High-Speed Hardware Architectures and Fair FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber

Viet Ba Dang, Kamyar Mohajerani and Kris Gaj

Cryptographic Engineering Research Group,
George Mason University
Fairfax, VA, U.S.A.

Abstract. Performance in hardware has typically played a significant role in differentiating among leading candidates in cryptographic standardization efforts. Winners of two past NIST cryptographic contests (Rijndael in case of AES and Keccak in case of SHA-3) were ranked consistently among the two fastest candidates when implemented using FPGAs and ASICs. Hardware implementations of cryptographic operations may quite easily outperform software implementations for at least a subset of major performance metrics, such as latency, number of operations per second, power consumption, and energy usage, as well as in terms of security against physical attacks, including side-channel analysis. Using hardware also permits much higher flexibility in trading one subset of these properties for another. This paper presents high-speed hardware architectures for four lattice-based CCA-secure Key Encapsulation Mechanisms (KEMs), representing three NIST PQC finalists: CRYSTALS-Kyber, NTRU (with two distinct variants, NTRU-HPS and NTRU-HRSS), and Saber. We rank these candidates among each other and compare them with all other Round 3 KEMs based on the data from the previously reported work.

Keywords: Post-Quantum Cryptography · lattice-based · Key Encapsulation Mechanism · hardware · FPGA

1 Introduction

Post-Quantum Cryptography (PQC) refers to a class of cryptographic algorithms that are resistant against all known attacks using quantum computers, and can be implemented on traditional non-quantum computing platforms. These platforms include microprocessors, microcontrollers, graphics processing units (GPUs), Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and many others. The main goal of PQC is to replace the existing public-key cryptography standards based on RSA and Elliptic Curve Cryptography. These standards seem to be the most vulnerable to quantum computing and impossible to defend using traditional approaches such as gradually increasing key sizes [73, 16, 76, 36].

To initiate a timely transition to a new class of cryptographic schemes, in December 2016, NIST launched its PQC standardization process with the release of a "Call for Proposals and Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms" [63]. Sixty-nine submissions were judged complete and accepted for Round 1, which started in December 2017 [65, 1]. In January 2019, based on the initial security analysis and preliminary software benchmarking results, 26 submissions were qualified by NIST to Round 2. These submissions included multiple public-key encryption, key

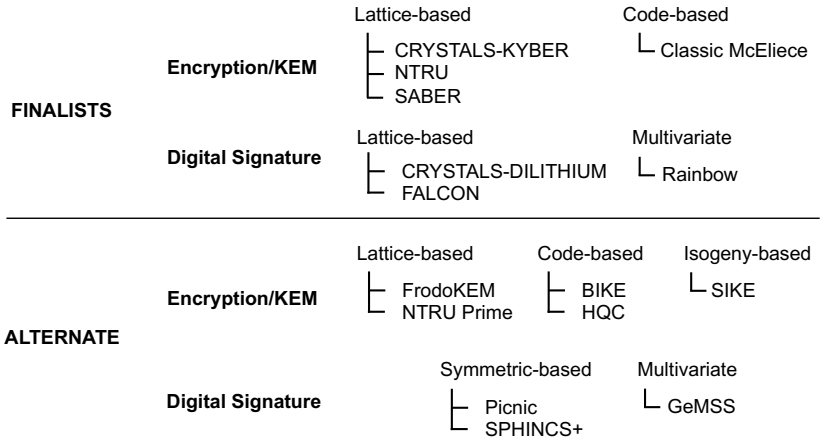

Figure 1: Finalists and alternate candidates qualified to Round 3 of the NIST PQC Standardization Process

encapsulation mechanism (KEM), and digital signature schemes, with many submissions describing more than one algorithm [66].

On July 22, 2020, NIST announced 15 candidates qualified for Round 3 of the standardization process. These candidates are summarized in Fig. 1. All Round 3 candidates represent five diverse families: lattice-based, code-based, multivariate, symmetric-based, and isogeny-based. Seven finalists are expected to be given priority in the standardization process. One encryption/KEM scheme and one digital signature scheme from this group may be selected as a PQC standard as early as 2022. Alternate candidates are treated as backup candidates. In Round 2, alternate candidates were judged to be either insufficiently investigated from the security point of view or were believed to lack some desired properties related to their performance (such as small public keys, small signatures, short execution time in software, etc.). In this paper, we focus on evaluating and contrasting the hardware efficiency of three lattice-based KEMs: CRYSTALS-KYBER, NTRU, and Saber.

There are multiple reasons for focusing our attention on the three candidates mentioned above. In Fig. 2, we show the relationship between the ciphertext and public-key sizes of all Round 3 candidates. All schemes based on structured lattices - Saber, CRYSTALS-KYBER, NTRU Prime, and NTRU - have their ciphertext and public key sizes in the range between 512 and 2048 bytes. The only candidate better than them is an isogeny-based SIKE, which is still considered relatively recent and not sufficiently scrutinized from the security point of view. As a result, this scheme was qualified for Round 3 only as an alternate candidate. The only other PKE/KEM finalist, Classic McEliece, has public-key lengths between 0.25 and 2 Megabytes. So large public key sizes may significantly impact the sizes of data exchanged between two parties in the key establishment phase of any modern secure communication protocol, such as TLS, IPSec, SSH, etc. The sizes of keys and ciphertexts used by the selected lattice-based schemes are significantly smaller than those of the alternate code-based schemes, BIKE and HQC, and the unstructured-lattice scheme FrodoKEM. As a result, the key establishment time and the amount of memory required to store public-key certificates are also the most practical among all Round 3 candidates other than SIKE.

Hardware benchmarking has played a major role in all recent cryptographic standardization efforts, such as the AES, eSTREAM, SHA-3 [10, 34, 50, 51], and CAESAR contests [19, 22]. With the emergence of commonly-accepted hardware application programming interfaces (APIs) [41], development packages [40, 44], specialized optimization tools [35, 29], new design methodologies based on High-Level Synthesis (HLS) [42, 43],

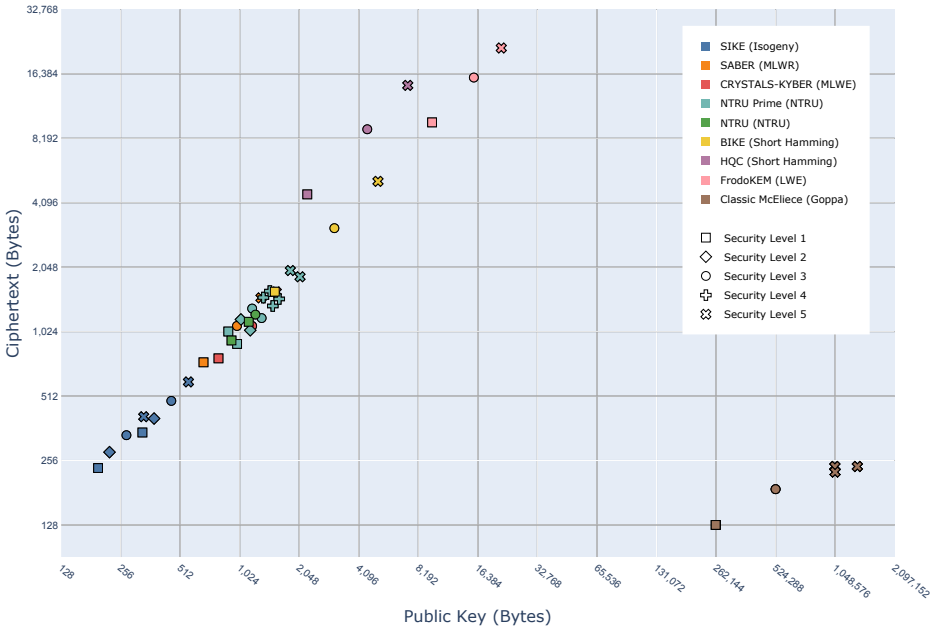

Figure 2: Relation between the ciphertext and public-key sizes for Round 3 PQC Key Encapsulation Mechanisms

and mandatory hardware implementations in the final round of the CAESAR contest [19], the percentage of initial submissions implemented in hardware grew from 27.5% in the SHA-3 contest [33] to 49.1% in the CAESAR competition [22, 32]. In Round 2, all AES, all SHA-3, and all but one CAESAR candidates had at least one hardware implementation reported by the end of the evaluation process. Unfortunately, this trend could not be sustained in the NIST PQC standardization process. In many respects, PQC schemes are diametrically different and at least an order of magnitude more complex to implement compared to those evaluated in previous cryptographic contests.

High-speed vs. lightweight. Assuming comparable technology, hardware implementations outperform software implementations using at least one, and typically multiple, metrics, such as latency, number of operations per second, power consumption, energy usage, and security against physical attacks. They also allow much higher flexibility in trading one subset of these metrics for another. From the point of view of benchmarking and ranking of candidates, such flexibility may become a curse, especially considering that no two metrics are likely to have a simple linear dependence on each other. A practical solution to this problem is to focus during the evaluation process on two major types of implementations: high-speed and lightweight.

In high-speed implementations, the primary target is speed, understood as either minimum latency (a.k.a. execution time) or the number of operations per second. For PQC schemes, this target amounts to optimizing the implementations of major operations involving the public and private key, respectively. For Key Encapsulation Mechanisms (KEMs), these operations are encapsulation and decapsulation; for digital signature schemes, signature verification and generation; for public-key encryption (PKE), encryption and decryption. The time of key generation may also play a major role in the case when a public-private key pair cannot be reused for security reasons. The resource utilization is secondary. Still, hardware designers typically aim at achieving the Pareto optimality, in which any further speed improvement comes at a disproportionate cost in terms of resource utilization. The primary advantage of high-speed implementations is that they

reveal the inherent potential of a given algorithm for parallelization. As long as the resource-utilization limit is sufficiently high, this limit does not affect the ranking of algorithms in terms of latency. Consequently, this ranking is strongly correlated with algorithms' features and is not substantially influenced by any additional assumptions and technology choices. Additionally, only high-speed hardware implementations may effectively compete with optimized software implementations targeting high-performance processors with vector instructions (e.g., AVX2).

In lightweight implementations, the primary targets are typically minimum resource utilization and minimum power consumption, assuming that the execution time does not exceed a predefined maximum. Another way of formulating the goal is to achieve minimum execution time, assuming a given maximum budget in terms of resource utilization, power consumption, or energy usage. The maximum budget on resource utilization is related to the cost of implementation; the budget on power assures correct operation without overheating or devoting additional resources to cooling. The maximum energy usage affects how long a battery-operated device can function before the next battery recharge. In the context of the standardization process for cryptographic algorithms, the mentioned above maximum budgets are very hard to select. Any change in these thresholds may favor a different subset of candidates. With new standards remaining in use for decades, the timing, cost, and power requirements of new and emerging applications are very challenging to predict.

Additionally, changes in technology significantly affect which hardware architectures meet particular constraints. For example, an architecture capable of accomplishing the execution time of 0.1 seconds (or below) under a certain power or energy budget may substantially change with the improvements in technology. As a result, most current limits are selected arbitrarily by different designers or left undefined in their reports. Consequently, the ranking of PQC candidates based on their lightweight implementations, especially those developed by different groups, is extremely challenging and assumption-dependent. These rankings have little to do with the parallelization allowed by each algorithm, as most of the operations must be executed sequentially due to the small resource budget. The primary feature of algorithms these implementations reveal is the number and complexity of its distinct elementary operations. Each major operation infers an additional functional unit, increasing resource utilization and power consumption. Additionally, lightweight hardware implementations can outperform only software implementations targeting specific low-cost, low-power embedded processors, such as Cortex-M4.

In the case of FPGA implementations, resource utilization is a vector, such as (#LUTs, #flip-flops, #DSP units, #BRAMs). No single element of this vector can be expressed in terms of other elements. As a result, imposing a resource limit implies specifying the values of all components of this resource vector. One possible approach may be to choose the resources of the smallest FPGA of a given low-cost FPGA family. However, FPGA families and their resources change over time, so this limit has only a physical meaning during the limited time, covering the evaluation period, and may lose its significance just a few years after the standard is published and deployed. Finally, the same FPGA device may also need to accommodate any overhead associated with countermeasures against side-channel attacks. At the same time, this overhead or even effective countermeasures may remain unknown at the time of the candidates' evaluation.

As a result, in this paper, we focus on developing, benchmarking, and ranking high-speed implementations.

Choice of Algorithms to Implement. In terms of algorithms, we focus on KEMs with indistinguishability under chosen-ciphertext attack (IND-CCA). Our primary goal was to implement all lattice-based IND-CCA secure KEMs described in the specifications of PQC finalists. The submission package of NTRU describes two substantially different KEMs : NTRU-HRSS and NTRU-HPS. As a result, we have implemented four KEMs

representing three PQC finalists. For each implemented KEM, we generated results for all supported security levels.

2 Previous Work

Hardware and software/hardware implementations of all KEMs qualified to Round 3 of the NIST PQC Standardization Process are summarized in Table 1. The PQC candidates are grouped by family. All implementations are classified as either High-Speed or Lightweight. However, the dividing line is not always very clear, and, in multiple cases, the authors have not used these terms explicitly by themselves.

HLS-based implementations are distinguished with the superscript^{H}. These implementations were reported in only one paper [15]. They have been shown to give substantially different results than implementations developed using traditional Register-Transfer Level (RTL) methodology, in which HDL code is developed manually. Therefore, in this paper, we focus on implementations in which a hardware part of the design was developed using traditional RTL methodology.

NTRU Prime is the only Round 3 KEM that does not have any high-speed implementation reported to date. NTRU (as specified in Rounds 2 and 3 of the NIST process) and HQC have no RTL implementations. Additionally, NTRU and all code-based KEMs have not been reported to be implemented using the lightweight approach.

In Tables 2–5, we summarize major results for hardware and software/hardware implementations of KEMs. Most of the implemented schemes are KEMs with indistinguishability under the chosen-ciphertext attack (IND-CCA). Some are PKEs with indistinguishability under the chosen-plaintext attack (IND-CPA). If an IND-CPA-secure PKE is reported, this fact is marked with a superscript ^{cpa}. All mentioned above tables have the same fields. The first two columns contain a reference to the publication and the name of the algorithm variant, respectively. The superscript ^{z} next to the publication reference indicates the implementation using Zynq-7000 SoC FPGA. The implementations targeting Artix-7 and Zynq-7000 are grouped together because the programmable logic of both families is realized using the same technological process and composed of the same basic building blocks.

Table 1: Reported Hardware Implementations of KEMs qualified to Round 3

Algorithms	High-Speed	Lightweight
Lattice-based		
CRYSTALS-KYBER	[15]^{H},[80], [47], [81]	[11], [12]*, [3], [30]
FrodoKEM	[45], [15]^{H}, [24]	[11], [12]*
NTRU	[15]^{H}	–
NTRU Prime	–	[58]
Saber	[15]^{H}, [24], [57], [74], [83]	[30]
Isogeny-based		
SIKE	[53], [60], [27]	[60]
Code-based		
BIKE	[46], [67], [68]	–
Classic McEliece	[79],[78], [15]^{H}	–
HQC	[69]^{H}	–

^{H} design developed using the High-Level Synthesis (HLS) approach
* extended version of [11]

Table 2: Level I KEMs and PKEs on Artix-7 (default) and Zynq-7000 (indicated with the superscript ^{z})

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BRAM	Key Generation		Encaps./Enc.^{cpa}		Decaps./(Dec.+Enc.)^{cpa}	
										cycles	μs	cycles	μs	cycles	μs
Security Level 1															
[81]	Kyber512	HW	HS	161	7,412	4,644	2,126	2	3.0	3,800	24.3	5,100	30.5	6,700	41.3
[78]	mceliece348864^{cpa}	HW	HS	106	81,339	132,190	–	0	236.0	202,787	1,920.3	2,720	25.8	12,743	120.7
[78]	mceliece348864^{cpa}	HW	LW	108	25,327	49,383	–	0	168.0	1,599,882	14,800.0	2,720	25.2	18,358	169.8
[30]^{z}	Kyber512	SW/HW^{RV}	LW	–	23,925	10,844	–	21	32.0	150,106	–	193,076	–	204,843	–
[45]	FrodoKEM-640 16x	HW	HS	172	2,587	2,994	855	16	0	204,766	1,190.5	–	–	–	–
				171	5,796	4,694	1,692	16	0	–	–	207,269	1,212.1	–	–
				149	6,881	5,081	1,947	16	12.5	–	–	–	–	209,867	1,408.5
[69]^{H}	HQC-I	HW	HS	180	15029	11028	5295	0	28.5	59,485	330.0	158,251	870.0	265,836	1,460.0
[12]	Kyber512	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	54,861	2,194.4	134,965	5,398.6	146,068	5,842.7
[53]	SIKEp434	HW	HS	132	21,946	24,328	8,006	240	26.5	530,000	4,009.1	930,000	7,034.8	980,000	7,413.0
[53]	SIKEp503	HW	HS	130	24,610	27,759	9,186	264	33.5	640,000	4,926.9	1,140,000	8,776.0	1,200,000	9,237.9
[30]^{z}	LightSaber	SW/HW^{RV}	LW	–	23,925	10,844	–	21	32.0	366, 837	–	526, 496	–	657,583	–
[3]	Kyber512	SW/HW^{RV}	LW	59	1,842	1,634	–	5	34.0	710,000	11,993.2	971,000	16,402.0	870,000	14,695.9
[60]	SIKEp434	SW/HW^{c}	HS	162	22,595	11,558	7,491	162	37.0	1,474,200	9100	2,494,800	15,400.0	2,656,800	16,400.0
[60]	SIKEp503	SW/HW^{c}	HS	162	22,595	11,558	7,491	162	37.0	1,733,400	10,700.0	2,932,200	18,100.0	3,126,600	19,300.0
[45]	FrodoKEM-640 1x	HW	LW	191	971	433	290	1	0	3,237,288	16,949.2	–	–	–	–
				190	4,246	2,131	1,180	1	0	–	–	3,275,862	17,241.4	–	–
				162	4,446	2,152	1,254	1	12.5	–	–	–	–	3,306,122	20,408.2
[60]	SIKEp434	SW/HW^{c}	LW	143	10,976	7,115	3,512	57	21.0	2,187,902	15,300.0	3,718,004	26,000.0	3,946,804	27,600.0
[60]	SIKEp503	SW/HW^{c}	LW	143	10,976	7,115	3,512	57	21.0	2,602,603	18,200.0	4,390,104	30,700.0	4,676,105	32,700.0
[68]	BIKE Level 1	HW	LW	121	10,702	4,940	3,334	7	15.0	2.671,000	21,903.0	153,000	1,252.0	13,120,000	107,580.0
[68]	BIKE Level 1	HW	HS	96	29,448	5,498	8,419	7	28.0	259,000	2,691.0	12,000	127.0	13,120,000	136,443.0
[12]	FrodoKEM-640	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	11,453,942	458,157.7	11,609,668	464,386.7	12,035,513	481,420.5
[46]	BIKE-1 Level 1^{cs}	HW	HS	165	1,907	1,049	608	0	7.0	95,500	578.0	–	–	–	–
[46]	BIKE-3 Level 1^{cs}	HW	HS	170	1,397	925	453	0	4.0	98,500	579.0	–	–	–	–
[46]	BIKE-2 Level 1^{cs}	HW	HS	160	3,874	2,141	1,312	0	10.0	2,150,000	13,437.0	–	–	–	–

^{z} Design implemented on Zynq-7000

^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)

^{cs} Designs for the variants BIKE-1, BIKE-2, and BIKE-3 consolidated by submitters to BIKE on May 3, 2020

^{RV} co-design using RISC-V RV32IM

^{c} co-design using a custom processor

* Preliminary result

Table 3: Level 3 & 5 KEMs and PKEs on Artix-7 (default) and Zynq-7000 (indicated with the superscript ^{z})

