A Fast and Efficient SIKE Co-Design: Coarse-Grained Reconfigurable Accelerators with Custom RISC-V Microcontroller on FPGA

Jing Tian, Bo Wu, Lang Feng, Haochen Zhang, and Zhongfeng Wang

School of Electronic Science and Engineering, Nanjing University, Nanjing, China,
{tianjing,flang,zfwang}@nju.edu.cn,wubo@smail.nju.edu.cn,zhanghc0624@sina.com

Abstract. This paper proposes a fast and efficient FPGA-based hardware-software co-design for the supersingular isogeny key encapsulation (SIKE) protocol controlled by a custom RISC-V processor. Firstly, we highly optimize the core unit, the polynomial-based field arithmetic logic unit (FALU), with the proposed fast convolution-like multiplier (FCM) to significantly reduce the resource consumption while still maintaining low latency and constant time for all the four SIKE parameters. Secondly, we pack the small isogeny and point operations in hardware, devise a coarse-grained reconfigurable hardware architecture (CGRHA) based on FALU as the co-processor, and apply it to the RISC-V core with customized instructions, effectively avoiding extra time consumption for the data exchange with the software side and meanwhile increasing flexibility. Finally, we code the hardware in SystemVerilog language and the software in C language and run experiments on FPGAs. In the co-processor implementation, the experiment results show that our design for the four SIKE parameters achieves 2.6-4.4x speedup and obtains comparable or better area-time product to or than the state-of-the-art. In the hardware-software co-design experiments, we still have the superiority in speed and only <10% of extra time is introduced by mutual communication.

Keywords: Supersingular isogeny Diffie-Hellman (SIDH) key exchange, elliptic curve cryptography (ECC), modular reduction, Montgomery representation, Barrett reduction, polynomial multiplication.

1 Introduction

As summarized by Bernstein and Lange in [BL17], most of the popularly-used cryptosystems would heavily suffer from large quantum computers by using Grover's algorithm [Gro96] or Shor's algorithm [Sho94]. The public-key cryptography like the Rivest-Shamir-Adleman (RSA) algorithm [RSA78] and Elliptic-Curve Cryptography (ECC) [Mil85] would be entirely destroyed by Shor's algorithm. Recently, much progress has been made in quantum circuits [Vu16, KMT+17, ZWD+20], which accelerates the establishment of Post-Quantum Cryptography (PQC). Several standardization bodies like the National Institute of Standards and Technology (NIST) and Internet Engineering Task Force (IETF) have joined in the evaluation for PQC standardization.

From 2016 till now, four rounds of the evaluation process of PQC have been completed by the NIST [CCJ+16], in which the Supersingular Isogeny Key Encapsulation (SIKE) protocol [ACC+17] still exists as an alternate candidate for public-key encryption and key-establishment (PKE) algorithms based on the Key-Encapsulation Mechanism (KEM) [HHK17]. SIKE, evolved from the Supersingular Isogeny Diffie-Hellman (SIDH) key exchange [JDF11], is the only isogeny-based protocol that adopts the ECC arithmetic

operations. It has the advantage of relatively short key sizes and can be well compatible with conventional ECC protocols like the elliptic-curve Diffie–Hellman (ECDH) key exchange [Mil85]. Unlike ECDH, which moves around different points on a fixed curve, SIKE moves around different points and different curves [ACC+17]. Large-degree isogeny computations between supersingular elliptic curves are adopted to achieve high-level post-quantum security, which causes high computational cost and long latency. Therefore, techniques to speed up SIKE are essential in PQC research.

1.1 Related Work

To improve the performance, many researchers have provided optimizations for the SIKE implementation in pure software [Jao11, AFJ14, AJK+16, CLN16, FHLOJRH17, ZSP+18, ACC+17, SLLH18, JAK18, AAK21, SAJA20, TWL+21, CFGR22] or pure hardware [KAKJ17, KAK18, KAEK+20, EAMK20, TWW21, FBSMBA21a, FBSMBA21b, EAMK21b, NOL+22b, FBSMB20]. On the pure software side, apart from the portable generic ones in C language, many implementations in assembly language are targeted at specific CPU or ARM architectures to obtain high performance or low power. The software implementations have an advantage over the hardware ones in flexibility but are at a disadvantage in speed, limited by the fixed word length and usage of parallelism. On the pure hardware side, the mainstream designs are implemented on high-end FPGA platforms and specifically accelerated for a certain SIKE parameter without consideration for flexibility. Most of them are dedicatedly optimized in the underlying field arithmetic, mainly for modular multiplication. Two well-known modular multiplication algorithms, namely the Montgomery algorithm [MP85] and the Barrett algorithm [Bar86], are widely used and studied for SIKE. In terms of the original version, the former is more efficient than the latter [TLW20]. Usually, the high-radix Montgomery multiplication algorithm [Oru95] and its architecture in [BP01] are adopted by many SIKE implementations to make a tradeoff between efficiency and resource utilization.

Recently, Tian *et al.* [TWL+21] proposed a low-complexity Barrett-based multiplication algorithm (UR-Barrett) for SIKE by leveraging the special form of the prime. They introduced a reciprocal with a small denominator for the SIKE prime and constructed a relatively large-degree polynomial structure with an unconventional radix. All the field arithmetic operations are then processed over the corresponding polynomial domain. In the UR-Barrett algorithm, the large integer multiplications are broken down into small ones, reducing the multiplicative complexity from quadratically to linearly in the polynomial order. It means that UR-Barrett has the potential to be superior over the best Montgomery one in complexity even though it is based on the Barrett scheme [TWL+21, TLW19]. Meanwhile, parallelism can also be improved when processing the independent coefficients. A feed-forward circuit for UR-Barrett is presented in [TWW21], which facilitates the implementation of SIKEp751 achieving the fastest one among all so far at the cost of relatively more resource consumption. In this paper, we take this scheme as our starting point and further optimize it for all the SIKE parameters for high speed and low complexity.

As for the hardware-software co-design (HSC-design) scheme, it is expected to combine the advantages of hardware and software platforms but is still at the beginning for the SIKE protocol [MLRB20, RFS20, EAMK21a, EKAK22]. In [MLRB20], Massolino *et al.* proposed the first HSC-design methodology in simulation level. They designed a compact and scalable MAC unit based on an FPGA platform and extracted the simulated data to a software environment to evaluate the co-design. Two modes for the MAC are provided, *i.e.*, the Carmela128 (abbr. S) and the Carmela256 (abbr. F). The corresponding SIKE experimental results show tradeoffs between flexibility and speed, either of which, however, is slower than almost all of the previous pure hardware designs. In [RFS20], Roy *et al.* for the first time, integrated the customized finite field unit into ARM and RISC-V architectures, respectively, and implemented the SIKE designs on an FPGA development

board and evaluated the performance. Because of the limitation in clock frequency (150MHz for the ARM and 25MHz for the RISC-V), the runtime results of both cannot be comparable to the ARM-based pure software implementations. In [EAMK21a], Elkhatib *et al.* also presented a HSC-design with RISC-V based on FPGA platforms. They selected a very lightweight RISC-V core that can only support some basic integer operations as the controller and adopted a single modular multiplier and a modular adder with a size of 752 bits as the hardware accelerator. This way, their clock frequency reaches 243.6MHz, much faster than the other two. The place-and-route results based on a Virtex-7 FPGA core show that their design has improvements in both time and area compared with the designs in [MLRB20]. In [EKAK22], they further optimized the clock frequency up to 303MHz to improve the timing performance and achieved the best area-time product (ATP) among previous works. However, the HSC-design is still much slower than most pure hardware implementations. It should be pointed out that the HSC-design in [RFS20] is the only one so far whose timing performance is evaluated based on an FPGA development board.

Regarding the HSC-design for SIKE, it has a great challenge and potential to make a better tradeoff among speed, area, and flexibility. The timing performance need still be improved with an acceptable area increase while maintaining flexibility.

1.2 Our Contribution and Paper Organization

The main contributions are summarized as follows:

1. We review the popularly used modular multiplication algorithms for SIKE and analyze their complexities in formula and numeric, respectively. The results show that the modified UR-Barrett (mUR-Barrett) evolved from [TWL+20] has the lowest computational complexity.

2. We take the polynomial-based field arithmetic logic unit (FALU) specifically designed for SIKEp751 in [TWW21] as the starting point and propose an advanced FALU supporting all the four SIKE parameters. The new FALU includes a modular multiplier and a modular adder/subtractor.

   (a) We present a general fast convolution-like multiplier (FCM) that is scalable and configurable for both integer and polynomial multiplications. The core idea is to reduce the complexity of multiplication in a divide-and-conquer manner like the Karatsuba algorithm, with an organized and lower-complexity way. We exhaustively adopt this architecture in serial and parallel for all the involved integer and polynomial multiplications to achieve better area efficiency.

   (b) By fully leveraging FCM, we propose a new constant-time modular multiplier compatible with all the four SIKE parameters in low complexity and small latency. The number of DSPs of the new modular multiplier is reduced by about 70%, and that of Slices is by about 40%, much lower hardware complexity compared with the multiplier in [TWW21]. At the same time, the interleaved latency is only three cycles, much smaller than other previous designs. In this way, we can still use a single modular multiplier for SIKE to achieve high performance.

   (c) For the modular adder and subtractor, we merge them and make them configurable for all modes and parameters. Additionally, only one stage of the pipeline is inserted for low latency.

3. We make the higher-level operations, *i.e.*, the small isogeny and point-addition operations, form a coarse-grained reconfigurable hardware architecture (CGRHA) in the hardware side to ease the communication cost. Note that it is not worthy to directly connect the proposed FALU with the software side since its latency is too

Table 1: Timing results of our design for the four SIKE parameters by using simulation on EDA and testing on an FPGA development board, respectively. Measured in cycle counts (rounded to 10^3).

SIKE Parameter	Key Gen.		Encaps.		Decaps.		Encaps. + Decaps.	
	simu.	test.	simu.	test.	simu.	test.	simu.	test.
SIKEp751	496	503	869	880	916	928	1,784	1,807
SIKEp610	389	411	790	826	773	813	1,563	1,639
SIKEp503	316	333	560	596	586	625	1,146	1,221
SIKEp434	278	290	485	497	525	527	1,010	1,024

small to cover the overhead of the communication. It can be realized by adding a program ROM module to save the necessary instructions. Meanwhile, all operations have constant execution time in any parameter settings.

4. We take CGRHA as a co-processor and integrate it into a powerful RISC-V platform. The overview of the proposed HSC-design can be illustrated in Figure 1. We test the area based on post-place and route and evaluate the time according to the Vivado simulation and the runtime of an FPGA development board, respectively, where the cycle counts are summarized in Table 1. Compared with the state-of-the-art, our design achieves the fastest speed for all four SIKE parameters and keeps high area efficiency.

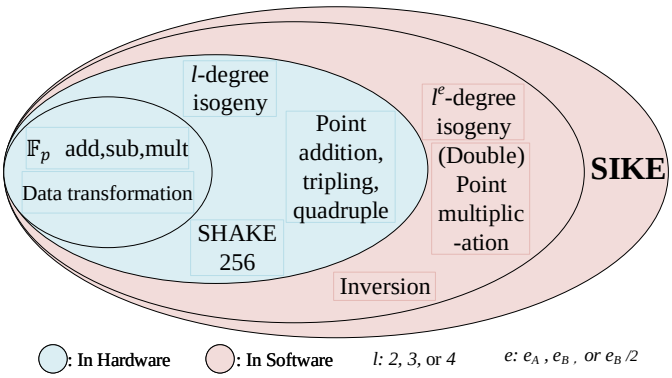

Figure 1: Overview of the proposed HSC-design of SIKE. The operations in the blue ellipses are processed in the pure hardware accelerator, and those in the pink are carried out by the software controller calling the required operations from the hardware. Note that the Data Transformation module on the hardware side is necessary for transforming the alternating data in the right format. The hash function is kept in hardware to accelerate this operation and reduce the data exchange between software and hardware.

The rest of this paper is organized as follows. Section 2 gives a brief review of the necessary mathematical foundations and the SIKE protocol. In addition, the efficient modular multiplication algorithms for SIKE are summarized and compared. Section 3 presents the details of the proposed CGRHA. The integration with RISC-V is provided in Section 4. Experimental results and comparisons are given in Section 5. Finally, Section 6 concludes the paper.

