High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware*

Thomas Blum† Christof Paar
tblum@ece.wpi.edu christof@ece.wpi.edu
ECE Department
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609, USA

to appear in the IEEE Transactions on Computers

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. 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 a high–radix Montgomery modular multiplication algorithm with a new systolic array design. The designs are flexible, allowing any choice of operand and modulus. The new architecture also allows the use of high radices.

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 commercially available FPGAs. We found that 1024 bit RSA decryption can be done in 3.1 ms with our fastest architecture.

Keywords: Montgomery, modular arithmetic, FPGA, exponentiation, RSA, systolic array

*The research was supported in part through NSF CAREER award #CCR-9733246.
†The author is now with Ergon Informatik, Switzerland.

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. Our goal was to develop a modular exponentiation architecture which is optimized for modern FPGAs, based on a high–radix Montgomery's modular reduction scheme and a novel systolic array architecture. The design should be performance optimized and use considerably fewer logic resources than many other systolic array architectures for modular arithmetic.

As will be shown, we met the stated goals. Our design outperforms all reported software and FPGA implementations of the public-key algorithms listed, and can be realized on a single commercially available FPGA.

This contribution is structured as follows. In Section 2, we summarize some of the previous work on modular exponentiation. Section 3 describes algorithms for modular exponentiation. Section 4 outlines our architecture for modular exponentiation. Section 5 of this contribution shows the timing and area results obtained. In Sections 6 and 7, we compare our results to previous work and draw some conclusions.

2 Previous Work

the following, we will summarize relevant previous work in the field of modular multiplication. Most presented approaches are based on the algorithm proposed by Peter Montgomery in 1985 [1], either in conjunction with a redundant radix number system [2, 3, 4, 5] or in a systolic array architecture, e.g. [6].

In [4] Montgomery's modular multiplication algorithm is adapted for an efficient hardware implementation. A gain in speed results from a faster clock, due to simpler combinatorial logic. Compared to previous techniques, a speed-up factor of two is reported. The Research Laboratory of Digital Equipment Corp. in Paris implemented modular exponentiation architectures on FPGAs [2, 3]. They utilized an array of 16 XILINX 3090 FPGAs. Their design uses several speed-up methods [3] including the Chinese remainder theorem, asynchronous carry completion adder, and a windowing exponentiation method. The problem of using high radices in Montgomery's modular multiplication algorithm is the more complex determination of the quotient. This behavior made a pipelined execution of the algorithm impossible. Reference [5] rewrites the algorithm and avoids thereby any operation involved in the quotient determination. The necessary pre–computation has to be done only once for a given modulus. Our work presented in this contribution extends [5] to reconfigurable hardware and a systolic array architecture.

There have been a number of proposals for systolic array architectures for modular arithmetic. However, no implementations have been reported to our knowledge. Reference [6] describes an architecture based on one row of processing elements and a radix of two. Squarings and multiplications are computed in parallel. The system requires n systolic processing elements for an n–bit modular exponentiation, and the resulting execution time is $2n^2$ clock cycles.

A detailed description of the software implementation option of the Montgomery algorithm is provided in [7].

Section 6 lists the fastest software and hardware implementations presented in technical literature and compares them to our architectures.

3 Preliminaries: Modular Exponentiation

In this section we review a parallel version of the square and multiply algorithm, which is the standard algorithm for exponentiation. Secondly we review a version of Montgomery's modular multiplication algorithm which is well suited for hardware implementations.

3.1 Square and Multiply Algorithm

The public–key schemes described in Section 5 are based on modular exponentiation or repeated point addition. Both operations are in their most basic forms done by the following version of the square and multiply algorithm [6]:

Algorithm 3.1 *computes* $P = X^E \bmod M$, *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. $Z_{i+1} = Z_i^2 \bmod M$

4. *IF* $e_i = 1$ *THEN* $P_{i+1} = P_i \cdot Z_i \bmod M$

ELSE $P_{i+1} = P_i$

5. *END FOR*

Algorithm 3.1 takes $2n$ operations in the worst case and $1.5n$ on average. In this version of the square and multiply algorithm there exists no data dependency between the modular squaring and multiplying operations. Hence, both operations can be performed in parallel.

