HECC Goes Embedded: An Area-Efficient Implementation of HECC

Junfeng Fan, Lejla Batina, and Ingrid Verbauwhede

Katholieke Universiteit Leuven, ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium
{junfeng.fan,lejla.batina,ingrid.verbauwhede}@esat.kuleuven.be

Abstract. In this paper we describe a high performance, area-efficient implementation of Hyperelliptic Curve Cryptosystems over GF(2^m). A compact Arithmetic Logic Unit (ALU) is proposed to perform multiplication and inversion. With this ALU, we show that divisor multiplication using affine coordinates can be efficiently supported. Besides, the required throughput of memory or Register File (RF) is reduced so that area of memory/RF is reduced. We choose hyperelliptic curves using the parameters $h(x) = x$ and $f(x) = x^5 + f_3x^3 + x^2 + f_0$. The performance of this coprocessor is substantially better than all previously reported FPGA-based implementations. The coprocessor for HECC over GF(2^{83}) uses 2316 slices and 2016 bits of Block RAM on Xilinx Virtex-II FPGA, and finishes one scalar multiplication in 311 μs.

Keywords: Hyperelliptic Curve Cryptosystems, Modular multiplication, Modular inversion, FPGA.

1 Introduction

Public-Key Cryptography (PKC) [10], introduced in the mid 70's by Diffie and Hellman, ensures a secure communication over an insecure network without prior key agreement. PKC is widely used for digital signatures, key agreement and data encryption. The best-known and most commonly used public-key cryptosystems are RSA [26] and Elliptic Curve Cryptography (ECC) [23,19], but recently HyperElliptic Curve Cryptography (HECC) [20] is catching up. The main benefit for curve-based cryptography *e.g.* ECC and HECC is that they offer equivalent security as RSA for much smaller parameter sizes. The advantages result in smaller data-paths, less memory and lower power consumption.

Implementing HECC on a resource-constrained platform has been a challenge in both area and performance. Over the past few years, HECC have been implemented in both software [25,27] and hardware [4,7,11,15]. However, the implementations so far failed in reaching the performance of ECC implementations with comparable hardware cost. Table 1 compares the computational complexity of point/divisor operations in ECC and HECC as in [2]. Here I, M and S denote modular inversion, multiplication and squaring, respectively. Note that Table 1 is not exhaustive, and a comprehensive description of different coordinates as

R. Avanzi, L. Keliher, and F. Sica (Eds.): SAC 2008, LNCS 5381, pp. 387–400, 2009.

Table 1. Modular Operations Required by Point/Divisor Operations in GF(2^m) [2]

		PA/DA	PD/DD	Coordinates Conversion
ECC	Affine	I+2M+S	I+2M+S	-
	Projective	15M+3S	7M+4S	I+2M
HECC	Affine	I+22M+3S	I+20M+6S	-
	Projective	49M+4S	38M+7S	I+4M

well as their computational complexity can be found in [2]. In addition, state of the art regarding various types of coordinates for all types of curve-based cryptosystem can be found in [9]. For example, ECC over GF(2^{163}) and HECC over GF(2^{83}) are supposed to offer equivalent security as 1024-bit RSA [2]. Using projective coordinates, one EC Point Addition (PA) requires 15 multiplications and 3 squarings in GF(2^{163}), while one HEC Divisor Addition (DA) requires 49 multiplications and 4 squarings in GF(2^{83}), which is much more complex even with parameters of half bit-lengths. In order to speed up HECC implementations, parallel multipliers [4,7] or inverters [15] were used. As a result, an ALU becomes large in the area. In order to efficiently feed data to parallel multipliers and inverters, a high-throughput Register File (RF) with an additional control logic *i.e.* a MUX array connected to ALU is required. This adds even more area to implementations.

In this paper, we describe a compact HECC coprocessor on an FPGA platform. The coprocessor utilizes a unified multiplier/inverter, which supports both multiplication and inversion. This architecture brings three main advantages. First, the fast inverter makes affine coordinates very efficient. Second, as the multiplier and inverter share partial data-path, it is much smaller in area compared to previous implementations. Third, using only one multiplier/inverter, the required throughput of Memory or RF is comparably low. Therefore we can reduce the area of the memory. Note that the architecture proposed here for FPGA design can also lead to an area-efficient design in ASICs. The coprocessor was synthesized with Xilinx ISE8.1i. On Virtex-II FPGA (XC2V4000), this coprocessor finishes one scalar multiplication of HECC over GF(2^{83}) in 311 μs using 2316 slices and 2016 bits memory. To the best of our knowledge, this implementation is faster than all proposed FPGA-based implementations of HECC, while the area is much smaller than that of the fastest reported implementation [15].

The rest of the paper is organized as follows. Section 2 gives a brief introduction on the previous work. Section 3 describes the mathematical background of HECC and field arithmetic. Section 4 describes the architecture of the proposed HECC coprocessor. In Sect. 5 we show the implementation results. We conclude the paper and give some future work in Sect. 6.

