Elliptic-Curve-Based Security Processor for RFID

Yong Ki Lee, *Student Member*, *IEEE*, Kazuo Sakiyama, *Member*, *IEEE*, Lejla Batina, *Member*, *IEEE*, and Ingrid Verbauwhede, *Senior Member*, *IEEE*

Abstract—Radio Frequency IDentification (RFID) tags need to include security functions, yet at the same time, their resources are extremely limited. Moreover, to provide privacy, authentication, and protection against tracking of RFID tags without losing the system scalability, a public-key-based approach is inevitable. In this paper, we present an architecture of a state-of-the-art processor for RFID tags with an Elliptic Curve (EC) processor over $\mathrm{GF}(2^{163})$. It shows the plausibility of meeting both security and efficiency requirements even in a passive RFID tag. The proposed processor is able to perform EC scalar multiplications and general modular arithmetic (additions and multiplications), which are needed for the cryptographic protocols. As we work with large numbers, the register file is the most critical component in the architecture. By combining several techniques, we are able to reduce the number of registers from nine to six in the EC processor. To obtain an efficient modulo arithmetic, we introduce a redundant modular operation. Moreover, the proposed architecture can support multiple cryptographic protocols. The synthesis results with a 0.13-$\mu\mathrm{m}$ CMOS technology show that the gate area of the most compact version is 12.5 Kgates.

Index Terms—RFID systems, security processor, elliptic curve cryptography, scalable hardware, arithmetic and logic units, public-key cryptosystems.

1 Introduction

Designing a Radio Frequency IDentification (RFID) system is one of the most challenging tasks since it requires compact and power-efficient solutions, especially when security-related processing is needed. The most commonly required security properties are anticloning and untractability. Besides these security properties, the systems should be scalable since the number of tags can be very large. For example, it can be millions for large libraries or warehouses. To satisfy those security and system requirements, it is proven that a public-key cryptosystem is necessary [1]. An Elliptic Curve (EC)-based cryptosystem would be one of the best candidates for the RFID systems due to its small key size and efficient computation.

In this paper, the proposed RFID processor is composed of a microcontroller, an EC processor (ECP), and a bus manager, where the ECP is over $\mathrm{GF}(2^{163})$. For an efficient computation with restrictions on the gate area and the number of cycles, several techniques are introduced in the algorithms and the architecture level. The optimization techniques can be considered in two parts: the ECP and the microcontroller. First, noting that the ECP is dominated by the registers due to a large key size of 163 bits, the optimization of the ECP is mostly concentrated on the register file. By proposing a common Z-coordinate system (and its corresponding formulas) and by introducing a register reuse technique, we reduce the number of registers from nine to six. In addition, we design a new register file architecture that reduces around 30 percent of gate area of the register file with small overhead in cycles. Second, the microcontroller is designed to perform general modular operations efficiently. For efficient general modular operations, we propose a redundant representation that results in an efficient computation with less memory compared to conventional methods.

Those techniques result in the most compact ECP of 10.1 Kgates with 276 Kcycles for one point multiplication. The ECP is attached to the micro controller of a tag. General modular operations are also needed for the computation of the authentication protocols. In general, the minimally required operations are modular additions and multiplications. The modular additions and multiplications take 574 cycles and 25 Kcycles, respectively, for a word size of 163 bits. Since the modular operations can be performed in parallel with the EC scalar multiplication, the former operations do not contribute to the latency. As a result, the overall architecture takes 12.5 Kgates. The architecture is also scalable in the digit size of the ECP (the ECP's ALU performs the field multiplication in digit serial), and hence, a better performance can be easily obtained. We also demonstrate the proposed processor for an EC-based authentication protocol.

The remainder of this paper is organized as follows: In Section 2, an overview of arithmetic operations for EC Cryptography (ECC) is introduced, and in Section 3, the starting points are summarized. The system overview of the proposed RFID processor architecture is shown in Section 4, and several techniques to minimize the ECP are described

- *Y.K. Lee is with the Electrical Engineering Department, University of California, 56-125B Engineering IV Building, 420 Westwood Plaza, Los Angeles, CA 90095-1594. E-mail: jfirst@ee.ucla.edu.*
- *K. Sakiyama is with the Department of Information and Communication Engineering, University of Electro-Communications, 1-5-1 Chofugaoka, Chofu, Tokyo 182-8585, Japan. E-mail: saki@ice.uec.ac.jp.*
- *L. Batina is with the Katholieke Universiteit Leuven, ESAT/SCD, Kasteelpark Arenberg 10, Bus 2446, B-3001 Leuven-Heverlee, Belgium. E-mail: Lejla.Batina@esat.kuleuven.be.*
- *I. Verbauwhede is with the Katholieke Universiteit Leuven, ESAT/SCD, Kasteelpark Arenberg 10, Bus 2446, B-3001 Leuven-Heverlee, Belgium, and also with the University of California, Los Angeles, CA 90095-1594. E-mail: Ingrid.Verbauwhede@esat.kuleuven.be.*

Manuscript received 16 Nov. 2007; revised 10 June 2008; accepted 7 July 2008; published online 12 Aug. 2008.
Recommended for acceptance by R. Steinwandt, W. Geiselmann, and Ç.K. Koç.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-11-0603.
Digital Object Identifier no. 10.1109/TC.2008.148.

in Section 5. The architecture and the instructions of the RFID microcontroller are given in Section 6, and the synthesis results and the performance analysis are summarized in Section 7 followed by the conclusion in Section 8.

2 Overview of Arithmetic Operations for Elliptic Curve Cryptography

ECC includes protocols that are based on the arithmetic of ECs. Curves that are commonly used in applications are usually defined over $\mathrm{GF}(p)$ or $\mathrm{GF}(2^n)$, where p is a prime number. EC systems over both types of fields provide the same level of security, but the so-called binary fields have some implementation advantages. Namely, binary arithmetic is "carry-free," squaring can be implemented very efficiently in some cases, etc. The properties are very convenient for hardware implementations. Binary fields offer also more arithmetic options as there are many choices for bases, irreducible polynomials, fields, etc. In general, the EC arithmetic consists of several hierarchical levels. The top level is EC scalar multiplication, which is executed by point addition and doubling. The point operations can be performed by different formulas, which depend on the representation chosen, i.e., coordinates. The formulas for point arithmetic are sequences of finite field operations: addition/subtraction, multiplication/squaring, and inversion.

2.1 EC Scalar Multiplication

All ECC protocols include one or a few scalar or point multiplications. This operation is achieved by repeated point additions and doublings. The basic algorithm for scalar multiplication is the so-called binary method [2].

Algorithm 1. Scalar multiplication: Binary method [2]

```
Require: A point $P$, a $t$-bit integer $k$, $k = (k_{t-1}, k_{t-2}, \dots k_0)_2$,
         $k_i \in \{0, 1\}$
Ensure: $Q = kP$
1: $Q \leftarrow O$;
2: for $i$ from $t-1$ down to 0 do
3:     $Q \leftarrow 2Q$;
4:     If $k_i = 1$, then $Q \leftarrow Q + P$;
5: end for
6: Return $Q$;
```

For scalar multiplication, one often chooses the Montgomery ladder [3]. In the Montgomery ladder, the computation is balanced and independent of k_i in the iteration, and therefore, it is secure against simple side-channel attacks.

Algorithm 2. Montgomery ladder [3]

```
Require: a $t$-bit integer $k > 0$ and a point $P$
Ensure: $kP$
1: $k \leftarrow 1, k_{t-2}, \dots, k_1, k_0$;
2: $P_1 \leftarrow P$, $P_2 \leftarrow 2P$;
3: for $i$ from $t-2$ down to 0 do
4:     If $k_i = 1$ then $P_1 \leftarrow P_1 + P_2$, $P_2 \leftarrow 2P_2$;
5:         else $P_2 \leftarrow P_2 + P_1$, $P_1 \leftarrow 2P_1$;
6: end for
7: Return $P_1$;
```

1) **Common Input:** The set of system parameters consists of: (q, a, b, P, n, h). Here, q specifies the finite field, a, b, define an elliptic curve, P is a point on the curve of order n and h is the cofactor.
2) **Prover-Tag Input:** The secret k such that $Z = -k \cdot P$.
3) **Protocol:** It involves exchange of the following messages:

Prover P		Verifier V
$r \in_R \mathbb{Z}_n$		
$X \leftarrow r \cdot P$	$\xrightarrow{X}$	
	$\xleftarrow{e}$	$e \in_R \mathbb{Z}_n$
$y = (ke + r) \bmod n$	$\xrightarrow{y}$	
		If $y \cdot P + e \cdot Z = X$ then accept, else reject.

Fig. 1. Schnorr's identification protocol.

For the point operation, there exist several formulas, depending on the choice of coordinates. For example, formulas based on affine coordinates and projective coordinates can be found in [6] and [9], respectively. Projective coordinates are commonly used to avoid the field inversion.

2.2 General Modular Arithmetic Operation

Besides the EC scalar multiplier, general modular arithmetic operations are required to perform the cryptographic protocols. For example, the Schnorr protocol [4] is shown in Fig. 1. When the prover calculates $y (= ke + r \bmod n)$, general modular operations (multiplication and addition) should be performed. General modular operations include also the reduction operation. Efficient reduction is possible for Mersenne primes, but since n is the order of a curve, the reduction should work for an arbitrary n. In this case, the reduction needs more computation than the addition and multiplication themselves.

There exist efficient reduction algorithms such as Montgomery's reduction algorithm [7] and Barrett's algorithm [8]. However, since Montgomery's algorithm requires the transformation overhead, it is not convenient to use it in this situation. In Barrett's algorithm, the reduction can be performed after calculating the multiplication and the quotient. Hence, it requires temporary memory of five times the size of n. Considering the scarceness of resources in RFID systems, the required memory should be minimized.

