Are standards compliant Elliptic Curve Cryptosystems feasible on RFID?

Sandeep S. Kumar and Christof Paar

Horst Görtz Institute for IT Security,
Ruhr-Universität Bochum, Germany

Abstract. With *elliptic curve cryptography* emerging as a serious alternative, the desired level of security can be attained with significantly smaller key sizes. This makes ECC very attractive for small-footprint devices with limited computational capability, memory and low-bandwidth network connections. However ECC is still considered to be impracticable for very low-end constrained devices like sensor networks and RFID tags. We present a stand-alone highly area optimized ECC processor design for standards compliant binary field curves. We use the fast squarer implementation to construct an addition chain that allows inversion to be computed efficiently. Hence, we propose an affine co-ordinate ASIC implementation of the ECC processor using a modified Montgomery point multiplication method for binary curves ranging from $113 - 193$ bits. An area between 10k and 18k gates on a 0.35um CMOS process is possible for the different curves which makes the design very attractive for enabling ECC in constrained devices.

Key Words: Elliptic curve cryptography (ECC), finite fields, Fermat's little theorem.

1 Introduction

Elliptic Curve Cryptography is a relatively new cryptosystem, suggested independently in 1986 by Miller [14] and Koblitz [11]. At present, ECC has been commercially accepted, and has also been adopted by many standardizing bodies such as ANSI [2], IEEE [8], ISO [9], SEC [1] and NIST [15]. A number of hardware implementations for standardized elliptic curve cryptography have been suggested in literature, but very few of them are aimed for low-end devices. Most implementations focus on speed and are mostly only suitable for server end applications due to their huge area requirements. A survey of different ECC implementations can be found in [3]. An Instruction Set Extension (ISE) based implementation as shown in [12] provides a simple solution if a processor is already available on the device. However, there is an equally important need for a stand-alone ECC engines in small constrained devices used for different

applications like sensor networks and RFID tags. This is normally dictated by the needs for better performance required by a communication protocol or energy constraints (as a stand-alone engine can be selectively switched off when not in use).

The different ECC processor implementations that have been suggested for such low-end applications [17, 16, 5] normally use non-standardized curves and hence are not acceptable for commercial applications. Standards compliant implementations are however very important for mass acceptance of a reliable public key infrastructure. The work in [18] presents an ECC implementation aimed for low-area for both $\mathbb{F}_{2^m}$ and $\mathbb{F}_p$ curves. The implementations use field sizes in the range of 191 to 256-bits for certain standardized curves.

In this work we try to find the limits of a low-area stand-alone public-key processor for standardized ECC curves. Therefore, we tradeoff flexibility in a design for a specific standardized binary field curve which is quite reasonable for constrained devices. We also note from previous implementations [18], that the memory requirements for storage of points and temporary variables can contribute substantially (more than 50%) to the overall size of the ECC processor. Hence we aim for algorithms that require lesser temporary variables even if it leads to a small computational drawback.

This paper is organized as follows: In Section 2, we give the proper choice and tweaks of the different algorithms that allow us to reduce area without drastically affecting the performance. Section 3 presents the implementation design for the different arithmetic units, memory unit and the overall processor design. Finally we analyze the area and performance in Section 4.

2 Mathematical Background

A detailed introduction to Elliptic Curve Cryptography and its implementation can be found in [7]. We present here only the algorithms specific for our implementation. Characteristic two fields $\mathbb{F}_{2^m}$ are often chosen for hardware realizations [3] as they are well suited for hardware implementation due to their "carry-free" arithmetic. This not only simplifies the architecture but reduces the area due to the lack of carry arithmetic.

An elliptic curve E over $\mathbb{F}_{2^m}$ is the set of solutions (x, y) which satisfy the simplified Weierstrass equation:

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

where $a, b \in \mathbb{F}_{2^m}$ and $b \neq 0$, together with the point at infinity $\mathcal{O}$. Due to the Weil Descent attack [6], m is chosen to be a prime. The point addition and doubling is then define in the underlying field $\mathbb{F}_{2^m}$ as follows: Let $P = (x_0, y_0) \in E(\mathbb{F}_{2^m})$ and $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$, where $Q \neq -P$.
Then $P + Q = (x_2, y_2)$, where

$$\begin{aligned} x_2 &= \lambda^2 + \lambda + x_0 + x_1 + a \\ y_2 &= \lambda(x_0 + x_2) + x_2 + y_0 \end{aligned} \tag{2}$$

and

$$\lambda = \begin{cases} \frac{y_0+y_1}{x_0+x_1} & \text{if } P \neq Q \\ \\ x_1 + \frac{y_1}{x_1} \text{ if } P = Q \end{cases} \tag{3}$$

The *standard polynomial basis* representation is used for our implementations with the reduction polynomial $F(x) = x^m + G(x) = x^m + \sum_{i=0}^{m-1} g_i x^i$ where $g_i \in \{0, 1\}$, for $i = 1, ..., m-1$ and $g_0 = 1$.
Let α be a root of $F(x)$, then we represent $A \in \mathbb{F}_{2^m}$ in polynomial basis as

$$A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i, \;\; a_i \in \mathbb{F}_2 \tag{4}$$

The field arithmetic is implemented as polynomial arithmetic modulo $F(x)$. Notice that by assumption $F(\alpha) = 0$ since α is a root of $F(x)$. Therefore,

$$\alpha^m = -G(\alpha) = \sum_{i=0}^{m-1} g_i \alpha^i \tag{5}$$

gives an easy way to perform modulo reduction whenever we encounter powers of α greater than $m-1$. Throughout the text, we will write A mod $F(\alpha)$ to mean *explicitly* the reduction step.

