Chapter 17

Architecture and FPGA Implementation of a Digit-serial RSA Processor

Alessandro Cilardo[1], Antonino Mazzeo[2], Luigi Romano[3], Giacinto Paolo Saggese[4]

[1] *Dipartimento di Informatica e Sistemistica, Universita' degli Studi di Napoli*
acilardo@unina.it

[2] *Dipartimento di Informatica e Sistemistica, Universita' degli Studi di Napoli*
mazzeo@unina.it

[3] *Dipartimento di Informatica e Sistemistica, Universita' degli Studi di Napoli*
lrom@unina.it

[4] *Dipartimento di Informatica e Sistemistica, Universita' degli Studi di Napoli*
saggese@unina.it

Keywords: Field-Programmable Gate Array (FPGA), RSA cryptosystem, Modular Multiplication, Modular Exponentiation

Introduction

In the recent years, we have witnessed an increasing deployment of hardware devices for providing security functions via cryptographic algorithms. In fact, hardware devices provide both high performance and considerable resistance to tampering attacks, and are thus ideally suited for implementing computationally intensive cryptographic routines which operates on sensitive data.

Among the various techniques found in the cryptography realm, the Rivest-Shamir-Adleman (RSA) algorithm [1] constitutes the most widely adopted public-key scheme. In particular, it is useful for security applications which need confidentiality, authentication, integrity, and non-repudiation [2]. The basic operation of this algorithm is modular exponentiation on large integers,

P. Lysaght and W. Rosenstiel (eds.),
New Algorithms, Architectures and Applications for Reconfigurable Computing, 209–218.

i.e. $Y = X^E \bmod N$, which is used for both decryption/signature and encryption/verification.

The security level of an RSA cryptosystem is tied to the length of the modulus N. Typical values of the modulus size range from 768 to 4096 bits, depending on the security level required. All operands involved in the computation of modular exponentiation have normally the same size as the modulus.

All existing techniques for computing $X^E \bmod N$ reduce modular exponentiation to a sequence of modular multiplications. Several sub-optimal algorithms have been presented in the literature to compute the sequence of multiplications leading to the Eth power of X, such as binary methods (RL-algorithm and LR-algorithm), M-ary methods, Power Tree, and more [3, 4]. In particular, the method known as Binary Right-to-Left Algorithm [4] consists of repeated squaring and multiplication operations and is well-suited for simple and efficient hardware implementations.

Since modular multiplication is the core computation of all modular exponentiation algorithms, the efficiency of its execution is crucial for any implementation of the RSA algorithm. Unfortunately, modular multiplication is a complex arithmetic operation because of the inherent multiplication and modular reduction operations. Several techniques have been proposed in the last years for achieving efficient implementations of modular multiplication. In particular, Blakley's method [5] and Montgomery's method [6] are the most studied techniques. Actually, they are the only algorithms suitable for practical hardware implementation [3].

Both Blakley's method and Montgomery's method perform the modular reduction during the multiplication process. No division operation is needed at any point in the process. However, Blakley's method needs a comparison of two large integers at each step of the modular multiplication process, while the Montgomery's method does not, by means of a representation of the operands as a residue class modulo N. Furthermore, the Montgomery's technique requires some preprocessing and postprocessing steps, which are needed to convert the numbers to and from the residue based representation. However, the cost of these steps is negligible when many consecutive modular multiplications are to be executed, as in the case of RSA. This is the reason why the Montgomery's method is considered the most efficient algorithm for implementing RSA operations. There exist several versions of the Montgomery's algorithm, depending on the number r used as the radix for the representation of numbers. In hardware implementations r is always a power of 2.

The Montgomery's algorithm is in turn based on repeated additions on integer numbers. Since the size of operands is as large as the modulus, the addition operation turns out to be the critical step from the implementation viewpoint.

