Montgomery Modular Exponentiation on Reconfigurable Hardware*

Thomas Blum
Worcester Polytechnic Institute
ECE Department
Worcester, MA 01609-2280, USA
tblum@ece.wpi.edu

Christof Paar
christof@ece.wpi.edu

Abstract

It is widely recognized that security issues will play a crucial role in the majority of future computer and communication systems. Central tools for achieving system security are cryptographic algorithms. For performance as well as for physical security reasons, it is often advantageous to realize cryptographic algorithms in hardware. In order to overcome the well-known drawback of reduced flexibility that is associated with traditional ASIC solutions, this contribution proposes arithmetic architectures which are optimized for modern field programmable gate arrays (FPGAs). The proposed architectures perform modular exponentiation with very long integers. This operation is at the heart of many practical public-key algorithms such as RSA and discrete logarithm schemes. We combine the Montgomery modular multiplication algorithm with a new systolic array design, which is capable of processing a variable number of bits per array cell. The designs are flexible, allowing any choice of operand and modulus.

Unlike previous approaches, we systematically implement and compare several variants of our new architecture for different bit lengths. We provide absolute area and timing measures for each architecture. The results allow conclusions about the feasibility and time-space trade-offs of our architecture for implementation on Xilinx XC4000 series FPGAs. As a major practical result we show that it is possible to implement modular exponentiation at secure bit lengths on a single commercially available FPGA.

1 Introduction

It is widely recognized that security issues will play a crucial role in many future computer and communication systems. A central tool for achieving system security are cryptographic algorithms. For performance as well as for physical security reasons it is often required to realize cryptographic algorithms in hardware. Traditional ASIC solutions, however, have the well-known drawback of reduced flexibility compared to software solutions. Since modern security protocols are increasingly defined to be *algorithm independent*, a high degree of flexibility with respect to the cryptographic algorithms is desirable. A promising solution which combines high flexibility with the speed and physical security of traditional hardware is the implementation of cryptographic algorithms on reconfigurable devices such as FPGAs and EPLDs. In the case of public-key schemes, algorithm independence can mean not only a change of the actual crypto algorithm but also change of parameters such as bit length, modulus, or exponents. This contribution deals with arithmetic architectures for modular exponentiation with very long integers which is at the heart of most modern public-key schemes. Most notably, both RSA and discrete logarithm-based (e.g., Diffie-Hellman key exchange or the Digital Signature Algorithm, DSA) schemes require modular long number exponentiation.

The challenge at hand is to design such arithmetic architectures for operands with up to 1024 bit on current FPGAs. The very long word lengths prohibit the application of many proposed architectures as they would result in unrealistically large resource requirements. In this contribution we derive a modular exponentiation architecture which combines Montgomery's modular reduction scheme and a novel systolic array architecture. The systolic array architecture requires considerably fewer logic resources than many other systolic array architectures for modular arithmetic. This is crucial, as one of our goals was to derive solutions that can fit into a single FPGA. Clearly a design which fits in a single FPGA has many cost and design advantages over multi–FPGA solutions. Another important objective was to systematically implement various architecture options for different bit lengths.

This contribution is structured as follows. In Section 2, we summarize some of the previous work on modular ex-

*The research was supported in part through an NSF CAREER award #CCR–9733246.

ponentiation. Section 3 describes algorithms for modular exponentiation and multiplication and some simplifications and speed-ups for their hardware implementation. In this section we also describe some of the relevant features of the Xilinx XC4000 FPGA series. Section 4 outlines our architecture for modular exponentiation. Section 5 briefly describes our methodology and tools that were used for this research. Section 6 of this contribution posts the timing and area results obtained. A comparison to other architectures and an outlook conclude this contribution.

2 Previous Work

In the following, we will summarize relevant previous work in the field of modular multiplication. Most proposed approaches are based on Montgomery's algorithm [10], either in conjunction with a redundant number representation or in an systolic array architecture. Solutions using other algorithms have also been presented.

To avoid the carry propagation in multiplication/addition architectures several solutions have been proposed in the literature. They either use Montgomery's algorithm, in combination with a redundant radix number system [15, 13, 17, 4, 18, 6] or the Residue Number System [1].