For the implementation of our stand-alone ECC processor, we use standardized binary fields that provide short-term security and also fields which are required for high security applications. The four different field sizes chosen for our implementation range from 113 to 193-bits, and are recommended in the standards SECG [1] and NIST [15]) (the different recommended curves and the reduction polynomials are shown in Table 1). For some of the constrained devices, short-term keys in 113-bit fields can provide the adequate security required for the application and therefore are a good option when area is extremely constrained.

Table 1. Standards recommended field sizes for $\mathbb{F}_{2^m}$ and reduction polynomial

Standard	**Field Size** (m)	**Reduction polynomial**
SECG	113	$F_{113}(x) = x^{113} + x^9 + 1$
SECG	131	$F_{131}(x) = x^{131} + x^8 + x^3 + x^2 + 1$
NIST, SECG	163	$F_{163}(x) = x^{163} + x^7 + x^6 + x^3 + 1$
SECG	193	$F_{193}(x) = x^{193} + x^{15} + 1$

A second reason for the use of the binary fields, is the simplified squaring structure, which is a central idea used in the algorithms chosen for the processor design.

2.1 Squaring

Squarings in $\mathbb{F}_{2^m}$ could be done in two steps, first an expansion with interleaved 0's as:

$$\begin{aligned} C &\equiv A^2 \bmod F(\alpha) \\ &\equiv (a_{m-1}\alpha^{2(m-1)} + a_{m-2}\alpha^{2(m-2)} + \ldots + a_1\alpha^2 + a_0) \bmod F(\alpha) \end{aligned} \tag{6}$$

and then reducing the double-sized result with the reduction polynomial using the equivalence in Eq. 5. However in hardware these two steps can be combined if the reduction polynomial has a small number of non-zero co-efficients (which is the case with the trinomial and pentanomial reduction polynomials as in Table 1). Hence, the squaring can be efficiently implemented to generate the result in *one single clock cycle* without huge area requirements. The implementation and the area costs are discussed in detail in the implementation section.

2.2 Inversion

A common observation is that performing point arithmetic in affine co-ordinates requires lesser number of temporary variables. This is a very good argument to help reduce the memory requirements. However, the disadvantage is that the point operations in the affine co-ordinates requires an inversion operation. Dedicated inversion units using binary Euclidean methods are themselves costly to implement and require extra storage variables. The other more simpler method to perform inversion is using the *Fermat's Little Theorem* [4]:

$$A^{-1} \equiv A^{2^m-2} = (A^{2^{m-1}-1})^2 \bmod F(x) \quad \text{for } A \in \mathbb{F}_{2^m}. \tag{7}$$

Since $2^m - 2 = 2^1 + 2^2 + \cdots + 2^{m-1}$, a straightforward way of performing this exponentiation would be a binary square-and-multiply as $A^{-1} = A^{2^1} \cdot A^{2^2} \cdots A^{2^{m-1}}$, requiring a total of $(m-2)$ multiplications and $(m-1)$ squarings. This is an extremely costly due to the large number of multiplications and hence can considerably slow down an implementation of an ECC point multiplication. Therefore, projective co-ordinates are chosen even for low-area implementations [18].

However, Itoh and Tsujii proposed in [10], a construction of an addition chain such that the exponentiation could be performed in $O(log_m)$ multiplications. Though the algorithm was proposed for optimal normal basis implementations where squarings are almost for free (cyclic rotations), the area requirements for the squaring structure in our implementation is within bounds but has the same timing efficiency of 1-clock cycle as in the normal basis.

We first present here the addition chain construction and discuss the arithmetic costs for the different fields that we use. Representing $m-1$ in binary format, we can write

$$m - 1 = m_{q-1}2^{q-1} + m_{q-2}2^{q-2} + \cdots + m_1 2 + m_0, \; m_{q-1} = 1 \tag{8}$$

where $m_i \in 0, 1$ and $q = \lfloor log_2(m-1) \rfloor + 1$, the bit-length of $m-1$. We can then represent $m-1$ as a bit vector: $[m_{q-1}m_{q-2}\cdots m_1 m_0]_2$.
The Itoh-Tsujii method is based on the idea that we can represent

$$\begin{aligned} 2^{m-1} - 1 &= 2^{[m_{q-1}m_{q-2}\cdots m_1 m_0]_2} - 1 \\ &= 2^{m_0}(2^{2\cdot[m_{q-1}m_{q-2}\cdots m_1]_2} - 1) + 2^{m_0} - 1 \\ &= 2^{m_0}(2^{[m_{q-1}m_{q-2}\cdots m_1]_2} - 1) \cdot (2^{[m_{q-1}m_{q-2}\cdots m_1]_2} + 1) + m_0 \end{aligned} \tag{9}$$

If we define

$$T_i = (2^{[m_{q-1}m_{q-2}\cdots m_i]_2} - 1), \tag{10}$$

then a recursive equation can be constructed as follows

$$T_i = \begin{cases} 2^{m_i} T_{i+1} \cdot (2^{[m_{q-1}m_{q-2}\cdots m_{i+1}]_2} + 1) + m_i & \text{for } \; 0 \le i \le (q-2) \\ 1 & i = q-1 \end{cases}$$

The exponentiation $A^{2^{m-1}-1} = A^{T_0}$ can then be shown as an recursive operation:

$$\begin{aligned} A^{T_i} &= A^{2^{m_i} T_{i+1} \cdot (2^{[m_{q-1}m_{q-2}\cdots m_{i+1}]_2} + 1) + m_i} \\ &= \{(A^{T_{i+1}})^{2^{[m_{q-1}m_{q-2}\cdots m_{i+1}]_2}} \cdot (A^{T_{i+1}})\}^{2^{m_i}} \cdot A^{m_i} \quad \text{for } \; 0 \le i \le (q-2) \end{aligned}$$

Thus each recursive step requires $[1m_{q-2} \cdots m_{i+1}]_2$ squarings + 1 multiplication, and if $m_i = 1$, an additional squaring and multiplication. It can be easily shown that the inverse A^{-1} can then be obtained in $(\lfloor log_2(m-1) \rfloor + H_w(m-1) - 1)$ multiplications and $(m-1)$ squarings using this addition chain, where $H_w(.)$ denotes the Hamming weight of the binary representation.

