

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

[PP89] K. C. Posch and R. Posch. Approaching encryption at ISDN speed using partial parallel modulus multiplication. IIG report 276, Institutes for Information Processing Graz, Graz, Austria, November 1989.

[QC82] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *IEE Electronics Letters*, 18(21), pp. 905–907, October 1982.

[Rab96] J. M. Rabaey. Digital Integrated Circuits - A Design Perspective. *Prentice Hall Electronics and VLSI Series*, Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the Association for Computing Machinery*, 21(2) pp. 120–126, February 1978.

[Sch96] V. Schindler. A low-power true single phase clocked (TSPC) full-adder. *Proceedings of the 22nd ESSCIRC*, Neuchâtel, Switzerland, pp. 72–75, 1996.

[SV93] M. Shand and J. Vuillemin. Fast Implementation of RSA Cryptography. *Proceedings of 11th Symposion on Computer Arithmetic*, 1993.

[Wal64] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computation*, EC-13(1), pp. 14–17, 1964.

[YS89] J. Yuan and C. Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1), pp. 62–70, 1989.