3.2 Montgomery Modular Multiplication

As shown in the previous section, modular exponentiation is reduced to a series of modular multiplications and squaring steps. The algorithm for modular multiplication described below has been proposed by P. L. Montgomery in 1985 [1]. It is a method for multiplying two integers modulo M, while avoiding division by M. The idea is to transform the integers in m-residues and compute the multiplication with these m-residues. Finally we transform back to the normal representation. This approach is only beneficial if we compute a series of multiplications in the transform domain (e.g., modular exponentiation).

In [8] we presented a version of Montgomery's algorithm optimized for a radix two hardware implementation. The version shown below was taken from [5]. It is suitable for high radix hardware implementations of modular exponentiation.

Algorithm 3.2 *[5] MONT(A,B):Montgomery Modular Multiplication for computing* $A \cdot B \bmod M$*, where* $M = \sum_{i=0}^{m-1}(2^k)^i m_i$*,* $m_i \in \{0, 1 \ldots 2^k - 1\}$*;*
$\tilde{M} = (M' \bmod 2^k)M$*,* $\tilde{M} = \sum_{i=0}^{m}(2^k)^i \tilde{m}_i$*,* $\tilde{m}_i \in \{0, 1 \ldots 2^k - 1\}$*;*
$B = \sum_{i=0}^{m+1}(2^k)^i b_i$*,* $b_i \in \{0, 1 \ldots 2^k - 1\}$*;*
$A = \sum_{i=0}^{m+2}(2^k)^i a_i$*,* $a_i \in \{0, 1 \ldots 2^k - 1\}$*,* $a_{m+2} = 0$*;*
$A,B < 2\tilde{M}$*;* $4\tilde{M} < 2^{km}$*;* $M' = -M^{-1}$

1. $S_0 = 0$

2. *FOR* $i = 0$ *to* $m + 2$ *DO*

3. $q_i = (S_i) \bmod 2^k$

4. $S_{i+1} = (S_i + q_i\tilde{M})/2^k + a_i B$

5. *END FOR*

The result of Algorithm 3.2 is $S_{m+3} = ABR^{-1} \bmod M$ where $R = 2^{k(m+2)} \bmod M$. To get the desired result $S_{m+3} = AB \bmod M$ a pre-computation and post–computation step need to be performed: Pre–multiply all inputs by the factor $2^{2k(m+2)} \bmod M$. Thus every intermediate result carries a factor $2^{k(m+2)}$. We just need to Montgomery multiply the final result by 1 to eliminate that factor.

4 A New Architecture

In this section we describe our new architecture. The goal was to design a speed efficient architecture using a systolic array which realizes Algorithm 3.2. As target devices we use the Xilinx XC4000 family [9] as this appears to be a good representative of a modern FPGA architecture. An XC4000 CLB consists of three look–up tables, two flip–flops and programmable multiplexers. Two boolean functions of four inputs can be computed in one CLB. Note that Altera, Lucent, and Actel have FPGA families with related architectures and we expect that our design is suitable for those too. For a more detailed description and timing diagrams of the design described in this section refer to [10]. The results of the actual implementation of the architecture will be described in Section 5.

4.1 Design Overview

In [8] we described an architecture which was optimized in terms of resource usage and uses a radix of 2. In order to speed–up the design we describe in the following a high radix version of Montgomery's algorithm (3.2) which reduces the amount of cycles per modular multiplication.

One of the major problems when implementing Algorithm 3.2 is computing multiples of B and $\tilde{M}$ in step 4. Reference [5] proposes a multiplexer network. This approach is not suitable for a systolic array implementation into FPGA because of the following reasons:

1. For a radix of 2^2 the multiplexer could be implemented in one CLB per bit length, but already a radix of 2^4 uses more than four CLBs per bit. This would result in unrealistically large CLB counts for secure bit lengths in cryptographic applications.