Algorithm 4, shows the steps involved for calculation of the inverse for the field size $163 = [10100011]_2$. As shown, an inverse in $\mathbb{F}_{2^{163}}$ requires 9 field multiplications and 162 squaring, and one extra variable (denoted as T here) for the temporary storage (which is blocked only during the inversion process and hence can be used later as a temporary variable in the point multiplication algorithm). As we already mentioned, a squaring can be computed in a single clock cycle, and hence the overall cost for inverse is approximately 10 multiplications (assuming multiplication takes 163 clock cycles). Hence, we take the different approach of using the affine co-ordinates for our implementation of the ECC processor. Similar addition chains can be achieved for the other field sizes. We give here only the costs in terms of the field multiplications and squarings for each in the Table 2

Table 2. $\mathbb{F}_{2^m}$ inversion cost using Itoh-Tsuji method

Field size (m)	**Cost**
113	8 $\mathbb{F}_{2^{113}}$ **M** + 112 $\mathbb{F}_{2^{113}}$ **S**
131	8 $\mathbb{F}_{2^{131}}$ **M** + 130 $\mathbb{F}_{2^{131}}$ **S**
163	9 $\mathbb{F}_{2^{163}}$ **M** + 162 $\mathbb{F}_{2^{163}}$ **S**
193	9 $\mathbb{F}_{2^{193}}$ **M** + 192 $\mathbb{F}_{2^{193}}$ **S**

2.3 Point multiplication

Montgomery point multiplication is a very efficient algorithm which is used widely because of the computational savings it gives in projective co-ordinates. It also has the added advantage that, it computes only over the x co-ordinates in each iteration and hence requires lesser storage area. It is based on the fact that a running difference $P = (x, y) = P_1 - P_2$ can be used to derive the x co-ordinate of $P_1 + P_2 = (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

as:

$$x_3 = \begin{cases} x + (\frac{x_1}{x_1+x_2})^2 + \frac{x_1}{x_1+x_2} \text{ if } P_1 \neq P_2 \\ \\ x_1^2 + \frac{b}{x_1^2} \qquad\qquad\qquad \text{if } P_1 = P_2 \end{cases} \tag{11}$$

In affine co-ordinates, the Montgomery algorithm as shown in Algorithm 2.3, has the disadvantage that it requires two inversions to be computed in each iteration.

Algorithm 1 Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ in affine co-ordinates [13]

Input: P, k, where $P = (x, y) \in E(\mathbb{F}_{2^m}), k = [k_{l-1} \cdots k_1 k_0]_2 \in \mathbf{Z}^+$ and $\log_2 k < m$

Output: $Q = k \cdot P$, where $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$

1: if $k = 0$ or $x = 0$ then Return$Q = (0, 0)$ and stop.
2: Set $x_1 \leftarrow x$, $x_2 \leftarrow x^2 + b/x^2$.
3: **for** $i = l - 2$ downto 0 **do**
4: Set $t \leftarrow \frac{x_1}{x_1+x_2}$.
5: **if** $k_i = 1$ **then**
6: $x_1 \leftarrow x + t^2 + t, \quad x_2 \leftarrow x_2^2 + \frac{b}{x_2^2}$.
7: **else**
8: $x_2 \leftarrow x + t^2 + t, \quad x_1 \leftarrow x_1^2 + \frac{b}{x_1^2}$.
9: **end if**
10: **end for**
11: $r_1 \leftarrow x_1 + x$, $r_2 \leftarrow x_2 + x$
12: $y_1 \leftarrow r_1(r_1 r_2 + x^2 + y)/x + y$
13: Return $(Q = (x_1, y_1)$

The overall cost of the point multiplication using this algorithms is:

$$\sharp INV. = 2\lfloor log_2 k \rfloor + 2, \quad \sharp MUL. = 2\lfloor log_2 k \rfloor + 4,$$
$$\sharp ADD. = 4\lfloor log_2 k \rfloor + 6, \quad \sharp SQR. = 2\lfloor log_2 k \rfloor + 2.$$

We can reduce the inversions required by performing a simultaneous inversion. For $a_1, a_2 \in \mathbb{F}_{2^m}$ and non-zero, we first compute the product $A = a_1 \cdot a_2 \bmod F(\alpha)$ and perform a single inversion $A^{-1} \bmod F(\alpha)$. Then the individual inverses are obtained by the two multiplication $a_1^{-1} =$

$A^{-1} \cdot a_2 \bmod F(\alpha)$ and $a_2^{-1} = A^{-1} \cdot a_1 \bmod F(\alpha)$. Hence, we can trade-off one inversion for three extra multiplications. From Table 2, we know that inversions for our implementation are more costly than 3 multiplications and therefore a simultaneous inversion always gives a better performance.

There is however, the cost for one extra memory location to temporarily save the product A during the computation of the inverse (apart from the temporary location T). We use two different options here: a) we allocate an extra memory location (denoted as R here) to store the temporary variable A, and b) the product A is computed each time it is required during the inverse computation.

