RSA Acceleration with Field Programmable Gate Arrays

Alexander Tiountchik[1] and Elena Trichina[2]

[1] Institute of Mathematics, National Academy of Sciences of Belarus,
11 Surganova str, Minsk 220072, Belarus
aat@im.bas-net.by
[2] Advanced Computing Research Centre, University of South Australia,
Mawson Lakes, SA 5095, Australia
elena.trichina@unisa.edu.au

Abstract. An efficient implementations of modular exponentiation, i.e., the main building block in the RSA cryptographic scheme, is achieved by first designing a bit-level systolic array such that the whole procedure of modular exponentiation can be carried out entirely by a single unit without using global interconnections or memory to store intermediate results, and then mapping this design onto Xilinx XC6000 Field Programmable Gate Array.

1 Introduction

Many popular cryptographic schemes, such as the RSA scheme [13], ElGamal scheme [6], Fiat-Shamir scheme [8], etc., make extensive use of modular exponentiation of long integers. However, it is a very slow operation when performed on a general purpose computer. A cheap and flexible modular exponentiation hardware accelerator can be achieved using Field Programmable Gate Arrays [2]. In this paper we do not compete with industrial-strength special purpose hardware for modular exponentiation. Rather we use a complexity of the problem as a benchmark for evaluating computing power of fine grained FPGAs, and for developing a more systematic methodology for their programming.

We propose a two-step procedure for an implementation of modular exponentiation on FPGAs. The main idea is as follows. Bit-level systolic arrays share many limitations and constraints with FPGAs; both favor regular repetitive designs with local interconnections, simple synchronisation mechanisms and minimal global memory access. While programming FPGAs is still pretty much an *ad hoc* process, there is a mature methodology of bit-level systolic systems design. Thus, to achieve a good FPGA implementation, it may be beneficial first to design a systolic array for a given application, and then map this array onto FPGAs in a systematic fashion, preserving the main properties of the systolic design.

In this paper an efficient systolic array for a modular exponentiation such that the whole exponentiation procedure can be carried out entirely by the single systolic unit without global interconnections or use of global memory to store

J. Pieprzyk, R. Safavi-Naini, and J. Seberry (Eds.): ACISP'99, LNCS 1587, pp. 164–176, 1999.

intermediate results is designed first. This procedure is based on a Montgomery multiplication, and uses a high-to-low binary method of exponentiation. Moreover, this array is expected to be faster than similar devices performing exponentiation by repeated modular multiplications of an integer by itself [17, 9].

The next step consists of a systematic mapping of the systolic array onto fine grained FPGAs. During this experiment a number of observations emerged, which we present in this paper. Our final design accommodates a modular exponentiation of a 132-bit long number on one Xilinx XC6000 chip comprising 64×64 elementary logic cells.

Reported in this paper hardware implementation relies on configurability of FPGAs, but does not use run-time reprogrammability or/and SRAM memory (intermediate results are stored in registers implemented within individual cells). This makes our design simpler and easy to implement. The price to pay is that more chips are needed to implement RSA with a longer key. 4 Kgates, or one XC6000 chip, is required for modular exponentiation of 132-bit long integers. 512-bit long integers need 4 XC6000 chips connected in a pipeline fashion, or 16 Kgates. The bit rate for a clock frequency of 25 MHz can be estimated to be approximately 800 Kb/sec for 512 bit keys, which is comparable with the rate reported in a fundamental paper of Shand and Vuillemin [15], and an order of magnitude better than that the ones in [9] and [12].

2 Modular Exponentiation of Long Integers

The main and most time consuming operation in the RSA algorithm is modular exponentiation of long integers. The RSA Laboratories recommended key sizes are now 768 bits for personal use, 1024 bits for corporate use, and 2048 bits for extremely valuable keys. A 768-bit key is expected to be secure until at least the year 2004.

A modular exponentiation operation $M^e \bmod n$ cannot be implemented in a naive fashion by first exponentiating M^e and then performing reduction modulo n, since even if M and e have only 256 bits each, the intermediate result M^e contains $\approx 10^{80}$ digits. Hence, the intermediate results of the exponentiation are to be reduced modulo n at each step. The straightforward reduction modulo n involves a number of arithmetic operations (division, subtraction, etc.), and is very time consuming. Therefore, special algorithms for modular operations are to be used.

In 1985, P. L. Montgomery [10] proposed an algorithm for modular multiplication $AB \bmod m$ without trial division. In [1] different modular reduction algorithms for large integers were compared with respect to their performance and the conclusion was drawn that for general modular exponentiation the exponentiation besed on Montgomery's algorithm has the best performance.

2.0.1 Montgomery Multiplication

Let A, B be elements of $\mathbf{Z}_m$, where $\mathbf{Z}_m$ is the set of integers between 0 and $m-1$. Let h be an integer coprime to m, and $h > m$.

