Implementing and Benchmarking Seven Round 2 Lattice-Based Key Encapsulation Mechanisms Using a Software/Hardware Codesign Approach

Farnoud Farahmand[1], Viet Ba Dang[1], Michal Andrzejczak[2] and Kris Gaj[1]

[1] Cryptographic Engineering Research Group,
George Mason University
Fairfax, VA, U.S.A.
[2] Military University of Technology
Warsaw, Poland

Abstract. In this paper we present the results of implementing and benchmarking seven lattice-based key encapsulation mechanisms (KEMs), representing five NIST PQC Round 2 PQC candidates, using a software/hardware codesign approach. This approach is particularly applicable to the current stage of the NIST PQC standardization process, where the large number and high complexity of the candidate algorithms makes traditional hardware benchmarking extremely challenging. We propose and justify the choice of a suitable platform and design methodology. The results obtained indicate the potential for very substantial speed-ups vs. purely software implementations, reaching 396x for encapsulation and 712x for decapsulation. At the same time these speed-ups depend strongly on the features of each particular algorithm, which leads to noticeable changes in the ranking of evaluated candidates using software/hardware vs. purely-software benchmarking.

Keywords: Post-Quantum Cryptography · software/hardware codesign · lattice-based · hardware accelerator · System on Chip · programmable logic

1 Introduction

Hardware benchmarking has played a major role in all recent cryptographic standardization efforts, such as the AES, eSTREAM, SHA-3, and CAESAR contests. As with the current NIST Post-Quantum Cryptography (PQC) standardization effort, the number of candidates was reduced after each round of public evaluation. With the emergence of commonly-accepted hardware application programming interfaces (APIs) [20], development packages [19, 21], specialized optimization tools [13, 8], new design methodologies based on High-Level Synthesis (HLS) [17, 18], and mandatory hardware implementations in the final round of the CAESAR contest [5], the percentage of initial submissions with hardware implementations grew from 27.5% in the SHA-3 contest [12] to 49.1% in the CAESAR competition [6, 11].

Unfortunately this trend is not likely to be sustained in the NIST PQC standardization process by simply following prior practices and hardware benchmarking approaches. In many respects PQC schemes are diametrically different from those evaluated in previous cryptographic contests, and new challenges call for new substantially different solutions [11, 7].

Traditionally software and hardware benchmarking were conducted separately, by different groups of experts, equipped with different knowledge and tools. Even the units for measuring speed were different - cycles per byte for software, and megabits per second for hardware. For PQC algorithms this approach is hard to maintain. These algorithms are simply too complex and too different from the current state-of-the-art in public-key cryptography to permit the development of optimized purely hardware implementations for a significant fraction of the remaining candidates by any single group within the time frame imposed by the NIST evaluation process (12-18 months in case of Round 2).

At the same time there is little (if any) consensus regarding basic design choices such as hardware API, optimization target, or hardware platform (e.g., a single FPGA family or a single ASIC standard cell library).

NIST has not indicated that a hardware implementation will be required for each submission to the next round of the process.

In the 16 months since the start of the NIST PQC process only a few purely hardware implementations of Round 1 candidates have been announced: [29], [39], [22], [25], [9], and even fewer have been made open source. These implementations use different APIs, target different platforms, and are aimed at different optimization targets from high-speed to low-area. No conclusions regarding ranking of these algorithms in terms of their performance in hardware can be reached based on such divergent efforts.

In this paper we present an alternative approach to evaluating candidates in cryptographic contests, based on software/hardware codesign. This technique has been used for years in industry and studied extensively in academia, with the goal of reaching performance targets using a shorter development cycle than is typical for hardware-only implementations. To the best of our knowledge no benchmarking of software/hardware designs was reported during any previous cryptographic competitions. As a result multiple problems specific to cryptographic contests, such as the choice of the most representative platform(s) and the fairness of software/hardware partitioning schemes, have never been addressed.

It should be clearly stated that software/hardware benchmarking is not intended as a replacement for purely-hardware benchmarking. On the contrary, applying this approach for the 26 candidates advanced to Round 2, and developing a library of hardware accelerators for major operations of these candidates, will make it much easier to develop hardware-only implementations in subsequent rounds.

Within the proposed framework the first issue to address is the choice of the representative device. In particular we need a computing platform allowing fast communication across the software/hardware boundary. We also need reconfigurable hardware, as the timing measurements must be performed experimentally, and the platform must be well-suited for attempting various software/hardware partitioning schemes.

In recent years several such platforms have emerged. The most popular in industry are those based on integrating an ARM-based processor and FPGA fabric on a single chip. Examples include Xilinx Zynq 7000 System on Chip (SoC), Xilinx Zynq UltraScale+ MPSoC, Intel Arria 10 SoC FPGAs, and Intel Stratix 10 SoC FPGAs. These devices support software/hardware codesigns based on a traditional high-level language program running on an ARM processor, with the most time-critical computations performed on a dedicated hardware accelerator. The advantages of these platforms include: the use of the most popular embedded processor family (ARM) operating at high speed (1 GHz or above), state-of-the-art commercial tools (available for free, or at a reduced price for academic use), availability of inexpensive prototyping boards, and practical deployment in multiple environments.

The primary alternatives are FPGA-based systems with so-called "soft" processor cores implemented in reconfigurable logic. Examples include Xilinx MicroBlaze, Intel Nios II, and the open-source RISC-V, originally developed at the University of California, Berkeley [30, 40, 41]. The main advantage of these systems over "hard" processor cores is flexibility in the allocation of resources to processor cores, including the possibility of extending them with special instructions specific to PQC. Additionally they are easy to port between different FPGA families, and even between FPGAs and ASICs. A disadvantage compared to the "hard" option is that the "soft" processors operate at much lower clock frequencies (typically 200-450 MHz).

During a presentation at PQCrypto 2019, NIST asked designers to focus on the ARM Cortex-M4 for embedded software implementations and the Artix-7 for FPGA implementations. However we are not aware of any all programmable SoC device that contains a Cortex-M processor and the Artix-7 FPGA fabric on a single chip. Even if such a chip existed it would be more suitable for benchmarking of lightweight implementations (optimized for minimal cost and power consumption), rather than benchmarking of the high-speed implementations targeted by our study.

As a result we have based our choice of platform primarily on the projected practical importance of various platforms during the initial period of deployment of new PQC standards, and the expected speed-up over purely software implementations. These priorities led us to choose devices from the "hard" processor class, with a hard-wired ARM processor, and among them the Zynq UltraScale+ family from Xilinx Inc., the vendor with the biggest market share in this device category. Zynq UltraScale+ and similar SoCs are likely to be used for practical deployments of PQC in the near future, wherever device speed and time-to-market are of primary concern. Implementations using these devices are more likely than implementations using only hardware.

However the use of soft-core processors, and in particular the free and open-source RISC-V, should be considered as a natural next step, especially in light of DARPA's recent selection of the RISC-V Instruction

Table 1: Features of selected NIST Round 2 PQC KEMs

Feature	FrodoKEM	Round5	Saber
Underlying problem	LWE : Learning With Errors	RLWR : Ring Learning With Rounding	Mod-LWR : Module Learning with Rounding
Element of a matrix or vector in	Z_q	Z_q	$\mathbb{Z}_q[x]/(x^n+1)$
Modulus q	Power of 2	Power of 2	Power of 2
Major parameters	n: matrix dimensions, B: number of bits encoded in each matrix entry, σ: standard deviation	n: degree of reduction polynomial, p, t: other moduli	n: degree of reduction polynomial, l: number of polynomials per vector, p, T: other moduli, μ: parameter of CBD
Hash-based functions	SHAKE	cSHAKE	SHAKE, SHA3-256, SHA3-512
Sampling	Integers are sampled from an approximation of a rounded continuous Gaussian distribution.	Integers from a uniform distribution are produced by a DRBG taking a random seed.	Integers are sampled from a centered binomial distribution (CBD).
Decryption failures	Yes	Yes	Yes
#Multiplications in Encapsulation	2 matrix-by-matrix	2 vector-by-vector	2 matrix-by-vector 1 vector-by-vector
#Multiplications in Decapsulation	3 matrix-by-matrix	3 vector-by-vector	1 matrix-by-vector 2 vector-by-vector

Set Architecture (ISA) for investigation within its cybersecurity-related programs [27].

With the preferred platform identified, our second major concern is the fairness of software/hardware benchmarking, especially in terms of deciding which operations within each evaluated scheme should be offloaded to hardware. In this paper we propose a comprehensive approach to address this issue, aimed at achieving the best possible trade-off between the speed-up compared to software and the required development time. This approach is described in detail in Section 4.

The proposed methodology was applied to the evaluation of seven IND-CCA-secure [3, 15] key encapsulation mechanisms (KEMs), belonging to the following five different Round 2 PQC submissions: FrodoKEM [34], Round5 [37], Saber [38], NTRU [36], and NTRU Prime [35].

2 Basic Features of Compared Algorithms

Basic features of FrodoKEM, Round5, and Saber are summarized in Table 1. These algorithms are based on the Learning with Errors (LWE), General Learning With Rounding (GLWR), and Module Learning with Rounding (Mod-LWR) problems, respectively. The implemented variant of Round5 relies specifically on the RLWR (Ring Learning With Rounding) variant of GLWR, and thus only features of this variant are discussed below. All three KEMs are based on underlying IND-CPA public key encryption schemes, converted to IND-CCA KEMs using very similar variants of the Fujisaki–Okamoto transform [10], [16].

In all three schemes the elementary operation is integer multiplications modulo a power of two (denoted as q). In FrodoKEM the most time-consuming operation is a matrix-by-matrix multiplication, where each component of a matrix is an element of Z_q. In Saber the most time-consuming operations are matrix-by-vector and vector-by-vector multiplications, where each element of a matrix or a vector is a polynomial with n coefficients in Z_q, and the multiplication of such polynomials is performed modulo the reduction polynomial x^n+1. In the implemented variant of Round5 the most time consuming operation is a vector-by-vector

Table 2: Features of NIST Round 2 NTRU-based PQC KEMs

Feature	NTRU-HPS	NTRU-HRSS	Streamlined NTRU Prime	NTRU LPRime
Underlying problem	Shortest Vector Problem	Shortest Vector Problem	Shortest Vector Problem	Shortest Vector Problem
Polynomial P	$x^n - 1$	$\Phi_n = (x^n - 1)/(x - 1)^{**}$	$x^n - x - 1$ irreducible in $Z_q[x]$	$x^n - x - 1$ irreducible in $Z_q[x]$
Degree n^*	prime	prime	prime	prime
Modulus q	power of 2 with $q/8 - 2 \leq 2n/3$	power of 2 with $q > 8\sqrt{2}(n + 1)$	prime	prime
Weight w	Fixed weight for f and r	N/A	Fixed weight for f and r. $3w \leq 2n$ $16w + 1 \leq q$	Fixed weight for b and a. $3w \leq 2n$ $16w + 2\delta + 3 \leq q$
Hash-based functions	SHA3-256	SHA3-256	SHA3-512	SHA3-512
Decryption failures	No	No	No	No
Quotient rings	R/q: $Z_q[x]/(x^n - 1)$ S/q: $Z_q[x]/(\Phi_n)^{**}$ S/3: $Z_3[x]/(\Phi_n)^{**}$	R/q: $Z_q[x]/(x^n - 1)$ S/3: $Z_3[x](x - 1)/(x^n - 1)$	R/q: $Z_q[x]/(x^n - x - 1)$ R/3: $Z_3[x]/(x^n - x - 1)$	R/q: $Z_q[x]/(x^n - x - 1)$ R/3: $Z_3[x]/(x^n - x - 1)$
#Poly Mults for Encapsulation	1 in R/q	1 in R/q	1 in R/q	2 in R/q
#Poly Mults for Decapsulation	1 in R/q 1 in S/q 1 in S/3	1 in R/q 1 in S/q 1 in S/3	2 in R/q 1 in R/3	3 in R/q

* denoted by p in the specification of Streamlined NTRU Prime and NTRU LPRime
** $\Phi_n = (x^n - 1)/(x - 1)$ irreducible in $Z_q[x]$

multiplication, where components of one vector are elements of Z_q, and the components of the other vector are in the set {-1, 0, 1}.

All three algorithms use SHAKE [26] or cSHAKE [24] as an auxiliary cryptographic operation. Saber uses SHA3-256 and SHA3-512 in addition to SHAKE. Sampling is the easiest to implement in Round5 (uniform distribution), followed by Saber (centered binomial distribution), and then FrodoKEM (approximation of a rounded continuous Gaussian distribution).

Basic features of the four NTRU-based KEMs submitted to the NIST PQC process (NTRU-HPS and NTRU-HRSS from the NTRU submission package, and Streamlined NTRU Prime and NTRU LPRime from the NTRU Prime submission package) are summarized in Table 2. In each of these algorithms the underlying security problem is the Shortest Vector Problem (SVP) in a lattice. The most time-consuming operation in each is a polynomial multiplication, where the degree of the reduction polynomial is a prime. For operations on the polynomial coefficients the modulus is a power of 2 for NTRU-HPS and NTRU-HRSS, and a prime for Streamlined NTRU Prime and NTRU LPRime. The modulus chosen for each NTRU Prime algorithm may potentially lead to a higher resistance against future attacks, but its effect on the maximum clock frequency and resource utilization is clearly negative.

Additionally, NTRU LPRime requires two polynomial multiplications per encapsulation vs. one for the other three algorithms listed in Table 2. For decapsulation, the exact types of multiplications vary, but the number of multiplications required is three for each algorithm.

Parameter sets of seven investigated algorithms are summarized in Table 3. Because we compared IND-CCA KEMs [15], the parameter sets for Round5 were adopted from the IND-CCA PKE variant, rather than from the IND-CPA KEM. The submission package of Round5 does not contain the recommended parameter values for the IND-CCA KEM as this scheme is treated only as a building block of the IND-CCA PKE.

The specification of NTRU associates two different security categories with each parameter set for NTRU-HPS and NTRU-HRSS. In this paper we conservatively assumed the lower security category based on the so called non-local computational models (see [36], Section 5.3 Security Categories). The same computation model is implicitly assumed by the submitters of the other investigated algorithms.

In Table 3 we have divided parameter sets into three groups with security categories 1 and 2, 3 only, and 4 and 5, respectively. Only the first group contains all 7 investigated algorithms. However the second group contains the largest number of {algorithm, parameter set} pairs (6 out of 7) with exactly the same security level.

Table 3: Parameter sets of investigated algorithms

Algorithm	**Parameter Set**	**Security Category**	**Degree** n	**Modulus** q	**Other Major Parameters**	**Auxiliary Functions**
FrodoKEM	Frodo-640	1	640	2^{15}	B=2, $\sigma = 2.8$	SHAKE128
Round5	R5ND-1PKE_0d	1	586	2^{13}	$p = 2^9$, $t = 2^4$	cSHAKE128
Saber	LightSaber-KEM	1	256	2^{13}	$l = 2$, $T = 2^3$, $\mu = 10$	SHAKE128 SHA3-256 SHA3-512
NTRU-HPS	ntruhps2048677	1*	677	2^{11}	N/A	SHA3-256
NTRU-HRSS	ntruhrss701	1*	701	2^{13}	N/A	SHA3-256
Str NTRU Prime	kem/sntrup653	2	653	$4621 < 2^{13}$	$w = 288$	SHA3-512
NTRU LPRime	kem/ntrulpr653	2	653	$4621 < 2^{13}$	$w = 252$, $\delta = 289$	SHA3-512
FrodoKEM	Frodo-976	3	976	2^{16}	B=3, $\sigma = 2.3$	SHAKE256
Round5	R5ND-3PKE_0d	3	852	2^{12}	$p = 2^9$, $t = 2^5$	cSHAKE256
Saber	Saber-KEM	3	256	2^{13}	$l = 3$, $T = 2^4$, $\mu = 8$	SHAKE128 SHA3-256 SHA3-512
NTRU-HPS	ntruhps4096821	3*	821	2^{12}	N/A	SHA3-256
Str NTRU Prime	kem/sntrup761	3	761	$4591 < 2^{13}$	$w = 286$	SHA3-512
NTRU LPRime	kem/ntrulpr761	3	761	$4591 < 2^{13}$	$w = 250$, $\delta = 292$	SHA3-512
FrodoKEM	Frodo-1344	5	1344	2^{16}	B=4, $\sigma = 1.4$	SHAKE256
Round5	R5ND-5PKE_0d	5	1170	2^{13}	$p = 2^9$, $t = 2^5$	cSHAKE256
Saber	FireSaber-KEM	5	256	2^{13}	$l = 4$, $T = 2^6$, $\mu = 6$	SHAKE128 SHA3-256 SHA3-512
Str NTRU Prime	kem/sntrup857	4	857	$5167 < 2^{13}$	$w = 322$	SHA3-512
NTRU LPRime	kem/ntrulpr857	4	857	$5167 < 2^{13}$	$w = 281$, $\delta = 329$	SHA3-512

* assuming non-local computational models

3 Previous Work

Only a few candidates in the NIST PQC standardization process have been fully implemented in hardware to date. These implementations are reported in [29], [39], [22], [25], [9].

Only a few attempts to accelerate software implementations of post-quantum cryptosystems have been made through software/hardware (SW/HW) codesign by other groups. A coprocessor consisting of the

PicoBlaze soft-core and several parallel acceleration units for the code-based McEliece cryptosystem was implemented on Spartan-3AN FPGAs by Ghosh et al. [14]. No speed-up vs. purely software implementation using PicoBlaze was reported.

