Montgomery Reduction Algorithm for Modular Multiplication Using Low-Weight Polynomial Form Integers[*]

Jaewook Chung[†] and M. Anwar Hasan
jaewook.chung@gmail.com and ahasan@secure.uwaterloo.ca
Department of Electrical and Computer Engineering,
University of Waterloo, Ontario, Canada

Abstract

In this paper, we extend a recent piece of work on low-weight polynomial form integers (LWPFIs). We present a new coefficient reduction algorithm based on the Montgomery reduction algorithm and provide its detailed analysis results. We give a condition for eliminating the final subtractions at the end of our Montgomery reduction algorithm adapted to perform the coefficient reduction. Our experimental results show that a new coefficient reduction algorithm is indeed more efficient than the one presented in [1].

Keywords: More generalized Mersenne numbers, Low-weight polynomial form integers, adapted modular number system, polynomial modular number system, Montgomery reduction algorithm

1. Introduction

In [1], Low-weight polynomial form integers (LWPFIs) are defined as integers that can be represented in a degree-l monic polynomial form, $f(t) = t^l + f_{l-1}t^{l-1} + \cdots + f_1 t + f_0$, where $|f_i| \leq 1$. Modular multiplication using an LWPFI is performed in two phases. First, integers represented in polynomial form are multiplied in the polynomial ring $\mathbb{Z}[t]/f(t)$. Then the coefficients of the resulting polynomial are reduced. In [1], the coefficient reduction algorithms are based on a division algorithm derived from the Barrett reduction algorithm. In this paper, we present a new coefficient reduction algorithm based on the Montgomery reduction algorithm. Moreover, we further generalize LWPFIs by removing the restriction on f_i's. We analyze the new coefficient reduction algorithm in this general framework.

The remainder of this article is organized as follows. After a brief discussion on related work in Section 2, we present the definition of our extended LWPFIs in Section 3. Then, in Section 4, we present our new coefficient reduction method based on the Montgomery reduction algorithm. We give detailed analysis and consider some interesting special cases. We show a condition on parameters for which our new coefficient reduction algorithm can be performed without any final subtractions. In Section 5, we discuss how the performance of our coefficient reduction algorithm is compared to that of the Montgomery reduction algorithm, and show experiemental results. In Section 6, we consider the applications of LWPFIs and conclusions follow in 7.

2. Related Work

LWPFIs have been first introduced in [2] and the refined results are shown in [1]. In both [2] and [1], integers are represented in polynomial $\mathbb{Z}[t]/f(t)$, where its coefficients are approximately the size of t. Bajard, Imbert and Plantard have proposed the Adapted Modular Number System (AMNS) [3] and the Polynomial Modular Number Systems (PMNS) [4]. Both AMNS and PMNS are similar to number systems using LWPFIs in that they use low-weight polynomial form moduli. However, integers are represented quite differently in such number systems. In AMNS, arithmetic operations are efficient for careful choice of the parameters. It is shown in [3] that the modular multiplication in AMNS is more efficient than the usual modular multiplication of integers using the Montgomery reduction algorithm. The main drawback of PMNS is that it requires a large look-up table for coefficient reduction.

[*]This work was supported in part by Natural Science and Engineering Research Council of Canada (NSERC) discovery and strategic project grants awarded to Dr. Hasan.

[†]Jaewook Chung's research was funded by Bell 125th Anniversary Scholarship.

3. Extended Definition of Low-Weight Polynomial Form Integers

In [1], LWPFIs are defined as integers expressed in low-weight, monic polynomial form: $p = f(t) = t^l + f_{l-1}t^{l-1} + \cdots + f_1 t + f_0$, where $l \geq 2$, $f_i \in \{0, \pm 1\}$ and $t > 2(2^{2l+1} - 1)(2^l - 1)$.

Here we loosen the restriction on f_i's so that $|f_i| \leq \xi$ for some small positive integer $\xi < t$. The condition $t > 2(2^{2l+1} - 1)(2^l - 1) \approx 2^{3l+2}$ is applied in [1] due to the use of coefficient reduction based on the Barrett division algorithm. However, such a condition is not needed in our improved coefficient reduction presented here. In this paper, we work in this general framework and narrow down conditions on parameters that allow efficient implementation of modular arithmetic modulo an LWPFI.

Definition 1 (LWPFI Redefined) *For a degree-l, monic polynomial $f(t) = t^l + f_{l-1}t^{l-1} + \cdots + f_1 t + f_0$, where t is a positive integer and $|f_i| \leq \xi$ for some small positive integer $\xi < t$, $p = f(t)$ is a* **low-weight polynomial form integer**.

In modular arithmetic based on LWPFI moduli, we express elements of $\mathbb{Z}_p$ as polynomials in $\mathbb{Z}[t]/f(t)$. Such a representation always exists for any element in $\mathbb{Z}_p$ using coefficients at most $(t + \xi)/2$ in magnitude.

Proposition 1 *For any integer $x \in \mathbb{Z}_p$, there exists a degree-$(l-1)$ polynomial $x(t) = \sum_{i=0}^{l-1} x_i t^i$ such that $x \equiv x(t) \pmod{p}$ and $|x_i| \leq \psi$, if $\psi \geq (t + \xi)/2$.*

Proof: Let $p_{\max} = t^l + \xi t^{l-1} + \cdots + \xi t + \xi$. Then $p_{\max}$ is the maximum possible LWPFI of the form $f(t) = t^l + \sum_{i=0}^{l-1} f_i t^i$, where $|f_i| \leq \xi$. Let $x(t) = \sum_{i=0}^{l-1} x_i t^i$. If $\max(x(t)) - \min(x(t)) \geq p_{\max}$ holds, then $x(t)$ can represent any element in $\mathbb{Z}_{f(t)}$. It is straightforward that

$$\max(x(t)) = \sum_{i=0}^{l-1} \psi t^i = -\min(x(t)).$$

It follows that

$$\max(x(t)) - \min(x(t)) \geq p_{\max}$$
$$\iff (2\psi - \xi) \cdot \frac{t^l - 1}{t - 1} \geq t^l.$$

It is easy to see that $2\psi - \xi = t - 1$ does not satisfy the above inequality, but $2\psi - \xi \geq t$ does. Therefore $\psi \geq (t + \xi)/2$.