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BRAM	Key Generation		Encaps./Enc.^{cpa}		Decaps./(Dec.+Enc.)^{cpa}	
										cycles	μs	cycles	μs	cycles	μs
Security Level 3															
[81]	Kyber768	HW	HS	161	7,412	4,644	2,126	2	3.0	6,300	39.2	7,900	47.6	10,000	62.3
[78]	mceliece460896^{cpa}	HW	LW	107	38,669	74,858	-	0	303.0	5,002,044	46,704.4	3,360	31.4	31,005	289.5
[45]	FrodoKEM-976 16x	HW	HS	169	2,869	3,000	908	16	0	476,05	2,816.9	–	–	–	–
				168	6,188	4,678	1782	16	0	–	–	479,993	2,857.1	–	–
				157	7,213	5,087	2042	16	19.0	–	–	–	–	483,073	3,076.9
[57]^{z}	Saber	SW/HW^{A9}	HS	125	7,400	7,331	-	28	2.0	–	3,273.0	–	4,147.0	–	3,844.0
[12]	Kyber768	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	84,110	3,364.4	184,080	7,363.2	198,011	7,920.4
[53]	SIKEp610	HW	HS	125	29,447	33,198	10,843	312	39.5	900,000	7,182.8	1,810,000	14,445.3	1,780,000	14,205.9
[60]	SIKEp610	SW/HW^{c}	HS	162	22,595	11,558	7,491	162	37.0	2,916,000	18,000.0	5,443,200	33,600.0	5,508,000	34,000.0
[45]	FrodoKEM-976 1x	HW	LW	189	1,243	441	362	1	0	7,560,000	40,000.0	–	–	–	–
				187	4,650	2,118	1,272	1	0	–	–	7,480,000	40,000.0	–	–
				162	4,888	2,153	1,390	1	19.0	–	–	–	–	7,714,286	47,619.0
[60]	SIKEp610	SW/HW^{c}	LW	143	10,976	7,115	3,512	57	21.0	4,347,204	30,400.0	8,108,108	56,700.0	8,208,208	57,400.0
[68]	BIKE Level 3	HW	LW	122	9,808	5,075	2,996	7	23.0	11,600,207	95,122.6	601,099	4,929.1	37,596,111	308,291.2
[68]	BIKE Level 3	HW	HS	96	28,784	5,553	8,184	7	33.0	930,179	9,674.2	42,162	438.5	37,596,111	391,015.2
[12]	FrodoKEM-976	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	26,005,326	1,040,213.0	29,749,417	1,189,976.7	30,421,175	1,216,847.0
Security Level 5															
[81]	Kyber1024	HW	HS	161	7,412	4,644	2,126	2	3.0	9,400	58.2	11,300	67.9	13,900	86.2
[12]	Kyber1024	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	116,841	4,673.6	236,886	9,475.4	256,828	10,273.1
[30]^{z}	Kyber1024	SW/HW	LW	–	23,925	10,844	–	21	32.0	349,673	–	405,477	–	424,682	–
[53]	SIKEp751	HW	HS	127	40,792	49,982	15,794	512	43.5	1,250,000	9,842.5	2,210,000	17,401.6	2,340,000	18,425.2
[30]^{z}	FireSaber	SW/HW	LW	–	23,925	10,844	–	21	32.0	1,300,272	–	1,622,818	–	1,898,051	–
[3]	Kyber1024	SW/HW^{RV}	LW	59	1,842	1,634	–	5	34.0	2,203,000	37,212.8	2,619,000	44,239.9	2,429,000	41,030.4
[60]	SIKEp751	SW/HW^{c}	HS	162	22,595	11,558	7,491	162	37.0	3,742,200	23,100.0	6,188,400	38,200.0	6,658,200	41,100.0
[60]	SIKEp751	SW/HW^{c}	LW	143	10,976	7,115	3,512	57	21.0	7,965,108	55,700.0	13,156,013	92,000.0	14,185,614	99,200.0
[12]	FrodoKEM-1344	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	67,994,170	2,719,766.8	71,501,358	2,860,054.3	72,526,695	2,901,067.8

^{z} Design implemented on Zynq-7000
^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)
^{RV} co-design using RISC-V RV32IM
^{c} co-design using a custom processor
^{A9} co-design using ARM Cortex-A9
* Preliminary result

Table 4: KEMs on Virtex-7

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BR AM	Key Generation		Encap./Enc.[cpa]		Decap./Dec.[cpa]	
										cycles	μs	cycles	μs	cycles	μs
Security Level 1															
[27]	SIKEp434	HW	HS	250	12,818	18,271	5,527	195	32.0	–	–	1,095,000	4,400.0	1,095,000	4,400.0
[27]	SIKEp503	HW	HS	244	13,963	19,935	6,163	225	34.0	–	–	1,440,000	5,900.0	1,440,000	5,900.0
[53]	SIKEp434	HW	HS	168	21,059	23,819	8,121	240	26.5	530,000	3,147.3	930,000	5,522.6	980,000	5,819.5
[53]	SIKEp503	HW	HS	166	23,746	27,609	8,907	264	33.5	640,000	3,857.7	1,140,000	6,871.6	1,200,000	7,233.3
[60]	SIKEp434	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	981,180	6,900.0	1,677,960	11,800.0	1,777,500	12,500.0
[60]	SIKEp503	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	1,166,040	8,200.0	1,976,580	13,900.0	2,104,560	14,800.0
[60]	SIKEp434	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	2,191,781	14,400.0	3,713,851	24,400.0	3,957,382	26,000.0
[60]	SIKEp503	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	2,602,740	17,100.0	4,383,562	28,800.0	4,672,755	30,700.0
Security Level 3															
[78]	mceliece460896[cpa]	HW	HS	131	109,484	168,939	-	0	446.0	515,806	3,943.5	3,360	25.7	17,931	137.1
[27]	SIKEp610	HW	HS	239	16,226	26,757	7,461	270	38.5	–	–	2,280,000	9,550.0	2,280,000	9,550.0
[53]	SIKEp610	HW	HS	166	28,217	33,297	10,675	312	39.5	900,000	5,428.2	1,810,000	10,916.8	1,780,000	10,735.8
[60]	SIKEp610	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	1,962,360	13,800.0	3,654,540	25,700.0	3,711,420	26,100.0
[60]	SIKEp610	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	4,353,120	28,600.0	8,097,412	53,200.0	8,219,178	54,000.0
Security Level 5															
[78]	mceliece6960119[cpa]	HW	HS	130	116,928	188,324	-	0	607.0	974,306	7,500.4	5,413	41.7	25,135	193.5
[78]	mceliece6688128[cpa]	HW	HS	137	122,624	186,194	-	0	589.0	1,046,139	7,658.4	5,024	36.8	29,754	217.8
[78]	mceliece8192128[cpa]	HW	HS	130	123,361	190,707	-	0	589.0	1,286,179	9,901.3	6,528	50.3	32,765	252.2
[78]	mceliece6960119[cpa]	HW	LW	141	44,154	88,963	-	0	563.0	11,179,636	79,570.4	5,413	38.5	46,141	328.4
[78]	mceliece6688128[cpa]	HW	LW	136	44,345	83,637	-	0	446.0	12,389,742	91,034.1	5,024	36.9	52,333	384.5
[78]	mceliece8192128[cpa]	HW	LW	134	45,150	88,154	-	0	525.0	15,185,314	113,154.4	6,528	48.6	55,330	412.3
[27]	SIKEp751	HW	HS	233	20,207	39,339	11,136	452	41.5	–	–	2,965,000	12,750.0	2,965,000	12,750.0
[53]	SIKEp751	HW	HS	163	39,953	50,079	15,834	512	43.5	1,250,000	7,664.0	2,210,000	13,550.0	2,340,000	14,347.0
[60]	SIKEp751	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	2,516,940	17,700.0	4,166,460	29,300.0	4,479,300	31,500.0
[60]	SIKEp751	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	7,960,426	52,300.0	13,150,685	86,400.0	14,185,693	93,200.0

[cpa] Design of a KEM variant resistant against Chosen-Plaintext Attack (CPA)

Table 5: All KEMs and PKEs on Zynq Ultrascale+

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BRAM	Key Gen.		Encap./Enc.^{cpa}		Decap./(Dec.+Enc.)^{cpa}	
										cycles	us	cycles	us	cycles	us
Security Level 1															
[83]^{cpa}	LightSaber	HW	HS	100	34,886	9,858	–	85	6.0	519	5.1	664	6.6	990	9.9
[24]	LightSaber	SW/HW	HS	322	12,343	11,288	1,989	256	3.5	–	–	–	53.0	–	56.0
[24]	FrodoKEM-640	SW/HW	HS	402	7,213	6,647	1,186	32	13.5	–	–	–	1,223.0	–	1,319.0
Security Level 3															
[83]^{cpa}	Saber	HW	HS	100	34,886	9,858	–	85	6.0	1,039^{a}	10.3^{a}	1,396^{a}	14.0^{a}	1,684^{a}	16.8^{a}
[74]	Saber	HW	HS	250	45,895	18,705	–	0	2	4,320	17.3	5,231	20.9	6,461	25.8
[74]	Saber	HW	HS	250	25,079	10,750	–	0	2	5,435	21.8	6,618	26.5	8,034	32.1
[24]	Saber	SW/HW	HS	322	12,566	11,619	1,993	256	3.5	–	–	–	60.0	–	65.0
[58]	StrNTRUPrime761	HW	LW	271	9,538	7,803	1,841	19	14.0	1,302,968	4,808.0	142,004	524.0	259,618	958.0
[24]	FrodoKEM-976	SW/HW	HS	402	7087	6693	1190	32	17	–	–	–	1,642.0	–	1,866.0
Security Level 5															
[83]^{cpa}	FireSaber	HW	HS	100	34,886	9,858	–	85	6.0	1,513	15.1	1,811	18.1	2,301	23.0
[24]	FireSaber	SW/HW	HS	322	12,555	11,881	2,341	256	3.5	–	–	–	74.0	–	80.0
[24]	FrodoKEM-1344	SW/HW	HS	417	7,015	6,610	1,215	32	17.5	–	–	–	2,186.0	–	3,120.0

All SW/HW co-designs using ARM Cortex-A53

^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)

^{a} The results are estimated through the existing PKE results and additional hash functions.

The type of implementation is indicated in the third column, with HW standing for hardware and SW/HW standing for software/hardware. Among the software/hardware implementations, we specify the embedded processors used with the following notation: ^{RV} represents a RISC-V processor with the RV32IM ISA, i.e., RISC-V with the base 32-bit integer ISA and the standard Integer Multiplication and Division extension. ^{c} represents a custom processor, and ^{A9} a hard processor of the Zynq-7000 SoC FPGA family, namely ARM Cortex-A9. Unlike the first two options, this processor operates with a frequency significantly higher than the maximum clock frequency of programmable logic. At the same time, the transfer of control and data between the processor and the hardware accelerator contributes a non-negligible transfer overhead to all reported execution times.

The next column, Max. Freq., corresponds to the maximum clock frequency in MHz. The next five columns are used to report FPGA resource utilization, described as a vector (LUT, FF, Slice, DSP, BRAM), where the subsequent fields represent the number of look-up tables, flip-flops, slices, DSP units, and 36 kbit Block RAMs. For the last of these values, BRAM, 0.5 represents the use of an 18-kbit block RAM. In the case of KEMs, the remaining six columns are used to show the execution time of Key Generation, Encapsulation, and Decapsulation, expressed in clock cycles and μs, respectively. In the cases when only results for the IND-CPA PKE are reported, the last two columns represent the sum of the execution times of Encryption and Decryption. This convention is used because the most popular transformations between an IND-CPA-secure PKE and the corresponding IND-CCA-secure KEM involve both the Decryption and Encryption operations on the receiver's side. Additionally, these two operations dominate the total Decapsulation time. For all execution times, the value in μs can be obtained by dividing the corresponding number of clock cycles by the maximum clock frequency in MHz.

In Tables 2 and 3, we summarize implementations targeting Xilinx Artix-7 FPGAs and related Xilinx Zynq-7000 SoC FPGAs. For security level 1, five candidates - Classic McEliece, CRYSTALS-Kyber, FrodoKEM, SIKE, and Saber - have implementations of all three operations reported. The preliminary implementations of BIKE focused on key generation only [46, 4]. For security level 5, the results are missing for Classic McEliece.

For most KEMs, the time of decapsulation is longer than the time of encapsulation. Table entries are ordered according to the time of decapsulation in μs (and, if needed, according to the decapsulation time in clock cycles).

The ranking of candidates listed in Tables 2 and 3 is very challenging to determine based on available results. First, it may be unfair to compare pure hardware implementations with software/hardware implementations. Secondly, it is hard to compare lightweight implementations with high-speed implementations, as they are optimized with different primary metrics in mind. Third, software/hardware implementations based on different processors are very challenging to compare with one another. Finally, even for implementations using exactly the same type of implementation (software/hardware) and the same type of processor (RISC-V), such as those reported in [30], the comparison may be unintentionally biased. In the specific case of [30], significantly different hardware support was provided for algorithms that can take advantage of the Number Theoretic Transform - Kyber and NewHope - vs. the algorithm that cannot - Saber. An additional, relatively minor factor is that several results for Classic McEliece concern their IND-CPA-secure PKEs rather than IND-CCA-secure KEMs.

Taking all these factors into account, almost the only ranking that is quite clear from Tables 2 and 3 is the ranking of candidates that have results available for pure hardware implementations, developed using the RTL methodology, targeting high-speed. In this specific category, the ranking for security level 1 is: *1.* Kyber, *2.* Classic McEliece, *3.* FrodoKEM, *4.* SIKE, and *5.* BIKE. At level 3, the ranking remains the same, even though the implementation of Classic McEliece is parameterized for low resource usage. At level 5, only Kyber and SIKE have high-speed pure hardware implementations reported. For

decapsulation, Kyber outperforms SIKE by a factor of over 200.

In Table 4, we summarize implementations targeting Xilinx Virtex-7 FPGAs. Unfortunately, the only conclusion that can be drawn from these tables is an advantage of Classic McEliece over SIKE in terms of all performance metrics other than the number of LUTs and flip-flops.

All results reported in Table 5 were obtained using the same SoC FPGA, Zynq UltraScale+. Only Saber and NTRU Prime are implemented in pure hardware. Additionally, their implementations are of different types, high-speed and lightweight, respectively.

3 Basic Features of Compared Algorithms

Selected features of all implemented KEMs are summarized in Table 6. All three KEMs are based on the underlying IND-CPA public-key encryption (PKE) schemes. In CRYSTALS-Kyber and Saber, the conversions to the corresponding IND-CCA KEMs are performed using very similar variants of the Fujisaki–Okamoto transform [31], [38]. NTRU uses a generic transformation from a deterministic public-key encryption scheme to construct a KEM. The NTRU KEM transformation provides IND-CCA2 security with a tight reduction to the well-studied OW-CPA (one-way CPA) security of the NTRU PKE [72]. The only KEMs with no Decryption Failure in the underlying PKE are NTRU-based KEMs, NTRU-HPS and NTRU-HRSS. Consequently, these schemes require no re-encryption during decapsulation.

In all of these KEMs, the elementary operation is multiplication mod q. In Saber, NTRU-HPS, and NTRU-HRSS, q is a power of two, significantly simplifying the reduction mod q. In Kyber, q is a special prime, selected in such a way to support speeding up

Table 6: Features of lattice-based NIST Round 3 finalists in the category of KEMs

Feature	CRYSTALS-Kyber	Saber	NTRU-HPS	NTRU-HRSS
Underlying problem	Module-LWE: Module Learning with Errors	Mod-LWR: Module Learning with Rounding	Shortest Vector Problem	Shortest Vector Problem
Degree n	Power of 2	Power of 2	Prime	Prime
Modulus q	Prime	Power of 2	Power of 2 with $q/8-2 \leq 2n/3$	Power of 2 with $q > 8\sqrt{2}(n+1)$
Other major parameters	k: the lattice dimension as a multiple of n, η: noise parameter	l: number of polynomials per vector, p, T: other moduli, μ: parameter of CBD	w: Fixed weight for f and r	N/A
Hash-based functions	SHA3-256, SHA3-512, SHAKE128, SHAKE256	SHA3-256, SHA3-512, SHAKE128	SHA3-256	SHA3-256
Sampling	Integers are sampled from a centered binomial distribution (CBD)	Integers are sampled from a centered binomial distribution (CBD)	Fixed-weight and variable -weight polynomials are sampled from a uniform distribution	Variable-weight polynomials are sampled from a uniform distribution
Decryption failures	Yes	Yes	No	No
Polynomial Rings	$\mathbb{Z}_q[x]/(x^n+1)$	$\mathbb{Z}_q[x]/(x^n+1)$	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: $Z_3[x](x-1)/(x^n-1)$
#Polynomial Multiplications in Encapsulation	k^2+k	l^2+l	1 in R/q	1 in R/q
#Polynomial Multiplications in Decapsulation	k^2+2k	l^2+2l	1 in R/q 1 in S/q 1 in S/3	1 in R/q 1 in S/q 1 in S/3

* $\Phi_n = (x^n-1)/(x-1)$ irreducible in $Z_q[x]$

Table 7: Parameter sets of investigated algorithms. Notation: Sk - Secret Key, Pk - Public key, Ct - Ciphertext.

Algorithm	**Parameter Set**	**Security Level**	**Degree** n	**Modulus** q	**Sk Size [bytes]**	**Pk Size [bytes]**	**Ct Size [bytes]**
Kyber	Kyber512	1	256	3329	1,632	800	768
NTRU-HPS	ntruhps2048677	1*	677	2^{11}	1,235	931	931
NTRU-HRSS	ntruhrss701	1*	701	2^{13}	1,452	1,138	1,138
Saber	LightSaber-KEM	1	256	2^{13}	1,568	672	736
Kyber	Kyber768	3	256	3329	2,400	1,184	1,088
NTRU-HPS	ntruhps4096821	3*	821	2^{12}	1,592	1,230	1,230
Saber	Saber-KEM	3	256	2^{13}	2,304	992	1,088
Kyber	Kyber1024	5	256	3329	3,168	1,568	1,568
Saber	FireSaber-KEM	5	256	2^{13}	3,040	1,312	1,472

* assuming non-local computational models

polynomial multiplication in $\mathbb{Z}_q[x]/(x^n+1)$ using the Number Theoretic Transform (NTT).

All four algorithms use SHA3-256. Saber additionally employs SHA3-512 and SHAKE128. Kyber requires the same set of hash-based algorithms as Saber, extended with SHAKE256. NTRU-based KEMs use sampling from the uniform distribution. In Kyber and Saber, a Centered Binomial Distribution (CBD) is employed.

There are two variants of NTRU described in the specification, the NTRU-HPS based on Hoffstein, Pipher, and Silverman's NTRU encryption scheme [37] and NTRU-HRSS introduced by Hülsing, Rijnveld, Schanck, and Schwabe in [48]. The NTRU-HPS parameter sets follow the approach to use fixed-weight sample spaces and allow several choices of modulus q for each degree n. Meanwhile, the NTRU-HRSS allows arbitrary-weight sample spaces but restricts q as a function of n.

In Kyber and 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 NTRU-based KEMs, the most time-consuming operation is polynomial multiplication in the rings specified in Table 6.

Parameter sets of three investigated candidates are summarized in Table 7. The specification of NTRU associates two different security categories with each parameter set of 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 [70], Section 5.3 Security Categories). The same computation model is implicitly assumed by the submitters of the other investigated algorithms. We implemented three parameter sets of NTRU-HPS and NTRU-HRSS, which are ntruhrss701, ntruhps677, and ntruhps821, corresponding to security levels 1, 1, and 3, respectively in non-local models of computation.

4 Methodology

Hardware design methodologies are developed by the industry over the period of decades. The Register-Transfer Level (RTL) methodology is the most popular design methodology codified by academic textbooks and supported by most industry-grade computer-aided design tools. This methodology assumes designing/coding at a level that is manageable for humans and easy for tools to turn into efficient hardware. The entire system is divided into the Datapath and Controller. The Datapath is described using a hierarchical block diagram using medium-scale components (e.g., adders, multipliers, multiplexers, registers, and memories). The Controller is described using hierarchical algorithmic state machine (ASM) charts or state diagrams. Indirectly, the designer specifies what happens in the circuit in every clock cycle. Thus, the latency (the execution time of a particular major

operation) in clock cycles is an inherent feature of the design. The tools determine the maximum clock frequency at which the circuit can operate and the amount of hardware resources used.

Any other approaches to hardware design are often mistrusted. In some cases, justifiably so. For example, recent attempts at replacing RTL with High-Level Synthesis resulted in PQC designs 2-4 orders of magnitude less efficient [15, 23]. Similarly, the use of the software/hardware co-design for PQC led to inconclusive results disregarded by NIST at the end of Round 2 [23, 2].

Therefore, the development of hardware implementations described in this paper follows the traditional RTL methodology. The designers of each implementation worked very closely with each other to ensure a consistent approach to all optimizations. Our designs started when no pure hardware implementations of CRYSTALS-Kyber, NTRU, or Saber were reported in the literature yet. All major designs decisions were made independently of those made in related concurrent projects described in [81], [80], [74], and [83]. All code was developed from scratch without using any library components or any parts of other groups' designs. Consequently, our designs are fully portable, well-documented, and easy to improve and maintain.

All modules common for multiple algorithms, such as the SHA-3/SHAKE unit, were reused. The designs for NTRU and Saber are encoded using VHDL. The design for CRYSTALS-Kyber is encoded using Chisel [8]. We believe that in the RTL methodology, the choice of a hardware description language has a negligible effect on the obtained results. Functional verification of the hardware description language (HDL) code has been performed by comparing simulation results with precomputed outputs generated by a reference software implementation.

On top of this well-known and trusted design methodology, we define a quite straightforward benchmarking methodology. The primary goal is fairness, not a novelty.

