RUHR-UNIVERSITÄT BOCHUM

Efficient Hardware Implementation of Post-Quantum Cryptography using High-Level Synthesis

Fabian Buschkowski

Master's Thesis. January 10, 2021.
Chair for Embedded Security – Prof. Dr.-Ing. Tim Güneysu
Advisor: Georg Land

Abstract

The implementation of Post-Quantum Cryptography schemes in hardware becomes increasingly important with the NIST post-quantum standardization process coming closer to its end. Implementing in hardware can be a difficult and time-consuming task, and late changes to the design can require an enormous amount of work. The concept of High-Level Synthesis offers a quicker way of creating a hardware implementation by letting a tool transform high-level language code into equivalent hardware description language code. However, the quality of resulting implementations has barely been studied so far.

In this work, the use of High-Level Synthesis in the development of hardware implementations of Post-Quantum Cryptography is evaluated, and general strategies for hardware implementations using High-Level Synthesis are developed. For this purpose, High-Level Synthesis is performed on four important components of lattice-based cryptography and the FrodoKEM key encapsulation scheme, and the resulting implementations are compared to existing hardware implementations. Results indicate that High-Level Synthesis can be a very helpful tool for the implementation of Post-Quantum Cryptography in hardware by speeding up the development process while the created implementations perform similarly to the direct hardware implementations.

Keywords: Post-quantum cryptography, lattices, Frodo, FPGA, High-Level Synthesis, hardware

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Statutory Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of High School.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

10.01.2021
Datum / Date

Fabian Buschkowski
Unterschrift / Signature

Contents

1 Introduction

The research field of quantum computers has seen numerous advances in recent times. While a major breakthrough, allowing to build large-scale quantum computers with hundreds or thousands of qubits, has not yet been achieved, some scientists expect large-scale quantum computers to be available within the next two decades. Once they are available, it will be possible to solve the previously hard mathematical problems of factorization and discrete logarithms. These two problems are the basis of many asymmetric cryptographic schemes currently in use, especially the RSA encryption and the Diffie-Hellman Key Exchange.

As a consequence, the National Institute of Standards and Technology (NIST) has started a competition in 2017 with a call for proposals [NIS20], asking to submit asymmetric encryption and signature schemes that can withstand attacks with a classical and a quantum computer. Over 60 candidates were submitted to the first round of the competition and then reviewed and analysed regarding their security, execution speed and flexibility. Many candidates were eliminated from the competition, and as of 2020, the competition is in the third and final round with 4 encryption and 3 signature schemes remaining. In the end, one or several winners will be standardized to replace the old Public-Key Infrastructure (PKI). The submitted candidates make use of very different approaches to achieve resistance to quantum computers, such as lattices, codes, hash functions or multivariate polynomials. Among the different approaches, lattice-based cryptography seems to be the most popular and promising approach, with the majority of the submitted candidates being based on lattices.

Lattice-based cryptography schemes can be grouped into three classes, depending on the underlying lattice-problem: Encryption schemes such as FrodoKEM [ABD+19a] or Round5 [BBF+19] are based on the standard Learning With Errors (LWE) problem, resulting in superior security due to the lack of any structure and scalability at the cost of decreased efficiency due to large parameters, keys and ciphertexts. In contrast, schemes based on the Ring-LWE problem such as NewHope [AAB+19] or NTRU [CDH+19] offer better performance as well as smaller keys and ciphertexts than those based on standard lattices. However, it is unclear whether a quantum computer might be able to exploit the additional structure to break the cryptosystem. The NTRU encryption scheme is one of the 4 remaining candidates in the third round of the NIST competition. The third class of lattice-based cryptosystems is based on the Module-LWE problem and offers a trade-off between the security of unstructured lattices and the performance of fully structured lattices. Examples of encryption schemes from this class are CRYSTALS-Kyber [ABD+19b] or SABER [BBMD+20], both of them candidates in the third round of the competition.

While a software implementation in the high-level language `C` exists for all of the submitted

candidates, a hardware implementation exists for only a few schemes such as FrodoKEM or NewHope. Software implementations are executed on a processor and can easily be ported between different types of processors. In contrast, hardware implementations can be executed using Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). They are usually written in a Hardware Description Language (HDL) such as VHDL or Verilog. Compared to high-level languages, HDLs offer new features such as timing and the concurrent (parallel) execution of code. The design process for a hardware implementation typically takes longer than for a software implementation for a number of reasons: First of all, code written in a HDL is generally longer than equivalent `C`-code. Multiple thousand lines of code for a cryptography scheme with all required sub-modules is not a rare sight in hardware, whereas less than a thousand lines of code may suffice for a software implementation of the same scheme. In addition, the time required to verify the correctness of the implementation, that is to execute the code, is significantly longer for a hardware implementation. While the software implementation can be executed within a few microseconds to seconds, the hardware implementation simulated on a processor can take several minutes. Finally, late changes to the design of the hardware implementation can be a very time-consuming task as many lines of code may have to be changed, potentially inducing unrecognised errors in the code. In contrast, late changes to a software implementation can be relatively easy, depending on the type of change, by exchanging only one or a few functions.
To reduce the long development times of hardware implementations, two possible approaches have emerged: Hardware/Software (HW/SW) Co-Design and High-Level Synthesis (HLS). In Hardware/Software Co-Design, both software and hardware are used to implement the desired functionality. The majority of tasks is implemented in software, but the remaining tasks, usually the most time-consuming ones, are implemented and executed in hardware, allowing a significant speed-up in execution time over a pure software implementation, and a shorter design process compared to a pure hardware implementation. In HLS, the code of a software implementation is used as input to a High-Level Synthesis tool that transforms the given code into equivalent HDL code, resulting in a pure hardware implementation with the same functionality as the software implementation. The main advantage of this approach is the potentially short design process since the code can be written in `C` or a different high-level language, and the potentially high performance of the resulting pure hardware implementation. However, the quality regarding speed and size of the resulting hardware implementations has barely been studied so far, and it is unclear whether hardware implementations created by HLS can achieve a similar performance as those written directly in an HDL or not.
This work aims at evaluating the possibilities of using HLS to create hardware implementations of Post-Quantum Cryptography (PQC), and their quality regarding execution speed and hardware utilization. For this purpose, software implementations of different components of lattice-based cryptography such as polynomial multiplication, random sampling or Pseudo-Random Number Generators (PRNGs) are processed into an equivalent hardware implementation using HLS, and then compared to direct hardware implementations of these components. In a second step, the existing software implementation of the FrodoKEM encapsulation scheme is synthesized using HLS and again compared

to the existing hardware implementation of this scheme. Based on the results of the comparisons, general strategies to achieve the best possible implementations with HLS are described.

1.1 Related Work

Various works have evaluated implementations of PQC schemes on FPGAs. However, this research field is still very limited as these schemes usually require a large amount of resources. Howe et al. implemented the FrodoKEM key encapsulation scheme on an FPGA in [HOKG18] using a balanced approach between hardware utilization and latency. Güneysu et al. implemented the NewHope-Simple Key Exchange in [OG19]. Unlike FrodoKEM, NewHope-Simple is based on ideal lattices, allowing more efficient implementations. In [BFM+18], the authors present a hardware implementation of the NTRU encryption scheme, also based on ideal lattices, on an FPGA. PQC schemes based on other classes of problems have been implemented on FPGAs as well. The code-based encryption scheme McEliece is implemented on an FPGA in [LGCN20]. A key exchange scheme based on supersingular isogenies has been implemented on an FPGA by Koziel et al. [KAMK16].
The use of HLS for the implementation of cryptography in hardware has been studied in only a few works. In [HG15], the authors evaluate the use of HLS on 5 candidates of the SHA-3 contest. Their study shows a noticeable penalty in performance compared to the direct implementations. Analysing HLS on the candidates of the CAESAR contest, the authors come to very similar conclusions [HG17]. Farahmand et al. have evaluated the use of HW/SW Co-Design and HLS on the NTRU encryption scheme [FND+19]. Their results indicate that HLS and HW/SW Co-Design can be very useful tools to reduce the development time of a hardware implementation by allowing quick implementation of functionalities that have little influence on the total latency. In [DFA+20], the authors evaluate and compare the hardware implementations of many Round 2 candidates and analyse possible speed-ups in the design process using HW/SW techniques and HLS.

1.2 Contribution

In this work, the possible use of HLS in the development of PQC hardware implementations is evaluated by means of comparing already existing direct hardware implementations with HLS implementations made in this work. In addition, general strategies to achieve good results with HLS on PQC are developed. The HLS versions of four components that play an important role in PQC – Number-Theoretic Transform (NTT), polynomial multiplication, the SHAKE hash function and a Gaussian sampler – achieve very similar performances as the direct implementations in terms of hardware usage, latency and frequency.
The HLS implementation of FrodoKEM developed in this work is on par or surpasses the direct implementation regarding the hardware usage, but suffers a penalty on the latency of no more than 3 % compared to the direct implementation. The frequencies of

both implementations are identical for the key generation and the encapsulation, and 20 MHz lower in the HLS version for the decapsulation. All HLS implementations are fully compliant with the software implementations they are based on to ensure their compatibility.
The results of this work show that HLS can be a highly useful tool in the development of hardware implementations as it allows to quickly implement simpler functions and more complex designs with a performance comparable to that of direct implementations.

2 Background

This chapter provides some background on PQC in general and lattice-based cryptography in more detail, FPGAs and their components, High-Level Synthesis, and the FrodoKEM encryption scheme.

2.1 Post-Quantum Cryptography

The term Post-Quantum Cryptography refers to cryptographic schemes that are intended to be secure against attacks with a normal and a quantum computer. To achieve quantum security, these schemes no longer rely on currently hard problems like factoring and discrete logarithms since these can be efficiently solved with a large enough quantum computer. Instead, these schemes rely on new approaches with underlying problems that are believed to be hard even for a quantum computer. Among the candidates for the NIST PQC competition, four different approaches were used to construct quantum-resistant cryptography schemes:

- Code-Based Cryptography: Cryptography schemes based on codes such as the McEliece cryptosystem [BCL+19] rely on the hardness of decoding a random linear code [OS09]. They typically have large public keys and small ciphertexts, and the underlying problems are rather well understood and analysed. So far, only Key Encapsulation Mechanisms (KEMs) have been built on this approach.
- Hash-Based Cryptography: Hash-based schemes rely on the hardness of finding a pre-image of a hash function. Among the candidates for the second round, only the signature scheme SPHINCS+ [ABB+20] chose this approach. The signature schemes XMSS [BDH11] and LMS [MCF] have already been standardized by the NIST.
- Multivariate-Quadratic: These schemes make use of the hard problem of finding the solution for a set of multivariate-quadratic systems over a finite field. So far, only signature schemes have been constructed using this approach. They have very small signatures and fast execution time at the cost of large public keys, and their security has been barely analysed so far.
- Lattice-Based Cryptography: Lattice-based schemes are the most popular approach among the candidates in the second round. The underlying problems rely on the difficulty of finding the closest or shortest vector in a lattice, which, up to date, has not been efficiently done using a classical or quantum computer. Both encryption and signature schemes can be constructed using this approach.

2.1.1 Lattices

A lattice $\mathcal{L}$ is the linear combination of independent vectors $\mathbf{b}_1, ..., \mathbf{b}_n$ that form a basis $\mathbf{B}$. The lattice can be written as $\mathcal{L} = \sum_{i=1}^{n} c_i \cdot b_i$ for $c_i \in \mathbb{Z}$, that is the linear combinations of the basis vectors. The minimum distance λ_1 of $\mathcal{L}$ is the length of the shortest non-zero vector in the lattice: $\lambda_1(\mathcal{L}) := \min_{v \in \mathcal{L} \setminus \{0\}} ||v||$, where $||v||$ denotes the Euclidean Norm [CCKK15]. Several hard problems over lattices exist, with the most basic one being the Shortest Vector Problem (SVP): Given a basis $\mathbf{B}$ and $\gamma \geq 1$, find a non-zero $v \in \mathcal{L}(\mathbf{B})$ with $||v|| \leq \gamma \cdot \lambda_1(\mathcal{L})$. Several variations of the SVP exist, such as the Approximate Shortest Vector Problem (SVP_γ), the Decisional Approximate SVP (GapSVP_γ) or the Approximate Shortest Independent Vectors Problem (SIVP_γ) [Pei16]. The security of lattice-based cryptosystems can be proven on the basis of the hardness of the aforementioned lattice-problems.

2.1.2 (Ring-) Learning With Errors

In 2005, Regev introduced the Learning With Errors (LWE) problem [Reg05], whose hardness can be proven with a reduction to the GapSVP_γ and SIVP_γ problems. Two equivalent variants of the LWE problem exist, the Search-LWE and the Decisional-LWE problem. The two variants are defined as follows [CCKK15]:

- Search-LWE: Find $\mathbf{s} \in \mathbb{Z}_q^n$ given noisy random inner products

$$a_1 \leftarrow \mathbb{Z}_q^n, b_1 = \langle \mathbf{b}, a_1 \rangle + e_1 \tag{2.1}$$
$$a_2 \leftarrow \mathbb{Z}_q^n, b_2 = \langle \mathbf{b}, a_2 \rangle + e_2 \tag{2.2}$$
$$\vdots \tag{2.3}$$

  where $e_i \leftarrow \chi$ for a Gaussian distribution χ over $\mathbb{Z}$. $\mathbf{s}$ is often referred to as the secret, and the e_i are usually called error terms.

- Decisional-LWE: Distinguish $(A, \mathbf{b}^t = \mathbf{s}^t A + e^t)$ from a uniform $(A, \mathbf{b}^t)$, where $A = (a_1, ..., a_m)$.

The Ring-LWE problem is a variant of the LWE problem defined over rings. It is very similar to the normal LWE with the difference that $\mathbf{s}$ and the a_i are now taken from a ring R_q. Ring elements are often represented using polynomials.
While cryptosystems based on the normal LWE problem typically include computations such as vector-matrix multiplications or matrix-matrix multiplications, systems based on Ring-LWE usually include some sort of polynomial multiplications. The added structure in the Ring-LWE based systems can significantly increase their performance and decrease the parameter sizes, but at the same time, the structure might account for additional attack vectors on these schemes. Due to the lack of any structure, schemes like FrodoKEM are believed to offer superior security at the cost of slower encryption and decryption.

2.2 Field Programmable Gate Arrays

A Field Programmable Gate Array is an integrated circuit whose functionality can be configured after its production, unlike an ASIC that is manufactured for a specific purpose and cannot be modified afterwards. The reconfigurability of an FPGA allows to use them as prototypes to test the functionality of a design and potentially modify it, whereas ASICs are used for mass production once the design has been tested and is found to be working as intended.
The HDL source code, describing the desired behaviour of the FPGA at a Register-Transfer Level (RTL), is transformed into a netlist of logic components and their connections by a synthesis tool. Based on this netlist, a bitstream is generated that contains the programming information for the FPGA. In order for the FPGA to behave in the desired way, the generated bitstream has to be loaded into the device. The two major manufacturers of FPGAs, Xilinx and Altera, both provide proprietary tools that complete the needed tasks in the design process, including the simulation of the written HDL code to verify its correctness, the synthesis process and the transfer to the FPGA.
Different devices can be grouped into FPGA families. Inside a family, all devices share the same internal structure and process technology, but they can vary in size and speed to meet the needs of different applications. The families range from low-cost families such as the Xilinx Artix-7, featuring a rather small size at a small price, to high-performance families like the Xilinx Virtex-7 that are significantly larger, but also more expensive. FPGAs include different types and, depending on the family, different number of resources that are used to realize the desired functionality. These resources include:

- Flip Flops (FFs): An FF is the most basic storage unit in an FPGA and can be used to store a single bit. Multiple FFs together, often referred to as a register, can be used to store multiple bits. Writing to and reading from an FF can be done in a very short period of time, making them the fastest way to store data. However, when a large amount of data is to be stored, the available number of FFs on the FPGA can quickly be exceeded.

- Look-Up Tables (LUTs): LUTs are the key component to implementing logic on an FPGA. By storing the truth table of a function, an n-bit LUT can implement any n-bit boolean function. In modern FPGAs, LUTs typically have 5–6 input bits. If a more complex mathematical function such as an addition or even a multiplication is needed, multiple LUTs can be combined to implement the operation. In addition, LUTs can be used to store larger amounts of data as a Distributed RAM (LUTRAM). Storing and reading data with a LUTRAM is slower than with FFs, especially read accesses which take 2 clock cycles. The advantage of a LUTRAM over FFs is that it uses less of the available resources. However, for very large amounts of data such as keys of PQC, it is more efficient to use Block RAMs (BRAMs).

- BRAMs: BRAMs are dedicated storage elements located on the FPGA that are used to store large amounts of data. They exist in different sizes and configurations, for Xilinx FPGAs, the typical size is 36 Kbits. Such a BRAM can either be used

as a single 36 Kb RAM or as two independent 18 Kb RAMs. An 18 Kb RAM can have different configurations regarding the width of read and write ports, ranging from 18k × 1 with 18.000 1 bit elements to 512 × 36 with 512 36 bit elements. If larger data widths are needed, for example to store 64 or 128 bit elements, multiple BRAMs can be combined to reach the desired data width. BRAMs can be operated in different modes that influence the number of read and write ports of the BRAM. In the simplest, the single-port mode, the BRAM has one read and one write port, meaning that one element can be read and then overwritten in the same clock cycle. In simple dual-port mode, two read ports and one write port exist, allowing for an additional read over the single-port use. Finally, in true dual-port mode, two read and two write ports exist, enabling reading and then overwriting two stored elements in the same clock cycle.

- DSPs: The main use of a DSP is to implement complex mathematical logic such as multiply-accumulate, where two inputs are multiplied and then added to a third input. While such a functionality can also be implemented using LUTs, a DSP can execute it significantly faster. The multiply-accumulate operation is used, for example, in the matrix-matrix multiplication or in polynomial multiplications, and can therefore play an important role in the implementation of PQC on an FPGA.
- Programmable Interconnects (PICs): PICs are used to connect logic blocks on the FPGA. They are pre-laid on the device and can be programmed to provide routing paths for the logic blocks.
- I/O blocks: These blocks can be used to connect the FPGA with its exterior, for example to to load data onto the FPGA from an external source, or vice versa to write data from the device into an external destination.

Depending on the used FPGA family, the number of available resources can heavily vary from less than 10.000 FFs and 5 BRAMs for the smallest member of the Spartan-7 family, to over 1 million LUTs and 1.000 BRAMs for some members of the Virtex-7 family.

2.3 High-Level Synthesis

The main goal of High-Level Synthesis (HLS) is to speed up the design process of an implementation on an FPGA by letting a tool generate the HDL code from code written in a high-level language. The most common HLS tool, which is also used for HLS in this work, is Vivado HLS by Xilinx. This tool can perform the HLS on code written in the languages `C` and `C++` as well as `SystemC`, a system-level modelling language that is based on `C++`, but allows the concurrent execution of processes. As the software implementations submitted to the NIST competition are all written in `C`, only the functionalities for `C`-synthesis are used in this work.
The design process with the HLS tool consists of three repeating steps, `C`-Validation, `C`-Synthesis and `C`/RTL Cosimulation. During `C`-Validation, the correctness of the `C`-code is tested against a test bench provided by the user to find any errors in the code. Ideally,

the test bench should cover every possible case of parameter combinations. This step can be executed in a short period of time due to the speed of C. If the code is found correct, C-Synthesis can be started. The HLS tool now transforms the input code into equivalent code in Verilog and VHDL. After synthesis, the tool reports an estimation of the latency and hardware usage for the top function and any sub-functions. These estimations can be an apt indicator on the performance of the design and the efficiency of changes to it, but they are not exact. For the exact hardware usage, the user has to input the generated HDL files into a regular synthesis tool and have it perform the normal synthesis and implementation. Typically, the hardware usage estimations by the HLS tool are higher than the actual usage. Depending on the complexity of the C-code, C-Synthesis can take from a few seconds to a few minutes. Following C-Synthesis, the generated HDL-code can be verified in the C/RTL Cosimulation. The verification is done using the same test bench as in C-Validation. If the Cosimulation fails, the user can inspect the waveform of the executed code to find errors. Apart from correctness, the Cosimulation also reports the minimum, average and maximum latencies for the different runs of the test bench. As the Cosimulation step simulates the behaviour of hardware on a processor, it can take a lot longer than the initial C-Validation. Depending on the complexity of the code, Cosimulation can take from a few seconds to dozens of minutes. Although it is not necessary to always execute these three steps during optimization of the code, they certainly help to avoid errors and the long search for them, as well as giving a good overview of the positive and negative effects on latency and hardware usage of the applied changes.
To optimize the design, the user generally has two major options: Changes to the source code itself, and special instructions to the HLS tool informing it how to synthesize the code. Possible changes to the code include the (un-) rolling of loops, inlining of functions or wrapping multiple statements into a function, rearranging instructions, changing data types, and many more. Techniques used for software optimization can sometimes be applied; however, the user always has to keep in mind that the C-code is not meant to be executed in software, but in hardware.
Instructions to the tool can be made using special *pragmas* that are placed in the source code and read by the tool. Vivado HLS currently offers more than 20 *pragmas* that have very different use cases and effects. The below list shows some of the *pragmas* frequently used in this work and their effect:

- `INLINE`: Tells the tool to inline a function. As calling a function takes two clock cycles – one to start the function and read the parameters, one to end it and return values if necessary – this can significantly reduce the total latency if the function is called often, for example inside a loop. However, if the function is called on more than one occasion in the code, inlining it will result in additional hardware cost since the exact same hardware will have to be built multiple times. The tool will automatically inline smaller functions based on heuristics. To prevent this, the option `off` can be added to the *pragma*.

- `UNROLL`: If this *pragma* is specified inside a loop, it will be fully unrolled, and every loop iteration is executed within the same clock cycle if possible. This can reduce

the latency, but also increase the hardware usage as more operations have to be executed in parallel. By adding a `factor`, the user can specify how many loop iterations are to be executed in one clock cycle.

- `RESOURCE`: This *pragma* allows to specify which memory resource is to be used to implement an array. If no resource is specified, the tool will automatically decide which one to use based on heuristics. Smaller arrays will generally be implemented using LUTRAMs, but starting from a certain array size, BRAMs will be used. When specifying a resource, the user can decide what storage type is to be used (ROM, BRAM or LUTRAM) and in which mode it should operate (single-port, simple dual-port, or dual-port). If no storage type or mode is selected, the tool automatically selects them based on heuristics and based on how the array is used in the code.