We let $\psi_{\min} = (t + \xi)/2$. However, in practice, the magnitudes of the coefficients do not have to be limited to ψ_{min}. To find a polynomial that corresponds to a given integer, Algorithm 1 can be used. The resulting polynomial has coefficients that are at most $(t/2 + \xi)$ in magnitude. Since $t/2 + \xi > \psi_{\min}$, Algorithm 1 results in a slightly redundant representation.

Algorithm 1 Conversion to Polynomial Form

```
Require: an integer $0 \leq x < p$, where $p = f(t) = t^l + f_{l-1}t^{l-1} + \cdots + f_1 t + f_0$.
Ensure: a polynomial $x(t) = \sum_{i=0}^{l-1} x_i t^i$, such that $x \equiv x(t) \pmod{p}$, where $|x_i| \leq t/2 + \xi$.
1: $c_{-1} \leftarrow x$.
2: for $i$ from 0 to $l-1$ do
3:     Find $c_i$ and $x_i$ such that $c_{i-1} = c_i t + x_i$, where $-t/2 \leq x_i < t/2$.
4: end for
5: for $i$ from 0 to $l-1$ do
6:     $x_i \leftarrow x_i - f_i \cdot c_{l-1}$.        (Note: $|c_{l-1}| \leq 1$ )
7: end for
8: return $x(t) = \sum_{i=0}^{l-1} x_i t^i$.
```

4. Modular Multiplication Using LWPFI moduli

In this section, we present an efficient modular multiplication scheme using LWPFI moduli. The modular multiplication using LWPFI moduli is performed in the following steps.

1. POLY-MULT: $\hat{z}(t) = x(t) \cdot y(t)$.
2. POLY-REDC: $z'(t) = \hat{z}(t) \bmod f(t)$.
3. COEFF-REDC: coefficient reduction of $z'(t)$.

The above modular multiplication scheme is called the *LWPFI modular multiplication*. POLY-MULT step can be performed by at most l^2 multiplications of coefficients using the schoolbook method. Sub-quadratic multiplication algorithms may be applied to achieve better performance [5, 6, 7, 8]. POLY-REDC step requires at most $(l-1)\tau$ constant multiplications by integers at most ξ in magnitude, where τ is the number of non-zero f_i's. The range of f_i we use here is larger than that in [1]. Note that, due to this extended range for f_i's, our POLY-REDC step is potentially slower than that in [1]. However, we will not go over the details on POLY-REDC and focus only on the establishment of a new coefficient reduction algorithm based on the Montgomery reduction algorithm. For fixed $f(t)$, one may consider combining POLY-MULT and POLY-REDC steps for better performance as we propose in [1].

Suppose that the coefficients of $x(t)$ and $y(t)$ are at most ψ in magnitude. It easily follows that the result of POLY-REDC has coefficients that are at most $\psi^2((\xi+1)^l-1)/\xi$ in magnitude as shown in Proposition 2. Throughout this paper, we will use λ to denote $((\xi+1)^l-1)/\xi$.

Proposition 2 $|z'_i| \le \lambda\psi^2$.

Proof:
Let $x(t)$ and $y(t)$ be the polynomials whose coefficients are at most ψ in magnitude. Let $\hat{z}(t) = (\hat{z}_{2l-2}, \ldots, \hat{z}_1, \hat{z}_0) = x(t) \cdot y(t)$. It follows that $|\hat{z}_i| \le (i+1)\psi^2$ for $i = 0, \ldots, l-1$ and $|\hat{z}_i| \le (2l-1-i)\psi^2$ for $i = l, \ldots, 2l-2$. The magnitudes of coefficients in $z'(t) = \hat{z}(t) \bmod f(t)$ are maximum when $f(t) = t^l + \xi\sum_{i=0}^{l-1} t^i$ and the maximum coefficient is at z'_{l-1}. In this case, it can be shown that $\max\{(z'_{l-1})\} = \Gamma_{l-1}\psi^2$, where $\Gamma_i = 1 + (\xi+1)\Gamma_{i-1}$ for $i > 0$ and $\Gamma_0 = 1$. Note that $\Gamma_i = ((\xi+1)^{i+1}-1)/\xi$. Therefore $|z'_i| \le ((\xi+1)^l-1)/\xi \cdot \psi^2$.

In Section 4.6, we discuss how the value ψ is related to other parameters, t, ξ and l. In [1], $\psi = t + 2^{l+2} - 2$ is fixed and a division algorithm derived from the Barrett reduction algorithm is used to perform COEFF-REDC step. In this work, we apply the Montgomery reduction algorithm to perform COEFF-REDC step and determine appropriate value ψ.

Note that the output of our COEFF-REDC based on the Montgomery reduction algorithm (MONT-COEFF-REDC) is different from the output from Algorithm 5 in [1]. In [1], Algorithm 5 computes $z(t)$ such that $z(t) \equiv x(t) \cdot y(t) \pmod p$. However, the MONT-COEFF-REDC presented here outputs $z(t) \equiv x(t) \cdot y(t) \cdot b^{-q} \pmod p$, where b is the radix used to represent coefficients of polynomials in $\mathbb{Z}[t]/f(t)$ and q is a positive integer. Consider two integers $\bar{x}(t) \equiv x(t) \cdot b^q \pmod p$ and $\bar{y}(t) \equiv y(t) \cdot b^q \pmod p$. These are the transformation of $x(t)$ and $y(t)$ to the so-called *the Montgomery domain*. The direct product of $\bar{x}(t)$ and $\bar{y}(t)$ in $\mathbb{Z}[t]/f(t)$ results in $\bar{x}(t)\bar{y}(t) \equiv x(t)y(t) \cdot b^{2q} \pmod p$. Applying our new coefficient reduction algorithm results in $\bar{z}(t) \equiv x(t)y(t) \cdot b^q \pmod p$, whose coefficients are at most ψ. Note that the result is the transformation of $x(t)y(t)$ to the Montgomery domain. We discuss the relationship between the value q and other parameters of LWPFI in Section 4.6.