2. In a systolic array we typically compute k bits per processing element. With a multiplexer solution the internal bit length becomes $2k$ resulting in twice the cost for adders and registers.

To avoid the doubling of the internal bit length of a unit the following approach which is optimized for the CLB architectures at hand can be taken:

- Pre-compute the multiples of B and $\tilde{M}$ at the beginning of the execution of Montgomery's algorithm and store the results for further use.

- Let the carries of this pre–computations propagate to the units to the left.

If a unit processes k bits, the stored multiples will also have k bits and the internal bit length will not exceed $k+2$ bits (addition of 3 operands). The cost is additional 2^k clock cycles for calculating the 2^k multiples of B. For small k values, this expense is negligible compared to the total amount of $2(m+3)$ cycles for the whole algorithm. As storage elements we can either use registers or RAM elements. For k larger than 2, registers are not suitable as they utilize one CLB per 2 stored bits. RAM elements are very efficient up to an address width of 4 bits [9]. Their implementation requires only one CLB per two bits data width (2 CLBs for a 16×4 bit RAM). The resource requirements grow rapidly, though, for larger address width. A 64×6 bit implementation ($k = 6$) utilizes 18 CLBs, a 256×8 bit implementation ($k = 8$) utilizes 96 CLBs. Both would result in unrealistically large CLB counts for secure bit length. Additionally the $2^6 - 1$ or even $2^8 - 1$ clock cycles for pre–computing the multiples of B are not negligible any more. To achieve an optimal time–area product we implemented therefore an architecture with a radix $r = 2^4$ and compute 4 bits per processing element.

Similar to the approaches in [6] and [8], we use the square and multiply Algorithm 3.1 and compute squares and multiplications in parallel.

Our design can be divided hierarchically into three levels.

Processing Element Computes four bits of a modular multiplication.

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

Modular Exponentiation Combines modular multiplications to a modular exponentiation according to Algorithm 3.1.

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

4.2 Processing Elements

Figure 1 shows the implementation of a processing element. The *$B/\tilde{M}$ Adder* does the pre–computation of the 15 multiples of B and $\tilde{M}$. These values are stored into *B-RAM* and *$\tilde{M}$-RAM*. *B+$\tilde{M}$-Adder* and *S+B+$\tilde{M}$-Adder* add the previous result S_i to the relevant multiples of B and $\tilde{M}$. Additionally we need several registers to store the control word, a_i, q_i and the result of a modular multiplication.

The registers need a total of 14 CLBs, the adders 13 CLBs and the RAM blocks 4 CLBs. The possibility of re–using registers for combinatorial logic allows some savings of CLBs. Thus a processing element utilizes a total of 24 CLBs, which is equal to 6 CLBs per processed bit.

For a detailed description of the processing elements, please refer to [10].

