High Performance Architecture for Elliptic Curve Scalar Multiplication over $GF(2^m)$

Junjie Jiang, Jing Chen, Jian Wang, Duncan S. Wong, and Xiaotie Deng

Abstract

We propose a new architecture for performing Elliptic Curve Scalar Multiplication (ECSM) on elliptic curves over $GF(2^m)$. This architecture maximizes the parallelism that the projective version of the Montgomery ECSM algorithm can achieve. It completes one ECSM operation in about $2(m-1)(\lceil m/D \rceil+4)+m$ cycles, and is at least three times the speed of the best known result currently available. When implemented on a Virtex-4 FPGA, it completes one ECSM operation over $GF(2^{163})$ in $12.5\mu s$ with the maximum achievable frequency of 222MHz. Two other implementation variants for less resource consumption are also proposed. Our first variant reduces the resource consumption by almost 50% while still maintaining the utilization efficiency, which is measured by a performance to resource consumption ratio. Our second variant achieves the best utilization efficiency and in our actual implementation on an elliptic curve group over $GF(2^{163})$, it gives more than 30% reduction on resource consumption while maintaining almost the same speed of computation as that of our original design. For achieving this high performance, we also propose a modified finite field inversion algorithm which takes only m cycles to invert an element over $GF(2^m)$, rather than $2m$ cycles as the traditional Extended Euclid algorithm does, and this new design yields much better utilization of the cycle time.

Index Terms

Elliptic Curve Cryptography, Elliptic Curve Scalar Multiplication, FPGA, Finite Field Inversion

I. Introduction

Elliptic Curve Cryptography (ECC), independently introduced by Miller [1] and Koblitz [2] in 1980's, is a promising technology for new cryptographic applications. It can achieve high security levels as that of RSA [3], but with much smaller key sizes and faster computation, which result in lower power consumption as well as better memory and bandwidth savings. A 163-bit elliptic curve cryptosystem is known to provide a comparable security level to that of a 1024-bit RSA-based cryptosystem; and a 224-bit elliptic curve cryptosystem is comparable to a 2048-bit RSA-based cryptosystem [4], [5]. We refer readers to [6], [4], [7], [8] for its general background and recent applications.

J. Jiang, D. S. Wong and X. Deng are with the Department of Computer Science, City University of Hong Kong, China. J. Chen is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge MA 02139, USA. J. Wang is with the Department of Computer Science and Technology, Tsinghua University, China. The work was done when J. Chen and J. Wang were visiting City University of Hong Kong under the support of CityU research grants (Project Nos. 9440053 and 7001959).

In ECC, Elliptic Curve Scalar Multiplication (ECSM) is the core and also the most expensive operation, just as the modular exponentiation does in RSA. The study of its implementation performance has attracted great interest due to the rapid development and deployment of ECC lately [8]. On software implementation, there are several open-source libraries such as Crypto++ and MIRACL[1] available. They have been widely used and deployed, and also been used as the up-to-date benchmarks for the software implementation performance of ECSM. According to MIRACL, one ECSM operation on an elliptic curve over $GF(2^{163})$ takes $1.05ms$ on a 3GHz Pentium IV system.

On the hardware implementation of ECSM, many different architectures have been proposed [9], [10], [11], [12]. One of the most efficient FPGA implementation of ECSM on elliptic curves over $GF(2^m)$ is due to Ansari and Hasan [12]. Their implementation for $m = 163$ takes $41\mu s$ with 100MHz the maximum achievable frequency on a Virtex2-2000 FPGA to compute one ECSM operation in the projective coordinate, and that is more than 25 times faster than software implementation. In this paper, we propose a new architecture for ECSM on elliptic curves over $GF(2^m)$. Our design maximizes the execution parallelism of individual modules and also improves the utilization of cycle time for some of the modules. When implemented on a Virtex-4 FPGA, it completes one ECSM operation on an elliptic curve over $GF(2^{163})$ in just $12.5\mu s$, which is at least three times the speed of [12].

A. Our Results

We propose a new architecture for doing ECSM on elliptic curves over $GF(2^m)$. Our architecture maximizes the parallelism of the execution of independent functional modules and results in taking only $2(m-1)(\lceil m/D \rceil+4)+m$ cycles to complete one ECSM operation, where D is the digit size of the underlying digit-serial finite field multiplier. This is about one third of the number of cycles required by a design due to Ansari and Hasan [12], which has been known as one of the most efficient designs previously available. When implemented, our design also achieves at least three times the speed of the one in [12] and reaches a much higher maximum achievable frequency. Implemented on a Virtex4-LX200 FPGA, our implementation completes one ECSM operation on an elliptic curve over $GF(2^{163})$ in $12.5\mu s$ with a maximum frequency of 222MHz achieved.

In our design, we also maximize the parallelism that the Montgomery ECSM algorithm in projective coordinate can achieve. Based on our analysis, we find that the optimal implementation of Montgomery ECSM algorithm in terms of speed requires two serial finite field multiplications. In our proposed ECSM architecture, we employ this optimized design. We hope that our design will also serve as a useful reference for estimating how fast the Montgomery algorithm can achieve on FPGA in the future.

Higher resource consumption implies higher cost of production (i.e. larger chip area). For reducing the resource consumption, we investigate two other architecture variants. Our first variant reduces the number of finite field multiplication modules from three to one, the number of division modules from two to one and eliminates the inversion module. It turns out that the resource consumption is reduced by almost 50% while still maintaining the utilization efficiency (i.e. the ratio of performance to resource consumption, or, the performance-to-chip-area ratio).

[1]Crypto++: `http://www.cryptopp.com`
MIRACL: `http://www.shamus.ie`

Our second variant is similar to the first variant but having two multiplication modules instead of one. This modification is significant because it allows much higher parallelism to be achieved than the first variant does. It actually yields the best utilization efficiency, by achieving 3.16:1 performance-area ratio, while which is 2.37:1 for our original architecture. This variant also gives us more than 30% reduction on resource consumption while maintaining almost the same speed of computation as that of our original architecture when implemented on an elliptic curve group over $GF(2^{163})$.

Besides proposing a highly optimized architecture through parallelism, we also propose a new implementation method for carrying out an Extended Euclid algorithm based finite field inversion operation over $GF(2^m)$. By unrolling two iterations into one, our method takes only m cycles to invert an element rather than $2m$ cycles in the traditional implementation of the Extended Euclid algorithm. More importantly, each iteration in our method has much better utilization of the time of one clock cycle than the traditional one. This gives a significant improvement on the number of cycles required for completing one finite field inversion while without increasing the cycle time. This improved implementation contributes significantly to the high performance and high performance-area ratio of our proposed ECSM architecture.