In this paper, we propose a redundant representation based on the addition of a few guard bits for general modular operations, which is efficient and requires a small temporary memory, as will be explained in detail in Section 6.

3 Starting Points

In this section, we describe two building blocks that are our starting points. The first one is the Montgomery ladder with the López-Dahab algorithm. This approach allows an implementation that does not need the storage of the Y-coordinate. The second one is a compact arithmetic unit to perform the field operations.

TABLE 1
López-Dahab's Addition and Doubling Algorithms

Addition Algorithm $(X_1, Z_1) \leftarrow \text{Madd}(X_1, Z_1, X_2, Z_2)$	Doubling Algorithm $(X, Z) \leftarrow \text{Mdouble}(X, Z)$
1. $T_1 \leftarrow x$	1. $T_1 \leftarrow c$
2. $X_1 \leftarrow X_1 \cdot Z_2$	2. $X \leftarrow X^2$
3. $Z_1 \leftarrow Z_1 \cdot X_2$	3. $Z \leftarrow Z^2$
4. $T_2 \leftarrow X_1 \cdot Z_1$	4. $T_1 \leftarrow Z \cdot T_1$
5. $Z_1 \leftarrow Z_1 + X_1$	5. $Z \leftarrow Z \cdot X$
6. $Z_1 \leftarrow Z_1^2$	6. $T_1 \leftarrow T_1^2$
7. $X_1 \leftarrow Z_1 \cdot T_1$	7. $X \leftarrow X^2$
8. $X_1 \leftarrow X_1 + T_2$	8. $X \leftarrow X + T_1$

3.1 Montgomery Ladder with the López-Dahab Algorithm

The Montgomery ladder with López-Dahab's algorithm shown in Algorithm 3 uses a projective coordinate system. The point addition formulas of $(X_{Add}, Z_{Add}) \leftarrow \text{Madd}(X_1, Z_1, X_2, Z_2)$ are defined by

$$\begin{aligned} Z_{Add} &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2, \\ X_{Add} &= x \cdot Z_{Add} + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1). \end{aligned} \tag{1}$$

Algorithm 3. Montgomery ladder with the López-Dahab algorithm [9]

```
Require: An EC $y^2 + xy = x^3 + ax^2 + b$, a point $P$, a $t$-bit
    integer $k$, $k = (1, k_{t-2}, \ldots, k_0)_2$, $k_i \in \{0, 1\}$
Ensure: $Q = kP$
 1: If ($k = 0$ or $x = 0$) then output (0, 0) and stop;
 2: $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$;
 3: for $i$ from $t - 2$ down to 0 do
 4:     If $k_i = 1$ then
 5:         $(X_1, Z_1) \leftarrow \text{Madd}(X_1, Z_1, X_2, Z_2)$,
 6:         $(X_2, Z_2) \leftarrow \text{Mdouble}(X_2, Z_2)$;
 7:     else $(X_2, Z_2) \leftarrow \text{Madd}(X_2, Z_2, X_1, Z_1)$,
 8:         $(X_1, Z_1) \leftarrow \text{Mdouble}(X_1, Z_1)$;
 9: end for
10: Return $Q \leftarrow \text{Mxy}(X_1, Z_1, X_2, Z_2)$;
```

The doubling formulas of $(X_{Double}, Z_{Double}) \leftarrow \text{Mdouble}(X_2, Z_2)$ for the case of $k_i = 0$ are defined by

$$\begin{aligned} Z_{Double} &= (X_2 \cdot Z_2)^2, \\ X_{Double} &= X_2^4 + b \cdot Z_2^4. \end{aligned} \tag{2}$$

$Q \leftarrow \text{Mxy}(X_1, Z_1, X_2, Z_2)$ is the conversion from projective coordinate to affine coordinate. López-Dahab's addition and doubling algorithms are described in Table 1, where $c^2 = b$.

The total number of registers required in Table 1 is six, which is for storage of X_1, Z_1, X_2, Z_2, T_1, and T_2. The required field operations for the Addition Algorithm are four multiplications, one squaring, and two additions, and for the Doubling Algorithm, one needs two multiplications, four squarings, and one addition. Note that it is not necessary to maintain the Y-coordinate during the iterations since it can be derived at the end of the computation.

3.2 Modular Arithmetic Logic Unit (MALU)

In order to perform the field operations, i.e., the multiplications, squarings, and additions in Table 1, we need a Modular Arithmetic Logic Unit (MALU). The MALU architecture is a compact architecture that performs the arithmetic field operations shown as follows [12]:

$$\begin{aligned} A(x) &= B(x) \cdot C(x) \bmod P(x), \qquad \text{if } cmd = 1, \\ A(x) &= A(x) + C(x) \bmod P(x), \qquad \text{if } cmd = 0, \end{aligned} \tag{3}$$

where $A(x) = \Sigma a_i x^i$, $B(x) = \Sigma b_i x^i$, $C(x) = \Sigma c_i x^i$, and $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

In the MALU, the squaring operation uses the same logic as the multiplication by duplicating the operand. With a digit size of d, the field multiplication and addition take $\lceil \frac{163}{d} \rceil$ and one cycle, respectively. The benefit of this architecture is that the multiplication and addition operations share the XOR array, and by increasing the digit size, the MALU can be easily scaled. More explanation about the MALU is given in Section 5.

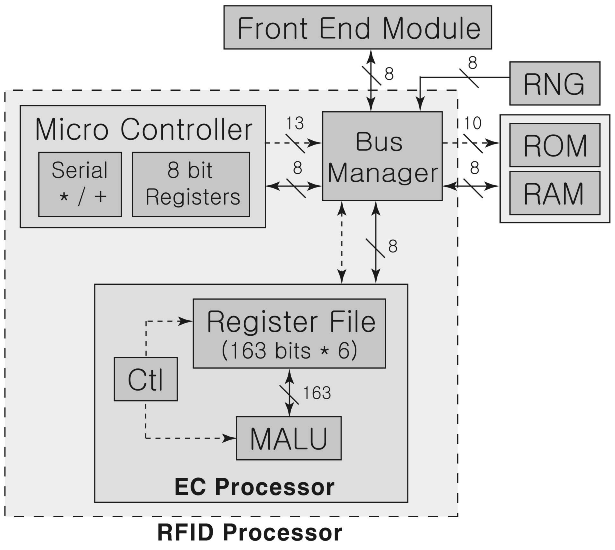

Fig. 2. Proposed RFID processor architecture.

4 System Overview

The RFID processor is composed of a microcontroller, a bus manager, and an ECP. It is connected with a front-end module, a random number generator (RNG), ROM, and RAM, as illustrated in Fig. 2. A front-end module includes an antenna, an Analog-to-Digital (A/D) converter, a modulator, and a demodulator and provides an interface with the bus manager on an 8-bit bus. The solid lines are for data exchange, the dash lines with numbers are for addressing, and the dash lines without numbers are control signals.

The ROM stores program codes and data. The program is executed by the microcontroller and the data may include a tag's private key, the server's public key and system parameters. The program is basically an authentication protocol. Hardwiring of an authentication protocol is not a flexible solution because protocols are often changed due to a constant progress in cryptanalysis. Therefore, we choose a design that can be programmed for several authentication protocols even after ASIC is produced. The private key and the system parameters should be changeable since the private key of a tag should be different for each tag, and the system parameters could be different, depending on RFID system users. Therefore, the used ROM should be EPROM or EEPROM. RAM is used to store intermediate or final results of calculations by the microcontroller and the ECP. Even though we exclude RAM and ROM from our design, they should be embedded in the design when the actual ASIC is produced.

The microcontroller is able to perform general modular arithmetic operations (additions and multiplications) in a byte-serial fashion. It also gives commands for the execution of the ECP via the bus manager.

The ECP loads a key (k) and an EC point (P) from ROM or RAM and executes the EC scalar multiplication (kP). After finishing the scalar multiplication, it stores the results in RAM.

The bus manager takes a role as the bridge for the data flow from/to outside of the RFID processor. It also arbitrates the memory access of the microcontroller and the ECP. A higher priority is given on the ECP than the microcontroller since the execution time is more critical in the former. For this priority setting, the ECP signals the bus manager in advance of memory access.

5 Elliptic Curve Processor

5.1 Implementation Considerations

If López-Dahab's algorithm is implemented based on the MALU in a conventional way, the total number of registers is nine, i.e., three registers for the MALU plus six registers for the Montgomery ladder algorithm. In [13], three registers and five RAMs are used (eight memory elements in total). One register is reduced by modifying López-Dahab's algorithm and assuming that constants can be loaded directly to the MALU without using a register. In our architecture, we are able to reduce the number of registers to six even without relying on these assumptions. As the registers occupy more than 80 percent of the gate area in a conventional architecture, reducing the number of the registers and the complexity of the register file are very effective to minimize the total gate area.

Accordingly, our compact architecture achieves two things: reducing the number of registers (one register reduction by using the common Z projective coordinate system and two registers reduction by register reuse) [14] and designing a compact register file architecture by limiting the access to the registers.

5.2 Common Z Projective Coordinate System

We propose new formulas for the common Z projective coordinate system where the Z values of two EC points in the Montgomery ladder algorithm are kept the same during the process. New formulas for the common Z projective coordinate system have been proposed over prime fields in [10]. However, this work is different from ours. First, they made new formulas over a prime field, while ours is over a binary field. Second, they made new formulas to reduce the computation in a special addition chain, while our formulas slightly increase the computation amount in order to reduce the number of the registers. Again, note that reducing even one register decreases the total gate area considerably.

