

Acceleration of Modular Exponentiation on System-on-a-Programmable-Chip

Panu Hämäläinen, Ning Liu, Marko Hännikäinen, and Timo D. Hämäläinen
Institute of Digital and Computer Systems
Tampere University of Technology
Tampere, Finland

Abstract—Computing modular exponentiations with long integers is required in a number of security protocols. Since security procedures typically consume large amount of processing capacity in network devices, efficient implementations are needed. As a solution, this paper presents an exponentiation accelerator suited for efficient processing in security protocols using public key schemes, such as TLS and IPsec. The accelerator is implemented on a System-on-a-Programmable-Chip, partitioned into software control and hardware processing. Compared to previous radix-2 designs, significantly higher performance is achieved. The design computes a full exponentiation in $(n+k)(n+4)$ clock cycles, in which n is the bit length of the modulus and the exponent and k is the number of ones in the binary representation of the exponent. In the average case, the design executes the exponentiation 25% faster than the previous hardware designs at equal clock speeds. The proposed exponentiation control and 1-cycle processing mode can also be utilized for improving higher radix designs.

I. INTRODUCTION

One of the most utilized operations in security protocols is modular exponentiation of long integers. It is required in most public key cryptosystems for a number of purposes, including encryption, authentication, key agreement, and digital signing. For example, Transport Layer Security (TLS) and Internet Protocol Security (IPsec) require the exponentiations in their widely utilized RSA and Diffie-Hellman schemes. Authentication methods using large exponentiations are also extending to wireless technologies for improving network access control.

Large exponentiations are recognized as expensive operations. For example, in [8] it is measured that they take 90% of the processing time in the TLS handshake. Security procedures are also generally among the tasks requiring most processing capacity in network devices. Especially, in wireless devices overall processing limitations and energy consumption can be significantly decreased with efficient implementations. On the other hand, e.g., virtual private networks require high-speed implementations in busy corporate firewalls. With an efficient design, compromising the security level and degenerating response times due to limited processing resources is avoided. As a solution, this paper presents an accelerator for large modular exponentiations.

Altera Excalibur System-on-a-Programmable-Chip (SoPC) [1], consisting of programmable logic integrated with an ARM processor, is used as the implementation platform for the exponentiation accelerator. The exponentiation processing is implemented in hardware and the control in software. In [5] it is estimated that with a dedicated hardware the cycle count

of a full-width modular exponentiation can be reduced by the factor of several tenfolds from software, which is also proven with the results of this work. As the expensive computations are completely performed by the hardware, the software part of the accelerator can be run even in a microcontroller with very limited capabilities without decreasing the performance.

The paper continues the work of [4], in which an exponentiation hardware was designed for accelerating Secure Remote Password protocol (SRP) [7]. Similarly to public key cryptosystems, most of the processing time in SRP is used for the exponentiations [5]. SRP has been proposed to be used in the TLS handshake as well as in the Wireless Local Area Network (WLAN) authentication. In our previous work the execution time of SRP using 256-bit exponents was reduced to 1/4 compared to ARM software [4]. With longer exponents the performance increase is even higher. Compared to our previous work, improvements in terms of performance as well as design scalability and reusability are achieved in this work.

II. MODULAR MULTIPLICATION AND EXPONENTIATION

Without applying a proper algorithm, modular exponentiations of large integers can consume substantially large amount of time and space. Generally, a square-and-multiply method with a reduction algorithm is utilized. The power is calculated as a composition of squarings, multiplications, and reductions, e.g., $a^5 \bmod N = ((a^2 \bmod N)^2 \bmod N)a \bmod N$.

Each squaring and multiplication is performed with a modular multiplication algorithm. A widely used algorithm is Montgomery Multiplication (MM) [6]. It is particularly suitable for hardware since it only requires additions and shifts. Even though pre-processing and post-processing are needed, MM is advantageous when the multiplication is performed several times with the same modulus – as in exponentiation.

Algorithm 1 presents the radix-2 MM algorithm utilized in this work [2]. The radix refers to that one bit of the multiplier is processed at a time. Higher radices require less clock cycles but they are also more complex and consume more hardware resources [3]. However, the accelerator is designed considering its extension to higher radices in the future. The extra term r^{-1} is eliminated by first performing MM on the inputs (α, β) with $r^2 \bmod N$ and then on the result with 1.

