Incomplete Reduction in Modular Arithmetic [*][†][‡]

T. Yanık, E. Savaş, and Ç. K. Koç
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Abstract

We describe a novel method for obtaining fast software implementations of the arithmetic operations in the finite field $GF(p)$ with an arbitrary prime modulus p which is of arbitrary length. The most important feature of the method is that it avoids bit-level operations which are slow on microprocessors and performs word-level operations which are significantly faster. The proposed method has applications in public-key cryptographic algorithms defined over the finite field $GF(p)$, most notably the elliptic curve digital signature algorithm.

1 Introduction

The basic arithmetic operations (i.e., addition, subtraction, and multiplication) in the finite field $GF(p)$ have several applications in cryptography, such as decipherment operation of the RSA algorithm [11], the Diffie-Hellman key exchange algorithm [2], elliptic curve cryptography [5, 8], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm (ECDSA) [10]. These applications demand high-speed software implementations of the arithmetic operations in $GF(p)$ for $160 \leq \lceil \log_2(p) \rceil \leq 2048$. In this paper, we describe a new method for obtaining high-speed software implementations of the arithmetic operations on microprocessors and general-purpose computers. The most important feature of this method is that it avoids bit-level operations which are slow on modern microprocessors. The algorithms proposed in this paper perform word-level operations, trading them off for bit-level operations, and thus, resulting in much higher speeds. We provide the timing results of our implementations on a Pentium II computer, supporting our speedup claims.

2 Representation of the Numbers

The arithmetic of $GF(p)$ is also called modular arithmetic where the modulus is p. The elements of the field are the set of integers $\{0, 1, \ldots, (p-1)\}$, and the arithmetic functions (addition, subtraction, and multiplication) takes two input operands from this set and produces the output which is also in this set. We are assuming that the modulus p is a k-bit integer where $k \in [160, 2048]$. A number in this range is represented as an array of words, where each word is of length w. Most software

implementations require that $w = 32$, however, w can be selected as 8 or 16 on 8-bit or 16-bit microprocessors.

In order to create a scalable implementation, we are not placing any restrictions on the prime p or its length k. The prime number does not need to be in any special form, as some methods require, for example, the method in [1] requires that $p = 2^k - c$. Furthermore, the length of the prime p does not need to be an integer multiple of the wordsize of the computer.

We have the following definitions:

- k : The exact number of bits required to represent the prime modulus p, i.e., $k = \lceil \log_2 p \rceil$.

- w : The word-size of the representation (i.e., the computer). Usually, $w = 8, 16, 32$.

- s: The exact number of words required to represent the prime modulus p, i.e., $s = \lceil \frac{k}{w} \rceil$.

- m : The total number of bits in s words, i.e., $m = sw$.

Since our computers are capable of performing only two's complement binary arithmetic, we represent the numbers as unsigned binary numbers. A number from the field $GF(p)$ is represented as an s-word array of unsigned binary integers. We will use the notation $A = (A_{s-1} A_{s-2} \ldots A_1 A_0)$, where the words A_i for $i = 0, 1, \ldots, (s-1)$ are unsigned binary numbers of length w. The most significant word (MSW) of A is A_{s-1}, while the least significant word (LSW) of A is A_0. The bit-level representation of A is given as $A = (a_{k-1} a_{k-2} \ldots a_1 a_0)$. Similarly, the most significant bit (MSB) of A is a_{k-1}, while the least significant bit (LSB) of A is a_0. We will use exactly s words to represent a number. If k is not an integer multiple of w, then we have $k = (s-1)w + u$ where $u < w$ is a positive integer, and thus, only the least significant u bits of the MSW of A_{s-1} are occupied, and the most significant $(w - u)$ bits are all zero.

A_{s-1}	A_{s-2}	$\cdots$	A_1	A_0
$\underbrace{0 \cdots 0}_{w-u} a_{(s-1)w+u-1} \cdots a_{(s-1)w}$	$a_{(s-1)w-1} \cdots a_{(s-2)w}$	$\cdots$	$a_{2w-1} \cdots a_w$	$a_{w-1} \cdots a_0$

2.1   Incompletely Reduced Numbers

This representation methodology has a shortcoming: it requires bit-level operations on the MSW in order to perform arithmetic, which affects the speed of the operations in software implementations. In order to overcome this shortcoming, we introduce the concept of *incomplete modular arithmetic*. In order to explain the mechanics of the method, we make the following definitions:

- Completely Reduced Numbers: the numbers from 0 to $(p-1)$.

$$\mathbf{C} = \{0, 1, \ldots, (p-1)\} .$$

- Incompletely Reduced Numbers: the numbers from 0 to $(2^m - 1)$.

$$\mathbf{I} = \{0, 1, \ldots, p-1, p, p+1, \ldots, (2^m - 1)\} .$$

- Unreduced Numbers: the numbers from p to $(2^m - 1)$.

$$\mathbf{U} = \{p, p+1, \ldots, (2^m - 1)\} .$$

2

Note that we have the following relationship between these sets:

$$\mathbf{C} \subset \mathbf{I}$$
$$\mathbf{U} \subset \mathbf{I}$$
$$\mathbf{U} = \mathbf{I} - \mathbf{C}$$

