IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 2, pp. 328–356. DOI:10.46586/tches.v2021.i2.328-356

A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA

Yufei Xing and Shuguo Li

Institute of Microelectronics, Tsinghua University, Beijing, China
xingyf10@gmail.com, lisg@tsinghua.edu.cn

Abstract. Post-quantum cryptosystems should be prepared before the advent of powerful quantum computers to ensure information secure in our daily life. In 2016 a post-quantum standardization contest was launched by National Institute of Standards and Technology (NIST), and there have been lots of works concentrating on evaluation of these candidate protocols, mainly in pure software or through hardware-software co-design methodology on different platforms. As the contest progresses to third round in July 2020 with only 7 finalists and 8 alternate candidates remained, more dedicated and specific hardware designs should be considered to illustrate the intrinsic property of a certain protocol and achieve better performance. To this end, we present a standalone hardware design of CRYSTALS-KYBER, a module learning-with-errors (MLWE) based key exchange mechanism (KEM) protocol within the 7 finalists on FPGA platform. Through elaborate scheduling of sampling and number theoretic transform (NTT) related calculations, decent performance is achieved with limited hardware resources. The way that Encode/Decode and the tweaked Fujisaki-Okamoto transform are implemented is demonstrated in detail. Analysis about minimizing memory footprint is also given out. In summary, we realize the adaptive chosen ciphertext attack (CCA) secure Kyber with all selectable module dimension k on the smallest Xilinx Artix-7 device. Our design computes key-generation, encapsulation (encryption) and decapsulation (decryption and re-encryption) phase in 3768/5079/6668 cycles when $k = 2$, 6316/7925/10049 cycles when $k = 3$, and 9380/11321/13908 cycles when $k = 4$, consuming 7412/6785 LUTs, 4644/3981 FFs, 2126/1899 slices, 2/2 DSPs and 3/3 BRAMs in server/client with 6.2/6.0 ns critical path delay, outperforming corresponding high level synthesis (HLS) based designs or hardware-software co-designs to a large extent.

Keywords: CRYSTALS-KYBER · post-quantum cryptography · key exchange mechanism · Module-LWE · full hardware implementation

1 Introduction

Powerful quantum computers would render the public key cryptosystems currently used in our daily life insecure, with which the underlying mathematical hard problem integer factorization, discrete logarithm that considered to be infeasible to handle in classic computer architecture now can be broken in polynomial time by Shor's algorithm [Sho94]. As a result, the post-quantum cryptography (PQC), study of cryptosystems that would be secure against adversaries who have access to both classic and quantum computers, is under intensive research not only in academia but also within industrial community. In December 2016, NIST launched a contest of new post-quantum cryptographic schemes and called for proposals from all over the world, aiming to form public key cryptography standards before the upcoming quantum era. In December 2017 and January 2019, the

Licensed under Creative Commons License CC-BY 4.0.
Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-02-23

first and second round of candidates were announced respectively, each followed by a public comment period. Very recently, the third round of candidates were announced in July 2020. In total 15 out of 26 second round candidates get through, of which 7 have been selected as finalists and the other 8 as alternate candidates [AASA+20]. These schemes base their security on different mathematical hard problems, and the evaluation criteria used to compare schemes throughout standardization process mainly focuses on three aspects, namely security, cost & performance, and algorithm & implementation characteristics, in a decreasing order of importance.

There have been lots of works concentrating on achievable performance of PQC protocols in hardware-software co-design, aiming to evaluate several candidates with similar computational tasks on the same platform, obtaining relatively precise rankings among them, for instance [NCD19][NVB+20][FDAG19]. It would give preference to some protocols over others in certain use cases, and further impact the standardization process to some extent [FDAG19]. The most time-consuming parts of the protocol are offloaded to separate hardware accelerators, referring mainly to NTT and hash functions, while others are implemented with software in processors such as ARM Cortex series. The hardwired ARM core can be replaced with popular RISC-V soft cores, which can be implemented with configurable logic in FPGA, making it possible to implement the whole design in hardware, avoiding slow data transmission across hardware/software boundary to a large extent, including works in [BUC19][AEL+20][FSS20][XHY+20]. The accelerators can be designed as vector coprocessor, being able to speed up corresponding operations massively [XHY+20]. In addition to performance gain compared with pure software design, flexibility is another primary advantage in consideration that the PQC contest is still in progress and different tweaks may be involved in a certain protocol. By dividing tasks into hardware and software parts, procedures that are standard and fixed by a few parameters would be implemented in hardware, while the ones that are subject to tweak and often intractable to handle in hardware, would resort to software design.

Complete hardware designs are still rare up to now. As the contest progresses to the third round, and is expected to last 12-18 months from July 2020, more dedicated designs would better demonstrate the strength and illustrate the intrinsic property of a certain protocol, fitting the expectation from NIST that more performance data of seven finalists and eight alternate candidates for hardware implementation would emerge [AASA+20]. A few published ones that reported results for Kyber are [DFA+20] and [HHLW20]. Moreover, some procedures that are intractable to handle in hardware but shared among different participants in the contest, can be properly designed once and used in a similarly way among corresponding protocols, especially tasks required by the Fujisaki-Okamoto transform in re-encryption phase. To this end, we design and implement all building blocks of a specific protocol — CRYSTALS-KYBER in hardware.

As one of the finalists, CRYSTALS-KYBER [SAB+] is a KEM protocol basing its security on the presumed hardness of the MLWE problem [LS12]. It first constructs public key encryption from the original method in [Reg04] and then achieves adaptive CCA-secure through the tweaked Fujisaki-Okamoto transform [FO99]. The scheme possesses high performance on both hardware and software platforms, and another advantage over some Ring-LWE-based candidates in round 2 is that the security level can be easily adjusted by changing module dimension k. Specifically, security category 1,3 and 5, equivalent to strength of AES-128,AES-192 and AES-256 respectively are supported in Kyber.

Contributions. A manually designed hardware implementation of Kyber is presented to illustrate its strength over HLS-based designs and hardware-software co-designs. Through compact scheduling of sampling and NTT related processes, decent performance is achieved with limited computational resources in NTT core, and the method, especially the use of predefined order sequence table and endpoint table can be extended to other candidates. Unified butterfly pair is proposed, within which NTT-related procedures, including the

special point-wise multiplication (PWM) are all well supported and can be easily switched by short control code. A reference realization of the Fujisaki-Okamoto transform is given out, through use of a set of FIFOs with different specifications. We also explore the minimal memory footprint of all the RAMs and FIFOs adopted in out design. By means of unified computation cores and minimal memory footprint, the whole design is able to be deployed in the smallest device in Xilinx Artix-7 series. Our design has been verified against known answer test (KAT) files of Kyber in the round-3 submission to NIST and the Vivado project has been uploaded to https://github.com/xingyf14/CRYSTALS-KYBER to ease comparison.

Structure. The rest of our paper is organized as follows: Section 2 clarifies the notations adopted in our paper firstly, then gives a brief introduction to CRYSTALS-KYBER and describes the special NTT and PWM. Section 3 describes design rationale of main hardware modules and our optimizations in implementing the protocol, from both hardware and algorithm point of view. The way towards minimizing memory footprint is discussed in Section 4. Performance results on the selected FPGA device and comparison with related works are given out in Section 5, followed by discussion about other design choices. Finally we draw our conclusion in Section 6.

2 Preliminary

2.1 Notation

The ring of integers modulo prime q is denoted as $\mathbb{Z}_q$. The ring of integer polynomials modulo prime q is denoted as $\mathbb{Z}_q[X]$. Then polynomials modulo both q and $X^n + 1$ form the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. In MLWE-based protocol Kyber, polynomial vector that contains k elements in $\mathcal{R}_q$ is represented in bold lowercase where k represents module dimension in MLWE, while pure lowercase corresponds to a single element. Furthermore, polynomial matrix that consists of $k \times k$ elements in $\mathcal{R}_q$ is represented in bold uppercase. The i-th entry within a polynomial vector $\mathbf{s}$ is denoted by $\mathbf{s}_i$, and the entry that lies in i-th row, j-th column in polynomial matrix $\mathbf{A}$ is denoted by $\mathbf{A}_{ij}$. Greek symbol ζ is selected to denote the n-th primitive root of unity in $\mathbb{Z}_q$, in conformance with Kyber specification, then by ξ we mean the square of ζ. The binomial distribution with parameter η is denoted as $\mathcal{B}_\eta$, where η directly relates with possible range of noise samples. In Kyber, n, q, ζ, η are set to $256, 3329, 17, 2$ or 3 respectively and k takes value from 2,3,4.

2.2 CRYSTALS-KYBER KEM

Kyber, as part of the Cryptographic Suite for Algebraic Lattics (CRYSTALS), is a post-quantum key exchange mechanism that bases its security on the MLWE problem. It follows conventional construction method to build an IND-CPA public key encryption (PKE) scheme firstly and then turns it into an IND-CCA KEM through the tweaked Fujisaki-Okamoto transform. Detailed procedures corresponding to KYBER.CPAPKE and KYBER.CCAKEM are listed in Algorithm 5-Algorithm 10 in appendix. Parameter sets and instantiation of symmetric primitives are listed in Table 6 and Table 7 respectively, with definition of auxiliary functions depicted in Equation 12. In the first-round submission to NIST, prime q is chosen to be 7681 and satisfies $q \equiv 1 \bmod 2n$ with degree of modular polynomial $n = 256$, implying classic NTT can be utilized to speed up polynomial multiplication in $\mathcal{R}_q$. Current version of Kyber adjusts the protocol in several places, including the change of prime q from 7681 to 3329, resulting in several differences in detailed procedure, of which the most important one is that there is no $2n$-th primitive root of unity in $\mathbb{Z}_q$, thus polynomials during NTT process are not evaluated at all the $2n$-th roots of unity in $\mathbb{Z}_q$, but rather at all the n-th roots of unity. Although detailed

Algorithm 1 Polynomial Multiplication over $\mathbb{Z}_q[X]/(X^n+1)$ using negative wrapped convolution (NWT)

Input: $\mathbf{a}=\sum_{i=0}^{n-1} a_i x^i$, $\mathbf{b}=\sum_{i=0}^{n-1} b_i x^i$, both in $\mathbb{Z}_q/(X^n+1)$, the n-th primitive root of unity ω_n in $\mathbb{Z}_q$, the square root of ω_n, denoted as φ_{2n}, prime q satisfying $q \equiv 1 \bmod 2n$.
Output: polynomial $\mathbf{c}=\mathbf{a}*\mathbf{b} \bmod (q, X^n+1)$
1: $\overline{\mathbf{a}}$ =pre-scale($\mathbf{a}$) $=\sum_{i=0}^{n-1} a_i\varphi_{2n}^i x^i$
2: $\overline{\mathbf{b}}$ =pre-scale($\mathbf{b}$) $=\sum_{i=0}^{n-1} b_i\varphi_{2n}^i x^i$
3: $\hat{\mathbf{a}}=\mathrm{NTT}(\overline{\mathbf{a}})$, $\hat{\mathbf{b}}=\mathrm{NTT}(\overline{\mathbf{b}})$
4: $\hat{\mathbf{c}}=\hat{\mathbf{a}}\circ\hat{\mathbf{b}}$
5: $\overline{\mathbf{c}}=\mathrm{INTT}(\hat{\mathbf{c}})=\sum_{i=0}^{n-1} c_i\varphi_{2n}^i x^i$
6: $\mathbf{c}=$ post-scale($\overline{\mathbf{c}}$) $=\sum_{i=0}^{n-1} c_i x^i$
7: **return c**

algorithm is changed, the computational complexity remains the same, which would be demonstrated further in Subsection 2.3. On the basis of a rather small prime, the noise in Kyber can be selected within a tight range.

2.3 NTT in Kyber

NTT resembles discrete Fourier transform (DFT), which is widely used in signal processing and conducted in complex field $\mathbb{C}$. Differently, NTT is conducted in finite field $\mathbb{Z}_q$ and it transforms an integer polynomial from coefficient representation into point-value representation, and can be sped up in the same way as fast Fourier transform (FFT) used in DFT case, only if we choose a set of special points to evaluate the polynomial. By the usage of NTT, computational complexity of multiplying two n-term integer polynomial can be decreased from $\mathcal{O}(n^2)$ to $\mathcal{O}(n\log n)$. Typically, prime q is chosen such that there exists $2n$-th primitive root of unity in $\mathbb{Z}_q$, where the polynomial modulo X^n+1 completely factors into n degree 1 polynomials. Defining φ an arbitrary $2n$-th primitive root of unity and ω the square of φ, the n roots of X^n+1 can be generated completely by φ^{2k+1} with integer k ranges from 0 to $n-1$. The n-point NTT/INTT of vector $\mathbf{x}/\mathbf{X}$ are defined in Equation 1 and Equation 2, both referred to as classic NTT hereafter.

$$X_m=\sum_{k=0}^{n-1} x_k\varphi^{(2m+1)k}=\sum_{k=0}^{n-1}\left(x_k\varphi^k\right)\omega^{mk} \bmod q \tag{1}$$

$$x_k=\frac{1}{n}\sum_{m=0}^{n-1} X_m\varphi^{-(2m+1)k}=\varphi^{-k}\cdot\frac{1}{n}\sum_{m=0}^{n-1} X_m\omega^{-mk} \bmod q \tag{2}$$

Compared with evaluation process in FFT, pre-scaling is needed for input polynomial terms, corresponding to multiplication $x_k\varphi^k$ in Equation 1. Similarly, post-scaling (multiplication with φ^{-k} in Equation 2) should be conducted on output polynomial terms. These two auxiliary procedures result from the chosen polynomial modulo X^n+1, and help to make NTT applicable to this special modular polynomial multiplication. The complete processes are also known as negative wrapped convolution (NWC), as depicted in Algorithm 1.

