FPGA-based Implementation of a serial RSA processor

A. Mazzeo, L. Romano, G. P. Saggese - Universita' degli Studi di Napoli "Federico II"
N. Mazzocca - Seconda Universita' degli Studi di Napoli
{mazzeo, lrom, saggese, n.mazzocca}@unina.it

Abstract

In this paper we present an hardware implementation of the RSA algorithm for public-key cryptography. The RSA algorithm consists in the computation of modular exponentials on large integers, that can be reduced to repeated modular multiplications. We present a serial implementation of RSA, which is based upon an optimized version of the RSA algorithm originally proposed by P.L. Montgomery. The proposed architecture is innovative, and it widely exploits specific capabilities of Xilinx programmable devices. As compared to other solutions in the literature, the proposed implementation of the RSA processor has smaller area occupation and comparable performance. The final performance level is a function of the serialization factor. We provide a thorough discussion of design tradeoffs, in terms of area requirements vs performance, for different values of the key length and of the serialization factor.

1. Introduction and RSA algorithm

In the recent years, we have witnessed increasing deployment of hardware devices for the provisioning of security functions, such as confidentiality, authentication, integrity and non-repudiation [1]. Hardware devices appear to be a promising solution to inherent performance issues of symmetric (private key) and asymmetric (public-key) cryptosystems, and provide greater resistance to tampering [1].

Among the existing algorithms the Rivest-Shamir-Adleman (RSA) is the most widely adopted [2] public-key cryptography algorithm. Its security lies in the difficulty of factorizing large integers. The basic operation of such an algorithm is modular exponentiation on large integers. The private key consists of two large primes p and q and an exponent D. The public key consists of the modulus $N = p \cdot q$, and an exponent E such that $E = D^{-1} \bmod (p-1) \cdot (q-1)$. To encrypt a message X, the algorithm requires the computation of $Y = X^E \bmod N$. Decryption is done by calculating $X = Y^D \bmod N$. Both encryption and decryption require the computation of modular exponentiation. In this paper, we thus concentrate on the calculation of modular exponentiation $Y = X^E \bmod N$.

Two are the major problems that make an effective implementation of modular exponentiation difficult:

1) modular exponentiation is typically carried out via repeated modular multiplication. The problem of determining the minimum sequence of multiplications is itself a hard task. In fact, for a given positive exponent *E*, the problem of computing the minimum sequence of multiplications for *E* is known as Addition Chains, and it is established to be an NP-complete problem [3]. Heuristic search can be used only when the modulus is known in advance and a preprocessing step is thus possible. However, many algorithms are known, which can guarantee shorter sub-optimal addition chains, such as: binary methods (RL-algorithm, LR-algorithm), M-ary methods, Power Tree, etc. [4]);

2) the design and the implementation of an effective algorithm for modular multiplication is also an issue. In fact, modular multiplication is generally considered a complex arithmetic operation because of the inherent multiplication and division operations. This is particularly true if the size of the operands is large, i.e. greater than 64 bits or so. It is worth noting that the involved operands are indeed large, since in order to thwart currently known attacks, the modulus N, X and Y have lengths of K of 512-1024 bits at least.

As far as the first issue is concerned, i.e. the computation of an addition chain for a given exponent, we adopted the method known as Binary Right-to-Left Algorithm [5] which consists in repeated squarings and multiplications. This choice was motivated by the simple hardware implementation of the method. This is actually the only feasible alternative, especially when limited hardware resources are available.

The main focus of the paper is thus the second issue, i.e. the problem of modular multiplication. Two are the main approaches to calculating modular multiplication in the literature: *division-after-multiplication*, and *division-during-multiplication*. In the former, modulo operation follows multiplication: the K-bit multiplication is carried out, and then the $2K$-bit result is divided, thus leading to the desired remainder. In the latter, the modulo operation is repeated after each iteration of the multiplication procedure. An overview of different algorithms employed in the division-after-multiplication method is in [4]. This approach requires more hardware resources than the division-during-multiplication counterpart, mainly because of multiplication and computation of remainder. The only reason why it was originally adopted is its intuitiveness. The division-during-multiplication method is widely recognized as a more effective method and it is thus the preferred one.

1530-1591/03 $17.00 © 2003 IEEE

Two are the main division-during-multiplication techniques: *Blakley's method* and *Montgomery's method*. Blakley's method [6] computes a modular multiplication by interleaving the shift-add steps of the multiplication and the shift-subtract steps of the division. Montgomery's method [7] instead, computes $M = A \cdot B \bmod N$, without performing any division by the modulus N.