Definition. *Montgomery multiplication (MM)* is an operation

$$A \overset{h,m}{\otimes} B = A \cdot B \cdot h^{-1} \bmod m. \tag{1}$$

Implementation of this operation is much easier than a normal reduction modulo m ; and is based on some facts from number theory. The use of MM does not result in the desirable speed-up immediately. To compute $AB \bmod m$, a computation of MM is to be performed twice:

1. $C = A \overset{h,m}{\otimes} B = A \cdot B \cdot h^{-1} \bmod m$, and
2. $C \overset{h,m}{\otimes} (h^2 \bmod m) = ABh^{-1} \cdot h^2 \cdot h^{-1} \bmod m = AB \bmod m$,

where $h^2 \bmod m$ is computed in advance. The advantage of using two Montgomery multiplications instead of one operation of plain modular multiplication is uncertain.

2.0.2 Montgomery Exponentiation

An efficient way to compute $AB \bmod m$ using MM is by exploiting special representations of A and B.

Definition. $\widehat{X}$ is called an *image* of X if $\widehat{X} = X \cdot h \bmod m$, $h > m$.

If h and m are relatively prime, then there exists a one-to-one correspondence between X and $\widehat{X}$. MM of $\widehat{A}$ and $\widehat{B}$ is isomorphic to the modular multiplication of A and B. Indeed, $\widehat{A} \overset{h,m}{\otimes} \widehat{B} = (Ah \bmod m \overset{h,m}{\otimes} (Bh \bmod m) = (AB)h \bmod m = \widehat{A \cdot B}$. The reduction of X to $\widehat{X}$ and vice versa can be carried out on the basis of MM:

$$X \overset{h,m}{\otimes} (h^2 \bmod m) = X \cdot h^2 h^{-1} \bmod m = \widehat{X}, \tag{2}$$

$$\widehat{X} \overset{h,m}{\otimes} 1 = X \cdot h \bmod m \cdot 1 \cdot h^{-1} \bmod m = X. \tag{3}$$

By virtue of the isomorphism of modular multiplication and MM, the use of the images is very convenient for exponentiation. Let $(\overset{h,m}{\otimes} X)^n$ denote $(n-1)$ MMs of X by itself. To compute $Y = X^n \bmod m$, we should perform three steps: first, convert X to $\widehat{X}$ by (2); next, realize $\widehat{Y} = (\overset{h,m}{\otimes} \widehat{X})^n = \widehat{X^n}$; and finally, convert $\widehat{Y}$ to Y by (3).

2.1 Algorithm for Implementation of Montgomery Multiplication

Several algorithms suitable for hardware implementation of MM are known [9, 14,17,4,3]. In this paper, the design of a systolic array is based on the algorithm described and analysed in [17]. Let numbers A, B and m be written with radix 2:

$$A = \sum_{i=0}^{N-1} a_i \cdot 2^i, \qquad B = \sum_{i=0}^{M} b_i \cdot 2^i, \qquad m = \sum_{i=0}^{M-1} m_i \cdot 2^i,$$

where a_i, b_i, $m_i \in \mathbf{GF}(2)$, N and M are the numbers of digits in A and m, respectively. B satisfies condition $B < 2m$, and has at most $M + 1$ digits. m is odd (to be coprime to the radix 2). Extend a definition of A with an extra zero digit $a_N = 0$. The algorithm for MM is given below (4).

$$\begin{array}{l} s := 0; \\ \textbf{For } i := 0 \textbf{ to } N \textbf{ do} \\ \textbf{Begin} \\ \qquad u_i := ((s_0 + a_i * b_0) * w) \bmod 2 \\ \qquad s := (s + a_i * B + u_i * m)\mathrm{div}2 \\ \textbf{End} \end{array} \tag{4}$$

Initial condition $B < 2m$ ensures that intermediate and final values of s are bounded by $3m$. The use of an iteration with $a_N = 0$ ensures that the final value $s < 2m$ [17]. Hence, this value can be used for B input in a subsequent multiplication. Since 2 and m are relatively prime, we can precompute value $w = (2 - m_0)^{-1} \bmod 2$. An implementation of the operations div2 and mod2 is trivial (shifting and inspecting the lowest digit, respectively). Algorithm (4) returns either $s = A \cdot B \cdot 2^{-n-1} \bmod m$ or $s + m$ (because $s < 2m$). In any case, this extra m has no effect on subsequent arithmetics modulo m. It should be noted, that the number of iterations in (4) affects h [17]. In our case, (4) presents the implementation of (1) with $h = 2^{N+1}$.

