Fast Modular Inversion in the Montgomery Domain on Reconfigurable Logic

Alan Daly[†], **Liam Marnane**[††] **and Emanuel Popovici**[*]

[†]*Department of Electrical and Electronic Engineering, University College Cork, IRELAND*

[*]*Department of Microelectronic Engineering, University College Cork, IRELAND*

E-mail: [†]aland@rennes.ucc.ie [††]marnane@ucc.ie [*]e.popovici@ucc.ie

Abstract — **Modular multiplication and inversion are frequently used operations in modern cryptographic applications. Multiplication bitlengths of 1024 to 2048 are common in RSA implementations, while inversion bitlengths of 128 to 256 are used in ECC *(Elliptic Curve Cryptography)* systems. In both operations, an addition carry chain forms part of the critical path. In order to improve performance, we investigate methods of breaking the carry chain by arithmetic manipulation of the addition/subtraction operations. The target device used for this research is an FPGA *(Field Programmable Gate Array)* where the length of the carry chain is an important design consideration due to the underlying architecture imposed by the reconfigurable technology. However, carry propagation delays also affect performance in ASIC *(Application Specific Integrated Circuits)* designs, and the methods proposed in this work may be implemented to improve efficiency.**

I Introduction

MODULAR Arithmetic is the basis of many important operations in cryptographic applications. The RSA algorithm[1] relies on modular exponentiation to provide secure public key encryption and decryption. This modular exponentiation is achieved through a combination of modular multiplications and squarings of the inputs.

By definition, modular multiplication reduces the product of two numbers through division by the modulus, resulting in a bounded remainder. Classically, this made it a difficult operation to perform. However, in 1985, P. L. Montgomery[2] proposed a method to perform modular multiplication without trial division, through a series of shifts and additions which are easier to perform in hardware.

Modular inversion is used in applications such as the generation of public/private key pairs in the RSA system, the Diffie-Hellman key exchange algorithm [3] and more recently in elliptic curve cryptography (ECC). Modular inversion can also be used to accelerate modular exponentiation in conjunction with addition-subtraction chains, where canonical recoding is used to reduce the average number of non-zero multiplications[4][5].

In Elliptic Curve Cryptosystems [6][7] over prime fields $GF(p)$, an elliptic curve is defined by the set of solutions or points $\mathrm{P} = (x, y)$ to the equation $y^2 = x^3 + ax + b$ for $x, y \in GF(p)$. Encryption is performed by point addition and point doubling. The number of additions/doublings performed is a function of the private key, and each operation requires a series of modular integer additions, subtractions, multiplications and inversions. Therefore, fast modular multiplication and inversion are required to provide efficient ECC Encryption.

For FPGA implementation, the carry chain length becomes an important design consideration. Methods of improving performance by shortening the critical carry-chain length in Montgomery multiplication and exponentiation have previously been presented in [8].

The methods of breaking the carry chain for Montgomery multiplication are not applicable to inversion, since each iteration of the inversion algorithm is dependent on a sign/magnitude comparison of the previous iteration. Therefore, the entire intermediate result must be calculated each iteration before operations can commence on the next. The aim of this paper is to present another method of reducing the maximum carry chain length and hence shorten the critical delay path.

II Montgomery Multiplication and the Montgomery Domain

Montgomery multiplication is an efficient technique for multiplying two integers modulo M, proposed by P.L. Montgomery in 1985[2]. The *MonPro* algorithm computes the Montgomery product of the two integers through an iterative process of additions and right-shifts. The algorithm eliminates the need for division by M, which is accepted to be a more difficult operation.

The *MonPro* operation performs the following computation:

$$C = \text{MonPro}\ (\ a,\ b,\ M\ ) := a\ b\ 2^{-n}\ (\ \text{mod}\ M\ )$$

where M is at most an m-bit integer (ie: $0 < M < 2^m$) and the bit length of the multiplier, n, is defined to be equal to $m+2$. It is necessary to keep the bit length of the multiplier equal to $m+2$ to ensure that the result C remains bounded (ie: $0 \leq C < 2^m$).