The advantage of this approach is that it exploits a representation of A and B as a residue class modulo N, thus replacing the division by N operation with a division by R. Since R can be chosen as a power of 2, this operation is a low-cost operation for binary represented numbers. However, the preprocessing and postprocessing steps, which are needed to convert the numbers to and from residue based representation are expensive. The cost of the conversion may be unacceptable if a few modular multiplications are to be performed, but it becomes negligible as the number of modular multiplications with the same modulus increases. Since this is the case of modular exponentiation, we resorted to the Montgomery algorithm.

The rest of the paper is organized as follows. In Section 2 we explain the algorithms which we have adopted in the proposed implementation of RSA. Section 3 describes the architecture of the RSA processor, parameterized with respect to the length of the key, K, and the size S of the operands of the elementary steps. Sections 4, addresses issues related to the implementation of the proposed RSA processor on a COTS (Commercial Off The Shelf) reconfigurable device, namely Xilinx VirtexE. Section 5, discusses performance results and analyzes area vs throughput trade-offs, with respect to other architectures proposed in the literature. Finally, section 6 concludes the paper with some final remarks.

2 Algorithms used in 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 Montgomery Product first described by Walter [8]. We suppose that N can be represented with K bit, and we take $R = 2^{K+2}$. The N-residue of A with respect to R is defined as the positive integer $\bar{A} = A \cdot R \bmod N$. Montgomery Product [7] 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 [8] and [4], the following radix-2 binary add-shift algorithm can be employed to calculate $MonPro$:

Algorithm 2.1 - *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 exponentiation algorithm for computing $X^E \bmod N$ known as Right-To-Left binary method [5], modified in order to take advantage of Montgomery Product.

Algorithm 2.2 - *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. $P_0 = MonProd(1, R^2 \bmod N)$
2. $Z_0 = MonProd(X, R^2 \bmod N)$
3. For $i = 0$ to $H - 1$ do
4. $Z_{i+1} = MonProd(Z_i, Z_i)$
5.   if ($E_i = 1$) then $P_{i+1} = MonProd(P_i, Z_i)$
6.   else $P_{i+1} = P_i$
7. end for
8. $P = MonProd(P_H, 1)$
9. if ($P \geq N$) then return $P - N$
10.   else return P

The first phase (lines 1-2) calculates residues of initial values 1 and X. For a given key value, the factor $R^2 \bmod N$ remains unchanged. It is thus possible to use a precomputed value for such a factor and reduce residue calculation to a $MonProd$. The core of the computation is a loop in which modular squares are performed, and 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 in parallel. Instruction 8 allows to switch back from residue domain to 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 $X^E \bmod N$. In fact, while it is acceptable in intermediate loop executions that the $MonProd$ 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.

3 Description of RSA processor

This section describes the architecture of our RSA processor, which implements the algorithms described in the previous section. From Alg. 2.2 it follows that basic operation is modular product, so the RSA processor must be able to properly sequence repeated modular products on data, and store intermediate results in a register file, according to the control flow of Alg. 2.2. Again, each $MonProd$ (see Alg. 2.1) is composed of different micro-operations consisting of load-store on registers, shifts, and additions. Operands are long integers with length related to K. In our implementation, each instruction is broken up into several parts and executed in a serial fashion on S-bit long operands.

At a high level of abstraction, the RSA processor is composed of two modules: a *Data-path Unit* performing data-processing operations, and a *Control Unit* which determines the sequence of operations. To master the complexity of the design, we used a modular approach. We divided the *Control Unit* in two entities, in accordance to Alg. 2.2 (which is composed by a main routine, calling the $MonProd$ routine). The *ModExp_Controller* block implements the routine

corresponding to Alg. 2.2. It is thus in charge of generating control flow signals (for loops, conditional and unconditional jumps), of activating the *MonProd_Controller* when a modular product is met, and of waiting until this has finished. The MonProd_Controller supervises correct modular product execution, i.e. it sequences long integer operations, which are performed serially on S bits at a time.

The *Data-path* is designed to operate on S-bit wide words. As such, S represents the serialization factor, or in other terms, S is digit-size in multiprecision arithmetic. Data-path is composed by three macro blocks (Fig. 1): a *Memory Section* block storing intermediate data inherent in Alg. 2.2, and two processing units named *P-processor* and *Z-processor*, implementing in parallel respectively modular product and modular squaring.

