A Reconfigurable Arithmetic Logic Unit for Elliptic Curve Cryptosystems over $GF(2^m)$

Mathias Schmalisch, Dirk Timmermann
University of Rostock, Institute of Applied Microelectronics and Computer Science
Richard-Wagner-Str. 31, 18119 Rostock, Germany
{mathias.schmalisch, dirk.timmermann}@etechnik.uni-rostock.de

***Abstract*—In this paper we present an arithmetic logic unit (ALU) for elliptic curve cryptosystems over $GF(2^m)$. The novelty of these ALU is the inclusion of a hardware inversion operation, which is as fast as a multiplication. So we can use faster algorithms for the computation of $k{\cdot}P$. Although we use a serial multiplication and inversion, we achieve a faster computation than other parallelized hardware implementations.**

***Index Terms*—Elliptic curve cryptosystems, arithmetic logic unit, finite field arithmetic, polynomial basis**

I. Introduction

The key of classic public key algorithms like RSA, ElGamal and Diffie Hellmann that uses the modular exponentiation must have today a length of 1024 bits or more to be secure. A new approach is the Elliptic Curve Cryptography (ECC) which only needs a key length of 160 bits at the same level of security [1].

With ECC it is possible to do key exchange, digital signature, authentication, encryption and so forth. The function which is needed for this is the scalar multiplication $k{\cdot}P$, where k is an integer and P a point on an elliptic curve. The scalar multiplication can be computed with point addition and point doubling on an elliptic curve. These again are based on computations in finite fields.

For cryptography the finite fields $GF(p)$ and $GF(2^m)$ are used. The finite field GF(p) uses the modular arithmetic with a prime number p as modulus. This finite field is very suitable for software computation. For hardware computation the finite field of $GF(2^m)$ is more interesting. The field $GF(2^m)$ can be viewed as a vector space of dimension m over $GF(2)$. So an addition of two numbers can be performed by bitwise XOR combination of the vector representation, and takes only one clock cycle. In the finite field $GF(2^m)$ the arithmetic operations addition, multiplication, squaring and inversion have to be supported to compute the point operations.

To compute the scalar multiplication several algorithms with different numbers of finite field operations exist. The algorithms are split in two great groups. One group which compute the point operations on the elliptic curve in the affine plane and the other group using the projective plane. If we compute the scalar multiplication in affine coordinates, then we need an inversion for each point addition and point doubling. On the other hand if we compute the scalar multiplication in projective coordinates we need only one inversion for a scalar multiplication but more field multiplications and squarings. The inversion is the operation which is the hardest to compute. Therefore the known hardware implementations use algorithms with projective coordinates.

In this paper we present an ALU for finite field operations in $GF(2^m)$ which uses a serial hardware inversion. This inversion is as fast as a serial multiplication. So it is possible to use algorithms in affine coordinates. At the end of the paper we compare our serial hardware implementation with other parallelized hardware implementations.

II. Elliptic Curves

For the implementation of elliptic curve cryptosystems in hardware are the elliptic curves over finite field $GF(2^m)$ best suitably. In this case the field has the characteristic 2 and the equation for a non-supersingular elliptic curve has the form

$$E : y^2 + xy = x^3 + ax^2 + b, \tag{1}$$

with $b \neq 0$. To compute the scalar multiplication we have to compute the point addition and point doubling on this curve.

To compute the point addition of $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ in affine coordinates we need the sloop λ of the line through P and Q. If $P = Q$ than we have a point doubling, in this case the line is the tangent to the curve P. Therefore the equation for the slope λ of the line has the following form

$$\lambda = \begin{cases} \dfrac{y_1 + y_2}{x_1 + x_2}, & P \neq Q, \\ x_1 + \dfrac{y_1}{x_1}, & P = Q. \end{cases} \tag{2}$$

The new coordinates for $R = P + Q$ with $R = (x_3, y_3)$ can be computed with the equation

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + b, \\ y_3 &= \lambda\left(x_1 + x_3\right) + x_3 + y_1. \end{aligned} \tag{3}$$