The extra factor of 2^{-n} must be removed to produce the correct result $c = a\ b\ (\ \text{mod}\ M\ )$. This can be achieved by performing another Montgomery multiplication on the result and $2^{2n}\ (\ \text{mod}\ M\ )$. This must be performed if the output is to be used as an input for further multiplications, such as in modular exponentiation.

$$\begin{aligned} c = \text{MonPro}(\ C,\ 2^{2n},\ M\ ) &= a\ b\ 2^{-n}\ 2^{2n}\ 2^{-n} \\ &= a\ b\ (\ \text{mod}\ M\ ) \end{aligned}$$

Note that the value of $2^{2n}(\ \text{mod}\ M\ )$ must be precomputed externally.

For a single multiplication, this does not appear to be very efficient since two MonPro operations are required per multiplication.

If however, the numbers to be multiplied are converted to the so-called Montgomery domain or m-residue, the output from one multiplication may be used as an input to the next. The m-residue $\mathcal{A}$ of an integer $a < M$ is defined as:

$$\mathcal{A} := a\ 2^n\ (\ \text{mod}\ M\ )$$

This mapping may be performed by using the MonPro operation:

$$\begin{aligned} \mathcal{A} &:= \text{MonPro}\ (\ a\ ,\ 2^{2n}(\text{mod}\ M),\ M\ ) \\ &= a\ 2^n\ (\ \text{mod}\ M\ ) \end{aligned}$$

When two numbers in the Montgomery domain are multiplied together, the result is a number which itself is in the domain:

$$\mathcal{C} := \text{MonPro}\ (\ \mathcal{A},\ \mathcal{B},\ M\ ) = c\ 2^n\ (\ \text{mod}\ M\ )$$

To convert the result back to the real domain, it must be Montgomery multiplied by '1':

$$\text{MonPro}\ (\ \mathcal{C},\ 1,\ M\ ) = c\ 2^n\ 1\ 2^{-n} = c\ (\ \text{mod}\ M\ )$$

Bitlength (m)	Area (Slices)	Clk Period (ns)	Throughput (Mbit/s)
128-bit	646	12.31	78.2
256-bit	1,292	17.17	57.1
512-bit	2,588	17.84	55.5

Table 1: Sample Area and speed results for the Montgomery Multiplier.

So for applications such as exponentiation, three stages are required: mapping to the Montgomery domain, exponentiation in the Montgomery domain, and re-mapping back to the real integer domain. For large exponents in the range of 1024-bits or greater, the additional two multiplications are not a significant penalty.

Some sample speed/area figures for a Montgomery Multiplier based on the designs in [8] and implemented on a Xilinx Virtex XCV2000e FPGA are given in Table 1. Each Multiplication requires $(m + 5)$ clock cycles using this architecture.

III Montgomery Modular Inversion

The Modular inverse of an integer $a \in [1, M-1]$ modulo a prime M is defined as the integer x, such that:

$$a * x = 1\ (\ \text{mod}\ M\ )$$

which can be written:

$$x := \text{ModInv}\ (\ a\ ) = a^{-1}\ (\ \text{mod}\ M\ )$$

Note that the multiplicative inverse of an integer $a\ (\text{mod}\ M)$ exists if, and only if, a and M are relatively prime.

Kaliski [9] describes an algorithm which computes the Montgomery modular inverse of an integer a, based on the extended binary GCD algorithm. The Montgomery modular inverse of a is given by:

$$\mathcal{X} := \text{MonInv}\ (\ a\ ) = a^{-1}\ 2^m\ (\ \text{mod}\ M\ )$$

where m is the bit-length of the modulus M.