All our hardware implementations assume the use of the FIFO interface defined in [28]. This interface is similar to the interface of the AXI4-Stream Protocol [5].

In terms of functionality of designed units, several options are possible: 1) separate units for encapsulation, decapsulation, and key generation; 2) one unit supporting encapsulation, decapsulation, and key generation, with resource sharing; 3) one unit supporting encapsulation and decapsulation and the second unit responsible for key generation; 4) one unit (on the server-side) supporting key generation and decapsulation, and the second unit (on the client-side) supporting encapsulation. None of these assumptions meet the requirements of all applications. In this paper, we assume Scenario 1). However, whenever possible, we also report results for Scenario 2).

Similarly, there are two major assumptions regarding support for multiple parameter sets: 1. choice among parameters sets at the time of synthesis; 2. choice among parameters sets at run time. The advantage of Approach 1) is the ability to determine the minimum possible resource utilization separately for each security level. Approach 2) demonstrates the flexibility of hardware implementation. However, it will likely require a larger amount of resources than the implementation supporting the highest security level. In this paper, we adopted Approach 1.

The primary design goal is speed. The speed is characterized using two primary metrics: a) the minimum latency in time units and b) the maximum number of operations per second. These two metrics are related. However, any particular application may have independent requirements in terms of their values. For example, real-time applications, such as secure communication between two autonomous vehicles, may have very strict requirements regarding the time required to establish secure communication and thus the total time required for encapsulation and decapsulation. At the same time, the required number of operations per second may be very small and thus not limiting. On the other hand, a high-traffic server may have to handle thousands of secret key establishments per

second. Simultaneously, the time allowed for each individual transaction (and thus the latency of decapsulation) may be quite large.

Taking into account that specific thresholds depend strongly on an application and the state of technology, no specific values are assumed in this benchmarking effort. Instead, we assume that both decreasing latency and increasing the number of operations per second are worthy goals as they will broaden the range of applications that can use a new PQC standard at a given stage of technology. For simplicity, we assume, in agreement with most of the literature, that each design processes only one set of inputs (keys, ciphertexts, random bits) at a time. As a result, the number of operations per second becomes a direct inverse of latency in time units. One, however, should keep in mind an important difference between them: duplicating a design doubles the number of operations per second, but it does not change the latency.

When choosing between multiple potential solutions during the design-space exploration, we give priority to the designs that minimize the product $Latency^2 \cdot Area$ and thus maximize the ratio $\#Operations\_per\_seconds^2/Area$. Thus, for high-speed implementations, minimizing *Latency* can be accomplished at the cost of a relatively higher increase in *Area*. However, the parallelization is pursued only until it gives substantial gain in speed as compared to the area increase in LUTs.

For our target platforms, we chose representative devices of two different FPGA / FPGA SoC families: Artix-7 and Zynq UltraScale+. Specifically, we choose the largest devices of both families supported by free versions of Xilinx tools. For each device, we assume that its highest speed grade is used. These assumptions led us to choosing a) Artix-7 XC7A200T-3, with 134,600 LUTs, 365 BRAMs, and 740 DSP units, and Zynq UltraScale+ ZU7EV-3, with 230,400 LUTs, 312 BRAMs, 96 Ultra BRAMs, and 1,728 DSP units. Based on the previous work, summarized in Section 2, these devices are sufficient for a vast majority of designs reported to date. Out of their resources, the number of LUTs is the most limiting. The use of BRAMs and DSP units is typically negligible. Therefore, for the purpose of design-space exploration, we use the number of LUTs as a measure of the circuit *Area*. The maximum clock frequency is determined using binary search. Only final results obtained after placing and routing are reported.

5 Results

5.1 CRYSTALS-Kyber

In Table 8, we report our results for CRYSTALS-Kyber and compare them with previous work.

The implementation of Kyber presented in this work outperforms the best previous implementation, reported in [81], by approximately a factor of two in terms of the execution time in microseconds for all major operations (key generation, encapsulation, and decapsulation). The comparison in terms of resource utilization is less obvious, considering that all operations are allowed to share the same resources in this work. In [81], the resource utilization for the server side (executing key generation and decapsulation) and the client side (executing encapsulation) are reported separately. However, based on our design, extending the coverage of operations from the server side to include encapsulation has negligible influence on the circuit area. Thus, it seems to be fair to compare our resource utilization numbers with the corresponding numbers for the server unit in [81].

Previous software/hardware implementations, such as those reported in [80], [30], [3], are clearly inferior in terms of both the latency and the product of the latency and the number of LUTs.

Table 8: Implementation results of different Kyber instances on various FPGAs and ASIC technologies. S/C - denotes results for Server/Client, respectively.

Scheme	Key/Encaps/Decaps [K Cycles]	Freq. [MHz]	Key/Encaps/Decaps [us]	LUT	FF	DSP	BR AM	Device
Kyber-CCAKEM L1								
Kyber R3 [this work]	2.2/3.2/4.5	220	10.0/14.7/20.5	9,457	8,543	4	4.5	Artix-7 XC7A200
Kyber R3 [81]	3.8/5.1/6.7	S/C 161/167	23.4/30.5/41.3	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2/2	Artix-7 XA7A12
Kyber R2 [80]	18.6/45.9/80.0	300	61.9/153.0/267.0	-	-	-	-	ASIC 28nm
Kyber R2 [30]	150.1/193.1/204.8	-	-	23,925	10,844	32	21	Zynq7000 XC7Z020
Kyber R2 [3]	710.0/971.0/870.0	59	11,993.2/16,402.0/14,695.9	1,842	1,634	34	5	Artix-7 XC7A35T
Kyber R3 [this work]	2.2/3.2/4.5	450	4.9/7.2/10.0	9,504	8,957	4	4.5	Zynq-UltraScale+ XCZU7EV
Kyber-CCAKEM L3								
Kyber R3 [this work]	2.6/3.7/4.9	220	12.0/17.0/22.2	10,530	9,837	6	6.5	Artix-7 XC7A200
Kyber R3 [81]	6.3/7.9/10.0	S/C 161/167	39.2/47.6/62.3	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2/2	Artix-7 XA7A12
Kyber R3 [this work]	2.6/3.7/4.9	450	5.9/8.3/10.9	10,590	10,458	6	6.5	Zynq-UltraScale+ XCZU7EV
Kyber-CCAKEM L5								
Kyber R3 [this work]	3.6/4.8/5.8	220	16.2/21.7/26.4	11,623	11,131	8	8.5	Artix-7 XC7A200
Kyber R3 [81]	9.4/11.3/13.9	S/C 161/167	58.2/67.9/86.2	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2/2	Artix-7 XA7A12
Kyber R2 [80]	39.7/81.6/136.5	300	132.0/272.0/455.0	-	-	-	-	ASIC 28nm
Kyber R2 [30]	349.7/405.5/424.7	-	-	23,925	10,844	32	21	Zynq7000 XC7Z020
Kyber R2 [3]	2,203.0/2,619.0/2,429.0	59	37,212.8/44,239.9/9,639.1	1,842	1,634	34	5	Artix-7 XC7A35T
Kyber R3 [this work]	3.6/4.8/5.8	450	7.9/10.6/12.9	11,676	11,959	8	8.5	Zynq-UltraScale+ XCZU7EV

5.2 NTRU

Table 9: Implementation results of NTRU on Zynq UltraScale+

Design	Module	Freq	LUT	FF	Slice	DSP	BRAM	Latency	
								Cycles	μs
Security Level 1									
NTRU-HRSS701	Key Gen.	300	49,001	39,957	9,357	45	2.5	51,812	172.7
	Encaps.	300	31,494	25,120	6,652	0	2.5	2,219	7.4
	Decaps.	300	37,702	34,441	8,032	45	2.5	8,826	29.4
NTRU-HPS677	Key Gen.	250	41,047	39,037	7,968	45	6	48,179	192.7
	Encaps.	250	26,325	17,568	4,638	0	5	3,687	14.7
	Decaps.	300	29,935	19,511	5,217	45	2.5	7,522	25.1
Security Level 3									
NTRU-HPS821	Key Gen.	250	50,347	44,281	10,127	45	6.5	67,157	268.6
	Encaps.	250	33,698	30,551	7,370	0	5.5	4,576	18.3
	Decaps.	300	38,642	33,003	7,785	45	2.5	10,211	34.0

The results of our implementations of two variants of NTRU, NTRU-HRSS (at the security level 1) and NTRU-HPS (at the security levels 1 and 3), are summarized in Table 9. At security level 1, NTRU-HRSS outperforms NTRU-HPS for key generation and encapsulation. However, it slightly lags behind for decapsulation. NTRU-HRSS operates at a higher clock frequency (except for decapsulation) but requires consistently more resources than NTRU-HPS. With the increase in the security level, NTRU-HPS requires more FPGA resources, with the exception of DSP units, the number of which remains the same.

Table 10: Comparison between the implementations of NTRU and Streamlined NTRUPrime at the same security levels for Zynq UltraScale+

Scheme	Module	Freq [MHz]	LUT	FF	Slices	DSP	BRAM	Latency	
								cycles	us
NTRU-HPS821	Key Gen.	250	50,347	44,281	10,127	45	6.5	67,157	268.6
	Encaps.	250	33,698	30,551	7,370	0	5.5	4,576	18.3
	Decaps.	300	38,642	33,003	7,785	45	2.5	10,211	34.0
Streamlined NTRUPrime [59]	Key Gen.	271.6	5,935	3,204	1,068	12	8.5	1,289,959	4,749.5
	Encaps.	271.6	4,570	2,843	844	8	7.5	119,250	439.1
	Decaps.	271.6	5,117	2,958	902	8	7.0	260,307	958.4

Table 11: Implementation results of Saber on Zynq UltraScale+

Design	Module	Freq	LUT	FF	Slice	DSP	BRAM	Latency	
								Cycles	μs
Security Level 1									
LightSaber	Key Gen.	370	23,557	14,190	3,844	0	1.5	1,607	4.3
	Encaps.	370	24,199	14,457	3,984	0	1.5	2,153	5.8
	Decaps.	370	24,655	14,879	4,364	0	1.5	2,794	7.6
Security Level 3									
Saber	Key Gen.	370	20,496	13,939	3,634	0	1.5	2,709	7.3
	Encaps.	370	21,069	14,074	3,503	0	1.5	3,735	10.1
	Decaps.	370	21,342	14,233	3,816	0	1.5	4,682	12.7
Security Level 5									
FireSaber	Key Gen.	370	19,752	14,358	3,321	0	1.5	4,895	13.2
	Encaps.	370	20,696	13,949	3,455	0	1.5	5,867	15.9
	Decaps.	370	20,868	14,237	3,460	0	1.5	7,128	19.3

In NTRU-HPS, the maximum clock frequency for the key generation and encapsulation is limited by the sort-based sampling unit. This unit is not a part of the decapsulation core. Consequently, decapsulation can be performed at a 50 MHz higher clock frequency.

In Table 10, we compare our implementation of NTRU-HPS with the lightweight implementation of Streamlined NTRU Prime, reported in [59]. Both variants have the same security level. In both implementations, key generation is implemented separately. In this comparison, NTRU outperforms NTRU Prime by a factor of 17.6, 24.0, and 28.1 for key generation, encapsulation, and decapsulation, respectively. At the same time, it uses significantly more FPGA resources, e.g., about 8.5x more LUTs for key generation and about 7.4x, 7.6x more LUTs for encapsulation and decapsulation unit, respectively.

5.3 Saber

The results of our implementations of Saber at the security levels 1, 3, and 5, targeting Zynq UltraScale+, are summarized in Table 11. This table demonstrates three clear advantages of Saber: 1) the resource utilization stays almost the same, independently of the security level, 2) the maximum clock frequency is independent of the security level, 3) implementations use no DSP units and a very small number of BRAMs. Only latency is affected considerably by using higher security levels.

The comparison with the best implementations of Saber reported in the literature to date is shown in Table 12 and Figs 12–14. In Table 12, our implementations are marked in bold.

The designs with the terms x2 and x4 in the name are obtained by unrolling the polynomial multiplication unit by 2 and 4 times, respectively. These designs offer significant

Table 12: Implementation results of Saber and comparison with related works on ZynqUltrascale+ platform

Design	Key/Encaps/Decaps [K Cycles]	Freq [MHz]	Key/Encaps/Decap [us]	LUT	FF	Slices	DSP	BRAM
Security Level 1								
LightSaber x4	0.9/1/1.3	310	2.9/3.3/4.2	65,890	28,230	10,404	0	1.5
LightSaber x2	1.1/1.4/1.8	345	3.2/4.1/5.2	39,423	21,467	6,610	0	1.5
LightSaber	1.6/2.2/2.8	370	4.3/5.8/7.6	24,688	14,785	4,309	0	1.5
Unified Saber [83]	0.5/0.7/1	100	5.2/6.6/9.9	34,886	9,858	—	85	6.0
Unified Saber [74]	2.8/4/5	150	18.4/26.9/33.6	24,979	10,732	—	0	2.0
Security Level 3								
Saber x4	1.3/1.5/1.9	310	4.3/4.8/6	48,895	27,715	7,726	0	1.5
Saber x2	1.8/2.2/2.8	345	5.2/6.5/8.1	32,099	21,037	5,294	0	1.5
Saber	2.7/3.7/4.7	370	7.3/10.1/12.7	21,352	14,232	3,763	0	1.5
Unified Saber [83]	0.9/1.4/1.7	100	9.4/14.0/16.8	34,886	9,858	—	85	6.0
Saber [74]	5.5/6.6/8	250	21.8/26.5/32.1	25,079	10,750	—	0	2.0
Unified Saber [74]	5.5/6.6/8	150	36.4/44.1/53.6	24,979	10,732	—	0	2.0
Security Level 5								
FireSaber x4	2/2.1/2.6	310	6.5/6.9/8.5	38,268	27,677	6,348	0	1.5
FireSaber x2	2.9/3.4/4.1	345	8.4/9.8/11.9	25,760	21,035	4,239	0	1.5
FireSaber	4.9/5.9/7.1	370	13.2/15.9/19.3	20,383	14,239	3,408	0	1.5
Unified Saber [83]	1.5/1.8/2.3	100	15.3/18.1/23	34,886	9,858	—	85	6.0

improvements in latency at the cost of a substantial increase in the number of LUTs, flip-flops, and slices. Overall, they are inferior in terms of the metric $Latency^2 \cdot Area$. As a result, they are not taken into account in other comparisons presented in this paper. In Figs 12–14, the implementation described in [83] is denoted as Saber-Tsinghua, the implementation from [74] as Saber-U.Birmingham, and our design as Saber-TW. Based on these figures and Table 12, our implementation is clearly the fastest and the smallest in terms of the number of LUTs, and maintains this advantage for all Saber operations.

5.4 Comparison of Round 3 candidates

In Figs. 3–14, we illustrate the dependence between the speed of the Round 3 candidates (in the number operations per second, which, for all considered designs, is equivalent to the inverse of latency in time units) and their resource utilization in LUTs. All other components of resource utilization, such as the number of BRAMs or DSP units, are omitted for simplicity. In terms of the percentage of the total amount of FPGA resources, the utilization of LUTs is typically the highest. However, some exceptions to this typical scenario may occasionally occur. In the legend of these figures, TW refers to This Work.

For security level 1, the number of implementations on Artix-7 FPGAs, illustrated in Figs. 3–5, is 9 for key generation and 10 for encapsulation and decapsulation. These implementations represent seven candidates, including all four finalists in the category of KEMs. Saber is the fastest for all three major operations. CRYSTALS-Kyber is clearly the second for key generation and decapsulation. In the case of encapsulation, it is practically tied with NTRU-HRSS. NTRU-HRSS and NTRU-HPS are about 3x slower than Saber for decapsulation and over 30x slower for key generation. FrodoKEM is at least two orders of magnitude slower than Saber for all three operations. However, it can be implemented using several times fewer LUTs. Classic McEliece is more than two orders of magnitude slower than Saber for key generation, about an order of magnitude slower for decapsulation, and only a few times slower for encapsulation. It also always requires more LUTs. BIKE trails Saber by more than two orders of magnitude for key generation, one order of magnitude for encapsulation, and almost four orders of magnitude for decapsulation. The HQC results are available only for key generation. Its performance is comparable to that of NTRU algorithms, i.e., over 30x lower than for Saber. Overall,

Table 13: Artix-7 results for designs proposed and documented in this work

Key Generation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber	9.5	1.00	Kyber	12.0	1.00	Kyber	16.2	1.00
Kyber	10.0	1.05	Saber	15.9	1.33	Saber	28.8	1.78
NTRU-HRSS	323.8	34.08	NTRU-HPS	516.6	43.05			
NTRU-HPS	370.6	39.01						

Encapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber	12.7	1.00	Kyber	17.0	1.00	Kyber	21.7	1.00
NTRU-HRSS	13.9	1.09	Saber	22.0	1.29	Saber	34.5	1.59
Kyber	14.7	1.16	NTRU-HPS	35.2	2.07			
NTRU-HPS	28.4	2.24						

Decapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber	16.4	1.00	Kyber	22.2	1.00	Kyber	26.4	1.00
Kyber	20.5	1.25	Saber	27.5	1.24	Saber	41.9	1.59
NTRU-HPS	47.0	2.87	NTRU-HPS	63.8	2.87			
NTRU-HRSS	55.2	3.37						

four finalists – Saber, Kyber, NTRU, and Classic McEliece – clearly outperform three alternates – FrodoKEM, BIKE, and HQC. Based on the data from Table 4, we can clearly establish that SIKE is much slower than the four finalists as well. Among the finalists, Saber and Kyber perform overall much better than NTRU and Classic McEliece.

The results for the security level 5 are shown in Figs. 6–8. The majority of Round 3 candidates either do not have implementations, or these implementations have exceeded the resources of Artix-7 FPGAs. Kyber and Saber are in a virtual tie, with Kyber slightly ahead for all operations.

For the security level 3, we present results for both Artix-7 (in Figs 9–11) and Zynq UltraScale+ (in Figs 12–14). In the case of Artix-7, results are reported for all four finalists and two alternates (FrodoKEM and BIKE). In the case of Zynq UltraScale+, the graphs cover three lattice-based finalists and one alternate candidate, NTRU Prime. For all operations, at the security level 3, Kyber outperforms Saber by a very small factor in terms of both speed and area. NTRU (represented at this level only by NTRU-HPS) is more than an order of magnitude slower for key generation and 2-3 times slower for encapsulation and decapsulation. Classic McEliece slightly exceeds the speed of NTRU for encapsulation, but lags behind by almost an order of magnitude for decapsulation and two orders of magnitude for key generation. FrodoKEM and BIKE are orders of magnitude slower than finalists for encapsulation and decapsulation, and better only than Classic McEliece for key generation. The results obtained using Zynq UltraScale+ (or UltraScale+) seem to indicate that Streamlined NTRU Prime lags at least two orders of magnitude behind the best two candidates for each major operation. It is possible, however, that these results are sub-optimal and biased by the fact that the designer's primary goal was small resource utilization. Three Saber designs are comparable in terms of speed and resource utilization. However, the design proposed and documented in this work is clearly the best in terms of both the speed and the usage of LUTs.

In Tables 13 and 14, the exact numerical results are presented for the execution times of implementations proposed and described in this paper. These results clearly indicate that NTRU is between 30 and 50 times slower than Saber for the key generation at both level 1 and level 3. NTRU is also about 2-4 times slower than Saber for decapsulation. Only for encapsulation, the performance of NTRU becomes comparable. Kyber is between 5% and

Table 14: Zynq UltraScale+ results for designs proposed and documented in this work

Key Generation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber	4.3	1.00	Kyber	5.9	1.00	Kyber	7.9	1.00
Kyber	4.9	1.14	Saber	7.3	1.24	Saber	13.2	1.67
NTRU-HRSS	172.7	40.16	NTRU-HPS	268.6	44.81			
NTRU-HPS	192.7	48.18						

Encapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber	5.8	1.00	Kyber	8.3	1.00	Kyber	10.6	1.00
Kyber	7.2	1.24	Saber	10.1	1.22	Saber	15.9	1.50
NTRU-HRSS	7.4	1.28	NTRU-HPS	18.3	1.81			
NTRU-HPS	14.7	2.53						

Decapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber	7.6	1.00	Kyber	10.9	1.00	Kyber	12.9	1.00
Kyber	10.0	1.32	Saber	12.7	1.17	Saber	19.3	1.50
NTRU-HPS	25.1	3.30	NTRU-HPS	34.0	3.12			
NTRU-HRSS	29.4	3.87						

32% slower at level 1. It outperforms Saber in all rankings at levels 3 by a factor ranging between 17% and 33%. At level 5, the advantage of Kyber increases to the range 50%-80%. The reasons for the change in the ranking of Kyber and Saber depending on the security level are as follows. In Kyber, the NTT-based multiplier is quite small and sequential. Therefore, it is justifiable to use 2, 3, and 4 multipliers at the security levels 1, 3, and 5, respectively (as described in Appendix A). In Saber, the schoolbook multiplier is big and parallel. Therefore, increasing the number of multipliers is not justifiable, as a small increase in speed causes a large increase in area. Consequently, the relative performance of Kyber increases at higher security levels.