Based on the discussion on the inverse computation and as seen in Algorithm 4, the value of A is required at $H(m-1)$ different steps in the inverse operation and hence has to be recomputed $H(m-1)-1$ times (since the computation at the beginning would have to be done anyways). This is quite low and if area is the main constraint, replacing A with extra multiplications would not drastically affect performance. Hence, the steps in the computation of the inverse in Algorithm 4:

$T \leftarrow B \cdot A$;

are replaced with the computational sequence:

$T \leftarrow a_1 \cdot a_2$; $\{T = A\}$

$T \leftarrow B \cdot T$;

As we mentioned, simultaneous inversion requires three extra multiplications to trade-off one inversion. However, for the Montgomery point multiplication algorithm, we can reduce this to just two multiplications based on the observation that the x co-ordinate of $P_1 - P_2$ (as in Eq. 11) can as well be replaced with x co-ordinates of $P_2 - P_1$ as shown here:

$$x_3 = \begin{cases} x + (\frac{x_2}{x_1+x_2})^2 + \frac{x_2}{x_1+x_2} \text{ if } P_1 \neq P_2 \\ \\ x_1^2 + \frac{b}{x_1^2} \qquad\qquad\qquad \text{if } P_1 = P_2 \end{cases} \tag{12}$$

The modified Montgomery algorithm is as shown in Algorithm 2. We now require one inversion and four multiplications in each iteration. The total cost of the $\mathbb{F}_{2^m}$ point multiplication using this algorithms is:

$$\sharp INV. = \lfloor log_2 k \rfloor + 2, \quad \sharp MUL. = 4\lfloor log_2 k \rfloor + 4,$$
$$\sharp ADD. = 4\lfloor log_2 k \rfloor + 6, \quad \sharp SQR. = 3\lfloor log_2 k \rfloor + 2.$$

The algorithm also allows us to compute each iteration without the need for any extra temporary memory locations (apart from the memory location T for inversion, and based on the implementation option the memory location R)

Algorithm 2 Modified Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ in affine co-ordinates

Input: P, k, where $P = (x, y) \in E(\mathbb{F}_{2^m}), k = [k_{l-1} \cdots k_1 k_0]_2 \in \mathbf{Z}^+$ and $\log_2 k < m$

Output: $Q = k \cdot P$, where $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$

1: if $k = 0$ or $x = 0$ then Return$Q = (0, 0)$ and stop.
2: Set $x_1 \leftarrow x$, $x_2 \leftarrow x^2 + b/x^2$.
3: **for** $i = l - 2$ downto 0 **do**
4: Set $r_0 \leftarrow x_1 + x_2$.
5: **if** $k_i = 1$ **then**
6: $R = \frac{1}{(x_1+x_2)\cdot x_2}$
7: $x_1 \leftarrow x + (x_2^2 \cdot R)^2 + (x_2^2 \cdot R), \quad x_2 \leftarrow x_2^2 + b \cdot (r_0 \cdot R)^2$.
8: **else**
9: $R = \frac{1}{(x_1+x_2)\cdot x_1}$
10: $x_2 \leftarrow x + (x_1^2 \cdot R)^2 + (x_1^2 \cdot R), \quad x_1 \leftarrow x_1^2 + b \cdot (r_0 \cdot R)^2$.
11: **end if**
12: **end for**
13: $r_1 \leftarrow x_1 + x$, $r_2 \leftarrow x_2 + x$
14: $y_1 \leftarrow r_1(r_1 r_2 + x^2 + y)/x + y$
15: Return $(Q = (x_1, y_1)$

3 Implementation Aspects

Based on the mathematical analysis, the main units that are required for the ECC processor are the adder, multiplier and squaring units in $\mathbb{F}_{2^m}$.

3.1 $\mathbb{F}_{2^m}$ Adder Unit

Addition is a simple bit wise XOR operation implemented using XOR gates. Therefore, a $\mathbb{F}_{2^m}$ addition is implemented in our design using m XOR gates with the output latency of 1 clock cycle.

3.2 $\mathbb{F}_{2^m}$ Multiplier Unit

Multipliers are normally the next biggest component in an ECC processor and therefore the appropriate multiplier design needs to be chosen

based on the implementation goals (speed or area). When implementing for constrained devices, which requires extreme savings in area, bit-serial multipliers are the most efficient that reduce area and maintain good performance. We implement the Most-Significant Bit-serial (MSB) multiplier (Algorithm 3).

Algorithm 3 Shift-and-Add Most Significant Bit $\mathbb{F}_{2^m}$ multiplication

Input: $A = \sum_{i=0}^{m-1} a_i \alpha^i$, $B = \sum_{i=0}^{m-1} b_i \alpha^i$ where $a_i, b_i \in \mathbb{F}_2$.
Output: $C \equiv A \cdot B \bmod F(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i$ where $c_i \in \mathbb{F}_2$.

1: $C \leftarrow 0$
2: **for** $i = m-1$ downto 0 **do**
3: $\quad C \leftarrow b_i \cdot (\sum_{i=0}^{m-1} a_i \alpha^i) + (\sum_{i=0}^{m-1} c_i \alpha^i) \cdot \alpha \bmod F(\alpha)$
4: **end for**
5: Return (C)