The algorithm to compute the Montgomery inverse of an integer a, as introduced in [9] is given below. It is broken into two phases. Phase I is the Almost Montgomery Inverse. It's output is the integer r such that:

$$r := \text{AlmMonInv}\ (\ a,\ M\ ) = a^{-1}\ 2^k\ (\ \text{mod}\ M\ )$$

where $m \leq k \leq 2m$.

Therefore, the AlmMonInv algorithm will require between $(m + 1)$ and $(2m + 2)$ clock cycles to complete the operation. The extra cycles come from the adjustment phase after the while loop.

Algorithm 1 : Almost Montgomery Inverse

AlmMonInv (a, M)
Input : $a \in [1, M-1]$ and M
Output : r and k where $r = a^{-1}2^k \pmod M$
and $m \leq k \leq 2m$

$u := M$, $v := a$, $r := 0$, $s := 1$
$k := 0$
while $(v > 0)$
if u is even then $u := u/2$, $s := 2s$
else if v is even then $v := v/2$, $r := 2r$
else if $u > v$ then $u := (u - v)/2$, $r := r + s$,
$s := 2s$
else if $v \geq u$ then $v := (v - u)/2$, $s := s + r$,
$r := 2r$
$k := k + 1$
if $r \geq M$ then $r := r - M$
return $r := M - r$
return k

The result r of Phase I is then corrected using Phase II to obtain the Montgomery inverse.

Phase II requires $(k - m)$ clock cycles (ie: between 0 and m, depending on the value of k). It is clear that the number of clock cycles required to perform the Montgomery inversion is not constant, however the upper and lower limits are well defined.

Algorithm 2 : Montgomery Inverse (Phase II)

MonInv (r, M, k)
Input : r, M and k from AlmMonInv
Output : x, where $x = a^{-1}2^m \pmod M$

for $i = 1$ to $(k - m)$ do
if r is even then $r := r/2$
else $r := (r + M)/2$
return $x := r$

The output of Phase II may be Montgomery multiplied by 1 to produce the real inverse a^{-1}. Alternatively, it is observed that phase II of the algorithm may be altered so as to output the real inverse of a, by replacing $(k - m)$ in the for loop by k as given in algorithm 3.

This variation of Phase II requires k clock cycles (between m and $2m$). Again this value relies on the choice of a and M and cannot be estimated in advance.

If the input to the Almost Montgomery Inverse algorithm is already in the Montgomery domain, ie: $a\ 2^m$, then the output of phase I is given by:

$$\text{AlmMonInv } (a2^m, M) = a^{-1}\ 2^{-m}\ 2^k = a^{-1}\ 2^{(k-m)}$$

Algorithm 3 : Real Modular Inverse (Phase II)

ModInv (r, M, k)
Input : r, M and k from AlmMonInv
Output : x, where $x = a^{-1} \pmod M$

for $i = 1$ to k do
if r is even then $r := r/2$
else $r := (r + M)/2$
return $x := r$

By using this as input to phase II, the output is in the integer domain:

$$\text{MonInv } (\ a^{-1}\ 2^{(k-m)},\ M,\ k\ ) = a^{-1}$$

To convert this result back to the Montgomery domain, the Montgomery product of a^{-1} and 2^{2m} can be computed.

Table 2 gives the number of clock cycles required to perform each of the inversions, to and from each domain.

The number of clock cycles per inversion does not simply depend on the bitlength of the inverter, but instead varies depending on the values of a and M. In cryptographic applications this could pose a security weakness in the system. In order to avoid this situation, the number of clock cycles per inversion should be standardised to the maximum for the given operation and domain. This requires including superfluous clock cycles which do not alter any valid data, between the time of completing the inversion and reaching the maximum number of clock cycles possible.

A Montgomery product based phase II was proposed in [10] for a software based system which requires 2 or 3 Montgomery product operations after the inverse function.