2 Previous Work

In 2001, Wollinger described the first hardware architecture for HECC implementations using Cantor's algorithm [6] in his thesis [32]. However, the architecture was only outlined. The first complete hardware implementation of HECC was presented in [4]. It is also based on Cantor's algorithm, but with improvement on the calculation of Greatest Common Divisor (GCD). This implementation, using 16600 slices on Xilinx Virtex II FPGA, supports a genus-2 HEC over GF(2^{113}). One scalar multiplication takes 20.2 ms on this coprocessor running at 45MHz. This work was further improved in [7].

In 2002, Lange generalized the explicit formulae for HECC over finite fields with arbitrary characteristic [21]. This was first implemented on 32-bit embedded processors (ARM7TDMI and PowerPC) in [25]. The inversion in this algorithm was performed with Extended Euclidean Algorithm (EEA). The first hardware implementation of HECC using explicit formulae was described in [12]. Further improvement by using mixed coordinates and simplified curves were proposed in [11]. In [11] the coprocessor, running at 45.3MHz, deploys 25272 slices on Xilinx Virtex II FPGA. With this implementation 2.03 ms is required to perform one scalar multiplication of HECC over GF(2^{113}). There are some ASIC implementations of HECC using projective coordinates. For example, Sakiyama proposed a HECC coprocessor [28] using 0.13-μm CMOS technology. The coprocessor runs at 500 MHz, and can perform one scalar multiplication of HECC over GF(2^{83}) in 63 μs.

The first hardware implementations of HECC using affine version of explicit formulae were described in [31], which described so far the fastest FPGA-based HECC coprocessor. This coprocessor uses three modular multipliers and two modular inverters. It uses 7785 slices on Xilinx Virtex II FPGA(XC2V4000), and can reach a clock frequency of 56.7MHz. One scalar multiplication of HECC over GF(2^{81}) takes 415 μs.

3 Mathematical Background

3.1 Hyperelliptic Curve Cryptography

Hyperelliptic curves are a special class of algebraic curves; they can be viewed as generalization of elliptic curves. Namely, a hyperelliptic curve of genus $g = 1$ is an elliptic curve, while in general, hyperelliptic curves can be of any genus $g \geq 1$.

Let $\overline{\mathrm{GF}}(2^m)$ be an algebraic closure of the field GF(2^m). Here we consider a hyperelliptic curve C of genus $g = 2$ over GF(2^m), which is given with an equation of the form:

$$C : y^2 + h(x)y = f(x) \quad in \quad \mathrm{GF}(2^m)[x, y], \tag{1}$$

where $h(x) \in \mathrm{GF}(2^m)[x]$ is a polynomial of degree at most g ($deg(h) \leq g$) and $f(x)$ is a monic polynomial of degree $2g + 1$ ($deg(f) = 2g + 1$). Also, there are no solutions $(x, y) \in \overline{GF}(2^m) \times \overline{GF}(2^m)$ which simultaneously satisfy the equation (1) and the equations: $2v + h(u) = 0, h'(u)v - f'(u) = 0$. These points are called singular points. For the genus 2, in the general case the following equation is used $y^2 + (h_2x^2 + h_1x + h_0)y = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$.

A divisor D is a formal sum of points on the hyperelliptic curve C *i.e.* $D = \sum m_P P$ and its degree is $degD = \sum m_P$. Let Div denotes the group of all divisors on C and Div_0 the subgroup of Div of all divisors with degree zero. The Jacobian J of the curve C is defined as quotient group $J = Div_0/P$. Here P is the set of all principal divisors, where a divisor D is called principal if $D = div(f)$, for some element f of the function field of C $(div(f) = \sum_{P \in C} ord_P(f)P)$. The discrete logarithm problem in the Jacobian is the basis of security for HECC. In practice, the Mumford representation according to which each divisor is represented as a pair of polynomials $[u, v]$ is usually used. Here, u is monic of degree 2, $degv < degu$ and $u|f - hv - v^2$ (so-called reduced divisors). For implementations of HECC, we need to implement the multiplication of elements of the Jacobian *i.e.* divisors with some scalar.

The main operation in any hyperelliptic curve based primitive is scalar multiplication, *i.e.* mD where m is an integer and D is a reduced divisor in the Jacobian of some hyperelliptic curve C. The first algorithm for arithmetic in the Jacobian is due to Cantor [6]. However, until "explicit formulae" were invented, the HECC was not considered a suitable alternative to EC based cryptosystems. For geni 2 and 3, there was some substantial work on the formulae and algorithms for computing the group law on the Jacobian have been optimized. Algorithms for the group operation for the case of genus 2 hyperelliptic curves, which we used are due to Lange [22].

The main operation in any curve-based primitive (ECC or HECC) is the scalar multiplication. Looking at the arithmetic for both ECC/HECC the only difference between ECC and HECC is in the group operations. On this level both ciphers consist of different sequences of operations. Those for HECC are more complex when compared with the ECC point operation, but they use shorter operands. The divisor scalar multiplication is achieved by repeated divisor addition and doubling. Many techniques that help to speed up ECC scalar multiplication are also applicable to HECC. For example, using Non-Adjacent Form (NAF) for scalar representation or window method can also improve HECC performance.

3.2 Field Arithmetic

An element α in GF(2^m) can be represented as a polynomial $A(x)=\sum_{i=0}^{m-1} a_i x^i$, here $a_i \in$ GF(2). Addition of two elements in GF(2^m) is performed as polynomial addition in GF(2)

$$\sum_{i=0}^{m-1} a_i x^i + \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} (a_i \oplus b_i) x^i,$$

where $\oplus$ is XOR operation.