B. Related Work

In 2000, Orlando and Paar [9] proposed an FPGA-based processor for ECSM on elliptic curves over $GF(2^m)$. Their implementation requires $(6\lceil 167/D\rceil + 24) * 166 + 20\lceil 167/D\rceil + 764$ clock cycles to complete one ECSM operation over $GF(2^{167})$, where D is the word size of the underlying finite field multiplier. The reported performance on a Virtex-E-400 FPGA is $210\mu s$. In their design, the finite field multiplication operations in each iteration of the main loop of ECSM are carried out serially.

In 2003, Eberle et al. [10] proposed an FPGA-based processor for ECSM on elliptic curves over $GF(2^m)$ where m is configurable up to the value of 255. Several standardized curves have also been preloaded for optimized implementation. They reported the speed of their implementation is $302\mu s$ for ECSM on generic elliptic curves over $GF(2^{163})$ on a Virtex-E 2000 FPGA, and $144\mu s$ for preloaded curves. However, as their design has to include additional logics for handling generic curves, its resource consumption (i.e. chip area) is noticeably high when compared with the corresponding speed it can achieve. As we will see later, in order to get a better performance-area ratio, in our design, we provide developers a slightly less freedom on choosing parameters without any reconfiguration, but in return, we improve the performance by more than 10 times. On the parameters that we still allow developers to choose without any reconfiguration, they include the choice of elliptic curves, base points and certain finite field parameters. Reconfiguration is needed only if the underlying finite field is fundamentally changed, for example, from $GF(2^{163})$ to $GF(2^{255})$. As there are usually many suitable elliptic curves for each particular finite field, developers do not need to do reconfiguration that often in practice, while the gain on performance of our design is much significant.

Another FPGA-based processor for ECSM proposed recently is due to Mentens et al. [11]. The algorithms they used are not the optimized ones, and hence it takes longer time to compute. The reported result is $3.801ms$ for

completing one ECSM operation on elliptic curves over $GF(2^{160})$.

In [12], Ansari and Hasan proposed a hardwired logic for performing ECSM on elliptic curves over $GF(2^m)$. In their design, a pseudo-pipelined finite field multiplier over $GF(2^m)$ was constructed in such a way that the multiplier can complete one finite field multiplication in exactly $\lceil m/D \rceil$ cycles rather than $\lceil m/D \rceil + c$ cycles, where D is the word size of the multiplier and c is some non-zero positive constant, which is needed for a traditional implementation of the finite field multiplier. This optimization and several others make their design efficient. To complete one ECSM operation on an elliptic curve over $GF(2^m)$ in the *projective coordinate*, their design takes approximately $6(m-1)\lceil m/D \rceil$ clock cycles, or $41\mu s$ with a maximum achievable frequency of 100MHz on a Virtex2-2000 FPGA when $m = 163$. In their design, six finite field multiplications are carried out serially. As we will see in subsequent sections, our design allows some of these finite field multiplication operations to be carried out in parallel and hence reduces the effective sequential operations to two.

Paper Organization. The rest of the paper is organized as follows. In Sec. II, we introduce the individual algorithms that constitute our ECSM system. Optimizations that we have done on some of the algorithms are also described. In particular, we propose an improved finite field inversion implementation method over $GF(2^m)$ in Sec. II-C and show that it doubles the performance of the traditional one and yields better utilization of the cycle time. The design of our ECSM architecture is described in Sec. III. This is followed by performance analysis, complexity analysis, and comparison with previous results in Sec. IV. Two variants of our architecture for reducing resource consumption are proposed and analyzed in Sec. V, and the second variant is shown to achieve the best performance-area ratio. The paper is concluded in Sec. VI.

II. Algorithms and Optimizations

The most commonly used elliptic curves are defined over prime fields $GF(p)$ and binary fields $GF(2^m)$. In this paper, we focus on the latter one, where an element in $GF(2^m)$ can be represented as $a = \Sigma_{i=0}^{m-1} a_i x^i$, $a_i \in \{0, 1\}$. An elliptic curve $\mathcal{C}$ over $GF(2^m)$ is a Weierstrass equation and is defined by $y^2 + xy = x^3 + ax^2 + b$, where a and $b \neq 0$ are elements of $GF(2^m)$ [7]. Each curve also includes an imaginary point O called point at infinity, which acts as the identity element in the corresponding elliptic curve additive group.

A point P on an elliptic curve $\mathcal{C}$ over $GF(2^m)$ can be represented in *Affine Coordinate* or *Projective Coordinate* [7]. In the affine coordinate, P is denoted by two elements (x, y) of $GF(2^m)$; in the projective coordinate, P needs three elements to represent, namely (X, Y, Z), where $X, Y, Z \in GF(2^m)$ and $Z \neq 0$. Therefore, affine coordinate representation can help reduce network bandwidth and memory space for transmission and storage, respectively. On the other hand, performing ECSM in affine coordinate involves a large number of finite field inversion operations[2], whose performance is much poorer than that of finite field multiplication operations [7], [9].

In projective coordinate, it is known [7], [13], [14] that much more efficient ECSM algorithms which do not involve any finite field inversion operation can be constructed. Therefore, almost all of the recent hardware

[2]One of the most commonly used finite field multiplication algorithms in affine coordinate is double-and-add [7].

implementations (such as those reviewed in Sec. I-B) [9], [10], [11], [12] of ECSM are in the projective coordinate. In our design, we also have the core ECSM operation carried out in the projective coordinate, but we use the affine coordinate for the input and output of the whole system so that the network bandwidth and storage space of elliptic curve points can be reduced. This implies that our design also has two conversion modules included for transforming elliptic curve points from affine coordinate to projective coordinate and vice versa.

In the following, we introduce the individual algorithms that we have chosen in our ECSM implementation. We start our presentation with the highest level ECSM algorithm first, which is followed by the algorithms for the lower level finite field element operations such as addition, multiplication, square and inversion over $GF(2^m)$.

A. Elliptic Curve Scalar Multiplication (ECSM)

ECSM is an operation which on input an integer k and a point P on an elliptic curve $\mathcal{C}$, computes another point Q such that $Q = kP$. In our ECSM architecture, we use a variant of the algorithm due to López and Dahab [13], which is an improvement of the traditional Montgomery ECSM algorithm [15]. The algorithm consists of three stages: (1) convert P from affine coordinate to projective coordinate; (2) compute $Q = kP$ in projective coordinate; and (3) convert Q from projective coordinate back to affine coordinate. The algorithm is shown in Algorithm 1.

