FPGA Implementation of Modular Exponentiation

Alexander Tiountchik
Institute of Mathematics, National Academy of Sciences of Belarus,
11 Surganova st, Minsk 220072, Belarus
e-mail: aat@im.bas-net.by

Elena Trichina
Advanced Computing Research Centre, University of South Australia,
Mawson Lakes, SA 5095, Australia
e-mail: elena.trichina@unisa.edu.au

Abstract. An efficient implementations of the main building block in the RSA cryptographic scheme is achieved by mapping a bit-level systolic array for modular exponentiation onto Xilinx FPGAs. One XC6000 chip, or 4 Kgates accommodates 132-bit long integers. 16 Kgates is required for modular exponentiation of 512 bit keys, with the estimated bit rate 800 Kb/sec.

1 Systolic Array for Modular Exponentiation

The design of public key cryptography hardware is an active area of research because the speed of cryptographical schemes is a serious bottleneck in many applications. A cheap and flexible modular exponentiation hardware accelerator can be achieved using Field Programmable Gate Arrays. FPGA design presented in this paper is based on 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 use of global memory. This procedure is based on a Montgomery multiplication [2], and uses a high-to-low binary method of exponentiation.

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 $h > m$. *Montgomery multiplication* (MM) is an operation $A \overset{h,m}{\otimes} B = A \cdot B \cdot h^{-1} \bmod m$. Several algorithms suitable for hardware implementation of MM are known [1, 4, 7]. In this paper, as a starting point we use the algorithm described and analysed in [7]. Let numbers A, B and m be written with radix 2: $A = \sum_{i=0}^{N-1} a_i \cdot 2^i$, $B = \sum_{i=0}^{M} b_i \cdot 2^i$, $m = \sum_{i=0}^{M-1} m_i \cdot 2^i$, 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. Extend a definition of A with an extra zero digit $a_N = 0$. The algorithm for MM is given below (1).

```
$s := 0;$
For $i := 0$ to $N$ do
Begin
    $u_i := \big((s_0 + a_i * b_0) * w^1\big) \bmod 2$        (1)
    $s := (s + a_i * B + u_i * m)\mathrm{div}2$
End
```

Using B instead of A results in calculating $B \overset{h,m}{\otimes} B = (\overset{h,m}{\otimes} B)^2$, called M-*squaring*. A fast way to compute $B^n \bmod m$ is by reducing the computation to a sequence of modular squarings and multiplications [5]. Let $[n_0 \ldots n_k]$ 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.

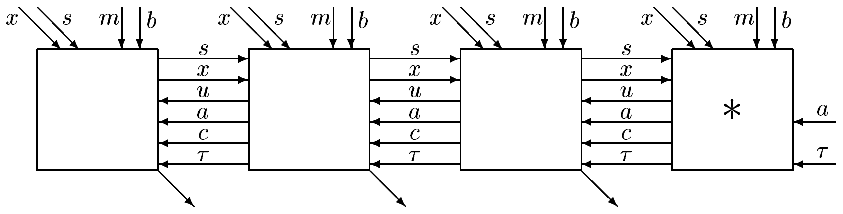

Fig. 1. Linear Systolic Array for Montgomery Exponentiation; M = 3.

A linear systolic array for high-to-low modular exponentiation is presented in Fig. 1. Details of its systematic design can be found in [6]. The systolic array comprises $M + 1$ processing elements (PE), each PE is able to operate in two modes, multiplication and squaring. 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 of M-multiplication, if $\tau = 1$, M-squaring. The order in which control signals are input is determined by the binary representation of n. Each vertex 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. A starred vertex, i.e., a vertex marked with "$*$", performs calculations of $u_i := ((s_0 + a_i * b_0) * w) \bmod 2$ besides an ordinary operation.