Multiplication. In the literature there are various algorithms and architectures [3,30] proposed for modular multiplication in GF(2^m). The bit-serial algorithms can be classified into two categories, the Most Significant Bit (MSB) first algorithms and the Least Significant Bit (LSB) first algorithms. It is important to point out that LSB-first bit-serial multiplier has shorter critical path than MSB-first bit-serial multipliers [3]. In this paper, we use the LSB-first algorithm.

Algorithm 1. LSB-first bit-serial modular multiplication in GF(2^m) [3]

```
Input: $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$, irreducible binary polynomial $P(x)$ with $\deg(P(x)) = m$.
Output: $A(x)B(x) \bmod P(x)$.
$C(x) \leftarrow 0$, $A'(x) \leftarrow A(x)$;
for $i = 0$ to $m - 1$ do
    $C(x) \leftarrow C(x) + b_i A'(x)$;
    $A'(x) \leftarrow xA'(x) \bmod P(x)$;
end for
Return: $C(x)$.
```

Inversion. A multiplicative inverse of $A(x)$ is a polynomial $A^{-1}(x)$ in GF(2) such that $A^{-1}(x)A(x) \equiv 1 \bmod P(x)$. Compared with the other modular operations, modular inversion is considered as a computationally expensive operation. The most commonly used methods to perform the modular inversion are based on Fermat's little theorem [1], Extended Euclidean Algorithm [18] and Gaussian elimination [16]. EEA is widely used to perform inversion in practice.

The schoolbook EEA-based inversion algorithm in GF(2^m) is commonly considered inefficient due to the long polynomial division in each iteration. This problem was partially solved by replacing degree comparison with a counter [5].

In [34], Yan *et al.* proposed a modified inversion algorithm based on the EEA. Algorithm 2 shows this inversion algorithm. Here we use $S^i(x)$ to denote the value of $S(x)$ after i^{th} iteration, and d_0^{i-1} the LSB of d^{i-1}. The complement of C_1 is represented as $\bar{C_1}$. Unlike many other EEA variants [14,5,18], this algorithm has no modular operations, thus a short critical path delay can be easily achieved. Besides, with a fixed number of iterations, it is more secure against side-channel analysis.

4 HECC Coprocessor Architecture

In this section we describe a compact coprocessor architecture for HECC over GF(2^m). Two main approaches are used to reduce the area: using compact ALU and reducing memory area. First, we propose a unified digit-serial modular multiplier/inverter, which enables a small ALU. Second, we investigate the characteristics of the ALU, and reduce area of memory block as well as its interconnecting network.

4.1 Modular Multiplier

As shown in Algorithm 1, the main operation in LSB-first multiplication is $(bA(x) + C(x))$, which can be performed by a row of AND gates and XOR gates

Algorithm 2. EEA-Based Inversion Algorithm [34]

Input: irreducible binary polynomial $P(x)$ with $\deg(P(x)) = m$, polynomial $A(x)$ with $\deg(A(x)) < m$.
Output: $A^{-1}(x) \bmod P(x)$.

```
$R^0(x) \leftarrow P(x)$, $S^0(x) \leftarrow xA(x)$, $H^0(x) \leftarrow 0$, $J^0(x) \leftarrow x^m$, $d^0 \leftarrow 2$, $sign^0 \leftarrow 1$;
for $i = 1$ to $2m - 1$ do
    $C_1 \leftarrow s_m^i$, $C_2 \leftarrow C_1 \wedge sign^{i-1}$;
    $sign^i \leftarrow \begin{cases} \bar{C_1} & \text{if } sign^{i-1} = 1; \\ d_0^{i-1} & \text{if } sign^{i-1} = 0; \end{cases}$
    $S^i(x) \leftarrow \begin{cases} x(R^{i-1}(x) + S^{i-1}(x)) & \text{if } C_1 = 1; \\ xS^{i-1}(x) & \text{if } C_1 = 0; \end{cases}$
    $J^i(x) \leftarrow \begin{cases} H^{i-1}(x) + J^{i-1}(x) & \text{if } C_1 = 1; \\ J^{i-1}(x) & \text{if } C_1 = 0; \end{cases}$
    $R^i(x) \leftarrow \begin{cases} S^{i-1}(x) & \text{if } C_2 = 1; \\ R^{i-1}(x) & \text{if } C_2 = 0; \end{cases}$
    $H^i(x) \leftarrow \begin{cases} J^{i-1}(x)/x & \text{if } C_2 = 1; \\ H^{i-1}(x)/x & \text{if } C_2 = 0; \end{cases}$
    $d^i \leftarrow \begin{cases} 2d^{i-1} & \text{if } sign^i = 1; \\ d^{i-1}/2 & \text{if } sign^i = 0; \end{cases}$
end for
```

