Optimized Modular Multiplication for Supersingular Isogeny Diffie-Hellman

Weiqiang Liu, *Senior Member, IEEE*, Jian Ni, Zhe Liu, *Senior Member, IEEE*, Chunyang Liu, and Máire O'Neill, *Senior Member, IEEE*

Abstract—Recent progress in quantum physics shows that quantum computers may be a reality in the not too distant future. Post-quantum cryptography (PQC) refers to cryptographic schemes that are based on hard problems which are believed to be resistant to attacks from quantum computers. The supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol shows promising security properties among various post-quantum cryptosystems that have been proposed. In this paper, we propose two efficient modular multiplication algorithms with special primes that can be used in SIDH key exchange protocol. Hardware architectures for the two proposed algorithms are also proposed. The hardware implementations are provided and compared with the original modular multiplication algorithm. The results show that the proposed finite field multiplier is over 6.79 times faster than the original multiplier in hardware. Moreover, the SIDH hardware/software codesign implementation using the proposed FFM2 hardware is over 31 percent faster than the best SIDH software implementation.

Index Terms—Post-quantum cryptography, supersingular isogeny Diffie-Hellman (SIDH), modular multiplication

1 Introduction

The computing capability of quantum computers is significantly higher than classical computers. It is found that a 30-qubit quantum computer would have the same processing power as a conventional computer processing commands at 10 teraflops per second [1]. In 1994, Shor [2] proposed an algorithm that can be used to quickly factorise large numbers, which shows exponential speedup of the computation. Later in 1996, Grover's algorithm [3] is proposed to search an unsorted database with quadratic speedup over a conventional computer(in $\mathcal{O}(\frac{N}{2})$ time rather than $\mathcal{O}(N)$). Recent progress in the design and development of quantum computers shows that real quantum computers may be available in the not too distant future [4].

As a result, commonly used public-key cryptographic algorithms, such as RSA [5] and Elliptic curve cryptography (ECC) [6], which rely on integer factorization and the discrete log problem that are used in all of today's communications and internet security will be vulnerable to attacks from quantum computers. Post-quantum cryptographic (PQC) [7], or quantum-safe schemes, which refer to conventional non-quantum cryptographic algorithms that are secure today but should remain secure even after practical quantum computing is a reality, have been shown to be just as practical as classical RSA and ECC schemes [8]. Recently, NIST [9] and ETSI [10] have held workshops to discuss the importance of quantum-safe cryptography. NIST are currently hosting a standardisation process to select new post-quantum cryptographic signature and encryption schemes [11].

Much research is now being conducted into PQC. Among the various post-quantum techniques, the latest supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol scheme [12] shows promising security properties. The SIDH key exchange scheme offers significantly smaller key sizes than other post quantum key exchange and encryption counterparts such as the commonly cited lattice-based [13], [14], code-based [15], hash-based [16] and multivariate quadratic [17] cryptography. As SIDH is more than a decade younger than the other types of PQC schemes, little research has been conducted into evaluating its practicality. The supersingular isogeny key encapsulation (SIKE) protocol which is based on SIDH has been submitted to the NIST post-quantum cryptography process in November 2017 [18].

In 2011, Jao and Feo presented software implementation results showing that their proposed SIDH key-exchange protocols are over two orders of magnitude faster than classical isogeny-based cryptosystems over ordinary curves. Later Azarderakhs et al. implemented the same SIDH protocol on PC (x86-64) and ARM (ARMv7) platforms [19]. Their implementation is between 18-26 percent faster depending on the security level. In 2016, Costello et al. presented a high-speed implementation of SIDH, which is more than 2.5 times faster than the previous SIDH software results [20]. The first hardware implementation of SIDH was recently proposed, targeting a Virtex-7 Field Programmable Gate Array (FPGA) [21], and is 1.5 times faster than the best software implementation for the 512-bit SIDH scheme.

In the recent studies [22], [23], it is found that the Montgomery reduction for the primes of special structure used in isogeny based cryptography is not optimal. There are special moduli can be used for faster implementations. In SIDH, the prime, p, is in the form, $p = f \cdot 2^a 3^b - 1$, where f is a small number. Similar to other public key cryptosystems, the modular multiplication plays a very important role. In [24], Karmakar et al. proposed an efficient finite field multiplication (EFFM) algorithm, in which the prime field has a special structure.

In this paper, we improve upon EFFM [24] and propose two new algorithms. The first algorithm proposed, referred to as the improved EFFM (FFM1), improved upon the original EFFM by reducing the number of operands. The second new finite field multiplication (FFM2) algorithm proposed is very different from the original EFFM and the FFM1, and allows for larger operand sizes while reducing the number of operations. Both proposed algorithms speed up the computation significantly. Hardware architectures for both proposed algorithms are also proposed. The hardware implementation results are provided and compared with implementations of original EFFM algorithm. This paper is an extension of previous research by the authors in [25]. It improves on [25] as follows:

1) A more detailed description of the two proposed algorithms is given with a brief review of Barrett Division;
2) A new modular multiplication algorithm (FFM2) is proposed;
3) A hardware architecture for the FFM2 algorithm is proposed;
4) The hardware results of the FFM2 algorithm are provided and compared with original EFFM algorithm and the FFM1 algorithm in [25]. The hardware results show that the FFM2 algorithm is the fastest;
5) A complete SIDH hardware/software (HW/SW) codesign result using the proposed FFM2 is provided and compared with the best SIDH software implementation.