Since in López-Dahab's algorithm, two EC points must be maintained during EC scalar multiplication, the required number of registers is four (X_1, Z_1, X_2, and Z_2), and including two registers for intermediate values (T_1 and T_2), the total number of registers is six. The idea of the common Z projective coordinate system is to make sure that $Z_1 = Z_2$ at each iteration of López-Dahab's algorithm. This condition is satisfied at the beginning of the iterations since the algorithm starts from $Z_1 = Z_2 = 1$. Even if $Z_1 \neq Z_2$, we can satisfy this condition using three extra field multiplications, as shown in the following equation, where the resulting coordinate set is (X_1, X_2, Z):

$$\begin{aligned} X_1 &\leftarrow X_1 \cdot Z_2, \\ X_2 &\leftarrow X_2 \cdot Z_1, \\ Z &\leftarrow Z_1 \cdot Z_2. \end{aligned} \tag{4}$$

TABLE 2
Comparison between the Original and the Modified Formulas

The original equation	The new equation assuming $Z = Z_1 = Z_2$
$Z_{Add} = (X_1 Z_2 + X_2 Z_1)^2$ $X_{Add} = x Z_{Add} + X_1 Z_2 X_2 Z_1$	$Z_{Add} = (X_1 + X_2)^2$ $X_{Add} = x Z_{Add} + X_1 X_2$

Since we now assume that $Z_1 = Z_2$, we can start the point addition algorithm with the common Z projective coordinate system. With $Z = Z_1 = Z_2$, (1) is rewritten as shown by (5). Now, Z_{Add} and X_{Add} have a common factor of Z^2:

$$\begin{aligned} Z_{Add} &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2 = (X_1 + X_2)^2 \cdot Z^2, \\ X_{Add} &= x \cdot Z_{Add} + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1) \\ &= x \cdot Z_{Add} + (X_1 \cdot X_2 \cdot Z^2). \end{aligned} \tag{5}$$

Due to the property of projective coordinate systems, we can divide Z_{Add} and X_{Add} by the common factor Z^2. The comparison of the original equation with the modified equation is summarized in Table 2. Note that the new formula of the point addition algorithm is independent of the previous Z-coordinate value.

In the point doubling algorithm, there is no such reduction since it deals with only one EC point. Nevertheless, we can simplify the point doubling algorithm by noticing that $T_1^2 + X^2 \equiv (T_1 + X)^2$ at steps 6, 7, and 8 in Table 1. One field multiplication can be reduced using this mathematical equality. Equation (2) is rewritten as follows, where $c^2 = b$:

$$\begin{aligned} Z_{Double} &= (X_2 \cdot Z)^2, \\ X_{Double} &= \left(X_2^2 + c \cdot Z^2\right)^2. \end{aligned} \tag{6}$$

Note that the resulting Z-coordinate values are different between the point addition and doubling formulas. In order to maintain a common Z-coordinate value, extra steps similar to (4) are required. These extra steps must follow every pair of the point addition and doubling algorithms. The final mathematical expression and its algorithm are shown by the following equation and in Table 3, respectively:

$$\begin{aligned} X_1 &\leftarrow X_{Add} Z_{Double} = \left\{x(X_1 + X_2)^2 + X_1 X_2\right\}(X_2 Z)^2, \\ X_2 &\leftarrow X_{Double} Z_{Add} = \left(X_2^2 + cZ^2\right)^2 (X_1 + X_2)^2, \\ Z &\leftarrow Z_{Add} Z_{Double} = (X_1 + X_2)^2 (X_2 Z)^2. \end{aligned} \tag{7}$$

In Table 3, the mark of (T_1) at each squaring operation indicates that the T_1 register is free to store some other

TABLE 3
Proposed Point Addition and Doubling Algorithms

Addition Algorithm	Doubling Algorithm	Extra Steps
1. $T_2 \leftarrow X_1 + X_2$	1. $X_2 \leftarrow X_2^2\ (T_1)$	1. $X_1 \leftarrow X_1 \cdot Z$
2. $T_2 \leftarrow T_2^2\ (T_1)$	2. $Z \leftarrow Z^2\ (T_1)$	2. $X_2 \leftarrow X_2 \cdot T_2$
3. $T_1 \leftarrow X_1 \cdot X_2$	3. $T_1 \leftarrow c$	3. $Z \leftarrow Z \cdot T_2$
4. $X_1 \leftarrow x$	4. $T_1 \leftarrow Z \cdot T_1$	
5. $X_1 \leftarrow T_2 \cdot X_1$	5. $Z \leftarrow Z \cdot X_2$	
6. $X_1 \leftarrow X_1 + T_1$	6. $X_2 \leftarrow X_2 + T_1$	
	7. $X_2 \leftarrow X_2^2\ (T_1)$	

value. The reason for this will be explained in this section. The comparison of the amount of field operations between López-Dahab's algorithm and our algorithm is shown in Table 4.

Noting that the multiplication and the squaring are equivalent in the MALU operation, the workload of our algorithm is the same as that of López-Dahab's algorithm, and we still reduce the storage by one register. Moreover, if an EC with $b = 1$ is chosen in $y^2 + xy = x^3 + ax^2 + b$, one additional multiplication can be saved in the point doubling algorithm. In our work, a is randomly selected, and $b = 1$.

5.3 ECP's MALU Architecture

The MALU architecture of the ECP, which reuses the MALU in [12], is shown in Fig. 3. The registers in the MALU are combined with the external registers to reduce the number of registers. At the completion of the multiplication or addition operation, only register RegA is updated, while registers RegB and RegC hold the same data as at the beginning of the operations. Therefore, RegB and RegC can be used not only to store field operands but also to store some intermediate values of the proposed point addition and doubling algorithm where we need five registers for X_1, X_2, Z, T_1, and T_2 in Table 3.

An extra care should be taken at this point since the same value must be placed in both of RegB and RegC for squaring. Therefore, during squaring, only one register can be reused. This fact would conflict with our purpose to

TABLE 4
Comparison of the Computational Workload

Field Operation	López-Dahab's algorithm	Our algorithm
Multiplication	6	7
Squaring	5	4
Addition	3	3

reuse each of RegB and RegC as a storage of the point addition and doubling algorithms. Fortunately, it is possible to free one of the registers to hold another value during squaring. As shown in Table 3, T_1 can be reused whenever a squaring operation is required.

In Fig. 3, cmd signals the command to perform multiplication or addition as shown by (3). When the MALU performs a multiplication, each digit of d bits of RegB must be provided to the MALU. Instead of addressing each digit of the 163-bit word, the most significant digit (MSD) is provided, and a circular shift is performed by d bits. The shift operation must be circular, and the last shift must be the remainder of $163/d$ since the value must be kept as the initial value at the end of the operation. During the MALU operation, an intermediate result is stored in RegA.

5.4 Circular Shift Register File Architecture

By reusing the MALU registers for the Montgomery ladder algorithm, we reduce two of the registers as discussed in the previous section. This means that all the registers of the MALU and the Montgomery ladder algorithm should be organized in a single register file. Therefore, the register file of our system consists of six registers. We use a circular shift register file to reduce the complexity of the multiplexer. The area complexity of a multiplexer in a randomly accessible register file increases as the square of the number of registers. On the other hand, the area complexity of the multiplexer in a circular shift register file is a constant. As a result, this model reduces about 30 percent of the gate area of the register file.

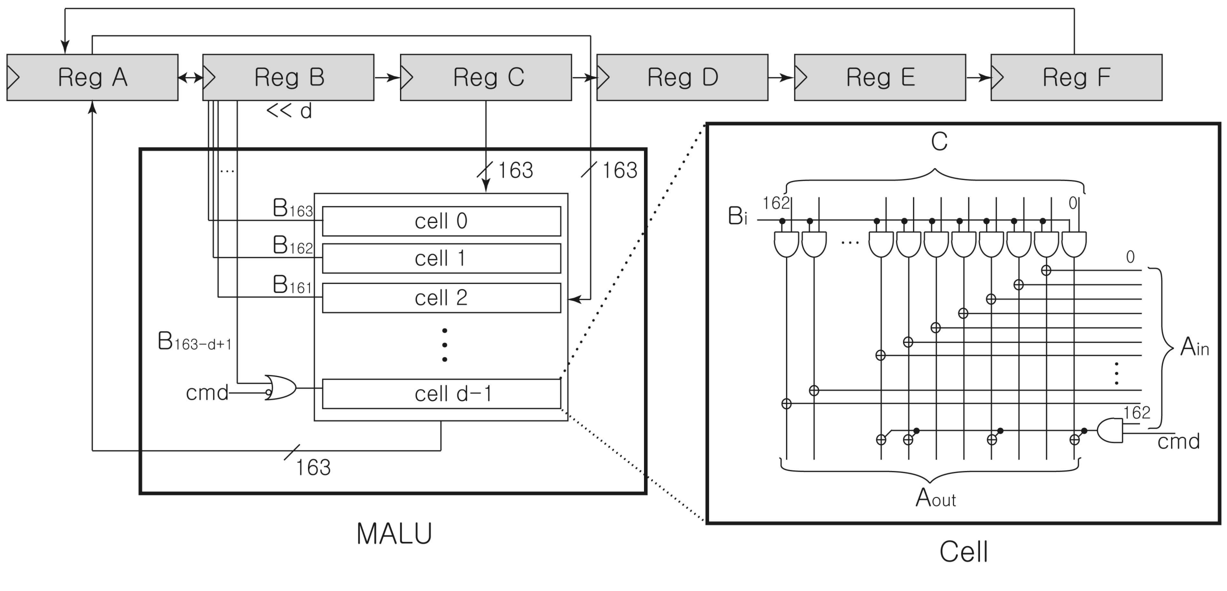

Fig. 3. ECP's MALU architecture.

