An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA

Guerric Meurice de Dormale, Philippe Bulens and Jean-Jacques Quisquater.

***Abstract*— Modular multiplication and inversion/division are the most common primitives in today's public key cryptography. Elliptic Curve Public Key Cryptosystems (ECPKC) are becoming increasingly popular for use in mobile appliances where bandwidth and chip area are strongly constrained. For the same level of security, ECPKC use much smaller key length than the commonly used RSA but need modular inversion/division.**

This paper presents an improved algorithm for prime field Montgomery modular inversion. The first important contribution lies in the reduction of the number of operations needed. Resource sharing is also used to lighten the control part of the algorithm. The second contribution is the minimization of the set of different instructions to enable powerful FPGA implementations. Resulting 256-bit circuit achieves a ratio throughput/area improved by at least 70% compared to the only known Montgomery inverse design in FPGA technology.

Though the implementations are first oriented towards FPGA, some improvements are generic. So, they could prove to be also efficient for ASIC designs in terms of area and power consumption.

***Index Terms*— Montgomery Modular Inversion, Prime Field, Elliptic Curves, ECC over GF$(\mathbf{p})$, Reconfigurable Logic.**

I. Introduction

Modular arithmetic plays an important role in public key cryptographic systems. In mobile appliances, very efficient implementations are needed to meet the cost constraint while preserving good computing performance. In this field, modular multiplication has received great attention and numerous papers have been published. The modular inversion problem has also been extensively studied. It can be performed using Fermat little theorem or the well-known extended Euclidian algorithm (or any of the binary variants like the Montgomery inverse [3], [6]). The modular division is believed to be slow and can be replaced by a modular inversion followed by a modular multiplication.

Despite their slowness and their cost, inversion or division are needed in several cases: when creating public-private key pairs for RSA or when deciphering in an ElGamal cryptosystem. Although, the main bottleneck arises in the context of Elliptic Curve Cryptosystems (ECC). In this paper, we will focus on this topic and more precisely over GF(p).

Despite the fact that modular inversion has been extensively studied, not many FPGA implementations have been published. For the Montgomery modular inverse, we found only one paper [2], based on the binary extended Euclidian algorithm.

Microelectronics Laboratory, Crypto Group, UCL, BELGIUM, {gmeurice,bulens,quisquater}@dice.ucl.ac.be

The interest for the developed algorithmic modifications are twofold. First, we reduce the number of operations needed and second, we reduce the set of possible operations in order to meet efficiency in selected hardware devices.

This paper is structured as follows: section II reminds the theoretical bases of ECC over GF(p). Section III describes some constraints of the selected hardware devices needed to understand the algorithmic modification choices. We introduce the Montgomery inverse algorithm, the modified algorithm and its implementation in section IV. Practical results and comparisons with the only known published design lie on section V. Finally, section VI concludes the paper.

II. Elliptic Curve Operations over GF(p)

An elliptic curve E over $GF(p)$, with p a prime number, is defined as the set of points (x, y) verifying the reduced Weierstraß equation:

$$E : f(X,Y) \triangleq Y^2 - X^3 - aX - b \equiv 0 \bmod p.$$

In ECC, the data to be encrypted is represented by a point P on a chosen curve. The encipherment by the key k is performed by computing $Q = P + P + \cdots + P = kP$. This operation, called scalar multiplication, is usually achieved through the *double and add* method (the adaptation of the well-known *square and multiply* to elliptic curves).

When the resulting point is not the point at infinity $\mathcal{O}$, the addition of points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ leads to the resulting point $R = (x_R, y_R)$ through the following computation:

$$\begin{cases} x_R &= \lambda^2 - x_P - x_Q \bmod p \\ y_R &= -y_P + \lambda(x_P - x_R) \bmod p \end{cases}$$

where

$$\lambda = \begin{cases} \frac{y_P - y_Q}{x_P - x_Q} \bmod p & \text{if } P \neq Q \\ \frac{3x_P^2 + a}{2y_P} \bmod p & \text{if } P = Q \end{cases}$$

It should be noticed that it is also possible to represent points in other coordinate systems. The reader can find an interesting overview in [1].

III. FPGA Constraints

ECC need computation over important bit-length modulus (typically 160 bits). The inversion algorithm used in this paper is based on adder, so it is obvious that the critical path resides in the carry chain of the simple carry-propagate adders.