- *W. Liu, J. Ni, and C. Liu are with the College of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210016, China. E-mail: {liuweiqiang, nijian}@nuaa.edu.cn, lcygzyj@163.com.*
- *Z. Liu is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210016, China. He is also with State Key Laboratory of Cryptology, P.O. Box 5159, Beijing, 100878, China. E-mail: sduliuzhe@gmail.com.*
- *M. O'Neill is with the Centre for Secure Information Technologies, Queens University Belfast, Belfast BT7 1NN, United Kingdom. E-mail: m.oneill@ecit.qub.ac.uk.*

Manuscript received 1 July 2018; revised 30 Jan. 2019; accepted 6 Feb. 2019. Date of publication 20 Feb. 2019; date of current version 17 July 2019.
(Corresponding author: Weiqiang Liu and Zhe Liu.)
Recommended for acceptance by J. D. Bruguera.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2019.2899847

The rest of the paper is organized as follows. Section 2 reviews the original EFFM algorithm with the special primes, the Barrett reduction algorithm and the Barrett Division algorithm. The two proposed algorithms are presented in details in Sections 3 and 4, respectively, where hardware architectures are also provided. Section 5 presents hardware implementation results for both the proposed modular multiplication algorithms, and provides a comparison with the previously proposed EFFM algorithm. A SIDH HW/SW codesign using the proposed FFM2 is also provided in this section. Section 6 concludes the paper.

2 Review

2.1 Efficient Finite Field Multiplication (EFFM) Algorithm for SIDH

One of the main computational bottlenecks in SIDH is computing arithmetic modulo $p = f \cdot 2^a 3^b - 1$, where the value of f is fixed to 2 and b is even. a and b for the prime p are chosen in such a way that $2^a \approx 3^b$. For instance, the prime $p = 2 \cdot 2^{386} 3^{242} - 1$ is 771 bits in [12] to guarantee 128-bit post-quantum security. [24] proposes an algorithm using a special structure of the prime to optimize the modular multiplication and reduction. This algorithm is briefly reviewed in this section.

Assume that $p = 2 \cdot 2^a 3^b - 1$, where a and b are even. By using a radix $R = 2^{a/2} 3^{b/2}$, a field element $A \in F_p$ can be represented as follows:

$$A = a_1 \cdot R^2 + a_2 \cdot R + a_3, a_1 \in \{0, 1\}, a_2, a_3 \in [0, R). \tag{1}$$

The field elements are converted once at the start and once at the end of the algorithm in the above representation from the radix R. Suppose there are two numbers A and B in the representation of Eq. (1). By multiplying them we can get the product C as follows:

$$\begin{aligned} C &= a_1 b_1 \cdot R^4 + (a_1 b_2 + a_2 b_1) \cdot R^3 \\ &\quad + (a_1 b_3 + a_2 b_2 + a_3 b_1) \cdot R^2 \\ &\quad + (a_2 b_3 + a_3 b_2) \cdot R + a_3 b_3. \end{aligned} \tag{2}$$

As $2^a 3^b = 2^{-1} (mod\ p)$, R^2 and R^4 can be replaced by $2^{-1} (mod\ p)$ and $2^{-2} (mod\ p)$, respectively, or 0, which can be precomputed for a fixed prime. Therefore, we can get Eq. (3), where 4 multiplication operations are required: $a_2 b_2$, $a_2 b_3$, $a_3 b_2$ and $a_3 b_3$. The other products multiplied by either a_1 or b_1 can be computed by simply selecting the correct result. Therefore, C can be rewritten as $C = c_1 \cdot R^2 + c_2 \cdot R + c_3$. As c_2 and c_3 are in the range of $[0, R^2)$, they need to be further reduced by improved Barrett reduction.

$$\begin{aligned} C &= (a_1 b_3 + a_2 b_2 + a_3 b_1)(mod\ 2) \cdot R^2 \\ &\quad + (\lfloor (a_1 b_2 + a_2 b_1)/2 \rfloor + (a_2 b_3 + a_3 b_2)) \cdot R \\ &\quad + (2^{-2} (mod\ p) a_1 b_1 + ((a_1 b_2 + a_2 b_1)(mod\ 2)) \cdot \tfrac{R}{2} \cdot \\ &\quad + \lfloor (a_1 b_3 + a_3 b_1 + a_2 b_2 / 2) \rfloor + a_3 b_3) \end{aligned} \tag{3}$$

The original EFFM algorithm is shown in Algorithm 1, where the Barrett Division is the improved Barrett reduction used in [24].

Algorithm 1. The EFFM Algorithm [24]

```
Input: $A, B \in F_p$, $A = a_1 \cdot R^2 + a_2 \cdot R + a_3$ and $B = b_1 \cdot R^2 + b_2 \cdot R + b_3$, $2^{-2}(mod\ p)$ is precalculated.
Output: $C = A \cdot B(mod\ p) = c_1 \cdot R^2 + c_2 \cdot R + c_3$, $(R = 2^{a/2} 3^{b/2})$.
$c_1 = 0, c_2 = 0, c_3 = 0$
$c_3 = a_1 b_1 \cdot 2^{-2}(mod\ p) + a_3 b_3$
$c_2 = a_2 b_3 + a_3 b_2$
$t = a_1 b_2 + a_2 b_1, c_2 = c_2 + \lfloor t/2 \rfloor, c_3 = c_3 + t[0] \cdot \frac{R}{2}$;
$t = a_1 b_3 + a_2 b_2 + a_3 b_1, c_3 = c_3 + \lfloor t/2 \rfloor, c_1 = t[0]$;
Barrett Division$(c_3) \Rightarrow c_3 = r, c_2 = c_2 + r$;
Barrett Division$(c_2) \Rightarrow c_2 = r, c_1 = c_1 + r$;
$c_3 = c_3 + \lfloor c_1/2 \rfloor, c_1 = c_1[0]$;
```