6 Conclusions

In this paper, we have proposed, documented, and benchmarked a) the first complete hardware implementations of two variants of NTRU (NTRU-HRSS and NTRU-HPS), as defined in the submissions to Rounds 2 and 3 of the NIST PQC standardization process; b) the best high-speed implementation of Saber, outperforming competing designs in terms of both speed and resource utilization, and c) the fastest implementation of CRYSTALS-KYBER. All designs are fully reproducible, and their source code will be released as open-source after the acceptance of this paper to a journal or a conference with proceedings.

We also have comprehensively reviewed the related literature and collected information about hardware and software/hardware implementations of all Round 3 candidates in the category of Key Encapsulation Mechanisms (KEMs). Our analysis reveals that four NIST PQC finalists significantly outperform all alternate candidates when implemented in hardware with speed as a primary optimization target. Among the four finalists, Saber and CRYSTALS-Kyber significantly outperform NTRU and Classic McEliece for at least a subset of all operations. The differences between the two top candidates are relatively minor. Saber seems to exhibit a minor advantage at the security level 1. However, this advantage dissipates when the security level is increased. In particular, at the security level 5, Kyber is at least 50% faster than Saber.

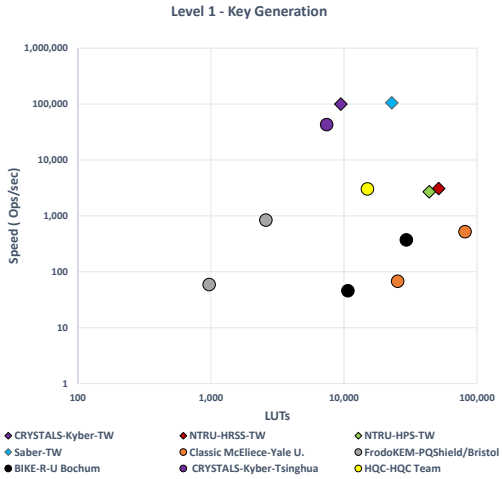

Figure 3: L1, KeyGen, Artix-7

Figure 4: L1, Encaps, Artix-7

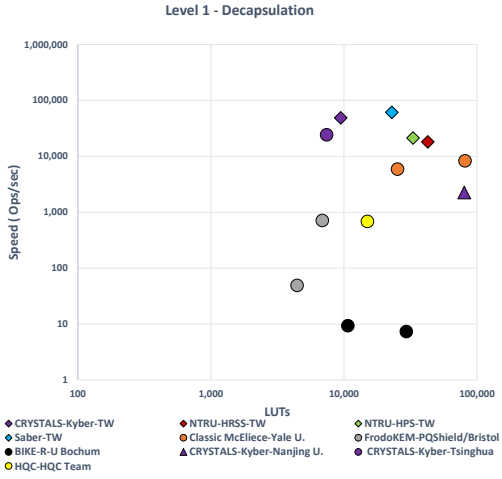

Figure 5: L1, Decaps, Artix-7

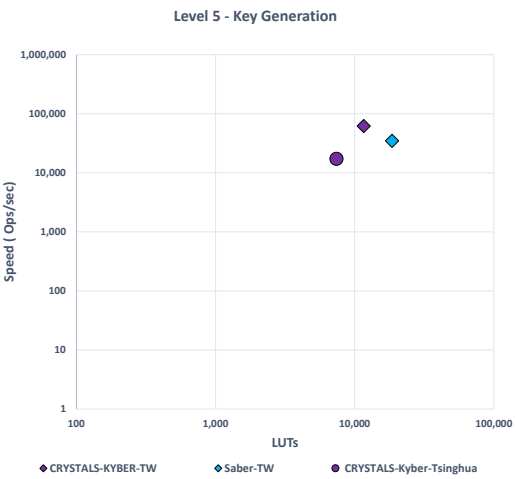

Figure 6: L5, KeyGen, Artix-7

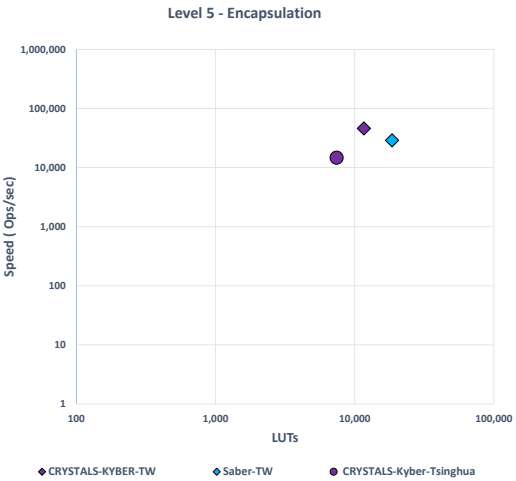

Figure 7: L5, Encaps, Artix-7

Figure 8: L5, Decaps, Artix-7

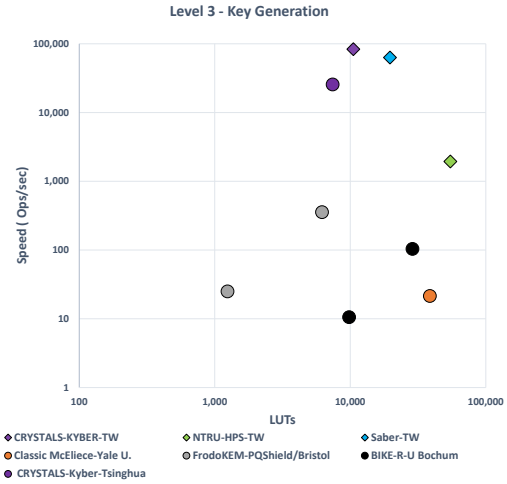

Figure 9: L3, KeyGen, Artix-7

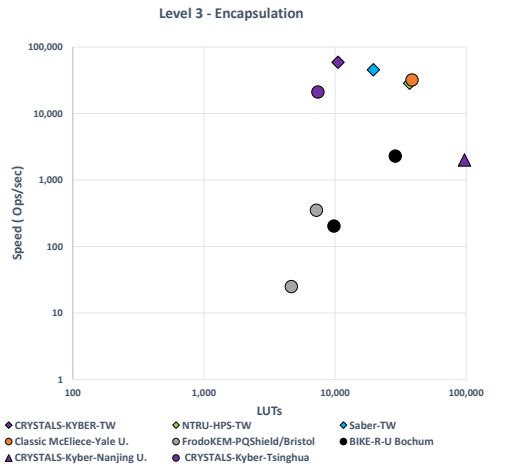

Figure 10: L3, Encaps, Artix-7

Figure 11: L3, Decaps, Artix-7

Figure 12: L3, KeyGen, Zynq UltraScale+

Figure 13: L3, Encaps, Zynq UltraScale+

Figure 14: L3, Decaps, Zynq UltraScale+

References

1. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Liu, Y.-K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., and Smith-Tone, D.: Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. Tech. rep. NIST IR 8240, Gaithersburg, MD: National Institute of Standards and Technology (2019)
2. Alagic, G., Apon, D.C., Cooper, D.A., Dang, Q.H., Kelsey, J.M., Liu, Y.-K., Miller, C.A., Moody, D., Peralta, R.C., Perlner, R.A., Robinson, A.Y., Smith-Tone, D.C., and Alperin-Sheriff, J.: Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. Tech. rep. NISTIR 8309, Gaithersburg, MD: National Institute of Standards and Technology (2020)
3. Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., and Petri, R.: ISA Extensions for Finite Field Arithmetic. IACR Transactions on Cryptographic Hardware and Embedded Systems **Volume 2020**, 219–242 (2020)
4. Aragon, N., Barreto, P.S.L.M., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Gaborit, P., Gueron, S., Güneysu, T., Melchor, C.A., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J.-P., Vasseur, V., and Zémor, G.: BIKE: Bit Flipping Key Encapsulation: Submission for Round 3 Consideration. Tech. rep., (2020)
5. ARM: AMBA 4 AXI4-Stream Protocol Specification. Tech. rep., (2010)
6. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., and Stehlé, D.: CRYSTALS-KYBER: Algorithm Specifications And Supporting Documentation (Version 2.0). Tech. rep., (2019)
7. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., and Stehlé, D.: CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation (Version 3.01), (2021)
8. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K.: Chisel: Constructing Hardware in a Scala Embedded Language. In: DAC Design Automation Conference 2012, pp. 1212–1221 (2012)
9. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K.: Chisel: Constructing Hardware in a Scala Embedded Language. In: DAC Design Automation Conference 2012, pp. 1212–1221 (2012)
10. Baldwin, B., Byrne, A., Lu, L., Hamilton, M., Hanley, N., O'Neill, M., and Marnane, W.P.: FPGA Implementations of the Round Two SHA-3 Candidates. In: 2010 International Conference on Field Programmable Logic and Applications, FPL 2010, pp. 400–407, Milan, Italy (2010)
11. Banerjee, U., Ukyab, T.S., and Chandrakasan, A.P.: Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-Based Protocols. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(4) (2019)
12. Banerjee, U., Ukyab, T.S., and Chandrakasan, A.P.: Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-Based Protocols (Extended Version). Cryptology ePrint Archive 2019/1140, (2020)
13. Barrett, P.: Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. Lecture Notes in Computer Science, pp. 311–323. Springer, Berlin, Heidelberg (1987)
14. Basso, A., and Roy, S.S.: Optimized Polynomial Multiplier Architectures for Post-Quantum KEM Saber. In: 58th Design Automation Conference, DAC 2021, San Francisco (2021)
15. Basu, K., Soni, D., Nabeel, M., and Karri, R.: NIST Post-Quantum Cryptography- A Hardware Evaluation Study. Cryptology ePrint Archive 2019/047, (2019)
16. Bernstein, D.J., Heninger, N., Lou, P., and Valenta, L.: Post-Quantum RSA. In: 8th International Workshop on Post-Quantum Cryptography, PQCrypto 2017. Lecture Notes in Computer Science, pp. 312–329. Springer International Publishing, Cham (2017)
17. Bernstein, D.J., and Yang, B.-Y.: Fast Constant-Time Gcd Computation and Modular Inversion. TCHES (2019). https://tches.iacr.org/index.php/TCHES/article/view/8298 (visited on 04/16/2021)
18. Bodrato, M., and Zanoni, A.: Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In: International Symposium on Symbolic and Algebraic Computation, ISSAC 2007, pp. 17–24 (2007)

19. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - Web Page*, https://competitions.cr.yp.to/caesar.html (2019). 2019
20. Chen, C., Danba, O., Rijneveld, J., Schanck, J.M., Saito, T., Schwabe, P., Whyte, W., Xagawa, K., Yamakawa, T., and Zhang, Z.: NTRU: Algorithm Specifications And Supporting Documentation, (2020)
21. Chung, C.-M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.-J., and Yang, B.-Y.: NTT Multiplication for NTT-Unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. TCHES **2021**(2), 159–188 (2021)
22. Cryptographic Engineering Research Group (CERG) at George Mason University: *Hardware Benchmarking of CAESAR Candidates*, https://cryptography.gmu.edu/athena/index.php?id=CAESAR (2019). 2019
23. Dang, V., Farahmand, F., Andrzejczak, M., Mohajerani, K., Nguyen, D.T., and Gaj, K.: Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-Design Approaches. Cryptology ePrint Archive 2020/795, p. 86 (2020)
24. Dang, V.B., Farahmand, F., Andrzejczak, M., and Gaj, K.: Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign. In: 2019 International Conference on Field Programmable Technology, FPT 2019, pp. 206–214. IEEE, Tianjin, China (Dec. 9-13, 2019)
25. de Dinechin, F.: Reflections on 10 Years of FloPoCo. In: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 187–189. IEEE, Kyoto, Japan (2019)
26. Dubois, and Venetsanopoulos: The Discrete Fourier Transform Over Finite Rings with Application to Fast Convolution. IEEE Transactions on Computers **C-27**(7), 586–593 (1978)
27. Elkhatib, R., Azarderakhsh, R., and Mozaffari-Kermani, M.: High-Performance FPGA Accelerator for SIKE. IEEE Trans. Comput. (2021)
28. Farahmand, F., Dang, V.B., Nguyen, D.T., and Gaj, K.: 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, pp. 23–43. Springer, Chongqing, China (2019)
29. Farahmand, F., Ferozpuri, A., Diehl, W., and Gaj, K.: Minerva: Automated Hardware Optimization Tool. In: 2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, pp. 1–8. IEEE, Cancun (2017)
30. Fritzmann, T., Sigl, G., and Sepúlveda, J.: RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 239–280 (2020)
31. Fujisaki, E., and Okamoto, T.: Secure Integration of Asymmetric and Symmetric Encryption Schemes. Journal of Cryptology **26**(1), 80–101 (2013)
32. Gaj, K.: Challenges and Rewards of Implementing and Benchmarking Post-Quantum Cryptography in Hardware. In: 2018 Great Lakes Symposium on VLSI, GLSVLSI 2018, pp. 359–364. ACM Press, Chicago, IL, USA (2018)
33. Gaj, K., Homsirikamol, E., and Rogawski, M.: 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. LNCS, pp. 264–278, Santa Barbara, CA (2010)
34. Gaj, K., Homsirikamol, E., Rogawski, M., Shahid, R., and Sharif, M.U.: Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive 2012/368, (2012)
35. Gaj, K., Kaps, J.-P., Amirineni, V., Rogawski, M., Homsirikamol, E., and Brewster, B.Y.: 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, pp. 414–421. IEEE, Milan, Italy (2010)
36. Gambetta, J.: IBM's Roadmap For Scaling Quantum Technology, IBM Research Blog. (2020). https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/ (visited on 06/01/2021)
37. Hoffstein, J., Pipher, J., and Silverman, J.H.: NTRU: A Ring-Based Public Key Cryptosystem. In: Algorithmic Number Theory. Ed. by G. Goos, J. Hartmanis, J. van Leeuwen, and J.P. Buhler, pp. 267–288. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
38. Hofheinz, D., Hövelmanns, K., and Kiltz, E.: A Modular Analysis of the Fujisaki-Okamoto Transformation. In: Theory of Cryptography. Ed. by Y. Kalai and L. Reyzin, pp. 341–371. Springer International Publishing, Cham (2017)

39. Hofheinz, D., Hövelmanns, K., and Kiltz, E.: A Modular Analysis of the Fujisaki-Okamoto Transformation. In: Theory of Cryptography. Ed. by Y. Kalai and L. Reyzin, pp. 341–371. Springer International Publishing, Cham (2017). http://link.springer.com/10.1007/978-3-319-70500-2_12 (visited on 04/15/2019)
40. Homsirikamol, E., Yalla, P., Farahmand, F., Diehl, W., Ferozpuri, A., Kaps, J.-P., and Gaj, K.: Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API. GMU Report, Fairfax, VA: GMU (2016)
41. Homsirikamol, E., Diehl, W., Ferozpuri, A., Farahmand, F., Yalla, P., Kaps, J.-P., and Gaj, K.: CAESAR Hardware API. Cryptology ePrint Archive 2016/626, (2016)
42. Homsirikamol, E., and Gaj, K.: Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study. In: Applied Reconfigurable Computing - ARC 2015. LNCS, pp. 217–228. Springer International Publishing, Cham (2015)
43. Homsirikamol, E., and Gaj, K.: Toward a New HLS-Based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study. In: 2017 International Conference on Field Programmable Technology, FPT 2017, pp. 120–127. IEEE, Melbourne, Australia (2017)
44. Homsirikamol, E., Yalla, P., and Farahmand, F.: Development Package for Hardware Implementations Compliant with the CAESAR Hardware API, v2.0, (2017). https://cryptography.gmu.edu/athena/index.php?id=CAESAR
45. Howe, J.: Optimised Lattice-Based Key Encapsulation in Hardware. In: Second NIST Post-Quantum Cryptography Standardization Conference 2019, p. 13 (2019)
46. Hu, J., Wang, W., Cheung, R.C., and Wang, H.: Optimized Polynomial Multiplier Over Commutative Rings on FPGAs: A Case Study on BIKE. In: 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 231–234. IEEE, Tianjin, China (2019)
47. Huang, Y., Huang, M., Lei, Z., and Wu, J.: A Pure Hardware Implementation of CRYSTALS-KYBER PQC Algorithm through Resource Reuse. IEICE Electronics Express (2020)
48. Hülsing, A., Rijneveld, J., Schanck, J., and Schwabe, P.: High-Speed Key Encapsulation from NTRU. In: Cryptographic Hardware and Embedded Systems – CHES 2017. Ed. by W. Fischer and N. Homma, pp. 232–252. Springer International Publishing, Cham (2017)
49. Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., and Bachrach, J.: Reusability Is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In: Proceedings of the 36th International Conference on Computer-Aided Design. ICCAD '17, pp. 209–216. IEEE Press, Piscataway, NJ, USA (2017). http://dl.acm.org/citation.cfm?id=3199700.3199728 (visited on 07/21/2019)
50. Kaps, J.-P., Surapathi, K.K., Habib, B., Vadlamudi, S., Gurung, S., and Pham, J.: Lightweight Implementations of SHA-3 Candidates on FPGAs. In: 12th International Conference on Cryptology in India, Indocrypt 2011. LNCS, pp. 270–289, Chennai, India (2011)
51. Knezevic, M., Kobayashi, K., Ikegami, J., Matsuo, S., Satoh, A., Kocabas, Ü., Fan, J., Katashita, T., Sugawara, T., Sakiyama, K., Verbauwhede, I., Ohta, K., Homma, N., and Aoki, T.: Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **20**(5), 827–840 (2012)
52. Knezevic, M., Vercauteren, F., and Verbauwhede, I.: Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods. IEEE Transactions on Computers **59**(12), 1715–1721 (2010)
53. Koziel, B., Ackie, A.-B., El Khatib, R., Azarderakhsh, R., and Kermani, M.M.: SIKE'd Up: Fast Hardware Architectures for Supersingular Isogeny Key Encapsulation. IEEE Transactions on Circuits and Systems I: Regular Papers (2020)
54. Kumm, M., Gustafsson, O., Garrido, M., and Zipf, P.: Optimal Single Constant Multiplication Using Ternary Adders. IEEE Transactions on Circuits and Systems II: Express Briefs **65**(7), 928–932 (2018). https://ieeexplore.ieee.org/document/7752883/ (visited on 03/23/2020)
55. Liu, B., and Wu, H.: Efficient Architecture and Implementation for NTRUEncrypt System. In: 2015 IEEE 58th International Midwest Symposium on Circuits and Systems, MWSCAS 2015, Fort Collins, CO, USA (2015)
56. Longa, P., Naehrig, M., Longa, P., and Naehrig, M.: Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In: Cryptology and Network Security - CANS 2016, pp. 124–139. Springer International Publishing, Cham (2016). http://link.springer.com/10.1007/978-3-319-48965-0_8 (visited on 02/17/2019)
57. Maria Bermudo Mera, J., Turan, F., Karmakar, A., Sinha Roy, S., and Verbauwhede, I.: Compact Domain-Specific Co-Processor for Accelerating Module Lattice-Based KEM. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE, San Francisco, CA, USA (2020)