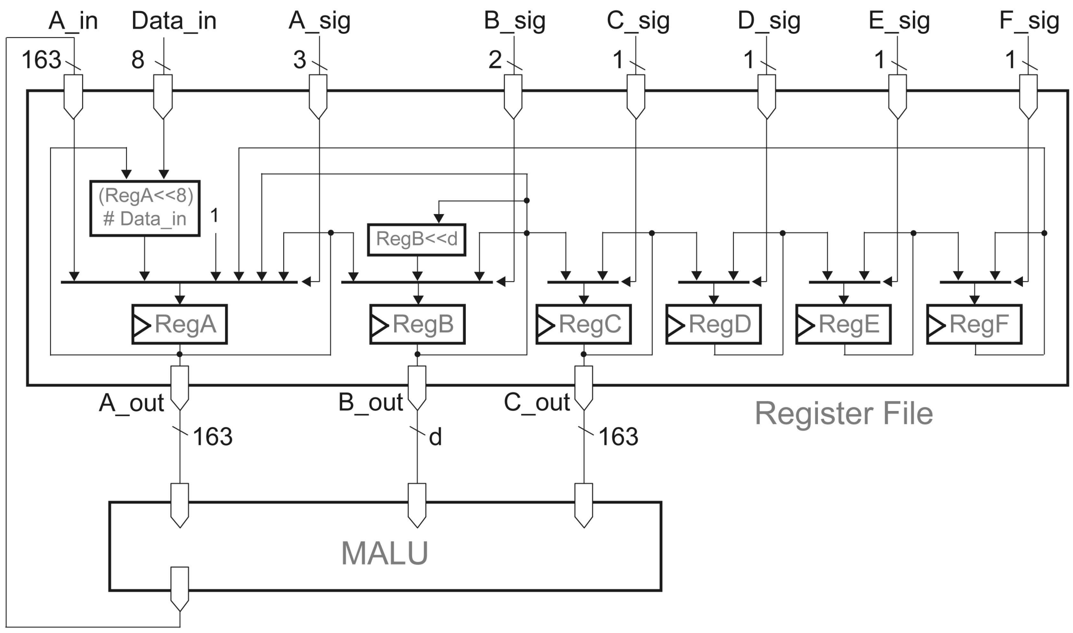

Fig. 4. Register file architecture.

Although the register file in Fig. 4 is a circular shift register file, each register is independently controlled for efficient management. RegA is the only register that gets values from outside of the register file. Data_in is for the data from a memory unit. This data can be either a scalar k or a point P for EC point multiplications of $k \cdot P$. Since the data is input as 8 bits, RegA performs 8-bit shifts to keep the previously loaded data. The signal "1" is for the initialization of the Z-coordinate value. In the multiplexer for RegB, the shift of "d" positions is a circular shift so that RegB goes back to the original value after finishing the field multiplication. Except for RegA and RegB, all the registers can be updated only by the preceding register. Note that the multiplexers for RegC, RegD, RegE, and RegF are not implemented since the enable signals of flip-flops can be utilized to indicate whether to update with new values or to keep their previous values.

With the given multiplexers, any replacement or reordering of the register values can be achieved. Since only RegA and RegB get multiple inputs, only two fixed-size multiplexers are necessary. Note that RegA, RegB, and RegC in Fig. 4 are used as the three registers for the MALU in Fig. 3.

5.5 EC Processor (ECP) Architecture

The ECP architecture is shown in Fig. 5. EC point add and doubler (EC Add&Doubler) consists of Control1, the MALU, and the register file. Control1 receives the EC parameters and gives the result of EC scalar multiplication through Control2. Control2 conveys the data from/to Control1 and reads a key (or a scalar) through the bus manager. The key is read in bytes and stored in a 1-byte buffer in Control2. Control2 also controls the EC Add&Doubler according to the Montgomery algorithm in Algorithm 3.

In our system, we assume that the coordinate conversion to the affine coordinate system and the calculation of Y-coordinate value are performed on a tag reader or a back-end system.

5.6 Register File Management for Algorithm Implementation

The register file management for the point addition algorithm is shown in Table 5. Each step requires one cycle except for the field multiplications and the read operation of x (step 14). The read operation of x requires 28 cycles, which is composed of seven cycles for the synchronization with the bus manager and 21 cycles for the reading of 21 bytes.

For the field multiplication, only the final results are shown. At the beginning of the operation, we assume that X_2, X_1, and Z are stored in RegA, RegB, and RegC, respectively. RegD, RegE, and RegF are marked with "$-$" since meaningful values are not stored yet. On each step of the register file management, each register value is updated

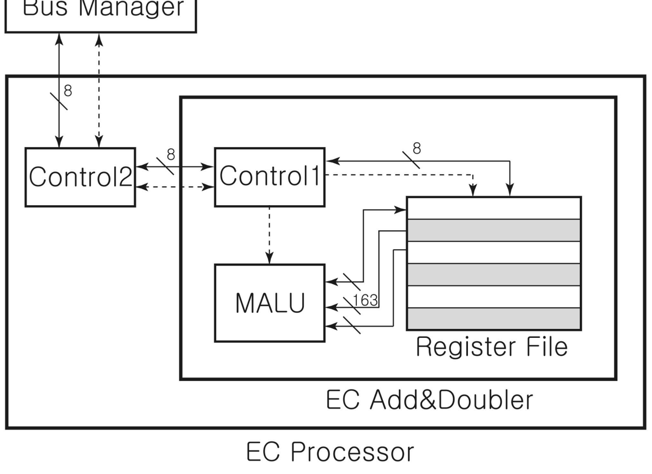

Fig. 5. ECP architecture.

TABLE 5
Register File Management for the Addition Algorithm

Step	Field Operation	RegA	RegB	RegC	RegD	RegE	RegF	cycles
(1)	Initial	X_2	X_1	Z	–	–	–	–
(2)		X_2	X_2	X_1	Z	–	–	1
(3)	1. $T_2 \leftarrow X_1 + X_2$	T_2	X_2	X_2	X_1	Z	–	1
(4)		T_2	T_2	X_2	X_1	Z	Z	1
(5)		T_2	T_2	T_2	X_2	X_1	Z	1
(6)	2. $T_2 \leftarrow T_2^2$	T_2	–	–	X_2	X_1	Z	$\lceil 163/d \rceil$
(7)		Z	T_2	–	X_2	X_2	X_1	1
(8)		X_1	Z	T_2	X_2	X_2	X_2	1
(9)		X_2	X_1	Z	T_2	X_2	X_2	1
(10)		X_2	X_2	X_1	Z	T_2	X_2	1
(11)	3. $T_1 \leftarrow X_1 \times X_2$	T_1	X_2	–	Z	T_2	X_2	$\lceil 163/d \rceil$
(12)		X_2	T_1	–	Z	Z	T_2	1
(13)		X_2	X_2	T_1	Z	Z	T_2	1
(14)	4. $X_1 \leftarrow x$	X_1	X_2	T_1	Z	Z	T_2	28
(15)		T_2	X_1	X_2	T_1	Z	T_2	1
(16)		T_2	X_1	X_2	T_1	T_1	Z	1
(17)		T_2	T_2	X_1	X_2	T_1	Z	1
(18)	5. $X_1 \leftarrow T_2 \times X_1$	X_1	T_2	–	X_2	T_1	Z	$\lceil 163/d \rceil$
(19)		Z	X_1	T_2	X_2	T_1	T_1	1
(20)		X_1	Z	T_2	X_2	X_2	T_1	1
(21)		T_1	X_1	Z	T_2	X_2	T_1	1
(22)		T_1	X_1	X_1	Z	T_2	X_2	1
(23)	6. $X_1 \leftarrow X_1 + T_1$	X_1	–	–	Z	T_2	X_2	1

according to the architecture in Fig. 4. While RegA can be updated by either RegB or RegF, the other registers can be updated only by their preceding registers. During the procedure, registers are marked with "$-$" whenever the previous values are not used any more. The field addition and multiplication are performed as $\mathrm{RegA} \leftarrow \mathrm{RegB} \times \mathrm{RegC}$ and $\mathrm{RegA} \leftarrow \mathrm{RegA} + \mathrm{RegC}$, respectively. The register file management for other parts can be similarly described.

The use of this register file increases the number of cycles due to the control overhead. However, considering that a field multiplication takes a large number of cycles, the number of overhead cycles is relatively small. One thing that we need to consider is the peak power consumption. If all the registers are updated at the same time, the large peak power consumption can be a problem. In the proposed architecture, at most four registers are updated at a time. This number can be reduced up to two by introducing more overhead cycles, but it cannot be less than two, since the field multiplication in the MALU updates two registers.

6 MICROCONTROLLER

In general, modular additions and multiplications are needed in protocols. Because they are not part of the critical calculations and thus do not contribute to the latency, we decided to perform these modular operations on the 8-bit microcontroller in a byte-serial fashion. In order to reduce the computation amount of modular reductions, we use a form of redundant representation by using five extra guard bits.

6.1 General Modular Arithmetic Operation

All the scalar values are 163 bits long, so a scalar needs 21 bytes (168 bits) in an 8-bit controller. Therefore, we can utilize the extra five bits as bits for computation efficiency without extra overhead. In the redundant modular operation, a scalar is not reduced to the fully minimized form of 163 bits, but it is allowed to have a length of 168 bits. Those extra guard bits make the computation efficient.