4.1. Montgomery Reduction Algorithm

The Montgomery algorithm performs modular reduction without using any division instruction of the underlying processor [9]. Let m be a modulus, and T be a positive integer which is to be reduced. We choose an integer R such that $R > m$, $\gcd(m, R) = 1$ and $0 \le T < mR$.

Algorithm 2 Montgomery Algorithm for Integers Reduction (MAIR)

Require: integers T and $m = (m_{k-1} \cdots m_1 m_0)_b$, such that $R = b^q$, $0 \le T < mR$ and $\gcd(b, m) = 1$.
Ensure: $T \cdot b^{-q} \bmod m$.

```
1: $T_0 \leftarrow T$.
2: for $i$ from 0 to $q-1$ do
3:     $u_i \leftarrow -m^{-1} \cdot T_i \bmod b$.
4:     $T_{i+1} \leftarrow (T_i + u_i \cdot m)/b$.
5: end for
6: if $T_q \ge m$ then
7:     $T_q \leftarrow T_q - m$.
8: end if
9: return $T_q$.
```

Algorithm 2 computes $T \cdot b^{-q} \bmod m$, given an integer $0 \le T < mR$, where $R = b^q$. In each iteration of Algorithm 2, a multiple of the modulus M is added to T_i such that the least significant digit becomes zero. Then, the division of T_{i+1} by b can be performed simply by shifting all digits of T by one place to the right. If q is chosen to be the digit length of T, then it can be easily shown that $T_q \in [0, 2m)$. Therefore, one final subtraction by m may be required to output an integer within $[0, m)$. Some researchers have proposed ways to eliminate this final subtraction [10, 11, 12]. Walter proposed using q such that $2m < b^{q-1}$ [13]. Hachez and Quisquater improved this condition to $m < b^{q-1}$ for $b = 2$ [14]. Walter improved this condition again to $4m < b^q$ [15]. Line 3 of MAIR requires one single-precision multiplication and line 4 requires k single-precision multiplications, where k is the digit length of m. Therefore, MAIR requires a total of $q(k+1)$ single-precision multiplications.

4.2. COEFF-REDC based on Montgomery Reduction Algorithm

Here, we construct a new coefficient reduction algorithm which is similar to Algorithm 2. Given an input polynomial $z'(t)$ of degree $(l-1)$, our new algorithm computes a polynomial whose evaluation at t is congruent to $z'(t) \cdot b^{-q} \bmod p$.

Before, we begin the description of a new coefficient reduction algorithm, we clarify notations that we use in this paper. Let $\vec{u}$ and $\vec{v}$ be the column vectors in $\mathbb{Z}^l$ such that the following condition is satisfied:

$$[t^{l-1}, \ldots, t, 1] \cdot \vec{u} \equiv [t^{l-1}, \ldots, t, 1] \cdot \vec{v} \pmod p. \quad (1)$$

Then we say $\vec{u}$ *is congruent to* $\vec{v}$ *modulo* p and write as $\vec{u} \cong_p \vec{v}$. We slightly abuse this notation and write as $\vec{u} \cong_b v$ for some integer v satisfying $[t^{l-1}, \ldots, t, 1] \cdot \vec{u} \equiv v \pmod{b}$. We also say $\vec{u}$ *is congruent to* v *modulo* b, if $\vec{u} \cong_b v$. We use '$\equiv$', to express element-wise congruence relation, i.e., $\vec{u} \equiv \vec{v} \pmod{b}$. In "$\vec{u} \bmod b$", modulo operation applies to each element of $\vec{u}$.

Let $x(t) = (x_{l-1}, \ldots, x_1, x_0)_t$ be the result of POLY-REDC step and b be the radix used for representing x_i's. When performing multiplication in $GF(p)[t]/f(t)$, we can apply Algorithm 2 individually to each coefficient to reduce them modulo p. However, individual reduction of coefficients does not output correct results when working in $\mathbb{Z}[t]/f(t)$. To reduce coefficients in $\mathbb{Z}[t]/f(t)$, we must apply the Montgomery reduction algorithm to all coefficients simultaneously.

The coefficient reduction is closely related to the *closest vector problem* from lattice theory. A lattice $\mathcal{L}$ is a discrete subgroup of $\mathbb{R}^l$. Let $\vec{V} = \{\vec{v}_1, \ldots, \vec{v}_{d-1}, \vec{v}_d\}$ be a set of linearly independent vectors in $\mathbb{R}^l$. The lattice $\mathcal{L} = \mathcal{L}(\vec{V})$ is a set of all integral combination of $\vec{v}_i$'s. The set $\vec{V}$ is called the basis of the lattice $\mathcal{L}(\vec{V})$. If $d = l$, $\mathcal{L}$ is called a full-rank lattice. If $\vec{v}_i \in \mathbb{Z}^l$ for all i, then $\mathcal{L}$ is called an integral lattice. For our purpose, we assume that $\mathcal{L}$ is a full-rank, integral lattice.

Suppose $\vec{v}_i \cong_p 0 \pmod{p}$ for all $i = 1, \ldots, l$. Then all the lattice points in $\mathcal{L}$ represent 0 modulo p. Let $\vec{x}$ be a vector whose elements are the coefficients of $x(t)$. Suppose $\vec{y} \in \mathcal{L}(\vec{V})$ is the closest lattice point (with respect to L_∞ norm) to $\vec{x}$, then $\vec{z} = \vec{x} - \vec{y}$ belongs to the fundamental domain of $\mathcal{L}$. The coordinate values of $\vec{z}$ forms a polynomial $z(t)$ such that $z(t) \equiv x(t) \pmod{p}$ and it has only reasonably small coefficients. However, closest vector problem is believed to be NP-hard. There are polynomial time algorithms that give approximate solutions [16], but they require arithmetic using floating point or rational numbers and are too cumbersome to use for our purposes.

Rather than solving the closest vector problem, we search for $\vec{z'}$ such that $\vec{x} \cong_p \vec{z'} \cdot b^q \pmod{p}$ and the elements of $\vec{z'}$ are reasonably small. Below we show how to find such a vector $\vec{z'}$ using a method similar to the Montgomery reduction algorithm. This approach requires only simple integer arithmetic and enjoys good features of the Montgomery reduction algorithm for integers.