For the point doubling, this equation can be simplified to

$$x_3 = \lambda^2 + \lambda + b,$$
$$y_3 = \lambda x_3 + x_3 + x_1^2 . \qquad (4)$$

With these equations it is possible to compute the point addition and point doubling in affine coordinates. To compute the scalar multiplication with the “Double and Add/Subtract” algorithm we need also a point subtract. For a point $P = (x_1, y_1)$ in affine coordinates on an elliptic curve E is $-P = (x_1, y_1 + x_1)$, therefore the point subtraction can be computed with a point addition and an additional finite field addition. For the “Double and Add/Subtract” Algorithm we have to recode the integer k to a singed digit representation, this can be done with well known algorithms like the Canonical Recoding Algorithm or the Modified Booth Algorithm.

Double and Add/Subtract Algorithm:
Input: An Integer $k > 0$ and a point P
Output: $Q = k{\cdot}P$

```
1. $k := (k_{n-1}, \ldots, k_1, k_0)_{SD}$,  $k_i \in \{0, 1, -1\}$
2. $Q := P$
3. for $i$ from $n$ - 2 downto 0 do
4.     $Q := 2Q$
5.     if $k_i$ = "1" then
6.         Q := Q + P
7.     elseif ki = "-1" then
8.         $Q := Q$ - P
9. return $Q$
```

For this algorithm there exists an improvement that was presented in [2]. Whit this improvement it is possible to save multiplications in the finite field.

III. Inversion

The speed of the inversions determines what algorithms for the scalar multiplication is the best. For the computation of the inversion there exist two different methods. One method is Fermat’s Theorem and the other one is Euclid’s Algorithm.

A. *Fermat’s Theorem*

Fermat’s Theorem means

$$a^{n-1} \equiv 1 \bmod n, \qquad (5)$$

if n is relative prime to a. Therefore

$$a^{n-2} \equiv a^{-1} \bmod n. \qquad (6)$$

This theorem also works for finite fields of $GF(2^m)$. So the inversion can be computed with squarings and multiplications. An optimized algorithm was presented in [3]. If the inversion will be computed with this method, than the speed is very slow compared to the other functions in the finite filed. If the scalar multiplication $k{\cdot}P$ will be computed with projective coordinates this method is used to compute the inversion because in this case the inversion is only needed once.

B. *Euclid’s Algorithm*

The inversion with Euclid’s Algorithm for finite fields $GF(2^m)$ in polynomial basis, where $F(x)$ is the irreducible polynomial, can be computed with the following algorithm

Inversion using Euclid’s Algorithm:
Input: $F(x)$ the irreducible polynomial
$B(x)$ the polynomial for which inversion is needed
Output: $B^{-1}(x)$

```
1. $s := F(x)$, $r := B(x)$, $v := 0$, $u := 1$
2. repeat
3.     $q := \lfloor s / r \rfloor$
4.     $t := s - q \cdot r$, $s := r$, $r := t$
5.     $t := v - q \cdot r$, $v := u$, $u := t$
6. until $r = 0$
7. return $v$
```

Hardware implementations for this algorithm do exist. A very interesting implementation was presented in [4]. This implementation can compute the inversion in m clock cycles, where m is the field width of the finite field $GF(2^m)$. Therefore this implementation is suitable for a computation of the scalar multiplication in affine coordinates where you need an inversion for each point operation. The biggest advantage of this solution is the speed. Then this inversion is as fast as a serial multiplication.

IV. Implementation

For our ALU we chose the finite field $GF(2^m)$ on polynomial basis. Through reconfiguration this ALU can be optimized for any field order m and the corresponding irreducible polynomial.