- `ARRAY_PARTITION`: By using this *pragma*, a single array can be split up into multiple arrays, resulting in an implementation with multiple BRAMs. The user can specify the number of arrays and how the elements from the original array are distributed into the new arrays. If the partition is cyclic with n arrays used, the first array element is stored in the first array, the second element in the second array and so on, until the n-th element is stored in the first array again. By contrast, if the partition is block-wise, the first block of elements is stored in the first array, the second block in the second array and so on. With the additional BRAMs, more read and write ports for the array are available, potentially decreasing the latency if the additional ports are used, at the cost of a higher BRAM usage of the design. The third possible partition type is the complete partition. In that case, the array is not implemented using BRAMs, but using FFs. This can significantly speed up the design since all the array elements can be accessed at the same time. However, additional multiplexers are used to choose the needed array element, increasing the FF- and LUT-usage.

- `INTERFACE`: This *pragma* is especially important when arrays are passed to a function. If the array is passed by reference, that is if only a pointer is passed to the function, the `INTERFACE` *pragma* has to be used with the option `ap_bus`. The array is then implemented as a FIFO bus with one read or write access per clock cycle. Burst access, where multiple elements are read or written in successive clock cycles, is supported. Using the `depth` option, the user can specify the maximum number of elements in the bus. If, by contrast, the entire array is passed to the function (pass-by-value), the *pragma* has to be used together with the `ap_memory` option. The array is then implemented as an external RAM with data, address and control ports. The user can specify the type of the external RAM using the `RESOURCE` *pragma*, allowing up to two read and write accesses per clock cycle.

- `PIPELINE`: One of the most powerful *pragmas* that can be specified inside loops or functions. To demonstrate its effects, think of a loop where two array elements are added and then written into a third array. Without pipelining, each loop iteration

will take 3 clock cycles: In the first two clock cycles, the elements are read from the arrays, and in the third cycle, they are added and the result is written into the third array. Only after the result has been written, the next loop iteration is started. However, assuming that the loop iterations are independent of each other, the second iteration could start earlier, while the first is still being executed. In an ideal scenario such as the one above, a new iteration can start every clock cycle, reducing the effective iteration latency to just one clock cycle. Such an ideal scenario is not always given, in other cases a new loop iteration might only be started every second or third clock cycle. The potential speed-up for the design is enormous, and the additionally needed hardware is generally very small.

While the effects of single *pragmas* can already be noticeable, the best possible design can usually be achieved when several *pragmas* are combined. As an example, the additional read and write ports achieved by using `ARRAY_PARTITION` can have the maximum effect if a loop using the arrays is unrolled with the `UNROLL` *pragma*. For this example, it would make sense to adapt the unrolling factor to the partitioning factor to use every available port of the BRAMs. In a similar way, completely partitioning an array only makes sense if loops working with this array are fully unrolled so as to make full use of the simultaneous access to every array element.

2.4 FrodoKEM

Algorithm 2.4.1: Key Generation of FrodoKEM

Input: None

Output: Key pair (pk, sk') with $pk \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_\mathbf{A}}+D\cdot n\cdot\bar{n}}$, $sk' \in \{0,1\}^{\mathsf{len}_\mathbf{s}+\mathsf{len}_{\mathsf{seed}_\mathbf{A}}+D\cdot n\cdot\bar{n}} \times \mathbb{Z}_q^{n\times\bar{n}} \times \{0,1\}^{\mathsf{len}_{\mathbf{pkh}}}$

1 Choose uniformly random seeds $\mathbf{s}||\mathsf{seed}_{\mathbf{SE}}||\mathbf{z} \leftarrow^{\$} U(\{0,1\}^{\mathsf{len}_\mathbf{s}+\mathsf{len}_{\mathsf{seed}_\mathbf{SE}}+\mathsf{len}_\mathbf{z}})$

2 Generate pseudorandom seed $\mathsf{seed}_\mathbf{A} \leftarrow \text{SHAKE}(\mathbf{z}, \mathsf{len}_{\mathsf{seed}_\mathbf{A}})$

3 Generate the matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times\bar{n}}$ via $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_\mathbf{A})$

4 Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, ..., \mathbf{r}^{2n\bar{n}-1}) \leftarrow \text{SHAKE}(\texttt{0x5F}||\mathsf{seed}_{\mathbf{SE}}, 2n\bar{n}\cdot\mathsf{len}_\chi)$

5 Sample error matrix $\mathbf{S} \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, ..., \mathbf{r}^{(n\bar{n}-1)}), n, \bar{n}, T_\chi)$

6 Sample error matrix $\mathbf{E} \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(n\bar{n})}, \mathbf{r}^{(n\bar{n}+1)}, ..., \mathbf{r}^{(2n\bar{n}-1)}), n, \bar{n}, T_\chi)$

7 Compute $\mathbf{B} \leftarrow \mathbf{A}\cdot\mathbf{S}+\mathbf{E}$

8 Compute $\mathbf{b} \leftarrow \mathsf{Frodo.Pack}(\mathbf{B})$

9 Compute $\mathbf{pkh} \leftarrow \text{SHAKE}(\mathsf{seed}_\mathbf{A}||\mathbf{b}, \mathsf{len}_{\mathbf{pkh}})$

10 **return** public key $pk \leftarrow \mathsf{seed}_\mathbf{A}||\mathbf{b}$ and secret key $sk' \leftarrow (\mathbf{s}||\mathsf{seed}_\mathbf{A}||\mathbf{b}, \mathbf{S}, \mathbf{pkh})$

FrodoKEM [ABD+19a] is a lattice-based key encapsulation scheme and one of the candidates in the second round of the NIST competition. The German Federal Office for Information Security (BSI) recommends the use of the larger parameter sets of FrodoKEM

for the protection of confidential information over a long time [fISB20].
As FrodoKEM relies on the LWE and not on the Ring-LWE problem, it has larger parameters and a slower execution time than other lattice-based schemes, but offers strong security. Algorithms 2.4.1, 2.4.2 and 2.4.3 show the key generation, encapsulation and decapsulation of FrodoKEM. The subroutines called in the algorithms are briefly described in this section, a more detailed description can be found in the original specification. Frodo.Gen generates the matrix $\mathbf{A}$ row-wise, either with the hash function SHAKE or using AES. The generation is deterministic, based on a short seed and the index of the currently generated row. Using Frodo.SampleMatrix, matrices are sampled element-wise from a discrete Gaussian distribution. Frodo.Pack transforms a matrix into a bit string, Frodo.Unpack does the reverse operation. Frodo.Encode encodes a bit string as an integer matrix, Frodo.Decode decodes a matrix into a bit string.

Algorithm 2.4.2: Encapsulation of FrodoKEM

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

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

For the key generation, the main operation is the matrix multiplication to obtain the matrix $\mathbf{B}$. The seed that generates $\mathbf{A}$ as well as the packed version of $\mathbf{B}$ form the public key, whereas the secret matrix $\mathbf{S}$ and the hashed public key form the secret key.
In the encapsulation, three error matrices $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$ are generated the same way as in the key generation. Since a different seed is used, the matrices differ from $\mathbf{S}$ and $\mathbf{E}$ in the key generation. The matrix $\mathbf{B}'$ is calculated as $\mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ and then packed into $\mathbf{c}_1$. Note that $\mathbf{A}$ is now on the right side of the multiplication, whereas it was on the left

side in the key generation. $\mathbf{V}$ is calculated as $\mathbf{S}' \cdot \mathbf{B} + \mathbf{E}''$, where $\mathbf{B}$ is unpacked from the public key. The encoded random bit string μ is added to $\mathbf{V}$ and the result is packed into $\mathbf{c}_2$. Finally, the shared secret $\mathbf{ss}$ is calculated as the hash of the two ciphertexts and a short random value $\mathbf{k}$.

Algorithm 2.4.3: Decapsulation of FrodoKEM

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

Output: Shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len_{ss}}}$

$\mathbf{B}' \leftarrow$ Frodo.Unpack($\mathbf{c}_1$)
$\mathbf{C} \leftarrow$ Frodo.Unpack($\mathbf{c}_2$)
Compute $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}' \cdot \mathbf{S}$
Compute $\mu' \leftarrow$ Frodo.Decode($\mathbf{M}$)
Parse $pk \leftarrow \mathsf{seed}_\mathbf{A}||\mathbf{b}$
Generate pseudorandom values $\mathsf{seed}'_{\mathbf{SE}}||\mathbf{k}' \leftarrow$ SHAKE($\mathbf{pkh}||\mu, \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len}_\mathbf{k}$)
Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, ..., \mathbf{r}^{(2\bar{m}n+\overline{mn}-1)}) \leftarrow$ SHAKE($\mathsf{0x96}||\mathsf{seed}'_{\mathbf{SE}}, (2\bar{m}n + \overline{mn}) \cdot \mathsf{len}_\chi$)
Sample error matrix $\mathbf{S}' \leftarrow$ Frodo.SampleMatrix($(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, ..., \mathbf{r}^{(\overline{m}n-1)}), \overline{m}, n, T_\chi$)
Sample error matrix $\mathbf{E}' \leftarrow$ Frodo.SampleMatrix($(\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, ..., \mathbf{r}^{(2\overline{m}n-1)}), \overline{m}, n, T_\chi$)
Generate $\mathbf{A} \leftarrow$ Frodo.Gen($\mathsf{seed}_\mathbf{A}$)
Compute $\mathbf{B}'' \leftarrow \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$
Sample error matrix $\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix($(\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, ..., \mathbf{r}^{(2\overline{m}n+\overline{mn}-1)}), \overline{m}, \overline{n}, T_\chi$)
Compute $\mathbf{B} \leftarrow$ Frodo.Unpack($\mathbf{b}, n, \overline{n}$)
Compute $\mathbf{V} \leftarrow \mathbf{S}' \cdot \mathbf{B} + \mathbf{E}''$
Compute $\mathbf{C}' \leftarrow \mathbf{V} +$ Frodo.Encode(μ')
if $\mathbf{B}'||\mathbf{C} = \mathbf{B}''||\mathbf{C}'$ **then**
  return shared secret $\mathbf{ss} \leftarrow$ SHAKE($\mathbf{c}_1||\mathbf{c}_2||\mathbf{k}', \mathsf{len_{ss}}$)
else
  return shared secret $\mathbf{ss} \leftarrow$ SHAKE($\mathbf{c}_1||\mathbf{c}_2||\mathbf{s}, \mathsf{len_{ss}}$)

In the decapsulation, μ is first decrypted and then re-encrypted using the secret key. Then it is checked whether the re-encrypted ciphertexts match with the input ciphertexts. If that is the case, the shared secret is calculated by hashing the ciphertexts together with the short random value.
The authors suggest three different parameters sets, Frodo-640, Frodo-976 and Frodo-1344, named after the value of the parameter n, that achieve different security levels. The smallest parameter set Frodo-640 achieves security level 1 in the NIST competition which matches the security of AES-128. Frodo-976 targets security level 3 which matches the security of AES-192, and the largest parameter set Frodo-1344 matches or exceeds the security of AES-256 which is equivalent to security level 5. Table 2.1 shows the

three different parameters sets. The sampling of the error matrices is done using a discrete, symmetric distribution on $\mathbb{Z}$ that approximates a rounded continuous Gaussian distribution. The different probability density functions for the three parameter sets are displayed in Table 2.2. For the implementation, the probability density functions are transformed into discrete Cumulative Distribution Functions (CDFs) T_χ that return the probability for a value being x or less. 16 pseudorandom bits are required to sample one matrix entry. The first 15 bits are interpreted as an integer $y \in [0, 32767]$, and the sampled value x is the smallest integer such that $y \leq T_\chi(x)$. The least significant bit then determines the sign of the sampled value. As an example, if the 16 bit input is 37.287, or `0x91A7` in hexadecimal notation, y is then 18.643 or `0x48D3`. The smallest x such that $y \leq T_\chi(x)$ is 2, as $T_\chi(2) = 20.864$. Since the least significant bit of the input is 1, the sampled value is -2, or `0xFFFE` as a 16 bit hexadecimal value.

	Frodo-640	Frodo-976	Frodo-1344
D	15	16	16
q	32768	65536	65536
n	640	976	1344
$\overline{m} = \overline{n}$	8	8	8
B	2	3	4
$\mathsf{len}_{\mathsf{seed}_\mathbf{A}}$	128	128	128
$\mathsf{len}_\mathbf{z}$	128	128	128
$\mathsf{len}_\mu = l$	128	192	256
$\mathsf{len}_{\mathsf{seed}_\mathbf{SE}}$	128	192	256
$\mathsf{len}_\mathbf{s}$	128	192	256
$\mathsf{len}_\mathbf{k}$	128	192	256
$\mathsf{len}_\mathbf{pkh}$	128	192	256
$\mathsf{len}_\mathbf{ss}$	128	192	256
len_χ	16	16	16
χ	$\chi_{\text{Frodo-640}}$	$\chi_{\text{Frodo-976}}$	$\chi_{\text{Frodo-1344}}$
SHAKE	SHAKE128	SHAKE256	SHAKE256

Table 2.1: FrodoKEM parameter sets.

	σ	**Probability of (in multiples of 2^{-16})**												
		0	±1	±2	±3	±4	±5	±6	±7	±8	±9	±10	±11	±12
$\chi_{\text{Frodo-640}}$	2.8	9288	8720	7216	5264	3384	1918	958	422	164	56	17	4	1
$\chi_{\text{Frodo-976}}$	2.3	11278	10277	7774	4882	2545	1101	396	118	29	6	1		
$\chi_{\text{Frodo-1344}}$	1.4	18286	14320	6876	2023	364	40	2						

Table 2.2: Error distributions for the three parameter sets of FrodoKEM.

3 Analysis of Crucial Components

Despite the many differences between the lattice-based encryption schemes, certain crucial components are used in many of them. One of these components is the multiplication of polynomials that appears in many of the schemes relying on the Ring-LWE assumption where it is one of the key operations. For schemes such as FrodoKEM relying on the normal LWE assumption, matrix multiplication instead of polynomial multiplication is used. To generate large amounts of pseudorandom data based on a small amount of true randomness, many schemes make use of the SHAKE hash function as it can produce output data of arbitrary length. Another important component of lattice-based cryptography are samplers from either a Gaussian or a binomial distribution. This section describes the process of applying HLS to some of the mentioned components as well as the optimization of the results. After optimization, the resulting hardware implementations are compared with existing direct hardware implementations of the components. If not stated otherwise, the target device is a Xilinx Artix-7 FPGA, the HLS tool used is Xilinx Vivado HLS 2020.1 and the regular synthesis tool used is Xilinx Vivado 2020.1.

3.1 Polynomial Multiplication

The multiplication of two polynomials appears in many lattice-based PQC schemes, either using the NTT or not. The case where the NTT is not used appears for example in the NTRU encryption scheme [CDH+19]. Together with the submission paper, the authors also submitted a software implementation of their scheme that can be found at [NTR20]. The direct hardware implementation that is used for comparison in this section was published by Braun et al. [BFM+18] in 2018. It performs the multiplication of two polynomials with 509 coefficients in 509 clock cycles (for different parameter sets, the number of coefficients in a polynomial can vary, the runtime of the multiplication scales linearly with the number of coefficients). To achieve this high performance, a large number of LUTs and FFs is used in the implementation. The goal of the optimization process should therefore be to match the low latency, whereas hardware usage is only a secondary optimization target.

3.1.1 Naive HLS

Before starting any optimization, it makes sense to perform HLS on the unoptimized software implementation to find out how well it performs, and where the largest optimization potential is. In the implementation, polynomials are represented in a `struct poly` that contains the coefficients in an array named `coeffs`. The coefficients have the 16 bit data type `uint16_t`. Listing 3.1 shows the unoptimized software implementation of the

```
1  typedef struct{
2      uint16_t coeffs[NTRU_N];
3  } poly;
4
5  void poly_mul(poly *r, const poly *a, const poly *b)
6  {
7      int i, k;
8      L1: for(k = 0; k < NTRU_N; k++){
9          r->coeffs[k] = 0;
10         L1_1: for(i = 1; i < NTRU_N-k; i++){
11             r->coeffs[k] += a->coeffs[k+i] * b->coeffs[NTRU_N-i];
12         }
13         L1_2: for(i = 0; i < k+1; i++){
14             r->coeffs[k] += a->coeffs[k-i] * b->coeffs[i];
15         }
16         r->coeffs[k] = MODQ(r->coeffs[k]);
17     }
18 }
```

Listing 3.1: Unoptimized polynomial multiplication of the NTRU encryption scheme.

function `poly_mul` and the definition of `poly`. The two input polynomials are passed as parameters `a` and `b` of type `poly`, the output polynomial is passed to the function as a `poly` named `r`.
This code can be synthesized by the HLS tool without any changes. After synthesis, the tool estimates a hardware usage of 156 FFs, 324 LUTs, 3 BRAMs (one for each polynomial) and 2 DSPs. The total latency is estimated to be 779.280 clock cycles with a maximum frequency of 109,63 MHz. While the hardware usage is very low, the latency of the design is very long compared to the direct hardware implementation. As a consequence, reducing the latency and increasing the frequency should be the main goal of the optimization process.

3.1.2 First Optimizations

The first thing to notice is that in the two inner loops `L1_1` and `L1_2`, the product of two coefficients is always added to a coefficient in the output polynomial, resulting in many read and write accesses to `r`. While this is normal in a software implementation, it can be problematic in a hardware implementation since reading from or writing to a BRAM takes longer than accessing a register. As a consequence, it is more efficient to use a 16-bit variable `temp`, implemented using FFs, to accumulate the different products. `temp` is set to 0 at the start of every iteration of `L1` and then accumulates the products in the two inner loops. In the end, `temp` is written to `r->coeffs[k]`. This change reduces the hardware usage to 153 FFs and 297 LUTs while also slightly decreasing the total latency. In addition, the maximum frequency is increased to 157,48 MHz.
Another possible optimization is to change the data type of `coeffs` which is currently `uint16_t`. However, only 11 bits are needed to represent the polynomial coefficients. In normal `C`-code, declaring an 11-bit variable is impossible since the data length has to be a multiple of 8, but in hardware, registers of any length are possible. Including the

`ap_cint.h` header allows the user to define variables of any desired length in the `C`-code. Changing the data type to `uint11` makes sense for `coeffs` and the accumulator variable `temp`. This change slightly reduces the FF-usage to 133.

3.1.3 Pipeline

The two previous optimizations helped reduce the total hardware usage, but had only little effect on the latency of the function. In the current design, every iteration of one of the two inner loops takes 3 clock cycles: The coefficients of `a` and `b` are loaded in the first 2 clock cycles, then multiplied and added to the accumulator variable in the third clock cycle. To improve the design, the `PIPELINE` *pragma* can be used in the two inner loops to start a new loop iteration every clock cycle, reducing the effective latency of every loop iteration to 1. This reduces the latency of the function to 262.644 clock cycles while slightly increasing the LUT usage to 335. The maximum frequency remains unchanged.
While this optimization has already reduced the latency significantly, the design is still far off from the low latency of the direct hardware implementation, and further improvements of the latency with the code in its current form seem to be impossible. In order to further reduce the latency, it is necessary to execute more multiplications at the same time, for example by unrolling the two inner loops.
However, two obstacles are currently preventing these loops from being unrolled: First of all, `L1_1` and `L1_2` have variable upper bounds that depend on the value of k, the loop variable of the outer loop. Therefore, it is impossible for the HLS tool to fully unroll the loops. The only possibility would be a partial unrolling, with only a certain amount of loop iterations being executed at the same time. Unfortunately, such a partial unrolling of the inner loops would not be sufficient to reduce the latency as much as desired.
The second obstacle is the array `coeffs` currently being implemented using a BRAM. In order to execute many multiplications in one clock cycle, it is necessary to access many, if not all, coefficients in one clock cycle. This is impossible when implementing the coefficients as a BRAM since it is only possible to access a maximum of two of its elements in one clock cycle.
As a consequence, two major changes to the design are needed to further reduce the latency: In order to be able to fully unroll the two inner loops, it is necessary for these loops to have a fixed lower and upper bound, and instead of implementing the polynomials using BRAMs, they need to be implemented using FFs to allow many parallel accesses to the coefficients.

3.1.4 Code Redesign

To modify the two inner loops to loops with fixed bounds, it helps to visualize the first two iterations of `L1` for $k = 0$ and $k = 1$. They are shown in Equation 3.1. Currently, `r`[0] is accumulated first using the variable `temp`. Once `r`[0] is done, `r`[1] is accumulated, and so on. To express this behaviour with a single inner loop is impossible, however the equations can be rearranged to achieve the desired behaviour. Instead of arranging the multiplications around the coefficients of `r`, they can be arranged around the coefficients

of a. This means that every multiplication that includes a[0] is completed and added to the right coefficient of r first, then the same is done for a[1] and so on. To ensure that the products are always added to the correct coefficient of r, the polynomial b has to be rotated by one position after every iteration of L1. The rotation can be done efficiently once b is implemented using FFs as this allows to access every element at the same time.

$$\begin{aligned}
k = 0 : \texttt{temp} &+= \texttt{a}[1] \cdot \texttt{b}[508] \\
\texttt{temp} &+= \texttt{a}[2] \cdot \texttt{b}[507] \\
&\dots \\
\texttt{temp} &+= \texttt{a}[508] \cdot \texttt{b}[1] \\
\texttt{temp} &+= \texttt{a}[0] \cdot \texttt{b}[0] \\
\texttt{r}[0] &= \texttt{temp} \\
k = 1 : \texttt{temp} &+= \texttt{a}[2] \cdot \texttt{b}[508] \\
&\dots \\
\texttt{temp} &+= \texttt{a}[508] \cdot \texttt{b}[2] \\
\texttt{temp} &+= \texttt{a}[1] \cdot \texttt{b}[0] \\
\texttt{temp} &+= \texttt{a}[0] \cdot \texttt{b}[1] \\
\texttt{r}[1] &= \texttt{temp}
\end{aligned} \tag{3.1}$$

To achieve the implementation of the polynomials using FFs, it is sufficient to add the ARRAY_PARTITION *pragma* with the option complete at the beginning of the function. However, the HLS tool produces a rather odd behaviour when completely partitioned arrays are passed to a function: Instead of working on the passed array, the tool creates a local copy of the array inside the function and executes the function on this copy. While the code remains correct, this behaviour results in two versions of the same array and therefore in a doubled FF cost to implement the array. To circumvent this problem, the arrays to be partitioned can be defined as global arrays outside of any function, with the ARRAY_PARTITION *pragma* inside the top-level function. In this case, it is not necessary to pass the partitioned array to the sub-function and the tool only creates one version of the partitioned array.
The rewritten C-code, displayed in Listing 3.2, still has two inner loops, but with fixed upper bounds. The inner loop L1_1 executes the multiplications of the coefficients, and Rotate rotates the entire polynomial b by one position. To ensure the correctness of the code, the coefficients of r need to be set to 0 in the loop Reset before starting the multiplications. The ARRAY_PARTITION *pragma* is added for the coefficients of r and b. It is not necessary for a since only one coefficient per outer loop iteration is needed.
After synthesis, the new design needs over 11.000 FFs, mainly to implement the two polynomials r and b, and over 20.000 LUTs. The high amount of LUTs can be explained by the need for large multiplexers to select the needed coefficient. With the addition of the loop Rotate, the total latency is increased to 521.218 clock cycles. While the new design seems to be worse than the previous one in all aspects, it is now possible to fully unroll the two inner loops and thereby reducing the latency to the desired level.

```
1  uint11 r[NTRU_N], b[NTRU_N];
2  void poly_mul(poly *a)
3  {
4      #pragma HLS ARRAY_PARTITION variable=r complete dim=1
5      #pragma HLS ARRAY_PARTITION variable=b complete dim=1
6      uint11 temp;
7      int i, k;
8      Reset: for(i = 0; i < NTRU_N; i++){
9          r[i] = 0;
10     }
11     L1: for(k = 0; k < NTRU_N; k++){
12         L1_1: for(i = 0; i < NTRU_N; i++){
13             #pragma HLS PIPELINE
14             r[i] += a->coeffs[k] * b[i];
15         }
16         temp = b[NTRU_N-1];
17         Rotate: for(i = NTRU_N-1; i > 0; i--){
18             b[i] = b[i-1];
19         }
20         b[0] = temp;
21     }
22 }
```

Listing 3.2: Polynomial multiplication of the NTRU encryption scheme after the internal multiplications have been rearranged.