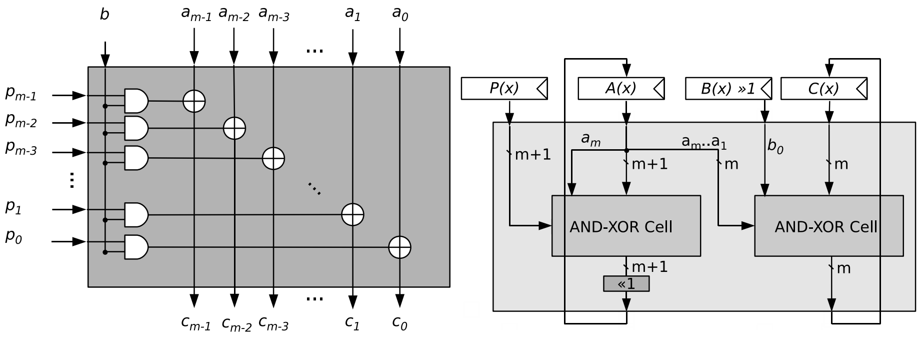
Return: $H^{2m-1}(x)$.

shown in Figure 1(a). Figure 1(b) shows the architecture of a LSB-first bit-serial multiplier. Two $(m+1)$-bit registers are used to hold the parameter $P(x)$, $A(x)$ and two m-bit registers to hold $B(x)$ and the partial product $C(x)$. Note that $B(x)$ is shifted to right by one bit in each clock cycle. Here $(a_m P(x) + A(x))$ and $(b_0 A(x) + C(x))$ is performed on the left and right side, respectively. If low Hamming weight irreducible polynomials are used, the AND-XOR cell on the left side can be simplified. For example, using $P(x) = x^{83} + x^7 + x^4 + x^2 + 1$, only 4 AND gates and 4 XOR gates are required to perform $(a_m P(x) + A(x))$.

It is clear that the critical path delay is $T_{\text{AND}} + T_{\text{XOR}}$, where T_{AND} and T_{XOR} denote the delay of a 2-input AND and XOR gate, respectively. One multiplication in GF(2^m) takes m clock cycles on this bit-serial multiplier.

4.2 Unified Modular Inverter and Multiplier

We propose a unified architecture which can perform both multiplication and inversion. In [8], Daly *et al.* have proposed a unified ALU for GF(p). It can perform addition, subtraction, multiplication and inversion. Compared with this ALU, our unified inverter/multiplier in GF(2^m) has a shorter critical path delay, and can be implemented in a digit-serial manner to achieve a higher throughput. Figure 2 shows the data-path of our proposed bit-serial inverter and multiplier.



(a) AND-XOR Cell

(b)Bit-serial modular multiplier

Fig. 1. Bit-serial modular multiplier

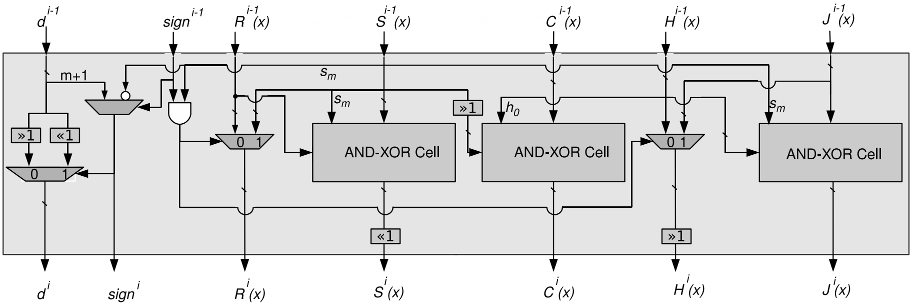


Fig. 2. Bit-serial modular multiplication/inversion unit

It realizes both Algorithm 1 and Algorithm 2. The multiplier and the inverter share one AND-XOR cell and three registers. The critical path delay is $2T_{\mathrm{MUX}}$. Here T_{MUX} denotes the delay of a 2-input multiplexer. This multiplier/inverter finishes one inversion operation in GF(2^m) in $(2m-1)$ clock cycles.

This data-path supports the following operations:

1. Modular Multiplication
   - Initialization ($i = 0$), $R(x) \leftarrow P(x)$, $S(x) \leftarrow xA(x)$, $H(x) \leftarrow B(x)$, $C(x) \leftarrow 0$, $d \leftarrow 0$, $sign \leftarrow 0$;
   - During the whole loop ($0 < i < m+1$), $d^i = 0$, $sign^i = 0$, thus, $R^i(x) = R^{i-1}(x) = P(x)$, $H^i(x) \leftarrow H^{i-1}(x)/x$, $A^i(x) \leftarrow x(A^{i-1}(x) + a_m P(x))$, and $C^i(x) \leftarrow h_0 A^{i-1}(x)/x + C^{i-1}(x)$;
   - Return $C^m(x)$.
2. Modular Inversion
   - Initialization ($i = 0$), $R(x) \leftarrow P(x)$, $S(x) \leftarrow xA(x)$, $H(x) \leftarrow 0$, $J(x) \leftarrow x^m$, $d \leftarrow 2$, $sign \leftarrow 1$;

- During the whole loop ($0 < i < 2m$), $S^i(x) \leftarrow x(S^{i-1}(x) + s_m R^{i-1}(x))$, $J^i(x) \leftarrow J^{i-1}(x) + s_m H^{i-1}(x)$,
  - If $C_2 = 1$, then $R^i(x) \leftarrow S^{i-1}(x)$, $H^i(x) \leftarrow J^{i-1}(x)/x$;
  - If $C_2 = 0$, then $R^i(x) \leftarrow R^{i-1}(x)$, $H^i(x) \leftarrow H^{i-1}(x)/x$;
- Return $H^{2m-1}(x)$.

