Generic Implementations of Elliptic Curve Cryptography using Partial Reduction

Nils Gura
Sun Microsystems Laboratories
2600 Casey Ave
Mountain View, CA 94043
Nils.Gura@sun.com

Hans Eberle
Sun Microsystems Laboratories
2600 Casey Ave
Mountain View, CA 94043
Hans.Eberle@sun.com

Sheueling Chang Shantz
Sun Microsystems Laboratories
2600 Casey Ave
Mountain View, CA 94043
Sheueling.Chang@sun.com

ABSTRACT

Elliptic Curve Cryptography (ECC) is evolving as an attractive alternative to other public-key schemes such as RSA by offering the smallest key size and the highest strength per bit. The importance of ECC has been recognized by the US government and the standards bodies NIST and SECG. Standards for preferred elliptic curves over prime fields $GF(p)$ and binary polynomial fields $GF(2^m)$ as well as the Elliptic Curve Digital Signature Algorithm (ECDSA) have been created. A security protocol based on ECC requires support for different curves representing different security levels. This is particularly true for server applications that are exposed to requests for secure connections with different parameters generated by a multitude of client devices. Reported implementations of ECC over $GF(2^m)$ typically choose to implement each curve as a special case so that modular reduction can be optimized, thus improving the overall performance. In contrast, this paper focuses on generic implementations of ECC point multiplication for arbitrary curves over $GF(2^m)$. We present a novel reduction algorithm that allows hardware and software implementations for variable field degrees m. Though not as high in performance as an implementation optimized for a specific curve, it offers an attractive solution to supporting infrequently used curves or curves not known at the time of the implementation.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Public key cryptosystems

General Terms

Algorithms

Keywords

Elliptic curve cryptography, modular reduction

1. INTRODUCTION

Elliptic Curve Cryptography (ECC) is evolving as an attractive alternative to other public-key schemes such as RSA by offering the smallest key size and the highest strength per bit. Extensive research has been done on the underlying math, security strength and efficient implementations.

Among the different fields that can underlie elliptic curves, prime fields $GF(p)$ and binary polynomial fields $GF(2^m)$ have shown to be best suited for cryptographic applications. In particular, binary polynomial fields allow for fast computation in software as well as in hardware.

Small key sizes and computational efficiency make ECC not only applicable to hosts processing security protocols over wired networks, but also to small wireless devices such as cell phones, PDAs and Smartcards.

To make ECC commercially viable, its integration into security protocols needs to be standardized. As an emerging alternative to RSA, the US government has adopted ECC for the Elliptic Curve Digital Signature Algorithm (ECDSA) and specified *named curves* [19]. For binary polynomial fields, these curves were chosen for key sizes of 163, 233, 283, 409 and 571 bits. Additional curves for commercial use were recommended by the Standards for Efficient Cryptography Group (SECG) [6]. However, only few ECC-enabled protocols have been deployed so far. Today's dominant Internet security protocols such as SSL and IPsec rely on RSA and the Diffie-Hellman key exchange. Although standards for the integration of ECC have been proposed [9] [5], they have not yet been finalized.

The mathematical simplicity of RSA and the Diffie-Hellman key exchange allow for a straight-forward implementation of the underlying arithmetic operations. Standard implementations are available in various cryptographic libraries. Arithmetically, RSA and the Diffie-Hellman key exchange operate on prime fields and primarily involve modular multiplications. In comparison, ECC is more complex. It is specified over both prime and binary polynomial fields and involves modular addition, multiplication and division. Not only the implementation, but also the algorithm may be tailored to the system architecture and constraints such as processor speed, data path width or memory size.

Our approach towards an end-to-end solution is driven by a scenario of a wireless and web-based environment where millions of client devices connect to secure servers. Clients may choose different key sizes and curves depending on vendor preferences, security requirements and processor capabilities. In addition, different types of transactions may require different security levels. The aggregation of client connections/transactions leads to high computational demand on the server side, which calls for hardware acceleration.

While optimized implementations for specific named curves and field degrees can provide high performance, it is a desired security feature for both ECC software libraries and hardware accelerators to support elliptic curves over a wide range of binary polynomial fields $GF(2^m)$. In particular, the implementer of an ECC library or hardware platform may not know all curves that will eventually be used. Vendors may change their selection of curves according to security considerations, computational efficiency, market conditions and corporate policies. For hardware implementations in ASIC technology, this may result in architectural changes and costly redesigns. Also, there may be a need to support curves that are infrequently used and do not call for optimized performance. *Partial reduction* addresses these problems by allowing for generic, curve-independent implementations of ECC over $GF(2^m)$ in hardware and software.

The paper is structured as follows. Section 2 summarizes related work on ECC hardware and software implementations. In Section 3, we describe point multiplication, the fundamental operation underlying ECC. A more detailed presentation of the arithmetic operations is given in Section 4. The problem of modular reduction is addressed in Sections 5 and 6. Applications of partial reduction in hardware and software are presented in Section 7. A hardware implementation of a programmable processor optimized for ECC point multiplications is presented in Section 8. Section 9 contains performance measurements. We conclude and discuss future directions in Section 10.

2. RELATED WORK

Software and hardware implementations of ECC point multiplication over $GF(2^m)$ for different values of m have been reported in numerous publications. Software implementations are described in [17], [20] and [15]. Hardware implementations can be found in [16], [2], [1] and [8].

Schroeppel et al. implemented an ECC equivalent of the Diffie-Hellman key exchange for $GF(2^{155})$ on 32-bit and 64-bit processor architectures [17]. A design for $GF((2^8-17)^{17})$, optimized for 8-bit processors, is described by Woodbury et al. in [20]. The implementation targets a Smartcard based on an Intel 8051 microcontroller. López and Dahab proposed a version of Montgomery's point multiplication algorithm using projective coordinates and implemented it for $GF(2^{163})$, $GF(2^{191})$ and $GF(2^{239})$ [15].