3.1.5 Loop Unrolling

To fully unroll the two inner loops, the `UNROLL` *pragma* can be added inside the two loops. In addition, `Reset` can also be fully unrolled using this *pragma*. The synthesis results are now much closer to the results of the direct hardware implementation: The usage of FFs remains almost unchanged and the usage of LUTs is reduced to 12.302. This is due to the fact that less multiplexers are required since every element of the partitioned arrays is used in every iteration of the outer loop. Furthermore, the total latency of the function is reduced to only 2.037 clock cycles. However, the design now needs 509 DSPs to execute every multiplication of `L1_1` within one clock cycle. Since the direct implementation does not use any DSPs, further optimizations are needed to eliminate the need for DSPs.
Before doing so, two smaller optimizations are possible. Currently, every iteration of `L1` takes 4 clock cycles to complete. For some reason, the tool schedules multiple read operations on `a`, despite only one coefficient of `a` being needed, which takes 2 clock cycles. By loading `a->coeffs[k]` into a local variable `a_k` at the start of every iteration, this behaviour disappears and the loop iterations take only 3 clock cycles instead of 4.
Now that the two inner loops `L1_1` and `Rotate` are fully unrolled, it is possible to pipeline `L1` by adding the `PIPELINE` *pragma*. This results in an effective iteration latency of 1 clock cycle, and therefore 509 clock cycles for the entire loop. The number of needed FFs and LUTs has almost remained unchanged by these two optimizations.

3.1.6 Removing DSPs

In order to eliminate the need for DSPs, it is worth noting that the input polynomial `a` is a ternary polynomial, meaning that its coefficients can only be ± 1 or 0. As a consequence, a true multiplication using DSPs is not necessary. Instead, it is sufficient to check whether `a_k` is $+1$ or -1, and, depending on which is the case, either adding or subtracting the coefficient of `b`. If the coefficient is 0, nothing has to be done. The final code of the polynomial multiplication is shown in Listing 3.3. The `if/else` statement starting in line 14 replaces the multiplication in the previous versions. After synthesis, the tool reports an increased LUT-usage of 41.849 since 509 parallel additions have to be executed, but no more DSPs are needed. In addition, the maximum frequency is increased to 153,47 MHz, compared to only 129,87 MHz when using DSPs.
The described hardware usage is only an estimation from the HLS tool, the actual hardware usage after synthesis and implementation can differ a lot and will usually be lower. In the case at hand, the usage of FFs is almost identical, but the usage of LUTs is reduced to 16.073, less than half of what the HLS tool estimated.

```
1  void poly_mul(poly *a)
2  {
3      uint11 temp, a_k, b_i;
4      int i, k;
5      Reset: for(i = 0; i < NTRU_N; i++){
6          #pragma HLS UNROLL
7          r[i] = 0;
8      }
9      L1: for(k = 0; k < NTRU_N; k++){
10         #pragma HLS PIPELINE
11         a_k = a->coeffs[k];
12         L1_1: for(i = 0; i < NTRU_N; i++){
13             #pragma HLS UNROLL
14             if(a_k == 1)
15                 b_i = b[i];
16             else if(a_k == NTRU_Q-1)
17                 b_i = -b[i];
18             else
19                 b_i = 0;
20             r[i] += b_i;
21         }
22         temp = b[NTRU_N-1];
23         Rotate: for(i = NTRU_N-1; i > 0; i--){
24             #pragma HLS UNROLL
25             b[i] = b[i-1];
26         }
27         b[0] = temp;
28     }
29 }
```

Listing 3.3: Final design for the polynomial multiplication of the NTRU encryption scheme.

3.1.7 Comparison with the Direct Hardware Implementation

The information about the hardware usage of the direct implementation in [BFM+18] is incomplete, therefore it is necessary to make some reasonable assumptions. The paper does not state how many FFs are needed, but the number should be almost identical to the one in this work, since the only FFs needed are those to implement the two polynomials. Regarding LUTs, the paper only gives the total amount for encryption and decryption. As both operations are similar, it can be expected that encryption alone takes about half the total amount of 38.240. The polynomial multiplication is the most expensive operation in the encryption, therefore, a large amount of the 19.120 LUTs will be used for it. 16.073 LUTs in the HLS implementation seems to be a good amount compared to the direct implementation.
Regarding latency, the direct implementation needs 509 clock cycles, scaling linearly with the value of the parameter `NTRU_N`. The HLS implementation takes 2 clock cycles more due to the use of a function. A latency smaller than 511 clock cycles is not achievable in HLS with the current design.
Table 3.1 shows the synthesis results for the naive HLS implementation, the subsequent optimization steps, and the reference direct implementation. The values for all but the final synthesis version are estimations by the HLS tool, while the final synthesis version displays the actual usage calculated by the synthesis tool.

Implementation	**FF**	**LUT**	**BRAM (18K)**	**DSP**	**Latency (Cycles)**	**Frequency (MHz)**
Naive HLS	156	324	3	2	779.280	109,63
First Optimizations	133	297	3	2	778.771	157,48
Pipeline	123	335	3	2	262.644	157,48
Code Redesign	11.379	22.657	0	1	521.218	129,87
Loop Unrolling	11.297	12.302	0	509	2.037	129,87
Removing DSPs	11.247	41.849	0	0	511	153,47
Final Synthesis	11.454	16.073	0	0	511	153,47
[BFM+18]	?	<19.120	0	0	509	?

Table 3.1: Synthesis results for the different optimization steps of the HLS version and the direct implementation of the polynomial multiplication of the NTRU encryption scheme.

3.2 Number Theoretic Transform

The NTT is a transformation that can help speed up the multiplication of polynomials. Instead of performing a schoolbook multiplication with around $\mathcal{O}(n^2)$ single multiplications and additions, the NTT is first applied to both polynomials, followed by a point-wise multiplication of the transformed polynomials. Finally, the inverse NTT is applied to the result. The NTT can speed up the polynomial multiplication to $\mathcal{O}(n \cdot \log n)$ single

multiplications and additions. However, it can only be applied if the polynomials and the underlying ring fulfil certain criteria.
As the pointwise multiplication of two polynomials is a rather simple task, this section only focuses on the HLS of the NTT itself. The reference hardware implementation by Güneysu et al. [OG19] implements the NewHope-Simple Key Exchange that uses the NTT for the polynomial multiplication. This work uses the software implementation of the NewHope cryptosystem by Alkim et al. [AAB+19] submitted to the NIST competition as a basis for the High-Level Synthesis.

3.2.1 Naive HLS

The `C`-code of the function `ntt` cannot be synthesized without initial changes due to the input arrays `a` and `omega`. As these arrays are passed by reference, it is necessary to inform the tool about the maximum array size using the `INTERFACE` *pragma*. The `depth` option of this *pragma* specifies the maximum number of elements in the referenced array. In this function, both arrays have a `depth` of 1024. The code with the added *pragmas* is shown in Listing B.1 in the Appendix.
After synthesis, the tool expects a hardware usage of 1.021 FFs, 1.126 LUTs, 3 DSPs, and a total latency of 124.942 clock cycles with a maximum frequency of 114,29 MHz. This is more than 5 times the amount of FFs and LUTs the direct implementation needs, and more than 3 times slower. In addition, one more DSP is needed. Reducing the latency as well as reducing the FF- and LUT-usage should be the main goal of the optimization process.

3.2.2 Polynomials as Parameters

Polynomials are implemented in the same way as for NTRU, only that the number of elements in the array `coeffs` is different. Instead of passing the entire `poly`, only the array `coeffs` is passed to the function. This results in long array access times of 3 clock cycles instead of the normal 2 clock cycles.
Therefore, passing the entire `poly` to the function significantly reduces the latency. Array elements now have to be accessed via `a->coeffs[k]` instead of via `a[k]`, the accesses to `omega` remain unchanged. This change reduces the estimated hardware usage to 829 FFs and 1.115 LUTs. In addition, the latency is reduced by over 25.000 clock cycles to 99.342, and the maximum frequency increases to 131,18 MHz.

3.2.3 Modulo Operation

A major parameter influencing the latency is the modulo reduction in line 33 of the unoptimized code, shown again in Listing 3.4. As the modulus `NEWHOPE_Q` (12289) is not a power of 2, this reduction cannot be done with a simple `AND` operation. In addition, it is impossible to make use of the Montgomery-Reduction since the input to the reduction might not be large enough. Therefore, it needs to be implemented as a true modulo reduction, a very slow and costly operation in hardware.
However, the inputs to this reduction have a maximum size of 16 bits or a maximum

value of 65.535, which is less than 6 times the modulus. As a consequence, instead of using a true modulo reduction, it is possible to use a few conditional statements and subtractions to implement it. The modulo reduction can be replaced by the code shown in Listing 3.5. The thresholds of the conditional statements as well as the subtrahends are the different multiples of `NEWHOPE_Q`.

```
1    ...
2    a[j] = (temp + a[j + distance]) % NEWHOPE_Q;
3    ...
```

Listing 3.4: Modulo reduction in the unoptimized implementation of the NTT.

This optimized modulo operation significantly reduces the FF usage to 598 and almost halves the latency to 50.703 clock cycles while the LUT usage remains almost unchanged. Due to a longer critical path in the modulo operation, the frequency is reduced to 118,30 MHz.

```
1  x = temp + a->coeffs[j + distance];
2  if(x >= 61445)
3      x -= 61445;
4  else if(x >= 49156)
5      x -= 49156;
6  else if(x >= 36867)
7      x -= 36867;
8  else if(x >= 24578)
9      x -= 24578;
10 else if(x >= 12289)
11     x -= 12289;
12 a->coeffs[j] = x;
```

Listing 3.5: Optimized modulo operation for small inputs using only comparators and subtractions.

3.2.4 Loop Rolling

In the unoptimized version, the loop `L1` is partially unrolled with a factor of 2: 2 loop iterations are written out (`L1_1` and `L1_2`), and the loop index `i` is always increased by 2. An excerpt of `L1` is shown in Listing 3.6. The only difference between `L1_1` and `L1_2` is that in `L1_2`, the previously described modulo reduction is executed, whereas it is omitted in `L1_1`. While unrolling a loop can speed up the design if multiple loop iterations can be executed in parallel, this is not the case for this loop since `L1_2` can only be executed once `L1_1` has been completed. As a consequence, the loop unrolling in this function does not result in a speed up while requiring considerably more resources to create the almost identical hardware twice for the two inner loops.
To roll `L1`, one of the two inner loops needs to be removed from the code, and `i` needs to only be increased by 1 per iteration. To ensure that the modulo reduction is only executed in odd iterations, it needs to be moved inside a conditional statement that is

only true for odd values of `i`, for example `if(i & 0x1)`.
Rolling `L1` significantly reduces the usage of FFs and LUTs to 356 and 776, respectively, while slightly increasing the latency to 53.267 clock cycles. In addition, the amount of needed DSPs is reduced by 1 as the multiplication for the input of the Montgomery Reduction now only appears once instead of appearing twice.

```
...
L1: for(i = 0; i < 10; i += 2)
{
    distance = (1 << i);
    L1_1: for(start = 0; start < distance; start++){
        ...
    }
    distance <<= 1;
    L1_2: for(start = 0; start < distance; start++){
        ...
    }
}
...
```

Listing 3.6: Partially unrolled loop `L1` in the unoptimized software implementation of the NTT.

3.2.5 Further Optimizations

The coefficient `a->coeffs[j + distance]` is needed twice in every iteration of `L1_1_1`, resulting in two accesses to the array. This can be avoided by loading the coefficient into a variable `adistance` and then accessing this variable whenever the coefficient is needed. In every iteration of `L1_1_1`, 3 clock cycles are saved by this optimization, reducing the latency to 37.907 clock cycles as well as slightly reducing the hardware usage.
Furthermore, variables can have smaller data types in some cases, depending on the maximum value they have to contain. If the tool can determine the maximum value of a variable, for example of loop variables where the loop has a fixed upper bound, the tool implements these with as few FFs as possible, and changing the data type is not necessary. This is the case for `i` and `j`. For other variables, the tool is unable to determine the maximum size, and therefore implements them as indicated by the data types. `jTwiddle` is at most a 9 bit value, while `distance` is at most a 10 bit value. Declaring the variables as shown in Listing 3.7 results in a reduction of hardware usage to only 247 FFs and 667 LUTs. The total latency and the maximum frequency remain unchanged by this optimization.

3.2.6 Combining Loops

The function currently has 3 nested loops, and every change from an outer to an inner loop (either from `L1` to `L1_1` or from `L1_1` to `L1_1_1`) takes one clock cycle. To avoid this, it is possible to modify the design to have only one loop with 5120 iterations, the total number of iterations of the old `L1_1_1` loop. The loop body of `L1_1_1` remains unchanged, but

```
1 uint9 jTwiddle = 0;
2 uint10 start = 0, distance = 1;
3 uint14 W;
4 uint15 temp, adistance;
5 uint16_t i, j = 0, x;
```

Listing 3.7: Changed data types for variables.

after each iteration, the design has to check which variables to be incremented or reset. This check is displayed in Listing 3.8.
The `if`-statement in line 1 represents the old loop `L1_1_1`, checking if another incrementation of `j` is possible. If that is not the case, then `L1_1_1` has been completed and it has to be checked if another iteration of the old `L1_1` loop can be done, represented by the second `if`-statement in line 8. If `start` cannot be incremented, `L1_1` has been completed and the next iteration of `L1` can start. This change slightly reduces the hardware usage to 261 FFs and 804 LUTs, and the latency to 35.841 clock cycles.
The critical path of the design is currently the reduction using subtractions and the write operation `a->coeffs[j] = x`, which happen within the same clock cycle. As the reduction takes rather long, this results in a low frequency. By manually choosing a faster clock, for example 8 ns, the tool can be forced to add a register in the critical path, writing the result of the reduction into the register, and writing to `a->coeffs` from the register in the next clock cycle. This slightly increases the number of FFs needed to 275 while increasing the maximum frequency to 146,77 MHz.

```
1  if((j + 2*distance) < NEWHOPE_N-1)
2  {
3      j += 2*distance;
4  }
5  else
6  {
7      jTwiddle = 0;
8      if((start + 1) < distance)
9      {
10         start++;
11         j = start;
12     }
13     else
14     {
15         start = 0;
16         j = 0;
17         distance <<= 1;
18     }
19 }
```

Listing 3.8: Loop control for the NTT that checks if another iteration can be executed and if variables have to be incremented or reset for the next interation.

3.2.7 Inlining the Montgomery Reduction

After combining the loops, the Montgomery Reduction, executed by the function `montgomery_reduce`, only appears at one place in the code. As it is not executed in parallel to another function, there is no reason to keep the reduction inside a function. Instead, it can be inlined to further decrease the latency of the design. After inlining, the latency is reduced to 30.721 clock cycles since one clock cycle per loop iteration can be saved. However, the HLS tool now reports the use of 3 DSPs, one more than before and one more than in the direct implementation.
To limit the number of DSPs to 2, the `ALLOCATION` *pragma* with the option `instances=mul limit=2` can be added. The HLS tool now only uses 2 DSPs; however, the `C`/RTL Cosimulation fails with the new design. Another way to control the number of used DSPs is to move multiplications into a function. This could potentially increase the latency since multiplications could not be executed in parallel (with only one multiplication function used), but allows the user to control how many DSP are used. Listing 3.9 shows the updated Montgomery Reduction where every multiplication is executed using the multiplication function `mul`. Using this approach, the HLS tool reports a hardware usage of 291 FFs, 815 LUTs and 2 DSPs with a total latency of 30.721 clock cycles. The maximum frequency is slightly reduced to 142,85 MHz.
The synthesis tool reports a hardware usage of 293 FFs, 254 LUTs and 2 DSPs with a total latency of 30.721 clock cycles. The maximum frequency of 136,99 MHz is a bit lower than estimated by the HLS tool.

```
1  uint32_t mul(uint32_t a, uint32_t b)
2  {
3      #pragma HLS INLINE off
4      return a * b;
5  }
6  red_in = mul(W, ((uint32_t) temp + 3*NEWHOPE_Q - adistance));
7  u = mul(red_in, qinv);
8  u &= ((1 << rlog) - 1);
9  u = mul(u, NEWHOPE_Q);
10 red_out = (red_in + u) >> 18;
```

Listing 3.9: Inlined Montgomery Reduction. Multiplications are executed using the function `mul`.

3.2.8 Comparison with the Direct Hardware Implementation

Table 3.2 displays the hardware usage and latency for the different optimization steps of the HLS version and of the direct hardware implementation. Overall, the performance of both implementations is similar, albeit a few differences. The HLS implementation uses about 40 FFs more than the direct implementation, but only needs half the amount of LUTs. However, it has to be said that the direct implementation can execute the forward and the backward NTT, whereas the HLS version can only execute the forward NTT. This might explain the rather low LUT usage of the HLS version. The total latency of

Implementation	FF	LUT	BRAM (18K)	DSP	Latency (Cycles)	Frequency (MHz)
Naive HLS	1.021	1.126	0	3	124.942	114,29
Modulo Operation	598	1.098	0	3	50.703	118,30
Loop Rolling	431	788	0	2	53.267	119,90
Further Optimizations	247	667	0	2	37.907	119,90
Combining Loops	275	804	0	2	35.841	146,77
Inlining Reduction	291	815	0	2	30.721	142,85
Final Synthesis	293	254	0	2	30.722	136,99
[OG19]	251	509	0	2	35.840	125

Table 3.2: Synthesis results for the different optimization steps of the HLS version and the direct implementation of the NTT.

the HLS version is about 5.000 clock cycles lower than that of the direct implementation, which is due to one clock cycle being saved in every iteration of the main loop. The maximum frequency of the HLS implementation is about 10 MHz higher. However, the maximum frequency of the direct implementation is most likely bounded by other functionalities in the NewHope scheme and could therefore be higher if the NTT were implemented alone.
Regarding the two used DSPs, one thing that has to be mentioned is that in the direct implementation, only one DSP is bounded to the NTT whereas the other is a multi-purpose DSP that is used for other functionalities as well. By contrast, the two DSPs used in the function `ntt` are both bounded to this function and cannot be used by other functions. As a consequence, if the entire NewHope scheme were implemented using HLS, it would most likely need more than just two DSPs.

3.3 SHAKE Hash Function

The hash functions SHA-3 and SHAKE play an important role in Post-Quantum Cryptography since they are widely used as PRNGs. Both make use of the same round function to permute the input and generate the output. The advantage of SHAKE is that it can take inputs of arbitrary length and generate outputs of any length, whereas SHA-3 has different fixed input and output lengths. There are numerous hardware implementations of SHAKE, the one used for reference in this section is from [HOKG18] where cSHAKE, a variant of SHAKE, is used. Both variants are very similar and should yield comparable implementation results. The software implementation used for optimization is from the software implementation of FrodoKEM [Fro20].
To test the function for multiple combinations of input and output length, the test set used for C-Validation and Cosimulation contains test cases with different input and output lengths, resulting in very different execution times. The main metric for the latency of the design is therefore the average latency of the entire test set.
Hashing using SHAKE consists of two major phases: In the first phase, the absorbing

phase, the input is XORed to the state until a full block (1344 bit for SHAKE-128, 1088 for SHAKE-256) has been absorbed. The state is then permuted, and the next input block is XORed to the state until the entire input has been used. Following the next permutation, the output can be squeezed from the state. Whenever a full block of output has been squeezed, the state is permuted again until the desired output length is reached.

3.3.1 Naive Synthesis

Listing B.3 in the Appendix displays the unoptimized software code for the SHAKE-128 hash function with all needed sub functions. The top function `shake128` calls the functions `keccak_absorb` for the absorbing phase, and `keccak_squeezeblocks` for the squeezing. Before the code can be synthesized, the `INTERFACE` *pragma* for the arrays `input` and `output` has to be added to inform the tool about their maximum size. In addition, the `INLINE off` *pragma* is added to the sub-functions to prevent the tool from inlining these functions. Listing 3.10 shows the top-level function `shake128` with the added *pragmas*.

```
1  void shake128(unsigned char *output, unsigned long long outlen, const unsigned char *input,
       unsigned long long inlen)
2  {
3      #pragma HLS INTERFACE ap_bus depth=21000 port=output
4      #pragma HLS INTERFACE ap_bus depth=10000 port=input
5      uint64_t s[25] = {0};
6      unsigned char t[SHAKE128_RATE];
7      unsigned long long nblocks = outlen/SHAKE128_RATE;
8      size_t i;
9      keccak_absorb(s, SHAKE128_RATE, input, inlen, 0x1F);
10     keccak_squeezeblocks(output, nblocks, s, SHAKE128_RATE);
11     output += nblocks*SHAKE128_RATE;
12     outlen -= nblocks*SHAKE128_RATE;
13     if (outlen)
14     {
15         keccak_squeezeblocks(t, 1, s, SHAKE128_RATE);
16         for (i = 0; i < outlen; i++)
17         output[i] = t[i];
18     }
19 }
```

Listing 3.10: Unoptimized function SHAKE-128

Following synthesis, the tool estimates a hardware usage of 12.282 FFs, 54.572 LUTs, 12 BRAMs, 16 DSPs, and an average latency of 11.792 clock cycles. Compared to the direct hardware implementation, the hardware usage is very high in every aspect, especially the large amount of BRAMs and DSPs used catches the eye. In addition, the latency of the naive HLS version is very high.
Before starting the optimization process, it might be useful to analyse where the HLS version wastes so many resources. Three major problems become apparent:

- The function `keccak_squeezeblocks` appears twice, once in line 10 where the first parameter is the array `output`, and once in line 15 where it is the array `t`. As these

arrays are of different size and therefore need addresses of different length, the tool creates two almost identical instances of the function, the only difference being the address length for the first parameter. Each instance of `keccak_squeezeblocks` uses around 3.300 FFs and 17.200 LUTs as well as 2 BRAMs. Modifying the code in a way that only one instance is created, for example by using the same array for the first parameter, might significantly reduce the hardware usage.