4.3 Compact Digit-Serial Inverter/Multiplier for HECC

In order to achieve higher throughput, a digit-serial inverter/multiplier can be implemented with multiple bit-serial multiplication and inversion units. We propose a flexible architecture which allows us to explore the trade-off between performance and hardware cost. Figure 3 shows the architecture where 3 unified inversion multiplication units ($w_1 = 3$) and 4 bit-serial multipliers ($w_2 = 7$) are used. Here w_1 and w_2 denote the equivalent digit-size of this digit-serial inverter and multiplier, respectively. When choosing $m = 83$, one inversion takes $\lceil \frac{2m-1}{w_1} \rceil = 55$ clock cycles, while one multiplication takes $\lceil \frac{m}{w_2} \rceil = 14$ clock cycles.

Given a constant w_2, increasing w_1 will reduce the number of clock cycles required by one inversion. However, it will increase the area as well as the critical path delay. As a result, the multiplication will be slowed down slightly. Therefore, w_1/w_2 can be chosen for different design targets such as high performance, low hardware cost or smallest area-time product. Theoretical exploration for optimal (w_1, w_2) for a specific design target is out of the scope of this paper. Table 2 shows

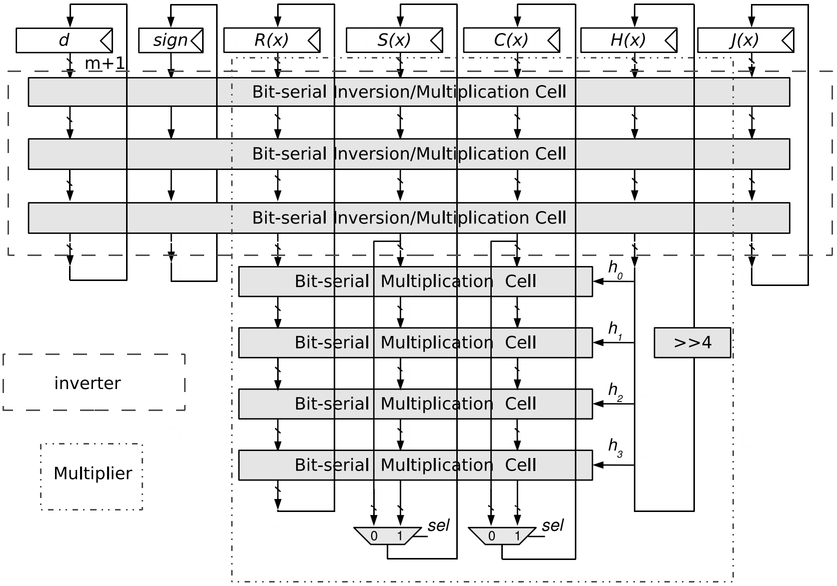

d
sign
R(x)
S(x)
C(x)
H(x)
J(x)
m+1
Bit-serial Inversion/Multiplication Cell
Bit-serial Inversion/Multiplication Cell
Bit-serial Inversion/Multiplication Cell
Bit-serial Multiplication Cell
Bit-serial Multiplication Cell
Bit-serial Multiplication Cell
Bit-serial Multiplication Cell
h0
h1
h2
h3
>>4
inverter
Multiplier
0 1 sel
0 1 sel

Fig. 3. Digit-serial modular multiplication inversion unit ($w_1 = 3, w_2 = 7$)

Table 2. Performance comparison of multiplication and inversion unit in GF(2^m)

Ref. Design	Configuration	Area [Slices]	Freq. [MHz]	Finite Field	Mul. Perf. [ns]/[#cycle]	Inv. Perf. [ns]/[#cycle]
Fig. 3	$w_1 = 1, w_2 = 14$	977	127	GF(2^{83})	47.1 / 6	1296 / 165
	$w_1 = 2, w_2 = 14$	1117	126	GF(2^{83})	47.3 / 6	654 / 83
	$w_1 = 3, w_2 = 14$	1500	125	GF(2^{83})	47.9 / 6	439 / 55
	$w_1 = 4, w_2 = 14$	1718	113	GF(2^{83})	52.7 / 6	372 / 42
	$w_1 = 5, w_2 = 14$	1987	104	GF(2^{83})	57.4 / 6	315 / 33
Mult. [31]	$w = 8$	342	108.7	GF(2^{81})	101 / 11	-
	$w = 16$	554	87.5	GF(2^{81})	69 / 6	-
	$w = 27$	882	71.0	GF(2^{81})	42 / 3	-
Inv. [31]	MAIA	663	87.8	GF(2^{81})	-	1014 / 89

the performance and area of the proposed ALU with different configurations. Here Xilinx Virtex II (XC2V4000) FPGA is used. In this HECC implementation we choose $w_1 = 3$ and $w_2 = 14$ as the best performance/area trade-off for this architecture. With this configuration, one multiplication and one inversion in GF(2^{83}) take 47.9 and 439 ns, respectively.