Kobayashi and Morita proposed a method for improving the speed of Montgomery inversions in [11] by modifying the extended binary GCD algorithm to perform matrix multiplications instead of simple multiplications by two. However, their method was also intended for software implementation and will not be of benefit for hardware implementation.

Domain	Max number of Clock Cycles			
from → to	Ph I	Ph II	MonPro	Total
Int→Mont	$2m+2$	m	0	$3m+2$
Int→Int	$2m+2$	$2m$	0	$4m+2$
Mont→Int	$2m+2$	m	0	$3m+2$
Mont→Mont	$2m+2$	m	$m+2$	$4m+4$

Table 2: Clock cycles required to perform m-bit inversions in Montgomery/integer domains.

IV Underlying FPGA Architecture

Unlike ASIC design, there is an uderlying architecture imposed upon FPGA circuit design. Thus an optimum design for FPGA will exploit the features available.

Typically, FPL *(Field Programmable Logic)* devices today are comprised of configurable units which consist of 4-input lookup tables, simple D-type Latches, and control logic. Each Xilinx Virtex CLB *(Configurable Logic Block)* contains 2 *slices* and each slice contains 2 of each of the basic elements[12].

Some devices contain high speed interconnect lines between vertically adjacent logic blocks which are designed to provide efficient carry propagation. This dedicated logic provides carry capability for high speed arithmetic functions. In order to exploit the fast carry chain, the logic blocks in which the carry signals are generated and propagated must be placed in a single column on the FPGA. This dedicated carry logic improves both the speed and area of arithmetic operations.

However, once the maximum carry chain length of the target device is reached, the carry must be routed from the top of one column to the bottom of the next. Therefore any design with a carry chain exceeding the column height of the target device will suffer a performance penalty.

The architectures presented in this paper are optimal for implementation on any FPL device which has dedicated carry logic capability. Speed and area results may differ between technologies due to routing and different physical layout, however by exploiting the maximum carry chain length, and avoiding overflow of the carry chain, the optimum design can be tailored to a given target device.

V Basic Inverter Architecture

By exploiting the dedicated carry logic provided in the FPGA and pipelining the additions, fast Montgomery multipliers can be implemented based on designs proposed in [8], provided that the pipeline bitlength is shorter than the maximum carry chain length (ie: 1 column of the FPGA). This design reduces the performance loss caused by the carry being routed from the top of one column to the bottom of the next.

However, due to the nature of the modular inversion algorithm, it is not possible to pipeline the calculation. Each iteration is dependent on the MSB of the previous iteration, meaning that each step must be fully completed before commencing the next. In modular multiplication, the next iteration is dependent on the LSB of the previous iteration. Arithmetic manipulation is proposed as an alternative to pipelining as a method of reducing the critical carry chain length.

The basic inverter design similar to that proposed in [13] consists of 2 m-bit subtractors to perform the $(u - v)$ and $(v - u)$ operations, an m-bit adder to perform the $(r + s)$ operation, and multiplexors to control inputs and outputs as shown in Fig.1. These components are re-used for phase II of the algorithm where the subtractor inputs are changed so as to perform $(M - r)$ and $(r - M)$, and the adder inputs change to perform $(r + M)$.

If the bitlength of the inverter required is larger than the maximum carry chain of the target device, this basic design suffers a degradation in performance. The interconnect between columns of carry chains contributes largely to the overall delay, and a method of avoiding this carry overflow is desirable to improve speed.

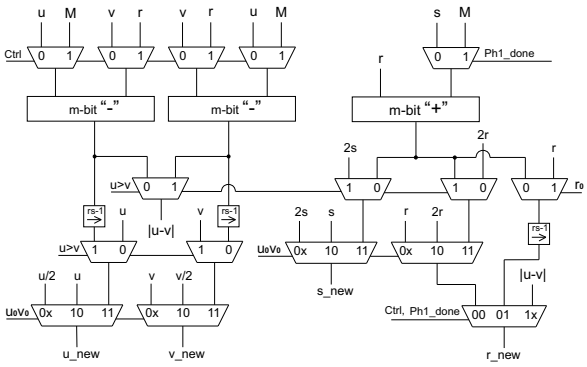

Fig. 1: Basic Inverter Design

VI Proposed Inverter Architecture

A new architecture is proposed which greatly increases the speed of an m-bit inverter by using 4 $((\frac{n}{2}) + 1)$-bit subtractors and 3 $(\frac{m}{2})$-bit adders. This new architecture halves the length of the critical carry chain, requiring just one extra level of multiplexors and one level of bit-wise inverters .