- The functions `keccak_absorb` and `keccak_squeezeblocks` both call the round permutation `KeccakF1600_StatePermute`, resulting in two instances of the permutation being created. The round permutation is a large component, thus creating two of it only makes sense if they can be used at the same time. However, this is not the case since `keccak_absorb` and `keccak_squeezeblocks` are called subsequently. To avoid the creation of two round permutations, the permutation has to be called directly by the function `shake128`, for example by inlining the two aforementioned functions.

- The round permutation itself is very large due to the fact that the loop in it is partially unrolled. By rolling this loop, the hardware usage of the permutation should almost be halved.

3.3.2 Function Inlining

By manually inlining the functions `keccak_absorb` and `keccak_squeezeblocks`, the second problem is solved since only one instance of the round permutation is created now that it is directly called by `shake128`. In addition, the first problem is solved as well since the two instances of `keccak_squeezeblocks` no longer exist. This optimization significantly reduces the hardware usage to 5.615 FFs, 20.189 LUTs and 7 BRAMs while not changing the usage of DSPs and the latency. The decrease in BRAM usage is due to the possible reuse of the array `t`. Before inlining the functions, there was an array `t` with 200 elements created in `keccak_absorb`, and one with 168 elements in `shake128`. After inlining, a single array `t` with 200 elements can be used for both functionalities. The reduction of FF- and LUT-usage is mainly due to the fact that only one instance of `KeccakF1600_StatePermute` is created after inlining, whereas three instances where created before.

3.3.3 Optimizing the Permutation

Currently, the internal state of the hash function is stored in an array and passed to the round function. In the round function, the state is copied from the array to local variables on which the permutation is executed. After 24 rounds, the permuted state is copied back to the state array. Since the amount of array accesses is limited to 2 per clock cycle, copying the state to and from the state array takes a total of 26 clock cycles. By fully partitioning the state array and implementing it using FFs, it is possible to execute the permutation directly on the state, making the local variables in the function unnecessary. Similar to Section 3.1, the partitioned state should not be passed to the

function as a parameter, but instead be defined as a global array outside the function. The `ARRAY_PARTITION` *pragma* can be added inside the function `shake128`. To ensure the correctness of the code, the state array needs to be set to 0 at the start of the function with an additional loop `Reset`.
Partitioning the state reduces the amount of FFs needed to 4.436 and the amount of BRAMs to 3 while slightly increasing the LUT usage. In addition, the latency of the round function is reduced from 50 to 14 clock cycles, thus significantly reducing the average latency of the design.
As mentioned above in the third point, the loop in the permutation function is partially unrolled with a factor of 2, resulting in a very low latency for the permutation, but also a high LUT usage. By rolling the loop and only writing out one iteration, the usage of LUTs can be reduced to 12.471. However, this increases the latency of the permutation function from 14 to 26 clock cycles, and therefore the average latency to 11.466 clock cycles. If a very fast implementation of SHAKE is needed, the loop can of course be kept unrolled, but for many applications of SHAKE, the speed of the rolled version seems to be sufficient.
Another possible optimization is the implementation of the round constants. Currently, they are stored in a 64-bit array with 24 elements, which is implemented in hardware using 2 BRAMs. However, as they are constants and do not need to change their value during the execution of the function, the round constants can be implemented using a `switch/case` statement setting the variable `RC` depending on the current round of the loop. Listing 3.11 shows an excerpt of the implementation of the round constants.
While this change has no influence on the latency and slightly increases the LUT usage, it saves another 2 BRAMs. The optimized code for the round permutation function with all described changes is shown in Listing B.4 in the Appendix.

```
1 switch(round)
2 {
3     case  0: RC = 0x0000000000000001ULL; break;
4     case  1: RC = 0x0000000000008082ULL; break;
5     ...
6     case 23: RC = 0x8000000080008008ULL; break;
7     default: RC = 0x0000000000000000ULL; break;
8 }
```

Listing 3.11: Implementation of the SHAKE round constants using `case`-statements (Excerpt).

3.3.4 Optimizing the Absorbing Phase

The implementation in its current form assumes an unpadded input with a bit length that is a multiple of 8 (only full bytes). If, however, the input were already padded and the bit length of the input were a multiple of 64 (or the byte length a multiple of 8), the code can be simplified, saving both resources and latency. Achieving this assumption is rather easy in cryptography since the sizes of arrays are generally known. Therefore,

it is fairly easy to add the padding at the end of the data to be hashed, and to fill the remaining array elements with zeros until the size of the array is a multiple of 8, assuming an 8 bit array.
Listing 3.12 shows the code for the optimized absorbing phase that assumes a padded input. The conditional statement in line 3 checks if a full block can be absorbed, setting `end` accordingly, or if the remaining input is less than a full block. In `L1`, the input is absorbed, 8 bytes at a time, and XORed to the state. The conditional statement in line 11 is necessary to avoid the state being permuted once too often: If no full block was absorbed, the permutation happens at the start of the squeezing phase, and is not needed in the absorbing phase. In addition, the padding is terminated in this case.
This change reduces the hardware usage to 4.042 FFs and 11.679 LUTs while also decreasing the average latency by about 500 clock cycles. The reduced latency is mainly due to the fact that the array `t` is no longer accessed in the absorbing phase.
To further optimize the absorbing phase, it is worth noting that the left shift in the function `load64` is implemented as a true left shift, costing many LUTs. However, a true left shift is not necessary. By fully unrolling the loop inside the function `load64`, two improvements can be achieved at once: With the loop disappearing due to the unrolling, the left shift is no longer implemented as a true left shift, but as eight left shifts with a fixed offset. In addition, the process of setting `r` now only takes 8 clock cycles instead of 16. Since the function `load64` only appears once in the code, it also makes sense to inline it to save a few more FFs and LUTs.
These optimizations result in a hardware usage of 4.002 FFs and 11.417 LUTs, a slight decrease in FF usage and a rather large decrease in LUT usage. Furthermore, the average latency is reduced to 9.148 clock cycles, a decrease by over 2.000 cycles.

```
1  Absorb: while(inlen > 0)
2  {
3      if(inlen >= SHAKE128_RATE)
4          end = SHAKE128_RATE >> 3;
5      else
6          end = inlen >> 3;
7      L1: for(i = 0; i < end; i++)
8      {
9          s[i] ^= load64(input + 8*i);
10     }
11     if(end == (SHAKE128_RATE >> 3))
12         KeccakF1600_StatePermute();
13     else
14         s[20] ^= 0x8000000000000000;
15     inlen -= (end << 3);
16     input += (end << 3);
17 }
```

Listing 3.12: Optimized SHAKE absorbing phase, assuming a padded input.

3.3.5 Optimizing the Squeezing Phase

The calculation of the variable `nblocks` in line 7 of Listing 3.10 needs 16 DSPs since the operand `outlen` and the result are 64 bit variables. Avoiding the use of `nblocks` could therefore eliminate all the DSPs that are currently used by the function. Instead of using `nblocks` and decrementing it after every iteration of the `while`-loop in `keccak_squeezeblocks`, it is sufficient to use `outlen` and to decrement it by `SHAKE128_RATE` after every iteration. The only change to be made is the condition of the `while`-loop: it has to be rewritten from `while(nblocks > 0)` to `while(outlen >= SHAKE128_RATE)`. This simple change makes all 16 DSPs unnecessary, saves over 1.300 FFs and 1.000 LUTs and reduces the average latency by about 70 clock cycles.
In its current form, the hash function can produce an output of any given length, as long as the output length is a multiple of 8. However, similar to the absorbing phase, it makes sense to assume that the output length is not only a multiple of 8, but also of 64, to reduce the hardware cost of the squeezing phase. With this assumption, it is possible to change the `while`-loop in `squeeze_blocks` in a similar way to that in the absorbing phase, making the `if`-condition in line 13 of Listing 3.10 unnecessary. The optimized code is shown in Listing 3.13.
After synthesis, the tool estimates a hardware usage of 2.452 FFs and 10.008 LUTs. Furthermore, the one BRAM previously necessary to implement the array `t` is not needed any more as the array is no longer used in the code. The average latency is reduced by around 300 clock cycles.
As the function `store64` now appears only once in the code, it makes sense to inline it, slightly reducing the hardware usage and the maximum latency. Unlike in the squeezing phase, unrolling or pipelining the loop of the former function `store64` has no benefit since each loop iteration takes only 1 clock cycle which cannot be sped up.

```
1  Squeeze: while(outlen > 0)
2  {
3      KeccakF1600_StatePermute();
4      if(outlen >= SHAKE128_RATE)
5          end = SHAKE128_RATE >> 3;
6      else
7          end = outlen >> 3;
8      L2: for (i = 0; i < end; i++)
9      {
10         store64(output+8*i, s[i]);
11     }
12     output += (end << 3);
13     outlen -= (end << 3);
14 }
```

Listing 3.13: Optimized SHAKE squeezing phase.

3.3.6 Further Optimizations

The loop `Reset` currently takes 50 clock cycles to complete. However, since the state array is implemented using FFs, it is possible to access and set every element at the same

time. By fully unrolling the loop using the `UNROLL` *pragma*, the latency of the loop can be reduced to just one clock cycle.
A further decrease in latency is possible by changing the data type of the input and output array. Currently, both arrays are implemented as 8 bit arrays, and it takes 8 clock cycles to complete the innermost loop in the absorbing or squeezing phase. If both arrays were implemented as 16 bit arrays, the latency of these loops could be reduced to 4 clock cycles, which can be a massive improvement if a long input or output has to be treated. However, this change is only possible if the function that calls SHAKE can work with 16 bit arrays.
Changing the data type of input and output arrays to 16 bits slightly increases the usage of FFs and LUTs, but significantly reduces the average latency to about 5.500 clock cycles. Using even larger data types for the arrays, especially 32 or 64 bit, could further reduce the latency while not having a large effect on the hardware usage. However, arrays with larger data types entail other drawbacks such as the increased BRAM usage to implement them.
If the maximum length of input and output is known, it is also possible to change the data type of the input parameters `inlen` and `outlen`, which are currently defined as 64 bit variables. This allows for extremely large inputs and outputs, but might not be needed in many applications of SHAKE. For the test cases used in this work, the maximum of input and output length is about 20.000 bytes, which allows to define the two parameters as 16 bit values. Changing the type to `uint16_t` saves about 100 FFs and 150 LUTs. In some cases with even smaller input or output lengths, defining the parameters as `uint8_t` might be possible. If one of the two parameters were definitely smaller than `SHAKE128_RATE`, the `while`-loops of the absorbing or the squeezing phase could be omitted since they would be executed only once. This can save a small amount of FFs and LUTs, but only works in very special scenarios.
The design is highly flexible regarding frequency. It is currently running with a frequency of 114,29 MHz, but even a frequency of 190,48 MHz results in only a very slight increase of hardware usage and 1 additional clock cycle of latency. Even higher frequencies might be possible if needed.
Finally, the way input and output are passed to the function can be changed to pass-by-value. The `INTERFACE` *pragma* then needs to have the option `ap_memory` instead of `ap_bus`, which allows for two read and write accesses on the passed arrays instead of one. As a consequence, the absorbing and squeezing phase can be completed much quicker since two elements can be absorbed or squeezed in one clock cycle. This change saves about 140 FFs, but needs an additional 100 LUTs. The average latency is reduced by almost 1.500 to 4.072 clock cycles while the maximum frequency is increased to 198,74 MHz. The final code for the optimized implementation of `shake128` is displayed in Listing B.5 in the Appendix.

3.3.7 Comparison with the Direct Hardware Implementation

Table 3.3 shows the synthesis results for the different optimization steps of SHAKE and the direct hardware implementation. The HLS version uses more hardware than

Implementation	FF	LUT	BRAM (18K)	DSP	Latency (Cycles)	Frequency (MHz)
Naive HLS	12.282	54.572	12	16	11.792	114,29
Function Inlining	5.615	20.189	7	16	11.787	114,29
Optimizing Permutation	4.433	12.565	1	16	11.466	114,29
Optimizing Absorb	4.002	11.417	1	16	9.148	114,29
Optimizing Squeeze	2.318	9.900	0	0	8.067	114,29
Further Optimizations	2.128	9.881	0	0	4.072	198,73
Final Synthesis	1.972	5.249	0	0	4.072	198,73
[HOKG18]	1.685	2.744	0	0	-	172

Table 3.3: Synthesis results for the different optimization steps of the HLS version and the direct implementation of the SHAKE hash function. The latency shows the average latency for the entire test set.

the direct implementation, especially the usage of LUTs is very high with over 5.000, compared to less than 3.000 in the direct implementation. This can partly be due to the higher frequency of the HLS version, but the frequency alone does not explain the big difference. A better result seems to be impossible to achieve with HLS. However, if SHAKE is called by another function, as is the case in FrodoKEM in Section 4.2, the SHAKE modules with an identical code have a smaller hardware usage of about 2.000 FFs and 3.000 LUTs. This hardware usage is much closer to that of the direct implementation, albeit still about 10 % higher.

3.4 Discrete Gaussian Sampling

Random sampling from a given distribution takes a crucial role in cryptography since the quality of sampled values and the underlying distribution can widely influence the security of a scheme. Sampling is usually done using either a binomial or a Gaussian distribution. For this work, the discrete Gaussian sampler used in the FrodoKem encapsulation scheme is analysed. Its functionality is briefly described in Section 2.4. The unoptimized `C`-code, taken from the software implementation of FrodoKEM, is shown in Listing 3.14. For the comparison, the direct implementation made for this work is used. The test set contains cases with different input lengths, therefore the average latency is used to measure the execution time.
Before HLS can be performed on the `C`-code, the `INTERFACE` *pragma* has to be added for the parameter `s` since it is passed by reference. Following synthesis, the tool reports a hardware usage of 162 FFs and 271 LUTs with an average latency of 107.287 clock cycles and a maximum frequency of 114,29 MHz. The main optimization potential lies within the loop `L1_1`, where the value of `sample` is determined on the basis of the first 15 bits (named `prnd` in the code) of the input. The difference between the entry of the CDF-table and `prnd` is calculated, and the most significant bit of it is added to `sample`. As long as `prnd` is smaller than the entries in the CDF-table, 1 is added in every iteration,

```
1  uint16_t CDF_TABLE[13] = {4643, 13363, 20579, 25843, 29227, 31145, 32103, 32525, 32689, 32745,
       32762, 32766, 32767};
2  uint16_t CDF_TABLE_LEN = 13;
3  void frodo_sample_n(uint16_t *s, const size_t n)
4  {
5      #pragma HLS INTERFACE ap_bus depth=5120 port=s
6      unsigned int i, j;
7      L1: for (i = 0; i < n; ++i)
8      {
9          uint8_t sample = 0;
10         uint16_t prnd = s[i] >> 1;
11         uint8_t sign = s[i] & 0x1;
12         L1_1: for (j = 0; j < (unsigned int)(CDF_TABLE_LEN - 1); j++)
13         {
14             sample += (uint16_t)(CDF_TABLE[j] - prnd) >> 15;
15         }
16         s[i] = ((-sign) ^ sample) + sign;
17     }
18 }
```

Listing 3.14: Unoptimized Gaussian sampling of FrodoKEM

but once `prnd` is larger, only zeroes will be added.
To ensure that the function runs in constant time, `L1_1` loops over all the entries in the CDF-table, even if multiple zeroes are added this way. With a total of 12 iterations, this loop takes 24 clock cycles every time it is executed. In hardware, constant time can be achieved even if multiple conditional statements are present in the code. Replacement of `L1_1` by multiple conditional statements allows to obtain the value of `sample` in one clock cycle. Listing 3.15 shows an excerpt of the conditional statements used to determine the value of `sample`. This change increases the LUT usage to 420 due to the large amount of comparators needed, but also reduces the average latency to 19.168 clock cycles, a fifth of the initial value.

```
1  if(prnd > 32766)
2      sample = 12;
3  else if(prnd > 32762)
4      sample = 11;
5  ...
```

Listing 3.15: Calculation of `sample` using only conditional statements instead of looping over the entire CDF-table (Excerpt).

In a similar way, the sign of the sampled value (line 16 in Listing 3.14) is currently determined using multiple operations, rather than a conditional statement. By using a conditional statement checking whether `sign` is positive or negative and setting the result accordingly, another 20 LUTs can be saved.
Every iteration of `L1` currently takes four clock cycles to complete. Using the `PIPELINE` *pragma*, the effective latency can be reduced to three clock cycles per iteration at the cost of a slightly increased LUT usage. A further reduction of the iteration latency is impossible with the current design since the sampled results are used to overwrite the

inputs. However, if the results were written into a different array, the effective iteration latency could be reduced to just one clock cycle. Writing the sampled values into a second arrays decreases the average latency to 3.861 clock cycles, but also increases the hardware usage to 243 FFs and 504 LUTs. Whether or not this change can be applied depends on the way the sampler is used in a design.
The parameter `n`, defining how many values are to be sampled, currently has the data type `size_t`, which is synthesized as a 64 bit value. In the case of FrodoKEM where this function is used, the maximum input length is 5.120 for the smallest parameter set. Therefore, it is sufficient for `n` to be a 13 bit variable. Changing the data type to `uint13` can slightly reduce the FF usage of the design. Similarly, the data types of `prnd` and `sign` can be changed to `uint15` and `uint1`, respectively. These changes reduce the FF usage to 189.
`sample` is currently set using a long cascade of conditional statements with one `if` case and many `else if` cases. By reordering the conditional statements in a similar way to a binary search tree, another 13 LUTs can be saved. Listing 3.16 shows the beginning of the changed `if/else` statements.

```
1  if(prnd > 32525)
2  {
3      if(prnd > 32762)
4      {
5          if(prnd > 32766)
6              sample = 12;
7          else
8              sample = 11;
9          }
10     else if(prnd > 32689)
11     {
12         if(prnd > 32745)
13             sample = 10;
14         else
15             sample = 9;
16         }
17 ...
18 }
```

Listing 3.16: Calculation of `sample` using a balanced comparator tree (Excerpt).

The frequency can be increased by selecting a higher frequency and thereby forcing the tool to implement the function accordingly. At a frequency of 190,51 MHz, the used hardware is only increased by 11 FFs, and even higher frequencies might be possible if needed. However, for many of the applications including FrodoKEM, a frequency of 190 MHz should be more than sufficient.
Finally, instead of passing pointers to the input and output arrays, the arrays can be entirely passed to the function. This has no influence on the latency, but slightly reduces the hardware usage to 180 FFs and 390 LUTs. Furthermore, passing the arrays makes it possible to specify them as dual-port BRAMs, allowing to access and sample two values at once. However, sampling two values at once also requires twice the amount of hardware for the additional comparators. The decision whether this is needed or not

Implementation	FF	LUT	BRAM (18K)	DSP	Latency (Cycles)	Frequency (MHz)
Naive HLS	162	271	0	0	107.287	114,29
Conditional Statements	155	399	0	0	19.168	114,29
Pipeline	243	504	0	0	3.861	114,29
Further Optimizations	180	390	0	0	3.860	190,51
Final Synthesis	55	83	0	0	3.860	190,00
Direct Implementation	10	113	0	0	3.860	167,00

Table 3.4: Synthesis results for the different optimizations steps of the HLS version and the direct implementation of the discrete Gaussian sampler. The latency shows the average latency over the entire test set.

depends on the use case of the function and the available hardware resources. For this work, the version with only one sample at a time is used.
In Table 3.4, the hardware usage and average latency are listed for the described optimization steps. After synthesis and implementation of the generated VHDL code, the synthesis tool reports a hardware usage of only 55 FFs and 83 LUTs with a frequency of 190 MHz.
By comparison, the direct implementation uses less FFs with only 10, but needs 30 additional LUTs, in all a very similar hardware usage. Both implementations have the same average latency, but the HLS version has a higher frequency of 190 MHz compared to 167 in the direct version. However, the frequency of the direct implementation is limited by other components and could potentially be higher if only the sampler were analysed.

4 Implementation of FrodoKEM

This chapter describes two hardware implementations of FrodoKEM, one created by implementing directly in hardware and the other using the HLS tool based on the existing software implementation. Because a direct hardware implementation of FrodoKEM already exists, the main purpose of the implementation in this work is to serve as a reference for the HLS version regarding development time. At the end of the chapter, the three implementations are briefly compared. The target device for both the direct implementation and the HLS version is a Xilinx Artix-7 FPGA. The HLS tool used is again Xilinx Vivado HLS 2020.1, and the synthesis tool is again Xilinx Vivado 2020.1.

4.1 Hardware Implementation of FrodoKEM

The direct hardware implementation in this work follows the design idea proposed in [HOKG18], which is briefly described in this section. Three main components are needed for key generation, encapsulation and decapsulation: SHAKE to generate pseudorandom data, a discrete Gaussian sampler and the matrix multiplication. Additional components include packing and unpacking as well as encoding and decoding. The operations for key generation, encapsulation and decapsulation being very similar, they are only described for the encapsulation.

To implement the matrix-matrix multiplications, a vector-matrix multiplier is used that multiplies one row of $\mathbf{S}'$ on the left with the matrix on the right. This has to be done eight times for the eight rows of $\mathbf{S}'$. To avoid storage of the entire matrix $\mathbf{S}'$, a double-buffered store with two BRAMs is used. In a pregeneration phase, the first row of $\mathbf{S}'$ is generated using a SHAKE module together with the discrete Gaussian sampler and stored in the first BRAM. Once the generation has finished, the second row can be generated and stored in the second BRAM. At the same time, the first row can be multiplied with the matrix on the right using the vector-matrix multiplier. Once both operations are completed, the roles of the BRAMs are switched: The next row is generated into the first BRAM, and the row in the second is used in the vector-matrix multiplier. This way, only 2 instead of 8 BRAMs are needed to store $\mathbf{S}'$, and the generation of $\mathbf{S}'$ happens in parallel to the multiplication, and therefore has no influence on the total latency of the design.

As soon as one vector-matrix multiplication has finished, the result is input into a second SHAKE module that computes the shared secret $\mathbf{ss}$ in parallel to the next multiplication. Thus, the hashing does not account for any additional latency.