4.4 Memory/RF Analysis

Besides ALU, memory/RF is another main component that decides the overall area and performance of a coprocessor. The size, throughput and delay of memory/RF must be chosen according to the requirement of the ALU. We analyze different design strategies of HECC coprocessor here.

Both memory and RF have their own advantages and disadvantages. While registers are larger than memory of the same capacity, memory usually has one clock delay in read operation. This delay may cause performance degradation when multiple data-path work in a pipelining mode, see [31]. Thus, HECC coprocessors using multiple data-path [7,33,31] require an efficient register file to feed data to parallel multipliers and inverters. The register file and its interconnecting network make a big part of the whole area.

The area of memory/RF is dependent on the size and throughput [24,29]. Higher throughput results in a more complex decoder and a larger interconnecting network, which cause the area increase. Thus, reducing the memory/RF throughput reduces the area. Table 3 shows the required memory/RF throughput of different ALUs. Note that here we use GF(2^{83}) for all the ALUs, D denotes the delay of multiplication. For example, when using three multipliers, the ALU reads 6 operands from memory/RF and writes 3 data back. In [33,31], 3 clock cycles are required for one multiplication. If each operand is 84-bit, then the ALU needs to read 168 bits in each clock cycle. The proposed multiplier/inverter shown in Figure 3 requires 56-bit read and 14-bit write in each clock cycles. The required memory throughput is much smaller than that in [33] and [31].

Table 3. Comparison of memory throughput required by different ALUs

Ref. Design	Configuration	Read [Bits]	Write [Bits]	Total [Bits]
[33]	3 Mult. ($D = 3$)	168	84	252
[31]	2 Mult. ($D = 3$)	112	56	168
Fig.4	Unified M/I. ($D = 6$)	56	14	80

4.5 Coprocessor Architecture

The HECC Coprocessor is shown in Figure 4. It contains an Instruction ROM, a main controller and a unified modular multiplier/inverter. The Instruction ROM contains the field operation sequences of divisor addition and doubling. As only a single data-path is used, the coprocessor does not require high-throughput register files. Instead, a data RAM is used to keep the curve parameters, base divisor and intermediate data. On FPGAs, Block RAMs are used.

The coprocessor supports four instructions, namely,

```
Add Ra,Rb,Rc            // Ra=Rb+Rc
Mul Ra,Rb,Rc            // Ra=Rb*Rc
Mac Ra,Rb,Rc,Rd,Re      // Ra=Rb*Rc+Rd+Re
Inv Ra,Rb               // Ra=Rb^{-1}
```

Here one `Add` instruction takes two cycles. As $w_1 = 3$, one `Inv` instruction takes 55 clock cycles. One `Mul` instruction takes 6 clock cycles. One `Mac` instruction consists of one `Mul` and two `Add` instructions. However, it takes also 6 clock cycles. This is because fetching and adding data `Rd` and `Re` are performed during the multiplication. Two `Add` and one `Mul` instructions cause 6 operand fetches and 3 result stores, while one `Mac` instruction requires only 4 operand fetches and one result store. Therefore, the use of `Mac` instruction reduces the number of memory access and speeds up the scalar multiplication.

In this implementation, we choose hyperelliptic curves with the following parameters: $h(x) = x$ and $f(x) = x^5 + f_3x^3 + x^2 + f_0$. One DA operation consists of 36 instructions, which include 11 `Add`, 24 `Mac` and 1 `Inv` instructions. One DD operation consists of 14 instructions, which includes 2 `Add`, 11 `Mac` and 1 `Inv` instructions.

Note that the architecture of the coprocessor can be slightly modified so that it can be integrated into a SoC where memory is shared. The required throughput of memory needs to be further reduced. In the InsRom `Mac` instruction needs to be replaced by a `Mul` and two `Add` instructions, thus only two instead of four operands need to be loaded for each instruction. In this case, the required throughput of memory is $\frac{2*84}{6} = 28$ bits, the amount that a 32-bit dual-port SRAM is able to offer. However, the `add` instruction requires 6 instead of 2 clock cycles, which slightly degrades the performance of the coprocessor.

5 Implementation Results

In order to check the area and performance of the proposed coprocessor, we implemented the architecture from Figure 4 on a Xilinx Virtex-II (XC2V4000) FPGA. The coprocessor is described with Gezel [13] language and synthesized with Xilinx ISE8.1. It uses 2316 slices and 6 Block RAMs. A clock frequency of 125 MHz can be reached. Table 4 compares the area and performance with previous FPGA-based implementations of HECC in GF(2^m).

The proposed HECC coprocessor in [7] uses Cantor's method to perform divisor addition and doubling. It has two modular multipliers, one inverter, one GCD module and several other logics. Register file is connected to the datapath with MUX arrays. When supporting HECC in $GF(2^{83})$, it uses 22000 slices on Xilinx Virtex-II FPGA and can finish one scalar multiplication in 10 ms.

