5 Efficient Elliptic Curve Cryptographic Hardware Design for Wireless Security

Lo'ai A. Tawalbeh and Çetin Kaya Koç

CONTENTS

5.1 Introduction 154
5.2 Elliptic Curve Theory 155
5.2.1 Elliptic Curves Defined over GF(p) 156
5.2.1.1 Affine Coordinates 157
5.2.1.2 Projective Coordinates 157
5.2.2 Elliptic Curves Defined over GF(2^n) 158
5.2.2.1 Affine Coordinates 159
5.2.2.2 Projective Coordinates 159
5.2.3 Arithmetic Complexity of Affine and Projective Coordinates 160
5.3 Elliptic Curve Cryptosystems 161
5.3.1 Elliptic Curve Digital Signature Algorithm 161
5.3.1.1 ECDSA Key Generation 161
5.3.1.2 ECDSA Signature Generation 161
5.3.1.3 ECDSA Signature Verification 162
5.3.2 Elliptic Curve ElGamal Cryptosystem 162
5.4 Scalable Hardware Design for Elliptic Curve Cryptography 163
5.4.1 Unified Division/Multiplication Algorithm 163
5.4.2 Top Level Hardware Architecture Implementing UDMA 165
5.4.2.1 Register File 165
5.4.2.2 Datapath 166
5.4.2.3 Control Block 167
5.4.3 Experimental Results for UMDM 168
5.4.3.1 ASIC Results for the UMDM Scalable Design 168
5.4.3.2 FPGA Results for the UMDM Scalable Design 169
5.5 Elliptic Curve Crypto-Processor over GF(2^n) 171
5.5.1 ECCP Hardware Architecture 171
5.5.2 Experimental Results and Analysis for GF(2^n) ECCP 172
References 174

The spread of wired and wireless communications, the continuous growth of the Internet, and the E-commerce transactions increased the necessity for security in applications that involve sharing or exchange of secret or private information. Public-key cryptography is widely used in establishing secure communication channels between the users on the Internet and in wireless communication networks.

5.1 INTRODUCTION

A small set of public-key cryptosystems are used extensively, which includes ElGamal cryptosystem [1], Diffie–Hellman (DH) key exchange algorithm [2], the digital signature algorithm (DSA) [3], and elliptic curve cryptography–based algorithms such as EC–ElGamal and ECDSA. Elliptic curve cryptography (ECC), which was introduced by Miller [4] and Koblitz [5], is based on a more difficult mathematical problem to solve than the one used in traditional public-key algorithms. Thus, ECC stands out from this crowd of algorithms because of its unique property of providing the highest degree of security with the smallest key sizes. For example, an elliptic curve system with 313-bits can replace a certain 4096-bit key size conventional system [6]. Using smaller key sizes to gain the same level of security leads to a big reduction in hardware resources used in implementations.

In this chapter, we mainly concentrate on efficient hardware realization of elliptic curve cryptography for wireless applications. Elliptic curve cryptography involves huge arithmetic operations performed over finite fields (most commonly used fields are the prime extension fields, GF(p), and the binary extension fields, GF(2^n)), and therefore, an efficient ECC system requires efficient hardware implementations of finite field operations. Once realized, similar hardware can also be used to support other public-key cryptographic functions. Furthermore, long-term deployment of public-key cryptography hardware requires flexibility in key size as better cryptanalytic techniques are developed.

Recently, two important developments took place in this area. The first one is called *scalability* which refers to the ability of the hardware to reconfigure itself to support longer key sizes, limited only by the amount of available input, output, and scratch memory space. The second one is about designing a single hardware to support all kinds of elliptic curves based on finite fields of different characteristics. This property of hardware is called *unified* or *dual field*.

Our research starts from these premises and moves on to create better algorithms to support long-term, efficient, scalable, and unified hardware implementations. We address and provide solutions for dual-field Montgomery multipliers, modular dividers, and unified dividers and inverters. Particularly, we introduce a novel algorithm suitable for hardware design which computes division (inverse) and multiplication in a very efficient way for

GF(p) and GF(2^n) fields. The new algorithm is called the unified division/multiplication algorithm (UDMA). In addition, we propose the hardware architecture that efficiently supports all operations in the UDMA and uses carry-save unified adders for reduced critical path delay, making the proposed architecture faster than other previously proposed designs. We present example designs of our algorithms using field programmable gate arrays (FPGAs) and the benchmark results of our implementations.

At the end of this chapter, we introduce an elliptic curve crypto-processor (ECCP) architecture over GF(2^n) that is based on the efficient UDMA hardware implementation. The scalability feature of the proposed crypto-processor allows the adjustment of the word size used in the datapath to meet area and performance requirements. On the other hand, the processor allows the user to choose the value of the field parameter (n). Finally, the experimental results obtained for the ECCP are analyzed and compared with other proposed designs.

5.2 ELLIPTIC CURVE THEORY

In the mid-1980s, Niel Koblitz and Victor Miller proposed the elliptic curve cryptography (ECC) [4,5]. It is based on the discrete logarithm (DL) problem over the points on an elliptic curve (EC). Recently, the elliptic curve cryptosystems started to replace many known conventional public-key cryptography algorithms. This is due to the high level of security they provide and their fast and compact size implementations over finite fields.

Data in an ECC are represented as points on an elliptic curve. They are called elliptic because they arose historically from the problem of computing the solutions for an equation of an ellipse. These curves have special characteristics and provide the base for particular arithmetic operations.

In cryptography, we are interested in the elliptic curves defined over finite fields. In other words, the coefficients of the defining equation ($F(x,y) = 0$) are elements of GF(q), and the points on the curve are of the form $P = (x,y)$, where x and y are the elements of GF(q) that satisfy the equation. The general form for an elliptic curve equation is

$$y^2 + axy + by = x^3 + cx^2 + dx + e.$$

A point at infinity (O) is also defined [7]. O plays a role similar to zero in ordinary addition. It is computed as the sum of three points that lie on a straight line on the EC.

The complexity of elliptic curve arithmetic operations that includes rules used to add two points (point addition) or add a point to itself (point doubling) on the elliptic curves, depends on the finite field (GF(p) or GF(2^n)) and on the coordinate system (affine or projective) that is used. Moreover, choosing the

suitable representation for the elements of the finite field may lead to more efficient implementations of the field arithmetic in hardware or in software.

