Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA

Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka

Fujitsu Laboratories Ltd.
64 Nishiwaki, Ohkubo-cho, Akashi 674-8555, Japan
{sokada,torii,kito,takenaka}@flab.fujitsu.co.jp

Abstract. We describe the implementation of an elliptic curve cryptographic (ECC) coprocessor over $GF(2^m)$ on an FPGA and also the result of simulations evaluating its LSI implementation. This coprocessor is suitable for server systems that require efficient ECC operations for various parameters. For speeding-up an elliptic scalar multiplication, we developed a novel configuration of a multiplier over $GF(2^m)$, which enables the multiplication of any bit length by using our data conversion method. The FPGA implementation of the coprocessor with our multiplier, operating at 3 MHz, takes 80 ms for 163-bit elliptic scalar multiplication on a pesudo-random curve and takes 45 ms on a Koblitz curve. The 0.25 μm ASIC implementation of the coprocessor, operating at 66 MHz and having a hardware size of 165 Kgates, would take 1.1 ms for 163-bit elliptic scalar multiplication on a pesudo-random curve and would take 0.65 ms on a Koblitz curve.
Keywords: Elliptic curve cryptography (ECC), coprocessor, elliptic scalar multiplication over $GF(2^m)$, IEEE P1363, Koblitz curve, multiplier.

1 Introduction

We describe the implementation of an elliptic curve cryptographic (ECC) [8] [13] [12] coprocessor over $GF(2^m)$ that is suitable for server systems. A cryptographic coprocessor for server systems must be flexible and provide a high-performance to process a large number of requests from various types of clients. A flexible coprocessor should be able to operate for various elliptic curve parameters. For example, it should be able to operate for arbitrary irreducible polynomials at any bit length. We therefore chose a polynomial basis (PB), because with reasonable hardware size it provides more flexibility than a normal basis (NB). And a high-performance coprocessor should perform fast elliptic scalar multiplication. The elliptic scalar multiplication is based on the multiplication over $GF(2^m)$. We therefore developed and implemented an efficient algorithm for bit parallel multiplication over $GF(2^m)$.

There have been many proposals regarding fast multipliers over $GF(2^m)$ [10] [7] [11]. A classical bit parallel multiplier made by piling up bit serial multipliers (each of which is known as a linear feedback shift register (LFSR) [10]) was

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 25–40, 2000.
© Springer-Verlag Berlin Heidelberg 2000

proposed by Laws [10] and improved by Im [7]. One of the fastest multipliers was proposed by Mastrovito [11] but it is extremely difficult to implement with reasonable hardware size if the irreducible polynomials and the bit length are not fixed.

There have also been studies concerned with the hardware implementation of an ECC over $GF(2^m)$ [1] [2] [16] [15] [5]. The hardware described in [1] and [2] is based on NB. And that described in [16] is based on composite field arithmetic on PB. To reduce the hardware size needed for implementation of the ECC on PB, some new multiplier have been proposed. The basic idea behind them is that an m-bit$\times m$-bit multiplication is calculated by a w_1-bit$\times w_2$-bit multiplier (where $w_1, w_2 \leq m$). Hasan proposed a look-up table based algorithm for $GF(2^m)$ multiplication [5]. This method uses an m-bit$\times w_2$-bit multiplier. And Orlando and Paar developed a new sliced PB multiplier, called a super serial multiplier (SSM) [15], which is based on the LFSR. The SSM uses a w_1-bit$\times$1-bit multiplier.

In this paper we describe a fast multiplier over $GF(2^m)$ that can operate for arbitrary irreducible polynomials at any bit length, and we also describe the implementation of an ECC coprocessor with this multiplier on an FPGA. Our multiplier has two special characteristics. Our multiplier architecture extends the concept of the SSM. That is, our multiplier folds the bit parallel multiplier, whereas the SSM folds the LFSR. Our multiplier is a w_1-bit$\times w_2$-bit multiplier (where $w_1 > w_2$, $w_1, w_2 \leq m$), which offers better performance when w_1 is larger or w_2 is smaller in case of fixed hardware size. Our multiplier also does fast multiplication at any bit length by using a new data conversion method, in which the data is converted, a sequence of multiplications is done, and the result is obtained by inverse conversion. This method enables fast operation when a sequence of multiplications is required, as in ECC calculation.

We implemented an ECC coprocessor with our multiplier on a field programmable gate array (FPGA), EPF10K250AGC599-2 by ALTERA[3]. Our coprocessor performs a fast elliptic scalar multiplication on a pseudo-random curve [14] and a Koblitz curve [14] [9] for various parameters. It operates at 3 MHz and includes an 82-bit$\times$4-bit multiplier. For 163-bit elliptic scalar multiplication, it takes 80 ms on a pseudo-random curve and 45 ms on a Koblitz curve. We also evaluated the performance and the hardware size of our coprocessor with 0.25 μm ASIC by FUJITSU [4]. Our coprocessor can operate at up to 66 MHz using a 288-bit$\times$8-bit multiplier and its hardware size is about 165 Kgates. For 163-bit elliptic scalar multiplication, it takes 1.1 ms on a pseudo-random curve and 0.65 ms on a Koblitz curve. And for 571-bit elliptic scalar multiplication, it takes 22 ms on a pseudo-random curve and 13 ms on a Koblitz curve.

We describe our multiplication algorithm in section 2, the configuration of our multiplier in section 3, and the ECC coprocessor implementation in section 4.

2 Multiplication Algorithm

2.1 Polynomial Representation

In this paper we represent elements over $GF(2^m)$ in three different types: **bit-string**, **word-string**, and **block-string**. An element a over $GF(2^m)$ is expressed as a polynomial of degree less than m. That is,

$$a(x) = \sum_{i=0}^{m-1} a_i x^i, (a_i \in GF(2)).$$

In the bit-string the element a is represented as $a = (a_{m-1}, a_{m-2}, \ldots, a_0)$.

