Modular Exponent Realization on FPGAs

Jüri Põldre | Kalle Tammemäe | Marek Mandre

Tallinn Technical University
Computer Engineering Department
jp@pld.ttu.ee

Abstract. The article describes modular exponent calculations used widely in cryptographic key exchange protocols. The measures for hardware consumption and execution speed based on argument bit width and algorithm rank are created. The partitioning of calculations is analyzed with respect to interconnect signal numbers and added delay. The partitioned blocks are used for implementation approximations of two different multiplier architectures. Examples are provided for 3 families of FPGAs: XC4000, XC6200 and FLEX10k

1 Introduction

Modular exponent calculations are widely used in Secure Electronic Trading (SET) protocols for purchasing goods over Internet. One transaction in SET protocol requires the calculation of six full-length exponents. Because of advances in factoring and ever-increasing computing power the exponent size has to be at least 1024 bits now and predicted 2048 in 2005 to guarantee the security [1].

The calculations with very large integers are managed in software by breaking them down to the host processor word size. The exponent is usually calculated by progressive squaring method and takes $2 \times N$ modular multiplications to complete. One modular multiply takes at least $(A / W) \times (N \times N)$ instructions to complete, where N is argument bit length, W is host processor word size and $A>1$ is a constant depending on the algorithm used. As exponent calculation demands N multiplications that leaves us with N^3 complexity.

As of now the need for SET transactions is about one per second. It is estimated to reach over 200 per second in servers after 18 months. Today the common PC Pentium 200 MHz processors can calculate one exponent in 60 msec. Taking 6 exponents per SET transaction we have 100% load in 2¾ SET transactions per second.

As the calculations are very specific it is not likely that they will become a part of general-purpose processors, although internal study by Intel Corporation has been carried out to find possible instruction set expansion for cryptographic applications in 1997. Sadly only abstract of that is available for public review. A separate unit in system for cryptographic calculations also increases security by creating "secure area" for sensitive information.

Recently many companies have come up with the product to solve this problem. Usually these consist of RISC processor core, flash ROM, RAM and exponent accelerator unit. Several products on the market are ranging from Rainbow CryptoSwift with 54 msec to Ncipher Nfast 3 msec per exponent. They also have different physical interfaces – Cryptoswift uses PCI bus and Ncipher is a SCSI device. Cryptoswift uses ARM RISC processors and Ncipher ASICs. Several other designs have been created including authors IDEA/RSA processor [7]. Prices for these products range from $1600 – $3000 per device.

Standards are being developed and new algorithms proposed, so the design lifespan of these accelerators is quite low. A solution here is to use a cryptography system library with certain functions being accelerated in reconfigurable hardware – FPGA.

FPGA based accelerator board supplied with PCI-interface is universal device, which can be plugged into any contemporary PC for accelerating RSA-key encoding-decoding task. Considering the fact, that Sun has already included PCI-interface into Ultrasparc workstation configuration, the board suits there as well, reducing computation load of main processor(s).

In the course of this work we will look into accelerating exponent calculations using two different methods. Both of them use progressive squaring, Montgomery reduction, and redundantly represented partial product accumulation. The difference is in the architecture of modular multiplier unit.

In following pages we:

- Select the appropriate exponentiation (multiplication) algorithm.
- Define the hardware building blocks for the algorithm.
- Analyze two different architectural approaches using the blocks from previous stage.

In every step the alternative approaches and reasoning behind selection are presented.

2 Modular exponentiation

will be handled by right-left binary method. It gives the possibility to run two multiplications per iteration in parallel and thus half the execution time if sufficient hardware resources are available. It can also be utilized easily for interleaved calculations as two arguments (N, P) are same for both multiplications

To find $C := M^e \bmod N$ proceed as follows:

Input:	base M; exponent e; moduli N;
Output:	$C := M^e \bmod N$
Temporary variable:	P
Exponent size:	h
Ith bit of e:	e_i

Algorithm:

1. $C := 1; P := M$
2. **for** $i = 0$ **to** $h - 2$
2a. **if** $e_i = 1$ **then** $C := C \times P \pmod N$
2b. $P := P \times P \pmod N$
3. **if** $e_{h-1} = 1$ **then** $C := C \times P \pmod N$
4. **return** C

Further possibilities for reducing number of multiplications are not considered in this work. These methods involve precalculated tables and short exponents [6]. The support for precalculated tables can be added at higher level of hierarchy using host processor. For the course of the article let the time for exponentiation be equal to number of bits in the exponent:

$$T_{exp} = h \times T_{mult} \tag{1}$$

h: number of bits in exponent,
T_{exp}: time for exponent calculation,
T_{mult}: time for multiplication.

