High-Speed RSA Hardware Based on Barret's Modular Reduction Method*

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 public-key cryptosystems like the RSA encryption scheme or the Diffie-Hellman key agreement scheme is primarily determined by an efficient implementation of the modular arithmetic. This paper presents the basic concepts and design considerations of the RSAγ crypto chip, a high-speed hardware accelerator for long integer modular exponentiation. 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.

RSAγ uses a hardware-optimized variant of Barret's modular reduction method to avoid the division in the modular multiplication. From an architectural viewpoint, a high degree of parallelism in the multiplier core is the most significant characteristic of the RSAγ crypto chip. The actual prototype contains a 1056$*$16 bit partial parallel multiplier which executes a 1024-bit modular multiplication in 227 clock cycles. Due to massive pipelining in the long integer unit, the RSAγ crypto chip reaches a decryption rate of 560 kbit/s for a 1024-bit exponent. The decryption rate increases to 2 Mbit/s if the Chinese Remainder Theorem is exploited.

Keywords: Public-key cryptography, RSA algorithm, modular arithmetic, partial parallel multiplier, pipelining, full-custom VLSI design.

1 Introduction

Security is an important aspect in many applications of modern information technology, including electronic commerce, virtual private networks, secure internet access, and digital signatures. All these services apply public-key cryptography. Practical and secure examples for public-key algorithms are the Rivest, Shamir and Adleman (RSA) encryption scheme [RSA78], the Diffie-Hellman (DH) key agreement scheme [DH76], and the Digital Signature Algorithm (DSA) for generation/verification of digital signatures [Nat94]. From a mathematical viewpoint,

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

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 191–203, 2000.

the algorithms mentioned have a common characteristic: they perform long integer modular exponentiation.

In order to meet modern security demands, the modulus should have a length of at least 1024 bits. The calculation of a 1024-bit RSA decryption in software causes a very high computational cost since the complexity of the modular exponentiation is n^3 for n-bit numbers. Special hardware accelerators like the RSAγ crypto chip contain an optimized long integer multiplier and for this reason they are more efficient for RSA decryption than a general-purpose 32-bit CPU.

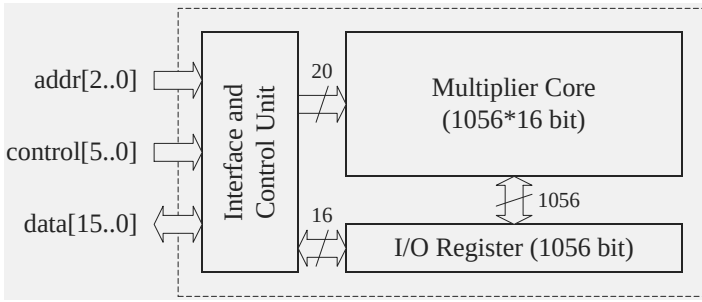

Fig. 1. Main components of the RSAγ crypto chip.

Figure 1 shows the most important components of the RSAγ crypto chip on the system-level. For communication with the off-chip world, the RSAγ provides a 16-bit standard microcontroller *interface*. Data interchange and command invocation are performed via this interface. The *control unit* is responsible for feeding the multiplier core with control signals. Since the multiplier is clocked with 200 MHz, and the control signals have to be provided with the same frequency, they can not be delivered from outside the chip via the pads. Therefore, the control sequences for the square and the multiply operation are stored in a FIFO within the control unit. Other parts of the control unit like the register for the exponent and the non-speed critical portions of the control logic operate at 25 MHz. The frequency ratio between the core and the interface is 8:1.

The *I/O register* supports 16-bit serial data transfer with the interface unit, and 1056-bit parallel data exchange with the multiplier core. Note that the data transfer from and to the multiplier core does not affect the throughput since the fetching of the exponentiation result and the loading of the next data block takes place independently of the multiplier core. The performance of the RSAγ crypto chip mainly relies on the efficiency of the modular arithmetic, therefore this paper focusses on the *multiplier core*. We present the basic algorithmic and architectural concepts of the multiplier, and describe how they were combined and optimized for each other in order to reach maximum performance.

Since the publication of the RSA public-key cryptosystem in 1978, many algorithms for modular multiplication have been proposed; the most important are

summarized in [NM96]. Nevertheless, it turns out that two algorithms are preferred for hardware implementation: the Barret modular reduction method [Bar87] and the Montgomery algorithm [Mon85]. In recent years, most proposed approaches are based on Montgomery's algorithm, either in conjunction with a redundant number representation or in an systolic array architecture. Blum et al. reported an implementation of Montgomery modular exponentiation on FPGAs [BP99]. They reached a decryption time of approximately 10 ms for a 1024-bit modulus when the Chinese Remainder Theorem is applied. Compared to the architectures based on Montgomery's algorithm, the multiplier core of the RSAγ crypto chip differs in the following characteristics:

- *Implemented algorithms* – RSAγ uses an optimized variant of Barret's modular reduction method, termed *FastMM algorithm* [MPPS95], instead of the more frequently used Montgomery algorithm. The FastMM algorithm is very well suited for hardware implementation as it avoids the division in the modular reduction operation and calculates a modular multiplication by three simple n-bit multiplications and one addition. Additionally, the RSAγ crypto chip can exploit the Chinese Remainder Theorem (CRT) to speed up the decryption process [SV93].
- *Multiplier architecture* – From an architectural viewpoint, the multiplier in the RSAγ crypto chip is a *partial parallel multiplier* (PPM). The actual prototype contains a $1056*16$ bit PPM, which schedules the multiplicand fully parallel and the multiplier sequentially in 16-bit words. Compared to $r*r$ bit multipliers with $r=32$ or 64, the partial parallel multiplier is much faster because it is able to process the long integers directly. For this reason, the complexity of an n-bit RSA exponentiation is reduced from n^3 to n^2. Due to its high degree of parallelism, the multiplier core computes a 1024-bit modular multiplication in 227 clock cycles. Additionally, pipelining significantly increases the throughput in RSA encryption.
- *Circuit technique and design methodology* – Although the architecture (theoretically) may accept an arbitrary degree of parallelism, it must be noticed that area and power resources are limited on a single chip. Therefore, the goal of achieving optimum performance involves low-power as well as low area design. The *true single-phase circuit technique* (TSPC) turns out to be useful for applications that can take use of pipelining and massive parallelism [YS89]. The RSAγ datapath is implemented in non-precharged TSPC logic to simplify the clock generation and clock distribution. Since the whole multiplier core consists of very few basic cells and is highly regular, it can be realized rather simple in a *full-custom design methodology.*

The rest of the paper is structured as follows: Section 2 describes the implemented algorithms for exponentiation and modular multiplication in detail. Section 3 covers the architecture of the RSAγ multiplier core and explains how it executes a simple multiplication and a modular multiplication, respectively. In section 4, VLSI design related topics like floorplanning and clock distribution are sketched. The paper finishes with conclusions in section 5.

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:

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

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 $q = \lfloor \frac{Z}{N} \rfloor$ is avoided by computing the quotient $\widetilde{q}$ instead:

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

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 $\lfloor 2^{2n}/N \rfloor$ only depends on the modulus N 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 performing 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 required 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 wordsize w of the multiplier hardware (usually 16 or 32 bits) 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, the quotient $\widetilde{q}$ is calculated as follows [MPPS95]:

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

The *FastMM algorithm* combines multiplication and modified Barret reduction to implement the modular multiplication by three multiplications and one addition according to the following formulas:

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

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 only depend on the modulus N, they can be precomputed:

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

$$NegN = 2^{n+2w} - N \quad (NegN \text{ is the two's complement of } N) \tag{5}$$

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] and [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.

2.4 Chinese Remainder Theorem

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, the modular exponentiation can be performed separately $\bmod P$ and $\bmod Q$ with shorter exponents, as described in [QC82] and [SV93]. Since the length of the operands is about $n/2$, there are only about $3n/4$ modular multiplications needed for a single exponentiation. The RSAγ crypto chip is able to compute both exponentiations in parallel, as the n-bit multiplier core can be divided into two $n/2$-bit multipliers. Running the two $n/2$-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 almost 4.

3 Multiplier Architecture

In the previous section we explained how a modular exponentiation can be calculated by continued modular multiplications, and how three simple multiplications and one addition result in a modular multiplication. The multiplier hardware introduced in this section is optimized for the execution of long integer multiplications according to the FastMM algorithm.

3.1 Partial Parallel Multiplier

Figure 2 illustrates the architecture of the high-speed partial parallel multiplier (PPM) of the RSAγ crypto chip. In order to reach a high degree of parallelism, a wordsize of $w = 16$ was chosen for the PPM. The actual RSAγ prototype is optimized for a modulus length of $n = 1024$, thus the multiplier core has a dimension of $(n+2w)*w = 1056*16$ bit. The PPM could be implemented in an *array-type architecture* [Rab96] or a *Wallace tree architecture* [Wal64]. It turns out that the array architecture is the better choice since it offers a more regular layout and less routing effort, especially when Booth recoding is applied.

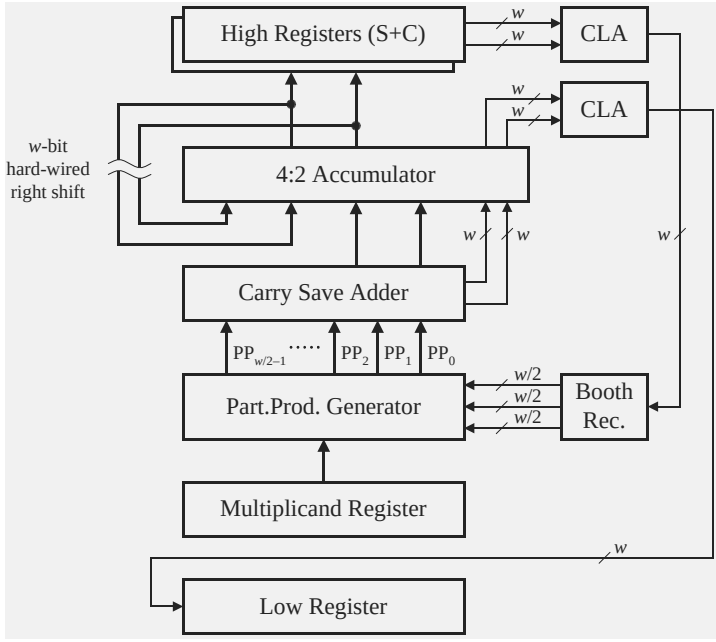

Fig. 2. Basic architecture of the partial parallel multiplier.