Orlando and Paar describe a programmable elliptic curve processor for FPGA technology in [16]. Different curves can be handled by parameterizing the hardware architecture and reconfiguring the logic. The prototype performs point multiplication on curves over $GF(2^{167})$. Bednara et al. [2] designed an FPGA-based ECC processor architecture that allows for using multiple squarers, adders and multipliers. They researched hybrid coordinate representations in affine, projective, Jacobian and López-Dahab form. Two prototypes were synthesized for $GF(2^{191})$. Agnew et al. [1] built an ASIC implementing ECC point multiplication for $GF(2^{155})$. The chip uses an optimal normal basis multiplier exploiting the composite field property of $GF(2^{155})$. Goodman and Chandrakasan [8] designed a generic public-key processor optimized for low power consumption that executes modular operations on prime and binary polynomial fields. The architecture is designed around a bit-serial arithmetic unit that can be dynamically reconfigured to support different field operations. Point multiplication over binary polynomial fields is computed by a microcoded double-and-add algorithm. To our knowledge, this is the only reported implementation that supports $GF(2^m)$ for variable field degrees m. All other implementations described above target either one or a small number of specific curves. That is, none of them can handle a curve that is not specified at implementation time without requiring the software to be modified or the hardware to be reconfigured.

An implementation of ECC over various prime fields $GF(p)$ using incomplete reduction was presented by Yanık et al. in [21]. Similar to partial reduction for $GF(2^m)$, this scheme tries to speed up modular arithmetic in $GF(p)$ by reducing operands to multiples of machine words (e.g. 32-bit) to avoid bit-level operations that are costly on most microprocessors.

3. POINT MULTIPLICATION

The fundamental and most expensive operation underlying ECC is *point multiplication*, which is defined over finite field operations[1]. For a non-supersingular elliptic curve $C: y^2+xy = x^3+ax^2+b; x, y \in GF(2^m)$ with curve parameters $a, b \in GF(2^m)$ over a binary polynomial field $GF(2^m)$, an additive Abelian group of points $G = (S, +)$ can be defined. $S = \{(x,y)|(x,y)\,\text{satisfies}\,C\} \cup 0$ includes all points on C and a point at infinity denoted by 0. The neutral element of G is 0 and the inverse of a point $P = (x, y)$ is $-P = (x, x+y)$. The addition of two points is defined by

$$P = (x,y) = P_1 + P_2 =$$
$$\begin{cases} \text{if } P_1 = 0\text{:} \\ P_2 \\ \text{if } P_2 = 0\text{:} \\ P_1 \\ \text{if } P_1 \neq P_2, P_1 \neq -P_2\text{:} \\ x = (\frac{y_1+y_2}{x_1+x_2})^2 + (\frac{y_1+y_2}{x_1+x_2}) + a + x_1 + x_2 \\ y = \frac{y_1+y_2}{x_1+x_2} * (x_1+x) + x + y_1 & (1a) \\ \text{if } P_1 \neq P_2, P_1 = -P_2\text{:} \\ 0 & (1b) \\ \text{if } P_1 = P_2, x_1 \neq 0\text{:} \\ x = x_1^2 + \frac{b}{x_1^2} \\ y = x_1^2 + (x_1 + \frac{y_1}{x_1}) * x + x & (1c) \\ \text{if } P_1 = P_2, x_1 = 0\text{:} \\ 0 & (1d) \end{cases}$$

Cases $(1a)$ and $(1b)$ describe a *point addition* and cases $(1c)$ and $(1d)$ describe a *point doubling*. For a point P in G and a positive integer k, the point multiplication kP is defined by adding P $k-1$-times to itself, e.g. $4P = P+P+P+P$.

Various algorithms have been proposed to efficiently compute point multiplications[2]. For our implementation, we experimented with different point multiplication algorithms and settled on Montgomery's point multiplication algorithm

[1]For a detailed mathematical background on ECC the reader is referred to [4].

[2]For a survey of point multiplication algorithms the reader is referred to [12].

using projective coordinates as proposed by López and Dahab [15]. This algorithm allows for simple implementations in both hardware and software. It avoids expensive divisions by representing affine point coordinates (x, y) as projective triples (X, Y, Z) with $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$. In addition, it reduces the number of arithmetic operations by only computing the x-coordinate of intermediate points. Hardware implementations can exploit the fact that most multiplications can be executed in parallel to squarings or additions. Using projective coordinate representation, Montgomery point multiplication requires $6\lfloor log_2(k)\rfloor + 9$ multiplications, $5\lfloor log_2(k)\rfloor + 3$ squarings, $3\lfloor log_2(k)\rfloor + 7$ additions and 1 division.

4. ECC ARITHMETIC IN $GF(2^M)$

Elliptic curve cryptography over finite fields is based on modular addition, subtraction, multiplication, squaring and division. These operations are specific to the underlying field. In this paper, we will focus on binary polynomial fields denoted by $GF(2^m)$. $GF(2^m)$ is an extension field of $GF(2)$ with m being the extension or field degree. More precisely, $GF(2^m)$ can be represented by an irreducible polynomial $M = t^m + t^k + \sum_{j=1}^{k-1} M_j t^j + 1, M_j \in GF(2), 1 \leq k < m$. The elements of $GF(2^m)$ are represented as polynomials over $GF(2)$ and can be expressed using different bases such a polynomial (standard) bases and normal bases. Using polynomial bases, a polynomial $a \in GF(2^m)$ in canonical form can be written as $a = a_{m-1}t^{m-1} + a_{m-2}t^{m-2} + \cdots + a_1 t + a_0$. The coefficients a_i are elements of $GF(2)$, i.e. they can be either 0 or 1. For efficient computation, polynomials can be stored as bit strings representing their coefficients $(a_{m-1}a_{m-2} \ldots a_1 a_0)$. For the remainder of this paper, we assume that polynomials are represented using polynomial bases.

The addition of two elements $a, b \in GF(2^m)$ is defined as the sum of the two polynomials obtained by adding the coefficients, i.e.

$$\begin{aligned} c = &a + b \\ =&(a_{m-1} + b_{m-1})t^{m-1} \\ &+ (a_{m-2} + b_{m-2})t^{m-2} \\ &+ \cdots + (a_1 + b_1)t + (a_0 + b_0) \end{aligned}$$

$GF(2)$ addition of two coefficients $a_i + b_i$ corresponds to a logical XOR and can be implemented efficiently in both software and hardware. Since every element of $GF(2^m)$ is identical to its additive inverse, subtraction is identical to addition.