2 Preliminaries

In this section, we will first introduce the necessary mathematical foundations. Then, the SIKE protocol will be reviewed. Finally, the modular multiplication algorithms for SIKE will be summarized and analyzed.

2.1 Mathematical Foundations

The supersingular elliptic curve over F_{p^2} is usually used the Montgomery curve with the form as:

$$E/\mathsf{F}_{p^2} : Dy^2 = x^3 + Cx^2 + x, \tag{1}$$

where $C, D \quad \mathsf{F}_{p^2}$, $D(C^2 - 4) = 0$, and $p = f \cdot a^{e_A} b^{e_B} \pm 1$. The prime for SIKE usually is set as $p = 2^{e_A} 3^{e_B} - 1$. Two pairs of independent public points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ are selected from the curve E/F_{p^2} (simply denoted as E), which satisfy $P_A, Q_A = E[2^{e_A}]$ and $P_B, Q_B = E[3^{e_B}]$, respectively.

In the SIKE protocol, the isogeny $\varphi : E_1 \to E_2$ is defined as a non-constant rational map over F_{p^2}. In practical computing, a large-degree isogeny is resolved into many small-degree isogenies that are 2, 3 or 4-degree computed by the Vélu's formula [Vél71] iteratively. Assume a curve E over F_{p^2} and a point S of order l^e on E, where $l \quad \{2, 3, 4\}$ and $e \quad \{e_A, e_B, e_A/2\}$. The l^e-degree isogney $\varphi : E \to E\ S$ are computed as shown in Algorithm 1. It needs to iteratively compute the l-degree isogeny and the point multiplication e times.

Algorithm 1: l^e-Degree Isogney Computation [JAC+]

Input: A curve E and a ponit S.

1: $E_0 = E$, $S_0 = S$
2: **for** $i = 0 \to e - 1$ **do**
3: $\quad E_{i+1} = E_i / [l^{e-i-1}]S_i$
4: $\quad \varphi_i : E_i \to E_{i+1}$
5: $\quad S_{i+1} = \varphi_i(S_i)$
6: **end for**

Output: The isogeny $\varphi = \varphi_{e-1} : E \to E/\ S$.

Two isogeny algorithms, *i.e.*, the isogen algorithm for computing public keys and the isoex algorithm for establishing shared keys, are involved in the SIKE protocol. More details of the two algorithms can be referred to the provided documentation of [JAC+]. Both are mainly composed of Algorithm 1, where the point S is computed by a double-point multiplication based on the selected secret keys and public points or keys. For the isogen algorithm, the goal is to find the image of the public points over the isogenous curve $E/\ S$ as public keys. The isoex algorithm is to compute the j-invariant of $E/\ S$ as the shared keys, which is calculated as:

$$j(E/\ S\ ) = \frac{256(C^2 - 3)^3}{C^2 - 4}. \tag{2}$$

2.2 The SIKE Protocol

Assume that two parties, e.g., Alice and Bob, communicate with each other based on the SIKE protocol, and Alice sends a message to Bob. The computing process is shown in Algorithm 2, which consists of a tuple of there algorithms: **Key Generation**, **Encapsulation**, and **Encapsulation**.

Algorithm 2: SIKE Protocol [JAC+]

Input: Public parameters: E/F_{p^2}, $p = 2^{e_A}3^{e_B} - 1$, P_A, Q_A, P_B, and Q_B.

1: **Key Generation**: (Bob's round)
$sk_B = \text{random}\{0, 1, ..., 2^{\log_2 3^{e_B}} - 1\}$,
$pk_B = \text{isogen}(sk_B, P_B, Q_B, P_A, Q_A)$
$fm_B = \text{random}\{0, 1, ..., 2^M - 1\}$
Output: $\{sk_B,\ pk_B,\ fm_B\}$

2: **Encapsulation**: (Alice's round)
Input: $\{m_A,\ pk_B\}$
$sk_A = H(\{m_A, pk_B\}, e_A)$,
$pk_A = \text{isogen}(sk_A, P_A, Q_A, P_B, Q_B)$
$j = \text{isoex}(sk_A, pk_B)$,
$ss = H(j, M)$
$c_A = ss \quad m_A$,
$em_A = H(\{m_A, pk_A, c_A\}, K)$
Output: $\{pk_A,\ c_A,\ em_A\}$

3: **Decapsulation**: (Bob's round)
Input: $\{pk_A,\ sk_B,\ c_A,\ pk_B,\ fm_B\}$
$j = \text{isoex}(sk_B, pk_A)$,
$ss = H(j, M)$
$m_A = ss \quad c_A$,
$sk_A = H(\{m_A, pk_B\}, e_A)$
$pk_A = \text{isogen}(sk_A, P_A, Q_A, P_B, Q_B)$
if $(pk_A == pk_A)$ **then**
$em_B = H(\{m_A, pk_A, c_A\}, K)$
else
$em_B = H(\{fm_B, pk_A, c_A\}, K)$
Output: em_B

At the beginning, Bob uses the **Key Generation** algorithm to generate his secret and public keys, and discloses the public key. Meanwhile, he produces a random number for use in the later. Then, Alice adopts the **Encapsulation** algorithm to generate her keys in an encapsulating way [HHK17] and discloses her public key. Based the shared key,*i.e.* the j-invariant, she encrypts her plaintext and sends the ciphertext to Bob. In addition, she keeps a second shared key em_A for verification. At the last step, Bob decrypts the ciphertext with the shared key, recovers Alice's keys, and checks her message using the **Encapsulation** algorithm. He discloses the calculated value em_B to notify Alice whether their communication is attacked or not. This protocol has been proven IND-CCA secure.

From Algorithm 2, it can be seen that the SIKE protocol requires five large-degree isogeny computations, six Hash functions, two random-number operations, and two XOR operations. In existing hardware implementations for SIKE, the random numbers are taken as given. The main focus is on the isogeny computations as they take up most of the whole time. Note that the isogeny computations are over a finite field, dominated by modular multiplications. We will introduce and compare several efficient modular multiplication algorithms for SIKE in the following.

2.3 Modular Multiplication Algorithms for SIKE

The Montgomery algorithm [MP85] and Barrett algorithm [Bar86] are two widely used modular multiplication algorithms since both of them remove the division and only involve multiplication and addition operations.

Algorithm 3: Montgomery Modular Multiplication Algorithm [MP85]

Input: $p,\ A,\ B$.
Modulus p with $2^{N-1} < p < 2^N$;
Precompute $p' = (-p^{-1}) \bmod 2^N$;
Operands $A, B \quad [0, p)$.
1: $C = A \cdot B$
2: $t = ((C \mod R) \cdot p') \mod 2^N$
3: $r = (C + t \cdot p)/2^N$
4: **if** $r \geq p$ **then**
5: $\quad r = r - p$
6: **end if**
Output: The residue $r = A \times B \times 2^{-N} \bmod p$.

Algorithm 4: Barrett Modular Multiplication Algorithm [Bar86]

Input: $p,\ A,\ B$.
Modulus p with $2^{N-1} < p < 2^N$;
Precompute $\lambda = \lfloor 2^{2N}/p \rfloor$;
Operands $A, B \quad [0, p)$.
1: $C = A \cdot B$
2: $q = \frac{C \cdot \lambda}{2^{2N}}$
3: $r = C - q \cdot p$
4: **if** $r \geq p$ **then**
5: $\quad r = r - p, q = q + 1$
6: **end if**
Output: The quotient $q = \lfloor C/p \rfloor$ and the remainder $r = C \bmod p$.

Their procedures are summarized in Algorithms 3 and 4, respectively, where the modulus

p is an arbitrary number with the data width of N. We can see that the Montgomery takes 1 $2N + 2N$, 1 $N + N$ additions and 3 $N \times N$ multiplications, while the Barrett requires 1 $2N + 2N$, 1 $N + N$, 1 $N + 1$ additions and 1 $2N \times N$, 2 $N \times N$ multiplications. The Barrett is more complex than the Montgomery. However, the Barrett can directly obtain the remainder and the quotient, while the Montgomery needs extra operations for the domain transformation. Moreover, the complexity of the Barrett can be further reduced if the input operands are constrained much smaller than p, while that of the Montgomery is consistent with p. Hence, they can be effectively applied to different scenarios. For SIKE, the Montgomery one is usually preferred.

Algorithm 5: High-Radix Montgomery (HR-Montgomery) Modular Multiplication Algorithm [Oru95]

Input: p, A, B.
Modulus $p = \sum_{i=0}^{m-1} p_i \left(2^k\right)^i$, $p_i \quad \{0, 1, ..., 2^k - 1\}$, $2^{N-1} < p < 2^N$;
Precompute $\tilde{p} = (-p^{-1} \bmod 2^k)p = \sum_{i=0}^{m} \tilde{p}_i \left(2^k\right)^i$, $\tilde{p}_i \quad \{0, 1, ..., 2^k - 1\}$;
Operands $A = \sum_{i=0}^{m+2} a_i \left(2^k\right)^i$, $B = \sum_{i=0}^{m+1} b_i \left(2^k\right)^i$, where $a_i, b_i \quad \{0, 1, ..., 2^k - 1\}$, $a_{m+2} = 0$.
Requirements: $A, B < 2\tilde{p}$ and $4\tilde{p} < 2^{km}$.

1: $S_0 = 0$
2: **for** $0 \le i \le m + 2$ **do**
3: $\quad q_i = S_i \bmod 2^k$
4: $\quad S_{i+1} = (S_i + q_i \cdot \tilde{p}) / 2^k + a_i \cdot B$
5: **end for**

Output: : The residue $S_{m+3} = A \times B \times 2^{-km} \bmod p$.

To make a good trade-off between speed and area, the high-radix Montgomery (HR-Montgomery) modular multiplication algorithm [Oru95] is adopted for SIDH/SIKE by the SIKE team [JAC+] and extensively studied in existing works [KAKJ17, KAK18, KAEK+20, EAMK20, FBSMBA21a, FBSMBA21b, EAMK21b, FBSMB20]. The flow of this algorithm is shown in Algorithm 5, where the modulus and operands all are represented in 2^k-radix polynomials with the highest degree of $(m + 2)$. The integer multiplication and reduction are interleaved and computed in $(m + 3)$ iterations. The parameters p and p are merged together as $\tilde{p}$, which needs to satisfy $4\tilde{p} < 2^{km}$. It means that, Generally, it requires $k + N + 2 < km$. For SIKE, as the formula $-p^{-1} \bmod 2^k = 1$ when $k < e_A$, it has $\tilde{p} = p$ and the requirement becomes $N < km - 2$. In this situation, this algorithm takes about $(m + 3)$ $(k + N + m) + (k + N)$ and $(m + 3)$ $(k + m) + (k + N)$ additions, and $2(m + 3)$ $k \times N$ multiplications. To reduce the complexity, the km should be selected close to $N + 3$ as much as possible.

By leveraging the prime form of supersingular isogeny elliptic curves, Tian *et al.* presented a low-complexity Barrett-based multiplication algorithm (UR-Barrett) for SIKE [TWL+21]. The operands are both represented in R-radix polynomials with a degree of $(n - 1)$. The main idea is to transform the prime as $p = 2^{e_A} 3^{e_B} - 1 = \alpha R^n - 1$ and reduce the partial sums based on R in sequence. The most difficult thing is constructing a small factor α to make the power n large while keeping the modulus p prime. In the earlier works [KRVV16, BF18, LNL+19, TLW20], researchers set α to 2 and therefore n to 2. As demonstrated in [TWL+21], such algorithm is inferior to the optimized multi-precision Montgomery algorithm [FHLOJRH17]. Moreover, such settings cannot apply to most parameters of SIKE. In UR-Barrett, α is cleverly set as a fraction where the numerator equals 1 and the denominator is related to 2 or 3. With this setting, a relatively large n can be easily found. For example, in [TWL+21], when α is set to 1/2 or 1/3, the orders n for all the SIKE parameters are no less than 6. We propose the modified UR-Barrett

Algorithm 6: Modified UR-Barrett (mUR-Barrett) Modular Multiplication Algorithm for SIKE

Input: $p,\ A,\ B$.
Modulus $p = 2^{e_A}3^{e_B} - 1 = \alpha R^n - 1 = (\alpha R - 1)R^{n-1} + \sum_{i=0}^{n-2}(R-1)R^i < 2^{wn}$;
Operands $A, B \quad [0, p)$, where $A = \sum_{i=0}^{n-1} a_i R^i,\ B = \sum_{i=0}^{n-1} b_i R^i$,
$a_i, b_i \quad \{0, 1, ..., R-1\},\ a_{n-1}, b_{n-1} \quad \{0, 1, ..., \alpha R - 1\}$.
Partial MAC:
1: **for** $0 \le i \le n-1$ **do**
2: $\quad S_i = \sum_{j=0}^{i} a_j b_{i-j} + (\sum_{j=i+1}^{n-1} a_j b_{i-j+n}) \cdot \frac{1}{\alpha}$
3: **end for**
Reduction:
4: $(q_{n-2},\ S_{n-2}) = \mathrm{IBR}(S_{n-2},\ R,\ \gamma)$
5: $(q_{-1},\ S_{n-1}) = \mathrm{IBR}(S_{n-1} + q_{n-2},\ \alpha R,\ \gamma)$
6: **for** $0 \le i \le n-2$ **do**
7: $\quad (q_i,\ S_i) = \mathrm{IBR}(S_i + q_{i-1},\ R,\ \gamma)$
8: **end for**
9: $S_{n-1} = S_{n-1} + q_{n-2}$
10: **if** $S_{n-1} > \alpha R - 1$ **then**
11: $\quad S_{n-1} = S_{n-1} - \alpha R,\ S_0 = S_0 + 1,$
12: **end if**
Output: : The residue $\sum_{i=0}^{n-1} S_i \cdot R^i \equiv A \times B \bmod p$.