In the word-string the element a is represented with words which have a w_2-bit length. We denote the i-th word as A_i, and it can be represented with a bit-string as $A_i = (a_{w_2 \cdot i+w_2-1}, a_{w_2 \cdot i+w_2-2}, \ldots, a_{w_2 \cdot i})$. When $m = n_2 \cdot w_2$, the element a can be represented as $a = (A_{n_2-1}, A_{n_2-2}, \ldots, A_0)$ and we can express the element a by using the following equations:

$$a(x) = \sum_{i=0}^{n_2-1} A_i(x) \cdot x^{w_2 \cdot i},$$

$$A_j(x) = \sum_{k=0}^{w_2-1} a_{w_2 \cdot j+k} \cdot x^k.$$

In the block-string the element a is represented with blocks, which are sequences of words. We denote the block $\boldsymbol{A}_{[i,j]} = (A_i, A_{i-1}, \ldots, A_j)$ (where $i \geq j$). When $m = n_1 \cdot w_1$ and $w_1 = s \cdot w_2$, we can express the element a by using the following equations:

$$a(x) = \sum_{i=0}^{n_1-1} \boldsymbol{A}_{[s \cdot i+s-1, s \cdot i]}(x) \cdot x^{w_1 \cdot i},$$

$$\boldsymbol{A}_{[s \cdot i+s-1, s \cdot i]} = \sum_{j=0}^{s-1} A_{s \cdot i+j}(x) \cdot x^{w_2 \cdot j}.$$

2.2 Irreducible Polynomial Representation

The irreducible polynomial $f(x)$ over $GF(2^m)$ can be represented as

$$f(x) = x^m + \sum_{i=0}^{m-1} f_i \cdot x^i, \ (f_i \in GF(2)).$$

And the lowest m-bit sequence of f_i is denoted as $f^* = (f_{m-1}, f_{m-2}, \ldots, f_0)$. In this paper we also call $f^*(x)$ an irreducible polynomial.

2.3 Partial Multiplication Algorithm

In this section, we describe the multiplication algorithm over $GF(2^m)$ with w_1-bit×w_2-bit partial multiplications. The multiplication with m-bit×w_2-bit partial multiplication algorithm over $GF(2^m)$ has already reported by Hasan [5]. We extend this algorithm to w_1-bit×w_2-bit partial multiplication.

Multiplication over $GF(2^m)$.
The multiplication algorithm over $GF(2^m)$ is the following Algorithm 1.

Algorithm 1.
Input : $a(x), b(x), f(x)$
Output : $r(x) = a(x) \cdot b(x) \pmod{f(x)}$

Step 1. $t(x) = a(x) \cdot b(x)$
Step 2. $e(x) = \lfloor t(x)/f(x) \rfloor$
Step 3. $r(x) = t(x) + e(x) \cdot f(x)$

Here $t(x)$ is a temporary variable, and $\lfloor t(x)/f(x) \rfloor$ is a quotient in which $t(x)$ is divided by $f(x)$.

Multiplication with m-Bit×w_2-Bit Partial Multiplications.
We show Algorithm 2 so that $b(x)$ is handled word-by-word. This algorithm is based on the multiplication reported by Hasan [5].

Algorithm 2.
Input : $a(x), b(x), f(x)$
Output : $r(x) = a(x) \cdot b(x) \pmod{f(x)}$

Step 1. $r(x) = 0$
Step 2. *for* $(j = n_2 - 1; j \geq 0; j = j - 1)$ {
Step 3. $t(x) = r(x) \cdot x^{w_2} + a(x) \cdot B_j(x)$
Step 4. $e(x) = \lfloor t(x)/f(x) \rfloor$
Step 5. $r(x) = t(x) + e(x) \cdot f(x)$
Step 6. }

Multiplication with w_1-bit×w_2-bit Partial Multiplications.
In Algorithm 2, $r(x)$ is calculated by m-bit×w_2-bit partial multiplications. We have extended Algorithm 2 the following Algorithm 3 in which $a(x)$ is handled block-by-block.

Algorithm 3. (Proposed Algorithm)
Input : $a(x), b(x), f(x)$
Output : $r(x) = a(x) \cdot b(x) \pmod{f(x)}$

Step 1. $r(x) = 0$
Step 2. *for* $(j = n_2 - 1; j \geq 0; j = j - 1)$ {
Step 3. $Uc(x) = 0$

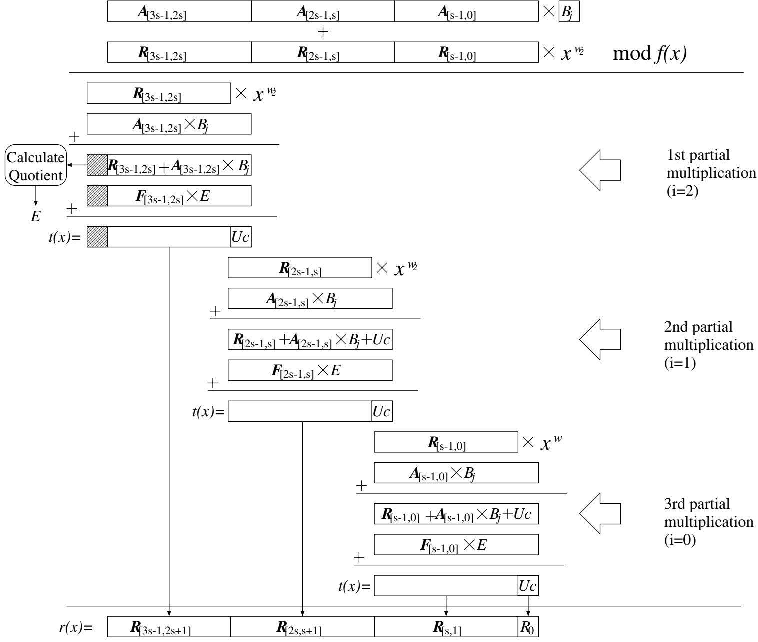


Fig. 1. m-bit$\times m$-bit Multiplication with w_1-bit$\times w_2$-bit Partial Multiplications (n_1=3).