The structure of the 163-bit MSB multiplier is as shown Fig 1. Here,

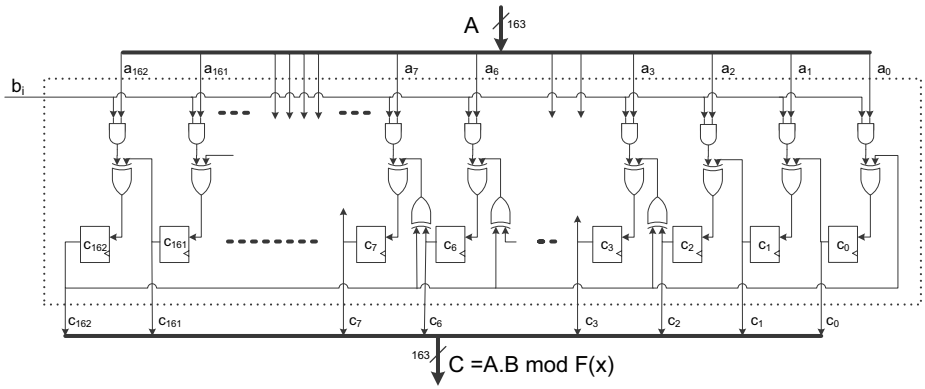

Fig. 1. $\mathbb{F}_{2^{163}}$ Most Significant Bit-serial (MSB) Multiplier circuit

the operand A can be enabled onto the data-bus A of the multiplier, directly from the memory register location. The individual bits of b_i can be sent from a memory location by implementing the memory registers as a cyclic shift-register (with the output at the most-significant bit). The value of the operand register remains unchanged after the completion of the multiplication as it makes one complete rotation.

The reduction within the multiplier is now performed on the accumulating result c_i, as in step 4 in Algorithm 3. The taps that are feeded back to c_i, are based on the reduction polynomial. In Fig 1, which shows the implementation for $F_{163}(x)$ reduction polynomial, the taps XOR the result of c_{162} to c_7, c_6 c_3 and c_0.

The complexity of the multiplier is n AND + $(n+t-1)$ XOR gates and n FF where $t = 3$ for a trinomial reduction polynomial ($F_{113}(x)$ and $F_{193}(x)$) and $t = 5$ for a pentanomial reduction polynomial ($F_{131}(x)$ and $F_{163}(x)$). The latency for the multiplier output is n clock cycles. The maximum critical path is $2\Delta_{XOR}$ (independent of n) where, Δ_{XOR} represents the delay in an XOR gate.

3.3 $\mathbb{F}_{2^m}$ Squarer Unit

As mentioned previously, we can implement a very fast squarer with a latency of a single clock cycle. Squaring involves first the expansion by interleaving with 0's, which in hardware is just an interleaving of 0 bit valued lines on to the bus to expand it to $2n$ bits. The reduction of this polynomial is inexpensive, first, due to the fact that reduction polynomial used is a trinomial or pentanomial, and secondly, the polynomial being reduced is sparse with no reduction required for $\lfloor n/2 \rfloor$ of the higher order bits (since they have been set to 0's). The squarer is implemented as a hard wired XOR circuit as shown in Fig. 2. The XOR requirements and the maximum critical path (assuming an XOR tree implementation) for the four reduction polynomials used are given in the Table 3.

Table 3. $\mathbb{F}_{2^m}$ Squaring unit requirements

Reduction Polynomial	**XOR gates**	**Critical Path**
$x^{113} + x^9 + 1$	56 XOR	2 Δ_{XOR}
$x^{131} + x^8 + x^3 + x^2 + 1$	205 XOR	3 Δ_{XOR}
$x^{163} + x^7 + x^6 + x^3 + 1$	246 XOR	3 Δ_{XOR}
$x^{193} + x^15 + 1$	96 XOR	2 Δ_{XOR}

3.4 ECC Processor design

The three units: $\mathbb{F}_{2^m}$ addition (ADD), $\mathbb{F}_{2^m}$ multiplication (MUL), and $\mathbb{F}_{2^m}$ squaring (SQR) are closely interconnected inside a single *Arithmetic Unit*

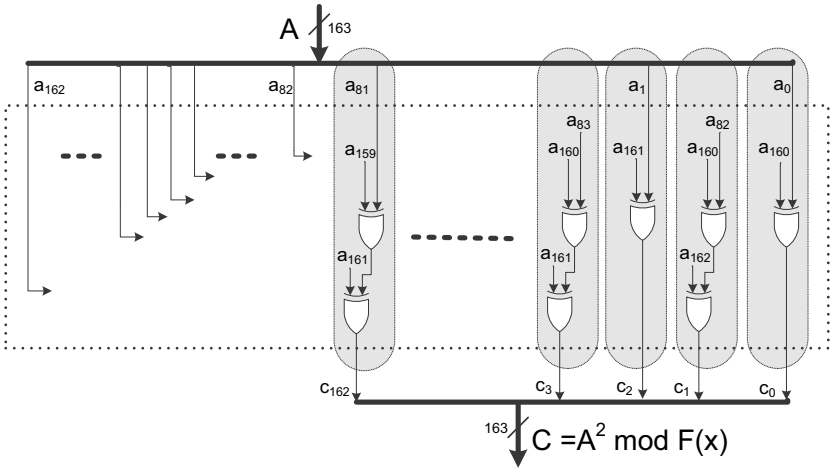

Fig. 2. $\mathbb{F}_{2^{163}}$ squaring circuit

(as shown in Fig. 3) sharing the common input data-bus A. The appropriate result is selected at the output data-bus C by the *Controller* signal C_{sel}. The adder needs an additional data-bus B for the second operand and the multiplier requires a single bit b_i signal for the multiplicand. The operands are stored in the *Memory* as registers (some of them as cyclic registers) with the output being selected for A, B and b_i using multiplexors with control signals (A_{sel}, B_{sel} and b_{i_sel}) from the *Controller*. All the operand register are connected in parallel to the data-bus C, with the appropriate register being loaded based on the *Controller* load signal C_{ld_reg}.