2.2 Barrett Reduction

According to Euclids division lemma, it is known that there exists q and r such that $a = q \cdot b + r, r \in [0, b)$ for any two positive integers a and b. Therefore, we have $a = r(mod\ b)$. In order to get such a q and r, one division is required. However, division is a very expensive operation, which is more complex and much slower than multiplication. Thus, to speed up the division, it can be converted to a multiplication by Barrett reduction [26], i.e., $\times 1/b$. Furthermore, $1/b$ can also be expressed as follows:

$$\frac{1}{b} = \frac{2^k/b}{b \cdot 2^k/b} = \frac{2^k/b}{2^k} \approx \frac{x}{2^k}. \tag{4}$$

Generally, the value x is taken as $x = \lfloor 2^k/b \rfloor$. However, an error (denoted as e) is produced from the approximation of $1/b$, which equals $1/b - x/2^k$. To make sure the final result is correct, q must be smaller than 1. This condition can be met when $k = log_2\ a$. The whole process is shown in Algorithm 2.

Algorithm 2. The Barrett Reduction Algorithm [26]

```
Input: Two numbers $a$ and $b$, parameter $k$, $x = \lfloor 2^k/b \rfloor$
Output: $a(mod\ b)$
$q = (a \times x) \gg k$
$r = a - q \times b$
if $r \geqslant b$ then
    $r = r - b$
end
return $r$;
```

2.3 Barrett Division

An efficient division algorithm was proposed in [24]. The algorithm can divide a number $c_i \in [0, 2^a 3^b)$ by $2^{a/2} 3^{b/2}$ and calculate the quotient q and remainder r in an efficient way. As division by two is a simple right shift operation, the division can be performed by $2^{a/2} 3^{b/2}$ according to the following steps:

Step1: Extract the $a/2$ least significant bits of c_i and store them in a variable r_1;

Step2: Right shift c_i by $a/2$ bits to obtain $c_i^{'}$;

Step3 Divide $c_i^{'}$ by $3^{b/2}$ to get the quotient q and remainder r_2.

Therefore, c_i can be rewritten as follows:

$$c_i = q \cdot 2^{a/2} 3^{b/2} + (r_2 2^{a/2} + r_1) = q \cdot 2^{a/2} 3^{b/2} + r. \tag{5}$$

The division by $3^{b/2}$ in Step 3 is more complex than the division by $2^{a/2}$. As b is a fixed integer, it is possible to speed up the division by performing multiplication similar to the Barrett reduction technique, as shown in Algorithm 3. Therefore, this efficient division algorithm is referred to as the Barrett Division in this paper.

Once the quotients and remainders are obtained, it is easy to represent $c = c_1 \cdot 2^a 3^b + c_2 \cdot 2^{a/2} 3^{b/2} + c_3$ with desired finite field element.

Algorithm 3. The Barrett Division Algorithm [24]

```
Input: 2 numbers $Q \in [0, 2^a 3^b)$ and $P = 2^{a/2} 3^{b/2}$ and $log_2 P \approx 2 log_2 Q$. $P^{'} = P/2^{a/2}$ precomputed $x = 2^k/P^{'}$, $k$ is as described in Section 2.2
Output: $q$ and $r$ such that $Q = q \cdot P + r$
$t = \lfloor Q/2^{a/2} \rfloor, s = Q(mod\ p)$;
$q = t \times x \gg k$;
$r = t - P^{'} \times q$;
$r = r \times 2^{a/2} + s$;
if $r > P$ then
$r = r - P$;
$q = q - 1$;
end
return $q, r$
```

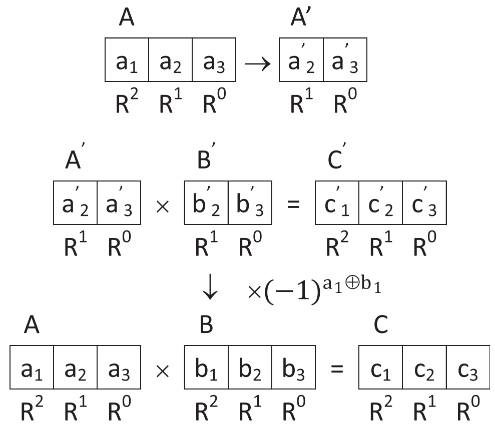
A
A'
a1 a2 a3 → a'2 a'3
R2 R1 R0 R1 R0
A' B' C'
a'2 a'3 × b'2 b'3 = c'1 c'2 c'3
R1 R0 R1 R0 R2 R1 R0
↓ ×(−1)^(a1⊕b1)
A B C
a1 a2 a3 × b1 b2 b3 = c1 c2 c3
R2 R1 R0 R2 R1 R0 R2 R1 R0

Fig. 1. The transforming process of the operands, where the radix is: $R = 2^{a/2}3^{b/2}$.

3 The FFM1 Algorithm and Its Hardware Architecture

Improving upon [24], it is found that the modular multiplication with the special field can be further simplified based on the fact that:

$$(p - A)(p - B) = A \cdot B(mod\ p). \tag{6}$$

The improvement is further explained in detail in the following subsections.