The value of $|u - v|$ is required in the algorithm. In the basic design both $(u - v)$ and $(v - u)$ were calculated in m-bit subtractors. If instead, u and v are broken up into $(\frac{m}{2})$-bit words, u_H, u_L, v_H and v_L, the upper and lower differences, $(u_H - v_H)$ and $(u_L - v_L)$ may be determined concurrently.

A leading '1' is appended to u_L and a leading '0' to v_L. The subtraction $(u_L - v_L)$ is performed, and the result is an $((\frac{m}{2}) + 1)$-bit word. This is illustrated in Fig.2. The leading bit of the result is an indication that $(u_L \geq v_L)$. If this bit is '0', then $(v_L > u_L)$ and hence one bit needs to be borrowed from u_H to determine $(u - v)$. Therefore, the subtraction $(u_H - v_H - '1')$ must be performed. However using two's complement notation, $(u_H - v_H - '1')$ is identical to $(\overline{v_H - u_H})$. Since $(v_H - u_H)$ has also been determined, no extra subtractions are required, just a bitwise inverter.

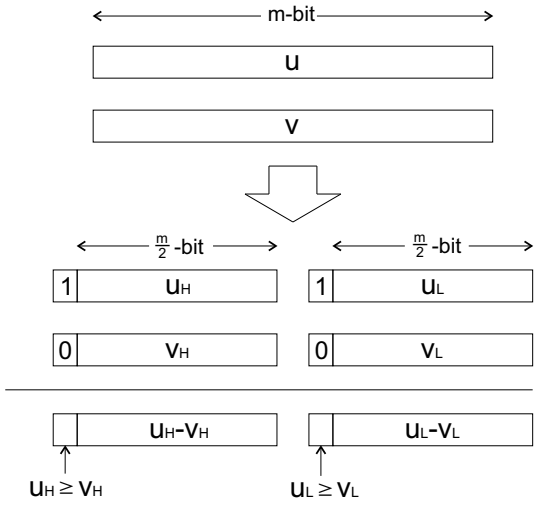

Fig. 2: Split Subtraction

Similarly, the $(v - u)$ subtraction is broken up into $(v_H - u_H)$ and $(v_L - u_L)$, and the values of $(v_H \geq u_H)$ and $(v_L \geq u_L)$ are determined. The value of $(u > v)$, which is needed to determine $|u - v|$ is given by:

$$(u > v) = ((u_H \geq v_H) \text{ and } (u_L > v_L)) \text{ or } (u_H > v_H)$$

The proposed architecture to evaluate u_{new} and v_{new} is presented in Fig.3. Comparing this with the basic design in Fig.1, it is clear that only 2 extra $(\frac{m}{2})$-bit multiplexors and 2 $(\frac{m}{2})$-bit inverters, are required.