Modern FPGA are built to efficiently implement the addition operation. High speed performance is achieved by the use of a dedicated optimized carry chain. While FPGAs are structured as an array of programmable logic and routing resources, the carry chain is physically wired and therefore exhibits very low delays. It can be used without constraining the routing as long as the carry chain size does not exceed the column height. For this purpose, adders must be adapted to keep satisfactory speed performance.

In the algorithms we deal with, the control logic uses the result of a magnitude comparison, preventing techniques like pipelining. Experience shows that intricate adder structure on FPGA leads to poor speed/area results. So, we decided to choose a simple carry conditional adder [5] to break the carry chain exceeding the column height.

We decided to move towards a Xilinx Virtex or Spartan FPGA for our design. For this kind of device, a simplified Logical Element (LE), equivalent to half a slice, is illustrated in figure 1. For this kind of architecture, another constraint is that 4-input Look-Up Table (LUT) in Adder mode can not use combinatorial circuit with one of the adder input. Indeed, with this kind of carry chain, we must have almost one input directly connected to the serial carry chain multiplexor to enable the full adder function. In figure 1, for example, we can only apply a combinatorial process with the LUT on the A input. So, if we want to spare as much FPGA ressources as possible, we must take care in the algorithmic choices of always keeping an input identical to the adders.

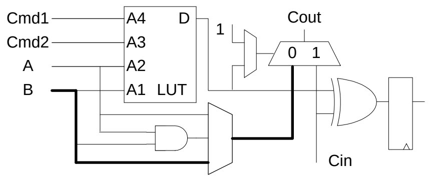

Fig. 1. A simplified Xilinx Virtex Logical Element

The first constraint is implementation specific while the last constraint has been taken into account all along the algorithmic modifications. It will explicitly come to light in the modified algorithms presented below.

IV. The Montgomery Modular Inverse

In this section, we will discuss about integer Montgomery modular inverse. First, we introduce the original algorithm and the Montgomery notation. Then, we present our modified algorithm with motivations about the choices made. The implementation details end this section.

A. Montgomery Modular Inverse Algorithm

The Montgomery modular inverse was first introduced by Kaliski in 1995 [3]. It is defined as the Montgomery representation of the modular inverse, i.e. $X^{-1}2^n \bmod M$, where n is the bit-length of M. It is based on the extended binary GCD algorithm. This algorithm takes place in two phases: the first one, presented in algorithm 1, outputs a partial result $X^{-1}2^k \bmod M$ with $n \leq k \leq 2n$, called the *almost Montgomery inverse* (introduced in [6]). The second part may differ according to the desired output domain: Montgomery domain or real integer. This result can be reached either by iterative halving modulo M or multiplication modulo M.

Algorithm 1 Kaliski Montgomery inverse algorithm, Phase 1

Input: $X \in [1, M-1]$ and M

Output: $R \in [1, M-1]$ and k, where $R = X^{-1}2^k \bmod M$ and $n \leq k \leq 2n$

```
$U \leftarrow M,\ V \leftarrow X,\ R \leftarrow 0,\ S \leftarrow 1,\ k \leftarrow 0$
while $V > 0$ do
  if $U$ even then $U \leftarrow U/2,\ S \leftarrow 2S$
  elsif $V$ even then $V \leftarrow V/2,\ R \leftarrow 2R$
  elsif $U > V$ then $U \leftarrow (U-V)/2,\ R \leftarrow R+S,\ S \leftarrow 2S$
  elsif $V \geq U$ then $V \leftarrow (V-U)/2,\ S \leftarrow S+R,\ R \leftarrow 2R$
  $k \leftarrow k+1$
if $R \geq M$ then $R \leftarrow R - M$
return $R \leftarrow M - R$ and $k$
```

During Phase 1, the following invariants can be verified by induction:

$$\begin{aligned} XR &\equiv -U\,2^k \bmod M \\ M &= US + VR \end{aligned}$$

At the end of the while loop, the values of $U = 1$ and $V = 0$ allow to check that $R = -X^{-1}2^k \bmod M$ which is then corrected to bring the result back in the range $[1, M-1]$. Moreover, the register containing S will end the first phase with the value of the prime modulus M. Also, it has been proved [3] that the number of operations needed to complete algorithm's Phase 1 is at least n and at most $2n$ cycles if X and M are relatively prime.