In this paper we present an innovative hardware architecture and an FPGA-based implementation of the Montgomery's algorithm based on a digit-serial approach, which allows the basic arithmetic operations to be broken into words and processed in a serialized fashion. As a consequence, the architecture implementation takes advantage of a short critical path and low area requirements. In fact, as compared to other solutions in the literature, the proposed implementation of the RSA processor has smaller area requirements and comparable performance. The serialization factor S of our serial architecture is taken as a parameter and the final performance level is given as a function of this factor. We thoroughly explore the design tradeoffs, in terms of area requirements vs time performance, for different values of the key length and the serialization factor.

17.1 Algorithm Used for the RSA Processor

This section describes the algorithms we have used in our RSA processor. For implementation of modular multiplication we exploit some optimizations of the Montgomery Product first described by Walter [7]. We assume that N can be represented with K bits, and we take $R = 2^{K+2}$. The N-residue of A with respect to R is defined as the positive integer $\bar{A} = A \cdot R \bmod N$. The Montgomery Product [6] of residues of A and B, $MonPro(\bar{A}, \bar{B})$, is defined as $(\bar{A} \cdot \bar{B} \cdot R^{-1}) \bmod N$, that is the N-residue of the desired $A \cdot B \bmod N$. If $A, B < 2N$, combining [7] and [3], the following radix-2 binary add-shift algorithm can be employed to calculate *MonPro*:

Algorithm 17.1.1— *Montgomery Product MonPro(A,B) radix-2.*
Given $A = \sum_{i=0}^{K+2} A_i \cdot 2^i, B = \sum_{i=0}^{K} B_i \cdot 2^i, N = \sum_{i=0}^{K-1} N_i \cdot 2^i$, where $A_i, B_i, N_i \in \{0, 1\}, A_{K+1}, A_{K+2} = 0$, computes a number falling in $[0, 2N]$ which is modulo N congruent with desired $(A \cdot B \cdot 2^{-(K+2)}) \bmod N$
1. $U = 0$
2. `For` $j = 0$ `to` $K + 2$ `do`
3. `if` $(U_0 = 1)$ `then` $U = U + N$
4. $U = (U/2) + A_j \cdot B$
5. `end for`

We report the exponentiation algorithm for computing $X^E \bmod N$ known as Right-To-Left binary method [4], modified here in order to take advantage of the Montgomery Product.

Algorithm 17.1.2— *Right-To-Left Modular Exponentiation using Montgomery Product.*
Given X, N, and $E = \sum_{i=0}^{H-1} E_i \cdot 2^i$, $E_i \in \{0, 1\}$, computes $P = X^E \bmod N$.

1. $P_0 = MonPro(1, R^2 \bmod N)$
2. $Z_0 = MonPro(X, R^2 \bmod N)$
3. `For` $i = 0$ `to` $H - 1$ `do`
4. $\quad Z_{i+1} = MonPro(Z_i, Z_i)$
5. $\quad$ `if` $(E_i = 1)$ `then` $P_{i+1} = MonPro(P_i, Z_i)$
6. $\quad\quad$ `else` $P_{i+1} = P_i$
7. `end for`
8. $P = MonPro(P_H, 1)$
9. `if` $(P \geq N)$ `then return` $P - N$
10. $\quad\quad$ `else return` P

The first phase (lines 1–2) calculates the residues of the initial values 1 and X. For a given key value, the factor R^2 mod N remains unchanged. It is thus possible to use a precomputed value for such a factor and reduce residue calculation to a *MonPro*. The core of the computation is a loop in which modular squares are performed, and the previous partial result P_i is multiplied by Z_i, based on a test performed on the value of i-th bit of E (H is the number of the bits composing E). It is worth noting that, because of the absence of dependencies between instructions 4 and 5–6, these can be executed concurrently. Instruction 8 allows to switch back from the residue domain to the normal representation of numbers. After line 8 is executed, a further check is needed (lines 9–10) to ensure that the obtained value of P is actually X^E mod N. In fact, while it is acceptable in intermediate loop executions that the *MonPro* temporary result (line 4 and line 5) be in the range $[0, 2N]$, this cannot be in the last iteration. Thus, if the final value of P is greater than N, it must be diminished by N.