Input: Point $P = (x, y)$ and k, where $x, y, k \in GF(2^m)$
Output: Point $Q = (x_q, y_q) = kP$, where $x_q, y_q \in GF(2^m)$
/**Affine to Projective*/
$X_1 = x$, $Z_1 = 1$, $X_2 = x^4 + b$, $Z_2 = x^2$
/**Projective Scalar Multiplication*/
for i from $l - 2$ downto 0 do
/**Point Addition*/
$A = X_1 Z_2 X_2 Z_1 + x(X_1 Z_2 + X_2 Z_1)^2$
$B = (X_1 Z_2 + X_2 Z_1)^2$
/**Point Doubling*/
if $(k_i = 1)$ then $C = X_2^4 + bZ_2^4$, $D = X_2^2 Z_2^2$
else $C = X_1^4 + bZ_1^4$, $D = X_1^2 Z_1^2$
endif
if $(k_i = 1)$ then $X_1 = A$, $Z_1 = B$, $X_2 = C$, $Z_2 = D$
else $X_2 = A$, $Z_2 = B$, $X_1 = C$, $Z_1 = D$
endif
endfor
/**Projective to Affine*/
$x_q = X_1/Z_1$,
$y_q = ((X_1/Z_1 + x)(X_2/Z_2 + x) + (x^2 + y))(X_1/Z_1 + x)/x + y$
Return (x_q, y_q)

Algorithm 1. Our ECSM Algorithm - A Variant of [13]

In the algorithm, k_i represents the i-th bit of k, for $i = 0, \cdots, l-1$ and k_{l-1} is the most significant bit of k. The symbol b represents the constant of the underlying elliptic curve, that is, $y^2 + xy = x^3 + ax^2 + b$ over $GF(2^m)$. In practice, it is possible that $l < m$, but for analyzing the worst-case performance of an ECSM architecture, in the rest of the paper, we assume that $l = m$.

In each iteration of the main loop of Algorithm 1, *Point Addition* and *Point Doubling* can be run in parallel. We modify the algorithm of [13] slightly to Algorithm 1 solely for making the parallelism of *Point Addition* and *Point Doubling* explicit. Readers may refer to Appendix (page 20) for the original one.

B. Finite Field Algorithms

In Algorithm 1, finite field element operations such as addition, multiplication, square, inversion and/or division over $GF(2^m)$ are used. Addition over $GF(2^m)$ is merely the bitwise exclusive-or operation, and therefore is very efficient.

The finite field multiplication algorithm used in our design is the LSD-first digit-serial multiplication algorithm proposed by Song and Parhi [16]. Let $f(x) = x^m + \Sigma_{i=0}^{m-1} f_i x^i$, $f_i \in \{0, 1\}$, be an irreducible polynomial. The finite field multiplication operation over $GF(2^m)$ is defined by $c(x) = a(x)b(x) \bmod f(x)$, or in short, $c = ab \bmod f$. The algorithm is shown in Algorithm 2.

Input: $a, b \in GF(2^m)$
Output: $c \in GF(2^m)$, $c = ab$ over $GF(2^m)$
Set: $A^{(0)} = a, C^{(0)} = 0, d = \lceil m/D \rceil$
for i from 1 to d do
$A^{(i)} = A^{(i-1)} x^D \bmod f(x)$, (1)
$C^{(i)} = A^{(i-1)} B_{i-1} + C^{(i-1)}$, (2)
where
$A^{(i)} = \Sigma_{j=0}^{m-1} A_j^{(i)} x^j$,
$C^{(i)} = \Sigma_{j=0}^{m+D-2} C_j^{(i)} x^j$, and
$$B_i = \begin{cases} \Sigma_{j=0}^{D-1} b_{Di+j} x^j, & 0 \le i \le d-2 \\ \Sigma_{j=0}^{m-1-D(d-1)} b_{Di+j} x^j, & i = d-1 \end{cases}$$
endfor
return $C^{(d)} \bmod f(x)$

Algorithm 2. LSD-first Digit-Serial Multiplication over $GF(2^m)$

The algorithm divides the two operands a and b into digit blocks, with word/block size D. Operations in each block are performed in parallel, while the blocks are processed serially. Therefore, it takes $\lceil m/D \rceil$ cycles to compute one multiplication over $GF(2^m)$.

The multiplication algorithm can be used for performing finite field square and it takes $\lceil m/D \rceil$ cycles to compute. For efficiency, cryptosystems usually choose to use irreducible polynomials which have low Hamming weight, for example, 3, 5 or 7. In this case, the square operation can greatly be simplified, for example, by using the algorithm proposed by Wu [17] with low Hamming weight irreducible polynomials, we can complete one finite field square

operation over $GF(2^m)$ in just one cycle. We refer readers to [17] for details. In our design, we use this algorithm to do finite field square.

C. Finite Field Inversion Algorithm

Inverting an element over $GF(2^m)$ is much more expensive than multiplying two elements together, and this is also the reason why almost all recent ECSM hardware architectures have chosen to implement it in projective coordinate. This can help avoid hundreds of finite field inversion operations by introducing only a few additional finite field multiplication operations, and only a few inversions or divisions are needed at the end of the computation, for converting the result back to affine coordinate. One of the commonly used inversion algorithms is the Extended Euclid Algorithm. In Algorithm 3, it shows the implementation of the algorithm in [18].

Input: $a \in GF(2^m)$, irreducible f
Output: $a^{-1} \bmod f$
$S = f$, $R = a$, $U = 1$, $V = 0$, $\delta = 0$, $q = 0$, $t = 0$
for i from 0 to $2m - 1$ do
  if ($r_m = 0$) then
    $R = xR,\ \ U = (xU) \bmod f,\ \ \delta = \delta + 1$
  else /*$r_m = 1$*/
    $q = s_m,\ \ S = S - qR,\ \ V = V - qU,\ \ S = xS$
    if ($\delta = 0$) then
      $t = R,\ \ R = S,\ \ S = t,\ \ t = U,\ \ U = V,\ \ V = t,\ \ U = (xU) \bmod f,\ \ \delta = \delta + 1$
    else
      $U = (U/x) \bmod f,\ \ \delta = \delta - 1$
    endif
  endif
endfor
return U

Algorithm 3. Extended Euclid Algorithm over $GF(2^m)$

We notice that the maximum path delay in Algorithm 3 is very short, that is, consisting of only one LUT (lookup table) and one FF (flip-flop). As a result, when implemented, the module corresponding to Algorithm 3 is idle for most of the time in a cycle as some other simultaneous operations are still running in the same clock cycle. This implies that the corresponding clock cycle for performing one iteration of the main loop of Algorithm 3 is not fully utilized. In consideration of this, we propose to unroll the main loop so that every two iterations are now combined into one. As a result, the main loop of our modified algorithm takes only m iterations rather than $2m$ iterations. More importantly, the maximum path delay of each iteration in our revised algorithm is now corresponding to five LUTs and one FF. They are comparable to the time elapsed of one clock cycle. As a result, the clock cycle for performing one iteration in the main loop of our algorithm can be fully utilized. Algorithm 4 summarizes the modified algorithm.