B. Modified Montgomery Modular Inverse Algorithm

The modified Montgomery modular inverse is presented in algorithm 2. It is clearly hardware oriented to enable an easy transposition towards the FPGA implementation. The UV variables are allowed to be negative and the two's complement representation has been chosen. In the following, sign(T) stands for the sign of T, which is defined as the bit sign in two's complement representation. The absolute value of T will be expressed as abs(T).

The initialization phase has been extended to meet area efficiency with the FPGA. In fact, the original load of the X operand in the register Luv means that we need a multiplexor between the adder and the memory element, wasting one LE per input bit. The added clock cycle is negligible with respect to the global running time of the algorithm.

The first major improvement relies on the removal of the $U-V$ or $V-U$ operation. Indeed, the substraction needed for the magnitude comparison is reused for the $U = (U - V)/2$ or $V = (V-U)/2$ operation. We simply allow results to be negative and we take care of their signs to perform the appropriate next operation. The output sign of the current Luv + Ruv or Luv − Ruv operation also tells which register must be updated (if a switch of the variables must be achieved or not).

Algorithm 2 Modified Montgomery inverse algorithm, Phase 1 and 2

Input: $X \in [1, M-1]$ and M

Output: Lrs $\in [1, M-1]$, where Lrs $= X^{-1}2^n \bmod M$

```
PHASE 1
  $k \leftarrow -\text{bitsize}(M),\ \text{Luv} \leftarrow 0,\ \text{Ruv} \leftarrow 2X,\ \text{Lrs} \leftarrow 0,\ \text{Rrs} \leftarrow 1$
  $\text{Luv} \leftarrow \text{Luv}/2 + \text{Ruv},\ \text{Ruv} \leftarrow M,\ \text{Lrs} \leftarrow \text{Lrs} + \text{Rrs},\ \text{Rrs} \leftarrow 0$
  while true do
    $\text{SLuv} \leftarrow \text{sign}(\text{Luv}),\ \text{SRuv} \leftarrow \text{sign}(\text{Ruv})$
    if $\text{Luv}/2$ even then
      if $\text{SLuv} = \text{sign}(-\text{Luv})$ then break while loop
      $\text{Luv} \leftarrow \text{Luv}/2,\ \text{Rrs} \leftarrow 2\text{Rrs},\ k \leftarrow k+1$
    else
      $\text{tmpuv} \leftarrow \text{Luv}/2,\ \text{tmprs} \leftarrow \text{Lrs},\ \text{Lrs} \leftarrow \text{Lrs} + \text{Rrs}$
      if $(\text{SLuv xor SRuv}) = 1$ then $\text{Luv} \leftarrow \text{Luv}/2 + \text{Ruv}$
      else $\text{Luv} \leftarrow \text{Luv}/2 - \text{Ruv}$
      $k \leftarrow k+1$, ctrl $\leftarrow (\overline{\text{SLuv}}$ and $\overline{\text{SRuv}})$ or $(\overline{\text{SLuv}}$ and $\text{SRuv})$
      if $\text{sign}(\text{Luv}) = \text{ctrl}$ then $\text{Ruv} \leftarrow \text{tmpuv},\ \text{Rrs} \leftarrow 2\text{tmprs}$
      else $\text{Rrs} \leftarrow 2\text{Rrs}$
  /* thanks to the invariant, we have $\text{Lrs} = M$ */
  $\text{Lrs} \leftarrow \text{Lrs} - \text{Rrs},\ \text{Rrs} \leftarrow M$
  if $\text{sign}(\text{Lrs}) = 1$ then $\text{Lrs} \leftarrow \text{Lrs} + \text{Rrs}$

PHASE 2
  while $k \neq 0$ do
    $k \leftarrow k - 1$
    if Lrs even then $\text{Lrs} \leftarrow \text{Lrs}/2$
    else $\text{Lrs} \leftarrow (\text{Lrs} + \text{Rrs})/2$
  return Lrs
```

The second major modification is based on the reusing of the Luv adder at the time of *simple Luv bitshift* to verify the $\text{Luv} = 0$ end criterion. Instead of using a *not so negligible* zero comparison, we check whether the stored Luv sign is equal to the $-\text{Luv}$ sign computed with the right adder. With two's complement representation, this equality means that $\text{Luv} = 0$.

For the last operation of the Phase 1, $U = 1$ and $V = 0$. Fortunately, the algorithm invariant tells us that S is equal to the prime modulus M. We can then directly execute the $M - R$ operation needed to output the correct result.