Algorithm 3 shows our Montgomery reduction algorithm adapted to perform COEFF-REDC step. Note that we have used $\vec{x}_q^{(i)}$ to denote the element of $\vec{x}_q$ at the i-th row in Algorithm 3. Moreover, F is an $l \times l$ integral matrix such that the following holds for any column vectors $\vec{x}$ and $\vec{u} \in \mathbb{Z}^l$:

$$\vec{x} + F \cdot \vec{u} \cong_p \vec{x}. \tag{2}$$

Algorithm 3 MONT-COEFF-REDC

```
Require: $x(t) = (x_{l-1}, \ldots, x_1, x_0)_t$, a matrix $F$ and $F' = -F^{-1} \bmod b$, where $\det F \neq 0$ and $\gcd(\det F, b) = 1$.
Ensure: $z(t) \equiv x(t) \cdot b^{-q} \pmod{p}$.
1: $\vec{x}_0 \leftarrow [x_{l-1}, x_{l-2}, \ldots, x_0]^T$.
2: for $i$ from 0 to $q - 1$ do
3:     $\vec{u}_i \leftarrow F' \cdot \vec{x}_i \bmod b$.
4:     $\vec{x}_{i+1} \leftarrow (\vec{x}_i + F \cdot \vec{u}_i)/b$.
5: end for
6: Perform final subtractions if necessary.
7: return $z(t) = \sum_{i=0}^{l-1} z_i t^i$, where $z_i = \vec{x}_q^{(i)}$.
```

A non-trivial matrix F that satisfies (2) can be constructed by collecting l column vectors that are congruent to 0 modulo p. Such a matrix F must be invertible modulo b, since we need $F' = -F^{-1} \bmod b$ in line 3 of Algorithm 3. The invertibility of F modulo b can be verified by checking if $\det F \neq 0$ and the determinant has no common factor with b, i.e., $\gcd(\det F, b) = 1$.

Theorem 1 *Algorithm 3 returns* $z(t) \equiv x(t) \cdot b^{-q} \pmod{p}$.

Proof: It is easily seen that each iteration of Algorithm 3 computes the following:

$$\vec{x}_{i+1} \leftarrow \frac{\vec{x}_i + F \cdot (-F^{-1} \cdot \vec{x}_i \bmod b)}{b}. \tag{3}$$

Since F is a collection of column vectors that are congruent to 0 modulo p, adding any integral linear combination of the column vectors in F to $\vec{x}_i$ does not change its value in $\mathbb{Z}_p$. Hence, $\vec{x}_{i+1} \cong_p (\vec{x}_i + F \cdot (-F^{-1} \cdot \vec{x}_i \bmod b)) \cdot b^{-1}$. The division by b in (3) is exact and requires no division, since

$$\begin{aligned}\vec{x} + F \cdot \vec{u} &= \vec{x} + F \cdot (-F^{-1} \cdot \vec{x} \bmod b) \\ &\equiv [0, \ldots, 0, 0]^T \pmod{b}.\end{aligned} \tag{4}$$

Therefore, $\vec{x}_{i+1} \cong_p \vec{x}_i \cdot b^{-1}$. In Algorithm 3, the process (3) is performed iteratively q times starting with $\vec{x}_0 = \vec{x}$ resulting in $\vec{x}_q \equiv \vec{x} \cdot b^{-q} \pmod{p}$. This is quite similar to the original Montgomery reduction algorithm. The only difference is that Algorithm 3 uses vectors and matrix, while the original Montgomery reduction algorithm deals with integers.

At this point, a number of questions arise: what are the conditions for q such that $\vec{x}_q$ are sufficiently reduced, so that the result can be used as input to the subsequent LWPFI modular multiplications? How do we construct the matrix F? Is Algorithm 3 efficient? Can we eliminate the final subtractions? We answer these questions in the following.

4.3. Construction of F and Analysis of Algorithm 3

For $p = f(t) = t^l + f_{l-1}t^{l-1} + \cdots f_1 t + f_0$, where $|f_i| \le \xi$, consider the following $l \times l$ matrix F:

$$F = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & -t - f_{l-1} \\ -t & 1 & \cdots & 0 & 0 & -f_{l-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -t & 1 & -f_1 \\ 0 & 0 & \cdots & 0 & -t & -f_0 \end{bmatrix}. \quad (5)$$

We have constructed the matrix F such that the column vectors of F are congruent to 0 modulo p, i.e., $F \cong_p [0, \ldots, 0, 0]$. It remains to verify whether F has its inverse modulo b. The invertibility of F modulo b can be easily checked as shown in Proposition 3.

Proposition 3 *The $l \times l$ matrix F as shown in* (5) *is invertible modulo b if and only if* $\gcd(p = f(t), b) = 1$ *and* $f(t) \neq 0$.

Proof: We perform some elementary row operations on both sides of $I_l \cdot F = F$, where I_l is an $l \times l$ identity matrix, to obtain

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 \\ t & 1 & \cdots & 0 & 0 & 0 \\ t^2 & t & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ t^{l-2} & t^{l-3} & \cdots & t & 1 & 0 \\ t^{l-1} & t^{l-2} & \cdots & t^2 & t & 1 \end{bmatrix} \cdot F$$
$$= \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & -C_{l-1} \\ 0 & 1 & \cdots & 0 & 0 & -C_{l-2} \\ 0 & 0 & \cdots & 0 & 0 & -C_{l-3} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 1 & -C_1 \\ 0 & 0 & \cdots & 0 & 0 & -C_0 \end{bmatrix}, \quad (6)$$

where $C_i = (t^l + \sum_{j=i}^{l-1} f_j t^j)/t^i$. Using the fact that the determinant of a triangular matrix is the product of all diagonal entries, we easily obtain that $\det(F) = -C_0 = -f(t)$ and F is invertible modulo b if and only if $\gcd(f(t), b) = 1$ and $f(t) \neq 0$. We remark that this invertibility condition of F modulo b is always satisfied when $p = f(t)$ is an odd number, for an even radix b.

