UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Acceleration of Post-Quantum Cryptography on OpenTitan Big Number Accelerator using Instruction Set Extensions

Emma Urquhart

Trinity College

June 2024

Submitted in partial fulfillment of the requirements for the Master of Philosophy in Advanced Computer Science

Total page count: 54

Main chapters (excluding front-matter, references and appendix): 41 pages (pp 9–49)

Main chapters word count: 14939

Methodology used to generate that word count:

```
$ make wordcount
gs -q -dSAFER -sDEVICE=txtwrite -o - -dFirstPage=9 -dLastPage=48 \
./MPhil_Thesis.pdf | egrep '[A-Za-z]{3}' | wc -w
14939
```

I, Emma Urquhart of Trinity College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this project report and the work described in it are my own work, unaided except as may be specified below, and that the project report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this project report I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my project report to be made available to the students and staff of the University.

Signed:

Date: 31/05/2024

Abstract

Acceleration of Post-Quantum Cryptography on OpenTitan Big Number Accelerator using Instruction Set Extensions

The impending threat of quantum computing is advancing alongside the proliferation of the internet of things (IoT). In an era of ubiquitous computing and evolving security risks, post-quantum cryptography is emerging as a critical safeguard that may soon become indispensable. The release of the first open-source silicon chip by OpenTitan in February 2024 marks a major breakthrough in secure and trustworthy hardware [26]. Security is a fundamental aspect of the OpenTitan project and the platform is equipped with a custom cryptographic co-processor, the OpenTitan Big Number Accelerator (OTBN). Ideally suited for integration into IoT devices, a challenge still remains in the optimisation of OTBN for post-quantum cryptography. We present 8 new instructions for acceleration of the Kyber number theoretic transform and inverse number theoretic transform on OTBN and integrate them into optimised implementations. We demonstrate a significant performance improvement factor of 21.1$\times$ over the baseline implementation on OTBN for the number theoretic transform and a performance improvement factor of 24.3$\times$ for its inverse. Through hardware/software co-design, our approach fully leverages the potential for parallelism, maximally exploits the existing capabilities of OTBN and proposes some moderate hardware modifications to the platform.

Acknowledgements

I would like to thank my supervisor, Prof. Frank Stajano, for all his support, guidance and feedback on my research project throughout the year. I would also like to thank Prof. Robert Mullins and Dr. Andreas Kurth of lowRISC for helping to define the project, offering access to relevant resources and supporting the implementation efforts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As post-quantum cryptography (PQC) progresses towards mainstream adoption, classical processors remain unoptimised for PQC algorithms. Through hardware/software co-design, we demonstrate the acceleration potential of classical cryptographic co-processors, such as the OpenTitan Big Number Accelerator (OTBN), for PQC. At present, public key cryptography (PKC) algorithms such as Rivest-Shamir-Adleman (RSA) [39] and Elliptic Curve Cryptography (ECC) [28] are used to ensure the confidentiality and integrity of online communications. Execution of these algorithms can be expensive on resource-constrained devices. Therefore, many are equipped with specialized cryptographic co-processors which are optimised for their most expensive operations. In February 2024, the first open-source silicon chip was released by lowRISC and the OpenTitan coalition [26], marking a significant development in secure and transparent hardware. Based on the EarlGrey discrete root of trust (RoT), it incorporates OTBN, a cryptographic co-processor, designed for use in IoT devices. Security-critical functionality offered by the chip includes secure boot, safeguarding of cryptographic keys and secure communication.

The advent of quantum computing poses an impending threat to the public-key cryptosystems which are integral to the security of communications between IoT devices. The hard mathematical problems on which algorithms such as RSA and ECC rely include factorization of large integers and the discrete logarithm problem. These tasks are computationally infeasible for classical computers, but can be solved in polynomial time by quantum adversaries, as proven by Shor [43]. To mitigate this threat, PQC solutions are being developed. This form of cryptography can be implemented on classical hardware but is capable of withstanding quantum attacks. The National Institute for Standards and Technology (NIST) is conducting a standardization process for PQC, currently in its fourth round [12]. Although the standardisation process is still underway, various algorithmic commonalities between candidates are already emerging. Therefore, it is possible to optimise for costly operations which are likely to become prevalent in the future.

Lattice-based cryptography is emerging as a promising approach to PQC, encompassing three of the four algorithms already selected for standardisation: Kyber [35], a key encapsulation mechanism and digital signature algorithms Dilithium [14] and Falcon [18]. However, implementation of these algorithms poses practical challenges as their core operations incur significant overhead on existing platforms. Modern cryptographic co-processors such as OTBN are optimised for RSA and ECC, providing large integer arithmetic capabilities. However, lattice-based PQC algorithms do not involve such operations and therefore do not benefit from these capabilities. OTBN is not yet specialised for PQC. Our project brief was to accelerate PQC on OTBN through hardware/software co-design. We propose realistic instruction set extensions for OTBN. We leverage our new instructions in optimised implementations and significantly enhance performance of PQC functions. We exploit OTBN's wide data path to maximise vectorisation. In addition, distinctive hardware features such as its 512-bit barrel shifter are leveraged and/or re-purposed.

Due to the predicted prevalence of lattice-based PQC, we focus on a characteristic bottleneck of such schemes: polynomial multiplication. We accelerate polynomial multiplication via the number theoretic transform (NTT) and inverse number theoretic transform (INTT) in Kyber. Our techniques should also be applicable to Dilithium, due to the high level of similarity between the `ntt` and `invntt` function implementations. The target platform is OTBN. The proposed instruction set extensions and software implementations leverage its existing capabilities and incorporate viable extensions to the architecture. We have submitted a paper to the 23rd International Conference on Cryptology And Network Security (CANS) (pending acceptance at the time of writing) and our code is publicly available on the author's fork of the OpenTitan repository (submitted in .zip format), including documentation for reproducibility of results. Our main contributions include:

- Baseline implementations of NTT and INTT in Kyber using the existing OTBN instruction set, replicating the reference implementations as closely as possible.
- Identification and analysis of the bottlenecks in the baseline implementations.
- 9 new instructions for acceleration of (I)NTT on OTBN, cycle count estimates for each and descriptions of the required hardware modifications.
- Vectorized implementations of the (I)NTT in Kyber, which leverage the new instructions and reduce cycle cost from 91,939 to 4,356 (21.1$\times$) for NTT and 149,435 to 6,133 (24.3$\times$) for INTT, over the baseline implementations.
- Analysis of performance optimisations and comparison with the performance of the reference (I)NTT implementations on RISC-V Ibex, the architecture on which OpenTitan is modelled.
- Extension to the OTBNSim testing infrastructure, automating the generation of input/output file pairs. Enhancement of debugging capabilities within OTBNsim by re-purposing of existing code from the OpenTitan repository.

Chapter 2

Background

2.1 Post-Quantum Cryptography

We begin this section with an overview of cryptography and the motivation for its evolution in anticipation of quantum advancement. We consider the relevance of post-quantum cryptography to the OpenTitan platform. We describe the current status of the NIST PQC standardisation process, which informs our choice of Kyber as our target algorithm for acceleration.

2.1.1 Cryptography and the transition to the quantum era

Cryptography is the practice of securing communications using mathematical techniques, such that only the sender and the intended recipient can view the message contents [38]. Cryptography underpins the security of digital communications, which have become intrinsic to everyday life. The internet now encompasses a complex web of interconnected devices. Currently, PKC enables confidential communication between devices over insecure channels such as the internet. PKC utilises corresponding public and private keys for encryption and decryption, respectively. Compared to their symmetric (private-key) counterparts, public-key algorithms are more computationally complex due to the different key types and the requirement for much longer keys to guarantee an equivalent level of security. However, on the internet, communicating parties cannot meet to securely exchange private keys in advance, therefore the use of PKC is required.

Current standards for PKC include *Rivest-Shamir-Adleman (RSA)* [39] and *Elliptic Curve Cryptography (ECC)* [28]. The security of both of these algorithms is based on the hardness of factoring a large positive integer into its prime factors. In 1994, Shor published a quantum algorithm capable of computing the prime factors of any large positive integer in polylogarithmic time [42]. Currently, quantum computers do not have the capacity to break PKC algorithms if a sufficiently large key is used. However, quantum computing is a rapidly developing field. It seems that the realisation of large-scale quantum computers is imminent and inevitable, and once this happens, current PKC cryptosystems will be rendered obsolete [4].

Bernstein and Lange [4] present approaches to cryptography which are invulnerable to cryptanalysis by a quantum adversary. PQC describes a class of algorithms which are capable of withstanding both classical and quantum attacks, can be deployed on general-purpose hardware and are interoperable with existing protocols and networks, which will facilitate a seamless transition. However, PQC algorithms are prohibitively inefficient on general-purpose hardware in its current form. There are a number of contributing factors to this challenge:

- PQC algorithms are based on different and often more complex mathematical problems than traditional algorithms. This increases the computational effort required.
- Certain operations, such as those involving large polynomials, can be very resource-intensive on hardware that has not been specialised for their execution.
- The maturity of algorithms such as RSA and ECC has resulted in highly optimised deployments of the classical PKC standards. This is the result of decades of focused research. As PQC is still in its infancy, the full potential for optimisations and specialised support has not yet been explored.

The potential rise to prominence of PQC is prompting research in the optimisation domain. Analysis of current platforms and their incremental adaptation to support faster PQC are prerequisites for the transition to quantum-secure communication infrastructure. As an emerging and pioneering platform in the open-source hardware ecosystem, the same considerations and investigations should be tailored to OpenTitan, specifically OTBN.

2.1.2 NIST PQC Standardisation Process

NIST began a standardisation process for PQC in 2016 [12], which is now in its fourth round. The candidate algorithms are potential substitutes for classical PKC solutions, aiming to address the tasks of general encryption and digital signatures, for which RSA and ECC currently employed. Two main classes of algorithms have emerged:

- **Digital signature schemes:** algorithms that enable the sender to sign a message with their private key and the receiver to authenticate that message by verifying the signature with the sender's public key.
- **Key Encapsulation mechanisms (KEM):** algorithms that enable establishment of a shared secret key between two parties over non-confidential communication channels. KEMs have 3 main stages and progress as follows.

  In the *key generation* phase, a public/private key pair is generated. Any party with access to another party's public key can establish a shared secret key with them.

The sender provides as input to the *encapsulation* algorithm the intended recipient's public key. The encapsulation algorithm generates and returns a shared secret key. It also encapsulates (effectively, encrypts) the shared secret key using the receiver's public key. The resulting encapsulation is then forwarded to the receiver.

The receiver can then *decapsulate* it using their private key. The decapsulation process will yield the unencrypted shared secret key. This key may then be used for secure communication between the two parties, e.g. via symmetric encryption.

NIST has so far selected four PQC algorithms for standardisation: CRYSTALS-Kyber [35], a key encapsulation mechanism, along with three digital signature schemes: CRYSTALS-Dilithium [14], Sphincs$^+$ [4] and Falcon [18]. Kyber was selected for general encryption, e.g. in accessing secure websites. Its main advantages include short encryption keys which facilitate ease of transmission and its speed of execution, relative to competitors. Of the three digital signature schemes, NIST endorses Dilithium as the primary algorithm. Falcon was selected for its applicability to scenarios requiring smaller signatures than Dilithium. Sphincs$^+$, although larger and slower than the other two, was selected for its foundation in hash functions, to avoid over-reliance on lattice-based schemes.

In our selection of a suitable target algorithm for acceleration on OpenTitan, we consulted experts from lowRISC who suggested focusing on a KEM. Since only one had been selected for standardisation (Kyber), it was the logical choice. In the context of real-world deployment of OpenTitan, which is specialised for integration in lightweight IoT devices, a key encapsulation mechanism such as Kyber would be critical to ensuring efficient key establishment between interconnected devices on public channels.

2.2 The Kyber Key Encapsulation Mechanism

Kyber is currently the only quantum-secure KEM which has been approved for standardisation. In this section, we explore the fundamental hard mathematical problem on which Kyber is based, the ring-learning-with-errors (RLWE) problem. We profile the reference implementation of Kyber and analyse the distribution of computational effort within the algorithm. Based on the profiling results, we discuss why the NTT and INTT are ideal candidates for acceleration on OTBN.

2.2.1 The Learning with Errors (LWE) Problem

The learning with errors (LWE) problem is a computationally hard problem, meaning that no algorithm has been found which can solve it efficiently [37]. It was originally proposed by Regev in 2005 [36]. It forms the basis for the cryptographic strength of lattice-based schemes, a prominent class of PQC algorithms which accounts for three of the four algorithms selected by NIST for standardisation.

LWE and its derivatives are based on the concept of solving a "noisy" system of linear equations, i.e. a small amount of error is added to the system to create a computationally hard problem. Mathematically, such a system may be represented as follows, where the matrix of coefficients A and the vector b combine to form the public key and the vector s represents the secret key. e represents an error vector, which is typically a random vector of small values used to perturb the equations slightly.

$$A * (s + e) = b \tag{2.1}$$

The addition of error to the equation greatly amplifies the complexity of the problem. To solve the system, an adversary would need to extract the value s. The sizes of the matrix and vectors vary according to the target security level of the algorithm.