3 Modular multiplication

is the only operation in exponentiation loop. The modular multiplication is multiplication followed by dividing the result by moduli and returning quotient:

$$C = A \times B \bmod N \tag{2}$$

$T = A \times B$
$Q = T / N$
$C = T - Q \times N$

We can calculate the multiplication and then divide by moduli, but these operations can also be interleaved. This reduces the length of operands and thus hardware consumption. The algorithms rank k is the amount of bits handled at one step.

The main updating line in interleaved k-ary modular multiply algorithm is:

$$S_{i+1} = S_i << k + A \times B_i - Q_i \times N \tag{3}$$

Not going any further into details [2] let us point out that most time-consuming operation is to find Q_i, what is S_{i-1} / N. Several approaches have been proposed. All of them use approximation to find Q_i. Some of them involve multiplication, others table lookup. All of them consume silicon resources, but mainly they increase cycle time significantly.

In 1985 Montgomery [3] proposed new method for solving the problem of quotient digit calculation. After some preprocessing it is possible to calculate the loop updating line as:

$$S_{i+1} = S_i >> k + \tilde{A} \times B_i + Q_I \times \tilde{N} \quad (4)$$

Q_i is equal to k least significant bits of partial sum S_i. This comes at a price of transforming the initial arguments and moduli to Montgomery residue system and the result back. The transformations can be carried out using the same Montgomery modular multiplication operation (arguments conversion constant $2^{(2 \times h)} \bmod N$ is needed, but it can easily be calculated in software). Another restriction is that 2^k and N should be relatively prime. As application area is cryptography even N will never occur and thus this condition is satisfied.

The exponentiation process now takes two more multiplications to complete for transforming arguments. Because argument length h is usually large (more than 1024 bits) it is *ca* two tenths of percent. This introduced delay is negligible taken into account the cycle speedup and hardware savings.

4 Hardware blocks

To describe hardware architectures the main building blocks for them are needed. Montgomery multiplication (4) needs multiplication, summation and shift operators. Because the argument sizes are large and result is not needed in normal form before the end of the exponentiation it is reasonable to use redundant representation of arguments - 2 digits to represent one. The exact value of the digit is the sum of these two components.

Double amount of hardware is needed for handling redundant numbers, but it postpones carry ripple time until the result is needed in normal form. Even if we would construct a fast adder (CLA) it spends more than twice hardware and definitely has larger power consumption.

Shifting these digits is straightforward and involves shifting both numbers.

For addition normal full-adder cells can be used. Two such cells forms an adder for redundant numbers what is called 4-2 adder (Add4-2):

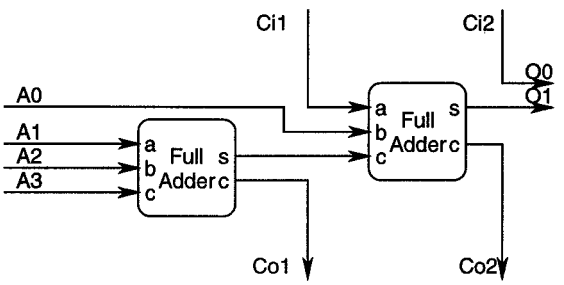

Fig. 1. 4-2 adder from two full adders

Connecting carries $ci \rightarrow co$ of h such blocks generates h-bit redundant adder. As it can be seen from the figure the maximal delay for such addition does not depend on argument length and equals to two full-adder cell delays.