58. Marotzke, A.: A Constant Time Full Hardware Implementation of Streamlined NTRU Prime. In: 19th International Conference on Smart Card Research and Advanced Applications, CARDIS 2020. LNCS, pp. 3–17. Springer International Publishing, Cham (2020)
59. Marotzke, A.: A Constant Time Hardware Implementation of Streamlined NTRU Prime, (2021). https://github.com/AdrianMarotzke/SNTRUP (visited on 06/04/2021)
60. Massolino, P.M.C., Longa, P., Renes, J., and Batina, L.: A Compact and Scalable Hardware/Software Co-Design of SIKE. IACR Transactions on Cryptographic Hardware and Embedded Systems (2020)
61. Mera, J.M.B., Turan, F., Karmakar, A., Roy, S., and Verbauwhede, I.: Compact Domain-Specific Co-Processor for Accelerating Module Lattice-Based Key Encapsulation Mechanism. Cryptology ePrint Archive 2020/321, p. 15 (2020)
62. Montgomery, P.L.: Modular Multiplication without Trial Division. Mathematics of computation **44**(170), 519–521 (1985)
63. NIST: *Post-Quantum Cryptography: Call for Proposals*, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals (2019). 2019
64. NTRU Submission Team: Round 2 Submissions - NTRU Candidate Submission Package, Post-Quantum Cryptography: Round 2 Submissions. (2019)
65. *Post-Quantum Cryptography: Round 1 Submissions*, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (2019). Apr. 2019
66. *Post-Quantum Cryptography: Round 2 Submissions*, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions (2019). Apr. 2019
67. Reinders, A.H., Misoczki, R., Ghosh, S., and Sastry, M.R.: Efficient BIKE Hardware Design with Constant-Time Decoder. In: 2020 IEEE International Conference on Quantum Computing and Engineering (QCE), pp. 197–204. IEEE, Denver, CO, USA (2020)
68. Richter-Brockmann, J., Mono, J., and Guneysu, T.: Folding BIKE: Scalable Hardware Implementation for Recongurable Devices. IEEE Trans. Comput. (2021)
69. *Round 3 Submissions - HQC Candidate Submission Package*, https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions (2020). Oct. 2020
70. *Round 3 Submissions - NTRU Candidate Submission Package*, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions (2021). Apr. 2021
71. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I., Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., and Verbauwhede, I.: Compact Ring-LWE Cryptoprocessor. In: Salinesi, C., Norrie, M.C., and Pastor, Ó. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014. LNCS, pp. 371–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). http://link.springer.com/10.1007/978-3-662-44709-3_21 (visited on 02/17/2019)
72. Saito, T., Xagawa, K., and Yamakawa, T.: Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model. In: Advances in Cryptology – EUROCRYPT 2018. Lecture Notes in Computer Science, pp. 520–551. Springer International Publishing, Cham (2018)
73. Shor, P.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE Comput. Soc. Press, Santa Fe, NM, USA (1994)
74. Sinha Roy, S., and Basso, A.: High-Speed Instruction-Set Coprocessor for Lattice-Based Key Encapsulation Mechanism: Saber in Hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 443–466 (2020)
75. Sinha Roy, S., and Verbauwhede, I.: Lattice-Based Public-Key Cryptography in Hardware. Springer Singapore, Singapore (2020)
76. Vandersypen, L.: *A "Spins-inside" Quantum Computer*, Invited Talk (2017). Utrecht, the Netherlands, June 2017
77. Wang, W., Jungk, B., Wälde, J., Deng, S., Gupta, N., Szefer, J., and Niederhagen, R.: XMSS and Embedded Systems - XMSS Hardware Accelerators for RISC-V. Cryptology ePrint Archive 2018/1225, (2018)
78. Wang, W., Szefer, J., and Niederhagen, R.: FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes. In: Lange, T., and Steinwandt, R. (eds.) 9th International Conference on Post-Quantum Cryptography, PQCrypto 2018. LNCS, pp. 77–98. Springer International Publishing, Fort Lauderdale, Florida (2018)
79. Wang, W., Szefer, J., Niederhagen, R., Szefer, J., and Niederhagen, R.: FPGA-Based Key Generator for the Niederreiter Cryptosystem Using Binary Goppa Codes. In: Cryptographic Hardware and Embedded Systems, CHES 2017. LNCS, pp. 253–274. Springer International Publishing, Cham (2017)

80. Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., and Zeng, X.: VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. IEEE Transactions on Circuits and Systems I: Regular Papers **67**(8), 1–13 (2020)

81. Xing, Y., and Li, S.: A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. TCHES (2021)

82. Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., and Liu, L.: Highly Efficient Architecture of NewHope-NIST on FPGA Using Low-Complexity NTT/INTT. IACR Transactions on Cryptographic Hardware and Embedded Systems (2020). https://tches.iacr.org/index.php/TCHES/article/view/8544 (visited on 03/13/2020)

83. Zhu, Y., Zhu, M., Yang, B., Zhu, W., Deng, C., Chen, C., Wei, S., and Liu, L.: LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR. IEEE Transactions on Circuits and Systems I: Regular Papers **68**(3), 1146–1159 (2021)

A CRYSTALS-Kyber

A.1 Algorithms

CRYSTALS-Kyber [7] is a lattice-based CCA-secure key encapsulation mechanism (KEM) based on the Module Learning with Errors problem (MLWE). Kyber provides three parameter sets, Kyber512, Kyber768, and Kyber1024, corresponding to NIST security levels 1, 3, and 5. Kyber constructs its CCA-KEM primitives (key generation, encapsulation, and decapsulation) over CPA-secure public-key encryption (PKE) primitives (KeyGen, Encrypt, Decrypt) through a variant of the Fujisaki-Okamoto transform [31, 39].

For all security levels, polynomials are of the same degree $n = 256$, and their coefficients are members of the base prime field $\mathbb{Z}_q$, where $q = 3329$. However, a different number of polynomials is required for each security level. These polynomials are treated as a vector. The size of this vector is specified using the parameter k. k is 2, 3, and 4 for security levels 1, 3, and 5, respectively. Secret noise polynomials are sampled from a Centered Binomial Distribution (CBD), where η is either 2 or 3.

Pseudocode of the Kyber CPAPKE Key Generation, Encryption, and Decryption are given in algorithms 1, 2, and 3, respectively. Kyber CCA KEM schemes are built upon the CPAPKE operations, multiple hashing operations, and the FO transformation to achieve the IND-CCA2 security.

The detailed algorithms of the Kyber CCAKEM Key Generation, Encapsulation, and Decapsulation are shown in algorithms 4, 5 and 6. Here is the meaning of notation used in these algorithms:

- XOF: SHAKE128
- H: SHA3-256
- G: SHA3-512
- PRF(s, b): SHAKE256(s||b)
- KDF: SHAKE256
- CBD_{eta}: Sample from centered binomial distribution $\eta \in \{2, 3\}$.
- Parse: Sample from uniform distribution using rejection sampling and then perform NTT.
- Decode_l: deserialize an array of $32l$ bytes into a polynomial $f = f_0 + f_1X + \ldots + f_{255}X^{255}$ with each coefficient f_i in $0, \ldots, 2^l - 1$
- Encode_l: inverse of Decode_l
- $\text{Compress}_q(x, d)$: perform compression by taking an element $x \in \mathbb{Z}_q$ and outputting an integer in $0, \ldots, 2^d - 1$, where $d < \lceil log_2(q) \rceil$
- $\text{Decompress}_q(x, d)$: perform decompression, defined such that $x' = \text{Decompress}_q(\text{Compress}_q(x, d), d)$ is an element close to x, specifically: $|x' - x \; mod^{\pm} q| \leq B_q := \lceil \frac{q}{2^{d+1}} \rfloor$

Algorithm 1 KYBER.CPAPKE.KeyGen() [6]

Output: Secret key sk
Output: Public key pk

$d \xleftarrow{\$} \{0,1\}^{256}$
$(\rho|\sigma) \leftarrow G(d)$
for i from 0 to $k-1$ **do**
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}[i][j] \leftarrow \text{Parse}(\text{XOF}(\rho, j, i))$
  end for
end for
for i from 0 to $k-1$ **do**
  $\mathbf{s}[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, i))$
end for
for i from 0 to $k-1$ **do**
  $\mathbf{e}[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, i+k))$
end for
$\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \leftarrow (\hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}})$
$pk \leftarrow \text{Encode}(\hat{\mathbf{t}}) \ |\rho$
$sk \leftarrow \text{Encode}(\hat{\mathbf{s}})$
return (pk, sk)

Algorithm 2 KYBER.CPAPKE.Enc(pk, m, r) [6]

Input: Public key pk
Input: Message m
Input: Random coins $coins$
Output: Ciphertext c

$(pk'|\rho) \leftarrow pk, \quad pk' \in \{0,1\}^{256 \cdot 12k}$
$\hat{t} \leftarrow \text{Decode}(pk')$
for i from 0 to $k-1$ **do**
  for j from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}^T[i][j] \leftarrow \text{Parse}(\text{XOF}(\rho, i, j))$
  end for
end for
for i from 0 to $k-1$ **do**
  $\mathbf{r}[i] \leftarrow \text{CBD}_\eta(\text{PRF}(coins, i))$
end for
for i from 0 to $k-1$ **do**
  $\mathbf{e}_1[i] \leftarrow \text{CBD}_\eta(\text{PRF}(coins, i+k))$
end for
$e_2 \leftarrow \text{CBD}_\eta(\text{PRF}(coins, 2k))$
$\hat{\mathbf{r}} \leftarrow \text{NTT}(\mathbf{r})$
$\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
$v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}(m, 1)$
$c_1 \leftarrow \text{Compress}(\mathbf{u}, d_u)$
$c_2 \leftarrow \text{Compress}(v, d_v)$
$c \leftarrow (c_1|c_2)$
return c

Algorithm 3 KYBER.CPAPKE.Dec(sk, c) [6]

Input: Secret key sk
Input: Ciphertext c
Output: Message m

$(c_1|c_2) \leftarrow c$
$\mathbf{u} \leftarrow \text{Decompress}(c_1, d_u)$
$v \leftarrow \text{Decompress}(c_2, d_v)$
$\hat{\mathbf{s}} \leftarrow \text{Decode}(sk)$
$\hat{\mathbf{u}} \leftarrow \text{NTT}(\mathbf{u})$
$\mu \leftarrow v - \text{NTT}^{-1}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}})$
$m \leftarrow \text{Compress}(\mu, 1)$
return m

Algorithm 4 KYBER.CCAKEM.KeyGen() [6]

Output: Public key pk
Output: Secret key sk

$(pk, sk') \leftarrow$ KYBER.CPAPKE.KeyGen()
$z \xleftarrow{\$} \{0,1\}^{256}$
$sk \leftarrow (sk'|pk|\text{H}(pk)|z)$
return (pk, sk)

Algorithm 5 KYBER.CCAKEM.Encap(pk) [6]

Input: Public key pk
Output: Ciphertext c
Output: Shared secret ss

$m' \xleftarrow{\$} \{0,1\}^{256}$
$m \leftarrow \text{H}(m')$
$(\bar{ss}|r) \leftarrow \text{G}(m|\text{H}(pk)) \quad \bar{ss}, r \in \{0,1\}^{256}$
$c \leftarrow$ KYBER.CPAPKE.Enc(pk, m, r)
$ss \leftarrow \text{KDF}(\bar{ss}|\text{H}(c))$
return (c, ss)

Algorithm 6 KYBER.CCAKEM.Decap(c, sk) [6]

Input: Ciphertext: ct
Input: Secret key: sk, public key: pk, $h_{pk} = H(pk)$, failure random z
Output: Shared secret $ss \in \{0,1\}^{256}$

$m' \leftarrow$ KYBER.CPAPKE.Dec(sk', c)
$(\bar{ss}|r') \leftarrow \text{G}(m'|h_{pk}) \quad \bar{ss}, r \in \{0,1\}^{256}$
$c' \leftarrow$ KYBER.CPAPKE.Enc(pk, m', r')
$ss \leftarrow$ **if** $(\text{H}(c') = \text{H}(c))$ KDF$(\bar{ss}|\text{H}(c))$ **else** KDF$(z|\text{H}(c))$
return ss

A.2 Polynomial Multiplication

A basic operation in lattice-based cryptography (LBC) schemes is the multiplication of two polynomials. In Kyber the polynomials are elements of $R_q = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$.

An efficient method for the computation of polynomial convolution in R_q is through the use of the Number-Theoretic Transform (NTT) [26] which is a generalization of the Discrete Fourier Transform (DFT) to the finite ring $\mathbb{Z}_q$. Since the Round 2 version, Kyber uses $n = 256$ and $q = 3329 = 13 \cdot 2^8 + 1$ where $2n \nmid q - 1 = 13 \cdot 2^8$. To make efficient NTT multiplication possible, a new definition of NTT was provided, which transforms a polynomial of degree 256 to a polynomial of degree 128 made up of degree one polynomials as its coefficients.

$$\hat{f}_k = f \bmod (X^2 - \zeta^{(2k+1)}) \tag{1}$$

In other words $\hat{f}$ consists of 128 polynomials of degree one:

$$\hat{f}_k = f \bmod (X^2 - \zeta^{(2k+1)}) = \hat{\hat{f}}_{2k} + \hat{\hat{f}}_{2k+1} X \tag{2}$$

The sequence of 128 coefficient pairs of degree 1 polynomials can be viewed as a polynomial of degree 256 and then the NTT transform can be expressed separately for the odd and even coefficients:

Point-wise multiplication consists of 128 basic products $\hat{f} \cdot \hat{g} \bmod X^2 - \zeta^{(2i+1)}$:

$$\begin{aligned} \hat{\hat{h}}_{2i} + \hat{\hat{h}}_{2i+1} X &= (\hat{\hat{f}}_{2i} + \hat{\hat{f}}_{2i+1} X)(\hat{\hat{g}}_{2i} + \hat{\hat{g}}_{2i+1} X) \\ &= \left(\hat{\hat{f}}_{2i}\hat{\hat{g}}_{2i} + \zeta^{(2i+1)} \hat{\hat{f}}_{2i+1}\hat{\hat{g}}_{2i+1}\right) + \left(\hat{\hat{f}}_{2i}\hat{\hat{g}}_{2i+1} + \hat{\hat{f}}_{2i+1}\hat{\hat{g}}_{2i}\right) X \end{aligned} \tag{3}$$

A.3 Hardware Architecture

The proposed hardware architecture for Round 3 Kyber supports the following variants and operations: a) CPA-PKE: Key Generation, Encryption, and Decryption, and b) CCA-KEM: Key Generation, Encapsulation, and Decapsulation. The top-level unit is shown in Fig. 15. The hardware is implemented in Chisel hardware design language [9][49] and incorporates state of the art techniques for optimizing speed and minimizing the resource overhead.

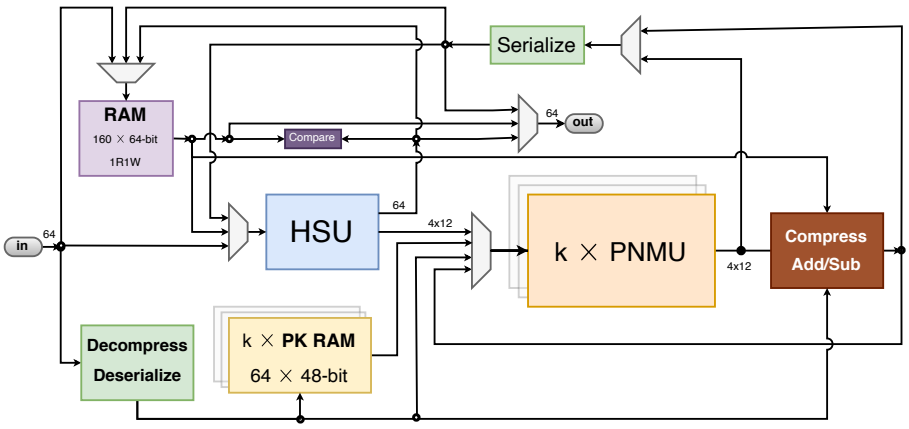

Figure 15: Block diagram of the Kyber top-level datapath

A.3.1 Polynomial NTT and Multiplication Unit

The Polynomial NTT/Multiplication Unit (PNMU) performs forward and inverse NTT operations, as well as point-wise multiplication and accumulation. The top-level block diagram of PNMU is shown in Fig. 16. For security level K, we use k instances of the PNMU module to allow for overlapping NTT and MAC operations on k polynomials of a vector. Each PNMU instance has its own dedicated operating RAM (NTT RAM) as well as a dedicated input FIFO (inFIFO), to minimize stalls when the module is busy.

A logical word of the NTT RAM consists of four coefficients. In each butterfly unit, two consecutive operations are performed on two pairs of read coefficients, enabling an efficient memory access scheme similar to [71]. In the DIT configuration, two input coefficients (out of four) need to be swapped. The Head Reorder and Tail Reorder units are responsible for this reordering inside coefficient pairs. The same reordering is required for the output of the DIF butterfly. No extra reordering or scaling steps are required for either of the forward or inverse NTT operations.

The data-path of the NTT units consists of two parallel configurable radix-2 butterflies, which can operate in three modes of operation: DIT NTT, DIF iNTT, and point-wise multiply-accumulate (MAC). As the computation of NTT (and INTT) for odd and even coefficients can be carried out independently, we deploy two butterflies operating in parallel. . This architecture allows for a more efficient implementation as the address generation and control circuits are shared. Also as four coefficients can be efficiently packed in three 18-bit wide BRAM banks in a Simple Dual-Port configuration.

The result of the inverse NTT operation needs to be scaled by n^{-1}. In straightforward software and hardware implementations of NTT the scaling is performed in a separate step requiring n additional field multiplications for each polynomial. By performing a division by $2 \pmod q$ at each layer of inverse NTT, the scaling step can be entirely avoided. This observation was also used by Zhang et al. [82]. In that implementation, two divide-by-2 hardware units are utilized to scale both outputs of the radix-2 INTT butterfly. In our implementation, we use a single divide-by-2 unit for each butterfly, and the other output of each butterfly is scaled by adjusting twiddle factors of the inverse transform.

The twiddle factors are stored in a separate ROM, which is mapped to BRAM-based memory during the FPGA synthesis. While only 128 twiddle constants are sufficient for both forward and inverse NTT operations, as we're storing the twiddle factors in a Block RAM with available extra capacity, an additional scaled copy (scaled by $2^{-1} \bmod q$) of the twiddle factors is also kept to eliminate an extra scaling step and the need for scaling hardware in one side of the DIF butterfly. This also eliminates need for generation of different twiddle-factor index sequence during the inverse NTT operation.

The point-wise multiplication of polynomials a and b (both in NTT domain) is performed on base degree 1 polynomials in the form of $a_{2i} + a_{2i+1}X$ and $b_{2i} + b_{2i+1}X$. The resulting polynomial $c = a * b$ is calculated using the following formula:

$$c_{2i} + c_{2i+1}X = (a_{2i} + a_{2i+1}X)(b_{2i} + b_{2i+1}X) \bmod X^2 - \zeta_i$$

which results to:

$$\begin{cases} c_{2i} & = a_{2i}b_{2i} + a_{2i+1}b_{2i+1}\zeta_i \\ c_{2i+1} & = a_{2i}b_{2i+1} + a_{2i+1}b_{2i} \end{cases}$$

The straightforward formulation requires 5 modular multiplications for producing a pair of coefficients. By using the Karatsuba method, the number of modular multiplications can be reduced to 4:

$$\begin{cases} c_{2i} & = a_{2i}b_{2i} + a_{2i+1}b_{2i+1}\zeta_i \\ c_{2i+1} & = (a_{2i} + a_{2i+1})(b_{2i} + b_{2i+1}) - a_{2i}b_{2i} - a_{2i+1}b_{2i+1} \end{cases}$$

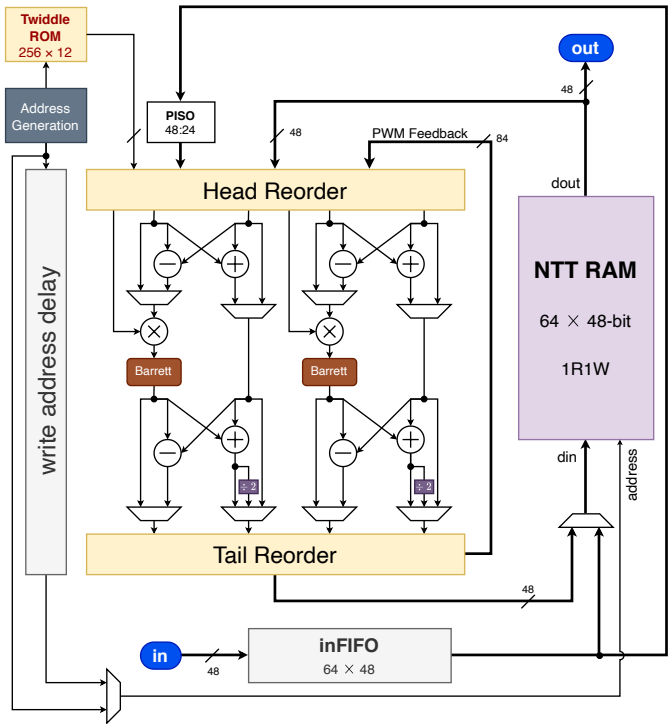

Figure 16: Block diagram of the NTT.

By adding some multiplexers and careful scheduling of the butterfly pipelines, the same resources are used to perform the point-wise multiplication and accumulation (MAC). The scheduling of the butterflies for point-wise multiplication is shown in Figure 17. The butterfly pipeline is interleaved with the first pass of operations entering the pipeline and the second pass of operations coming from the feedback loop. This results in the complete utilization of the multiplier and reduction units of both butterflies in each cycle. The multiplication and accumulation of each polynomial require 128 cycles (plus 12 additional cycles to clear the pipeline).

A.3.2 Barrett reduction with support for division

Coefficients of polynomials are elements of a finite field (or ring) $\mathbf{Z}_q$, where q is a small constant modulus (less than 20 bits). In Kyber q is a prime. This choice requires a modular reduction step after most arithmetic operations to keep the bit width of the data bounded. Variants of Barrett [13], Montgomery [62], K-RED [56], and SAMS2 [55] reduction algorithms have been widely used in software and hardware implementations of R-LWE schemes.