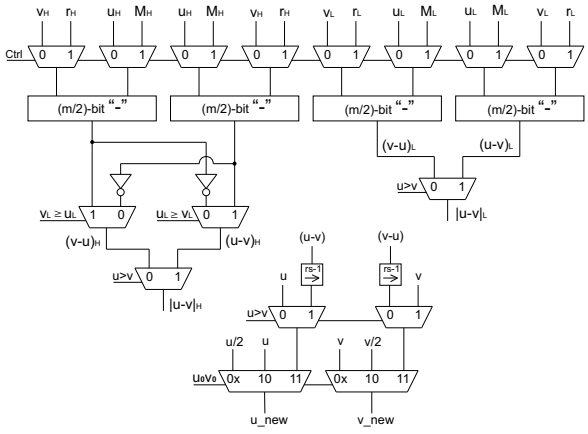

Fig. 3: Proposed Inverter Design (U&V)

For the evaluation of r_{new} and s_{new}, a carry-select type adder is utilised. As illustrated in Fig.4, 3 $(\frac{m}{2})$-bit adders are required. Similar to u and v, r and s are broken up into $(\frac{m}{2})$-bit words r_H, r_L, s_H and s_L. The $(r + s)$ addition is broken up into $(r_L + s_L)$, $(r_H + s_H)_0$ and $(r_H + s_H)_1$. $(r_H + s_H)_0$ is the sum of $(r_H + s_H)$ with a carry-in of '0', and $(r_H + s_H)_1$ is the sum of $(r_H + s_H)$ with a carry-in of '1'. The carry-out of $(r_L + s_L)$ determines which

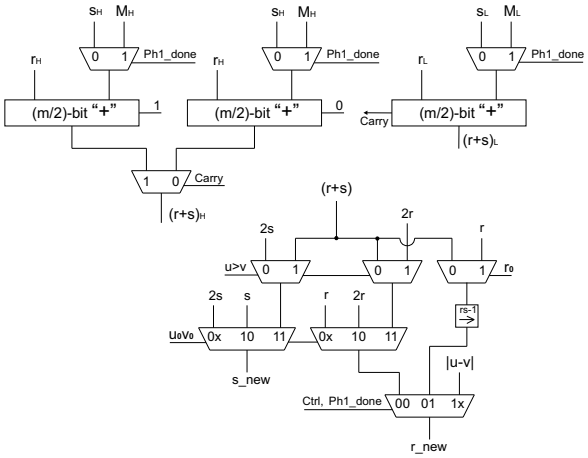

Fig. 4: Proposed Inverter Design (R&S)

of the two is used as the result. Note that when *Ph1_ done* = '1', the input s is replaced by M.

VII Speed/Area Results

Speed and area comparisons were performed on 4 different bitlength inverters, with both the basic architecture and the new proposed architecture. The results are listed in Table 3. The clock speed figures are post place and route. The VHDL Synthesis was performed using Leonardo Spectrum, and the Place and Route using Xilinx Design Manager.

The target FPGA device used for this research is the Xilinx Virtex XCV2000e-6bg560 and has 80 CLB's per column, which means the maximum unbroken carry chain length is 160 bits. Using the design presented here, an inverter up to 318 bits can be implemented without the penalty of routing a carry bit across columns. This is ideal for ECC applications where the bit length of the numbers is in the region of 256 bits. Without using the proposed inverter design, the carry chain would exceed the column height and incur a performance penalty. However, using the proposed design will improve speed significantly.

The (32-bit x 2) design gives little performance improvement over the 64-bit design. In this case the delay caused by the carry chain is not as significant as the control and multiplexor delays.

The results of the (64-bit x 2) design shows the direct performance improvement obtained with an unbroken carry-chain compared to the 128-bit design.

However, a greater improvement is seen when comparing the (128-bit x 2) design to the 256-bit design, due to the fact that the carry must overflow and be routed from the top of one CLB column to the bottom of the next. This results in considerable propagation delay in the basic 256-bit architecture design. Because the new design

Design	Area (Slices)	Equivalent Gates	Max Freq. (MHz)
64-bit	515	10,221	59.93
32-bit x 2	549	10,631	61.97
128-bit	1,019	18,613	39.03
64-bit x 2	1,023	19,757	46.80
256-bit	2,020	37,097	21.38
128-bit x 2	2,022	39,369	34.73
512-bit	3,331	60,243	13.12
256-bit x 2	3,481	65,686	18.15

Table 3: Area and speed results for the two designs.