2.2 Graph Model for Montgomery Multiplication

Using standard methods for systolic systems design, first we construct a data dependency graph (also referred as DG, or graph model) for Algorithm (4). This graph is depicted in Fig. 1 (see also [17,16]). For N– and M–digit integers A and B, a graph consists of $N+2$ rows and $M+1$ columns. The i-th row represents the i-th iteration of (4). Arrows are associated with digits transferred along indicated directions. Each vertex $v(j, i)$, $i \in \{0, ..., N\}, j \in \{0, ..., M\}$ is associated with the operation

$$s_{j-1}^{(i+1)} + 2 \cdot c_{out} := s_j^{(i)} + a_i \cdot b_j + u_i \cdot m_j + c_{in},$$

where $s_j^{(i)}$ denotes the j-th digit of the i-th partial product of s, c_{out} and c_{in} are the output and input carries. Rightmost starred vertices, i.e., vertices marked with "$*$", perform calculations of $u_i := ((s_0 + a_i * b_0) * w) \bmod 2$ besides an ordinary operation. Using standard notation, the vertex operations can be specified in terms of inputs/outputs as follows:

$$\begin{gathered} s_{out} + 2 \cdot c_{out} := s_{in} + a_{in} \cdot b_{in} + u_{in} \cdot m_{in} + c_{in}, \\ a_{out} := a_{in}, \qquad b_{out} := b_{in}, \\ u_{out} := u_{in}, \qquad m_{out} := m_{in}, \end{gathered} \tag{5}$$

for plain vertices, and

$$u_{out} := (s_{in} + a_{in} \cdot b_{in}) \cdot w_{in},$$

$$c_{out} := \mathrm{maj}_2(s_{in}, a_{in} \cdot b_{in}, u_{in} \cdot m_{in}), \tag{6}$$

$$a_{out} := a_{in}, \qquad b_{out} := b_{in}, \qquad m_{out} := m_{in},$$

for starred vertices, where $\mathrm{maj}_2(s_{in}, a_{in} \cdot b_{in}, u_{in} \cdot m_{in})$ is 1 if at least two out of three entries are 1s; otherwise it is 0.

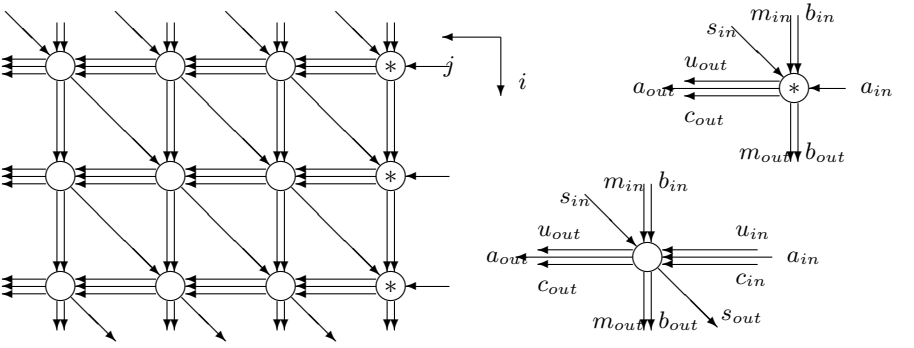

Fig. 1. Graph model for $A \overset{h,m}{\otimes} B$: case of $N = 2$, $M = 3$.

2.2.1 High-to-Low Binary Method of Exponentiation

If instead of digits A we input digits B both, at the topmost and rightmost vertices of the graph in Fig. 1, then the graph model represents a calculation of

$$B \overset{h,m}{\otimes} B = (\overset{h,m}{\otimes} B)^2,$$

called M-*squaring*. To represent the computation of $(\overset{h,m}{\otimes} B)^3$, two graphs can be joined in a single graph by connecting s_j-outputs of the first graph with b_j-inputs of the next (identical) graph, in which rightmost inputs a_i get digits of B as before. To compute $(\overset{h,m}{\otimes} B)^n$, we will need $n-1$ joined graphs. The resulting graph model consists of vertices located in a rectangular domain $V_1 = \{v(i,j) | 0 \le i \le n \times (M+2) - 1,\ 0 \le j \le M+1\}$. The graph is almost homogeneous, with exceptional starred vertices in the rightmost column.

However, a faster way to compute $B^n \bmod m$ is by reducing the computation to a sequence of modular squares and multiplications [15]. Let $[n_0 \dots n_k]$ be a binary representation of n, i.e., $n = n_0 + 2n_1 + + \cdots + 2^k n_k$, $n_j \in \mathbf{GF}(2)$, $k = \lfloor \log_2 n \rfloor$, $n_k = 1$. Let β denote a partial product. We start out with $\beta = B$ and run from n_{k-1} to n_0 as follows: if $n_j = 0$, then $\beta := \beta^2$; if $n_j = 1$, then $\beta := \beta^2 * B$. Thus, we need at most $2k$ operations to compute B^n. This algorithm has an advantage over a low-to-high binary method of exponentiation since, when implemented in hardware, it requires only one set of storage registers for intermediate results as opposed to two for a low-to-high method [15].