Input: $a \in GF(2^m)$, irreducible f
Output: $a^{-1} \bmod f$
Set: $S = f$, $R = a$, $U = 1$, $V = 0$, $\delta = 0$, $t = 0$, $q = 0$, $e = 0$
for i from 0 to $m-1$ do
  if ($r_m r_{m-1} = 00$) then
    $R = x^2 R,\ U = (x^2 U) \bmod f,\ \delta = \delta + 2$
  else if ($r_m r_{m-1} = 01$) then
    $q = s_m,\ R = xR,\ S = x(S - qR),\ V = V - q(xU \bmod f)$
  else /*$r_m = 1$*/
    if ($\delta \geq 2$) then
      $\delta = \delta - 2,\ q = s_m,\ e = s_{m-1} - s_m r_{m-1},\ S = x^2(S - qR) - x(eR),$
      $V = V - qU - e(U/x \bmod f),\ U = U/x^2 \bmod f$
    else if ($\delta = 1$) then
      $q = s_m,\ e = s_{m-1} - s_m r_{m-1},\ t = R,\ R = x(x(S - qR) - eR),\ S = t,$
      $t = U/x \bmod f,\ U = x(V - qU - et) \bmod f,\ V = t$
    else /*$\delta = 0$*/
      $q = s_m,\ e = s_{m-1} - s_m r_{m-1}$
      if ($e = 0$) then
        $\delta = \delta + 2,\ t = R,\ R = x^2(S - qR),\ S = t,\ t = U,\ U = x^2(V - qU) \bmod f,\ V = t$
      else /*$e = 1$*/
        $t = xR - x^2 e(S - qR),\ R = x(S - qR),\ S = t,\ t = U - e(x(V - qU) \bmod f),\ U = V - qU,\ V = t$
      endif
    endif
  endif
endfor
return U

Algorithm 4. Our Modified Inversion Algorithm Over $GF(2^m)$

To divide one element by another, the finite field division algorithm over $GF(2^m)$ proceeds almost identically the same as that of the inversion algorithm, with the exception that the variable U should be initialized by the dividend, rather than by 1.

III. Our ECSM Architecture

In this section, we describe our ECSM architecture and estimate the execution time as well as the computational complexity of each individual module.

A. Implementation of Our ECSM Algorithm (Algorithm 1)

As explained in Sec. II and also shown in Algorithm 1 (page 5), our ECSM architecture consists of three stages: (1) *Affine-to-Projective* conversion of input point P; (2) *Projective-Scalar-Multiplication* to get output point $Q = kP$; and (3) *Projective-to-Affine* conversion of the output point Q. Let A, M, S and D be the number of

clock cycles required for finite field addition, multiplication, square and division over $GF(2^m)$, respectively. They are used for analyzing the execution time of individual modules, such as *Affine-to-Projective* module, etc.

1) Affine-to-Projective: According to Algorithm 1, this stage has the following operations involved.

$$X_1 = x,\ Z_1 = 1,\ X_2 = x^4 + b,\ Z_2 = x^2$$

Most of them can be carried out in parallel, except the computation of X_2. In the following, the time steps of our implementation are given.

Affine-to-Projective: $2S + 1A$

1 : $Z_2 = x^2,\ X_1 = x,\ Z_1 = 1;\ (1S)$

2 : $T_1 = Z_2^2;\ (1S)$

3 : $X_2 = T_1 + b.\ (1A)$

In time step 1, three operations are carried out in parallel. This is followed by time step 2 and 3 for computing X_2. The execution time of *Affine-to-Projective* is $2S + 1A$.

2) Projective-Scalar-Multiplication: In the main loop of this module, the two sub-modules carried out in each iteration are *Point-Addition* and *Point-Doubling*. The time steps of our implementations of these two sub-modules are given as below.

Point Addition: $2M + 1S + 2A$

1 : $T_1 = X_1 Z_2,\ T_2 = X_2 Z_1;\ (1M)$

2 : $T_3 = T_1 + T_2;\ (1A)$

3 : $B = T_3^2;\ (1S)$

4 : $T_4 = T_1 T_2, T_5 = xB;\ (1M)$

5 : $A = T_4 + T_5.\ (1A)$

Point Doubling: $2M + 1S + 1A + 1MUX$

1 : $X = (k_i = 1)?X_2 : X_1,$
$Z = (k_i = 1)?Z_2 : Z_1;\ (1MUX)$

2 : $T_6 = X^2,\ T_7 = Z^2;\ (1S)$

3 : $T_8 = T_6^2,\ T_9 = T_7^2,\ D = T_6 T_7;\ (1M)$

4 : $T_{10} = bT_9;\ (1M)$

5 : $C = T_8 + T_{10}.\ (1A)$

In time step 1 of *Point-Doubling*, the operation $(b?a_1 : a_2)$ corresponds to a multiplexer MUX. Its delay is one LUT (lookup table), which is equivalent to that of one finite field addition. Therefore, the execution time of *Point-Doubling* and *Point-Addition* are the same, that is $2M + 1S + 2A$. As shown in Algorithm 1 (page 5), there is a set of conditional assignments at the end of each iteration. This set of assignments incurs an additional MUX for each iteration. Since the execution time of one MUX is equivalent to one A, the total execution time for each iteration is $2M + 1S + 3A$.

3) Projective-to-Affine: Below are the time steps of the third stage of our ECSM implementation.

Projective-to-Affine: $1D + 3A + 2M$

1 : $T_1 = X_1/Z_1,\ T_2 = X_2/Z_2,\ T_3 = x^{-1},\ T_4 = x^2;\ (1D)$

2 : $T_5 = T_1 + x,\ T_6 = T_2 + x,\ T_7 = T_4 + y;\ (1A)$

3 : $T_8 = T_5T_6,\ T_9 = T_5T_3;\ (1M)$

4 : $T_{10} = T_7 + T_8;\ (1A)$

5 : $T_{11} = T_9T_{10};\ (1M)$

6 : $T_{12} = T_{11} + y.\ (1A)$

The execution time is $1D + 3A + 2M$. Note that we compute T_9 by one inversion and one multiplication rather than one division. The reason of making this implementation rather than using division is that the execution time will becomes $2D + 3A + 1M$ if we use division, which results in larger execution time. Therefore, we 'divide' the division into inversion and multiplication so that the inversion part can be computed in time step 1.

Computational Complexity. A summary of the resources that are used in the three stages of our ECSM implementation is shown in Table I.

Complexity	Affine to Proj.	Proj. Scalar Mul.	Proj. to Affine
Square	2	$5(m-1)$	1
Addition	1	$3(m-1)$	5
Multiplication	0	$6(m-1)$	3
Division/ Inversion	0	0	3

TABLE I

RESOURCE CONSUMPTION OF THE ECSM ARCHITECTURE (UNOPTIMIZED)

In each iteration of the main loop, we need 5 finite field square modules, 3 addition modules and 6 multiplication modules, while most of them can be reused for reducing the resource consumption (i.e. chip area). More details on resource reuse are given in Sec. III-C.

Next, we describe our implementation of the finite field element operations and estimate their performance and complexity.

B. Finite Field Multiplication over $GF(2^m)$

Our implementation of finite field multiplication is based on the LSD-first digit-serial algorithm described in Sec. II-A. In Fig. 1, one iteration of our implementation is shown. In the figure, the value next to each signal line represents the width of the corresponding signal.