If $A \in \mathbf{C}$ and $2p < 2^m$, then A has an incompletely reduced equivalent $B \in \mathbf{I}$ such that $A = B$ (mod p). Thus, instead of working with A, we can also work with B in our arithmetic operations. The incompletely reduced numbers completely occupy s words as follows:

B_{s-1}	B_{s-2}	$\cdots$	B_1	B_0
$b_{sw-1}\cdots b_{(s-1)w}$	$b_{(s-1)w-1}\cdots b_{(s-2)w}$	$\cdots$	$b_{2w-1}\cdots b_w$	$b_{w-1}\cdots b_0$

When we perform arithmetic with incompletely reduced numbers, we do not need to perform bit-level operations on the MSW. All operations are word-level operations, and checks for carry bits are performed on the word boundaries, not within the words. Furthermore, we skip unnecessary reductions until the actual output is produced, in which case, we make sure that it belongs to $\mathbf{C}$. This representation yields high-speed implementations of the arithmetic operations.

An incompletely reduced number B can be converted to its completely reduced equivalent A by subtracting integer multiples of p from B as many times as necessary (until it is less than p).

2.2 Positive and Negative Numbers

The numbers in the range $[0, p-1]$ as defined are positive numbers. The implementation of the subtraction operation requires a method of representation for negative numbers as well. We use the *least positive residues* which allow simultaneous representation of the positive and negative numbers modulo p. In this representation, the numbers are always left as positive, i.e., if the result of a subtraction operation is a negative number, then it is converted back to positive by adding p to it. For example, for $p = 7$, the operation $s = 3 - 4$ is performed as $s = 3 - 4 + 7 = 6$. The numbers from 0 to $(p-1)/2$ can be interpreted as positive numbers modulo p, while the numbers from $(p-1)/2 + 1$ to $p - 1$ can be interpreted as negative numbers modulo p. However, numbers are always kept as unsigned positive integers.

2.3 A Representation Example

We take the prime modulus as $p = 11 = (1011)$ and the word size as $w = 3$. Thus, we have $k = 4$ and $s = \lceil k/w \rceil = \lceil 4/3 \rceil = 2$, which gives $m = 2 \cdot 3 = 6$. The completely reduced set of numbers is $\mathbf{C} = \{0, 1, \ldots, 9, 10\}$, while the incompletely reduced set contain the numbers ranging from 0 to $(2^m - 1) = (2^6 - 1) = 63$ as $\mathbf{I} = \{0, 1, \ldots, 62, 63\}$. The incompletely reduced numbers occupy 2 words as $A = (A_1 A_0) = (a_5 a_4 a_3 \ a_2 a_1 a_0)$. For example, the decimal number 44 is represented as (101 100) in binary or (5 4) in octal.

An incompletely reduced equivalent of A is given as $B = A + i \cdot p$, where $B \in [0, 63]$ and i is a positive integer. For example, if $A = 5$, then the incompletely reduced equivalents of A are given as $\{5, 16, 27, 38, 49, 60\}$. The incompletely reduced representation is a redundant representation: we use the notation $\bar{5} = \{5, 16, 27, 38, 49, 60\}$ to represent the residue class $\bar{5}$. We will show in the following sections that this redundant representation provides more efficient arithmetic in $GF(p)$.

3   Modular Addition

Since we use the incompletely reduced numbers, our numbers are allowed to grow as large as $2^m - 1$. The incompletely reduced representation avoids unnecessary reduction operations. We add the input operands as $X := A + B \pmod{p}$. If the result does not exceed $2^m - 1$, we do not perform reduction. This check is simple to perform: since $m = sw$, we are only checking to see if there is a carry-out from the MSW. We will use the notation

$$(c, S_i) := A_i + B_i + c \tag{1}$$

to denote the word-level addition operation which adds the 1-word numbers A_i and B_i and the 1-bit carry-in c, producing the outputs c and S_i such that c is the 1-bit carry-out and S_i is the 1-word sum. The addition algorithm is given below:

<div align="center">

Modular Addition Algorithm

</div>

Inputs:	$A = (A_{s-1} \cdots A_1 A_0)$ and $B = (B_{s-1} \cdots B_1 B_0)$
Auxilary:	$F = (F_{s-1} \cdots F_1 F_0)$
Output:	$X = (X_{s-1} \cdots X_1 X_0)$
Step 1:	$c := 0$
Step 2:	for $i = 0$ to $s - 1$
Step 3:	$\quad (c, S_i) := A_i + B_i + c$
Step 4:	if $c = 0$ then return $X = (S_{s-1} \cdots S_1 S_0)$
Step 5:	$c := 0$
Step 6:	for $i = 0$ to $s - 1$
Step 7:	$\quad (c, T_i) := S_i + F_i + c$
Step 8:	if $c = 0$ then return $X = (T_{s-1} \cdots T_1 T_0)$
Step 9:	$c := 0$
Step 10:	for $i = 0$ to $s - 1$
Step 11:	$\quad (c, U_i) := T_i + F_i + c$
Step 12:	return $X = (U_{s-1} \cdots U_1 U_0)$