We use an optimized variant of the Barrett reduction algorithm shown in Algorithm 7. As shown by Knezevic et al. [52], by careful selection of parameters α and β, only one level of conditional subtraction will be required. The hardware generator code creates optimized single constant multipliers (SCM) based on shift-adder trees and ternary adders based on [54].

A.3.3 Keccak and Sampling Unit

Kyber uses the SHA3-256 and SHA3-512 hash functions as well as SHAKE128 and SHAKE256 extendable-output functions. All of them are based on the Keccak permutation.

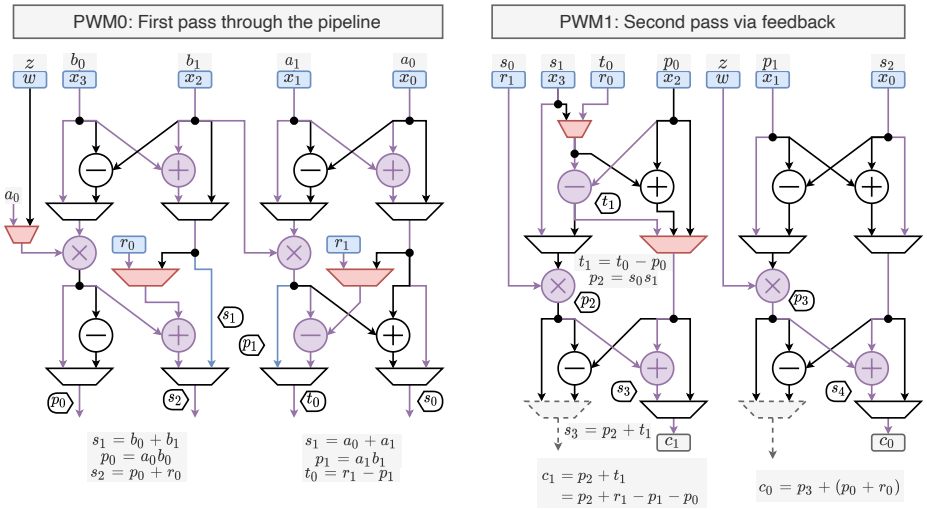

Figure 17: Point-wise multiplication and accumulation (MAC) using the NTT butterflies

Our Keccak implementation takes advantage of the full-width, basic iterative architecture, which performs 24 rounds in 24 clock cycles. The data input and output are 64 bits wide with the valid-ready (decoupled) interface. Keccak and Sampling Unit (KSU) integrates the Keccak with CBD and rejection-based samplers.

A.3.4 Centered Binomial Sampler

The CBD module in Kyber is responsible for performing binomial sampling. Kyber requires 12 bits of random data generated by SHAKE module to generate four coefficients per clock cycle. Two CBD parameters η_1 and η_2 are used. $\eta_2 = 2$ for all security levels, $\eta_1 = 3$ for security Level 1 and $\eta_1 = 2$ for the other security levels. The samples are calculated from formula 4.

$$B_\eta = \sum_{i=1}^{\eta} (a_i - b_i) \tag{4}$$

Hamming weights of the input chunks of the size $\eta \in \{2, 3\}$ are calculated. Negative results are mapped to positive values through a lookup table.

A.3.5 Rejection-based Sampler

In order to minimize the size of the public key, the public matrix A (or its transpose A^T) is generated through the rejection-based sampling of a deterministic random source. The uniform random is generated using SHAKE128 from the public key seed. The output from SHAKE is partitioned into groups of 12 bits, and the resulting unsigned value is only accepted as a valid coefficient if it is less than $q = 3329$. This gives a probability of 81.27% for a sample to be valid. As k^2 sampled polynomials need to be generated through multiple invocations of the Keccak permutation and filtering of coefficients, this step is one of the bottlenecks in Kyber hardware scheduling. The rejection-based sampling of A is inherently not constant time, but any timing variation entirely depends on the public key seed and therefore would not expose any secrets. The Rejection Sampling submodule of the KSU is able to construct a polynomial of 256 coefficients in an average time of 82 cycles. The output will then be delivered to the PNMU for the execution of the matrix-vector multiplication.

Algorithm 7 Optimized Barrett Modular Reduction and Division

Require: $0 \leq u \leq (q-1)^2$
Ensure: $r = u \bmod q,\ r \in [0, q)$ ▷ remainder
Ensure: $u = d.q + r \bmod q,\ d \in [0, q)$ ▷ quotient

Generation Time: Find optimal values for α and β such that:
1. Only a single conditional subtraction is required
2. Multiplication with the constant μ has minimal hardware complexity.

For Kyber Round 3:
$q := 3329,\ n := \lceil \log_2(q) \rceil = 12,\ \alpha := 12,\ \beta := -2,\ \mu := \lfloor \frac{2^{n+\alpha}}{q} \rfloor = 5039$

function BarrettReduce(u)
  $u_h \leftarrow u \gg (n+\beta)$ ▷ discard $n+\beta$ least-significant bits
  $d \leftarrow (\mu \cdot u_h) \gg (\alpha - \beta)$ ▷ discard $\alpha - \beta$ least-significant bits
  $r \leftarrow u - d \cdot q$
  if $r \geq q$ **then** ▷ conditional subtraction
    $r \leftarrow r - q$
    $d \leftarrow d + 1$
  end if
end function

A.3.6 Comparison of re-encrypted Ciphertext

During decapsulation, instead of comparing the re-encrypted ciphertext with the received ciphertext, we first generate $H(ct)$ of the original ciphertext. After re-encryption the hash of the re-encrypted ciphertext ($H(ct')$) is computed, and then only the hashes are compared. This eliminates the need for keeping the original ciphertext. This design decision has negligible cycle overhead but allows a simpler control circuit and also provides a path towards protection against ciphertext malleability side-channel attacks. Additional protection against power and electromagnetic side-channel attacks for this design is under development and will be presented in our future work.

A.3.7 Improvements over Previous Work

A state-of-the-art hardware implementation of Kyber is reported in [81]. Our design has been conducted independently. Both designs employ all relevant optimization techniques reported before, including:

- Flexible DIF/DIT butterflies for performing forward/inverse NTT transforms with efficient resource sharing and avoiding any extra shuffling (bit-reverse ordering) steps.
- Efficient division by two at each step of inverse NTT, eliminating the need for the scaling step.
- Parallel processing of even and odd coefficients using a double-butterfly structure.
- Reuse of DIF/DIT butterflies for performing Kyber's point-wise multiplication.
- Use of Karatsuba algorithm to reduce number of field multiplications for point-wise multiplications from 5 to 4.

Our improvements over previous work are as follows:

Our high-level architecture and scheduling are based on the use of K Polynomial NTT/Multiplication Units (PNMUs) and a single Hash/Sampling unit (HSU). In [81], only a single set of these units is used. Our PNMUs are developed to have low area (around 940 LUTs each). As a result, they allow efficient exploitation of Kyber's algorithm-level

parallelism by setting K to 2, 3, and 4 for the security levels 1, 3, and 5, respectively. Through the design space exploration, we determined that using K PNMUs is optimal from the point of view of our optimization metric, $Latency^2 \cdot Area$.

We support efficient reuse of NTT butterflies for point-wise multiplication (PWM) and the Multiply-Accumulate (MAC) operation. As in [81], we utilize the Karatsuba method to reduce the number of required field multiplications. However, we developed a more resource-efficient mapping of operations. Unlike [81], we support the accumulation of NTT-domain polynomials, which eliminates the extra cycles for load/add/store during multiplication by matrix A (used in Keygen, Encap, Decap) and the extra NTT-domain addition (in KeyGen). We also eliminate the need for an extra "accumulate" unit used in [81].

We support a more efficient NTT/PWM memory access, reducing the memory requirement of each PNMU to 1-read 1-write (1R1W) 64x48-bit RAM. In Xilinx FPGAs, this memory is mapped to a single BRAM tile (36 Kb block RAM) in the simple dual-port (SDP) mode of operation. Efficient "Head/Tail Re-order" units of the double-butterfly structure perform online re-ordering of coefficients entering/exiting the butterfly pipeline in NTT/invNTT (as a Multi-path Delay Commutator) as well as the re-ordeing required for PWM/MAC. The double-butterfly structure computes the point-wise multiplication through interleaved reiteration of the pipeline as depicted in Figure 17.

Our deeply pipelined butterfly implementation, including 12 stages, results in a higher maximum clock frequency. The optimized control circuit can skip pipeline flushing stalls whenever possible.

We have developed an optimized reduction unit based on a tweaked version of Barrett's algorithm. This unit has been shown to be faster and more efficient than the other implementations of modular reduction suggested in the literature. It also computes the division by q, required for a fast and efficient implementation of the compression step. As a bonus, our hardware generation code works perfectly for any value of q, including the value used in CRYSTALS-DILITHIUM.

Our fast and efficient implementation of the Rejection sampler processes four coefficients at a time, reducing the "Parse" step to an average of 116 cycles per polynomial.

Our fast implementations of Keccak and the CBD sampler are integrated together into an optimized Hash/Sampler Unit.

Our fast and efficient Keecak implementation has input and output widths of 64 bits, with decoupled output and efficient SHA3/SHAKE padding of the input words.

Efficient implementation of the CBD sampler which can simultaneously supports η values 2 and 3 (for security level 1) and provide output in the standard range.

Finally, unlike [81], our design is technology-independent and does not employ any vendor-specific IPs. These features allow for easy deployment on FPGA platforms other than Xilinx, use of synthesis flows other than Vivado (including open-source FPGA flows), as well as porting to ASICs.

B NTRU

B.1 Algorithms

Definitions and Parameters: Φ_1 is $(x-1)$. Φ_n is $(x^n-1)/\Phi_1 = x^{n-1}+x^{n-2}+\ldots+x+1$. From the implementation point of view, all operations in NTRU are polynomial operations over the quotient rings R_q, S_q and S_p where $R_q : \mathbb{Z}_q[x]/\Phi_1\Phi_n$, $S_q : \mathbb{Z}_q[x]/\Phi_n$, and $S_p : \mathbb{Z}_p[x]/\Phi_n$. Parameter p is fixed to 3 in all parameter sets of NTRU. Thus, polynomials in S_p are in ternary form, i.e., have their coefficients in $\{-1, 0, 1\}$. In this paper, for NTRU, we use the notation S_p and S_3 interchangeably. Coefficients of polynomials in R_q and S_q have bit-widths of $\epsilon_q = log_2 q$ and those of polynomials in S_p have bit-widths of $\epsilon_p = \lceil log_2 p \rceil$.

Algorithm 8 NTRU PKE Keypair

Input: fg_bits
Output: $pk = packed\_h$ and $sk = (packed\_f, packed\_f_p, packed\_h_q)$

1: $(f, g) \leftarrow \mathsf{Sample}(fg\_bits)$
2: $f_p \leftarrow f^{-1} \bmod (3, \Phi_n)$
3: $G \leftarrow 3 \cdot g$
4: $v_0 \leftarrow (G \cdot f) \bmod (q, \Phi_n)$
5: $v_1 \leftarrow v_0^{-1} \bmod (q, \Phi_n)$
6: $h \leftarrow (v_1 \cdot G \cdot G) \bmod (q, \Phi_1\Phi_n)$
7: $h_q \leftarrow (v_1 \cdot f \cdot f) \bmod (q, \Phi_1\Phi_n)$
8: $sk \leftarrow (\mathsf{pack\_}\epsilon_\mathsf{p}(f), \mathsf{pack\_}\epsilon_\mathsf{p}(f_p), \mathsf{pack\_}\epsilon_\mathsf{q}(h_q))$
9: $pk \leftarrow \mathsf{pack\_}\epsilon_\mathsf{q}(h)$

Algorithm 9 NTRU KEM Keypair

Input: Random seed $seeds$
Output: $pk = packed\_h$ and $sk = (packed\_f, packed\_f_p, packed\_h_q, s)$

1: $(fg\_bits, prf\_key) \leftarrow \mathsf{SHAKE128}(seeds)$
2: $packed\_h, packed\_f, packed\_f_p, packed\_h_q \leftarrow \mathsf{PKE.KeyPair}(fg\_bits)$
3: $sk \leftarrow (packed\_f, packed\_f_p, packed\_h_q, bits\_to\_bytes(prf\_key))$
4: $pk \leftarrow packed\_h$

Algorithm 10 NTRU PKE Encryption

Input: $pk = packed\_h$, r and m
Output: $packed\_c$

1: $m' \leftarrow \mathsf{Lift}(m)$
2: $h \leftarrow \mathsf{unpack\_}\epsilon_q(packed\_h)$
3: $c \leftarrow (r \cdot h + m') \bmod (q, \Phi_1\Phi_n)$
4: $packed\_c \leftarrow \mathsf{pack\_}\epsilon_\mathsf{q}(c)$

Algorithm 11 NTRU DPKE Decryption

Input: $sk = (packed\_f, packed\_f_p, packed\_h_q)$ and $packed\_c$
Output: $r, m, fail$

1: if $c \not\equiv 0 \pmod{(q, \Phi_1)}$ return $(0, 0, 1)$
2: $c \leftarrow \mathsf{unpacked\_}\epsilon_q(packed\_c)$
3: $f \leftarrow \mathsf{unpacked\_}\epsilon_p(packed\_f)$
4: $a' \leftarrow (c \cdot f) \bmod (q, \Phi_1\Phi_n)$
5: $a \leftarrow \mathsf{R_q\_to\_S_3}(a')$
6: $f_p \leftarrow \mathsf{unpacked\_}\epsilon_p(packed\_f_p)$
7: $m \leftarrow (a \cdot f_p) \bmod (3, \Phi_n)$
8: $h_q \leftarrow \mathsf{unpacked\_}\epsilon_q(packed\_h_q)$
9: $m' \leftarrow \mathsf{Lift}(m)$
10: $r \leftarrow ((c - m') \cdot h_q) \bmod (q, \Phi_n)$
11: if (r, m) valid return $(r, m, 0)$ else return $(0, 0, 1)$

Algorithm 12 NTRU KEM Encapsulation

Input: $pk = packed\_h$ and $seed$
Output: $packed\_c$ and shared key K

1: $seed_{rm} \leftarrow \mathsf{SHAKE\text{-}128}(seed)$
2: $(r, m) \leftarrow \mathsf{Sample}(seed_{rm})$
3: $packed\_c \leftarrow \mathsf{PKE.Encrypt}(pk, (r, m))$
4: $packed\_rm \leftarrow (\mathsf{pack\_}\epsilon_\mathsf{p}(r), \mathsf{pack\_}\epsilon_\mathsf{p}(m))$
5: $K \leftarrow \mathsf{Hash}(packed\_rm)$

Algorithm 13 NTRU KEM Decapsulation

Input: $sk = (packed\_f, packed\_f_p, packed\_h_q, s)$ and $packed\_c$
Output: Shared key K

1: $(r, m, fail) \leftarrow \mathsf{PKE.Decrypt}((packed\_f, packed\_f_p, packed\_h_q), packed\_c)$
2: $packed\_rm \leftarrow (\mathsf{pack\_}\epsilon_\mathsf{p}(r), \mathsf{pack\_}\epsilon_\mathsf{p}(m))$
3: $k_1 \leftarrow \mathsf{H_1}(r, m)$
4: $k_2 \leftarrow \mathsf{H_2}(s, packed\_c)$
5: **if** $fail == 0$ **then**
6: $\quad K \leftarrow k_1$
7: **else**
8: $\quad K \leftarrow k_2$
9: **end if**

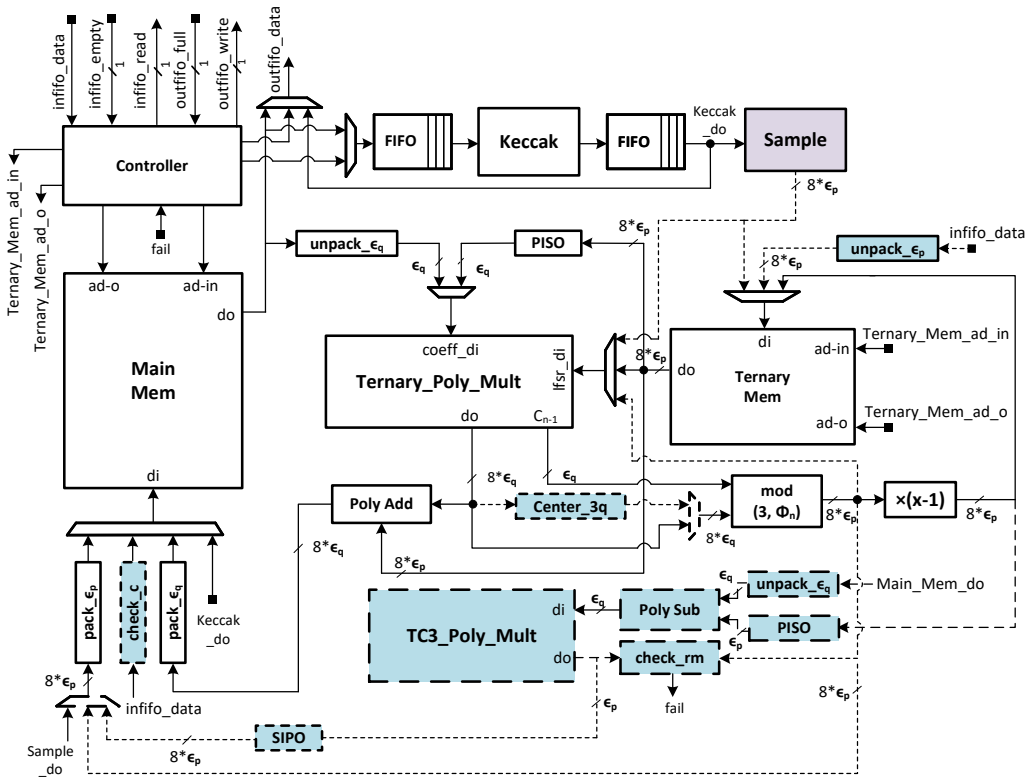

Figure 18: Top-level block diagrams of the Encapsulation and Decapsulation modules of NTRU. The purple, blue modules are used only in Encapsulation and Decapsulation, respectively.

In NTRU-HRSS, polynomial f, which is a part of the secret key, is required to have non-negative correlation property, $\sum_i f_i f_{i+1} \geq 0$. In NTRU-HPS, polynomial m in S_p has the fixed-weight property, consisting of $d/2$ coefficients equal to 1 and $d/2$ coefficients equal to -1, with $d = q/8 - 2$. Having the fixed-weight property of m ensures that the ciphertext $c \equiv 0 \pmod{(q, \Phi_1)}$ in NTRU-HPS. In NTRU-HRSS, in order to achieve the same property of c, m is lifted from S_3 to R_q by the map $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$.

The key generation, encryption and decryption of the PKE scheme of NTRU are shown in Algorithms 8, 10 and 11, respectively [20]. The IND-CCA2 NTRU KEM in Algorithms 9, 12 and 13, is based on the Saito-Xagawa-Yamakawa variant of the NTRU-HRSS KEM, with improvements that eliminate re-encryption during decapsulation. In the reference implementation of NTRU, the Sample function performs ternary sampling on random input, which requires kilobytes of random data per each operation of key generation or encapsulation. We chose to deviate from the reference implementation by using only 32-byte random input data and expanding it using SHAKE128. Sample generates polynomials in ternary form, which may have either an arbitrary or a fixed weight and/or non-negative correlation property. The top-level diagram of NTRU is shown in Fig. 18.

During key generation, two polynomial inversions are performed in $S_3 \pmod{(3, \Phi_n)}$ and $S_q \pmod{(q, \Phi_n)}$. To reduce space requirements, all coefficients of polynomials modulo q or p are packed together by unpack_ϵ_q and unpack_ϵ_p. Thus, they must be unpacked before being used in any operation. The Lift function lifts polynomial in S_3 to R_q. The most critical operation is polynomial multiplication in R_q (mod (q, Φ_n)). Other multiplication operations in S_3 or S_q can be performed by doing multiplication in R_q, followed by modulo

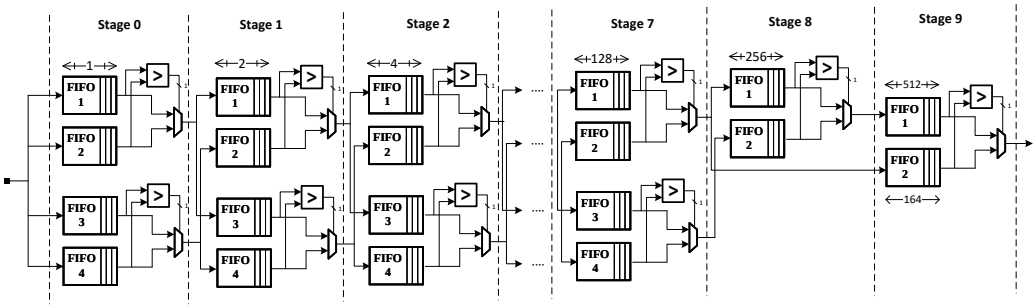

Figure 19: FIFO-based merge sort module for NTRUHPS2048677.