6.1.1 Modular Addition with 8-Bit ALU

We start the modular operations by assuming that all the scalars have a length of 168 bits. The modular addition is described in Algorithm 4.

```
Algorithm 4. Modular addition with 8-bit ALU
Require: $A = \sum_{k=0}^{167} a_k 2^k$, $B = \sum_{k=0}^{167} b_k 2^k$
Ensure: $C = A + B \bmod n$
 1: $C \leftarrow \mathbf{Add}(A, B)$;
 2: $C_0 \leftarrow \sum_{k=0}^{166} c_k 2^k$, $D \leftarrow \sum_{k=0}^{1} c_{k+167} 2^k$,
    $C_1 \leftarrow \mathbf{Multiply}(N_1, D)$;
 3: $C \leftarrow \mathbf{Add}(C_0, C_1)$;
 4: Return $C$;
```

Step 1 is for addition, and steps 2 and 3 are for reduction, where $N_1 = 2^{167} \bmod n$. To provide the validity of the reduction, we consider the following:

$$\begin{aligned} C \bmod n &= \left(\sum_{k=0}^{168} c_k 2^k \right) \bmod n \\ &= \left(\sum_{k=0}^{166} c_k 2^k + 2^{167} \cdot \sum_{k=0}^{1} c_{k+167} 2^k \right) \bmod n \\ &= \left(\sum_{k=0}^{166} c_k 2^k + N_1 \cdot \sum_{k=0}^{1} c_{k+167} 2^k \right) \bmod n. \end{aligned} \quad (8)$$

Since the size of N_1 is 163 bits, $N_1 \cdot \sum_{k=0}^{1} c_{k+167} 2^k$ is 165 bits long, and the final result will be up to 168 bits.

Therefore, the modular addition requires two $\mathbf{Add}(\cdot,\cdot)$ operations and one $\mathbf{Multiply}(\cdot,\cdot)$ operation. $\mathbf{Add}(\cdot,\cdot)$ and $\mathbf{Multiply}(\cdot,\cdot)$ are described in Algorithms 5 and 6, respectively. Though the inputs of $\mathbf{Add}(\cdot,\cdot)$ in Algorithm 4 are 21 bytes long, $\mathbf{Add}(\cdot,\cdot)$ is defined for 22-byte inputs so that it can be used for the reduction of the modular multiplication as well.

Algorithm 5. Addition of 22-byte operands: $\mathbf{C} \leftarrow \mathbf{ADD}(\mathbf{A},\mathbf{B})$
Require: $A = \sum_{k=0}^{175} a_k 2^k$, $B = \sum_{k=0}^{175} b_k 2^k$
Ensure: $C = A + B$

```
1: $Carry_{Bit} \leftarrow 0$;
2: for $i$ from 0 to 21 do
3:     $A^{(i)} \leftarrow \sum_{k=0}^{7} a_{8i+k}2^k$, $B^{(i)} \leftarrow \sum_{k=0}^{7} b_{8i+k}2^k$;
4:     $\{C^{(i)}, Carry_{Bit}\} \leftarrow A^{(i)} + B^{(i)} + Carry_{Bit}$;
5: end for
6: $C^{(22)} \leftarrow Carry_{Bit}$;
7: Return $C$;
```

In Algorithm 5, $Carry_{Bit}$ is a 1-bit variable that stores the carry of the addition of two 1-byte values. $A^{(i)}$ is the ith byte of A. In Algorithm 6, $Carry_{Byte}$ is a 1-byte variable that stores the most significant byte of the multiplication of two 1-byte values. M_0 is the least significant byte, and M_1 is the most significant byte of $A^{(i)} \times D$.

Algorithm 6. Multiplication of a 21-byte value by a 1-byte value: $\mathbf{C} \leftarrow \mathbf{Multiply}(\mathbf{A},\mathbf{B})$
Require: $A = \sum_{k=0}^{167} a_k 2^k$, $D = \sum_{k=0}^{7} d_k 2^k$
Ensure: $C = A \cdot D$

```
1: $Carry_{Byte} \leftarrow 0$, $Carry_{Bit} \leftarrow 0$;
2: for $i$ from 0 to 20 do
3:     $A^{(i)} \leftarrow \sum_{k=0}^{7} a_{8i+k}2^k$;
4:     $\{M_0, M_1\} \leftarrow A^{(i)} \cdot D$;
5:     $\{C^{(i)}, Carry_{Bit}\} \leftarrow M_0 + Carry_{Byte} + Carry_{Bit}$;
6:     $Carry_{Byte} \leftarrow M_1$;
7: end for
8: $C^{(21)} \leftarrow Carry_{Byte} + Carry_{Bit}$;
9: Return $C$;
```

6.1.2 Modular Multiplication with 8-Bit ALU

The algorithm of byte-serial modular multiplication is described in Algorithm 7.

Algorithm 7. Modular multiplication with 8-bit ALU
Require: $A = \sum_{k=0}^{k=167} a_k 2^k$, $B = \sum_{k=0}^{k=167} b_k 2^k$
Ensure: $C = A \cdot B \bmod n$

```
1: for $i$ from 20 down to 0 do
2:     $C_0 \leftarrow 2^8 \cdot C_0$;
3:     $D \leftarrow \sum_{k=0}^{7} b_{8i+k}2^k$, $C_1 \leftarrow \mathbf{Multiply}(A, D)$;
4:     $C \leftarrow \mathbf{Add}(C_0, C_1)$;
5:     $C_0 \leftarrow \sum_{k=0}^{169} c_k 2^k$, $D \leftarrow \sum_{k=0}^{6} c_{k+170}2^k$,
       $C_1 \leftarrow \mathbf{Multiply}(N_2, D)$;
6:     $C \leftarrow \mathbf{Add}(C_0, C_1)$;
7:     $C_0 \leftarrow \sum_{k=0}^{166} c_k 2^k$, $D \leftarrow \sum_{k=0}^{3} c_{k+167}2^k$,
       $C_1 \leftarrow \mathbf{Multiply}(N_1, D)$;
8:     $C \leftarrow \mathbf{Add}(C_0, C_1)$;
9: end for
10: Return $C$;
```

Steps 2-4 are for a 1-byte shift, multiplication by 1 byte, and accumulation. The remainder, steps 5-8, are for the reduction, whose validity is described by (9) and (10), where $N_1 = 2^{167} \bmod n$, and $N_2 = 2^{170} \bmod n$. The size of C before the reduction, i.e., in step 4, is 177 bits since the both of C_0 and C_1 are 176 bits long and the addition of the two will produce up to 177 bits. The following equation describes steps 5 and 6, and (10) describes the steps 7 and 8 in Algorithm 7:

$$\begin{aligned} C \bmod n &= \left(\sum_{k=0}^{176} c_k 2^k\right) \bmod n \\ &= \left(\sum_{k=0}^{169} c_k 2^k + 2^{170} \cdot \sum_{k=0}^{6} c_{k+170}2^k\right) \bmod n \\ &= \left(\sum_{k=0}^{169} c_k 2^k + N_2 \cdot \sum_{k=0}^{6} c_{k+170}2^k\right) \bmod n = C'. \end{aligned} \tag{9}$$

In (9), since the size of N_2 is 163 bits and the size of $N_2 \cdot \sum_{k=0}^{6} c_{k+170}2^k$ is 170 bits, the size of C' is at most 171 bits. C' can be reduced again according to

$$\begin{aligned} C' \bmod n &= \left(\sum_{k=0}^{170} c'_k 2^k\right) \bmod n \\ &= \left(\sum_{k=0}^{166} c'_k 2^k + 2^{167} \cdot \sum_{k=0}^{3} c'_{k+167}2^k\right) \bmod n \\ &= \left(\sum_{k=0}^{166} c'_k 2^k + N_1 \cdot \sum_{k=0}^{3} c'_{k+167}2^k\right) \bmod n. \end{aligned} \tag{10}$$

Since the size of N_1 is 163 bits and the size of $N_1 \cdot \sum_{k=0}^{3} c'_{k+167}2^k$ is 167 bits, the size of the final result is at most 168 bits. Therefore, the reduced C (i.e., C after finishing step 8 in Algorithm 7) can be used for the next iteration.

6.1.3 Comparison with Barrett's Modular Reduction in Modular Multiplication

Barrett's algorithm [8] is one of the most efficient modular reduction algorithms for a general number n. Let the size of n be t bits and $M = A \times B$, where we calculate $A \times B \bmod n$. Barrett's reduction can be expressed by

$$\begin{aligned} \mu &= \left\lfloor \frac{2^{2t}}{n} \right\rfloor, \\ q' &= \left\lfloor \left\lfloor \frac{M}{2^t} \right\rfloor \cdot \frac{\mu}{2^t} \right\rfloor, \\ R &= M - q' \cdot n. \end{aligned} \tag{11}$$

μ can be precalculated, and hence, it does not contribute to the computation amount. Since R is congruent to $M \bmod n$ and $R < 3n$, the final result requires at most two t-bit subtractions after calculating R. Therefore, the required computation in the worst case is two t-bit multiplications, one $2t$-bit subtraction, and two t-bit subtractions. For the calculation, it requires $5t$-bit temporary memory since M requires $2t$-bit memory, q' requires t-bit memory, and the result of $q' \cdot n$ needs to be stored in separate $2t$-bit memory. Therefore, the required memory is 105 bytes if t is 163 and the unit of memory is 8 bits (each t bits will need 21 bytes).

TABLE 6
RAM Usage for Modular Multiplication

Block #	Initial	2)	3)	4)	5)	6)	7)	8)
0	C(168)	C_0(176)	C_0(176)	C(177)	C_0(170)	C(171)	C_0(167)	C(168)
1			C_1(176)		C_1(170)		C_1(167)	
2	A(168)	A(168)	A(168)	A(168)	A(168)	A(168)	A(168)	A(168)
3	B(168)	B(168)	B(168)	B(168)	B(168)	B(168)	B(168)	B(168)

In the proposed modular multiplication in Algorithm 7, steps 3-4 are for the multiplications, and steps 5-8 are for the reduction. The complexity of the reduction is exactly two n-bit multiplication, and the required temporary memory is 44 bytes. Therefore, the proposed modular reduction is more efficient in both computation and memory. The comparison between two algorithms is summarized in Table 7.