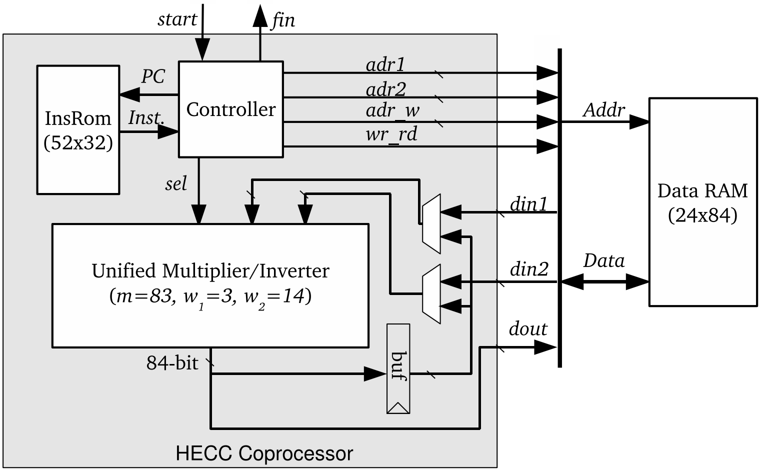


Fig. 4. Block diagram of the proposed HECC coprocessor

The proposed HECC coprocessor in [11] uses the mixed coordinates of explicit formulae proposed in [21]. The ALU contains three modules, namely divisor addition module, divisor doubling module and coordinates conversion module. Each of them has four field multipliers, while only the coordinates conversion module has a inverter. It supports Right-to-Left binary expansion method, which scans the key from LSB to MSB, and can perform divisor addition and doubling in parallel. It also supports NAF method. Here we list the performance of scalar multiplication using NAF method as it is slightly faster than the binary method.

The HECC coprocessor proposed in [17] uses projective coordinates, and a superscalar architecture is used to support parallel field operations. Several digit-serial ($w = 12$) multipliers are used. Our coprocessor, using one unified multiplier/inverter, is faster than the coprocessor in [17] that uses three multipliers.

Table 4. Performance comparison of FPGA-based HECC implementations in GF(2^m)

Ref. Design	FPGA	Freq. [MHz]	Area [Slices]	RAM [bits]	Finite Field	Irreducible Polynomial	Perf. [μs]	Comments
Clancy [7]	Xilinx Virtex-II	N/A	23000	0	GF(2^{83})	Arbitrary	10000	Two mult. One inv. Using NAF
Elias *et al.* [11]	Xilinx Virtex-II (XC2V8000)	45.3	25271	0	GF(2^{113})	Fixed	2030	12 mult. One inv. Using NAF
Sakiyama *et al.* [17]	Xilinx Virtex-II Pro (XC2VP30)	100	6586	8064	GF(2^{83})	Arbitrary	420	Three mult. Using NAF
			4749	5376	GF(2^{83})	Arbitrary	549	Two mult. Using NAF
			2446	2688	GF(2^{83})	Arbitrary	989	One mult. Using NAF
Wollinger [31]	Xilinx Virtex-II (XC2V4000)	56.7	7785	0	GF(2^{81})	Fixed	415	Three mult. Two inv.
		47.0	5604	0	GF(2^{81})	Fixed	724	Two mult. One inv.
		54.0	3955	1536	GF(2^{81})	Fixed	831	Two mult. One inv.
This work	Xilinx Virtex-II (XC2V4000)	125	2316	2016	GF(2^{83})	Fixed	311	Unified mult./inv. Using NAF

The architectures proposed in [31], however, uses affine coordinates of the explicit formulae. Three different architectures ranging from high speed to low hardware cost are proposed. For the high speed version, with three multipliers and two inverters, only 415 μs is required to finish one scalar multiplication. The area of the coprocessor is also much smaller than that of [7,11]. The area can be further reduced to 3955 slices but, in that case it requires 831 μs for one scalar multiplication.

Compared with all the previous FPGA-based implementations our implementation has the best performance, to the best of our knowledge. The area reduction is attributed to the use of compact ALU and the reduction of the memory throughput. The ALU in [31] contains two multipliers and one inverter, which in total use 2427 slices. The ALU used in this paper requires only 1500 slices. The performance gain is mainly due to the efficient inverter. When running at 56.7 MHz, the inverter in [31] requires 1570 ns in average for one inversion in GF(2^{81}), while the proposed ALU finishes one inversion in GF(2^{83}) in 439 ns. Though we use only one multiplier, which is also slower than the one in [31], the overall performance of divisor addition/doubling is better.

6 Conclusions

We describe a compact architecture for HECC over binary extension field. This architecture uses a unified modular multiplier/inverter, and reduces the throughput of the memory. Thus, the area of the coprocessor is largely reduced. On a Xilinx Virtex II (XC2V4000) FPGA, the proposed coprocessor takes 311 μs to finish one scalar multiplication in HECC over GF(2^{83}).

The proposed implementation can be further speeded up by exploring instruction level parallelism. Besides, if more space is available in the data memory, precomputation can be used to drastically improve the performance.

Acknowledgments

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by FWO projects G.0475.05, and G.0300.07, by the European Comission through the IST Programme under Contract IST-2002-507932 ECRYPT NoE, and by the K.U. Leuven-BOF.