Algorithm 7: Improved Barrett Reduction (IBR) for Modulus $R = 2^x 3^y$ [TWL+21]

Input: $S,\ R,\ \gamma$.
Operand $S \quad [0, 2^{2w+\gamma})$;
Modulus $R = 2^x 3^y$, where $w_1 = x$, $w_2 = \lfloor \log_2(3^y) \rfloor$, $w_1 + w_2 = w$, $w_1 \approx w_2$;
Precompute $\lambda = \lfloor 2^{2w+\gamma+1}/R \rfloor$.
1: $t = \lfloor S/2^{w_1} \rfloor,\ s = S \bmod 2^{w_1}$
2: $q = \lfloor \frac{\frac{t}{2^{w_2-2}} \cdot \lambda}{2^{w+\gamma+3}} \rfloor$
3: $t_1 = (q \bmod 2^{w_2+1}) \cdot 3^y$
4: $r = ((t \bmod 2^{w_2+1}) - (t_1 \bmod 2^{w_2+1})) \bmod 2^{w_2+1}$
5: **if** $r \ge 3^y$ **then**
6: $\quad r = r - 3^y,\ q = q + 1$
7: **end if**
8: $r = \{r, s\}$
Output: The quotient $q = \lfloor S/R \rfloor$, and the remainder $r = S \bmod R$.

(mUR-Barrett) modular multiplication algorithm as shown in Algorithm 6, where the improved Barrett Reduction (IBR) function is presented in Algorithm 7. It must be noted that the reductions for the partial multiply–accumulates (MAC) are in cyclic and the cyclic reduction can be started at any term. In [TWL+21], the reduction is started from the constant term. Though a lazy reduction is applied, one loop with $(n-1)$ subtractions and additions is needed for adjustment. Hence, we here move the starting point to the $(n-2)$-th term, and then only one subtraction and one addition are used, as shown in Steps 10-12 of Algorithm 6. It should be pointed out that $(n+1)$ IBR functions are used and the complexity of IBR is related to the value of γ. For the first round of IBR reductions, γ's are small positive integers. For the second reduction of the $(n-2)$-th term, γ should be a very small negative integers. For simplicity, We make γ as a constant positive integer all the time, which is much smaller than w. In general, the addition operations in mUR-Barrett include $(3n-4)(n-1)/2$ $2w+2w$, n $(2w+\gamma)+(2w+\gamma)$, 1 $w+w$, 1 $w+\gamma$, $(n+2)$ $w+1$, and$(2n+2)$ $w/2+w/2$. The multiplications are n^2 $w \times w$, $(n+1)$ $(w+\gamma+2) \times (w+\gamma+2)$, and $(n+1)$ $w/2 \times w/2$.

Table 2: Complexity analysis and comparison for modular multiplication algorithms.

Algorithms	Mont. [MP85]	Barrett [Bar86]	HR-Mont. [BP01]	mUR-Barrett (proposed)
Addition operations	1 $2N+2N$ 1 $N+N$	1 $2N+2N$ 1 $N+N$ 1 $N+1$	$(m+3)$ $(k+N+m)+(k+N)$ $(m+3)$ $(k+m)+(k+N)$	$\frac{(3n-4)(n-1)}{2}$ $2w+2w$ n $(2w+\gamma)+(2w+\gamma)$ 1 $w+w$ 1 $w+\gamma$ $(n+2)$ $w+1$ $(2n+2)$ $w/2+w/2$
Multiplication operations	3 $N \times N$	1 $2N \times N$ 2 $N \times N$	$2(m+3)$ $k \times N$	n^2 $w \times w$ $(n+1)$ $(w+\gamma+2) \times (w+\gamma+2)$ $(n+1)$ $w/2 \times w/2$
Complexity	$3N^2+3N$	$4N^2+4N$	$2(m+3)kN$ $+(m+3)(2k+2N+m)$	$(n^2+(n+1)/4)w^2$ $+(n+1)(w+\gamma+2)^2$ $+(3n^2-5n+9)w+n(2w+\gamma)$
SIKEp751 ($N=751$)	1,694,256 *1.00*	2,259,008 *1.33*	$(m=32, k=24)^{\dagger}$ 1,317,050 *0.78*	$(n=12, w=63, \gamma=6)^{\ddagger}$ 675,555 0.40
SIKEp610 ($N=610$)	1,118,130 *1.00*	1,490,840 *1.33*	$(m=26, k=24)$ 886,646 *0.79*	$(n=6, w=102, \gamma=4)$ 484,521 0.43
SIKEp503 ($N=503$)	760,536 *1.00*	1,014,048 *1.33*	$(m=22, k=23)$ 605,300 *0.80*	$(n=10, w=51, \gamma=5)$ 318,536 0.42
SIKEp434 ($N=434$)	566,370 *1.00*	755,160 *1.33*	$(m=20, k=22)$ 460,644 *0.81*	$(n=6, w=73, \gamma=4)$ 252,108 0.45

Count the multiplications quadratically and additions linearly.
† Parameters (m and k) are referred to the newest SIKE project [JAC+] provided by the SIKE team.
‡ Parameters (n, w, and γ) are referred to the project of [TWL+21].

We summarize the addition and multiplication operations of the four aforementioned modular multiplication algorithms as shown in Table 2. The complexities are also evaluated in formulas, where we count the multiplications quadratically and the additions linearly to make them fairer. It should be noted that the additions are aligned to the worst case in the counting, such as $w+1$ to $w+w$.

To be more intuitive, we also list the numerical results for the four SIKE parameters in the table. The variables of HR-Montgomery are referred to in the newest SIKE project [JAC+] provided by the SIKE team and those of mUR-Barrett are referred to in the project of [TWL+21]. Clearly, mUR-Barrett has the lowest computational complexity in all parameters, and the original Barrett [Bar86] is the highest. Being normalized to the original Montgomery's [MP85], we can see that the computational complexity of HR-Montgomery [Oru95] is reduced by about 20% in all cases thanks to the prime form of SIKE. Compared to HR-Montgomery, the complexity of mUR-Barrett is further decreased by nearly 50% in all situations. Note that less complexity means less time or resources

required for the implementation. So, we can conclude that the mUR-Barrett modular algorithm has the potential to obtain a better tradeoff in performance than others. In the following, we will further reduce the complexity, optimize the tradeoff between speed and area, and achieve flexibility for all four SIKE parameters.

3 Proposed Coarse-Grained Reconfigurable Hardware Architectures for SIKE

3.1 Top Level of Hardware Architecture: CGRHA

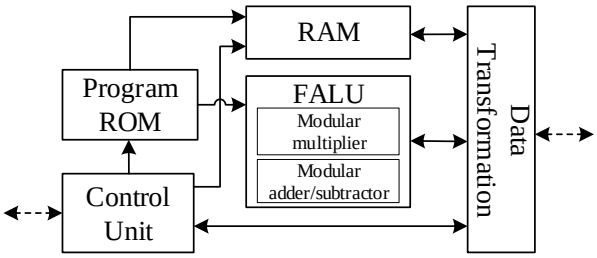

Figure 2: Top level of hardware architecture for SIKE: CGRHA.

The top level of hardware architecture for SIKE named CGRHA is shown in Figure 2, where the dashed lines denote the connections with CPU. It includes five modules: FALU, RAM, Program ROM, Control Unit, and Data Transformation. FALU is used to compute the basic field arithmetic over F_p, made up of the modular multiplier and modular adder/subtractor. RAM is to read or write the calculated or initial data dynamically. Program ROM is to save the computing flows of small isogeny computations and point operations in instruction form (ROM instruction). Control Unit is for the interaction between software and hardware. Data Transformation is to translate data into available forms for FALU, RAM, and the software side. The computing process can be commonly summarized as follows.

In the beginning, the initial data and configurations from CPU are written into RAM. Then, Program ROM provides some ROM instructions to FALU to compute specific small isogeny computations or point operations. Finally, those coarse-grained results are sent back to CPU and reconfigured for large isogeny computations or point multiplications. Beginnings and ends are all supervised by Control Unit. In the following, we will mainly introduce the proposed hardware modules, *i.e.*, FALU and Data Transformation. The software and interactive process will be detailed in the next section.

3.2 Modular Multiplier of FALU

As stated above, the modular multiplications are the most time-consumed, so we present a high-efficient modular multiplier which consists of the partial MAC unit and the reduction unit. Before introducing the modular multiplier, we present the proposed general fast convolution-like multiplier (FCM) adopted by the involved integer and polynomial multipliers.

3.2.1 A General Fast Convolution-Like Multiplier (FCM)

In [TWW21], the authors adopted a 6-point FCM to reduce the complexity of the partial MAC. In this paper, we extend this method into a general one and exhaustively apply it to the whole modular multiplier to achieve better area efficiency.

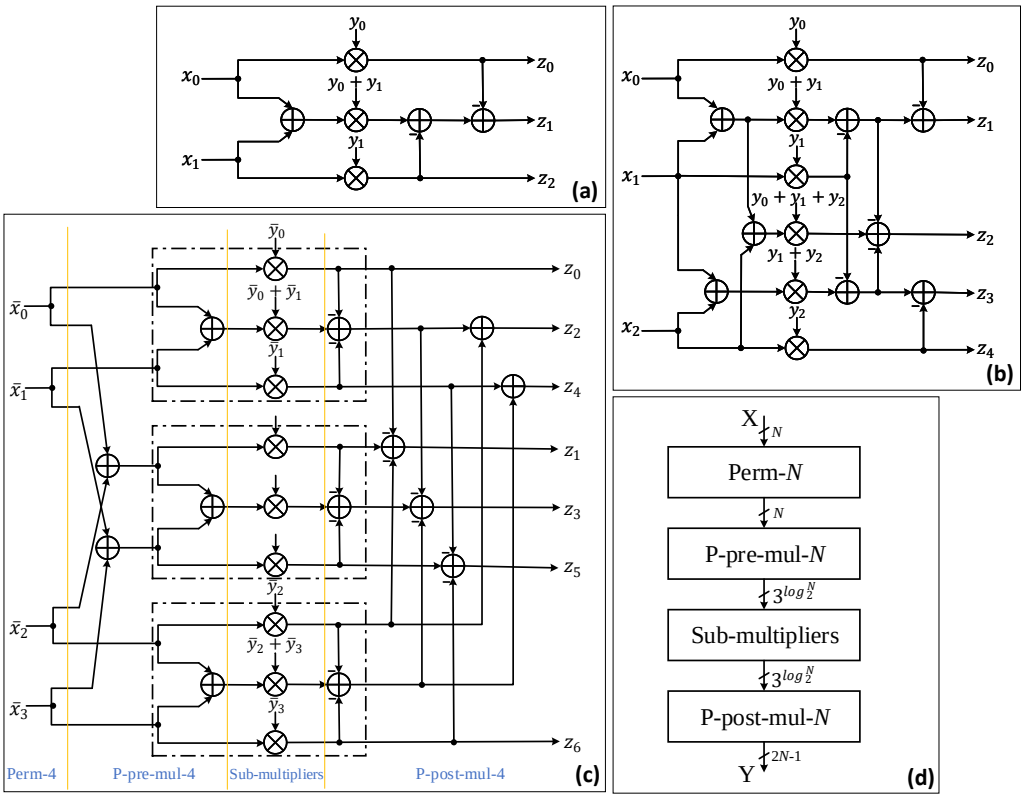

Figure 3: FCMs' architectures: (a) 2-point FCM, (b) 3-point FCM, (c) 4-point FCM, and (d) N-point FCM.

