High Performance Modular Multiplication for SIDH

Weiqiang Liu, *Senior Member, IEEE*, Ziying Ni, Jian Ni, Ciara Rafferty, *Member, IEEE*, and Máire O'Neill, *Senior Member, IEEE*

***Abstract*—The latest research indicates that quantum computers will be realized in the near future. In theory, the computation speed of a quantum computer is much faster than current computers, which will pose a serious threat to current cryptosystems. Post-quantum cryptography (PQC) is a class of cryptography based on underlying mathematical problems that are considered infeasible to crack even with access to a quantum computer. The supersingular isogeny Diffie–Hellman (SIDH) key exchange protocol is a new post-quantum cryptosystem, which offers advantages in reduced secret key length and attack resistance. SIDH is the basis of the supersingular isogeny key encapsulation (SIKE) protocol, which is in the second round of the U.S. National Institute of Standards and Technology (NIST) PQC standardization process. In this article, we propose a new modular multiplication algorithm and a new interleaved hardware architecture for SIDH. Performance results for the proposed modular multiplier using four parameter sets for the prime, p that correspond to the SIKE Round 2 parameter sets show significant advantages in speed.**

***Index Terms*—FPGA, modular multiplication, post-quantum cryptography (PQC), supersingular isogeny Diffie–Hellman (SIDH).**

I. Introduction

The computational performance of a quantum computer is expected to be much higher than that of a traditional computer. In 1994, Shor [1] proposed a fast decomposition method for large numbers based on a quantum computer, whose complexity is $\mathcal{O}(\log N)$, and the larger the number, the better the performance of Shor's algorithm. Later in 1996, Grover [2] proposed a quantum algorithm that can be used to search an unsorted database faster than a traditional computer [quadratic speedup $\mathcal{O}(N/2)$ time rather than $\mathcal{O}(N)$]. In recent years much effort has gone into the development of quantum computers by industry, and in 2019, Google developed a new 54-qubit processor, which is 9 orders of magnitude faster than the fastest supercomputer [3]. Thus, it is predictable that quantum computers will become practical in the near future.

Traditional encryption methods, such as RSA [4] based on the large number factorization problem, and elliptic curve cryptography (ECC) [5], based on the discrete logarithm problem, will be easily broken by quantum computers, which will have an important impact on current communications and network security.

Post-quantum cryptography (PQC) [6] is a form of public-key cryptography with high complexity, and includes encryption, digital signatures, and key encapsulation mechanisms. PQC contains a variety of algorithms based on different hard problems [7], such as lattice-based cryptography, multivariate quadratic cryptography, hash-based signatures, and code-based cryptography. The supersingular isogeny Diffie–Hellman (SIDH) [8] key exchange protocol is the most recently proposed PQC scheme. SIDH is based on the theory of point addition and point doubling in ECC, combined with the theory of isogenies in elliptic curves. Two curves E/K and E'/K are isogenous over K if there exists a morphism $\phi : E \rightarrow E'$ with coefficients in K mapping the neutral element of E to the neutral element of E'. Therefore, SIDH is more complex than ECC, and is believed to be resistant to attacks by quantum computers [9]. In addition, SIDH is characterized by relatively small key sizes compared with other PQC schemes. SIDH is the basis of the supersingular isogeny key encapsulation (SIKE) protocol [10], which is a candidate in the NIST PQC standardization process. However, as SIDH was proposed more than a decade after other PQC cryptosystems, there is still much work needed to understand its practical capability.

In 2011, Jao and De Feo [8] proposed the SIDH algorithm and implemented it for the first time in software. They used a 768-bit modulus to satisfy 128-bit security and the result is two orders of magnitude faster than isogeny-based cryptosystems over ordinary curves. In 2016, Costello *et al.* [9] proposed a fast software implementation for SIDH, whose computing speed is 2.5 times faster than the original software implementations. In the same year, the first hardware implementation of SIDH on Virtex-7 FPGA was presented in [11]. The implementation is 50% faster than the fastest known software implementation. Most recently, in 2019, Koziel *et al.* [12] improved the Montgomery multiplier with an optimized radix, which is faster than the previous designs.