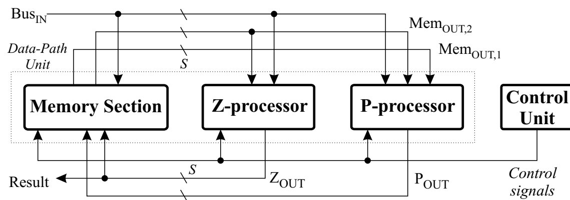

Figure 1. Overall architecture of RSA processor

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 scratchpad registers, containing current partial result U for squaring and product. The second stage (Adder/Sub block) operates on the data. The last stage writes results back into Registers U.

In the following, we will first describe the P-processor, which performs a serial modular product and a reduction step. Then, we will describe the Z-processor. Since squaring can be realized as a product, the Z-processor is a simplified version of a P-processor. We will thus limit our description to those characteristics of the Z-processor which differ from the P-processor. Finally, we will describe the Memory Section block and the microcoded controller has been designed.

3.1 P-processor

The P-processor realizes different operations in several phases of the RSA algorithm:
1) it can act, along with the Memory Section, as a serial Montgomery multiplier implementing Alg. 2.1;
2) it can carry out a simple preprocessing phase to accelerate modular product (i.e. $2B$+N computation);
3) finally, it can realize reduction step (a comparison and if necessary a subtraction) to ensure that result is actually the modulus of requested exponentiation.

$MonProd(A, B)$ of Alg. 2.1 is a sequence of K+3 conditional sums, in which the operands depend both on the least significant bit U_0 of partial result U, and on bits 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, since serial implementation of an algorithm makes it possible to use smaller operators (saving is actually achieved provided that area-overhead due to serial to parallel data formatting and subsequent inverse conversion does not frustrate area-saving deriving from smaller data-path). Second, a serial approach avoids long carry chains. Long carry chains cannot be accommodated in a single column of a Xilinx logic block array and must thus be broken down into several columns, which results in an increased net delay. These advantages

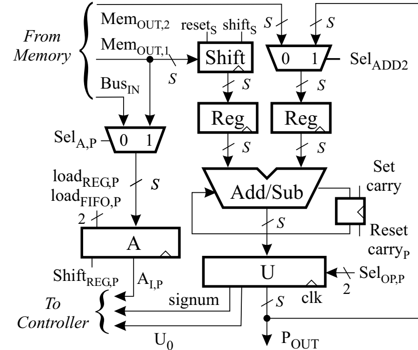

Figure 2. Structure of P-processor

come at the price of: i) a slightly more complex control unit, ii) time-area overhead due to serial-to-parallel and vice versa conversions, and iii) a potential increase of the overall delay due to the need of allowing the flip-flop delay to elapse S times (as opposed to only one time, in the completely parallel alternative).

The major drawback is certainly the second one. To address this, we use the RAM as a set of register files, storing (K+3)-bit operands as composed by M words, of size S. We then access directly the required portion of operands. By doing so, 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 is reported in Fig. 2. Steps 3-4 of Alg. 2.1 are actually implemented as:

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

where $V \in \{0, 2B, 2B{+}N, N\}$ depending on A_i and U_0. It can be proved that U 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 [9]. 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 a price of one register of K+3 bit, we can save M·((K+3)/4-1) clock ticks, and also save hardware, since a two word adder can be used instead of a three word adder.

The sequential circuit implementing the *Shift* block, computes a multiplication by two when required (i.e. when adding $2B$ to N in the preprocessing phase or adding $2B$ during multiplication). It uses a flip-flop to store the most significant bit of a word, and an S-wide multiplexer, indicating whether to execute the shift or not. The multiplexer on one of the inputs of *Adder/Subtracter* is controlled by

Sel_{ADD2}: in the preprocessing phase of a modular multiplication it switches output of memory bank Mem_{OUT2} to calculate $2B$+N, then it closes in reaction the partial result U, in order to update it with the next addendum $2B$, $2B$+N, or N. The S-bit registers *Reg* are pipelining registers. The *Adder/Subtracter* is a simple block which sums operands, or subtracts operand on the right of Fig. 2 from the one on the left. The flip-flop in reaction stores carry-out of the previous operation. The Adder/Subtracter can add or subtract simply xor-ing the second operand with a S-bit word of '0' or '1', in accordance with the $Op_{ADD/SUB}$ signal. In the first step of individual operations, proper activation of the synchronous set/reset inputs of the flip-flop is necessary. *Register U* stores the value of U prior the shifting, as required by the modular product algorithm. It shows its content shifted by 1 bit on the right. It also outputs least significant bit of U, necessary for the *Controller* to choose which is the next operand that is to be added to U. $Signum$ signal states if content of register is less than zero. In fact, the final reduction step is actually performed by storing U (P_H in Alg. 2.2) in the Memory Section, subtracting N from final U. If the result is greater than 0, the current result is output, otherwise the previously stored one is output. *Register A* holds the value of A operand of modular product, which can be loaded serially through Mem_{OUT1} or Bus_{IN}. It can shift one bit at a time, showing the least significant bit to the controller.