Multiplication of two elements $a, b \in GF(2^m)$ is carried out in two steps. First, the operands are multiplied using polynomial multiplication resulting in

$$\begin{aligned} c_0 = a * b = &c_{0,2(m-1)}t^{2(m-1)} \\ &+ c_{0,2(m-1)-1}t^{2(m-1)-1} \\ &+ \cdots + c_{0,1}t + c_{0,0} \end{aligned}$$

The degree of c_0 is less than $2m - 1$, i.e. $\deg(c_0) < 2m - 1$. The coefficients of c_0 are calculated through convolution of a and b

$$c_{0,i} = \sum_{k=0}^{i} a_k b_{i-k} \tag{2}$$

c_0 may not be in reduced canonical notation since its degree may be greater than $m - 1$. Second, c_0 is reduced by the irreducible polynomial M to a polynomial of less than the field degree m. The reduced canonical result $c \equiv c_0 \mod M$, $\deg(c) < m$ is defined as the residue of the polynomial division of c_0 by M.

Polynomial multiplication can be efficiently implemented using well-known techniques such as the Ofman-Karatsuba method [14]. Field multiplication, i.e. polynomial multiplication combined with reduction, can be implemented using techniques such as the LSD or MSD multiplication method [18]. Implementations of reduction will be discussed in detail in Sections 5 and 6.

The first step of a squaring operation, which is a special case of polynomial multiplication, does not require a full multiplication. When squaring a polynomial $a \in GF(2^m)$, all mixed terms $a_i a_j t^k, k = 1 \ldots 2(m-1), k = i + j, i \neq j$ occur twice cancelling each other out. Therefore, $a^2 = a_{m-1}t^{2(m-1)} + a_{m-2}t^{2(m-2)} + \cdots + a_1 t^2 + a_0$ can be computed by inserting zeros into the corresponding bit string. For example, squaring $(t^3 + t^2 + t + 1)$ results in $(1111)^2 = 1010101$.

Division $\frac{a}{b}$, $a, b \in GF(2^m)$ is defined as a multiplication of the dividend a with the multiplicative inverse of the divisor b. Algorithms for finding the inverse element include the extended Euclidean algorithm [3] and methods using Fermat's theorem such as the one proposed by Itoh and Tsujii [13]. A method for efficiently implementing division was proposed by Shantz [7].

5. REDUCTION

Field multiplication and squaring operations require reduction by an irreducible polynomial M. Rather than computing a full polynomial division, reduction can be done by executing a sequence of polynomial multiplications and additions based on the congruency

$$u \equiv u + vM \mod M \tag{3}$$

for arbitrary polynomials u and v over $GF(2)$. A special case of Equation (3), used for reduction, is

$$t^m \equiv M - t^m \mod M \tag{4}$$

Reduction of a product $c_0 = a * b$, $a, b \in GF(2^m)$, $\deg(a) < m$, $\deg(b) < m$ can be computed iteratively as follows. Since the degree of c_0 is less than $2m - 1$, c_0 can be split up into two polynomials $c_{0,h}$ and $c_{0,l}$ with $\deg(c_{0,h}) < m - 1$, $\deg(c_{0,l}) < m$ such that

$$c_0 = a * b = c_{0,h} * t^m + c_{0,l} \tag{5}$$

Using (4), the following congruency is obvious

$$\begin{aligned} c_1 &= c_{0,h} * (M - t^m) + c_{0,l} \\ &\equiv c_0 \mod M \end{aligned}$$

Given that $\deg(c_{0,h}) < m - 1$ and $\deg(M - t^m) < m$, it follows that $\deg(c_1) < 2m - 2$. By iteratively splitting up c_j into polynomials $c_{j,h}$ and $c_{j,l}$ such that

$$c_{j+1} = c_{j,h} * (M - t^m) + c_{j,l} \tag{6}$$

until

$$c_{j,h} = 0 \Leftrightarrow \deg(c_j) < m$$

the reduced result $c = c_i$ can be computed in a maximum of $i \leq m-1$ reduction iterations. The minimum number of required iterations depends on the second highest term of the irreducible polynomial M [18], [11]. For

$$M = t^m + t^k + \sum_{j=1}^{k-1} M_j t^j + 1, 1 \leq k < m \tag{7}$$

it follows that a better upper bound for $\deg(c_1)$ is $\deg(c_1) < m + k - 1$. Applying (6), $\deg(c_j)$ gradually decreases such that

$$\deg(c_{j+1,h}) = \begin{cases} \text{if } \deg(c_{j,h}) > m-k : \\ \quad \deg(c_{j,h}) + k - m \\ \text{if } \deg(c_{j,h}) \leq m-k : \\ \quad 0 \end{cases} \tag{8}$$

The minimum number of iterations i is given by

$$m - 1 - i(m-k) \leq 0 \Leftrightarrow i \geq \lceil \frac{m-1}{m-k} \rceil \tag{9}$$

To enable efficient implementations, M is often chosen to be either a trinomial M_t or pentanomial M_p:

$$\begin{aligned} M_t &= t^m + t^{k_3} + 1 \\ M_p &= t^m + t^{k_3} + t^{k_2} + t^{k_1} + 1 \\ m &> k_3 > k_2 > k_1 \geq 1 \end{aligned}$$

Choosing M such that $k_3 \leq \frac{m-1}{2}$ apparently limits the number of reduction iterations to two. This is the case for all irreducible polynomials recommended by NIST [19] and SECG [6]. As we will show in Section 7.1, the multiplications $c_{j,h} * (M - t^m)$ can be optimized if $(M - t^m)$ is a constant sparse polynomial, i.e. if M is fixed and contains only few terms.