If the carry-out from the MSW is zero, the algorithm produces the correct result in Step 4 as $X = S = (S_{s-1} \cdots S_1 S_0)$. If the carry-out is one, we first ignore the carry-out and then correct the result. By ignoring the carry-out, we are essentially performing the operation $S := S - 2^m$. Since we need to perform modulo p arithmetic, we are allowed only add or subtract integer multiples of p, therefore, we need to correct the result as $T := (S - 2^m) + F$, where $F = (F_{s-1} \cdots F_1 F_0)$ is called *the correction factor for addition* and is defined as

$$F = 2^m - Ip , \tag{2}$$

where I is largest possible integer which brings F to the range $[1, p-1]$, in other words, $I = \lfloor 2^m/p \rfloor$. The number F is precomputed and saved. By performing the operation $T := (S - 2^m) + F$, we essentially perform a modulo p reduction as

$$T := (S - 2^m) + F = S - 2^m + 2^m - Ip = S - Ip . \tag{3}$$

Thus, the result $X = T$ will be correct as a modular number after Step 8. However, there is a danger in performing the operation $T := S + F$ since this might also cause a carry-out from the MSW. The input operands A and B are arbitrary numbers, they can be as large as $2^m - 1$, which

gives $S = 2^{m+1} - 2$. By ignoring the carry in Step 3, we obtain $S = 2^m - 2$. Therefore, $T := S + F$ in Step 4 may exceed 2^m, and we need to correct the result one more time, which is accomplished in Steps 9–11. This is the last correction we need to perform. There is no need for another correction after Step 11, since the maximum value of U is strictly less than 2^m.

$$U = (T - 2^m) + F = 2^m - 2 - 2^m + F = -2 + F \le -2 + p - 1 < 2^m \ . \tag{4}$$

3.1 Addition Examples

Let $p = 11$, $k = 4$, $w = 3$, $m = 6$, and $s = 2$. We also compute F as

$$F = 2^m - \lfloor 2^m/p \rfloor \cdot p = 64 - \lfloor 2^6/11 \rfloor \cdot 11 = 64 - 5 \cdot 11 = 9 \ . \tag{5}$$

- We illustrate the addition of $\overline{4} = \{4, 15, 26, 37, 48, 59\}$ and $\overline{5} = \{5, 16, 27, 38, 49, 60\}$ and using the incompletely reduced numbers 26 and 27.

$$
\begin{aligned}
S &= 26 + 27 \\
&= 53 \qquad (c = 0 \ \text{ return Step 4})
\end{aligned}
$$

  The result is indeed correct since 53 is equivalent to $\overline{9} = \{9, 20, 31, 42, 53\}$.

- We give an addition example where the first correction (Steps 5–8) would be required. The addition of $\overline{4} = \{4, 15, 26, 37, 48, 59\}$ and $\overline{5} = \{5, 16, 27, 38, 49, 60\}$ and using the incompletely reduced numbers 37 and 49 is such an example.

$$
\begin{aligned}
S &= 37 + 49 \\
&= 86 \qquad (c = 1 \ \text{ Step 4}) \\
&= 86 - 64 \quad (\text{ignore carry Step 4}) \\
&= 22 \\
T &= 22 + 9 \quad (\text{correction Steps 5–7}) \\
&= 31 \qquad (c = 0 \ \text{ return Step 8})
\end{aligned}
$$

  The result is indeed correct since 31 is equivalent to $\overline{9} = \{9, 20, 31, 42, 53\}$.

- We give an addition example where the second correction (Steps 10 and 11) would also be required. The addition of $\overline{6} = \{6, 17, 28, 39, 50, 61\}$ and $\overline{7} = \{7, 18, 29, 40, 51, 62\}$ using the incompletely reduced numbers 61 and 62 is such an example.

$$
\begin{aligned}
S &= 61 + 62 \\
&= 123 \qquad (c = 1 \ \text{ Step 4}) \\
&= 123 - 64 \quad (\text{ignore carry Step 4}) \\
&= 59 \\
T &= 59 + 9 \quad (\text{correction Steps 5–7}) \\
&= 68 \qquad (c = 1 \ \text{ Step 8}) \\
&= 68 - 64 \quad (\text{ignore carry Step 8}) \\
&= 4 \\
U &= 4 + 9 \quad (\text{correction Steps 9–11}) \\
&= 13 \qquad (\text{return Step 12})
\end{aligned}
$$

  The result is indeed correct since 13 is equivalent to $\overline{2} = \{2, 13, 24, 35, 46, 57\}$.

4 Modular Subtraction

The subtraction is performed using two's complement arithmetic. The input operands are least positive residues represented using incompletely reduced representation. We will use the notation

$$(b, S_i) := A_i - B_i - b \tag{6}$$

to denote the word-level subtraction operation which subtracts the 1-word number B_i and the 1-bit borrow-in b from the 1-word number A_i, producing the outputs which are the 1-word number S_i and the 1-bit borrow-out b. The field subtraction algorithm for computing $X = A - B \pmod{p}$ is given below:

Modular Subtraction Algorithm

Inputs:	$A = (A_{s-1} \cdots A_1 A_0)$ and $B = (B_{s-1} \cdots B_1 B_0)$
Auxilary:	$G = (G_{s-1} \cdots G_1 G_0)$ and $F = (F_{s-1} \cdots F_1 F_0)$
Output:	$X = (X_{s-1} \cdots X_1 X_0)$
Step 1:	$b := 0$
Step 2:	for $i = 0$ to $s - 1$
Step 3:	$\quad (b, S_i) := A_i - B_i - b$
Step 4:	if $b = 0$ then return $X = (S_{s-1} \cdots S_1 S_0)$
Step 5:	$c := 0$
Step 6:	for $i = 0$ to $s - 1$
Step 7:	$\quad (c, T_i) := S_i + G_i + c$
Step 8:	if $c = 0$ then return $X = (T_{s-1} \cdots T_1 T_0)$
Step 9:	$c := 0$
Step 10:	for $i = 0$ to $s - 1$
Step 11:	$\quad (c, U_i) := T_i + F_i + c$
Step 12:	return $X = (U_{s-1} \cdots U_1 U_0)$

If $b = 0$ after Step 4, then the result is positive, and it is a properly reduced modular number. If $b = 1$, then the result is negative, we obtain the two's complement result, i.e., we essentially compute

$$S := A - B = A + 2^m - B . \tag{7}$$

The result S is in the range $[0, 2^m - 1]$, as required. However, it is incorrectly reduced, i.e., 2^m is added, and thus, we need to correct the result by adding $G = (G_{s-1} \cdots G_1 G_0)$ which is the *correction factor for subtraction* defined as

$$G = Jp - 2^m , \tag{8}$$

where J is the smallest integer which brings G to the range $[1, p-1]$, i.e., $J = \lceil 2^m/p \rceil$. It is easily proven that the sum of the correction factors for addition and subtraction, i.e., the sum of F and G, is equal to p since

$$F + G = 2^m - Ip + Jp - 2^m = (J - I)p = (\lceil 2^m/p \rceil - \lfloor 2^m/p \rfloor)p = p , \tag{9}$$

in other words, we have $G = p - F$ or $F = p - G$. The result S is corrected to obtain T in Steps of 5–8. After the correction of S in Step 8, we obtain

$$T = S + G = A + 2^m - B + Jp - 2^m = A - B + Jp . \tag{10}$$

Similar to the addition algorithm in Step 8, this correction might cause a carry from the MSW, requiring another correction which we need to take care of using F. This is accomplished in Steps 9–11. There is no need for another correction after Step 12, since the maxium value $S = (2^m - 1)$ gives

$$U \leq (2^m - 1) + G - 2^m + F = -1 + p < 2^m . \tag{11}$$

4.1 Subtraction Examples

Let $p = 11$, $k = 4$, $w = 3$, $m = 6$, and $s = 2$. We also compute G as

$$G = \lceil 2^m/p \rceil \cdot p - 2^m = \lceil 2^6/11 \rceil \cdot 11 - 64 = 6 \cdot 11 - 64 = 2 . \tag{12}$$

Since $F + G = p$, we could have also obtained G using $G = p - F = 11 - 9 = 2$.

- We illustrate the subtraction operation $S := 5 - 7$, where $\overline{5} = \{5, 16, 27, 38, 49, 60\}$ and $\overline{7} = \{7, 18, 29, 40, 51, 62\}$, using the incompletely reduced equivalents 49 and 40.

$$
\begin{aligned}
S \;&=\; 49 - 29 \\
&=\; 20 \qquad (b = 0 \text{ return Step 4})
\end{aligned}
$$

  The result is indeed correct since 20 is a incompletely reduced number represents the reduced number $\overline{9} = \{9, 20, 31, 42, 53\}$ which is equal to $5 - 7 = -2 = 9 \pmod{11}$.

- On the other hand, the same subtraction $S := 5 - 7$ operation using the incompletely reduced equivalents 16 and 40 is performed as

$$
\begin{aligned}
S \;&=\; 16 - 40 \\
&=\; -24 \qquad (b = 1 \text{ Step 4}) \\
&=\; 64 - 24 \quad (\text{two's complement Step 4}) \\
&=\; 40 \\
T \;&=\; 40 + 2 \qquad (\text{correction Steps 5–8}) \\
&=\; 42 \qquad (c = 0 \text{ eturn Step 8})
\end{aligned}
$$

  The incompletely reduced number 42 is also the correct result since it represents $\overline{9} = \{9, 20, 31, 42, 53\}$.

- We now give an addition example where the second correction (Step 9–12) would be required. The subtraction operation $5 - 6$, where $\overline{5} = \{5, 16, 27, 38, 49, 60\}$ and $\overline{6} = \{6, 17, 28, 39, 50, 61\}$, using the incompletely reduced numbers 49 and 50 is such an example:

$$
\begin{aligned}
S \;&=\; 49 - 50 \\
&=\; -1 \qquad (b = 1 \text{ Step 4}) \\
&=\; 64 - 1 \qquad (\text{two's complement Step 4}) \\
&=\; 63 \\
T \;&=\; 63 + 2 \qquad (\text{correction Steps 5–8}) \\
&=\; 65 \qquad (c = 1 \text{ Step 8}) \\
&=\; 65 - 64 \quad (\text{ignore carry Step 8}) \\
&=\; 1 \\
U \;&=\; 1 + 9 \qquad (\text{correction Steps 9–11}) \\
&=\; 10 \qquad (\text{return Step 12})
\end{aligned}
$$

  The result is indeed correct since 10 is equal to (-1) modulo 11.