References

1. Asano, Y., Itoh, T., Tsujii, S.: Generalised fast algorithm for computing multiplicative inverses in GF(2^m). Electronics Letters 25(10), 664–665 (1989)
2. Avanzi, R.M., Cohen, H., Doche, C., Frey, G., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. CRC Press, Boca Raton (2005)
3. Beth, T., Gollman, D.: Algorithm engineering for public key algorithms. IEEE Journal on Selected Areas in Communications 7(4), 458–466 (1989)
4. Boston, N., Clancy, T., Liow, Y., Webster, J.: Genus two hyperelliptic curve coprocessor. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 400–414. Springer, Heidelberg (2003)
5. Brent, R.P., Kung, H.T.: Systolic VLSI Arrays for Polynomial GCD Computation. IEEE Trans. Computers 33(8), 731–736 (1984)
6. Cantor, D.G.: Computing in the Jacobian of a Hyperelliptic curve. Mathematics of Computation 48, 95–101 (1987)
7. Clancy, T.: FPGA-based Hyperelliptic Curve Cryptosystems. invited paper presented at AMS Central Section Meeting (April 2003)
8. Daly, A., Marnane, W., Kerins, T., Popovici, E.: An FPGA implementation of a GF(p) ALU for encryption processors. Elsevier Journal on Microprocessors and Microsystems (Special issue on FPGAs: Applications and Designs) 28(5-6), 253–260 (2004)
9. Explicit-Formulas Database, `http://www.hyperelliptic.org/EFD`
10. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22, 644–654 (1976)
11. Elias, G., Miri, A., Yeap, T.H.: On efficient implementation of FPGA-based hyperelliptic curve cryptosystems. Computers and Electrical Engineering 33(5-6), 349–366 (2007)
12. Yeap, T.H., Elias, G., Miri, A.: High-Performance, FPGA-Based Hyperelliptic Curve Cryptosystems. In: The Proceeding of the 22nd Biennial Symposium on Communications (May 2004)
13. GEZEL, `http://rijndael.ece.vt.edu/gezel2/`
14. Guo, J.-H., Wang, C.-L.: A novel digit-serial systolic array for modular multiplication. In: ISCAS 1998. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems, ISCAS 1998, 31 May-3 Jun 1998, vol. 2,2, pp. 177–180 (1998)

15. Kim, H.W., Wollinger, T., Choi, Y., Chung, K.-I., Paar, C.: Hyperelliptic curve coprocessors on a FPGA. In: Lim, C.H., Yung, M. (eds.) WISA 2004. LNCS, vol. 3325, pp. 360–374. Springer, Heidelberg (2005)
16. Hasan, M.A., Bhargava, V.K.: Bit-serial systolic divider and multiplier for finite fields GF(2^m). IEEE Transactions on Computers 41(8), 972–980 (1992)
17. Preneel, B., Sakiyama, K., Batina, L., Verbauwhede, I.: Superscalar coprocessor for high-speed curve-based cryptography. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 415–429. Springer, Heidelberg (2006)
18. Knuth, D.E.: The Art of Computer Programming, vol. 2. Addison-Wesley, Reading (1981)
19. Koblitz, N.: Elliptic Curve Cryptosystem. Math. Comp. 48, 203–209 (1987)
20. Koblitz, N.: Hyperelliptic Cryptosystems. Journal of Cryptology 1(3), 129–150 (1989)
21. Lange, T.: Inversion-free arithmetic on genus 2 hyperelliptic curves. Cryptology ePrint ARchive (2002)
22. Lange, T.: Formulae for Arithmetic on Genus 2 Hyperelliptic Curves. Applicable Algebra in Engineering, Communication and Computing 15(5), 295–328 (2005)
23. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
24. Mulder, J.M., Quach, N.T., Flynn, M.J.: An area model for on-chip memories and its application. IEEE Journal of Solid-State Circuits 26(2), 98–106 (1991)
25. Pelzl, J.: Hyperelliptic Cryptosystems on Embedded Microprocessors. Master's thesis, Ruhr-Universitat Bochum (September 2002)
26. Rivest, R.L., Shamir, A., Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
27. Sakai, Y., Sakurai, K.: Design of hyperelliptic cryptosystems in small characteristic and a software implementation over F_{2^n}. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 80–94. Springer, Heidelberg (1998)
28. Sakiyama, K.: Secure Design Methodology and Implementation for Embedded Public-key Cryptosystems. PhD thesis, Katholieke Universiteit Leuven, Belgium (2007)
29. Shiue, W.-T.: Memory synthesis for low power ASIC design. In: ASIC 2002: Proceedings of 2002 IEEE Asia-Pacific Conference, pp. 335–342 (2002)
30. Song, L., Parhi, K.K.: Low-energy digit-serial/parallel finite field multipliers. J. VLSI Signal Process. Syst. 19(2), 149–166 (1998)
31. Wollinger, T.: Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems. PhD thesis, Ruhr-University Bochum, Germany (2004)
32. Wollinger, T.: Computer Architectures for Cryptosystems Based on Hyperelliptic Curves. Master's thesis, Worcester Polytechnic Institute, Worcester, Massachusetts (May 2001)
33. Wollinger, T., Bertoni, G., Breveglieri, L., Paar, C.: Performance of HECC Coprocessors Using Inversionfree Formulae. In: International Workshop on Information Security and Hiding, Singapore (ISH 2005), May 2005, pp. 1004–1012 (2005)
34. Yan, Z., Sarwate, D.V., Liu, Z.: High-speed systolic architectures for finite field inversion. Integration, VLSI Journal 38(3), 383–398 (2005)