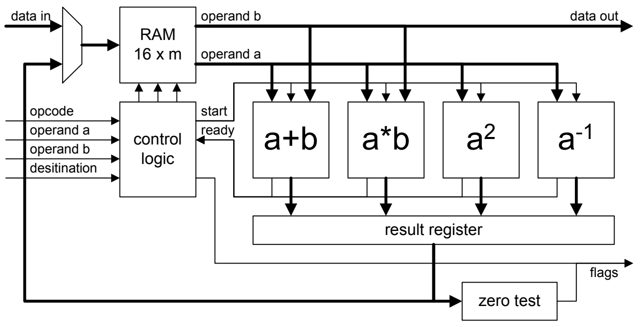

Fig. 1. Architecture of the arithmetic logic unit

Additional to the four operations a memory, a control logic and a zero test were implemented. The memory stores operands, temporary values and the final result. The memory can be written and read from outside. To read two operands at the same time we used dual ported RAM for this memory. Typical inputs are the parameters of the elliptic curve a and b and the coordinates of the starting point P.

The control logic selects the operands from the memory and controls the operations. If the computation is finished, the control logic indicates this with a flag. To control the ALU an opcode and three addresses for the memory can be send to the control logic. The opcode selects on of the four finite field functions. The first two addresses selects the operands for the finite field function and the third address selects the address where the result will be stored. There also exist two opcodes to read and write the memory.

The zero test compares the result register with zero and shows this comparison result with a flag. The zero test could be easily realized with an OR tree over the result register. This is needed for the inversion because an inversion of zero is not defined.

A. *Addition*

The addition in the finite field of $GF(2^m)$ is very easy to compute. For the chosen field the addition of two numbers is the simplest operation, since it is only a XOR combination of the bits of the two addends. Therefore we need only m XOR gates and one clock cycle for this operation.

B. *Multiplication*

For the multiplication we chose a serial implementation, where the reduction with the irreducible polynomial was integrated. So we need m - 1 XOR gates for the addition, several additional gates for the integrated reduction with the irreducible polynomial, two shift registers, one register for the multiplicand and two multiplexers.

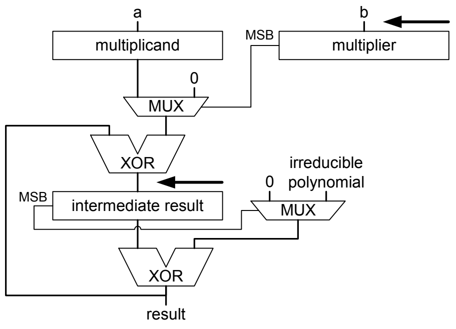

Fig. 2. Serial multiplication with integrated reduction

Such a serial multiplier computes the multiplication in m clock cycles.

C. *Squaring*

Squaring is a special case of the multiplication but it is faster to compute. The binomial equation in the finite field of $GF(2^m)$ has the form

$$(a+b)^2 = a^2 + \underbrace{2ab}_{\text{mod } 2=0} + b^2 = a^2 + b^2 . \qquad (7)$$

So the squaring of a number is very easy. We only need to add zero bits between each bit of the number. Than we have a number that is 2m-1 bit long. This number must be reduced with the irreducible polynomial. The Reduction only need $(m + k - 1)/2$ XOR gates for irreducible trinomials $F(x) = x^m + x^k + 1$ and therefore the squaring can be computed in one clock cycle [5].

D. *Inversion*

The most complex operation is the inversion. A serial implementation exist which computes the operation in m clock cycles [4] with Euclid's Algorithm. Out of all operations this one is the most logic intensive. In Figure 1 the architecture of the ALU is shown.

V. Results

The prototype was coded in the hardware description language VHDL. For the net list synthesis we used the tool Synplify Pro 7.3 from Synplicity. The target synthesis has been realized with Xilinx ISE 5.2i. The prototype was implemented in a Xilinx FPGA XCV400E-8-BG560. To compare our results to the fastest implementation, we used the finite field $GF(2^{167})$ with the irreducible polynomial $F(x) = x^{167} + x^6 + 1$. Table 1 summarizes the results.

TABLE I
Key performance figures of the prototype (Utilization)

Clock frequency	Lookup Tables	Flip Flops	Equivalent gates
82.3 MHz	4,245 (44%)	1,393 (14%)	56,932

For the scalar multiplication we used the "double and add/subtract" algorithm for affine coordinates with the improvement from [2]. With this algorithm our ALU computes a scalar multiplication in 1.3 ms.