6.2 Eight-Bit ALU

ALU has an 8-bit adder and an 8-bit multiplier. The detailed algorithms for general modular operations are implemented in hardware and performed using these two blocks.

6.3 Addressing and Memory Management

The addressing is composed of 13 bits where the first two 4 bits are used for a device ID and a block address, and the last 5 bits for a byte address (Table 8). The device ID indicates the ROM for the program, the ROM for data, the RAM, the RNG, the front-end module, and the ECP. The block address and the byte address are used only for ROM and RAM. Even the nonmemory devices are memory mapped so that the microcontroller gives an input or gets an output.

Since all the scalar values are 21 bytes long, each block is composed of 21 bytes. All the basic data managements are based on blocks. This makes the program and the control very simple. For example, the program needs to specify only the block address for the general modular operations.

The process of intermediate value storage for the modular addition is described in Table 9, where we assume that the operands of the modular addition are stored in Blocks 2 and 3. The numbering of 1), 2), and 3) indicate the steps of Algorithm 4, and the numbers inside of the parentheses are the size of the variables in bits. For the modular addition, two blocks of RAM must be reserved to hold intermediate values and the final result.

The process of intermediate value storage for the modular multiplication is described in Table 6, where we assume that the operands are stored in Blocks 2 and 3, and the steps 1), 2), etc., represent the steps of Algorithm 7.

For the reduction, n is not used but N_1 and N_2 are used, which should be precalculated and stored in ROM. The modular multiplication also needs to reserve the first two memory blocks in RAM, so they should not store meaningful values before starting modular operations. Note that the size of Blocks 0 and 1 is 22 bytes, which is 1 byte larger than the other blocks. Though the results of the modular operations are not fully reduced in tags, such reduction can be taken care of in the reader or the server.

6.4 ROM for Data

There are some data that should be stored in tags for their processing. The data may include the private key of a tag, the public key of the reader, and system parameters. If this kind of data is hardwired, the architecture can be simplified since tags do not need to access the memory and the hardwired data is simpler than any other memory structure. However, the hardwired data cannot be changed if the architecture is once produced as an ASIC. Therefore, some of the data should be stored in nonvolatile memory. Table 10 shows the ROM usage of the Schnorr protocol as an example, where N_1 and N_2 are used for general modular operations, k is the private key of a tag, and $x(P)$ is the X-coordinate value of the base point P.

6.5 Instructions of the Microcontroller

The instructions must be carefully designed since they are directly related to the system performance and the program size. If the instructions are designed at a lower level, the

TABLE 7
RAM Usage for Modular Multiplication

Reduction Algorithm	Computation Complexity	Memory Requirement
Barrett's Algorithm	two t-bit multiplications one $2t$-bit subtraction two t-bit subtractions	105 bytes
Proposed Algorithm	two t-bit multiplications	44 bytes

TABLE 8
Physical Memory Address Map

0	1	2	3	4	5	6	7	8	9	10	11	12
Device ID(4)				Block Address(4)				Byte Address(5)				

TABLE 9
RAM Usage for Modular Addition

Block #	Initial	1)	2)	3)
0		C(169)	C_0(167)	C(168)
1			C_1(167)	
2	A(168)	A(168)	A(168)	A(168)
3	B(168)	B(168)	B(168)	B(168)

TABLE 10
ROM for Schnorr Protocol

Block Address in Data ROM	0	1	2	3
Data	N_1	N_2	k	$x(P)$

TABLE 11
Instructions of the Processor

Instruction	Description	Number of cycles
Block_Mov (A, B)	Move one block of memory from the block address A to the block address B.	121
Block_Add (A, B)	Add the block A and the block B, and store in RAM[0].	574
Block_Mul (A, B)	Multiply the block A by the block B, and store in RAM[0].	25,486
Activate_ECP (A)	Start the ECP with k of RAM[4] and P of the block A: $k \cdot P$. (When the ECP finishes, the result is stored in RAM[2] and RAM[3])	≥ 11
Wait_for_ECP	The processor waits until the ECP completes its computation.	≥ 10
End_of_code	The end of the program.	6

memory for a program will grow and the performance will be degraded since most of the control must be done in software. Instructions are at a high level to reduce the control overhead. Programmability is still required as we want to use the device for multiple protocols.

In the proposed architectures, the program is stored in ROM starting from the address 0. Each instruction is composed of 1, 2, or 3 bytes, where the first byte is for a command, and the remaining bytes are for block addressing. Since most of the data is constructed in blocks and the designed instructions manipulate data in blocks, the program is significantly simplified, and the programmer does not have to care about the details of the computation. Since the block-level instructions are implemented in hardware, the processor runs fast and efficiently. Table 11 summarizes some important commands where each of the instructions and operands is one byte and RAM[i] is the ith block of RAM.

Table 12 shows an example program for the Schnorr protocol of Fig. 1.

7 Synthesis Results and Performance Analysis

In order to find the best trade-offs, we have designed three different architectures of the ECP, as shown in Table 14. Type 1 is the minimal version described so far. Type 2 uses an extra register to hold the X-coordinate value of the base EC point (i.e., P at the EC scalar multiplication of $k \cdot P$), say, $x(P)$. Therefore, this extra register makes the ECP load $x(P)$ only once and use for the whole calculation of an EC scalar multiplication. Otherwise, the ECP has to load $x(P)$ at every iteration in the Montgomery algorithm, which means that the ECP has to load 163 times for a 163-bit key. Type 3 has an extra register and a randomly accessible register file. The use of the extra register and the randomly accessible register file increase the gate area and reduce the number of cycles. Therefore, the ECP Type 1 has the least gate area and the most number of cycles, and Type 3 has the most gate area and the least number of cycles in the same digit size.

The proposed architectures are synthesized using a low leakage power library of UMC's 0.13 μm (fsc0l_d_sc_tc.db). The synthesized architectures include the microcontroller, the bus manager, and the ECP corresponding to everything within the dashed lines in Fig. 2. Some samples of the synthesis results and the performances are shown in Table 13. The number of cycles is to finish the Schnorr protocol, which includes one EC scalar multiplication, some general modular operations, the random number generation, and the data transmission/reception.

The clock frequency is chosen to finish the Schnorr protocol within 250 ms and to be a factor of 13.56 MHz, i.e., the carrier frequency of a reader in our system. Therefore, tags can use a simple division logic of the carrier frequency for their internal clock frequency so that a separate pulse generator is not needed. Though the Schnorr protocol requires only one EC scalar multiplication, some other protocols such as the Okamoto protocol [5] requires two EC scalar multiplications. We expect one EC scalar multiplication to finish in 250 ms so that even the protocols that have two EC scalar multiplications can finish in 500 ms. Five hundred milliseconds is a very reasonable response time though it is too much delay for sequential accesses of multiple tags. However, it is possible to solve the throughput problem by applying a multiple-access protocol that can handle multiple tags in parallel. This is possible because most of the time taken in the processor is caused by the calculation inside of tags, and therefore, if we can make multiple tags start the authentication in parallel and the radio communication of each tag exclusive, the overall throughput can be effectively increased.

The gate area is dominated by the register file. In order to minimize the gate area, we minimize the flip-flops. The UMC standard cell library of 0.13 μm offers a very compact D flip-flop combined with a multiplexer, which can be implemented in 6.25 gate area. In the case of Type 1 and the digit size of 1, the register file occupies 7.53 Kgates. This is around 7.7 gates per bit including the multiplexers.

TABLE 12
Program for the Schnorr Protocol

Program Code	Comment
Block_Mov (RNG, RAM[4])	Generate a block of random data and store in RAM[4].
Activate_ECP (ROM[3])	Start the ECP with X-coordinate value of ROM[3].
Wait_for_ECP	Wait until the ECP finishes.
Block_Mov (RAM[2], Tran.)	Transmit RAM[2].
Block_Mov (RAM[3], Tran.)	Transmit RAM[3].
Block_Mov (Recv., RAM[2])	Receive a block and store in RAM[2].
Block_Mul (ROM[2], RAM[2])	Do modular multiplication of ROM[2] and RAM[2], and store in RAM[0].
Block_Add (ROM[0], RAM[4])	Do Modular addition of ROM[0] and RAM[4], and store in RAM[0].
Block_Mov (RAM[0], Tran.)	Transmit RAM[0].
End_of_code	The end of the program.