$(3, \Phi_n)$ or (q, Φ_n), respectively. During decryption, the ciphertext c is checked to determine if $c \equiv 0 \pmod{(q, \Phi_1)}$. As described in the specification [20], if c is unpacked by $\mathsf{unpack\_}\epsilon_{\mathsf{q}}$, we only need to check whether the unused bits of the final byte of c are all zeros. r and m are also needed to be checked if they are in the plaintext space, which means their coefficients are in the ternary form, and for NTRU-HPS, m must have the correct fixed weight.

B.2 Hardware Architecture

B.2.1 Ternary Sampling

For NTRU-HRSS, the generation of f and g is performed in S_3 during key generation. Random bytes coming from SHAKE128 are reduced modulo 3 to obtain the ternary coefficients stored in a first-in first-out (FIFO) unit. The sum of products of consecutive coefficients $s = \sum_i f_i f_{i+1}$ is computed at the same time. After finishing generating all coefficients, if $s < 0$, coefficients at even indices are signed-flipped before being transferred to the next computational stage. Thus, the non-negative correlation properties of f and g are satisfied. g is later multiplied by $x - 1$, which can be carried out trivially during the transfer. During encryption, r and m do not have either the non-negative correlation property or fixed-weight. They can be computed by simply reducing random data modulo 3.

For NTRU-HPS, f and r have arbitrary weight and can be sampled in a straightforward manner. However, m and g have fixed weight and are sampled by creating a random permutation of a list with a fixed number of values $-1, 0$ and 1. One can simply perform Fisher-Yates shuffle to have a random non-biased permutation of such a list. However, Fisher-Yates shuffle is not constant-time and creates a risk of potential timing attacks. Given that, we adopt a constant-time merge sorting approach for the permutation. The merge-sort module requires n random elements. Each element includes 30 random bits concatenated with "01" for the first $w/2$ elements, "10" for the next $d/2$ elements, and "00" for the rest. To get a 30-bit block, a 64-bit input is passed through a PISO, to be divided into two 32-bit blocks. Each 32-bit block is then processed using a buffer register and a variable shifter to get a 30-bit block. The leftover bits are stored in the buffer register to be concatenated with the subsequent output of PISO. After sorting, the upper 30 bits are discarded, and the lower 2 bits are converted from $\{0, 1, 2\}$ to $\{0, 1, -1\}$.

Related works: Wang et al. [77] proposed a fully pipelined constant-time merge sort module to generate random permutation in the Key Generation operation of Classic McEliece. To sort a random list of n elements, the module needs $log_2(n)$ iterations, where each step requires $O(n)$ comparison operations. Therefore, the total cycle count is approximately equal to $nlog_2(n)$ cycles. Marotzke [58] implemented an iterative Batcher's merge exchange sort module for a very similar sampling function in the Streamlined NTRU

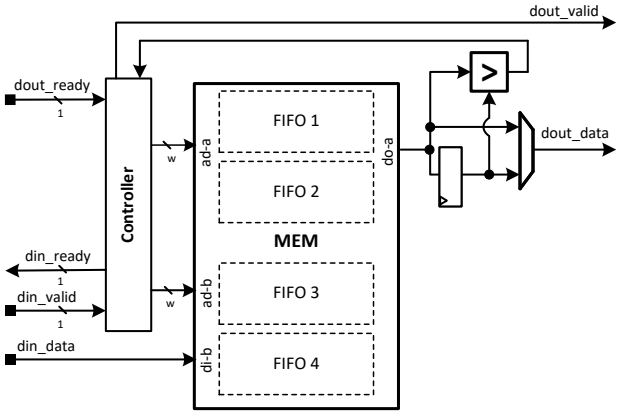

Figure 20: One stage of a FIFO-based merge sort module implemented using dual-port memory.

Table 15: Implementation Results of the FIFO-based Merge Sort module and comparison with related works.

	Freq.	LUT	FF	BRAM	DSP	Cycles
NTRUPRime: $n = 761$, $w = 32$, Zynq Ultrascale+						
Batcher's Merge Exchange Sort [58]	279	231	87	1.0	0	49,400
FIFO-based Merge Sort	250	1,441	940	3.5	0	2,762
ClassicMcEliece: $n = 8192$, $w = 45$, Zynq Ultrascale+						
4x Pipelined Merge Sort [78]	250	583	411	20.0	0	147,505
FIFO-based Merge Sort	250	2,533	1,589	33.0	0	26,646

Prime. Its operation also have asymptotic complexity of $O(nlog_2(n))$.

To speed up this operation, we use a merge-sort module consisting of $log_2(n)$ cascaded Sort Stages to sort the random sequences. The FIFO-based merge-sort module for NTRU-HPS677 is shown in Fig. 19. The inputs to each Sort Stage are two sorted lists, and the output is a sorted list of double input length, including all elements from the two input lists. Each input list is stored in a separate segment of memory. While the lower stages can be implemented by registers, the higher stages are implemented in dual-port memory. This approach can reduce the number of LUTs and FFs used to construct the large FIFO in higher stages at the cost of a small number of BRAMs. The internal structure of a Sort Stage is shown in Fig 20. By making use of the dual-port memory, the controller in each stage can write out the sorted list to the next stage and receive other input lists from the previous stage at the same time. By pipelining the operation of multiple Sort Stages, we can achieve a highly optimized latency for sorting. Our merge-sort module requires n clock cycles for reading n elements, roughly n cycles for sorting, and another n cycles to write out a sorted sequence.

The comparison of our FIFO-based merge sort module with previous work is shown in Table 15. We synthesize our module with the parameters used in [58] and [77]. Since the code of [77] is open-source, we can synthesize their merge-sort module targeting the same platform, Zynq Ultrascale+, and obtain results. Our FIFO-based merge sort module outperforms the previous designs by roughly an order of magnitude, excluding the time to load input and unload output. Although the increase in resource utilization is significant, it is still a quite compact design, suitable for high-speed applications that require random constant-time permutation.

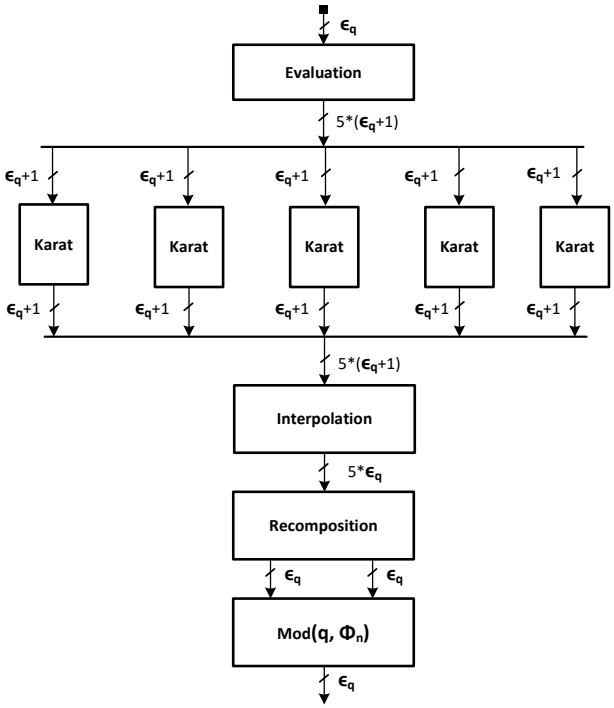

Figure 21: Toom-Cook 3 Polynomial Multiplier w/ Overlap-free Karatsuba.

B.2.2 Polynomial Multiplication

In all previous work on hardware implementations of NTRU, the polynomial multipliers always exploited the property of small ternary coefficients. The schoolbook multiplication has quadratic-complexity but enables simple, parallel, easy-to-parameterize, and very fast architecture for polynomial multiplication in NTRU. In [57], an efficient architecture based on Toom-Cook algorithm is proposed in a Software/Hardware codesign platform. Toom-Cook 4-way was applied to divide polynomial multiplication of 256 coefficients into seven multiplications with 64 coefficients. These seven multiplications are run in parallel using seven schoolbook polynomial multipliers.

In the AVX2 implementation of the NTRU submission package [64], a multi-layer Toom-Cook and Karatsuba is used to speed up the multiplication. In the recent work [21], an NTT-based polynomial multiplication is proposed, which outperforms the Toom-Cook method. However, the NTT-based polynomial multiplication was also applied to only multiplication with ternary polynomials. Therefore, it is not applied to speed up key generation and the final multiplication in decryption, which does not have any input polynomial in ternary form.

Toom-Cook Polynomial Multiplier. In this work, for multiplication without involving ternary polynomial, we implement a Toom-Cook 3-way polynomial multiplier, which splits an n-coefficient polynomial multiplication into five multiplications with $n/3$ coefficients. The five multiplications are performed in parallel using five Odd-Even Karatsuba multipliers. Our improvements over [57] include:

- Our implementation supports splitting input polynomials into three smaller polynomials before Evaluation step. The Toom-Cook core in [57] relies on software to do this operation.
- Using the Odd-Even Karatsuba method significantly improves the latency of the multiplication step.

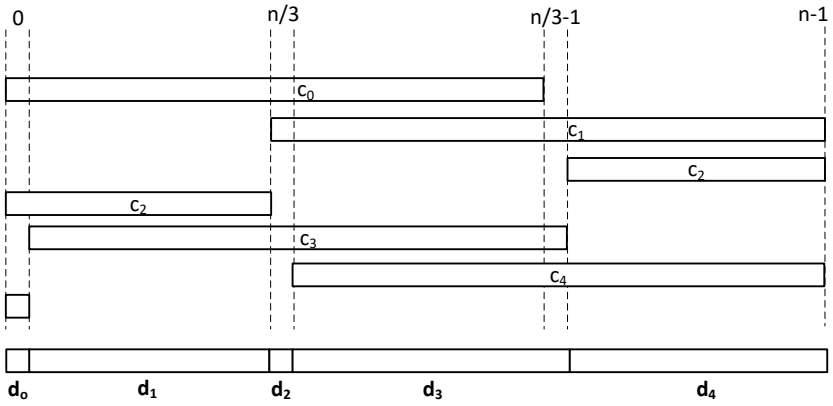

Figure 22: Recomposition in R_q.

- Our core supports Recomposition, which has the output polynomial in the ring R_q. In [57], 5 output polynomials are transferred to software and are then recomposed into a single polynomial.

Toom-Cook and Karatsuba are multiplication algorithms that have better asymptotic complexity compared to the schoolbook method. Toom-Cook k-way is a generalization of Karatsuba with $k = 2$. Both algorithms generally follow five steps: splitting, evaluation, pointwise-multiplication, interpolation, and recomposition. The input polynomials are split into $2k - 1$ polynomials with n/k coefficients. These polynomials are then evaluated at $2k - 1$ points. The evaluated polynomials are multiplied in the pointwise-multiplication steps. The results are interpolated as an opposite of the evaluation step. The output polynomials of the interpolation step are finally recomposed into the final product.

The top-level diagram of Toom-Cook 3-way module is shown in Fig. 21. Toom-Cook 3-way splits input polynomial $A(x)$ into three polynomial a_0, a_1 and a_2 such that $A(y) = a_0 + a_1 y + a_2 y$, where $y = \lceil n/3 \rceil$. a_0, a_1 and a_2 are then evaluated at five points $\{0, 1, -1, 2 \text{ and } \infty\}$. The pointwise multiplications are performed by Odd-Even Karatsuba modules. We adopt the optimal sequence for evaluation and interpolation in the Toom-Cook 3-way from Bodrato et al. [18]. We would like to highlight that during evaluation, there is a division by 2, which becomes a one-bit shift and causes a one-bit loss of precision. Therefore, the pointwise multiplication and interpolation steps require one extra bit for each coefficient.

After interpolation steps, we have 5 output polynomials $c_0, c_1, \ldots c_4$ with $2n/3$ coefficients needed to be recomposed and reduced modulo $x^n - 1$ in the ring R_q. Fig. 22 shows the positions of polynomials $c_0, c_1 \ldots c_4$ in the final product polynomial d modulo $x^n - 1$. Since the recomposition module receives five coefficients with the same index from c_0 to c_4, we need two registers d_0, d_2 and three shift registers of the size $\lfloor n/3 \rfloor - 1$. For example, d_0 will be initialized with the coefficient from c_0 at the cycle 0, then it is added to a coefficient from c_2 in the cycle $\lfloor n/3 \rfloor - 1$ and lastly added with the last coefficient from c_4 in the cycle $\lfloor 2n/3 \rfloor - 1$.

The overlap-free Karatsuba splits input polynomial $A(x)$ into two polynomials a_0 and a_1 such that $A(y) = a_0 + a_1 y$ where $y = x$. It means that a_0 consists of all even coefficients of A(x); meanwhile, a_1 consists of all odd coefficients of A(x). The overlap-free Karatsuba scheme enables a more efficient alignment of product coefficients compared to the classic Karatsuba scheme. The diagram of our overlap-free Karatsuba module is shown in Fig. 23. Two polynomials are stored in `RAM_a` and `RAM_b`. The multiplication between two coefficients from `RAM_a` and `RAM_b` would normally cost 12 integer multipliers. However, this number is reduced to 9 multipliers thanks to

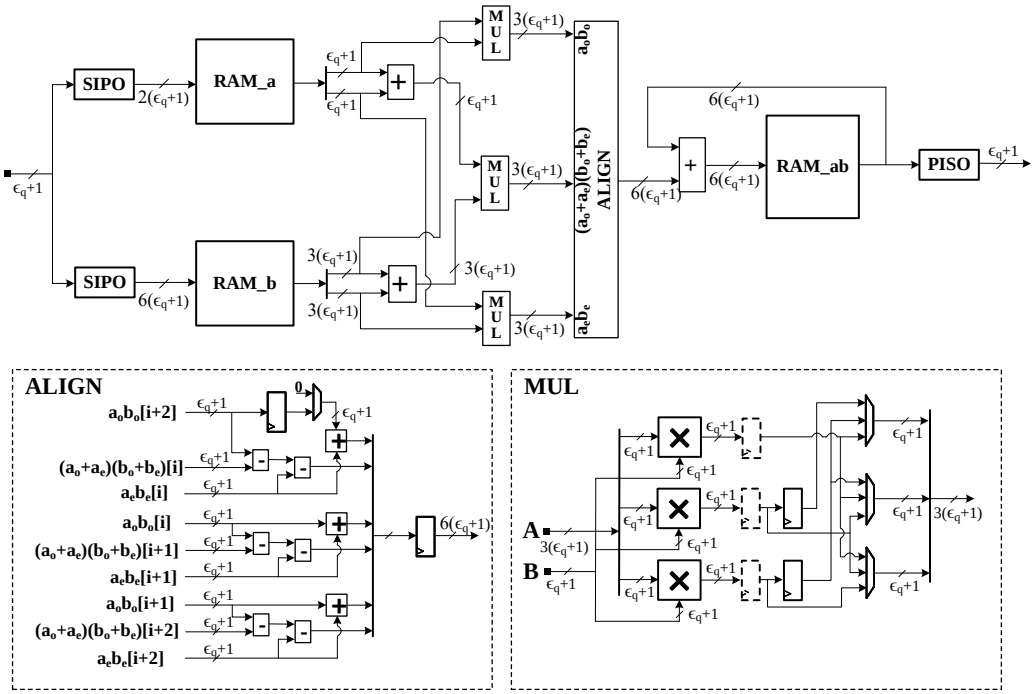

Figure 23: Overlap-free Karatsuba polynomial multiplier.

Table 16: Implementation results of the Toom-Cook 3 polynomial multiplier and comparison with related work for Saber with $n = 256$ and $q = 2^{13}$.

	Freq. [MHz]	LUT/FF/BRAM/DSP	Split/Eval. [cycles]	Mult. [cycles]	Inter./Recomp. [cycles]	Total [cycles]
Toom-Cook 3	130	3963/3389/0/45	174	688	174	1,036
Toom-Cook 4 [61]	125	2927/1279/2/28	Not Avail./128	1,168	128/Not Avail.	$> 1,424$

the Karatsuba algorithm. The latency of this module can be calculated as follows:

$$\text{Multiplication Latency} = (\tfrac{n}{6\times 3} + 1) \times (\tfrac{n}{6} + 1)$$

The comparison with the previous work in [57] is shown in Table 16. We synthesize our design for Saber with $n = 256$ and $q = 2^{13}$ on the Xilinx ZedBoard Zynq-7000. The recomposition module is also adjusted to support the ring $\mathbb{Z}_q[x]/(x^n + 1)$. The increase in the number of LUTs and FFs comes partially from the large shift registers used in splitting and recomposition steps. We note that the splitting and recomposition steps are merged into evaluation and interpolation, respectively. Our Toom-Cook multiplier finishes one polynomial multiplication in R_q or S_q in 5507, 5098 and 7274 cycles for $n = 701, 677$ and 821, respectively.

Ternary Polynomial Multiplier. For multiplications involving polynomial in the ternary form $\{-1, 0, 1\}$, we use the constant-time LFSR-based polynomial multiplier, proposed in [28], which has the latency of n clock cycles. By loading the ternary polynomial with coefficients in $\{-1, 0, 1\}$ to the LFSR, instead of a polynomial with "big" coefficients, we reduce the number of flip-flops required to realize this LFSR by a factor of four. We also shorten the time required to load a polynomial into the LFSR, since eight 2-bit coefficients can be loaded in a single clock cycle. All integer multiplication-and-accumulation operations between coefficients of two operands and one product polynomials are reduced to addition, pass-through, or subtraction. The LFSR is initialized to a polynomial with ternary coefficients. Let us denote the initial state of this LFSR as $a(x)$. In each subsequent

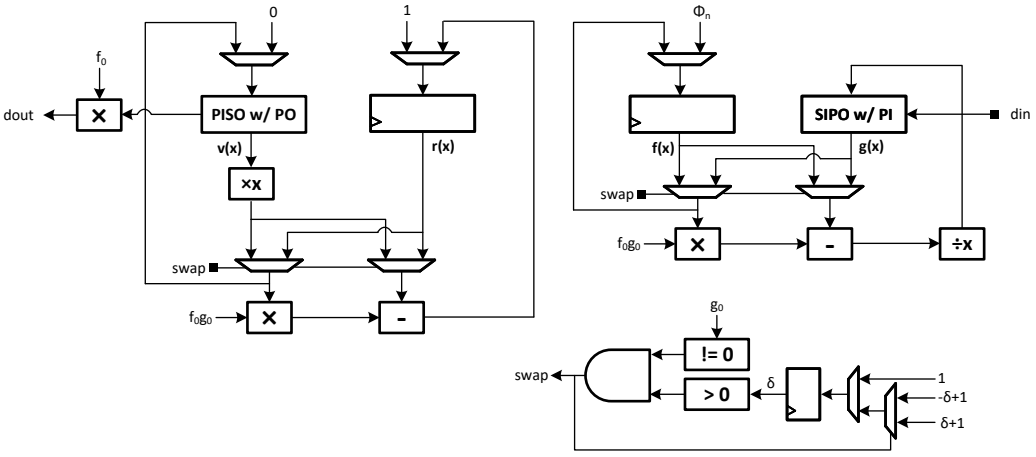

Figure 24: S_2/S_3 Inversion Module

Table 17: Implementation results of the Extended GCD module and comparison with related work for Streamlined NTRU Prime in Zynq Ultrascale+ platform.

	Freq.	LUT	FF	BRAM	DSP	Cycles
Extended GCD w/ $n = 761$ [58]	271	518	216	0	0	1,168,899
Extended GCD w/ $n = 821$	250	8,534	5,479	0	0	1,846

iteration, the output from LFSR contains the value $a(x) \cdot x^i \bmod x^n - 1$. In a single clock cycle, a simple multiplication by x, namely $a(x) \cdot x^{i+1} \bmod x^n - 1 = a(x) \cdot x^i \cdot x \bmod x^n - 1$, is performed.

B.2.3 Inversion in S_3 and R_q

Inverse of polynomials in R_q and S_3 plays an important role in key generation. We need to compute f_p which is an inverse of f in S_3 for the secret key. Computation of v_1, which is an inverse of v_0 in S_q, must be completed before any later operations could proceed.