In SIDH, the modulus $p = l_a^{e_A} l_b^{e_B} \cdot f \pm 1$, where l_a and l_b are two small prime numbers, and f is a small number, so that p conforms to the form of a prime number. In general, $\log_2 l_a^{e_A}$ is required to be approximately $\log_2 l_b^{e_B}$ to ensure that either side of A and B is equally difficult for attackers. The modulus plays an important role in public key cryptography schemes. Karmakar *et al.* [13] proposed a fast modulus multiplication method for the modulus in this special field. Liu *et al.* [14] improved the modular multiplication algorithm over [13] by proposing a new algorithm. Bos and Friedberger [15] compared several modular multiplication methods for SIDH mathematically, but did not provide hardware results.

This article proposes a new modular multiplication algorithm named high-performance finite field multiplication (HFFM) for the special field F_p, $p = f \cdot 2^a \cdot l_b^{e_B} - 1$, which is suitable for a larger modulus range. In addition, to reduce the overall computation time, a new multipipelined interleaved architecture running on FPGA is also proposed.

The rest of this article is organized as follows. Section II reviews the special field modulus multiplication method. Section III proposes the new HFFM algorithm and its hardware architecture. Section IV

Manuscript received April 23, 2019; revised September 18, 2019; accepted December 10, 2019. Date of publication December 17, 2019; date of current version September 18, 2020. This work was supported in part by the National Natural Science Foundation of China under Grant 61871216, in part by the Fundamental Research Funds for the Central Universities China under Grant NE2019102, and in part by the Six Talent Peaks Project in Jiangsu Province under Grant 2018XYDXX-009. This article was recommended by Associate Editor Y. Makris. *(Corresponding author: Weiqiang Liu.)*

Weiqiang Liu, Ziying Ni, and Jian Ni are with the College of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: liuweiqiang@nuaa.edu.cn; nzy@nuaa.edu.cn; nijian@nuaa.edu.cn).

Ciara Rafferty and Máire O'Neill are with the Centre for Secure Information Technologies, Queens University Belfast, Belfast BT3 9DT, U.K. (e-mail: c.m.raffertyg@qub.ac.uk; m.oneill@ecit.qub.ac.uk).

Digital Object Identifier 10.1109/TCAD.2019.2960330

presents the results of the hardware implementation and compares it with the existing hardware implementations. Section V concludes this article.

II. Review of Finite Field Multiplication for SIDH

The Barrett reduction [16] is introduced first, which is used in the following algorithms. Barrett reduction consists of several multiplications and shift operations instead of the division operation. As a result, it is efficient for hardware implementation. The transformation of division in $a \bmod b$ is shown in (1), where $x = 2^k/b$, $k = \log_2 a$. For more details on Barrett reduction, please refer to [16]

$$\frac{1}{b} = \frac{\left(2^k\right)/b}{b \cdot 2^k/b} = \frac{\left(2^k\right)/b}{2^k} \approx \frac{x}{2^k}. \tag{1}$$

One of the moduli in the SIDH public key exchange protocol is $p = f \cdot 2^a 3^b - 1$, where f is a very small positive integer, such as 1, 2, etc. Based on this form of special field, [13] proves that a good performance can be achieved using Barrett reduction directly and proposed a fast modulus multiplication method named efficient finite field multiplication (EFFM) through mathematical transformation. Assume $p = 2 \cdot 2^a \cdot 3^b - 1$ and that both a and b are even numbers, and the radix $R = 2^{a/2} \cdot 3^{b/2}$. Therefore, the number A in this field can be expressed as follows:

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

B in this field also satisfies the structure of (2), and the product C can be expressed as follows:

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

As $2 \cdot R^2 = p + 1$, we can get $R^2 \equiv 2^{-1} (\bmod p)$. R^2 and R^4 in (3) can be replaced with $2^{-1} (\bmod p)$ and $2^{-2} (\bmod p)$, respectively, which can be precomputed and stored directly in the hardware to save computation time. Through the above transformation, (3) can be converted into (4) and C can be expressed as $C = c_1 \cdot R^2 + c_2 \cdot R + c_3$

$$\begin{aligned} C = {} & (a_1 b_3 + a_2 b_2 + a_3 b_1) (\bmod 2) \cdot R^2 \\ & + (\lfloor (a_1 b_2 + a_2 b_1)/2 \rfloor + (a_2 b_3 + a_3 b_2)) \cdot R \\ & + \left(2^{-2} (\bmod p) a_1 b_1 + (a_1 b_2 + a_2 b_1) (\bmod 2)\right) \cdot R/2 \\ & + \lfloor (a_1 b_3 + a_3 b_1 + a_2 b_2/2) \rfloor + a_3 b_3. \end{aligned} \tag{4}$$