TABLE 13
Synthesis Result and Performance

Type	Digit Size	ECP Gate Area	Overall[1] Gate Area	Cycles[2]	Frequency (KHz)	Time[2] ($msec$)	Dynamic Power (μW)	Leakage Power (μW)	Total Power (μW)
1	1	10,106	12,506	302,457	1,130	244.43	36.5780	0.0509	36.6289
	2	11,383	14,064	171,480	590	246.33	21.4927	0.0553	21.5480
	3	12,236	14,729	127,821	411	247.18	15.6854	0.0609	15.7463
	4	12,863	15,356	105,183	323	244.47	12.0117	0.0641	12.0758
	5	13,497	15,989	92,247	266	248.20	11.3389	0.0674	11.4063
2	1	11,133	13,624	298,111	1,130	240.58	38.6624	0.0559	38.7183
	2	12,696	15,191	167,136	565	249.35	22.5107	0.0622	22.5729
	3	13,319	15,808	123,475	399	243.77	16.0403	0.0654	16.1057
	4	13,934	16,433	100,837	301	247.51	12.7105	0.0686	12.7791
	5	14,570	17,251	87,901	251	245.50	11.0944	0.0721	11.1665
3	1	14,307	16,799	293,587	1,130	236.58	43.3769	0.0673	43.4442
	2	15,967	18,451	162,608	565	241.33	24.0127	0.0751	24.0878
	3	16,568	19,074	118,951	377	246.10	16.7974	0.0783	16.8757
	4	17,200	19,693	96,311	283	247.99	13.0870	0.0814	13.1684
	5	17,837	20,316	83,375	230	248.54	11.1542	0.0845	11.2387

(1) *The synthesis results are for RFID processor which includes the micro controller, the bus manager and ECP.*
(2) *The number of cycles and the time are to complete the Schnorr protocol of Fig. 1 and Table 12.*

The trade-offs of the gate area and the number of cycles depending on the digit size are shown in Fig. 6, where each of the three line graphs represents each of different types. On each graph, the leftmost-side dot is for the digit size of one, and the digit size grows one by one through the rightmost-side dot until the digit size of 10. The most compact architecture is Type 1 with the digit size of one. If the digit size is increased to more than five, Type 2 shows better performance than Type 1 in terms of the cycle number and area product. Again, if the digit size of Type 2 is increased enough, Type 3 will show better performance. This result is due to a constant factor in the number of cycles, which is independent of the digit size in Type 1. Note that Type 1 has to perform some register file management operations due to its special architecture of the shift register file and also has to load the X-coordinate value of the base EC point at each iteration in the Montgomery algorithm. Type 2 also has some constant factor of the cycles due to the shift register file management, but it is smaller than Type 1. Since Type 3 does not have such constant factors, the number of cycles can be more effectively decreased by increasing the digit size.

One of the most important factors in RFID tags is power consumption, especially if tags are passive. In order to get the average power estimation in the gate level, we used Design Vision and ModelSim SE. We generated Value Change Dump (VCD) files in ModelSim using a test bench data. Then, VCD files are translated to Switching Activity Interchange Format (SAIF) files, which are used in Design Vision to get average power consumptions. Since we used a low leakage power library of UMC, the leakage power is negligible, as shown in Table 13.

TABLE 14
ECP Types

	An extra buffer register	The register file type
Type 1	No	Circular Shift Register File
Type 2	Yes	Circular Shift Register File
Type 3	Yes	Randomly Accessible Register File

Fig. 7 shows the synthesis results for the power consumption. According to the synthesis results, while increasing the digit size is an effective way to reduce the dynamic power consumption, it increases the leakage power. Since the leakage power is negligible, we get lower total power consumption as we increase the digit size. However, considering the gate area, we should limit the digit size. Although Types 2 and 3 show lower power consumptions than Type 1 when the digit size is increased, since the gate area is larger, increasing the digit size of Type 1 would be a better choice rather than choosing Type 2 or Type 3.

According to Zhou et al. [17], the maximal allowed power for tags is less than 100 μW, and in [19], the author presents 30 μW for a security processor. ISO 18000-3 (13.56 MHz) [18] requires the power consumption of less than 15 μW at 1.5 V to guarantee 1-m operating range. In our synthesis results, if we increase the digit size, the power consumption becomes close to 10 μW. This power consumption would be low enough for even a passive tag.

7.1 Parallelism in ECP and General Modular Operations

The operations in the ECP are most critical in the number of cycles and afterward come general modular operations. For efficiency, the system is designed to run the EC operations and the general modular operations in parallel. In the Schnorr protocol, this parallelism is not useful since the general modular operations must be performed after the EC operations are finished. However, depending on the protocol, this parallelism can effectively reduce the number of cycles.

7.2 Required Memory Space of ROM and RAM

The required memory amount is dependent on the cryptographic protocol. In the case of the Schnorr protocol, the required memories are summarized in Table 15. Note that one block of memory is 21 bytes, and 2 extra bytes are required in RAM, which are used for modular operations. During the modular operations, blocks 0 and 1 of RAM require an extra byte each (refer to Table 6).

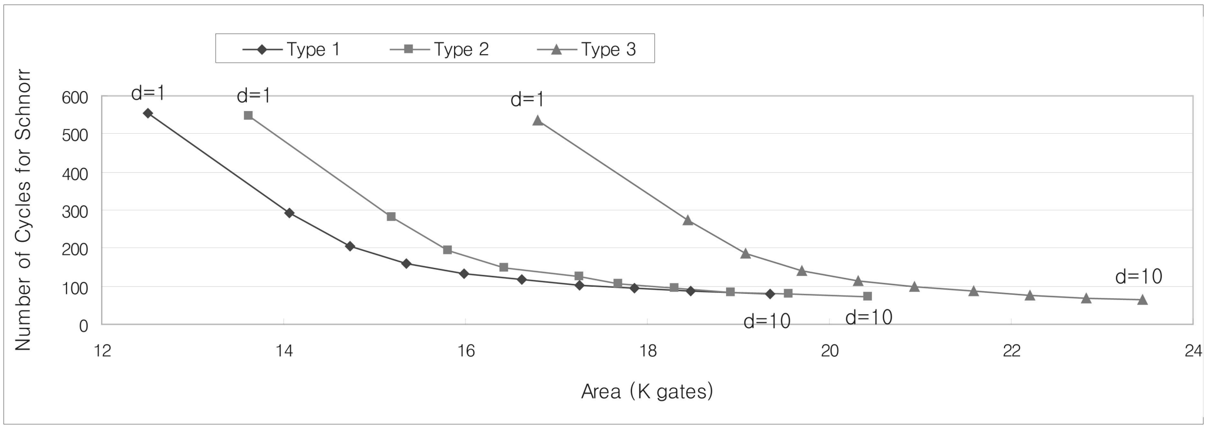

Fig. 6. The trade-offs of gate area and number of cycles for three different types.

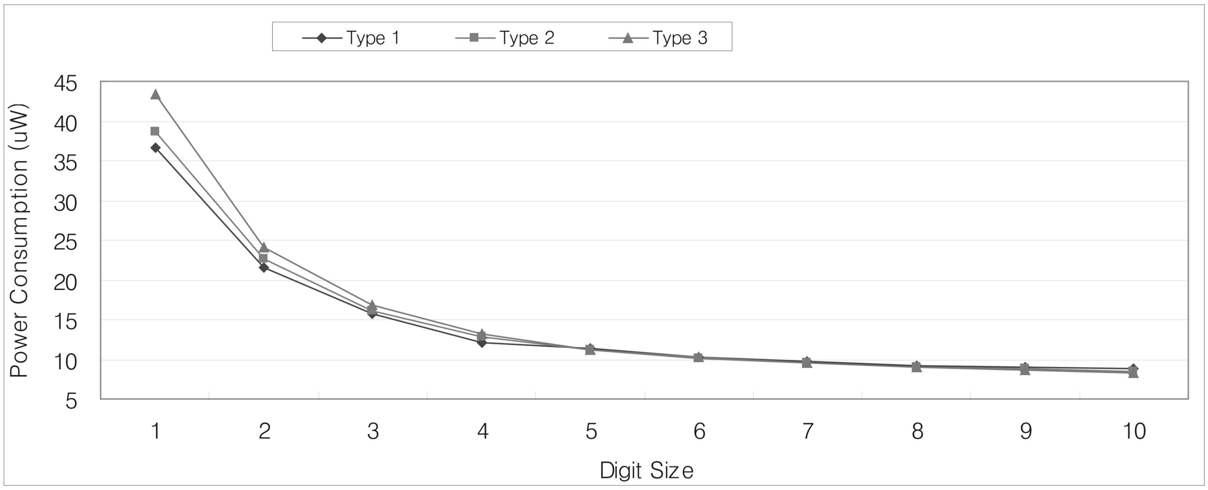

Fig. 7. The trade-offs of digit size and power consumption for three different types.

7.3 Memory Access

The synthesis results do not include ROM and RAM, and hence, the power consumption should be considered separately. The number of memory accesses for the Schnorr protocol is summarized in Table 16 to give an idea for the estimation of power consumption in memory. Note that the number of memory accesses is independent of the digit size of the ECP. Type 1 reads ROM more than Types 2 and 3 since it does not have a register for the EC base point (it has to read the base point every time it is needed). There is no difference between Type 2 and Type 3 except in the register file type in the ECP.

7.4 Comparison with Related Work

Table 17 shows the comparison with related work. Even though the key size of [20] is much smaller than ours, it has a larger gate area and a larger number of cycles (when the digit size is more than one). The result in [21] shows 15,094 gates, which is still larger than the digit size of three of our proposal and requires more than three times cycles. The results in [22] also show a larger gate area and a larger number of cycles. Among related work (except for our designs), [19] has the least power consumption. Though the power consumption is smaller than our design of the digit size of one, the delay is much larger. Moreover, in our design, the power consumption can be effectively reduced by increasing the digit size. One notable thing is that only our designs can perform the general modular operations. Based on this comparison, our designs require small gate areas and small numbers of cycles and consume small power and energy.

TABLE 15
Required Memories for the Schnorr Protocol

Memory	Size
ROM for program	25 bytes
ROM for data	4 blocks = 84 bytes
RAM	5 blocks + 2 bytes = 107 bytes

TABLE 16
Memory Access for the Schnorr Protocol

	Memory	Read	Write
Type 1	ROM	4,330 times	–
	RAM	3,911 times	3,448 times
Type 2	ROM	928 times	–
	RAM	3,911 times	3,448 times
Type 3	ROM	928 times	–
	RAM	3,911 times	3,448 times

* One byte is accessed on each time.