The Research Laboratory of Digital Equipment Corp. in Paris implemented modular exponentiation in architectures on FPGAs [17, 13]. They utilized an array of 16 XILINX 3090 FPGAs. Compared to XILINX 4000 series in terms of flip–flops, this is equivalent to a chip with 5100 configurable logic blocks (CLBs). In terms of logic resources this is equivalent to a chip of 4000 CLBs. In their work they used several speed-up methods [13] including the Chinese remainder theorem, an asynchronous carry completion adder, and a windowing method. The implementation computes a 970bit RSA decryption at a rate of 185kb/s (5.2ms per 970 bit decryption) and a 512 bit RSA decryption in excess of 300 kb/s (1.7ms per 512 bit decryption). A drawback of this solution is that the binary representation of the modulus is hardwired into the logic representation so that the architecture has to be reconfigured with every new modulus.

There has been a number of proposals for systolic array architectures for modular arithmetic. However, no implementations have been reported to our knowledge. In [5] a VLSI solution is presented where a modular multiplication is calculated in $(4n+1) \cdot 3n/2$ clock cycles (n is the number of bits of the modulus). That is approximately four times more cycles than in a conventional solution. In terms of resources this design would be suitable for FPGA.

Similar two-dimensional systolic arrays are presented in [7, 19, 20, 16]. For a radix of two they all propose an $n \times n$ matrix of one bit processing elements. With this configuration $2n$ modular multiplications are calculated at the same time and the theoretical throughput is one modular multiplication per clock cycle. In terms of resources, such a solution is not feasible in either VLSI or FPGA for the bit length required in public-key algorithms. Even implementing only one row of processing elements, (resulting in n times slower throughput) into presently available FPGAs is difficult in terms of resources. We tried to overcome the shortage of resources per chip by using larger processing elements and thus saving overhead.

Reference [2] provides a good overview of previously presented architectures for VLSI implementations of modular integer arithmetic. Reference [3] summarizes the chips available in 1990 for performing RSA encryption. More recently an approach [23] has been presented that utilizes precomputed complements of the modulus and is based on the iterative Horner's rule. Compared to Montgomery's algorithms these approaches use the most significant bits of an intermediate result to decide which multiples of the modulus to subtract. The drawback of these solutions is that they either need a large amount of storage space or many clock cycles to complete a modular multiplication. The authors attempted to overcome the later problem by a higher clock frequency which is possible due to a simplified modulo reduction operation.

3 Preliminaries

3.1 Modular Exponentiation and RSA

We start this section with a short description of the RSA algorithm, proposed by Rivest, Shamir and Adleman [12] in 1978. The algorithm is based on modular exponentiation of integers. The private key of a user consists of two large primes p and q and an exponent D. The public key consists of the modulus $N = p \times q$ and an exponent E such that $E = D^{-1} \bmod (p-1)(q-1)$. In the remainder of the article we always assume that N can be represented by n bits. To encrypt a message X the user computes:

$$Y = X^E \bmod N$$

Decryption is done by calculating:

$$X = Y^D \bmod N$$

The identical operations are utilized for the RSA digital signature scheme. In order to thwart currently known attacks, the modulus N and thus X and Y should have a length of 512 – 1024 bits. Both encryption and decryption require algorithms for computing a modular exponentiation. This can be realized by using the square and multiply algorithm [14]. To compute squaring and multiplication in parallel we can use the following version [20]:

Algorithm 1: computes $P = X^E \bmod N$, where $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0, 1\}$

1. $P_0 = 1$, $Z_0 = X$
2. for i = 0 to $n - 1$ do
3. $Z_{i+1} = Z_i^2 \bmod N$
4. if $e_i = 1$ then $P_{i+1} = P_i \cdot Z_i \bmod N$

Algorithm 1 takes $2n$ operations in the worst case and $1.5n$ on average.

For speeding up encryption the use of a short exponent E has been proposed [8]. Recommended by ITU is the the Fermat prime $F_4 = 2^{16} + 1$. Using F_4, the encryption is executed in only 17 operations. Other short exponents proposed include $E = 3$ and $E = 17$.

Obviously the same trick can not be used for decryption, as the decryption exponent D must be kept secret. But using the knowledge of the factors of $N = q \times p$, the Chinese Remainder Theorem [11] can be applied by the decrypting party. Two $n/2$ size modular exponentiations and an additional recombination instead of one n size modular exponentiations are computed in this case. Each modular exponentiation of length $n/2$ takes $1/4$ of the time required for an n – bit exponentiation. If both exponentiations are performed serially, an over–all speed–up factor of two is achieved. If they are performed in parallel, a speed–up factor of four is achieved.