The LWE problem has several variants which satisfy different requirements. The Ring Learning with Errors (RLWE) problem restricts computations and values to polynomial rings, instead of matrices and vectors in LWE. This leads to significantly smaller key sizes and more efficient computations.

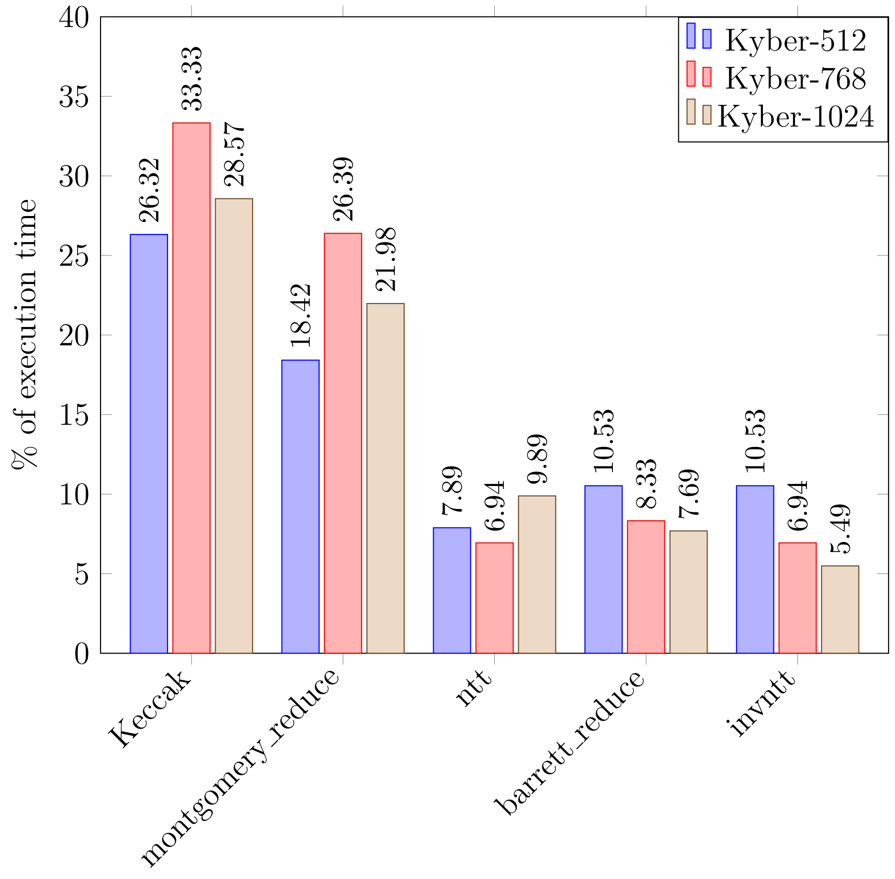
The variant of LWE used in Kyber is known as Module Learning with Errors (MLWE) [23]. MLWE is a variant of LWE, which reconciles the flexibility of LWE with the efficiency and lower bandwidth requirements of RLWE. Module lattices, used in MLWE, are more complicated than the ideal lattices used in RLWE, yet more structured (i.e. less complicated) than the Euclidean lattices used in LWE [44]. Therefore, the hardness of MLWE schemes is based on an intermediate problem between RLWE and LWE. The use of MLWE circumvents various security risks associated with the use of RLWE. These risks have been highlighted in numerous works demonstrating vulnerabilities of RLWE which exploit the algebraic structure of ideal lattices [6], [16], [10], [9], [11]. MLWE also offers the advantage of straightforward scalability to higher security levels, requiring only minimal changes to the parameters of Kyber [7].

2.2.2 Profiling the reference Kyber C implementation

After selecting Kyber as the focus of our optimisation efforts, we conducted an analysis of the reference C implementation to inform our choice of algorithmic components to target for acceleration. Two implementations of Kyber are available within the official repository [35], a platform-agnostic implementation (within `ref/`) and an optimised AVX2 implementation. The optimised implementation may be run on processors which support the AVX2 instruction set. AVX2 offers capabilities for signed and unsigned processing of high and low parts of packed values within SIMD registers. These processors also offer out-of-order execution, meaning that instructions can be interleaved. Our target platform is OTBN, which has a restricted instruction set, does not support advanced extensions like AVX2 and does not support out-of-order execution. Therefore, the platform-independent implementation was the most suitable reference point for our work. It should, however, be noted that the reference implementation has not been optimised for any platform and instead favours readability over performance. Given that OTBN is not yet equipped with a compiler, the reference C code cannot be directly excuted on the platform. We obtained the profiling results by executing the reference code on a regular laptop (Core i7 processor). We assume an approximate equivalence in terms of distribution of computational effort within the Kyber algorithm between the reference C code and an OTBN implementation.

Three executables are generated for each parameter set (512, 768 and 1024) by compiling the test program; `test_kyber$ALG`, `test_kex$ALG` and `test_vectors$ALG`, where `$ALG` identifies the parameter set. According to the repository documentation, `test_kyber$ALG` runs 1,000 tests which encompass key generation, encapsulation and decapsulation. We used `test_kyber$ALG` for profiling. The parameter sets correspond to the different security levels of Kyber. We obtained results for all three security levels. The algorithm remains the same for each security level; all that changes are parameter values. We generated a flat profile for each parameter set using `gprof` [17]. The flat profile captured the amount of time spent in the execution of each function. We set the profiling (`-pg`) flag in the Makefile's `CFLAGS`. This enabled generation of the `gmon.out` file containing the profiling data, which was compiled into a text file report using the `gprof` command. A visualisation of the percentage of execution time spent within each function is presented in Figure 2.1.

Figure 2.1: Profiling results for reference (I)NTT implementations



From the data in Figure 2.1, we firstly note that the distributions across the parameter sets are similar, which aligns with expectations given that the code is the same. The Keccak function is the most computationally intensive component of the algorithm. However, the Keccak core is most conducive to acceleration in pure hardware, as it was designed as a hardware-oriented implementation of the SHA-3 hashing algorithm [5]. The purpose of our research is to investigate acceleration of Kyber on OTBN through hardware/software co-design. Therefore, while a custom Keccak accelerator could likely be integrated as a hardware extension, it is not the ideal candidate for optimisation via instruction set extensions which do not aim to drastically alter the hardware architecture.

The subsequent four most expensive functions are `montgomery_reduce`, `barrett_reduce`, `ntt`, and `invntt`. It is important to note that the `montgomery_reduce` function is called from both `ntt` and `invntt`, while the `barrett_reduce` function is called from the `invntt` function. Algorithmically, the `ntt` and `invntt` functions are closely (inversely) related and hence share a number of similarities. Although the implementation of `invntt` is slightly longer and more complex, it appeared likely that it would be possible to design certain optimisations to target both functions. Because `ntt` and `invntt` enclose `montgomery_reduce` and `barrett_reduce`, these functions were also targeted in our acceleration strategies.

2.2.3 The Number Theoretic Transform and its Inverse

Efficient polynomial multiplication is one of the most performance-critical implementation elements of lattice-based PQC schemes such as Kyber and Dilithium. The number theoretic transform (NTT) is commonly adopted for this purpose, enabling a reduction in complexity of polynomial multiplication from $O(n^2)$ to $O(n)$, where n is the number of terms in each polynomial. This is compared to the most basic method of polynomial multiplication, sometimes known as the schoolbook method, which entails multiplying each term of the first polynomial with all of the terms of the second polynomial and adding the results. In the context of the ring-based polynomials in Kyber, the final result would need to be reduced by $\phi(x) = x^n + 1$. For the large polynomials processed by lattice-based schemes, the magnitude of this efficiency optimisation is particularly significant.

The number theoretic transform is a specialised form of the Discrete Fourier Transform, however it operates on the finite field $\mathbb{Z}_q$ instead of complex numbers. The NTT operates by transforming polynomials into a domain in which multiplication is highly efficient. The multiplications are performed within the NTT domain before the results are transformed back into the normal domain using the inverse number theoretic transform (INTT). Specifically, the NTT transforms polynomials from their coefficient representation (a vector of coefficients) to their point-value form (a vector of points along the polynomial). For polynomials of degree n, n points are sufficient to form the basis for the point-value representation. Multiplication of two polynomials in their point-value form is a straightforward pointwise product. Given two polynomials f and g, we compute their product according to Equation 2.2, where $\circ$ denotes multiplication within the NTT domain:

$$f \cdot g = INTT(NTT(f) \circ NTT(g)) \tag{2.2}$$

The operation of the NTT is given in Equation 2.3. This equation describes the transformation on a polynomial g of degree n, where $g = \sum_{i=0}^{n-1} g_i X^i$ and $g_i \in \mathbb{Z}_q$. Furthermore, ω represents the primitive n-th root of unity, where $\omega^n \equiv 1 \mod q$ and for any $1 \leq k < n$, $\omega^k \not\equiv 1 \mod q$. Values of ω^{ij} are known as twiddle factors. Multiplication by these values is equivalent to evaluating the polynomial at powers of the n-th root of unity. In the Kyber reference implementation and in our implementations, the twiddle factors are stored as pre-computed constants. The INTT reverses this transformation (Equation 2.4).

$$\hat{g} = \mathrm{NTT}(g) = \sum_{i=0}^{n-1} \hat{g}_i X^i, \quad \text{with} \quad \hat{g}_i = \sum_{j=0}^{n-1} g_j \omega_n^{ij} \pmod{q}. \tag{2.3}$$

$$g = \mathrm{INTT}(\hat{g}) = \sum_{i=0}^{n-1} g_i X^i, \quad \text{with} \quad g_i = n^{-1} \sum_{j=0}^{n-1} \hat{g}_j \omega_n^{-ij} \pmod{q}. \tag{2.4}$$

In the Kyber reference implementation and in this work, Montgomery reduction and Barrett reduction are integrated into the NTT and INTT functions, to efficiently perform modular arithmetic within the NTT domain. Montgomery reduction is used in both NTT and INTT, while Barrett reduction is used within INTT only. In the Kyber implementation, the `ntt` and `invntt` functions are computed in-place for efficiency reasons, as this method incurs no additional memory overhead. The consequence of this is that the outputs are in bit-reversed order. However, this is handled within the functions, e.g. the `invntt` function expects its input in reverse.

Montgomery reduction converts numbers to a specialised form known as Montgomery form, to enable faster modular multiplication. Numbers of the form $a \pmod{m}$ are represented as $aR \pmod{m}$, where R is a power of 2 and $R > m$. This allows modular multiplication without explicit division by a modulus. Instead of directly computing $a \times b \pmod{m}$, the same result is more efficiently obtained by computing $((a \times b) \times R^{-1}) \pmod{m}$, assuming a and b are already in Montgomery form. By incorporating multiplication by a power of 2 for computational efficiency, this approach leverages the native bitwise operations in hardware and hence benefits from the speed of bit-shifting, which is used as a substitute for costly division operations.

Barrett reduction also enables fast modular computations, but by following a different approach. A precomputed value μ is designed to approximate $\frac{1}{m}$ in a form that can efficiently be handled in hardware, according to the formula $\left\lfloor \frac{2^k}{m} \right\rfloor$, where k is selected as a value slightly larger than the bitlength of m. Division of a by m can then be approximated efficiently by: $a \pmod{m} = a - \mu p$. The intrinsic link with binary representation once again facilitates the use of bit-shifting as a substitute for costly arithmetic multiplication and division operations. Barrett reduction is used for efficiently computing long divisions required for evaluating modulus in INTT, by replacing division with multiplication.

2.3 OpenTitan Big Number Accelerator

The OpenTitan project is a collaboration between nine coalition members [26]. The project is hosted by lowRISC C.I.C. [25], a not-for-profit company based in Cambridge, U.K. which aims to develop and maintain an open-source silicon ecosystem. It was originally launched by Google, lowRISC and their partners in 2018. The project reached a significant milestone in February 2024, becoming the first open-source silicon project to reach commercial availability with the release to market of validated chips based on the OpenTitan Earl Grey discrete root of trust (RoT).

A RoT is a highly reliable component which is inherently trusted within a cryptographic system [15]. Because RoTs are responsible for performing security-critical operations and safeguarding secret values such as cryptographic keys, they must be secure by design. A fundamental hardware security primitive of OpenTitan is the OpenTitan Big Number Accelerator (OTBN), a co-processor for acceleration of asymmetric cryptography. OTBN provides large number arithmetic capabilities which are fundamental to PKC algorithms such as RSA and ECC. Its wide data path and specialized instruction set enhance the efficiency of PKC. However, OTBN is not yet optimised for PQC. As OpenTitan forges a path to the future of secure hardware, the emerging threat of quantum computing cannot be overlooked. To ensure the resilience of OTBN in the future, it is imperative to optimise this platform for secure and efficient PQC before existing solutions are rendered obsolete.