$$S_{i+1} = (S_i + q_i \cdot \tilde{M})/2^k + a_i \cdot B$$

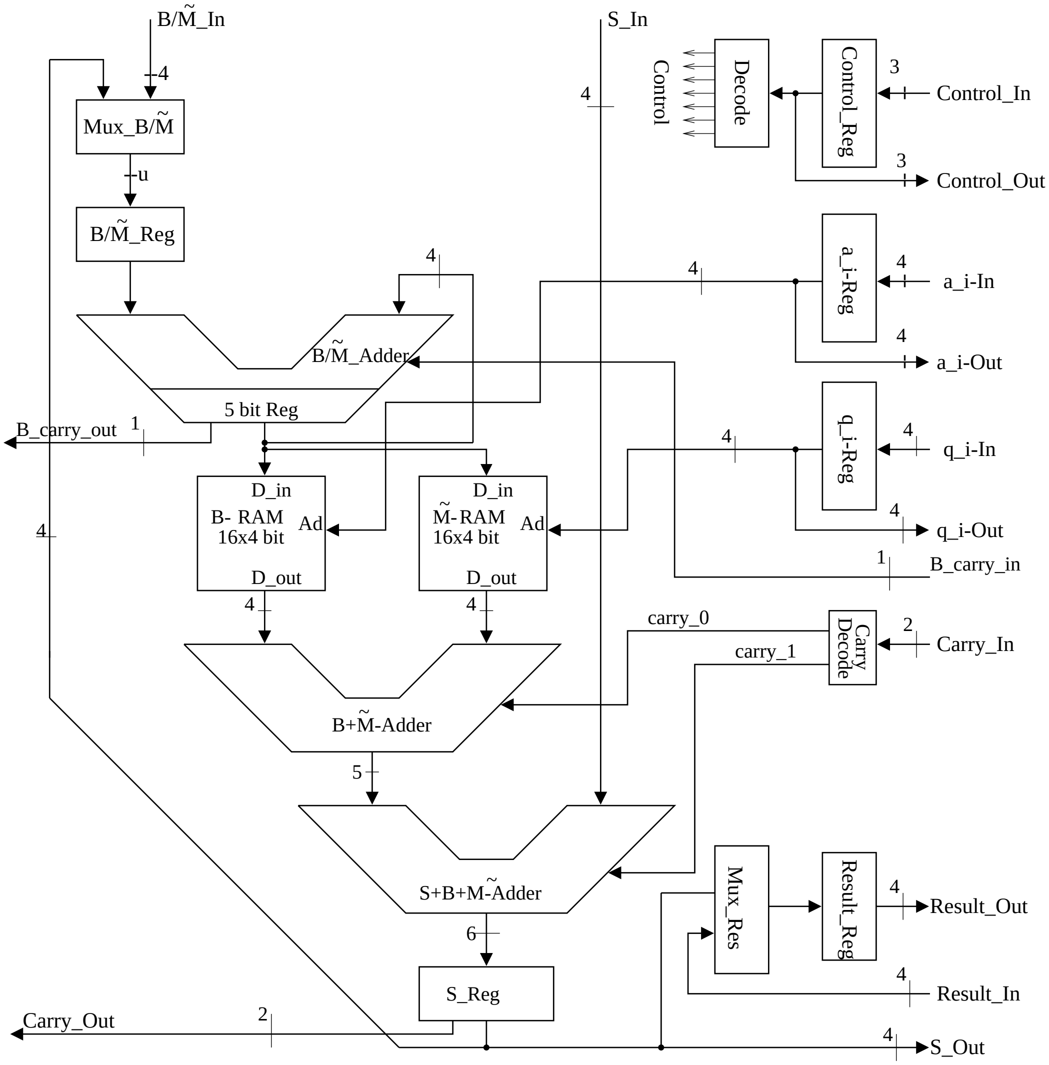

Figure 1: Processing element (unit)

4.3 Modular Multiplication

Figure 2 shows how the processing elements are connected to an array for computing a full size modular multiplication. To compute $S_{i+1} = (S_i + q_i \cdot \tilde{M})/16 + a_i \cdot B$ with operand $B = \sum_{i=0}^{m+1} 16^i \cdot b_i$, we need $m+3$ units. Units $1 \ldots m+1$ are designed as described in Section 4.2. *Unit*$_0$ does not have a *B-RAM* as B is not shifted by 4 bits (division by 16)