Before making multiplier we will look at multiplicand recording. If both positive and negative numbers are allowed in multiplication partial product accumulation, then the number of terms can be halved. This process is called Booth encoding. Following table should illustrate the idea:

Table 1. Booth recording of 3-bit multiplicand

B	*Term1*	*Term2*
0	0	4×0
1	1	4×0
2	2	4×0
3	-1	4×1
4	0	4×1
5	1	4×1
6	2	4×1
7	-1	4×2

Calculating multiple of $A \times B$ in usual way requires 3 terms: A , $A << 1$, $A << 2$. By recording B differently we can do away with only two$_k$ The multiplication constants for terms are 0,1,2 and –1. Multiplications by 2,4,2^k can be handled with shift. Negation uses complementary code: $-A$ = (not A) +1. Carry inputs to partial sum accumulation tree structure are utilized for supplying additional carries.

One term of partial sum is thrown away by adding 4-input multiplexers (Mux4). The same method can be used to make $5 \rightarrow 3$, $7 \rightarrow 4$, ... etc. encodings. Generally we will have:

$$(N\text{-}1) \rightarrow N/2 \tag{5}$$

Booth recorder is required for generating multiplexer control information from multiplier bits. As this circuit is small and only one is needed for multiplier we will not look into that more deeply.

The multiplier consists of Booth encoder, shifter and redundant adder tree. It has redundantly represented base N, Booth recorded multiplicand B and redundant result O. The terms from Booth encoders will be accumulated using tree of 4-2 adders. Carry inputs and outputs are for expansion purposes and for negation control at LSB end of digit.

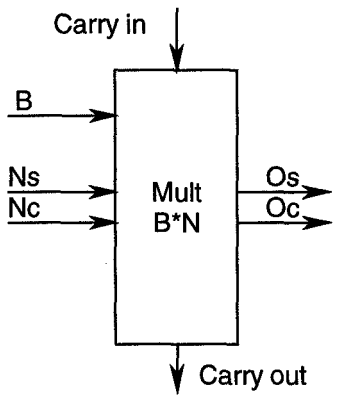

Fig. 2. *B*×*N* multiplier

The total component delay of this circuit is the sum of multiplexer delay and delay introduced by 4-2 adder tree. This delay is proportional to tree depth or log2 of size of B in digits. We can write delay as following:

$$T_{mult} = T_{mux4} + \log_2(\text{size}(B)) \times T_{add4\text{-}2} \quad (6)$$

T_{mult}	Time for multiplication.
T_{mux4}	Mux4 cell delay.
$T_{add4\text{-}2}$	Add4-2 cell delay.
Size(B)	size of Booth recorded number in digits.

The number of elements required for building such block is:

$$Count_{Mux4} = N \times 2 \times \text{size}(B) \quad (7)$$

$$Count_{Add4\text{-}2} = N \times 2 \times (\log_2(\text{size}(B)) - 1) + 1 \quad (8)$$

Here is an example to clarify the formulas:

Device is 4 booth digits × 8 bits or 7 × 8 bit multiplier. The result is calculated in $T_{mux4} + 2 \times T_{add4\text{-}2}$ time units. The number of 4 - input multiplexers is $8 \times 2 \times 4$ and the count of 4 - 2 adders is 16 + 1. The multiplier structure for one output bit is described in figure below.

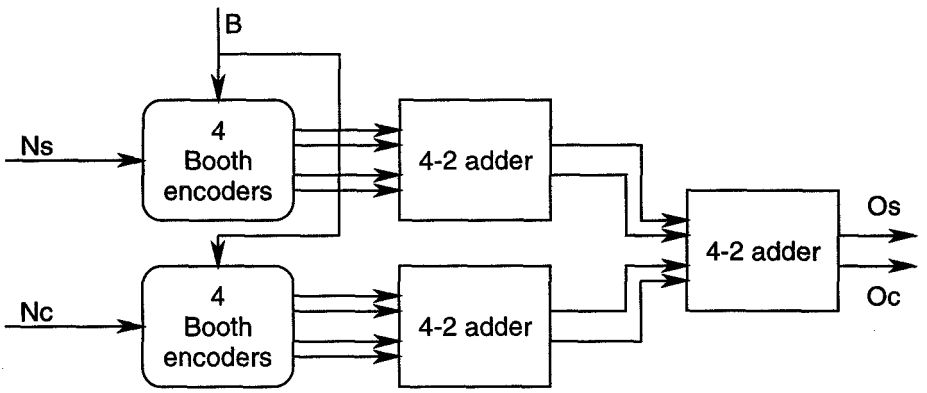

Fig. 3. 7×8 multiplier structure

Each 4-2 adder generates two carries. Booth encoder needs higher bits of previous operand to generate terms, adding two times the size of B carries for both input operands. The total is thus $3 \times 2 + 4 \times 2 + 2 = 16$ carries in and 16 carries out. The formula for counting carry signal numbers is:

$$(\log_2(\text{size(b)}) + \text{size(b)} + 1) \times 2 \quad (9)$$

5 FPGA resources

For FPGA realizations the resource allocation formulas for these operators are needed. We will consider 3 series: Xilinx 4000, Xilinx 6200 and Altera FLEX10K. As all previously described blocks contain simple routing what is connected only to closest neighbors we will ignore the routing expenses in calculations and concentrate in CLB count.

The above described two operators demand the following hardware resources:

- MUX4 (may be built from two 2MUX cells).
- ADD4-2 (consists of two full-adder cells).

The following table will sum the resources needed to build these blocks in each of mentioned families:

Table 2. Cell hardware requirements

Cell name	XC4000	XC6200	FLEX10K
MUX4	1	2	2
ADD4-2	1*	6	2

* Actually it is 2 ADD4-2 cells per 2 blocks, because one block implements 2-bit full adder.

6 Architectural solutions

Having the blocks let us now consider different architectures for implementation.

6.1 Traditional k-ary algorithm

Calculates the Montgomery multiplication by directly implementing the loop updating statement:

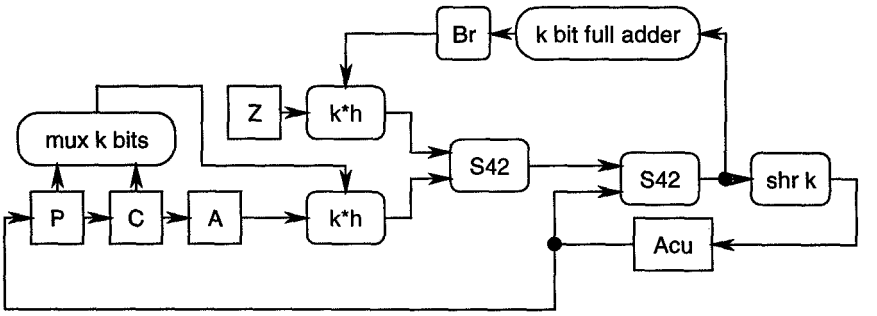

Fig. 4. Traditional architecture for calculating Montgomery multiplication

Z, *A*, *ACU* are *h* bit registers. *k*×*h* multiplier forming $B_I \times A$ is the multiplier cell described above. $Q_i \times Z$ is the same cell with only half of hardware, because *Z* is in normal (non-redundant) representation. Two 4-2 adders (S42) accumulate the results. By adding registers in accumulator structure it is possible to interleave two multiplications as described earlier. Control unit must multiplex B_i each clock tick from *P* or *C*. At the end of calculations *C* and *P* are updated. The updating of *P* is done conditionally depending on value of e_i, the ith bit of exponent.

6.2 K-ary systolic architecture

This approach uses array of identical blocks. Each block calculates digit(s) of result. By connecting these blocks it is possible to generate a hardware structure of arbitrary bitlength. Remarkable features of this approach are expandability and relative ease of construction. Once you have mastered the primitive block it is easy to place them on CLB array or silicon. The structure of systolic array multiplier consisting of $h / (2 \times k)$ cells implementing (4) is:

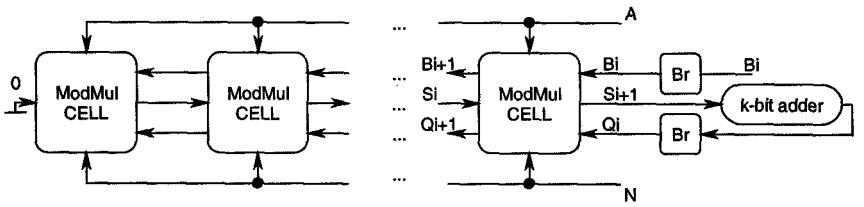

Fig. 5. Systolic multiplier structure

In each cell two terms of partial sum are calculated and summed [5]. Sum term is represented redundantly, but *k*-bit full adder converts it back to normal form. Therefore cell contains four $k \times k$ non-redundant input multipliers and accumulator tree of 4-2 adders (Fig 6). *B*, *Q* and *S* are *k* bit registers. Cell also contains carry memory what adds twice the number of S42 block count registers to cell memory requirements.