Inversion is done as mentioned using the Itoh-Tsuji method and thus requires no additional hardware apart from the multiplier and squarer unit with some additional control circuitry to enable the proper variables to the appropriate unit. We define a single INV instruction which performs the required sequence of steps. Since the inversion can be computed with $(\lfloor log_2(m-1) \rfloor + H_w(m-1) - 1)$ multiplications and $(m-1)$ squaring, the latency of the inversion in clock cycles is:
$(\lfloor log_2(m-1) \rfloor + H_w(m-1) - 1) \cdot m + (m-1) = (\lfloor log_2(m-1) \rfloor + H_w(m-1)) \cdot m - 1$

The instruction set of the processor is as shown in Table 4. The *Controller* executes the required instructions by enabling the appropriate control signals to the *Arithmetic Unit* and *Memory*. The scalar k is stored within the *Controller* as a shift register. The input operands x,y, and k

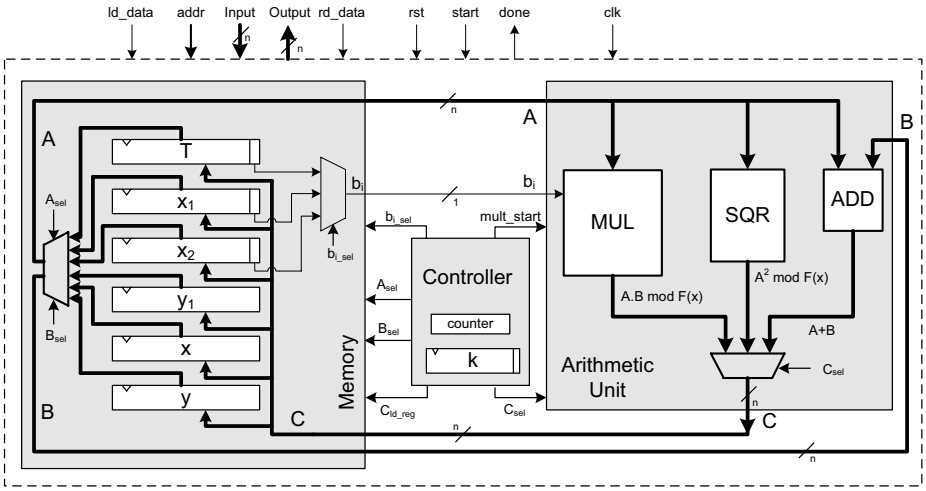

Fig. 3. Area optimized $\mathbb{F}_{2^n}$ ECC processor

are loaded externally on an n bit data-bus using the *addr* signal. The final results x_1 and y_1 is similarly read out using the *addr* signal.

Table 4. Controller Commands of ECC Processor over $\mathbb{F}_{2^m}$

Command	**Action**
LOAD $[addr_A]$	Load data into the register $[A]$ (in parallel)
READ $[addr_A]$	Read data out of the register $[A]$ (in parallel)
MUL $[addr_A]$, $[addr_B]$, $[addr_C]$	Perform $\mathbb{F}_{2^m}$ multiplication on $[A]$ (loaded in parallel), $[B]$ (loaded in serially) and stores the result in $[C]$ (in parallel)
ADD $[addr_A]$ $[addr_B]$ $[addr_C]$	Perform $\mathbb{F}_{2^m}$ addition on $[A]$, $[B]$ (both loaded in parallel) and stores the result in $[C]$ (in parallel)
SQR $[addr_A]$ $[addr_C]$	Perform $\mathbb{F}_{2^m}$ squaring on $[A]$ (loaded in parallel) and stores the result in $[C]$
INV $[addr_A]$ $[addr_C]$	Perform $\mathbb{F}_{2^m}$ inversion on $[A]$ (loaded in parallel) and stores the result in $[C]$ (in parallel)

The next important aspect of the design was to find the optimum sequence of computation steps such that the least number of temporary

memory is required. Another requirement during this optimization process, was to make sure that not all memory variables be connected to the data-bus B and b_i signal. This reduces the area requirement for the selection logic that is implemented with multiplexors and additionally, the fact that memory variables which are implemented without a cyclic shift requires much lesser area.

We use the modified Algorithm 2 to construct the sequence of instructions as shown in Algorithm 5. As mentioned before we can get rid of the extra register R by performing additional multiplications and hence Fig. 3 is shown without this register. The processor requires only 6 memory locations, the inputs x, y , the outputs x_1, x_2, and registers x_2 and T. No other extra temporary memory registers are needed during the computation. Only the registers x_1, x_2 and T are connected to the b_i signal and hence the others are implemented as simple registers with no shift operation. Similarly, only the register x_1 and x_2 are connected to the data-bus B. Data-bus A is connected to all the memory locations.

4 Performance Analysis

Based on the implementation described in the last section, we build two ECC processors: a) with the extra register R (not a cyclic shift register) and hence fewer multiplications and b) without the extra register R but smaller area. The implementations were synthesized for a custom ASIC design using AMI Semiconductor $0.35\mu m$ CMOS technology using the Synopsys Design Compiler tools. Timing measurements were done with Modelsim simulator against test vectors generated with a Java based model.

The area requirements (in terms of equivalent gate counts) for individual units of both the processor implementations is shown in the Table 5. The *Controller* area differs only slightly between the implementations and we give here only for the case without the register R. The area requirements for the *Memory* is given in Table 6 (without the extra R register) and Table 7 (with the extra R register). As can be seen, memory requirements are more than 50% of the whole design. The designs have a total area ranging from 10k equivalent gates for 113-bit field for short-term security, to 18k for 193-bit field for high security applications.
The latency of the ECC processor in clock cycles for a single scalar multiplication is shown in Tables 6 and 7. Since the structure is extremely simple, it can be clocked at very high frequencies, but we present here the absolute timings at 13.56 Mhz which is normally the frequency used

Table 5. $\mathbb{F}_{2^m}$ ECC processor area (in equivalent gates)

Field size	**MULT**	**SQR**	**ADD**	**Controller**
113	1210.58	188.38	226.00	1567.51
131	1402.46	406.00	262.00	1833.23
163	1743.58	502.00	326.00	2335.61
193	2068.38	321.98	386.00	2957.83

in RFID applications. We also make a comparison of the efficiency between the two ECC processor using the area-time product (normalized to the implementation with register R) and shown in Table 6. We see that the ECC processor with the extra register has a better efficiency, and so should be the preferred implementation if a slight area increase is acceptable.