3.1 The FFM1 Algorithm

Assume $A \in F_p$, which is in the form of Eq. (1). Then we get a number $A' \in F_p$ as follows:

$$\begin{cases} A' = A, & a_1 = 0 \\ A' = p - A, & a_1 = 1 \end{cases}. \tag{7}$$

When $a_1 = 1$, A can be written as $A = R^2 + a_2R + a_3$ with a radix $R = 2^{a/2}3^{b/2}$, and p can be written as $p = R^2 + (R-1)\cdot R + (R-1)$, so we have:

$$a_i^{'} = R - 1 - a_i, i \in \{2, 3\}. \tag{8}$$

Therefore, A' can be represented in the following form:

$$A' = a_2^{'} \cdot R + a_3^{'},\ a_2^{'}, a_3^{'} \in [0, R). \tag{9}$$

Suppose two numbers A and B are in the form of Eq. (1). A' and B' are in the form of Eq. (7). Due to the fact that Eq. (6) holds, the following relationship between $A \cdot B(mod\ p)$ and $A' \cdot B'(mod\ p)$ can be obtained as follows:

$$A \cdot B(mod\ p) = A' \cdot B'(mod\ p) \cdot (-1)^{a_1 \oplus b_1}. \tag{10}$$

The transforming process of the operands is shown in the Fig. 1.

According to Eq. (10), we can get the result of $A \cdot B(mod\ p)$ by computing $A' \cdot B'(mod\ p)$, which saves about 5 additions. The product of A' and B' can be expressed as follows:

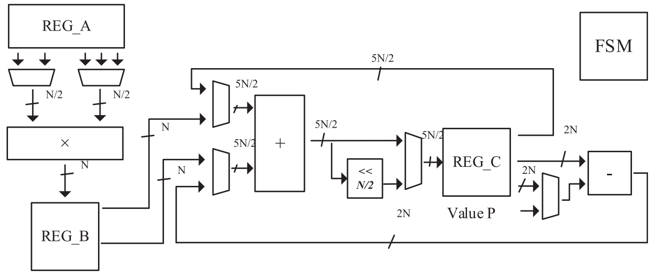

Fig. 2. The proposed hardware architecture for FFM1.

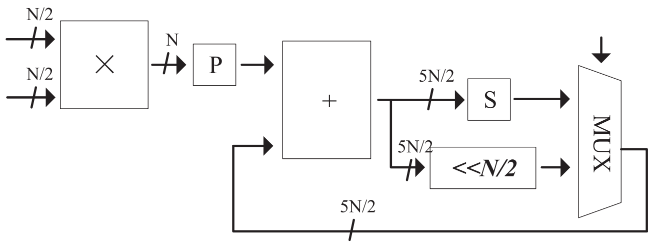

Fig. 3. The pipeline structure of a full multiplication.

$$A' \cdot B' = a_2^{'}b_2^{'} \cdot R^2 + (a_2^{'}b_3^{'} + a_3^{'}b_2^{'}) \cdot R + a_3^{'}b_3^{'}. \tag{11}$$

$$\begin{aligned} A' \cdot B' &= (a_2^{'}b_2^{'}(mod\ 2)) \cdot R^2 + (a_2^{'}b_3^{'} + a_3^{'}b_2^{'}) \cdot R \\ &\quad + (a_3^{'}b_3^{'} + \lfloor a_2^{'}b_2^{'}/2 \rfloor) \\ &= c_1^{'} \cdot R^2 + c_2^{'} \cdot R + c_3^{'}. \end{aligned} \tag{12}$$

Rewriting Eq. (11) by replacing the coefficients, we can get Eq. (12). As $c_2^{'}$ and $c_3^{'}$ can be larger than R, they need to be further reduced by the Barrett reduction as described in Section 2.2. The proposed algorithm is presented in Algorithm 4.

Algorithm 4. The FFM1 Algorithm

```
Input: $A', B' \in F_p, A = a_2^{'} \cdot R + a_3^{'}$ and $B = b_2^{'} \cdot R + b_3^{'}$
Output: $C' = A' \cdot B'(mod\ p) = c_1^{'} \cdot R^2 + c_2^{'} \cdot R + c_3^{'}$,
        $(R = 2^{a/2}3^{b/2})$.
$c_1^{'} = 0, c_2^{'} = 0, c_3^{'} = 0$;
$c_1^{'} = a_2^{'}b_2^{'}[0]$;
$c_3^{'} = a_3^{'}b_3^{'} + \lfloor a_2^{'}b_2^{'}/2 \rfloor$;
Barrett Division$(c_3^{'}) \Rightarrow c_3^{'} = r, c_2^{'} = c_2^{'} + r$;
Barrett Division$(c_2^{'}) \Rightarrow c_2^{'} = r, c_1^{'} = c_1^{'} + r$;
$c_3^{'} = c_3^{'} + \lfloor c_1^{'}/2 \rfloor, c_1^{'} = c_1^{'}[0]$;
```

The difference between the original EFFM [24] and the FFM1 is as follows. The two algorithms both need to compute 4 multiplications: $a_2 \times b_2, a_2 \times b_3, a_3 \times b_2$ and $a_3 \times b_3$ in the EFFM and $a_2^{'} \times b_2^{'}, a_2^{'} \times b_3^{'}, a_3^{'} \times b_2^{'}$ and $a_3^{'} \times b_3^{'}$ in the proposed algorithm. However, the EFFM has 5 multiplication terms to get $a_1 \times b_1 \times 2^{-2}(mod\ p), a_1 \times b_2, a_1 \times b_3, b_1 \times a_2$ and $b_1 \times a_3$; while our improved one can get $a_2^{'}, a_3^{'}, b_2^{'}$ and $b_3^{'}$ by at most 4 subtractions. Before the Barrett reduction, the EFFM algorithm has 6 to 9 additions/subtractions, 2 right-shifts to calculate the coefficients c_1, c_2 and c_3; while the proposed algorithm only needs 2 additions and 1 right-shift to get $c_1^{'}, c_2^{'}$ and $c_3^{'}$. The EFFM needs to precalculate and store $2^{-2}(mod\ p)$ using 21 registers while the proposed algorithm only needs 16 registers. Most importantly, the terms with R^2 have been removed in the proposed algorithm, which saves on the number of operations significantly. A detailed comparison with the hardware implementations is further discussed in Section 5.