OTBN features a 32-bit wide control path and a 256-bit wide data path, each containing 32 registers. Its security-centric design incorporates a reduced instruction set which comprises a base subset for control flow and a big number subset for wide-integer arithmetic in data flow. The separation of paths reduces the risk of data leakage. OTBN supports data integrity protection and secure wipe of internal states, which is initiated upon encountering unexpected errors. Security is enhanced by its internal random number generation mechanism which is connected to the Entropy Distribution Network. OTBN contains two dedicated memories of 4kiB: instruction memory (IMEM) and data memory (DMEM). Each addresses different aspects of cryptographic processing and their separation bolsters security. The integrity of the instruction stream is ensured by the inaccessibility of IMEM to user-invoked load/store operations, which are performed on the DMEM.

2.4 Use Cases of OTBN

This section describes the potential for real-world impact of our work on accelerating PQC on OTBN. We detail the broad array of applications and use cases of OpenTitan, exploring their potential vulnerability to quantum computing and how they may be adapted to support PQC as a defensive measure. The official documentation [34] identifies three distinct uses cases for OpenTitan —platform integrity modules, trusted platform modules and universal 2nd-factor security keys, each of which encompass a broad spectrum of applications. We analyse how OTBN contributes to the core functionality of OpenTitan in each use case. We describe how our contributions may help to ensure the robustness of OpenTitan in these application scenarios in the quantum era.

2.4.1 Platform Integrity Modules

Platform integrity modules play a crucial role in securing the boot process. This process provides assurance that the device is starting in a trustworthy state and verifies that its firmware has not been tampered with. This guarantee is crucial in situations where technology is entrusted with handling sensitive data or performing critical operations. Devices which are likely to incorporate platform security modules include surveillance cameras, healthcare monitors, autonomous vehicles and industrial process control systems.

Within platform integrity modules, OpenTitan performs the following tasks:

1. **Firmware Integrity Verification.** OpenTitan acts as an intermediary between the device's boot flash and its other components. Once the device starts, OpenTitan verifies the integrity of the initial stages of the boot firmware. OTBN currently uses RSA with SHA-256 for digital signature verification in this process. Once verification is complete, OpenTitan grants the other boot devices access to the boot flash —the program that defines the boot sequence.

2. **Monitoring of downstream devices**. When the device is operational, OpenTitan monitors the resets and heartbeat signals of the downstream devices. Deviation from expected patterns could indicate compromise and triggers interrupt service routines. The role of OTBN in this process is likely minor, potentially including verification of downstream devices using ECDSA (a form of ECC) with SHA-256.

3. **Enforcement of runtime boot device access policies.** OpenTitan manages access of boot devices to the boot flash and A/B firmware updates. A/B updates install seamlessly and include a mechanism for recovery from corruption. OTBN may be used to verify that the firmware updates have indeed come from the manufacturer via digital signatures.

4. **Root key storage and attestation flows.** OpenTitan securely stores and manages cryptographic keys. Keys and internal flash data are encrypted with AES.

Attestation flows use the keys from the secure storage to generate a digital signature for the device. This can be used to prove the identity of the device via remote attestation. OTBN uses ECDSA with SHA-256 to generate the digital signature.

Digital signature algorithms are critical in platform integrity modules, with PKC also being used for securing transmission of data between communicating devices. With the impending quantum threat, classical algorithms such as RSA and ECDSA (a form of ECC) will be replaced by Dilithium for digital signature generation and Kyber for secure data transmission. AES is used to secure keys and data within the internal flash. Although quantum computers have the potential to reduce its security level, it is currently considered quantum-safe. Given that our (I)NTT implementations are targeted at Kyber, which shares similarities with Dilithium, our contributions are applicable to this use case.

2.4.2 Trusted Platform Modules

The Trusted Platform Module specification (TPM 2.0) is an international standard for secure cryptoprocessors. The purpose of this technology is to provide secure storage for artifacts which are used to authenticate computing platforms, e.g. cryptographic keys, digital certificates and passwords. Compared to platform integrity modules, trusted platform modules provide a broader range of functionality. In addition to enabling device authentication, attestation and secure boot, trusted platform modules also generate cryptographic keys and provide disc encryption. Confidence in trusted platform modules is strengthened through conformance to the TPM specification, which is a widely-recognised standard. Trusted platform modules play a pivotal role in many applications, including e-commerce, confidential government communications, password/PIN management and secure file storage.

OpenTitan can be used to implement TPM 2.0 for both clients and servers. Because trusted platform modules generate their own cryptographic keys, an internal entropy source is required. OTBN is equipped with a hardware random number generator, connected to its entropy distribution network. The algorithms for which these random keys are generated are RSA and ECDSA. OpenTitan uses these algorithms for digital signature generation and securing data during communication with other devices. Both of these algorithms will eventually be phased out in favor of their quantum-secure counterparts, Dilthium and Kyber. In the cases of secure file storage and disk encryption, symmetric key encryption (AES) is used and therefore this feature may persist into the quantum era.

2.4.3 Universal 2nd-Factor Security Key

When used as a Universal 2nd-Factor Security Key, OpenTitan operates under the U2F authentication standards developed by the FIDO Alliance (v1.2). These standards outline a two-factor authentication protocol, whereby a physical security key is required in addition to a password. Identity-based services, such as online banking, confidential communication platforms like email and cloud computing can benefit significantly from the security afforded by the use of U2F authentication.

The stages of U2F authentication are as follows:

1. **Login attempt.** The user attempts to login to an account by entering their username and password. Upon initial association of the U2F security key with the service, a public/private key pair is generated. The key pair is generated for either RSA or ECDSA, using OTBN's entropy distribution network as a source of randomness. The private key remains securely stored and is never exposed externally.

2. **Cryptographic Challenge.** Once the user's credentials have been verified by the service, the service generates a cryptographic challenge. The user is required to connect their U2F security key as a peripheral to receive this challenge, using a USB port in the case of OpenTitan.

3. **Completion of the Challenge.** The user is required to physically interact with their U2F security key to prove their presence, e.g. by pressing a button. The challenge is a value which must be signed by the private key of the U2F and returned to the service. This process is completed by generating a digital signature using ECDSA on OTBN.

4. **Verification.** The service verifies the integrity of the received signed challenge using the public key associated with the user's U2F. ECDSA is used for verification.

The primary cryptographic operations in this use case are performed by digital signature algorithms, since the purpose of the application is to enable secure authentication procedures for users' interactions with services. In a post-quantum context, the most applicable algorithm will likely be Dilithium, acting as a quantum-secure substitute for ECDSA. We anticipate that our techniques will translate almost seamlessly to Dilithium due to the implementation commonalities it shares with Kyber.

In summary, the primary use cases of OpenTitan are intrinsically associated with processes such as secure boot, attestation, authentication and integrity verification. Because OpenTitan is intended for use in IoT devices, secure communication of data between such devices is also an important requirement of its cryptographic co-processor, OTBN. Although it currently offers support for efficient execution of PKC algorithms, these algorithms will likely be replaced by quantum-secure standards in the future. Both digital signature algorithms such as Dilithium and key encapsulation mechanisms such as Kyber have the potential to play a prominent role in post-quantum use cases of OpenTitan.

Chapter 3

Design and implementation

3.1 Development environment (OTBNSim)

During the development phase of our project, OTBN was still being taped out as an engineering sample and therefore the physical chip was unavailable. A Python simulator for OTBN, OTBNSim [33], is available as part of the OpenTitan repository and this was used for development. OTBNSim comprehensively simulates the OTBN block of OpenTitan, accurately modelling the state of the core at each step of the simulation. It emulates the behaviour of internal state components using abstract models of the registers, DMEM and IMEM. By updating the state abstractions in a stepwise manner, OTBNSim facilitates detailed inspection and validation of instructions and algorithms. It is closely aligned with the SystemVerilog model of the OTBN module, which ensures its conformance to the device verification documentation. OTBNSim includes mechanisms for error injection, error detection and initiation of a secure wipe of the internal state in response to detected errors. These capabilities allow it to accurately replicate the security features which characterise OTBN as a suitable platform for cryptographic processing. By offering a layer of abstraction from the hardware, OTBNSim provides an efficient framework for development and validation of cryptographic implementations for OTBN.

Instructions are defined as subclasses of the `OTBNInsn` class in Python. They can be found within the `otbn/dv/otbnsim/sim/insn.py` file. Tests for each instruction are written in Python and stored within the `otbn/dv/otbnsim/test/` directory. The `execute()` method defines each instruction's behaviour. Instructions are also declared in the file: `otbn/data/bignum-insns.yml`, including mnemonics, operands, syntax and encoding.

OTBNSim processes tasks sequentially and can respond to various external inputs during execution. Successful execution of a task progresses through the following stages:

1. **Decoding of the program.** The encoded instructions are fetched from the IMEM. Each instruction is then decoded to extract the opcode. The opcode uniquely identifies the operation within the instruction set to be executed by the processor.

2. **Loading of the program.** The decoded program (sequence of decoded instructions) is then loaded into dedicated local storage within the simulator.

3. **Stepwise simulation of execution.** Within the simulated SystemVerilog environment, each execution step triggers updates to one or more of the following: the state of the core, the registers and the memory. The relevant abstractions are updated by modifying the values within the corresponding Python data structures.

4. **Generating traces**. Following each execution step, a trace is generated, which provides a detailed description of the operations performed by the processor during that step and the effects on the processor's state. The traces are eventually forwarded to the OTBNTraceChecker, a tool designed to examine them for correctness and alignment with expected instruction behaviour.

OTBNSim incorporates a cycle-accurate performance reporting mechanism for all existing OTBN instructions, which execute in either one or two CPU cycles. This is based on cycle count measurements taken from a prototype chip. As we did not have access to a prototype chip during development, when designing new instructions, it was necessary to consider the required hardware modifications to accurately estimate cycle counts. The instructions we propose are designed to maximally leverage OTBN's existing hardware components and require only minor modifications. This not only facilitates a realistic and feasible solution which enhances performance, but ensures that cycle count estimates are tightly aligned with the ground-truth performance of existing instructions.

A compiler is not yet available for OTBN, so all development involved coding directly in OTBN assembly. OTBNSim is capable of directly processing `.s` assembly files, where the instruction sequence is defined in the `.text` section and constants to be stored in memory are defined in the `.data` section. At the end of the assembly routine, the `ecall` (environment call) command is used to trigger the `done` interrupt, which signals to the simulator that the execution is complete. Subsequently, the final state values are reported in the console. In the event of a runtime error, OTBNSim halts execution. However, the Python code which orchestrates the execution by default within the simulator does not explicitly handle different error types. Given that the repository provides simple instruction tests, this functionality is not typically required. However, for complex assembly routines in which a large number of instructions are called, such as the ones we developed, an accurate mechanism for error tracing is essential for debugging purposes. We inspected the internal error handling process, to discover that different errors have individual identifier codes, stored in a dedicated register called `ERR_BITS`. A Python enum is used to map these values to their error names within the simulator. We identified a script in the repository, external to the OTBNSim directory (`otbn/util/otbn_sim_test.py`), which runs a simulation and includes functionality to identify errors by accessing this enum and recording the instruction number at which the error occurred. We incorporated this enhanced error tracing feature into the OTBNSim framework to assist with debugging.

3.2 Testing Infrastructure

To ensure the correctness of our algorithmic implementations, we extended the testing infrastructure within the simulator to support more rigorous testing. The repository contains a simple testing framework for straightforward instructions, such as addition. For each test case, two files with the same base name are defined. The input assembly (`.s`) file defines the assembly routine to be executed, and the corresponding `.exp` file contains the expected values of registers upon completion. The `standalone.py` script is used to run the simulation. It operates by assembling and linking the given assembly file. The assembling process constitutes the translation of assembly code into object code. This is then linked to create an executable ELF file, using a custom linker script in `otbn_ld.py` which ensures the correct placement of the code and data in memory. The simulator is run via a subprocess call, with a command-line argument (`--dump-regs`) set which specifies that following execution, the values of GPRs (general-purpose registers) and WDRs (wide data registers) should be written to an output file. The `.exp` file is then parsed to extract the expected values, against which the actual values are compared. Any discrepancies are displayed in the console.

For each test, input values and expected output values are manually hard-coded into the `.s` and `.exp` files. This approach is acceptable for the purposes of testing simple cases on the simulator. However, in the context of extensively testing the cryptographic implementations we developed, it lacks the required flexibility to thoroughly test a wide range of inputs. Therefore, we created a Python script to automate the process of input/output file generation for arbitrary lists of input values. The input values are defined as a list within the Python script and the corresponding output values are computed by a prototype function of the assembly code being tested. For each assembly routine, we created a subdirectory, within the same directory as the script. Within each subdirectory, a file called `template.s` contains the code for the assembly routine. A file called `template.exp` defines the expected final register values. Within these files, placeholders are used to identify where the input and output values should be inserted before execution.

When the test script is run, a new `inputoutput` subdirectory is created. For every input, the expected output is computed by the function prototype. A copy of `template.s` and `template.exp` is made and regular expression matching in Python is used to replace the placeholders with the actual values. We used a character sequence which was not present anywhere else in the assembly code as a placeholder (e.g. "`[inp1]`") to ensure against false positives in the matching process. This pair of files is then assigned the base name of `template_inp`, where `inp` represents the number supplied as input to the given test case, and added to the `inputoutput` subdirectory. Once all the files have been created, the existing testing infrastructure processes the corresponding file pairs in `inputoutput/`. We developed different versions of this script for different aspects of the implementation, e.g., for testing the new instructions individually and for the function implementations as they were being developed. Our extension was easily integrable into the existing test suite due to its use of the `pytest` framework. Separation of different test cases facilitated straightforward enabling and disabling of tests, which streamlined the testing process.