5   Montgomery Modular Multiplication

The modular multiplication operation multiplies the input operands A and B and reduces the product modulo p, i.e., it computes $C := AB \pmod{p}$. The reduction operation often requires bit-level shift-subtract operations [6]. We are interested in algorithms which requires word-level operations. Instead of the classical modular multiplication operation, we prefer to use the Montgomery modular multiplication [9] which computes

$$T := ABR^{-1} \pmod{p} , \tag{13}$$

where R is an integer with property $\gcd(R, p) = 1$. The selection of R is very important since it determines the algorithmic details and the speed of the Montgomery multiplication. Generally R is selected as the smallest power of 2 which is larger than p, i.e., $R = 2^k$, where $k = \lceil \log_2 p \rceil$. Thus, we have $1 < p < R$, however, $2p > R$. If k is not an integer multiple of the word-length w, this selection requires that we perfom bit-level operations. Following the general premise of this paper, we propose the use of $R = 2^m$, where $m = sw$, in order to avoid performing bit-level operations. In this case, 2^m may be several times larger than p, as it was also the case for the addition and subtraction algorithms presented in the previous sections.

We propose to use the Montgomery multiplication algorithm for incompletely reduced numbers, which receives two numbers A and B in the range $[0, 2^m - 1]$, and computes the result T which is also an incompletely reduced number in the range $[0, 2^m - 1]$, given by (13). In the high-level view, the Montgomery multiplication algorithm computes the result T using

$$T = \frac{AB + p\,(ABp' \bmod R)}{R} , \tag{14}$$

where p' is defined using the multiplicative inverse $R^{-1} \pmod{p}$ as

$$RR^{-1} - pp' = 1 , \tag{15}$$

and computed using the extended Euclidean algorithm. The algorithm receives the inputs $A, B \in [0, R-1]$ and computes the result T in (14). Since $A, B < R$, the result of the operation in (14) will have the maximum value

$$\frac{(R-1)(R-1) + p(R-1)}{R} = \frac{(R-1)(R-1+p)}{R} < R - 1 + p . \tag{16}$$

In other words, T as computed by (14) exceeds R only by an additive factor of p, therefore, we need to perform only a single subtraction to bring it back to the range $[0, R-1]$.

The word-level description of the Montgomery multiplication involves a word-level multiplication operation, which we denote as

$$(c, T_j) := T_j + A_i \cdot B_j + c , \tag{17}$$

in which the new value of T_j and the new carry word c are computed using the old value of T_j and also the 1-word operands A_i and B, and the old carry word c. Here, all operands $A_i, B_j, T_j, c \in [0, 2^w - 1]$, i.e., they are all 1-word numbers. Since we have

$$(2^w - 1) + (2^w - 1) \cdot (2^w - 1) + (2^w - 1) = (2^w - 1)(2^w + 1) = 2^{2w} - 1 , \tag{18}$$

the result of the operation in (17) is a 2-word number represented using the 1-word numbers T_j and c.

The details of different Montgomery multiplication algorithms can be found in [7]. Here we describe an algorithm which computes T using the least significant word of the number p' defined in (15). Since $R = 2^{sw}$, we can reduce the equation (15) modulo 2^w, and obtain

$$-pp' = 1 \pmod{2^w} . \tag{19}$$

Let P_0 and Q_0 be the LSW of p and p', respectively. Then, Q_0 is the negative of the multiplicative inverse of the LSW of p modulo 2^w, i.e.,

$$Q_0 = -P_0^{-1} \pmod{2^w} . \tag{20}$$

This 1-word number can be computed very quickly using a variation of the extended Euclidean algorithm given in [3]. The Montgomery multiplication algorithm for computing $T = AB2^{-m} \pmod{p}$ using Q_0 is given below.

<div align="center">

Montgomery Modular Multiplication Algorithm

</div>

Inputs:	$A = (A_{s-1} \cdots A_1 A_0)$ and $B = (B_{s-1} \cdots B_1 B_0)$
Auxilary:	Q_0 and $p = (P_{s-1} \cdots P_1 P_0)$
Output:	$T = (T_{s-1} \cdots T_1 T_0)$
Step 1:	for $j = 0$ to $s - 1$
Step 2:	$\quad T_j := 0$
Step 3:	for $i = 0$ to $s - 1$
Step 4:	$\quad c := 0$
Step 5:	$\quad$ for $j = 0$ to $s - 1$
Step 6:	$\quad\quad (c, T_j) := T_j + A_i \cdot B_j + c$
Step 7:	$\quad T_s := c$
Step 8:	$\quad M := T_0 \cdot Q_0 \pmod{2^w}$
Step 9:	$\quad c := (T_0 + M \cdot P_0)/2^w$
Step 10:	$\quad$ for $j = 1$ to $s - 1$
Step 11:	$\quad\quad (c, T_{j-1}) := T_j + M \cdot P_j + c$
Step 12:	$\quad (c, T_{s-1}) := T_s + c$
Step 13:	if $c = 0$ return $T = (T_{s-1} \cdots T_1 T_0)$
Step 14:	$b := 0$
Step 15:	for $j = 0$ to $s - 1$
Step 16:	$\quad (b, T_j) := T_j - P_j - b$
Step 17:	return $T = (T_{s-1} \cdots T_1 T_0)$

The explanations and proofs of the steps are given below:

- In Steps 1 and 2, we clear the words of the result T to zero. The final result $T = AB2^{-m} \pmod{p}$ will be located in the s-word T at the end of the computation.