6. PARTIAL REDUCTION

For all $c \in GF(2^m)$, $\deg(c) < m$ there exist congruent polynomials over $GF(2)$ of a degree greater or equal to m. The use of congruent polynomials in non-canonical form is the idea underlying partial reduction. Instead of reducing $c_0 = a * b$, $a, b \in GF(2^m)$, $\deg(a) < m$, $\deg(b) < m$ to a polynomial of degree less than m, c_0 is only partially reduced resulting in a congruent polynomial of degree less than a chosen integer n with $n \geq m$. For hardware implementations, n could, for example, be the maximum operand size of a multiplier. Addition and multiplication of congruent polynomials again result in a congruent polynomial. For arithmetic operations on polynomials $a, b \in GF(2^m)$, $\deg(a) < m$, $\deg(b) < m$ and congruent polynomials over $GF(2)$ a', b', i.e. $a' \equiv a \mod M$ and $b' \equiv b \mod M$, the following properties can be used:

$$a' + b' \equiv a + b \mod M \tag{10}$$

$$a' * b' \equiv a * b \mod M \tag{11}$$

Consequently, all computations for a point multiplication in $GF(2^m)$ can be executed on congruent polynomials, whose degree may be greater or equal to m. Reduction of the results to polynomials in canonical form only needs to be done in a last step of the point multiplication.

For a multiplication $c_0 = a * b$ with $a, b \in GF(2^m)$, $\deg(a) < m$, $\deg(b) < m$, c_0 can be partially reduced to $c \equiv c_0 \mod M$, $\deg(c) < n$ as follows: For an integer $n \geq m$, c_0 can be split up into two polynomials $c_{0,h}$ and $c_{0,l}$ with

$$\begin{aligned} &\deg(c_{0,l}) < n \\ &\deg(c_{0,h}) < 2m - n - 1, && \text{if } n < 2m - 1 \\ &c_{0,h} = 0, && \text{if } n \geq 2m - 1 \end{aligned}$$

such that similar to (5)

$$\begin{aligned} c_0 = a * b &= c_{0,h} * t^n + c_{0,l} \\ &= c_{0,h} * t^{n-m} * t^m + c_{0,l} \end{aligned} \tag{12}$$

Applying (4), a congruent polynomial c_1 can be computed with

$$c_1 = c_{0,h} * t^{n-m} * (M - t^m) + c_{0,l} \tag{13}$$

$$\equiv c_0 \mod M \tag{14}$$

Given that $\deg(c_{0,h}) < 2m - n - 1$ and $\deg((M - t^m) * t^{n-m}) < n$, it follows that $\deg(c_1) < 2m - 2$. c_j can again be iteratively split up into polynomials $c_{j,h}$ and $c_{j,l}$ such that

$$c_{j+1} = c_{j,h} * t^{n-m} * (M - t^m) + c_{j,l} \tag{15}$$

To compute a congruent polynomial of degree less than n, the termination condition is

$$c_{j,h} = 0 \Leftrightarrow \deg(c_j) < n \tag{16}$$

The result $c = c_i$ can be computed in at most $i \leq 2m - n - 1$ reduction steps.

Given M as defined in (7), $\deg(c_j)$ gradually decreases as stated by Equation (8). The minimum number of iterations i is given by

$$\begin{aligned} 2m - n - 1 - i(m-k) \leq 0 \\ \Leftrightarrow i \geq \lceil \frac{2m - n - 1}{m - k} \rceil \end{aligned} \tag{17}$$

For polynomials over $GF(2)$ a', b' with $\deg(a') < n$ and $\deg(b') < n$, $c_0 = a' * b'$ can be partially reduced to $c \equiv c_0 \mod M$, $\deg(c) < n$ as follows: c_0 can be split up into $c_{0,h}$ and $c_{0,l}$ with $\deg(c_{0,h}) < n - 1$ and $\deg(c_{0,l}) < n$. Formulas (12), (14), (15), (16) and (8) can be applied for arbitrary $n \geq m$. Therefore, the minimum number of reduction steps i is

$$n - 1 - i(m-k) \leq 0 \Leftrightarrow i \geq \lceil \frac{n-1}{m-k} \rceil \tag{18}$$

7. APPLICATIONS OF PARTIAL REDUCTION

NIST and SECG specified named curves over fields $GF(2^m)$ with m being a prime number. Examples are $m = 113$, 131, 163, 193, 233, 239, 283, 409 and 571. On computer systems, polynomials of these fields can be efficiently represented by bit strings. The size of the bit strings is preferably a power of 2, i.e. n bits with $n = 2^u \geq m$ for a positive integer u, or multiples of a power of 2, i.e. $n = v * w$ with $w = 2^u$ bits for positive integers u, v, w and $n \geq m$. For general purpose processor architectures, w corresponds to the word size and v to the number of words. For example, on a 32-bit processor a polynomial $a \in GF(2^{163})$ could be represented with $v = 6$ words each $w = 32$ bit wide, i.e. $n = 6 * 32 = 192$. Partial reduction allows for a single implementation that can handle curves over any $GF(2^m)$ with $m \leq n$.

7.1 ECC Hardware Accelerators

To efficiently support ECC in hardware, $GF(2^m)$ arithmetic needs to be implemented for large operands. Design choices depend on the number of supported elliptic curves and irreducible polynomials. For a single field $GF(2^m)$ with a given field degree m and a fixed irreducible polynomial M, the reduction steps of field multiplications and squarings can be optimized. Referring to Equation (6), the multiplications $c_{j,h} * (M - t^m)$ constitute multiplications with a constant $(M - t^m)$. In addition, choosing M as a trinomial or pentanomial reduces the cost of a full multiplication to two, respectively four additions. An example of a reduction iteration for a pentanomial $M_p = t^m + t^{k_3} + t^{k_2} + t^{k_1} + 1$ is shown in Figure 1. The simplified multiplication typically