We analyze the performance of Algorithm 3 in terms of the number of single-precision multiplications and single-precision additions/subtractions. The overhead caused by additions and subtractions shall not be ignored. Additions and subtractions are ignored in many literature, however, the difference between addition/subtraction and multiplication is not significant in many modern microprocessors. The latency of `add` and `sub` instructions is only one clock cycle on Intel Pentium 4 Family 4 processors. However, when long integer addition operation is performed, they are used only when adding or subtracting the least significant digits. The rest of the digits are added or subtracted with slow `adc` (add with carry) and `sbb` (subtract with borrow) instructions, whose latency is 10 clock cycles. These instructions are only 9% faster than `mul` instruction, whose latency is 11 clock cycles [17].

For convenience, we use Intel x86 instructions `mul`, `add` and `adc` to denote the following operations:

- `mul`: single-precision multiplication,
- `add`: addition/subtraction without carry/borrow,
- `adc`: addition/subtraction with carry/borrow.

When multiplying n-digit integer with a single-digit integer, it is clear that n `mul` instructions are required. The numbers of required `add` and `adc` instructions are 1 and $(n-1)$, respectively. When adding i-digit and j-digit integers, the required number of `add` and `adc` instructions are one and $\min(i, j)$, respectively, assuming that carry does not propagate more than one digit place above the most significant digit of the shorter operand. The probability of having carry above the most significant digit place of the shorter integer is $1/2$. The probability that the carry will propagate one more digit place is only $1/b$. Similar argument holds for subtracting two long integers.

Straightforward computation of $\vec{u}_i = -F^{-1} \cdot \vec{x}_i \bmod b$ requires l^2 `mul` and $(l^2 - l)$ `add` instructions. However, exploiting the special structure of F, we can compute $\vec{u}_i$ using only $(2l-1)$ `mul` and $2(l-1)$ `add` instructions, provided that we are allowed to have l-digit pre-computed values that depend on the coefficients of $f(t)$ and the value t.

Theorem 2 *The computation $\vec{u}_i = -F^{-1} \cdot \vec{x}_i \bmod b$ can be performed using only* $(2l-1)$ `mul` *and* $2(l-1)$ `add` *instructions, using l-digit pre-computed values that depend only on the coefficients of $f(t)$ and the value t.*

Proof: Further row operations from (6) easily reveals the exact form of $F' = -F^{-1}$ as follows:

$$F' = \frac{-1}{C_0} \times \begin{bmatrix} C_0 - C_{l-1}t^{l-1} & -C_{l-1}t^{l-2} & \cdots & -C_{l-1} \\ tC_0 - C_{l-2}t^{l-1} & C_0 - C_{l-2}t^{l-2} & \cdots & -C_{l-2} \\ \vdots & \vdots & \ddots & \vdots \\ t^{l-2}C_0 - C_1t^{l-1} & t^{l-3}C_0 - C_1t^{l-2} & \cdots & -C_1 \\ -t^{l-1} & -t^{l-2} & \cdots & -1 \end{bmatrix}, \tag{7}$$

where $C_i = (t^l + \sum_{j=i}^{l-1} f_j t^j)/t^i$. Now, we can express F' as follows,

$$F' = F'_1 - F'_2,$$

where,

$$F'_1 = \begin{bmatrix} \frac{C_{l-1}}{C_0}\vec{v} \\ \frac{C_{l-2}}{C_0}\vec{v} \\ \frac{C_{l-3}}{C_0}\vec{v} \\ \vdots \\ \frac{C_1}{C_0}\vec{v} \\ \frac{1}{C_0}\vec{v} \end{bmatrix}, \qquad \vec{v} = [t^{l-1}, \ldots, t^1, t, 1],$$

$$F'_2 = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ t & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ t^{l-2} & t^{l-3} & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

The matrix-vector product $F'_2 \cdot \vec{x}_i \bmod b$ can be computed using Horner's rule, and it requires only $(l-2)$ single-precision multiplications and $(l-2)$ single-precision additions. The vector product $\vec{v} \cdot \vec{x}_i$ can be computed by multiplying $(t \bmod b)$ to the $(l-1)$-th entry of $F'_2 \cdot \vec{x}_i \bmod b$, and then adding $(f_0 \bmod b)$ to the result. Assuming that $1/C_0 \bmod b$ and $C_i/C_0 \bmod b$ for $i = 1, \ldots, l-1$ are precomputed, computing $F'_1 \cdot \vec{x}_i \bmod b$ requires only l single-precision multiplications. It only remains to compute $F' \cdot \vec{x}_i = F'_1 \cdot \vec{x}_i - F'_2 \cdot \vec{x}_i$ using l single-precision subtractions.

Algorithm 4 explicitly shows how $\vec{u}_i$ is computed using $(2l-1)$ `mul` and $2(l-1)$ `add` instructions. Since $l \geq 2$, Algorithm 4 always performs better than the straightforward matrix-vector product, which requires l^2 `mul` and $(l^2 - l)$ `add` instructions.

We analyze the line 4 of Algorithm 3. Since each row of F contains only one t and f_i, the matrix-vector product $F \cdot \vec{u}_i$ requires l multiplications of t and f_i's by a 1-digit integer, and some additions/subtractions. Let n and k ($\leq n$) be the digit length of t and f_i, respectively and let τ be the number of non-zero f_i's in $f(t)$. Then the number of `mul` required in line 4 is $(ln + \tau k)$.