2.3 Graph Model for Squaring

To perform M-squaring the dependency graph for M-multiplication can be modified in such a way that all the b_j's inputs enter the graph only via the top-row vertices [16]. This eliminates rightmost a_i-inputs entirely. To deliver all b_js to the rightmost vertices, we have to pump them through the graph in a direction determined by vector $(1, -1)$. To do it, additional arcs x_j's for propagation of b_j's digits have to be added to a dependency graph in Fig. 1. Vertex operations are to be slightly modified to provide propagation of these digits: each non-starred vertex just transmits its x-input data to an x-output, while when arriving at the rightmost vertices, these data are "reflected" and propagated to the left as if they were ordinary a_i's input data. It is known that the output value $s < 2m$. Hence, we need at most $M + 1$ rows for the "reflected" factor and an additional row for the extension with an extra zero digit. A graph model for M-squaring is depicted in Fig. 2.

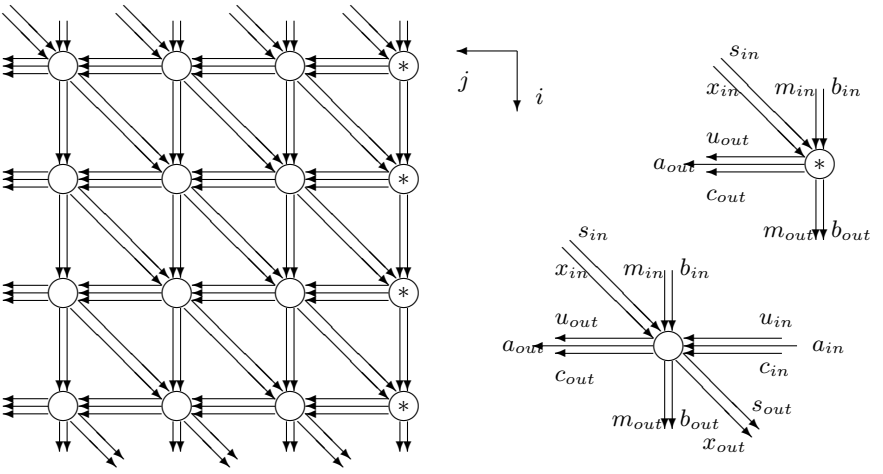

Fig. 2. Graph model for $\overset{h,m}{\otimes} B^2$: case of $M = 3$.

Using standard notation, the vertex operations for M-squaring can be specified in terms of inputs/outputs as follows:

$$\begin{gathered} s_{out} + 2 \cdot c_{out} := s_{in} + a_{in} \cdot b_{in} + u_{in} \cdot m_{in} + c_{in}, \\ a_{out} := a_{in}, \qquad x_{out} := x_{in}, \\ b_{out} := b_{in}, \qquad u_{out} := u_{in}, \qquad m_{out} := m_{in}, \end{gathered} \tag{7}$$

for plain vertices, and for starred vertices the operation is:

$$\begin{gathered} u := (s_{in} + x_{in} \cdot b_{in}) \cdot w_{in}, \\ s_{out} + 2 \cdot c_{out} := s_{in} + x_{in} \cdot b_{in} + u \cdot m_{in}, \\ c_{out} := \mathrm{maj}_2(s_{in}, x_{in} \cdot b_{in}, u_{in} \cdot m_{in}), \\ u_{out} := u, \qquad a_{out} := x_{in}, \qquad b_{out} := b_{in}, \qquad m_{out} := m_{in}. \end{gathered} \tag{8}$$

2.4 Linear Systolic Array for Modular Exponentiation

A graph model for an exponentiation as a whole is constructed as a composition of graphs for M-multiplication and M-squaring by joining outputs of one graph with corresponding inputs of the consecutive graph. There are at most $2k$ graphs altogether, and the precise number of required graphs for M-multiplication and M-squaring and the order in which they occur in the composition is fully determined only by the binary representation of n. The vertices of the resulting graph constitute a rectangular domain $V_2 = \{v(i,j) | 0 \leq i \leq 2k \times (M+2),\ 0 \leq j \leq M+1\}$, where $k = \lfloor \log_2 n \rfloor$. [1]

The next stage of the systolic design is a space-time mapping of domain V_2 onto a one-dimensional domain of processing elements (PE). Spatial mapping is determined by a linear operator with matrix $P = (1\ \ 0)$ [16], which maps an indefinitely long composition of the cohered DGs onto a linear systolic array with $M+1$ processing elements: each column of vertices is mapped onto one PE, as shown in Fig. 3. Hence, each PE in Fig. 3 has to be able to operate in two modes. To control the operation modes, a sequence of one-bit control signals τ is fed into the rightmost PE and propagated through the array. If $\tau = 0$ the PE implements an operation for M-multiplication, if $\tau = 1$, for M-squaring. The order in which control signals are input is determined by the binary representation of n. A timing function that provides a correct order of operations is $t(v) = 2i + j$ [16]. The total running time is thus at most $(4\lfloor \log_2 n \rfloor + 1)M + 8\lfloor \log_2 n \rfloor$ time units.