3.2 Montgomery Modular Multiplication

As shown in the previous section, modular exponentiation is reduced to a series of modular multiplications and squarings. The algorithm for modular multiplication described below has been proposed by P. L. Montgomery in 1985 [10]. Several optimizations were taken from reference [19]:

Algorithm 2: Montgomery Modular Multiplication (radix 2) for computing $A \cdot B \bmod N$, where $B = \sum_{i=0}^{n+1} b_i 2^i$, $b_i \in \{0, 1\}$, $b_0 = 0$, $A = \sum_{i=0}^{n+2} a_i 2^i$, $a_i \in \{0, 1\}$, $a_{n+1} = 0$, $a_{n+2} = 0$

1. $R_0 = 0$
2. for $i = 0$ to $n + 2$ do
3. $q_i = R_i(0)$
4. $R_{i+1} = (R_i + a_i \cdot B + q_i \cdot N)/2$

B is shifted up one bit with $b_0 = 0$. This measure simplifies the computation of q_i, compared to the original algorithm.

The loop of Algorithm 2 is executed three more times than originally proposed. With this step we make sure the inequalities $R_i < 3N$ and $R_{n+3} < 2N$ always hold. The result of a modular multiplication R_{n+3} can thus be reused as input A and B for the next multiplication. We avoid the originally proposed final comparison and subtraction and make a pipelined execution of the algorithm possible.

A precondition for the algorithm to work is that the modulus N has to be relatively prime to the radix. In RSA this is always satisfied as N is a multiple of two primes and therefore odd. The algorithm above calculates $R_n = (2^{-n-3}AB) \bmod N$. To get the right result we need an extra Montgomery modular multiplication by $2^{2n+6} \bmod N$. However if further multiplications are required as for exponentiation it is better to pre–multiply all inputs by the factor $2^{2n+6} \bmod N$. Thus every intermediate result carries a factor 2^{n+3}. We just need to Montgomery multiply the result by 1 to eliminate that factor.

The final Montgomery multiplication with 1 makes sure our final result is smaller than N. Consider Algorithm 2 with $B < 4N$ (B shifted up) and $A = (0, \ldots, 0, 1)$. We will get $R_1 = B/2 < 2N$. As all remaining $a_i = 0$, we get at most $R_{i+1} = (R_i + N)/2 \rightarrow N$. If only one $q_i = 0$ $(i = 1, 2 \ldots n + 2)$, then $R_{i+1} = R_i/2 < N$ (probability: $1 - 2^{-(n+2)}$).

The whole computational complexity of Algorithm 2 lies in the three additions of n bit operands for computing R_{i+1}. As the propagation of n carries is too slow and an equivalent carry look ahead logic requires too many resources, two different strategies have been pursued in the literature:

1. Redundant representation: The intermediate results are kept in redundant form. Resolution into binary representation is only done at the very end and for feeding the intermediate result back as a_i in Algorithm 2.
2. Systolic Arrays: n processing units calculate 1 bit per clock cycle. The computed carries, q_i and a_i are "pumped" through the processing units. As these signals have to be distributed only between adjacent processing units, a faster clock speed and a resulting higher throughput should be possible. The cost is a higher latency and possibly more resources.

3.3 Xilinx XC4000 Series FPGAs

In this section we present some of the relevant features of the Xilinx XC4000 Series FPGAs and introduce a metric for FPGA cost and performance evaluation.

An FPGA device consists of three types of reconfigurable elements, the Configurable Logic Blocks (CLBs), I/O blocks (IOBs) and routing resources [22]. An XC4000 CLB is made up of 3 look–up tables, two flip-flops and programmable multiplexers. Any boolean function of 5 inputs,

any 2 functions of 4 inputs and some functions of up to 9 inputs can be computed in one CLB. The multiplexers can route these signals directly to the outputs or to the flip-flops. In the first case the flip-flops can be utilized to store direct inputs. Programmable routing resources connect the CLBs and IOBs into a network. For signal distribution all over the device there are 8 global nets available.