NTT in Kyber is slightly different, where field $\mathbb{Z}_q$ contains n-th primitive roots of unity, but not $2n$-th primitive ones, thus the modulo X^n+1 can not fully factor into n degree 1 polynomials, but rather $n/2$ degree 2 ones. Specifically, $X^{256}+1=\prod_{i=0}^{127}(X^2-\zeta^{2i+1})$ in Kyber, and the evaluation process is applied to the set of all the 256-th root of unity $\{\zeta^1,\zeta^3,\cdots,\zeta^{255}\}$. For an arbitrary polynomial $f(x)=\sum_{i=0}^{255} f_i x^i$, it can be divided into two 128-term polynomials according to parity of index. Denoting x^2 by y, they are expressed as

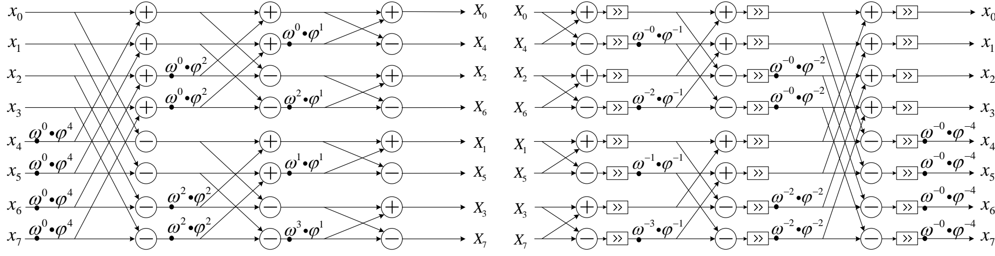

Figure 1: Schematic of rearranged NTT in NWC (left) and INTT in INWC (right)

$f^e(y) = \sum_{i=0}^{127} f_{2i}y^i, f^o(y) = \sum_{i=0}^{127} f_{2i+1}y^i$, and they are related with $f(x) = f^e(y) + xf^o(y)$. The evaluation process is then simplified as substituting y with each 256-th root of unity $\{\zeta^1, \zeta^3, \cdots, \zeta^{255}\}$, i.e. $f(\zeta^{2i+1}) = f^e(\zeta^{2i+1}) + xf^o(\zeta^{2i+1})$. In an alternative form that conforms the definition in Kyber specification, $F_i = f(\zeta^{2\mathrm{br}(i)+1}) = \hat{f}_{2i} + x\hat{f}_{2i+1}$, where

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\zeta^{(2\mathrm{br}(i)+1)j} = \sum_{j=0}^{127} f_{2j}\zeta^j\xi^{\mathrm{br}(i)j} \tag{3}$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathrm{br}(i)+1)j} = \sum_{j=0}^{127} f_{2j+1}\zeta^j\xi^{\mathrm{br}(i)j}, \tag{4}$$

$\xi = \zeta^2$ is the 128-th root of unity in $\mathbb{Z}_q$ and $\mathrm{br}(i)$ for $i = 0, \cdots, 127$ is the bit reversal of the unsigned 7-bit integer i. To this end, one 256-term NTT process in Kyber can be conducted by two *separate* 128-term classic ones as aforementioned in Equation 1.

After NTT, polynomial multiplication $h(x) = f(x) \cdot g(x) \bmod (X^n + 1)$ corresponds to PWM at these selected points, i.e. $H_i = F_i \cdot G_i \bmod (X^2 - \zeta^{2\mathrm{br}(i)+1})$. Specifically,

$$H_i = (\hat{f}_{2i}+x\hat{f}_{2i+1})\cdot(\hat{g}_{2i}+x\hat{g}_{2i+1}) = \hat{f}_{2i}\hat{g}_{2i}+\zeta^{2\mathrm{br}(i)+1}\hat{f}_{2i+1}\hat{g}_{2i+1}+x(\hat{f}_{2i}\hat{f}_{2i+1}+\hat{g}_{2i}\hat{g}_{2i+1}) \tag{5}$$

It is quite different from classic PWM where only one multiplication in $\mathbb{Z}_q$ would be conducted, here five multiplications are needed to complete the calculation, as the evaluation value at a certain point is not one element in $\mathbb{Z}_q$, but rather one degree 1 polynomial in $\mathbb{Z}_q$.

INTT in Kyber can be derived similarly as NTT, with one INTT corresponds to two classic ones as in Equation 2. When NTT and INTT are applied to vector or matrix of polynomials in $\mathcal{R}_q$, the respective operation is conducted on each element individually.

2.4 Order-relating issues and rearranged iterative NTT algorithms

In Kyber specification, the generated secret polynomial $\mathbf{s}, \mathbf{r}$, noise polynomial $\mathbf{e}, \mathbf{e}', e''$ are considered to be in natural order, e.g. elements $\mathbf{s}_{00}, \mathbf{s}_{01}, \mathbf{s}_{02} \cdots$ in $\mathbf{s}_0$ are generated one by one in order. Besides, the reference software implementation chooses decimation in time (DIT) strategy for NTT and decimation in frequency (DIF) strategy for INTT, which means input vector is considered to be in natural order, and the output vector of NTT is in bit-reversed order correspondingly, resulting from the property of NTT calculation. After INTT, the output vector would return to natural order. Noticing the order of input, output vector in original DIT NTT and DIF INTT is contrary to actual requirement in Kyber, both iterative DIT and DIF algorithm need modification to be applicable. Detailed schematic during each stage of rearranged DIT/DIF strategy is illustrated in left/right part of Figure 1 [POG15], with corresponding algorithm presented in Algorithm 2/Algorithm 3. It is noteworthy that twiddle factors are rearranged as well and further merged with scaling factors, and multiplication with n^{-1} in INTT is

Algorithm 2 Rearranged Iterative DIT NTT

Input: digits vector $\boldsymbol{x} = (x_0, x_1, \cdots, x_{n-1})$, precomputed φ^k for $k \in [0, n-1)$ stored in bitreversed order of k in ROM_forward.

Output: NTT of $\boldsymbol{x}$

1: **for** $i = \log_2 n$ downto 1 **do**
2: $m \leftarrow 2^i, r \leftarrow 0$
3: **for** $k = 0$ to $n-1$ by m **do**
4: $\omega \leftarrow$ ROM_forward $[r + \mathit{bitreverse}(m/2)]$
5: **for** $j = 0$ to $m/2 - 1$ **do**
6: $t \leftarrow \omega \cdot x_{j+k+m/2}$
7: $u \leftarrow x_{j+k}$
8: $x_{j+k} \leftarrow u + t$
9: $x_{j+k+m/2} \leftarrow u - t$
10: **end for**
11: $r \leftarrow r + 1$
12: **end for**
13: **end for**
14: **return** $\boldsymbol{X} = \text{NTT}(\boldsymbol{x})$

Algorithm 3 Rearranged Iterative DIF INTT

Input: digits vector $\boldsymbol{X} = (X_0, X_1, \cdots, X_{n-1})$, precomputed φ^{-k} for $k \in [0, n-1)$ stored in bitreversed order of k in ROM_backward.

Output: INTT of $\boldsymbol{X}$

1: **for** $i = 1$ to $\log_2 n$ **do**
2: $m \leftarrow 2^i, r \leftarrow 0$
3: **for** $k = 0$ to $n-1$ by m **do**
4: $\omega \leftarrow$ ROM_backward $[r + \mathit{bitreverse}(m/2)]$
5: **for** $j = 0$ to $m/2 - 1$ **do**
6: $u \leftarrow (X_{j+k} + X_{j+k+m/2})/2$
7: $t \leftarrow (X_{j+k} - X_{j+k+m/2})/2$
8: $X_{j+k} \leftarrow u$
9: $X_{j+k+m/2} \leftarrow \omega \cdot t$
10: **end for**
11: $r \leftarrow r + 1$
12: **end for**
13: **end for**
14: **return** $\boldsymbol{x} = \text{INTT}(\boldsymbol{X})$

distributed among stages where simple multiplication with 2^{-1} is conducted instead. These two methods are meant for decreasing computational complexity in terms of the number of multiplications in $\mathbb{Z}_q$. When conducting point-wise multiplication, the evaluation points are in bit-reversed order, requiring the corresponding 256-th primitive roots of unity in Equation 5 also in bit-reversed order, just as in Kyber specification $\zeta^{2br_7(i)+1}$ is used in an increasing order of i rather than ζ^{2i+1}. For the same reason, elements of public matrix $\hat{\mathbf{A}}, \hat{\mathbf{A}}^T$ should also be generated in bit-reversed order.

2.5 Binomial sampling

The noise polynomials $\mathbf{s}, \mathbf{e}, \mathbf{r}, \mathbf{e}', e''$ in Kyber obey binomial distribution with parameter $\eta = 2$ or 3, implying only four or six pseudo random bits are required to form one desired sample. Taking the former as an example, four random bits, denoted as $b[3:0]$ are firstly

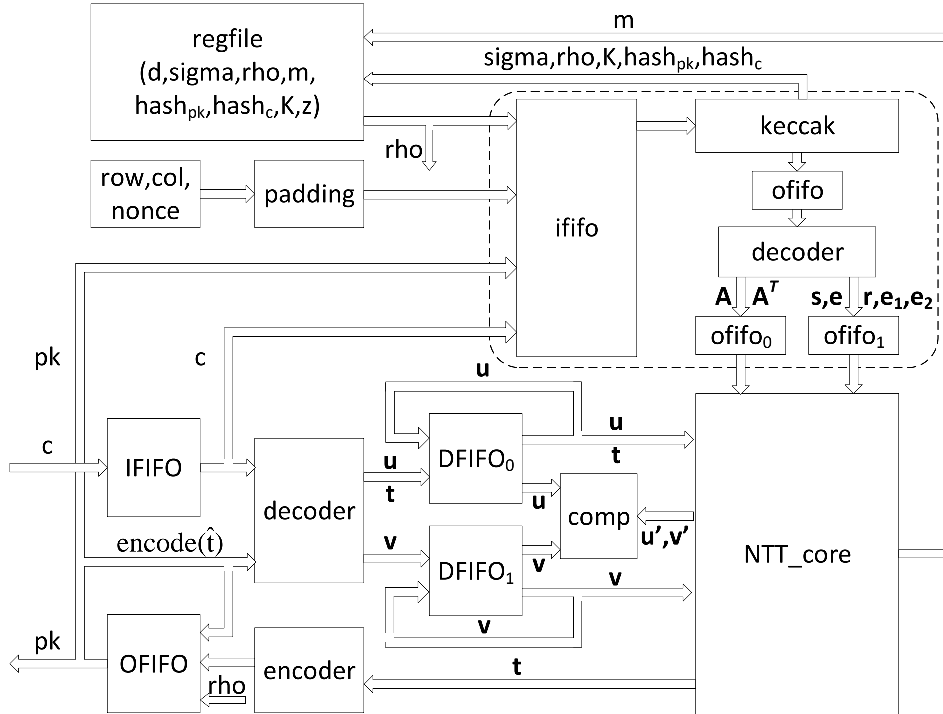

Figure 2: Data flow in server during the whole protocol. Symbols of intermediate values located above/to the left of data path arrows correspond to key generation and decryption phase, while values located below/to the right correspond to re-encryption phase.

segmented into $b[3:2]$ and $b[1:0]$, then Hamming weight of each part is calculated by $w_1 = b[3] + b[2]$, $w_0 = b[1] + b[0]$. The output sample would simply be $w_1 - w_0$, lying within $[-2, 2] \cap \mathbb{Z}$. As can be seen, the whole process is pretty succinct and easy to realize in constant time.

3 Design Rationale

The overall data flow chart in server is depicted in Figure 2, details corresponding to major parts will be demonstrated further in the following subsections, including NTT core and its components, encoder/decoder as well as hash module.