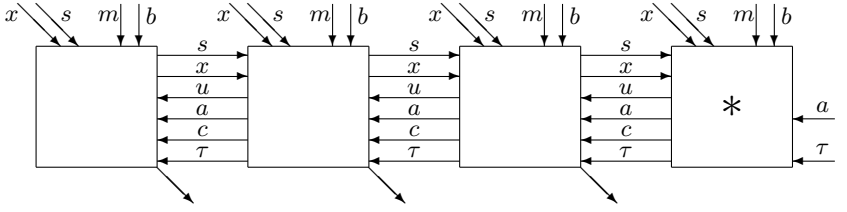

Fig. 3. Linear Systolic Array.

3 Logic Design of the FPGA Implementation of Montgomery Exponentiation

Our next step is to implement the systolic array on FPGAs. The purpose of this experiment is threefold: firstly, derive a systematic method of mapping systolic algorithms into a sea of cells; secondly, construct an efficient FPGA implementation of a particularly important application; and thirdly, investigate the limits of a fine grained FPGA chip for modular exponentiation. We conducted our experiments with Xilinx XC6000, comprising 64×64 logic cells.

3.1 Inputs and Outputs

To meet limitations of FPGAs that input/output ports are located along the borders of the chip, we found the following ideas fruitful. The first step of any

[1] $2k \times (M+2)$ is the largest number of all possible rows in a resulting graph model.

exponentiation is always squaring, β^2; but this step can be implemented as multiplication $\beta \cdot \beta$ (where $\beta = B$, an input number). Since all s_i's are 0's at the first step of any multiplication and squaring, and for all other steps the s_i's are to be the results of the previous step, we can use registers to store intermediate s_i's, and instead of loading 0's from the host, just set all these registers to 0 initially. The dependency graph for M-multiplication does not have inputs x_i's, $i \in \{0, ..., M+1\}$; hence, the final design does not need these inputs since input values for x_i's required for M-squaring will be generated later, and can be loaded from the registers containing the results s_j's of the previous stage of computations.

All inputs b_i's for the topmost row of the first graph for M-multiplication must receive corresponding bits of input B (the same as the rightmost inputs a_i's) while for all consecutive graphs these inputs are to be connected with s_j's- or b_j's-outputs of the previous graph, depending of whether this stage is multiplication or squaring. An additional control signal σ must be used to provide the correct assignment for registers associated with top row vertices. Thus, instead of having $M+1$ "vertical" inputs for b_i's, we shall use registers and load them initially with corresponding bits b_i's from inputs a_i's; new values computed at later stages and used as inputs for the consequent graph, will be reassigned to these registers as described above. Hence, the only I/O ports actually needed in the design, are located at the rightmost and leftmost processing elements.

3.2 Logic Design for the Non-starred PE

Consider now the logic design for implementation of an individual PE in details. A minor optimisation first. Modes of the plain PE for multiplication and squaring differ only by the transmission or absence of the transmission of data x_i, and the control signal τ is used to distinguish these two modes. However, x_i's do not affect computations in the non-starred PEs; they are used only in the rightmost (starred) PEs. Therefore, we can ignore the existence of two modes for the plain PE and let it transmit x_i regardless τ. Hence, we do not need a control signal τ in the plain PEs; τ should be used only in the rightmost PE where it defines whether an input data x_i is to be used or ignored. Nevertheless, we need a control signal σ to ensure the correct initial assignments to x_i's depending on whether M-multiplication or M-squaring should be carried out: $x_{out} := \mathrm{mux}(\sigma_{in} : x_{in}, s_{in})$.

Original input data b_i's are to be stored in the local memories of PEs for future M-multiplications. We use a special register $b_{in}^{<\mathrm{old}>}$ for this purpose. As above, a control signal σ is used to provide the correct initial assignments to variables b_i's depending on whether PEs are supposed to perform multiplication or squaring: $b_{in} := \mathrm{mux}(\sigma_{in} : b_{in}^{<\mathrm{old}>}, s_{in})$.

A computational part of the main PE includes control over input data, two logic multiplications, $a_{in} \cdot b_{in}$ and $u_{in} \cdot m_{in}$, and addition of these products with an intermediate sum s_{in} and input carries. Evidently, four-element addition can generate two carries meaning that all main PEs will have two input carries, and produce two output carries; the first carry $c_{out}^{<1>}$ is to be used by the nearest neighbor PE, and the second carry $c_{out}^{<2>}$ is to be used by the PE followed after

the nearest neighbor:

$$s_{out} + 2c_{out}^{<1>} + 4c_{out}^{<2>} = a_{in} \cdot b_{in} + u_{in} \cdot m_{in} + s_{in} + c_{in}^{<1>} + c_{in}^{<2>}.$$

We shall denote the carry that is just a transit from the right neighbor to the left one by $c^{<T>}$, hence $c_{out}^{<T>} := c_{in}^{<2>}$. It is not uncommon to implement addition of 5 entries using two full adders and one half adder, which can be found in a standard library XC6000 provided by EXACTStep6000, with the "communication" part surrounding this module. However, if the outputs of some gates are to be stored in registers, these gates and registers should not be at different hierarchical levels, because normally a gate–register pair may occupy only one cell but if the gate is embedded in a module, while the register is outside of this module they inevitably will be placed in different cells, and often rather far apart. Thus, if registers are to be used to store output data of a module, it is desirable to insert these registers inside the module. Fig. 4 presents the final design for a plain PE.