Another feature of the CLB is its dedicated hardware to accelerate the carry path of adders and counters [22]. An n bit ripple carry adder is implemented in $n/2 + 2$ CLBs. As the carry signal uses dedicated interconnects, there is no routing delay in the path and the total delay is fixed: $t_{pd} = 4.5 + n \cdot 0.35\ [ns]$.

On chip RAM reduces the cost of data storage. A single CLB can be used for a 16×2 bit or 32×1 bit ROM/RAM or for a 16×1 bit Dual Port RAM.

In previous work [20, 19, 4] the gate count model has been used for cost evaluation and the gate delay model for speed evaluation. This is not appropriate for FPGAs. As the functional unit of an FPGA is the CLB, we evaluate the cost (C) in number of CLBs. The operation time (T) consists of logic delay in the CLBs and routing delay and is obtained from Xilinx's Timing Analyzer software. As a third parameter we use the time – area product (TA). It is defined by time multiplied by cost.

4 A New Architecture

4.1 Design Overview

As described in Section 3.2, there have been two principle approaches proposed to compute Montgomery modular multiplication. A solution following approach 1 has already been implemented in FPGA [17]. The second approach using systolic arrays has drawn considerable attention in the research community. However, no architectures that specifically target FPGAs have been reported, nor are there reports of implementations of such systolic architectures. Our contribution targets these two goals. Our system can be divided hierarchically into three levels.

1. Processing Element: Compute u bits of a modular multiplication.

2. Modular Multiplication: An array of processing elements computes a modular multiplication.

3. Modular Exponentiation: Combine modular multiplications to modular exponentiation according to Algorithm 1.

In the following we describe the system with a bottom–up approach.

4.2 Processing Elements

A general radix 2 systolic array as proposed in [7, 19, 16, 5] utilizes n times n processing elements. As this approach would result in unrealistically large CLB counts for the bit length required in modern public key schemes, we implemented only one row of processing elements. To further reduce the required number of CLBs we implemented processing elements (units) of $u = 4$,8,16 bits. With this approach we need only n/u instead of n processing elements, and a considerable amount of overhead can be saved.

Similar to the approach in [9] we compute squarings and multiplications of Algorithm 1 in parallel. As explained in Section 4.3, this measure fully utilizes every cycle.

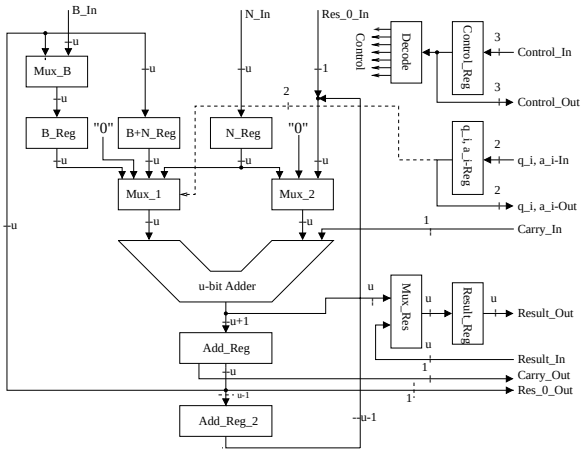

Figure 1. Processing Element (unit)

In the processing elements we need the following registers:

- *N-Reg* (u bits): storage of the modulus
- *B-Reg* (u bits): storage of the B multiplier
- *B+N-Reg* (u bits): storage of the intermediate result $B + N$
- *Add-Reg* ($u+1$ bits): storage of the intermediate result
- *Add-Reg-2* ($u - 1$ bits): storage of the intermediate result
- *Control-Reg* (3 bits): control of the multiplexers and clock enables
- a_i,q_i (2 bits): multiplier A, quotient Q, according to Algorithm 2
- *Result-Reg* (u bits): storage of the result at the end of a multiplication

The registers need a total of $(6u + 5)/2$ CLBs. Instead of computing $(R + a_i \cdot B + q_i \cdot N)/2$ in each iteration, we compute $N + B$ once and store the result in the *B+N-Reg*. Multiplexer Mux_1 selects one of its inputs 0, N, B, $B + N$ to be added to R according to the value of the binary variables a_i and q_i. The additional cost is a u – bit register, a slightly more complicated multiplexer Mux_1, and two more clock cycles per multiplication. The advantage is that only a two–operand adder is needed that can be implemented with the ripple carry adder optimized for the Xilinx XC4000 series (see Section 3.3). Also we need only one carry instead of two between units. The carry propagation delay of a 16–bit adder is equivalent to only one additional CLB delay. The adder can be combined into the CLBs of the *Add-Reg*; we need therefore no additional CLBs. An additional register *Add-Reg-2* allows storage of a multiplication while a squaring is computed and vice versa.