Table 6. $\mathbb{F}_{2^m}$ ECC processor performance @13.56 Mhz *without* extra register R

Field size	**Memory**	**Total Area**	**Cycles**	**Time** (ms)	**Area-Time**
113	6686.43	10112.85	195159	14.4	1.07
131	7747.17	11969.93	244192	18.0	0.99
163	9632.93	15094.31	430654	31.8	1.06
193	11400.83	17723.12	564919	41.7	1.00

Table 7. $\mathbb{F}_{2^m}$ ECC processor performance @13.56 Mhz *with* extra register R

Field size	**Memory**	**Total Area**	**Cycles**	**Time** (ms)
113	7439.01	10894.34	169169	12.5
131	8619.63	12883.51	226769	16.8
163	10718.49	16206.67	376864	27.9
193	12686.21	19048.22	527284	38.8

Comparing the results presented here to the work in [18] (the only standardized curve implementation for constrained devices), the clock cycles for our 193-bit size implementation is 19% more than for the 191-bit Montgomery projective co-ordinate algorithm, which was expected because we trade-off performance slightly to save on area. However, the

implementation is 4.4 times faster than the affine implementation presented. The area requirements of 18k gates for our 193-bit implementation is much smaller (22%) than the 23k gates for the 191-bit field mentioned in [18] (though the design is slightly more versatile due the dual field multipliers used). However, as mentioned before, not all constrained devices like RFID require such high security. We can choose smaller key sizes depending on the security requirements of the application for which the RFID is used. This allows for a much smaller area requirements as mentioned in this work. However power requirements are also an important consideration for the feasibility of the a cryptographic engine on an RFID device and we would be studying them in our future work.

5 Summary

We showed here an extremely small area implementation of an ECC processor in the affine co-ordinates. Though affine co-ordinate implementations are not normally favored for constrained devices due to the need for an inverter, we show through the use of an addition chain and fast squarer, they are equally good for use in low-area implementations. Further savings are possible at the point arithmetic level by tweaking the algorithm to save on temporary storage variables. The proposed ECC processor is also secure against timing attacks due to the regular structure of the instructions used independent of the scalar. Hence, we show that an ECC processor implementation for four different standardized binary fields ranging from 113 to 193 bits with area requirements ranging between 10k and 18k gates, which makes the design very attractive for enabling ECC in constrained devices. Thus to the answer "Are standards compliant Elliptic Curve Cryptosystems feasible on RFID?", it depends on the security requirements of the application and could as well be feasible for lower security applications and within reach for high security applications in terms of gate count. Accurate power requirements of the processor would be the major factor that would affect the final decision and would require more research in the future.

References

1. *SEC 2. Standards for Efficient Cryptography Group: Recommended Elliptic Curve Domain Parameters. Version 1.0*, 2000.
2. American National Standards Institute, New York, USA. *ANSI X9.62: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.

3. L. Batina, S. B. Ors, B. Prenee, and J. Vandewalle. Hardware architectures for public key cryptography. *Integration, the VLSI journal*, 34(6):1–64, 2003.
4. D. M. Burton. *Elementary Number Theory.* Allyn and Bacon, 1980.
5. J. W. Chung, S. G. Sim, and P. J. Lee. Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *LNCS*, pages 57–70, Berlin, Germany, August 2000. Springer-Verlag Inc.
6. P. Gaudry, F. Hess, and N. P. Smart. Constructive and destructive facets of weil descent on elliptic curves. *Journal of Cryptology*, 15(1):19–46, 2002.
7. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag Inc., Berlin, Germany, 2004.
8. IEEE Computer Society Press, Silver Spring, MD, USA. *IEEE P1363-2000: IEEE Standard Specifications for Public-Key Cryptography*, 2000.
9. International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 15946: Information Technology — Security Techniques: Cryptographic Techniques based on Elliptic Curves*, 2002.
10. T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.
11. N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
12. S. S. Kumar and C. Paar. Reconfigurable instruction set extension for enabling ECC on an 8-bit processor. In J. Becker, M. Platzner, and S. Vernalde, editors, *14th International Conference Field Programmable Logic and Application — FPL 2004*, volume 3203 of *LNCS*, pages 586–595, Berlin, Germany, August 2004. Springer-Verlag Inc.
13. J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2 m) without precomputation. In Ç. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES '99*, volume 1717 of *LNCS*, pages 316–327, Berlin, Germany, August 1999. Springer-Verlag Inc.
14. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in cryptology — CRYPTO '85*, volume 218 of *LNCS*, pages 417–426, Berlin, Germany, August 1986. Springer-Verlag Inc.
15. National Institute for Standards and Technology, Gaithersburg, MD, USA. *FIPS 186-2: Digital Signature Standard (DSS). 186-2*, February 2000. Available for download at http://csrc.nist.gov/encryption.
16. E. Öztürk, B. Sunar, and E. Savas. Low-power elliptic curve cryptography using scaled modular arithmetic. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems—CHES 2004*, volume 3156 of *LNCS*, pages 92–106, Berlin, Germany, August 2004. Springer-Verlag Inc.
17. R. Schroeppel, C. L. Beaver, R. Gonzales, R. Miller, and T. Draelos. A low-power design for an elliptic curve digital signature chip. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *LNCS*, pages 366–380, Berlin, Germany, August 2002. Springer-Verlag Inc.
18. J. Wolkerstorfer. *Hardware Aspects of Elliptic Curve Cryptography.* PhD thesis, Graz University of Technology, Graz, Austria, 2004.