Figures 3(a)-(d) show the architectures of 2-, 3-, 4-, and N-point FCMs, respectively, where N denotes the number of coefficients for a polynomial or the number of segments for an integer. The core idea is to reduce the number of sub-multipliers with an increase of additions, *i.e.*, replacing strong operations with weak operations, as the methods used for convolutions [Par07]. Actually, the MAC operations of convolution and multiplication are the same. However, the fast convolution methods cannot be directly used for multiplications as their outputs are quite different. For the convolution, N outputs are required to be aligned by delay units, while for the multiplication, the number of outputs is set to $2N - 1$ without such alignment. As shown in Figures 3(a) and (b), We remove the delay units, unfold the outputs, and arrange the subscripts in ascending order. Those outputs are the final results of a polynomial multiplication or the partial sums of an integer multiplication. It can be seen that the complexities of multiplication are reduced to 3/4 and 2/3, respectively. Such a method can be further directly generalized to high-point multiplications.

We have found that the nested method for a 4-point shown in Figure 3(c) is more efficient than the direct method. The order of inputs $x_i (0 \le i < N)$ need be permuted, which is computed as $\bar{x}_i (0 \le i < N)$ by using the depth-first recursive function of a binary tree presented in Algorithm 8, where $N = 4$ in this case. From top to bottom, we use three 2-point FCMs denoted as the even kernel, cross kernel, and odd kernel. The 6-point FCM can be obtained by replacing the three 2-point kernels with three 3-point kernels. Similarly, the high-level nested architecture can be obtained by substituting the three kernels. Moreover, we can find another interesting fact that the even outputs

Algorithm 8: Depth-First Recursive Function with the Binary Tree Model for the Input Permutation

Input: The maximum depth $log_2 N$, and the input vector x_i with $i \quad \{0, 1, ..., N-1\}$.

```
Initialization: k = 0, depth = 0, and index = 0.
Perm(depth, index)
if depth < log2N then
    Perm(depth + 1, index)
    Perm(depth + 1, index + 2^depth)
else
    x̄_k = x_index, k = k + 1
end if
```

Output: $\bar{x}_i$ with $i \quad \{0, 1, ..., N-1\}$.

$z_{2i}(0 < i < N-1)$ are computed by directly adding the outputs of the even kernel and odd kernel, and the odd outputs $z_{(2i+1)}, (0 \le i < N-1)$ are calculated by using the outputs of the cross kernel correspondingly minus those of even and odd kernels.

As shown in Figure 3(c), we divide the FCM architecture into four parts: Perm-4, P-pre-mul-4, Sub-multipliers, and P-post-mul-4, where 'P' means parallel, separated from the serial architectures. Except Perm-4, P-pre-mul-4 and P-post-mul-4 are made up of additions before and after the sub-multipliers. Based on this partition method, the schematic diagram of N-point FCM is shown in Figure 3(d), where '4' is replaced by 'N'. Assume that N is a power of 2 and $u = \log_2 N$. We can obtain that the number of adders of P-pre-mul-4 is $\sum_{i=0}^{u-1}(2^i 3^{u-1-i})$, that of P-post-mul-4 is $\sum_{i=0}^{u-1}((6 \times 2^i - 4)3^{u-1-i})$, and the number of sub-multipliers is 3^u. It should be noted that if N is equal or close to $2^{\log_2 N}$, selecting the 2-point FCM as the basic kernel and being nested $\log_2 N$ times would be the best choice.

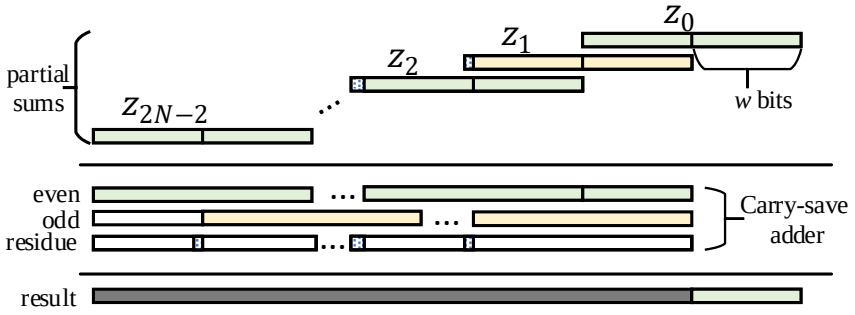

Figure 4: Sum-up module for an integer multiplier with a $(2N-1)w$-bit carry-save adder.

For the final output of an integer multiplier, we use a $(2N-1)w$-bit carry-save adder, where the three operands are obtained by aligning the even partial sums, odd partial sums, and their residues, respectively, as shown in Figure 4.

We propose a serial-parallel hybrid architecture by folding part of the layers to make a good tradeoff between time and area. As shown in Figure 5, a case study of serial-parallel hybrid architecture is presented for an 8-point FCM using a 4-point FCM in three iterations. The adders in the pre- and post-mul-8-4 are possibly reused to reduce the area further. Similar architectures can be applied to others with different configurations.

It should be pointed out that our method is somewhat similar to the recursive Karatsuba method by using the idea of divide-and-conquer. We call it a fast convolution-

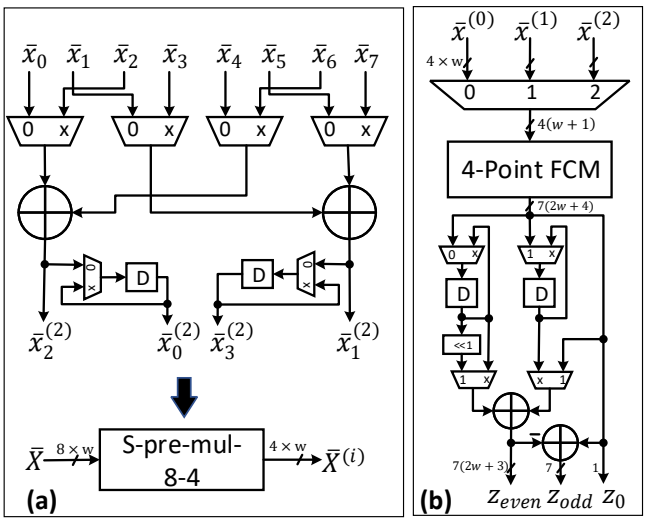

Figure 5: A case study of serial-parallel hybrid architectures: (a) the architecture of serial pre-mul-8-4 and (b) the architecture of 4-point FCM and serial post-mul-8-4.

like method because we were firstly inspired by the low-complexity cascading structure in [TWW21] which leverages the method used for convolutions [Par07]. The Karatsuba algorithm was firstly proposed by Karatsuba and Ofman in [KO62]. This method was generalized for polynomial multiplications by Weimerskirch and Paar in [WP06]. Usually, previous hardware designs for polynomial and integer multiplications are separately considered [JMEH02, vzGS05, MZB+08, FSGL10, FH15]. For the integer multiplier, the adders for the combination of each layer are needed after the sub-multipliers (e.g., Figure 8 of [TLW20]), while the polynomial multiplier does not need. In this paper, we carefully rearrange the inputs and outputs, postpone the combination in each stage of the integer multiplier, and find a method to combine the polynomial and integer multipliers together. Actually, most of their circuits are the same. We only need to apply an extra carry-save adder for the final result of the integer multiplier.

3.2.2 Partial MAC Unit for SIKE

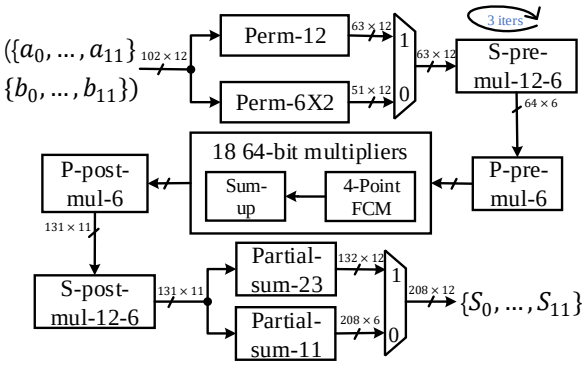

Figure 6: Architecture of partial MAC unit for the modular multiplier.

In Figure 6, the partial MAC unit is proposed to implement Steps 1-3 of Algorithm 6, where the four SIKE parameters (shown in the leftmost column of Table 3) all are covered.

The data width and the number of inputs are both set to the largest values, *i.e.*, 102 and 12, respectively. We adopt the FCM architecture for both the polynomial and integer multiplications. Since the degrees of the four constructed polynomials are not exactly the same, we divide them into two categories controlled by a signal *mode*: one includes SIKEp751 and SIKEp503 with the setting of $n = 12$ for $mode = 1$; the other covers SIKEp610 and SIKEp434 with $n = 6$ for $mode = 0$. To further reduce the complexity, the polynomial multiplication and integer multiplication are interleaved, whose correctness is demonstrated in Appendix A. As shown in Figure 6, the polynomial and integer multiplications are mixed, sharing the same multipliers resource, and the serial processing and parallel processing are hybrid. The number of 64-bit sub-multipliers is reduced from $12 \times 12 = 144$ to $144 \times 1/4 \times 3/4 \times 2/3 = 18$. Moreover, every 64-bit sub-multiplier is processed by a 4-point 16-bit FCM, further reducing the complexity by 7/16. Hence, this module takes about $18 \times 9 = 162$ DSPs. We will give more details of the flow in the following.

At first, the two groups of input coefficients are permuted in two ways. Perm-12 is to permute the 12 63-bit coefficients with three times of nesting, where the basic kernel is a 3-point 63-bit FCM, nested by the 2-point FCM two times. Perm-6X2 is to permute the first 6 102-bit coefficients with two times of nesting and split every permuted coefficient into 2 51-bit data, forming 12 51-bits data. After the permutation, the 12 63-bit variables are handled by two pre-mul modules. The first one is for in serial, leading to three iterative computations, and the second one is for in parallel. Then, every 18 64-bit data are simultaneously input into the multipliers module, which comprises 18 4-point FCMs and the corresponding Sum-up modules. Next, the corresponding parallel and serial post-mul modules are to deal with the outputs of the sub-multipliers. After three iterations, all candidates of partial MAC are ready. Finally, we adopt the rules of Algorithm 6 and the deduced function in Appendix A to compute the final results in two modes, respectively. In short, the partial MAC of all the four SIKE parameters can be computed by this unit with a constant time and meanwhile keeping a low complexity and low latency.

3.2.3 Reduction Unit for SIKE

To obtain a low-latency design, we still adopt the two-stage version of UR-Barrett in Algorithm 7 of [TWW21] to implement our reduction unit, where normal IBRs and mini IBRs (mIBRs) are considered. It should be noted that a relatively large of parameters are involved since the four different forms of prime need to be simultaneously considered. We still try our best to make resources shared and reused. Table 3 shows the used multiplications and data widths of the main arguments of the IBR and mIBR functions for the four SIKE parameters, where mul_1 or mul_{1s} corresponds to the multiplication operation in Step 2 and mul_2 or mul_{2s} to Step 3 of Algorithm 7. They are constant multiplications, but we use regular multipliers to match different parameters.

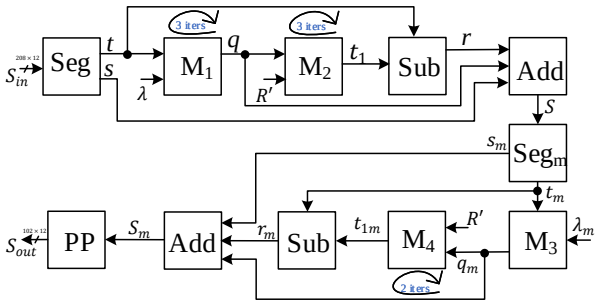

Figure 7: Architecture of reduction unit for the modular multiplier.

Table 3: Multiplications and data widths of the main arguments of the IBR and mIBR functions for the four SIKE parameters.