The core operation on ECC is the scalar point multiplication, which consists of a certain number of point additions. When a point P defined on the curve is added to itself k times, it is very difficult to find what was P without knowing k. That is the characteristic that provides security to ECC:

$$Q = kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}. \tag{5.1}$$

In the following subsections, we discuss the elliptic curves defined over GF(p) and GF(2^n) and the arithmetic algorithms defined in each field.

5.2.1 Elliptic Curves Defined over GF(p)

The elements of the field GF(p) are the integers in the set $\{0, 1, 2, \ldots, p-1\}$, where p is an n-bit prime modulus in the range of $2^{n-1} < p < 2^n$. The basic arithmetic operations defined in this field are

- *Addition modulo p*. The addition of elements in a prime field is a conventional integer addition with modulo reduction (mod p). For example, let X, Y, $R \in$ GF(p), then $R = X + Y \bmod p$, where R is the remainder of $(X + Y)$ divided by p.
- *Multiplication modulo p*. Let $M = X \cdot Y$, where X, Y, $M \in$ GF(p), M is the remainder of $X \cdot Y$ divided by p.
- *Squaring*. If $X \in$ GF(p), then $X^2 = X \cdot X$ is the remainder of X^2 divided by p.
- *Inversion modulo p*. Inversion is defined for a nonzero element $X \in$ GF(p) as X^{-1} to be the unique integer $W \in$ GF(p), such that $X \cdot W \equiv 1 \bmod p$.

The elliptic curves defined over GF(p) satisfy the following equation:

$$y_2 = x_3 + ax + b \bmod p,$$

where $p > 3$, $4a^3 + 27b^2 \neq 0$ and x, y, a, $b \in$ GF(p). As mentioned earlier, the point at infinity O plays a role similar to zero in the integer domain. But, there are some addition rules for O in this field. Assume that (x, y) is a point on an EC, then

1. $(x, y) + O = (x, y)$.
2. $(x, y) + (x, -y) = O$.
3. $O = -O$.

The points on the curves can be represented using affine or projective coordinates. A brief description of each coordinate is given in the following sections.

5.2.1.1 Affine Coordinates

To add two points on an elliptic curve represented in affine coordinates as $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, we compute $P_3 = (x_3, y_3) = P_1 + P_2$ and $P_1 \neq P_2$ According to the addition rules,

$$\alpha = \frac{y_2 - y_1}{x_2 - x_1},$$
$$x_3 = \alpha^2 - x_1 - x_2,$$
$$y_3 = \alpha(x_1 - x_3) - y_1,$$

and when $P_1 = P_2$ (point doubling $P_3 = 2P_1$ and $P_1 \neq 0$), the addition rules are

$$\alpha = \frac{3x_1^2 + a}{2y_1},$$
$$x_3 = \alpha^2 - 2x_1,$$
$$y_3 = \alpha(x_1 - x_3) - y_1.$$

If we assumed that the squaring calculation is equivalent to a multiplication, then the addition of two different points in GF(p) requires: six additions, one inversion, and three multiplication operations. On the other hand, to add a point to itself (point doubling) a total of four additions, one inversion, and four multiplications are required [8].

5.2.1.2 Projective Coordinates

Adding or doubling points represented in affine coordinates involve modular inversion calculations. The inversion is considered a time-consuming operation. The projective coordinates are used to almost eliminate the need for performing inversion [8].

The elliptic point, $P_1 = (x, y)$ defined over GF(p), is represented in the projective coordinates as (X, Y, Z), where $x = X/Z^2$ and $y = Y/Z^3$. This transformation is performed at the beginning to represent the point in projective coordinates. After performing the point addition operation, this transformation is carried out again to get the point back in affine coordinates. Algorithm 1 is used to add two points $(P + Q, P \neq Q)$ in projective coordinates:

$$
\begin{aligned}
P &= (X_1, Y_1, Z_1);\ Q = (X_2, Y_2, Z_2);\ P + Q = (X_3, Y_3, Z_3) \\
&\quad (x, y) = (X/Z^2, Y/Z^3), \\
T_1 &= X_1 Z_2^2, \\
T_2 &= X_2 Z_1^2, \\
T_3 &= T_1 - T_2, \\
T_4 &= Y_1 Z_2^3, \\
T_5 &= Y_2 Z_1^3, \\
T_6 &= T_4 - T_5, \\
T_7 &= T_1 + T_2, \\
T_8 &= T_4 + T_5, \\
Z_3 &= Z_1 Z_2 T_3, \\
X_3 &= T_6^2 - T_7 T_3^2, \\
T_9 &= T_7 T_3^2 - 2X_3, \\
Y_3 &= \frac{T_9 T_6 - T_8 T_3^3}{2}.
\end{aligned}
$$

The doubling point algorithm $(P + P)$ in projective coordinates is given by

$$
\begin{aligned}
P &= (X_1, Y_1, Z_1);\ P + P = (X_3, Y_3, Z_3) \\
&\quad (x, y) = (X/Z^2, Y/Z^3), \\
T_1 &= T_3 X_1^2 + a Z_1^4, \\
Z_3 &= 2 Y_1 Z_1, \\
T_2 &= 4 X_1 Y_1^2, \\
X_3 &= T_1^2 - 2T_2, \\
T_3 &= 8 Y_1^4, \\
T_4 &= T_2 - X_3, \\
Y_3 &= T_1 T_4 - X_3.
\end{aligned}
$$

From these algorithms, we found that the number of multiplication operations needed to add 2 points is 16, whereas the number of multiplications for doubling a point is found to be only 10 [8].

5.2.2 Elliptic Curves Defined over GF(2^n)

The elliptic curves defined over GF(2^n) satisfy the equation

$$E\colon y^2 + xy = x^3 + ax^2 + b,$$

where $a,b \in GF(2^n)$ and $b \neq 0$. The addition law for two points in affine coordinates involves multiplication, division, and squaring in the underlying finite field.