3.1 Butterfly unit

As aforementioned in Subsection 2.3, one 256-term NTT in Kyber can be regarded as two separate 128-term ones, namely the even index one and the odd index one, and each of them can be implemented in a classic way. These two separate NTTs share the same scaling factors and twiddle factors, thus can be calculated concurrently. Naturally two sets of butterfly units should be adopted to handle the even part and the odd part. Considering both DIT and DIF strategies should be supported, the unified butterfly structure would be more suitable than two separate operators that each supports one strategy [BUC19]. Details with regard to active operators during DIT NTT and DIF INTT is illustrated in upper part of Figure 3.

In addition to conventional use where it is either working in DIF mode or in DIT mode, several other operations are also adapted to functionality of the unified butterfly units, as

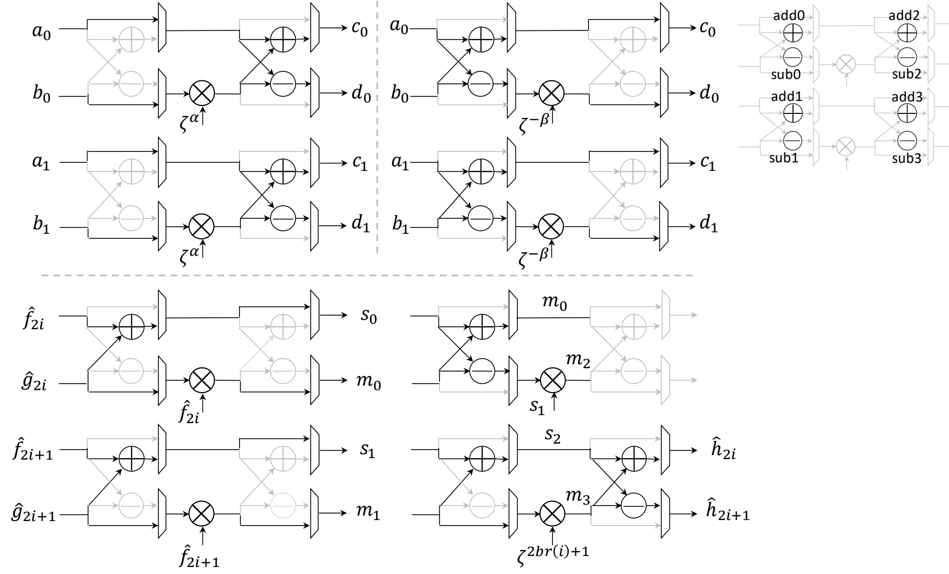

Figure 3: Active operators in two unified butterfly units during NTT (upper left), INTT (upper right), PWM0 (lower left) and PWM1 (lower right)

Table 1: Control code of each procedure. Symbol $'+'$ denotes corresponding operator is in use, while $'-'$ denotes it is idle at the moment.

function	sel[9:0]	add0	sub0	add1	sub1	add2	sub2	add3	sub3
NTT	110_000_0000	−	−	−	−	+	+	+	+
INTT	000_111_0000	+	+	+	+	−	−	−	−
PWM0	000_001_1010	+	−	+	−	−	−	−	−
PWM1	100_011_1101	+	+	+	−	−	+	−	+
INTTm	111_111_0000	+	+	+	+	+	+	+	+
COMPs	000_110_1000	−	+	−	+	−	−	−	−
DECOMP	000_000_0000	−	−	−	−	−	−	−	−

listed in Table 1. Correspondence between 10-bit control code *sel* with all the operators is depicted in Figure 10. The special point-wise multiplication in Kyber, corresponding to Equation 5, is dispatched among operators in two butterfly units and implemented in two cycles, namely PWM0 and PWM1. Active operators during each cycle are illustrated in lower part of Figure 3 and details would be further demonstrated in Subsection 3.3. Similarly, three other operations are also supported with some modifications made to operators in unified butterfly units or expression form of procedures in Kyber protocol. For the former one, INTTm function absorbs addition with $\text{Decompress}_q\,(\text{Decode}_1(m), 1)$ into the last stage of $\text{INTT}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}})$, from an observation that add2,sub2,add3 and sub3 are all idle during INTT. Obviously sub2 and sub3 should be modified in order to support extra addition functionality. For the latter one, one example is $\mathbf{u} = \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}'$ in encryption phase. Noticing $\mathbf{u}$ would be compressed before it forms ciphertext c_1, addition with noise samples can be delayed (thus $\mathbf{u} = \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}})$) and handled in Compress procedure by means of $\text{Compress}_q(\mathbf{u} - \mathbf{e}', d_u)$, equaling to $\lceil \frac{2^{d_u}(\mathbf{u}-\mathbf{e}')}{q} \rfloor \bmod 2^{d_u}$. The new operation is conformed to basic functionality of DIF butterfly unit and thus supported by our unified butterfly units, implying the addition with noise samples can be absorbed, saving both time and resources potentially. The same method also goes for $\text{Compress}_q(v, d_v)$, corresponding to COMPs in Table 1. The sign of noise samples should have no effect on

Figure 4: NTT structure, with symbols of intermediate values located above/under data path arrows correspond to data flow of server/client side.

security of the protocol, but to be compatible with software implementation, the centered binomial distributed noise samples in polynomial $\mathbf{e}'$ and e'' need to be negated, corresponding to position switch between two Hamming weights in subtraction. The DECOMP function realizes $\text{Decompress}_q(\text{Decode}_{d_u}(c_1), d_u)$ and $\text{Decompress}_q(\text{Decode}_{d_v}(c_2), d_v)$, in which multiplication with q would be conducted.

3.2 RAM structure

The corresponding even and odd index coefficients, arranged as a pair, reside in the same place within a RAM. During NTT process, two pairs of coefficients are fetched from RAM, with the even index coefficient from each pair sent to one butterfly unit, and the odd index coefficient from each pair sent to the other. In this way, the shared twiddle factors need to be read only once. An overall schematic of NTT core is illustrated in Figure 4.

The noise polynomials are written into RAM in natural order as discussed in Subsection 2.4. Each cycle two pairs of points would be fetched and two pairs would be written back concurrently in consideration that butterfly units function in a pipelined manner. At the first sight, the two pairs can be stored in the same place within a certain RAM, and output pairs between two consecutive cycles should be interleaved and stored back [RVM+14][FSS20]. The strategy is not suitable in our case as points with increasing index are stored continuously in RAM, thus not ready for application of the strategy unless an extra bit-reverse process is conducted, which is time-consuming. In our design, RAMs are arranged as two banks instead, each with the width of one pair of points. Strict in-place strategy is adopted, with additional advantage relating with PWM between data stored in RAMs and input points. Specifically, when two pairs of points are stored in the same place, one of them needs to be registered as two butterfly units can only handle one PWM per cycle. By each bank realized in separate RAM, granular data movement can be achieved. Detailed RAM structure is illustrated in Figure 5.

3.3 Adopting Karatsuba algorithm in point-wise multiplication

Point-wise multiplication, or basecase multiplication referred in Kyber specification $\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1}X) \cdot (\hat{g}_{2i} + \hat{g}_{2i+1}X) \bmod (X^2 - \zeta^{2br(i)+1})$, can be conducted in a

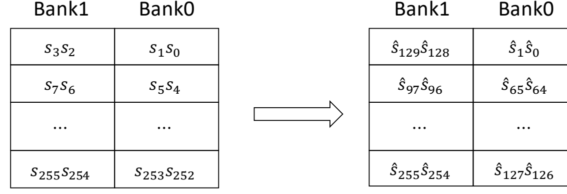

Figure 5: Detailed RAM structure, the left exhibits specific content after samples are written into the RAM and the right corresponds to results after NTT process.

straightforward way :

$$\hat{h}_{2i} = \hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1} \cdot \zeta^{2br(i)+1} \tag{6}$$

$$\hat{h}_{2i+1} = \hat{f}_{2i}\hat{g}_{2i+1} + \hat{f}_{2i+1}\hat{g}_{2i} \tag{7}$$

Clearly there are five multiplications in $\mathbb{Z}_q$ in total, and two consecutive ones, $\hat{f}_{2i+1}\hat{g}_{2i+1} \cdot \zeta^{2br(i)+1}$ form the longest path. Despite the longest path can not be shortened, the total number of multiplications can be decreased by means of Karatsuba algorithm, which is originally used in integer multiplication $a \times b = (a_0 + a_1 2^n) \times (b_0 + b_1 2^n)$ and depicted as follows :

$$a \times b = a_0 b_0 + ((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1)2^n + a_1 b_1 2^{2n} \tag{8}$$

The number of n-bit multiplications then decreases from four to three. On this basis, Karatsuba algorithm can also be used in PWM in our case:

$$\begin{aligned} &(\hat{f}_{2i} + \hat{f}_{2i+1}X)(\hat{g}_{2i} + \hat{g}_{2i+1}X) \bmod (X^2 - \zeta^{2br(i)+1}) \\ &= \hat{f}_{2i}\hat{g}_{2i} + ((\hat{f}_{2i} + \hat{f}_{2i+1})(\hat{g}_{2i} + \hat{g}_{2i+1}) - \hat{f}_{2i}\hat{g}_{2i} - \hat{f}_{2i+1}\hat{g}_{2i+1})X + \hat{f}_{2i+1}\hat{g}_{2i+1}X^2 \\ &= \hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}\zeta^{2br(i)+1} + ((\hat{f}_{2i} + \hat{f}_{2i+1})(\hat{g}_{2i} + \hat{g}_{2i+1}) - \hat{f}_{2i}\hat{g}_{2i} - \hat{f}_{2i+1}\hat{g}_{2i+1})X \end{aligned} \tag{9}$$

As can be seen, the number of multiplications decreases from five to four, and $\hat{h}_{2i} + \hat{h}_{2i+1}X$ is now obtained by

$$\hat{h}_{2i} = \hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1} \cdot \zeta^{2br(i)+1} \tag{10}$$

$$\hat{h}_{2i+1} = (\hat{f}_{2i} + \hat{f}_{2i+1})(\hat{g}_{2i} + \hat{g}_{2i+1}) - (\hat{f}_{2i}\hat{g}_{2i} + \hat{f}_{2i+1}\hat{g}_{2i+1}) \tag{11}$$

The calculation process is divided and allocated in two cycles in our design, according to functionality of butterfly units. The way that each step is realized with a specific operator in butterfly units has been shown in Figure 3 and Table 1.
PWM0 : $s_0 = \hat{f}_{2i} + \hat{f}_{2i+1},\ s_1 = \hat{g}_{2i} + \hat{g}_{2i+1},\ m_0 = \hat{f}_{2i} \cdot \hat{g}_{2i},\ m_1 = \hat{f}_{2i+1} \cdot \hat{g}_{2i+1}$
PWM1 : $s_2 = m_0 + m_1,\ m_2 = s_0 \cdot s_1,\ m_3 = m_1 \cdot \zeta^{2br(i)+1},\ \hat{h}_{2i} = m_0 + m_3,\ \hat{h}_{2i+1} = m_2 - s_2$

3.4 Modular reduction

In Kyber a fixed modulus $q = 3329$ is used, and dedicated modular reduction method can be implemented to accelerate corresponding operations, especially modular multiplication in $\mathbb{Z}_q$. There exist two multiplications in generic Barrett or Montgomery reduction method, demanding extra DSP resources or forcing the reduction unit to work in a time-multiplexed way with other operations that involve multiplications. In fact, these two multiplications can be replaced with shifts and additions by making use of property of q. We adapt Barrett reduction method to modular multiplication in our design, for the sake of its clarity. Another advantage is that approximate quotient is also generated along with the remainder, which would be beneficial in Compress procedure during encryption phase.

Algorithm 4 Modified Barrett Reduction

Input: Integer $prod$ with maximum length 24 bits, and fixed modulo q being 3329_{10}
Output: $res = prod \bmod q$, $quo = \lceil prod/q \rfloor$

1: $\mu \leftarrow \lceil 2^{24}/q \rfloor = 2^{12} + 2^{10} - 2^{6} - 2^{4}$
2: $\widehat{quo} \leftarrow prod_{[23:12]} + prod_{[23:14]} - prod_{[23:18]} - prod_{[23:20]} \approx \lfloor prod \cdot \mu / 2^{24} \rfloor$
3: $diff \leftarrow prod_{[14:0]} - (\widehat{quo} + \widehat{quo}_{[3:0]} << 11 + \widehat{quo}_{[4:0]} << 10 + \widehat{quo}_{[6:0]} << 8)$
4: **switch** $(diff_{[14:12]})$
5: **case 0:** $q_{mux} \leftarrow 0, \quad quo \leftarrow \widehat{quo}$
6: **case 1:** $q_{mux} \leftarrow -q, \quad quo \leftarrow \widehat{quo} + 1$
7: **case 5:** $q_{mux} \leftarrow 3q, \quad quo \leftarrow \widehat{quo} - 3$
8: **case 6:** $q_{mux} \leftarrow 3q, \quad quo \leftarrow \widehat{quo} - 3$
9: **case 7:** $q_{mux} \leftarrow 2q, \quad quo \leftarrow \widehat{quo} - 2$
10: **default:** $q_{mux} \leftarrow 0, \quad quo \leftarrow \widehat{quo}$
11: **end switch**
12: $res \leftarrow diff + q_{mux}$
13: **if** $res > q$ **then**
14: $\quad res \leftarrow res - q, \quad quo \leftarrow quo + 1$
15: **end if**
16: **if** $res > \lfloor q/2 \rfloor$ **then**
17: $\quad quo \leftarrow quo + 1$
18: **end if**
19: **return** res, quo

Detailed algorithm is given out in Algorithm 4, and $\widehat{quo}$ in line 2 is an approximation of actual quotient $\left\lfloor \frac{prod}{q} \right\rfloor$. As $\widehat{quo} \in \left[\left\lfloor \frac{prod}{q} \right\rfloor - 1, \left\lfloor \frac{prod}{q} \right\rfloor + 3\right]$ from analysis, the difference between input $prod$ and $\widehat{quo} \cdot q$ lies in $[-3q, 2q)$. With follow-up steps presented in line 4 to 15, precise remainder can be obtained. Noticing in Algorithm 4 the intermediate value $diff$ is a signed 15-bit integer.