SIKE Parameters	IBR					mIBR				
	w_S	y	w_λ	mul_1	mul_2	w_{Sm}	y_m	$w_{\lambda m}$	mul_{1m}	mul_{2m}
SIKEp751 ($2^{372}3^{239}-1$, 12×63, $\alpha=1/3$)	132	6	71	71×71	33×32	71	-55	10	10×10	9×32
SIKEp610 ($2^{305}3^{192}-1$, 6×102, $\alpha=1/2$)	208	4	108	108×108	52×51	108	-96	8	8×8	7×51
SIKEp503 ($2^{250}3^{159}-1$, 10×51, $\alpha=1/3$)	107	5	58	58×58	27×26	58	-44	9	9×9	8×26
SIKEp434 ($2^{216}3^{137}-1$, 6×73, $\alpha=1/3$)	150	4	79	79×79	38×37	79	-67	8	8×8	7×37

Figure 7 is the proposed architecture of the reduction unit, where the data widths of intermediate variables are omitted for clarity. The Seg and Seg_m modules are used to divide the input into two segments as Step 1 of Algorithm 7. Sub is to implement Step 4 of Algorithm 7. Add is used for plus the quotients and residues. Both of them are instantiated the same in the IBR and mIBR stages. PP is to adjust the final coefficients into the correct ranges, whose architecture is referred to in the design of *Post_Process* in Figure 9 of [TWW21].

The most complex modules, M_1, M_2, M_3, and M_4, are the corresponding multipliers, which are carefully designed respectively. To satisfy all the parameters, M_1 should contain 12 71×71 and 6 108×108 multipliers. Similar to the method for the partial MAC unit, we use the interleaved and hybrid architecture for M_1 to reduce resource consumption. We align the 12 multipliers into 6 142×142 multipliers and adopt 6 72×72 multipliers for M_1 to compute the results in three iterations, where each 72-bit multiplier is implemented by a 4-point 18-bit FCM. For M_2, we use 6 34×34 multipliers in three iterations and a 2-point 17-bit FCM for each multiplier. M_3 is directly devised with 12 10×10 multipliers. For M_4, we use 6 9×34 multipliers in two iterations. It should be noted that the interleaved latency of the proposed modular multiplier is three cycles whether we insert pipelines or not, as there are no feedback signals in this design. We only need to pick up the right results in the right cycles.

3.3 Modular Adder/Subtractor of FALU

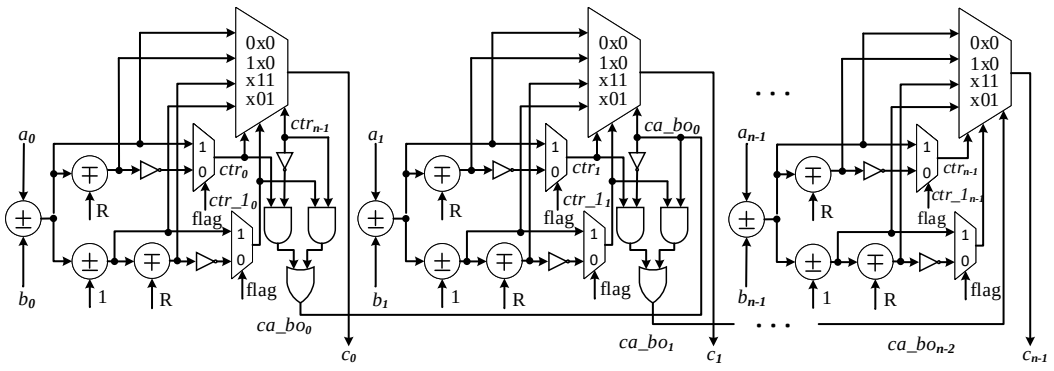

Figure 8: Architecture for the modular adder and subtractor.

In [TWW21], the modular adder and subtractor are separately devised. In this paper,

we merge them to reduce the resource. Figure 8 shows the proposed architecture, where the *flag* signal is used to select the two patterns with '1' for modular adder and '0' for modular subtractor. It can be seen that except the $(n-1)$ 1-bit *ca_bo* signals, the '$\pm$' and '*mp*' operations all are independently computed in parallel. Those operations are implemented by the Adder/Subtractor IPs where 1 LUT completes 2 bits in general. To further reduce the LUT resources, two kinds of IPs are adopted, *i.e.*, the $104+104$ and $65+65$. The former is used for the first six groups and the latter for the rests. The highest bits are the sign bits used as the control signals.

3.4 Data Transformation

The Data Transformation module implements four conversions: RAM to software (R2S), polynomial format to integer format (P2I), data from FALU to RAM (F2R), and RAM to FALU (R2F). It should be noted that all of them are not resource-intensive. R2S is to transform a 756-bit integer into 12 64-bit words for a 64-bit CPU by uniformly slicing the integer into 12 pieces. P2I transforms the data from the polynomial format back to the integer format. It is almost the same as the MU2NM architecture proposed in [TWW21] with some adjustments for the data widths and parameters. Meanwhile, the involved 102-bit multiplier is also replaced with FCM to reduce the complexity.

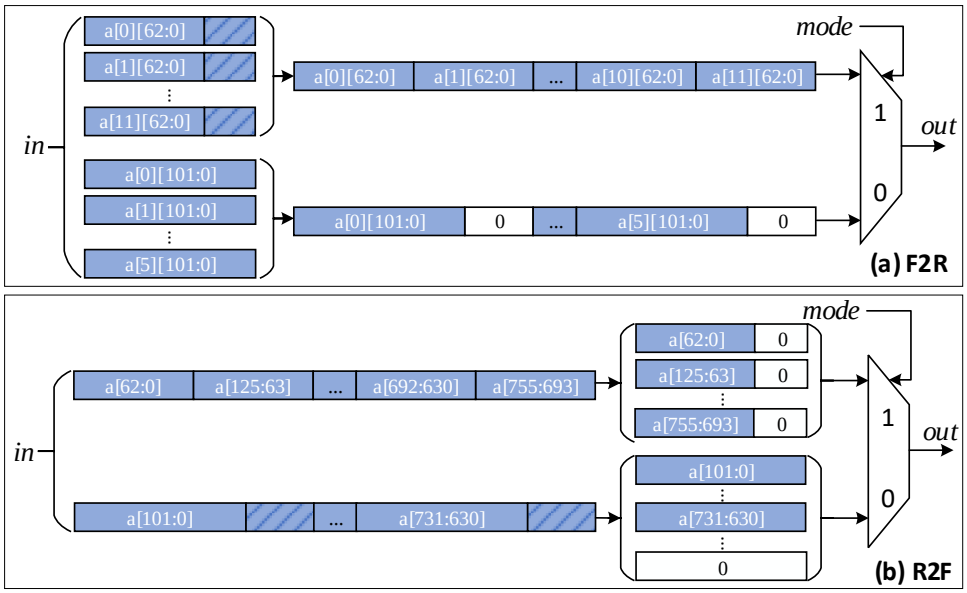

Figure 9: Procedures of F2R and R2F.

Figure 9 depicts the procedures of F2R and R2F. As shown in Figure 9 (a), F2R is used to transform the 12 102-bit coefficients ($a[0]$ã$[11]$) into a 756-bit integer saving in the RAM, where two modes are considered. When $mode = 1$, SIKEp751 and SIKEp503 are selected; otherwise, SIKEp610 and SIKEp434 are chosen. For the former, the 63 LSBs of each coefficient are picked and aligned; for the latter, only the first six coefficients are adopted and aligned, with zeros inserted. R2F is the inverse process as shown in Figure 9 (b), where a 756-bit integer is transformed into a vector with a width of 102 bits and a depth of 12. The corresponding two modes are also needed.

4 RISC-V Integration

In this section, we integrate CGRHA into RISC-V platform as a co-processor to realize high-speed and flexible SIKE co-design. RISC-V is a popular open source Instruction Set Architecture (ISA) standard designed in 2010 [RIS10], based on which various processors are proposed for research or industrial developments. Our platform is based on a powerful open source RISC-V SoC project called Rocket Chip [AAB+16].

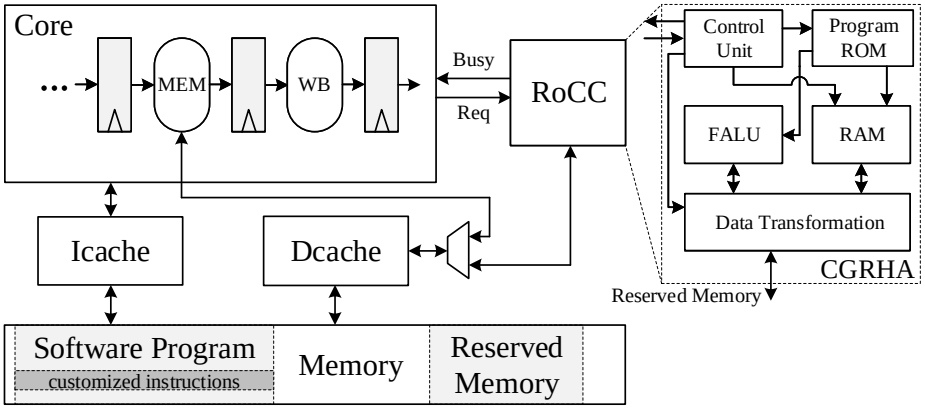

Figure 10: The proposed RISC-V architecture for SIKE. ("MEM" and "WB" represent the last two pipeline stages "memory access" and "writeback", respectively.)

Rocket Chip contains a processor core with a 5-stage pipeline, and a customizable co-processor named Rocket Custom Coprocessor (RoCC), where the proposed CGRHA is implemented. RoCC can be controlled by the customizable instructions in RISC-V ISA, and thus, the platform provides users with the flexibility to control CGRHA by writing software programs with the customized instructions.The RISC-V architecture with efficient and flexible hardware-software co-design is proposed and shown in Figure 10.

There are three main parts of the proposed system, which are listed as follows.

- **Customized Instructions:** To obtain the flexibility for supporting various SIKE parameters, customized instructions on the software side need to be designed. However, how to find the best tradeoff between flexibility and speed is challenging.
- **Software Program and RISC-V Core:** Given the customized instructions, the users can write software programs to conduct SIKE calculations with different parameters. The RISC-V core sends SIKE calculation requests to CGHRA according to users' programs.
- **Customized RoCC:** Being controlled by the customized instructions sent from the RISC-V core, the CGHRA in RoCC needs to be designed to perform the SIKE calculations accordingly.

The above parts are further elaborated in the following subsections.

4.1 Customized Instructions for SIKE

The user can write the software program containing the RISC-V customized instructions that can send requests ("Req" in Figure 10) to RoCC. The CGRHA in RoCC is controlled by these requests to conduct efficient SIKE calculations. The customizable instruction is named `custom`, and its format is shown as the first white row in Figure 11. There are two supported source registers indicated by `rs1/rs2`, and the register values are transferred to

RoCC once `xs1/xs2` is set to 1, respectively. One destination register `rd` is also supported, but it is not involved in our design. Besides, `funct7` is an indicator that can be set with specific values for controlling the CGRHA in RoCC.

	31 26	25 20	19 15	14	13	12	11 7	6 0
	funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
load_init_data	1	0	data_paddr	0	1	0	0	opcode
rom_instr_exec	2	instr_end	instr_start	0	1	1	0	opcode
results_writeback	3	ram_addr	data_paddr	0	1	1	0	opcode

Figure 11: The format of `custom` instruction in RISC-V ISA. (The first white row refers to the format of the original `custom`; The three grey rows refer to the implementation of our customized instructions by leveraging `custom`.)

When designing the detailed functionality of the `custom` to control RoCC for the SIKE calculations, there are two challenges as follows.

C1. The RoCC can be designed to finish the complete SIKE calculations, so only one `custom` is needed to trigger RoCC. Unfortunately, this can incur high hardware consumption, and more seriously, it has too limited flexibility to support various SIKE parameters.

C2. In contrast, we can use different `funct7` values in `custom` to represent the requests for different basic operations of SIKE, such as modular addition and modular multiplication. However, since the proposed FALU in RoCC is highly optimized, the time cost of the `custom` instructions could be more than that of the calculations on CGRHA's FALU. This could lead to time overhead compared with hardware-based SIKE solutions due to the extra cost of executing many `custom` instructions in the core's pipeline.