Since the simulator is Python-based, we began by writing the function prototypes in Python. This approach proved successful for simple assembly subroutines and instruction-level tests. For vectorised instructions, we simply repeated the functionality a number of times equivalent to the number of lanes, shifted the results to their correct lane indices and concatenated them. However, we encountered issues with this approach as our implementations became more complex. We noted a divergence in the behaviour of the Python and the reference code, which we traced back to the differences between Python and C in the handling of integer operations and type casting. The reference implementation of Kyber, which we aimed to replicate precisely, is in C. In C, integers have a fixed size and may be signed or unsigned. In contrast, in Python, integers are inherently signed, arbitrarily large and their representation does not correspond to C's precise type system. When attempting to replicate the integer operations of the C reference implementation in Python, issues arose regarding sign interpretation and maintaining fixed-size integer variables, particularly following operations which caused overflow. This would have required a large amount of additional processing for integer manipulation in Python. We attempted to implement this, but found that it confounded the core functionality of the implementation and made error tracing complex. It also led to various bugs not being identified promptly, necessitating a more accurate and efficient approach.

We identified `ctypes` as a means of incorporating the C reference implementation directly into the testing framework. Using this library, foreign functions, i.e. functions written in other programming languages, can be invoked from Python. We imported the relevant C and header files from the GitHub repository of Kyber into the OTBNSim environment. After compiling the C code into an object file using `gcc`, we created a shared library (`.so` file). This allowed us to wrap the required C functions (`ntt()` and `invntt()`) in pure Python. This facilitated incremental development, whereby we constructed the assembly code in a stepwise manner and ensured its correctness after each step. We committed each incremental development to GitHub once fully functional.

The (I)NTT implementations in Kyber perform in-place transformations on a 256-element array. Throughout development, we defined this array as random but constant. However, to thoroughly test the implementation to a cryptographic standard, it was necessary to ensure that the code computed the correct values for arbitrary 256-element arrays. A standard method of testing cryptographic implementations for correctness is by using NIST vector tests. Within the Kyber reference implementation, there exists a mechanism for conducting these tests, whereby the executable `test_vectors$ALG` generates 10,000 sets of test data for the security level specified by `$ALG` (512, 768 or 1024). The sets contain keys, ciphertexts and shared secrets, which are values that both sender and receiver should be able to compute independently if the encryption and decryption processes are correct.

Given that we implemented the `ntt()` and `invntt()` functions in isolation, as a full Kyber implementation in OTBN assembly was not available, it was necessary to extract the array values upon entering and exiting these functions during the vector tests. We achieved this by modifying the reference code to write the contents of the array before and after execution of `ntt()` and `invntt()` to text files. Separate text files were maintained for the inputs and outputs of the two functions, where entries of the same index in the input and output files corresponded to the same test case. Array elements were separated by a different delimiter to that which separated the arrays. This enabled Python parsing of the text files to successfully read the values with which to overwrite the placeholders in the `.s` and `.exp` files. This served as a black-box testing method once the implementation was complete, as we replaced the computation of the output values by a prototype function with the ground-truth values obtained by separately running the NIST vector tests.

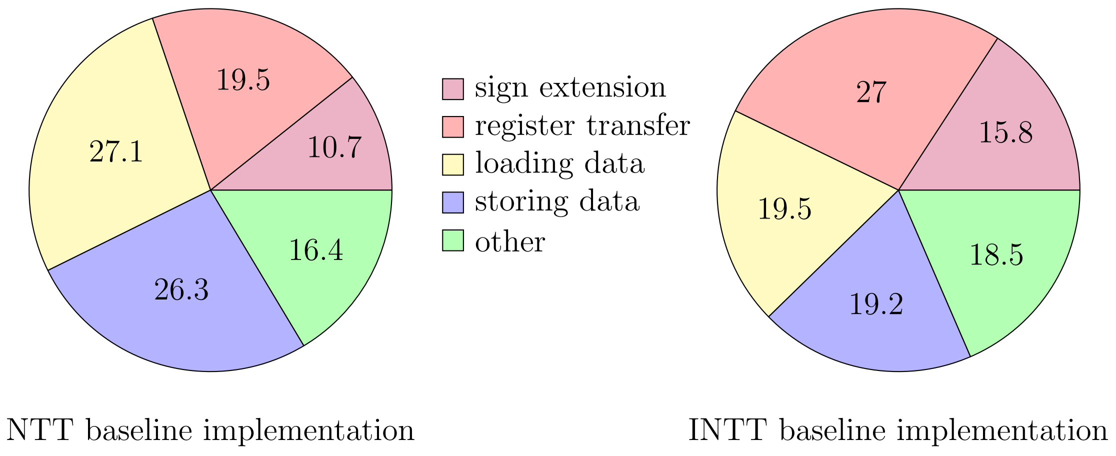
3.3 Baseline Implementations

We first developed baseline implementations of the NTT and INTT by translating the functions in the Kyber reference implementation as directly as possible to OTBN assembly, using only the existing instructions. This process established the current performance of OTBN in execution of these functions and provided baseline performance benchmarks against which to compare optimisation efforts. It should be noted that the Kyber reference implementation has not been optimised for any platform. However, the reduced instruction set of OTBN constrains the potential for optimisation without implementing instruction set extensions, so the baseline performance provides a reasonable estimate of its capabilities. Analysis of the baseline implementations on OTBN granted insights into performance bottlenecks and particularly inefficient operations, hence serving to motivate optimisations and inform the design of instruction set extensions. In Figure 3.1, we present our analysis of the baseline implementations of NTT and INTT on OTBN. Performance bottlenecks are analysed in terms of the percentage of overall cycle count.

The baseline implementation of NTT required 91,939 CPU cycles to execute, while the INTT required 149,435 cycles. During the implementation process, the most significant performance impediment we noticed was the restriction to scalar computations on the wide data registers (WDRs) of 256 bits. These registers were designed to perform arithmetic on large integers; however, in the case of `ntt`, the integers involved in computation, including intermediate results, do not exceed 32 bits and hence the large register capacity is not utilised. The computational effort spent on these operations is not being exploited to its full potential and the same effects could be achieved by operating on much smaller units. We noted that vectorisation of operations could be maximised to enable full use of WDRs.

Although the majority of instructions in both subsets execute in a single clock cycle, in the

Figure 3.1: % execution time spent on functionality types in baseline implementations



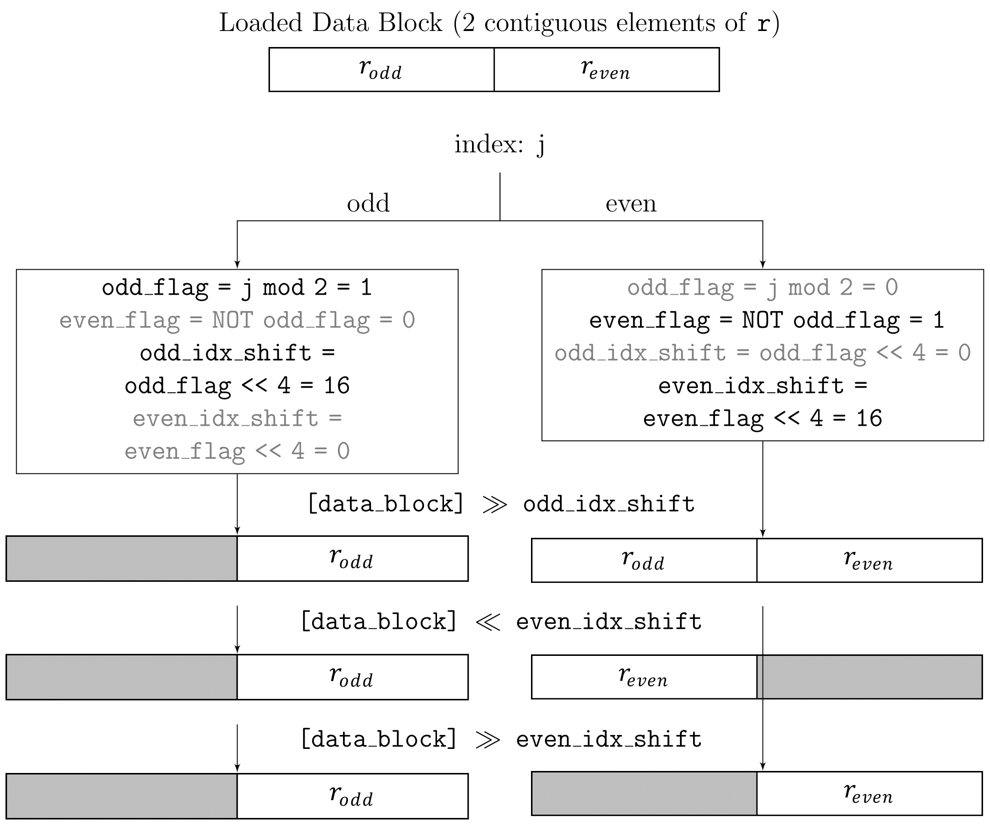
context of the Big Number subset, additional operations are required to perform certain computations. OTBN only supports unsigned arithmetic and the (I)NTT function operates on signed values. Two's complement is used to represent negative numbers. Although the values involved in multiplication operations are 16 bits in width, it was necessary to sign-extend these numbers to 64 bits before using `BN.MULQACC`, as it operates on 64-bit operands. This was required to be done manually if this data had been transferred from GPRs. In the interest of maintaining a constant-time implementation, we adopted the following approach. The sign bit is isolated through a right shift of 15 bits and multiplied by a 64-bit mask with the upper 48 bits set. The result is then XORed with the original 16-bit value, resulting in sign-extension to 64 bits. This process costs 4 cycles for each of two operands, before a multiplication can be performed. Sign extension required 10.7% and 15.8% of the total cycle count of the `ntt` and `intt` baselines, respectively (3.1).

In our efforts to design methods to enhance the efficiency of sign extension, we identified two viable strategies. The first involved reducing the operand size in a vectorised implementation. For example, if lanes were 16 bits in length, this would align with the operand width and therefore sign extension beyond this width would not be necessary. In designing instruction set extensions, we are granted flexibility at a hardware level. This led us to propose offloading of sign interpretation to hardware in the case of the INTT where vectorised arithmetic right shifts are required.

Another notable bottleneck was the requirement to transfer data between the register types to perform different operations. Certain instructions, such as multiplication, are only available in the Big Number subset. Conversely, other operations, such as left shift, are only available in the base subset. Given that the Big Number subset is designated for data flow, transferring data back and forth between WDRs and GPRs during the main computation is both inefficient and does not conform to standard practice. However, at certain points in the implementation this is necessary, for example, loading array values with fine granularity at data addresses and performing a left shift during the Barrett reduction phase of INTT. A resulting objective of our design of new instructions was to minimise the requirement for data transfer between register types. Transferring of data between register types cost 19.5% and 27% of the cycle count of the `ntt` and `intt` baselines, respectively (3.1).

The array `r`, which is processed by the NTT function, consists of 256 16-bit elements which are stored contiguously in memory. Each element is processed individually and reads from memory to GPRs can only be performed on 32-bit-aligned boundaries. Therefore, data can only be loaded in fixed 32-bit blocks. This complexified the element loading and storage procedures. In order to load and operate on `r[j]`, we floor divide the index `j` by 2 by performing a right shift by 1 bit, in order to identify the index of the 32-bit block containing `r[j]`. We then shift the result right by 2 to compute the byte offset from the base address of `r` from which to load the block. We determine whether index `j` is even or odd by computing `j AND 1`. In the case of an even index, we isolate `r[j]` via an `AND` of the loaded block with a 16-bit mask. In the case of an odd index, we shift the loaded block right by 16 bits. However, this approach contains a conditional statement, which may lead to violation of constant-time properties. In the case of development of the baseline and optimised implementations, we avoided the use of conditional statements in order to retain constant-time properties.

Therefore, we require a single execution path for loads and stores of odd- and even-indexed values. To load and isolate an individual array value (of either odd or even index) in a GPR, we follow the process outlined in Figure 3.3. We begin by loading a data block containing 2 contiguous array elements following the previous procedure, one of which is at the required index `j`. We compute `j AND 1` and its inverse. We shift both values left by 4 so the non-zero remainder represents 16. We then shift the loaded block right by the former value (`(j AND 1) << 16`), shift left by the same value and finally shift right by the latter value (`(NOT(j AND 1) << 16`). This method isolates the required element in the least significant position in the case of both odd and even indices, enabling subsequent computations. For storing the result, we only overwrite one element of `r`, leaving the other 16 bits of the 32-bit block unchanged. The opposite 16-bit value in the block is isolated in a similar way to `r[j]`. Before the final block is stored to memory, the two 16-bit components are shifted back to their original position and combined with an XOR. We noted the evidently large computational overhead introduced by replicating the elementwise loading procedure of the reference implementation. In the `ntt` and `intt` baselines, respectively (3.1), loading of values into GPRs (including subsequent manipulation of loaded values) cost 27.1% and 19.5% of the total cycles. Meanwhile, constructing the resulting data blocks and storing them to memory cost 26.3% and 19.2% of the total cycles, respectively. We aimed to reduce the number of load and store operations, eliminate manipulation of loaded values and facilitate parallel computation on loaded values directly.