5.2.2.1 Affine Coordinates

Adding two points in the affine coordinates can be achieved as follows: let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points defined on the curve; then $P_3 = (x_3, y_3) = P_1 + P_2$ is defined when $P_1 \neq P_2$ as

$$\begin{aligned} \alpha &= \frac{y_1 + y_2}{x_1 + x_2}, \\ x_3 &= \alpha^2 + \alpha + x_1 + x_2 + a, \\ y_3 &= (x_1 + x_3)\alpha + x_3 + y_1, \end{aligned}$$

and when $P_1 = P_2$ (point doubling) as

$$\begin{aligned} \alpha &= x_1 + \frac{y_1}{x_1}, \\ x_3 &= \alpha^2 + \alpha + a, \\ y_3 &= (x_1 + x_3)\alpha + x_3 + y_1. \end{aligned}$$

5.2.2.2 Projective Coordinates

To eliminate the need for performing inversion in $GF(2^n)$, the affine coordinates (x, y) are projected to (X, Y, Z), where $x = X/Z^2$ and $y = Y/Z^3$ [8]. The point doubling algorithm $(P + P)$ in projective coordinates is given by

$$\begin{aligned} P &= (X_1, Y_1, Z_1);\ P + P = (X_3, Y_3, Z_3), \\ Z_3 &= X_1 Z_1^2, \\ X_3 &= (X_1 + bZ_1^2)^4, \\ T &= Z_3 + X_1^2 + Y_1 Z_1, \\ Y_3 &= X_1^4 Z_3 + T X_3. \end{aligned}$$

On the other hand, the point addition of two elliptic curve points $(P + Q)$, where $P \neq Q$, is given by

$$\begin{aligned} P &= \ (X_1, Y_1, Z_1);\ Q = (X_2, Y_2, Z_2);\ P + Q = (X_3, Y_3, Z_3), \\ (x,y) &= (X/Z^2, Y/Z^3), \\ T_1 &= X_1 Z_2^2, \end{aligned}$$

$$\begin{aligned}
T_2 &= X_2 Z_1^2, \\
T_3 &= T_1 + T_2, \\
T_4 &= Y_1 Z_2^3, \\
T_5 &= Y_2 Z_1^3, \\
T_6 &= T_4 + T_5, \\
T_7 &= Z_1 T_3, \\
T_8 &= T_6 X_2 + T_7 Y_2, \\
Z_3 &= T_7 Z_2, \\
T_9 &= T_6 + Z_3, \\
X_3 &= a Z_3^2 + T_6 T_9 + T_3^3, \\
Y_3 &= T_9 X_3 + T_8 T_7^2.
\end{aligned}$$

When using GF(2^n), the number of multiplication processes for adding 2 points is found to be 20, whereas it is found to be 10 for doubling a point.

5.2.3 Arithmetic Complexity of Affine and Projective Coordinates

A research was carried out by Gutub [8] to evaluate the complexity of performing arithmetic operations in affine and projective coordinates, and in both finite fields (GF(p) and GF(2^n)). The research was based on using the binary algorithm to compute kP from a given point P on the elliptic curve. Assuming that k is n-bits, then the algorithm performs exactly n point doubling. To evaluate the average point additions, we assume that k has half ones and half zeros. This results in $n/2$ point additions.

Table 5.1 shows the total number of multiplications and inversions for both GF(p) and GF(2^n) needed to perform n point doubling and $n/2$ point additions. The table indicates that for an affine coordinates system to be faster than a projective system, the time to compute $1.5n$ inversions and $5.5n$ multiplications should be less than $18n$, GF(p) multiplications or $20n$, GF(2^n) multiplications. But, it is worth mentioning that even using projective coordinates did not eliminate the inversion step completely. It is still required at the end of the computations to convert the result back to affine coordinates. This fact motivates the research for efficient hardware implementations for the inverse operation.

TABLE 5.1
Comparison between Affine and Projective Coordinates

Finite Field	Affine Coordinates Operations	Projective Coordinates Operations
GF(p)	$1.5n$ inversions, $5.5n$ multiplications	$18n$ multiplications
GF(2^n)	$1.5n$ inversions, $5.5n$ multiplications	$20n$ multiplications

5.3 ELLIPTIC CURVE CRYPTOSYSTEMS

Computing kP from the point p can be carried out easily using the algorithms mentioned in the earlier sections, based on which field and coordinates are used. Now, computing the value of k from the points kP and P is very hard. This fact is used to build many elliptic curve–based cryptosystems and techniques.

To change conventional systems that are based on DL problem [9] into an elliptic curve system, the following two rules are applied:

- Any modular multiplication operation defined in the conventional system is replaced by the addition of points on the elliptic curve version.
- Any modular exponentiation operation is replaced by point multiplication on the elliptic curve version of the conventional system.

There are many conventional systems that can be transferred to elliptic curve systems. As an example, we mention the elliptic curve digital signature algorithm (ECDSA) and the elliptic curve ElGamal cryptosystem (ECEC).

5.3.1 Elliptic Curve Digital Signature Algorithm

The process of ECDSA is composed of three main steps: key generation, signature generation, and signature verification. Each step is described as follows.

5.3.1.1 ECDSA Key Generation

The following procedure shows how the users should generate the public and the private keys:

1. Choose an elliptic curve E over a finite field, GF(p), for example. Assume that n is a large prime, then the number of points on E should be divisible by n.
2. Choose a point $P = (x, y) \in$ GF(p) of order n (see [6] for more information about the order).
3. Choose randomly an integer $d \in [1, n-1]$.
4. Compute $Q = dP$.
5. The public keys for the users are (Q, n, P, E), and the private key is d.

5.3.1.2 ECDSA Signature Generation

The following steps describe how to generate a signature for a certain message m:

1. Choose k to be a random integer $\in [1, n-1]$.
2. Compute $kP = (x_1, y_1)$, and set x_1 mod $n = r$. If r is zero then go back to step 1.

3. Compute k^{-1} mod n.
4. Compute $s = k^{-1}\,(H(m) + dr)$ mod n, where $H(m)$ is the hash value of the message m obtained using a suitable hash function.
5. If $s = 0$, go to step 1. This is because s^{-1} mod n does not exist, and the signature cannot be verified.
6. The pair of integers (s, r) is included in the message m as a signature.

5.3.1.3 ECDSA Signature Verification

The last step is to verify the signature (s, r) on the message m, which is executed as follows:

1. Obtain an authentic copy of the public key (Q, n, P, E).
2. Make sure that the integers r and $s \in [1, n-1]$.
3. Compute $w = s^{-1}$ mod n and $H(m)$.
4. Compute $u_1 = H(m) \cdot w$ mod n and $u_2 = r \cdot w$ mod n.
5. Compute $u_2Q + u_1P = (x_0, y_0)$ and $v = x_0$ mod n.
6. If $r = v$, the signature is accepted, otherwise it is not verified.

To reduce the public-key size (Q, n, P, E), the users can agree on a fixed curve E and a base point P as system parameters, instead of generating different E and P for each user. After that, each user defines only the point Q.