17.2 Architecture of the RSA Processor

As shown in Algorithm 17.1.2, the modular multiplication constitutes the basic operation of the RSA exponentiation, which basically consists of repeated multiplication operations. Thus, the RSA processor must be able to properly sequence modular products on data and store intermediate results in a register file according to the control flow of Algorithm 17.1.2. In turn, the *MonPro* operation (see Algorithm 17.1.1) is composed of different micro-operations consisting of load/store on registers, shifts and additions. All operands are $(K + 3)$-bit long at most, where K is the modulus length. In our implementation, each arithmetic operation is broken up into a number of steps and executed in a serialized fashion on S-bit words.

At a high level of abstraction, the RSA processor is composed of two modules (see Figure 17.1): a *Data-path Unit* performing data-processing operations, and a *Control Unit* which determines the sequence of operations.

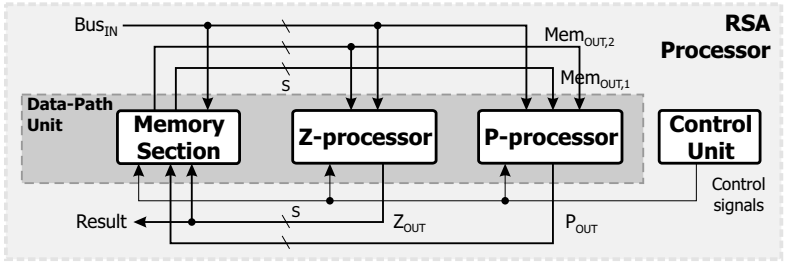

Figure 17.1. Overall architecture of RSA processor.

The *Control Unit* reflects the structure of Algorithm 17.1.2, in that it is composed of two hierarchical blocks corresponding to the main modular exponentiation routine of Algorithm 17.1.2 and the *MonPro* subroutine of Algorithm 17.1.1, respectively.

More precisely, a block named *MonExp_Controller* is in charge of generating the control flow signals (for loops, conditional and unconditional jumps), activating a second block (the *MonPro_Controller* block) when a modular product is met, and waiting until this has finished. The *MonPro_Controller* supervises modular product execution, i.e. it sequences long integer operations, which are performed serially on *S*-bit words.

The *Data-path* has a data width of *S* bits. Thus, *S* represents the serialization factor or, in other terms, *S* is the digit-size in multiprecision arithmetic. Data-path is composed of three macro blocks (see Figure 17.2): a *Memory Section* block storing intermediate data inherent in Algorithm 17.1.2, and two processing units named *P-processor* and *Z-processor*, executing concurrently modular products and modular squaring operations, respectively.

The data-path is organized as a 3-stage pipeline. The first stage fetches the operands. Data are read from the register file in the Memory Section and from

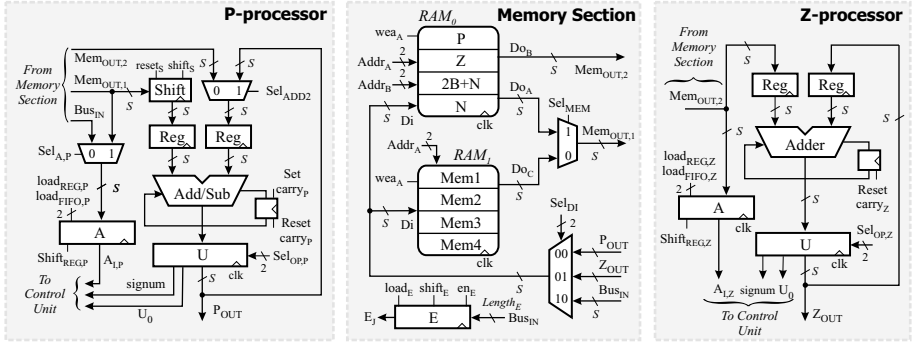

Figure 17.2. Structure of the Data-path.

scratchpad registers, containing the current partial result U for squaring and multiplication. The second stage (Adder/Sub block) operates on the fetched data. The last stage writes results back into Registers U. More details about the presented architecture are provided in [9].