Loaded Data Block (2 contiguous elements of r)
r_{odd}
r_{even}
index: j
odd
even
odd_flag = j mod 2 = 1
even_flag = NOT odd_flag = 0
odd_idx_shift = odd_flag << 4 = 16
even_idx_shift = even_flag << 4 = 0
odd_flag = j mod 2 = 0
even_flag = NOT odd_flag = 1
odd_idx_shift = odd_flag << 4 = 0
even_idx_shift = even_flag << 4 = 16
[data_block] ≫ odd_idx_shift
r_{odd}
r_{odd}
r_{even}
[data_block] ≪ even_idx_shift
r_{odd}
r_{even}
[data_block] ≫ even_idx_shift
r_{odd}
r_{even}

Figure 3.2: Loading and isolating data elements in GPRs

3.4 Vectorised Implementation Design

We designed the vectorised implementations of the `ntt()` and `invntt()` functions, outlined in pseudocode in Appendix A, with the aim of maximising parallelism, minimising load/store overhead, minimising data transfer between register types and more efficiently handling signed multiplication. We designed an implementation which maximises the vectorisation potential of OTBN and enables full usage of the capacity of the WDRs. To implement it, we designed new instruction set extensions which complement the existing capabilites of OTBN and incorporate some moderate hardware modifications, which could realistically be introduced to the platform. The optimised implementation loads and stores 16 polynomial coefficients at a time, significantly reducing the load/store overhead of the baseline. Array elements are operated on in-place using vectorised instructions, which enables us to avoid manipulation of loaded values. Explicit sign extension is no longer required due to the narrower lane widths and offloading of the sign interpretation to hardware in the case of a vectorised arithmetic right shift.

The values of `zeta` are broadcasted at 32-bit intervals across WDRs before entering the `fqmul` computation. Because the broadcast instruction replicates the value in a GPR across all lanes of a WDR, we must load `zeta` from memory directly into a GPR. However, we aimed to minimise the overhead of loading and isolating elements. We achieved this by unrolling two iterations of each loop which required a new value of `zeta` to be assigned. This enabled us to load two values at once, isolate them and retain the second one in a separate GPR instead of performing a second load operation. In the baseline implementation, loading and isolating two values of `zeta` cost 26 cycles, whereas this approach costs only 6 cycles.

The implementation is split into two parts: the first deals with values of `len` which are multiples of 16. This means that the number of elements between `r[j]` and `r[j+len]` can be stored in a distinct number of wide data registers. Therefore the new values of `r[j]` and `r[j+len]` can be separately computed and written to memory in batches of 16 elements. Within the `fqmul` function, intermediate values can occupy up to 32 bits. At this point, the lane widths are effectively expanded from 16 to 32 bits. This process is illustrated in Figure 3.4, where shaded sections of mask registers represent all 0s and non-shaded sections all 1s. Note that shifts applied are vectorised on 32-bit lanes. The lower and upper 16-bit elements in each 32-bit lane of the wide data register are extracted into two separate registers. We isolate the elements of even index by pre-loading a 256-bit mask with every even-indexed 16-bit element set and performing an `AND` operation between this register and the loaded values. Then, to isolate the values at odd indices, a vectorised right shift by 16 bits of each 32-bit lane is used to place them in the lower positions. The `fqmul` operations can then proceed in the same way for both vectors. The results of the two `fqmul` computations are then combined by reversing the shift and performing an XOR between the two registers.

The second part of the implementation deals with values of `len` which are factors of 16. The number of elements between `r[j]` and `r[j+len]` is less than the capacity of a wide data register. As data elements are loaded contiguously, computations of the new values of `r[j]` and `r[j+len]` are combined within registers. Iteration levels are merged to maximise computational capacity. This part of the implementation has one less nested loop than the first, merging the loading of zetas into the innermost loop. `8/len` zetas are loaded into a single register, occupying equal proportions. Each iteration reads 16 consecutive elements as a vector of `r[j]`. Since loads to wide data registers are only permitted at 256-bit boundaries, OTBN's 512-bit barrel shifter, which produces a 256-bit output, is used to load `r[j+len]` at the required level of finer granularity. The subsequent block of 16 elements is then loaded, concatenated with the previous and shifted right by `len` elements, returning the low 256 bits as the corresponding vector of `r[j+len]`. This construction of the vectors of zetas, `r[j]` and `r[j+len]` allows the rest of the computation to proceed in the same way as the first part of the implementation. Once `fqmul` has been computed however, the computations of the new values of `r[j]` and `r[j+len]` must be combined within the same resulting register. This is achieved using bitmasks and shifting to interleave the calculated values at offsets of length `len` within the register. Due to the direct operations on data in WDRs throughout the computation, the overhead of transferring data between register types in NTT and INTT is reduced to zero from cycle counts of 17,918 and 40,320 in the respective baselines (3.1).

Throughout the vectorised computation, elements are fully packed into the WDRs. The combination of the computation of new values of `r[j]` and `r[j+len]` within the same registers ensures that this potential remains maximised even for values of `len` which are less than the element capacity a WDR. The expansion of lane widths from 16 to 32 bits is implemented for the least possible number of instructions. Once 32-bit precision is no longer required for intermediate computations, the implementation transitions back to the initial mode of operation on 16 elements in parallel. Minimisation of load/store overhead is achieved, as the optimised implementation loads and stores 16 elements at once. Additional pre-processing of loaded array values is eliminated as all loaded values are operated on directly in the positions within the register at which they were loaded. As shown in Table 3.1, the number of cycles spent on the loading of values and manipulation of loaded values was reduced from 24,947 to 1,196 cycles for NTT and from 29,171 to 1,273 for INTT, over the baselines (3.1). Similarly, the cycles required for constructing and storing data blocks to memory was reduced from 24,192 to 592 for NTT and 28,672 to 672 for INTT. Loading of `zeta` values into GPRs before broadcasting is optimised by isolating and storing the two values that are loaded at once from memory. The multiplication process has been streamlined to eliminate the requirement for explicit sign extension in software. In the baseline implementations of INTT and NTT, respectively, 9,854 and 23,680 cycles were spent on sign extension, however this costs no additional cycles in the optimised implementations.

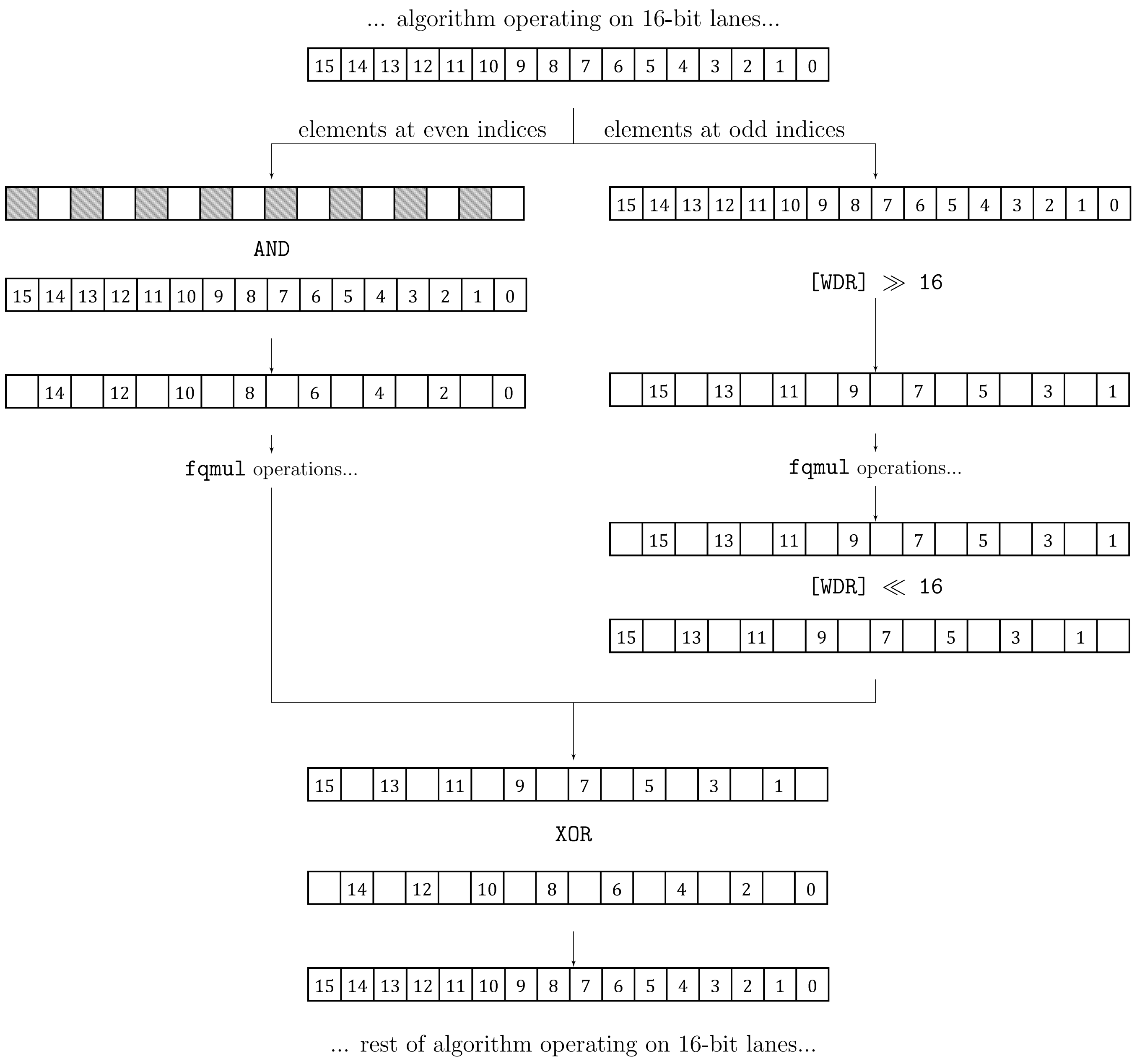


Figure 3.3: Loading and isolating data elements into GPRs

Table 3.1: Comparison of Cycle Count Distribution Between Implementations

Implementation	**Sign Extension**	**Register transfer**	**Load**	**Store**	**Other**	**Total**
`ntt_baseline`	9854	17918	24947	24192	15028	91939
`ntt_optimised`	0	0	1196	592	2568	4356
`invntt_baseline`	23680	40320	29171	28672	27592	149435
`invntt_optimised`	0	0	1273	672	4188	6133

3.5 Instruction Set Extensions for OTBN

In this section, we propose 9 new instructions for OTBN. We implemented these within the Python simulator by defining them in the file: `hw/ip/otbn/dv/otbnsim/ sim/insn.py`. We leveraged the new instructions in our vectorised implementations of `ntt()` and `invntt()`. Although they have not been implemented in hardware, OTBNSim granted us the ability to define and implement these instructions within the simulation environment for the purposes of demonstrating the correctness, efficiency and viability of the optimised implementations which are enabled by these instructions.

An important consideration when designing hardware modifications for PQC is that it is a rapidly-evolving field and the standardisation process is ongoing. Although PQC is poised for mainstream adoption, it is a dynamic field of research that will inevitably experience many changes. Therefore, developing extremely specialised and targeted hardware prematurely may result in technology which will rapidly become obsolete. The instructions we propose enable maximisation of the vectorisation potential of OTBN in the `ntt()` and `invntt()` computations, yet are not exclusive to these use cases. Rather, they are generic and could be leveraged in the implementation of many other algorithmic components.

Our design strategy focused on proposing realistic extensions to OTBN, which could be feasibly incorporated into the chip without extensive hardware modifications or addition of an entirely separate data path. We aimed to minimise the percentage increase in chip area that would be required by our instructions by aligning them as closely as possible with OTBN's existing hardware components. However, to precisely quantify this increase, we would need access to the physical hardware and would ideally consult with experts at lowRISC. We were careful to avoid proposing any drastic changes to the architecture of OTBN. This approach conforms to OTBN's intentionally minimalist design, which facilitates security countermeasures by minimising the attack surface. Therefore, we aimed to leverage OTBN's existing capabilities by re-using and re-purposing its underlying hardware features where possible. We carefully considered the required hardware modifications and aimed to minimise these, whilst still exploiting the speed of hardware for particularly inefficient operations in software on OTBN. This approach was important from the perspective of security, but also performance estimation. OTBNSim is a cycle-accurate model of OTBN. The cycle count of each existing OTBN instruction is fully accurate. However, OTBNSim does not have a mechanism for measuring the cycle count of new instructions. Therefore, we were required to estimate the cycle counts of our new instructions as accurately as possible, using inference from the ground-truth values wherever applicable.

The rest of this section describes the new instruction set extensions, the reasoning used to estimate their cycle counts, details of how they leverage OTBN's existing hardware components and descriptions of any hardware modifications required.