In 2015 Aysu et al. [2] built a high-speed implementation of a lattice-based digital signature scheme using SW/HW codesign techniques. The work focused on the acceleration of signature generation. The design targeted the Cyclone IV FPGA family and consisted of the NIOS II soft processor, a hash unit, and a polynomial multiplier. Compared to the C implementation running on the NIOS II processor, the most efficient software/hardware codesign reported in the paper achieved a speed-up of 26,250x at the expense of an increase in the number of Logic Elements by a factor of 20.

Wang et al. [39] reported a software/hardware implementation of the hash-based digital signature scheme XMSS. The selected platform was an Intel Cyclone V SoC, and the software part of the design was implemented using a RISC-V soft-core processor. Hardware accelerators supported a general-purpose SHA-256 hash function, as well as several XMSS specific operations. The design achieved a speed-up of 23x for signing and 18x for verification over a purely software implementation running on RISC-V.

All the aforementioned platforms were substantially different than the platform used in this work. The algorithms and their parameters were also substantially different. As a result, limited information could be inferred regarding the optimal software/hardware partitioning, expected speed-up, or expected communication overhead.

An earlier version of this work, representing three NIST PQC Round 1 candidates (NTRUEncrypt, NTRU-HRSS, and NTRU Prime) was reported in [7]. Compared to that work, all previously reported designs were updated to make them compatible with the Round 2 specifications of NTRU and NTRU Prime [36], [35]. It should be mentioned that although the NTRUEncrypt and NTRU-HRSS candidates merged, the obtained Round 2 candidate, NTRU, has two distinct variants, NTRU-HPS (somewhat similar to the Round 1 NTRUEncrypt) and NTRU-HRSS (somewhat similar to the Round 1 candidate with the same name), and thus the total number of the NTRU-based KEMs did not change. Our designs for FrodoKEM, Round5, and Saber have not been reported in any earlier paper. Other differences compared to [7] include reporting results for multiple parameter sets per algorithm, more complete exploration of the available software/hardware partitioning schemes, minimization of the software/hardware transfer overhead, and measuring separately the execution time of the function `randombytes()` used to obtain uniformly distributed random bytes during encapsulation.

4 Methodology

4.1 Software/Hardware Codesign Platform

The platform used in this work was selected based on the following criteria:

- modern technology, representing the current state of the art (vs. older generations of FPGAs, such as Xilinx Virtex-6 or Virtex-7, used in the majority of previous cryptographic competitions)
- reconfigurable logic large enough to demonstrate the full capability for parallelization in hardware of PQC algorithms
- a fast processor, representing the majority of the embedded system market, such as a variant of ARM
- a fast on-chip interface between the Processing System (based on a microprocessor) and Programmable Logic (based on reconfigurable fabric), such as the ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) version 4, the de facto standard for today's embedded processor bus architectures [1]
- relatively low cost and wide availability of a prototyping board containing the selected device, supporting practical experimental measurements by multiple groups
- a device with relatively large share of the market for embedded system applications, especially in the area of communications.

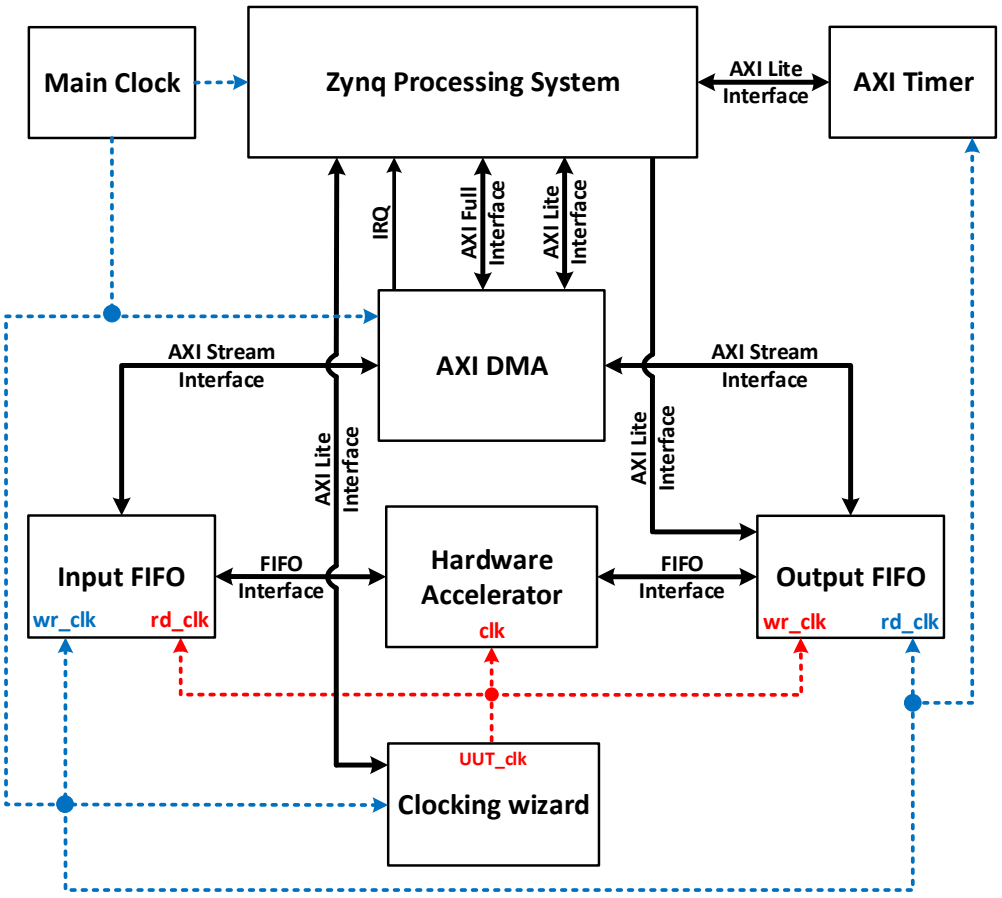


Figure 1: Block diagram of software/hardware codesign.

Based on these criteria we chose the Xilinx Zynq UltraScale+ MPSoC XCZU9EG-2FFVB1156E as our target device, and the Xilinx ZCU102 Evaluation Kit as a prototyping board. The device is composed of two major parts sharing the same chip. The primary component of the Processing System (PS) is a quad-core ARM Cortex-A53 Application Processing Unit, running at 1.2 GHz. As in the software benchmarking experiments conducted by other groups, we utilize only one core in all our experiments. The Programmable Logic (PL) includes a programmable FPGA fabric similar to that of Virtex UltraScale+ FPGAs, including Configurable Logic Block (CLB) slices, Block RAMs, DSP units, etc. The frequency of operation depends on the particular logic instantiated in the reconfigurable fabric, but typically does not exceed 400 MHz.

The software used is Xilinx Vivado Design Suite HLx Edition, Xilinx Software Development Kit (XSDK), and Xilinx Vivado HLS, all with version number 2018.2.

A high-level block diagram of the experimental software/hardware codesign platform is shown in Fig. 1. The Hardware Accelerator is connected, through the dual-clock Input and Output FIFOs, to the AXI DMA, supporting the high-speed communication with the Processing System. Timing measurements are performed using the popular Xilinx IP unit called AXI Timer, which is capable of measuring time in clock cycles of the 200 MHz system clock. The Hardware Accelerator can operate at a variable clock frequency, controlled from software using the Clocking wizard unit.

4.2 Software Profiling, C Source Code Analysis, and Software/Hardware Partitioning

Our first step in evaluating the suitability of cryptographic algorithms for software/hardware codesign was profiling of their software implementations using one core of the ARM Cortex-A53. Profiling produces a list of the most-time consuming functions, including their absolute execution time, percentage execution time, and the number of times they are called.

In the case of KEMs, the encapsulation operation uses multiple calls to the function `randombytes()` which produces a sequence of random bytes with uniform distribution. Other PQC benchmarking projects use a version of this function based on operating system functions and/or functions from OpenSSL [4, 32, 23, 33]. None of these options is available in our study, in which we perform benchmarking in the Bare Metal mode. Therefore in our code we use the implementation of `randombytes()` proposed by Saarinen in April 2018 [32], which is an improved version of the implementation developed by NIST for the generation of known-answer tests [28]. Since both of these implementations rely on the implementation of AES in the ECB mode from the OpenSSL library, we have replaced this implementation by the standalone, optimized implementation of AES in C, based on the use of T-boxes [31]. Compared to the OpenSSL implementation the selected implementation is written entirely in C rather than in an assembly language of a specific processor, and does not contain any countermeasures against cache-timing attacks. As a result, the selected implementation of `randombytes()` is likely to have different timing characteristics than the implementations used in other benchmarking studies, such as SUPERCOP [4], pqcbench [32], pqm4 [23], and liboqs [33]. Therefore for each encapsulation operation we measure the execution time including and excluding the execution time of `randombytes()`. Additionally, we report the total number of calls to `randombytes()`, as well as the total number of bytes generated using this function. This approach allows us to determine whether ranking of candidates may be possibly modified by the use of a different implementation of `randombytes()`, and/or by replacing calls to this function by calls to a different cryptographically-strong pseudorandom function, such as SHAKE.

We decided which functions to offload to hardware based on the highest potential for total speed-up, as well as fairness of comparison among investigated algorithms. The total speed-up obtained by offloading an operation to hardware depends on two major factors: the percentage of the execution time taken in software by the operation offloaded to hardware, and the speed-up for the offloaded operation itself (which we will call the "accelerator speed-up"). In order to maximize the first factor we gave priority to operations that take the largest percentage of the execution time, preferably more than 90%. These operations may involve a single function call, several adjacent function calls, or a sequence of consecutive instructions in C. It is preferred that a given operation is executed only once, or only a few times, as each transfer of control and data between software and hardware involves a certain fixed timing overhead, independent of the size of input and output to the accelerator. In order to maximize the second factor we gave priority to operations that have high potential for parallelization in hardware, and small total size of inputs and outputs (which will need to be transferred to and from the hardware accelerator, respectively)

Most of the data required to make informed decisions regarding software/hardware partitioning can be obtained by profiling the purely software implementation, possibly extended with some small modifications required to gather all relevant data. However, determining the potential for parallelization requires some knowledge of hardware or at least basic concepts of concurrent computing.

In order to assure fairness in our comparison, we endeavored to offload to hardware all operations common to or similar across the iplementat algorithms (e.g. all polynomial multiplications), and all operations that contribution significantly to the total execution time. Nevertheless it should be understood that this heuristic procedure may need to be repeated several times, because after the each round of offloading to hardware different software operations may emerge as taking the majority of the total execution time. This process can stop when the development effort required for offloading the next most-critical operation to hardware is disproportionately high compared to the projected speed-up.

4.3 Interface of Hardware Accelerators and the RTL Design Methodology

The interface of a hardware accelerator matches the interface of the Input and Output FIFOs. The default width of the data bus is 64 bits. Each particular operation, such as load public key, start encapsulation, etc., is initiated by sending an appropriate header (in the form of a single 64-bit word), from a program running on the ARM processor to the data input of a hardware accelerator. When an operation requires additional data, this data is transmitted using the subsequent Input FIFO words.

After the hardware accelerator produces results or detects an error, a header word is sent in the opposite direction. If an additional output is required, this output follows the header and is arranged in 64-bit words. The detailed format of the exchanged inputs and outputs is left up to the designer of a hardware accelerator.

The design of a hardware accelerator follows a traditional Register-Transfer Level (RTL) methodology.

The entire system is divided into a Datapath and a Controller. The Datapath is described using a hierarchical block diagram, and the Controller using hierarchical algorithmic state machine (ASM) charts. Multiple local controllers may be advantageous compared to a single global Controller. The RTL approach, although not novel by itself, is an important part of our methodology as it facilitates very efficient hardware accelerator designs. The block diagrams and ASM charts are very easy to translate to efficient and fully synthesizable VHDL code.

4.4 Verification and Generation of Results

Functional verification of the hardware description language (HDL) code is performed by comparing simulation results with precomputed outputs generated by a reference software implementation.

Fully verified and independently optimized VHDL code is then combined with the *optimized* software implementation of a given PQC candidate. Functional verification of the integrated software/hardware design is performed by running the code on the prototyping board and comparing the obtained outputs with outputs generated by a functionally equivalent reference implementation, run on the same ARM Cortex-A53 processor.

Experimental timing measurements follow, with the hardware accelerator's clock set (using the Clocking wizard) to the optimal target frequency identified during the synthesis and implementation runs. The execution time is measured by using the AXI Timer module, shown in Fig. 1, in clock cycles of the AXI Timer, which operates at the default clock frequency of 200 MHz.

5 Hardware Accelerators

5.1 FrodoKEM

The pseudocode of FrodoKEM, with parts offloaded to hardware surrounded by dotted rectangular frames, is shown in Fig. 9 in Appendix B. The top-level block diagram of the hardware accelerator is shown in Fig. 2.

The public key is composed of the 128-bit *seed_A* and B – an unpacked public-key matrix, of dimensions n x 8 $log_2 q$-bit words, where n=640, 976, 1344 for the security levels 1, 3, 5, respectively. Both of these elements are assumed to be loaded to the respective memories of the hardware accelerator, `Seed_Asm_Mem` and `Matrix_A_and_B_Dual_Mem` before the encapsulation or decapsulation starts.

During the encapsulation, shown in Fig. 9, the 256-bit *seed_SE* is first loaded to the asymmetric memory `Seed_Asm_Mem`, with the 8-bit data input and the 64-bit data output. SHAKE128 is run to generate a pseudorandom sequence $r^{(0)}..r^{(\overline{m}n-1)}$. This sequence is then fed to `Sampler`, which for every 16-bit word produces a w-bit output. The obtained samples, representing subsequent coefficients of the vector S', are stored in the asymmetric memory `Matrix_S'_Asym_Mem`. The internal block diagram of the Frodo Sampler is shown in Fig. 3.

The subsequent words generated by SHAKE128, denoted in the pseudocode as $r^{(\overline{m}n)}..\ r^{(2\overline{m}n-1)}$, are passed through `Sampler`, and its outputs are stored as subsequent coefficients of E', in the memory `Matrix_B'_and_V_Dual_Mems`. Similarly, the words $r^{(\overline{m}n)}..r^{(2\overline{m}n-1)}$ are passed through the sampler, and used to generate coefficients of E'', stored in the same memory.

Subsequently, SHAKE128 is used to generate elements of the n x n matrix A, with each element expressed using $log_2 q$ bits. In order to reduce the execution time and the size of the `Matrix_A_and_B_Dual_Mem` memory, only one row of the A matrix is generated at a time, and used for the computations of $B' = S'A + E'$, in parallel with calculating the subsequent row of A. The elements of A are multiplied by the corresponding elements of S', read from `Matrix_S'_Asym_Mem`, sign-extended to $log_2 q$ bits, and stored in one of the eight registers preceding the 4MAC units.

The internal block diagram of the 4MAC unit, processing 4 elements of A, S', and E' at a time is shown in Fig. 13 in Appendix B. The temporary results are stored back in `Matrix_B'_and_V_Dual_Mem`. B' is then transferred back to the processor using the `outfifo_data` bus. After the subsequent computation, $V = S'B + E''$, V is transferred to the processor for further computations in software.

The operations performed by the hardware accelerator during the decapsulation are identical to those performed during the encapsulation (with B' replaced by B''). The operation $M = C - B'S$ is not offloaded to hardware. This operation takes a very small percentage of the total execution time in the purely software implementation. It also requires a significant amount of data to be transferred to and from the hardware

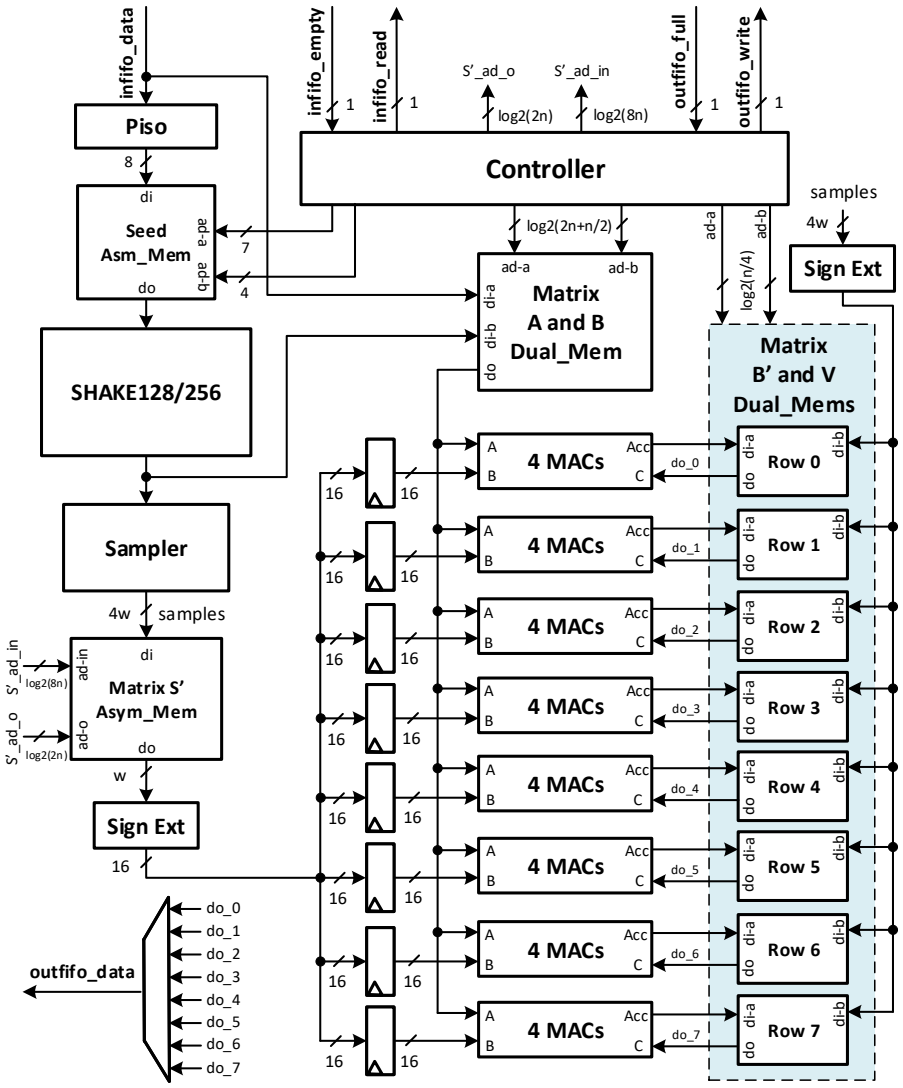


Figure 2: Block diagram of the hardware accelerator for FrodoKEM. All bus widths are 64-bit unless specified.

accelerator. As a result, any attempt at a hardware acceleration of this operation has resulted in increasing rather than decreasing the total execution time.