5.3.2 Elliptic Curve ElGamal Cryptosystem

First, we describe the conventional version of the ElGamal algorithm introduced by ElGamal [1]. If Alice has to send a message m to Bob, Bob needs to have both public and private keys. Bob selects a large prime p, an integer ι mod p, and a secret integer a. He computes $v = \iota^a$ mod p. The public key for Bob consists of (p, ι, v), whereas his private key is a. Now, to encrypt the message m, Alice chooses a random integer n and computes x_B, y_B such that

$$x_B \equiv \iota^n,\ y_B \equiv mv^n (\text{mod } p).$$

After that, x_B and y_B are sent to Bob to be decrypted. The decryption process is carried out by computing

$$m \equiv y_B x_B^a (\text{mod } p).$$

On the other hand, the ElGamal elliptic curve version can be described as follows: first, Bob selects an elliptic curve E mod p, a point ι on E, and a secret integer a. He computes

$$v = a\iota.$$

The public key consists of the two points ι and ν. The secret key is the integer a. The message m is translated into a point on E by Alice. Then she chooses a random integer n and computes

$$x_{\mathrm{B}} = n\iota \text{ and } y_{\mathrm{B}} = m + n\nu.$$

Then she sends x_{B} and y_{B} to Bob. Finally, the decryption is done by computing

$$m = y_{\mathrm{B}} - ax_{\mathrm{B}}.$$

5.4 SCALABLE HARDWARE DESIGN FOR ELLIPTIC CURVE CRYPTOGRAPHY

The main operation in elliptic curve cryptography is to compute the point multiplication that consists of point additions and point doubling. As discussed earlier, computing point multiplication involves huge arithmetic operations done over the finite fields (mostly GF(p) and GF(2^n)), and therefore, an efficient ECC system requires efficient hardware implementations of finite field operations. The main two operations are modular multiplication and modular division (inverse). The proposed elliptic curve hardware design has the following two features:

1. computing point multiplication based on efficient implementation of UDMA and
2. meeting the most required two features of any efficient hardware design: being scalable and unified.

In the following subsections, UDMA and its hardware implementation are proposed.

5.4.1 Unified Division/Multiplication Algorithm

We use a novel algorithm (UDMA) [10] to compute Montgomery modular multiplication (proved to be a very efficient modular multiplication method) and modular division in GF(p) and GF(2^n) finite fields. UDMA is presented in Figure 5.1.

The UDMA mode of operation is controlled by input *Op* (*div* or *mult*), and the finite field is controlled by the input field (GF(p) or GF(2^n)). For simplicity, the polynomials $X(x)$, $Y(x)$, and $p(x)$ are denoted as X, Y, and p, respectively, which correspond to the bit-vector representation of these polynomials.

Most of the arithmetic operations in the algorithm are common to both modes of operation. The initialization of variables depends on the operation.

```
Function: Modular Division and Multiplication in GF(p) and GF(2^n)
Inputs: 0 ≤ X < p, 0 < Y < p, 2^(n-1) < p < 2^n, Field, Op, n
Output: Z = XY2^(-n) mod p when Op = mult, Z = X/Y mod p when Op = div.
Algorithm:
C = Y.
  IF Op = mult THEN                    /* Multiplication Mode */
     D = 0, U = 0, W = X, δ = n
  ELSE                                 /* Division Mode */
     D = p, U = X, W = 0,  δ = 0
  END IF;
  WHILE [(C ≠ 0 AND Op = div) OR (δ ≠ 0 AND Op = mult)]
     IF c0 = 0 THEN
        C := C >> 1
        δ := δ − 1        /* Integer Operation */
     ELSE
        k = 1
        IF (Op = div) THEN
           IF δ < 0 THEN C ⇔ D, U ⇔ W,  δ := −δ  END IF; /*Swapping */
           IF((C + D) mod 4 ≠ 0 AND Field = GF(p))THEN k = −1
           ELSE    δ :=  δ − 1   END IF;
        ELSE     /*Op = mult */
          δ :=  δ −1
        END IF;
        C := (C + k * D) >> 1, U := (U + k * W)
     END IF;
     U := (U + u0 * p) >> 1
  END WHILE;
  IF Op = div THEN Z := W      ELSE Z := U
  END IF;
```

FIGURE 5.1 Unified modular division/multiplication algorithm (UDMA) for GF(p) and GF(2^n).

For a given field, all the additions or subtractions are done in the field, besides the arithmetic operations on δ (subtractions and change of sign) which are always integer operations.

The algorithm integrates the extended binary GCD algorithm and the Montgomery multiplication algorithm and it was verified using Maple. To compute Montgomery multiplication using an n-bit modulus p, UDMA performs n iterations. The counter δ is initialized with value n, and in each iteration it is decremented by 1. The variables used in the algorithm are initialized as $C = Y$, $D = 0$, $U = 0$, and $W = X$. The result is ready ($Z = U$), when $\delta = 0$. The partial product U is reduced mod p in each iteration. In both fields, while multiplying, addition is used in the operations that update C and $D(k = 1)$. The $\gg$ operator indicates a 1-bit right shift operation.

UDMA computes modular division using the same structure used by the extended binary GCD algorithm for modular division [11]. The variables are initialized as $C = Y$, $D = p$, $U = X$, $W = 0$, and $\delta = 0$. If the division is computed in GF(p), UDMA tests the least significant 2-bits of C and D (($C + D$) mod 4 $\neq$ 0) to conditionally subtract C from D (set $k = -1$). Otherwise, C is always added to D in both fields. The division is completed when $C = 0$, and the final result is available in W. For more details about the operation of UDMA, the reader is referred to [10,12].

5.4.2 Top Level Hardware Architecture Implementing UDMA

Figure 5.2 shows the top level architecture of the unified modular divider or multiplier (let us call it UMDM) that implements UDMA. The main functional blocks are Register file, Datapath, and Control.

5.4.2.1 Register File

The register file has five registers ($R1$ to $R5$). As the computations are done in carry-save form, each intermediate variable (C, U, D, W) is represented in two vectors (sum, carry). Therefore, the registers inside the register file are designed to store two n-bit vectors. In other words, the ith register R_i is represented as $R_i = (\text{sum}, \text{carry}) = (R_i s, R_i c)$.