Inside the vector-matrix multiplication, the generation of $\mathbf{A}$ is done in a very similar way to that of $\mathbf{S}'$. Storing the entire matrix would exceed the storage capacity of many FPGAs, therefore the double-buffer technique is used again. After the first row has been

generated and stored using yet another SHAKE module, the next row is generated in parallel to the multiplication of the previous row with the row of $\mathbf{S}'$. Apart from the pregeneration of the first row, the generation of $\mathbf{A}$ does not account for the total latency of the design.
For the multiplications, a single DSP is used that executes a multiplication in every clock cycle. To increase the frequency, a pipelined design is used where the addition of the error term does not happen in the same clock cycle as the multiplication, but in the second clock cycle instead. In the third clock cycle, the result is written into the BRAM. This design ensures that the implementation always runs in constant time.
A total of three SHAKE modules is used in the implementation. The first module is combined with a discrete Gaussian sampler to sample the matrices $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$ according to a Gaussian distribution. Its functionality is briefly described in Section 2.4. However, instead of looping over the entire CDF-table in the sampler, a large amount of comparators is used to instantly output the sampled value. A second SHAKE module is used to generate the rows of $\mathbf{A}$ based on a seed and the row index, and the third module is used to compute the shared secret.
The implementation results of key generation, encapsulation and decapsulation as well as the total latencies are shown in Tables 4.1, 4.2, and 4.3, respectively. Apart from a short initialization phase, the latency is only defined by the number of multiplications that have to be completed. The initialization takes no longer than 20.000 clock cycles, a negligible amount compared to the over three million clock cycles needed for the multiplications.

4.2 HLS of FrodoKEM

In this section, the existing software implementation of FrodoKEM [Fro20] is transformed into a hardware implementation using HLS. For each of the three algorithms key generation, encapsulation and decapsulation, the changes made to the `C`-code and the resulting changes in hardware usage and latency are described.

4.2.1 Key Generation

The algorithm for the key generation has been displayed in 2.4.1 earlier in this work. The initial true randomness and the two arrays `pk` and `sk` are passed to the function using a pointer. The top-level function `crypto_kem_keypair` calls various sub-functions such as `shake` to evaluate SHAKE, `frodo_sample_n` for Gaussian sampling and `frodo_pack` for the packing.

Naive HLS

Before the HLS can be performed, a few changes to the code have to be made. For the three parameter arrays `pk`, `sk` and `randomness`, the `INTERFACE` pragma has to be added to inform the synthesis tool about the maximum size of these arrays. Furthermore, instances of the function `memcpy` have to be replaced by code that copies the content from one array to another. While `memcpy` can be synthesized by the HLS tool, this is

only possible if one of the parameters of the function is a bus (like the passed arrays), and the other one a local memory (array). In addition, both arrays need to have the same width data type. The different appearances of `memcpy` breach at least one of the two requirements and therefore need to be replaced, either by a self-written function `copy` that writes the content of one array to another, or by a loop that does the same. Since `memcpy` gets synthesized as a loop and not as a function, it cannot be executed in parallel to other instructions. Therefore, replacing it by a loop should result in a very similar performance, whereas a new function can be expected to be faster since it might be executed in parallel to other functions.
In addition, some calls to `shake` contain an implicit pointer reinterpretation that has to be changed. One of these pointer reinterpretations is shown in Listing 4.1. The function `shake` is defined on 8 bit arrays, but the 16 bit array `S` is passed to it with a cast to `uint8_t`. This is possible in `C`, but not in hardware since the size and type of a BRAM on an FPGA cannot be changed while it is running. To circumvent this problem, two solutions are possible: First, the function could be reworked to operate on 16 bit arrays which makes the pointer reinterpretation unnecessary. However, this is a rather large change to the code, and the goal of the naive HLS is to show the performance of the code with as little changes as possible. The second solution is to make use of an additional 8 bit array that is passed to `shake`. After the function has finished, the contents of the additional array are written into the 16 bit array by concatenating two elements. This solution is very simple to implement, yet it results in an increased latency for the copying, and a higher BRAM usage for the second array.

```
1    shake((uint8_t*)S, 2*PARAMS_N*PARAMS_NBAR*sizeof(uint16_t), shake_input_seedSE, 1 +
         CRYPTO_BYTES);
```

Listing 4.1: Pointer reinterpretation in the FrodoKEM key generation.

Using the second solution, the HLS tool estimates a hardware usage of 18.634 FFs, 43.525 LUTs, 1.073 BRAMs and 1 DSP. The total latency of the key generation is 17.332.462 clock cycles with a maximum frequency of 114,29 MHz. These results are extremely bad compared to the direct hardware implementation; especially the very high BRAM usage and the high latency stand out. Two major changes can significantly reduce the hardware usage and the latency.
To reduce the usage of BRAMs, it is necessary to avoid storing the matrices needed for the matrix multiplication entirely, especially the very large matrix $\mathbf{A}$. Instead, enough of this matrix can be generated and stored to allow the start of the matrix multiplication. The rest of $\mathbf{A}$ can then be generated while the multiplication is running.
A considerable reduction in latency can be achieved by pipelining the inner-most loop in the matrix multiplication using the `PIPELINE` *pragma*. Without the pipeline, every iteration of said loop takes 3 clock cycles to complete, resulting in almost 10.000.000 clock cycles to complete the entire multiplication. By contrast, when using the pipeline, a new loop iteration can start every clock cycle, reducing the latency for the matrix multiplication to about a third of its original latency.

First Optimizations

Before applying the two major optimizations steps, a few smaller changes to the code can be applied to facilitate further optimizations. First of all, the function for the matrix multiplication, `frodo_mul_add_as_plus_e`, can be inlined since it only appears once in the code. This slightly reduces the LUT usage.
The code currently uses the unoptimized generic version of the SHAKE hash function, which can treat any input and output length, but is rather large and slow. As some applications of SHAKE in the code always have the same input and output length, a much more specialized instance, capable only of treating a fixed input and output length, can be used, reducing hardware usage and latency. The generation of the rows of **A** is one of these applications where a specialized version of SHAKE makes sense. With an input length of only 18 byte, the absorbing phase is very simple and does not need any loops. As the output length is a multiple of 64 bit, the squeezing phase can also be simplified as described in Section 3.3. In addition, this instance can be optimized to work on 16 bit instead of 8 bit arrays. This does not only speed up the function, but more importantly eliminates the need for the second array that was used to circumvent the problem of pointer reinterpretation. As a consequence, the BRAM usage is almost halved to 559, and the latency is reduced to 11.183.048 clock cycles after applying these changes. The usage of LUTs and FFs is also reduced. The resulting function `shake128_10240` (named after the output length of 10.240 byte) is displayed in Listing B.6 in the Appendix.
The arrays `A` and `S` are currently zero-initialized. However, this is not necessary as they are written first before being read. Removing the initialization slightly reduces the hardware usage and saves around 470.000 clock cycles.
Before the matrix multiplication starts, the contents of the error matrix **E** are copied to the result matrix **B**, and the accumulated sums of the multiplication are then added to **B**. This leads to many unnecessary array accesses, slowing down the design. Instead, the accumulated sum and the corresponding element of **E** can be added and only then be written to **B**, reducing the latency by over 10.000 clock cycles.

Reducing the BRAM Usage

To reduce the very high BRAM usage, it is important to optimize the generation and storage of the matrix **A** since it accounts for 512 of the 559 currently used BRAMs. As the rows of **A** can be generated independently, it is possible to only generate and store one row of **A** at a time, multiply it with a column of **S**, and then generate the next row of **A**. While only one BRAM would be needed to store **A**, this design would be very slow as **A** would have to be generated a total of 8 times for the multiplication with the 8 columns of **S**.
Instead, two BRAMs can be used to store **A**. The idea, often referred to as double-buffered store, page-flip method or ping-pong buffer, is to use two arrays `A_1` and `A_2` to implement **A**. The first row of **A** is generated and stored in `A_2` during the pre-computation phase. Once the row has been generated, the vector-vector multiplication with the first column of **S** can start using `A_2`. In parallel to this multiplication, the generation of the second

```
1  uint16_t vector_vector_mul(uint16_t *a, uint16_t *s)
2  {
3      #pragma HLS INLINE off
4      uint16_t i, sum = 0;
5      for(i = 0; i < PARAMS_N; i++)
6      {
7          sum += a[i] * s[i];
8      }
9      return sum;
10 }
```

Listing 4.2: Function `vector_vector_mul`, multiplying two vectors and returning the accumulated sum.

row of **A** can happen, but this row is stored in `A_1`. As the two operations do not access the same BRAMs, they can be executed completely in parallel, and the latency of the two operations is only defined by whichever operation takes longer to complete. In this case, the multiplication takes about 1.920 clock cycles, whereas the row generation takes less than 600 cycles. Once the first vector-vector multiplication has been completed, the second can start immediately. This time, the roles of the two arrays are switched: `A_1` is used for the multiplication, and the third row of **A** is generated into `A_2`. This procedure is continued until the entire multiplication has been completed. Apart from the latency needed for the pre-generation, the latency of the matrix multiplication is only defined by the latency of the vector-vector multiplication.
To achieve the described behaviour in HLS, the generation of rows and the vector-vector multiplication both have to be independent functions that do not share any parameters. The generation of rows is already implemented in the function `shake128_10240`, but the vector-vector multiplication is currently not implemented as a function. Listing 4.2 shows the function `vector_vector_mul` that implements the multiplication of two vectors. The `INLINE off` *pragma* has to be added to prevent the tool from inlining the function because an inlined function cannot run in parallel to the row generation.
Listing 4.3 shows the matrix multiplication using the ping-pong buffer for **A**. In lines 3 and 4, the first row of **A** is pre-generated into `A_2`. The loop iterates through the rows of **A** and multiplies them with a column of **S**. In iterations with an even `k` (lines 10 and 11), a row is generated into `A_1`, and the multiplication happens from `A_2`. When `k` is odd (lines 15 and 16), the arrays are switched. The two functions `shake128_10240` and `vector_vector_mul` can be executed at the same time as they do not share an array.
Using a ping-pong buffer reduces the BRAM usage to 49 since only 2 are needed to implement **A**, and the latency is reduced to around 10.150.000 clock cycles. The decrease in latency is due to the fact that the generation of **A** happens in parallel to the vector-vector multiplications and therefore has no effect on the latency.

Pipelining the Vector-Vector Multiplication

The function `vector_vector_mul` currently takes 1.923 clock cycles to complete since every iteration of its loop takes 3 clock cycles. In the first two clock cycles, the factors

```
1  Mat_Mul: for (i = 0; i < PARAMS_NBAR; i++)
2  {
3      seed_A[0] = 0;
4      shake128_10240(A_2, seed_A);
5      for (k = 0; k < PARAMS_N; k++)
6      {
7          seed_A[0] = k+1;
8          if((k & 0x1) == 0)
9          {
10             shake128_10240(A_1, seed_A);
11             sum = vector_vector_mul(A_2, S+i*PARAMS_N);
12         }
13         else
14         {
15             shake128_10240(A_2, seed_A);
16             sum = vector_vector_mul(A_1, S+i*PARAMS_N);
17         }
18         B[k*PARAMS_NBAR + i] = sum + E[k*PARAMS_NBAR + i];
19     }
20 }
```

Listing 4.3: Matrix multiplication of the key generation using a ping-pong buffer for **A**.

are loaded from the BRAMs and multiplied, the product is added to the accumulated sum in the third clock cycle. By pipelining the loop, a new loop iteration can be started every clock cycle, reducing the effective iteration latency to just one clock cycle. The total latency of the vector-vector multiplication is therefore reduced to 643 cycles. This change reduces the latency of the key generation to only 3.603.031 clock cycles while barely changing the hardware usage. As a consequence of this change, the latency of the design is already rather close to that of the direct hardware implementation.

Generation of S and E

The matrices **S** and **E** are currently generated using a generic instance of SHAKE with 8 bit inputs and outputs (Listing 4.1). However, **S** and **E** are supposed to be matrices with 16 bit elements. As described for the naive HLS, a second array for both **S** and **E** is needed to work around the pointer reinterpretation. By creating an optimized and specialized instance `shake_gen_S` of SHAKE with 16 bit outputs, the need for the second arrays can be avoided. The function is very similar to `shake128_10240`, but with a different input and output length. To avoid conflicts on the internal state of the hash functions, `shake_gen_S` operates on a different internal state than `shake128_10240`. Using the optimized SHAKE instance reduces the hardware usage to 13.149 FFs, 41.402 LUTs and 32 BRAMs, and also reduces the latency to 3.482.736 clock cycles.
Similar to the generation of **A**, it is also possible to optimize the generation of **S** and **E** using a ping-pong buffer. As the two matrices do not have to exist at the same time (**S** is needed first for the multiplication with **A**, then **E** is added to the result), it is not necessary to have two arrays for **S** and two for **E**. Instead, the two arrays can first be used to store **S**, and as soon as that is no longer needed, to store **E**. By contrast to **A**, the columns of **S** and **E** cannot be generated independently. To allow generating only

one column at a time, it is therefore necessary to modify `shake_gen_S` in a way that the internal state is not reset with every call to the function. In addition, it has to allow the squeezing to stop and resume somewhere in the middle of a SHAKE block since a column of the matrix does not have the same length as a SHAKE block. Keeping track of where squeezing was stopped and is to be resumed can be done with a static variable that keeps its value when the function is called again.
After a pre-generation phase where the first column of **S** is generated, the generation of the next column and the multiplication of the first one with **A** can happen in parallel. In order to be able to execute the vector-vector multiplications and `shake_gen_S` in parallel, it is necessary to implement the multiplications using a function `vector_matrix_mul`. This function generates the entire matrix **A** and multiplies it with one column of **S** using vector-vector multiplications, and writes the results into the matrix **B**. The modified code of the matrix multiplication is shown in Listing 4.4.

```
1  Mat_Mul: for(i = 0; i < PARAMS_NBAR+1; i++)
2  {
3      ...
4      if((i & 0x1) == 0)
5      {
6          shake_gen_S(S_2, seed_SE, 160, reset, reset);
7          frodo_sample_n(S_2, PARAMS_N);
8          write_sk(sk_S+2*i*PARAMS_N, S_2, begin_write_sk);
9          vector_matrix_mul(B, A_1, A_2, S_1, seed_A, i-1, begin_matrix_mul);
10     }
11     else
12     {
13         shake_gen_S(S_1, seed_SE, 160, reset, reset);
14         frodo_sample_n(S_1, PARAMS_N);
15         write_sk(sk_S+2*i*PARAMS_N, S_1, begin_write_sk);
16         vector_matrix_mul(B, A_1, A_2, S_2, seed_A, i-1, begin_matrix_mul);
17     }
18 }
```

Listing 4.4: Optimized matrix multiplication of the key generation. The generation of S is done in parallel to the multiplication.

Some further aspects are worth mentioning:

- The function `write_sk`: This function writes the columns of **S** to the secret key. In the original code, this happens after the matrix multiplication. However, since **S** is not stored entirely any more, this has to happen earlier, before the current column is overwritten. As the code to write the secret key appears two times in the code (once in the `if`-case in line 19 and once in the `else`-case), it makes sense to write a function to avoid creating the same hardware twice.

- One additional loop iteration: In Listing 4.3, the loop `Mat_Mul` had a total of 8 iterations, now it has 9. This is due to the fact that the HLS tool sometimes has problems with the synthesis process when a function appears both outside and inside a loop, which is the case for the function `shake_gen_S`. For the pre-generation of the first column of **S**, it appears before the loop, and then inside the loop for

the generation of the remaining columns. To circumvent this problem, one loop iteration is added.

- Additional parameters for `write_sk` and `vector_matrix_mul`: The additional parameters `begin_matrix_mul` and `begin_write_sk` have to do with the previously mentioned point. In the first iteration, the first column of **S** is generated and stored in `S_2`, but the matrix multiplication cannot start yet. This is ensured by `begin_matrix_mul`, which is 0 in the first and 1 in the remaining iterations. Starting from the second iteration, the functions can be executed normally until the very last iteration. In this iteration, `shake_gen_S` generates the first row of **E**, which need not be written to the secret key. Therefore, the function `write_sk` is not executed in the last iteration. This is controlled by the parameter `begin_write_sk`.

Applying the ping-pong buffer for **S** increases the LUT usage by around 300 for the additional control logic and the new function `write_sk`, but it also reduces the latency by about 15.000 clock cycles.
To implement **E**, the two arrays `S_1` and `S_2` can be reused since they are no longer needed for **S**. The first column of **E** has already been generated into `S_2` in the last iteration of `Mat_Mul`, therefore the addition can immediately start. It is implemented using a new function `add_E`, allowing it to run in parallel to `shake_gen_S` that generates the next row of **E**. This saves another 10.000 clock cycles due to the parallel execution of functions while slightly increasing the hardware usage for the increased control logic.

Data Type of Parameters

The three arrays `randomness`, `pk` and `sk` currently have the data type `uint8_t`, whereas most of the computations inside the function such as the generation of rows and columns and the vector multiplication operate on arrays of type `uint16_t`. Changing the data type of the parameters to `uint16_t` could have multiple positive effects on the performance of the hardware implementation: Reading and writing the parameters, especially `pk` and `sk`, can be done in less clock cycles as the amount of bits that can be read/written per clock cycle is doubled. In addition, the key generation would only operate on 16 bit values, making splitting up a 16 bit value or combining two 8 bit values unnecessary. This should improve both the latency and the hardware usage. Finally, further optimized functions can be used. The design currently uses two completely unoptimized instances of `shake128` which operate on 8 bit values. When only working with 16 bit values, the optimized version from Section 3.3 could be used, which is significantly faster and requires less hardware than the unoptimized one. In a similar way, the function `frodo_pack` can be optimized to work on 16 bit arrays. The new packing function is named `frodo_pack_16`. When all three parameters are changed to a 16 bit data type, and the optimized functions for `shake128` and `frodo_pack` are used, the hardware usage is reduced to 6.847 FFs and 15 BRAMs, while the usage of LUTs is increased 42.427. The latency is reduced by around 90.000 to 3.381.098 clock cycles.

Frequency

The design is currently synthesized with the default settings for the clock period (10 ns), resulting in a maximum frequency of 114,29 MHz. By manually increasing the frequency, the HLS tool can be forced to add registers at some points to reduce the length of the critical path. Increasing the frequency to 156,54 MHz comes at the cost of 400 additional FFs and 100 LUTs, while also increasing the latency by almost 5.000 clock cycles. However, the design is executed almost 50% faster due to the higher frequency. The additional hardware is mainly needed in the three functions `add_E`, `frodo_sample_n` and `frodo_pack`.
An even higher frequency can be achieved when the parameters `randomness`, `pk` and `sk` are passed by value instead of being passed by reference. At a frequency of 167 MHz, an additional register is added in the function `vector_vector_mul` to store the result of the multiplication before adding it to the accumulated sum. This increases the total latency by 7.000 clock cycles and the FF usage by 100, but allows to meet the frequency of the direct hardware implementation.

Further Optimizations

The two major optimization steps being done, there are a few smaller improvements left that help to further reduce both the hardware usage and the latency. First, the loop `Mat_Mul` from Listing 4.4 can be optimized. The three functions `shake_gen_S`, `frodo_sample_n` and `write_sk` are all dependent of `S_2` (or `S_1` in the `else`-case) and therefore have to be executed one after another, whereas `vector_matrix_mul` depends on the other array. Hence, it should be possible to execute the first three functions in parallel to the fourth. However, this is not how the HLS tool schedules the functions. Instead, it executes `shake_gen_S` and `vector_matrix_mul` in parallel, and once both functions have finished, starts `frodo_sample_n` and then `write_sk`. This results in almost 3.000 additional clock cycles per iteration, or 24.000 additional cycles in total.

```
1 void gen_S_sample_write(uint16_t *S, uint16_t *seed, uint16_t outlen, uint8_t reset, uint16_t *sk,
      uint8_t begin_write)
2 {
3     #pragma HLS INLINE off
4     shake_gen_S(S, seed, outlen, reset);
5     frodo_sample_n(S, 640);
6     write_sk_16(sk, S, begin_write);
7 }
```

Listing 4.5: Function `gen_S_sample_write` that calls the three sub-functions `shake_gen_S`, `frodo_sample_n` and `write_sk_16`.

To circumvent this problem, the three functions can be replaced by one function that simply calls them in the right order. The resulting function `gen_S_sample_write` is displayed in Listing 4.5. With this change, the tool executes the two functions `vector_matrix_mul` and `gen_S_sample_write` in parallel, saving almost 24.000 clock cycles while slightly increasing the hardware usage.

The 8 bit array `shake_input_seedSE` is currently serving as an intermediate array: It is first written to from the input `randomness`, and then its contents are written to the 16 bit array `seedSE` that serves as input to `shake_gen_S`. This intermediate step is unnecessary, instead, the input from `randomness` can be directly written to `seedSE`, resulting in one less BRAM needed and a few clock cycles saved.
Furthermore, the array `seedSE` is currently implemented using a BRAM. However, this array has only 8 elements, which makes using an entire BRAM for it a waste of resources. Instead of implementing it using a BRAM, it can be implemented as a distributed RAM (LUTRAM) using the `RESOURCE` *pragma* with the option `core=RAM_2P_LUTRAM`. This change reduces the BRAM usage by 1 while the FF usage is slightly increased. The latency remains unchanged.
Sampling from a discrete Gaussian distribution, done by the function `frodo_sample_n`, has further potential for optimizations. The function `shake_gen_S` generates a vector of pseudorandom values that serve as input to `frodo_sample_n`. A more efficient way of sampling would be to sample in `shake_gen_S` as soon as one pseudorandom value is ready. In addition, the optimized sampler from Section 3.4 can used. These changes reduce the hardware usage by around 200 FFs and LUTs while also decreasing the latency by almost 500 clock cycles.
After the loop `Add_E` has been completed, the resulting matrix $\mathbf{B}$ is packed using the function `frodo_pack_16`. The design can be sped up by starting the packing earlier, as soon as the first row of $\mathbf{B}$ is fully calculated, by executing `frodo_pack_16` in parallel to `add_E`. This can be achieved by using a ping-pong buffer, very similar to the ones described before. As the two arrays `A_1` and `A_2` from the matrix multiplication are no longer in use, they can now be used to implement the ping-pong buffer. The optimized code for the loop `Add_E` is shown in Listing 4.6.

```
1  Add_E: for(i = 0; i < PARAMS_NBAR+1; i++)
2  {
3      ...
4      if((i & 0x1) == 0)
5      {
6          add_E(A_2, B+i*PARAMS_N, S_2, begin_add);
7          gen_S_sample_write(S_1, seed_SE, 160, 0, sk_S+i*PARAMS_N, 0);
8          frodo_pack_16(pk_b+600*(i-1), A_1, PARAMS_N, begin_pack);
9      }
10     else
11     {
12         add_E(A_1, B+i*PARAMS_N, S_1, begin_add);
13         gen_S_sample_write(S_2, seed_SE, 160, 0, sk_S+i*PARAMS_N, 0);
14         frodo_pack_16(pk_b+600*(i-1), A_2, PARAMS_N, begin_pack);
15     }
16 }
```

Listing 4.6: Loop `Add_E` that generates the error matrix E and adds it to the result of the matrix multiplication.