3.2 The Proposed Hardware Architecture for FFM1

We also propose a hardware architecture of the improved modular multiplication with the special field, which is shown in Fig. 2. In this architecture, there is one N/2-bit multiplier, one 5N/2-bit adder and one 2N-bit subtractor. An adder and subtraction that support large word lengths are used to reduce the number of clock cycles. The whole modular multiplication process is controlled by a finite state machine (FSM).

The operands are stored in pipeline registers that are inserted between arithmetic units to further increase the performance. The inputs of the multiplier, adder and subtractor are selected by MUXs, which are controlled by the FSM.

The process of a full binary multiplication is shown in Fig. 3. Two N/2-bit operands are multiplied by the N/2-bit multiplier, and then partial products are accumulated by the 5N/2-bit adder.

There are three kinds of multiplications in the algorithm, which have different input sizes, namely, $N \times N/2$, $N \times N$, and $3N/2 \times 3N/2$. In the $3N/2 \times 3N/2$ multiplication, one of the inputs is a constant, whose most significant N/2 digits always equal to 2 and remain unchanged during the process. Therefore, the $3N/2 \times 3N/2$ multiplication is performed by a $3N/2 \times N$ multiplication and a shift operation. This can be performed by the circuit as shown in Fig. 3. We can get the most significant N/2 digits of the final result in the second clock cycle. On the next clock cycle, the lower N/2 digits can be obtained subsequently. Depending on the weights of the partial products, the summation is shifted.

For example, using only 5 clock cycles, we can get the product of an $N \times N$ multiplication. For the 4 $N \times N$ multiplications at the start of the algorithm, it only takes 17 clock cycles. Otherwise, if the multiplication is not pipelined, 20 clock cycles are required.

4 The FFM2 Algorithm and Its Hardware Architecture

As we mentioned in Section 2, the modulo $p = f \cdot 2^a 3^b - 1$ is chosen in that: (a) f is a small number, such as 1 or 2; (b) $2^a \approx 3^b$. In fact, the number of p that satisfy the above conditions is finite. In addition, the original algorithm in [24] provides rules on how to choose appropriate parameters so that the p is suitable for SIDH. It fixes f to 2 and makes sure that b must be even. For example, the modulo used in the [24] is $p = 2 \cdot 2^{386} 3^{242} - 1$. However, in the proposed FFM2, there is no such limitation on f or b.

In the original EFFM algorithm [24], to prevent the c_2 and c_3 values from increasing beyond the size of the modulus, they proposed efficient Barrett Division, as discussed in Section 2.3. Since Barrett Division uses the fact that division by two is a simple right shift operation, it can replace the complex division by simple shifting, multiplication and addition operations. Inspired by this fact, we notice that there exists a factor $f \cdot 2^a 3^b$ in the modulo p. Thus, a division whose divider is $f \cdot 2^a 3^b$ can be performed by the Barrett Division algorithm. If the modulo is $p = f \cdot 2^a 3^b$, we can complete the modular multiplication by only using Barrett Division. This form of modulo is an ideal modulo while a practical modulo is the ideal modulo plus or minus 1. In fact, there exists a relationship between the modular multiplication with an ideal modulo and the modular multiplication with a practical modulo, which will be discussed in the following subsections.

4.1 The FFM2 Algorithm

The ideal modulo equals to the practical modulo plus or minus 1. For these two cases, assume the following equations:

$$C = A \times B \tag{13}$$

$$C \div (f \cdot 2^a 3^b) = q \cdots r. \tag{14}$$

In Eq. (13), C represents the product of A and B. In Eq. (14), q and r represent the quotient and remainder, respectively, of the division $C \div (f \cdot 2^a 3^b)$.

4.1.1 *First Case: The Practical Modulo Equals the Ideal Modulo Minus 1*

In this case, we get:

$$f \cdot 2^a 3^b = p + 1. \tag{15}$$

Then, the operation $C \ mod \ p$ can be expressed as follows:

$$C \equiv q \cdot (p+1) + r \equiv qp + q + r \equiv (q+r) mod \ p. \tag{16}$$

Eq. (16) shows that the result of the modular multiplication with a practical modulo can be performed by adding the quotient and remainder of the modular multiplication with an ideal modulo. However, there is a good chance that the sum of the quotient and remainder may be longer than the modulus length. So we need to check the range of the sum, which determines the performance of the algorithm. According to Eq. (14), r is the remainder, so we can get:

$$r < f \cdot 2^a 3^b, \tag{17}$$

In this case,

$$p = f \cdot 2^a 3^b - 1, \tag{18}$$

Thus,

$$r \leqslant p, \tag{19}$$

As

$$\begin{array}{c} A \in [0, p-1] \\ B \in [0, p-1]. \end{array} \tag{20}$$

We have

$$C = A \times B \leqslant (p-1)^2. \tag{21}$$

Moreover,

$$f \cdot 2^a 3^b = p + 1. \tag{22}$$

Thus, we can get

$$q = \lfloor C/(f \cdot 2^a 3^b) \rfloor < p. \tag{23}$$

By considering both Eqs. (19) and (23), we have:

$$r + q < 2p. \tag{24}$$

It shows that the sum of the quotient and remainder lies in the range $[0, 2p)$. In conclusion, when the sum $r + q$ is larger than p, we need to perform another subtraction to get the final result, which is similar as the final step of the Montgomery algorithm [27].