BN.LSHI (*Concatenate and left shift*)

Concatenates the contents of the WDRs `<wrs1>` and `<wrs2>`, with `<wrs1>` forming the upper part and shifts left by an immediate value of up to 255 bits. Writes the uppermost 256 bits of the resulting value to the destination WDR `<wrd>`. The equivalent instruction for performing a right shift already exists in OTBN (`BN.RSHI`), where the final result is instead truncated to return the lowest 256 bits.

Syntax:	`BN.LSHI <wrd>, <wrs1>, <wrs2> << imm`
`wrd`	Destination WDR.
`wrs1`	Source WDR which will form the upper part of the 512-bit value to be shifted.
`wrs2`	Source WDR which will form the lower part of the 512-bit value to be shifted.
`imm`	Immediate value specifying the number of bits by which to shift (range: 0 to 255).

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: This instruction essentially operates in the same way as the existing `BN.RSHI` instruction for OTBN which executes in one cycle, but performs a left shift instead of right. Therefore it is reasonable to infer that a barrel shifter can operate in both directions within the same window.

Hardware Modifications Required: OTBN's current 512-bit barrel shifter only supports right shift operations. Therefore, the same hardware logic would need to be added, but configured to shift in the opposite direction.

BN.MULVEC (*Vectorised multiplication of low 16 bits of 32-bit lanes*)

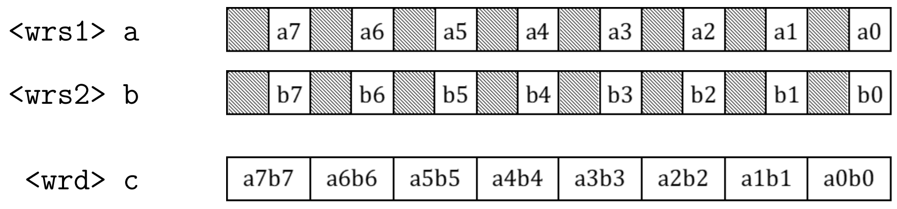
This instruction performs a vectorised multiplication on source WDRs `<wrs1>` and `<wrs2>`. It multiplies the values represented by the low 16 bits of each 32-bit lane and writes the resulting value (up to 32 bits) to the corresponding lane of the destination WDR (`<wrd>`). Its operation is illustrated in Figure 3.4.

Syntax:	`BN.MULVEC <wrd>, <wrs1>, <wrs2>`
`wrd`	Destination WDR.
`wrs1`	First source WDR.
`wrs2`	First source WDR.

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: OTBN currently has a 64-bit multiplier which acts on WDRs via the `BN.MULQACC` instruction and has a latency of 1 cycle. This instruction would require eight 32-bit multiplications to be performed in parallel. The combinatorial path of 32-bit multiplication is much shorter than that of the 64-bit multiplication which

Figure 3.4: Operation of `BN.MULVEC c, a, b`



is already supported. Since these multiplications would execute in parallel, the critical path through this part of the instruction should be shorter than that of `BN.MULQACC`. Given that this instruction acts on the low 16 bits of every 32-bit lane and produces a 32-bit result, it is similar to the operation of `BN.MULQACC`, which acts on the low 64 bits of source WDRs to produce a 128-bit output. Isolation of the low 16 bits of each lane could be done using a similar masking mechanism which is already used to isolate the low 64 bits of a WDR. Isolating values across multiple lanes would not incur any additional overhead, as the `AND` operation on source WDRs is applicable, using a different mask. Therefore, this instruction could realistically execute within a single cycle.

Hardware Modifications Required:

- The masking mechanism which is used to isolate the low 64 bits of the source WDRs would need to be modified to instead isolate the low 16 bits of every 32-bit lane.
- The 64-bit multiplier units would need to be segmented and the data paths to and from these units modified to process 16-bit elements separately.
- The 64-bit multiplier units would need to be reconfigured to operate independently on 16-bit segments in 32-bit lanes. This may involve replicating parts of the circuitry to enable parallel operations and reducing the size of the operands.

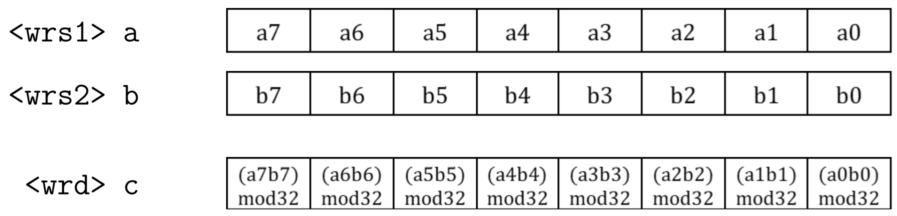
BN.MULVEC32 (*Vectorised multiplication of 32-bit lanes*)
This instruction performs a vectorised multiplication on source WDRs `<wrs1>` and `<wrs2>`. It multiplies the full 32-bit values in each 32-bit lane and truncates the resulting value to 32 bits before writing it to the corresponding lane of the destination WDR (`<wrd>`). Its operation is illustrated in Figure 3.5.

Syntax:	`BN.MULVEC32 <wrd>, <wrs1>, <wrs2>`
`wrd`	Destination WDR.
`wrs1`	First source WDR.
`wrs2`	First source WDR.

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: A similar justification of the cycle count estimate is relevant to this instruction as to the `BN.MULVEC` instruction. The main difference between

Figure 3.5: Operation of `BN.MULVEC32 c, a, b`



them is that this instruction operates on the full 32 bits of each lane. The product of two 32-bit values can occupy up to 64 bits and therefore the result needs to be truncated to 32 bits before being written to the destination register. Therefore, instead of applying a masking operation to isolate the low 16-bit values of each lane before multiplication, as is the case for `BN.MULVEC`, masking is required to truncate the result to 32 bits. The `AND` operation is effectively moved within the internal path of this instruction, which should not introduce additional overhead.

Hardware Modifications Required:

- The masking mechanism which is used to isolate the low 64 bits of the source WDRs would need to be modified to instead truncate the 64-bit values resulting from multiplication to 32-bit values.
- The 64-bit multiplier units would need to be segmented and the data paths to and from these units modified to process 32-bit elements separately.
- The 64-bit multiplier units would need to be reconfigured to operate independently on 32-bit segments. Some parts of the circuitry would need to be replicated to enable parallel operations on smaller operands.

BN.ADDVEC (*Vectorised addition of 16-bit lanes*)

This instruction performs a vectorised addition on source WDRs `<wrs1>` and `<wrs2>`. It adds the 16-bit values in each 16-bit lane and truncates the resulting value to 16 bits before writing it to the corresponding lane of the destination WDR (`<wrd>`).

Syntax:	`BN.ADDVEC <wrd>, <wrs1>, <wrs2>`
`wrd`	Destination WDR.
`wrs1`	First source WDR.
`wrs2`	First source WDR.

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: OTBN is already equipped with a 256-bit adder, which operates on the full 256 bits of WDRs via the `BN.ADD` instruction and executes within a single cycle. `BN.ADD` then truncates the result to 256 bits and writes it to the destination register. This instruction entails 16 parallel additions of 16-bit values. The combinatorial path of 16-bit additions is much smaller than that of a 256-bit addition. Therefore, given that these smaller additions will execute simultaneously, the critical path of this instruction is significantly shorter than that of the existing `BN.ADD` instruction. As a result, it should also execute within a single cycle.

Hardware Modifications Required:

- The 256-bit adder would need to be partitioned into 16 addition units which each operate on 16-bit values and produce 16-bit results.
- To ensure that results from additions in each lane do not overflow their 16-bit bounds, the carry logic within the adder would need to be modified. It should be updated to define boundaries for carry bits in each 16-bit lane, ensuring that they do not spill over into their neighbouring lanes of higher significance.

BN.SUBVEC (*Vectorised subtraction of 16-bit lanes*)

This instruction performs a vectorised subtraction on source WDRs `<wrs1>` and `<wrs2>`. It subtracts the 16-bit values in each 16-bit lane and truncates the resulting value to 16 bits before writing it to the corresponding lane of the destination WDR (`<wrd>`).

Syntax:	`BN.SUBVEC <wrd>, <wrs1>, <wrs2>`
`wrd`	Destination WDR.
`wrs1`	First source WDR.
`wrs2`	First source WDR.

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: This instruction will use OTBN's 256-bit adder, similar to `BN.ADDVEC`. The same argument can be used to justify the cycle count estimate for this instruction as for `BN.SUBVEC`, using the existing OTBN instruction `BN.SUB` in place of `BN.ADD`.

Hardware Modifications Required:

- The same partitioning of the 256-bit adder which was required by `BN.ADDVEC` would be required.
- In the case of the vectorised subtraction, handling of the borrow flag would need to be managed. To ensure that parallel subtractions do not borrow from values in their neighbouring lanes, the borrow logic within the adder would need to be modified. It should define boundaries for borrow bits at the most significant end of each 16-bit lane.

BN.LSHIFTVEC, BN.RSHIFTVEC, BN.ARSHIFTVEC (*Logical and arithmetic vectorised shift operations on 32-bit lanes*)
BN.LSHIFTVEC and BN.RSHIFTVEC perform vectorised logical right/left shift operations on each 32-bit lane of the source WDR `<wrs1>` of up to 31 bits, writing the resulting value to the corresponding lane of the destination WDR (`<wrd>`). BN.ARSHIFTVEC operates in the same way but performs a vectorised arithmetic right shift.

Syntax:	`BN.(L/(A)R)SHIFTVEC <wrd>, <wrs1>, <imm>`
`wrd`	Destination WDR.
`wrs1`	Source WDR.
`imm`	Immediate value specifying the number of bits by which to shift the contents of each lane.

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: OTBN features a 512-bit barrel shifter that produces a 256-bit output within one cycle. This is more complex than a 256-bit barrel shifter, which would be required for these instructions. Vectorised shifts would be performed across 256-bit registers, in parallel lanes of 32 bits. The combinatorial path of each of these shifts would be much shorter than that of a full 256-bit shift as computations within each lane can be computed independently. Therefore it should execute within one cycle. Following implementation of `BN.LSHI`, OTBN will support shifts on WDRs in both directions. The arithmetic right shift operation inserts a copy of the previous most significant bit into vacant bit positions instead of zero. The value of this bit can be extracted using a multiplexer in each lane, which is straightforward to implement in hardware and should not impact the cycle count estimate.

Hardware Modifications Required:

- Segmentation of the low 256 bits of the barrel shifter into eight 32-bit lanes.
- Reconfiguration of the data flow path to and from this unit to operate on separate values in parallel.
- Implementation of control logic, e.g. using masking, to isolate the shifts within each lane. As bits are shifted, masks or logical operations can be applied to each lane to make sure that they do not run over into adjacent lanes.
- Addition of multiplexers to extract the value of the most significant bit in each lane before performing an arithmetic right shift.

BN.BROADCAST (*Broadcast 32-bit value across 32-bit lanes of WDR*)
This instruction replicates the value in the source GPR `<grs1>` across each 32-bit lane of the destination WDR (`<wrd>`).

Syntax:	`BN.(L/R)SHIFTVEC <wrd>, <wrs1>, <imm>`
`wrd`	Destination WDR.
`wrs1`	Source WDR.
`wrs2`	Immediate value specifying the number of bits by which to shift the contents of each lane.

Cycle Count Estimate: 1 cycle.

Justification of Cycle Count Estimate: OTBN does not currently support any instructions from which the cycle count for `BN.BROADCAST` can be directly derived, however, broadcasting is a straightforward operation in hardware. OTBN features a GPR selector multiplexer, used for instructions which operate on source GPRs in the base instruction subset. The GPR value should be replicated 8 times across the destination WDR, which would require a simple fanout of wires from the output of the GPR selector multiplexer. Reproducing a signal at multiple destinations is primarily handled by wiring. Physical design tools may insert buffers to maintain signal integrity by preventing attenuation and interference between signals. This is standard practice and does not introduce much complexity. The existing WDR input multiplexer would need to be extended by one input to accept data from GPRs. For comparison, OTBN's `BN.SID` instruction uses the output from the GPR multiplexer to index the WDR output multiplexer and stores the value to memory within one cycle. Storing to memory is the most intensive part of the critical path of `BN.SID` due to the structure of SRAM cells, and `BN.BROADCAST` would not need to do this. Therefore, critical path requirements of `BN.BROADCAST` are relatively simple in the context of the existing instruction set, so this instruction should execute in one cycle.

Hardware Modifications Required:

- Addition of a fanout of wires from the output of the GPR selector multiplexer.
- Potential insertion of buffers by physical design tools
- Extension of the existing WDR input multiplexer by one input.

In summary, we designed 9 new instructions for OTBN, for optimisation of PQC. While they can be leveraged to significantly accelerate the (I)NTT in Kyber, the vector processing capabilities provided by these instructions have the potential to enhance performance of a wide range of algorithmic components. They have been implemented in Python in the `hw/ip/otbn/dv/otbnsim/insn.py` file. We designed these instructions with careful consideration of the required increase in chip area and hardware complexity. We aimed to minimise both of these costs and hence optimise the cost/performance trade-off.