In the first iteration, the first row of $\mathbf{B}$ is calculated and stored in `A_2` with the function `add_E`, but the packing does not start yet. Starting from the second iteration, addition,

generation of **E**, and packing can be executed in parallel. In the last iteration, no new row of **B** has to be calculated, and the packing of the last row is finished.
Using the ping-pong buffer increases the hardware usage by around 100 FFs and 250 LUTs, but decreases the latency by 8.000 clock cycles. As `A_1` and `A_2` can be reused, the BRAM usage does not change.
Public key and secret key both contain the seed `seedA` and the packed version of the matrix `B`. In the current version of the code, the seed and the matrix are first written into the public key, and from there copied into the secret key, which takes almost 5.000 clock cycles. When writing the packed matrix directly into both public and secret key, the copying is no longer necessary. This can be achieved by slightly modifying the function `frodo_pack_16` by not only writing the public key, but also the secret key in the function. This slightly increases the FF usage, while the latency is decreased by almost 5.000 clock cycles.
Regarding the generation of the matrix **A** in the matrix multiplication (Listing 4.3), there is still some room for improvement. Two things can be observed:

- Before the vector-vector multiplications can start, the first row of **A** has to be pre-generated. This happens in every call to the function `vector_matrix_mul`.
- In the last loop iteration of that function, another row of **A** is generated (the 641st row), but never used.

These observations lead to another possible optimization: The first row of **A** only has to be pre-generated when the function `vector_matrix_mul` is called for the first time. When the loop in the function reaches the last iteration, instead of generating an unneeded next row, the first row can be generated again. That way, it is ready when the function is called the next time, and the pre-generation is no longer necessary. The code for the optimized pre-generation of **A** is shown in Listing 4.7. `pregen` is an additional parameter for the function that is only set to 1 for the first call to it. In lines 8 and 9, the seed is set to generate the first row when the last iteration is reached, else the seed is set to generate the next row. Applying this change slightly increases the LUT usage while saving 4.000 clock cycles.
One final issue to solve has to do with the function `shake128`. It is called twice, once at the start of the key generation to calculate `seed_A`, and once at the end to calculate `pkh`. Ideally, the same instance of `shake128` should be used for both calls as they are independent of each other and need not run in parallel. However, the HLS tool creates two different instances of the function because the input and output arrays have different sizes. Regarding the output array, `seed_A` can be used in both instances: In the first function call, `seed_A` is already the desired output array, and it can be used for the second call as well since its previous contents are no longer needed. From `seed_A`, the generated data can be written to the secret key in a short final loop.
Regarding the input arrays, slightly more work needs to be done. On the first function call, the input is a part of `randomness`, whereas on the second call, the input is the parameter `pk`. To circumvent this problem, a local array `X` can be used that serves as input in both cases. A first approach could be to make `X` large enough that it can hold

```
if(pregen)
{
    seed_A[0] = 0;
    shake128_10240(A_2, seed_A);
}
for(i = 0; i < PARAMS_N; i++)
{
    if(i == PARAMS_N-1)
        seed_A[0] = 0;
    else
        seed_A[0] = i+1;
    ...
}
```

Listing 4.7: Optimized pregeneration of the first row of **A** in the matrix multiplication of the key generation.

the entire public key, copy the public key into `X` and then hash it in one go. However, `X` would be very large and require multiple BRAMs to implement it. Instead, `X` could be implemented using only one BRAM by hashing the public key in several steps. To make this work, `X` is passed to `frodo_pack_16`, and the packed row of **B** is written into it. Afterwards, this packed row can be hashed. Of course, this requires a modified version of `shake128` that allows to end and resume the hashing in the middle of a block. This modified instance `absorb_block` could also be used to compute the first hash in the beginning.
To further develop this idea, `X` can be implemented as a ping-pong buffer using two arrays `X_1` and `X_2`. That way, `absorb_block` can run in parallel to the other functions in the loop `Add_E`, which significantly reduces the latency. Listing 4.8 shows the modified loop with the added function `absorb_block`. This solution requires two additional BRAMs for the arrays `X_1` and `X_2`, but saves over 10.000 LUTs and 10.000 clock cycles.
One final optimization targets the storage of the matrix **B**. It is currently implemented using 8 BRAMs, but it is also possible to implement it using a distributed RAM. This would reduce the BRAM usage by 8, while needing about 1.300 additional LUTs. As there is no difference in latency or maximum frequency between the two solutions, none of them is necessarily to be favoured, but for this work, the implementation using a distributed RAM is chosen to meet the total BRAM usage by the direct implementation. After applying all the described changes, the HLS tool estimates a hardware usage of 6.693 FFs, 33.218 LUTs, 6 BRAMs and 1 DSP. The total latency is 3.336.364 clock cycles with a maximum frequency of 167 MHz.

4.2.2 Encapsulation

The algorithm for the encapsulation is shown in Algorithm 2.4.2. Many functionalities are shared between key generation and encapsulation, such as the generation of **A**, **S** and **E**, sampling and packing. Their optimization is therefore only briefly mentioned, and only new optimizations are described in more detail.

```
1  Add_E: for(i = 0; i < PARAMS_NBAR+1; i++)
2  {
3      ...
4      if((i & 0x1) == 0)
5      {
6          add_E(A_2, B+i*PARAMS_N, S_2, begin_add);
7          gen_S_sample_write(S_1, seed_SE, 160, 0, sk_S+i*PARAMS_N, 0);
8          frodo_pack_16(pk_b+600*(i-1), sk_pk+sk_offset, X_1, A_1, PARAMS_N, begin_pack);
9          absorb_block(seed_A, X_2, 0, 150, 0, begin_absorb);
10     }
11     else
12     {
13         add_E(A_1, B+i*PARAMS_N, S_1, begin_add);
14         gen_S_sample_write(S_2, seed_SE, 160, 0, sk_S+i*PARAMS_N, 0);
15         frodo_pack_16(pk_b+600*(i-1), sk_pk+sk_offset, X_2, A_2, PARAMS_N, begin_pack);
16         absorb_block(seed_A, X_1, 0, 150, 0, begin_absorb);
17     }
18 }
```

Listing 4.8: Optimized loop `Add_E` that performs the addition of the error terms, packs and then hashes the results.

Naive HLS and First Optimizations

The software implementation of the encapsulation algorithm requires some changes before the HLS can be done, very similar to those needed for the key generation. For the arrays passed to the function, the `INTERFACE` *pragma* has to be added, and any appearance of `memcpy` has to be replaced by a loop that copies data from one array to the other. In order to avoid pointer reinterpretations from `uint8_t` to `uint16_t`, additional intermediate arrays are required. Once these changes have been applied, the code can be synthesized. The HLS tool estimates a hardware usage of 22.456 FFs, 68.619 LUTs, 1.132 BRAMs and 2 DSPs with a total latency of 17.308.775 clock cycles and a maximum frequency of 114,29 MHz.
The first optimization steps are identical to those of the key generation: To generate the rows of $\mathbf{A}$, the function `shake128_10240` can be used, and the two functions for matrix multiplication can be inlined. In addition, the zero-initialization of the arrays is unnecessary and can be omitted. These changes alone reduce the hardware usage by 3.000 FFs, 1.500 LUTs and over 500 BRAMs while also decreasing the latency by over 7.000.000 clock cycles.
At the start of the encapsulation, the public key is hashed using `shake128`, and the result, together with the input `mu`, is hashed again to obtain `seedSE` and `k`. Due to the different sizes of the inputs, two instances of `shake128` are generated by the HLS tool. In order to only have one instance of `shake128` created, it makes sense to load the public key into a local array `shake_input` that serves as input to the hash function whenever it is called. Due to the size of the public key, it is more favourable to load it into the local array in multiple smaller blocks and to hash these blocks. The function `absorb_block`, which is already used in the key generation, can be used in the encapsulation as well. After the public key has been hashed this way, `pkh`$||\mu$ can be written into the local

array and then be hashed as well. To improve the latency of hashing the public key, `shake_input` can be implemented as a ping-pong buffer using two arrays `shake_input_1` and `shake_input_2`. Listing 4.9 shows the loop that reads the public key into the local arrays and then hashes it. To read the public key in parallel to the hashing, a new function `read_input` is used. In the first iteration, `begin_absorb_block` is 0 as hashing cannot begin yet, and `read_input` reads the first 640 elements from the public key into `shake_input_2`. Starting from the second iteration, `absorb_block` hashes the content of one array, while `read_input` loads the next block of the public key into the other array. Hashing this way reduces the hardware usage by about 400 FFs and 11.000 LUTs while requiring 2 additional BRAMs that can be used again later in the encapsulation. The latency is reduced by almost 20.000 clock cycles.

```
1  Hash_pk: for(i = 0; i < 8; i++)
2  {
3      ...
4      if((i & 0x1) == 0)
5      {
6          absorb_block(pkh, shake_input_1, 0, 160, 0, begin_absorb_block);
7          read_input(shake_input_2, pk+640*i, read_input_length, begin_read_input);
8      }
9      else
10     {
11         absorb_block(pkh, shake_input_2, 0, 160, 0, begin_absorb_block);
12         read_input(shake_input_1, pk+640*i, read_input_length, begin_read_input);
13     }
14 }
```

Listing 4.9: Hashing of the public key at the start of the encapsulation.

First Matrix Multiplication

In the first matrix multiplication, the product of the matrices $\mathbf{S}'$ and $\mathbf{A}$ is added to $\mathbf{E}'$ and then written into $\mathbf{B}$. By contrast to the matrix multiplication in the key generation, $\mathbf{A}$ is now on the right side instead of the left. As a consequence, when performing vector-vector multiplications like before, the matrix $\mathbf{A}$ is needed column-wise, while it is still being generated row-wise. Since generating $\mathbf{A}$ column-wise is impossible, it is necessary to implement the matrix multiplication in a different way that still uses $\mathbf{A}$ row-wise. To achieve this, a scalar-vector multiplication can be used that multiplies a scalar from the left matrix with a vector on the right. The first element from $\mathbf{S}'$ is multiplied with every element from the first row of $\mathbf{A}$, and the products are added to the first row of $\mathbf{B}$. Then, the second element from $\mathbf{S}$ is multiplied with the second row of $\mathbf{A}$, and the results are again added to the first row of $\mathbf{B}$. To maintain correctness, this solution requires the rows of $\mathbf{B}$ to be set to zero at the start before any additions happen. The function `scalar_vector_mul` is shown in Listing 4.10. To execute the addition of the error term in the function as well, the currently needed row of $\mathbf{E}$ is passed and added in the first call to the function, indicated by the parameter `index`. In the remaining calls, only zeroes are added. As the function is supposed to be used in the second, smaller

```
1  void scalar_vector_mul(uint16_t *output_vec, uint16_t *input_vec, uint16_t scalar, uint16_t n,
       uint16_t index, uint16_t *E)
2  {
3      #pragma HLS INLINE off
4      uint16_t i, temp, e;
5      L1: for(i = 0; i < n; i++)
6      {
7          #pragma HLS PIPELINE
8          if(index == 0)
9              e = E[i];
10         else
11             e = 0;
12         output_vec[i] += scalar*input_vec[i] + e;
13     }
14 }
```

Listing 4.10: Function `scalar_vector_mul`.

matrix multiplication as well, the parameter `n` is added that contains the number of columns of the right matrix (640 for $\mathbf{A}$, 8 in the second matrix multiplication).
With the help of this function and `shake128_10240`, $\mathbf{A}$ can be implemented as a ping-pong buffer, just like in the key generation. This saves over 500 BRAMs, and with the pipeline in `scalar_vector_mul`, the latency can be reduced by almost 7.000.000 clock cycles.
The matrices $\mathbf{S}'$ and $\mathbf{E}'$ need not be stored entirely, but can be implemented using a ping-pong buffer in a way similar to the key generation with the arrays `S_1` and `S_2` for $\mathbf{S}'$ and `E_1` and `E_2` for $\mathbf{E}'$. To generate a row of $\mathbf{S}'$ and $\mathbf{E}'$ in one go, it is necessary to modify the function `shake_gen_S` that was used for this task in the key generation. The modified function `shake_gen_S_E` generates a series of pseudorandom data, but only writes a certain interval of it to $\mathbf{S}'$ or $\mathbf{E}'$. To execute `shake_gen_S_E` in parallel to the scalar-vector multiplications, these have to be moved inside a function `vector_matrix_mul` that works very similar to the one used in the key generation. It multiplies one row of $\mathbf{S}'$ with the entire matrix $\mathbf{A}$ and adds one row of $\mathbf{E}'$ to the result. Using these two functions, the hardware usage can be reduced by 12 BRAMs and the latency by 30.000 clock cycles.
The matrix $\mathbf{B}'$ is packed once the entire matrix multiplication has finished. This requires storing $\mathbf{B}'$ entirely which uses 8 BRAMs. A more efficient way of packing $\mathbf{B}'$, both in terms of BRAM usage and latency, is to start the packing as soon as the first row has been calculated. When implementing $\mathbf{B}'$ with two arrays `B_1` and `B_2`, the packing function `frodo_pack_16` can be executed in parallel to `vector_matrix_mul` and `shake_gen_S_E`. That way, the latency of the packing can completely be ignored, and the number of BRAMs needed to store $\mathbf{B}'$ is reduced to 2.
After packing, the results are written to the beginning of the ciphertext which is hashed at the end of the encapsulation using `shake128`. To reduce the hardware usage, it is desirable to replace any instance of `shake128` with the function `absorb_block` that is already used for the hashing at the start of the encapsulation. As this function is able to hash an input block-wise, it makes sense to combine the hashing with the packing.

```
1  Mat_Mul1: for(i = 0; i < PARAMS_NBAR+2; i++)
2  {
3      ...
4      if((i & 0x1) == 0)
5      {
6          pack_reset(shake_input_2, B_1, PARAMS_N, begin_pack, PARAMS_N);
7          shake_gen_S_E(S_1, E_1, outlen, seed_SE, start1, end1, start2, end2, begin_shake_gen);
8          vector_matrix_mul(B_2, S_2, A_1, A_2, E_2, seed_A_separated, pk_b, pregen, PARAMS_N, 1,
               begin_mat_mul);
9          absorb_block_write(ss_prime, shake_input_1, 0, 150, reset, begin_absorb_block, ct+600*(i-2)
               , 600, begin_read_input);
10     }
11     else
12     {
13         pack_reset(shake_input_1, B_2, PARAMS_N, begin_pack, PARAMS_N);
14         shake_gen_S_E(S_2, E_2, outlen, seed_SE, start1, end1, start2, end2, begin_shake_gen);
15         vector_matrix_mul(B_1, S_1, A_1, A_2, E_1, seed_A_separated, pk_b, pregen, PARAMS_N, 1,
               begin_mat_mul);
16         absorb_block_write(ss_prime, shake_input_2, 0, 150, reset, begin_absorb_block, ct+600*(i-2)
               , 600, begin_read_input);
17     }
18 }
```

Listing 4.11: First matrix multiplication of the encapsulation. The results of the multiplication are packed and then hashed.

Instead of writing the results from `frodo_pack_16` into `ct` and then hashing them with `absorb_block` later, the more efficient way is to immediately hash the packed rows of $\mathbf{B}'$ once they are available. To achieve this, instead of packing into `ct`, `frodo_pack_16` writes its results to the two arrays `shake_input_1` and `shake_input_2`, from where they are hashed. Furthermore, the packed rows are written to the ciphertext from the two arrays of `shake_input`.
Listing 4.11 shows the loop that implements the described behaviour. All the functions used have a parameter `begin` that decides whether a function executes its code or not. The function `pack_reset` is a combination of two sub-functions, `frodo_pack_16` and `reset_vector`. The former packs the data from `B_1` or `B_2`, while the latter sets all elements in the array to 0, which is necessary for the matrix multiplication. Wrapping these two functions into one function is done to execute them in parallel to `vector_matrix_mul`. They are accessing the same array, and without wrapping them in one function, `reset_vector` would only be executed after `vector_matrix_mul` has been completed. For the same reason, the functions `absorb_block` and `write_ct` are wrapped into the function `absorb_block_write`. While the former hashes the packed row of $\mathbf{B}'$, the latter writes it into the ciphertext.

Second Matrix Multiplication

In the second matrix multiplication, the matrix $\mathbf{S}'$ on the left is multiplied with $\mathbf{B}$ on the right, and the result is added to $\mathbf{E}''$. $\mathbf{S}'$ and $\mathbf{E}''$ can be generated using the function `shake_gen_S_E`, with the data for $\mathbf{E}''$ coming after the data of $\mathbf{E}'$, whereas $\mathbf{B}$ is unpacked from the public key using the function `frodo_unpack`. This function can

```
...
L1: for(k = 0; k < PARAMS_N; k++)
{
    ...
    if((k & 0x1) == 0)
    {
        if(use_A)
        {
            shake128_10240_1(A_1, seed_A_separated);
            scalar_vector_mul(output_vec, A_2, s, n, k, E);
        }
        else
        {
            frodo_unpack_16(A_1, pk+offset+8, 8, 8);
            scalar_vector_mul(output_vec, A_2, s, n, k, E);
            offset += 15;
        }
    }
    ...
}
```

Listing 4.12: Modified function `vector_matrix_mul` that executes both matrix multiplications of the encapsulation (Excerpt).

be optimized to work on 16 bit values in order to decrease its latency. Similar to the first matrix multiplication, the matrices need not be generated and stored entirely before the multiplication can start. Instead, enough of the matrices is pregenerated so as to be able to start the multiplication, and the rest of the generation happens in parallel to the multiplications. To maximize the efficiency of the design, it is desirable to reuse and modify the function `vector_matrix_mul` from the first multiplication. That way, only one DSP is needed for the encapsulation.
Listing 4.12 shows an excerpt of the modified function `vector_matrix_mul`. The parameter `use_A` was added to the function to decide whether the matrix $\mathbf{A}$ or $\mathbf{B}$ is multiplied on the right side. In the case where `use_A` is 1, $\mathbf{A}$ is generated into the arrays `A_1` and `A_2` as described for the first matrix multiplication. In the other case, $\mathbf{B}$ is generated into `A_1` and `A_2` using the function `frodo_unpack_16`. Independent of whether $\mathbf{A}$ or $\mathbf{B}$ is used, the generation of the matrix is executed in parallel to the scalar-vector multiplication and therefore has no influence on the total latency.

Frequency

The design is currently running with a maximum frequency of only 114,29 MHz, with the critical path being the multiplication and additions in the function `scalar_vector_mul`. To increase the frequency, it is necessary to manually insert a register in the critical path after the multiplication and the addition with `e` has been executed, but before the result is added and written back to `output_vec[i]`. The HLS tool does not offer a pragma that forces it to insert a register at a certain point, instead, the register has to be added in the `C`-code. The code with added registers is shown in Listing 4.13. The new function `reg` implements the register by simply returning its input. To prevent the HLS tool from

```
1  uint16_t reg(uint16_t in)
2  {
3      #pragma HLS PIPELINE II=1
4      #pragma HLS INTERFACE ap_ctrl_none register port=return
5      #pragma HLS INLINE off
6      return in;
7  }
8  void scalar_vector_mul(uint16_t *output_vec, uint16_t *input_vec, uint16_t scalar, uint16_t n,
       uint16_t index, uint16_t *E)
9  {
10     ...
11     temp = scalar*input_vec[i] + e;
12     temp = reg(temp);
13     output_vec[i] += temp;
14 }
```

Listing 4.13: Function `scalar_vector_mul` with added registers to increase the maximum frequency.

optimizing the function away, the `INLINE` pragma is added with the option `off`, and the `PIPELINE` pragma allows the function to be executed every clock cycle. With this change, the maximum frequency can be increased to 156,54 MHz. To further increase the frequency, the parameters `mu_in`, `pk`, `ct` and `ss` can be passed by value instead of passing a pointer to the arrays. This allows to increase the frequency to 167 MHz, at the cost of higher hardware usage, especially FFs, and a latency that is increased by about 20.000 clock cycles. The additional clock cycles mainly result from the function `scalar_vector_mul` where the tool automatically inserts an additional register in the critical path.

Further Optimizations

After optimizing the two matrix multiplications and increasing the frequency, most of the optimization potential for hardware usage and latency has been used. The remaining optimizations can reduce the latency by a few thousand clock cycles and save a few hundred FFs and LUTs.
The generation of $\mathbf{B}$ in `vector_matrix_mul` is currently done using the function `frodo_unpack_16` which can unpack data of any given length. In this case, however, a row of $\mathbf{B}$ with 8 elements has to be unpacked. Therefore, it is possible to use a more specialized instance `frodo_unpack_8` which unpacks a fixed amount of 8 elements. This change saves about 200 FFs and LUTs.
For the two matrix multiplications, two loops `Mat_Mul1` and `Mat_Mul2` are used, the second one only starting once the first matrix multiplication, including the processing of the data with the function `absorb_block_write`, has been completed. While the last row of $\mathbf{B}'$ is processed, the vector-multiplication is idle. By combining the two matrix multiplication loops into one, the second multiplication can start while the last row of the first multiplication is processed, reducing the latency by around 3.000 clock cycles. The function `vector_matrix_mul` is never idle that way.

After the second matrix multiplication, the result $\mathbf{V}$ is added to the encoding of `mu` using the functions `frodo_key_encode` and `frodo_add`. Since these two functions do not appear anywhere else in the code, they can be inlined. In addition, they can be combined into one loop `Encode_Add`, reducing the hardware usage by around 50 FFs and 300 LUTs as well as reducing the latency by 100 clock cycles.
The final hash of the ciphertext, with the hashing of c_1 starting during the first matrix multiplication, can be done with the function `absorb_block` by packing the result of the second matrix multiplication, $\mathbf{V}$, into `shake_input_1`, writing the random value k behind it and then calling the hash function. This eliminates the last instance of `shake128`, reducing the hardware usage by another 2.000 FFs and 10.000 LUTs.
With all the described changes applied, the HLS tool expects a hardware usage of 7.694 FFs, 33.926 LUTs, 10 BRAMs and 1 DSP. The total latency is 3.412.950 clock cycles with a maximum frequency of 156,54 MHz. Following synthesis and implementation, the synthesis tool reports a hardware usage of 6.817 FFs and 14.169 LUTs with the other values remaining unchanged.

4.2.3 Decapsulation

Most of the functions used and optimized in the encapsulation can also be used for the decapsulation algorithm. One major change is the additional third matrix multiplication that requires modification of the function `vector_matrix_mul`.
Before the HLS can be performed on the software implementation, a few changes regarding pointer reinterpretation and the function `memcpy` are necessary, as already described for the key generation and the encapsulation. After synthesis, the tool expects a hardware usage of 26.277 FFs, 61.640 LUTs, 1.147 BRAMs and 3 DSPs. The decapsulation algorithm takes 17.682.782 clock cycles to complete with a maximum frequency of 109,63 MHz. The first optimization steps from the encapsulation – replacing instances of `shake128` with `absorb_block` and hashing block-wise, generating the rows of $\mathbf{A}$ with `shake128_10240`, and sampling $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$ with `shake_gen_S_E` – can be applied here as well. These changes alone significantly reduce the hardware usage, especially regarding LUTs and BRAMs, and the total latency.