3.2 Memory Section and Z-processor

The Memory Section and the Z-processor schematics are reported in Fig. 3. Memory Section supplies P and Z processors with contents of 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 words

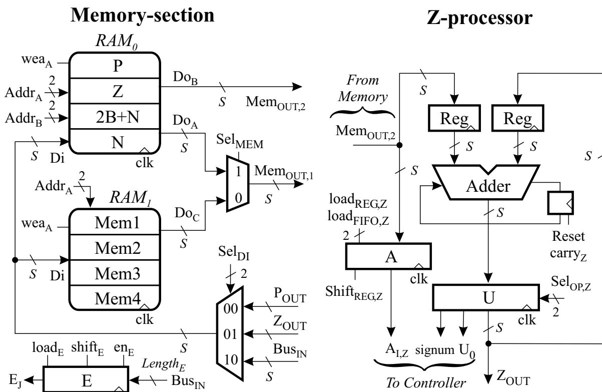

Figure 3. Memory Section and Z-processor

that can be added to partial sum U of Alg. 2.1 (P, Z, $2B$+N, N), while RAM_1 stores K-bit constants that are useful when calculating an N-residue, or returning to normal number representation and when the key is changed ($R^2 \bmod N$, P_0, 1, $R \bmod N$). 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 $RegisterE$, which stores exponent E.

Please note that, strictly speaking, we only need a P-processor and a Memory-Section to implement a 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 overall required area by a factor smaller than 2 (area of Memory Section is constant), so the version of RSA processor with both P and Z processors is characterized by a better value of the product $A \cdot T$. When the available area is reduced and performance is not an issue, the design option which relies solely on the P-processor can gain interest.

3.3 Control Unit

Due to complexity of the algorithm, we split *Control Unit*, in two different coordinated entities *ModExp_Controller* and *MonProd_Controller*, which alternatively take control of the data path. *ModExp_Controller* is implemented as a microcoded architecture (basically a ROM), since it has to sequence instructions managing control flow of Alg. 2.2. *MonProd_Controller*, instead, is designed as a simple FSM. It is in charge of coordinating actions for serial implementation, by means of P and Z processors, of simple instructions such as MonProd, reduction phase, and other steps of the algorithm (see Sec. 3.1). The Control Unit as a whole runs concurrently in pipelining with the Data-Path, in a such way that elaboration of data and sequencing of operations can be overlapped. It is worth noting that no control hazards can happen.

4. Implementation of the RSA processor

4.1 Target Device and Associated Tools

The Xilinx Virtex series of FPGAs, consists of a logic cell (or CLB) and interconnect tiled to form a chip. Each CLB consists of two slices, each slice containing 2 4-LUTs, 2 flip flops, and associated carry chain logic. Each LUT can either be used as 16x1 bit RAM, or a 1-17 cycle delay shift register (SRL16 mode), while the flip flop has a clock enable and a reset. Both the LUT and the flip flop can be accessed independently. A flip-flop is present in each slice of the Virtex architecture: so an S bit register requires S slices. Our approach to limit area growth, is to exploit whenever possible ability of a slice of Virtex to be configured as a memory block (named Block RAM): a LUT can be configured as a 16×1 single-ported RAM, and 2 LUTs as a 16×1 dual-ported RAM or a 16-bit shift-register. We have used a Xilinx Virtex-E 2000-8bg560. Xilinx XCV2000E presents 19200 slices and 19520 tristate buffers. For synthesis we used Synplicity Synplify Pro 7.1, integrated in Xilinx ISE 4.1 design flow in order to use Xilinx Place&Route, Timing Analyzer and tools generating post-mapping and post-place&route VHDL descriptions.