TABLE 17
Comparison with Related Work

Ref.	PKC	Digit Size	Area (gates)	Cycles	CMOS (μm)	Freq. (KHz)	Perf. ($msec$)	Power (μW)	Energy per op. (μJ)	Operations
This Work	ECC GF(2^{163}) Type 1	1	12,506	275,816	0.13	1,130	244.08	36.63	8.94	Point Mult.,
		2	14,064	144,842		590	245.49	21.55	5.29	General
		3	14,729	101,183		411	246.19	15.75	3.88	Modular
		4	15,356	78,544		323	243.17	12.08	2.94	Operation
[20]	ECC GF(p), $p = (2^{101}+1)/3$	N/A	18,720	205,225	0.13	500	410	394	161.5	ECDSA
[21]	ECC GF(2^{163})	N/A	15,094	376,864	0.35	13,560	31.8	N/A	N/A	Point Mult.
[22]	ECC GF(p_{160})	N/A	19,000	362,000	0.35	N/A	N/A	N/A	N/A	ECDSA
	ECC GF(p_{192})		23,600	502,000		83,333	6	141,000	846.0	
[19]$^{(1)}$	ECC GF(2^{191})	N/A	23,818	N/A	0.35	60	7,100	30	N/A	Point Mult.
					0.18	175	2,500	30		
					0.09	545	800	30		

* All the results are based on one EC scalar multiplication.
$^{(1)}$ *This work is a unified solution with* GF(p_{192}).

8 Conclusion

We propose a compact architecture of an EC-based security processor for RFID. For a compact ECP, we introduced several techniques such as the common Z-coordinate system, the register reuse, and a circular shift register file. We also proposed an algorithm for general modular operations with a redundant representation using a few extra guard bits. By utilizing the remainder bits of long scalars, we designed an efficient modular operation algorithm without overhead. The designed modular operations not only are computationally efficient but also reduce memory requirement compared to conventional modular algorithms. Moreover, the ECP and the modular operation, which are the two most critical operations, can be performed in parallel so that the number of cycles can be effectively reduced, depending on the cryptographic protocol.

We synthesized the proposed architectures with 0.13-μm CMOS technology for three different types and for different digit sizes of the ECP to show trade-offs for the number of cycle number, gate area, and power consumption. Compared to other reported results, our architecture not only minimizes the gate area and power consumption but also shows better performance. According to the synthesis results, the power consumption can be reduced to near to 10 μW, which would be low enough even for a passive RFID tag.

Acknowledgments

This work is supported by US National Science Foundation CCF-0541472, SRC, FWO, IUAP BCRYPT, and funds from the Katholieke Universiteit Leuven.

References

[1] M. Burmester, B. Medeiros, and R. Motta, "Robust Anonymous RFID Authentication with Constant Key Lookup," *Proc. ACM Symp. Information, Computer and Comm. Security (ASIACCS),* 2008.
[2] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.
[3] P. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Math. Computation,* vol. 48, pp. 243-264, 1987.
[4] C.-P. Schnorr, "Efficient Identification and Signatures for Smart Cards," *Proc. Ninth Ann. Int'l Cryptology Conf. (CRYPTO '89),* pp. 239-252, 1989.
[5] T. Okamoto, "Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes," *Proc. 12th Ann. Int'l Cryptology Conf. (CRYPTO '92),* pp. 31-53, 1992.
[6] I. Blake, G. Seroussi, and N.P. Smart, "Elliptic Curves in Cryptography," *London Math. Soc. Lecture Note Series.* Cambridge Univ. Press, 1999.
[7] P. Montgomery, "Multiplication without Trial Division," *Math. Computation,* vol. 44, pp. 519-521, 1985.
[8] P.D. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," *Proc. Advances in Cryptology (CRYPTO '86),* pp. 311-323, 1987.
[9] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation," *Proc. First Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '99),* pp. 316-327, 1999.
[10] N. Meloni, "Fast and Secure Elliptic Curve Scalar Multiplication over Prime Fields Using Special Addition Chains," *Cryptology ePrint Archive: Listing for 2006 (2006/216),* 2006.
[11] C. Paar, *Light-Weight Cryptography for Ubiquitous Computing,* Invited Talk at the Univ. of California, Los Angeles (UCLA), Inst. for Pure and Applied Math., Dec. 2006.
[12] K. Sakiyama, L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede, "Small-Footprint ALU for Public-Key Processors for Pervasive Security," *Proc. Workshop RFID Security (RFIDSec '06),* p. 12, 2006.
[13] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks," *Proc. Third European Workshop Security and Privacy in Ad Hoc and Sensor Networks (ESAS '06),* pp. 6-17, 2006.
[14] Y.K. Lee and I. Verbauwhede, "A Compact Architecture for Montgomery Elliptic Curve Scalar Multiplication Processor," *Proc. Eighth Int'l Workshop Information Security Applications (WISA '07),* pp. 115-127, 2007.
[15] E. Öztürk, B. Sunar, and E. Savas, "Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic," *Proc. Sixth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '04),* pp. 92-106, 2004.
[16] A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor," *IEEE Trans. Computers,* vol. 52, no. 4, pp. 449-460, Apr. 2003.
[17] F. Zhou, C. Chen, D. Jin, C. Huang, and H. Ming, *Evaluating and Optimizing Power Consumption of Anti-Collision Protocols for Application in RFID Systems,* AUTO-ID Labs, http://www.autoidlabs.org/uploads/media/AUTOIDLABS-WP-SWNET-014_01.pdf, white paper, 2008.
[18] ISO/IEC 18000-3:2004, *Information Technology—Radio Frequency Identification (RFID) for Item Management—Part 3: Parameters for Air Interface Communications at 13.56 MHz.*
[19] J. Wolkerstorfer, "Is Elliptic-Curve Cryptography Suitable to Secure RFID Tags?" *Proc. Workshop RFID and Light-Weight Cryptography,* Aug. 2005.

[20] G. Gaubatz, J.-P. Kaps, E. Öztürk, and B. Sunar, "State of the Art in Ultra-Low Power Public Key Cryptography for Wireless Sensor Networks," *Proc. Second IEEE Int'l Workshop Pervasive Computing and Comm. Security (PerSec '05),* pp. 146-150, 2005.

[21] S. Kumar and C. Paar, "Are Standards Compliant Elliptic Curve Cryptosystems Feasible on RFID?" *Workshop Record of the ECRYPT Workshop RFID Security,* p. 19, 2006.

[22] F. Fürbass and J. Wolkerstorfer, "ECC Processor with Low Die Size for RFID Applications," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS '07),* pp. 1835-1838, 2007.

Yong Ki Lee received the BS and MS degrees in computer science and engineering from Hanyang University, Korea, in 1997 and 1999 respectively. Since 2004, he has been a PhD student in the Electrical Engineering Department, University of California, Los Angeles (UCLA), studying in the field of communication and telecommunications. During his PhD program, he also received an MS degree from the same department in 2006. From 2000 to 2003, he was in the Korean Air Force as the chief of the Computer Department and the staff officer of information and communication in the 9785th base, where he worked on managing and developing computer- and communication-related tasks and on managing information security. His research interests are in the area of security protocol design and security hardware implementation, especially for radio frequency identification (RFID) and sensor networks. He is a student member of the IEEE.

Kazuo Sakiyama received the BEng and MEng degrees in electrical engineering from Osaka University, Japan, in 1994 and 1996, respectively, the MSc degree in electrical engineering from the University of California, Los Angeles (UCLA), and the PhD degree from the Katholieke Universiteit Leuven (KUL), Belgium, in 2007. From 1996 to 2004, he was with the Semiconductor and IC Division, Hitachi, Ltd., (now Renesas Technology Corp.). He is currently an associate professor in the Department of Information and Communication Engineering, University of Electro-Communications (UEC), Tokyo. His main research interest is efficient and secure embedded system architectures and design methodologies for cryptographic applications. He is a member of the IEEE. His research information can be found at http://www.oslab.ice.uec.ac.jp/~saki.

Lejla Batina received the MSc degree in mathematics from the University of Zagreb, Croatia, and the PhD degree in engineering from the Katholieke Universiteit Leuven (KU Leuven), Belgium. She is with the Computer Security and Industrial Cryptography (COSIC) Research Group, KU Leuven, as a postdoctoral researcher. Her research interests include efficient arithmetic for cryptographic algorithms, side-channel security, and protocols for lightweight cryptography. She is a member of the IEEE. More information on her research can be found at http://homes.esat.kuleuven.be/~lbatina.

Ingrid Verbauwhede received the electrical engineering degree and PhD degree from the Katholieke Universiteit Leuven (KU Leuven), Belgium, in 1991. From 1992 to 1994, she was a postdoctoral researcher and visiting lecturer at the Electrical Engineering and Computer Sciences Department, University of California, Berkeley. From 1994 to 1998, she worked for TCSI and ATMEL in Berkeley, California. In 1998, she joined the faculty of University of California, Los Angeles (UCLA). She is currently a professor at the KU Leuven and an adjunct professor at UCLA. At KU Leuven, she is a codirector of the Computer Security and Industrial Cryptography (COSIC) Laboratory. Her research interests include circuits, processor architectures and design methodologies for real-time embedded systems for security, cryptography, digital signal processing, and wireless communications. This includes the influence of new technologies and new circuit solutions on the design of next-generation systems on chip. She was the program chair of the Ninth International Workshop on Cryptographic Hardware and Embedded Systems (CHES '07), the 19th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '08), and the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '02). She was also the general chair of ISLPED 2003. She was a member of the executive committee of the 42nd and 43rd Design Automation Conference (DAC) as the design community chair. She is a senior member of the IEEE.