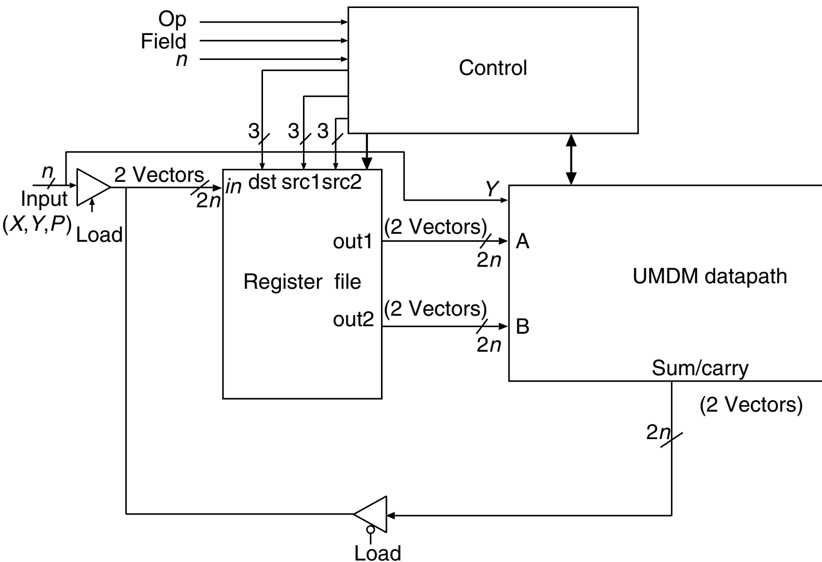

FIGURE 5.2 Top level hardware architecture of the unified modular divider/multiplier (UMDM).

The register file has one input and two output ports. The control block provides the register file with the signals necessary to perform reading or writing operations. The 3-bit signal dst determines the destination register to be written. The signals src1 and src2 (3-bits each) specify the registers to be read at output ports out1 and out2, respectively.

5.4.2.2 Datapath

The *n*-bit datapath implementing UDMA is shown in Figure 5.3. Each iteration of the algorithm is implemented in one clock cycle for multiplication mode, three clock cycles for division if *C* is odd, and two clock cycles if *C* is even, as explained later.

The proposed datapath has two inputs represented in carry-save form as $\mathrm{A} = (\mathrm{As}, \mathrm{Ac})$ and $\mathrm{B} = (\mathrm{Bs}, \mathrm{Bc})$, which receive their values from the register file ports out1 and out2, respectively. The main components of the datapath are two (3–2) unified carry-save adders (UCSAs), which are similar in complexity to full-adders [13]. The unified adders can perform bit addition with or without carry depending on the input FSEL (Field Select).

The unified adder may be used to implement a redundant or nonredundant adder. The use of nonredundant form of the operands and results reduces the register area but increases the addition time (because of carry propagation). We decided to use carry-save adders to make the addition time constant and independent of the operand's precision.

UCSAs. The first adder in the datapath is a UCSA with complement (UCSA1). Figure 5.4a shows the bit slice diagram for this adder and Figure 5.4b shows the connection of *n* slices to form an *n*-bit adder. The UCSA1 outputs are $(\mathrm{sum}, \mathrm{carry}) = a + b + c$, when $\mathrm{NEG} = C\ \ in = 0$, and

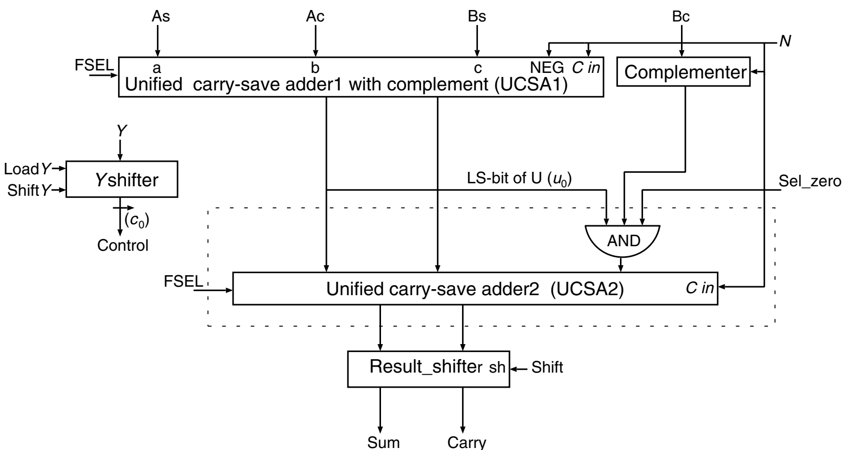

FIGURE 5.3 Unified datapath of the modular divider/multiplier (UMDM datapath).

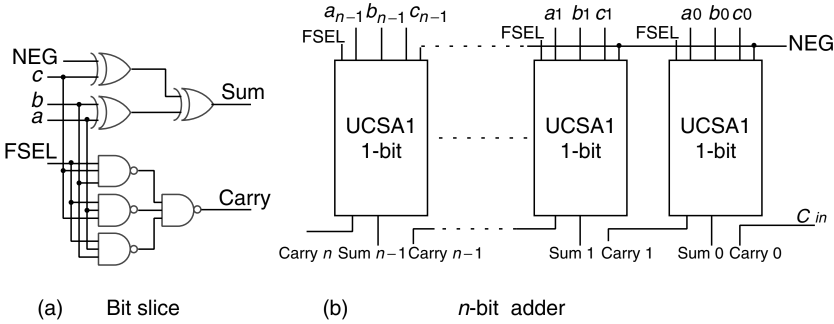

FIGURE 5.4 Unified carry-save adder with complement (UCSA1) for 1-bit and n-bit precision.

$(\text{sum}, \text{carry}) = a + b - c$, when NEG $= C\ in = 1$. Addition and subtraction in GF(2^n) are the same.

The delay of the two UCSAs, and the delay of the result_shifter ($2t_{\text{MUX}} \simeq 2t_{\text{XOR}}$), mainly determines the delay of the UMDM datapath (t_{datapath}). The delay of the AND gate is not considered because it was integrated with the second adder (shown in dashed box in Figure 5.3). As each UCSA has a delay of a full adder ($t_{\text{FA}} = 2t_{\text{XOR}}$), we get

$$t_{\text{datapath}} = t_{\text{USCA1}} + t_{\text{UCSA2}} + t_{\text{result_shifter}} = 4t_{\text{XOR}} + t_{\text{MUX}} = 5t_{\text{XOR}}.$$

The Yshifter shown in Figure 5.3 is a shift register used to implement. The operation ($C \gg 1$) in the multiplication mode is implemented by the shift register Yshifter shown in Figure 5.3. The least significant bit of the shifted C goes to the control section to be tested ($c_0 = 0$).