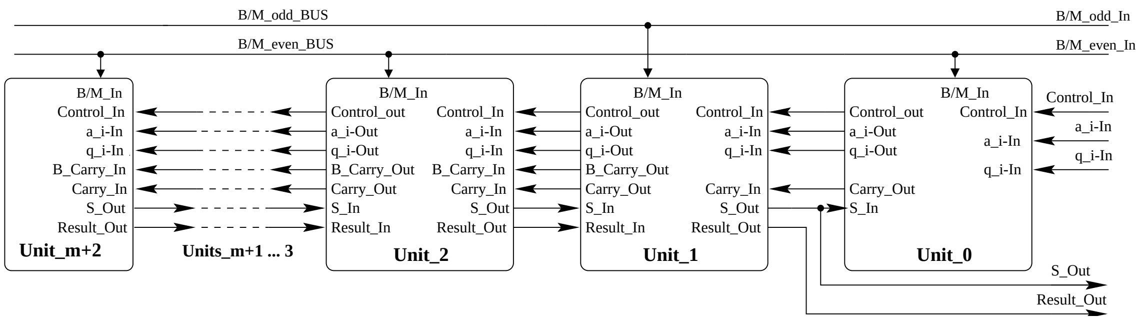

Figure 2: Systolic array for modular multiplication

in above mentioned equation. The four result bits $S_{3...0}$ are always equal to zero according to the properties of Montgomery's algorithm. $Unit_{m+2}$ on the other hand does not have an $\tilde{M}$-*RAM*. It processes the most significant bit of B and the temporarily occurring overflow of S_{i+1}.

The inputs and outputs of the units are connected to each other in the following way. The control word, q_i and a_i are pumped from right to left through the units. The result is pumped from left to right. The *carry-out* signals are fed to the *carry-in* inputs to the left. Output *S_out* is always connected to input *S_in* of the unit to the right. This represents the division by 16 of the computation.

For a detailed description of a modular multiplication, please refer to [10]. A squaring or multiplication takes $2m + 20$ cycles.

4.4 Modular Exponentiation

Figure 3 shows how the array of units is utilized for modular exponentiation. At the heart of our design is a finite state machine (FSM) which is responsible for loading the system parameters, and executing the pre-computation, the exponentiation, and the post-computation as needed in Algorithm 3.1 and Algorithm 3.2. The FSM is clocked at half the clock rate. The same is true for loading and reading the RAM and DP RAM elements. This measure makes sure the maximal propagation time is in the units. Thus the minimal clock cycle time and the resulting speed of a modular exponentiation relates to the effective computation time and not to the computation of overhead.

The design for modular exponentiation works similarly as described in [8]. Additionally, we need a q-counter and an a-counter for computing multiples of $\tilde{M}$ and B. The output of the counters $(0, 1, \cdots, 15)$ is fed via q_i and a_i to the address input of $\tilde{M}$-*RAM* and *B-RAM* while the multiples of $\tilde{M}$ and B are being computed (see Section 4.2).

A full modular exponentiation is computed in $(n + 2)(2m + 20)$ clock cycles. That is the delay it takes from inserting the first 4 bits of X (Algorithm 3.1) into the device, until the

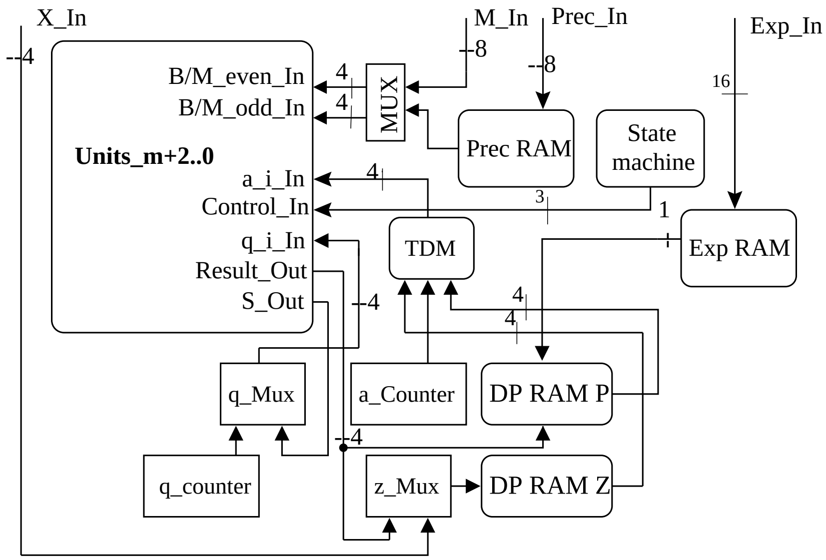

Figure 3: Design for a modular exponentiation

first 4 result bits appear at the output. At that point, another X value can enter the device. With an additional latency of $m + 2$ clock cycles the last 4 bits appear on the output bus.