4.2 FPGA implementation

Registers A and U are large registers which may occupy many slices, so implementations other than simple ones (flip-

flop-based), but less area expensive, are desirable. During modular product, two read and one write operations are executed every clock tick on Register U (see 4). In fact, two reads are needed to access an S-bit word which consists of $S-1$ most significant bits of one word of register U and one bit of the adjacent word, since such a word is not aligned with a memory word. This drawback does not affect the memory write back of the result of the serial addition, since this happens to be aligned with a word of Register U. Unfortunately,

CK	0	1	2	3	4	
Read	U1, U0	U2, U1	U3, U2	U4, U3	U5, U4	...
$Addr_{R,EVEN}$	0000	0001	0001	0010	0010	...
$Addr_{R,ODD}$	0000	0000	0001	0001	0010	...
Sel_{OP}	01	10	01	10	01	...
Adder	-	sum	sum	sum	sum	...
Write	-	-	U0	U1	U2	...

Figure 4. Timing of Register U

the Virtex device does not support concurrent execution of two reads and one write on the same RAM at a time. A solution is to increase parallelism using two RAM blocks, arranged as reported in Fig. 5. The *Mixing* block is composed only of wirings, which output four S-bit words in parallel, as reported in the figure. A *Multiplexer* is then used to select one of the S sized outputs. It can be easily proved that this block controlled with signals of Fig. 4, functions properly. We de-

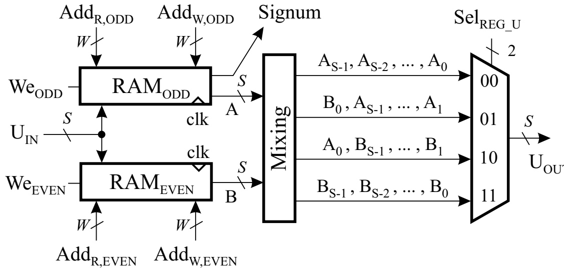

Figure 5. Structural view of register U

cided to implement Register A exploiting the capability of Virtex devices to realize compact shifting registers. Register A is implemented as a first-in-first-out queue of depth $M-1$ and length S, followed by a right shift register S bit wide (Fig. 6). Words composing operand A, are inserted from the least significant to the most significant ones: at the end of the loading phase, in the shift register there are $A_{S-1}, \ldots, A_0$, which can be shifted one bit at a time, showing its less significant bit. When S bits have been shown, the queue moves forward and the next word is loaded into the shift register. A 16-depth $\times 1$ shift register takes one LUT and one flip-flop slice.

4.3 Performance analysis

We implemented our design in RTL VHDL for $K = 1024$ and for widths of S from 32 to 256. First we verified the correctness of the design. We then performed synthesis,

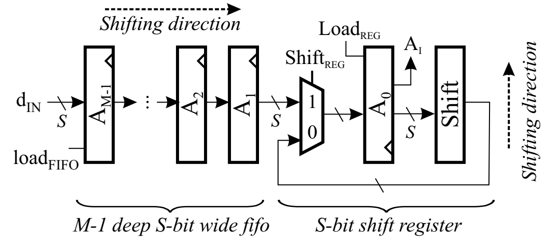

Figure 6. Structural view of register A

place&route step, and timing verification. A partial manually floorplan of different blocks was carried out occasionally, upon need.

Fig. 7 reports minimum clock period and total number of slices required for different values of S and for each stage of the pipeline. It is evident that the stage which limits clock frequency is the Adder. In fact, since net delay is minimized through mapping of carry path into dedicated hardware of CLBs, delay is entirely due to logic and no optimizations can be brought off. 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. Furthermore we exploited the capability of Virtex devices to realize multiplexers using tristate buffers instead of LUTs. In fact, Virtex is very rich of tristates that are used as routing resources. Clearly, tristate-based multiplexers show delays quite larger than their LUT-based counterparts, and so we used them only when it is possible to slow down the cycle period of stages, without reducing the operation frequency of the pipeline. This technique (see column "Tristate Buffers" of Fig. 7) can be adopted for Memory Section and for Write stage of P and Z processors, for every S, except for Memory Section in case $S = 32$.

Number of clock cycles for each modular product is given by $(2M+2) + (K+3) \cdot M$, whereas former contribution is due to A loading into Register A, and preprocessing phase of $2B + N$, and the latter concerns the computation of loop on j of Alg. 2.1, where additions are serial and require M clock ticks. The number of sequential modular products in Alg. 2.2 is $H + 2$ because squarings and products of the loop on i are executed in parallel, and a product is essential for residue calculation (step 2) and for turning residue of result in normal representation (step 8). Finally, M clock ticks are needed for subtraction (step 9) and $M + 1$ clock ticks for the last S bits of result to appear on the output P_{out}. The number of clock ticks (N_{CK}), total area, and total time for exponent $E = 2^{16} + 1$ as functions of parameter S are reported in Fig. 8.