As c_2 and c_3 are beyond the range of $[0, R)$, a final division operation is required. Karmakar *et al.* [13] proposed the use of the Barrett division algorithm based on the special structure of the modulus and Barrett reduction. $2^{a/2}$ is a factor of R and the multiplication and division of 2 in the digital circuit only need shift operation. Thus, we can extract $2^{a/2}$ from the divisor and divide $3^{b/2}$ only. Since b is a fixed integer, the Barrett reduction can be used. The above method can shorten the bit width of the dividend by a quarter. After the above process, the result of the Barrett division may be greater than p, so a subtraction is required for this case.

The values of a_1 and b_1 in (2) can only be 0 or 1, and dividing the number in this field into three parts for calculation is not the best choice. Liu *et al.* [14] proposed the improved EFFM (FFM1) algorithm to optimize (4). Suppose A in this special field still satisfies (2), as $(p - A)(p - B) (\bmod p) = A \cdot B (\bmod p)$, if A is greater than R^2, then A' can be found using

$$A' = p - A \quad (a_1 = 1). \tag{5}$$

Equation (6) is the expression expanded by A', where $a'_i = R - 1 - a_i, i = 2, 3$

$$A' = a'_2 \cdot R + a'_3, a'_2, a'_3 \in [0, R). \tag{6}$$

Then, the product C' of A' and B' is expressed as (7). Moreover, it is not guaranteed that the two multiplicands A and B are greater than R^2 at the same time, and the final results may need to be modified. The modified process is shown in

$$\begin{aligned} C' &= a'_2 b'_2 \cdot R^2 + \left(a'_2 b'_3 + a'_3 b'_2\right) \cdot R + a'_3 b'_3 \\ &= \left(a'_2 b'_2 (\bmod 2)\right) \cdot R^2 + \left(a'_2 b'_3 + a'_3 b'_2\right) \cdot R \\ &\quad + \left(a'_3 b'_3 + \left\lfloor a'_2 b'_2/2 \right\rfloor\right) \\ &= c'_1 \cdot R^2 + c'_2 \cdot R + c'_3 \end{aligned} \tag{7}$$

$$A \cdot B (\bmod p) = A' \cdot B' (\bmod p) \cdot (-1)^{a_1 \oplus b_1}. \tag{8}$$

In addition, c'_2 and c'_3 in (7) may be greater than R, and hence, may need to be reduced by Barrett division.

In order to solve the problem that only a small range of moduli can be used in the FFM1 algorithm, Liu *et al.* [14] proposed another finite field multiplication (FFM2) algorithm. The FFM2 algorithm no longer divides operands A and B. f, a, and b are no longer required to meet the above conditions as in FFM1. In addition, FFM2 algorithm using $p = 2^a 3^b \cdot f + 1$ and $p = 2^a 3^b \cdot f - 1$ are studied separately.

Assume that $p = 2^a 3^b \cdot f - 1$, in this case, we can get

$$f \cdot 2^a 3^b = p + 1. \tag{9}$$

Then, the product $C \bmod p$ can be expressed as follows:

$$C \equiv q \cdot (p + 1) + r \equiv qp + q + r \equiv (q + r) \bmod p. \tag{10}$$

From (10), we know that when p is $f \cdot 2^a 3^b - 1$, we can first find the quotient and remainder of C to $f \cdot 2^a 3^b$, and then add the quotient and remainder to get the result. However, the sum of the quotient and remainder may be greater than the modulus p; so the result needs to be modified. In (11), r is the remainder and q is the quotient. By deduction, we have

$$0 \le r + q < 2p. \tag{11}$$

As it can be seen from (11), in this case, a subtraction may be required in the final stage to modify the final result. Similarly, when $p = f \cdot 2^a 3^b + 1$, we have

$$C \equiv q \cdot (p - 1) + r \equiv qp - q + r \equiv (r - q) \bmod p. \tag{12}$$

By deduction, we have

$$-p \le r - q \le p. \tag{13}$$

When the difference is less than 0, we have to add p to the final result. For more details on the FFM1 and FFM2 algorithm, please refer to [11].