Matrix Multiplications

The main objective for the matrix multiplications is to modify the function `vector_matrix_mul` in such a way that it can be used for all 3 multiplications.
The first matrix multiplication calculates $\mathbf{C} - \mathbf{B}' \cdot \mathbf{S}$, where $\mathbf{B}'$ and $\mathbf{C}$ are unpacked from the ciphertext and $\mathbf{S}$ is a part of the secret key. $\mathbf{C}$ is very small and can be stored entirely after unpacking, while the larger matrices $\mathbf{B}'$ and $\mathbf{S}$ are stored row- or column-wise. To execute this multiplication with the same structure as the subsequent ones, $\mathbf{S}$ as the matrix on the right side has to be written into the arrays `A_1` and `A_2` inside the function `vector_matrix_mul`. In order to do this in parallel to the scalar-vector multiplication, it is necessary to read $\mathbf{S}$ using a function `read_S`. As no error terms have to be added, the parameter normally containing the error terms needs to be a zeroed array to ensure the

correctness.
The second matrix multiplication is identical to the first multiplication of the encapsulation, only that packing the resulting matrix is no longer necessary. Instead, the result $\mathbf{B}''$ is compared with the matrix $\mathbf{B}'$ which was used in the first matrix multiplication. As $\mathbf{B}'$ has not been stored, it has to be unpacked again from the ciphertext. This can happen in parallel to `vector_matrix_mul` and `shake_gen_S_E`. Once the first row of $\mathbf{B}''$ has been calculated and the first row of $\mathbf{B}'$ has been unpacked, they can be compared using a new function `compare`. This function loops over every element of the two input arrays and returns 1 if a difference occurs. Once the comparison is done, the two rows are no longer needed and can be overwritten.
The last matrix multiplication can happen in the same way as the second matrix multiplication of the encapsulation. In parallel to the multiplication, $c_1||c_2$ can be hashed by loading them block-wise into the two arrays of `shake_input`, and then hashing the blocks using `absorb_block`. Once the matrix multiplication has been completed, one last comparison between the result $\mathbf{C}'$ and $\mathbf{C}$ is necessary. $\mathbf{C}$ has to be unpacked from c_2 again. Depending on the outcome of the comparison, either k' or s must be hashed using `absorb_block`.

Frequency

The maximum frequency of the design can be increased by adding a register in `scalar_vector_mul` as described for the encapsulation, and by passing the arrays by value instead of passing them by reference. However, due to the increased BRAM usage of the design, the frequency can only reach a maximum of 142,57 MHz instead of 167.
With all the changes applied, the HLS tool expects a hardware usage of 7.142 FFs, 33.706 LUTs, 12 BRAMs and 1 DSP. The total latency is 3.476.753 clock cycles with a maximum frequency of 142,57 MHz.

4.3 Comparison of the Implementations

In general, the implementation from [HOKG18], the direct hardware implementation made for this work, and the HLS version perform very similar in terms of hardware usage, latency and frequency. However, several minor differences exist that are briefly discussed in this section.
The synthesis results for the key generation are shown in Table 4.1. In the HLS version, about 800 FFs less are needed than in the direct implementations, but it is difficult to state where exactly they are saved. On the other hand, the HLS version uses almost 3.500 additional LUTs. About 2.500 of them are used to store the matrix $\mathbf{B}$ before it is packed. This is not done in the direct implementations; however, it seems to be impossible to start packing earlier in the HLS version due to the order in which the entries of $\mathbf{B}$ are calculated: They are calculated column-wise, but the packing is done row-wise. Therefore, packing can only start once the full matrix has been calculated. The BRAM and DSP usage as well as the maximum frequency of all three implementations are identical. The latency of the HLS version is about 50.000 clock cycles higher. This

is mainly due to the use of functions, especially `vector_vector_mul` which takes 644 clock cycles to complete the 640 multiplications. As a consequence, with each call to this function, 4 clock cycles are lost compared to the direct implementation. More than 20.000 clock cycles are attributable to the 5.120 calls to the function alone. However, the additional latency is less than 2 % of the total latency.

Implementation	FF	LUT	BRAM (18K)	DSP	Latency (Cycles)	Frequency (MHz)
Naive HLS	18.634	43.522	1.073	1	17.332.462	114,29
Optimized HLS	6.112	15.135	6	1	3.336.364	167,00
[HOKG18]	6.855	12.003	6	1	3.276.800	167,00
Direct Implementation (this work)	6.719	12.687	6	1	3.291.149	167,00

Table 4.1: Synthesis results for the HLS- and direct implementations of the FrodoKEM key generation.

Table 4.2 displays the synthesis results for the encapsulation. The HLS version beats both direct implementations in terms of FF usage, and ranges in between the two implementations regarding the LUT usage. Just like for the key generation, it is difficult to state at what point exactly the hardware in the HLS version is saved or lost. The HLS version and the direct implementation in this work use 1 BRAM less than the implementation by Howe et al. All three implementations use only one DSP, and have the same frequency of 167 MHz. Mainly due to the fact that the encapsulation uses two matrix multiplications instead of one, the HLS implementation needs over 100.000 additional clock cycles, whereas the two direct implementations have a very similar latency. The function `scalar_vector_mul` uses 5 additional clock cycles, independent of the parameter `n`. As it is called over 10.000 times, this function alone is responsible for more than 50.000 of the additional clock cycles compared to the direct implementations. However, even though the difference has grown, the additional latency is just over 3 % of the total latency, a still rather small percentage.

Implementation	FF	LUT	BRAM (18K)	DSP	Latency (Cycles)	Frequency (MHz)
Naive HLS	22.456	68.619	1.132	2	17.308.775	114,29
Optimized HLS	6.891	13.939	10	1	3.423.522	167,00
[HOKG18]	8.583	14.977	11	1	3.317.760	167,00
Direct Implementation (this work)	7.024	11.396	10	1	3.328.907	167,00

Table 4.2: Synthesis results for the HLS- and direct implementations of the FrodoKEM encapsulation.

For the decapsulation, the performance results shown in Table 4.3 are very similar again. The HLS version surpasses both direct implementations in terms of FF usage and ranks

in between regarding the LUT usage. It uses 4 BRAMs less than the implementation of Howe et al., but 2 more than the implementation made for this work. All three implementations only use one DSP. The latency of the HLS version exceeds that of the direct implementations by about 120.000 clock cycles, an increase of almost 20.000 clock cycles over the encapsulation. This increase is mainly due to the third matrix multiplication and the additional clock cycles caused by the function `scalar_vector_mul`. Unlike for the key generation and the encapsulation, the HLS version is unable to match the frequency of the direct implementations. Even with added registers, the frequency of the HLS version only reaches 142,57 MHz, almost 20 less than the direct implementations with 162. The tendency towards a lower frequency of the decapsulation is already visible in the very first naive HLS versions, where key generation and encapsulation reached a frequency of 114,29 MHz, whereas the decapsulation only reached a maximum frequency of 109,63 MHz.

Implementation	**FF**	**LUT**	**BRAM (18K)**	**DSP**	**Latency (Cycles)**	**Frequency (MHz)**
Naive HLS	26.277	61.640	1.147	3	17.682.782	109,63
Optimized HLS	6.571	13.991	12	1	3.476.753	142,57
[HOKG18]	8.604	15.452	16	1	3.358.720	162,00
Direct Implementation (this work)	7.451	11.792	10	1	3.366.126	162,00

Table 4.3: Synthesis results for the HLS- and direct implementations of the FrodoKEM decapsulation.

5 Conclusion

This chapter first gives an overview of the results of this work. Then, general strategies to achieve good results with HLS on Post-Quantum Cryptography are identified and discussed. Finally, a short outlook on possible future work on this topic is given.

5.1 Results of this Work

In this work, the possibilities of using HLS to obtain hardware implementations of PQC instead of implementing directly in hardware were analysed. For the four analysed crucial components - NTT, polynomial multiplication without using the NTT, the SHAKE hash function and a discrete Gaussian sampler - the synthesis results of the optimized HLS implementations are very close to those of the direct hardware implementations, both in terms of resource utilization and latency. Similar results could be found for the FrodoKEM encapsulation scheme where the hardware usage of the HLS version is generally lower than that of the direct implementation, whereas the latency of the HLS version is slightly higher.
Apart from the performance of the implementation, its development time and flexibility play an important role. In this aspect, HLS truly begins to shine. The development time for the optimized HLS implementation of FrodoKEM was significantly lower than that of the direct implementation. The two main reasons for the lower development time are the very quick execution of the C-code, allowing to verify the correctness of changes in the design quickly, and the relatively small amount of code that has to be written. Especially when a software implementation already exists, which is the case for all of the cryptographic schemes submitted to the NIST competition, the amount of code to be written is rather small, and its correctness should be given, further reducing the development time. In addition to the shorter development time, HLS implementations are also more flexible regarding later changes. This is particularly useful for the rather new field of PQC where new attacks on schemes are quite likely, possibly resulting in replaced algorithms or changed parameter sets. Such changes are fairly easy to apply to the C-code by replacing a function or changing the data type of certain variables.
The price a user has to pay for the reduced development time and increased flexibility is a higher latency due to the use of functions for more complex designs like the FrodoKEM encapsulation scheme, and generally less control of the frequency and the translation of the design into hardware. While the user can inform the HLS tool about the desired frequency, it is up to the tool to generate the VHDL code in a way that it can match the frequency requirement, and the user has only limited options to influence the frequency manually. Another drawback results from the fact that DSPs cannot be shared between functions, which can lead to an increased DSP usage for complex designs with multiple

functions executing multiplications.
Despite these drawbacks, the overall performance results of the HLS versions in this work are very convincing and indicate that HLS can have multiple use cases for the hardware implementation of Post-Quantum Cryptography schemes. A first possible use case is to obtain initial benchmarks for a cryptographic scheme before beginning to implement it in hardware. With a few changes to the original `C`-code and some added *pragmas*, the hardware usage and latency of the design can be significantly improved and should allow an estimation on the possible performance results of the finished hardware implementation. Furthermore, the functionality of design ideas and their effect on hardware usage and latency can rather quickly and easily be tried out using HLS, for example the use of ping-pong buffers to reduce the storage requirements. If the HLS results indicate that the idea has a positive effect, it can then be implemented in hardware without the risk of applying changes with no benefit.
Apart from benchmarking for a direct implementation, HLS can also be used to create the entire hardware implementation, as was done for FrodoKEM in this work. Synthesizing complex designs does work well in the case of FrodoKEM, albeit a small penalty on the latency, but for other schemes, especially when many DSPs are used, the performance of the HLS version might be worse than a direct implementation.
Finally, a combination of direct implementation and HLS might work well. HLS can be used to quickly implement components that are not critical for the latency or the hardware usage, whereas time-critical components are implemented directly in hardware to achieve the best possible latency. In the case of FrodoKEM, the components that might be implemented using HLS are packing and unpacking, encoding and decoding, but also the SHAKE modules since they run in parallel to the multiplications. The only component that should be implemented directly in hardware is be the vector-matrix multiplication to ensure the continuous use of the DSP. Such an implementation would combine the best of two worlds: The implementation process would be accelerated due to the extended use of HLS, and at the same time would perform like the direct implementation.

5.2 General Strategies for HLS

Developing a single strategy that achieves the best possible results for the many different kinds of `C`-code and optimization goals is probably impossible. Instead, a few general techniques and optimizations that help to significantly improve the performance of the design are discussed in this section.
A first important guideline towards optimum results is to combine *pragmas* and code changes together in the optimization process. If only code changes are used, the design will most likely struggle with the latency since loops are not pipelined. On the other hand, when using only *pragmas*, the design will probably have a significantly higher hardware usage because the storage and data types can hardly be optimized. Combining both allows for the best possible optimizations. The `PIPELINE` *pragma* is certainly the most important one as it allows to reduce the latency of a loop to a fraction of its initial

value while barely increasing the hardware usage. Applying the `LOOP_UNROLL` *pragma*, albeit very powerful, only makes sense if the structure of the loop allows it. This is true especially for loops that access arrays, as arrays usually have a maximum of two read and write ports. As a consequence, an unrolling factor of more than 2 can often not be achieved, unless arrays are partitioned using the `ARRAY_PARTITION` *pragma*. This allows for a speed-up due to the unrolling, but also requires additional BRAMs for the storage. Many of the other *pragmas* can be applied under the right circumstances. Whether a *pragma* has a positive effect or not will often have to be found out by trial-and-error.
Functions are the best way of achieving the parallel execution of more complex functionalities, especially loops. While simple statements like additions are automatically executed in parallel by the HLS tool if they are independent, this is not possible for loops. The solution is to wrap the loop and associated statements into a function since functions can be executed in parallel. The only requirements for this are that the functions are independent, and that they do not share any arrays. The use of a function slightly increases the hardware usage, but potentially reduces the latency by a considerable amount, especially if two long functions can be executed in parallel. When working with functions, special attention has to be paid to the size and type of arrays that are passed to the function. If two arrays of different size are passed to a function, the HLS tool will create two instances of this function. The same applies if one array is implemented as a single-port BRAM and the other as a dual-port BRAM. To circumvent this problem and only create one instance of the function, the contents of one of the arrays can be loaded into the other if the contents of that array can be overwritten. Another option is to change the size of one array until both sizes match. When passing an array with an offset to a function, it is advisable to pass the offset as an additional parameter instead of adding it to the array. This avoids the creation of two instances of the function if the array is once passed with an offset and once without.
Conditional statements are often avoided in software implementations of cryptography because they are not executed in constant time and might provoke simple timing attacks. The conditional statements are then replaced by multiple operations such as XORs and additions. In hardware, conditional statements can be executed in constant time within one clock cycle, and replacing the mathematical operations from the original `C`-code by equivalent conditional statements can reduce the latency and the hardware usage.
Multiplications with a factor that is not a power of two can be costly in hardware as they are usually implemented using a DSP. As a consequence, such multiplications should be avoided whenever possible. Sometimes, when one of the factors is rather small, the multiplication can be implemented using multiple conditional statements and additions, in some other cases the multiplication can completely be avoided. The same applies for divisions where the divisor is not a power of two. Modular reductions are very costly as well if the modulus is not a power of two, and should therefore be avoided. When the input values to the reduction are not far bigger than the modulus, the reduction can be implemented using conditional statements and subtractions. This was done for the NTT in Section 3.2. Multiplications, divisions and reductions with a power of two are no problem at all since they can be implemented using left or right shifts, or with an AND-operation.

The language `C` only allows the declaration of variables with a bit length that is a power of two, starting from 8 bit types up to 64 bit types. In many cases, only a part of the available bit length of a variable us used, resulting in a waste of resources when the design is synthesized. To avoid this, the `ap_cint.h` header allows to define variables of any desired length. Changing the data types of variables in the `C`-code to the smallest possible type can save a large amount of FFs and LUTs. However, the user has to be careful not to choose a data type that is too small. If the maximum value of a variable is not known or not bounded, it makes sense to keep the initial data type of a variable to avoid a malfunction in the code.
Another functionality provided by the `ap_cint.h` header are several bit manipulation functions. The function `apint_get_range` allows to get the value of a bit or a range of bits at a specified point in a variable, `apint_set_range` allows to set a bit or a range of bits to a specified value. Finally, `apint_concatenate` allows to concatenate two smaller variables into one larger variable. All of these can be done with masks, shifts and AND-operations, but the functions are slightly easier to use and less error-prone. The synthesis results of both approaches should not differ.
One final aspect to discuss is whether arrays should be passed by reference or by value. Both options are possible and will be synthesized into different designs. When pass-by-reference is used, the passed array is treated as a bus with one read and write port. By contrast, if pass-by-value is used, the passed array is treated as an external BRAM, and the user can decide whether it is a single-port or a dual-port BRAM. The second option allows to use two read and write ports, which can potentially result in a large speed-up of the design. Therefore, passing arrays by value seems to be the better choice. This decision is only relevant for the top-level function. If an array is passed to a sub-function, one of two cases can occur: If the passed array was defined in the calling function, it will be treated as a BRAM, no matter whether it is passed by value or by reference. On the other hand, if the passed array is one of the arrays passed to the top-level function, the way it is treated depends on how it was passed to the top-level function, not how it is passed to the sub-function. As a consequence, passing arrays to sub-functions can be done by reference or by value without any difference in the synthesized result.

5.3 Future Work

For future work, it would be interesting to further explore the possibility of using HLS to implement Post-Quantum Cryptography. This work only covers the full synthesis of one lattice-based encapsulation scheme, but the variety of lattice-based schemes is wide, and the quality of implementations might differ for other lattice-based schemes. Furthermore, applying HLS to other types of Post-Quantum Cryptography such as code-based or multivariate-quadratic cryptography could be rewarding since such schemes include components that have not been covered in this work. Bearing in mind that Vivado HLS is not the only existing HLS tool, comparing the synthesis results of different tools could allow for comparative studies pinpointing the shines and disadvantages of the respective tools.

A Acronyms

ASIC Application-Specific Integrated Circuit

BRAM Block RAM

BSI Federal Office for Information Security

CDF Cumulative Distribution Function

DSP Digital Signal Processor

FF Flip Flop

FPGA Field Programmable Gate Array

HDL Hardware Description Language

HLS High-Level Synthesis

HW/SW Hardware/Software

LUT Look-Up Table

LWE Learning With Errors

NIST National Institute of Standards and Technology

NTT Number-Theoretic Transform

PIC Programmable Interconnect

PKI Public-Key Infrastructure

PRNG Pseudo-Random Number Generator

PQC Post-Quantum Cryptography

RTL Register-Transfer Level

SVP Shortest Vector Problem

XOR Exclusive OR

B Appendix

```
void ntt(uint16_t *a, const uint16_t *omega)
{
    #pragma HLS INTERFACE ap_bus depth=1024 port=omega
    #pragma HLS INTERFACE ap_bus depth=1024 port=a
    int i, start, j, jTwiddle, distance;
    uint16_t temp, W;

    L1: for(i = 0; i < 10; i += 2)
    {
        // Even level
        distance = (1 << i);
        L1_1: for(start = 0; start < distance; start++)
        {
            jTwiddle = 0;
            L1_1_1: for(j = start; j < NEWHOPE_N-1; j += 2*distance)
            {
                W = omega[jTwiddle++];
                temp = a[j];
                a[j] = (temp + a[j + distance]); // Omit reduction (be lazy)
                a[j + distance] = montgomery_reduce((W * ((uint32_t)temp + 3*NEWHOPE_Q - a[j +
                    distance])));
            }
        }

        // Odd level
        distance <<= 1;
        L1_2: for(start = 0; start < distance;start++)
        {
            jTwiddle = 0;
            L1_2_1: for(j = start; j < NEWHOPE_N-1; j += 2*distance)
            {
                W = omega[jTwiddle++];
                temp = a[j];
                a[j] = (temp + a[j + distance]) % NEWHOPE_Q;
                a[j + distance] = montgomery_reduce((W * ((uint32_t)temp + 3*NEWHOPE_Q - a[j +
                    distance])));
            }
        }
    }
}
uint16_t montgomery_reduce(uint32_t a)
{
    #pragma HLS INLINE off
    #pragma HLS ALLOCATION instances=mul limit=1 operation
    uint32_t u, u1;

    u = (a * qinv);
    u &= ((1 << rlog) - 1);
    u *= NEWHOPE_Q;
    a = a + u;
    return a >> 18;
}
```

Listing B.1: Unoptimized software implementation of the NTT

```
void ntt(poly *a, const poly *omega)
{
    uint9 jTwiddle = 0;
    uint10 start = 0, distance = 1;
    uint14 W;
    uint15 temp, adistance;
    uint16_t i, j = 0, x;

    L1: for(i = 0; i < 5120; i++)
    {
        W = omega->coeffs[jTwiddle++];
        temp = a->coeffs[j];
        adistance = a->coeffs[j+distance];

        x = temp + adistance;
        if(i & 0x200)
        {
            if(x >= 61445)
                x -= 61445;
            else if(x >= 49156)
                x -= 49156;
            else if(x >= 36867)
                x -= 36867;
            else if(x >= 24578)
                x -= 24578;
            else if(x >= 12289)
                x -= 12289;
        }
        a->coeffs[j] = x;
        a->coeffs[j + distance] = montgomery_reduce((W * ((uint32_t) temp + 3*NEWHOPE_Q - adistance
            )));
        if((j + 2*distance) < NEWHOPE_N-1)
        {
            j += 2*distance;
        }
        else
        {
            jTwiddle = 0;
            if((start + 1) < distance)
            {
                start++;
                j = start;
            }
            else
            {
                start = 0;
                j = 0;
                distance <<= 1;
            }
        }
    }
}
```