In this implementation, there are two loops corresponding to the two steps in the LSD-first digit-serial algorithm: the loop on the right performs step (1), and the loop on the left computes step (2) where finite field addition is implemented by XOR gates.

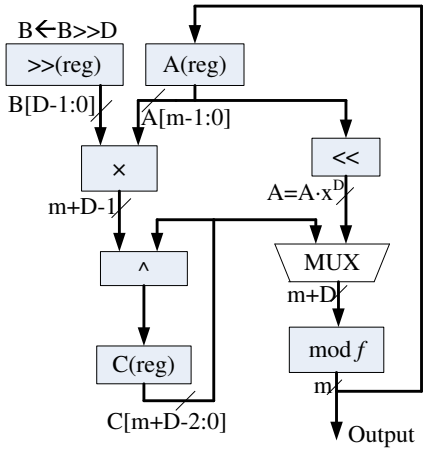

Fig. 1. Our Implementation of LSD-First Digit-Serial Multiplication Algorithm Over $GF(2^m)$

By considering the two extra clock cycles for initializing inputs and uploading outputs, our implementation takes $\lceil m/D \rceil + 2$ clock cycles to complete one finite field multiplication over $GF(2^m)$. Note that the pseudo-pipelined design of [12] cannot be applied in our architecture as the output of one finite field multiplication is used immediately as the input to the next one.

C. Resource Utilization and Performance of Finite Field Element Operation Modules

In our implementation, besides focusing on optimizing the overall performance, we also emphasize on the reuse of modules. In hardware architecture design, resource utilization is one of the most important issues that need to be considered. The reuse of some large modules will help reduce resource consumption or chip area tremendously. In this section, we discuss the resource utilization of the five basic finite field element operation modules in our implementation and also the reuse of them. They are finite field addition (ADD), square (SQU), multiplication (MUL), division (DIV) and inversion (INV). In Table II, the complexity and performance of these five modules are shown.

	LUTs	FFs	Clock Cycles
ADD	163	0	1
SQU	165	0	1
MUL	6200	1937	$\lceil m/D \rceil + 2$
DIV	4909	1506	m
INV	4456	1456	m

TABLE II

COMPLEXITY AND PERFORMANCE OF FIVE FINITE FIELD ELEMENT OPERATION MODULES OVER $GF(2^{163})$

ADD and SQU modules use much less resource than the other three modules. Also, only a few SQUs and ADDs are required, so we choose not to optimize the reusability of these two modules. In fact, to reuse a module, one additional multiplexer (MUX) is required, and too many MUXs may counteract the advantage of reusing modules when these modules are small and already efficient enough.

We put much effort on reusing the other three modules. In Table III, it shows the utilization of the SQU, ADD, DIV and INV modules in our implementation.

	Stage 1			Stage 2					Stage 3					
	S1	S2	S3	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5	S6
SQU1	Z_2	T_1			T_7									
SQU2						B								
SQU3						T_9								
SQU4					T_6									
SQU5						T_8								
SQU6									T_4					
ADD1			X_2											
ADD2								C						
ADD3					T_3			A						
ADD4										T_5				
ADD5										T_6				
ADD6										T_7				
ADD7												T_{10}		
ADD8														T_{12}
DIV1									T_1					
DIV2									T_2					
INV1									T_3					

TABLE III

UTILIZATION OF SQU, ADD, DIV AND INV MODULES IN OUR ECSM IMPLEMENTATION

In the table, Stage 1, Stage 2 and Stage 3 correspond to *Affine-to-Projective*, *Projective-Scalar-Multiplication* and *Projectibe-to-Affine*, respectively, where Stage 2 includes $m-1$ iterations. T_i corresponds to the variable in the analysis of execution time in Sec. III-A. We use 6 SQU modules and 8 ADD modules in our implementation, only one SQU module and one ADD module are reused.

Since two division modules are required to perform in parallel for shortening the execution time, in our implementation, we choose not to reuse the DIV module in preference of fast computation.

On the MUL modules, we make extra effort in allocating this type of modules. The reason is that MUL modules not only occupy large resources, but are also used very frequently. As we can see in Sec. III-A, 6 MUL modules (unoptimized) are needed for carrying out one iteration of the main loop of the *Projective-Scalar-Multiplication*. For reducing the number of MUL modules in this stage, we divide the 6 finite field multiplication operations over

$GF(2^m)$ in the time steps of *Point Addition* and *Point Doubling* into two groups:

1) In Step 4 of both *Point Addition* and *Point Doubling*, altogether three finite field multiplications can be carried out in parallel;
2) In Step 1 of *Point Addition* and Step 3 of *Point Doubling*, altogether three finite field multiplications can be carried out in parallel.

Therefore, we only require three MUL modules to run in parallel in each group and they can be reused in the other group. In addition, in the last stage, that is, *Projective-to-Affine* conversion, three MUL modules are needed and only two are required to be run in parallel. Therefore, the same set of MUL modules can also be reused in this stage. As a result, there are only three MUL modules in our implementation and Tabel IV shows the details of the utilization of these MUL modules.

	Stage 1			Stage 2					Stage 3					
	S1	S2	S3	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5	S6
MUL1				T_1			T_4				T_8			
MUL2				T_2			T_5				T_9			
MUL3						D	T_{10}						T_{11}	

TABLE IV

UTILIZATION OF MUL MODULES IN OUR ECSM IMPLEMENTATION

D. The Complete Architecture and Total Execution Time of Our ECSM Implementation

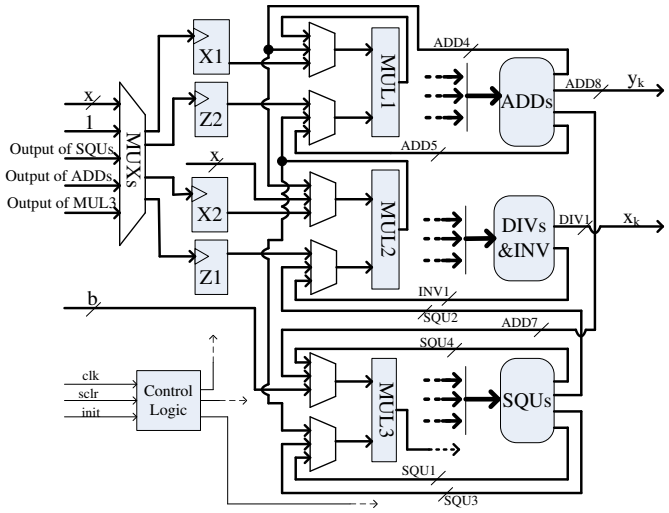

Fig. 2. Our ECSM Architecture

The architecture of our ECSM implementation is shown in Fig. 2. In the diagram, clk, sclr and init represent global clock signal, synchronous reset signal and global initial signal, respectively. Inputs for each MUL module are shown in the figure, and there are only three MUL modules in our design.