III. HFFM Multiplication Algorithm and Its Hardware Architecture

A. HFFM Algorithm for SIDH

The dividing and conquering method of the EFFM and FFM1 algorithms is an effective idea, and (10) can simplify the computation. Inspired by this, if $a/2$ is used to preprocess the operands, only one Barrett reduction is required. Therefore, we choose $f \cdot l_b^b$ as a radix. For $f \cdot l_b^b$, the values of f and b will be chosen to make sure that p is a prime.

Thus, the HFFM algorithm is proposed as follows. If $p = f \cdot 2^a \cdot l_b^b - 1$, then f is a very small positive integer, and a and b do not need to be even. The number A in this special field can be expressed as follows:

$$A = a_1 \cdot f \cdot l_b^b + a_2, a_1 \in \left[0, 2^a\right), a_2 \in \left[0, l_b^b\right). \tag{14}$$

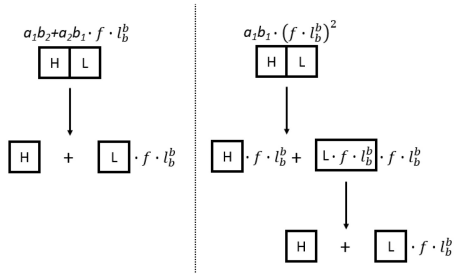

Fig. 1. Decomposition in the HFFM algorithm. (H: the first half of the operand, L: the second half of the operand).

B in this field can also be written in the same expression as (14). Therefore, the product C of A and B can be expressed as follows:

$$C = a_1b_1 \cdot \left(f \cdot l_b^b\right)^2 + (a_1b_2 + a_2b_1) \cdot f \cdot l_b^b + a_2b_2. \quad (15)$$

As $a_1b_2 + a_2b_1$ is less than $2p$ and the product of 2^a and $f \cdot l_b^b$ is $(p+1)$, $a_1b_2 + a_2b_1$ is split in half by 2^a. Set $m_1 = \lfloor (a_1b_2 + a_2b_1)/2^a \rfloor$, which is the first half of $a_1b_2 + a_2b_1$, and set $m_2 = (a_1b_2 + a_2b_1)(\mathrm{mod}\, 2^a)$, which is the second half of $a_1b_2 + a_2b_1$. Then $(a_1b_2 + a_2b_1) \cdot f \cdot l_b^b$ is converted into

$$\begin{aligned}(a_1b_2 + a_2b_1) \cdot f \cdot l_b^b &= \left(m_1 \cdot 2^a + m_2\right) f \cdot l_b^b \\ &= m_1 \cdot 2^a\left(f \cdot l_b^b\right) + m_2 \cdot f \cdot l_b^b \\ &= m_1 \cdot (p+1) + m_2 \cdot f \cdot l_b^b \\ &= m_1 + m_2 \cdot f \cdot l_b^b \\ &= \lfloor (a_1b_2 + a_2b_1)/2^a \rfloor \\ &\quad + (a_1b_2 + a_2b_1)\left(\mathrm{mod}\, 2^a\right) \cdot f \cdot l_b^b. \quad (16)\end{aligned}$$

Equation (16) breaks $(a_1b_2 + a_2b_1) \cdot f \cdot l_b^b$ into two parts. One part takes 1 as the radix and the other part takes $f \cdot l_b^b$ as the radix. The bit width of each part is only about half of p. Similarly, since a_1b_1 is less than $2p$, we split it into two terms by 2^a. After the first split, it will produce a radix of $(f \cdot l_b^b)^2$. As the bit width of $f \cdot l_b^b$ and $(a_1b_1)(\mathrm{mod}\, 2^a)$ are both about half of p, the product can be divided by 2^a as above. Therefore, $a_1b_1 \cdot (f \cdot l_b^b)^2$ is converted as follows:

$$\begin{aligned}&a_1b_1\left(f \cdot l_b^b\right)^2 \\ &= \lfloor (a_1b_1)/2^a \rfloor \left(f \cdot l_b^b\right) + (a_1b_1)\left(\mathrm{mod}\, 2^a\right)\left(f \cdot l_b^b\right)^2 \\ &= \lfloor (a_1b_1)/2^a \rfloor \left(f \cdot l_b^b\right) + \left\lfloor \left((a_1b_1)\left(\mathrm{mod}\, 2^a\right)\left(f \cdot l_b^b\right)/2^a\right) \right\rfloor \\ &\quad + \left(\left((a_1b_1)\left(\mathrm{mod}\, 2^a\right) \cdot \left(f \cdot l_b^b\right)\right)\left(\mathrm{mod}\, 2^a\right)\right) \cdot \left(f \cdot l_b^b\right). \quad (17)\end{aligned}$$

Equation (15) is finally expressed as follows:

$$\begin{aligned}C &= c_1 \cdot \left(f \cdot l_b^b\right) + c_2 \\ &= \Big((a_1b_2 + a_2b_1)\left(\mathrm{mod}\, 2^a\right) + \lfloor (a_1b_1)/2^a \rfloor \\ &\quad + \left(\left((a_1b_1)\left(\mathrm{mod}\, 2^a\right) \cdot \left(f \cdot l_b^b\right)\right)\left(\mathrm{mod}\, 2^a\right)\right)\Big) \cdot \left(f \cdot l_b^b\right) \\ &\quad + \Big(a_2b_2 + \lfloor (a_1b_2 + a_2b_1)/2^a \rfloor \\ &\quad + \left\lfloor \left((a_1b_1)\left(\mathrm{mod}\, 2^a\right)\left(f \cdot l_b^b\right)/2^a\right) \right\rfloor\Big). \quad (18)\end{aligned}$$

The above decomposition process is shown in Fig. 1. In addition, in the initial stage of the algorithm, the Karatsuba algorithm can be used to calculate $a_1b_2 + a_2b_1$ to save one multiplication. c_2 in (18) may be larger than $f \cdot l_b^b$; so Barrett reduction is used to ensure c_2 falls in $[0, f \cdot l_b^b)$. To further save multiplication operations, we only calculate the first half of the dividend in the Barrett reduction, which may produce a result of Barrett reduction that is more than $2 \cdot f \cdot l_b^b$, and one extra subtraction is required. After that, the results of adding the quotient and c_1 may also be greater than 2^a. Therefore, a shift operation may be required. In (15), splitting $a_1b_2 + a_2b_1$ is not necessary; however, it shortens the coefficient bit width of $f \cdot l_b^b$. The proposed HFFM algorithm is shown in Algorithm 1.

Algorithm 1 HFFM Multiplication With Special Field

1: **input** : $A = a_1 fl_b^b + a_2$; $B = b_1 fl_b^b + b_2$
2: **output** : $C = c_1 fl_b^b + c_2$
3: **compute** $a_1b_1, (a_1 + a_2)(b_1 + b_2), a_2b_2, a_1b_2 + a_2b_1 = (a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2, r_1 = a_1b_1 \bmod 2^a, q_1 = \lfloor a_1b_1/2^a \rfloor, r_1 fl_b^b$;
4: $c_2 = a_2b_2 + \lfloor (a_2b_1 + a_1b_2)/2^a \rfloor$;
5: $c_2 = c_2 + \lfloor r_1 fl_b^b/2^a \rfloor$;
6: $c_1 = q_1 + (a_2b_1 + a_1b_2) \bmod 2^a$;
7: $c_1 = c_1 + r_1 fl_b^b \bmod 2^a$;
8: Barrett Reduction(c_2)$\Rightarrow c_2 = r, c_1 = p + q$;
9: $c_1 = c_1 \bmod 2^a$; $c_2 = c_2 + \lfloor c_1/2^a \rfloor$.