Step 4. for $(i = n_1 - 1; i \geq 0; i = i - 1)$ {
Step 5. $\boldsymbol{T}_{[s,0]}(x) = \boldsymbol{R}_{[s\cdot i+s-1,s\cdot i]}(x)\cdot x^{w_2} + Uc(x)\cdot x^{w_1} + \boldsymbol{A}_{[s\cdot i+s-1,s\cdot i]}(x)\cdot B_j(x)$
Step 6. if $(i == n_1 - 1)$ $E(x) = \lfloor \boldsymbol{T}_{[s,0]}(x) \cdot x^{(n_1-1)\cdot w_1}/f(x)\rfloor$
Step 7. $\boldsymbol{T}_{[s,0]}(x) = \boldsymbol{T}_{[s,0]}(x) + E(x)\cdot \boldsymbol{F}_{[s\cdot i+s-1,s\cdot i]}(x)$
Step 8. if $(i == n_1 - 1)$ $\boldsymbol{R}_{[s\cdot i+s-1,s\cdot i+1]}(x) = \boldsymbol{T}_{[s-1,1]}(x)$
Step 9. else $\boldsymbol{R}_{[s\cdot i+s,s\cdot i+1]}(x) = \boldsymbol{T}_{[s,1]}(x)$
Step 10. $Uc(x) = T_0(x)$
Step 11. }
Step 12. $R_0(x) = Uc(x)$
Step 13. }

Figure 1 shows the calculations of Algorithm 3 from step 4 to step 12 when $n_1 = 3$. In Figure 1 there are three partial multiplications corresponding to step 5. Uc is the least significant word of intermediate value $t(x)$ and is added to the next partial product as the most significant word. The part of $r(x)$ is substituted with the highest s-word of $t(x)$. In the third partial multiplication, the lowest word of $r(x)$ is substituted with $Uc(x)$. Note that, $E(x)$ calculated in step 6 when $i = n_1 - 1$ is used when $i = n_1 - 1, n_1 - 2, \ldots, 0$ in step 7. This operation enables the partial multiplication over $GF(2^m)$.

2.4 An Example of Algorithm 3

In this example, $f(x) = x^{12} + x^{11} + x^8 + x^6 + 1$ over $GF(2^{12})$, $w_1 = 4$, and $w_2 = 2$. Thus the 12-bit×12-bit multiplication is executed by a 4-bit×2-bit partial multiplication When, $n_1 = 3$, $n_2 = 6$, $s = 2$,

$$a = (\boldsymbol{A}_{[5,4]}, \boldsymbol{A}_{[3,2]}, \boldsymbol{A}_{[1,0]}) = (A_5, A_4, A_3, A_2, A_1, A_0) = (11, 00, 10, 10, 10, 01),$$
$$b = (\boldsymbol{B}_{[5,4]}, \boldsymbol{B}_{[3,2]}, \boldsymbol{B}_{[1,0]}) = (B_5, B_4, B_3, B_2, B_1, B_0) = (01, 11, 00, 01, 11, 10),$$
$$f = (1100101000001), \text{ and } (F_5, F_4, F_3, F_2, F_1, F_0) = (10, 01, 01, 00, 00, 01).$$

The following is an example of the 12-bit×2-bit partial multiplication using Algorithm 3 when $j = 0$. For simplicity, we consider only steps 4 to 12, when r, t, E, and Uc are initialized by 0.

Step 4. $i = 2$
Step 5. $\boldsymbol{T}_{[2,0]}(x) = (00, 00) \cdot x^2 + (00) \cdot x^4 + (11, 00) \cdot (10) = (01, 10, 00)$
Step 6. $\boldsymbol{E}(x) = \lfloor (01, 10, 00) \cdot x^8 / (10, 01, 01, 00, 00, 01) \rfloor = (01)$
Step 7. $\boldsymbol{T}_{[2,0]}(x) = (01, 10, 00) + (01) \cdot (10, 01) = (01, 00, 01)$
Step 8. $\boldsymbol{R}_{[5,5]}(x) = (00)$
Step 10. $Uc(x) = (01)$

Step 4. $i = 1$
Step 5. $\boldsymbol{T}_{[2,0]}(x) = (00, 00) \cdot x^2 + (01) \cdot x^4 + (10, 10) \cdot (10) = (00, 01, 00)$
Step 7. $\boldsymbol{T}_{[2,0]}(x) = (00, 01, 00) + (01) \cdot (01, 00) = (00, 00, 00)$
Step 9. $\boldsymbol{R}_{[4,3]}(x) = (00, 00)$
Step 10. $Uc(x) = (00)$

Step 4. $i = 0$
Step 5. $\boldsymbol{T}_{[2,0]}(x) = (00, 00) \cdot x^2 + (00) \cdot x^4 + (10, 01) \cdot (10) = (01, 00, 10)$
Step 7. $\boldsymbol{T}_{[2,0]}(x) = (01, 00, 10) + (01) \cdot (00, 01) = (01, 00, 11)$
Step 9. $\boldsymbol{R}_{[2,1]}(x) = (01, 00)$
Step 10. $Uc(x) = (11)$

Step 12. $R_0(x) = (11)$

From the above calculation, $r = \boldsymbol{R}_{[5,0]} = (00, 00, 00, 01, 00, 11)$ is obtained.