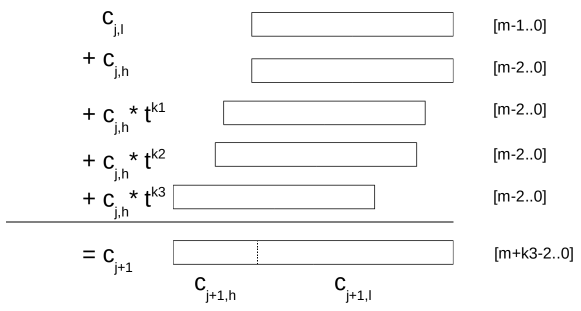

Figure 1: *Reduction for a Fixed Pentanomial M_p.*

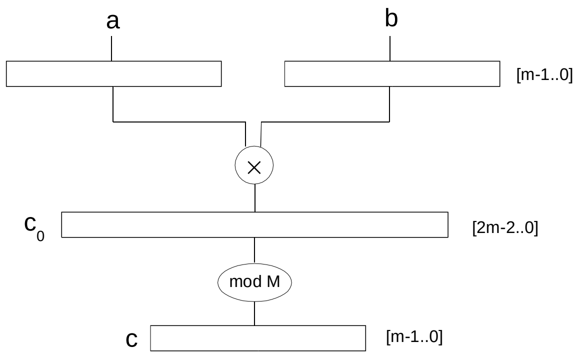

Figure 2: *Multiplier with Reduction Circuitry for a Fixed M.*

allows for implementing circuitry that can perform reduction in a single clock cycle as illustrated in Figure 2. After multiplying a and b, c_0 is reduced to a congruent polynomial $c \in GF(2^m)$, $\deg(c) < m$. Note that the register widths of a, b and c correspond to the field degree m.

In the case of squaring, both polynomial multiplication and reduction can typically be combined and executed in a single clock cycle. Since squaring only requires the insertion of zeros, no intermediate result c_0 needs to be computed making it possible to perform squaring and reduction in the same cycle.

For implementations of a small number of fields $GF(2^m)$ with given irreducible polynomials $\{M_1, M_2, \ldots, M_r\}$ it is a viable solution to add dedicated reduction logic for each irreducible polynomial. As shown in Figure 3, the register size n needs to be chosen according to the largest field de-

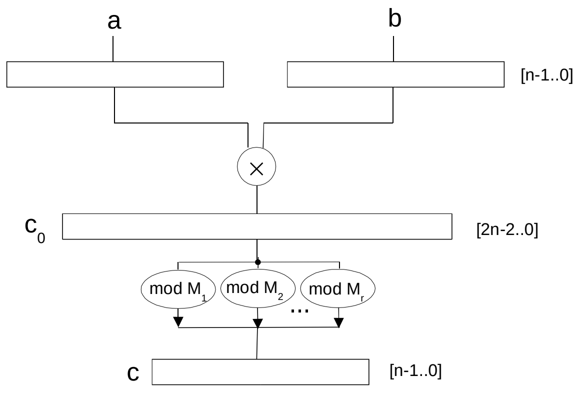

Figure 3: *Multiplier with Reduction Circuitry for a Set $\{M_1, M_2, \ldots, M_r\}$.*

gree m. Depending on the underlying field, the appropriate reduction logic can be selected by a multiplexer.

In the case of arbitrary curves, however, M is unknown, i.e. the multiplications $c_{j,h} * (M - t^m)$ cannot be optimized. In addition, for an $n \times n$-bit multiplier returning a $2n - 1$-bit result, data word c_0 may span both n-bit result registers depending on m as shown in Figure 4. Extracting $c_{0,h}$

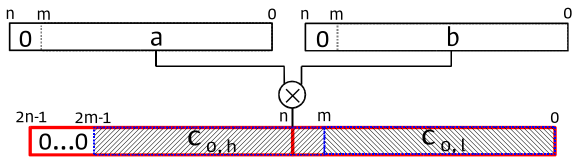

Figure 4: *Multiplication for Variable Field Degrees m.*

and subsequently $c_{j,h}$ to perform reduction requires complex multiplexer logic given that m may assume a range of values.

An alternative approach presented in [10] is shown in Figure 5. First, operand a is multiplied by the constant factor t^{n-m}, which is used to left-align operands to register boundaries. Second, the multiplication $c_0 = a * b$ is executed such that register r_h contains $c_{0,h}$. Reduction is performed as in Equation (6) until the condition $r_h = c_{j,h} = 0$ is met. Note that $(M - t^m) * t^{n-m}$ is a constant throughout the point multiplication and needs to be computed only once. Finally, the left-aligned reduction result in r_l is multiplied by t^m such that the reduced result $c \equiv c_0 \mod M$, $\deg(c) < m$ can be read from r_h. Note that the first and last multiplication can be omitted if the result is used as operand a in a subsequent multiplication.

Using partial reduction as described in Section 6 eliminates the two multiplications used for operand alignment. This is illustrated in Figure 6 for operand polynomials a', b', $\deg(a') < n$, $\deg(b') < n$ and an arbitrary irreducible polynomial $M, \deg(M) \le n$. Reduction of a partially reduced polynomial $c', \deg(c') < n$ to a congruent $c \equiv c' \mod M$, $\deg(c) < m$ can be performed with the algorithm in Figure 5 by setting $a = c'$ and omitting the second step.

To better support partial reduction, dedicated multiplier circuitry can be used. Figure 7 shows an $n \times n$-bit multiplier with data paths customized for partial reduction. Initially,