3.5 Encode and decode with shift register

Encode procedure packs polynomials into 32-bit block arrays. As the possible coefficient length (12 in $\hat{\mathbf{t}}$, d_u in $\mathbf{u}$, d_v in v) does not always divide 32-bit data bandwidth, points in polynomials need to be regularized before they are transmitted. Conversely, packed 32-bit block arrays should be decoded into data points with proper length, before they participate in calculation in subsequent procedure. Dedicated encoder and decoder are adopted in our design, both implemented with shift register. Detailed working flow and content in shift register during each cycle is clarified in Figure 6. For decoder, it requests 32-bit data blocks from FIFO whenever left bits in shift register are not enough to form one pair of data points with proper length. Decoded data points stream out continuously as long as corresponding FIFO is not empty, as input data rate is greater than output data rate. The encoder is working in a reversed way, receiving output data points continuously from two sets of butterfly units, then generating regularized 32-bit data blocks discontinuously.

Encoder and decoder can also be implemented with long buffer in the same way as [RB20] did to deal with length mismatch between coefficients in polynomial and data blocks stored in FIFO. Initially a 352-bit ($\mathrm{LCM}(2 \times d_u, 32)$) buffer should be involved. For decoder in server when $k = 4$ with $d_u = 11$, the buffer is filled after 11 cycles of data loading, then 22-bit data pieces can be read out from the most significant part or the least significant part, depending on shift direction in 16 cycles continuously. Dedicated cycles in data load drag down the overall performance severely. To address this issue, the authors in [RB20] proposed an improvement that data points can be fetched before the whole buffer

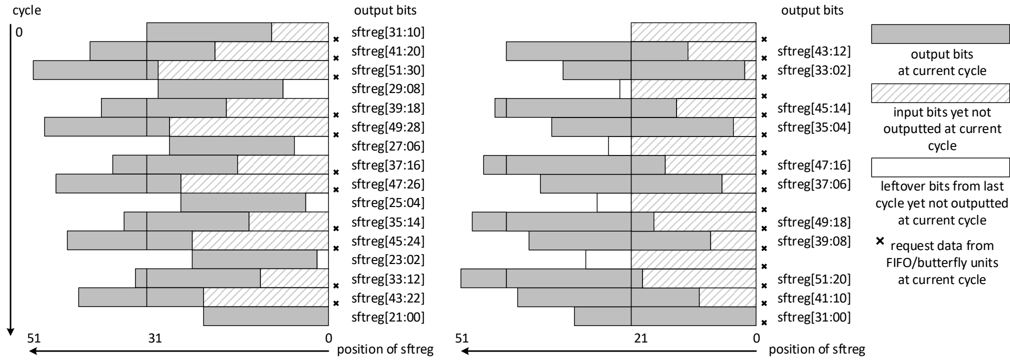

Figure 6: Detailed working flow and content in shift register during Decode($\mathbf{u}$) in decryption phase (left) and Encode($\mathbf{u}$) in encryption phase (right) with parameter $k = 4$

is filled, reducing both the buffer size and cycle overhead. However, the improvement comes at the cost of an extra large multiplexer, which would be a 11-to-1 22-bit-width one in our use case. Noticing in Figure 6, the multiplexer is 16-to-1 with the same 22-bit length, and the working flow is almost the same as long buffer in [RB20] whilst consuming much less register bits (52 in our design), the decoder adopted in our design should be advantageous. For $v = \text{Decode}(c_2)$ with different k, output bits are fetched from positions included in $\mathbf{u} = \text{Decode}(c_1)$ scenario. The longer output than needed (when $k < 4$ or during $v = \text{Decode}(c_2)$) would be truncated when fed into butterfly units for subsequent calculation. Decoder in client is more succinct as it receives 32-bit data and outputs fixed 24-bit data, corresponding to $\hat{\mathbf{t}} = \text{Decode}(pk)$.

3.6 Hash module

The hash module accounts for a great portion of the total resource consumption, thus special attention should be paid to reach an overall efficient design. It needs to support several function modes specified in SHA3 standard [oST15], including SHAKE-128, SHAKE-256, SHA3-256, SHA3-512. They share the same underlying function Keccak-f[1600], but with different rate r and different message suffix appended for domain separation. The constitution of r bits out of 1600-bit input is decided by the top module of either server or client. Parts of register data, appended suffix and part of padding data are selected through multiplexer under the control of a finite state machine, sent to ififo of hash module, as illustrated in Figure 2. Keccak core is working in an autonomous way that whenever ififo is not empty and it has done the last hash, data would be fetched from ififo until enough data has been collected to begin the next hash process. Detailed data flow is shown in Figure 7. It is especially useful in H(pk) and H(c) cases, where 32-bit data stream of pk or c flows into ififo and Keccak core decides when to fetch data from ififo. Three control signals are fed into ififo along with the 32-bit data, namely $mode, absorb, last$. $mode$ determines the number of capacity bits in Keccak-f[1600]. $absorb$ indicates the input data bits corresponding to rate part should be XORed with front r bits of the current 1600-bit Keccak state, which would happen in H(pk) and H(c) cases that one 24-round Keccak-f function can not fully absorb the input message bits. $last$ signal, as its name implies, indicates the current 32-bit data is the last block of message, and zeros corresponding to capacity should be appended to form the 1600 bits input to Keccak core. We adapt Keccak core from [OG17] to our design, which completes one round of Keccak-f function per cycle. One full 24-round Keccak-f function consumes 79 cycles, of which 25 are consumed in calculating hash value, the others mainly correspond to data input/output.

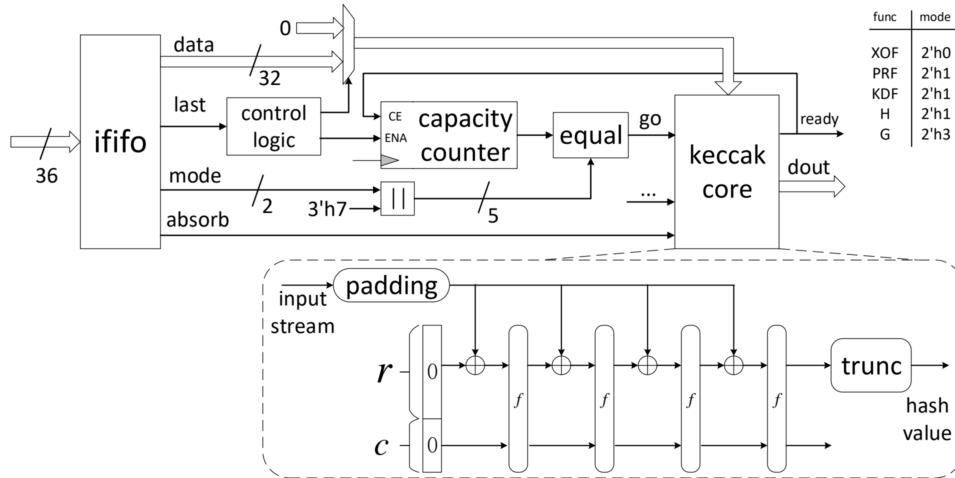

Figure 7: Detailed structure of hash module and sponge structure within Keccak core

The bandwidth of Keccak core is set to 32 bits, aiming to be compatible with software implementations, thus the 1600-bit internal state would return to its initial value after 50 cycles of shift, which would be helpful in Hash(pk) and Hash(c). As elements of matrix $\mathbf{A}/\mathbf{A}^{\mathbf{T}}$ are sampled from 12-bit pseudo random integers, the Keccak core should naturally generate 24-bit data per cycle in our design, which is not a divisor of 1600. Similarly, noise samples in $\mathbf{s}, \mathbf{e}, \mathbf{r}$ in Kyber512 ($\eta_1 = 3$) each consumes 6 pseudo random bits, then the problem that $6(\times 4)$ is not a divisor of 1600 arises as well. In both cases there would be samples split over two invocations of Keccak-f function, and updating internal state with data pieces from different indices would cause an obvious increase in total resource consumption. Considering these two kinds of samples are stored in separate places in our design, namely ofifo0 and ofifo1 to ease elaborate manipulation when read out, a data buffer is inserted between the Keccak core and ofifo0/ofifo1 in Figure 2, which receives 32-bit data pieces and generates 24-bit and 16-bit ($\eta_1, \eta_2 = 2$) output data. It consists of a small FIFO (32×32) and a small decoder, in a similar way with design rationale of Decode in the protocol.

The uniform and binomial distributed samples correspond to different input from register file (ρ/σ) as well as padding bits, as illustrated in Figure 2. Whether the samples should be fed into ofifo0/ofifo1 or not is determined before they are generated by the Keccak core. Considering the complexity in generating dedicated logic to control the order of sampling process, a predefined order table would be more suitable. The sort of samples that being generated is determined by the current bit fetched from order sequence, which we depict in Table 2. In Table 2, bit 1 (and specially 2&3) corresponds to centered binomial sampling, and bit 0 uniform random sampling. The underline beneath a certain bit indicates the end of sampling process, and a dedicated endpoint table highlights the endpoint position. The order sequence is fetched and stored in the 73-bit register *patt* upon reset signal is valid. The endpoint sequence, similarly, would be stored in the 73-bit *endp* concurrently. During sampling, the Keccak core would take register value and padding as input according to the most significant bit of *patt*, and *patt* would shift left one bit upon start signal *go* of the current Keccak process is valid. When the last bit corresponding to endpoint is fetched, the sampling process would terminate after current generation of samples and the top module moves to next procedure. The endpoint table is essential as the number of Keccak invocations varies among different system parameter k.

According to the specification of Kyber, noise polynomials in centered binomial distribution $\psi_{\eta_1}^n, \psi_{\eta_2}^n$ are generated by PRF, which is instantiated with SHAKE-256 and outputs

Table 2: Predefined order sequence and endpoint during sampling

Enc	k	order sequence
0	2	32000032000032000032000**0**0000000000000000000000000000000000000000000000000
1	2	3200003200000000100001**1**00000000000000000000000000000000000000000000000000
0	3	10000100001100000000100000000000010000000**0**0000000000000000000000000000000
1	3	100001000010000000010000000000001000000001**1**000000000000000000000000000000
0	4	10000100001110000000000001000000000000000010000000000000000100000000000**0**0
1	4	100001000011000000000000100000000000000001000000000000000010000000000001**1**

Enc	k	endpoint
0	2	0000000000000000000**1**00000000000000000000000000000000000000000000000000000
1	2	00000000000000000000**1**0000000000000000000000000000000000000000000000000000
0	3	00000000000000000000000000000000000000000**1**0000000000000000000000000000000
1	3	000000000000000000000000000000000000000000**1**000000000000000000000000000000
0	4	00000000000000000000000000000000000000000000000000000000000000000000000**1**0
1	4	000000000000000000000000000000000000000000000000000000000000000000000000**1**

1088 bits data after 24-round Keccak-f function. The parameter η_1 equals to 3 in $k = 2$ case, otherwise η_1 is set to 2, while η_2 always takes 2. For one 256-term polynomial obeying ψ_η^{256}, the total number of pseudo random bits needed is $2 \times \eta \times 256$, which would be as large as 1536, implying that the PRF in generating $\mathbf{s}, \mathbf{e}, \mathbf{r}$ corresponds to two consecutive invocations of Keccak-f function in Kyber512 case, illustrated as $'32'$ pairs in Table 2. In practice, both $'3'$ and $'2'$ need two bits space, so an auxiliary table that highlights the cases where $\eta_1 = 3$ is added along with order sequence table. Similarly, public matrix $\hat{\mathbf{A}}/\hat{\mathbf{A}}^T$ is sampled from pseudo random bit stream with XOF, which is instantiated with SHAKE-128 and outputs 1344 bits after 24-round Keccak-f function. Elements in matrix $\hat{\mathbf{A}}/\hat{\mathbf{A}}^T$ should be uniform random distributed in $\mathbb{Z}_q$, thus some samples are rejected when they fall outside the desired range. To be compatible with software implementation, the range is set to be $[0, q)$, corresponding to 12-bit pseudo random numbers, and the acceptance rate is $3329/2^{12} = 81.3\%$. Therefore, the number of generated elements in $\mathbb{Z}_q$ that one 24-round XOF generates is $1344/12 \times 81.3\% = 91.0$, implying with at least four XOFs can the generation of one element polynomial of $\hat{\mathbf{A}}/\hat{\mathbf{A}}^T$ in $\mathcal{R}_q$ be guaranteed, in consideration that *binocdf*$(255, 3 \times 112, 81.3\%) = 8.33 \times 10^{-3}$, which is not negligible. Sampling of one uniform polynomial in $\mathcal{R}_q$ corresponds to $'0000'$ bit strings (ahead of endpoint) in Table 2. Details would be further demonstrated in Subsection 4.2.