5.2 Round5

The pseudocode of Round5, with parts offloaded to hardware surrounded by dotted rectangular frames, is shown in Fig. 10 in Appendix B. The main computations of Round5 are performed in the polynomial ring $\mathbb{Z}_q[x]/(\Phi_{n+1}(x))$. The most time consuming operation is multiplication in the aforementioned ring, described by the equation

$$c_k = \sum_{i+j \equiv k \mod n} a_i * b_j \mod q \tag{1}$$

This operation is executed twice during encapsulation and three times during decapsulation.

Thus, polynomial multiplication is the most obvious candidate for hardware acceleration. Moreover, a polynomial multiplication can be implemented more efficiently than in general case, due to the special form of one of the polynomials. In each Round5 multiplication, one of the polynomials is always a ternary polynomial, which means that each of its coefficients is from the set $\{-1, 0, 1\}$. In this case, the multiplication is reduced only to addition or subtraction of the coefficients of the second polynomial.

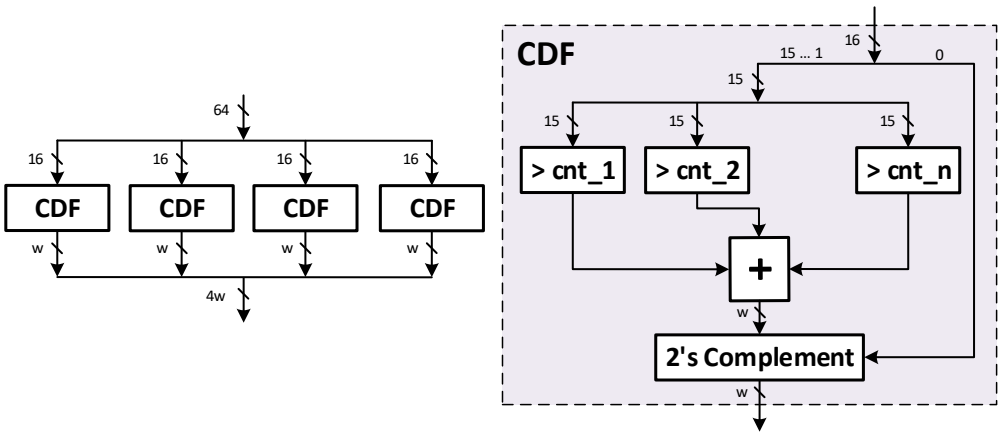


Figure 3: FrodoKEM sampler.

After initially, moving only polynomial multiplication to hardware, we have decided to accelerate the entire encryption and decryption. In this approach, cSHAKE is also implemented in hardware and used for the secret key and public key expansion. This approach allows generating the majority of polynomials used in multiplication directly in hardware, without the need of generating them in software and passing through the relatively slow communication channel. The inputs for encryption and decryption are directly passed to FPGA fabric without unpacking by CPU. The (un-)packing functions, based on bit-shifting operations are implemented in hardware. These operations are very inexpensive in hardware. Thus, the speed-up comes from both the faster execution of cSHAKE in hardware, as well as lower communication overhead, achieved by sending only the seed for cSHAKE instead of the expanded data. The remaining operations, such as rounding, addition, and subtraction are also fast and cost-efficient in hardware, providing additional speed-up. Thus, with little additional area, the design is able to execute encryption and decryption on the input data and return results in the already packed format.

We have decided to implement in hardware only r5_cpa_pke_encrypt, r5_cpa_pke_decrypt, and one of the additional calls to cSHAKE appearing during decapsulation (denoted as G(m'||pk in Fig. 10). A few remaining operations of Round5 CCA KEM are executed on the software side. Moving all operations to FPGA fabric would lead to a more complicated and area-consuming design. Moreover, the maximum clock frequency could decrease. Finally, the design with all operations executed in FPGA fabric would be a full hardware implementation and the comparison with other software/hardware codesigns described in this paper would not be any longer fair.

The top-level block diagram of *r5_cpa_pke* is shown in Fig. 4. The required data is being read from the input FIFO using the port `data_in`. The first data block must be a header block, which describes the command and the destination of the incoming transmission. Based on the header value, the main controller decides where the next data block should be written. The decision is sent to the `SIPO` module with selected input. If the incoming data is a seed for expansion, it is passed directly to the `cSHAKE` unit. In other cases, the specified input port of one of the two arithmetic modules is used.

The main controller is also responsible for managing the state of the accelerator. After all required data is received, including the expanded data generated by `cSHAKE`, the controller initializes the arithmetic modules and waits till the end of computations. The last step is to send the result back to software.

Encryption and decryption are performed by the arithmetic modules: `Rounding` and `Poly_Mul`, shaded with colors in Fig. 4. Provided with necessary data and operation type, the aforementioned modules execute specific instructions. At first, a polynomial multiplication is performed. Based on the operation type, temporary result can be then rounded. During encryption, the message is added at the end of the data flow, before the results are prepared to be sent back to software.

The majority of area taken by arithmetic modules is used by Poly Mult, shown in Fig. 5. The area requirements come from the construction of the multiplier. To achieve the best performance, we use n coefficient multipliers working in parallel. With this setting, the polynomial multiplication takes n clock cycles. A multiplication by a ternary coefficient is performed as an XOR and AND operation. We utilize the

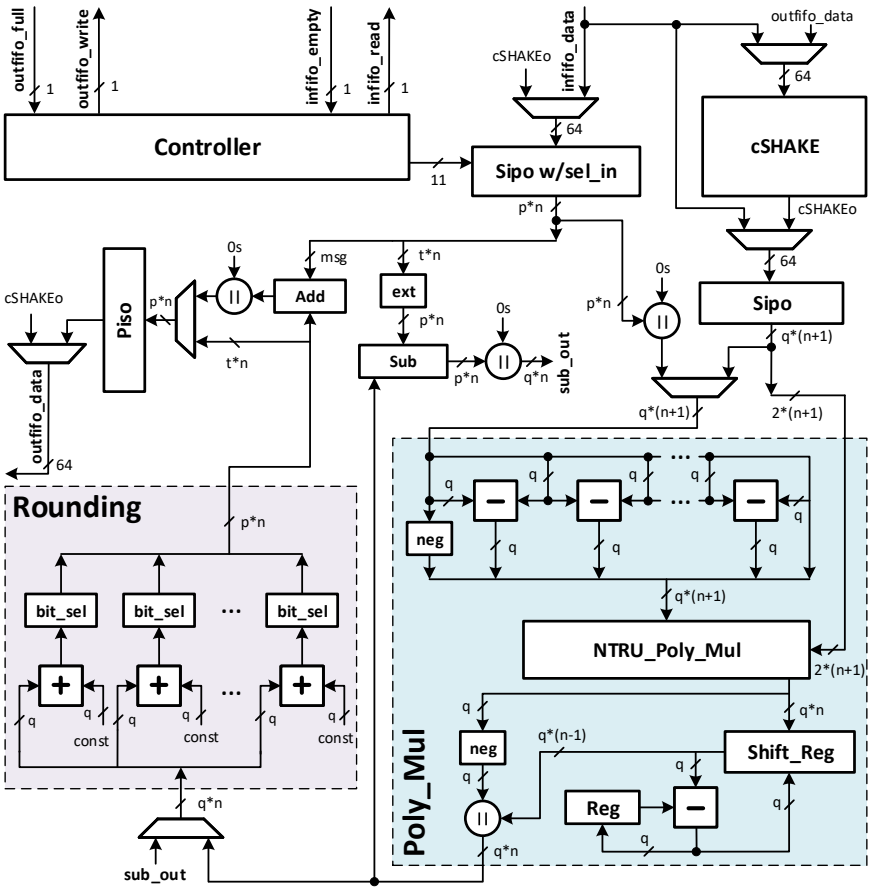


Figure 4: Block diagram of the hardware accelerator of Round5.

fact that one of the arguments is from the set $\{-1, 0, 1\}$. Thus, the second argument is XOR-ed bit-by-bit in parallel with the bit describing the sign of the first argument. Next, the parallel AND operation, with a bit denoting a zero value of the ternary coefficient is performed. The result is passed to an adder and next to an accumulator.

The `NTRU_Poly_Mul` is surrounded with additional logic performing necessary operations to prepare polynomials for multiplication. One of the polynomials is lifted from the ring

$$\mathbb{Z}_q[x]/(\Phi_{n+1}(x)) \tag{2}$$

to the ring

$$\mathbb{Z}_q[x]/(N_{n+1}(x)). \tag{3}$$

before multiplication. The coefficients in the lifted polynomial are equal to $-a_0+(a_0-a_1)x+(a_1-a_2)x^2+\cdots+a_{n-1}x^n$. Due to lack of data dependency, each coefficient is lifted in parallel and the operation takes always only one clock cycle. After multiplication, a polynomial is unlifted to previous ring. Unlifting is computed recursively as $a_i = a_{i-1} - a_i^L$. Unfortunately, this operation cannot be executed in parallel and has almost the same latency as multiplication. The second arithmetic module named `Rounding` is responsible for properly shrinking the bit size of coefficients by adding a rounding constant (specific to the given computational step) and applying a proper mask.

The presented design is able to encrypt and decrypt data by performing all necessary operations. This feature allows using the core directly in any Round5 scheme requiring the *r5_cpa_pke* encryption and decryption.

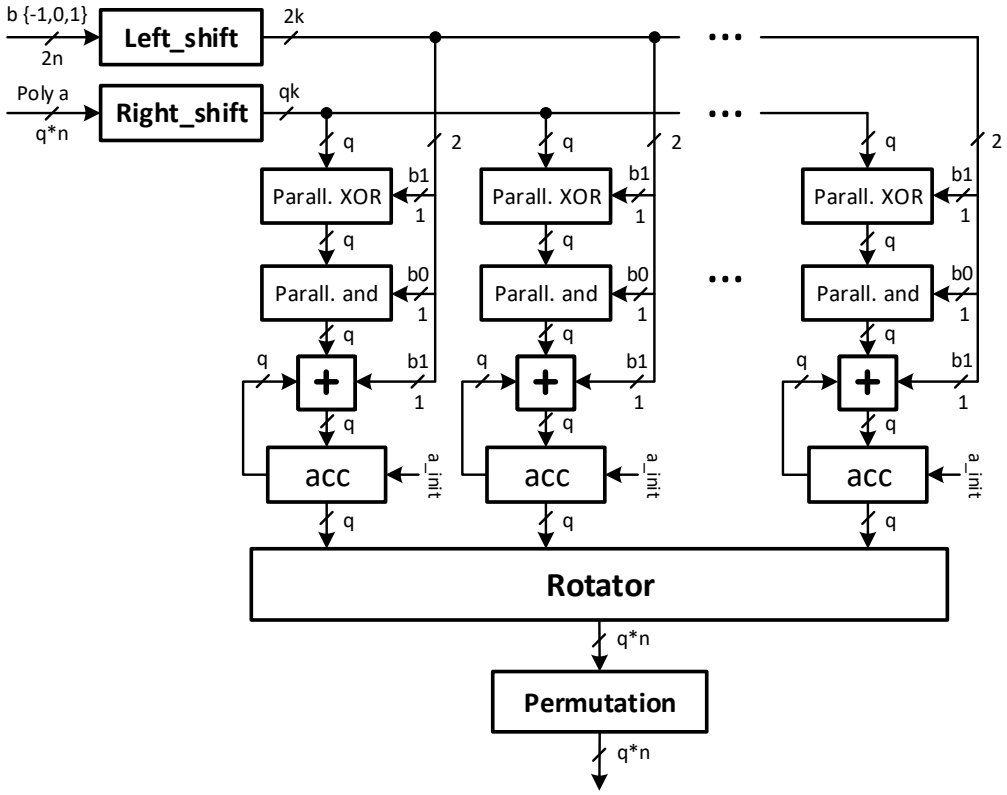


Figure 5: Block diagram of Round5 NTRU Poly Mult.

5.3 Saber

The pseudocode of Saber, with the operations offloaded to hardware surrounded by dotted rectangular frames, is shown in Fig. 11 in Appendix B. The top-level block diagram of the hardware accelerator is shown in Fig. 6.

The public key of Saber is composed of the 256-bit *seed_A* and the vector b composed of l polynomials with n=256 coefficients each (where l=2, 3, 4 for the security levels 1, 3, 5, respectively). The coefficients of polynomials are of the size of $log_2 q$=13 bits for all security levels. Both *seed_A* and b are assumed to be loaded to the respective memories of the hardware accelerator, `Seed_Mem` and `Vector_b_and_S_Asym_Mem`, using the 64-bit input bus `infifo_data`, before the encapsulation or decapsulation starts.

During the encapsulation, only the operations performed during Encryption Saber.PKE.Enc, shown in Fig. 11, are accelerated. Unlike in the pseudocode, in the hardware accelerator, vector s' is generated first. In order to make it possible, the 256-bit seed r is loaded first to `Seed_Mem`. The generation of s' involves `SHAKE128` followed by `Sampler`, generating w-bit integers using centered binomial distribution (CBD). The obtained samples, representing subsequent coefficients of the vector s', are stored in the asymmetric memory `Matrix_S'_Asym_Mem`.

Subsequently, `SHAKE128` is used to generate elements of the l x l matrix A, with each element representing a polynomial. In order to reduce the execution time and the size of `Matrix_A_Asym_Mem memory`, only one row of the A matrix is generated at a time, and used for the computations of $b' = (As' + h)\ mod\ q$, in parallel with calculating the subsequent row of A. h in the above equation is a constant. The elements of A are multiplied by the corresponding elements of s', read from `Matrix_S'_Asym_Mem`, sign-extended to 13-bits, and stored in the n-stage LFSR. With 4 coefficients loaded per clock cycle, the initialization of the 256-stage LFSR takes 64 clock cycles. The temporary results are stored in the registers shown to the right of `MACs` in Fig. 6. The internal structure of MACs is shown in Fig. 14 in Appendix B. Each coefficient of b' is then shifted right by 3 positions (corresponding to the division by q/p=$2^{13}/2^{10}$=8) and transferred back to the processor using the outfifo_data bus. In the subsequent operation, $v' = b^T(s' \bmod p)$, the reduction mod p is performed on the fly, and the result transferred to the processor for further computations in software.

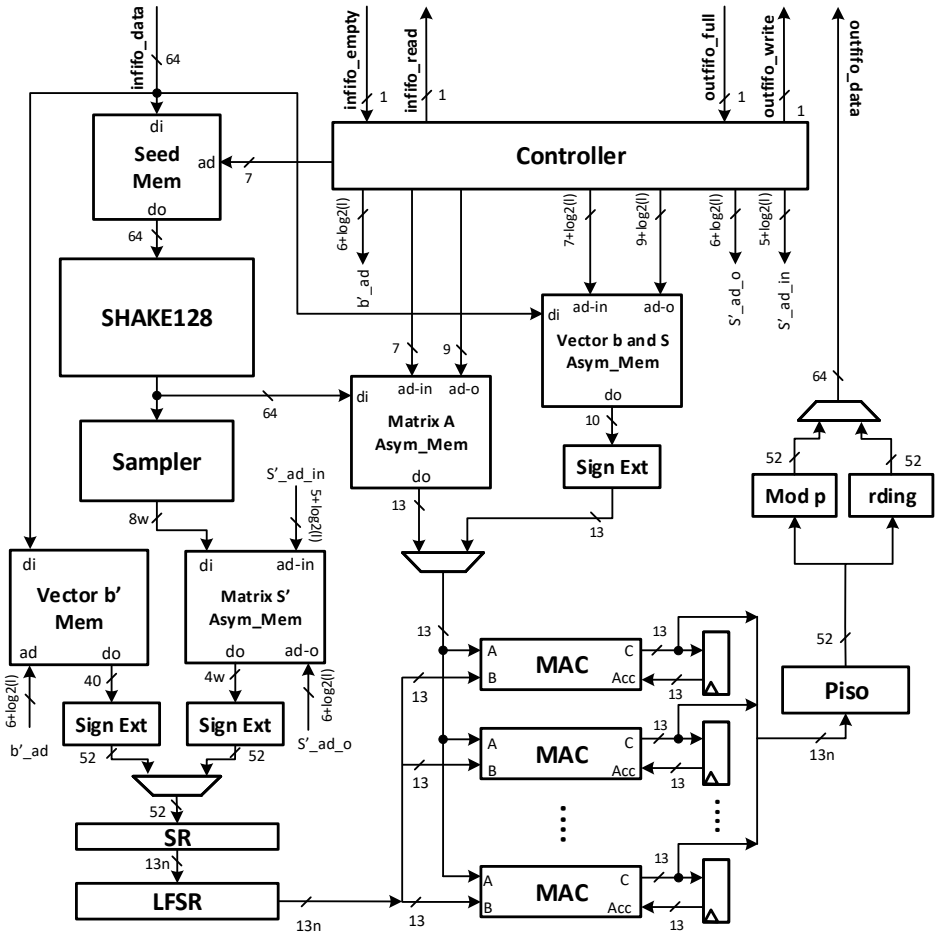


Figure 6: Block diagram of the hardware accelerator of Saber.