The decoded control–register signals and the a_i, q_i signals control the multiplexers Mux_B, Mux_1, Mux_2, Mux_{Res} and the clock enables of the registers. *N-Reg* is loaded only when the modulus is changed, *B-Reg* and *B+N-Reg* after each completion of Algorithm 2. Mux_1 feeds 0, B, N or $B + N$ into the adder according to the a_i and q_i bits. Mux_2 feeds N (for calculation of $N + B$) or the $u - 1$ most significant bits of the result plus the least significant result bit of the next unit (division by two / shift right) back into the adder. Mux_{Res} selects either the result of this unit or the one to the left to be stored into *Result-Reg*. Theoretically the implementation of the multiplexers and decoders would cost additional $4u + 4$ CLBs. The possibility of re–using registers for combinatorial logic allows some savings of CLBs. Mux_B and Mux_{Res} are implemented in the CLBs of *B-Reg* and *Result-Reg*, Mux_1 and Mux_2 partially in *N-Reg* and *B+N-Reg*. The resulting costs are approximately $3u+4$ CLBs per u–bit processing unit.

We compare this expense to the resources needed for a one bit unit implementation. The $B + N$ register would not be needed, as a ripple carry adder for such a small adder makes no sense. We would need a total of seven bit register space (N, B, a_i, q_i, control(2) and result) and a 4-bit input – 3 bit output (2 carries, result) adder. Together with one or two CLBs for decoding the control word and multiplexing, we would have a total of 6 or 7 CLBs per unit. A device that supports such a 1024 bit implementation would need $6.5 \cdot 10^3$ to $7.5 \cdot 10^3$ CLBs, including overhead.

4.3 Modular Multiplication

Figure 2 shows how the processing elements are connected to an array for computing an n – bit modular multiplication. Starting at the rightmost $unit_0$, the control word, a_i, and q_i are fed into their registers. The adder computes *Add-Reg-2* plus $B/N/B+N$ in one clock cycle according to

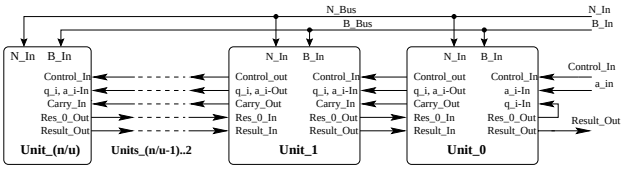

Figure 2. Systolic Array for modular multiplication

a_i and q_i. The least significant bit of the result is read back as q_{i+1} for the next computation. The resulting carry bit, the control word, a_i and q_i are pumped into the unit to the left, where the same computation takes place in the next clock cycle. In such a systolic fashion the control word, a_i, q_i, and the carry bits are pumped from right to left through the whole unit array. The division by two in Algorithm 2 leads also to a shift–right operation. The least significant bit of a unit's addition (Res_0) is always fed back into the unit to the right. After a modular multiplication is completed, the results are pumped from left to right through the units and consecutively stored in RAM for further processing.

A single processing element computes u bits of $R_{i+1} = (R_i + a_i \cdot B + q_i \cdot N)/2$ of Algorithm 2. In clock cycle i, $unit_0$ computes bits $0 \ldots u - 1$ of R_i. In cycle $i + 1$, $unit_1$ uses the resulting carry and computes bits $u \ldots 2u - 1$ of R_i. $Unit_0$ uses the right shifted (division by 2) bit u of R_i (Res_0) to compute bits $0 \ldots u - 1$ of R_{i+1} in clock cycle $i + 2$.

Clock cycle $i + 1$ is unproductive in $unit_0$ while waiting for the result of $unit_1$. This inefficiency is avoided by computing squares and multiplications in parallel according to Algorithm 2. Both p_{i+1} and z_{i+1} depend on z_i. We therefore store the intermediate result z_i in the B–Registers and feed z_i and p_i into the a_i input of the units for squaring and multiplication.

4.4 Modular Exponentiation

Figure 3 shows how the array of units is utilized for modular exponentiation.