P-processor. The P-processor implements several operations in distinct phases of the RSA algorithm: 1) along with the Memory Section, it acts as a serial Montgomery multiplier implementing Algorithm 17.1.1; 2) it carries out a simple preprocessing phase to accelerate modular multiplication (i.e. $2B$ and $2B + N$ computation); 3) finally, it performs the reduction step (a comparison and a conditional subtraction) to ensure that the result is actually the modulus of the requested exponentiation.

MonPro(A, B) of Algorithm 17.1.1 is a sequence of $K+3$ conditional sums, in which the operands depend on both the least significant bit U_0 of the partial result U, and the ith bit of the operand A. The $(K + 2)$-bit additions are performed serially with an S-bit adder in $M = \lceil (K + 2)/S \rceil$ clock cycles. This has two fundamental advantages. First, it allows area saving, provided that the area-overhead due to serial to parallel data formatting and the subsequent inverse conversion does not frustrate area-saving deriving from smaller operands and a narrower data-path. Second, a serial approach avoids long carry chains, which are difficult to implement in technologies such as FPGA, and result in larger net delays. These advantages come at the price of the time-area overhead due to serial-to-parallel (and vice versa) conversions. By addressing the RAM as a set of register files and accessing directly the required portion of operands, we can get rid of multiplexer/demultiplexer blocks. This results in a dramatic reduction of time-area overhead.

The computational core of the P-processor corresponds to Steps 3–4 of Algorithm 17.1.1 and is actually implemented as:

$$3.\quad U = (U + A_i \cdot 2B + U_0 \cdot N)/2 = (U + V)/2$$

where $V \in \{0, 2B, 2B + N, N\}$ depending on A_i and U_0. It can be proved that $U + V$ before division by two is always even, and so a simple shift can yield the correct result. In the proposed implementation, the modular product is composed by a preprocessing phase for computing (once for all) the value $2B + N$ [8]. This saves time, because $2B + N$ is added to U $(K + 3)/4$ times in average (assuming A_i and U_0 independent and equally distributed). Hence, at the price of one register of $K+3$ bit, we can save $M \cdot ((K + 3)/4 - 1)$ clock cycles, and also save hardware, since a two word adder can be used instead of a three word adder.

The circuit implementing the *Shift* block (see the P-processor structure in Figure 17.2), computes a multiplication by two when required. The S-bit

registers *Reg* are pipelining registers. The *Adder/Subtracter* sums operands, or subtracts the operand on the right input from the one on the left input. Note that the Adder/Subtracter performs multiprecision operations. *Register U* stores the value of U prior to shifting, as required by the modular product algorithm. It shows its contents shifted down by 1 bit. It also outputs the least significant bit of U, necessary for the *Controller* to choose which is the next operand that is to be added to U. *Register A* holds the $(K + 3)$-bit value of the A operand of modular product, which can be loaded serially from the Memory Section. It can shift down one bit at a time, showing the A_i bit to the controller.

Memory Section. The Memory Section schematics is reported in Figure 17.2. Memory Section supplies P and Z processors with the contents of the register file, in a serial fashion, S bits at a time. It receives output data from processors or from external through Bus_{IN}. Memory RAM_0 stores $(K + 3)$-bit numbers that can be added to the partial sum U of Algorithm 17.1.1 (P, Z, $2B + N$, N), while RAM_1 stores K-bit constants that are useful for calculating an N-residue, or returning to normal number representation. Each operand is stored as an array of S-bit words, in order to avoid the use of area-consuming multiplexers for selecting the correct part of the operand to be summed. The Memory Section also contains *Register E*, which stores the exponent E.