It is clear that when $j = 0$ we can get the same partial product as that in Algorithm 2. That is,

$$\begin{aligned} t(x) &= r(x) \cdot x^{w_2} + a(x) \cdot B_0(x) \\ &= 0 + (1100, 1010, 1001)(10) \\ &= (1, 1001, 0101, 0010) \\ e(x) &= \lfloor t/f \rfloor = 1 \\ r(x) &= t + e \cdot f = (0000, 0001, 0011). \end{aligned}$$

2.5 Calculation of Quotient $E(x)$

In step 6 of Algorithm 3, division by $f(x)$ is used to calculate wh$E(x)$. But because $f(x) = x^m + f^*(x)$, it can also be calculated with the highest w_2-bit of $T_s(x)$ in $\boldsymbol{T}_{[s,0]}(x)$ and $(f_{m-1}, f_{m-2}, \ldots, f_{m-w_2+1})$, which is the highest (w_2-1)-bit of $f^*(x)$. Algorithm 4 shows this calculation of $E(x)$.

Algorithm 4.
Input : $T_s(x), (f_{m-1}, f_{m-2}, \ldots, f_{m-w_2+1})$
Output : $E(x) = \lfloor \boldsymbol{T}_{[s,0]}(x) \cdot x^{(n_1-1)\cdot w_1}/f(x) \rfloor$

Step 1. $E(x) = 0$
Step 2. $U(x) = T_s(x)$
Step 3. *for* $(i = w_2 - 1; i \geq 0; i = i - 1)$ {
Step 4. *if* $(u_i == 1)$ {
Step 5. $e_i = 1$
Step 6. *for* $(j = i - 1; j \geq 0; j = j - 1)$ $u_j = u_j + f_{m-i+j}$
Step 7. }
Step 8. }

Here $U(x)$ is a temporary word variable.

2.6 Data Conversion Method

In the previous sections we have assumed that m is a multiple of w_1. In this section we discuss the case in which it is not. That is, m has an arbitrary bit length. Let α be the minimum positive integer that satisfies $m + \alpha = n_1 \times w_1$.

In Algorithms 3 and 4, the multiplication is processed from higher block/word to lower block/word. In Algorithm 4 the most significant bit (a coefficient of the highest degree) is used to calculate a quotient. To calculate these algorithms efficiently, the elements over $GF(2^m)$ should be converted to fill the most significant block. That is, elements should be multiplied by x^α. In addition this conversion is homomorphic, but in multiplication it is not.

Addition.

$$r(x) = a(x) + b(x) \pmod{f(x)}$$

$$\Rightarrow (a(x)x^\alpha + b(x)x^\alpha) = (a(x) + b(x))x^\alpha = r(x)x^\alpha \pmod{f(x)x^\alpha}$$

Multiplication.

$$r(x) = a(x) \cdot b(x) \pmod{f(x)}$$

$$\Rightarrow (a(x)x^\alpha)(b(x)x^\alpha) = (a(x) \cdot b(x))x^\alpha)x^\alpha \neq r(x)x^\alpha \pmod{f(x)x^\alpha}$$

To solve this problem, we need to multiply either multiplier $a(x)x^\alpha$ or multiplicand $b(x)x^\alpha$ by $x^{-\alpha}$. That is,

$$(a(x)x^\alpha) \cdot (b(x)x^\alpha \cdot x^{-\alpha}) = a(x) \cdot b(x)x^\alpha = r(x)x^\alpha \pmod{f(x)x^\alpha}$$

$r(x)$ can be retrieved by multiplying the above result by $x^{-\alpha}$. These processes are inefficient and cause a large overhead when a sequence of multiplications is required, as in ECC calculation.

So we propose a method in which all the input data is converted, before the sequence of multiplications is performed. In the final step, the data is inverse converted.

The element $a(x)$ is first converted to $a(x)x^{-\alpha} \pmod{f(x)}$ and then multiplied by x^{α}, as follows:

$$\overline{a}(x) = (a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha}.$$

The addition algorithm is clearly unchanged by this conversion. To see that the multiplication algorithm is unchanged, consider

$$\begin{aligned}
&\overline{a}(x) \cdot \overline{b}(x) \pmod{f(x)x^{\alpha}} \\
&= (a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \cdot (b(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \pmod{f(x)x^{\alpha}} \\
&= (a(x) \cdot b(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \pmod{f(x)x^{\alpha}} \\
&= \overline{a(x) \cdot b(x)} \pmod{f(x)x^{\alpha}}.
\end{aligned}$$

The inverse conversion for $\overline{a}(x)$ is processed by $\overline{a}(x) \pmod{f(x)}$. That is,

$$\begin{aligned}
\overline{a}(x) \pmod{f(x)} &= ((a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha}) \pmod{f(x)} \\
&= (a(x)x^{-\alpha} \cdot x^{\alpha}) \pmod{f(x)} \\
&= a(x) \pmod{f(x)}.
\end{aligned}$$

By doing this conversion, the multiplication of Algorithm 3 can be expanded for any bit length. The conversion and the inverse conversion can be summarized as follows:

$$\begin{aligned}
\text{Conversion}: \overline{a}(x) &= (a(x)x^{-\alpha} \pmod{f(x)})x^{\alpha} \\
\text{Inverse conversion}: a(x) &= \overline{a}(x) \pmod{f(x)}
\end{aligned}$$

2.7 An Example of the Data Conversion Method

In this example, $f(x) = x^5 + x^2 + 1$ over $GF(2^5)$, $w_1 = 8$, $n_1 = 1$, and $a = 3$. We convert the elements over $GF(2^5)$ for calculating with an 8-bit×8-bit multiplication.