We now estimate the total execution time in terms of the number of clock cycles. In one iteration of the main loop of *Projective-Scalar-Multiplication* (Sec. III-A2), the execution time is $2M+1S+3A$. Therefore, the execution time of *Projective-Scalar-Multiplication* is $(m-1)(2(\lceil m/D \rceil + 2)+4)$. For the two conversions, i.e. *Affine-to-Projective* (Sec. III-A1) and *Projective-to-Affine* (Sec. III-A3), the total execution time is $m + 2(\lceil m/D \rceil + 2) + 6$ which is about m for large D. Hence, the total number of clock cycles taken to compute one ECSM operation over $GF(2^m)$ is about $2(m-1)(\lceil m/D \rceil + 4) + m$.

IV. Performance of an Actual Implementation

We use Xilinx Integrated Software Environment (ISE) 9.1i to develop our ECSM hardwired logic. Simulation is done using Modelsim XE. The implementation is tested on a Xilinx Virtex-4 LX200 FPGA.

The finite field chosen in our actual implementation is $GF(2^{163})$ with the irreducible polynomial being set to $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. The elliptic curve is sect163r1[3]. When choosing the digit size D of the finite field multiplication (i.e. LSD-first Digit-Serial Multiplication) as 42, the whole system takes 39,584 lookup tables (LUTs) and 6,948 flip flops (FFs). The corresponding number of ASIC gates estimated by the ISE is 303,822. One ECSM operation can be completed in $12.5\mu s$ with the maximum achievable frequency of 222MHz. This result corresponds to the case when integer k is taken as $2^{163}-1$, that is, a 163-bit integer with all bits set to 1. This implies that in practice, our implementation will take less than $12.5\mu s$ to perform on ECSM operation over $GF(2^{163})$.

	FPGA	ECSM over $GF(2^{163})$ (μs)	Max Achievable Frequency (MHz)	Resources Occupied
Our Implementation	Virtex-4 LX 200	**12.5**	**222**	39,584 LUTs (≈303,822 gates)
[12]	Virtex-II 2000	41	100	8,300 LUTs, 7 RAM blocks
[9]	Virtex-E 400-8BG432	210	76.7	3,002 LUTs 10 RAM blocks
[10]	Virtex-E 2000-7	144	66.4	20,068 LUTs
[11]	Virtex 800-4HQ240	3,801	47	150,678 gates

TABLE V
Absolute Performance Comparison

[3]Standards for Efficient Cryptography Group, "Recommended Elliptic Curve Domain Parameters", September 2000. Available at `http://www.secg.org/index.php?action=secg,docs_secg`

In Table V, several recent results in hardware implementation of ECSM are given along with our result. We can see that our result ($12.5\mu s$) is at least three times the speed of the best result previously known (i.e. $41\mu s$) [12][4]. On the resource consumption (or chip area requirement), it seems that our implementation requires much more resources, for example, when compared with that of [12]. Actually, this is not the case. In fact, our implementation gives better performance-area ratio. This implies that our implementation yields faster computation than that of [12] if the same amount of chip area is given. In Table VI, the performance-area ratios of our implementation when compared with the best known results are given.

	Computation Time	Resources Occupied	Performance-Area Ratio
Ours : [12]	$3.28 : 1^{\dagger}$	$2.42 : 1^{\dagger\flat}$	$1.36 : 1^{\dagger}$
Ours : [9]	16.8:1	$7.1 : 1^{\ddagger}$	2.37:1
Ours : [10]	11.5:1	1.94:1	5.93:1
Ours : [11]	304:1	2.02:1	150.5:1

† Incomparable as [12] does not consider coordinate conversion.

$^{\flat}$ One 18k-bit RAM block in Virtex-II corresponds to 1,152 LUTs.

‡ One 4k-bit RAM block is approximately equivalent to 256 LUTs.

TABLE VI

RELATIVE PERFORMANCE COMPARISON

We can see in Table VI, our implementation also outperforms that of [12] even when considering the performance-area ratio. We should emphasize that in [12], *Affine-to-Projective* and *Projective-to-Affine* conversions are not implemented. Input and output of the ECSM in [12] are all in projective coordinate. Therefore, our implementation is actually much better in both absolute performance as well as performance-area ratio when compared with [12]. Comparing with [9], [10] and [11], the utilization efficiency of our implementation, i.e., performance-area ratio, is also much better.

V. OPTIMIZATION FOR RESOURCE CONSUMPTION

The estimated number of ASIC gates of our implementation is about 303,822 (Sec. IV). Higher the number of gates is, more expensive the ASIC chips are. For low-cost products, the resource consumption of our design may need to be reduced. In this section, we discuss how it can be reduced without introducing too much tradeoff on the performance.

[4]Note that in [12], only the ECSM in projective coordinate was implemented while the two conversions between affine coordinate and projective coordinate are not considered, while in our implementation, both conversions have been taken into account.

A. Variant 1

Our original ECSM implementation has three MUL modules. As mentioned, the MUL module is expensive and it gives significant reduction on resource consumption if fewer MUL modules are needed. As a result, our first variant from the original implementation is to reduce the number of MUL modules to only one. In other words, there is only one MUL module available for *Projective-Scalar-Multiplication*. Below are the time steps of the *Projective-Scalar-Multiplication* stage of this variant.

Variant 1: Projective-Scalar-Multiplication: $6M + 1A$

1 : $T_1 = (k_i = 1)?X_2Z_1 : X_1Z_2,\ \ X = (k_i = 1)?X_2 : X_1,\ Z = (k_i = 1)?Z_2 : Z_1$; $(1M)$

2 : $T_2 = (k_i = 1)?X_1Z_2 : X_2Z_1,\ \ T_6 = X^2,\ \ T_7 = Z^2$; $(1M)$

3 : $D = T_6T_7,\ \ T_3 = T_1 + T_2,\ \ T_8 = T_6^2,\ \ T_9 = T_7^2$; $(1M)$

4 : $T_4 = T_1T_2,\ \ B = T_3^2$; $(1M)$

5 : $T_{10} = bT_9$; $(1M)$

6 : $T_5 = xB,\ \ C = T_8 + T_{10}$; $(1M)$

7 : $A = T_4 + T_5$. $(1A)$

We can see that the execution time of one iteration in the main loop becomes $6M+1A$. Different from our original design in Sec. III and IV, the pseudo-pipelined technique [12] can now be employed, that is, the clock cycle for preparing the output of the previous execution of the MUL module can be done simultaneously with the loading of the input for the current execution of the MUL module. Hence, the effective number of clock cycles required for completing one finite field multiplication can be reduced to $\lceil m/D \rceil + 1$. Also note that registers storing the values of B, C and D are ready before the last time step of one iteration. Hence, the addition $(1A)$ in step 7 can be carried out simultaneously with the first time step $(1M)$ of the next iteration. As a result, the effective execution time of one iteration is only $6M$ rather than $6M + 1A$ for the *Projective-Scalar-Multiplication* stage.

In addition to the above, in *Projective-to-Affine* conversion, we also reduce the number of DIV modules from two in our original design to one, and eliminate the INV module. Below are the time steps of the *Projective-to-Affine* conversion of this variant.