Secret key s is assumed to be loaded before the decapsulation starts. In the first phase of decapsulation, a new operation, $v = b'^T s \bmod p$, specific to decapsulation, is performed by the hardware accelerator. b' is a part of the ciphertext, and thus must be loaded already after the start of decapsulation. In the second phase of decapsulation, the function Saber.PKE.Enc is called, and as a result, the hardware accelerator performs exactly the same operations as during the encapsulation.

5.4 NTRU-HPS and NTRU-HRSS

The pseudocode of NTRU, with parts offloaded to hardware surrounded by dotted rectangular frames, is shown in Fig. 12 in Appendix B. The top-level block diagram of the hardware accelerator is shown in Fig. 7.

Polynomial multiplications mod $(q, \Phi_1\Phi_n)$, located in the lines 2 of Encrypt() and 2 of Decrypt() are executed using `Zq_LFSR` and `MACs` located in the top portion of the block diagram. The `Zq_LFSR` is initialized with a polynomial with large coefficients (**h** for Encrypt() and **c** for Decrypt()). Let us denote the initial state of the LFSR as $a(x)$. In each subsequent iteration, the output from LFSR contains the value $a(x) \cdot x^i \bmod P$. In a single clock cycle, a simple multiplication by x, namely $a(x) \cdot x^{i+1} \bmod P = a(x) \cdot x^i \cdot x \bmod P$, is performed, as shown in Fig. 17a. The coefficients at the output of `Zq_LFSR` are then multiplied by the sign-extended small coefficient of **r** for Encrypt() and **f** for Decrypt(), read from the `f_r_Asym_RAM`, and added to the partial sum sum_fb. The internal structure of MAC is shown in Fig. 14. For the multiplication mod (q, Φ_n), located in the line 5 of Decrypt(), c0 is set to the output of `hq_Asym_RAM`. The multiplication mod $(3, \Phi_n)$, located in line 3 of Decrypt(), is performed using `Z3_LFSR` and the adders mod 3 located in the right portion of the block diagram in Fig. 7. c0r, generated by the controller, based on the value of the currently processed coefficient of f_p, is used to select between adding or subtracting the output of

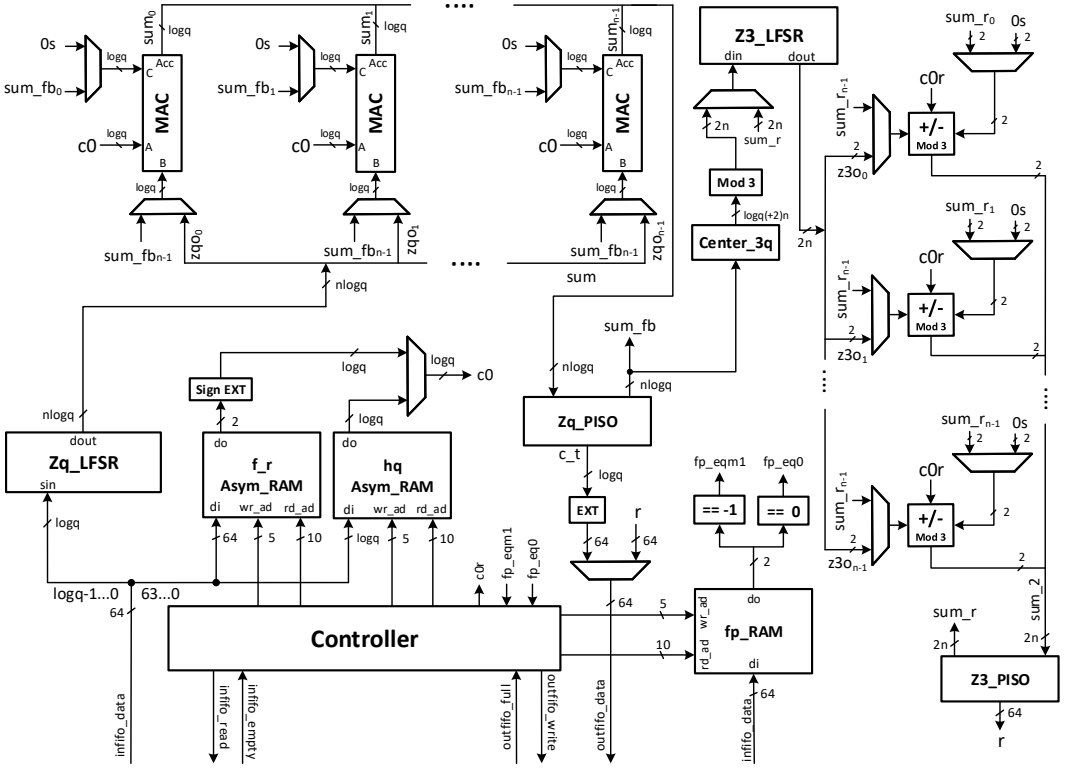


Figure 7: Block diagram of the hardware accelerator for NTRU.

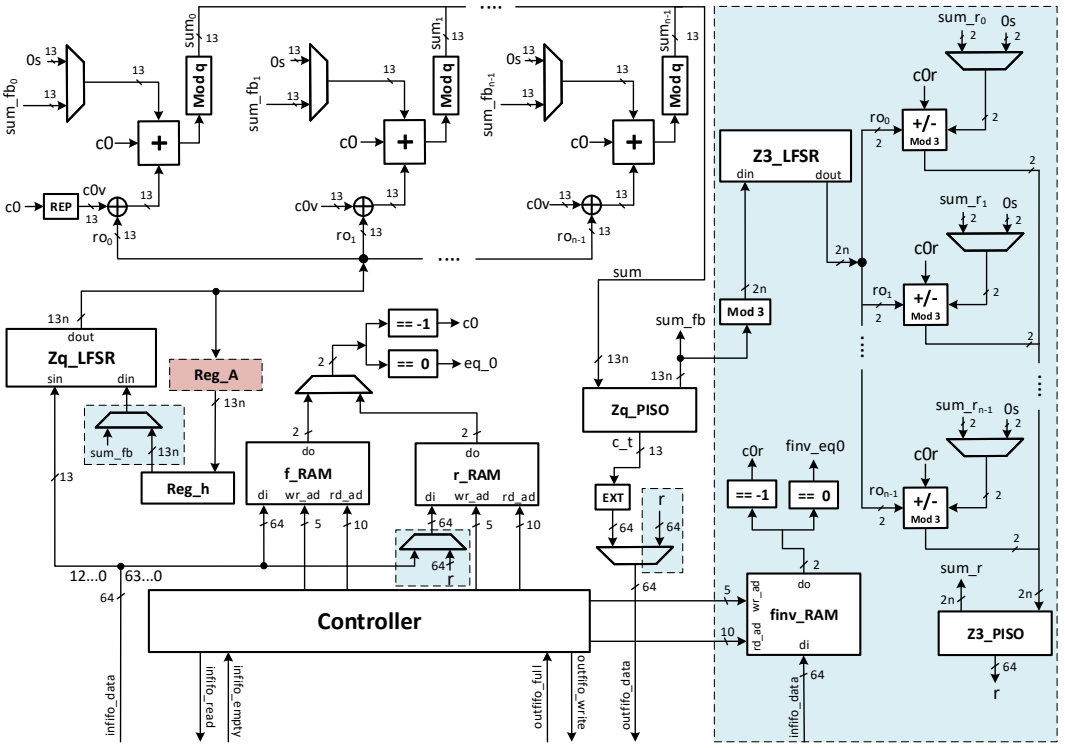


Figure 8: Block diagram of the hardware accelerator for NTRU LPRime and Streamlined NTRU Prime.

Z3_LFSR to/from the partial sum sum_r. The internal structure of the Z3_LFSR is shown in Fig. 17b. In case of multiplications in lines 3 and 5 of Decrypt(), one extra clock cycle is sufficient to convert the result of multiplication mod $\Phi_1\Phi_n{=}x^n-1$ to the result of multiplication mod Φ_n.

Coefficients of the private key **f** are preloaded to the asymmetric f_r_RAM, before the decryption starts. The partial and final results are stored in the Zq_PISO (Parallel-In Serial-Out) unit, with the parallel input of the width of $n \cdot logq$ bits, the parallel output of the same width (used to enable the accumulation of intermediate products), and the serial output of the width of $logq$ bits used to read out the final result to the output FIFO.

The Controller is responsible for generating suitable select and enable signals, communication with the Input and Output FIFOs, interpreting the input headers with instructions sent by the respective driver, and generating the output header containing the status and error codes that are sent back to the driver.

5.5 NTRU LPrime and Streamlined NTRU Prime

A block diagram of the hardware accelerators for Streamlined NTRU Prime and NTRU LPRime is shown in Fig. 8. The operations in R/3 are necessary only in case of Streamlined NTRU Prime and are similar to operations in S/3 for NTRU. Compared to NTRU, the main difference is the need for reduction of partial sums, involving large coefficients, mod q. Since now, q is a prime, a conditional subtraction is necessary. An additional register A is required for NTRU LPRime only, increasing the number of required flip-flops.

6 Results

The results of profiling for the purely software implementations, running on a single core of ARM Cortex-A53, at the frequency of 1.2 GHz, are presented in the left portions of Tables 7, 8, 9, 10, and 11 in Appendix A. For each of the seven investigated algorithms and each major operation (Encapsulation and Decapsulation), two to five most time-consuming functions are identified. For each of these functions, we provide their execution time in microseconds, and the percentage of the total execution time. In the right portions of the same tables, we list in bold functions offloaded to hardware. For the functions combined together, they are listed in the same field of the table, with sub-indices, such as 1.1, 1.2, 1.3, etc. A single execution time and a single percentage of the software/hardware execution time is given for such a combined function.

It should be mentioned that the number of functions offloaded to hardware may be misleading, as these functions may appear at different levels of hierarchy. For example, for the Round5 encapsulation, only one function is offloaded. However, it is a function involving the majority of operations of Round5, amounting to 99.6% of the total execution time in the software-only implementation. For the majority of algorithms, at least the first and the second most-time consuming functions are offloaded to hardware.

In Table 4, for each investigated KEM and each major operation (Encapsulation and Decapsulation), we list the total execution time in software (for the optimized software implementations in C running on ARM Cortex-A53 of Zynq UltraScale+ MPSoC), the total execution time in software and hardware (after offloading the most time consuming operations to hardware), and the obtained speed-up. The ARM processor runs at 1.2 GHz, DMA for the communication between the processor and the hardware accelerator at 200 MHz, and the hardware accelerators at the maximum frequencies, specific for the RTL implementations of each algorithm, listed in Table 6. All execution times were obtained through experimental measurements using the setup shown in Fig. 1. The speed up for the software part offloaded to hardware itself is given in the column Accel. Speed-up. This speed-up is a ratio of the execution time of the accelerated portion in software (column Accel. SW [ms]) and the execution time of the accelerated portion in hardware, including all overheads (column Accel. HW [ms]). The last column indicates how big percentage of the software-only execution time was taken by an accelerated portion of the program.

The time of Encapsulation is provided with and without the execution time of randombytes(). The reason for that is discussed in Section 4.2 and reinforced by measurements reported in Table 5. Optimized implementations included in the submission packages of FrodoKEM, Round5, Saber, and NTRU LPRime use randombytes() only to generate a 16, 24, or 32-byte seed for other pseudorandom functions, such as SHAKE. The implementations included in the submission packages of NTRU-HPS, NTRU-HRSS, and Streamlined NTRU Prime use randombytes() to generate significantly longer strings of bytes.

Table 4: Timing results.

Algorithm	Parameter Set	Total SW [ms]	Total SW/HW [ms]	Total Speed-up	Accel. SW [ms]	Accel. HW [ms]	Accel. Speed-up	SW part Sped up by HW [%]
Encaps								
FrodoKem	1:Frodo-640	16.192	1.414	**11.5**	15.10635	0.328	46.0	93.29
FrodoKem	3:Frodo-976	34.609	2.028	**17.1**	33.31272	0.732	45.5	96.26
FrodoKEM	5:Frodo-1344	62.076	1.977	**31.4**	61.39795	1.299	47.3	98.91
Round5	1:R5ND-1PKE_0d	9.899	0.055	**179.8**	9.86147	0.018	556.5	99.62
Round5	3:R5ND-3PKE_0d	20.807	0.077	**269.4**	20.75316	0.023	905.6	99.74
Round5	5:R5ND-5PKE_0d	39.097	0.100	**389.9**	39.12033	0.030	1,299.7	99.56
Saber	1:LightSaber-KEM	0.379	0.051	**7.4**	0.34173	0.014	23.7	90.25
Saber	3:Saber-KEM	0.725	0.069	**10.6**	0.67592	0.020	34.4	93.24
Saber	5:FireSaber-KEM	1.195	0.094	**12.7**	1.12595	0.025	44.8	94.21
NTRU-HPS	1:ntruhps2048677	3.066	0.386	**7.9**	2.69311	0.013	203.6	87.84
NTRU-HPS	3:ntruhps4096821	4.416	0.475	**9.3**	3.95545	0.015	271.5	89.58
NTRU-HRSS	1:ntruhrss701	3.044	0.171	**17.8**	2.88665	0.014	209.8	94.83
Str NTRU Prime	2:kem/sntrup653	34.936	0.540	**64.7**	34.40847	0.013	2,750.8	98.49
Str NTRU Prime	3:kem/sntrup761	47.343	0.646	**73.2**	46.70855	0.012	3,974.5	98.66
Str NTRU Prime	4:kem/sntrup857	59.930	0.727	**82.4**	59.21685	0.014	4,188.3	98.81
NTRU LPRime	2:kem/ntrulpr653	70.636	1.843	**38.3**	68.81706	0.024	2,863.2	97.42
NTRU LPRime	3:kem/ntrulpr761	95.490	2.093	**45.6**	93.41708	0.020	4,681.3	97.83
NTRU LPRime	4:kem/ntrulpr857	120.775	2.363	**51.1**	118.43359	0.022	5,432.4	98.06
Encaps without randombytes()								
FrodoKem	1:Frodo-640	16.191	1.413	**11.5**	15.10635	0.328	46.0	93.30
FrodoKem	3:Frodo-976	34.606	2.026	**17.1**	33.31272	0.732	45.5	96.26
FrodoKEM	5:Frodo-1344	62.076	1.977	**31.4**	61.39795	1.299	47.3	98.91
Round5	1:R5ND-1PKE_0d	9.898	0.054	**183.1**	9.86147	0.018	556.5	99.63
Round5	3:R5ND-3PKE_0d	20.806	0.076	**272.9**	20.75316	0.023	905.6	99.74
Round5	5:R5ND-5PKE_0d	39.096	0.099	**395.8**	39.02695	0.030	1,296.6	99.82
Saber	1:LightSaber-KEM	0.377	0.050	**7.6**	0.34173	0.014	23.9	90.61
Saber	3:Saber-KEM	0.723	0.067	**10.8**	0.67592	0.020	34.4	93.44
Saber	5:FireSaber-KEM	1.195	0.094	**12.7**	1.12595	0.025	44.8	94.22
NTRU-HPS	1:ntruhps2048677	2.954	0.274	**10.8**	2.69311	0.013	203.6	91.18
NTRU-HPS	3:ntruhps4096821	4.280	0.338	**12.7**	3.95661	0.015	271.6	92.44
NTRU-HRSS	1:ntruhrss701	2.995	0.122	**24.5**	2.88665	0.014	209.8	96.38
Str NTRU Prime	2:kem/sntrup653	34.638	0.242	**142.9**	34.40847	0.013	2,750.8	99.34
Str NTRU Prime	3:kem/sntrup761	46.997	0.300	**156.7**	46.70855	0.012	3,974.5	99.39
Str NTRU Prime	4:kem/sntrup857	59.543	0.340	**175.0**	59.21685	0.014	4,188.3	99.45
NTRU LPRime	2:kem/ntrulpr653	70.635	1.842	**38.4**	68.81706	0.024	2,863.2	97.43
NTRU LPRime	3:kem/ntrulpr761	95.489	2.092	**45.6**	93.41708	0.020	4,681.3	97.83
NTRU LPRime	4:kem/ntrulpr857	120.775	2.363	**51.1**	118.43359	0.022	5,432.4	98.06
Decaps								
FrodoKem	1:Frodo-640	16.192	1.414	**11.5**	15.10635	0.328	46.1	93.29
FrodoKem	3:Frodo-976	34.649	2.058	**16.8**	33.32329	0.733	45.5	96.18
FrodoKEM	5:Frodo-1344	62.377	2.608	**23.9**	61.06782	1.299	47.0	97.90
Round5	1:R5ND-1PKE_0d	14.826	0.043	**343.3**	14.80661	0.024	621.5	99.87
Round5	3:R5ND-3PKE_0d	31.177	0.063	**495.8**	31.14658	0.033	944.9	99.90
Round5	5:R5ND-5PKE_0d	58.598	0.082	**711.6**	58.55841	0.043	1,375.8	99.93
Saber	1:LightSaber-KEM	0.474	0.054	**8.8**	0.44317	0.024	18.8	93.56
Saber	3:Saber-KEM	0.867	0.069	**12.6**	0.82878	0.030	27.2	95.60
Saber	5:FireSaber-KEM	1.379	0.086	**16.0**	1.32991	0.037	35.7	96.43
NTRU-HPS	1:ntruhps2048677	8.175	0.114	**71.7**	8.09307	0.032	251.3	99.00
NTRU-HPS	3:ntruhps4096821	11.982	0.112	**107.1**	11.90773	0.038	313.7	99.38
NTRU-HRSS	1:ntruhrss701	8.790	0.128	**68.5**	8.69522	0.034	257.4	98.92
Str NTRU Prime	2:kem/sntrup653	106.391	0.341	**311.9**	106.07692	0.027	3,915.2	99.70
Str NTRU Prime	3:kem/sntrup761	144.361	0.392	**368.0**	143.9971	0.028	5,148.0	99.75
Str NTRU Prime	4:kem/sntrup857	182.965	0.437	**418.7**	182.55901	0.031	5,878.2	99.78
NTRU LPRime	2:kem/ntrulpr653	104.550	1.359	**77.0**	103.22538	0.034	3,043.2	98.73
NTRU LPRime	3:kem/ntrulpr761	141.615	1.526	**92.8**	140.12556	0.036	3,853.6	98.95
NTRU LPRime	4:kem/ntrulpr857	179.322	1.712	**104.7**	177.65014	0.040	4,407.9	99.07

Table 5: : The execution time of randombytes() in absolute units (Time [us]) and as a percentage of the total execution time of Encapsulation a) in software (% in SW) and b) using software/hardware codesign (% in SW/HW). #Calls denotes the total number of calls to the function randombytes(), and #Bytes – the total number of random bytes generated by these calls.