4.1.2 *Second Case: The Practical Modulo Equals the Ideal Modulo Plus 1*

In this case, we have:

$$f \cdot 2^a 3^b = p - 1. \tag{25}$$

Then, $C \ mod \ p$ can be expressed as follows:

$$C \equiv q \cdot (p-1) + r \equiv qp - q + r \equiv (r-q) mod \ p. \tag{26}$$

Eq. (26) shows that the result of the modular multiplication with a practical modulo can be performed by subtracting the remainder from the quotient of the modular multiplication with an ideal modulo.

Similar to above, the result may be longer than the modulus, hence we need to check the range of the difference. We know that r is the remainder, so Eq. (17) holds.

As

$$p = f \cdot 2^a 3^b + 1. \tag{27}$$

We have:

$$r < p. \tag{28}$$

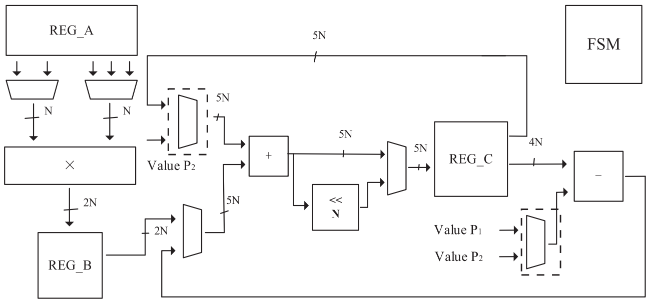

Fig. 4. The proposed hardware architecture for FFM2.

As Eqs. (19) and (20) hold, and

$$f \cdot 2^a 3^b = p - 1. \tag{29}$$

We have:

$$q = \lfloor C/(f \cdot 2^a 3^b) \rfloor \leqslant p. \tag{30}$$

As r and q are positive, we have

$$r - q \in [-p, p). \tag{31}$$

If the difference is smaller than 0, we need to perform another addition.

Algorithm 5. The FFM2 Algorithm

```
**Input:** $A, B \in F_p, p = f \cdot 2^a 3^b \pm 1$
**Output:** $C = A \times B (mod\ p)$
$C = A \times B$;
$q, r \rightarrow BarrettDivision(C, f \cdot 2^a 3^b)$;
if $p = f \cdot 2^a 3^b - 1$ then
    $C = q + r$;
    if $C > p$ then
        $C = C - p$;
    end
end
else
    $C = r - q$;
    if $C < 0$ then
        $C = C + p$;
    end
end
return $C$;
```

Referring to Algorithm 5, we can see that there are three main steps in implementing FFM2. The first step is to calculate the product of A and B. Then, we use Barrett Division to get the r and q. Finally, according to the practical modulo p, we may need to perform an extra subtraction or addition to obtain the correct result. Once the modulo p is determined, i.e., p equals either $f \cdot 2^a 3^b + 1$ or $f \cdot 2^a 3^b - 1$, no more than 4 steps are required in Algorithm 5. In conclusion, the FFM2 is much simpler than the EFFM and the FFM1. It has less steps and less complex operations. In particular, the FFM2 only needs to perform the Barrett Division once while the other two algorithms need to perform it twice.

4.2 The Proposed Hardware Architecture for FFM2

In order to have a comprehensive comparison, we also propose a hardware architecture for the FFM2, which is shown in Fig. 4. It is made of one N-bit multiplier, one 5N-bit adder and one 4N-bit subtractor. Similar as the FFM1 hardware architecture, the modular multiplication process is controlled by a FSM.

Compared with the EFFM and the FFM1, the FFM2 does not require the radix in the form, $R = 2^{a/2} 3^{b/2}$. Thus, the lengths of inputs of multiplier, adder and subtractor are doubled. There is one $2N \times N$ multiplication, one $2N \times 2N$ multiplication and one $3N \times 3N$ multiplication. All three multiplications can be converted into a summation of several $N \times N$ multiplications. If we do not break-up the multiplication, at most three additions and subtractions need to be performed, which is very simple. During the whole process, we only need 8 registers to store the precomputed values and intermediate values. Moreover, we use less MUXs to choose the different inputs. There are two dashed blocks in Fig. 4, which represent two different cases: $p = f \cdot 2^a 3^b \pm 1$. In the hardware design, only one MUX exists. The values P_1 and P_2 in Fig. 4 represent the precomputed value in the Barrett Division, and the practical modulo p, respectively.

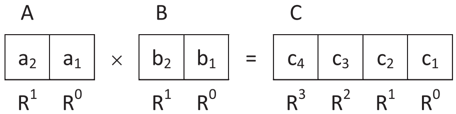

Fig. 5. Breaking $2N \times 2N$ multiplication into $N \times N$ multiplications: $R = 2^N$.

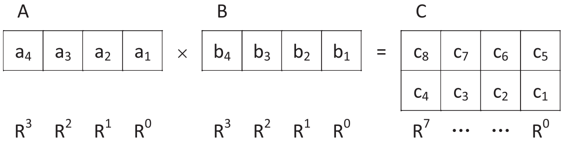

Fig. 6. Breaking $2N \times 2N$ multiplication into $N/2 \times N/2$ multiplications: $R = 2^{N/2}$.