The MM algorithm is utilized in the square-and-multiply exponentiation presented as Algorithm 2 [2]. In this work the exponent is processed one bit at a time. Similarly to higher

Algorithm 1 Montgomery multiplication (radix-2)

Inputs: α, β, N ; Output: $\delta_{n+3} = \alpha\beta r^{-1} \bmod N$
 $\alpha = \sum_{i=0}^{n+2} a_i 2^i, a_i \in \{0, 1\}, a_{n+1} = a_{n+2} = 0$
 $\beta = \sum_{i=0}^{n-1} b_i 2^i, b_i \in \{0, 1\}$
 $N = \sum_{i=0}^{n-1} m_i 2^i, m_i \in \{0, 1\}$
 $N \bmod 2 = 1; A, B < 2N; r = 2^{n+2}$

- 1: $\delta_0 = 0$
- 2: **for** $i = 0$ to $n + 2$ **do**
- 3: $q_i = \delta_i \bmod 2$
- 4: $\delta_{i+1} = (\delta_i + q_i N) / 2 + a_i \beta$
- 5: **end for**

radices, processing more exponent bits decreases the execution time but increases the resource consumption. Each round in Algorithm 2 uses MM twice, once for squaring and conditionally once for multiplication. As the squaring and multiplication are independent, they can be performed concurrently. When combined with MM, the pre-transformations are computed for X and P_0 and the post-transformation for P_l .

III. EXPONENTIATION ACCELERATOR DESIGN

The exponentiation accelerator is implemented on an Altera Excalibur SoPC and partitioned into software control and hardware processing. The hardware consist of a MM entity and control logic for utilizing it in square-and-multiply exponentiations. The overall processing is controlled by the software. Since the processing requirements for the software part of the accelerator are very low, it can be run even in a microcontroller with very limited processing capabilities.

A. Implementation Platform

The implementation platform of the accelerator is the Altera Excalibur EPXA10 DDR development kit [1], shown in Fig. 1. The main component of the board is the EPXA10F1020C2 SoPC, which consists of an integrated ARM922T processor core and the Altera APEX20KE-like Programmable Logic Device (PLD). The PLD contains a large number of programmable Logic Elements (LE) and Embedded System Blocks (ESB) are provided for implementing a variety of memory functions. ARM9 and the PLD are connected through two

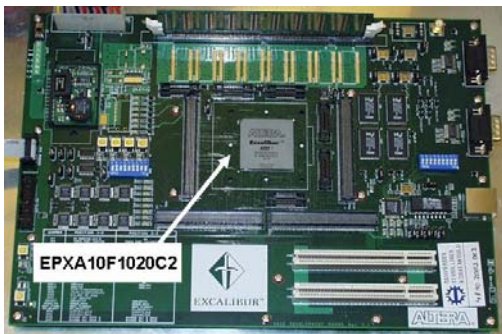


Fig. 1. Implementation platform of the accelerator.

Algorithm 2 Square-and-multiply modular exponentiation

Inputs: X, E, N ; Output: $P_l = X^E \bmod N$
 $E = \sum_{j=0}^{l-1} e_j 2^j, e_j \in \{0, 1\}$

- 1: $P_0 = 1, Z_0 = X$
- 2: **for** $j = 0$ to $l - 1$ **do**
- 3: $Z_{j+1} = Z_j^2 \bmod N$ (square)
- 4: **if** $e_j = 1$ **then**
- 5: $P_{j+1} = (P_j Z_j) \bmod N$ (multiply)
- 6: **else**
- 7: $P_{j+1} = P_j$
- 8: **end if**
- 9: **end for**

Advanced Microcontroller Bus Architecture (AMBA) High-performance Bus (AHB) bridges, a shared Dual-Port RAM (DPRAM), and interrupt lines.

B. Montgomery Multiplication Hardware

MM is implemented as a systolic array of Processing Units (PU) in the PLD. A PU, depicted in Fig. 2, processes an 8-bit piece of the line 4 of Algorithm 1 at a single clock cycle. A PU operates as follows:

- 1) An 8-bit piece of the modulus N is loaded to N_reg . Also, the next higher bit of N is loaded to n_reg .
- 2) An 8-bit piece of the multiplicand β is loaded to β_reg .
- 3) The signals a_i and q_i are input. The two additions are computed and the result is stored in δ_i_reg . The adder carries ($c1_out, c2_out$) are output to the higher PU and the lowest bit of the result ($\delta_i^0_out$) to the lower PU. Also, a_i, q_i , and the control signals are forwarded to the higher PU.
- 4) Step 3 is repeated until all the bits of α have been processed. The 8-bit piece of the MM result is written to res_reg and also conditionally stored in β_reg .