Algorithm	Parameter Set	Time [us]	% in SW	% in SW/HW	#Calls	#Bytes
FrodoKem	1:Frodo-640	1.6	0.000099	0.001131	1	16
FrodoKem	3:Frodo-976	2.2	0.001085	0.001085	1	24
FrodoKEM	5:Frodo-1344	2.22	0.000024	0.000759	1	32
Round5	1:R5ND-1KEM_0d	0.94	0.000009	0.017078	1	16
Round5	3:R5ND-3KEM_0d	0.96	0.000005	0.012428	1	24
Round5	5:R5ND-5KEM_0d	1.52	0.000039	0.007223	1	32
Saber	1:LightSaber-KEM	1.5	0.003961	0.029198	1	32
Saber	3:Saber-KEM	1.5	0.002069	0.021860	1	32
Saber	5:FireSaber-KEM	1.5	0.001260	0.010596	1	32
NTRU-HPS	1:ntruhps2048677	112.14	0.036576	0.290458	1	3211
NTRU-HPS	3:ntruhps4096821	135.61	3.070000	28.560000	1	3895
NTRU-HRSS	1:ntruhrss701	48.77	1.600000	48.770000	1	1400
Str NTRU Prime	2:kem/sntrup653	297.22	0.850764	55.073527	653	2612
Str NTRU Prime	3:kem/sntrup761	326.47	0.731825	53.593964	761	3044
Str NTRU Prime	4:kem/sntrup857	386.86	0.650000	53.210000	857	3428
NTRU LPRime	2:kem/ntrulpr653	1.52	0.000022	0.000825	1	32
NTRU LPRime	3:kem/ntrulpr761	1.5	0.000717	0.000717	1	32
NTRU LPRime	4:kem/ntrulpr857	1.54	0.000013	0.000642	1	32

From Tables 5 and 4, it can be clearly seen that the ranking of algorithms in terms of the total execution time is not affected by this choice for the purely software implementations, where the execution time of randombytes() does not exceed 3.1% for any investigated algorithm. Coincidentally, the ranking does not change significantly even for the software/hardware implementations, in spite of the fact that the execution time of randombytes() reaches 55.1% of the total execution time of Encapsulation for Streamlined NTRU Prime.

The total speed-up is by far the highest for Round5, due to the a) initial very high percentage of time taken by the accelerated operations (more than 99.56% for encapsulation and more than 99.87% for decapsulation), b) limited size of input to and output from the accelerator, and c) high potential for the parallelization in hardware (with the speed up of the accelerated portion reaching 1,299.7 for encapsulation and 1,375.8 for decapsulation). For similar reasons the total speed-up is also very high (greater than 38) for Streamlined NTRU Prime and NTRU LPrime, during both encapsulation and decapsulation.

NTRU-HPS and NTRU-HRSS achieve high overall speed-ups, but only for decapsulation, mostly because the accelerated portion of encapsulation takes less than 96.4% of the total execution time, even without counting the execution time of randombytes(). For FrodoKEM, the overall speed-up is comparable for encapsulation and decapsulation, and varies between 11.5 and 31.4 for encapsulation, and between 11.5 and 23.9 for decapsulation. For Saber, the total speed-up varies between 7.4 and 12.7 for encapsulation, and between 8.8 and 16.0 for decapsulation. Overall, the total speed-up is greater than 7 for all reported cases. As expected, the speed-up increases with the increase in the security level. This dependency exists because for larger parameter values, a higher level of parallelization can be typically achieved by the operations offloaded to hardware. Additionally, the operations offloaded to hardware tend to account for a larger percentage of the total execution time in software, as illustrated by the column SW part Sped up by HW [%] in Table 4.

Below, we describe the ranking of algorithms, separately for three groups of parameter sets listed in Table 3, with the security categories 1 and 2, 3 only, and 4 and 5, respectively. Only the first group contains all 7 investigated algorithms. In the second group NTRU-HRSS is missing, and in the third group both NTRU-HRSS and NTRU-HPS are not represented.

For all groups, the ranking of algorithms, in terms of the total execution time (in milliseconds), changes after offloading the most time-consuming operations to hardware. In particular, for the first group of parameter sets, covering the security categories 1 and 2, for encapsulation, the purely software ranking is:

Table 6: Maximum frequency and resource utilization.

Algorithm	Security Category: Parameter Set	Clock Freq. [MHz]	LUTs	Slices	FFs	36kb BRAMs	DSPs
FrodoKEM	1:Frodo-640	402	7,213	1,186	6,647	13.5	32
FrodoKEM	3:Frodo-976	402	7,087	1,190	6,693	17	32
FrodoKEM	5:Frodo-1344	417	7,015	1,215	6,610	17.5	32
Round5	1:R5ND-1PKE_0d	260	55,442	10,627	82,341	2	0
Round5	3:R5ND-3PKE_0d	249	73,881	14,307	109,211	2	0
Round5	5:R5ND-5KEM_0d	212	91,166	18,733	151,019	2	0
Saber	1:LightSaber-KEM	322	12,343	1,989	11,288	3.5	256
Saber	3:Saber-KEM	322	12,566	1,993	11,619	3.5	256
Saber	5:FireSaber-KEM	322	12,555	2,341	11,881	3.5	256
NTRU-HPS	1:ntruhps2048677	200	24,328	4,972	19,244	2.5	677
NTRU-HPS	3:ntruhps4096821	200	29,389	5,913	23,338	2.5	821
NTRU-HRSS	1:ntruhrss701	200	27,218	5,770	21,410	2.5	701
Str NTRU Prime	2:kem/sntrup653	244	55,843	8,134	28,143	3	0
Str NTRU Prime	3:kem/sntrup761	244	62,595	9,176	32,763	3	0
Str NTRU Prime	4:kem/sntrup857	244	70,604	9,894	37,018	3	0
NTRU LPRime	2:kem/ntrulpr653	244	50,911	7,874	34,050	2	0
NTRU LPRime	3:kem/ntrulpr761	244	51,295	7,978	39,600	2	0
NTRU LPRime	4:kem/ntrulpr857	244	58,056	8,895	44,719	2	0

1. Saber, 2-3. NTRU-HRSS and NTRU-HPS (with very similar results and the order swapped depending on counting or not the execution time of randombytes()) 4. Round5, 5. FrodoKEM, 6. Streamlined NTRU Prime, and 7. NTRU LPRime. For the software/hardware implementations, this ranking changes to 1. Saber, 2. Round5 (with results for Saber and Round5 very close to each other), 3. NTRU-HRSS, 4. NTRU-HPS (with NTRU-HRSS outperforming NTRU-HPS by more than a factor of 2), 5. Streamlined NTRU Prime, 6. FrodoKEM, and 7. NTRU LPRime. Thus, Round5 advances by two positions, ahead of NTRU-HRSS and NTRU-HPS. Additionally, Streamlined NTRU Prime advances ahead of FrodoKEM. The first position of Saber and the last position of NTRU LPRime remain unchanged.

For decapsulation, the software only ranking is 1. Saber, 2. NTRU-HPS, 3. NTRU-HRSS (with results for NTRU-HPS and NTRU-HRSS very close to each other), 4. Round5, 5. FrodoKEM, 6. NTRU LPRime, and 7. Streamlined NTRU Prime. Compared to encapsulation, only the positions of Streamlined NTRU Prime and NTRU LPRime have been swapped. For the software/hardware implementations, the ranking changes to 1. Round5, 2. Saber, 3. NTRU-HPS, 4. NTRU-HRSS, 5. Streamlined NTRU Prime, 6. NTRU LPRime, and 7. FrodoKEM. Thus, Round5 advanced by 3 positions, to the first place. Additionally, FrodoKEM and Streamlined NTRU Prime swapped positions 5 and 7. Thus, clearly Round5 and Streamlined NTRU Prime benefited the most from moving their most time consuming operations to hardware.

For the second group of parameter sets, covering the security category 3, for encapsulation, the ranking of candidates based on the purely software implementations is identical as in the first group, except that NTRU-HRSS is now missing. For the software/hardware implementations, the ranking is also almost identical as for group 1, except that now, the results for NTRU-HPS and Streamlined NTRU Prime are close to each other, and the ranking of these algorithms at positions 3 and 4 depends on the inclusion or exclusion of the execution time of randombytes(). For decapsulation, both rankings remain the same as in group 1 (except of the absence of NTRU-HRSS). Similarly, in the third group of parameter sets, covering the security categories 4 and 5, NTRU-HPS is not any longer represented. However, the rankings of remaining algorithms (for both software and software/hardware implementations) remain the same as in group 2.

The maximum clock frequencies and the corresponding resource utilizations, obtained after the synthesis and implementation tool optimizations, supported by Minerva [8], are summarized in Table 6. Clearly, the accelerators for NTRU-HPS and NTRU-HRSS involve the highest number of integer multiplications performed in parallel. These multiplications in the FPGA fabric are delegated to dedicated DSP units. The DSP units are also taken advantage of in Saber and to a lower extent in FrodoKEM. Round5, Streamlined NTRU Prime and NTRU LPRime do not involve any integer multiplications in hardware. This is because

the coefficients of one of the multiplied polynomials always belong to the set {-1, 0, 1}.

Because of the timing dependencies, and in particular, the bottleneck caused by SHAKE, our implementation of FrodoKEM cannot be easily sped up by trading additional resources for speed. This example clearly illustrates the potential algorithmic limits on the amount of potential parallelization (and thus the maximum speed-up), which is independent of the amount of hardware resources available to the designer. FrodoKEM is also an algorithm with the highest utilization of BRAMs, which reaches 17.5. The remaining algorithms use only between 2 and 3.5 36kb BRAMs. Round5, Streamlined NTRU Prime, and NTRU LPRime, which demonstrated the highest potential for hardware acceleration, use also the highest number of LUTs, Slices, and flip-flops (FFs). The amount of resources used increases noticeably with the increase in the security level for 5 out of 7 algorithms. The only exceptions are FrodoKEM and Saber, in which the security levels do not affect the resource utilization (except of the small increase in the number of BRAMs in FrodoKEM).

FrodoKEM is able to achieve the highest clock frequency, above 400 MHz for all parameter sets. This frequency is possible because the accelerator processes only 4 elements of each row of the product B'V at a time. This allows us to pipeline the Frodo MAC unit with additional registers between multipliers and adders. These registers are also built-in inside DSP units. The same optimization is not possible for Saber and NTRU, because the immediate feedback from the output registers is necessary for the next operation happening in the next clock cycle. NTRU design also suffers from additional logic for converting polynomials from R/q to S/q and from R/q to S/3. Operating frequency for the two variants of NTRU Prime is mainly limited by the "modulo q" block. To reduce numbers with the prime modulus q, we selected the conditional subtraction method, which is relatively simple, but comes with a long critical path.

7 Conclusions

In this paper we have demonstrated the feasibility of a new benchmarking approach, based on the software/hardware codesign, with application to 7 PQC schemes representing 5 submissions qualified to Round 2 of the NIST PQC standardization process. We have shown that the obtained speed-up depends strongly on the evaluated algorithm. For all analyzed schemes, and both major operations (encapsulation and decapsulation), the total speed-up always exceeded a factor of 7. For encapsulation the highest speed-up reached 396 for Round5 (without counting the execution time of randombytes()). For decapsulation the speed-ups were even more spectacular, reaching a factor of 712 for Round5, 419 for Streamlined NTRU Prime, 107 for NTRU-HPS, and 105 for NTRU LPRime. Only two out of seven evaluated algorithms (FrodoKEM and Saber) have decapsulation speed-ups smaller than 50.

Round5 benefited greatly from hardware acceleration due to its simple operations (not involving integer multiplication) and their inherent parallelism. Its achieved speed-up is the highest, but at the highest cost in terms of CLB Slice utilization. On the other hand no DSP units are used, and the use of Block RAMs is minimal. Due to accelerating almost 100% of the software execution time, it seems that maximum performance limit has been reached. Due to these significant speed-ups, Round5 is ranked the second for encapsulation and the first for decapsulation for all three investigated parameter groups. Saber, which is by far the fastest in software only implementations, for the software/hardware implementations remains on the first position for encapsulation, and moves to the second position for decapsulation. Saber has also relatively low resource utilization compared to other candidates (second in terms of the number of CLB Slices), but requires a substantial number of DSP units.

On the other end of the spectrum is FrodoKEM. Despite almost reaching its limit in terms of hardware acceleration, FrodoKEM is by far the slowest for decapsulation, and the second slowest for encapsulation. The results for NTRU-HPS and NTRU-HRSS place these candidates in the middle of the pack. A modification of the Round 1 NTRU algorithm resulted in a significant increase in DSP unit utilization. With at least twice as high logic utilization (in terms of CLB slices and DSPs) compared to Saber, the overall evaluation is clearly worse. The two KEMs associated with NTRU Prime start from the worst performance in embedded software. Despite impressive speed-ups (especially for decapsulation) their overall ranking improves only slightly, with only Streamlined NTRU Prime outperforming FrodoKEM for encapsulation, and both outperforming FrodoKEM for decapsulation.

Future work will include extending this analysis to the remaining NIST Round 2 PQC candidates, as well as the exploration of other software/hardware codesign platforms and development tools.

References

[1] ARM. *AMBA: The Standard for On-Chip Communication*. https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications. 2019.

[2] Aydin Aysu, Bilgiday Yuce, and Patrick Schaumont. "The Future of Real-Time Security: Latency-Optimized Lattice-Based Digital Signatures". In: *ACM Transactions on Embedded Computing Systems* 14.3 (Apr. 2015), pp. 1–18. ISSN: 15399087. DOI: 10.1145/2724714.

[3] Mihir Bellare and Phillip Rogaway. *Introduction to Modern Cryptography*. May 2005.

[4] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. https://bench.cr.yp.to. 2019.

[5] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - Web Page*. https://competitions.cr.yp.to/caesar.html. 2019.

[6] Cryptographic Engineering Research Group (CERG) at George Mason University. *Hardware Benchmarking of CAESAR Candidates*. https://cryptography.gmu.edu/athena/index.php?id=CAESAR. 2019.

[7] Farnoud Farahmand et al. "Evaluating the Potential for Hardware Acceleration of Four NTRU-Based Key Encapsulation Mechanisms Using Software/Hardware Codesign". In: *10th International Conference on Post-Quantum Cryptography, PQCrypto 2019*. LNCS. Chongqing, China: Springer, May 2019.

[8] Farnoud Farahmand et al. "Minerva: Automated Hardware Optimization Tool". In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun: IEEE, Dec. 2017, pp. 1–8. ISBN: 978-1-5386-3797-5. DOI: 10.1109/RECONFIG.2017.8279804.

[9] Ahmed Ferozpuri and Kris Gaj. "High-Speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme". In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun, Mexico: IEEE, Dec. 2018, pp. 1–8. ISBN: 978-1-72811-968-7. DOI: 10.1109/RECONFIG.2018.8641734.

[10] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Journal of Cryptology* 26.1 (Jan. 2013), pp. 80–101. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/s00145-011-9114-1.

[11] Kris Gaj. "Challenges and Rewards of Implementing and Benchmarking Post-Quantum Cryptography in Hardware". In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI - GLSVLSI '18*. Chicago, IL, USA: ACM Press, 2018, pp. 359–364. ISBN: 978-1-4503-5724-1. DOI: 10.1145/3194554.3194615.

[12] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs". In: *Cryptographic Hardware and Embedded Systems, CHES 2010*. Vol. 6225. LNCS. Santa Barbara, CA, Aug. 2010, pp. 264–278. ISBN: 978-3-642-15030-2 978-3-642-15031-9. DOI: 10.1007/978-3-642-15031-9_18.

[13] Kris Gaj et al. "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs". In: *2010 International Conference on Field Programmable Logic and Applications, FPL 2010*. Milan, Italy: IEEE, Aug. 2010, pp. 414–421. ISBN: 978-1-4244-7842-2. DOI: 10.1109/FPL.2010.86.