Algorithm 4 Computing $F' \cdot \vec{x} \bmod b$

```
Require: $f(t) = t^l + f_{l-1}t^{l-1} + \cdots f_1 t + f_0$, $\vec{x} = [x_{l-1}, \ldots, x_1, x_0]$ and pre-computed values $C_i/C_0 \bmod b$ for $i = 1, \ldots, l-1$ and $1/C_0 \bmod b$, where $C_i = (t^l + \sum_{j=i}^{l-1} f_j t^j)/t^i$.
Ensure: $F' \cdot \vec{x}^T = [u_{l-1}, \ldots, u_1, u_0]^T$.
 1: $v_l \leftarrow 0$.
 2: for $i$ from 0 to $l-1$ do
 3:     $v_{l-1-i} \leftarrow v_{l-i} \cdot t + x_{l-1-i} \bmod b$    ($l-1$ mul, $l-1$ add)
 4: end for
 5: for $i$ from 1 to $l-1$ do
 6:     $u_i \leftarrow v_0 \cdot C_i/C_0 \bmod b$.    ($l-1$ mul)
 7: end for
 8: $u_0 \leftarrow v_0/C_0 \bmod b$.    (1 mul)
 9: for $i$ from 1 to $l-1$ do
10:     $u_i \leftarrow u_i - v_i \bmod b$.    ($l-1$ add)
11: end for
12: return $[u_{l-1}, \ldots, u_1, u_0]^T$.
```

If f_i's are small powers of 2 or integers with very small Hamming weight, multiplications by f_i's can be efficiently computed, replacing τk multiplications with τ bit shifts.

We now count the numbers of `add` and `adc` instructions in line 4. There are l multiplications of t with single-digit integers from $\vec{u}_i$, and the total numbers of `add` and `adc` instructions required in this computation are l and $l(n-1)$, respectively. There are τ multiplications of f_i with one digit integer from $\vec{u}_i$, and the total numbers of `add` and `adc` instructions are τ and $\tau(k-1)$, respectively. The matrix vector product $F \cdot \vec{u}_i$ involves $(l-1)$ additions/subtractions of $(n+1)$-digit integer and a single digit integer. This can be computed with $(l-1)$ `add` and `adc` instructions. There are τ additions/subtractions of an $(n+1)$-digit integer with a $(k+1)$-digit integer. Since $k \leq n$ by definition, this computation requires τ `add` and $\tau(k+1)$ `adc` instructions. So far, the numbers of `add` and `adc` instructions in $F \cdot \vec{u}_i$ have been counted. It only remains to add $F \cdot \vec{u}_i$ to $\vec{x}_i$. This computation requires l `add` and $l(n+1)$ `adc` instructions. In total, the numbers of `add` and `adc` instructions required in line 4 are $3l + 2\tau - 1$ and $l(2n+1) + 2\tau k - 1$, respectively.

The total number of `mul`, `add` and `adc` instructions required in Algorithm 3, not considering the final subtraction step, is summarized as follows:

$$\begin{aligned} \#\texttt{mul} &= q(l(n+2) + \tau k - 1), \\ \#\texttt{add} &= q(5l + 2\tau - 3), \\ \#\texttt{adc} &= q(l(2n+1) + 2\tau k - 1). \end{aligned}$$

4.4. Conversions to and from the Montgomery Domain

To perform modular multiplication using Algorithm 3 as a coefficient reduction algorithm, we must transform operands to the Montgomery domain. For $x(t) \in \mathbb{Z}[t]/f(t)$, we compute $\overline{x}(t) \equiv x(t) \cdot b^q \pmod{p}$. This computation can be easily achieved by multiplying two polynomials $x(t)$ and $y(t) \equiv b^{2q} \bmod p$, and then reduce coefficients using Algorithm 3. The result will be $\overline{x}(t) \equiv x(t) \cdot b^q \bmod p$, as desired. It is convenient to have $y(t) \equiv b^{2q} \bmod p$ pre-computed for each p. Conversion from the Montgomery domain can be performed by directly applying Algorithm 3 on $\overline{x}(t)$. The result is $\overline{x}(t) \cdot b^{-q} \equiv x(t) \cdot b^q \cdot b^{-q} \equiv x(t) \pmod{p}$, as desired.

4.5. Interesting Implementation Options

We consider some special cases for which Algorithm 3 can speed up.

- Special Case I: $t \equiv 0 \pmod{b}$.
  In such a case, it can be shown that

$$F' = -F^{-1}$$
$$= \begin{bmatrix} -1 & 0 & 0 & \cdots & 0 & f_{l-1}/f_0 \\ 0 & -1 & 0 & \cdots & 0 & f_{l-2}/f_0 \\ 0 & 0 & -1 & \cdots & 0 & f_{l-3}/f_0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & f_1/f_0 \\ 0 & 0 & 0 & \cdots & 0 & 1/f_0 \end{bmatrix} \bmod b.$$

  Note that F is invertible if and only if $f_0 \neq 0$ and $\gcd(f_0, b) = 1$. Then, $\vec{u}_i = F' \cdot \vec{x}_i \bmod b$ can be computed with only τ `mul` and $(l-1)$ `add` instructions, provided that $f_i/f_0 \bmod b$ for $i = 1, \ldots, l-1$ and $1/f_0 \bmod b$ are pre-computed. Hence, compared to the general case, we save $(2l - \tau - 1)$ `mul` and $(l-1)$ `add` instructions in line 3 of Algorithm 3.

  When computing $F \cdot \vec{u}_i$, we can save l `mul` instructions and one `adc` instruction, since the least significant digit of t is zero. The total number of saved instructions is given as follows:

$$\#\texttt{mul}_{save} = q(3l - \tau - 1),$$
$$\#\texttt{add}_{save} = q(l-1),$$
$$\#\texttt{adc}_{save} = q.$$

- Special Case II: f_i's are powers of 2.
  In such a case, multiplication by f_i's can be simply performed by bit shifts. In line 3, there is no speed up. The number of instructions we can save in line 4 is given below.

$$\#\texttt{mul}_{save} = q\tau k,$$
$$\#\texttt{add}_{save} = q\tau,$$
$$\#\texttt{adc}_{save} = q\tau(k-1).$$

  Note that Algorithm 3 requires ql bit shift instructions in exchange for the above saved instructions.

4.6. Modular Multiplication Stability

Under some conditions with a $q \geq n$, it is possible to avoid the final subtractions using an approach similar to [15]. Due to space limitation, we only give the result here.

Theorem 3 (Stability) *Algorithm 3 does not require a final subtraction when q is chosen such that the following condition is satisfied:*

$$\lambda < \frac{b^q}{4 \cdot 2^{n'}}, \tag{8}$$

where n' is the bit length of $(t + \xi)$. In such a case, the magnitude of input and output coefficients will be bounded by $\Psi = \lfloor b^q/(4\lambda) \rceil$.