First, the exponent E and the pre–computation factor $2^{2n+6} \bmod N$ are read from I/O and stored into RAM (*Exp* and *Prec*). Then the modulus N is read from I/O and fed on the u–bit wide N bus to the N–registers of the units. These steps have to be executed only if the system parameters need to be changed.

Next we read the X value from I/O, u bits per clock cycle, and store it into the dual port (DP) RAM Z. At the same time the precomputation factor $2^{2n+6} \bmod N$ is read from *Prec RAM* and fed u bits per clock cycle via the B bus to the B–registers of the units.

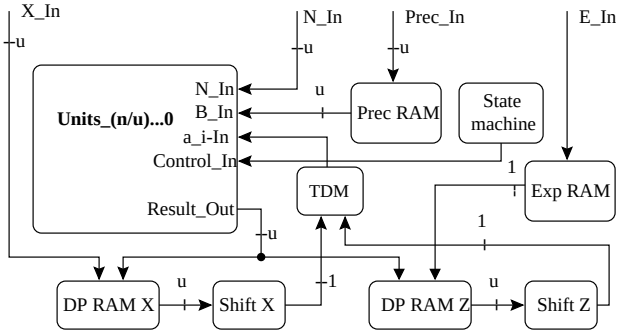

Figure 3. Design for a modular exponentiation

Execution of Algorithm 1 begins in parallel to the reading of X. Initially we have $P_0 = 1$ and $Z_0 = X$. First we multiply both values by the pre–computation factor $2^{2n+6} \bmod N$. This is done by time multiplexing X and $1, 0 \ldots 0$ in the time division multiplexing unit (TDM), pumping the result as a_i into the units and multiplying it by $2^{2n+6} \bmod N$ that is already stored in the B–registers. The results of the two pre–computations are stored into *DP RAM Z* and *DP RAM P*. Squaring is now straightforward: The intermediate result Z_i is always stored into the B–registers and into *DP RAM Z* and fed via a_i back into the units. Multiplication is done almost the same way. P_{i+1} is always computed by feeding P_i into the units, but the result is stored into *DP RAM P* only if the exponent e_i is equal to "1". In this way always the last stored P_i is pumped back into the units.

To eliminate the factor 2^{n+3} (see Section 4.3) from the result P_n, we compute a final Montgomery multiplication with inputs P_n and "1". $0, 0, \ldots 0, 1$ is stored via the B bus into the B–registers, P_n is fed from *DP RAM P* as a_i into the units.

A full modular exponentiation is computed in $2(n + 2)(n+4)$ clock cycles. That is the delay it takes from inserting the first u bits of X into the device until the first u result bits appear at the output. At that point, another X value can enter the device. With a latency of n/u clock cycles the last u bits appear on the output bus.

5 Methodology

In our implementation we adopted the following design flow approach that resulted in fast verification of gate level netlists as well as back annotated designs:

1. Design entry
2. Logic verification
3. Synthesis
4. Place and Route
5. Timing Verification

The entire design, with the exception of vendor specific soft macros, was entered in VHDL format. Once the design was developed in VHDL, boolean logic and major timing errors were verified by simulating the gate level description with Synopsys VHDL analyzer (vhdlan) and VHDL debugger (vhdldbx) version 1998.08. The next step involved the synthesis of the VHDL code with Synopsys Design Compiler (fpga_analyzer) version 1998.08. The output of this step was an optimized netlist describing the gate level design in XILINX format. The most time consuming step was the compilation of the synthesized design with the place and route tools available from Xilinx. This process was accomplished with the XILINX Design Manager tools version M1.5.19. The final step of the design flow was to verify the design once again but this time with the physical net, CLB, and pad delays introduced when the design was placed into a specific device. This was accomplished with the same test benches and simulation models that were used during the logic verification stage. Synopsys (vhdldbx) was used once again to verify back-annotated designs. The timing results from Section 6 were all computed by the Xilinx timing analyzer and verified by the Synopsis vhdl debugger. They were not verified with an actual chip.

6 Results

6.1 Modular Exponentiation

We implemented our design for various bit lengths and unit widths. Table 1 shows our results in terms of used CLBs (C), clock cycle time (T) and the time–area product (TA).