3.3 Logic Design for the Rightmost PE

The rightmost PE selects correct values for its b– and a–inputs, depending on control signals σ and τ, propagates data and signals to the left neighbor, computes value u and the sum $a_{in} \cdot b_{in} + u \cdot m_{in} + s_{in}$. For consistency, two zero carries should also be generated. Below we give a description of the rightmost PE, including the specification of data and control signals transmissions:
$a'_{in} = \mathrm{mux}(\tau : x_{in}, a_{in})$;
$b'_{in} = \mathrm{mux}(\sigma_{in} : b_{in}, s_{in})$;
$u_{in} = (a'_{in} \cdot b'_{in} \oplus s_{in}) \cdot w_{in}$;
$s_{out} + 2c_{out}^{<1>} = a'_{in} \cdot b'_{in} + u_{in} \cdot m_{in} + s_{in}$;
$u_{out} = u_{in}$; $a_{out} = a'_{in}$; $c_{out}^{<2>} = 0$; $c_{out}^{<T>} = 0$.
The main computational module of the rightmost PE after the optimisation and its structure as a whole are depicted in Fig. 5 and Fig. 6, respectively.

4 XACTStep 6000 Automatic Design and Its Optimisation

A high level of correspondence between the requirements of bit-level systolic arrays and FPGA designs provide an opportunity to implement a modular exponentiation algorithm on Xilinx XC6000 chips ensuring a very dense allocation of gates.

An ultimate design goal in our experiment was to find an absolute limit of the number of bits in Montgomery exponentiation, that can be handled by one 64×64 XC6000 chip without storing intermediate data in SRAM and without reprogramming the chip. By trial and error we found that automatic allocation provides successful routing for systolic arrays with a maximum of 67 PEs (1 starred, and 66 plain). Obviously, this is far from the limit. We decided to use it as a starting point for manual optimisation. Remote gates and registers were brought closer together so as to provide locality of interconnections and higher density of the overall design. For manual allocation of gates at the level

of ViewLogic design, the RLOC (relative location) attribute has to be used. The attribute determines the coordinates of a gate inside its module.

To embed a long and narrow one-dimensional array of PEs constituting a systolic design into a XC6000 64×64 square of logic cells, a natural solution is to partition this array into blocks of PEs with respect to the width of the chip, so that every block can be allocated in a side-to-side line on a chip; and then combine these blocks in a "zig-zag" snake-like structure. The length of the block is determined empirically, and better to be estimated conservatively, so as to allow for some extra space to permit successful routing in the corners. In our case one block constitutes 13 PEs. It should be noted that an allocation of PEs inside the block must be manual since we want a long narrow band of the gates while an automatic allocation is trying to provide a square–like allocation.

To eliminate irregularity and criscross connections between PEs in every second block of the zig-zag, we had to design a mirror image for a block by reflecting the block itself. Every PE inside the reflected block has to be a mirror image under reflection of a regular PE with the same functionality; and two additional types of mirror images under rotation were used for the leftmost and the rightmost PEs in a block to ensure locality of logic connections where the "snake" turns. All this allows us to allocate 132 PEs successfully on XC6000 64×64 logic cells. In other words, we can exponentiate a 132-bit long integer on one Xilinx XC6000 chip. An allocation of 132 PEs on a chip and successful routing is presented in Fig. 7.

5 Summary

We presented a new implementation of a modular exponentiation algorithm based on a Montgomery multiplication operation on fine–grained FPGAs. With hand–crafted optimisation we managed to embed a modular exponentiation of 132-bit long integers into one Xilinx XC6000 chip, which is to our knowledge one of the best fine-grained FPGA designs for a modular exponentiation reported so far. 2,615 out of 4,096 gates are used for computations, and 528 for registers, providing 75% density. This array can be used for both, reducing B to $\widehat{B}$ by (2) and for reducing a final value $\widehat{B^n}$ to B^n by (3).

Hence, 4 Kgates (one XC6000 chip) is required for modular exponentiation of 132-bit long integers. 512-bit long integers need 4 XC6000 chips connected in a pipeline fashion, or 16 Kgates. Taking into account the total running time (see Chapter 2), we can estimate the bit rate for a clock frequency of 25 MHz being approximately 800 Kb/sec for 512 bit keys, which is comparable with the rate reported in a fundamental paper of Shand and Vuillemin [15], and an order of magnitude better than that one in [9] and [12]. Further improvement of the proposed implementation can be achieved by simplification of the operation performed by the starred vertices to an ordinary (non-starred vertex) operation due to shifting B up to make $b_0 = 0$ [5]. The reduction of complexity of starred vertices decreases an overall time and provides higher clock rate.