To seek the best tradeoff between performance and flexibility and tackle challenges C1 and C2, we hardcode some instructions inside the ROM of RoCC, which are named ROM instructions. Each ROM instruction represents a basic operation of SIKE, such as multiplication, addition, and read/write data. By executing a group of ROM instructions, RoCC can perform a kind of coarse-grained SIKE calculations such as a small isogeny computation and a point addition shown in Figure 1. The user can use `custom` to select the ROM instruction group to be executed by using the source registers of `custom` to indicate the ROM address of the ROM instructions. By proposing multiple groups of ROM instructions, our RISC-V architecture can provide the users with the flexibility to support different SIKE parameters.Therefore, challenge C1 is tackled. Besides, since each group of ROM instructions contains multiple basic operations, the time cost is saved by reducing the amount of the `custom` instructions used for controlling RoCC, so challenge C2 is also tackled. Note that each group of the ROM instructions is designed to be finished in constant time for security.

In detail, we propose three customized instructions by using different `funct7` of the `custom`, which are `load_init_data`, `rom_instr_exec`, and `result_writeback`. They are shown as the three grey rows in Figure 11. The initial data (38 data in total) is loaded into RoCC from the memory once RoCC receives the request of `load_init_data`. The physical address `data_paddr` of the initial data is stored inside the source register `rs1`. For `rom_instr_exec`, two values `instr_start` and `instr_end` are stored in `rs1` and `rs2`, respectively. The ROM entries of the ROM instruction group range from `instr_start` to `instr_end`. When RoCC receives the request of `rom_instr_exec`, the calculations of the group of the ROM instructions begin. For `result_writeback`, it writes the results (12 data in total) in the RAM of RoCC back to the memory. The address of data to be written in the RAM is indicated by `ram_addr` in `rs2`, while the physical address of the memory is indicated by `data_paddr` in `rs1`.

With the support of the customized instructions for SIKE, the user can write the software program, such as a C program, to efficiently support different SIKE parameters.

4.2 Software Program and RISC-V Core

The user can write a software program to control RoCC for coarse-grained SIKE calculations and reconfigure the results from RoCC for completing the SIKE calculations. In the software program, the regular RISC-V instructions and the `custom` instructions are both executed in the RISC-V core. Once a `custom` instruction is committed after "WB" stage, a request from `custom` is sent to RoCC, along with the information in `custom`'s format and the values of the source registers. When RoCC is busy with the current request, the regular instructions in the core can keep executing before the next request of `custom` arrives. When the next `custom` is committed but RoCC is still busy with the current request, the core will be stalled until the current request is handled. The core and RoCC share the same L1Dcache for memory accesses. Besides, we modified the source code of the Linux system to reserve a part of the physical memory for SIKE calculations, shown as the "Reserved Memory" in Figure 10, so that the software and hardware can share data within the reserved memory.

By leveraging the RISC-V core of Rocket Chip, the `custom` instructions for coarse-grained calculations and the regular instructions for reconfiguration can both be well coordinated according to the user's software program, and the efficient hardware-software co-design can be realized.

4.3 Customized RoCC

The RoCC is customized to implement CGRHA for handling the requests from the core and conducting SIKE calculations. The overall architecture can be found in Figure 2 and Figure 10. During handling the software requests, the functionalities of the key modules in CGRHA are described as follows.

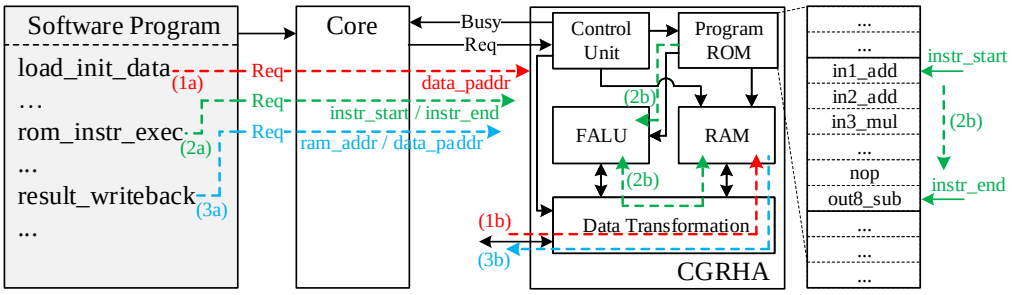

Figure 12: The flow of the request of each customized instruction. (The colored information refers to the example of the flows.)

- As mentioned in Section 3, FALU is proposed for efficient modular multiplication and modular addition/subtraction. Besides, a RAM is used to store the intermediate results, and a module for data transformation is designed to transform the data format from different sources.
- A ROM is proposed to store multiple groups of ROM instructions. Each group is designed to be finished in constant time for security. An example of the program ROM is shown on the right side of Figure 2, where each entry of the ROM stores a ROM instruction. The instructions between each solid line are from the same group. The syntax `ink_op` refers to a group's k-th instruction that inputs the data from the RAM to FALU for calculation, and `outk_op` refers to the k-th instruction that outputs the

calculated result from the FALU to the RAM. For example, `out8_sub` refers to the 8-th instruction that outputs the result, which is obtained from the modular subtraction of FALU.

- A control unit is designed to decode the request from the core. After decoding the request, it controls the program ROM, the RAM, and the data transformation module for handling the request. Upon receiving the requests, the detailed behaviors of the control unit are described as follows.

  - **Request from `load_init_data`:** In the software program, the user can prepare the initial data (38 data in total) inside the reserved physical memory by using the `mmap` function, and store the physical address in the source register of `load_init_data`. Once `load_init_data` in the program is executed and the request is sent to CGRHA (shown as (1a) in Figure 12), the control unit of CGRHA will send a memory read request according to the physical address. The data length is fixed in SIKE calculations. The data read from the memory are first transformed and buffered in the data transformation module. After all the initial data are read, the control unit moves the buffered data to the RAM in CGRHA (shown as (1b) in Figure 12).
  - **Request from `rom_instr_exec`:** In the user program, `rom_instr_exec` can be used to send request to CGRHA for executing ROM instructions. The values of `rom_instr_exec`'s two source registers are called `instr_start` and `instr_end`, which indicate the ROM address range of the selected ROM instruction group. Upon receiving the request of `rom_instr_exec`, the control unit in CGRHA obtains `instr_start` and `instr_end` (shown as (2a) in Figure 12). Then, the control unit fetch each instruction in the range between `instr_start` and `instr_end`, and control the FALU to execute the instructions of the selected ROM instruction group (shown as (2b) in Figure 12). During the execution, the input/output data of FALU are fetched/stored from/to the RAM in CGRHA, respectively.
  - **Request from `result_writeback`:** After SIKE calculations are finished, the user can use `result_writeback` to write the results (12 data in total) in RAM into the reserved memory, by specifying the physical address `data_paddr` in `rs1`, and the results' address of the RAM `ram_addr` in `rs2` (shown as (3a) and (3b) in Figure 12).

In summary, by applying the proposed CGRHA as the co-processor, our proposed RISC-V co-design architecture is able to efficiently execute SIKE calculations while keeping enough flexibility with the support of the customized instructions.

5 Experimental Results and Comparison

The implementation experiments are divided into two parts. The first part is to run the palace & route for the time and area evaluation of the co-processor over the Xilinx Vivado 2021.1 EDA platform based on a Xilinx Virtex-7 FPGA with a core of xc7vx690tffg1157-3 that is adopted by most of the previous SIKE implementations. The second part is to test the evaluation of the HSC-design over a HyperSilicon VeriTiger-H4000T FPGA development platform based on a Xilinx Virtex UltraScale FPGA with a core of xcvu440flga2892. The second part is based on a powerful open source RISC-V SoC project called Rocket Chip [AAB+16], and the software program is tested within Linux operating system running on the RISC-V platform. The Experimental results and comparison are presented in the following.

Table 4: Timing performance of the proposed modular units and the major coarse-grained operations for all the four SIKE parameters on a Virtex-7 FPGA.

Operation		# Cycles	Time (ns)
Modular unit in F_p	Add./Sub.	1	5.5
	Multiplier	31	170.5
Coarse-grained operation	Ladder Step	199	1,094.5
	Point Quad.	309	1,699.5
	Point Triple	272	1,496.0
	Get_4_Iso.	87	478.5
	Eval_4_Iso.	253	1,391.5
	Get_3_Iso.	148	814.0
	Eval_3_Iso.	163	896.5
	Get_2_Iso.	48	264
	Eval_2_Iso.	131	720.5

5.1 Hardware Implementations and Comparison

To make a fair comparison with previous works, we code the proposed co-processor in SystemVerilog language and implement it on Vivado equipped with the Virtex-7 690T core. The whole CGRHA achieves a maximum frequency of 181.8MHz. The clock cycles and time of the modular units and the major coarse-grained operations over the Virtex-7 FPGA are shown in Table 4. All of them maintain the same performance for the four SIKE parameters.

5.1.1 Results and Comparison of Modular Multiplier

Since the modular multiplier dominates the hardware resources, we separately pick out the corresponding terms and compare them with previous works. The area and timing comparisons are shown in Table 5, where all the works can support the SIKEp751 parameter. It should be noted that our multiplier is configurable for all the four SIKE parameters with constant time, while the others are delicately designed for SIKEp751. Overall, the proposed design obtains the best ATP among the existing works when considering the slice equivalent cost (SEC) and the interleaved latency.

The HR-Montgomery based multipliers [FBSMBA21a,EKAMK19,FBSMBA21b,NOL+22a, KAEK+20, EAMK20, NOL+22b] almost all have small area consumption. However, all of them suffer from high latency and interleaved latency caused by the intrinsic iterations and the inserted pipelines, which heavily constrain the upper limit of SIKE performance. The polynomial-based UR-Barrett multiplier in [TWW21] adopts a complete feed-forward scheme by leveraging the fully parallel schedule of the proposed algorithm. So, it has the lowest latency and interleaved latency and assists SIKEp751 in obtaining a new speed record at the cost of more resources. Due to the attractive timing performance, we take the UR-Barrett multiplier as our baseline. Compared with the original version in [TWW21], we reduce the number of Slices by about 40% and DSPs by about 70% and meantime extend the support to all four SIKE parameters. The total latency is nearly doubled, but the interleaved latency is only three cycles, much smaller than other previous designs. Therefore, we can still use a single modular multiplier for SIKE to achieve a high degree of parallelism.

5.1.2 Results and Comparison of SIKE Implementation

As introduced in Section 1, most previous works for SIKE implementations are in pure software or pure hardware. The pure hardware implementation can achieve better time

Table 5: Area and timing comparisons of modular multipliers, where ours can be configured for all the four SIKE parameters .

Work	Slices	DSPs	Freq. (MHz)	Latency (# CCs)	Interleaved Latency (# CCs)	ATP [1] (SEC×μs)
[FBSMBA21a] [2]	1,999	38	223	124	101	2,627
[EKAMK19] [3]	967	73	232.8	144	144	5,113
[FBSMBA21b]	6,939	64	205	30	26	1,694
[NOL+22a]	1,577	32	357	144	97	1,299
[KAEK+20]	-	128	167.4	100	69	5,276 [4]
[EAMK20]	-	113	294	138	90	3,459 [4]
[NOL+22b]	4,129	144	183	61	48	4,860
[TWW21]	23,311	828	155.8	16	1	681
Ours	14,626	258	181.8	31	3	667

[1] ATP = (Slices + DSPs × 100) × Interleaved Latency / Frequency.
[2] The efficient multiplier with (16, 32).
[3] O-FIOS.
[4] Only the DSPs are counted since the Slices are not provided.

performance but lack flexibility. Therefore, some researchers have begun to transfer the focus on HSC-designs. Until now, there are four works [MLRB20, RFS20, EAMK21a, EKAK22] for HSC-designs of SIKE. Except for the design in [RFS20], the rests are simulated and implemented on an EDA platform. To make a fair comparison, we provide the results of the proposed SIKE co-processor with post-place and route on Vivado. The area and timing results of ours and the other three works are shown in Table 6, where the two designs of [MLRB20] are included. Since the Slice number of the co-processor in [EKAK22] is not provided, we approximate it as LUTs/4 + FFs/8. We use 1 BRAM = 2 DSPs = 200 Slices for SEC as was used in [EKAK22]. The time is computed by counting the encapsulation and decapsulation together.