Inversion in S_3: Inversion in S_3 is done using the constant-time extended Greatest Common Divisor (GCD) unit proposed in [17]. The top-level diagram of our `S3_inverse` module is shown in Fig. 24. At first, $g(x)$ is initialized with an input polynomial in reverse order. $f(x)$, $r(x)$ and v(x) are initialized with Φ_n, 1 and 0 respectively. The module runs in exactly $2(n-1)$ cycles. All coefficients of four polynomials are updated simultaneously during each iteration according to the value of δ and g_0. All operations, including addition, subtraction, and multiplication, are reduced modulo 3. Multiply and divide by x are performed by simple bit shifting. Lastly, the inverse of input polynomial is $f_0 \times v(x)$. We note that the inverse polynomials are also stored in the reverse order. Our module also supports inversion in S_2, which is used in inversion in S_q. We compare our results for NTRU-HPS821 with $n = 821$ with the Reciprocal in $R/3$ module in the implementation of Streamlined NTRU Prime in [58]. We have shown that the extended GCD can be implemented in an unrolled fashion, achieving highly optimized latency.

Inversion in R_q: To compute the inverse of h in S_q, we perform $h^{-1} \bmod (2, \Phi_n)$ and then apply a variant of the Newton iteration in R_q to obtain $h_q \equiv h^{-1} \bmod (q, \Phi_n)$. The pseudocode of inversion in R_q is given in Algorithm 14. A similar approach is presented in [48], which finds an inverse mod $(2, \Phi_n)$ using $h^{-1} \equiv h^{2^{n-1}-2} \bmod (2, \Phi_n)$. Given that squaring operation in $\mathbb{Z}_2[x]$ is particularly very efficient in software, this approach is suitable for software implementation. In our case, we can re-use our `S3_inverse` module

Algorithm 14 Polynomial Inversion in S_q [48]

Input: Polynomial a in S_q
Output: Polynomial b in S_q such that $a \cdot b = 1 \bmod (q, \Phi_n)$

1: $v_0 \leftarrow a^{-1} \bmod (2, \Phi_n)$
2: $i \leftarrow 1$
3: **while** $i < logq$ **do**
4: $\quad v_0 \leftarrow v_0 \cdot (2 - a \cdot v_0)$
5: $\quad i \leftarrow 2i$
6: **end while**
7: $b \leftarrow v_0$

Algorithm 15 Lift in NTRU-HRSS [48]

Input: Polynomial v in S_3
Output: Polynomial $b = \Phi_1((v/\Phi_1) \bmod (3, \Phi_n)) \bmod (q, \Phi_1\Phi_n)$

1: $z = [1/\Phi_1] \bmod (3, \Phi_n) = \sum_{i=0}^{n-2} (1 - i) \cdot x^i \pmod 3$
2: $a = vz \bmod (q, \Phi_1\Phi_n)$
3: **for** $i = 0$ to $n - 1$ **do**
4: $\quad a_i = a_i - a_{n-1} \pmod 3$ $\quad \triangleright\ a = v/\Phi_1 \bmod (3, \Phi_n)$
5: **end for**
6: $b_0 = a_{n-1} - a_0 \pmod q$
7: **for** $i = 1$ to $n - 1$ **do**
8: $\quad b_i = a_{i-1} - a_i \pmod q$ $\quad \triangleright\ b = \Phi_1((v/\Phi_1) \bmod (3, \Phi_n)) \bmod (q, \Phi_1\Phi_n)$
9: **end for**

to compute inversion in S_2. All arithmetic operations are now reduced modulo 2 instead of 3 as in inversion in S_3. Operations from line 3 to 6 in Algorithm 14 are equivalent to 8 polynomial multiplications, which are performed by the Toom-Cook multiplier. Due to the long latency of the polynomial multiplication, inversion in R_q is the most time-consuming operation in Key Generation of NTRU, taking up to 90% of total latency.

B.2.4 `Lift` function in NTRU-HRSS

In NTRU-HRSS, the `Lift` function maps m from S_3 to R_q by doing $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$. An efficient implementation of `Lift` is shown in Algorithm 15. As shown in the pseudocode, `Lift` function can be performed by one multiplication with $z = 1\Phi_n$ then followed by reduction modulo $(3, \Phi_n)$ and lastly multiplied by Φ_1. Since z is a constant ternary polynomial, it is stored in the memory and the multiplication can be performed by the `Ternary_Poly_Mult` in n cycles. Reduction modulo $(3, \Phi_n)$ and multiplication by $\Phi_1 = x - 1$ can be performed on-the-fly while transferring result back to the memory.

C Saber

C.1 Algorithms

Saber uses a version of the Fujisaki-Okamoto transformation from an IND-CPA public-key encryption scheme to construct an IND-CCA KEM. By using only moduli that are powers of 2, modular reduction and rejection sampling are eliminated. A distinctive feature of Saber compared to LWE schemes is that rounding operations are used to avoid the noise addition step and reduce the amount of randomness required. In this paper, we implemented all three parameter sets of Saber: LightSaber, Saber, and FireSaber, corresponding to security levels 1, 3, and 5, respectively.

Definitions and Parameters: Saber involves operations on matrices and vectors of polynomials over the quotient rings $R_q : \mathbb{Z}_q[x]/(x^n + 1)$ with fixed $n = 256$. Polynomials in Saber are sampled from the uniform distribution or centered binomial distribution. β_μ denotes a centered binomial distribution with the parameter μ and the values of samples in the range $[-\mu/2; \mu/2]$. The module dimension l defines the size of vectors and matrices of polynomials as $l \times 1$ and $l \times l$, respectively. We denote $R_q^{l\times l}$ and $R_q^{l\times 1}$ as a matrix and vector of polynomials in R_q. The rounding operation includes coefficient-wise addition with a constant factor and is followed by bit shifting.

The pseudocode of Saber is shown as Algorithms 16, 17, 18, 20, 19, and 21. The KEM key generation includes sampling uniformly random matrix A using SHAKE128. Secret vector s is sampled in binomial distribution from the uniformly random output from SHAKE128. The vector product of $A^T s$ is rounded and served as a public vector b in the public key. The secret key includes the public key, hash of the public key, secret vector s, and a pseudo-random byte string z, which is used for implicit rejection in FO transform.

Encapsulation includes encryption with additional hashing. A "small" vector s is generated using sampling from the centered binomial distribution. The ciphertext has two parts. The first part has the rounded product of As'. The second one includes the sum of the inner product of b, s', and the encoded message m. We adopt the optimization in [83] to compute $b^T s'$ before As'. Since the generation of s' and A requires the same SHAKE128 function, we would need to finish generating s' before performing As' with the on-the-fly generation of A. The multiplication of b^T and s' can be performed in parallel with the sampling of s'. The shared secret is derived from the hashes of the public key, message, and ciphertext. Decapsulation involves decryption and re-encryption. During decryption, the secret key is used to compute v, which is used to extract the message. The obtained message is then re-encrypted to check whether the re-encrypted ciphertext is the same as the received one. To save bandwidth, all coefficients of polynomials modulo q or rounded to p or T are packed together by $\mathsf{pack\_\epsilon_q}$, $\mathsf{pack\_\epsilon_p}$ or $\mathsf{pack\_\epsilon_T}$. Thus, they must be unpacked before being used in any operation. The top-level block diagram is shown in Fig. 25.

Algorithm 16 Saber PKE Keypair

Input: $seed_A$ and $seed_s$
Output: $pk = (seed_A, packed\_b), sk = (s)$

1: $seed_A \leftarrow \mathsf{SHAKE\text{-}128}(seed_A)$
2: $A \leftarrow \mathsf{Unpack\_\epsilon_q}(\mathsf{SHAKE\text{-}128}(seed_A)) \in R_q^{l\times l}$
3: $s \leftarrow \mathsf{Sample}(\mathsf{SHAKE\text{-}128}(seed_s)) \in R_q^{l\times 1}$
4: $sk \leftarrow s$
5: $b \leftarrow \mathsf{Round_{qp}}(A^T \cdot s) \in R_p^{l\times 1}$
6: $pk \leftarrow (seed_A, \mathsf{Pack\_\epsilon_p}(b))$

Algorithm 17 Saber KEM Keypair

Input: $seed_A$, $seed_s$ and z.
Output: $pk = (seed_A, packed\_b), sk = (z, pkh, pk, s)$

1: $seed_A, packed\_b, s \leftarrow \mathsf{PKE\_Keypair}(seed_A, seed_s)$
2: $pk \leftarrow (seed_A, packed\_b)$
3: $pkh \leftarrow \mathsf{SHA3\text{-}256}(pk)$
4: $sk \leftarrow (z, pkh, pk, s)$

Table 18: Implementation results of the Optimized Polynomial Multiplier using optimized integer multipliers vs. the centralized multiplier architecture in [14]

	Optimized Multiplier	Centralized Multiplier
LightSaber	**12,492 LUTs, 8,727 FFs**	13,658 LUTs, 8,727 FFs
Saber	12,492 LUTs, 8,727 FFs	**11,426 LUTs, 8,727 FFs**
FireSaber	**8,726 LUTs, 8,215 FFs**	8,734 LUTs, 8,215 FFs

Algorithm 18 Saber PKE Encryption

Input: $pk = (seed_A, packed\_b)$, m and $seed_{s'}$
Output: $c = (packed\_c_m, packed\_b')$

1: $s' \leftarrow \mathsf{Sample}(\mathsf{SHAKE\text{-}128}(seed_{s'})) \in R_q^{l\times 1}$
2: $A \leftarrow \mathsf{Unpack\_}\epsilon_q(\mathsf{SHAKE\text{-}128}(seed_A)) \in R_q^{l\times l}$
3: $b' \leftarrow \mathsf{Round_{qp}}((A \cdot s' + h) \bmod q) \in R_p^{l\times 1}$
4: $packed\_b' \leftarrow \mathsf{Pack\_}\epsilon_\mathsf{p}(b')$
5: $b \leftarrow \mathsf{Unpack\_}\epsilon_\mathsf{p}(packed\_b)$
6: $v' \leftarrow b^T \cdot (s' \bmod p) \in R_p$
7: $c_m \leftarrow \mathsf{Round_{pT}}(v' + h_1 - 2^{\epsilon_p - 1} \cdot m \bmod p) \in R_T$
8: $packed\_c_m \leftarrow \mathsf{Pack\_}\epsilon_\mathsf{T}(c_m)$
9: $c \leftarrow (packed\_c_m, packed\_b')$

Algorithm 19 Saber PKE Decryption

Input: $sk = packed\_s$ and $c = (packed\_c_m, packed\_b'))$
Output: m

1: $s \leftarrow \mathsf{Unpack\_}\epsilon_\mathsf{q}(packed\_s) \in R_q^{l\times 1}$
2: $b' \leftarrow \mathsf{Unpack\_}\epsilon_\mathsf{p}(packed\_b') \in R_p^{l\times 1}$
3: $v \leftarrow b'^T \cdot s \bmod p \in R_p$
4: $m' \leftarrow \mathsf{Round_{p2}}(v + h_2 - 2^{\epsilon_p - \epsilon_T} \cdot c_m \bmod p) \in R_2$

Algorithm 20 Saber KEM Encapsulation

Input: $pk = (seed_A, BS\_b), m$
Output: $c = (packed\_c_m, packed\_b'))$ and a shared key K

1: $(\hat{K}, r) \leftarrow \mathsf{SHA3\text{-}512}(\mathsf{SHA3\text{-}256}(pk), \mathsf{SHA3\text{-}256}(m))$
2: $c \leftarrow \mathsf{Saber.PKE.Enc}(pk, m, r)$
3: $h\_c \leftarrow \mathsf{SHA3\text{-}256}(c)$
4: $K \leftarrow \mathsf{SHA3\text{-}256}(\hat{K}, h\_c)$

Algorithm 21 Saber KEM Decapsulation

Input: $sk = (packed\_s, z, pkh, pk = (seed_A, packed\_b))$ and $c = (packed\_c_m, packed\_b'))$
Output: Shared key K

1: $m' \leftarrow \mathsf{Saber.PKE.Dec}(packed\_s, c)$
2: $(\hat{K}', r') \leftarrow \mathsf{SHA3\text{-}512}(pkh, m')$
3: $c' \leftarrow \mathsf{Saber.PKE.Enc}(pk, m', r')$
4: $h\_c \leftarrow \mathsf{SHA3\text{-}256}(c)$
5: **if** $c = c'$ **then**
6: $\quad K \leftarrow \mathsf{SHA3\text{-}256}(\hat{K}', h\_c)$
7: **else**
8: $\quad K \leftarrow \mathsf{SHA3\text{-}256}(z, h\_c)$
9: **end if**

C.2 Hardware Architecture

C.2.1 Sampling

The diagram of our CBD sampling modules for three parameter sets of Saber is shown in Fig. 26. The values of coefficients sampled from CBD are in the range [-5; 5], [-4; 4], and [-3;3], corresponding to the bit-width $w = 4, 4, 3$. The 64-bit inputs are buffered in the dual-step shift register. After the shift register is full, chunks of data are read out and fed through a pure combinational logic to generate the coefficients. The output width of sampling modules is equal to $8 * w$. Therefore, we will have 8 samples generated per clock cycle.

C.2.2 Polynomial Multiplication

The high-speed SW/HW codesign of Saber in [24] uses a schoolbook-based multiplier, which requires 256 DSPs with 13-bit inputs. A Toom-Cook based multiplier for Saber is proposed in [57], also in the SW/HW co-design context. The Saber crypto-processor implementation in [75] uses a schoolbook-based multiplier, which exploits the small sizes of input coefficients. It can provide very good performance with moderate resource

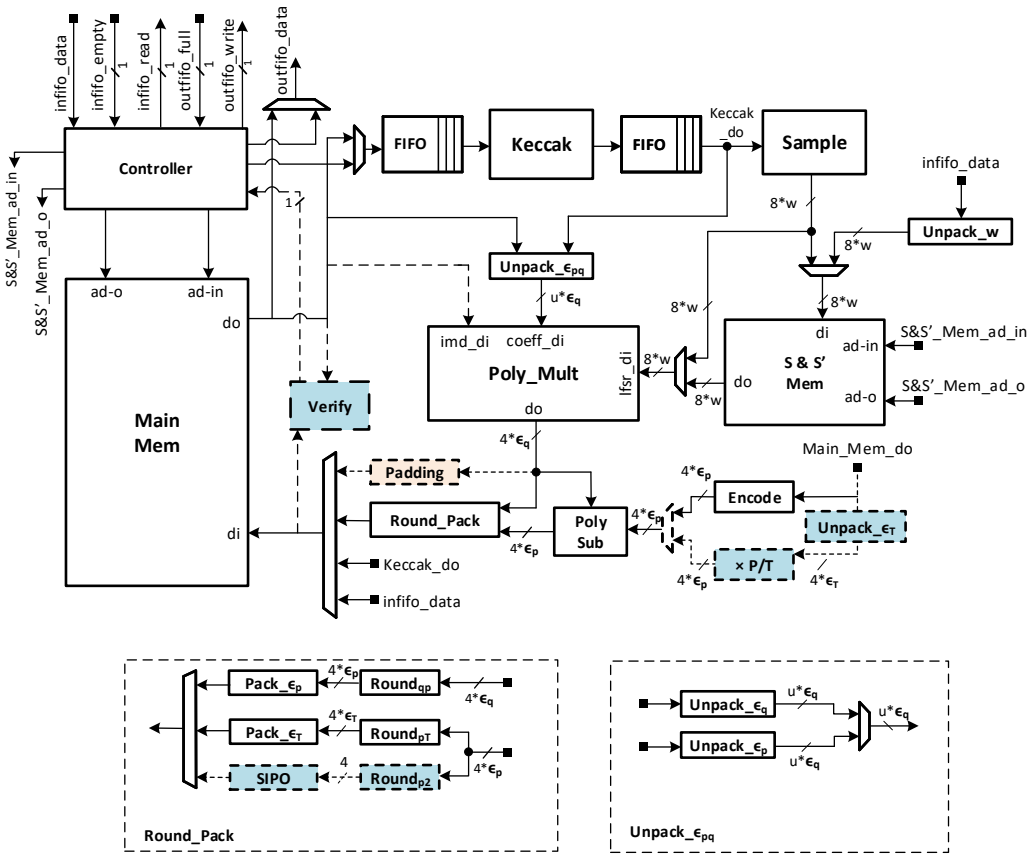

Figure 25: Top-level block diagrams of Saber. The orange, blue modules are used only in key generation and decapsulation, respectively.

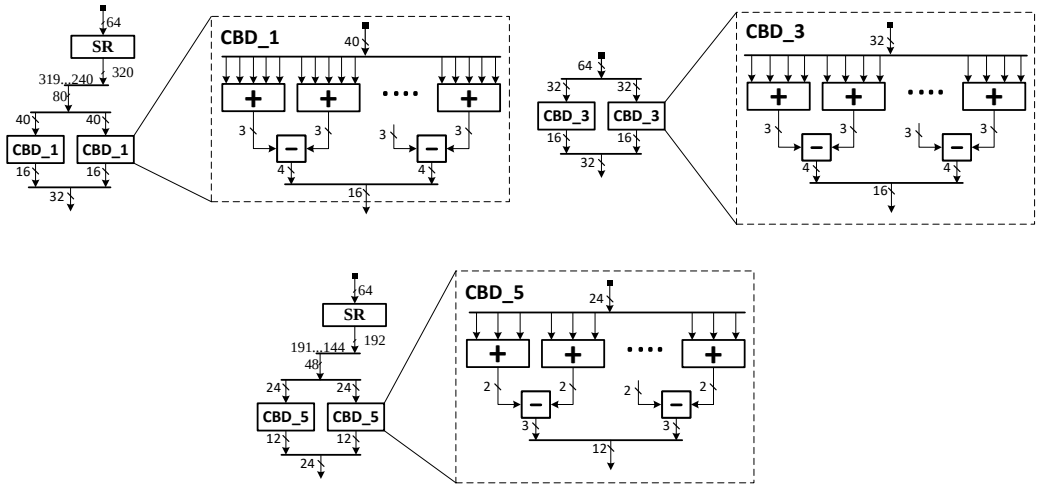

Figure 26: Sampling modules for three parameter sets of Saber.

consumption. [14] improves the multiplier used in [75] by centralizing coefficient-wise multiplication and replacing integer multipliers with simple multiplexers. RISC-V proposed an approach to use an NTT module to speed-up polynomial multiplication in Saber based

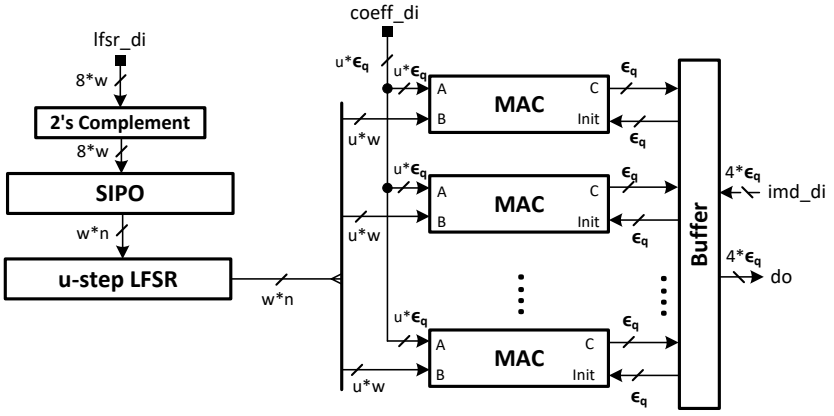

Figure 27: Schoolbook-based polynomial multiplier with unroll factor $u = 1, 2, 4$.

on the Chinese Remainder Theorem. Recently, [83] introduced an 8-level Karatsuba multiplier for Saber with efficient scheduling of operations, which achieves very small latency in terms of clock cycles. However, it requires a large area and a long critical path, which leads to low clock frequency.

For our high-speed application of Saber, we opt to use the schoolbook-based multiplier as shown in Fig. 27. Since there are multiple multiplications involved in vector-vector or matrix-vector multiplication, we improve the latency of multiplication by adding input and output buffers. The buffers are capable of pre-loading the next input polynomial as well as unloading the previous product polynomial at the same time as the current multiplication is performed. The `S&S'MEM` stores all small coefficients of secret polynomials in their unpacked form. Thus, it can provide one polynomial in 32 cycles. The latencies of loading and unloading polynomials are hidden in the multiplication latency. The multiplier can also be unrolled by a factor $u = 1, 2$, or 4, which can finish one polynomial multiplication in 256, 128 or 64 cycles, respectively. Instead, having simple integer coefficient-wise multipliers, which are based on shift-add operations, as in [75], we generated optimized integer multipliers using an open-source tool FloPoCo [25]. We also tried the centralized coefficient-wise multiplier approach proposed in [14]. We report the results of the two approaches in Table 18. The centralized multipliers approach has better area consumption in the case of Saber, so we use this approach for the specific parameter set. For LightSaber and FireSaber, the optimized integer multipliers are used.

C.2.3 Improvements over Previous Work

Compared to the previous work on the implementations of Saber, reported in [74] and [83], we further optimize the schoolbook multiplier. Additionally, we optimize the scheduling of all operations in hardware by fully exploiting the potential for parallel processing of operations without data dependencies. Our implementation achieves the best latency and the usage of LUTs.