4 Minimizing the Memory Footprint

4.1 RAM dimension

In order to reduce memory footprint, only the points that will participate in later calculation whilst other participants are not ready at the time would be stored in memory. For example, polynomial $\hat{\mathbf{s}}_i$ for $i \in \{0, 1, 2, 3\}$ must be preserved after key generation phase in server side, as $\hat{\mathbf{s}}_i$ would multiply with $\hat{\mathbf{u}}_i$ during decryption, thus dedicated memory block must be allocated for storage of *each* $\hat{\mathbf{s}}_i$ and it is free to use only after decryption. The same goes for $\hat{\mathbf{r}}_i$ in client side. In other words, data points should be preserved in RAM during their lifecycle, and both $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$ can be stored in a single large RAM block as under no circumstances would different $\hat{\mathbf{s}}_i$ parts or $\hat{\mathbf{r}}_i$ parts be accessed concurrently. As aforementioned in Subsection 3.2, memory block is divided into two banks, with one pair of

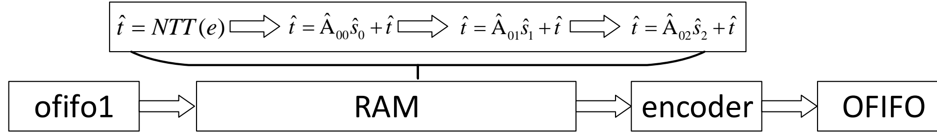

Figure 8: Data flow duing accumulation of $\hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{t}}$

points reside in the same place, corresponding to RAM0 and RAM1 in Figure 4. They are both in 24-bit width, 256-piece depth, adequate in $k = 4$ case. On the contrary, polynomial $\hat{\mathbf{e}}_i$ is only requested in $\hat{\mathbf{t}} = \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$. Noticing $\hat{\mathbf{t}}_i$ would be sent to encoder and then OFIFO once it is newly formed, as illustrated in Figure 2, both $\hat{\mathbf{e}}_i$ and $\hat{\mathbf{t}}_i$ can be overwritten. In fact, $\hat{\mathbf{t}}_i$ is obtained through accumulation, implemented in accumulator in Figure 4 (denoted as *acc* unit). The detailed procedures are depicted in Figure 8. Dedicated RAM should be allocated that is adequate for storing only one $\hat{\mathbf{t}}_i$. Similarly, one small RAM should be allocated in order to preserve intermediate values during accumulation of $\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}$ and subsequent INTT procedure. This part of memory corresponds to RAM2 and RAM3 in Figure 4, with each bank in 24-bit width, 64-piece depth.

In addition, one wider RAM block, depicted as RAM4, is involved and dedicated to PWM between two polynomials in NTT representation, including $\hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ in key generation phase, $\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}, \hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}$ in encryption phase as well as $\hat{\mathbf{s}} \circ \hat{\mathbf{u}}$ in decryption phase. Based on the observation that each PWM needs space as large as four intermediate points from Subsection 3.3, RAM4 is arranged as 48-bit-width, 128-piece-depth memory block. It is especially essential in $\hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ and $\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}$, where input $\hat{\mathbf{A}}$ and $\hat{\mathbf{A}}^T$ are fetched from ofifo0, $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$ fetched from RAM0/RAM1. The intermediate points can not be fully stored in the same place in RAM where elements of $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$ residue, as two points from RAM would explode to four intermediate ones, and ofifo0 is busy in generating other parts of public matrix $\hat{\mathbf{A}}$ or $\hat{\mathbf{A}}^T$ at the same time.

4.2 Specification of FIFOs

The depth of each FIFO depends on its functionality in our design. In Figure 2, the FIFOs with lowercase name ififo, ofifo, ofifo0 and ofifo1 reside in hash module. They are related with Keccak function and all serve as data buffer, thus their size should be adequate that no data is lost during transmission resulting from mismatch between read and write rate. The same goes for IFIFO in top module. Differently, DFIFO0, DFIFO1, and OFIFO in top module serve as data storage, thus their size is determined by the largest data block they would store.

IFIFO lies ahead of decoder and its size is determined by mismatch between input rate from client and processing ability of decoder. Detailed working flow of decoder in decoding polynomial $\mathbf{u}$ (with parameter $k = 4$) is illustrated in Figure 6. In 16 cycles, it outputs 22-bit data consecutively, and requests 32-bit data of c from IFIFO d_u times. Consequently, it needs $\frac{d_u \times 256 \times k}{32} \times \frac{16}{d_u} = 128k$ cycles to decode polynomial $\mathbf{u}$ from transmitted ciphertext c, which is greater than $\frac{d_u \times 256 \times k}{32} + \frac{d_v \times 256}{32} = 8kd_u + 8d_v$ in all cases when k, and correlated d_u, d_v take different values. As a result, the decoder is always decoding polynomial $\mathbf{u}$ when IFIFO receives ciphertext c from client, and in total $(8kd_u + 8d_v) \times \frac{d_u}{16} = \frac{1}{2}kd_u^2 + \frac{1}{2}d_u d_v$ 32-bit data is fetched from IFIFO, leaving behind $8kd_u + 8d_v - \frac{1}{2}kd_u^2 - \frac{1}{2}d_u d_v$ data, which equals to 72/102/122.5 when k takes 2/3/4. Consequently, the depth of IFIFO can be as small as 128, and the width is 32 bits, equaling to transmission bandwidth.

The size of DFIFO0, DFIFO1 is determined by the content that it stores, i.e. polynomial $\mathbf{u}/\hat{\mathbf{t}}$ and v respectively. Noticing $\mathbf{u}, \hat{\mathbf{t}}$ are both vector of 256-term polynomials while the width of $\mathbf{u}$ is smaller than $\hat{\mathbf{t}}$ in all cases when k takes different value, the DFIFO0 should

keccak s_0 $\hat{A}_{00}$ s_1 $\hat{A}_{01}$ s_2 e_0 $\hat{A}_{02}$ $\hat{A}_{10}$ e_1 $\hat{A}_{11}$ $\hat{A}_{12}$ $\hat{A}_{20}$ e_2 $\hat{A}_{21}$ $\hat{A}_{22}$
cycles 79 316 79 316 79 79 316 316 79 316 316 316 79 316 316
butterfly NTT(s_0) NTT(s_1) NTT(s_2) NTT(e_0) $\hat{A}_{00}\hat{s}_0$ $\hat{A}_{01}\hat{s}_1$ $\hat{A}_{02}\hat{s}_2$ NTT(e_1) $\hat{A}_{10}\hat{s}_0$ $\hat{A}_{11}\hat{s}_1$ $\hat{A}_{12}\hat{s}_2$ NTT(e_2) $\hat{A}_{20}\hat{s}_0$ $\hat{A}_{21}\hat{s}_1$ $\hat{A}_{22}\hat{s}_2$
cycles 512 512 512 512 256 256 256 512 256 256 256 512 256 256 256

keccak r_0 $\hat{A}^T_{00}$ r_1 $\hat{A}^T_{01}$ r_2 $\hat{A}^T_{02}$ $\hat{A}^T_{10}$ e'_0 $\hat{A}^T_{11}$ $\hat{A}^T_{12}$ $\hat{A}^T_{20}$ e'_1 $\hat{A}^T_{21}$ $\hat{A}^T_{22}$ e'_2 e''
cycles 79 316 79 316 79 316 316 79 316 316 316 79 316 316 79 79
butterfly NTT(r_0) NTT(r_1) NTT(r_2) $\hat{A}^T_{00}\hat{r}_0$ $\hat{A}^T_{01}\hat{r}_1$ $\hat{A}^T_{02}\hat{r}_2$ INTT(u_0) $\hat{A}^T_{10}\hat{r}_0$ $\hat{A}^T_{11}\hat{r}_1$ $\hat{A}^T_{12}\hat{r}_2$ INTT(u_1) $\hat{A}^T_{20}\hat{r}_0$ $\hat{A}^T_{21}\hat{r}_1$ $\hat{A}^T_{22}\hat{r}_2$ INTT(u_2)
cycles 512 512 512 256 256 256 576 256 256 256 576 256 256 256 576

Figure 9: Elaborate scheduling during sampling and corresponding NTT procedures

be set as 24×512 data storage, in order to fully save $\hat{\mathbf{t}}$ when k takes 4. Similarly, DFIFO1 should be set as 10×128 data storage.

OFIFO of server needs to store $\hat{\mathbf{t}}$ and publicseed ρ, in total $96k+8$ pieces of 32-bit data, whose maximal value is 392, implying the depth should be set to 512 at least. The width of OFIFO is 34. Besides 32-bit data transmitted to client, two flag bits are added in each piece, indicating whether the current piece corresponds to publicseed ρ and whether it is the last one that would be transmitted. The flag bits would not be sent to client, but rather are helpers that determine when the transmission is done and which data pieces should be sent back to the input port of OFIFO when they are requested and outputted in preparation for later use in re-encryption process $v = \mathrm{NTT}^{-1}\left(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}\right) + e'' + \mathrm{Decompress}_q\left(\mathrm{Decode}_1(m), 1\right)$. More details would be given out in Subsection 4.3.

As for data buffer ififo in hash module, the worst case happens when server/client requests c/pk data from OFIFO in client/server. At this time, continuous data stream sent to server/client would also be stored in ififo, corresponding to procedure $\mathrm{H}(c)/\mathrm{H}(pk)$. From Kyber specification we know H function is instantiated as SHA3-256 from FIPS-202 standard, where $1088 = 32 \times 34$ bits are absorbed in one 24-round Keccak-f function. In our design one 24-round Keccak-f would take 79 cycles. Consequently, the ratio of speed between continuous data stream fed into ififo and discontinuous data request from Keccak core is 79:34, thus the leftover 32-bit data in ififo during $\mathrm{H}(c)$ would be $(8kd_u + 8d_v) \times (1 - 34/79) = 109/155/223$ when k takes 2/3/4. For $\mathrm{H}(pk)$, that would be $(96k + 8) \times (1 - 34/79) = 114/169/223$. As a result, the lowest yet adequate depth of ififo is 256, and the width should be 32 bits, determined by bandwidth of Keccak core.

Up to now, we obtain the size constraint of IFIFO, DFIFO0, DFIFO1, OFIFO and ififo from two factors, namely the worst case of rate mismatch in data buffer and largest data block that storage would preserve. In both cases, there would be data losses when corresponding FIFO is not large enough. For ofifo0 and ofifo1, the constraint is looser in terms of their size. When they are not enough for storing the whole newly-sampled data pieces needed by following processing module, bubbles would be inserted into calculation procedure such that enough data can be collected and transmitted, implying the performance would be dragged down. On the other hand, we can arrange the sampling procedure elaborately to decrease the size of ofifo0 and ofifo1.