Variant 1: Projective-to-Affine: $3D + 2M + 2A$

1 : $T_1 = X_1/Z_1,\ \ T_4 = x_2$; $(1D)$

2 : $T_2 = X_2/Z_2,\ \ T_5 = T_1 + x,\ \ T_7 = T_4 + y$; $(1D)$

3 : $T_6 = T_2 + x,\ \ T_9 = T_5/x$; $(1D)$

4 : $T_8 = T_5T_6$; $(1M)$

5 : $T_{10} = T_7 + T_8$; $(1A)$

6 : $T_{11} = T_9T_{10}$; $(1M)$

7 : $T_{12} = T_{11} + y$; $(1A)$

The execution time of this stage is increased to $3D + 2M + 2A$ (in our original design, the time of this stage is $1D + 2M + 3A$). The first stage, that is, the *Affine-to-Projective* conversion, remains unchanged. Therefore, the total execution time of this ECSM variant is about $6(m - 1)(\lceil m/D \rceil + 1) + 3m$.

B. Variant 2

Another variant is to use two MUL modules rather than one as in Variant 1. The pseudo-pipelined technique [12] can also be applied in this variant. In the *Projective-to-Affine* conversion, similar to Variant 1, we only keep one DIV module, while having the INV module removed. The following is the time steps of the parts corresponding to *Point Addition* and *Point Doubling* in the *Projective-Scalar-Multiplication* stage (also refer to Algorithm 1 on page 5).

Variant 2: Point Addition and Point Doubling: $1MUX + 3M + 1A$

1 : $X = (k_i = 1)?X_2 : X_1,\ \ Z = (k_i = 1)?Z_2 : Z_1;\ (1MUX)$

2 : $T_1 = X_1Z_2,\ \ T_2 = X_2Z_1,\ \ T_6 = X^2,\ \ T_7 = Z^2;\ (1M)$

3 : $D = T_6T_7,\ \ T_{10} = bT_9,\ \ T_3 = T_1 + T_2,\ \ T_8 = T_6^2,\ \ T_9 = T_7^2;\ (1M)$

4 : $B = T_3^2,\ \ T_4 = T_1T_2,\ \ T_5 = xB;\ \ (1M)$

6 : $A = T_4 + T_5,\ \ C = T_8 + T_{10};\ \ (1A)$

In our implementation, step 1 in the time steps above is performed in parallel with the two multiplications of step 2. Before time step 2 ends, the remaining two finite field square operations can also be completed. Therefore, the execution time of *Point Addition* and *Point Doubling* in one iteration of the main loop in the *Projective-Scalar-Multiplication* stage is $3M + 1A$. As described in Algorithm 1, this is followed by a set of conditional assignments. Therefore, $1MUX$ has to be added for completing one iteration in *Projective-Scalar-Multiplication*. As a result, the total execution time of the *Projective-Scalar-Multiplication* stage in Variant 2 is $(m-1)(3(\lceil m/D \rceil + 1) + 2)$.

On the two coordinate conversions, they remain unchanged when compared with Variant 1. As a result, the total execution time of Variant 2 is about $(m-1)(3\lceil m/D \rceil + 5) + 3m$.

C. Comparison

Table VII shows the resource consumption and performance of these two variants for different values of D, where D is the digit size of the underlying LSD-first digit-serial multiplication operation. The resource consumption of LUTs and FFs varies with D and the MUL module is corresponding to this variability.

In Table VII, we can see that the variants, when having appropriate values of D chosen, can provide very good performance with tremendously reduced resource consumption. For example, Variant 2 with D chosen to be 42 or 32 can complete one ECSM operation over $GF(2^{163})$ within $20\mu s$, which is still more than twice the speed of the best result previously known [12], while having over 30% reduction on the estimated number of ASIC gates. In particular, for Variant 2 when $D = 42$, the speed is only slightly lower than our result in Sec. IV, in which we use three MUL modules, two DIV modules and one INV module. The advantage on utilization efficiency becomes more explicit when we compare the performance-area ratio of these variants with previous results. Table VIII shows the performance-area ratio of these two variants to the result of [9].

In Table VIII, significant improvement on the utilization efficiency can be found when D is set to 42 or 32 for Variant 2. The performance-area ratio of Variant 2 outperforms our original result in Sec. IV. The table shows that Variant 2 actual provides the best combination of utilization efficiency and speed. Similar results can also be found

	LUTs	FFs	Estimated ASIC Gates	Max. Achievable Freq. (MHz)	Time (μs)
Variant 1 ($D = 42$)	18,782	4,664	158,482	202.1	26.6
Variant 1 ($D = 32$)	16,747	4,640	145,948	200.7	36.5
Variant 1 ($D = 16$)	15,210	4,512	139,850	205.3	59.4
Variant 2 ($D = 42$)	24,972	5,695	201,536	220.2	**14.8**
Variant 2 ($D = 32$)	22,774	5,781	191,385	216.7	19.6
Variant 2 ($D = 16$)	18,416	5,597	163,735	217.0	30.8

TABLE VII

PERFORMANCE AND COMPLEXITY OF TWO VARIANTS OF OUR ECSM ARCHITECTURE

	Execution Time	Resources Occupied	Performance-Area Ratio
Variant 1 ($D = 42$) : [9]	7.90: 1	3.38:1	2.34:1
Variant 1 ($D = 32$) : [9]	5.75: 1	3.01:1	1.91:1
Variant 1 ($D = 16$) : [9]	3.54: 1	2.73:1	1.30:1
Variant 2 ($D = 42$) : [9]	14.19 : 1	4.49:1	**3.16:1**
Variant 2 ($D = 32$) : [9]	10.71: 1	4.09:1	2.62:1
Variant 2 ($D = 16$) : [9]	6.82: 1	3.31:1	2.06:1

TABLE VIII

COMPARISON WITH [9]

when compared with other related results such as those in [12], [10], [11]. We choose not to show the comparison with [12] in Table VIII, as explained earlier, the result shown in [12] does not include the *Affine-to-Projective* and *Projective-to-Affine* conversions.

VI. CONCLUSION

We proposed a highly efficient new architecture for performing ECSM on elliptic curves over $GF(2^m)$. Main efforts have been put on maximizing the parallelism of the execution of independent functional modules. Our implementation takes about $2(m-1)(\lceil m/D \rceil + 4) + m$ cycles to complete one ECSM operation. This is about one third of the number of cycles required by another best known result due to Anasari and Hansan [12]. When implemented on a Virtex4-LX200 FPGA, our implementation completes one ECSM operation on an elliptic curve

over $GF(2^{163})$ in $12.5\mu s$ with a maximum frequency of 222MHz achieved. This is at least three times the speed of [12]. Note that our implementation also includes coordinate conversions that are not considered in [12].