$$\begin{aligned}
a(x) &= x^4 + 1 \\
b(x) &= x^4 + x + 1 \\
c(x) &= a(x) \cdot b(x) = x^3 + x + 1 \pmod{f(x)}
\end{aligned}$$

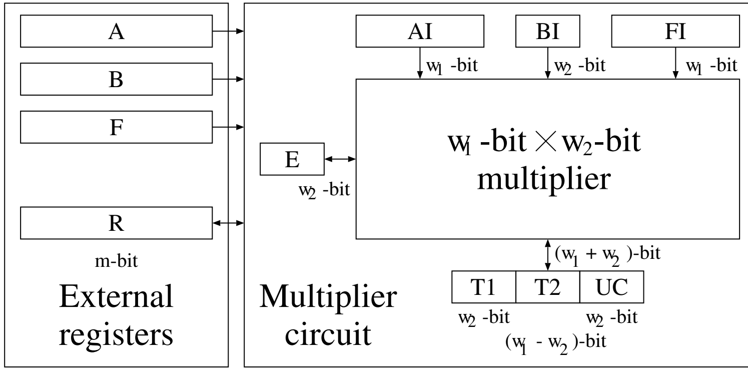


Fig. 2. Block Diagram of Our Multiplier

Conversion.
We convert $a(x)$ and $b(x)$ into $\overline{a}(x)$ and $\overline{b}(x)$ and calculate $\overline{c}(x)$, where $x^\alpha = x^3$ and $x^{-\alpha} = x^{-3} = x^4 + x^2 + x$.

$$\overline{a}(x) = (a(x) \cdot x^{-\alpha} \pmod{f(x)}) \cdot x^\alpha = x^7 + x^5$$
$$\overline{b}(x) = (b(x) \cdot x^{-\alpha} \pmod{f(x)}) \cdot x^\alpha = x^7 + x^6 + x^5 + x^3$$

Multiplication.

$$\begin{aligned}\overline{c}(x) &= \overline{a}(x) \cdot \overline{b}(x) \pmod{f(x) \cdot x^\alpha} \\ &= (x^7 + x^5) \cdot (x^7 + x^6 + x^5 + x^3) \pmod{x^8 + x^5 + x^3} \\ &= x^7 + x^6 + x^5 + x^4\end{aligned}$$

Inverse Conversion.

$$\begin{aligned}c(x) &= \overline{c}(x) \pmod{f(x) = x^5 + x^2 + 1} \\ &= x^3 + x + 1\end{aligned}$$

3 Our Multiplier

3.1 Block Diagram

Figure 2 is a block diagram of our multiplier. A, B, and F in Figure 2 are m-bit registers that store the multiplicand a, the multiplier b, and the irreducible polynomial f^*. R is an m-bit output register that stores the intermediate value of multiplication and the result. In this paper we call these registers "external registers." Each register A, F, and R is handled block by block, that is, A_i, F_i and R_i. And register B is handled word by word, that is, B_j. Moreover, R_i is divided into two sections: the highest $(w_1 - w_2)$-bit and the lowest w_2-bit. We denote

the highest $(w_1 - w_2)$-bit as RH_i and the lowest w_2-bit as RL_i . In addition, we call the registers $AI, BI, FI, E, T1, T2$, and UC in the multiplier "internal registers." AI, BI, and FI are respectively w_1-bit, w_2-bit and w_1-bit registers that store the input data, that is A_i, B_i and F_i, from the "external registers." These registers are used by the w_1-bit$\times w_2$-bit multiplier as input registers. E is a w_2-bit register that stores the intermediate value of the multiplier. $T1, T2$, and UC are respectively w_2-bit, $(w_1 - w_2)$-bit, and w_2-bit registers. They exchange the intermediate value and the result of the multiplier with R and are used by the multiplier as input and output registers.

3.2 Configuration

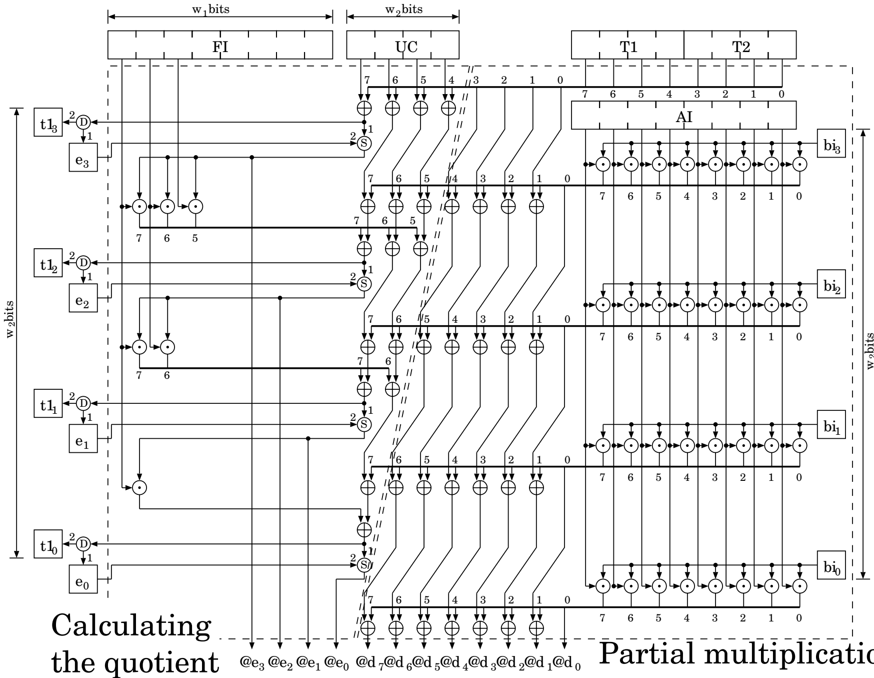
The following is the process flow for Algorithms 3 and 4 in our multiplier. Figure 3 shows our w_1-bit$\times w_2$-bit multiplier. We use $(w_1, w_2) = (8, 4)$ as an example.