Table 2 lists the performance of leading hardware implementations for FPGA's. In our implementation we use a serial multiplication and inversion, where others use parallelized implementations. Nevertheless our implementation is faster than most of the others. Since we use a serial implementation, we achieve also a high clock frequency. The fastest implementation uses a semi systolic multiplier [6]. Therefore it is faster than our implementation. By utilizing the potential of the presented architecture to parallelize the multiplication as well as the inversion operation it is likely that the architecture presented in [6] will be outperformed.

TABLE II
Performance comparison

Implementation	Target Platform	Field width	$k{\cdot}P$ (ms)	Performance (kbit/s)
Okada, Torii, Itoh, Takenaka [7]	Altera FPGA EPF10K, 3 MHz	163	80.7	2.0
Leung, Ma, Wong, Leong [8]	Xilinx FPGA XCV300, 45 MHz	113	3.7	29.8
Ernst, Jung, Madlener, Huss, Blümel [9]	Atmel FPSLIC AT94K40, 12 MHz	113	1.4	78.8
Ernst, Klupsch, Hauck, Huss [10]	Xilinx FPGA XC4085XLA, 37 MHz	155	1.3	116.4
Orlando, Paar [6]	Xilinx FPGA XCV400E, 76.7 MHz	167	0.21	776.7
Our ALU implementation	Xilinx FPGA XCV400E, 82.3 MHz	167	1.3	126.2

VI. Conclusion

In this paper we show that a serial implementation reaches a high performance if an efficient algorithm (presented in [2]) and hardware computation for the inversion (presented in [4]) is used. With this implementation of an arithmetic logic unit for finite fields it is possible to compute the scalar multiplication faster than other parallelized hardware implementations.

References

[1] A. Lenstra and E. Verheul, "Selecting Cryptographic Key Sizes," *Journal of Cryptology*, vol. 14, no. 4, pp. 255-293, 2001.

[2] K. Eisenträger, K. Lauter and P. Montgomery, "An efficient procedure to double and add points on an elliptic curve," *Cryptology ePrint Archive*, Report 2002/112, 2002.

[3] T. Itoh, S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ using normal bases," *Information on Computation*, vol. 78, no. 3, pp. 171-177, 1988.

[4] H. Brunner, A. Curiger and M. Hofstetter, "On computing multiplicative inverses in $GF(2^m)$," *IEEE Trans. on Computers*, vol. 42, no. 8, pp. 1010-1015, 1993.

[5] H. Wu, "Low complexity bit-parallel finite field arithmetic using polynomial basis," *Workshop on Cryptographic Hardware and Embedded Systems: Proceedings CHES '99*, Lecture Notes in Computer Science, vol. 1717, Springer Verlag, pp. 280-291, 1999.

[6] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for $GF(2^m)$," *Workshop on Cryptographic Hardware and Embedded Systems: Proceedings CHES 2000*, Lecture Notes in Computer Science, vol. 1965, Springer Verlag, pp. 41-56, 2000.

[7] S. Okada, N. Torii, K. Itoh and M. Takenaka, "Implementation of elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA," *Workshop on Cryptographic Hardware and Embedded Systems: Proceedings CHES 2000*, Lecture Notes in Computer Science, vol. 1965, Springer Verlag, pp. 25-40, 2000.

[8] K. H. Leung, K. W. Ma, W. K. Wong and P. H. W. Leong, "FPGA implementation of a microcoded elliptic curve cryptographic processor," in *Proc. IEEE FCCM 2000*, Napa Valley, pp. 68-76, 2000.

[9] M. Ernst, M. Jung, F. Madlener, S. Huss and R. Blümel, "A reconfigurable system on chip implementation for elliptic curve cryptography over $GF(2^n)$," *Workshop on Cryptographic Hardware and Embedded Systems: Proceedings CHES 2002*, Lecture Notes in Computer Science, vol. 2523, Springer Verlag, pp. 381-399, 2003.

[10] M. Ernst, S. Klupsch, O. Hauck and S. A. Huss, "Rapid prototyping for hardware accelerated elliptic curve public-key cryptosystems," in *Proc. 12th IEEE Workshop on Rapid System Prototyping*, Monterey, pp. 24-31, 2001.