4.3 Comparison of Hardware Architectures with Different Sizes of Multipliers

In the hardware implementation of the FFM2, we explore the impact of the operand sizes of the multipliers on the algorithm. We use two kinds of multipliers: one has the size of $N \times N$ as mentioned in the Section 4.2, while the other one has the size of $N/2 \times N/2$. In other words, we explore two ways to break large size multiplication into smaller ones. In this work, we break a $2N \times 2N$ multiplication into the following ways, as shown in Figs. 5 and 6.

In our design, there is only one multiplier no matter how large the size of the multiplier is. Thus, the smaller the multiplier, the greater the number of cycles needed to complete the operation. We know that for a $2N \times 2N$ multiplication, only 4 cycles are needed when we use the $N \times N$ multiplier, while we need 16 cycles when we use the $N/2 \times N/2$ multiplier. However, the operating frequency is lower and more resources are consumed when we use the larger multiplier. It is clear that there is a trade-off between throughput and hardware resources. We replace the $N \times N$ multiplier of the hardware architecture outlined in Section 4.2 with an $N/2 \times N/2$ multiplier, without changing other parts. We compare the hardware implementations of these two forms of multipliers, as shown in Table 1. It can be seen that the hardware architecture with the $N \times N$ multiplier needs less time to finish a full multiplication than the one with the $N/2 \times N/2$ multiplier. However, it requires double the number of LUTs and quadruple the number of DSPs.

TABLE 1
The Comparison of Hardware Architectures with Different Sizes of Multiplier of the FFM2

Size	$N/2 \times N/2$	$N \times N$
FFs	11585	11632
LUTs	17706	33049
DSPs	122	529
Frequency(MHz)	48	25
Operations	48 MP	12 MP
Time (μs)	1.33	1.12

TABLE 2
Comparison of FFM Hardware Results

Algorithms	EFFM [24]	FFM1	FFM2
FFs	11924	9675	11632
LUTs	12970	16629	33051
DSPs	0	122	529
Frequency (MHz)	31	55	25
Cycle	236	64	28
Time (μs)	7.61	1.16	1.12

TABLE 3
Comparison of the SIDH Software and HW/SW Codesign Implementations

Work	Platform	Total Time (ms)
Costello's Work [20]	i5-4210U@1.7 GHz	295.89
This Work	Kintex 7 FPGA+ i5-4210U@1.7 GHz	224.18

5 Results and Comparison

The proposed algorithms FFM1 and FFM2 are implemented in hardware and compared with the original EFFM algorithm [24]. We also apply the FFM2 hardware architecture in the HW/SW codesign implementation of the complete SIDH and compare it with the best SIDH software implementation [20] in this section.

5.1 Hardware Implementations of the Proposed FFMs

The proposed algorithms and the EFFM [24] are implemented using Vivado 16.4 on the KC705 evaluation board (with Kintex 7 FPGA chip, i.e., xc7k325tffg900-2). The proposed hardware architecture is applied. In order to have a fair comparison, we choose the same finite field, i.e., the field generated by the prime $p = 2 \cdot 2^{386}3^{242} - 1$, which is consistent with that in [24].

The hardware comparison with [24] is showed in Table 2. The proposed hardware architecture for the FFM1 algorithm uses 9,688 flip-flops (FFs), 17,247 LUTs and 122 DSP48s, which consumes 2, 8 and 15 of the resources available in the FPGA. One complete modular multiplication takes only 64 clock cycles and takes 1.16 μs. The operating frequency is 55 MHz. Compared with the hardware implementation in [24], our proposed FFM1 design is over 6.56 times faster.

As the hardware architecture for FFM2 with the $N \times N$ multiplier is faster than that with the $N/2 \times N/2$ multiplier as shown in Table 1, it is chosen to compare with other designs. It uses 11,632 flip-flops (FFs), 33,501 LUTs and 529 DSPs. It is the fastest design among all hardware implementations for the modular multiplication.

5.2 HW/SW Codesign Implementation of the SIDH

To evaluate the performance of the SIDH protocol using the proposed modular multiplier algorithm and hardware architecture, the FFM2 hardware is use for performing modular multiplications in the protocol. To compare with the previous best SIDH software implementation [20], the same prime of $2^{372}3^{239} - 1$ is chosen and the same processor (1.7 GHz Intel i5-4210U processor) is used for both the software and the HW/SW codesign implementations. However, for a higher performance codesign, the Zynq FPGA with ARM processor can be considered. FFM1 is not used as it is slower than FFM2 in hardware and also it requires that b in the prime should be even, which cannot be applied in the comparison. The complete SIDH protocol is implemented in the software except the modular multiplications which are performed in hardware (on Kintex 7 FPGA).

The results of both software and HW/SW codesign results are provided in Table 3. It can be seen that the SIDH codesign implementation using the proposed FFM2 hardware is 31.98 percent faster than the software implementation.