The register value is denoted by the register name, and is expressed such as $AI = (ai_{w_1-1}, ai_{w_1-2}, \ldots, ai_0)$. In addition, the concatenation of a w_1-bit register AI and a w_2-bit register BI is denoted as $AI||BI$. That is, $AI||BI = (ai_{w_1-1}, ai_{w_1-2}, \ldots, ai_0, bi_{w_2-1}, bi_{w_2-2}, \ldots, bi_0)$.

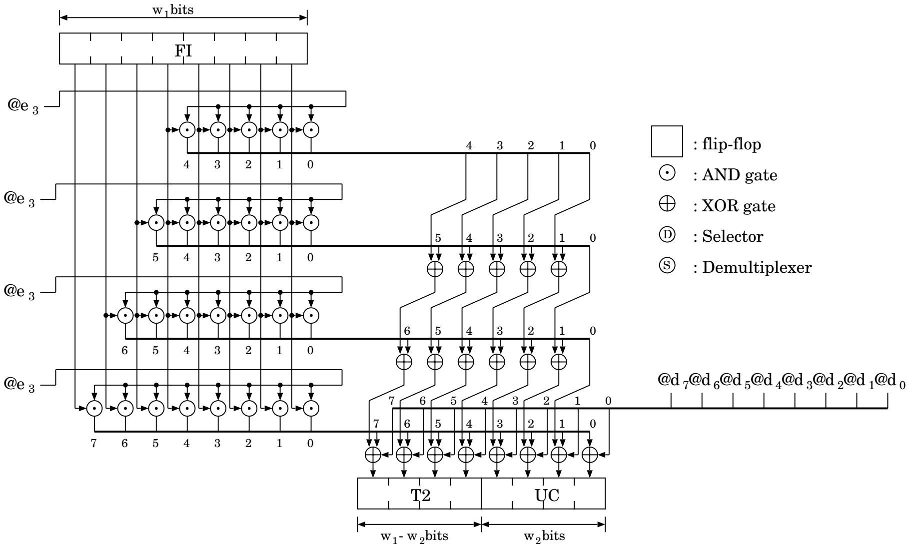
Input : a, b, f^*
Output : $r = a \cdot b \pmod f$

Proc. 1. $R \leftarrow 0$
Proc. 2. for $j = n_2 - 1$ to 0
Proc. 3. $T1 \leftarrow 0; T2 \leftarrow 0; E \leftarrow 0; UC \leftarrow 0$
Proc. 4. for $i = n_1 - 1$ to 0
Proc. 5. $(T1||T2) \leftarrow R_i$
Proc. 6. $AI \leftarrow A_i$
Proc. 7. $BI \leftarrow B_j$
Proc. 8. $FI \leftarrow F_i$
Proc. 9. $(T1||T2||UC) \leftarrow (T1||T2) \cdot x^{w_2} + UC \cdot x^{w_1} + AI \cdot BI$
Proc. 10. if $(i == n_1 - 1)$ $E \leftarrow \lfloor((T1||T2) \cdot x^{w_2} + AI \cdot BI)/F_{n_1-1})\rfloor$
Proc. 11. $(T1||T2||UC) \leftarrow (T1||T2||UC) + FI \cdot E$
Proc. 12. if $(i \neq n_1 - 1) RL_{i+1} \leftarrow T1$
Proc. 13. $RH_i \leftarrow T2$
Proc. 14. if $(i == 0)$ $RL_0 \leftarrow UC$
Proc. 15. next i
Proc. 16. next j

The modular multiplication in process 9-11 corresponds to that in steps 5-7 of Algorithm 3. The data storage in process 12-14 corresponds to that in steps 8-10 of Algorithm 3. Process 9 and 10 represent the w_1-bit$\times w_2$-bit multiplication which is executed by the multiplier shown in Figure 3(a), and process 11 represents the reduction by $f(x)$ which is executed by the multiplier shown in Figure 3(b). Process 10 represents the calculation of the quotient E, done by the left side of the circuit in Figure 3(a) which corresponds to Algorithm 4. If $i = (n_1 - 1)$ in process 10, selector S and demultiplexer D in Figure 3(1) are



(a) Partial Multiplication and Calculating Quotient Part



(b) Reduction Part

Fig. 3. An Example of Our w_1-bit$\times w_2$-bit Multiplier ($w_1 = 8$, $w_2 = 4$)

Table 1. Hardware Size and Critical Path.

Configuration	Hardware size					critical path delay		
	#XOR	#AND	#SEL	#DMUL	#FF	#XOR	#AND	#SEL
Our multiplier	$2w_1w_2$	$2w_1w_2$	w_2	w_2	$3(w_1+w_2)$	w_2+2	w_2	w_2
Laws' multiplier [10]	$2w_1w_2$	$2w_1w_2$	w_2	w_2	$3(w_1+w_2)$	$2w_2+1$	w_2	w_2
Im's multiplier [7]	$2w_1w_2$	$2w_1w_2$	w_2	w_2	$3(w_1+w_2)$	w_2+1	w_2	w_2
SSM [15]	$2w_1$	$2w_1$	–	–	$3w_1+3$	2	1	–

switched to side 1, so that the quotient is stored in register E. If $i \neq n_1 - 1$ in process 10, selector S and demultiplexer D in Figure 3(a) are switched to side 2.

3.3 Performance Analysis

Hardware Size and Performance.
The hardware size and critical path delay of our multiplier are listed in Table 1. As the values of w_1 and w_2 increase, the number of XOR gates and AND gates also increase to be a dominant factor determining hardware size.