5 Results

this section we present the results of the architecture described in Section 4. As target devices we used the Xilinx XC40250XV, speedgrade -09, 8464 CLBs, for the larger designs (> 5000 CLBs), and the XC40150XV, speedgrade -08, 5184 CLBs, for the smaller designs. The designs were developed in VHDL and synthesized with Synopsis Design Compiler (version 1998.08). The place and route process of the synthesized designs was accomplished with the Xilinx Design Manager tools (version M1.5.9). The timing results were computed by the Xilinx timing analyzer from the placed and routed designs, and verified by the Synopsis VHDL debugger. They were not verified with an actual chip. Note that the Xilinx tools assume the absolute worst possible operating conditions — highest possible operating temperature, lowest possible supply voltage, and worst-case fabrication tolerance for the speed grade of the FPGA [11].

It is highly relevant to compare our new architecture with a radix 2 Montgomery algorithm implementation on an FPGA. For this reason, we include the fastest results from our earlier study [8] in the tables of this section.

5.1 Modular Exponentiation

Table 1 shows our results in terms of used CLBs (C) and the clock cycle time (T). Operands and exponents have the same bit lengths in all cases.

Table 1: CLB usage and minimal clock cycle time of modular exponentiation architectures on Xilinx FPGAs

	160 bit		256 bit		512 bit		768 bit		1024 bit	
Radix	C [CLBs]	T [ns]	C [CLBs]	T [ns]	C [CLBs]	T [ns]	C [CLBs]	T [ns]	C [CLBs]	T [ns]
2 [8]	**951**	**17.3**	**1307**	**17.5**	**2555**	**17.7**	**3745**	**19.1**	**4865**	**19.2**
16	**1219**	**20.8**	**1818**	**21.3**	**3413**	**20.7**	**5071**	**20.1**	**6633**	**21.9**

The majority of CLBs is expended in the units, that is 6 CLBs per bit of the modulus. The overhead consists mainly of RAM, DP RAM, counters, registers, and the state machine. Between 300 CLBs for the 160–bit design and 500 CLBs for the 1024–bit design are used for overhead.

The clock cycle time T in Table 1 is the access delay $q_i \rightarrow D\_out$ of the $\tilde{M}$-RAM or $a_i \rightarrow D\_out$ of the B-RAM plus the delay through the two adders to the registered carry in S_Reg, plus the setup time of the flip-flop (see Figure 1). We compare this delay to the optimal cycle time calculated by the Xilinx timing analyzer; for the smaller designs (160–512 bits) the delay with optimal routing is 14.7 ns, for the larger designs 15.7 ns. The larger designs were implemented in larger FPGA devices featuring different delay specifications. Otherwise we expected the same cycle times for all designs as the difference between designs lies in the amount of units. The additional routing delay is about 30% above the optimal propagation delay.

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

	512 bit		768 bit		1024 bit	
Radix	C [CLBs]	T [ms]	C [CLBs]	T [ms]	C [CLBs]	T [ms]
2 [8]	**2555**	**9.38**	**3745**	**22.71**	**4865**	**40.05**
16	**3413**	**2.93**	**5071**	**6.25**	**6633**	**11.95**

Table 2 shows our results for a full length modular exponentiation, i.e., an exponentiation where base, exponent, and modulus have all the same bit length. A full modular exponentiation with an n bit exponent and an m digit modulus is computed in $2 \cdot (n+2)(m+10)$ clock cycles.

5.2 Application to RSA

RSA was proposed by Rivest, Shamir and Adleman [12] in 1978. 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 $M = p \times q$ and an exponent E such that $E = D^{-1} \bmod (p-1)(q-1)$.