It should be noted that the four previous works are with scalable basic modules for modular multipliers. The two multipliers in [MLRB20] adopt a 128-bit MAC for low area and a 256-bit MAC for high speed, respectively. These basic MAC units are repeatedly used to realize the modular multiplier. The multipliers in [EAMK21a] and [EKAK22] are iteratively computed by a 17-bit pipelined architecture. It means that the latency of their modular multiplier of each SIKE parameter differs, while that of ours is a constant. They can save more cycles for the smaller SIKE parameters, and the timing results of the four parameters have larger gaps than ours. However, since our modular multiplier already has a very small latency, we achieve about 2.6-4.4x faster speed than the state-of-the-art for the four SIKE parameters. Considering the ATP, our design obtains the best result for SIKEp751 and better results than most of prior arts except the design in [EKAK22] for the other three SIKE parameters. It should be pointed out that the timing performance is much more difficult to be improved because of the heavy data dependency. It can easily be found in Table VII of [EKAK22] that our design is the fastest among existing implementations for SIKEp434, SIKEp503, and SIKEp610, and slightly inferior to the record-holder design in [TWW21] for SIKEp751. At the same time, our design achieves a better ATP performance than most of the previous pure hardware implementations.

5.1.3 Comparison with Other NIST PQC KEM Protocols

After the announcement by NIST on July 5th, 2022, there are five KEM PKE algorithms left, *i.e.*, CRYSTALS-KYBER (lattice-based), BIKE (code-based), Classic McEliece (code-

Table 6: Area and timing comparisons of SIKE co-processors over a Virtex-7 FPGA. Regarding the time and ATP of each work, the results are for SIKEp434, SIKEp503, SIKEp610, and SIKEp751 from top to bottom.

Work	Area					Timing		ATP
	LUTs	FFs	Slices	DSPs	BRAMs	Freq. (MHz)	Time (ms)	SEC[†] × s
[MLRB20] (S)	11,984	7,268	3,855	57	21	153.9	49.8	685
							88.0	1,210
							105.9	1,457
							177.5	2,442
[MLRB20] (F)	21,321	13,756	8,131	162	39	141.6	24.4	784
							49.9	1,603
							52.0	1,671
							61.0	1,960
[EAMK21a]	9,888	13,922	4,087	78	21.5	243.6	19.2	311
							25.1	406
							38.7	626
							55.0	890
[EKAK22]	8,211	13,228	3,706	78	21	303.0	14.5	228
							19.2	302
							29.8	468
							42.7	671
Ours	52,814	69,152	19,145	276	56	181.8	5.5	319
							6.3	365
							8.6	498
							9.8	568

[†] SEC = Slices + DSPs × 100 + BRAMs × 200.
The number of Slices is approximately computed by using LUTs/4 + FFs/8.

Table 7: Area and time comparisons of different NIST PQC KEM algorithms on FPGA platforms.

PQC KEM	FPGA Platform	NIST Level	Public Key (in Bytes)	Freq. (MHz)	Area Slices/DSPs/BRAMs	Time (μs) KeyG/E/D
CRYSTALS -KYBER [XL21]	Artix-7	1	800	161	2,126/2/3	23/31/41
		3	1,184			39/48/62
		5	1,568			58/68/86
BIKE [RBCGG22]	Artix-7	1	1,540	113	7,332/13/34	1,672/132/1,892
Classic McEliece [CCU+20]	Artix-7	1	261,120	105.6	36,859 /-/236	1,920/26/95
	Virtex-7	3	524,160	130.8	48,488 /-/446	3,943/26/111
	Virtex 7	5†	1,044,992	136.6	53,930 /-/589	7,658/37/181
HQC‡ [MAB+]	Artix-7	1	3,024	148	6,600/0/12.5	270/590/1,200
SIKE (**Ours**)	Virtex-7	1	330	181.8	19,145/276/56	1,532/2,669/2,888
		2	378			1,737/3,077/3,223
		3	462			2,142/4,343/4,252
		5	564			2,728/4,778/5,037

The number of Slices is approximately computed by using LUTs/4 + FFs/8.
† The results are for mceliece6688128.
‡ The hardware implementation of HQC provided is coded by HLS language.

based), HQC (code-based), and SIKE (isogeny-based). CRYSTALS-KYBER is selected to be standardized for its strong security and excellent performance. The other four algorithms still move on to the fourth round. We list the latest FPGA results of area and time of the other four KEM algorithms [XL21, RBCGG22, CCU+20, MAB+] and our proposed SIKE implementation in Table 7 to show their merits and demerits. It should be noted that as there exist different FPGA platforms, the comparison is not so strict. The following discussion only considers the performance, without regard to the security.

Overall, the key size is the best advantage of SIKE, and the timing performance is the worst metric compared to others. Our work effectively reduces the gap to some degree. Compared to Classic McEliece, SIKE also has area superiority. Actually, CRYSTALS-KYBER has an obvious advantage over the rest in area and time performance and with a moderate length of key sizes. There is still more than one order of magnitude of time gap between CRYSTALS-KYBER and SIKE. Regarding the three code-based algorithms, all of them are required with an asymmetric time between encapsulation and decapsulation. Classic McEliece has better timing performance, but the key sizes and the resource consumption show much poorer performance than the other two's. BIKE and HQC have many similarities in performance and can make a tradeoff between key sizes and time performance.

5.2 System Evaluation

We first evaluate the runtime of the entire hardware-software co-design system. As shown in Table 8, given the flexibility of our system, four kinds of SIKE calculations under different parameter settings are tested. The runtime is separated into three steps, and the clock cycles needed are listed. During the running of the software program, the customized instructions can send requests to CGRHA (on RoCC) for completing coarse-grained SIKE calculations. Under different SIKE parameters, the results show that CGRHA spends 278K-496K, 485K-869K, and 525K-916K cycles on key generation, encapsulation, and decapsulation, respectively. When considering the runtime of the software program, the cycle cost slightly increases due to the software instruction execution and communication for controlling CGRHA. Benefiting from the coarse-grained calculations performed on CGRHA, the runtime overhead caused by the software instruction execution is less than 10%. This proves that the extra time cost of sending the request is limited, and the

CGRHA is fully utilized. Compared with work [RFS20], which only evaluates SIKE434, our work has several magnitude smaller runtime under the same parameter setting. Table 8 proves both the flexibility and the speed of our system.

Table 8: System runtime analysis under different SIKE parameters

		Pure Hardware (CGRHA)				Software Program with Co-Design				Hardware Calculation Utilization			
		KeyG	E	D	E+D	KeyG	E	D	E+D	KeyG	E	D	E+D
[RFS20]	SIKE434	-	-	-	-	10,700	17,800	19,100	36,900	-	-	-	-
Ours	SIKE751	496	869	916	1,784	503	880	928	1,807	96.82%	97.94%	98.66%	98.31%
	SIKE610	389	790	773	1,563	411	826	813	1,639	94.65%	95.64%	95.08%	95.36%
	SIKE503	316	560	586	1,146	333	596	625	1,221	94.89%	93.96%	93.76%	93.86%
	SIKE434	278	485	525	1,010	290	497	527	1,024	95.86%	97.59%	99.62%	98.63%

The unit of the listed integers is K clock cycles.

Hardware Calculation utilization represents the rate of the pure hardware calculation cycles to the software program execution cycles.

We tested our co-processor based on a full RISC-V system supporting Linux operating system. The area consumption of the co-design system is shown in Table 9. It is indicated that the additional area of RoCC's control circuits is negligible compared with the area of CGRHA. The area consumption is reasonable as the importance of PQC is rapidly increasing.

In conclusion, the system evaluation results show that the proposed hardware-software co-design realizes an excellent tradeoff between speed and flexibility, and thus, challenges C1 and C2 mentioned in Section 4.1 are well tackled. Meanwhile, the hardware resource consumption is reasonable enough for a real computer system.

6 Conclusions

In this paper, we have presented a fast and efficient hardware-software co-design for SIKE. Many optimization techniques, especially algorithmic strength reduction and novel architectural schemes, have been proposed and applied to the co-processor. By integrating the co-processor with an advanced RISC-V platform, we can flexibly implement the new HSC-design for all four SIKE parameters. Implementation results demonstrate that the proposed design is the fastest and more efficient than most of the prior arts.

References

[AAB+16] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical report, EECS Department, University of California, Berkeley, 2016.

Table 9: The area of our co-design system over a Virtex UltraScale FPGA.

	LUTs	FFs	CLBs	DSPs	BRAMs
Full System	130,894	121,467	24,597	294	90
RISC-V Core	49,437	38,420	9,443	18	51.5
RoCC	81,457	83,047	15,154	276	38.5
- CGRHA	81,166	82,808	15,076	276	38.5

[AAK21] Mila Anastasova, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast strategies for the implementation of sike round 3 on arm cortex-m4. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(10):4129–4141, 2021.

[ACC+17] Reza Azarderakhsh, Matthew Campagna, Craig Costello, LD Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, et al. Supersingular isogeny key encapsulation. *submission to the NIST post-quantum standardization project: https://sike.org/*, 152:154–155, 2017.

[AFJ14] Reza Azarderakhsh, Dieter Fishbein, and David Jao. Efficient implementations of a quantum-resistant key-exchange protocol on embedded systems. *Citeseer*, 2014.

[AJK+16] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. Key compression for isogeny-based cryptosystems. In *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, pages 1–10, 2016.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology-crypto 86, Santa Barbara, California, Usa*, pages 311–323, 1986.

[BF18] Joppe Bos and Simon Friedberger. Arithmetic considerations for isogeny based cryptography. *IEEE Transactions on Computers*, pages 1–1, 2018.

[BL17] Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.

[BP01] T. Blum and C. Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[CCJ+16] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology, 2016.

[CCU+20] Tung Chou, Carlos Cid, S UiB, J Gilcher, T Lange, V Maram, R Misoczki, R Niederhagen, KG Paterson, and E Persichetti. Classic mceliece: conservative code-based cryptography, 10 october 2020, 2020.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter YA Ryan. Highly vectorized sike for avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 41–68, 2022.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Annual International Cryptology Conference*, pages 572–601. Springer, 2016.

[EAMK20] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Highly optimized montgomery multiplier for SIKE primes on FPGA. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 64–71. IEEE, 2020.

[EAMK21a] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Accelerated RISC-V for SIKE. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, pages 131–138. IEEE, 2021.

[EAMK21b] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. High-performance fpga accelerator for sike. *IEEE Transactions on Computers*, 2021.

[EKAK22] Rami Elkhatib, Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Accelerated RISC-V for post-quantum SIKE. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[EKAMK19] Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Optimized algorithms and architectures for montgomery multiplication for post-quantum cryptography. In *International Conference on Cryptology and Network Security*, pages 83–98. Springer, 2019.

[FBSMB20] Mohammad-Hossein Farzam, Siavash Bayat-Sarmadi, and Hatameh Mosanaei-Boorani. Implementation of supersingular isogeny-based diffie-hellman and key encapsulation using an efficient scheduling. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(12):4895–4903, 2020.

[FBSMBA21a] Mohammad-Hossein Farzam, Siavash Bayat-Sarmadi, Hatameh Mosanaei-Boorani, and Armin Alivand. Hardware architecture for supersingular isogeny Diffie-Hellman and key encapsulation using a fast montgomery multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(5):2042–2050, 2021.

[FBSMBA21b] Sayed Mohammad-Hossein Farzam, Siavash Bayat-Sarmadi, Hatameh Mosanaei-Boorani, and Armin Alivand. Fast supersingular isogeny Diffie–Hellman and key encapsulation using a customized pipelined montgomery multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(3):1221–1230, 2021.

[FH15] Haining Fan and M Anwar Hasan. A survey of some recent bit-parallel gf (2n) multipliers. *Finite Fields and Their Applications*, 32:5–43, 2015.

[FHLOJRH17] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, 2017.

[FSGL10] Haining Fan, Jiaguang Sun, Ming Gu, and K-Y Lam. Overlap-free karatsuba–ofman polynomial multiplication algorithms. *IET Information security*, 4(1):8–14, 2010.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[JAC+] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation. Submission to the NIST Post-Quantum Standardization Project, 2022, [Online] https://sike.org.

[JAK18] Amir Jalali, Reza Azarderakhsh, and Mehran Mozaffari Kermani. NEON SIKE: supersingular isogeny key encapsulation on ARMv7. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 37–51. Springer, 2018.

[Jao11] David Jao. Software for "towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies". https://github.com/defeo/ss-isogeny-software, 2011.

[JDF11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[JMEH02] Michael Jung, Felix Madlener, Markus Ernst, and Sorin A Huss. A reconfigurable coprocessor for finite field multiplication in gf (2n). In *Proc. of IEEE Workshop on Heterogeneous Reconfigurable systems on Chip, Hamburg, Germany*, 2002.