- The first part of the multiplication loop (Steps 3-7) computes a partial product T which is of length $s + 1$. For $i = 0$, this value is given as

$$T := A_0 \cdot B .$$

Since $A_0 \in [0, 2^{w-1}]$ and $B \in [0, 2^{m-1}]$, this value of T is less than equal to

$$2^{w-1} \cdot 2^{m-1} = 2^{w-1} \cdot 2^{sw-1} = 2^{(s+1)w-2} .$$

- In Steps 8-12, we are reducing T modulo p in such a way that it is now of length s words at the end of Step 12. This is accomplished using the following substeps:

  - First, in Step 8, we multiply the LSW of T by Q_0 modulo 2^w. Recall that Q_0 is the LSW of p' or it is equal to $-P_0^{-1} \pmod{2^w}$. Thus, M (which is a 1-word number) is given as
  $$M := T_0 \cdot Q_0 = T_0 \cdot (-P_0^{-1}) = -T_0 P_0^{-1} \pmod{2^w} .$$

  - Then, in Step 9, we compute $T_0 + M \cdot P_0$ which is equal to
  $$X := T_0 + M \cdot P_0 := T_0 + (-T_0 P_0^{-1})P_0 .$$

  Note that X is a 2-word number, however, the LSW of X is zero since
  $$T_0 + (-T_0 P_0^{-1})P_0 = 0 \pmod{2^w} .$$

  Therefore, after the division by 2^w in Step 9, we obtain the 1-word carry c from the computation $T_0 + M \cdot P_0$.

  - Then, in the remaining steps, i.e., in Steps 10-12, we are finishing the computation of $T + M \cdot P$. Since the LSW of the result is zero, we are also shifting the result by 1 word to the right (towards the least significant) in order to obtain the s-word number given by Equation (14).

- According to Equation (16), the result computed at the end of Step 12 can exceed $R - 1$ by at most p, and thus, a single subtraction will bring it back to the range $[0, R-1]$. In Step 13, we check if the carry computed at the end of Step 12 is 1, i.e., if T exceeds $R - 1$. If there is no carry, then we return the result T in Step 13 as the final product.

- Otherwise, we perform a simple subtraction $T := T - p$ in order to bring back T to the range $[0, R - 1]$. The subtraction operation is accomplished in Steps 14-16, and the final product value is returned in Step 17.

Therefore, the Montgomery modular multiplication works even if the modulus $R = 2^{sw}$ is much larger than p, i.e., it need not be the smallest number of the form 2^i which is larger than p. While there may be several correction steps needed in the addition and subtraction operations, a single subtraction operation is sufficient for computing the Montgomery product $T = AB2^{-sw} \pmod{p}$.

The complete Montgomery multiplication and the incomplete Montgomery multiplication algorithms differ only slightly from one another. Algorithmically, these two algorithms are similar. Their main differences are in the way the input and output operands are specified:

- The radix R in the complete Montgomery multiplication algorithm is taken as 2^k, while the incomplete Montgomery multiplication algorithm uses the value 2^{sw}, therefore avoiding bit-level operations if k is not an integer multiple of w.

- The complete Montgomery multiplication algorithm requires that input operands be complete, i.e., numbers in the range $[0, p-1]$, while the incomplete Montgomery multiplication algorithm requires that input operands be in the range $[0, 2^m - 1]$.

- The complete Montgomery multiplication algorithm computes the final result as a complete number, i.e., a number in the range $[0, p-1]$, while the incomplete Montgomery multiplication algorithm computes the result in the range $[0, 2^m - 1]$.

5.1 Multiplication Examples

Let $p = 53$, $k = 4$, $w = 3$, $m = 6$, and $s = 2$. Since $p = 53 = (110101)$ and $P_0 = (101) = 5$, we compute $Q_0 = -P_0^{-1} \pmod{2^w}$ as

$$Q_0 = -5^{-1} \pmod 8 = -5 = 3 \ .$$

Also, we have $R = 2^m = 2^6 = 64$. Here are two multiplication examples:

- We illustrate the multiplication of $\overline{5} = \{5, 58\}$ and $\overline{7} = \{7, 60\}$ using the incompletely reduced numbers 58 and 60. Taking $A = 58 = (111\ 010)$ and $B = 60 = (111\ 100)$, we compute $T = A \cdot B \cdot R^{-1} \pmod p$ as follows:

  - Step 3: $i = 0$
  - Step 4,5,6 and $j = 0$: $(c, T_0) := A_0 \cdot B_0 = 2 \cdot 4 = 8 = (001\ 000)$.
  - Step 5,6 and $j = 1$: $(c, T_1) := A_0 \cdot B_1 + c = 2 \cdot 7 + 1 = 15 = (001\ 111)$.
  - Step 7: $T_2 = c = 1$. Therefore, we have T=$(001\ 111\ 000)$
  - Step 8: $M = T_0 \cdot Q_0 = 0 \cdot 3 \pmod 8 = 0$.
  - Step 9: $c = (T_0 + M \cdot P_0)/8 = (0 + 0 \cdot 5)/8 = 0$.
  - Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 7 + 0 \cdot 6 + 0 = 7 = (000\ 111)$.
  - Step 12: $(c, T_1) = T_2 + c = 1 + 0 = 1 = (000\ 001)$. We now have $T = (001\ 111)$.
  - Step 3: $i = 1$
  - Step 4,5,6 and $j = 0$: $(c, T_0) := T_0 + A_1 \cdot B_0 = 7 + 7 \cdot 4 = 35 = (100\ 011)$.
  - Step 5,6 and $j = 1$: $(c, T_1) := T_1 + A_1 \cdot B_1 + c = 1 + 7 \cdot 7 + 4 = 54 = (110\ 110)$.
  - Step 7: $T_2 = c = 6$. Therefore, we have $T = (110\ 110\ 011)$.
  - Step 8: $M = T_0 \cdot Q_0 = 3 \cdot 3 \pmod 8 = 1$.
  - Step 9: $c = (T_0 + M \cdot P_0)/8 = (3 + 1 \cdot 5)/8 = 1$.
  - Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 6 + 1 \cdot 6 + 1 = 13 = (001\ 101)$.
  - Step 12: $(c, T_1) = T_2 + c = 6 + 1 = 7 = (000\ 111)$. We now have $T = (111\ 101)$.
  - Step 13: since $c = 0$, the result $T = (111\ 101)$ is returned.

  The final result is $T = (111\ 101) = 61$ which is an incomplete number. The complete equivalent of 61 is 8 which is equal to $5 \cdot 7 \cdot 64^{-1} \pmod{53}$.

- We now illustrate the multiplication of $\overline{8} = \{8, 61\}$ and $\overline{10} = \{10, 63\}$ using the incompletely reduced numbers 61 and 63. Taking $A = 61 = (111\ 101)$ and $B = 63 = (111\ 111)$, we compute $T = A \cdot B \cdot R^{-1} \pmod p$ below. In this multiplication, the subtraction steps (Steps 14-17) are performed.

  - Step 3: $i = 0$
  - Step 4,5,6 and $j = 0$: $(c, T_0) := A_0 \cdot B_0 = 5 \cdot 7 = 35 = (100\ 011)$.
  - Step 5,6 and $j = 1$: $(c, T_1) := A_0 \cdot B_1 + c = 5 \cdot 7 + 4 = 39 = (100\ 111)$.
  - Step 7: $T_2 = c = 4$. Therefore, we have T=$(100\ 111\ 011)$
  - Step 8: $M = T_0 \cdot Q_0 = 3 \cdot 3 \pmod 8 = 1$.

- Step 9: $c = (T_0 + M \cdot P_0)/8 = (3 + 1 \cdot 5)/8 = 1$.
- Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 7 + 1 \cdot 6 + 1 = 14 = (001\ 110)$.
- Step 12: $(c, T_1) = T_2 + c = 4 + 1 = 5 = (000\ 101)$. We now have $T = (101\ 110)$.
- Step 3: $i = 1$
- Step 4,5,6 and $j = 0$: $(c, T_0) := T_0 + A_1 \cdot B_0 = 6 + 7 \cdot 7 = 55 = (110\ 111)$.
- Step 5,6 and $j = 1$: $(c, T_1) := T_1 + A_1 \cdot B_1 + c = 5 + 7 \cdot 7 + 6 = 60 = (111\ 100)$.
- Step 7: $T_2 = c = 6$. Therefore, we have $T = (110\ 110\ 011)$.
- Step 8: $M = T_0 \cdot Q_0 = 7 \cdot 3 \pmod{8} = 5$.
- Step 9: $c = (T_0 + M \cdot P_0)/8 = (7 + 5 \cdot 5)/8 = 4$.
- Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 4 + 5 \cdot 6 + 4 = 38 = (100\ 110)$.
- Step 12: $(c, T_1) = T_2 + c = 7 + 4 = 11 = (001\ 011)$.
- Step 13: since $c = 1$, we execute the subtraction steps below.
- Step 14: $b = 0$.
- Step 15,16 and $j = 0$: $(b, T_0) = T_0 - P_0 - b = 6 - 5 - 0 = 1 = (000\ 001)$.
- Step 15,16 and $j = 1$: $(b, T_1) = T_1 - P_1 - b = 3 - 6 - 0 = -3 \pmod{8} = 5(000\ 101)$.
- Step 17: $T = (101\ 001) = 41$ is returned.

The final result is $T = (101\ 001) = 41$ which is a complete number. The final result 41 is equal to $8 \cdot 10 \cdot 64^{-1} \pmod{53}$.

6 Implementation Results and Conclusions

In order to measure the speed of the incomplete addition, subtraction, and Montgomery multiplication algorithms together with their complete counterparts, we have implemented the ECDSA [10, 4] over the finite field $GF(p)$. The objective was to assess the performance impact of the incomplete arithmetic as compared to the complete arithmetic on the ECDSA. We executed the ECDSA code several hundred times using two different random elliptic curve sets. The addition, subtraction, and multiplication timings shown in Table 1 are obtained by running the ECDSA code and measuring the timings of the arithmetic operations in that context. The computer platform was a 450-MHz Pentium II PC running Windows NT 4.0 operating system, with 256 megabytes of memory. The code is written entirely in C. The timings of the complete and incomplete routines are tabulated in Table 1 in microseconds.