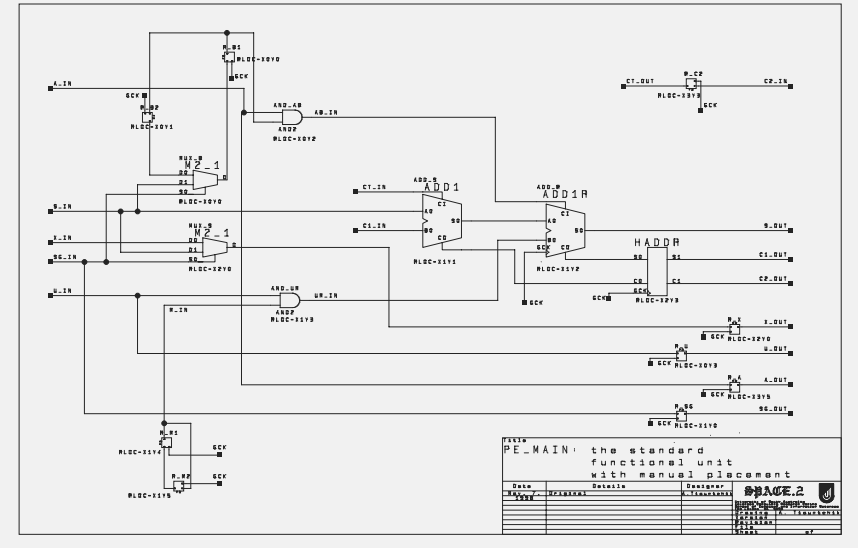

Fig. 4. Logic design for not-starred PE.

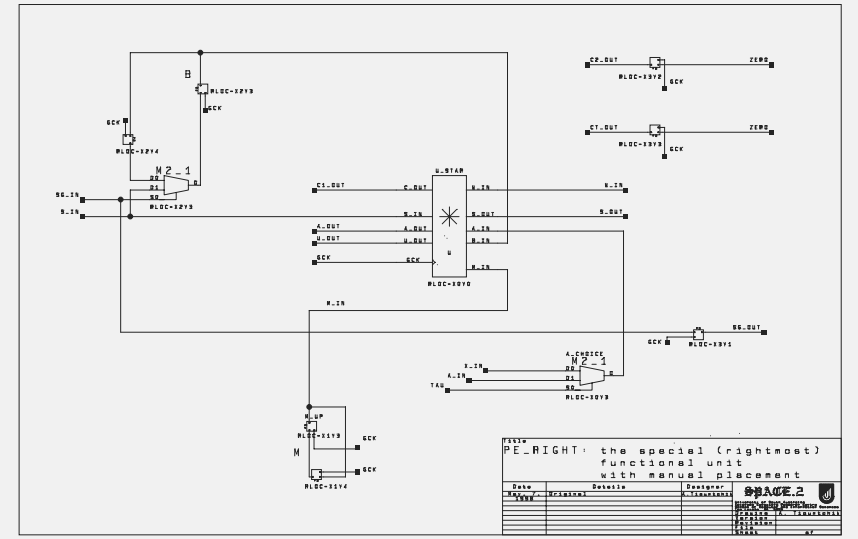

Fig. 5. Logic design for starred PE.

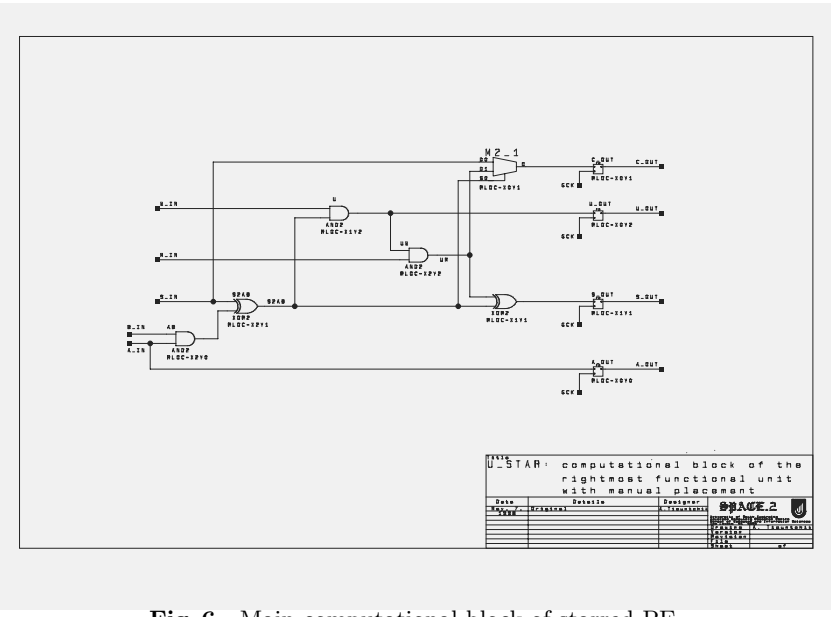

Fig. 6. Main computational block of starred PE.