[KAEK+20] Brian Koziel, A-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. SIKE'd up: Fast hardware architectures for supersingular isogeny key encapsulation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(12):4842–4854, 2020.

[KAK18] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. A high-performance and scalable hardware architecture for isogeny-based cryptography. *IEEE Transactions on Computers*, 67(11):1594–1609, 2018.

[KAKJ17] Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Post-quantum cryptography on FPGA based on isogenies on elliptic curves. *IEEE Transactions on Circuits and Systems I-regular Papers*, 64(1):86–99, 2017.

[KMT+17] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, 2017.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[KRVV16] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient finite field multiplication for isogeny based post quantum cryptography. *International Workshop on the Arithmetic of Finite Fields*, pages 193–207, 2016.

[LNL+19] Weiqiang Liu, Jian Ni, Zhe Liu, Chunyang Liu, and Máire O'Neill. Optimized modular multiplication for supersingular isogeny diffie-hellman. *IEEE Transactions on Computers*, 68(8):1249–1255, 2019.

[MAB+] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérome Lacan, et al. Hamming quasi-cyclic (HQC). NIST post-quantum cryptography standardization project (round 3), 2020.

[Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, 1985.

[MLRB20] Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. A compact and scalable hardware/software co-design of SIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 245–271, 2020.

[MP85] Montgomery and L. Peter. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–519, 1985.

[MZB+08] Mohsen Machhout, Medien Zeghid, Belgacem Bouallegue, Rached Tourki, et al. Efficient hardware architecture of recursive karatsuba-ofman multiplier. In *2008 3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, pages 1–6. IEEE, 2008.

[NOL+22a] Ziying Ni, Máire O'Neill, Weiqiang Liu, et al. A high performance SIKE accelerator with high frequency and low area-time product. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2022.

[NOL+22b] Ziying Ni, Máire O'Neill, Weiqiang Liu, et al. A high-performance SIKE hardware accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(6):803–815, 2022.

[Oru95] Holger Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 193–199. IEEE, 1995.

[Par07] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[RBCGG22] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: Improved polynomial multiplication and inversion in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 557–588, 2022.

[RFS20] Debapriya Basu Roy, Tim Fritzmann, and Georg Sigl. Efficient hardware/software co-design for post-quantum crypto algorithm SIKE on ARM and RISC-V based microcontrollers. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[RIS10] RISC-V: The Free and Open RISC Instruction Set Architecture. https://riscv.org/, 2010.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SAJA20] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (sike) round 2 on arm cortex-m4. *IEEE Transactions on Computers*, 70(10):1705–1718, 2020.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[SLLH18] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–20, 2018.

[TLW19] Jing Tian, Jun Lin, and Zhongfeng Wang. Ultra-fast modular multiplication implementation for isogeny-based post-quantum cryptography. In *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 97–102. IEEE, 2019.

[TLW20] Jing Tian, Jun Lin, and Zhongfeng Wang. Fast modular multipliers for supersingular isogeny-based post-quantum cryptography. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(2):359–371, 2020.

[TWL+20] Jing Tian, Piaoyang Wang, Zhe Liu, Jun Lin, Zhongfeng Wang, and Johann Großschädl. Faster software implementation of the SIKE protocol based on a new data representation. Cryptology ePrint Archive, Report 2020/660, 2020. https://eprint.iacr.org/2020/660.

[TWL+21] Jing Tian, Piaoyang Wang, Zhe Liu, Jun Lin, Zhongfeng Wang, and Johann Groszschaedl. Efficient software implementation of the sike protocol using new data representation. *IEEE Transactions on Computers*, 2021.

[TWW21] Jing Tian, Bo Wu, and Zhongfeng Wang. High-speed FPGA implementation of SIKE based on an ultra-low-latency modular multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(9):3719–3731, 2021.

[Vél71] Jacques Vélu. Isogénies entre courbes elliptiques. *CR Acad. Sci. Paris, Séries A*, 273:305–347, 1971.

[Vu16] C Vu. Ibm makes quantum computing available on ibm cloud to accelerate innovation. *IBM News Room*, 2016.

[vzGS05] Joachim von zur Gathen and Jamshid Shokrollahi. Efficient fpga-based karatsuba multipliers for polynomials over f˜ 2. In *Selected Areas in Cryptography*, volume 3897, pages 359–369. Springer, 2005.

[WP06] André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. *Cryptology ePrint Archive*, 2006.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 328–356, 2021.

[ZSP+18] Gustavo HM Zanon, Marcos A Simplicio, Geovandro CCF Pereira, Javad Doliskani, and Paulo SLM Barreto. Faster key compression for isogeny-based cryptosystems. *IEEE Transactions on Computers*, 68(5):688–701, 2018.

[ZWD+20] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. Quantum computational advantage using photons. *Science*, 370(6523):1460–1463, 2020.

A Appendix: Deduction for Interleaved Multiplication of Polynomials and Integers

We assume two kinds of polynomials, $P = p_0 + p_1 x_1 + ... + p_{2n-1}x^{2n-1} = [p_0, p_1, ..., p_{2n-1}]$ and $Q = q_0 + q_1 x_1 + ... + q_{n-1}x^{n-1} = [q_0, q_1, ..., q_{n-1}]$, where the order $2n$ of P is twice of that of Q while the data width w of the coefficients p_i is half of that of q_i. We will demonstrate in the following that the multiplication operation over the two polynomials can be efficiently designed and fully reused.

Assume two polynomials over P: $A = [a_0, a_1, ..., a_{2n-1}]$ and $B = [b_0, b_1, ..., b_{2n-1}]$. The product of A and B can be represented as:

$$\begin{aligned} C &= A \times B = [c_0, ..., c_i, ..., c_{4n-2}] \\ &= [a_0b_0, ..., \sum_{j=0}^{i} a_j b_{i-j}, ..., \sum_{j=0}^{2n-1} a_j b_{2n-1-j}, ..., \sum_{j=i-2n+1}^{2n-1} a_j b_{i-j}, ..., a_{2n-1}b_{2n-1}], \end{aligned} \tag{3}$$

where $4n^2$ w-bit multipliers are used, which can be reduced to $3^{\log_2 4n}$ by using the proposed $2n$-point FCM architecture.

Again, we suppose two polynomials over Q:

$$T = [t_0, t_1, ..., t_{n-1}] = [a_0 + a_1 2^w, a_2 + a_3 2^w, ..., a_{2n-2} + a_{2n-1}2^w], \tag{4}$$

$$S = [s_0, s_1, ..., s_{n-1}] = [b_0 + b_1 2^w, b_2 + b_3 2^w, ..., b_{2n-2} + b_{2n-1}2^w]. \tag{5}$$

We split the integer coefficients of T and S in this way just to show how to interleave the integer multiplications with the polynomial multiplications and reuse the previous pure polynomial multipliers. The product of the two polynomials equals:

$$\begin{aligned} V &= T \times S = [v_0, ..., v_i, ..., v_{2n-2}] \\ &= \{t_0 s_0, ..., \sum_{j=0}^{i} t_j s_{i-j}, ..., \sum_{j=0}^{n-1} t_j s_{n-1-j}, ..., \sum_{j=i-n+1}^{n-1} t_j s_{i-j}, ..., t_{n-1}s_{n-1}] \\ &= [a_0b_0 + (a_0b_1 + a_1b_0)2^w + a_1b_1 2^{2w}, ..., \\ &\quad \sum_{j=0}^{i} a_{2j} b_{2i-2j} + \sum_{j=0}^{2i+1} a_j b_{2i+1-j} 2^w + \sum_{j=0}^{i} a_{2j+1} b_{2(i+1)-2j-1} 2^{2w}, ..., \\ &\quad \sum_{j=i-n+1}^{n-1} a_{2j} b_{2i-2j} + \sum_{j=2i+1-2n+1}^{2n-1} a_j b_{2i+1-j} 2^w + \sum_{j=i-n+1}^{n-1} a_{2j+1} b_{2(i+1)-2j-1} 2^{2w}, ..., \\ &\quad a_{2n-2}b_{2n-2} + (a_{2n-2}b_{2n-1} + a_{2n-1}b_{2n-2})2^w + a_{2n-1}b_{2n-1}2^{2w}] \\ &= [c^e_0 + c_1 2^w + c^o_2 2^{2w}, ..., c^e_{2i} + c_{2i+1}2^w + c^o_{2i+2}2^{2w}, ..., c^e_{4n-4} + c_{4n-3}2^w + c^o_{4n-2}2^{2w}]. \end{aligned} \tag{6}$$

It should be noticed that the number of w-bit multiplications is also $4n^2$. The superscripts "e" and "o" of variables c^e_{2i} and c^o_{2i+2} indicate that all the subscripts of the involved multiplication operators are even and odd, respectively. Luckily, these even/odd accumulated terms are just the outputs of the even/odd kernel of the $2n$-point FCM, without the adders in the last step. Therefore, we can use almost the same low-complexity architecture to compute polynomial multiplications over P and Q. The order of the inputs is not the same, so the Perm module should be separately devised. Additionally, for the latter, we should use several adders to combine those ready terms to get the final results.

We give an example to make Eq. (6) more intuitive. We assume $n = 3$, two polynomials $A = [a_0, a_1, a_2, a_3, a_4, a_5]$ and $B = [b_0, b_1, b_2, b_3, b_4, b_5]$ over P, and two polynomials $T =$

$[t_0, t_1, t_2] = [a_0+a_1 2^w, a_2+a_3 2^w, a_4+a_5 2^w]$, $S = [s_0, s_1, s_2] = [b_0+b_1 2^w, b_2+b_3 2^w, b_4+b_5 2^w]$ over Q. Their separate multiplications are:

$$\begin{aligned} C &= A \times B = [c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}] \\ &= [a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0, \\ &\quad a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0, a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0, \\ &\quad a_1b_5 + a_2b_4 + a_3b_3 + a_4b_2 + a_5b_1, a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2, \\ &\quad a_3b_5 + a_4b_4 + a_5b_3, a_4b_5 + a_5b_4, a_5b_5]. \end{aligned} \tag{7}$$

$$\begin{aligned} V &= T \times S = [v_0, v_1, v_2, v_3, v_4] \\ &= [t_0s_0, t_0s_1 + t_1s_0, t_0s_2 + t_1s_1 + t_2s_0, t_1s_2 + t_2s_1, t_2s_2] \\ &\quad [(a_0b_0) + (a_0b_1 + a_1b_0)2^w + (a_1b_1)2^{2w}, \\ &\quad (a_0b_2 + a_2b_0) + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)2^w + (a_1b_3 + a_3b_1)2^{2w}, \\ &\quad (a_0b_4 + a_2b_2 + a_4b_0) + (a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0)2^w \\ &\quad +(a_1b_5 + a_3b_3 + a_5b_1)2^{2w}, \\ &\quad (a_2b_4 + a_4b_2) + (a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2)2^w + (a_3b_5 + a_5b_3)2^{2w}, \\ &\quad (a_4b_4) + (a_4b_5 + a_5b_4)2^w + (a_5b_5)2^{2w}]. \end{aligned} \tag{8}$$

We have:

$$\begin{aligned} v_0 &= a_0b_0 + (a_0b_1 + a_1b_0)2^w + a_1b_1 2^{2w} = c_0^e + c_1 2^w + c_2^o 2^{2w}, \\ v_1 &= a_0b_2 + a_2b_0 + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)2^w + (a_1b_3 + a_3b_1)2^{2w} = c_2^e + c_3 2^w + c_4^o 2^{2w} \\ v_2 &= a_0b_4 + a_2b_2 + a_4b_0 + (a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0)2^w \\ &\quad +(a_1b_5 + a_3b_3 + a_5b_1)2^{2w} = c_4^e + c_5 2^w + c_6^o 2^{2w} \\ v_3 &= a_2b_4 + a_4b_2 + (a_2b_5 + a_3b_4 + a_4b_3 + a_5b_2)2^w + (a_3b_5 + a_5b_3)2^{2w} = c_6^e + c_7 2^w + c_8^o 2^{2w} \\ v_4 &= a_4b_4 + (a_4b_5 + a_5b_4)2^w + a_5b_5 2^{2w} = c_8^e + c_9 2^w + c_{10}^o 2^{2w}. \end{aligned} \tag{9}$$

Clearly, the results are consistent with the formula in Eq.(6). It should be noted that Eq. (6) only shows the method of one-stage splitting. Actually, it can be recursively used according to required scenarios. In this paper, we only need the one-stage splitting.