The speedup in percentage is obtained by subtracting the incomplete timing result from the complete timing result and then dividing it by the complete timing result. As can be seen from Table 1, the incomplete addition is 34–43 % faster than the complete addition for the range of k from 161 to 256. Similarly, the incomplete subtraction is 17–23 % faster than the complete subtraction. The reduction in the speedup for the subtraction operation as compared to the addition operation is mainly due to the fact that we have to perform more corrections than the addition operation. On the other hand, we obtain only a small speedup (3–5 %) for the incomplete Montgomery multiplication operation since the incomplete and complete Montgomery multiplication algorithms differ only slightly.

Table 1: Incomplete and complete arithmetic timings in microseconds.

k	Addition			Subtraction			Multiplication		
	Complete	Incomplete	%	Complete	Incomplete	%	Complete	Incomplete	%
161	1.85	1.11	40	1.43	1.10	23	4.80	4.58	5
176	1.90	1.11	42	1.38	1.10	20	4.74	4.57	4
192	2.00	1.26	37	1.38	1.04	25	4.79	4.62	4
193	1.98	1.23	38	1.47	1.20	18	6.36	6.17	3
208	2.14	1.22	43	1.46	1.19	18	6.40	6.13	4
224	2.03	1.28	37	1.45	1.16	20	6.35	6.17	3
225	2.20	1.30	41	1.58	1.29	18	8.06	7.73	4
240	2.23	1.32	41	1.53	1.27	17	8.03	7.74	4
256	2.31	1.52	34	1.53	1.27	17	8.02	7.76	3

On the other hands, the timing resuls of the ECDSA signature generation algorithm are given in Table 2 in milliseconds. These ECDSA timings are obtained without using any precomputation. The ECDSA code was executed several hundred times using two different random elliptic curve sets for bit lengths as specified in Table 2. These random elliptic curves are not special in any meaning of the term; the curve parameters are randomly selected from the field $GF(p)$, i.e., they are full-size numbers. The implementation results have shown that the ECDSA algorithm can be made to execute 10–13 % faster using the incomplete modular arithmetic, which is very significant. The method of computation of the point multiplication, i.e., the computation of dP in ECDSA, where d is an integer and P is a point on the curve, is the addition-subtraction elliptic scalar multiplication [4]. Furthermore, we use the projective coordinate system to represent the points on the curve.

Table 2: ECDSA over $GF(p)$ signature generation timings in milliseconds.

k	C code only			C + Assembly
	Complete	Incomplete	%	Incomplete
161	13.6	12.0	12	5.3
176	14.8	12.9	13	5.8
192	16.5	14.7	11	6.6
193	20.8	18.4	12	8.5
208	22.6	19.7	13	9.1
224	23.7	21.1	11	9.7
225	29.8	26.5	11	12.2
240	31.1	27.9	10	12.8
256	34.2	30.8	10	14.0

According to the IEEE Standard [4], an elliptic curve point doubling requires 9 field additions, 3 field subtractions, and 10 field multiplications. On the other hand, an elliptic curve point addition requires 3 field additions, 5 field subtractions, and 11 field multiplications. As an example of performance estimation, let us take the scalar multiplication operation dP in which the integer d is 176 bits. This means, the addition-subtraction elliptic curve scalar multiplication requires approximately $\frac{176}{3} \approx 59$ point additions and 175 point doublings. Therefore, the computation of dP would require approximately $(175 \times 9 + 59 \times 3) = 1752$ field additions, $(175 \times 3 + 59 \times 5) = 820$ field subtractions, and $(175 \times 10 + 59 \times 11) = 2399$ field multiplications. This gives the total time

for the computation of the elliptic curve scalar multiplication using the complete and incomplete arithmetic as

$$\begin{aligned} \text{complete} &= (1752 \times 1.9 + 820 \times 1.38 + 2399 \times 4.74) \approx 15.83 \text{ ms} \\ \text{incomplete} &= (1752 \times 1.11 + 820 \times 1.1 + 2399 \times 4.57) \approx 13.81 \text{ ms} \end{aligned}$$

Thus, this rough estimation shows that the incomplete arithmetic based point multiplication operation is approximately $\frac{15.83}{13.81} \approx 1.15$ times faster than its complete arithmetic version. This means, the incomplete arithmetic for 176 bits would be about 15 percent faster than the complete arithmetic for ECDSA; our actual timing result in Table 2 gives this as 13 percent which is very close.

7   Conclusions

In this paper, we have presented a new methodology for speeding up arithmetic operations, and shown that the new method provides up to 13 % speedup in the execution of the ECDSA algorithm over the field $GF(p)$ for the length of p in the range $161 \le k \le 256$. Coupled with some machine-level programming, the ECDSA algorithm can be made significantly faster, as shown in the last column of Table 2.

In this paper, we applied the incomplete arithmetic only to the modular addition, the modular subtraction, and the Montgomery multiplication operations. A similar word-level methodology was described in [12] for the modular inverse and Montgomery modular inverse operations. These algorithms are also of type word-level, i.e., they avoid bit-level operation.

References

[1] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent Numbers 5,463,690 and 5,271,061 and 5,159,632, October 1995.

[2] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[3] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. Springer, Berlin, Germany, 1990.

[4] IEEE. P1363: Standard specifications for public-key cryptography. Draft Version 13, November 12, 1999.

[5] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[6] Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.

[7] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[8] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.

[9] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[10] National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.

[11] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

[12] E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(7):763–766, July 2000.