6 Conclusions and Future Work

In this paper, we proposed two new modular multiplication algorithms that exploit the special structure of primes, i.e., $p = 2 \cdot 2^{386}3^{242} - 1$, which can be applied in SIDH. One is improved from the original EFFM algorithm in [24] and the other one is a whole new algorithm, which differs in structure to the previous two algorithms. Building on the properties of the previous two algorithms, a mathematical transformation is applied to reduce the number of operations in the first new algorithm (FFM1). A hardware architecture is also proposed. The proposed FFM2 is over 6 times faster than the previous EFFM in hardware. The hardware implementation of the FFM2 algorithm is the fastest among the three algorithms. Furthermore, the FFM2 algorithm can be applied to a wide range of modulo, which is limited in the EFFM algorithm and the FFM1 algorithm. The FFM2 hardware is also applied in the complete SIDH HW/SW codesign implementation, which is over 31 percent faster than the best SIDH software implementation. Future work will look at the optimized modular multiplication on F_{p^2} as suggested in [23].

Acknowledgments

This work is supported by grants from the National Natural Science Foundation of China (61871216 and 61802180), the Fundamental Research Funds for the Central Universities China (NE2019102 and NE2018106), the Six Talent Peaks Project in Jiangsu Province (2018-XYDXX-009), NSF of Jiangsu Province (BK20180421) and the National Cryptography Development Fund (MMJJ20180105).

References

[1] K. Bonsor and J. Strickland, "How quantum computers work," *How Stuff Works*, 2016. [Online]. Available: http://computer.howstuffworks.com/quantum-computer1.htm

[2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Sci. Statistical Comput.*, vol. 26, pp. 1484–1509, 1994.

[3] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th ACM Symp. Theory Comput.*, 1996, pp. 212–219.

[4] M. Devoret and R. Schoelkopf, "Superconducting circuits for quantum information: An outlook," *Sci.*, vol. 339, pp. 1169–1174, 2013.

[5] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[6] V. Miller, "Use of elliptic curves in cryptography," in *Proc. 7th Annu. Int. Cryptology Conf. Advances Cryptology*, 1985, vol. 218, pp. 417–426.

[7] D. J. Bernstein, "Introduction to post-quantum cryptography," *Post-Quantum Cryptography*, Berlin, Germany: Springer, 2009, pp. 1–14.

[8] J. Howe, T. Pöppelmann, M. O'Neill, E. O'Sullivan, and T. Gneysu, "Practical lattice-based digital signature schemes," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, 2015, Art. no. 41.

[9] Cybersecurity in A Post-quantum World, [Online]. Available: http://nist.gov/itl/csd/ct/post-quantum-crypto-workshop-2015.cfm

[10] Fourth ETSI/IQC Workshop on Quantum-Safe Cryptography, [Online]. Available: http://www.etsi.org/standards/how-does-etsi-make-standards/10 news-events/events/1072-ws-on-quantumsafe-2016

[11] Post-Quantum Cryptography, [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions

[12] D. Jao and L. D. Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Proc. 1st Int. Conf. Post-Quantum Cryptography*, 2011, pp. 19–34.

[13] V. Lyubashevsky, C. Peikert, andO. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, pp. 1–35, 2013.

[14] V. Lyubashevsky, "Practical lattice-based cryptography: A signature scheme for embedded systems," in *Proc. 20th Int. Conf. Cryptographic Hardware Embedded Syst.*, 2013, pp. 530–547.

[15] R. Overbeck and N. Sendrier, "Code-based cryptography," *Post-Quantum Cryptography*, Berlin, Germany: Springer, 2009, pp. 95–145.

[16] D. J. Bernstein, D. Hopwood, A. Hlsing, T. Lange, R. Niederhagen, and L. Papachristodoulou, "SPHINCS: Practical stateless hash-based signatures," in *Proc. 24th Int. Conf. Theory Appl. Cryptographic Techn.*, 2005, pp. 368–397.

[17] J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," in *Proc. 3rd Int. Conf. Appl. Cryptography Netw. Security*, 2005, pp. 164–175.

[18] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello and L. D. Feo, *SIKE - Supersingular Isogeny Key Encapsulation*, 2018. [Online]. Available: http://www.sike.org.htm,

[19] R. Azarderakhsh, D. Fishbein, and D. Jao, "Efficient implementations of a quantum resistant key-exchange protocol on embedded systems," Univ. Waterloo, Waterloo, ON N2L 3G1, Canada, 2014, CACR2014-20.

[20] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny DH," in *Proc. 36th Annu. Int. Cryptology Conf. Advances Cryptology*, 2016, pp. 572–601.

[21] B. Koziel, R. Azarderakhsh, M. Mozaffari Kermani, and D. Jao, "Post-quantum cryptography on FPGA based on isogenies on elliptic curves," *IEEE Trans. Circuits Syst. I: Reg. Paper*, vol. 64, no. 1, pp. 86–99, Jan. 2017.

[22] J. Bos and S. Friedberger, "Fast arithmetic modulo $2^x p^y \pm 1$," in *Proc. 24th IEEE Symp. Comput. Arithmetic*, 2017, pp. 148–155.

[23] J. Bos and S. Friedberger, "Arithmetic considerations for isogeny based cryptography," *IEEE Trans. Comput.*, 2018, doi: 10.1109/TC.2018.2851238.

[24] A. Karmakar, S. S. Roy, F. Vercauteren, and I. Verbauwhede, "Efficient finite field multiplication for isogeny based post-quantum cryptography," in *Proc. 10th Int. Workshop Arithmetic Finite Fields*, 2016, pp. 193–207.

[25] C. Liu, J. Ni, W. Liu, Z. Liu and M. O'Neill, "Design and optimization of modular multiplication for SIDH," in *Proc. IEEE Int. Symp. Circuits Syst.*, May. 2018, pp. 1–5.

[26] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proc. 7th Annu. Int. Cryptology Conf. Advances Cryptology*, 1987, pp. 311–323.

[27] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 170, no. 44, pp. 519–521, 1985.