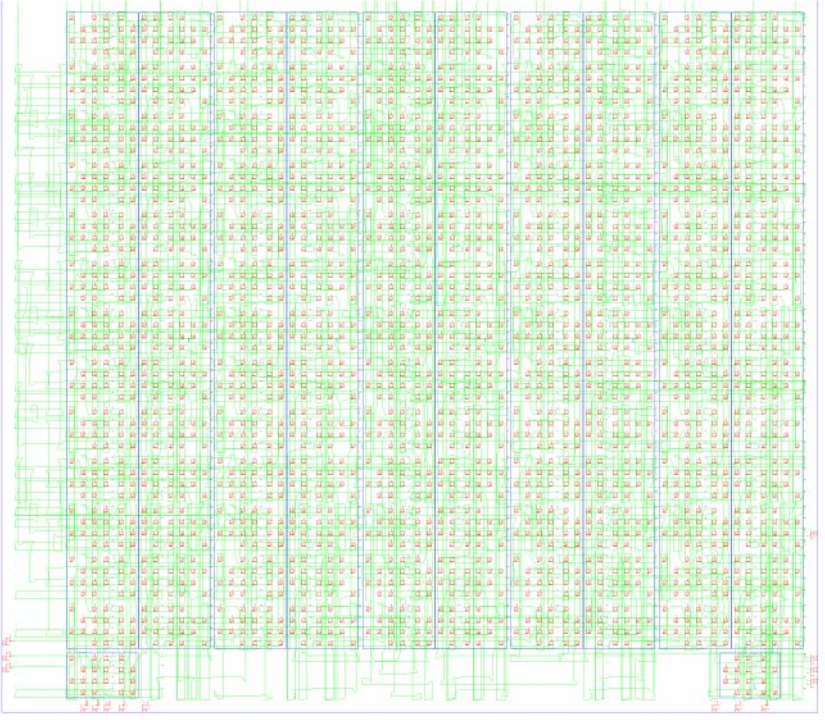

Fig. 7. Allocation and successful routing for 132 PEs.

Acknowledgement

We thank Raymond Keifer for continuous help and advice on XACTStep 6000 and for maintaining SPACE2 reconfigurable computer on which our experiments took place.

References

1. A. Bosselaers, R. Govaerts, and J. Vandewalle, Comparison of three modular reduction functions. pp 175–186
2. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder (Eds.), Splash 2: FPGAs in a custom computing machine. IEEE Computer Society Press, 1996.
3. S. R. Dusse and B. S. Kaliski, A Cryptographic Library for the Motorola DSP56000, in *Advances in Cryptology – EUROCRYPT'90*, Spinger-Verlag, LNCS, vol. 473 (1990) 230–244.
4. S. E. Eldridge, A faster modular multiplication algorithm. Intern. J. *Computer Math.*, 1993 (40) 63–68.
5. S. E. Eldridge and C. D. Walter, Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Trans. on Comput.*, 1993 (42) 693–699.
6. T. ElGamal, A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, 1985 (31) 469–472.
7. S. Even, Systolic modular multiplication, in *Advances in Cryptology – Crypto'90*, Springer-Verlag, LNCS, vol. 537 (1990) 619–624.
8. A. Fiat and A. Shamir, How to prove yourself, in *Advances in Cryptology – Crypto'86*, Springer-Verlag, LNCS, vol. 263 (1986) 186–194.
9. K. Iwamura, T. Matsumoto and H. Imai, Modular Exponentiation Using Montgomery Method and the Systolic Array, IEICE Technical Report, vol. 92, no. 134, ISEC92-7, 1992, pp.49–54.
10. P. L. Montgomery, Modular multiplication without trial division. *Mathematics of Computations*, 1985 (44) 519–521.
11. Ç. K. Koç, RSA Hardware Implementation, TR 801, RSA Laboratories, 30 pages, April 1996. http://www.ece.orst.edu/ koc/vita/v22ab.html
12. H. Orup, E. Svendsen, E. And, VICTOR an efficient RSA hardware implementation. In: *Eurocrypt 90*, LNCS, vol. 473 (1991) 245–252
13. R. Rivest, A. Shamir and L. Adleman, A method of obtaining digital signatures and public key cryptosystems. J. +em Commun. of ACM, 1978 (21) 120–126.
14. J. Sauerbrey, A Modular Exponentiation Unit Based on Systolic Arrays, in *Advances in Cryptology – AUSCRYPT'93*, Springer-Verlag, LNCS, vol. 718 (1993) 505–516.
15. M. Shand, J. Vuillemin, Fast Implementation of of RSA Cryptography. In *Proc. of the 11th IEEE Symposium on Computer Arithmetics*, 1993. pp.252–259.
16. A. A. Tiountchik, Systolic modular exponentiation via Montgomery algorithm. J. *Electronics Letters*, 1998 (34).
17. C. D. Walter, Systolic Modular Multiplication. *IEEE Trans. on Comput.*, 1993 (42) 376–378.