Chapter 4

Evaluation

In this section, we describe the evaluation process we followed to empirically assess the quality of the enhancements we propose in this work. The evaluation process consisted of the following phases:

1. Rigourous testing of the correctness of our implementation.
2. Quantification of the performance improvements contributed by our hardware/ software co-design approach over baseline measurements on OTBN.
3. Comparative analysis of the performance of our optimised OTBN implementation with the performance of the official reference implementation, executed on the most similar platform to OTBN —the RISC-V Ibex core.

4.1 Evaluation Methodology

We conducted each of our evaluation phases separately, and in a uniform manner on the four implementations we developed (baseline implementations of NTT and INTT, and optimised implementations of NTT and INTT). Each phase was carried out as follows:

1. **Testing of correctness**

   During development, we tested our implementations over arrays of 256 fixed random values. Once development was complete, we extensively validated both the correctness of our implementations using the gold-standard input/output value pairs generated by NIST test vectors. Using the test framework outlined in Chapter 3.2, we captured the expected input and output value pairs upon entry to and exit from the I(NTT) functions during execution of the official reference implementation. We used the `test_vectors$ALG` executable upon compiling the reference C code on Linux. This generates 10,000 sets of test vectors which contain keys, ciphertexts and shared secrets. The `$ALG` variable is used to identify the parameter set, which represents the security level in Kyber. We gathered values for all 3 parameter sets and for each set, tested our implementations in a black-box test setting, comparing expected and generated output values.

2. **Quantification of Performance Improvements on OTBN**

   OTBNSim is cycle-accurate for all existing instructions. The cycle counts of existing instructions were empirically obtained using a prototype chip. Because we did not have access to a prototype chip during development, we estimated the cycle counts of our new instructions using inference and hardware-oriented reasoning, described in Chapter 3.5. We then similarly hard-coded our estimated cycle counts into the simulator. We read the value of OTBN's `INSN_CNT` variable after execution of each implementation to record the number of cycles spent.

3. **Comparative Performance Analysis with RISC-V Ibex Core**

   Because there is no compiler available yet for OTBN, we were unable to run the official reference C code directly on OTBN. We translated the reference implementation as directly as possible into OTBN assembly (Chapter 3.3). However, only conducting comparisons between this baseline and our optimised implementation presented some limitations. Firstly, we had implemented this baseline ourselves and in addition, we acknowledged that OTBN may not be familiar to a general audience. Therefore, to ensure a balanced and objective evaluation, we searched for another platform that is comparable to OTBN and on which we could run the official reference C code, to obtain an "anchor" benchmark for comparison. As advised by experts from lowRISC, the RISC-V Ibex core is the most suitable reference for comparison. The OpenTitan chip is modelled on RISC-V Ibex, a 32-bit, customisable core intended for use in lightweight IoT devices [1].

   We cloned the GitHub repository containing the code for the Ibex simulator [24] and imported the required C and header files (for running NTT and INTT) from the Kyber reference implementation. We followed the instructions for compiling and executing code on the Ibex simulator that was available in the repository documentation. The cycle count was automatically output to the console. We did not modify any code within the Kyber reference implementation or change any of the default configuration settings for Ibex.

4.2 Results

All implementations that we developed passed the tests for correctness before they were analysed further in terms of performance.

The cycle counts spent in the execution of each implementation are presented in Table 4.1. The performance improvement factors achieved by the optimised implementations over the baseline implementations are shown in Table 4.2. Because the new vectorised implementation required a complete redesign of the algorithm, incremental performance measurements would not have led to meaningful results. Therefore, only the final results are reported for implementations that had been fully functionally verified.

Table 4.1: Comparison of Implementation Cycle Counts

Implementation	**Cycle Count**
`otbn_ntt_baseline`	91939
`ibex_ntt_baseline`	46497
`otbn_ntt_optimised`	4356
`otbn_invntt_baseline`	149435
`ibex_invntt_baseline`	71327
`otbn_invntt_optimised`	6133

Table 4.2: Performance Improvements over Baseline Implementations

Implementation	**OTBN Baseline**	**Ibex Baseline**
OTBN Optimised NTT	21.1x	10.7x
OTBN Optimised INTT	24.3x	11.6x

4.3 Discussion

The results showcase the superiority of our optimised implementations over the baseline implementations on both OTBN and RISC-V Ibex. The NTT implementation achieved 21.1x and 10.7x performance improvements and the INTT achieved 24.3x and 11.6x performance improvements over the baselines on OTBN and Ibex, respectively. This demonstrates the large performance enhancements that can be gained through integration of minor architectural modifications into the OTBN platform. In interpreting these results, there are a number of important considerations.

The results were obtained on simulators as the physical hardware was not readily available during development. While the simulators are of production quality, external factors such as concurrent executing processes and the heat of the computer do not affect results, whereas variations may be observed in real-world scenarios. Although we believe our cycle count estimates for our new instructions to be accurate, the optimised OTBN implementations assume the correctness of these estimates (Chapter 3.5) when the instruction set extensions are deployed on hardware. The Ibex simulator provides optimistic performance results as it assumes single-cycle latency for memory read operations, which may not always be the case due to memory hierarchies in deployment. This means that the reference implementations of (I)NTT may be slower to execute on a physical Ibex core. In this case, our techniques would demonstrate a larger performance improvement factor over Ibex. However, it is not unreasonable to assume that single-cycle memory reads would be possible for this application, as our code and data size are relatively small. In contrast, OTBN has 2-cycle latency for memory read operations, a value which was been empirically determined by maintainers of the project. It is also noted that the Kyber reference implementation has not been optimised for any platform and favours readability over performance. However, the limited nature of the existing OTBN instruction set constrains the potential for optimisation without incorporating extensions.

Some additional benefits offered by our implementations, including the baselines, is that they all execute in constant time, as there is a single execution path through the code and no branching or conditional statements. This reduces exposure to side-channel attacks such as timing attacks. Furthermore, these performance results were achieved on a platform that does not support out-of-order execution of instructions. This is a feature afforded by larger CPUs and can improve performance significantly by pipelining instructions to maximise resource usage. In spite of the lack of potential to interleave instructions, strong performance was achieved.

Chapter 5

Related work

A surge in interest in the field of PQC has been reflected in an abundance of recent research endeavours. Focus has broadened from a primarily theoretical perspective on PQC to include the practical implications of deploying quantum-secure cryptosystems. While developments in the domains of computer architecture and cryptography are advancing rapidly, the research community is responding to the urgent need to reconcile innovations in both fields in the realisation of viable post-quantum cryptosystems. The predominance of lattice-based approaches amongst NIST's selected candidates is an indicator of their likely prevalence in the future. Therefore, research is rapidly advancing in the area of lattice-based PQC. Existing research has explored acceleration methods for lattice-based PQC in three categories: pure software, custom hardware and hardware/software co-design. In this section, we will analyse prominent contributions to the field in each category and identify the research gap that we aim to bridge with our contributions.

5.1 Software-based approaches

Software approaches leverage modern instruction sets which exploit complex hardware features of proprietary platforms. In 2018, Seiler [41] published the optimised AVX2 implementation used in the official Kyber repository. It restructures Montgomery reduction by splitting multiplications into separate operations on low and high parts. AVX2 supports packed signed and unsigned multiplication of low and high integer values. The proposed approach is not directly applicable to Dilithium due to the mismatch in operand size for multiplications. Our approach exploits the similarities between Kyber and Dilithium and we predict that it should be straightforward to transfer it to Dilithium. AVX2 was also leveraged by Roy [40] in the design of a high-throughput implementation of the Saber algorithm. A batching technique is applied which allows four simultaneous Saber KEM operations to execute. These specialised approaches use complex, high-latency instructions, characteristic of high-powered instruction sets such as AVX2 found in large CPUs but less applicable to low-power platforms like OTBN.

Optimisation of lattice-based PQC has also been investigated for microcontrollers. In 2019, Botros et al. [8] presented an implementation of Kyber on ARM Cortex-M4, which is highly efficient in terms of memory and performance. They employ techniques such as link-time optimisations, packed vectorised operations on lower and upper register elements, and instruction alignment for smoother transitions between ARM instruction types. They also leverage the external eXtended Keccak Code Package library from the `pqm4` [21] framework. In 2022, Abdulrahman et al. [2] improve upon this work, using ARM's digital signal processing (DSP) instructions such as the mixed-width signed multiply-accumulate (`smlaw`) operation for reducing the cycle cost of Barrett reduction and floating-point operations for caching values within the NTT. Many of the optimisations were platform-specific and the ability to integrate existing optimised code from an external library is owed to the widespread popularity of the Cortex-M4 platform. Conversely, OTBN is an emerging technology to which these enhancements do not apply.

The ARM v8 processor series is targeted at higher-powered devices such as the Macbook Air. It has also been investigated extensively in the context of PQC acceleration. In 2021, Nguyen et al. [31] designed the first optimised implementations of Kyber, NTRU and Saber using NEON-Based Special Instructions of ARMv8, establishing state-of-the-art (SoA) performance results through matrix-to-vector polynomial multiplication. Becker et al. [3] subsequently designed an optimised version of the NTT on the Cortex-A72, improving on the prior SoA performance for Kyber, Dilithium and Saber. Their key innovation is the combination of Montgomery multiplication and Barrett reduction. These approaches rely on the advanced vectorised computational abilities of the Armv8-A Neon instruction set. These include specialised, fixed-point arithmetic instructions and out-of-order execution for instruction pipelining. These architectural features are beyond the scope of OTBN and would require extensive architectural reconfiguration, which could require OTBN's security countermeasures to become more complex.

5.2 Hardware-based Approaches

Custom hardware components have been proposed to accelerate performance-critical operations on low-power platforms. At the expense of integrating specialised modules into the architectures of these platforms, such solutions offer significant efficiency improvements. In 2021, Yaman et al. [46] proposed three hardware architectures balancing lightweight design and high performance for Kyber. They designed a custom polynomial multiplier using bitwise modular reduction and incorporated it into a unified butterfly structure. Derya et al. [13] similarly incorporated a unified butterfly structure into a hardware accelerator for polynomial multiplication based on NTT. The approach provides run-time configurability based on parameters and compile-time configurability based on throughput and area requirements. This was the first approach to offer this extent of customisation for NTT acceleration in hardware, and improves on previous works [27] in terms of hardware complexity. There may be potential for custom hardware accelerators to be integrated into OTBN but there are several factors to consider. Any drastic changes to the architecture may be expensive to validate and align with existing design principles. The data flow path of OTBN and its extensive use of registers for operations may not integrate well with structures such as butterfly units. Therefore, the benefit may not justify the cost.

While hardware acceleration of bottlenecks such as the NTT has been extensively investigated, other avenues have also been explored. Ni et al. [32] incorporate a combination of inter- and intra-module pipelining optimizations into a custom hardware module for parallelisation of Kyber on lightweight processors such as Artix-7. Introducing instruction scheduling and pipelining capabilities into OTBN would significantly increase complexity. Verification would be challenging and maintaining current security guarantees of OTBN may introduce complexity. Complete hardware implementations of PQC algorithms such as Kyber have also been proposed. In 2021, Huang et al. [20] similarly focus on parallelisation through pipelining, in addition to optimising resource reuse on a Xilinx FPGA. Other variations have been proposed, such as the high-performance and low-memory Kyber implementation for Artix-7 by Xing and Li [45]. Through scheduling and embedding of custom accelerators within the pipeline, they create a manual design and demonstrate its efficiency over approaches such as high-level synthesis (HLS), a tool that generates Verilog code from a higher-level language like C. Such designs are highly efficient yet highly specific. Kyber-specific processors serve use cases where Kyber is the primary algorithm, but this design choice sacrifices reusability of components. They also serve as standalone processors and are not designed to leverage existing features of platforms such as OTBN, instead using custom components exclusively and incurring a high design cost. Investing in such specific designs this early in the life cycle of PQC may be unwise as the field will undoubtedly undergo many changes.

5.3 Hardware/Software Co-Design Approaches

Hardware/software co-design has emerged as a promising approach for PQC acceleration, blending the design processes of hardware and software in the extension of existing platforms. Recent works in this area have proposed instruction set extensions for acceleration of lattice-based PQC, with a strong emphasis on the RISC-V architecture. Among these, Fritzmann et al. [19] introduced RISQ-V in 2020, a modified RISC-V architecture in which they embedded tightly-coupled hardware accelerators into the processing pipeline. Due to the close physical integration with the CPU, these accelerators offer reduced latency, increased throughput and efficient and fewer memory accesses. Karabulut and Aysu [22] developed a novel RISC-V-based architecture that flexibly accelerates different implementations of the NTT, characteristic of different algorithms. The architecture incorporates runtime tracking of program execution to recognise the implementation. The architecture then responds to the NTT configuration to optimise data flow, predictive memory operations and instruction scheduling. We also aim to optimise PQC through hardware/software co-design. However, although OpenTitan is derived from RISC-V, we seek to align with the more specific design principles of OTBN.

The development of specialised arithmetic logical units (ALUs) has also been explored. In 2021, Nannipieri et al. [30] presented instruction set extensions based on two distinct ALUs for acceleration of Kyber and Dilithium on RISC-V. They achieve large efficiency improvements, offset by a small area overhead. Although the performance enhancements are significant, in the context of OTBN, integration of a custom ALU presents challenges. The concerns listed previously regarding complex hardware integration apply and optimal communication interfaces between the ALU and OTBN would need to be designed. Miteloudi et al. [29] further investigate the development of specialised ALUs for PQC. They design a unified ALU for acceleration of both Kyber and Dilithium which is integrated into a 4-stage pipeline 32-bit RISC-V processor. This is more efficient than Nannipieri's approach [30] in terms of efficiency and resource utilisation.