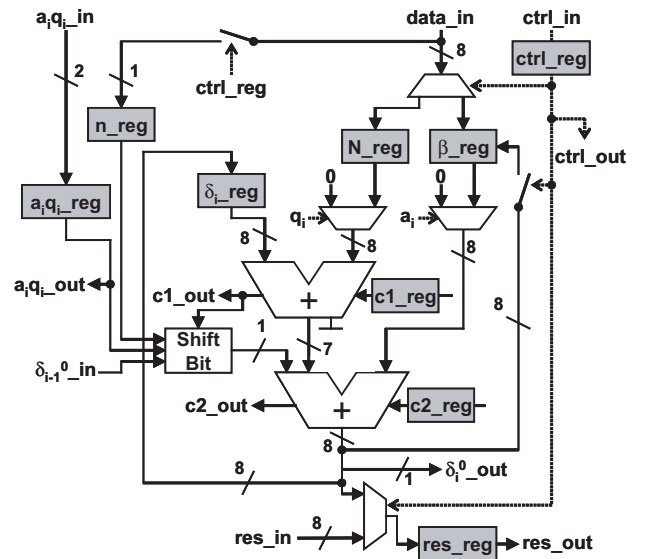


Fig. 2. An 8-bit processing unit.

As the processing is performed in the systolic architecture, the PUs compute their parts of the additions on concurrent clock cycles. Thus, δ_{i-1}^0 from the higher PU, n_reg , and $c1_out$ are required for precomputing the highest bit of the shifted operand of the second adder (*Shift Bit*).

The MM systolic PU array is presented in Fig. 3. The number of required PUs is $n/8$, in which n is the bit length of N . For example, 128 PUs are required for a 1024-bit modulus. The highest PU is slightly different as it has to handle the highest bit of β and the carries internally. The input bits (a_i, q_i) and the control signals flow into PU_0 and propagate through the array, a PU at a clock cycle. Each PU is fed with the 8-bits of β synchronously with the control flow. Successive PUs compute their pieces of the sum at consecutive clock cycles. Upon finishing, the 8-bit MM results are shifted backwards from the higher PUs to the lower ones, through the array of the res_reg registers in the PUs. The highest bit of the result δ_{n+3} (h_bit) is output separately and stored in the control entity for further use. The next MM can begin already before the previous one is completely finished, i.e., when PU_0 is ready.

C. Modular Exponentiation Accelerator

The modular exponentiation accelerator design on the SoPC is shown in Fig. 4. The hardware consists of the MM array, storage elements, and control logic. The overall processing is controlled by the ARM9 software. *I/O Ctrl* transfers the exponentiation inputs and outputs between the PLD and ARM9 through the DPRAM. *I/O Ctrl* contains an AHB bridge slave, which is utilized for the software control. When the hardware has finished an exponentiation, ARM9 is interrupted (IRQ). The figure also includes a pseudo-code example of using the Application Programming Interface (API) of the accelerator provided in ARM9.

In order to initiate the accelerator, ARM9 computes $R2 = r^2 \bmod N$. It loads the modulus N into the PUs through $R2_RAM$ and stores $R2$ in $R2_RAM$. The computation of $R2$ has to be performed only once for a given modulus. For example, in SRP it is likely that the modulus is changed infrequently [4]. In [7] it is also assumed that the modulus is known before the authentication is initiated, and thus, $R2$ can be precomputed and stored with N .

To begin an exponentiation, ARM9 loads the base X and the exponent E to Z_RAM and E_RAM , respectively, and releases the control to the hardware. *Loader* transforms the

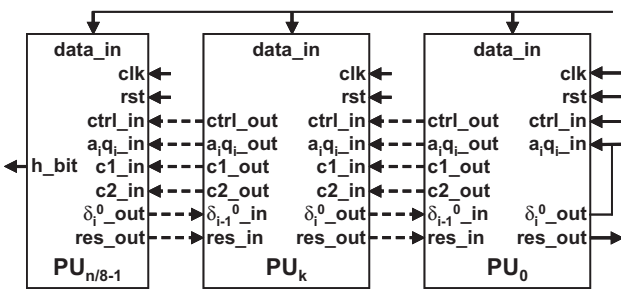


Fig. 3. Systolic processing array.