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 wordsize 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 $n = 1024$, eight PPGs are required to calculate the 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 bits 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.

3.2 Execution of a Simple Multiplication

In order to explain how the PPM executes a single multiplication, let us assume a modulus length of $n = 1024$ bits and a wordsize 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 bits 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 wordsize w on which the multiplier operates. The actual RSAγ prototype has a wordsize of $w = 16$ and needs exactly 80 clock cycles for a single 1024-bit multiplication.

3.3 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 2.3, 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 described in section 3.2, 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 3.2, 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 3.2, 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 clock cycles when it is performed on a PPM with a wordsize of $w = 16$.

4 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,* as illustrated in figure 3. 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.

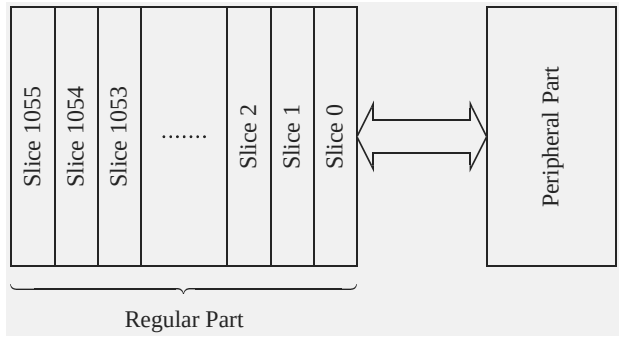

Fig. 3. Slice orientation of the regular part layout.

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, it turns out that the slice width is limited to approximately 35 μm with a corresponding slice length of approximately 1 mm, which results in a datapath layout size of approximately $35*1$ 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 (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.

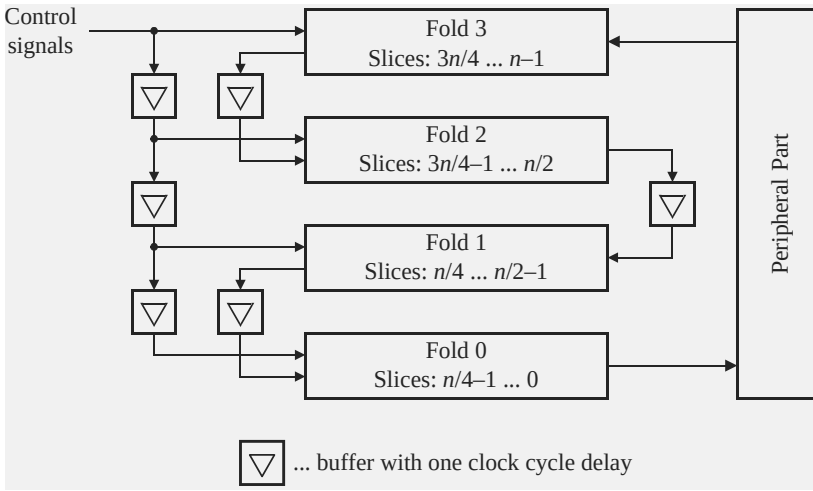

Fig. 4. The clocked folding principle.

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 bits wide regular part is divided into four folds, whereby each fold consists of 264 slices, as illustrated in figure 4. Note that folding can be applied if and only 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 drive the increased capacitive load. The additional delay caused by the buffers and the interconnection wires would compromise the overall performance. But 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, respectively. 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 rates than ordinary buffers.

5 Summary of Results and Monclusions

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 wordsize of $w = 16$. Based on a $0{,}6\mu$ standard CMOS process with one poly layer and two metal layers, the silicon area of the multiplier core is about 70 mm^2 and contains approximately 10^6 transistors. The execution of a 1024-bit modular multiplication requires 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. The 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 wordsize as well as the modulus length. The most limiting factor is the available silicon area. A modern $0{,}25\mu$ CMOS process would allow to increase the multiplier wordsize to $w = 32$, which doubles the performance.

References

Bar87. P. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, *Advances in Cryptology – CRYPTO '86 Proceedings*, vol. 263 of *Lecture Notes in Computer Science*, pp 311–323, Springer-Verlag, 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.

BP99. T. Blum and C. Paar. Montgomery Modular Exponentiation on Reconfigurable Hardware. *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 70–77, 1999.

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.

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

Mon85. P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170) pp. 519–521, 1985.

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.

NM96. D. Naccache and D. M'Raïhi. Arithmetic co-processors for public-key cryptography: The State of the Art. *IEEE Micro*, pp. 14–24, June 1996.

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 the 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.