The bitshift of the Luv variable is performed after its register to enable the use of the memory element directly available after the adder in the chosen FPGA.

We decided to include in the modified algorithm a phase 2 based on *iterative halving modulo* M to convert the almost inverse to the Montgomery domain. The algorithm can be trivially updated to eliminate this conversion or transform the almost inverse to the real inverse.

It must be noticed that a different but related *sign technique* has been used in the recent paper [4].

We will show in the next subsection that all these modifications lead to an efficient FPGA implementation.

C. Montgomery Modular Inverse Implementation

We begin with general observations of the implementation before inspecting specific features of the design.

At first glance, the implementation of the UV stage should be like the one shown in figure 2. This solution, more related to ASIC, exhibits a real algorithm improvement with the removal of one adder, but it is not optimal for FPGA implementation: the multiplexors between the adder and the memory element impose the use of extra LE. A better solution, presented in figure 3, can be achieved if we load the input variable in the right register and allow the *add 0* operation in the left register. The left adder can no longer be used to check the end criterion, so we convert the right multiplexor towards an adder for the purpose of computing $-\text{Luv}$. This implies no additional cost for selected FPGA when we use a simple carry ripple adder.

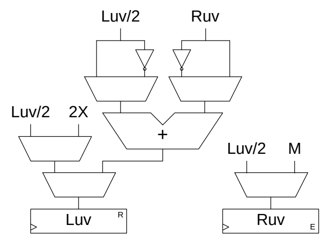

Fig. 2. Early implementation for UV stage, more ASIC related

The asynchronous reset of the Luv register is used to enable its load of the X variable on the next clock cycle. This variable must be left-loaded because the modulus is always odd and the *left add right* operation can only be performed in the left part.

The RS stage execution is delayed by one clock cycle with respect to the UV stage execution. This is motivated by the fact that the switch determined by the equality $\text{sign}(\text{Luv}) = \text{ctrl}$ is serially wired with a carry chain and constitute the critical path. Moreover, this control signal has a large fanout which increases even more the delay. These considerations also apply for the end criterion $\text{SLuv} = \text{sign}(-\text{Luv})$.

LE used for the multiplexor between the Lrs adder and its memory element only serves as a bitshifter to perform the Phase 2 of the algorithm. If only the almost Montgomery inverse is needed, we can simply remove it and then spare more area and control logic.

As explained above, no extra load of the constant M into the Lrs register is needed to perform the last Phase 1 operation.

1) Basic Adder Implementation: Figures 3 and 4 illustrate the two main stages of the basic adder. The small k counter is not shown. The two stages consume 5 LE per bit, but 1 LE can be removed if only the Phase 1 must be performed.

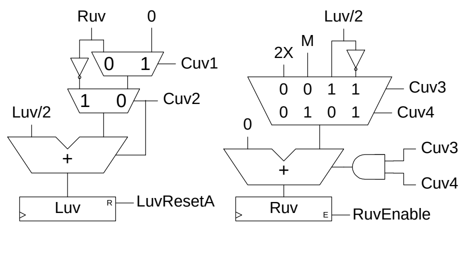

Fig. 3. Montgomery inverse operative part of the basic UV stage

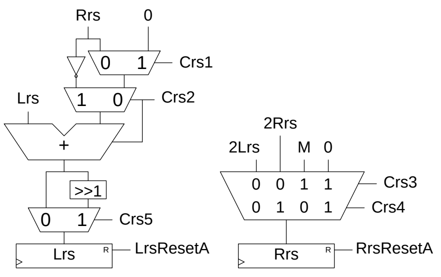

Fig. 4. Montgomery inverse operative part of the basic RS stage

2) Carry Conditional Implementation: Because carry chain can be too high for the FPGA, it can be necessary to use other type of adders to improve performances.

This modified implementation exhibits no difference with the basic adder design except that we replace carry ripple adders with carry conditional adders. However, the Ruv adder does not need all the traditional features of a carry conditional adder. As explained before, the only time it is used for the adder purpose, we only need the sign bit. So, it is more useful to replace this most significant part by a zero comparator. This can be efficiently achieved using LE as 4-input AND gates and the carry chain to process 4 Luv bits by LE.