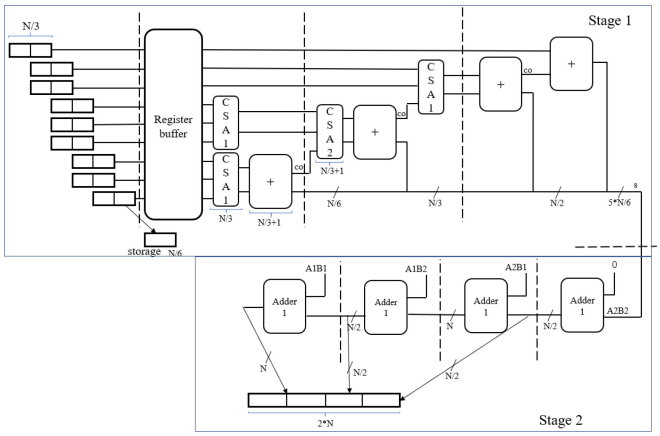

Fig. 2. Proposed new architecture for $N \cdot N$ multiplication in eight cycles.

Similar to the FFM1 algorithm, the HFFM algorithm divides the multiplier into two segments. However, the new algorithm uses $f \cdot l_b^b$ to segment operands and uses multiple shifting operations instead of partial division. As a result, the number of multiplications and additions in the HFFM algorithm are less than that of the FFM1 and FFM2 algorithms. The multiplication in the HFFM algorithm is also more friendly for pipelined hardware implementation. Compared with FFM2, if under the same modulus, the HFFM algorithm performs the same operation with only half of the bit width.

B. New Hardware Architecture for the Proposed HFFM Algorithm

We also propose a new interleaved hardware architecture to increase the operating frequency and reduce the computation time of the SIDH modular multiplication. In this architecture, there are 9 $N/6$-bit multipliers, 3 $N/3$-bit carry-save adders (CSAs), 1 $(N/3+1)$-bit CSA, 4 $(N/3+1)$-bit adders, and 2 N-bit adders. The whole design is controlled by a finite state machine (FSM).

As observed from previous designs, the large multiplier limits the operating frequency. Further splitting up the multiplication units can further increase the frequency. For this design, we choose the modulus $p = 2^a \cdot l_b^b \cdot f - 1$, and $K = a$ which is used for the shift operation, $N' = \max(\mathrm{digit}(2^a), \mathrm{digit}(l_b^b \cdot f))$. Complete N' to the point that it is divisible by 6 to get N. It takes 8 cycles to complete an $N \cdot N$ multiplication, which is divided into two stages. As shown in Fig. 2,

TABLE I
COMPARISON OF DIFFERENT HARDWARE ARCHITECTURES AND ALGORITHMS OF MODULAR MULTIPLICATION FOR SIDH ON VIRTEX-7 FPGA (F: FREQUENCY, MULT.: THE FIRST TWO MODULAR MULTIPLICATIONS)

Prime	Designs	LUTs	FFs	DSPs	F (MHz)	Mult. (Cycles)	Mult. (Time)	Interleave (Cycles)	Interleave (Time)	Improvement in Time(Mult.)	Improvement in Time (Interleave)
$SIKEp434$	Montgomery Multiplier [12]	-	-	56	207	88	425ns	61	295ns		
	HFFM Algorithm(N/6)	**6724**	**6216**	**36**	**236**	**66**	**280ns**	**54**	**229ns**	**34.12%**	**22.37%**
$SIKEp503$	Design in [12]	2952	5752	88	171	70	409ns	49	286ns		
	HFFM Algorithm(N/8)	**7757**	**7357**	**64**	**213**	**66**	**310ns**	**54**	**254ns**	**24.21%**	**11.19%**
$SIKEp610$	Montgomery Multiplier [12]	-	-	78	207	121	585ns	83	401ns		
	HFFM Algorithm(N/6)	**8600**	**8004**	**81**	**191**	**66**	**346ns**	**54**	**283ns**	**40.85%**	**29.43%**
$SIKEp751$	Design in [12]	-	-	128	167	100	597ns	69	412ns		
	HFFM Algorithm(N/6)	**10768**	**10481**	**144**	**161**	**66**	**410ns**	**54**	**335ns**	**31.32%**	**18.69%**
P_{771}	FFM1 Algorithm [14]	16502	9661	122	61	64	1049ns	64	1049ns		
	HFFM Algorithm(N/6)	**11007**	**10680**	**144**	**152**	**66**	**434ns**	**54**	**355ns**	**58.63%**	**66.16%**