Inverter Bitlength	Increase in Area (%)	Increase in Speed (%)
64-bit	6.6 %	3.4 %
128-bit	0.4 %	19.9 %
256-bit	0.1 %	62.4 %
512-bit	4.5 %	38.3 %

Table 4: Percentage increase in area/speed of new design over basic design.

is broken down to two 128-bit carry-chains, this overflow does not occur, and so no performance penalty is suffered.

The two 512-bit designs illustrate how the performance advantage is reduced when the carry-chain is broken in the new design. The new design is still significantly faster, but the margin is not as great.

The percentage increase in speed and area of the new design over the basic design for each bitlength is presented in Table 4. From this table it can be seen that the greatest improvement in performance is obtained from this design at inverter bitlengths greater than the maximum carry chain length, but less than twice it.

The results presented are for the critical path of phase I of the algorithm, excluding the control multiplexors for phase II. The proposed design improves the speed of the design significantly (up to 60% faster) at a very small increase in area ($< 1\%$).

VIII Conclusions

The critical path of the modular multiplication operation includes an n-bit carry chain. When this carry chain length exceeds the column height of the target FPGA, a large performance penalty is incurred due to routing delays between columns. A way to minimise this penalty is to pipeline the addition into blocks less than the column height of the FPGA.

Modular inversion is inherently slower than modular multiplication, and cannot be pipelined to improve the performance, due to the nature of the algorithm. The arithmetic manipulation method proposed in this paper is an alternative to pipelining for bitlengths less than twice the maximum column height. For ECC applications, m is in the order of 256 bits and this method proves to be well suited in this case. The critical path is shortened, providing a considerable improvement in performance at very little area cost. The speed of the carry chain propagation is doubled, but the registers, multiplexors and control contribute to the critical path, and so an overall speed improvement of up to 60% over the standard design was achieved.

Further performance improvements are possible by splitting the carry chain into shorter bit-lengths at the expense of more chip area and control logic.

IX Acknowledgement

This work is funded by a research innovation project from Enterprise Ireland.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for obtaining digital signatures and public-key cryptosystems". *Comm. ACM*, 21:120–126, 1978.

[2] P. L. Montgomery. "Modular Multiplication without Trial Division". *Math Computation*, 44:519–521, 1985.

[3] W. Diffie and M. E. Hellman. "New directions in cryptography". *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[4] C. K. Koc. "Parallel Canonical Recoding". *Electronics Letters*, 32(22):2063–2065, 1996.

[5] O. Egecioglu and C.K.Koc. "Exponentiation using Canonical Recoding". *Theoretical Computer Science*, 129(2):407–417, 1994.

[6] V. S. Miller. "Use of Elliptic Curves in Cryptography". *Advances in Cryptography Crypto'85*, (218):417–426, 1985.

[7] N. Koblitz. "Elliptic Curve Cryptosystems". *Math Comp*, 48:203–209, 1987.

[8] A. Daly and W. Marnane. "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". *International Symposium on FPGAs (FPGA2002)*, pages 40–49, Feb 2002.

[9] Burton S. Kaliski Jr. "The Montgomery Inverse and it's applications". *IEEE Trans. on Computers*, 44(8):1064–1065, Aug 1995.

[10] E. Savas and C. K. Koc. "The Montgomery Modular Inverse - Revisited". *IEEE Trans. on Computers*, 49(7):763–766, July 2000.

[11] T. Kobayashi and H. Morita. "Fast Modular Inversion Algorithm to Match any Operation Unit". *IEICE Trans. Fundamentals*, E82-A(5):733–740, May 1999.

[12] Xilinx Inc. Website & Data Sheets: http://www.xilinx.com.

[13] A. Gutub, A. F. Tenca, and C. K. Koc. "Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation". *IEEE Computer Society Annual Symposium on VLSI*, pages 53–58, April 2002.