The datapath outputs (sum, carry) are shifted right 1-bit by correct wiring using the result_shifter at the output of the UCSA2.

5.4.2.3 Control Block

The control block provides the necessary signals to control the flow of the operations in the system. The major component in the control unit is a finite state machine that was implemented using a hardwired control methodology. With the intention to design a robust and reliable control unit, the state machine was coded as a Moore machine in which the output signals depend solely on the present state, minimizing or eliminating glitches. More implementation details can be found in [10].

The algorithm's swap functions ($C \Leftrightarrow D$ and $U \Leftrightarrow W$) are accomplished within control unit to avoid actual data transfer between registers. An actual data transfer would be costly in terms of time, especially for a system with

large precision. Thus, the swap is performed, exchanging the addresses of the register in question, inside the control unit.

Another important component of the control unit is the delta counter. This counter is used to control the swapping operation and the major algorithm control flow. The functionality for delta counter includes decrementing and negating the count value. With the goal of implementing a fast counter, a ring counter design was chosen [14].

5.4.3 Experimental Results for UMDM

The UMDM design was implemented in ASIC and FPGAs. Therefore, we present two sets of experimental data in this section.

5.4.3.1 ASIC Results for the UMDM Scalable Design

The experimental data presented in this section were generated using Mentor Graphics CAD tools. The target technology was set to AMI05_fast auto (0.5 μm CMOS with hierarchy preserved) provided in the ASIC Design Kit (ADK) from the same company [15].

The UMDM architecture was described in VHDL and simulated in ModelSim for functional correctness. It was synthesized using Leonardo synthesis tool for the mentioned technology.

Figure 5.5 shows the critical path delays (in nanoseconds) of the UMDM for the precision range from 128 to 512-bits. The maximum delay at 512-bits is around 12.8 ns.

Table 5.2 shows the total number of gates for the UMDM design as a function of operand size. The area for the UMDM design was extracted from the experimental data presented in Table 5.2 as

$$A_{\mathrm{UMDM}} = 236.12 * n + 180 = O(n) \text{ gates.}$$

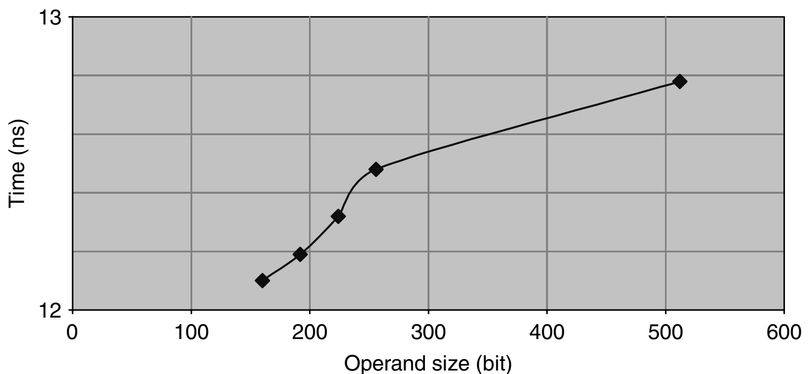

FIGURE 5.5 Critical path delays of the UMDM in nanoseconds (operand size from 160 to 512-bits).

TABLE 5.2
Area of the UMDM Design in Gates for Different Operand Sizes

Operand Size (Bits)	Area (Gates)
128-bits	30,403
160-bits	37,059
192-bits	45,513
224-bits	53,075
256-bits	60,629
512-bits	121,070

The integration of Montgomery multiplication and modular division in one design adds extra gates when compared with a dedicated divider. In the design proposed in this work, Montgomery multiplication is computed in almost the same time and complexity of a separate multiplication unit. In addition to that, this design allows the ability to compute division in the same unit with the flexibility to choose the required finite field.

5.4.3.2 FPGA Results for the UMDM Scalable Design

The scalable divider or multiplier design was synthesized for the FPGAs VertixII chip. The technology was set to $xc2vp50-7ff148$. The following paragraphs present the area and the critical path delay results obtained for the design.

Figure 5.6 shows the area synthesis results (in number of slices) of the scalable UMDM. The area is presented as function of the operand size (n)

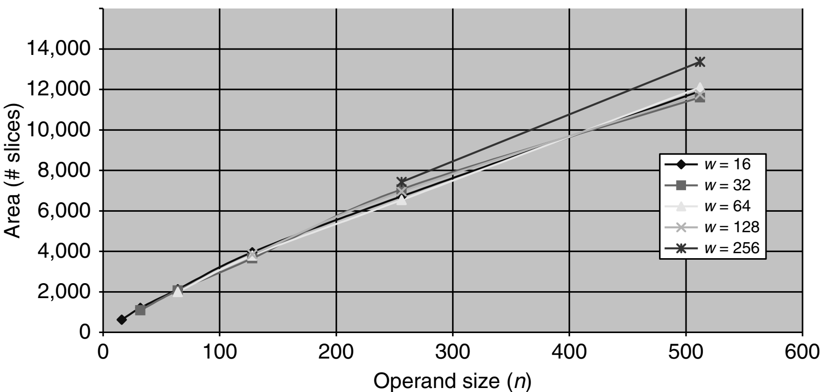

FIGURE 5.6 Area (FPGA technology) of the scalable UMDM in number of slices for combinations of operand size (n) from 16 to 512-bits and datapath word size (w) from 16 to 256-bits.

with different combinations of the datapath word sizes (w). The area results were obtained for the operand size in the range from 16 to 512-bits. The datapath word size was in the range from 16 to 256-bits. The reason why we did not use larger operand sizes is because the machines we are using could not handle operand size greater than 512-bits.

From the figure, we note that the area increases linearly as the operand size increases. There is a little difference in the number of slices when using different datapath word sizes for the same operand size.

The area for the scalable UMDM design was extracted from the experimental data presented in Figure 5.6 approximately as

$$A_{\text{scUMDM}} = 28 * n + 275 = O(n).$$

The same as in the area results, the experimental data for the critical path delay were obtained for the operand size (n) in the range from 16 to 512-bits, and the datapath word size (w) range from 16 to 256-bits. Table 5.3 shows the critical path delay (clock period) for all the possible combinations of the operand size and the datapath word size. The symbol—indicates that the combination is not possible.

The operating frequency of the UMDM design can be found by taking the reciprocal of the clock period at any point. From the table, the lowest clock period (19.83 ns) is at $n = 16$ and $w = 16$, and therefore, the maximum operating frequency is around 50 MHz.