The performance of our multiplier is evaluated by the critical path delay and $C(m)$. The critical path delay is proportional to w_2, and $C(m)$ is the number of partial multiplications required for multiplication over $GF(2^m)$ with the w_1-bit$\times w_2$-bit multiplier:

$$C(m) = \lceil m/w_1 \rceil \times \lceil m/w_2 \rceil + \beta,$$

where $\lceil x \rceil$ is the least integer greater than x. We assume that the w_1-bit$\times w_2$-bit multiplier is performed in one cycle, and we ignore the data transfer cycle for pipeline processing in which data is transferred during the multiplication. The variable β is a constant, and it is the sum of the number of cycles for the input and the output.

The number of cycles for multiplication over $GF(2^m)$ is shown in Figure 4 for $(w_1, w_2) = \{(288, 8), (144, 16), (96, 24), (72, 32), (48, 48)\}$. Note that the hardware size of the multipliers is the same for each of these pairs. In our evaluation, the number of processing cycles is the product of the critical path delay and the $C(m)$ based on (288,8). For example, when $m = 288$ and $(w_1, w_2) = (144, 16)$, the $C(m)$ remains unchanged, but the critical path delay is twice as long. Thus the number of processing cycles for (144, 16) is evaluated twice as that for (288, 8).

From Figure 4, it can be seen that processing is faster when w_1 is larger and w_2 is smaller. Theoretically, then the processing is the fastest when $w_2 = 1$. But, the processing is not always the fastest when $w_2 = 1$. This is due to an upper boundary of the processing clock that depends on the hardware characteristics. So we selected $w_2 = 4$ for the FPGA implementation and $w_2 = 8$ for the ASIC implementation to get the best performance.

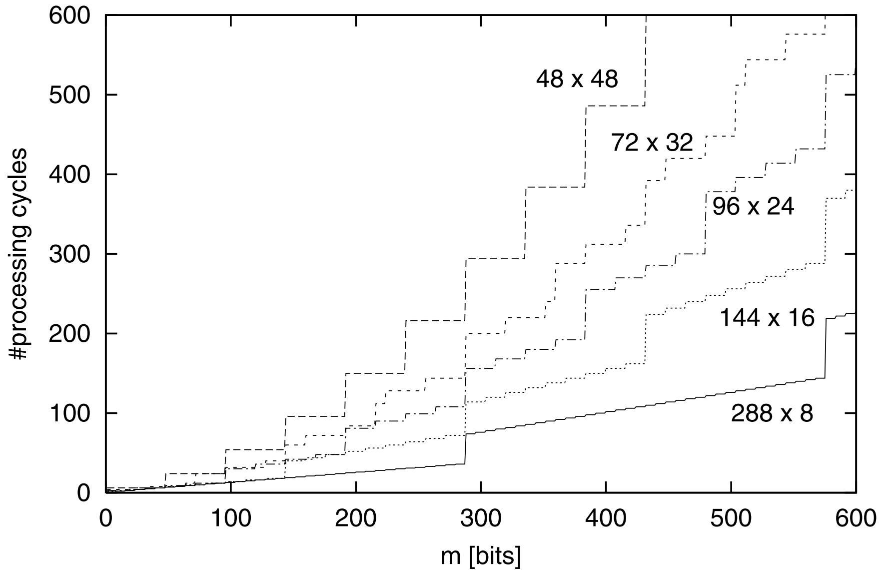


Fig. 4. Processing Cycles for Multiplication

Comparison.
In this section, we compare our multiplier with the three others based on the LFSR [10] [7] [15]. In [10] and [7], the multipliers are bit parallel m-bit$\times m$-bit multipliers based on the LFSR. In [10] polynomial is programmable, while in [7] it is fixed. The SSM is a bit serial multiplier based on the LFSR.

First we compare our multiplier with Im's multiplier and Laws' multiplier. For this comparison we modify Laws' multiplier and Im's multiplier so they can perform w_1-bit$\times w_2$-bit multiplications for arbitrary irreducible polynomials at any bit length. The hardware sizes and critical path delays are listed in Table 1.

The hardware size of our multiplier is the same as that of Laws' multiplier and Im's multiplier, and the difference of critical path delay is negligible in comparison with Im's multiplier the shortest one. If our methods were applied to Im's multiplier, it would improve flexibility and the performance.

Next, we compare our multiplier with the SSM. We evaluate the hardware sizes and critical path delays of the SSM, which are shown in Table 1. We consider the SSM is the special case of our multiplier when $w_2 = 1$. That is, the hardware size of the SSM is the same as that of our multiplier when $w_2 = 1$ and the difference of critical path delay is negligible in comparison with our multiplier and is the same as the Im's multiplier when $w_2 = 1$. We consider our multiplier architecture to be an extension of the concept of the SSM.

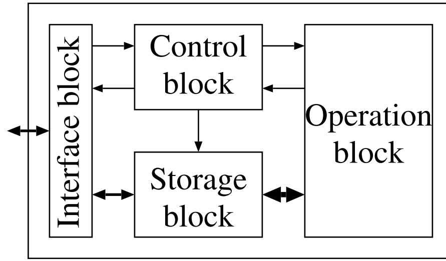


Fig. 5. Basic Diagram of the ECC Coprocessor

4 ECC Coprocessor Implementation

4.1 Basic Design

Figure 5 shows the basic design of the ECC coprocessor with our multiplier. It consists of four function blocks: an interface block, an operation block, a storage block, and a control block. The interface block controls the communications between host and coprocessor. The operation block contains the w_1-bit$\times w_2$-bit multiplier described in section 3. The storage block stores the input, output, and intermediate values. The control block controls the operation block in order to perform elliptic scalar multiplication.