We also maximized the parallelism that the Montgomery ECSM algorithm in projective coordinate can achieve. By adjusting the maximum delay paths of Point Addition and Point Doubling, we eliminated any idle time of Point Doubling when these two modules are running in parallel. Our optimized design consists of only two finite field multiplications in the maximum delay path. This is much shorter than previously proposed designs.

For reducing the resource consumption, we implemented two variants. Our first variant reduced the number of MUL modules from three to one, DIV modules from two to one and eliminated the INV module. Our results showed that the resource consumption can be reduced by almost 50% while still maintaining the utilization efficiency (i.e. the performance-area ratio). Our second variant is similar to the first variant but having two MUL modules. This modification is significant because it allows high parallelism to be achieved by our core ECSM computation. Through this optimization, we achieved the best resource utilization efficiency. It yielded 3.16:1 performance-area ratio when compared with 2.37:1 in our original implementation. It also gave us more than 30% reduction on resource consumption while maintaining almost the same speed of computation as that of our original implementation.

On the development of underlying finite field element operations, we proposed a highly efficient finite field inversion algorithm and deploy it in our ECSM implementation. Our algorithm is a variant of the Extended Euclid algorithm but it takes only m cycles to invert an element in $GF(2^m)$ rather than $2m$ cycles for a traditional implementation of the Extended Euclid algorithm. More importantly, it makes one iteration of the main loop of our algorithm matches to the time elapsed for one clock cycle much better than the traditional one. This modification helps improve the overall performance of our implementation significantly without reducing much on the maximum achievable frequency.

REFERENCES

[1] V. Miller, "Uses of elliptic curves in cryptography," in *Proc. CRYPTO 85*. Springer-Verlag, 1985, pp. 417–426, lecture Notes in Computer Science No. 218. (Cited on page 1.)

[2] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987. (Cited on page 1.)

[3] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. (Cited on page 1.)

[4] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*. Cambridge University Press, 1999. (Cited on page 1.)

[5] A. K. Lenstra and E. R. Verheul, "Selecting cryptographic key sizes," *Journal of Cryptology*, vol. 14, no. 4, pp. 255–293, 2001. (Cited on page 1.)

[6] J. H. Silverman, *The arithmetic of elliptic curves*. Springer-Verlag, 1986. (Cited on page 1.)

[7] IEEE, *IEEE 1363-2000: Standard Specifications for Public Key Cryptography*, 2000, `http://grouper.ieee.org/groups/1363`. (Cited on pages 1 and 4.)

[8] NSA, *Suite B Cryptography*, 2005, `http://www.nsa.gov/ia/industry/crypto_suite_b.cfm`. (Cited on pages 1 and 2.)

[9] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for $GF(2^m)$," in *CHES 2000*. Springer-Verlag, 2000, pp. 41–56, lecture Notes in Computer Science No. 1965. (Cited on pages 2, 3, 4, 5, 14, 15, 17, and 18.)

[10] H. Eberle, N. Gura, and S. C. Shantz, "A cryptographic processor for arbitrary elliptic curves over $GF(2^m)$," in *14th International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2003)*, Jun. 2003, pp. 444–454. (Cited on pages 2, 3, 5, 14, 15, and 18.)

[11] N. Mentens, S. Berna Örs, and B. Preneel, "An FPGA implementation of an elliptic curve processor over $GF(2^m)$," in *ACM Proceedings of the 2004 Great Lakes Symposium on VLSI (GLSVLSI 2004)*, 2004, pp. 454–457. (Cited on pages 2, 3, 5, 14, 15, and 18.)

[12] B. Ansari and M. A. Hasan, "High performance architecture of elliptic curve scalar multiplication," The University of Waterloo, Tech. Rep. CACR 2006-01, 2006. (Cited on pages 2, 4, 5, 11, 14, 15, 16, 17, 18, and 19.)

[13] J. López and R. Dahab, "Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation," in *CHES'99*. Springer-Verlag, 1999, pp. 316–327, lecture Notes in Computer Science No. 1717. (Cited on pages 4, 5, 6, and 20.)

[14] ——, "Improved algorithms for elliptic curve arithmetic in $GF(2^m)$," in *SAC'98*. Springer-Verlag, 1998, pp. 201–212, lecture Notes in Computer Science No. 1556. (Cited on page 4.)

[15] P. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987. (Cited on page 5.)

[16] L. Song and K. Parhi, "Low-energy digit-serial/parallel finite field multipliers," *Journal of VLSI Signal Processing*, vol. 19, pp. 149–166, 1999. (Cited on page 6.)

[17] H. Wu, "Low complexity bit-parallel finite field arithmetic using polynomial bases," in *CHES'99*. Springer-Verlag, 1999, pp. 280–291, lecture Notes in Computer Science No. 1717. (Cited on pages 6 and 7.)

[18] T. Kerins, E. M. Popovici, and W. P. Marnane, "Algorithms and architectures for use in FPGA implementations of identity based encryption schemes," in *FPL 2004*. Springer-Verlag, 2004, pp. 74–83, lecture Notes in Computer Science No. 3203. (Cited on page 7.)

APPENDIX

ECSM ALGORITHMS OF [13]

Montgomery-Scalar-Multiplication(x, y, k)
Input: Point $P = (x, y)$ and k where $x, y, k \in GF(2^m)$
Output: Point $Q = (x_q, y_q) = kP$, $x_q, y_q \in GF(2^m)$
$(X_1, Z_1, X_2, Y_2) =$ **Affine-to-Projective**(x, y);
for i from $l - 2$ downto 0 do
 if $(k_i = 1)$ then
  $(X_1, Z_1) =$ **Point-Addition**(X_1, Z_1, X_2, Z_2, x);
  $(X_2, Z_2) =$ **Point-Doubling**(X_2, Z_2);
 else
  $(X_2, Z_2) =$ **Point-Addition**(X_2, Z_2, X_1, Z_1, x);
  $(X_1, Z_1) =$ **Point-Doubling**(X_1, Z_1);
 endif
endfor
Return (**Projective-to-Affine**$(X_1, Z_1, X_2, Z_2, x, y)$)

Affine-to-Projective(x, y)
Set $X_1 = x; Z_1 = 1; X_2 = x^4 + b; Z_2 = x^2$;
Return (X_1, Z_1, X_2, Z_2)

Point-Addition(X_1, Z_1, X_2, Z_2, x)
Set $X = X_1Z_2X_2Z_1 + x(X_1Z_2 + X_2Z_1)^2$,
 $Z = (X_1Z_2 + X_2Z_1)^2$,
Return (X, Z)

Point-Doubling(X, Z)
Set $X = X^4 + bZ^4$,
 $Z = X^2Z^2$
Return (X, Z)

Projective to Affine$(X_1, Z_1, X_2, Z_2, x, y)$
Set $x_q = X_1/Z_1$,
 $y_q = ((X_1/Z_1 + x)(X_2/Z_2 + x) + (x^2 + y))$
  $*(X_1/Z_1 + x)/x + y$
Return: (x_q, y_q)