The question now is how to choose the best design points, or in other words, the (n, w) combinations that give the lowest delay. By looking at Table 5.3, we note that at a given operand size n, the minimum delay happens at the datapath word size $w = n$. For example, the best combination at the operand size $n = 256$ happens when the word size $w = 256$ also, with a minimum delay equal to 28.4 ns.

TABLE 5.3
Critical Path Delay (Clock Period) of the Scalable UMDM in Nanoseconds for Combinations of Operand Size (16 to 512-bits) and Datapath Word Size from 16 to 256-Bits

	Datapath Word Size (w)				
Operand size (n)	**16**	**32**	**64**	**128**	**256**
16	19.83	—	—	—	—
32	24.55	22.13	—	—	—
64	25	26.55	24.7	—	—
128	32	31	27.9	25.4	—
256	34.7	37.3	34.3	31.9	28.4
512	47.15	38.71	38.5	37.4	35.4

5.5 ELLIPTIC CURVE CRYPTO-PROCESSOR OVER GF(2^n)

After introducing UDMA and its efficient hardware implementation, we propose an ECCP over the binary extension field GF(2^n) to compute the point multiplication operation kP. The ECCP architecture is based on the UDMA hardware implementation shown in the previous sections, with some simplifications applied in GF(2^n).

5.5.1 ECCP Hardware Architecture

Figure 5.7 shows the top level diagram of the ECCP. Its components are the arithmetic unit (AU) data section and control, and the main control block. The AU unit represents the UDMA architecture. The main control block interacts with the user to get the scalar multiple (k) and the point to be multiplied (P), passing them to the AU.

The details of the main blocks in the ECCP are similar to that presented in the previous sections, taking into consideration the simplifications applied to the algorithm and its implementation due to the use of GF(2^n).

The scalability feature of the proposed crypto-processor allows the adjustment of the word size used in the datapath to meet area and performance requirements. On the other hand, the processor allows the user to choose the value of the field parameter (n).

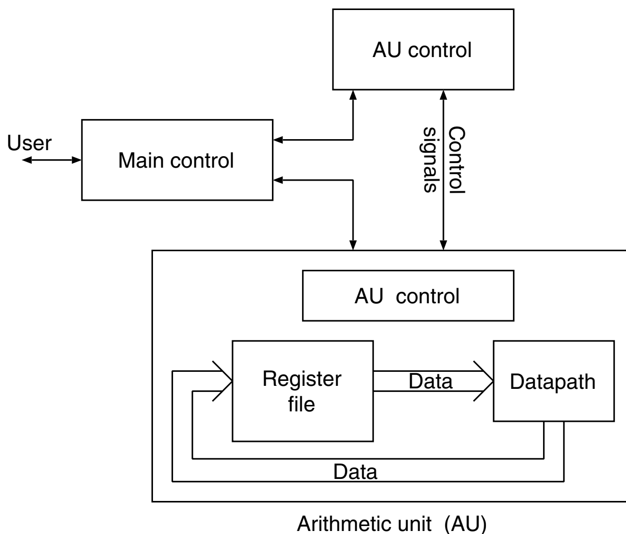

FIGURE 5.7 Top level diagram of the elliptic curve crypto-processor (ECCP).

5.5.2 Experimental Results and Analysis for GF(2^n) ECCP

As performed for the UDMA design, the experimental data presented in this section were generated using Mentor Graphics CAD tools with the target technology set to AMI05_fast auto (0.5 μm CMOS with hierarchy preserved) provided in the ADK from the same company [15]. The scalable architecture of the ECCP was described in VHDL and simulated in ModelSim to validate functional correctness. It was synthesized using Leonardo synthesis tool for the available technology.

Table 5.4 shows the critical path delays (in nanoseconds) of the ECCP for the precision range from 16 to 512-bits at different combinations of the datapath word size (from 16 to 512-bits).

We can see in the table that the minimum delay happens when the datapath word size is 16. When the word size increases, the delay increases slightly for a fixed operand precision, and the delay increases as the number of bits increases and it saturates at higher precision.

The ECCP architecture based on UDMA performs one iteration of the algorithm in each clock cycle when computing Montgomery multiplication. This means that we need n cycles to compute Montgomery modular multiplication. The ECCP has no dedicated hardware for squaring (x^2), and therefore the multiplication algorithm is used for squaring.

On the other hand, it takes a maximum of 2 iterations/bit and on an average 1.5 iterations/bit to compute the modular inverse in GF(2^n) using the simplified algorithm. The ECCP architecture performs each iteration of the algorithm in two clock cycles on an average, one to compute $(C + c_0 \cdot D)$ and another to compute $U + W$ with the modulus reduction. Therefore, the GF(2^n) inversion by the simplified algorithm takes on an average of $1.5 \times 2 = 3$ cycles for each bit.

TABLE 5.4
Critical Path Delay of the ECCP in Nanoseconds for Operand Precision 16 to 512-bits and Different Datapath Word Sizes

Delay (ns)	Datapath Word Size (w)					
Precision (bits)	16	32	64	128	256	512
16-bits	17.2	—	—	—	—	—
32-bits	17.6	17.8	—	—	—	—
64-bits	17.6	19.2	20.4	—	—	—
128-bits	17.5	19.2	20.8	20	—	—
256-bits	16.5	19.1	20.7	20.4	19	—
512-bits	16.7	18.2	20.7	20.5	19.5	20.2

In computing kP using the double-add method [7], where $P=(X_1,Y_1,Z_1)$, $Q=(X_2,Y_2,Z_2)$ are the points on the curve in the projective coordinates, we can assume that $Z_2=1$, computing point addition ($P\neq Q$) requires 13 field multiplications and computing point doubling ($P=Q$) requires 7 field multiplications [16]. To compute the scalar point multiplication (kP) using Equation 5.1, n point doubling operations are needed (n is the order of the field), and ~$n/2$ point additions are needed (given that the number of ones in the binary expansion of k is $0.5n$).