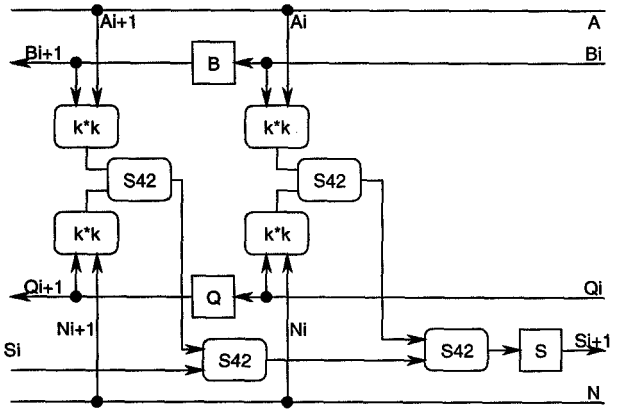

Fig. 6.. One systolic array cell

7 Analysis of implementations

Both the systolic and classical solution calculate the statement (2) with the same delay. As systolic solution is accumulating 2 terms it ads one S42 delay. Thus the formulas for calculating cycle length are (calculated in 4-2 adder delays):

$$T_{classic} = \log_2(k/2) + 1 \quad \textbf{(10)}$$

$$T_{systol} = \log_2(k/2) + 2 \quad \textbf{(11)}$$

We can further decrease time by adding registers at S42 outputs and using quotient pipeline [4]. This reduces cycle delay to one S42 cell delay. It can be reduced further, but registers in ASIC are expensive. This is not the case with FPGAs because the ratio of register/logic is high and flip-flops are already there.

Systolic array is made of $h / (2 \times k)$ cells and each cell consists of four $k \times k$ multipliers. Comparing that to standard approach with $1½\ k \times h$ multipliers:

$$4 \times (k \times k \times ½) \times H / (k \times 2) = 2 \times ½ \times k \times h = k \times h \quad \textbf{(12)}$$

In systolic array we have the result of multiplication in normal format, therefore we need 1/3 less hardware. The following table sums the hardware consumption for both architectures for 3 different algorithm ranks (k)

Table. 3. Hardware (CLB count) requirements for exponent calculator

bits	k	XC4000 systol	XC4000 classic	XC6200 systol	XC6200 classic	FLEX10K systol	FLEX10K classic	cycles systol	cycles classic
512	2	4096	6144	16384	24576	8192	12288	256	128
512	4	7168	10752	26624	39936	14336	21504	128	64
512	8	12288	18432	40960	61440	24576	36864	64	32
1024	2	8192	12288	32768	49152	16384	24576	512	256
1024	4	14336	21504	53248	79872	28672	43008	256	128
1024	8	24576	36864	81920	122880	49152	73728	128	64

The systolic structure calculates result in $2 \times N / k$ steps. For exponent calculations it is possible to either use twice the hardware or run two multiplications sequentially. In the table above hardware consumption for single multiplication is provided.

Cycle speed is increased by having to partition the design on several FPGAs, for large exponents do not fit into single FPGA. This additional delay consists of CLB→IOB→PCB→IOB→CLB path. Each component adds it's own delay. We will use the 20 ns safe figure here for this entire path. Thus the cycle times for chosen families are:

$$T_{4000} = 20 + (1 + \log_2(k)) \times 5 \tag{13}$$

$$T_{6200} = 20 + (2 + 3 \times \log_2(k)) \times 4$$

$$T_{FLEX10K} = 20 + (2 + 2 \times \log_2(k)) \times 5$$

First term is communication delay, then 4mux delay for Booth encoder and finally logarithmic component for accumulator. The numbers behind parenthesis is CLB delay added to closest neighbor routing of fastest member in the family. These are optimistic values, but as structure is regular and routing is between the closest neighbors the expected results should not differ from calculated more than 10%. The values in the following table are exponent calculation times in msec.