To perform M-squaring we need only the b_j's inputs. This eliminates the rightmost a-input entirely. To deliver all b_js to the starred vertex, we have to pump them through the graph using additional arcs x_js. Vertex operations are to be slightly modified to provide propagation of these digits: each non-starred vertex just transmits its x-input data to x-output, while when arriving at the rightmost vertex, these data are "reflected" and propagated to the left as if they were ordinary a_i's input data. The operations for M-squaring can be specified exactly as above (apart from x's transmissions) for main vertices; the starred vertex computes $u_i := ((s_0 + x_i * b_0) * w) \mod 2$, assigns $a_i := x_i$, and performs an operation as above (with $c_in = 0$).

A timing function that provides a correct order of operations is $t(v) = 2i + j$ [6]. The total running time is thus at most $(4\lfloor \log_2 n \rfloor + 1)M + 8\lfloor \log_2 n \rfloor$ time units.

2 Mapping a Systolic Array onto FPGAs

To map a systolic array onto FPGAs, one has to:

- find suitable logic designs for a FPGA realisation of individual PEs and compile these designs into blocks of FPGA cells;
- specify logic designs that combine these blocks into an array and compile it;
- optimise the design.

A straightforward realization of the first two steps leads to a successful routing for designs with 67 PEs. Our "trial and error" experience with optimisation of the mapping systolic arrays onto FPGAs has shown that the following ideas ensure 100% improvement.

– It is wise to base your manual optimisation on the XACTStep6000 system proposals because the XACTStep6000 system does it with the objective of successful routing; a complex set of criteria takes into account a number of parameters which are known only to the system that optimises the allocation from the point of view of all these criteria.

– Using a hierarchy of modules facilitates the design process. However, an automatic routing may spread gates that belong to different levels of the hierarchy rather far apart. The more levels of the hierarchy the less control one may exercise over the overall design, so try to minimise the number of levels.

– If the outputs of some gates are to be stored in registers, these gates and registers should not be at different hierarchical levels, because 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.

– When a systolic array cannot be mapped in a straightforward fashion onto a FPGA chip (e.g., a long and narrow line of PEs has to be mapped onto a square array of logic cells), it is advisable to partition this array into blocks of PEs with respect to the width of the board, so that every block can be allocated on a chip in a form of a border-to-border straight line. In our case, one block comprises 13 PEs. Pack these blocks in a zig-zag "snake" to fill in the whole 64×64 sea of cells on a chip.

– To ensure regularity and locality of interconnections between blocks, and between PEs in different blocks, create for each PE a few types of logic designs with identical functionality, but with input/output gates representing some suitable permutation of the gates in the original design. Use these designs to build "mirror images" of the original block under rotation and reflection. These images must be used at every second row of the zig-zag and where a "snake"' turns.

3 Summary

We presented a new implementation of modular exponentiation based on Montgomery multiplication 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. 3,000 out of 4,096 gates are used for computations and registers, providing 75% density.

Reported in this paper hardware implementation relies on configurability of FPGAs, but does not use run-time reprogrammability or/and SRAM memory. This makes our design simpler and easy to implement. The price to pay is that more chips are needed to implement RSA with longer keys. 512-bit keys need four XC6000 chips connected in a pipeline fashion, or 16 Kgates. Taking into account the total running time, 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 [5], and an order of magnitude better than that one in [1] and [3].

References

1. 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.
2. P. L. Montgomery, Modular multiplication without trial division. *Mathematics of Computations*, 1985 (44) 519–521.
3. H. Orup, E. Svendsen, E. And, VICTOR an efficient RSA hardware implementation. In: *Eurocrypt 90*, LNCS, vol. 473 (1991) 245–252
4. J. Sauerbrey, A Modular Exponentiation Unit Based on Systolic Arrays, in *Advances in Cryptology – AUSCRYPT'93*, Springer-Verlag, LNCS, vol. 718 (1993) 505–516.
5. M. Shand, J. Vuillemin, Fast Implementation of of RSA Cryptography. In *Proc. of the 11th IEEE Symposium on Computer Arithmetics*, 1993. pp.252–259.
6. A. A. Tiountchik, Systolic modular exponentiation via Montgomery algorithm. J. *Electronics Letters*, 1998 (34).
7. C. D. Walter, Systolic Modular Multiplication. *IEEE Trans. on Comput.*, 1993 (42) 376–378.

This article was processed using the LaTeX macro package with LLNCS style