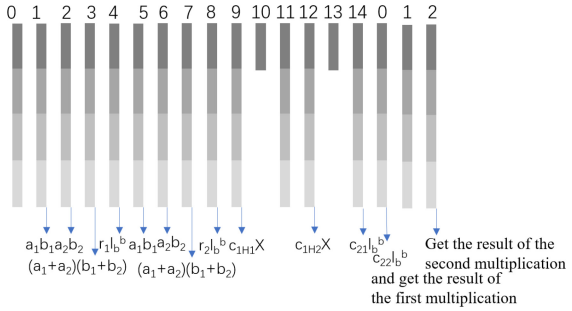

Fig. 3. Steps of $N \cdot N$ multiplications of the proposed modular multiplier.

it takes 4 cycles for the first stage to complete a multiplication of $N/2 \cdot N/2$.

In the first cycle of the first stage, we use 9 $N/6 \cdot N/6$ multipliers to calculate the partial products. In the second cycle, the results from the first cycle are stored in a register. In addition, we calculate the sum of the lowest three partial products using an adder and a CSA, then store the $N/6$-bit sum and pass the $(N/6+1)$-bit carry to the next cycle. In order to reduce the delay of the third cycle, a CSA is used to perform preliminary operations on the middle three partial products. In the third cycle, a CSA and $(N/3+1)$-bit adders are used to obtain the addition result of the last six partial products. After that, the $(N/6+1)$-bit carry and two partial products are sent to a CSA for calculation. In the fourth cycle, we use two $(N/3+1)$-bit adders to get the most significant $(5N/6)$-bit of the $N/2 \cdot N/2$ multiplication result.

If the data pipeline passes the 4 cycles in the first stage, then the next cycle after the first $N/2 \cdot N/2$ multiplication operation is complete will get the result of the second $N/2 \cdot N/2$ multiplication. In order to use an adder with a smaller number, the partial product of $N \cdot N$ is calculated in the order from the least significant part to the most significant part. Therefore, for the 4 cycles of the second stage, an N-bit adder is used, and the adder gets the N-bit addition result in every clock cycle. If the second half of the result no longer participates in the operation, it will be stored in the register in advance. Fig. 2 shows the calculation of an $N \cdot N$ multiplication in 8 cycles.

However, the last multiplication $[((a_1 b_1) \mod 2^a)(f \cdot l_b^b)]$ is dependant on the result of $a_1 b_1$, while the other multiplications are independent of each other. For multiple $N \cdot N$ multiplications, the pipelined architecture can also be used. A four-stage pipeline architecture is proposed for use here, which can save 12 cycles.

Since only the first N-bit of the dividend is calculated when performing the first step of the Barrett reduction, the reduction result may be greater than 2 times $f \cdot l_b^b$. As it is an odd number (extract a multiple of 2 in f and add it to a), the lowest significant bit (LSB) of the inversion of $f \cdot l_b^b$ is changed from 0 to 1 to get its complement. For the same reason, the complement of $2f \cdot l_b^b$ is inverted by $f \cdot l_b^b$ first, and the LSB is replaced by 1 and then it is shifted one bit to the left. In addition, we only need to subtract the lower $(b+1)$ bits of c_2. The modular multiplication is complete.

It can be seen that all blocks of the proposed modular multiplier are not working at the same time, which causes low computational efficiency. There is a lengthy waiting time between the initial stage of the modular multiplier and the multiplication required by Barrett reduction; therefore, we consider interleaving the two modular multiplications. The specific steps of the operation are shown in Fig. 3.

We call each four cycles a layer. As we can see from Fig. 3, the interleaved multiplier calculates the four partial products required for the first modular multiplication at layers 0–4, and then calculates four partial products required for the second modular multiplication at layers 5–8. At the same time, we calculate c_1 and c_2 of the first modular multiplication. Then the two multiplications in Barrett reduction are calculated in turn. There is only one cycle in the 10th layer and 13th layer because the operand of the next multiplication has not yet been calculated and needs to be extended by one cycle. Since the next two modular multiplications can be calculated at the beginning of each 0 layer, the iteration period is also shortened. At this time, the multiplier requires 66 cycles for the first two modular multiplications, and a multiplication iteration is completed every 54 cycles.