Let the total average computation time of a given design to compute multiplication or division be T_{design}, which is given by

$$T_{\text{design}} = (\text{cycles/bit}) * n * \text{clock period}.$$

At operand precision of $n=512$-bits, the time required to compute one multiplication by the ECCP is $T_{\text{mult}} = 1 * 512 * 20.2 * 10^{-9} = 10.3\ \mu\text{s}$. Then, at $n=512$-bits, the ECCP computes point addition in

$$T_{\text{P Add}} = 13 * T_{\text{mult}} \approx 134\ \mu\text{s},$$

and half of that time is required to compute point doubling $T_{\text{P Double}}=0.5 * T_{\text{P Add}}$. To compute the scalar point multiplication (kP), an inversion operation is required to transform back the result from the projective to the affine coordinates. The total time to compute the modular division (inverse) by the ECCP is $T_{\text{inv}} = 3 * T_{\text{mult}} \approx 31\ \mu\text{s}$. Then, the total time to compute kP by the proposed ECCP is

$$\begin{aligned} T_{kP} &= 0.5n * T_{\text{P Add}} + n * T_{\text{P Double}} + T_{\text{inv}} = 13/2n * T_{\text{mult}} + 7n * T_{\text{mult}} + 3T_{\text{mult}} \\ &= (13.5n+3) * T_{\text{mult}} = (13.5n+3)(n * \text{clock period}) \\ &= (13.5n^2+3n) * \text{clock period}. \end{aligned}$$

At precision $n=512$-bits, $T_{kP}=71$ ms. The proposed ECCP computes the kP faster than previously proposed elliptic curve architectures. As an example, the FPGA implementation of the elliptic curve processor presented in [17] computes the scalar point multiplication in 80.3 ms at operand size of 163-bits. In addition, the ECCP has an advantage over other designs by its scalablity (i.e., the user can choose the word size to achieve the required performance).

Table 5.5 shows the total area (in number of gates) for the ECCP design as a function of operand precision and different datapath word sizes. The area for the ECCP design was extracted from the experimental data presented in Table 5.5 as

$$A_{\text{ECCP}} = 236.12 * n + 180 = O(n) \text{ gates}.$$

TABLE 5.5
Total Area of ECCP in Gates for Operand Precision 16 to 512-bits and Different Datapath Word Sizes

Area (Gates)	Datapath Word Size (w)					
Precision (bits)	16	32	64	128	256	512
16-bits	3,857	—	—	—	—	—
32-bits	4,251	5,567	—	—	—	—
64-bits	5,012	6,267	8,945	—	—	—
128-bits	6,389	7,664	10,265	15,727	—	—
256-bits	8,212	10,310	12,928	18,274	29,434	—
512-bits	12,602	13,861	18,109	23,473	34,458	56,570

From Table 5.5, we can see that the proposed ECCP has area complexity of $O(n)$ at a given datapath word size. These results are compatible with many other designs [18,19].

REFERENCES

1. T. ElGamal, A public key cryptosystem and signature scheme based on discrete logarithms, *IEEE Transactions on Information Theory*, Vol. IT-31, No. 4, pp. 469–472, July 1998.
2. M.E. Hellman and W. Diffie, New directions on cryptography, *IEEE Transactions on Information Theory*, Vol. 22, pp. 644–654, November 1976.
3. National Institute for Standards and Technology, Digital Signature Standard (DSS), Technical Report 168–2, FIPS PUB, January 2000.
4. V. Miller, Elliptic curves in cryptography, in *Advances in Cryptology CRYPTO '85*. Editor H.C. Williams, Lecture Notes in Computer Science, No. 218, pp. 417–426, Springer 1985.
5. N. Koblitz, Elliptic curve cryptosystems, *Mathematics of Computation*, Vol. 48, No. 177, pp. 203–209, January 1987.
6. G. Seroussi, I. Blake, and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, UK, 1st ed., 1999.
7. P1363, Standard specifications for public key cryptography (draft version 13), IEEE, November 1999.
8. Adnan Abdul-Aziz Gutub, New Hardware Algorithms and Designs for Montgomery Modular Inverse Computation in Galois Fields GF(p) and GF(2^n), Ph.D. thesis, Oregon State University, Oregon, USA, June 2002.
9. W. Trappe and L.C. Washington, *Introduction to Cryptography with Coding Theory*, Prentice Hall, Englewood Cliffs, NJ, 2002.
10. L.A. Tawalbeh, A Novel Unified Algorithm and Hardware Architecture for Integrated Modular Division and Multiplication in GF(p) and GF (2^n) Suitable for Public-Key Cryptography, Ph.D. thesis, Oregon State University, Oregon, USA, October 2004.

11. A.F. Tenca and L.A. Tawalbeh, An algorithm for unified modular division in $GF(p)$ and $GF(2^n)$ suitable for cryptographic hardware, *IEE Electronics Letters*, Vol. 40, No. 5, pp. 304–306, March 2004.
12. L.A. Tawalbeh and A.F. Tenca, An algorithm and hardware architecture for integrated modular division and multiplication in $GF(p)$ and $GF(2^n)$, in *The IEEE 15th International Conference on Application-Specific Systems, Architecture, and Processors (ASAP)*. September 27–29, 2004, pp. 247–257, ieeecs.
13. E. Savas, A.F. Tenca, and Ç.K. Koç, A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$, in *Cryptographic Hardware and Embedded Systems—CHES 2000*, Ç.K. Koç and C. Paar, Eds. 2000, Lecture Notes in Computer Science, No. 1717, pp. 281–296, Springer, Berlin, Germany.
14. M. Stan, A. Tenca, and M. Ercegovac, Long and fast up/down counters, *IEEE Transactions on Computers*, Vol. 47, No. 7, pp. 722–734, July 1998.
15. ASIC Design Kit. Mentor Graphics Co, http://www.mentor.com/partners/hep/AsicDesignKit/dsheet/ami05databook.html
16. 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*, Vol. 11, No. 5, pp. 804–813, 1993.
17. G. Orlando and C. Paar, Implementation of elliptic curve cryptographic Coprocessor over $GF(2^m)$ on an FPGA, in *Cryptographic Hardware and Embedded Systems—CHES 2000*, Ç.K. Koç and C. Paar, Eds. 2000, Lecture Notes in Computer Science, No. 2162, pp. 25–40, Springer, Berlin, Germany.
18. J. Goodman and A.P. Chandrakasan, An energy-efficient reconfigurable public-key cryptography processor, *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, pp. 1808–1820, November 2001.
19. J. Wolkerstorfer, Dual-field arithmetic unit for $GF(P)$ and $GF(2^n)$, in *Cryptographic Hardware and Embedded Systems—CHES 2002*, B.S. Kaliski Jr. et al., Eds. 2003, Lecture Notes in Computer Science, No. 2523, pp. 484–499, Springer, Berlin, Germany.