[14] Santosh Ghosh et al. "A Speed Area Optimized Embedded Co-Processor for McEliece Cryptosystem". In: *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2012*. Delft, Netherlands: IEEE, July 2012, pp. 102–108. ISBN: 978-1-4673-2243-0 978-0-7695-4768-8. DOI: 10.1109/ASAP.2012.16.

[15] Shafi Goldwasser and Mihir Bellare. *Lecture Notes on Cryptography*. July 2008.

[16] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *Theory of Cryptography*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Cham: Springer International Publishing, 2017, pp. 341–371. ISBN: 978-3-319-70499-9 978-3-319-70500-2. DOI: 10.1007/978-3-319-70500-2_12.

[17] Ekawat Homsirikamol and Kris Gaj. "Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study". In: *Applied Reconfigurable Computing - ARC 2015*. Vol. 9040. LNCS. Cham: Springer International Publishing, 2015, pp. 217–228. ISBN: 978-3-319-16213-3 978-3-319-16214-0. DOI: 10.1007/978-3-319-16214-0_18.

[18] Ekawat Homsirikamol and Kris Gaj. "Toward a New HLS-Based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study". en. In: *2017 International Conference on Field Programmable Technology (ICFPT)*. Melbourne, Australia: IEEE, Dec. 2017, pp. 120–127. ISBN: 978-1-5386-2656-6. DOI: 10.1109/FPT.2017.8280129.

[19] Ekawat Homsirikamol, Panasayya Yalla, and Farnoud Farahmand. *Development Package for Hardware Implementations Compliant with the CAESAR Hardware API*. 2016.

[20] Ekawat Homsirikamol et al. *CAESAR Hardware API*. Cryptology ePrint Archive 2016/626. 2016.

[21] Ekawat Homsirikamol et al. *Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API*. GMU Report. Fairfax, VA: George Mason University, 2016.

[22] James Howe et al. "Standard Lattice-Based Key Encapsulation on Embedded Devices". en. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (Aug. 2018), pp. 372–393. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i3.372-393.

[23] Matthias J. Kannwischer et al. *Pqm4 - Post-Quantum Crypto Library for the {ARM} {Cortex-M4}*. https://github.com/mupq/pqm4. 2019.

[24] John Kelsey, Shu-jen Chang, and Ray Perlner. *NIST Special Publication 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash*. Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, Dec. 2016. DOI: 10.6028/NIST.SP.800-185.

[25] Brian Koziel et al. "Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.1 (Jan. 2017), pp. 86–99. ISSN: 1549-8328, 1558-0806. DOI: 10.1109/TCSI.2016.2611561.

[26] National Institute of Standards and Technology. *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Aug. 2015. DOI: 10.6028/NIST.FIPS.202.

[27] Richard Newell. *Survey of Notable Security-Enhancing Activities in the RISC-V Universe*. 17th International Workshop on Cryptographic Architectures Embedded in Logic Devices, CryptArchi 2019. Pruhonice, Czech Republic, June 2019.

[28] NIST. *PQC - API Notes*. 2017.

[29] Tobias Oder and Tim Guneysu. "Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs". en. In: *LATINCRYPT 2017*. Havana, Cuba, Sept. 2017.

[30] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Book version: 0.0.1. Strawberry Canyon LLC, Oct. 2017.

[31] Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto. *Optimized ANSI C Code for the Rijndael Cipher (Now AES), Rijndael-Alg-Fst.c, v3.0*. Dec. 2000.

[32] Markku-Juhani O. Saarinen. *Pqcbench*. https://github.com/mjosaarinen/pqcbench. 2019.

[33] Douglas Stebila and Michele Mosca. *Liboqs - Master Branch*. https://github.com/open-quantum-safe/liboqs. 2019.

[34] FrodoKEM Submission Team. *Round 2 Submissions - FrodoKEM Candidate Submission Package*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions. Apr. 2019.

[35] NTRU Prime Submission Team. *Round 2 Submissions - NTRU Prime Candidate Submission Package*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions. Apr. 2019.

[36] NTRU Submission Team. *Round 2 Submissions - NTRU Candidate Submission Package*. Apr. 2019.

[37] Round5 Submission Team. *Round 2 Submissions - Round5 Candidate Submission Package*. Apr. 2019.

[38] Saber Submission Team. *Round 2 Submissions - Saber Candidate Submission Package*. Apr. 2019.

[39] Wen Wang, Jakub Szefer, and Ruben Niederhagen. "FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes". In: *9th International Conference on Post-Quantum Cryptography, PQCrypto 2018*. Ed. by Tanja Lange and Rainer Steinwandt. Vol. 10786. LNCS. Fort Lauderdale, Florida: Springer International Publishing, Apr. 2018, pp. 77–98. ISBN: 978-3-319-79062-6 978-3-319-79063-3. DOI: 10.1007/978-3-319-79063-3_4.

[40] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual . Volume I: Unprivileged ISA v2.2*. Tech. rep. 20190608-Base-Ratified. June 2019, p. 236.

[41] Andrew Waterman and Krste Asanovic. "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, v1.12". In: (June 2019), p. 113.

A Profiling Results

Table 7: Results of profiling for FrodoKEM

Function	Time [us]	Time [%]	Function	Time [us]	Time [%]
Software			**Software/Hardware**		
FrodoKEM640 - Encaps					
1. frodo_mul_add_sa_plus_e	13,794.27	85.19	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,002.40	6.19	**1.2 Shake128 and frodo_sample_n**	328.16	23.20
3. frodo_mul_add_sb_plus_e	309.68	1.91	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_pack	291.83	1.80	2. frodo_pack	291.83	20.64
5. frodo_unpack	277.26	1.71	3. frodo_unpack	277.26	19.61
Total	16,192.37	96.81	Total	1,414.18	63.45
FrodoKEM640 - Encaps without Randombytes()					
1. frodo_mul_add_sa_plus_e	13,794.27	85.20	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,002.40	6.19	**1.2 Shake128 and frodo_sample_n**	328.16	23.23
3. frodo_mul_add_sb_plus_e	309.68	1.91	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_pack	291.83	1.80	2. frodo_pack	291.83	20.66
5. frodo_unpack	277.26	1.71	3. frodo_unpack	277.26	19.63
Total	16,190.75	96.82	Total	1,412.56	63.52
FrodoKEM640 - Decapsulation					
1. frodo_mul_add_sa_plus_e	13,793.01	85.23	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,002.85	6.20	**1.2 Shake128 and frodo_sample_n**	327.97	23.19
3. frodo_unpack x3	548.74	3.39	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_mul_add_sb_plus_e	309.21	1.91	2. frodo_unpack x3	548.74	38.81
5. frodo_mul_bs	242.40	1.50	3. frodo_mul_bs	242.40	17.14
Total	16,182.80	98.23	Total	1,413.99	79.15
FrodoKEM976 - Encaps					
1. frodo_mul_add_sa_plus_e	31,430.38	90.82	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,410.18	4.07	**1.2 Shake128 and frodo_sample_n**	732.09	36.10
3. frodo_mul_add_sb_plus_e	472.16	1.36	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_pack	357.58	1.03	2. frodo_pack	357.58	17.63
5. frodo_unpack	297.73	0.86	3. frodo_unpack	297.73	14.68
Total	34,608.54	98.15	Total	2,027.91	68.42
FrodoKEM976 - Encaps without Randombytes()					
1. frodo_mul_add_sa_plus_e	31,430.38	90.82	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,410.18	4.07	**1.2 Shake128 and frodo_sample_n**	732.09	36.14
3. frodo_mul_add_sb_plus_e	472.16	1.36	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_pack	357.58	1.03	2. frodo_pack	357.58	17.65
5. frodo_unpack	297.73	0.86	3. frodo_unpack	297.73	14.70
Total	34,606.34	98.16	Total	2,025.71	68.49
FrodoKEM976 - Decaps					
1. frodo_mul_add_sa_plus_e	31,441.14	90.74	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,410.86	4.07	**1.2 Shake128 and frodo_sample_n**	732.58	35.60
3. frodo_unpack x3	594.63	1.72	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_mul_add_sb_plus_e	471.29	1.36	2. frodo_unpack x3	594.63	28.90
5. frodo_mul_bs	368.32	1.06	3. frodo_mul_bs	368.32	17.90
Total	34,648.58	98.95	Total	2,057.87	82.39
FrodoKEM1344 - Encaps					
1. frodo_mul_add_sa_plus_e	58,577.48	94.41	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,416.27	2.28	**1.2 Shake128 and frodo_sample_n**	1,298.85	65.71
3. frodo_mul_add_sb_plus_e	654.64	1.06	**1.3 frodo_mul_add_sb_plus_e**		
4.frodo_pack	386.22	0.62	2. frodo_pack	386.22	19.54
5.frodo_unpack	276.00	0.44	3. frodo_unpack	276.00	13.96
Total	62,048.92	98.81	Total	1,976.73	99.21
FrodoKEM1344 - Encaps without Randombytes()					
1. frodo_mul_add_sa_plus_e	58,577.48	94.41	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	1,416.27	2.28	**1.2 Shake128 and frodo_sample_n**	1,298.85	65.76
3. frodo_mul_add_sb_plus_e	654.64	1.06	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_pack	386.22	0.62	2. frodo_pack	386.22	19.55
5. frodo_unpack	276.00	0.44	3. frodo_unpack	276.00	13.97
Total	62,046.72	98.81	Total	1,975.23	99.28
FrodoKEM1344 - Decaps					
1. frodo_mul_add_sa_plus_e	58,754.02	94.22	**1.1 frodo_mul_add_sa_plus_e**		
2. Shake128 and frodo_sample_n x3	883.14	1.42	**1.2 Shake128 and frodo_sample_n**	1,298.53	49.79
3. frodo_unpack x3	765.56	1.23	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_mul_add_sb_plus_e	649.68	1.04	2. frodo_unpack x3	765.56	29.36
5. frodo_mul_bs	507.08	0.81	3. frodo_mul_bs	507.08	19.44
Total	62,359.42	98.72	Total	2,607.89	98.59

Table 8: Results of profiling for Round5

Function	Time [us]	Time [%]	Function	Time [us]	Time [%]
Software			**Software/Hardware**		
R5ND_5PKE_0d - Encapsulation					
1. r5_cpa_pke_encrypt	39,026.95	99.82%	1. hash	68.65	68.46%
2. hash	68.65	0.1756%	**2. r5_cpa_pke_encrypt**	30.10	30.02%
3. randombytes	1.52	0.0039%	3. randombytes	1.52	1.52%
Total	39,097.13	100.00%	Total	100.28	99.99%
R5ND_5PKE_0d - Encapsulation without randombytes()					
1. r5_cpa_pke_encrypt	39,026.95	99.82%	1. hash	68.65	69.50%
2. hash	68.65	0.1756%	**2. r5_cpa_pke_encrypt**	30.10	30.47%
Total	39,095.61	100.00%	Total	98.78	99.97%
R5ND_5PKE_0d - Decapsulation					
1. r5_cpa_pke_encrypt	39,021.09	66.59%	1. hash_2	35.75	43.42%
2. r5_cpa_pke_decrypt	19,504.52	33.29%	**2.1 r5_cpa_pke_decrypt**		
3. hash_2	35.75	0.06%	**2.2 hash_1**	42.56	51.69%
4. hash_1	32.80	0.06%	**2.3 r5_cpa_pke_encrypt**		
Total	58,598.19	99.99%	Total	82.34	95.11%
R5ND_3PKE_0d - Encapsulation					
1. r5_cpa_pke_encrypt	20,753.16	99.74%	1. hash	52.87	68.44%
2. hash	52.87	0.2541%	**2. r5_cpa_pke_encrypt**	22.92	29.67%
3. randombytes	1.01	0.0049%	3. randombytes	1.01	1.31%
Total	20,807.49	100.00%	Total	77.25	99.42%
R5ND_3PKE_0d - Encapsulation without randombytes()					
1. r5_cpa_pke_encrypt	20,753.16	99.74%	1. hash	52.87	69.35%
2. hash	52.87	0.2541%	**2. r5_cpa_pke_encrypt**	22.92	30.06%
Total	20,806.48	100.00%	Total	76.24	99.41%
R5ND_3PKE_0d - Decapsulation					
1. r5_cpa_pke_encrypt	20,748.66	66.55%	**1.1 r5_cpa_pke_decrypt**		
2. r5_cpa_pke_decrypt	10,373.33	33.27%	**1.2 hash_1**	32.98	52.45%
3. hash_2	27.34	0.09%	**1.3 r5_cpa_pke_encrypt**		
4. hash_1	24.59	0.08%	2. hash_2	27.34	43.48%
Total	31,176.50	99.99%	Total	62.88	95.92%
R5ND_1PKE_0d - Encapsulation					
1. r5_cpa_pke_encrypt	9,861.47	99.62%	1. hash	34.30	62.32%
2. hash	34.30	0.3465%	**2. r5_cpa_pke_encrypt**	17.72	32.19%
3. randombytes	0.99	0.0100%	3. randombytes	0.99	1.80%
Total	9,898.79	99.98%	Total	55.04	96.31%
R5ND_1PKE_0d - Encapsulation without randombytes()					
1. r5_cpa_pke_encrypt	9,861.47	99.63%	1. hash	34.30	63.46%
2. hash	34.30	0.3465%	**2. r5_cpa_pke_encrypt**	17.72	32.78%
Total	9,897.80	99.98%	Total	54.05	96.24%
R5ND_1PKE_0d - Decapsulation					
1. r5_cpa_pke_encrypt	9,857.37	66.49%	**1.1 r5_cpa_pke_decrypt**		
2. r5_cpa_pke_decrypt	4,932.58	33.27%	**1.2 hash_1**	23.82	55.28%
3. hash_2	17.46	0.12%	**1.3 r5_cpa_pke_encrypt**		
4. hash_1	16.66	0.11%	2. hash_2	17.46	40.51%
Total	14,825.97	99.99%	Total	43.10	95.79%

Table 9: Results of profiling for Saber

Function	Time [us]	Time [%]	Function	Time [us]	Time [%]
Software			**Software/Hardware**		
LightSaber - Encaps					
1. MatrixVectorMul	204.70	54.06	1. Hash	28.27	55.03
2. InnerProduct	102.57	27.09	**2.1 MatrixVectorMul**	14.443	28.11
3. GenMatrix	23.64	6.24	**2.2 InnerProduct**		
4. Hash	28.27	7.47	**2.3 GenMatrix**		
5. GenSecret	10.82	2.86	**2.4 GenSecret**		
Total	378.66	97.71	Total	51.37	83.14
LightSaber - Encaps without Randombytes()					
1. MatrixVectorMul	204.70	54.27	1. Hash	28.27	56.68
2. InnerProduct	102.57	27.20	**2.1 MatrixVectorMul**	14.443	28.96
3. GenMatrix	23.64	6.27	**2.2 InnerProduct**		
4. Hash	28.27	7.50	**2.3 GenMatrix**		
5. GenSecret	10.82	2.87	**2.4 GenSecret**		
Total	377.16	98.10	Total	49.87	85.64
LightSaber - Decaps					
1. MatrixVectorMul	203.74	43.01	1. Hash	15.49	28.64
2. InnerProduct x2	204.78	43.23	**2.1 MatrixVectorMul**	23.59	43.61
3. GenMatrix	23.81	5.03	**2.2 InnerProduct x2**		
4. Hash	15.49	3.27	**2.3 GenMatrix**		
5. GenSecret	10.84	2.29	**2.4 GenSecret**		
Total	473.67	96.83	Total	54.09	72.25%
Saber - Encaps					
1. MatrixVectorMul	458.14	63.20	1. Hash	39.05	56.91
2. InnerProduct	153.34	21.15	**2.1 MatrixVectorMul**	19.65	28.64
3. GenMatrix	53.46	7.37	**2.2 InnerProduct**		
4. Hash	39.05	5.39	**2.3 GenMatrix**		
5. GenSecret	10.98	1.51	**2.4 GenSecret**		
Total	724.89	98.63	Total	68.62	85.54
Saber - Encaps without randombytes()					
1. MatrixVectorMul	458.14	63.33	1. Hash	39.05	58.18
2. InnerProduct	153.34	21.20	**2.1 MatrixVectorMul**	19.65	29.28
3. GenMatrix	53.46	7.39	**2.2 InnerProduct**		
4. Hash	39.05	5.40	**2.3 GenMatrix**		
5. GenSecret	10.98	1.52	**2.4 GenSecret**		
Total	723.39	98.84	Total	67.12	87.46
Saber - Decaps					
1. MatrixVectorMul	457.70	52.79	1. Hash	20.73	30.21
2. InnerProduct x2	306.54	35.36	**2.1 MatrixVectorMul**	30.47	44.39
3. GenMatrix	53.56	6.18	**2.2 InnerProduct x2**		
4. Hash	20.73	2.39	**2.3 GenMatrix**		
5. GenSecret	10.98	1.27	**2.4 GenSecret**		
Total	866.94	97.99	Total	68.63	74.60
FireSaber - Encaps					
1. MatrixVectorMul	815.40	68.48	1. Hash	44.82	47.49
2. InnerProduct	204.60	17.18	**2.1 MatrixVectorMul**	25.157	26.66
3. GenMatrix	92.58	7.78	**2.2 InnerProduct**		
4. Hash	44.82	3.76	**2.3 GenMatrix**		
5. GenSecret	12.46	1.05	**2.4 GenSecret**		
Total	1,190.70	98.25	Total	94.38	74.15
FireSaber - Encaps without randombytes()					
1. MatrixVectorMul	815.40	68.57	1. Hash	44.82	48.26
2. InnerProduct	204.60	17.20	**2.1 MatrixVectorMul**	25.157	27.09
3. GenMatrix	92.58	7.79	**2.2 InnerProduct**		
4. Hash	44.82	3.77	**2.3 GenMatrix**		
5. GenSecret	12.46	1.05	**2.4 GenSecret**		
Total	1,189.20	98.37	Total	92.88	75.34
FireSaber - Decaps					
1. MatrixVectorMul	815.98	59.29	1. Hash	44.82	51.85
2. InnerProduct x2	408.96	29.72	**2.1 MatrixVectorMul**	37.24	43.09
3. GenMatrix	92.60	6.73	**2.2 InnerProduct x2**		
4. Hash	24.50	1.78	**2.3 GenMatrix**		
5. GenSecret	12.44	0.90	**2.4 GenSecret**		
Total	1,376.14	98.43	Total	86.43	94.94