Now we demonstrate how the sampling procedures are arranged in cooperation with NTT calculation, with details illustrated in Figure 9. Firstly, one NTT on a certain polynomial in $\mathcal{R}_q$ would take $128/2 \times \log_2 128 = 448$ cycles. In $\mathrm{NTT}(\mathbf{s}_i), \mathrm{NTT}(\mathbf{e}_i), \mathrm{NTT}(\mathbf{r}_i)$, process that writing data into RAM takes another 64 cycles, and in Figure 9 they are merged into the cycles relating with NTT for clarity, amounting to 512 cycles in total. Similarly, cycles corresponding to $\mathrm{INTT}(\mathbf{u}_i)$ includes compression procedure in addition to INTT, which takes 128 cycles. During one NTT or INTT, one noise polynomial *and* one element polynomial of public matrix $\hat{\mathbf{A}}/\hat{\mathbf{A}}^T$ are guaranteed to be generated. Secondly,

one point-wise multiplication between $\hat{\mathbf{A}}_{ij}/\hat{\mathbf{A}}_{ij}^T$ and $\hat{\mathbf{s}}_i/\hat{\mathbf{r}}_i$ takes 256 cycles. As a result, during two such multiplications, one element of public matrix and one noise polynomial are guaranteed to be sampled when $k = 2$. For $k = 3$, during three such multiplications, two elements of public matrix and one noise polynomial are guaranteed to be sampled. Similarly, the number would be three and one respectively in $k = 4$ case. To this end, we present a compact sampling procedure that cooperates with NTT-related calculation. The complete sequences in both server and client are listed before in Table 2. Noticing Figure 9 means to illustrate the margin between sampling and corresponding NTT. In practice, the sampling procedures are continuous except when corresponding ofifo that data should be sent to is full. At that moment, sampling would be halted until corresponding ofifo is ready to receive new samples. Basically, it follows the so called *just-in-time* principle [HOKG18][KMRV18][BUC19], which corresponds to minimal memory footprint. The compactness comes from two aspects, of which the first refers to clock cycles, i.e. the samples generated from Keccak core are always ready to be fetched from ofifo where they are stored, thus no bubble needs to be inserted and performance would not be affected by sampling. The other goes to the size of ofifo0 and ofifo1. ofifo0 is designed to store only two elements of matrix $\hat{\mathbf{A}}$ or $\hat{\mathbf{A}}^T$, and ofifo1 only one element of noise polynomial vector $\mathbf{s}, \mathbf{e}, \mathbf{r}, \mathbf{e}', e''$. Thus the size of ofifo0 and ofifo1 is set to 24×256 and 24×64 respectively. Noticing the bandwidth of ofifo1 is 24 bits, enough for 4 centered binomial distributed samples when $\eta_1 = 3$. They are transformed to samples in NTT module before written into RAM, as illustrated in Figure 4.

4.3 Making full use of data storage FIFOs

In Kyber.CCAKEM.Dec, re-encryption process is required, which implements almost all the procedures in client. Besides the increase of computational complexity, an important problem arises that data stream of c_1, c_2, received from client during decryption phase as well as data points of $\hat{\mathbf{t}}$ in key generation phase would be requested again in the re-encryption phase. Message c_1 and c_2 would be requested and compared with the newly formed c_1' and c_2' to validate correctness of decryption, influencing generation of the final agreed key, and $\hat{\mathbf{t}}$ would be requested in restoring message v' with $v' = \mathrm{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}') + e'' + \mathrm{Decompress}_q\left(\mathrm{Decode}_1(m'), 1\right)$, implying these related message must be stored properly in some place. Instead of initializing more memory blocks, we make full use of the data storage FIFOs, in consideration that they are free to use in re-encryption phase after occupation in corresponding process.

To implement the strategy, output data of DFIFO0/DFIFO1 and OFIFO are redirected to its corresponding input port if they are not working in re-encryption phase. For DFIFO0 and DFIFO1, the redirected data points actually belong to $\mathrm{Compress}_q(\mathbf{u}, d_u)$ and $\mathrm{Compress}_q(v, d_v)$ respectively, and they are compared with $\mathrm{Compress}_q(\mathbf{u}', d_u)$ and $\mathrm{Compress}_q(v', d_v)$ in re-encryption directly, without further resorting to Encode procedure. The results of above comparisons are equivalent to comparison between c_1, c_2 and c_1', c_2', with two encode procedures eliminated. Different from DFIFO0 and DFIFO1 where all data points are needed in later comparison, register value ρ stored in OFIFO would not be requested. To distinguish which part of data points would be redirected, two flag bits are added, expanding the width of OFIFO from 32 bits to 34 bits, as depicted in Subsection 4.2. Noticing the redirected message is actually $\mathrm{Encode}(\hat{\mathbf{t}})$, a decode process should be involved such that the message is unpacked into data points and ready for participating in point-wise multiplication with $\hat{\mathbf{r}}'$ in re-encryption phase. The decoded data points are fed into DFIFO0, just after $\mathrm{Compress}_q(\mathbf{u}, d_u)$ being read out for comparison. Detailed data flow is depicted in Figure 2.

Table 3: Cycles counts/execution time (μs) of Kyber with different module dimension k at different protocol phases. Execution time is calculated at maximum frequency

k	key generation	encapsulation	decapsulation
2	3768/23.4	5079/30.5	6668/41.3
3	6316/39.2	7925/47.6	10049/62.3
4	9380/58.2	11321/67.9	13908/86.2

5 Results, Comparison and Discussion

The proposed design of CRYSTALS-KYBER has been simulated, synthesized and implemented with Vivado 2017.3 design suite on Xilinx XC7A12TCPG238-1 device, with all building blocks implemented in hardware. All module dimension k are supported and can be selected through dedicated input port. Bandwidth of data transmission in both server and client is set to be 32 bits, in order to be compatible with bandwidth of hash module.

5.1 Results

Cycle counts at different protocol phases with different parameter k are depicted in Table 3. The critical path delay is 6.2/6.0 ns, corresponding to 161/167 MHz maximum frequency respectively in server/client, and execution time of each phase is calculated accordingly. Detailed cycle counts corresponding to each procedure in server side when k takes 3 is listed in Table 8, where procedures conducted concurrently in NTT core and hash module reside in the same line. The execution time of sampling, both noise polynomials and public matrices, is basically hidden behind NTT and related PWM processes. The procedures H(pk) and H(c) are conducted simultaneously along with transmission/reception of pk and c. Overall, the execution time occupied by pure hash function accounts for rather small portion of the total time, as a result of elaborate scheduling between sampling and NTT procedures, as depicted in Figure 9. Consequently, timing performance is determined by computational ability of NTT cores, or more specifically butterfly units to a great extent. Key generation, encapsulation and decapsulation phase in CCA-secure Kyber can be completed in 23.4/30.5/41.3 μs when $k = 2$, and 39.2/47.6/62.3 μs when $k = 3$, corresponding figures would be 58.2/67.9/86.2 μs when $k = 4$.

Resource consumption of our proposed design is depicted in Table 4, where occupation by major submodules are also listed out. As can be seen, Keccak core accounts for more than 40% LUTs and 35% FFs of total consumption, resulting from that a high speed Keccak design is involved to keep up with computational ability of butterfly units, generating enough samples before they are required. The second most area-consuming module is NTT core, where two sets of butterfly units are allocated, cooperating with each other to realize several extra functionalities in addition to conventional use in NTT and INTT, as depicted in Table 1. There are differences in input/output multiplexing relating to inner adder/subtracter of each butterfly unit as well as intercommunication between two sets of them, so they are implemented in a single butterfly module, where two reduction units are adopted. In NTT core, RAM0 and RAM1, arranged as 24×256 memory block, each consumes one 18k BRAM. Similarly, RAM2 and RAM3, arranged as 24×64 memory block, each consumes one 18k BRAM, but only occupies $\frac{1}{12}$ storage capacity. Differently, RAM4 consumes one 36k BRAM because that its data width is 48 bits, exceeding 36-bit boundary that one 18k BRAM can hold. In total, five RAM blocks occupy three 36k BRAMs. Twiddle factors as well as scaling factors are stored in distributed RAM, in consideration that each one of ROM0, ROM1, ROM2 contains only 12×128 data bits, used in NTT, INTT, PWM respectively. Overall, our design consumes 7412/6785 LUTs, 4644/3981 FFs, 2126/1899 slices, 3/3 BRAMs and 2/2 DSPs in server/client side, and can

Table 4: Resource consumption of server/client and major submodules involved

module	LUT	FF	BRAM	DSP
hash core	4014/3825	1980/1980	0/0	0/0
⌊ Keccak	2966/2956	1610/1610	0/0	0/0
⌊ ififo	263/263	72/72	0/0	0/0
⌊ ofifo	45/45	56/56	0/	0/
⌊ ofifo0	187/187	60/60	0/0	0/0
⌊ ofifo1	60/60	53/53	0/0	0/0
NTT core	1737/1579	1167/1058	3/3	2/2
⌊ butterfly	708/647	574/501	0/0	2/2
×2 ⌊ reduction	135/135	96/96	0/0	0/0
IFIFO	132/131	64/64	0/0	0/0
DFIFO0	376/352	92/70	0/0	0/0
DFIFO1	60/–	42/–	0/–	0/–
OFIFO	484/463	80/78	0/0	0/0
decoder	239/57	99/86	0/0	0/0
encoder	70/144	53/59	0/0	0/0
total	7412/6785	4644/3981	3/3	2/2

fit in the smallest device in Xilinx Artix-7 series (XA7A12 series with different package and speed, providing 8000 LUTs, 16000 FFs, 20 BRAMs and 40 DSPs).

5.2 Comparison with related works

Comparison with related works that concentrate on hardware implementation of contest candidates, especially CRYSTALS-KYBER is demonstrated in Table 5. Both [DFA+20] and [HHLW20] report performance and resource consumption of standalone hardware implementation of Kyber. In [DFA+20], besides summarizing and analyzing massive results reported by other groups, four CCA-secure KEM candidates of the second round are implemented in pure hardware as the authors report, including Kyber. Compared with [DFA+20], our proposed design is more than two times slower, resulting from 1.5 – 2× more cycle counts and lower maximum frequency. It needs to be noted that key generation is assumed to be performed in software in [DFA+20] and only one module dimension k is supported at a time. As [DFA+20] does not give out any detail of either hardware architecture or implementation procedures, we can only make some speculations from the reported resource consumption. Speed grade of our device is -1, which is the slowest among the device family. Beside, more than two times of FFs are utilized in [DFA+20] compared with our design, facilitating shorter critical path, thus higher maximum frequency can be obtained. 4× of DSPs, 5× of BRAMs as well as 1.5× LUTs are involved, implying more computational cores participate in calculation and total cycle counts would decrease. In [HHLW20], BRAMs are inserted between different modules for communication, and bottom modules are reused in realizing related upper functionalities extensively. Through elaborate scheduling with limited computational resources, cycle counts of our design is 10 times smaller than that of [HHLW20], and resource consumption corresponding to LUT and FF are more than 10× and nearly 40× smaller as well.

[BUC19][FSS20][AEL+20] are all crypto-processors constructed upon a RISC-V core. The RISC-V soft core is implemented in fabric, configured in pipeline structure with different depth. Accelerators are inserted into the pipeline, and would be invoked in the same way as arithmetic logical unit (ALU) by user defined instructions, which is extended based on existing instruction set architecture (ISA). Procedures in the protocol would be translated to supported instructions, and conducted in the RISC-V core. Total cycle counts

Table 5: Comparison with related works. Designs are deployed on Xilinx Artix-7 device unless otherwise specified. Three rows correspond to cases when k takes 2,3,4 respectively. The slashes denote separation of three phases: key generation, encapsulation and decapsulation.

	cycle counts (k)	freq. MHz	execution time μs	area[1]				
				LUT	FF	slice	BRAM	DSP
This work	3.8/5.1/6.7 6.3/7.9/10.0 9.4/11.3/13.9	161	23.4/30.5/41.3 39.2/47.6/62.3 58.2/67.9/86.2	7412	4644	2126	3	2
[DFA+20]	–/3.0/4.4	210	–/14.3/20.9	11864	10348	3989	15	8
	–/4.0/5.6		–/19.2/26.5	11884	10380	3984	15	8
	–/5.7/7.4		–/27.4/35.2	12183	12441	4511	15	8
[HHLW20][2]	–/49.0/68.8	155	–/316/444	88901	152875	–	202	354
	–/77.5/102.1		–/500/659	110260	167293		202	292
	–/107.1/135.6	192	–/558/706	132918	172489		202	548
[BUC19][3]	74.5/131.7/142.3 111.5/177.5/190.6 148.5/223.5/241.0	25	2980/5268/5692 4461/7102/7623 5942/8939/9639	14975	2539	4173	14	11
[FSS20]	150.1/193.1/204.8 273.4/325.9/340.4 349.7/405.5/424.7	–	–	24306	10837	–	32	18
[AEL+20]	710/971/870 – 2203/2619/2429	59	12034/16458/14746 – 37339/44390/41169	1842	1634	–	34	5
[BSNK19][4]	–/31669/43018	67	–/475/645	1977896	194126	–	–	–
[RB20][5]	2.8/4.0/5.0 5.5/6.6/8.0 9.0/10.3/12.3	150	18.4/26.9/33.6 36.4/44.1/53.6 60.2/68.4/82.0	24950	10720	–	2	0

1 Figures correspond to server side when resource consumption of two sides are summarized separately.
2 Figures in last row corresponds to Vertex-7 device, as hardware resources are not sufficient to allocate the whole design when module dimension $k = 4$. Speed grade of both devices is -2. Figures corresponding to FFs are originally counted as slice consumption, which according to 7-series product selection guide exceed the maximal available number of corresponding devices (33650 and 75900).
3 Resource consumption corresponds to *Sapphire* crypto-core, excluding the RISC-V micro processor.
4 HLS-based implementation of Kyber when $k = 2$.
5 Hardware implementation of Saber on Xilinx UltraScale+ XCZU9EG-2FFVB1156 device, which is manufactured in 16 nm technology.

would generally be much larger compared with dedicated, standalone design. Besides, maximum frequency is subject to timing performance of the RISC-V core severely. It can be seen from Table 5, our design is hundreds of times faster than RISC-V-based designs, while their resource consumption varies in a large range. It can be as little as 1842 LUTs and 1634 FFs in [AEL+20], while it can also be as large as 24306 LUTs and 10837 FFs in [FSS20]. In summary, a general purpose computational platform is able to implement cryptographic protocol, but it possesses no particular advantage except flexibility in deploying different protocols.