	256 bit			512 bit		
u	C [CLBs]	T [ns]	TA $[CLB \cdot ns]$	C [CLBs]	T [ns]	TA $[CLB \cdot ns]$
4	**1307**	**17.5**	**22875**	**2555**	**17.7**	**45223**
8	**1122**	**19.8**	**22215**	**2094**	**19.1**	**39995**
16	**1110**	**21.7**	**23870**	**2001**	**21.8**	**43621**

	768 bit			1024 bit		
u	C [CLBs]	T [ns]	TA $[CLB \cdot ns]$	C [CLBs]	T [ns]	TA $[CLB \cdot ns]$
4	**3745**	**19.1**	**71529**	**4865**	**19.2**	**93408**
8	**3132**	**19.4**	**60760**	**4224**	**23.4**	**98842**
16	**2946**	**21.6**	**63633**	**3786**	**23.7**	**89728**

Table 1. CLB usage, minimal clock cycle time, and time–area product of modular exponentiation architectures on Xilinx FPGAs

The majority of CLBs is expended in the units. In Section 4.2 we derived an approximation of $3u + 4$ CLBs per unit. The overhead consists mainly of RAM, dual port RAM, shift registers, counters and the state machine. An n bit RAM is implemented in $n/32$ CLBs, a dual port RAM in $n/16$ CLBs. Counters and their decoding for addressing RAM and dual port RAM are more costly for larger designs. On the other hand, we used the same state machine for all designs in Table 1.

The clock cycle time T in Table 1 is the propagation delay from *B-Reg* through Mux_1 and the carries of the adder to the registered carry, plus the setup time of the flip-flop. We compare this delay to the optimal cycle time calculated by the Xilinx timing analyzer; for a 4–bit unit the delay with optimal routing is 10.5ns (256 and 512 bit designs) and 12.7ns (768 and 1024 bit designs); for an 8–bit unit 11.2ns and 13.7ns and for a 16–bit unit 12.8ns and 15.5ns. The larger designs were implemented in larger FPGA devices featuring different delay specifications. Otherwise we expect the same cycle times for designs with the same unit size. The additional routing delay is between 50% and 80% above the optimal propagation delay. For designs up to 768 and 1024 ($u = 4$) bits it remains approximately constant; it deteriorates for 1024 bit designs with unit sizes $u = 8$ and $u = 16$. The same can be said about the place and route time: we experienced run–times of a couple of hours on a AMD–K6–2/300 MHz PC for designs up to 768 and 1024 ($u = 4$) bits, up to a week for the 1024 ($u = 8$ and $u = 16$) bit designs. Different design methods, such as hard–macros for a single unit, would probably improve routing delay and place and route time.

The time–area product shows that designs with 8–bit units are generally most efficient.

	512 bit		768 bit		1024 bit	
u	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]
4	**2555**	**9.38**	**3745**	**22.71**	**4865**	**40.50**
8	**2094**	**10.13**	**3123**	**23.06**	**4224**	**49.36**
16	**2001**	**11.56**	**2946**	**25.68**	**3786**	**49.78**

Table 2. CLB usage and execution time for a full modular exponentiation

Table 2 shows the application of our results to public–key schemes where the Chinese remainder theorem cannot be applied. A full modular exponentiation with an n bit exponent is computed in $2(n + 2)(n + 4)$ clock cycles.

6.2 Application to RSA

Table 3 shows our results from the tables above, applied to RSA. The encryption time is calculated for the F_4 exponent, requiring $2 \cdot 19(n + 4)$ clock cycles. Using the F_4 exponent, only one multiplication can be calculated in parallel to a squaring.

	512 bit		1024 bit	
u	C CLBs	T [ms]	C CLBs	T [ms]
4	**2555**	**0.35**	**4865**	**0.75**
8	**2094**	**0.37**	**4224**	**0.91**
16	**2001**	**0.43**	**3786**	**0.93**

Table 3. Application to RSA: Encryption

For decryption we apply the Chinese remainder theorem. We either decrypt n bits with an $n/2$ bit architecture serially, or with two $n/2$ bit architectures in parallel. The first approach uses only half as many resources, the later is twice as fast.