Z-Processor. The Z-processor (see Figure 17.2) is a simplified version of the P-Processor, intended to perform a *MonPro* operation concurrently with the P-Processor. Note that, strictly speaking, we only need a P-processor and a Memory-Section to implement modular product and squaring. Hence, the Z-processor could be discarded (at the cost of doubling the time for modular exponentiation). This simplification however scales down the overall required area by a factor smaller than 2 (area of Memory Section is constant), so the version of the RSA processor with both the P and the Z processors is characterized by a better value of the product $A \cdot T$. When the available area is reduced and performance is not critical, the design option which relies solely on the P-processor can gain interest.

17.3 FPGA Implementation and Performance Analysis

We have used a Xilinx Virtex-E 2000-8bg560 for implementing the proposed architecture. Xilinx XCV2000E has 19200 slices and 19520 tristate buffers. For synthesis we used Synplicity Synplify Pro 7.1, integrated in Xilinx ISE 4.1.

We implemented our design in RTL VHDL for $K = 1024$ and evaluated it for values of the widths S ranging from 32 to 256. First we verified the correctness of the design. We then performed synthesis, place&route step, and

S	Stage	T_{ck} [ns]	Total Slices	FF/LUT slices	Lut for Dual-port RAM/Single-port RAM	Lut for Shift Reg	Tristate buffers	Mux implementation
32	Memory	8,7	627	44 / 486	512 / 256	0	0	LUTs
	Adder	7,4	32	2 / 64	0 / 0	0	0	
	Write	8,6	336	385 / 143	256 / 0	128	448	Tristates
64	Memory	10,7	563	44 / 108	512 / 256	0	960	Tristates
	Adder	11,6	64	2 / 128	0 / 0	0	0	
	Write	11,5	561	577 / 209	512 / 0	128	1152	Tristates
128	Memory	17,5	688	44 / 34	512 / 256	0	1344	Tristates
	Adder	18,6	132	2 / 264	0 / 0	0	0	
	Write	17,8	1050	1025 / 278	1024 / 0	256	2816	Tristates
256	Memory	30,5	553	44 / 30	512 / 256	0	2122	Tristates
	Adder	30,9	264	2 /528	0 / 0	0	0	
	Write	30,2	2085	2049 / 543	2048 / 0	512	4608	Tristates

Figure 17.3. Hardware resources and clock periods for each pipeline stage varying S.

timing verification. A partial manually floorplan of individual blocks was carried out occasionally, upon need.

Figure 17.3 reports the minimum clock period and the total number of slices required for different values of S and for each stage of the pipeline. Results show that the stage which limits the clock frequency is the Adder. Hence, for a fixed S, we determined the maximum sustainable clock rate from the Adder, and used that as the target clock for other stages. We exploited the capability of Virtex devices to implement multiplexers using tristate buffers instead of LUTs. From a detailed analysis of the architecture, it follows that the number of clock cycles for each modular product is given by $(2M + 2) + (K + 3) \cdot M$. The number of sequential modular products is $H + 2$ (where H is the number of bits composing the exponent E) because squarings and products of the modular exponentiation loop are executed in parallel, and a product is necessary for residue calculation and for turning the residue of the result in normal representation. Finally, M clock cycles are needed for subtraction, and $M + 1$ clock cycles are needed for the last S bits of the result to appear on the output P_{out}. The number of clock cycles (N_{CK}), the total area, and the total time for the exponent $E = 2^{16} + 1$ are reported in Figure 17.4 as functions of parameter S. These results are contrasted against other implementations in the following section.

S	T_{ck} [ns]	Area [Slices]	N_{CK}	Total Time [ms]	A·T [Slices·ms]
32	8,74	995	645288	5,64	5612
64	11,6	1188	176016	3,86	4586
128	18,6	1870	332440	3,27	6115
256	30,9	2902	97804	2,99	8677

Figure 17.4. RSA encryption with 1024 bit key.