Table 4. Exponent calculation timing

bits	k	XC4000 systol	XC4000 classic	XC6200 systol	XC6200 classic	FLEX10K systol	FLEX10K classic	cycles systol	cycles classic
512	2	3,9	2,0	5,2	2,6	5,2	2,6	256	128
512	4	2,3	1,1	3,4	1,7	3,3	1,6	128	64
512	8	1,3	0,7	2,1	1,0	2,0	1,0	64	32
1024	2	15,7	7,9	21,0	10,5	21,0	10,5	512	256
1024	4	9,2	4,6	13,6	6,8	13,1	6,6	256	128
1024	8	5,2	2,6	8,4	4,2	7,9	3,9	128	64

For partitioning the largest circuits from each family were used. These are at the current moment:

- XC6264 (16384 CLBs).
- XC4025 (1024 CLBs).
- EPF10K100 (4992 CLBs).

Utilizing them the following number of chips is needed for implementation (table 5). To compare the speed-up of calculations the data from RSA Inc. Bsafe cryptographic library is in table 6.

Table 5. Number of ICs for implementation

bits	k	XC4000 systol	XC4000 classic	XC6200 systol	XC6200 classic	FLEX10K systol	FLEX10K classic
512	2	4,0	6,0	1,0	1,5	1,6	2,5
512	4	7,0	10,5	1,6	2,4	2,9	4,3
512	8	12,0	18,0	2,5	3,8	4,9	7,4
1024	2	8,0	12,0	2,0	3,0	3,3	4,9
1024	4	14,0	21,0	3,3	4,9	5,7	8,6
1024	8	24,0	36,0	5,0	7,5	9,8	14,8

Table 6. Bsafe cryptolibrary execution benchmarks in seconds

Operand length in bits	Intel Pentium 90 MHz	Power Macintosh 80 MHz	Sun SparcStation 4 110 MHz	Digital AlphaStation 255 MHz
768	0.066	.220	.212	0.024
1024	.140	.534	.461	0.043

8 Conclusions

In this paper we have analyzed the implementation of modular exponent calculator on FPGAs. The appropriate algorithm for exponentiation and multiplication has been selected. Realizations on three families of FPGAs were considered.

While two XC6216 circuits would nicely fit onto PCI board and give over 10 times acceleration of calculations we must bear in mind that these circuits are quite expensive.

Maybe simpler approach would help? If we use 1-bit-at-a-time algorithm we can fit 1024 bit calculator into one package. $k = 1$ classic structure demands two 4-2 adders per bit and requires H steps to complete. 1024 bit exponent is calculated with 1024x1024x2 cycles. As the structure is simpler the cycle delay can be decreased on condition that we stay in limits of one package. That leaves us with 512 bit for XC4K, 1024 bit for XC6264 and 2500 for FLEX10K. The clock frequency can now be lifted up to one CLB delay plus routing between closest neighbors. This can be as high 100 MHz calculating one exponent in 20 msec. This is comparable with Digital 255 MHz processor. This is approximately 50 Kgates of accelerator hardware running at twice slower speed.

As to now programmable hardware is still too expensive to be included on motherboards but these figure shows a clear tendency that the devices together with hardware-software co-development system and downloadable modules will become a part of functioning computer system in nearest future.

References

[1] *Schneier, Bruce. "Applied Cryptography Second Edition: protocols, algorithms and source code in C", 1996, John Wiley and Sons, Inc.*

[2] *Ç. K. Koç, "RSA Hardware implementation", RSA laboratories, 1995.*

[3] *Peter L. Montgomery. "Modular multiplication without trial division", Mathematics of Computation, 44(170):519-521. April 1985.*

[4] *Holger Orup. "Simplifying Quotient Determination in High-Radix Modular Multiplication", Aarhus University, Denmark. 1995.*

[5] *Colin D. Walter. "Systolic Modular Multiplication" IEEE transactions on Computers, C-42(3)376-378, March 1993.*

[6] *B.J.Phillips, N.Burgess. "Algorithms of Exponentiation of Long Integers – A survey of Published Algorithms", The University of Adelaide, May 1996.*

[7] *Jüri Põldre, Ahto Buldas: "A VLSI implementation of RSA and IDEA encryption engine", Proceedings of NORCHIP'97 conference. November 1997.*