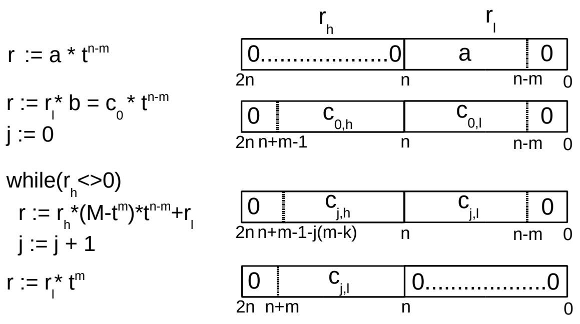

Figure 5: *Implementation of Reduction.*

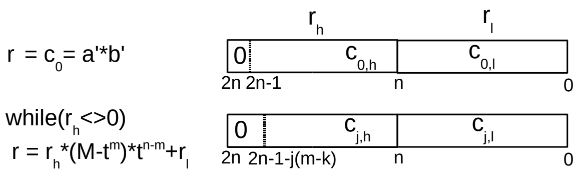

Figure 6: *Implementation of Partial Reduction.*

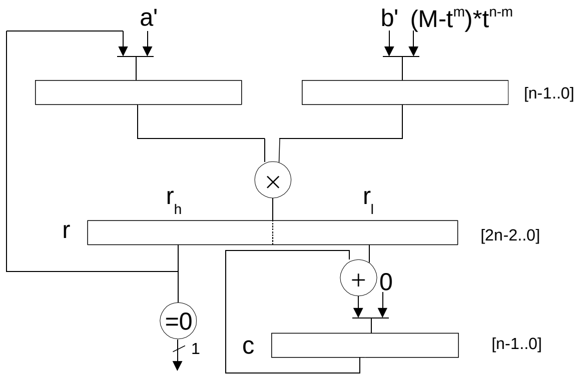

Figure 7: *Multiplier supporting Partial Reduction.*

the operand registers are loaded with n-bit operands a' and b'. The operands are multiplied using the logic denoted by $\bigotimes$. Depending on the design constraints, the multiplier logic can either implement serial, digit-serial or parallel polynomial multiplication. The result of the multiplication $c_0 = a' * b'$ is stored in register r, which has a width of $2n-1$ bits and is split into high word r_h and low word r_l. Note that $c_{j,h}$ and $c_{j,l}$ are aligned to the register boundaries of r_h and r_l as in Figure 6. A reduction iteration as specified by Equation (15) can be performed by loading the operand registers with $c_{j,h}$ and $(M - t^m) * t^{n-m}$. The sum of low words $c_{j,l}$ is accumulated in result register c. c contains the reduced result one cycle after r_h becomes 0.

7.2 ECC Software

Partial reduction can also be employed in the implementation of a compact and complete ECC software library. Besides high performance, a design goal for a software library may be to support arbitrary curves that are not known at implementation time. The following approach might be taken: In addition to hardcoded implementations for known curves, a generic point multiplication routine using partial reduction is provided; calls to the library can be dispatched according to whether an accelerated implementation exists or not. Furthermore, partial reduction can be useful in verifying implementations optimized for known curves.

On today's general purpose processors, polynomial multiplication is commonly implemented through a sequence of shift and XOR instructions. However, future processors may have dedicated binary polynomial multiplication instructions.

Combined with such instructions, partial reduction allows for operating on word-sized operands without having to extract bit fields. For example, to implement point multiplication over $GF(2^{163})$ on a 32-bit processor it may be more efficient to operate on $n = 6 * 32 = 192$ bits aligned to 32-bit word boundaries than to extract bits from non-aligned $m = 163$-bit bit strings. By applying partial reduction, all interim computations would include partial reduction to 192 bits. Only in the last step of a point multiplication, the operands would be reduced to 163 bits.

Further advantages of implementations using partial reduction include a small memory footprint and code that can be easily verified.

8. IMPLEMENTATION

We developed a programmable processor optimized for ECC point multiplications. The design was driven by the need to both provide high performance for named elliptic curves and support point multiplications for arbitrary, less frequently used curves. The architecture targets binary polynomial fields $GF(2^m)$, $m \leq 255$. We implemented hardwired reduction circuitry to accelerate curves over three fields recommended by SECG, of which two are also recommended by NIST. The curves are specified over $GF(2^{163})$, $GF(2^{193})$ and $GF(2^{233})$ and use the irreducible polynomials $M_{163} = t^{163} + t^7 + t^6 + t^3 + 1, M_{193} = t^{193} + t^{15} + 1$ and $M_{233} = t^{233} + t^{74} + 1$, respectively. Point multiplication for all other curves can be performed using partial reduction.

The data path of the processor shown in Figure 8 implements a 256-bit architecture. Parameters and variables are stored in an 8kB data memory DMEM and program instructions are contained in a 1kB instruction memory IMEM.

Both memories are dual-ported and accessible by the host machine through a 64-bit/66MHz PCI interface. The register file contains eight general purpose registers R0-R7, a register RM to hold the irreducible polynomial and a register RC for curve-specific configuration information. The arithmetic units implement division (DIV), multiplication (MUL) and squaring/addition/shift left (ALU). Source operands are transferred over the source bus SBUS and results are written back into the register file over the destination bus DBUS. Program execution is orchestrated by the Control Unit, which fetches instructions from the IMEM and controls the DMEM, the register file and the arithmetic units.

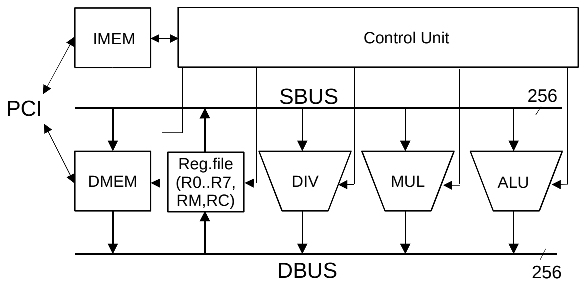

Figure 8: *Data Path and Control Unit.*

Corresponding to the arithmetic units we defined arithmetic and logic instructions MUL/MULNR, DIV, ADD, SQR and SL. While MUL multiplies two polynomials of degree less than m and returns a reduced result of degree less than m, MULNR executes a polynomial multiplication on two polynomials of order up to the register width n and returns a $2n$-bit result. Instructions ADD and MULNR can be used to implement reduction as shown in Figures 5 and 6. Computing a multiplication or squaring operation including reduction by an arbitrary irreducible polynomial requires 3+i MULNR and i ADD instructions for full reduction (with i as in Equation (9)) and 1+i MULNR and i ADD instruction (with i as in Equation (18)) when partial reduction is applied.