17.4 Related Work

Most hardware implementations of modular exponentiation are either dated or they rely on an ASIC implementation. As a consequence, a fair comparison is difficult. In [7] the Montgomery technique and the systolic multiplication were combined for the first time, resulting in a bidimensional systolic array architecture which gave a throughput of one modular multiplication per clock cycle and a latency of $2K + 2$ cycles. The great area requirements constituted the major drawback of this structure, deriving from its inherent high-parallelism.

A unidimensional systolic architecture was proposed in [8]. This work implements the same algorithm as ours (radix-2 Montgomery algorithm for modular product Algorithm 17.1.1), on top of a Xilinx device with the same basic architecture, but with a different technology, namely a XC40250XV-09 device. In a successive study [10], Blum and Paar improved their architecture using a high-radix formulation of Algorithm 17.1.1. To output the entire result, [8] requires $2(H + 2)(K + 4) + K/U$ clock cycles, where U is the dimension of the processing elements. The fastest design ($U = 4$) of [8] requires 0.75 ms for the same encryption of Figure 17.4 and requires 4865 XC4000 CLBs that are equivalent to 4865 Virtex slices. Our fastest design ($S = 256$) requires 2.99 ms (4 times slower), but it requires 2902 slices (with a saving of area equal to 40%). Our design requiring the least area ($S = 32$) occupies only 995 slices, while the smallest one in [8] requires 3786 slices. Finally, our design with the best $A \cdot T$ product ($S = 64$) presents $A \cdot T = 4586$, while the corresponding design of [8], presents $A \cdot T = 3511$.

In summary, the solution presented in [8] exhibits better performance, as compared to ours. This was made possible by the improved parallelism due to pipelining, inherent in the systolic paradigm of computation. On the other hand, our implementation is slower, but it has lower area requirements. This would allow the RSA processor to be replicated on the same physical device and more modular exponentiation operations to be performed in parallel.

17.5 Conclusions

We presented a novel serial architecture for RSA encryption/decryption operation. The design is targeted for implementation on reconfigurable logic, and exploits the inherent characteristics of the typical FPGA devices. The design allows to tradeoff area for performance, by modifying the value of the serialization factor S. Quantitative evaluation of this tradeoff was conducted via simulation. The results obtained showed that the presented architecture achieves good performance with low area requirements, as compared to other architectures.

Acknowledgements

This work was partially funded by Regione Campania within the framework of the Telemedicina and Centri di Competenza Projects.

References

[1] R. L. Rivest et al., "A Method for Obtaining Digital Signatures", Commun. ACM, vol. 21, pp. 120–126, 1978.
[2] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.
[3] Ç. K. Koç, "High-speed RSA Implementation", Technical Report TR 201, RSA Laboratories, November 1994.
[4] D. E. Knuth, "The Art of Computer Programming: Seminumerical Algorithms", vol. 2, Addison-Wesley, 1981.
[5] G. R. Blakley, "A computer algorithm for the product . . . ", IEEE Trans. on Computers, Vol. 32, No. 5, pp. 497–500, May 1983.
[6] P. L. Montgomery, "Modular multiplication without trial division", Math. of Computation, 44(170):519–521, April 1985.
[7] C. D. Walter, "Systolic Modular Multiplication", IEEE Trans. on Computers, Vol. 42, No. 3, pp. 376–378, March 1993.
[8] T. Blum, and C. Paar, "Montgomery Modular Exponentiation . . . ", Proc. 14th Symp. Comp. Arith., pp. 70–77, 1999.
[9] A. Mazzeo, N. Mazzocca, L. Romano, and G. P. Saggese, "FPGA-based Implementation of a Serial RSA processor", Proceedings of the Design And Test Europe (DATE) Conference 2003, pp. 582–587.
[10] T. Blum, and C. Paar, "High-Radix Montgomery Modular . . . ", IEEE Trans. on Comp., Vol. 50, No. 7, pp. 759–764, July 2001.