We referred to IEEE P1363 draft [6] for elliptic curve algorithms on a pseudo-random curve, and to FIPS186-2 draft [14] for elliptic curve algorithms on a Koblitz curve [9].

4.2 Implementation

We designed the above coprocessor by using Verilog-HDL, and implemented the ECC coprocessor on an FPGA. The FPGA we used was the EPF10K250AGC599-2 by ALTERA [3]. It operates at 3 MHz. We implemented the coprocessor using an 82-bit$\times$4-bit multiplier. It can do up to 163-bit elliptic scalar multiplication, and the processing times for 163-bit elliptic scalar performance for 163-bit elliptic scalar multiplication with this coprocessor are listed in Table 2.

We also designed and simulated an ECC coprocessor including a 288-bit$\times$8-bit multiplier for up to 572-bit elliptic scalar multiplication. In this simulation we used CE71 series of 0.25 μm ASIC, which is macro-embedded cell arrays by

Table 2. Performance of FPGA Implementation

$Length[bit]$	processing time [msec]	
	Pseudo-random curve	Koblitz curve
163	80.3	45.6

Table 3. Performance of ASIC Implementation (Simulation)

$Length[bit]$	processing time [msec]	
	Pseudo-random curve	Koblitz curve
163	1.1	0.65
233	1.9	1.1
283	3	1.7
409	11	6.6
571	22	13

FUJITSU [4]. Our coprocessor can operate at up to 66 MHz and its hardware size is about 165 Kgates. The processing time for elliptic scalar multiplication on a pseudo-random curves and a Koblitz curves when $m = 163, 233, 283, 409$, and 571 are listed in Table 3. These bit lengths are recommended in FIPS 186-2 draft [14].

5 Conclusion

We described the implementation of an elliptic curve cryptographic coprocessor over $GF(2^m)$ on an FPGA (EPF10K250AGC599-2, ALTERA). This coprocessor is suitable for server systems and enables efficient ECC operations for various parameters.

We proposed a novel multiplier configuration over $GF(2^m)$ that makes ECC calculation faster and more flexible. Its two special characters are that its bit parallel multiplier architecture is an expansion of the concept of the SSM, and that its data conversion method makes possible fast multiplication at any bit length.

The ECC coprocessor implemented with our multiplier on an FPGA performs a fast elliptic scalar multiplication on a pseudo-random curve and on a Koblitz curve. For 163-bit elliptic scalar multiplication, operating at 3 MHz, it takes 80 ms on a pseudo-random curve and 45 ms on a Koblitz curve. We also simulated the operation of this coprocessor implemented as a 0.25 μm ASIC that can operate at 66 MHz and has a hardware size of 165 Kgates. For 163-bit elliptic scalar multiplication, it would take 1.1 ms on a pseudo-random curve and 0.65 ms on a Koblitz curve. And for 571-bit, it would take 22 ms on a pseudo-random curve and 13 ms on a Koblitz curve.

References

1. G.B. Agnew, R.C. Mullin, and S.A. Vanstone, "An implementation of elliptic curve cryptosystems over $GF(2^{155})$," IEEE Journal on Selected Areas in Communications, 11(5), pp. 804-813, 1993.
2. G.B. Agnew, R.C. Mullin, I.M. Onyschuk, and S.A. Vanstone, "An implementation for a fast public-key cryptosystem," Journal of Cryptography, vol.3, pp. 63-79, 1991.

3. Altera, "FLEX 10K Embedded programmable logic Family Data Sheet ver.4.01," 1999. http://www.altera.com/document/ds/dsf10k.pdf
4. Fujitsu, "CMOS Macro embedded type cell array CE71 Series," 2000 http://www.fujitsu.co.jp/hypertext/Products/Device/ CATALOG/AD0000-00001/10e-5b-2.html
5. M.A. Hasan, "Look-up Table Based Large Finite Field Multiplication in Memory Constrained Cryptosystems," IEEE Trans. Comput., vol.49, No.7, July 2000 (to be appear).
6. IEEE, "IEEE P1363/D13(Draft Version 13). Standard Specifications for Public Key Cryptography Annex A (Informative). Number-Theoretic Background," 1999.
7. J.H. Im, "Galois field multiplier," U.S Patent #5502665, 1996.
8. N. Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation 48, pp.203-209, 1987.
9. N. Koblitz, "CM-curve with good cryptographic properties," Advances in Cryptology, Proc. Crypto'91, Springer-Verlag, pp.279-287 1992.
10. B.A. Laws and C.K. Rushforth, "A cellular-array multiplier for $GF(2^m)$," IEEE Trans. Comput., vol.C-20, pp.1573-1578, 1971.
11. E.D. Mastrovito, "VLSI design for multiplication over finite fields $GF(2^m)$," In Lecture Notes in Computer Science 357, pp.297-309. Springer-Verlag, 1989.
12. A.J. Menezes, "Elliptic Curve Public Key Cryptosystems," Kluwer Academic Publishers, 1993.
13. V.S. Miller, "Use of elliptic curve curves in cryptography," Advances in Cryptology, Proc.Crypto'85, Springer-Verlag, pp.417-426, 1986.
14. NIST, "FIPS 186-2 draft, Digital Signature Standard (DSS)," 2000. http://csrc.nist.gov/fips/fips186-2.pdf
15. G. Orlando and C. Paar, "A Super-Serial Galois Fields Multiplier for FPGAs and its Application to Public-Key Algorithms," in Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '99, April 1999.
16. M. Rosner, "Elliptic Curve Cryptosystems on Reconfigurable Hardware," Master's Thesis, Worcester Polytechnic Institute, 1998. http://www.ece.wpi.edu/Research/crypt/theses/index.html