A more detailed block diagram of the multiplier is shown in Figure 9. We studied and implemented several different architectures and, finally, settled on a digit-serial shift-and-add multiplier. The result is computed in two steps. First, the product of the polynomial multiplication is computed by iteratively multiplying a digit of operand X with Y, and accumulating the partial products in Z. In the case of MUL, the product Z is reduced by one of the irreducible polynomials M_{163}, M_{193} or M_{233}. In our implementation, the input operands X and Y can have a size of up to $n = 256$ bits, and the reduced result $P = X * Y \mod M_m$ has a size of $m = 163, 193, 233$ bits according to the specified named curve. The digit size d is 64. We optimized the number of iterations needed to compute the product Z such that the four iterations it takes to perform a full 256-bit multiplication are only executed for $m = 193, 233$ whereas three iterations are executed for $m = 163$. To compensate for the missing shift operation in the latter case, a multiplexer was added to select the bits of Z to be reduced. The hardwired reduction takes another clock cycle.

We can rely on the hardwired reducers only for the named curves. All other curves need to be handled by partial reduction. We, therefore, need a multiplier architecture that

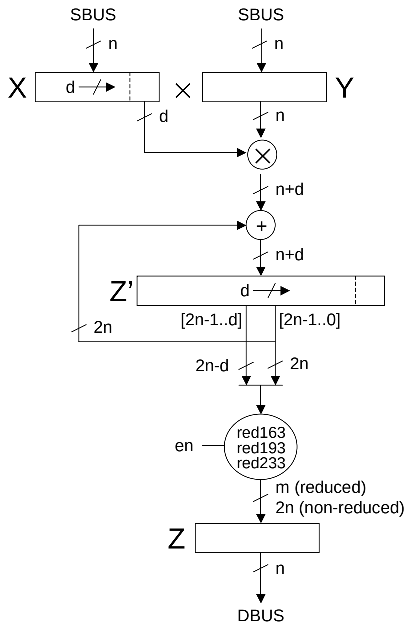

Figure 9: *Shift-and-Add Multiplier.*

either provides a way to reduce by an arbitrary irreducible polynomial as shown in Figure 7 or offers the option to calculate a non-reduced product. We opted for the latter option and added a path to bypass the reducer; i.e. the product of the polynomial multiplication $Z = X * Y$ can be written back into two result registers.

We specified our design in Verilog and prototyped it in a Xilinx Virtex XCV2000E-FG680-7 FPGA using the design tools Synplify 7.0.2 and Xilinx Design Manager 3.3.08i. Area constraints were given for the ALU, the divider and the register file, but no manual placement had to be done. The prototype runs off the PCI clock at a frequency of 66.4 MHz. An image of the floorplan is shown in Figure 10.

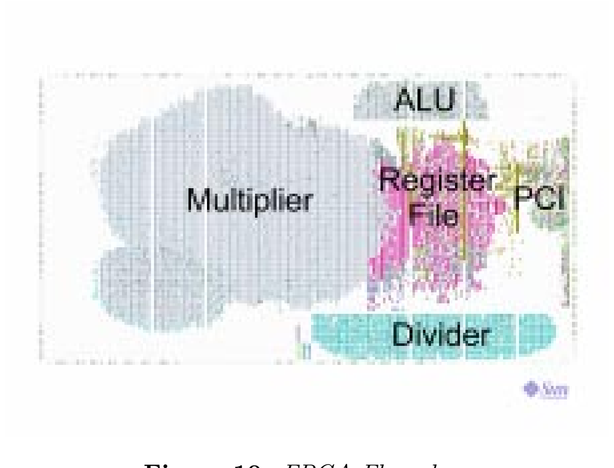

Figure 10: *FPGA Floorplan.*

9. PERFORMANCE

Table 1 summarizes performance numbers for hardware and software implementations of Montgomery's point multiplication algorithm over three fields $GF(2^{163})$, $GF(2^{193})$ and $GF(2^{233})$. The hardware numbers for named curves were obtained by using multiplication and squaring operations with hardwired reduction circuitry. The numbers for generic curves were measured for the same curves with multiplication and squaring operations implemented with full reduction as shown in Figure 5 and with partial reduction as shown in Figure 6, respectively. For comparison, we implemented and benchmarked software libraries in C for generic curves. The first implementation uses full reduction and assumes the irreducible polynomials to be either trinomials or pentanomials. It uses a variable number of words to represent different field degrees and was compiled for 64-bit processors. The second implementation uses partial reduction and allows an arbitrary number of terms in the irreducible polynomials. It uses a fixed number of eight 32-bit words and was compiled for 32-bit processors. Execution times were measured on a 900MHz Sun Fire™280R server.

	Hardware		**Software**	
	ops/s	**ms/op**	**ops/s**	**ms/op**
Named Curves				
$GF(2^{163})$	6987	0.14		
$GF(2^{193})$	5359	0.19		
$GF(2^{233})$	4438	0.23		
Generic Curves (full)				
$GF(2^{163})$	644	1.55	322	3.11
$GF(2^{193})$	544	1.84	294	3.40
$GF(2^{233})$	451	2.22	223	4.48
Generic Curves (partial)				
$GF(2^{163})$	1075	0.93	50	20.11
$GF(2^{193})$	911	1.10	42	23.83
$GF(2^{233})$	757	1.32	35	28.87

Table 1: *Hardware and Software Performance.*

Although hardwired reduction in hardware offers approximately a 6-fold speedup over partial reduction, partial reduction is about 1.7 times faster than an implementation using full reduction. Compared to full reduction, partial reduction not only reduces the number of multiplications needed for a reduction iteration, but also allows for a better usage of the registers. For our processor, this results in more efficient code with fewer memory operations.

As for the software numbers, the implementation using full reduction outperforms partial reduction. This can be explained by the high cost of multiplications of large operands, which we used to implement partial reduction. However, partial reduction allows for compact implementations such that a complete implementation of point multiplication for arbitrary field degrees up to $m = 255$ fits onto one letter-sized sheet of paper.

The execution time of a point multiplication kP depends on the execution times of the arithmetic operations in $GF(2^m)$ and the size of the integer k in bits, which is in the order of the field degree m. In our hardware implementation, addition and squaring with hardwired reduction can be performed in one clock cycle independent of m. A multiplication requires three cycles for $m \leq 192$ and four cycles for $m > 192$ with an additional cycle for hardwired reduction. A division can be computed in $2m$ cycles. Since Montgomery point multiplication using projective coordinates requires only one division, the execution time of a point multiplication on named curves is dominated by the number of multiplications and grows approximately linearly to number of bits in k in the intervals $1 \leq m \leq 192$ and $193 \leq m \leq 256$.