Table 10: Results of profiling for NTRU

Function	Time [us]	Time [%]	Function	Time [us]	Time [%]
Software			**Software/Hardware**		
NTRU-HPS2048677 - Encaps					
1. poly_Rq_mul	2,693.11	87.84	1. owcpa_samplemsg	217.38	56.30
2. owcpa_samplemsg	217.38	7.09	2. randombytes	112.14	29.05
3. randombytes	112.14	3.66	3. poly_S3_frombytes x2	25.96	6.72
4. poly_S3_frombytes x2	25.96	0.85	**4. poly_Rq_mul**	13.23	3.43
5. sha3_256	10.45	0.34	5. sha3_256	10.45	2.71
Total	3,065.84	99.78	Total	386.08	98.21
NTRU-HPS2048677 - Encaps without randombytes()					
1. poly_Rq_mul	2,693.11	91.18	1. owcpa_samplemsg	217.38	79.35
2. owcpa_samplemsg	217.38	7.36	2. poly_S3_frombytes x2	25.96	9.48
3. poly_S3_frombytes x2	25.96	0.88	**3. poly_Rq_mul**	13.23	4.83
4. sha3_256	10.45	0.35	4. sha3_256	10.45	3.81
Total	2,953.70	99.77	Total	273.94	97.47
NTRU-HPS2048677 - Decaps					
1. poly_S3_mul	2,706.80	33.11	**1.1 poly_Z3_to_Zq/poly_Rq_mul**		
2. poly_Sq_mul	2,693.15	32.94	**1.2 poly_Rq_to_S3/poly_S3_mul**	32.25	28.29
3. poly_Rq_mul	2,693.12	32.94	**1.3 poly_Sq_mul**		
4. poly_S3_frombytes x2	25.86	0.32	2. poly_S3_frombytes x2	25.86	22.68
5. sha3_256	20.50	0.25	3. sha3_256	20.50	17.98
Total	8,174.88	99.57	Total	114.02	68.95
NTRU-HPS4096821 - Encaps					
1. poly_Rq_mul	3,955.45	89.58	1. owcpa_samplemsg	272.44	57.37
2. owcpa_samplemsg	272.44	6.17	2. randombytes	135.61	28.56
3. randombytes	135.61	3.07	3. poly_S3_frombytes x2	31.06	6.54
4. poly_S3_frombytes x2	31.06	0.70	**4. poly_Rq_mul**	14.57	3.07
5. sha3_256	10.66	0.24	5. sha3_256	10.66	2.24
Total	4,415.75	99.76	Total	474.87	97.78
NTRU-HPS4096821 - Encaps without randombytes()					
1. poly_Rq_mul	3,955.45	92.41	1. owcpa_samplemsg	272.44	80.58
2. owcpa_samplemsg	272.44	6.37	2. poly_S3_frombytes x2	31.06	9.19
3. poly_S3_frombytes x2	31.06	0.73	**3. poly_Rq_mul**	14.57	4.31
4. sha3_256	10.66	0.25	4. sha3_256	10.66	3.15
Total	4,280.14	99.75	Total	338.10	97.23
NTRU-HPS4096821 - Decaps					
1. poly_S3_mul	3,972.12	33.15	**1.1 poly_Z3_to_Zq/poly_Rq_mul**		
2. poly_Sq_mul	3,960.27	33.05	**1.2 poly_Rq_to_S3/poly_S3_mul**	37.97	33.92
3. poly_Rq_mul	3,955.44	33.01	**1.3 poly_Sq_mul**		
4. poly_S3_frombytes x2	31.06	0.26	2. poly_S3_frombytes x2	31.06	18.40
5. sha3_256	24.07	0.20	3. sha3_256	24.07	14.26
Total	11,981.69	99.68	Total	111.93	83.18
NTRU-HRSS - Encaps					
1. poly_Rq_mul	2,886.65	94.83	1. randombytes	48.77	28.70
2. randombytes	48.77	1.60	**2. poly_Rq_mul**	13.57	7.99
3. owcpa_samplemsg	32.96	1.08	3. owcpa_samplemsg	32.96	19.40
4. poly_lift	27.89	0.92	4. poly_lift	27.89	16.41
5. poly_S3_frombytes x2	26.83	0.88	5. poly_S3_frombytes x2	26.83	15.79
Total	3,043.87	99.32	Total	169.93	88.29
NTRU-HRSS - Encaps without Randombytes()					
1. poly_Rq_mul	2,886.65	96.38	1. owcpa_samplemsg	32.96	27.20
2. owcpa_samplemsg	32.96	1.10	2. poly_lift	27.89	23.02
3. poly_lift	27.89	0.93	3. poly_S3_frombytes x2	26.83	22.14
4. poly_S3_frombytes x2	26.83	0.90	**4. poly_Rq_mul**	13.76	11.36
Total	2,995.10	99.31	Total	121.16	72.37
NTRU-HRSS - Decaps					
1. poly_S3_mul	2,900.79	33.00	**1.1 poly_Z3_to_Zq/poly_Rq_mul**		
2. poly_Sq_mul	2,890.74	32.89	**1.2 poly_Rq_to_S3/poly_S3_mul** **1.3 poly_Sq_mul**	33.78	26.32
3. poly_Rq_mul	2,886.63	32.84	2. poly_lift	22.33	21.19
4. poly_lift	27.19	0.31	3. sha3_256	13.29	17.40
5. sha3_256	22.33	0.25	4. poly_S3_frombytes	13.29	10.36
Total	8,789.77	99.29	Total	128.33	75.27

Table 11: Results of profiling for NTRU Prime

Function	Time [us]	Time [%]	Function	Time [us]	Time [%]
Software			**Software/Hardware**		
NTRU LPRime - Encaps					
1. Rq_mult_small x2	118,433.60	98.06	1. Hash	1,715.70	71.48
2. Hash	1,715.70	1.42	2. Short_fromlist	343.14	14.30
3. Short_fromlist	343.14	0.28	3. Round	72.44	3.02
4. Round	72.44	0.06	4. Rounded_decode	67.03	2.79
5. Rounded_decode	67.03	0.06	**5. Rq_mult_small x2**	52.33	2.18
Total	120,774.96	99.88	Total	2,400.31	93.76
NTRU LPRime - Encaps without randombytes()					
1.Rq_mult_small x2	118,433.60	98.06	1. Hash	1,715.70	71.52
2. Hash	1,715.70	1.42	2. Short_fromlist	343.14	14.30
3. Short_fromlist	343.14	0.28	3. Round	72.44	3.02
4. Round	72.44	0.06	4. Rounded_decode	67.03	2.79
5. Rounded_decode	67.03	0.06	**5. Rq_mult_small x2**	52.33	2.18
Total	120,773.42	99.88	Total	2,398.77	93.82
NTRU LPRime - Decaps					
1. Rq_mult_small x3	177,650.43	99.07	1. Hash	947.72	53.62
2. Hash	947.72	0.53	2. Short_fromlist	326.38	18.46
3. Short_fromlist	326.38	0.18	3. Rounded_decode x2	134.00	7.58
4. Rounded_decode x2	134.00	0.07	**4. Rq_mult_small x3**	95.92	5.43
5. Round	72.45	0.04	5. Round	72.45	4.10
Total	179,326.77	99.89	Total	1,767.57	89.19
Strl NTRU Prime - Encaps					
1. Rq_mult_small	59,216.81	98.81	1. Randombytes	386.86	53.21
2. Randombytes	386.86	0.65	2. Hash	72.66	9.99
3. Hash	72.66	0.12	3. Round	72.45	9.96
4. Round	72.45	0.12	**4. Rq_mult_small**	43.73	6.01
5. Rounded_encode	14.76	0.02	5. Rounded_encode	14.76	2.03
Total	59,929.86	99.72	Total	727.08	81.21
Strl NTRU Prime - Encaps without randombytes()					
1. Rq_mult_small	59,216.81	99.45	1. Hash	72.66	21.36
2. Hash	72.66	0.12	2. Round	72.45	21.30
3. Round	72.45	0.12	**3. Rq_mult_small**	43.73	12.85
4. Rounded_encode	14.76	0.02	4. Rounded_encode	14.76	4.34
Total	59,543.00	99.72	Total	340.22	59.84
Strl NTRU Prime - Decaps					
1. Rq_mult_small x2	118,433.60	64.33	1. Hash	1,174.55	78.42
2. R3_mult	64,124.99	34.83	**2.1 Rq_mult_small x2**	91.63	6.12
3. Hash	1,174.55	0.64	**2.2 R3_mult**		
4. Round	72.46	0.04	3. Round	72.46	4.84
5. Rounded_decode	67.17	0.04	4. Rounded_decode	67.17	4.48
Total	184,095.24	99.88	Total	1,497.82	93.86

B Pseudocode of investigated algorithms and block diagrams of lower-level operations

Algorithm FrodoKEM.Encaps.

Input: Public key $pk = \mathsf{seed}_{\mathbf{A}} \| \mathbf{b} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}} + D \cdot n \cdot \overline{n}}$.
Output: Ciphertext $\mathbf{c}_1 \| \mathbf{c}_2 \in \{0,1\}^{(\overline{m} \cdot n + \overline{m} \cdot \overline{n}) D}$ and shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len}_{\mathsf{ss}}}$.

1: Choose a uniformly random key $\mu \leftarrow\!\!\$\ U(\{0,1\}^{\mathsf{len}_{\mu}})$
2: Compute $\mathbf{pkh} \leftarrow \text{SHAKE}(pk, \mathsf{len}_{\mathbf{pkh}})$
3: Generate pseudorandom values $\mathsf{seed}_{\mathbf{SE}} \| \mathbf{k} \leftarrow \text{SHAKE}(\mathbf{pkh} \| \mu, \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len}_{\mathbf{k}})$
4: Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n + \overline{m}\overline{n} - 1)}) \leftarrow \text{SHAKE}(\mathtt{0x96} \| \mathsf{seed}_{\mathbf{SE}}, 2\overline{m}n + \overline{m}\overline{n} \cdot \mathsf{len}_{\chi})$
5: Sample error matrix $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\overline{m}n-1)})), \overline{m}, n, T_{\chi})$
6: Sample error matrix $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n-1)})), \overline{m}, n, T_{\chi})$
7: Generate $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
8: Compute $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
9: Compute $\mathbf{c}_1 \leftarrow \mathsf{Frodo.Pack}(\mathbf{B}')$
10: Sample error matrix $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{m}\overline{n}-1)})), \overline{m}, \overline{n}, T_{\chi})$
11: Compute $\mathbf{B} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{b}, n, \overline{n})$
12: Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
13: Compute $\mathbf{C} \leftarrow \mathbf{V} + \mathsf{Frodo.Encode}(\mu)$
14: Compute $\mathbf{c}_2 \leftarrow \mathsf{Frodo.Pack}(\mathbf{C})$
15: Compute $\mathbf{ss} \leftarrow \text{SHAKE}(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{k}, \mathsf{len}_{\mathbf{ss}})$
16: **return** ciphertext $\mathbf{c}_1 \| \mathbf{c}_2$ and shared secret $\mathbf{ss}$

Algorithm FrodoKEM.Decaps.

Input: Ciphertext $\mathbf{c}_1 \| \mathbf{c}_2 \in \{0,1\}^{(\overline{m} \cdot n + \overline{m} \cdot \overline{n}) D}$, secret key $sk' = (\mathbf{s} \| \mathsf{seed}_{\mathbf{A}} \| \mathbf{b}, \mathbf{S}, \mathbf{pkh}) \in \{0,1\}^{\mathsf{len}_{\mathbf{s}} + \mathsf{len}_{\mathsf{seed}_{\mathbf{A}}} + D \cdot n \cdot \overline{n}} \times \mathbb{Z}_q^{n \times \overline{n}} \times \{0,1\}^{\mathsf{len}_{\mathbf{pkh}}}$.
Output: Shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len}_{\mathsf{ss}}}$.

1: $\mathbf{B}' \leftarrow \mathsf{Frodo.Unpack}(\mathbf{c}_1)$
2: $\mathbf{C} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{c}_2)$
3: Compute $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
4: Compute $\mu' \leftarrow \mathsf{Frodo.Decode}(\mathbf{M})$
5: Parse $pk \leftarrow \mathsf{seed}_{\mathbf{A}} \| \mathbf{b}$
6: Generate pseudorandom values $\mathsf{seed}_{\mathbf{SE}}' \| \mathbf{k}' \leftarrow \text{SHAKE}(\mathbf{pkh} \| \mu', \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len}_{\mathbf{k}})$
7: Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n + \overline{m}\overline{n} - 1)}) \leftarrow \text{SHAKE}(\mathtt{0x96} \| \mathsf{seed}_{\mathbf{SE}}', 2\overline{m}n + \overline{m}\overline{n} \cdot \mathsf{len}_{\chi})$
8: Sample error matrix $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\overline{m}n-1)})), \overline{m}, n, T_{\chi})$
9: Sample error matrix $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n-1)})), \overline{m}, n, T_{\chi})$
10: Generate $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
11: Compute $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
12: Sample error matrix $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{m}\overline{n}-1)})), \overline{m}, \overline{n}, T_{\chi})$
13: Compute $\mathbf{B} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{b}, n, \overline{n})$
14: Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
15: Compute $\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Frodo.Encode}(\mu')$
16: **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **then**
17: **return** shared secret $\mathbf{ss} \leftarrow \text{SHAKE}(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{k}', \mathsf{len}_{\mathbf{ss}})$
18: **else**
19: **return** shared secret $\mathbf{ss} \leftarrow \text{SHAKE}(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{s}, \mathsf{len}_{\mathbf{ss}})$

Figure 9: Pseudocode of FrodoKEM [34]

Algorithm r5_cca_kem_encapsulate(pk)

parameters: Integers $p, t, q, n, d, \overline{m}, \overline{n}, \mu, b, \kappa, f, \tau$; $\xi \in \{\Phi_{n+1}(x), x^{n+1}-1\}$
input : $pk \in \{0,1\}^{\kappa} \times \mathcal{R}_{n,p}^{d/n \times \overline{n}}$
output : $ct = (\tilde{\boldsymbol{U}}, \boldsymbol{v}, g) \in \mathcal{R}_{n,p}^{\overline{m} \times d/n} \times \mathbb{Z}_t^{\mu} \times \{0,1\}^{\kappa}, k \in \{0,1\}^{\kappa}$

$m \xleftarrow{\$} \{0,1\}^{\kappa}$
$(L, g, \rho) = G(m||pk)$
$(\tilde{\boldsymbol{U}}, \boldsymbol{v}) = \text{r5\_cpa\_pke\_encrypt}(pk, m, \rho)$
$ct = (\tilde{\boldsymbol{U}}, \boldsymbol{v}, g)$
$k = H(L||ct)$
return (ct, k)

Algorithm r5_cca_kem_decapsulate(ct, sk)

parameters: Integers $p, t, q, n, d, \overline{m}, \overline{n}, \mu, b, \kappa, f, \tau$; $\xi \in \{\Phi_{n+1}(x), x^{n+1}-1\}$
input : $ct = (\tilde{\boldsymbol{U}}, \boldsymbol{v}, g) \in \mathcal{R}_{n,p}^{\overline{m} \times d/n} \times \mathbb{Z}_t^{\mu} \times \{0,1\}^{\kappa}, sk = (sk_{CPA-PKE}, y, pk) \in \{0,1\}^{\kappa} \times \{0,1\}^{\kappa} \times (\{0,1\}^{\kappa} \times \mathcal{R}_{n,p}^{d/n \times \overline{n}})$
output : $k \in \{0,1\}^{\kappa}$

$m' = \text{r5\_cpa\_pke\_decrypt}(sk_{CPA-PKE}, (\tilde{\boldsymbol{U}}, \boldsymbol{v}))$
$(L', g', \rho') = G(m'||pk)$
$(\tilde{\boldsymbol{U'}}, \boldsymbol{v'}) = \text{r5\_cpa\_pke\_encrypt}(pk, m', \rho')$
$ct' = (\tilde{\boldsymbol{U'}}, \boldsymbol{v'}, g')$
if $(ct = ct')$ **then**
    return $k = H(L'||ct)$
else
    return $k = H(y||ct)$
end if

Algorithm r5_cpa_pke_encrypt(pk, m, ρ)

parameters: Integers $p, t, q, n, d, \overline{m}, \overline{n}, \mu, b, \kappa, f, \tau$; $\xi \in \{\Phi_{n+1}(x), x^{n+1}-1\}$
input : $pk = (\sigma, \boldsymbol{B}) \in \{0,1\}^{\kappa} \times \mathcal{R}_{n,p}^{d/n \times \overline{n}}, m, \rho \in \{0,1\}^{\kappa}$
output : $ct = (\tilde{\boldsymbol{U}}, \boldsymbol{v}) \in \mathcal{R}_{n,p}^{\overline{m} \times d/n} \times \mathbb{Z}_t^{\mu}$