4.7. A Numerical Example

We show how a modular multiplication can be performed using Algortihm 3 on a 16-bit processor using an LWPFI modulus $p = f(t) = t^3 - 2t + 2$. Note that $f(t)$is an irreducible polynomial.

First, we randomly choose 16-bit integer t until p becomes a prime integer.

$$t = \texttt{0x9261},$$
$$p = \texttt{0x2FDBAD3AFE61}.$$

The modulus p is a 46-bit integer. Then we precompute values $1/C_0 \bmod b$ and $C_i/C_0 \bmod b$ for $0 < i < l$, where $b = 2^{16}$ as follows:

$$C_2/C_0 \bmod b = \texttt{0x9261},$$
$$C_1/C_0 \bmod b = \texttt{0xC8BF},$$
$$1/C_0 = \texttt{0xFE61}.$$

Since $\xi = 2$, $\lambda = (3^3 - 1)/2 = 13$. The number of iterations q in Algorithm 3 must be at least $\lceil \log_b(4 \cdot 13 \cdot 2^{n'}) \rceil$, where n' is the bit length of $t + \xi$. In our example, $n' = 16$. Therefore,

$$q = \lceil \log_b(4 \cdot 13 \cdot 2^{n'}) \rceil = 2.$$

Suppose the two input $A(t)$ and $B(t)$ are given as follows:

$$\begin{aligned} A(t) &= \texttt{0x7CE0}t^2 + \texttt{0xDF53}t + \texttt{0x481A}, \\ &= \texttt{0x28D4555F616D}, \\ B(t) &= \texttt{0x1233}t^2 + \texttt{0x6F85}t + \texttt{0xDB60}, \\ &= \texttt{0x5F37E808738}. \end{aligned}$$

Then,

$$\begin{aligned} C(t) &= A(t) \cdot B(t) \bmod f(t) = c_2t^2 + c_1t + c_0 \\ &= \texttt{0xE32CE98D}t^2 + \texttt{0x15993E234}t \\ &- \texttt{0x4EC36012}, \\ &= \texttt{0x4A46FCB26FA338EF} \\ &\equiv \texttt{0x2C8237C4F3E} \pmod p \end{aligned}$$

We construct a vector $\vec{x_0}$ using the coefficients of $C(t)$ as follows:

$$\vec{x}_0 = (c_2, c_1, c_0)^T.$$

Now we compute $\vec{u} = F' \cdot \vec{x}_0 \bmod b$ in line 3 of Algorithm 3.

$$\vec{u} = F' \cdot \vec{x}_0 \bmod b = (\texttt{0xFB62}, \texttt{0x8F50}, \texttt{0x594F})^T.$$

The computation $\vec{x}_0 + F \cdot \vec{u}$ in line 4 of Algorithm 3 is performed as follows:

$$\vec{x}_0 + F \cdot \vec{u} = \begin{bmatrix} \texttt{0xB01D0000} \\ \texttt{0xC9D80000} \\ -\texttt{0xA0B60000} \end{bmatrix}.$$

Since all elements of $\vec{x}_0$ are divisible by b, we can perform $\vec{x}_1 \leftarrow (\vec{x}_0 + F \cdot \vec{u})/b$ without using a division algorithm.

$$\vec{x}_1 = (\texttt{0xB01D}, \texttt{0xC9D8}, -\texttt{0xA0B6})^T.$$

This completes the first iteration of Algorithm 3.

In the second iteration,

$$\vec{u} = F' \cdot \vec{x}_1 \bmod b = (\texttt{0xB3E2}, \texttt{0x9E0C}, \texttt{0x525F})^T.$$

Then $\vec{x}_2$ is computed as,

$$\vec{x}_2 = (-\texttt{0x2F18}, -\texttt{0x66D9}, -\texttt{0x5A60})^T.$$

Therefore,

$$\begin{aligned} C(t) \cdot b^{-2} &\equiv A(t) \cdot B(t) \cdot b^{-2} \equiv -\texttt{0x2F18}t^2 \\ &- \texttt{0x66D9}t - \texttt{0x5A60}, \\ &\equiv -\texttt{0xF65E66D55B1} \pmod p \\ &\equiv \texttt{0x2C8237C4F3E} \cdot b^{-2} \pmod p. \end{aligned}$$

5. Comparisons and Experimental Results

In this section, we compare our new coefficient reduction algorithm with Algorithm 2 and show our experimental results. For fairness of comparison, we let the modulus m used in Algorithm 2 be an nl-digit integer. Note that a degree-l polynomial $f(t)$ with n-digit t generates an nl-digit LWPFI.

5.1. Comparisons

We use the same technique that we use in Section 4.3 to analyze Algorithm 2. In line 3, only one `mul` is required. In line 4, the computation of $u_i \cdot m$ requires nl `mul`, 1 `add` and $(nl-1)$ `adc` instructions. Adding $u_i \cdot m$, which is at most $(nl+1)$ digits long, to T_i requires 1 `add` and $(nl+1)$ `adc` instructions. Therefore, Algorithm 2 requires the following number of instructions, not considering the final subtraction:

$$\begin{aligned} \#\texttt{mul} &= q(nl+1), \\ \#\texttt{add} &= 2q, \\ \#\texttt{adc} &= 2qnl. \end{aligned}$$

Note that q in Algorithm 2 is not the same as the one used in Algorithm 3. Final subtractions in Algorithm 2 can be avoided in the case $b \geq 4$ by simply letting $q = nl+1$. In Algorithm 3, we suppose $q = n+1$ eliminates the necessity of final subtractions. Note that such a value for q can be chosen if ξ is reasonably small.

In Table 1, we observe that Algorithm 2 requires $O(n^2l^2)$ operations, whereas Algorithm 3 requires $O(ln^2)$ operations. Hence, Algorithm 3 does have better asymptotic behavior than Algorithm 2. However, this does not mean that Algorithm 3 is always faster than Algorithm 2. If actual values for parameters n, l, τ and k are substituted in Table 1, the required number of operations for Algorithm 3 may be larger. However, the larger n and l, the better Algorithm 3 will perform and eventually outperform Algorithm 2.