Using partial reduction, both squaring and multiplication require $i + 1$ polynomial multiplications and i additions as shown in Section 6. Both multiplications and squarings account for the bulk of the execution time of a point multiplication. The execution time of a point multiplication for generic curves increases with both the growth of i and the number of bits in k.

Named Curve	**Irreducible Polynomial**	**i for n= m**	**256**
sect113r1/r2	$x^{113} + x^9 + 1$	2	3
sect131r1/r2	$x^{131} + x^8 + x^3 + x^2 + 1$	2	3
sect163k1/r1/r2	$x^{163} + x^7 + x^6 + x^3 + 1$	2	2
sect193r1/r2	$x^{193} + x^{15} + 1$	2	2
sect233k1/r1	$x^{233} + x^{74} + 1$	2	2
sect239k1	$x^{239} + x^{36} + 1$	2	2

Table 2: *Reduction Iterations for SECG-Recommended Curves.*

The number of reduction iterations i for a particular field $GF(2^m)$ depends on the field degree m, the irreducible polynomial M and the register size n. Table 2 shows values of i for curves recommended by SECG in [6] for a register size equal to the field degree and a fixed register size of $n = 256$ bit. Among the named curves, only the two smallest require three reduction iterations, whereas all others can be computed with two iterations. Referring to Equation (18), the performance of partial reduction can be optimized by choosing a register size n close to the maximal field degree m and irreducible polymials with a large distance between the two highest terms, i.e. a large $m - k$.

10. CONCLUSIONS

Partial reduction allows for computing ECC point multiplication on arbitrary elliptic curves over binary polynomial fields up to a maximum degree n where n is typically defined by the maximum operand size that the implementation can handle. Partial reduction is applicable to software as well as hardware implementations. It can serve as the basis of a generic ECC software library or a reference implementation used to verify optimized implementations. In addition, implementations containing optimized support for specific curves may choose to use partial reduction as a backup solution for realizing other non-optimized curves and curves not known at implementation time.

We presented an ECC hardware accelerator that provides optimized performance for a limited number of named curves and support for generic curves over arbitrary fields $GF(2^m)$, $m \leq 255$. While many reported implementations make use of reconfigurable logic to address different curves, our approach is well suited for implementations in ASIC technology. Such implementations are needed to make ECC hardware acceleration commercially successful as they offer lower cost at high volumes, lower power consumption, higher clock frequencies and an easier way to provide tamper resistance.

An analysis of the execution times of point multiplication algorithms for named and generic curves shows that in the latter case multiplication and squaring operations are much more costly due to reduction. It is particularly noteworthy that squaring operations become as expensive as multiplication operations.

As for future work, we want to look into efficiently integrating partial reduction into the multiplier circuit of our accelerator architecture.

Acknowledgments

We would like to acknowledge Edouard Goupy and Daniel Finchelstein for implementing point multiplication on the hardware accelerator using different reduction schemes. We would also like to thank André Weimerskirch for reviewing our paper and for his suggestions regarding the formal representation of our ideas.

11. REFERENCES

[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $f_{2^{155}}$. *In IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[2] M. Bednara, M. Daldrup, J. von zur Gathen, and J. Shokrollahi. Reconfigurable implementation of elliptic curve crypto algorithms. *Reconfigurable Architectures Workshop, 16th International Parallel and Distributed Processing Symposium*, April 2002.

[3] Berlekamp. *Algebraic Coding Theory.* Aegan Park Press, 1984.

[4] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999. London Mathematical Society Lecture Note Series 265.

[5] S. Blake-Wilson, D. Brown, Y. Poeluev, and M. Salter. *Additional ECC Groups For IKE.* IETF Internet Draft, July 2002.

[6] Certicom Research. Sec 2: Recommended elliptic curve domain parameters. Standards for Efficient Cryptography Version 1.0, September 2000.

[7] S. Chang-Shantz. From euclid's gcd to montgomery multiplication to the great divide. Technical report, Sun Microsystems Laboratories TR-2001-95, June 2001.

[8] J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[9] V. Gupta, S. Blake-Wilson, B. Möller, and C. Hawk. *ECC Cipher Suites for TLS.* IETF Internet Draft, August 2002.

[10] N. Gura, H. Eberle, and S. C. Shantz. An end-to-end systems approach to elliptic curve cryptography. In *CHES '2002 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science. Springer-Verlag, August 2002.

[11] A. Halbutoğulları and Ç. K. Koç. Mastrovito multiplier for general irreducible polynomials. *IEEE Transactions on Computers*, 49(5):503–518, May 2000.

[12] D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1965. Springer-Verlag, August 2000.

[13] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $gf(2^m)$ using normal bases. *Information and Computation*, (78):171–177, 1988.

[14] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk*, (145):293–294, 1963. Translation in Physics-Doklady 7, 595-596.

[15] J. López and R. Dahab. Fast multiplication on elliptic curves over gf(2m) without precomputation. In *CHES '99 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1717. Springer-Verlag, August 1999.

[16] G. Orlando and C. Paar. A high-performance reconfigurable elliptic curve processor for $gf(2^m)$. In *CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1965. Springer-Verlag, August 2000.

[17] R. Schroeppel, H. Orman, and S. O'Malley. Fast key exchange with elliptic curve systems. In *Advances in Cryptography, Crypto '95*, Lecture Notes in Computer Science 963. Springer-Verlag, 1995.

[18] L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *IEEE Journal of VLSI Signal Processing Systems*, (19):149–166, 1998.

[19] U.S. Department of Commerce and National Institute of Standards and Technology. Digital signature standard (dss). Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[20] A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *The Fourth Smart Card Research and Advanced Applications (CARDIS2000) Conference*, September 2000. Bristol, UK.

[21] T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings: Computers and Digital Technique*, 149(2), March 2002.