$\boldsymbol{A} = \boldsymbol{f}_{d,n}^{(\tau)}(\sigma$,
$\boldsymbol{R} = f_R(\rho)$
$\boldsymbol{U} = \text{R}_{q \to p, h_2}(\langle \boldsymbol{A}^T \boldsymbol{R} \rangle_{\Phi_{n+1}})$
$\tilde{\boldsymbol{U}} = \boldsymbol{U}^T$
$\boldsymbol{v} = \langle \text{R}_{p \to t, h_2}(\text{Sample}_{\mu}(\langle \boldsymbol{B}^T \boldsymbol{R} \rangle_{\xi})) + \frac{t}{b} xef\_compute_{\kappa,f}(m) \rangle_t$
$ct = (\tilde{\boldsymbol{U}}, \boldsymbol{v})$
return ct

Algorithm r5_cpa_pke_decrypt(sk, ct)

parameters: Integers $p, t, q, n, d, \overline{m}, \overline{n}, \mu, b, \kappa, f$; $\xi \in \{\Phi_{n+1}(x), x^{n+1}-1\}$
input : $sk \in \{0,1\}^{\kappa}, ct = (\tilde{\boldsymbol{U}}, \boldsymbol{v}) \in \mathcal{R}_{n,p}^{\overline{m} \times d/n} \times \mathbb{Z}_t^{\mu}$
output : $\hat{m} \in \{0,1\}^{\kappa}$

$\boldsymbol{v}_p = \frac{p}{t} \boldsymbol{v}$
$\boldsymbol{S} = f_s(sk)$
$\boldsymbol{U} = \tilde{\boldsymbol{U}}^T$
$\boldsymbol{y} = \text{R}_{p \to b, h_3}(\boldsymbol{v}_p - \text{Sample}_{\mu}((\boldsymbol{S}^T(\boldsymbol{U} + h_4 \boldsymbol{J}))_{\xi}))$
$\hat{m} = xef\_decode_{\kappa,f}(\boldsymbol{y})$
return $\hat{m}$

Figure 10: Pseudocode of Round5 [37]

Algorithm $\texttt{Saber.KEM.Encaps}(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$

$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
$c = \texttt{Saber.PKE.Enc}(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
return (c, K)

Algorithm $\texttt{Saber.KEM.Decaps}(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$

$m' = \texttt{Saber.PKE.Dec}(\boldsymbol{s}, c)$
$(\hat{K}', r') = \mathcal{G}(pkh, m')$
$c' = \texttt{Saber.PKE.Enc}(pk, m'; r')$
if $c = c'$ **then**
  return $K = \mathcal{H}(\hat{K}', c)$
else
  return $K = \mathcal{H}(z, c)$

Algorithm $\texttt{Saber.PKE.Enc}(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$

$\boldsymbol{A} = \mathsf{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
if *r is not specified* **then**
  $r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
$\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
$v' = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in R_p$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
return $c := (c_m, \boldsymbol{b'})$

Algorithm $\texttt{Saber.PKE.Dec}(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$

$v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
$m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
return m'

Figure 11: Pseudocode of SABER [38]

$\mathsf{KeyGen}'(seed)$	$\mathsf{Encrypt}(\mathbf{h},(\mathbf{r},\mathbf{m}))$	$\mathsf{Decrypt}((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q),\mathbf{c})$
1. $(\mathbf{f},\mathbf{g}) \leftarrow \mathsf{Sample\_fg}(seed)$	1. $\mathbf{m}' \leftarrow \mathsf{Lift}(\mathbf{m})$	1. if $\mathbf{c} \not\equiv 0 \pmod{(q,\Phi_1)}$ return $(0,0,1)$
2. $\mathbf{f}_q \leftarrow (1/\mathbf{f}) \bmod (q,\Phi_n)$	2. $\mathbf{c} \leftarrow (\mathbf{r}\cdot\mathbf{h}+\mathbf{m}') \bmod (q,\Phi_1\Phi_n)$	2. $\mathbf{a} \leftarrow (\mathbf{c}\cdot\mathbf{f}) \bmod (q,\Phi_1\Phi_n)$
3. $\mathbf{h} \leftarrow (3\cdot\mathbf{g}\cdot\mathbf{f}_q) \bmod (q,\Phi_1\Phi_n)$	3. return $\mathbf{c}$	3. $\mathbf{m} \leftarrow (\mathbf{a}\cdot\mathbf{f}_p) \bmod (3,\Phi_n)$
4. $\mathbf{h}_q \leftarrow (1/\mathbf{h}) \bmod (q,\Phi_n)$		4. $\mathbf{m}' \leftarrow \mathsf{Lift}(\mathbf{m})$
5. $\mathbf{f}_p \leftarrow (1/\mathbf{f}) \bmod (3,\Phi_n)$		5. $\mathbf{r} \leftarrow ((\mathbf{c}-\mathbf{m}')\cdot\mathbf{h}_q) \bmod (q,\Phi_n)$
7. return $((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q),\mathbf{h})$		6. if $(\mathbf{r},\mathbf{m}) \in \mathcal{L}_r \times \mathcal{L}_m$ return $(\mathbf{r},\mathbf{m},0)$
		7. else return $(0,0,1)$

The DPKE for the NTRU submission.

$\mathsf{KeyGen}\,(seed)$	$\mathsf{Encapsulate}\,(\mathbf{h})$	$\mathsf{Decapsulate}\,((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q,s),\mathbf{c})$
1. $(\mathbf{f},\mathbf{f}_p,\mathbf{h}_q),\mathbf{h}) \leftarrow \mathsf{KeyGen}'(seed)$	1. $coins \leftarrow_{\$} \{0,1\}^{256}$	1. $(\mathbf{r},\mathbf{m},fail) \leftarrow \mathsf{Decrypt}((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q),\mathbf{c})$
2. $s \leftarrow_{\$} \{0,1\}^{256}$	2. $(\mathbf{r},\mathbf{m}) \leftarrow \mathsf{Sample\_rm}(coins)$	2. $k_1 \leftarrow H_1(\mathbf{r},\mathbf{m})$
7. return $((\mathbf{f},\mathbf{f}_p,\mathbf{h}_q,s),\mathbf{h})$	3. $\mathbf{c} \leftarrow \mathsf{Encrypt}(\mathbf{h},(\mathbf{r},\mathbf{m}))$	3. $k_2 \leftarrow H_2(s,\mathbf{c})$
	4. $k \leftarrow H_1(\mathbf{r},\mathbf{m})$	4. if $fail = 0$ return k_1
	5. return $(\mathbf{c},k)$	5. else return k_2

The KEM for the NTRU submission.

Figure 12: Pseudocode of NTRU [36]

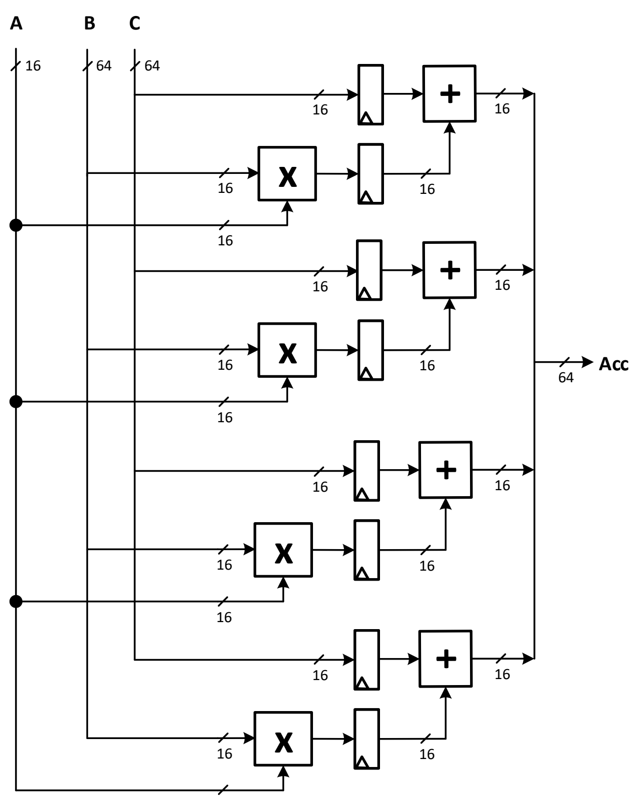


Figure 13: Block diagram of the unit 4 MACs used in FrodoKEM.

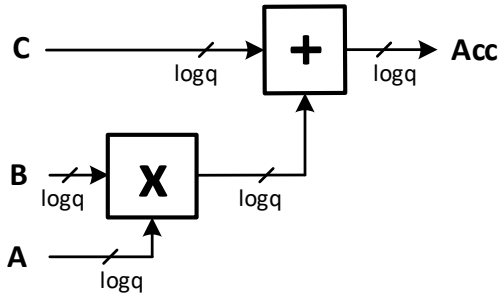


Figure 14: MAC used in Saber, NTRU-HPS, and NTRU-HRSS. logq=13 for Saber and NTRU-HRSS, logq=11 for NTRU-HPS with the security category 1 (ntruhps2048677) and 12 for NTRU-HPS with the security category 3 (ntruhps4096821).

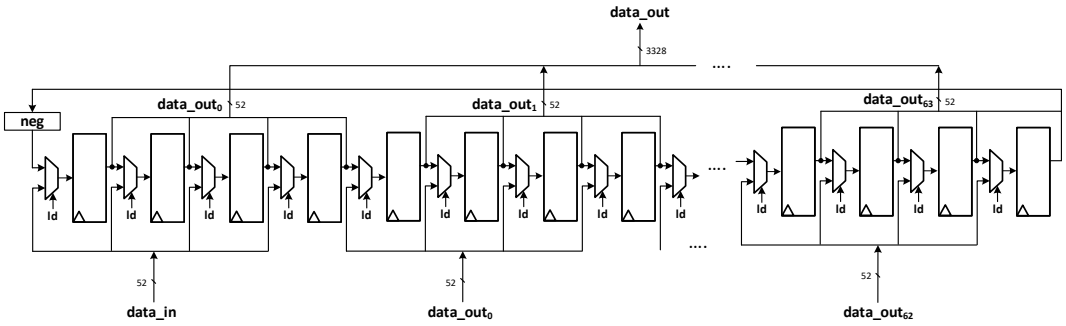
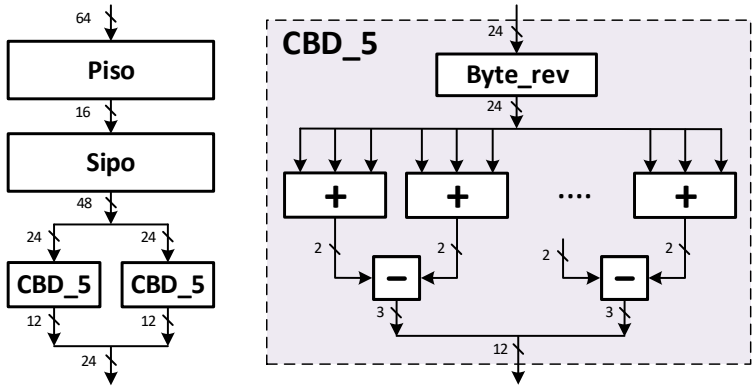
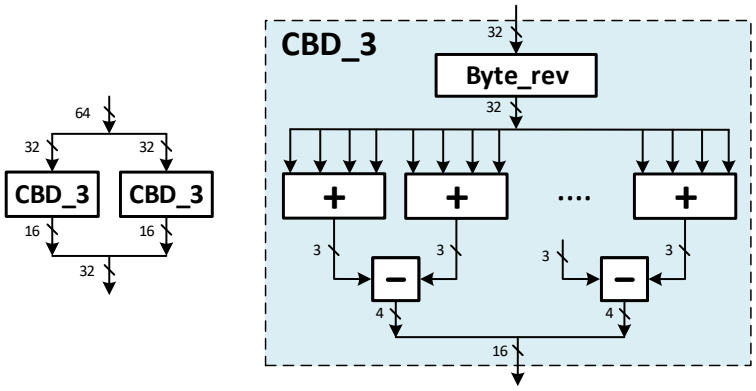


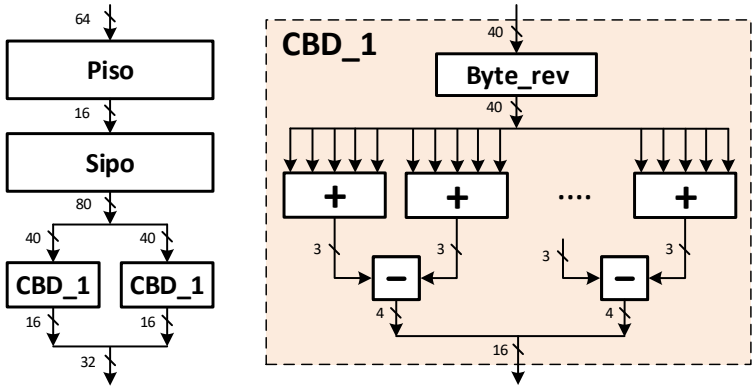
Figure 15: Block diagram of Saber LFSR. All buses are 13-bit wide unless specified.



(a) Security Level 5.

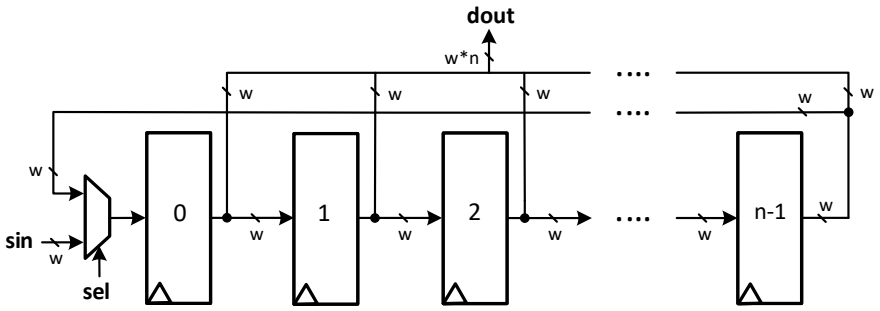


(b) Security Level 3.

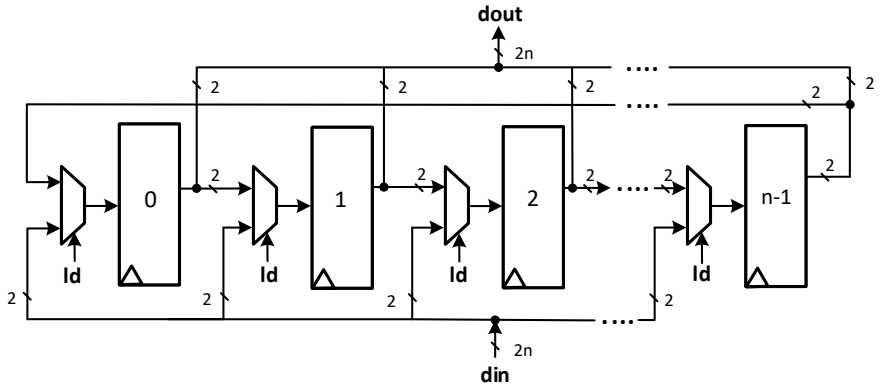


(c) Security Level 1.

Figure 16: Block diagrams for the centered binomial distribution (CBD) samplers of Saber for three different security levels.

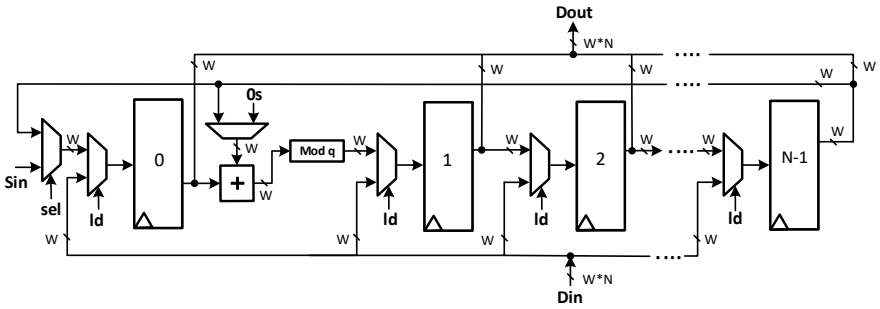


(a) Zq_LFSR.

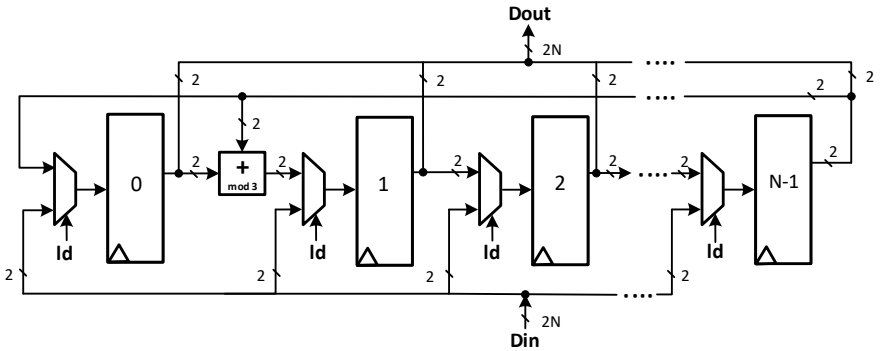


(b) Z3_LFSR

Figure 17: Block diagrams of LFSRs used in NTRU-HPS and NTRU-HRSS.



(a) Zq_LFSR.



(b) Z3_LFSR

Figure 18: Block diagrams of LFSRs used in NTRU LPrime and Streamlined NTRU Prime .