V. Implementation Results

The modified algorithm has been implemented to provide speed and area comparison of 64, 128, 160 and 256-bit basic and carry conditional adder designs. The VHDL synthesis and place & route have been achieved on Xilinx ISE 6.2.02i. The first device selected is a Xilinx Virtex-E XCV2000e-6bg560 FPGA to guarantee fair comparison with the Daly, Marnane and Popovici design [2]. The second is a Xilinx Spartan3 XC3S200-5tq144 to exhibit performances over a small and low cost FPGA. The operand is loaded by a 32 bits clocked interface and stored in 32-bit shift registers. The result is also sent to 32-bit shift registers to access the 32-bit clocked output interface.

The tables I show implementation results for the two selected devices.

TABLE I
Virtex-E and Spartan3 implementation results

VirtexE / Spartan3 / Design	Frequency (MHz)		Area (Slices)		Thr./Area (kbit/s.Slices)	
64-bit	67	70	302	295	72.5	77.0
32-bit ×2	69	77	394	425	56.9	58.9
128-bit	46	49	558	562	27.2	28.7
64-bit ×2	55	63	723	806	25.0	25.7
160-bit	38	42	687	684	18.2	20.3
80-bit ×2	50	59	890	971	18.5	20.0
256-bit	25	29	1057	1063	7.8	9.0
128-bit ×2	41	41	1390	1490	9.8	9.1

It can be seen that carry conditional adder designs exhibit a better throughput/area only when the carry chain exceeds the height of the selected FPGA. Indeed, the area increase cannot be compensated by small frequency increase. The column height of the selected Virtex-E is 80 CLB per column (and 2 Slices per CLB) while the selected Spartan3 is 20 CLB per column (and 4 Slices per CLB). So, when we reached the 160-bit limit, carry conditional design begin to be attractive.

The throughput/area ratio comparison between our best implementation and the Daly, Marnane and Popovici best implementation is given in the table II. To compute the throughput, we consider the worst case of $(3n+3)$ clock cycles (Phase 1 and 2) for their design and $(3n+5)$ for our design. Unfortunately, they do not give the 160-bit design results.

TABLE II
Performance comparisons

VirtexE Design	Freq. (Mhz)	Area (Slices)	Thr./area (kbit/s.Slices)	Improv.
64-bit	59.93	515	38.3	
Our 64-bit	**67**	**302**	**72.5**	**89 %**
128-bit	46.80	1023	15.2	
Our 128-bit	**46**	**558**	**27.2**	**79 %**
Our 160-bit	**50**	**890**	**18.5**	
256-bit	34.73	2022	5.7	
Our 256-bit	**41**	**1390**	**9.8**	**72 %**

VI. Conclusion

We presented an improved algorithm for prime field Montgomery modular inversion. Those improvements took place in two main phases: first we reduced the number of operations needed in the algorithm and second, we rewrote the operations to take into account the targeted device constraints. To validate the approach, we implemented them in two different FPGA devices.

We achieved a throughput/area ratio of 9.8 kbit/(s.Slices) for a 256-bit Virtex-E design. It represents an improvement by at least 70% of the only known Montgomery inverse design on FPGA technology. We also produced results for Spartan3 FPGA to exhibit performance over a small and low cost FPGA device. Results can be even more improved if only the almost Montgomery inverse is needed.

Though the modified algorithms are first oriented towards FPGA, some improvements are generic. Further work could use the proposed algorithms to issue efficient designs for ASIC in terms of area and power consumption.

References

[1] H. Cohen, A. Miyaji and T. Ono, *Efficient Elliptic Curve Exponentiation using Mixed Coordinates*, ASIACRYPT 1998, Springer-Verlag, 1998, LNCS 1423, pp. 51–65.
[2] A. Daly, L. Marnane and E. Popovici, *Fast Modular Inversion in the Montgomery Domain on Reconfigurable Logic,* Irish Signals and Systems Conference — ISSC 2003, Limerick, July 1–2, 2003.
[3] B. S. Kaliski Jr., *The Montgomery Inverse and its Applications*, IEEE Transactions on Computers, 44(8), pp. 1064–1065, August 1995.
[4] M. Naseer and E. Savaş, *Hardware Implementation of a Novel Inversion Algorithm*, The 46th IEEE Midwest Symposium on Circuits and Systems, Cairo, Egypt, December 27–31, 2003.
[5] J. M. Rabaey, *Digital integrated circuits: a design perspective*, Prentice-Hall upper saddle river (N.J.), 1996
[6] E. Savaş and Ç. K. Koç, *The Montgomery Modular Inverse - Revisited*, IEEE Transactions on Computers, 49(7), pp. 763–766, July 2000.