5.2. Experimental Results

We have implemented the modular multiplication algorithm using LWPFI moduli and the new coefficient reduction algorithm presented in this paper using C programming language. We used GNU Multiple Precision library (GMP) to perform long integer arithmetic. We used `gcc` 4.1.2 to compile all programs including GMP. When compiling GMP, we used `--disable-shared` option to prevent the overhead due to the runtime address resolution and `--build=none` option to disable assembly routines

Table 1. Comparison of Algorithm 2 and Algorithm 3

Instruction	Algorithm 2	Algorithm 3
`mul`	$(nl+1)^2$	$ln^2 + (\tau k + 3l - 1)n + 2l + \tau k - 1$
`add`	$2(nl+1)$	$(n+1)(5l + 2\tau - 3)$
`adc`	$2n^2l^2 + 2nl$	$2ln^2 + (2\tau k + 3l - 1)n + 2\tau k + l - 1$

in GMP. We ran GMP's tuneup program to maximize the performance of multiplication routines. Our LWPFI modular multiplication algorithms have been implemented using `mpz_*` and `mpn_*` functions provided in GMP. Experiements were performed in Linux running on Intel Pentium 4 3.20 GHz (Family 7, Model 4).

Figures 1(a) and 1(b) each shows the timing results of modular multiplication using the coefficient reduction algorithm in [1] and Algorithm 3. In each figure, we have also plotted the timing results of long integer modular multiplication using Algorithm 2 to simply show that modular multiplication using LWPFI is indeed asymptotically faster. It is clearly seen from the figures that our new coefficient reduction algorithm is more efficient than that in [1].

6. Applications of LWPFI Modular Multiplications

Many cryptosystems rely on the ability to perform modular arithmetic modulo large integers. Among the modular arithmetic, modular multiplication is the most frequently used operation. In most cases, the modulus has to be a prime number. One can randomly try t until $f(t)$ is a prime to use it in cryptosystems requiring modular multiplications. One may find t such that $f(t)$ has a large enough prime factor suitable for certain cryptosystems. We denote such a factor p'. In this case, we can embed any modular arithmetic modulo p' in slightly larger ring $\mathbb{Z}_{f(t)}$, where we can use efficient modular multiplications using LWPFI moduli. Note, however, this method is faster only if the modular multiplication using LWPFI is faster than the modular multiplication modulo p' using usual integer arithmetic. After all computations have been performed, the result must be converted to the usual representation of integers and be taken modulo p'.

The idea of embedding arithmetic into a larger ring, where computations are easy, is not at all new. A similar technique is used for efficient multiplication in finite fields [18] [19].

7. Conclusions

In this paper, we have extended LWPFIs presented in [1], and have proposed a new coefficient reduction reduction based on Algorithm 2. Our new coefficient reduction algorithm have been analyzed using the extended definition of LWPFIs. Performance have been thoroughly analyzed. A condition on parameters for eliminationg the final subtractions in the new coefficient reduction algorithm has been given. We have presented experimental results to show that our new coefficient reduction algorithm based on Montgomery reduction is indeed more efficient than the one presented in [1].

References

[1] J. Chung and M. A. Hasan, "Low-weight polynomial form integers for efficient modular multiplication," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 44–57, 2007.

[2] J. Chung and A. Hasan, "More generalized Mersenne numbers," in *Selected Areas in Cryptography - SAC 2003*, LNCS 3006, pp. 335–347, Springer-Verlag, 2003.

[3] J.-C. Bajard, L. Imbert, and T. Plantard, "Modular number systems: Beyond the Mersenne family," in *Selected Areas in Cryptography 2004*, LNCS 3357, pp. 159–169, Springer-Verlag, 2004.

[4] J.-C. Bajard, L. Imbert, and T. Plantard, "Arithmetic operations in the polynomial modular number system," in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ARITH'05, pp. 206–213, 2005.

[5] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady (English translation)*, vol. 7, no. 7, pp. 595–596, 1963.

[6] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Math*, vol. 3, pp. 714–716, 1963.

(a) $p = t^2 + 1$

(b) $p = t^3 - t + 1$

Figure 1. Timing of LWPFI modular multiplication

[7] S. A. Cook, *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, May 1966.

[8] P. L. Montgomery, "Five, six, and seven-term Karatsuba-like formulae," *IEEE Transaction on Computers*, vol. 54, no. 3, pp. 362–369, 2005.

[9] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[10] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology - CRYPTO '96*, LNCS 1109, pp. 104–113, Springer-Verlag, 1996.

[11] W. Schindler, "A timing attack against RSA with the Chinese remainder theorem," in *Cryptographic Hardware and Embedded Systems - CHES 2000*, LNCS 1965, pp. 109–124, Springer-Verlag, 2000.

[12] C. D. Walter and S. Thompson, "Distinguishing exponent digits by observing modular subtractions," in *Progress in Cryptology - CT-RSA 2001*, LNCS 2020, pp. 192–207, Springer-Verlag, 2001.

[13] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, 1999.

[14] G. Hachez and J.-J. Quisquater, "Montgomery exponentiation with no final subtractions: Improved results," in *Cryptographic Hardware and Embedded Systems - CHES 2000*, LNCS 1965, pp. 293–301, Springer-Verlag, 2000.

[15] C. D. Walter, "Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli," in *Topics in Cryptology - CT-RSA 2002*, LNCS 2271, pp. 30–39, Springer-Verlag, 2002.

[16] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger, "Closest point search in lattices," *IEEE Transactions on Information Theory*, vol. 48, pp. 2201–2214, August 2002.

[17] T. Granlund, "Instruction latencies and throughput for AMD and Intel x86 processors," 2005. Available at `http://swox.com/doc/x86-timing.pdf`.

[18] H. Wu, M. A. Hasan, I. F. Blake, and S. Gao, "Finite field multiplier using redundant representation," *IEEE Transactions on Computers*, vol. 51, no. 11, pp. 1306– 1316, 2002.

[19] G. Drolet, "A new representation of elements of finite fields $GF(2^m)$ yielding small complexity arithmetic circuits," *IEEE Transactions on Computers*, vol. 47, no. 9, 1998.