5. Comparison to previously reported implementations

Most hardware implementations of modular exponentiation are either dated or they rely on an ASIC implementation. As a consequence, a fair comparison is difficult.

Reference [8] first combined Montgomery technique and systolic multiplication, proposing a bidimensional systolic

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 implemen-tation
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 7. Hardware resources and clock periods for each pipeline stage varying S

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 8. RSA encryption with 1024 bit key

array architecture which gave a throughput of one modular multiplication per clock cycle and a latency of $2K+2$ cycles. Major drawback of this structure is great area request, deriving from inherent high-parallelism. We compare our results to unidimensional systolic architecture proposed in [9]. This work implements the same algorithm as ours (radix-2 Montgomery algorithm for modular product Alg. 2.1), on top of a Xilinx device with the same basic architecture, but with a different technology. More precisely, a Virtex-E 2000-8 and a XC40250XV-09 were used in our study and in theirs, respectively. In a successive study, Blum and Paar in [10] improved their architecture using a high-radix formulation of Alg. 2.1. Our architecture can also exploit a higher radix, and we are planning to do so. At the time of this writing, we can only compare our current implementation to their radix-2 based one. It should be noted, however, that both [9] and [10] solutions have a major flaw, that is occasionally an incorrect value of $(X^E \bmod N) + N$ is return (instead of $(X^E \bmod N)$). Authors of [9] argue that this situation is infrequent because it has a probability of $2^{-(K+2)}$. So based on the observation that most of the times correct result is available prior the reduction pass, they simply do not execute the final reduction pass (steps 9-10 of Alg. 2.2), thus saving both time and hardware resources. In contrast, our architecture always provides the correct result. This happens at a cost, in terms of time and hardware resources, as compared to [9], due to the extra operation.

To output the entire result, [9] 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 [9] requires 0.75 ms for the same encryption of Fig. 8 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 in [9] requires 3786 slices. Finally, our design with best $A \cdot T$ (S = 64) presents $A \cdot T = 4586$, while the corresponding design of [9], presents $A \cdot T = 3511$.

In summary, the solution presented in [9] 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. However, the solution presented in [9] has occasionally an incorrect behavior, resulting in wrong outputs. On the other side, our implementation is slower, but it always produces correct outputs, and also has lower area requirements.

6. Conclusions

We presented a novel serial architecture for RSA encryption/decryption operation. The design is targeted for implementation on reconfigurable logic, and exploits inherent characteristics of the Xilinx devices, for compact implementation of shift registers via distributed RAM blocks, and for multiplexers via tristate buffers. The design allows to tradeoff area for performance, by modifying the value of the serialization factor S. Quantitative evaluation of such a tradeoff is conducted via simulation, and results are discussed. Results show that the presented architecture achieves good performance with low area requirements, as compared to other architectures.

References

[1] B. Schneier, Applied Cryptography, New York: Wiley, 1996.

[2] R. L. Rivest et al., “A Method for Obtaining Digital Signatures”, Commun. ACM, vol. 21, pp. 120-126, 1978.

[3] P. Downey et al., “Computing sequences with addition chains”, SIAM J. on Computing, 3:638-696, 1981.

[4] Ç. K. Koç, “High-speed RSA Implementation”, Technical Report TR 201, RSA Laboratories, November 1994

[5] D.E. Knuth, “The Art of Computer Programming: Seminumerical Algorithms”, vol. 2, Addison-Wesley, 1981.

[6] G. R. Blakley, “A computer algorithm for the product...”, IEEE Trans. on Computers, Vol.32, No.5, pp. 497-500, May 1983.

[7] P. L. Montgomery, “Modular multiplication without trial division”, Math. of Computation, 44(170):519-521, April 1985.

[8] C. D. Walter, “Systolic Modular Multiplication”, IEEE Trans. on Computers, Vol.42, No.3, pp. 376-378, March 1993.

[9] T. Blum, and C. Paar, “Montgomery Modular Exponentiation...”, Proc. 14th Symp. Comp. Arith., pp. 70-77, 1999.

[10] T. Blum, and C. Paar, “High-Radix Montgomery Modular...”, IEEE Trans. on Comp., Vol.50, No.7, pp. 759-764, July 2001.