Table 3 shows our RSA encryption results. The encryption time is calculated for the Fermat prime $F_4 = 2^{16} + 1$ exponent [13], requiring $2 \cdot 19(m+4)$ clock cycles for the radix 2 design [8], and $2 \cdot 19(m+10)$ clock cycles if the radix 16 design is used. Please note that $M = \sum_{i=0}^{m-1} (2^k)^i m_i$ for $k = 1$ in Design [8], and $k = 4$ in the design presented in this contribution.

Table 3: Application to RSA: Encryption

	512 bit		1024 bit	
Radix	C [CLBs]	T [ms]	C [CLBs]	T [ms]
2 [8]	**2555**	**0.35**	**4865**	**0.75**
16	**3413**	**0.11**	**6633**	**0.22**

For decryption we apply the Chinese remainder theorem [14]. We either decrypt m bits with an $m/2$ bit architecture serially, or with two $m/2$ bit architectures in parallel. The first approach uses only half as many resources, the latter is almost twice as fast. We lose a little time here because of the slower delay specifications of the larger devices.

Table 4: Application to RSA: Decryption

	512 bit 2×256 serial		512 bit 2×256 parallel		1024 bit 2×512 serial		1024 bit 2×512 parallel	
Radix	C [CLBs]	T [ms]	C [CLBs]	T [ms]	C [CLBs]	T [ms]	C [CLBs]	T [ms]
2 [8]	**1307**	**4.69**	**2614**	**2.37**	**2555**	**18.78**	**5110**	**10.18**
16	**1818**	**1.62**	**3636**	**0.79**	**3413**	**5.87**	**6826**	**3.10**

6 Comparison to Previously Reported Implementations

We compared our fastest RSA 512/1024 bit designs of Table 4 to the fastest soft- and hardware solutions we found in the literature [2, 3, 15]. Our 0.8 ms decryption time is about 11 times faster than the 512 bit software implementation (9.1 ms) on a 150MHz Alpha [3]. The fastest 1024 bit software implementation [15] of 43.3 ms running on a PPro–200 based PC is about 14 times slower than our best result (3.1 ms).

Most reported hardware implementations of modular arithmetic are somewhat dated, making a fair comparison difficult. It is nevertheless interesting to look at previously reported performances. The fastest reported FPGA design [2] (1.7 ms for a 512 bit modulus and 5.2 ms for a 970 bit modulus) is a factor 2.1/1.8 slower than ours, which requires 2.8 ms for a 970 bit modulus. It is possible, though, that their solution upgraded to currently available FPGA technology, would run considerably faster. A drawback of the solution in [2] 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.

7 Conclusions

A modular exponentiation architecture was derived that combines a high radix version of Montgomery's algorithm with a novel systolic array architecture. The design was optimized for modern FPGAs. For an optimal speed area trade–off a radix of 16 was chosen. We showed that it is possible to implement 1024–bit modular exponentiation on a single commercially available FPGA. 1024–bit RSA is performed in 3.1 ms using a clock rate of 45.6 MHz and an area of 6826 CLB's on a Xilinx XC40250XV, speedgrade -09. These performances are better than all previously reported implementations presented in technical literature.

References

[1] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–21, April 1985.

[2] 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*, vol. 4, pp. 56–69, Mar 1996.

[3] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," in *Proceedings 11th IEEE Symposium on Computer Arithmetic*, pp. 252–259, 1993.

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

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

[6] P. Kornerup, "A systolic, linear-array multiplier for a class of right-shift algorithms," *IEEE Transactions on Computers*, vol. 43, pp. 892–8, August 1994.

[7] C. K. Koc, T. Acar, and B. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, pp. 26–33, June 1996.

[8] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proceedings 14th Symposium on Computer Arithmetic*, pp. 70–7, 1999.

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

[10] T. Blum, "Modular exponentiation on reconfigurable hardware," Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1999.

[11] P. Alfke, "Xilinx M1 Timing Parameters." Electronic Mail Personal Correspondence, December 1999.

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

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

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

[15] 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*, pp. 252–266, Springer-Verlag, 1998.