Appendix

Algorithm 4 Inversion with Itoh-Tsuji method over $\mathbb{F}_{2^{163}}$

Input: $A \in \mathbb{F}_{2^{163}}$ and irreducible polynomial $F(t)$.

Output: $B \equiv A^{-1} \bmod F(t) = A^{2^m-2}$ where $m = 163$.

1: $B \leftarrow A^2 = A^{(10)_2}$ { 1 SQR }

2: $T \leftarrow B \cdot A = A^{(11)_2}$ { 1 MUL }

3: $B \leftarrow T^{2^2} = A^{(1100)_2}$ { 2 SQR }

4: $T \leftarrow B \cdot T = A^{(1111)_2}$ { 1 MUL }

5: $B \leftarrow T^2 = A^{(11110)_2}$ { 1 SQR }

6: $T \leftarrow B \cdot A = A^{(11111)_2}$ { 1 MUL }

7: $B \leftarrow T^{2^5} = A^{(1111100000)_2}$ { 5 SQR }

8: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{10}\underbrace{0\cdots0}_{10})_2}$ { 1 MUL }

9: $B \leftarrow T^{2^{10}} = A^{(\underbrace{1\cdots1}_{10})_2}$ { 10 SQR }

10: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{20}\underbrace{0\cdots0}_{20})_2}$ { 1 MUL }

11: $B \leftarrow T^{2^{20}} = A^{(\underbrace{1\cdots1}_{20})_2}$ { 20 SQR }

12: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{40}\underbrace{0\cdots0}_{40})_2}$ { 1 MUL }

13: $B \leftarrow T^{2^{40}} = A^{(\underbrace{1\cdots1}_{40})_2}$ { 40 SQR }

14: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{80}0)_2}$ { 1 MUL }

15: $B \leftarrow T^2 = A^{(\underbrace{1\cdots1}_{80})_2}$ { 1 SQR }

16: $T \leftarrow B \cdot A = A^{(\underbrace{1\cdots1}_{81}\underbrace{0\cdots0}_{81})_2}$ { 1 MUL }

17: $B \leftarrow T^{2^{81}} = A^{(\underbrace{1\cdots1}_{81})_2}$ { 81 SQR }

18: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{162}0)_2}$ { 1 MUL }

19: $B \leftarrow T^2 = A^{\underbrace{1\cdots1}_{162}}$ { 1 SQR }

20: **Return** B

Algorithm 5 Instruction sequence for the Modified Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ in affine co-ordinates

Input: P, k, where $P = (x, y) \in E(\mathbb{F}_{2^m}), k = [k_{l-1} \cdots k_1 k_0]_2 \in \mathbf{Z}^+$ and $\log_2 k < m$

Output: $Q = k \cdot P$, where $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$

1: if $k = 0$ or $x = 0$ then Return$Q = (0, 0)$ and stop.
2: $x_1 \leftarrow x$,
3: $y_1 \leftarrow \text{SQR}\ (x)$
4: $x_2 \leftarrow \text{INV}\ (y_1)$.
5: $x_2 \leftarrow \text{MUL}\ (b, x_2)$.
6: $x_2 \leftarrow \text{ADD}\ (y_1, x_2)$.
7: **for** $i = l - 2$ downto 0 **do**
8: **if** $k_i = 1$ **then**
9: $x_1 \leftarrow \text{ADD}\ (x_1, x_2)$
10: $R \leftarrow \text{MUL}\ (x_1, x_2)$
11: $y_1 \leftarrow \text{INV}\ (R)$
12: $x_1 \leftarrow \text{MUL}\ (y_1, x_1)$
13: $x_2 \leftarrow \text{SQR}\ (x_2)$
14: $y_1 \leftarrow \text{MUL}\ (y_1, x_2)$
15: $x_1 \leftarrow \text{SQR}\ (x_1)$
16: $x_1 \leftarrow \text{MUL}\ (b\ , x_1)$
17: $x_2 \leftarrow \text{ADD}\ (x_1, x_2)$
18: $x_1 \leftarrow \text{SQR}\ (y_1)$
19: $x_1 \leftarrow \text{ADD}\ (y_1, x_1)$
20: $x_1 \leftarrow \text{ADD}\ (x, x_1)$
21: **else**
22: $x_2 \leftarrow \text{ADD}\ (x_1, x_2)$
23: $R \leftarrow \text{MUL}\ (x_1, x_2)$
24: $y_1 \leftarrow \text{INV}\ (R)$
25: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
26: $x_1 \leftarrow \text{SQR}\ (x_1)$
27: $y_1 \leftarrow \text{MUL}\ (y_1, x_1)$
28: $x_2 \leftarrow \text{SQR}\ (x_2)$
29: $x_2 \leftarrow \text{MUL}\ (b\ , x_2)$
30: $x_1 \leftarrow \text{ADD}\ (x_1, x_2)$
31: $x_2 \leftarrow \text{SQR}\ (y_1)$
32: $x_2 \leftarrow \text{ADD}\ (y_1, x_2)$
33: $x_2 \leftarrow \text{ADD}\ (x, x_2)$
34: **end if**
35: **end for**
36: $y_1 \leftarrow \text{ADD}\ (x, x_1)$
37: $x_2 \leftarrow \text{ADD}\ (x, x_2)$
38: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
39: $T \leftarrow \text{SQR}\ (x)$
40: $x_2 \leftarrow \text{ADD}\ (T\ , x_2)$
41: $x_2 \leftarrow \text{ADD}\ (y\ , x_2)$
42: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
43: $y_1 \leftarrow \text{INV}\ (x)$
44: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
45: $y_1 \leftarrow \text{ADD}\ (y, x_2)$
46: Return $(Q = (x_1, y_1)$