	512 bit $2 \cdot 256$ serial		512 bit $2 \cdot 256$ parallel		1024 bit $2 \cdot 512$ serial		1024 bit $2 \cdot 512$ parallel	
u	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]	C CLBs	T [ms]
4	**1307**	**4.69**	**2533**	**2.37**	**2555**	**18.78**	**4995**	**10.18**
8	**1122**	**5.31**	**2183**	**2.56**	**2094**	**20.26**	**4153**	**12.41**
16	**1110**	**5.82**	**2131**	**2.92**	**2001**	**23.12**	**3922**	**12.52**

Table 4. Application to RSA: Decryption

6.3 Comparison and Outlook

We compare our fastest RSA 512/1024 bit designs of Table 4 to the fastest soft- and hardware solutions we found in the literature [17, 13, 21]. Our 2.37ms decryption time is about four times faster than the 512 bit software implementation (9.1ms) on a 150MHz Alpha [13]. The fastest 1024 bit software implementation [21] of 43.3ms running on a PPro–200 based PC is about 4 times slower than our best result (10.2ms). The fastest reported hardware design [17] (1.7ms for a 512 bit modulus and 5.2ms for a 970 bit modulus) is a factor 1.4/1.7 faster than ours (9.1ms for a 970 bit modulus). A drawback of the solution in [17] is, however, that the binary representation of the modulus is hardwired into the logic representation so that the architecture has to be reconfigured with every new modulus. The user of such an implementation needs to own the full development tools for synthesis, placing and routing of FPGAs, if RSA with different moduli should be executed. Our design stores the modulus, the exponent and the pre–computation factor in registers and RAM. A second advantage of our design is that it is implemented into one device instead of a matrix of 16 devices. Using currently available FPGA technology, however, the design [17] would probably also fit in a single device.

To improve our design in terms of speed, three approaches can be taken:

1. Computation of one bit per processing unit (25% improvement estimated).
2. Montgomery multiplication with a radix $r = 2^u$, $u \geq 2$. Computation of a full modular exponentiation in $O(n^2/u)$ cycles instead of $O(n^2)$.

Both approaches have the major disadvantage that considerably more resources will be used. We will concentrate our future research on trying to implement a higher radix design according to approach 3). The challenge at hand is to accommodate simplifications as proposed in [6] to systolic array and FPGA technology.

References

[1] J. Bajard, L. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7):766–76, July 1998.

[2] T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–65, May 1989.

[3] E. Brickell. A survey of hardware implementations of RSA. In *Advances in Cryptology — CRYPTO '89*, pages 368–70. Springer-Verlag, 1990.

[4] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, July 1993.

[5] W. Gai and H. Chen. A systolic linear array for modular multiplication. In *2nd International Conference on ASIC*, pages 171–4, 1996.

[6] H.Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings 12th Symposium on Computer Arithmetic*, pages 193–9, 1995.

[7] K. Iwamura, T. Matsumoto, and H. Imai. Montgomery modular-multiplication method and systolic arrays suitable for modular exponentiation. *Electronics and Communications in Japan, Part 3*, 77(3):40–51, March 1994.

[8] D. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[9] P. Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on Computers*, 43(8):892–8, August 1994.

[10] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–21, April 1985.

[11] J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public–key cryptosystem. *Electronics Letters*, 18:905–7, October 1982.

[12] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–6, Feb. 1978.

[13] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proceedings 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, 1993.

[14] D. R. Stinson. *Cryptography, Theory and Practice*. CRC Press, 1995.

[15] N. Takagi. A radix-4 modular multiplication hardware algorithm efficient for iterative modular multiplications. In *Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 35–42, 1991.

[16] A. Tiountchik. Systolic modular exponentiation via Montgomery algorithm. *Electronic Letters*, 34(9):874–5, April 1998.

[17] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, Mar 1996.

[18] C. Walter. Fast modular multiplication using 2-power radix. *International Journal of Computer Mathematics*, 39(1–2):21–8, 1991.

[19] C. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–8, March 1993.

[20] P. Wang. New VLSI architectures of RSA public key cryptosystems. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, volume 3, pages 2040–3, 1997.

[21] E. D. Win, S. Mister, B. Preneel, and M. Wiener. On the performance of signature schemes based on elliptic curves. In *Algorithmic Number Theory Symposium III*, pages 252–266. Springer-Verlag, 1998.

[22] Xilinx Inc., San Jose, CA. *The Programmable Logic Data Book*. 1996.

[23] J. Yong-Yin and W. Burleson. VLSI array algorithms and architectures for RSA modular multiplication. *IEEE Transactions on VLSI Systems*, 5(2):211–17, June 1997.