Listing B.2: Optimized software implementation of the NTT

```
void shake128(unsigned char *output, unsigned long long outlen, const unsigned char *input,
    unsigned long long inlen)
```

```
2  {
3      #pragma HLS INTERFACE ap_bus depth=21000 port=output
4      #pragma HLS INTERFACE ap_bus depth=10000 port=input
5      uint64_t s[25] = {0};
6      unsigned char t[SHAKE128_RATE];
7      unsigned long long nblocks = outlen/SHAKE128_RATE;
8      size_t i;
9
10     /* Absorb input */
11     keccak_absorb(s, SHAKE128_RATE, input, inlen, 0x1F);
12
13     /* Squeeze output */
14     keccak_squeezeblocks(output, nblocks, s, SHAKE128_RATE);
15
16     output += nblocks*SHAKE128_RATE;
17     outlen -= nblocks*SHAKE128_RATE;
18
19     if (outlen)
20     {
21         keccak_squeezeblocks(t, 1, s, SHAKE128_RATE);
22         for (i = 0; i < outlen; i++)
23             output[i] = t[i];
24     }
25 }
26 void keccak_absorb(uint64_t *state, unsigned int shake_rate, const unsigned char *message,
       unsigned long long int message_length, unsigned char padding)
27 {
28     unsigned long long i;
29     unsigned char t[200];
30
31     while (message_length >= shake_rate)
32     {
33         for(i = 0; i < shake_rate / 8; ++i)
34         {
35             state[i] ^= load64(message + 8 * i);
36         }
37
38         KeccakF1600_StatePermute(state);
39         message_length -= shake_rate;
40         message += shake_rate;
41     }
42
43     for(i = 0; i < shake_rate; ++i)
44         t[i] = 0;
45     for(i = 0; i < message_length; ++i)
46         t[i] = message[i];
47
48     t[i] = padding;
49     t[shake_rate - 1] |= 128;
50
51     for (i = 0; i < shake_rate / 8; ++i)
52         state[i] ^= load64(t + 8 * i);
53 }
54 void keccak_squeezeblocks(unsigned char *output, unsigned long long int nblocks, uint64_t *s,
       unsigned int shake_rate)
55 {
56     #pragma HLS INLINE off
57     unsigned int i;
58
59     while(nblocks > 0)
60     {
61         KeccakF1600_StatePermute(s);
```

```
62          for (i = 0; i < (shake_rate>>3); i++)
63          {
64              store64(output+8*i, s[i]);
65          }
66          output += shake_rate;
67          nblocks--;
68      }
69  }
70  void KeccakF1600_StatePermute(uint64_t *state)
71  {
72      int round;
73
74      uint64_t Aba, Abe, Abi, Abo, Abu;
75      uint64_t Aga, Age, Agi, Ago, Agu;
76      uint64_t Aka, Ake, Aki, Ako, Aku;
77      uint64_t Ama, Ame, Ami, Amo, Amu;
78      uint64_t Asa, Ase, Asi, Aso, Asu;
79      uint64_t BCa, BCe, BCi, BCo, BCu;
80      uint64_t Da, De, Di, Do, Du;
81      uint64_t Eba, Ebe, Ebi, Ebo, Ebu;
82      uint64_t Ega, Ege, Egi, Ego, Egu;
83      uint64_t Eka, Eke, Eki, Eko, Eku;
84      uint64_t Ema, Eme, Emi, Emo, Emu;
85      uint64_t Esa, Ese, Esi, Eso, Esu;
86
87      //copyFromState(A, state)
88      Aba = state[ 0];
89      Abe = state[ 1];
90      Abi = state[ 2];
91      Abo = state[ 3];
92      Abu = state[ 4];
93      Aga = state[ 5];
94      Age = state[ 6];
95      Agi = state[ 7];
96      Ago = state[ 8];
97      Agu = state[ 9];
98      Aka = state[10];
99      Ake = state[11];
100     Aki = state[12];
101     Ako = state[13];
102     Aku = state[14];
103     Ama = state[15];
104     Ame = state[16];
105     Ami = state[17];
106     Amo = state[18];
107     Amu = state[19];
108     Asa = state[20];
109     Ase = state[21];
110     Asi = state[22];
111     Aso = state[23];
112     Asu = state[24];
113
114     for(round = 0; round < NROUNDS; round += 2)
115     {
116         //    prepareTheta
117         BCa = Aba^Aga^Aka^Ama^Asa;
118         BCe = Abe^Age^Ake^Ame^Ase;
119         BCi = Abi^Agi^Aki^Ami^Asi;
120         BCo = Abo^Ago^Ako^Amo^Aso;
121         BCu = Abu^Agu^Aku^Amu^Asu;
122
123         //thetaRhoPiChiIotaPrepareTheta(round , A, E)
```

```
124         Da = BCu^ROL(BCe, 1);
125         De = BCa^ROL(BCi, 1);
126         Di = BCe^ROL(BCo, 1);
127         Do = BCi^ROL(BCu, 1);
128         Du = BCo^ROL(BCa, 1);
129 
130         Aba ^= Da;
131         BCa = Aba;
132         Age ^= De;
133         BCe = ROL(Age, 44);
134         Aki ^= Di;
135         BCi = ROL(Aki, 43);
136         Amo ^= Do;
137         BCo = ROL(Amo, 21);
138         Asu ^= Du;
139         BCu = ROL(Asu, 14);
140         Eba =  BCa ^((~BCe)& BCi );
141         Eba ^= (uint64_t)KeccakF_RoundConstants[round];
142         Ebe =  BCe ^((~BCi)& BCo );
143         Ebi =  BCi ^((~BCo)& BCu );
144         Ebo =  BCo ^((~BCu)& BCa );
145         Ebu =  BCu ^((~BCa)& BCe );
146 
147         Abo ^= Do;
148         BCa = ROL(Abo, 28);
149         Agu ^= Du;
150         BCe = ROL(Agu, 20);
151         Aka ^= Da;
152         BCi = ROL(Aka, 3);
153         Ame ^= De;
154         BCo = ROL(Ame, 45);
155         Asi ^= Di;
156         BCu = ROL(Asi, 61);
157         Ega =  BCa ^((~BCe)& BCi );
158         Ege =  BCe ^((~BCi)& BCo );
159         Egi =  BCi ^((~BCo)& BCu );
160         Ego =  BCo ^((~BCu)& BCa );
161         Egu =  BCu ^((~BCa)& BCe );
162 
163         Abe ^= De;
164         BCa = ROL(Abe, 1);
165         Agi ^= Di;
166         BCe = ROL(Agi, 6);
167         Ako ^= Do;
168         BCi = ROL(Ako, 25);
169         Amu ^= Du;
170         BCo = ROL(Amu, 8);
171         Asa ^= Da;
172         BCu = ROL(Asa, 18);
173         Eka =  BCa ^((~BCe)& BCi );
174         Eke =  BCe ^((~BCi)& BCo );
175         Eki =  BCi ^((~BCo)& BCu );
176         Eko =  BCo ^((~BCu)& BCa );
177         Eku =  BCu ^((~BCa)& BCe );
178 
179         Abu ^= Du;
180         BCa = ROL(Abu, 27);
181         Aga ^= Da;
182         BCe = ROL(Aga, 36);
183         Ake ^= De;
184         BCi = ROL(Ake, 10);
185         Ami ^= Di;
```

```
186         BCo = ROL(Ami, 15);
187         Aso ^= Do;
188         BCu = ROL(Aso, 56);
189         Ema =  BCa ^((~BCe)& BCi );
190         Eme =  BCe ^((~BCi)& BCo );
191         Emi =  BCi ^((~BCo)& BCu );
192         Emo =  BCo ^((~BCu)& BCa );
193         Emu =  BCu ^((~BCa)& BCe );
194
195         Abi ^= Di;
196         BCa = ROL(Abi, 62);
197         Ago ^= Do;
198         BCe = ROL(Ago, 55);
199         Aku ^= Du;
200         BCi = ROL(Aku, 39);
201         Ama ^= Da;
202         BCo = ROL(Ama, 41);
203         Ase ^= De;
204         BCu = ROL(Ase, 2);
205         Esa =  BCa ^((~BCe)& BCi );
206         Ese =  BCe ^((~BCi)& BCo );
207         Esi =  BCi ^((~BCo)& BCu );
208         Eso =  BCo ^((~BCu)& BCa );
209         Esu =  BCu ^((~BCa)& BCe );
210
211         //   prepareTheta
212         BCa = Eba^Ega^Eka^Ema^Esa;
213         BCe = Ebe^Ege^Eke^Eme^Ese;
214         BCi = Ebi^Egi^Eki^Emi^Esi;
215         BCo = Ebo^Ego^Eko^Emo^Eso;
216         BCu = Ebu^Egu^Eku^Emu^Esu;
217
218         //thetaRhoPiChiIotaPrepareTheta(round+1, E, A)
219         Da = BCu^ROL(BCe, 1);
220         De = BCa^ROL(BCi, 1);
221         Di = BCe^ROL(BCo, 1);
222         Do = BCi^ROL(BCu, 1);
223         Du = BCo^ROL(BCa, 1);
224
225         Eba ^= Da;
226         BCa = Eba;
227         Ege ^= De;
228         BCe = ROL(Ege, 44);
229         Eki ^= Di;
230         BCi = ROL(Eki, 43);
231         Emo ^= Do;
232         BCo = ROL(Emo, 21);
233         Esu ^= Du;
234         BCu = ROL(Esu, 14);
235         Aba =  BCa ^((~BCe)& BCi );
236         Aba ^= (uint64_t)KeccakF_RoundConstants[round+1];
237         Abe =  BCe ^((~BCi)& BCo );
238         Abi =  BCi ^((~BCo)& BCu );
239         Abo =  BCo ^((~BCu)& BCa );
240         Abu =  BCu ^((~BCa)& BCe );
241
242         Ebo ^= Do;
243         BCa = ROL(Ebo, 28);
244         Egu ^= Du;
245         BCe = ROL(Egu, 20);
246         Eka ^= Da;
247         BCi = ROL(Eka, 3);
```

```
248         Eme ^= De;
249         BCo = ROL(Eme, 45);
250         Esi ^= Di;
251         BCu = ROL(Esi, 61);
252         Aga =  BCa ^((~BCe)& BCi );
253         Age =  BCe ^((~BCi)& BCo );
254         Agi =  BCi ^((~BCo)& BCu );
255         Ago =  BCo ^((~BCu)& BCa );
256         Agu =  BCu ^((~BCa)& BCe );
257
258         Ebe ^= De;
259         BCa = ROL(Ebe, 1);
260         Egi ^= Di;
261         BCe = ROL(Egi, 6);
262         Eko ^= Do;
263         BCi = ROL(Eko, 25);
264         Emu ^= Du;
265         BCo = ROL(Emu, 8);
266         Esa ^= Da;
267         BCu = ROL(Esa, 18);
268         Aka =  BCa ^((~BCe)& BCi );
269         Ake =  BCe ^((~BCi)& BCo );
270         Aki =  BCi ^((~BCo)& BCu );
271         Ako =  BCo ^((~BCu)& BCa );
272         Aku =  BCu ^((~BCa)& BCe );
273
274         Ebu ^= Du;
275         BCa = ROL(Ebu, 27);
276         Ega ^= Da;
277         BCe = ROL(Ega, 36);
278         Eke ^= De;
279         BCi = ROL(Eke, 10);
280         Emi ^= Di;
281         BCo = ROL(Emi, 15);
282         Eso ^= Do;
283         BCu = ROL(Eso, 56);
284         Ama =  BCa ^((~BCe)& BCi );
285         Ame =  BCe ^((~BCi)& BCo );
286         Ami =  BCi ^((~BCo)& BCu );
287         Amo =  BCo ^((~BCu)& BCa );
288         Amu =  BCu ^((~BCa)& BCe );
289
290         Ebi ^= Di;
291         BCa = ROL(Ebi, 62);
292         Ego ^= Do;
293         BCe = ROL(Ego, 55);
294         Eku ^= Du;
295         BCi = ROL(Eku, 39);
296         Ema ^= Da;
297         BCo = ROL(Ema, 41);
298         Ese ^= De;
299         BCu = ROL(Ese, 2);
300         Asa =  BCa ^((~BCe)& BCi );
301         Ase =  BCe ^((~BCi)& BCo );
302         Asi =  BCi ^((~BCo)& BCu );
303         Aso =  BCo ^((~BCu)& BCa );
304         Asu =  BCu ^((~BCa)& BCe );
305     }
306
307     //copyToState(state, A)
308     state[ 0] = Aba;
309     state[ 1] = Abe;
```

```
310     state[ 2] = Abi;
311     state[ 3] = Abo;
312     state[ 4] = Abu;
313     state[ 5] = Aga;
314     state[ 6] = Age;
315     state[ 7] = Agi;
316     state[ 8] = Ago;
317     state[ 9] = Agu;
318     state[10] = Aka;
319     state[11] = Ake;
320     state[12] = Aki;
321     state[13] = Ako;
322     state[14] = Aku;
323     state[15] = Ama;
324     state[16] = Ame;
325     state[17] = Ami;
326     state[18] = Amo;
327     state[19] = Amu;
328     state[20] = Asa;
329     state[21] = Ase;
330     state[22] = Asi;
331     state[23] = Aso;
332     state[24] = Asu;
333
334     #undef   round
335 }
336 uint64_t load64(const unsigned char *x)
337 {
338     #pragma HLS INLINE off
339     unsigned long long r = 0, i;
340
341     for (i = 0; i < 8; ++i)
342     {
343         r |= (unsigned long long)x[i] << 8 * i;
344     }
345     return r;
346 }
347
348 void store64(uint8_t *x, uint64_t u)
349 {
350     #pragma HLS INLINE off
351     unsigned int i;
352
353     for (i = 0; i < 8; ++i)
354     {
355         x[i] = (uint8_t)u;
356         u >>= 8;
357     }
358 }
```

Listing B.3: Unoptimized software implementation of SHAKE-128 with all used subfunctions

```
1  uint64_t s[25];
2  void KeccakF1600_StatePermute()
3  {
4      int round, i, j, x, y;
5      uint64_t B[25], D[5], RC;
6      #pragma HLS ARRAY_PARTITION variable=B complete dim=1
7      #pragma HLS ARRAY_PARTITION variable=D complete dim=1
8
```

```
9     L1: for(round = 0; round < NROUNDS; round++)
10    {
11        #pragma HLS PIPELINE
12
13        switch(round)
14        {
15            case  0: RC = 0x0000000000000001ULL; break;
16            case  1: RC = 0x0000000000008082ULL; break;
17            case  2: RC = 0x800000000000808AULL; break;
18            case  3: RC = 0x8000000080008000ULL; break;
19            case  4: RC = 0x000000000000808BULL; break;
20            case  5: RC = 0x0000000080000001ULL; break;
21            case  6: RC = 0x8000000080008081ULL; break;
22            case  7: RC = 0x8000000000008009ULL; break;
23            case  8: RC = 0x000000000000008AULL; break;
24            case  9: RC = 0x0000000000000088ULL; break;
25            case 10: RC = 0x0000000080008009ULL; break;
26            case 11: RC = 0x000000008000000AULL; break;
27            case 12: RC = 0x000000008000808BULL; break;
28            case 13: RC = 0x800000000000008BULL; break;
29            case 14: RC = 0x8000000000008089ULL; break;
30            case 15: RC = 0x8000000000008003ULL; break;
31            case 16: RC = 0x8000000000008002ULL; break;
32            case 17: RC = 0x8000000000000080ULL; break;
33            case 18: RC = 0x000000000000800AULL; break;
34            case 19: RC = 0x800000008000000AULL; break;
35            case 20: RC = 0x8000000080008081ULL; break;
36            case 21: RC = 0x8000000000008080ULL; break;
37            case 22: RC = 0x0000000080000001ULL; break;
38            case 23: RC = 0x8000000080008008ULL; break;
39            default: RC = 0x0000000000000000ULL; break;
40        }
41
42        for(i = 0; i < 5; i++)
43        {
44            B[i] = s[i] ^ s[i+5] ^ s[i+10] ^ s[i+15] ^ s[i+20];
45        }
46
47        for(i = 0; i < 5; i++)
48        {
49            D[i] = B[(i+4) % 5] ^ ROL(B[(i+1) % 5], 1);
50        }
51
52        for(x = 0; x < 5; x++)
53        {
54            for(y = 0; y < 5; y++)
55            {
56                s[x + 5*y] ^= D[x];
57            }
58        }
59
60        B[0] = s[0];
61        B[1] = ROL(s[6], 44);
62        B[2] = ROL(s[12], 43);
63        B[3] = ROL(s[18], 21);
64        B[4] = ROL(s[24], 14);
65        B[5] = ROL(s[3], 28);
66        B[6] = ROL(s[9], 20);
67        B[7] = ROL(s[10], 3);
68        B[8] = ROL(s[16], 45);
69        B[9] = ROL(s[22], 61);
70        B[10] = ROL(s[1], 1);
```

```
        B[11] = ROL(s[7], 6);
        B[12] = ROL(s[13], 25);
        B[13] = ROL(s[19], 8);
        B[14] = ROL(s[20], 18);
        B[15] = ROL(s[4], 27);
        B[16] = ROL(s[5], 36);
        B[17] = ROL(s[11], 10);
        B[18] = ROL(s[17], 15);
        B[19] = ROL(s[23], 56);
        B[20] = ROL(s[2], 62);
        B[21] = ROL(s[8], 55);
        B[22] = ROL(s[14], 39);
        B[23] = ROL(s[15], 41);
        B[24] = ROL(s[21], 2);

        for(x = 0; x < 5; x++)
        {
            for(y = 0; y < 5; y++)
            {
                s[x+5*y] = B[x+5*y] ^ ((~B[(x+1)%5+5*y]) & B[(x+2)%5+5*y]);
            }
        }
        s[0] ^= RC;
    }
    #undef   round
}
```

Listing B.4: Optimized permutation of SHAKE

```
uint64_t s[25];
void shake128(uint16_t *output, uint16_t outlen, const uint16_t *input, uint16_t inlen)
{
    #pragma HLS ARRAY_PARTITION variable=s complete dim=1
    #pragma HLS INTERFACE ap_bus depth=21000 port=output
    #pragma HLS INTERFACE ap_bus depth=10000 port=input

    size_t i, j;
    uint16_t end;
    uint64_t r;

    /* Absorb input */

    Reset: for(i = 0; i < 25; i++)
    {
        #pragma HLS UNROLL
        s[i] = 0;
    }

    Absorb: while(inlen > 0)
    {
        if(inlen >= SHAKE128_RATE)
            end = SHAKE128_RATE >> 3;
        else
            end = inlen >> 3;

        L1: for(i = 0; i < end; i++)
        {
            r = 0;
            r = SET_RANGE(r, 15, 0, ROL_16(input[0], 8));
            r = SET_RANGE(r, 31, 16, ROL_16(input[1], 8));
            r = SET_RANGE(r, 47, 32, ROL_16(input[2], 8));
            r = SET_RANGE(r, 63, 48, ROL_16(input[3], 8));
```

```
            s[i] ^= r;
            input += 4;
        }

        if(end == (SHAKE128_RATE >> 3))
            KeccakF1600_StatePermute();

        inlen -= (end << 3);
    }

    s[20] = SET_RANGE(s[20], 63, 63, (0x1 ^ GET_RANGE(s[20], 63, 63)));

    /* Squeeze output */
    Squeeze: while(outlen > 0)
    {
        KeccakF1600_StatePermute();
        if(outlen >= SHAKE128_RATE)
            end = SHAKE128_RATE >> 3;
        else
            end = outlen >> 3;

        L2: for (i = 0; i < end; i++)
        {
            r = s[i];
            L2_2: for(j = 0; j < 4; j++)
            {
                output[4*i+j] = ROL_16((r & 0xFFFF), 8);
                r >>= 16;
            }
        }
        output += (end << 2);
        outlen -= (end << 3);
    }
}
```

Listing B.5: Optimized implementation of SHAKE

```
void shake128_10240(uint16_t *output, const uint16_t *input)
{
    #pragma HLS ARRAY_PARTITION variable=s complete dim=1
    unsigned char i, j;

    //Reset state
    Reset: for(i = 0; i < 25; i++)
    {
        #pragma HLS UNROLL
        s[i] = 0;
    }

    //Absorb the 18 Byte input and apply the padding
    s[0]  = SET_RANGE( s[0], 15, 0, input[0]);
    s[0]  = SET_RANGE( s[0], 31, 16, input[1]);
    s[0]  = SET_RANGE( s[0], 47, 32, input[2]);
    s[0]  = SET_RANGE( s[0], 63, 48, input[3]);
    s[1]  = SET_RANGE( s[1], 15, 0, input[4]);
    s[1]  = SET_RANGE( s[1], 31, 16, input[5]);
    s[1]  = SET_RANGE( s[1], 47, 32, input[6]);
    s[1]  = SET_RANGE( s[1], 63, 48, input[7]);
    s[2]  = SET_RANGE( s[2], 15, 0, input[8]);
    s[2]  = SET_RANGE( s[2], 23, 16, 0x1F);
    s[20] = SET_RANGE(s[20], 63, 63, 1);

```

```
26      //Squeeze 8 full blocks
27      Squeeze: for(i = 0; i < 8; i++)
28      {
29          KeccakF1600_StatePermute();
30          Squeeze_1: for(j = 0; j < 21; j++)
31          {
32              output[4*j+0] = GET_RANGE(s[j], 15, 0);
33              output[4*j+1] = GET_RANGE(s[j], 31, 16);
34              output[4*j+2] = GET_RANGE(s[j], 47, 32);
35              output[4*j+3] = GET_RANGE(s[j], 63, 48);
36          }
37      output += (21 << 2);
38      }
39  }
```

Listing B.6: Function `shake128_10240` that generates one row of A

List of Tables

List of Algorithms

List of Listings

Bibliography

[AAB+19] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. NewHope – Algorithm Specifications and Supporting Documentation. 2019.

[ABB+20] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+: Submission to the NIST post-quantum project, v.3. 2020.

[ABD+19a] Erdem Alkim, Joppe Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM - Learning With Errors Key Encapsulation. 2019.

[ABD+19b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation. 2019.

[BBF+19] Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and Fast Post-Quantum Public-Key Encryption. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 83–102, Cham, 2019. Springer International Publishing.

[BBMD+20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel van Beirendock, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 3 Submission). 2020.

[BCL+19] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic McEliece: Conservative Code-Based Cryptography. 2019.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS – A Practical Forward Secure Signature Scheme Based on Minimal Security

Assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 117–129, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BFM+18] K. Braun, T. Fritzmann, G. Maringer, T. Schamberger, and J. Sepúlveda. Secure and Compact Full NTRU Hardware Implementation. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 89–94, 2018.

[CCKK15] D. Chi, J. W. Choi, J. S. Kim, and T. Kim. Lattice Based Cryptography for Beginners. *IACR Cryptol. ePrint Arch.*, 2015:938, 2015.

[CDH+19] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU – Algorithm Specifications And Supporting Documentation. 2019.

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

[fISB20] Federal Office for Information Security (BSI). Cryptographic Mechanisms: Recommendations and Key Lengths, March 2020. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile`.

[FND+19] F. Farahmand, D. T. Nguyen, V. B. Dang, A. Ferozpuri, and K. Gaj. Software/Hardware Codesign of the Post Quantum Cryptography Algorithm NTRUEncrypt Using High-Level Synthesis and Register-Transfer Level Design Methodologies. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 225–231, 2019.

[Fro20] FrodoKEM – Code, 2020 (accessed December 02, 2020). `https://frodokem.org/#code`.

[HG15] Ekawat Homsirikamol and Kris Gaj. Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 217–228, Cham, 2015. Springer International Publishing.

[HG17] E. Homsirikamol and K. G. George. 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 (ICFPT)*, pages 120–127, 2017.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR Cryptology ePrint Archive*, 2018:686, 2018.

[KAMK16] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology – INDOCRYPT 2016*, pages 191–206, Cham, 2016. Springer International Publishing.

[LGCN20] Mariano López-García and Enrique Cantó-Navarro. Hardware-Software Implementation of a McEliece Cryptosystem for Post-quantum Cryptography. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Advances in Information and Communication*, pages 814–825, Cham, 2020. Springer International Publishing.

[MCF] D. McGrew, M. Curcio, and S. Fluhrer. Leighton-Micali Hash-Based Signatures. `https://www.rfc-editor.org/info/rfc8554`.

[NIS20] Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, 2017 (accessed November 24, 2020). `https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf`.

[NTR20] NTRU - Software, 2020 (accessed November 27, 2020). `https://ntru.org/software.shtml`.

[OG19] Tobias Oder and Tim Güneysu. Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 128–142, Cham, 2019. Springer International Publishing.

[OS09] Raphael Overbeck and Nicolas Sendrier. *Code-based cryptography*, pages 95–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. `https://doi.org/10.1007/978-3-540-88702-7_4`.

[Pei16] C. Peikert. *A Decade of Lattice Cryptography*. 2016.

[Reg05] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. volume 56, pages 84–93, 01 2005.