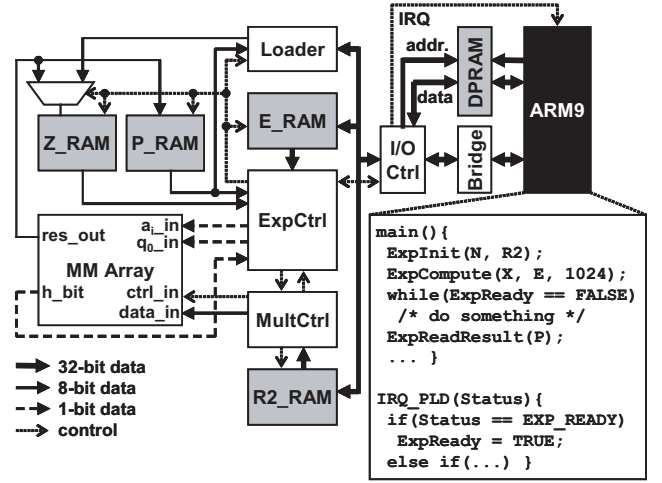


Fig. 4. Modular exponentiation accelerator on the SoPC.

32-bit Z_RAM inputs from *I/O Ctrl* into 8-bit pieces. All the RAM entities are implemented in the ESBs of the PLD.

In the exponentiation *ExpCtrl* reads in 8-bits of the multiplier (either from P_RAM or Z_RAM) and 32-bits of the exponent at a time and feeds the systolic array with a_i . It stores the multiplication results into their corresponding RAMs and instructs *Loader* to output the final exponentiation result. *MultCtrl* initializes the systolic array and feeds in the control signals during a single multiplication.

In the beginning of the exponentiation, MM is performed using X as the multiplier and $R2$ as the multiplicand for the pre-transformation. The multiplicand is read from $R2_RAM$ in 8-bit pieces and input to the PUs through $data_in$ as β . The multiplier is read from Z_RAM also in 8-bit pieces. Initially, q_0 is the lowest bit of the multiplier and the next values are the internal δ_i^0 feedbacks in PU_0 . The MM result is written to Z_RAM and P_RAM in 8-bit pieces and also kept as the next multiplicand (Z_0) in the PUs.

The hardware performs the modular exponentiation according to Algorithm 2 as follows. Computation is finished when all the exponent bits have been processed. The exponent length is defined as the third parameter of *ExpCompute* API function.

- 1) The contents of Z_RAM is used as the multiplier (Z_j) and the same Z_j in PUs as the multiplicand. The MM result is written back to Z_RAM as Z_{j+1} and also kept in PUs. This step is repeated until $e_{j+1} = 1$. When $e_{j+1} = 1$ the result is also stored in P_RAM , j is incremented, and the execution jumps to Step 3.
- 2) (*Multiply*) If $e_j = 0$ and $j < l - 1$, j is incremented and the execution jumps to Step 3. If $e_j = 0$ and $j = l - 1$, the PUs contain P_l and the execution jumps to Step 4. Otherwise, the contents of P_RAM is used as the multiplier (P_j) and Z_j in PUs as the multiplicand. If $j < l - 1$, the MM result is written to P_RAM as P_{j+1} , Z_j is kept in PUs, and the execution moves to Step 3. If $j = l - 1$, the computed P_l is kept in PUs and the execution moves to Step 4.

TABLE I

SYNTHESIS RESULTS FOR THE 1024-BIT ACCELERATOR.

Entity	LEs	Memory bits
Exponentiation	12,853	4,096
ARM-PLD Communications	461	0
I/O Ctrl	394	0
Loader	180	0
Others	72	0
<i>Total</i>	13,960	4,096
Maximum clock		63 MHz
Ref. [4]	9,644	5,120
Maximum clock		65 MHz
Ref. [2]	8,448	n/a
Maximum clock		42 MHz

- 3) (*Square*) If $j = l - 1$, the execution jumps to Step 2. Otherwise, the contents of Z_RAM is used as the multiplier (Z_j) and the same Z_j in PUs as the multiplicand. The MM result is written to Z_RAM as Z_{j+1} and kept in PUs. The execution moves to Step 2 and j is updated.
- 4) P_l in the PUs is multiplied with 1 to obtain the exponentiation result, which is written to P_RAM .

The final result is output via *Loader* and *I/O Ctrl* to the DPRAM and ARM9 is interrupted. The processor reads the exponentiation result from the DPRAM.

IV. RESULTS & COMPARISON

The synthesis results for the exponentiation accelerator with 1024-bit inputs on EPXA10F1020C2 produced by the Quartus v4.1 tool are presented in Table I. The table also presents results from our previous implementation [4] and the radix-2 implementation on which it was based [2]. In order to estimate the equivalent LE count of [2], the reported number of Xilinx Configurable Logic Blocks (CLB) has been doubled.