[BSNK19] is based on HLS design methodology, aiming to overcome the long deployment time problem. A widely accepted view is that results from HLS design are much worse, in terms of both timing performance and resource utilization than designs obtained through manual hardware description language (HDL) coding, which is also true in comparison between [BSNK19] and our design.

In addition to related hardware implementation of Kyber on FPGA, [RB20] is a perfect counterpart as comparison work, in which a similar module lattice-based KEM protocol Saber [DKRV] is implemented. Saber is another finalist in the third round contest, basing its security on module learning-with-rounding problem. Dimension of polynomial ring $n = 256$ is the same with Kyber, as well as module dimension parameter $k = 2, 3, 4$. Binomial distributed noise is also adopted with parameter $\eta = 3, 4, 5$, corresponding to different k. As NTT is not directly applicable in Saber, NTT procedures are not involved in its protocol, making it possible to take advantage of small norm of noise polynomials during polynomial multiplications by simple schoolbook method. 256 multiply-and-accumulate units are instantiated in parallel, being able to compute one polynomial multiplication in only 256 cycles. Besides, as noise term is within a small range, such multiplication in field $\mathbb{Z}_q$ can be realized with simple shifts and additions, which means the whole design can be implemented without DSP. Timing performance of our design is pretty close to that in [RB20], while LUT, FF consumption is more than $3\times$,$2\times$ less than the design of Saber.

5.3 Discussions

5.3.1 Discussions about performance and resource utilization

Our presented design targets to explore intrinsic relation between two core modules, NTT and Keccak to achieve decent performance with limited computational resources. The second purpose is to give out a design method in implementing procedures involved in the Fujisaki-Okamoto transform through reusing as much hardware structure and memory as possible. In pursuit of higher performance design, more butterfly units should be exploited, with other modules keeping up with data input/output requirement of them. The butterfly cores can be organized as vector processor, with dedicated permutation networks located in front of and behind them to arrange data points in desired order at the current stage of NTT process [PNPM15][XHY+20]. Alternatively, constant geometry NTT [Pea68] can be adopted to avoid complex and variable read/write pattern in different stages [BUC19][XL20]. Noticing Kyber is based on MLWE construction, NTT process is conducted on each element in $\mathcal{R}_q$ within a vector separately, implying only one extra memory block for storing one element in $\mathcal{R}_q$, resulting from the out-of-place property of the strategy, would be adequate. With more butterfly cores, the number of reduction units, located behind multiplier in butterfly units should also be increased accordingly. Besides, RAM structure should be modified elaborately to ensure enough data bandwidth [BUC19][XL20]. Similarly, bandwidth of Keccak core needs to be tuned accordingly, which is relatively easy as data input/output and round function are conducted separately. In a low area design contrarily, the first thing to do is adopting a low-cost Keccak core [Tea20] where Keccak round function is conducted in multiple cycles, as many works have reported hash module accounts for a great portion of the overall resource consumption.

5.3.2 Discussions about a constant-time design

Almost all the procedures in the CCA-secure Kyber are implemented in constant-time in our design, except the generation of public matrices $\mathbf{A}/\mathbf{A}^{\mathbf{T}}$. As samples of $\mathbf{A}/\mathbf{A}^{\mathbf{T}}$ are obtained from pseudo random bits with abortion, the total cycles would be variable with different publicseed ρ. However, this would not cause any trouble in a constant-time design from two aspects. Firstly, the publicseed ρ and the generation algorithm Parse,XOF are all public, which means the variation in time would not leak any information that should be private. Secondly, the generation processes are conducted along with NTT-related calculations, and the number of total cycles consumed in our design is irrelevant with these processes. With these facts, our design should be resistant to timing attacks.

6 Conclusion

In this work, a pure hardware implementation of CRYSTALS-KYBER, one of the third round finalists of PQC contest is presented. The whole implementation is manually designed, without resort to either hard-wired processor such as ARM Cortex series or soft core that can be implemented with reconfigurable logic such as popular RISC-V processors. Achievable performance is explored with limited computational resources. Through elaborate scheduling of NTT related procedures and noise sampling, our design achieves decent performance and it can fit in the smallest device in Xilinx Artix-7 series FPGA. The number of utilized butterfly units is two in our design on the basis of special form of the NTT in Kyber, and we also discuss different solutions in cases where either higher performance or lower cost design is demanded. Similar scheduling method can be derived according to relation of speed between NTT core and hash module. In addition, detailed design methods corresponding to data transmission, reception and the Fujisaki-Okamoto transform are illustrated, through adoption of a set of FIFOs with different specification. We further explore minimal memory footprint of each storage block, and avoid involving more memory blocks in re-encryption phase through data redirection of corresponding FIFOs.

Overall, our pure hardware implementation can execute key generation, encapsulation, decapsulation in 23.4/30.5/41.3 μs when module dimension k takes 2, in 39.2/47.6/62.3 μs when k takes 3, and in 58.2/67.9/86.2 μs when k takes 4, with 7412/6785 LUTs, 4644/3981 FFs, 2126/1899 slices, 2/2 DSPs and 3/3 BRAMs consumed in server/client side, shedding light on achievable performance in a standalone hardware design. The results verify computational efficiency of schemes based on structured lattice, and better performance of manual hardware design compared with that implemented with hardware-software co-design or HLS method.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant NO.61974083.

References

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the nist post-quantum cryptography standardization process. *NIST, Tech. Rep., July*, 2020.

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020, Issue 3:219–242, 2020.

[BSNK19] Kanad Basu, Deepraj Soni, Mohammed Nabeel, and Ramesh Karri. NIST Post-Quantum Cryptography-A Hardware Evaluation Study. *IACR Cryptol. ePrint Arch.*, 2019:47, 2019.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version). Cryptology ePrint Archive, Report 2019/1140, 2019. https://eprint.iacr.org/2019/1140.

[DFA+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. *Cryptology ePrint Archive: Report 2020/795*, 2020.

[DKRV] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Mod-LWR based KEM (round 3 submission), 2020. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[FDAG19] Farnoud Farahmand, Viet Ba Dang, Michal Andrzejczak, and Kris Gaj. Implementing and benchmarking seven round 2 lattice-based key encapsulation mechanisms using a software/hardware codesign approach. In *Second PQC Standardization Conference*, 2019.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020, Issue 4:239–280, 2020.

[HHLW20] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. *IEICE Electronics Express*, pages 17–20200234, 2020.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018, Issue 3:372–393, 2018.

[KMRV18] Angshuman Karmakar, Jose M. Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018, Issue 3:243–266, 2018.

[LS12] Adeline Langlois and Damien Stehle. Worst-Case to Average-Case Reductions for Module Lattices. Cryptology ePrint Archive, Report 2012/090, 2012. https://eprint.iacr.org/2012/090.

[NCD19] Hamid Nejatollahi, Rosario Cammarota, and Nikil Dutt. Flexible ntt accelerators for RLWE lattice-based cryptography. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 329–332. IEEE, 2019.

[NVB+20] Hamid Nejatollahi, Felipe Valencia, Subhadeep Banik, Francesco Regazzoni, Rosario Cammarota, and Nikil Dutt. Synthesis of flexible accelerators for early adoption of ring-LWE post-quantum cryptography. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(2):1–17, 2020.

[OG17] Tobias Oder and Tim Güneysu. Implementing the NewHope-Simple key exchange on low-cost FPGAs. In *International Conference on Cryptology and Information Security in Latin America*, pages 128–142. Springer, 2017.

[oST15] National Institute of Standards and Technology. FIPS PUB 202 SHA-3 standard: Permutation-based hash and extendable-output functions. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf, 2015.

[Pea68] Marshall C Pease. An adaptation of the fast fourier transform for parallel processing. *Journal of the ACM (JACM)*, 15(2):252–264, 1968.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating Homomorphic Evaluation on Reconfigurable Hardware. In *CHES*, pages 143–163. Springer, 2015.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015.

[RB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020, Issue 4:443–466, 2020.

[Reg04] Oded Regev. New lattice-based cryptographic constructions. *Journal of the ACM (JACM)*, 51(6):899–942, 2004.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE Cryptoprocessor. In *CHES*, pages 371–391. Springer, 2014.

[SAB+] P Schwabe, R Avanzi, J Bos, L Ducas, E Kiltz, T Lepoint, V Lyubashevsky, JM Schanck, G Seiler, and D Stehle. CRYSTALS-Kyber–Algorithm Specifications And Supporting Documentation. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[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.

[Tea20] Keccak Team. Keccak in VHDL. https://keccak.team/hardware.html, accessed on October. 2020.

[XHY+20] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng. VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2672–2684, 2020.

[XL20] Y. Xing and S. Li. An Efficient Implementation of the NewHope Key Exchange on FPGAs. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(3):866–878, 2020.

Appendix

Table 6: Parameter sets for Kyber

	n	k	q	η_1	η_2	d_u, d_v
Kyber512	256	2	3329	3	2	10,4
Kyber768	256	3	3329	2	2	10,4
Kyber1024	256	4	3329	2	2	11,5

Table 7: Instantiation of different symmetric primitives from the FIPS-202 standard

XOF	H	G	PRF(s,b)	KDF
SHAKE-128	SHA3-256	SHA3-512	SHAKE-256$(s\|\|b)$	SHAKE-256

$$\text{Compress}_q(x,d) = \lceil (2^d/q) \cdot x \rfloor \bmod {}^+ 2^d \tag{12}$$

$$\text{Decompress}_q(x,d) = \lceil (q/2^d) \cdot x \rfloor \tag{13}$$

Algorithm 5 KYBER.CPAPKE.KeyGen(): key generation

Output: Secret key $sk \in \mathcal{B}^{12\cdot k\cdot n/8}$
Output: Public key $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$
1: $d \leftarrow \mathcal{B}^{32}$
2: $(\rho, \sigma) := \text{G}(d)$
3: $N := 0$
4: **for** i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k\times k}$ in NTT domain
5: **for** j from 0 to $k-1$ **do**
6: $\hat{\mathbf{A}}[i][j] := \text{Parse}(XOF(\rho, j, i))$
7: **end for**
8: **end for**
9: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from B_η
10: $\mathbf{s}[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$
11: $N := N+1$
12: **end for**
13: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from B_η
14: $\mathbf{e}[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$
15: $N := N+1$
16: **end for**
17: $\hat{\mathbf{s}} := \text{NTT}(\mathbf{s})$
18: $\hat{\mathbf{e}} := \text{NTT}(\mathbf{e})$
19: $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
20: $pk := \left(\text{Encode}_{12}\left(\hat{\mathbf{t}} \bmod {}^+q\right) \| \rho\right)$ ▷ $pk := \mathbf{As} + \mathbf{e}$
21: $sk := \text{Encode}_{12}\,(\hat{\mathbf{s}} \bmod {}^+q)$ ▷ $sk := \mathbf{s}$
22: return (pk, sk)

Algorithm 6 KYBER.CPAPKE.Enc (pk, m, r) : encryption

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

1: $N := 0$
2: $\hat{\mathbf{t}} := \text{Decode}_{12}(pk)$
3: $\rho := pk + 12 \cdot k \cdot n/8$
4: **for** i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
5: **for** j from 0 to $k-1$ **do**
6: $\hat{\mathbf{A}}^T[i][j] := \text{Parse}(\text{XOF}(\rho, i, j))$
7: **end for**
8: **end for**
9: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from B_η
10: $\mathbf{r}[i] := \text{CBD}_{\eta_1}(\text{PRF}(r, N))$
11: $N := N + 1$
12: **end for**
13: **for** i from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}' \in R_q^k$ from B_η
14: $\mathbf{e}'[i] := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$
15: $N := N + 1$
16: **end for**
17: $e'' := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ ▷ Sample $e'' \in R_q$ from B_η
18: $\hat{\mathbf{r}} := \text{NTT}(\mathbf{r})$
19: $\mathbf{u} := \text{NTT}^{-1}\left(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}\right) + \mathbf{e}'$ ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}'$
20: $v := \text{NTT}^{-1}\left(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}\right) + e'' + \text{Decompress}_q\left(\text{Decode}_1(m), 1\right)$
21: ▷ $v := \mathbf{t}^T\mathbf{r} + e'' + \text{Decompress}_q(m, 1)$
22: $c_1 := \text{Encode}_{d_u}\left(\text{Compress}_q\left(\mathbf{u}, d_u\right)\right)$
23: $c_2 := \text{Encode}_{d_v}\left(\text{Compress}_q\left(v, d_v\right)\right)$
24: **return** $c = (c_1 \| c_2)$ ▷ $c := \left(\text{Compress}_q\left(\mathbf{u}, d_u\right), \text{Compress}_q\left(v, d_v\right)\right)$