IV. RESULTS AND COMPARISON

To offer a fair comparison with previous work, the proposed hardware modular multiplier architecture is implemented on a Virtex-7 xc7vx690tffg1157-3 device, and compiled with Xilinx Vivado 2018.2. The modulus $p = 2^{387} \cdot 3^{242} - 1$ and four parameter sets corresponding to the NIST Round 2 SIKE parameter sets are used.

The FPGA implementation results for the five parameter sets are listed in Table I. The proposed modular multiplier design is compared with the latest design [12] for parameter sets *SIKEp*503 and *SIKEp*751, while it is compared with the systolic array Montgomery multiplier from [12] for *SIKEp*434 and *SIKEp*610. Since there is no implementation for reference and every block in the systolic array Montgomery multiplier are the same, we select the highest frequency of *SIKEp*503 and *SIKEp*751 in [12] to estimate the frequency of the systolic array Montgomery multiplier of *SIKEp*434 and *SIKEp*610. We also compare with [14] for P_{771}. In order to save DSP resources, we use 16 $N/8$ multipliers in the *SIKEp*503, which reduces DSPs from 81 to 64.

From Table I, all of the proposed HFFM designs are faster than the previous designs. Compared with the optimized Montgomery multiplier in [12], the proposed design can still achieve 11–18% speed improvement and compared with the conventional Montgomery multiplier, the proposed design achieves a 22–29% speed improvement. As can be seen from the table, In the case of *SIKEp*434 and

*SIKEp*503, the proposed design uses less DSP resources. Since a DSP can calculate a 17×17 unsigned multiplication [12] and can be converted into 345LUTs in Multiplier 12.0 of Vivado, the resources we used in *SIKEp*434 are similar to the previous design. The proposed design with *SIKEp*503 uses less resources. In addition, the proposed design for prime, P_{771}, is nearly two times faster than the previous reported equivalent design.

V. Conclusion

In this article, a new high performance modular multiplication algorithm named HFFM for the specific fields in SIDH is proposed. This algorithm saves multiplications and additions compared with the previous algorithms. In addition, we propose a new interleaved multiplication architecture that allows two multiplications to be interleaved, which significantly saves computation time. Compared with previous work, the proposed multipliers are much faster. In addition, the proposed multiplier for the *SIKEp*503 parameter set uses even less hardware resources. Overall, it is evident that the proposed multipliers can be used to achieve high speed SIKE implementations in hardware.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Sci. Stat. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1994.

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

[3] F. Arute *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, pp. 505–510, Oct. 2019.

[4] R. L. 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.

[5] V. S. Miller, "Use of elliptic curves in cryptography," in *Proc. 7th Annu. Int. Cryptol. Conf. Adv. Cryptol.*, vol. 218, 1985, pp. 417–426.

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

[7] L. Chen *et al.*, "NIST: Report on post-quantum cryptography," Nat. Inst. Stand. Technol., Gaithersburg, MD, USA, Rep. NISTIR 8105, 2016.

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

[9] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie-Hellman," in *Proc. 36th Annu. Int. Cryptol. Conf. Adv. Cryptol.*, vol. 9814, 2016, pp. 572–601.

[10] D. Jao *et al.*, *Supersingular Isogeny Key Encapsulation* NIST Post Quantum Standardization Project, Gaithersburg, MD, USA, 2017.

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

[12] B. Koziel, A.-B. Ackie, R. E. Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "SIKE'd up: Fast and secure hardware architectures for supersingular isogeny key encapsulation," IACR Cryptol. ePrint Archive, Rep. 2019/711, 2019. [Online]. Available: https://eprint.iacr.org/2019/711

[13] 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.

[14] W. Liu, J. Ni, Z. Liu, C. Liu, and M. O'Neill, "Optimized modular multiplication for supersingular isogeny Diffie-Hellman," *IEEE Trans. Comput.*, vol. 68, no. 8, pp. 1249–1255, Aug. 2019.

[15] J. W. Bos and S. J. Friedberger, "Arithmetic considerations for isogeny-based cryptography," *IEEE Trans. Comput.*, vol. 68, no. 7, pp. 979–990, Jul. 2019.

[16] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. 7th Annu. Int. Cryptol. Conf. Adv. Cryptol.*, vol. 263, 1987, pp. 311–323.