As shown, the LE count in this work has increased. In [4] and [2] the PUs were carefully tuned and placed to fit in a compact area whereas in this work the placement of the modified PUs was left to the synthesis tool. Also, certain buses are wider and the new design supports full length inputs compared to [4]. Memory bits have been saved from [4] since the modulus N is stored in the PUs instead of ESBs. The clock frequency has slightly decreased because the combinatorial path in PUs has been made longer to reduce cycle counts.

Despite the frequency decrease, the reduced cycle count increases the overall performance of the accelerator compared to the reference designs. The references interleave the squaring and multiplication of Algorithm 2. For interleaving, each PU contains an additional 8-bit register and operate in a 2-cycle mode, in which one cycle is used for squaring and the other for multiplication. The design of this paper operates in a 1-cycle mode, in which the PUs compute the multiplications and squarings in series. Whereas in the 2-cycle mode the line 5 of Algorithm 2 is computed every time (if $e_i = 0$, the result is discarded), in the 1-cycle mode it is only computed when $e_i = 1$. A MM operation is saved whenever $e_i = 0$. Statistically, the 1-cycle mode is expected to require $1.5 \times l$

TABLE II

COMPARISON OF CYCLE COUNTS OF A FULL EXPONENTIATION.

Design	Cycles
This work	$(n+k)(n+4)$; $0 < k \leq n$
Ref. [4]	$(2n+4)(n+4)$
Ref. [2]	$(2n+4)(n+4)$

The parameter k is the number of ones in the binary representation of the exponent E . The parameter n is the bit length of N and E .

MM operations whereas the 2-cycle mode always requires $2 \times l$ MM operations. With equal clock speeds the design of this paper is always faster than the reference designs.

Compared to the reference implementations, the exponentiation control has also been tuned. Step 1 of Section III-C makes the pre-transformation of $P_0 = 1$ and the first multiplication with P_j unnecessary, saving two more MM operations. In addition, the control, e.g., skips squaring when $j = l - 1$.

Table II presents the comparison of the cycle counts of the designs for a full exponentiation ($l = n$). The table assumes that N and $R2$ are already computed and stored in the PLD. In the average case of $n = 1024$ and $k = 512$, the accelerator computes the exponentiation 25% faster than the previous designs at equal clock speeds. Compared to the software measurements of [5], the cycle counts are decreased by the factor of 11 from Pentium III and 19 from ARM9. The 1-cycle PUs with the tuned control can also be utilized for improving, e.g., the higher radix design [3] in the same way.

V. CONCLUSIONS

This paper presented a modular exponentiation accelerator utilizable in a number of security protocols. The accelerator processing was implemented in hardware and the control in software on a SoPC. The performance was increased by the factor of several tenfolds from software implementations. Compared to the previous radix-2 designs, significant performance increase through 1-cycle PUs and improved control was achieved. Also, the required amount of resources can be decreased to the same level with manual tuning.

REFERENCES

- [1] "Altera website," <http://www.altera.com>, 2005.
- [2] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proc. 14th IEEE Symp. Computer Arithmetic (ARITH 14)*, Adelaide, Australia, Apr. 14–16, 1999, pp. 70–77.
- [3] —, "High radix Montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. Computers*, vol. 50, no. 7, pp. 759–764, 2001.
- [4] P. Groen, P. Hämäläinen, B. Juurlink, and T. D. Hämäläinen, "Accelerating the Secure Remote Password protocol using reconfigurable hardware," in *Proc. 2004 ACM Computer Frontiers Conf. (CF'04)*, Ischia, Italy, Apr. 14–16, 2004, pp. 471–480.
- [5] P. Hämäläinen, M. Hännikäinen, M. Niemi, and T. D. Hämäläinen, "Performance evaluation of Secure Remote Password protocol," in *Proc. 2002 IEEE Int. Conf. Circuits and Systems (ISCAS 2002)*, vol. 3, Scottsdale, AZ, USA, May 26–29, 2002, pp. 29–32.
- [6] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
- [7] T. Wu, "The SRP authentication and key exchange system," RFC 2945, Sept. 2000.
- [8] L. Zhao, S. Mäkinen, and L. Bhuyan, "Anatomy and performance of SSL processing," in *Proc. 2005 IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS 2005)*, Austin, TX, USA, Mar. 20–22, 2005, pp. 197–206.