Algorithm 7 KYBER.CPAPKE.Dec (sk, c) : decryption

Input: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$

1: $\mathbf{u} := \text{Decompress}_q\left(\text{Decode}_{d_u}(c), d_u\right)$
2: $v := \text{Decompress}_q\left(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v\right)$
3: $\hat{\mathbf{s}} := \text{Decode}_{12}(sk)$
4: $m = \text{Encode}_1\left(\text{Compress}\left(v - \text{NTT}^{-1}\left(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})\right), 1\right)\right)$
5: ▷ $m := \text{Compress}_q\left(v - \mathbf{s}^T\mathbf{u}, 1\right)$
6: **return** m

Table 8: Detailed procedures in server side and corresponding cycle counts in implementing Kyber.CCAKEM protocol when k=3.

operation	cycles
$d \xleftarrow{\$} \mathcal{B}^{32}$	
$\{\rho, \sigma\} \leftarrow \text{G}(d)$	79
$\mathbf{s}_0 \leftarrow \text{PRF}(\cdot)$	79

Table 8: Detailed procedures in server side and corresponding cycle counts in implementing Kyber.CCAKEM protocol when k=3 (continued).

operation	cycles
$\hat{\mathbf{s}}_0 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{s}_0)), A_{00} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{s}_1 \leftarrow \text{PRF}(\cdot)$	512
$\hat{\mathbf{s}}_1 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{s}_1)), A_{01} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{s}_2 \leftarrow \text{PRF}(\cdot)$	512
$\hat{\mathbf{s}}_2 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{s}_2)), \mathbf{e}_0 \leftarrow \text{PRF}(\cdot)$	512
$\hat{\mathbf{e}}_0 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{e}_0))$	512
$acc \leftarrow A_{00}\hat{\mathbf{s}}_0 + \hat{\mathbf{e}}_0, A_{02} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$acc \leftarrow A_{01}\hat{\mathbf{s}}_1 + acc, A_{02} \leftarrow \text{Parse}(\text{XOF}(\cdot)), A_{10} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$\hat{\mathbf{t}}_0 \leftarrow A_{02}\hat{\mathbf{s}}_2 + acc, A_{10} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{e}_1 \leftarrow \text{PRF}(\cdot)$	256
$\hat{\mathbf{e}}_1 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{e}_1)), A_{11} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	512
$acc \leftarrow A_{10}\hat{\mathbf{s}}_0 + \hat{\mathbf{e}}_1, A_{12} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$acc \leftarrow A_{11}\hat{\mathbf{s}}_1 + acc, A_{12} \leftarrow \text{Parse}(\text{XOF}(\cdot)), A_{20} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$\hat{\mathbf{t}}_1 \leftarrow A_{12}\hat{\mathbf{s}}_2 + acc, A_{20} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{e}_2 \leftarrow \text{PRF}(\cdot)$	256
$\hat{\mathbf{e}}_2 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{e}_2)), A_{21} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	512
$acc \leftarrow A_{20}\hat{\mathbf{s}}_0 + \hat{\mathbf{e}}_2, A_{22} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$acc \leftarrow A_{21}\hat{\mathbf{s}}_1 + acc, A_{22} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$\hat{\mathbf{t}}_2 \leftarrow A_{22}\hat{\mathbf{s}}_2 + acc$	256
transmit $pk = \hat{\mathbf{t}}\|\|\rho$ to client side, h=H(pk)	696
receive $c = c_1\|\|c_2$ from client side, H(c)	713
$\mathbf{u}_0 \leftarrow \text{Decompress}(c_1), \hat{\mathbf{u}}_0 \leftarrow \text{NTT}(\mathbf{u}_0)$	576
$acc \leftarrow \hat{\mathbf{u}}_0 \cdot \hat{\mathbf{s}}_0 + 0$	256
$\mathbf{u}_1 \leftarrow \text{Decompress}(c_1), \hat{\mathbf{u}}_1 \leftarrow \text{NTT}(\mathbf{u}_1)$	576
$acc \leftarrow \hat{\mathbf{u}}_1 \cdot \hat{\mathbf{s}}_1 + acc$	256
$\mathbf{u}_2 \leftarrow \text{Decompress}(c_1), \hat{\mathbf{u}}_2 \leftarrow \text{NTT}(\mathbf{u}_2)$	576
$\hat{us} \leftarrow \hat{\mathbf{u}}_2 \cdot \hat{\mathbf{s}}_2 + acc$	256
$us \leftarrow \text{INTT}(\hat{us})$	448
$v \leftarrow \text{Decompress}(c_2)$	128
$m' \leftarrow \text{Encode}(\text{Compress}(v - us))$	128
$\{\overline{K}, r'\} \leftarrow \text{G}(m'\|\|h)$	79
$\mathbf{r}_0 \leftarrow \text{PRF}(\cdot)$	79
$\hat{\mathbf{r}}_0 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{r}_0)), A^T_{00} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{r}_1 \leftarrow \text{PRF}(\cdot)$	512
$\hat{\mathbf{r}}_1 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{r}_1)), A^T_{01} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{r}_2 \leftarrow \text{PRF}(\cdot)$	512
$\hat{\mathbf{r}}_2 \leftarrow \text{NTT}(\text{CBD}_{\eta_1}(\mathbf{r}_2))$	512
$acc \leftarrow A^T_{00}\hat{\mathbf{r}}_0 + 0, A^T_{02} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$acc \leftarrow A^T_{01}\hat{\mathbf{r}}_1 + acc, A^T_{02} \leftarrow \text{Parse}(\text{XOF}(\cdot)), A^T_{10} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$\hat{\mathbf{u}}_0 \leftarrow A^T_{02}\hat{\mathbf{r}}_2 + acc, A^T_{10} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{e}'_0 \leftarrow \text{PRF}(\cdot)$	256
$\mathbf{u}_0 \leftarrow \text{INTT}(\hat{\mathbf{u}}_0) + \mathbf{e}'_0, c'_1 = \text{Encode}(\text{Compress}(\mathbf{u}_0)), A^T_{11} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	576
$acc \leftarrow A^T_{10}\hat{\mathbf{r}}_0 + 0, A^T_{12} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$acc \leftarrow A^T_{11}\hat{\mathbf{r}}_1 + acc, A^T_{12} \leftarrow \text{Parse}(\text{XOF}(\cdot)), A^T_{20} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$\hat{\mathbf{u}}_1 \leftarrow A^T_{12}\hat{\mathbf{r}}_2 + acc, A^T_{20} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{e}'_1 \leftarrow \text{PRF}(\cdot)$	256
$\mathbf{u}_1 \leftarrow \text{INTT}(\hat{\mathbf{u}}_1) + \mathbf{e}'_1, c'_1 = \text{Encode}(\text{Compress}(\mathbf{u}_1)), A^T_{21} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	576
$acc \leftarrow A^T_{20}\hat{\mathbf{r}}_0 + 0, A^T_{22} \leftarrow \text{Parse}(\text{XOF}(\cdot))$	256
$acc \leftarrow A^T_{21}\hat{\mathbf{r}}_1 + acc, A^T_{22} \leftarrow \text{Parse}(\text{XOF}(\cdot)), \mathbf{e}'_2 \leftarrow \text{PRF}(\cdot)$	256
$\hat{\mathbf{u}}_2 \leftarrow A^T_{22}\hat{\mathbf{r}}_2 + acc$	256
$\mathbf{u}_2 \leftarrow \text{INTT}(\hat{\mathbf{u}}_2) + \mathbf{e}'_2, c'_1 = \text{Encode}(\text{Compress}(\mathbf{u}_2)), e'' \leftarrow \text{PRF}(\cdot)$	576
$acc \leftarrow \hat{\mathbf{t}}_0 \cdot \hat{\mathbf{r}}_0 + 0$	256
$acc \leftarrow \hat{\mathbf{t}}_1 \cdot \hat{\mathbf{r}}_1 + acc$	256

Table 8: Detailed procedures in server side and corresponding cycle counts in implementing Kyber.CCAKEM protocol when k=3 (continued).

operation	cycles
$\hat{tr} \leftarrow \hat{\mathbf{t}}_2 \cdot \hat{\mathbf{r}}_2 + acc$	256
$v \leftarrow \text{INTT}(\hat{tr}) + e'' + \text{Decompress}(\text{Decode}(m))$	448
$c_2' = \text{Encode}(\text{Compress}(v), \text{Compare}(c_2, c_2'))$	128
$K \leftarrow \text{KDF}(\overline{K}'\|\|\text{H}(c))$ *or* $\text{KDF}(z\|\|\text{H}(c))$	79

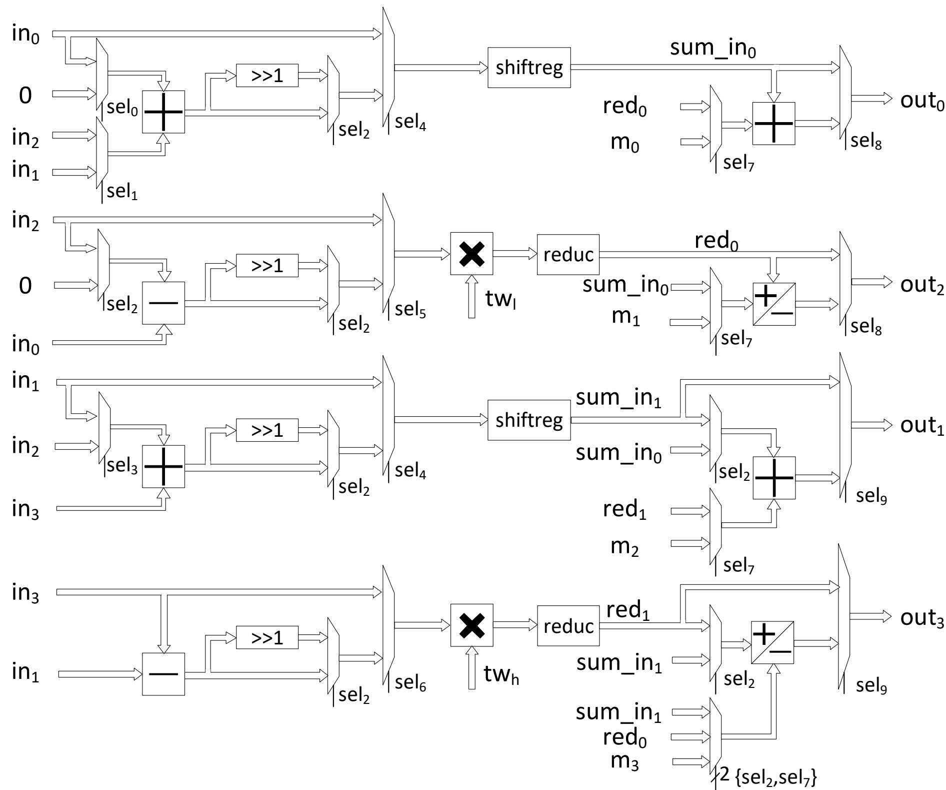

Figure 10: Structure of two sets of butterfly units. Each path is illustrated separately, with two input data pairs (in_1, in_0), (in_3, in_2) and corresponding output pairs (out_1, out_0), (out_3, out_2). Control codes in different operations are presented in Table 1.

Algorithm 8 KYBER.CCAKEM.KeyGen()

Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') :=$ KYBER.CPAPKE.KeyGen ()
3: $sk := (sk' \| pk \| \mathrm{H}(pk) \| z)$
4: **return** (pk, sk)

Algorithm 9 KYBER.CCAKEM.Enc (pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$
1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \mathrm{H}(m)$ ▷ Do not send output of system RNG
3: $(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
4: $c :=$ KYBER.CPAPKE.Enc (pk, m, r)
5: $K := \mathrm{KDF}(\bar{K} \| \mathrm{H}(c))$
6: **return** (c, K)

Algorithm 10 KYBER.CCAKEM.Dec (c, sk)

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
Output: Shared key $K \in \mathcal{B}^*$
1: $pk := sk + 12 \cdot k \cdot n/8$
2: $h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
3: $z := sk + 24 \cdot k \cdot n/8 + 64$
4: $m' :=$ KYBER.CPAPKE.Dec $(\mathbf{s}, (\mathbf{u}, v))$
5: $\left(\bar{K}', r'\right) := \mathrm{G}\left(m' \| h\right)$
6: $c' :=$ KYBER.CPAPKE.Enc (pk, m', r')
7: **if** $c = c'$ **then**
8: $\quad K := \mathrm{KDF}\left(\bar{K}' \| \mathrm{H}(c)\right)$
9: **else**
10: $\quad K := \mathrm{KDF}\left(z \| \mathrm{H}(c)\right)$
11: **end if**