The cutting edge of hardware-software co-design generally exercises a high level of freedom within hardware design and this has resulted in extremely efficient hardware extensions which can be leveraged through software. In contrast, our work tightly aligns with OTBN's architecture and proposes feasible extensions that could be incorporated without the requirement for custom hardware components. It is possible that custom modules could be integrated into future iterations of OpenTitan. However, without access to the physical chip and in favor of a more immediately-integrable solution, our work focuses on maximising utility of OTBN's existing architecture and incorporating moderate extensions which are unlikely to violate design principles or security assumptions.

Chapter 6

Summary and conclusions

Our work demonstrates the optimisation potential of cryptographic co-processors such as OTBN for PQC. By incorporating minor architectural modifications to enable vectorized processing, instruction set extensions can exploit OTBN's wide data path to accelerate core cryptographic components such as the (I)NTT. In summary, our main contributions include baseline implementations of (I)NTT using OTBN's existing instruction set, instruction set extensions for acceleration of (I)NTT on OTBN and optimised implementations which leverage the new instructions, achieving 21.1× and 24.3× performance improvement factors. We also integrated extensions to the OTBNSim testing framework, enhancing its test case generation and debugging capabilities. Our findings suggest that further exploration within this field could yield significant findings.

A direct extension of this work would involve the implementation of the proposed instruction set extensions in hardware and their integration into OTBN. Hardware verification would be required to confirm the seamless translation from our instruction prototypes to physical implementations. The precise percentage increase in area required by these extensions could be quantified following hardware implementation. The assembly code implementations which we developed could then be executed and benchmarked directly on OTBN, leveraging the new extensions. The remaining components of the Kyber algorithm could be implemented in OTBN assembly, which would be the final stage in development of a fully-functional, complete implementation for this platform. Another promising endeavour would lie in the investigation of the general applicability of our technique to other cryptographic contexts. For example, we predict that it will be straightforward to transfer our techniques to the (I)NTT in Dilithium given the algorithmic similarities it shares with Kyber. The (I)NTT is a prominent feature in many lattice-based schemes and therefore our techniques may prove applicable or extensible in even broader contexts. Finally, the domain of PQC is dynamic and constantly evolving. We hope that our contributions will serve as a foundation or inspiration in the adaptation of platforms such as OTBN to support quantum-safe solutions as the ecosystem develops.

Bibliography

[1] 2022. URL: `https://github.com/lowRISC/ibex`.

[2] Amin Abdulrahman et al. “Faster Kyber and Dilithium on the Cortex-M4”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2022, pp. 853–871.

[3] Hanno Becker et al. “Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1”. In: *Cryptology ePrint Archive* (2021).

[4] Daniel J Bernstein and Tanja Lange. “Post-quantum cryptography”. In: *Nature* 549.7671 (2017), pp. 188–194.

[5] Guido Bertoni et al. “Keccak”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2013, pp. 313–314.

[6] Jean-François Biasse and Fang Song. “Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields”. In: *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2016, pp. 893–902.

[7] Joppe Bos et al. “CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 353–367.

[8] Leon Botros, Matthias J Kannwischer, and Peter Schwabe. “Memory-efficient high-speed implementation of Kyber on Cortex-M4”. In: *Progress in Cryptology–AFRICACRYPT 2019: 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9–11, 2019, Proceedings 11*. Springer. 2019, pp. 209–228.

[9] Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. “On error distributions in ring-based LWE”. In: *LMS Journal of Computation and Mathematics* 19.A (2016), pp. 130–145.

[10] Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. “Provably weak instances of Ring-LWE revisited”. In: *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*. Springer. 2016, pp. 147–167.

[11] Hao Chen. “Ring-LWE over two-to-power cyclotomics is not hard”. In: *Cryptology ePrint Archive* (2021).

[12] Information Technology Laboratory Computer Security Division. *Post-Quantum Cryptography Standardization - Post-Quantum Cryptography — CSRC — CSRC*. 2017. URL: `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization`.

[13] Kemal Derya et al. "CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication". In: *Microprocessors and Microsystems* 89 (2022), p. 104451.

[14] Léo Ducas et al. "CRYSTALS-Dilithium: A lattice-based digital signature scheme". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 238–268.

[15] CSRC Content Editor. *roots of trust - Glossary — CSRC*. URL: `https://csrc.nist.gov/glossary/term/roots_of_trust`.

[16] Yara Elias et al. "Provably weak instances of Ring-LWE". In: *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I 35*. Springer. 2015, pp. 63–92.

[17] Jay Fenlason and Richard Stallman. *GNU gprof*. URL: `https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html#SEC11`.

[18] Pierre-Alain Fouque et al. "Falcon: Fast-Fourier lattice-based compact signatures over NTRU". In: *Submission to the NIST's post-quantum cryptography standardization process* 36.5 (2018), pp. 1–75.

[19] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 239–280.

[20] Yiming Huang et al. "A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse". In: *IEICE Electronics Express* 17.17 (2020), pp. 20200234–20200234.

[21] Matthias J Kannwischer et al. "pqm4: Benchmarking NIST Additional Post-Quantum Signature Schemes on Microcontrollers". In: *Cryptology ePrint Archive* (2024).

[22] Emre Karabulut and Aydin Aysu. "RANTT: A RISC-V architecture extension for the number theoretic transform". In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2020, pp. 26–32.

[23] Adeline Langlois and Damien Stehlé. "Worst-case to average-case reductions for module lattices". In: *Designs, Codes and Cryptography* 75.3 (2015), pp. 565–599.

[24] lowRISC. *Ibex RISC-V Core*. 2022. URL: `https://github.com/lowRISC/ibex`.

[25] lowRISC. *lowRISC: Collaborative open silicon engineering*. URL: `https://lowrisc.org/`.

[26] lowRISC. *OpenTitan® Partnership Makes History as First Open-Source Silicon Project to Reach Commercial Availability · lowRISC: Collaborative open silicon engineering*. 2024. URL: `https://lowrisc.org/news/2024/02/opentitan-commercial-availability/`.

[27] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. “FPGA implementation of a run-time configurable NTT-based polynomial multiplication hardware”. In: *Microprocessors and Microsystems* 78 (2020), p. 103219.

[28] Victor S Miller. “Use of elliptic curves in cryptography”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1985, pp. 417–426.

[29] Konstantina Miteloudi et al. “PQ. V. ALU. E: Post-quantum RISC-V Custom ALU Extensions on Dilithium and Kyber”. In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2023, pp. 190–209.

[30] Pietro Nannipieri et al. “A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms”. In: *IEEE Access* 9 (2021), pp. 150798–150808.

[31] Duc Tri Nguyen and Kris Gaj. “Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special instructions of ARMv8”. In: *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*. 2021.

[32] Ziying Ni et al. “HPKA: A High-Performance CRYSTALS-Kyber Accelerator Exploring Efficient Pipelining”. In: *IEEE Transactions on Computers* (2023).

[33] OpenTitan. *OTBN Simulation Software - OpenTitan Documentation*. URL: `https://opentitan.org/book/hw/ip/otbn/dv/otbnsim/index.html`.

[34] OpenTitan. *Use Cases - OpenTitan Documentation*. URL: `https://opentitan.org/book/doc/use_cases/index.html`.

[35] PQ-CRYSTALS. *Kyber*. 2022. URL: `https://github.com/pq-crystals/kyber`.

[36] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography.” In: *In Proceedings of the 37th ACM Symposium on Theory of Computing (STOC)*. 7.30 (2010), p. 11.

[37] Oded Regev. “The learning with errors problem”. In: *Invited survey in CCC* 7.30 (2010), p. 11.

[38] Ronald L Rivest. “Cryptography”. In: *Algorithms and complexity*. Elsevier, 1990, pp. 717–755.

[39] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[40] Sujoy Sinha Roy. “Saberx4: High-throughput software implementation of Saber key encapsulation mechanism”. In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE. 2019, pp. 321–324.

[41] Gregor Seiler. “Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography”. In: *Cryptology ePrint Archive* (2018).

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

[43] Peter W Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM review* 41.2 (1999), pp. 303–332.

[44] Yang Wang and Mingqiang Wang. “Module-LWE versus Ring-LWE, revisited”. In: *Cryptology ePrint Archive* (2019).

[45] Yufei Xing and Shuguo Li. “A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 328–356.

[46] Ferhat Yaman et al. “A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1020–1025.

Appendix A

Pseudocode for Optimised NTT and INTT implementations

Algorithm 1: mont_reduce_vec function

```
Function mont_reduce_vec($a$):
    for $i \leftarrow 0$ to $7$ do
        t[i] ← a[i] × QINV
        t[i] ← t[i] × KYBER_Q
        t[i] ← a[i] - t[i]
        t[i] ← t[i] ≫ 16
    return t
```

Algorithm 2: fqmulvec function

```
Function fqmulvec($vec\_a$, $vec\_b$):
    return mont_reduce_vec($a[i]\&0xFFFF \times b[i]\&0xFFFF$ for $i \leftarrow 0$ to $7$)
```

Algorithm 3: barrett_reduce_vec function

```
Function barrett_reduce_vec($a$):
    for $i \leftarrow 0$ to $7$ do
        t[i] ← a[i] × v
        t[i] ← t[i] AND 0XFFFFFFFF
        t[i] ← t[i] + (1≪25)
        t[i] ← t[i] ≫ 26
        t[i] ← t[i] × KYBER_Q
    return t
```

Algorithm 4: Vectorized NTT Implementation

```
Masks[mask_len8] ← [0x00000000] * 4 + [0xFFFFFFFF] * 4
Masks[mask_len4] ← ([0x00000000] * 2 + [0xFFFFFFFF] * 2) * 2
Masks[mask_len2] ← [0x00000000FFFFFFFF] * 4
k ← 1
Function ntt_vec(r[256]):
    for len in {128, 64, 32, 16} do
        for start ← 0 to 255 by 2×len do
            zetavec ← broadcast zeta[k++]
            for i ← 0 to 15 do
                idx ← i×16 + start
                Vec[rj_vec] ← r[idx ... idx+15]
                Vec[rjlen_vec] ← r[idx+len ... idx+len+15]
                Vec[rjlen_vec_low] ← rjlen_vec AND [0x0000FFFF] * 8
                Vec[rjlen_vec_upp] ← rjlen_vec ≫ 16
                t_low ← fqmulvec(zetavec, rjlen_vec_low)
                t_upp ← fqmulvec(zetavec, rjlen_vec_upp)
                t ← t_low XOR t_upp
                rjlen_vec_new ← rj_vec - t
                rj_vec_new ← rj_vec + t
                store rjlen_vec_new, rj_vec_new to r[idx+len], r[idx]
    for len in {8, 4, 2} do
        for i ← 0 to 15 do
            num_zetas ← 8 / len
            zetavec ← 0
            zeta_mask ← (1 ≪ (len ≪ 4)) - 1
            for z ← 0 to num_zetas - 1 do
                tmp ← broadcast zeta[k++]
                tmp ← tmp AND zeta_mask
                zetavec ← zetavec XOR tmp
                zeta_mask ← zeta_mask ≪ (len ≪ 5)
            idx ← i × 16
            Vec[rjvec] ← r[idx ... idx+15]
            Vec[nextvec] ← r[idx+16 ... idx+31]
            Vec[rjlen_vec] ← (nextvec ⊕ rjvec) ≫ (len ≪ 4)
            Vec[rjlen_vec_low] ← rjlen_vec AND [0x0000FFFF] * 8
            Vec[rjlen_vec_upp] ← rjlen_vec ≫ 16
            t_low ← fqmulvec(zetavec, rjlen_vec_low)
            t_upp ← fqmulvec(zetavec, rjlen_vec_upp)
            t ← t_low XOR t_upp
            Vec[rjlen_vec_new] ← rjvec - t
            rjlen_vec_new ← rjlen_vec_new AND Masks[mask_len{len}]
            rjlen_vec_new ← rjlen_vec_new ≪ (len ≪ 4)
            Vec[rjvec_new] ← rjvec + t
            rjvec_new ← rjvec_new AND Masks[mask_len{len}]
            res ← rjvec_new XOR rjlen_vec_new
            store res to r[idx]
```

Algorithm 5: Vectorized INTT Implementation (Part 1)

```
Masks[$mask\_len8$] ← [0x00000000] * 4 + [0xFFFFFFFF] * 4
Masks[$mask\_len4$] ← ([0x00000000] * 2 + [0xFFFFFFFF] * 2) * 2
Masks[$mask\_len2$] ← [0x00000000FFFFFFFF] * 4
Const($f$) ← 1441
Vec[$fvec$] ← broadcast_32b f
Const($v$) ← 20159 $k$ ← 127
Function invntt_vec($r[256]$):
    for $len$ in {2, 4, 8} do
        for $i$ ← 0 to 15 do
            num_zetas ← 8 / len
            zetavec ← 0
            zeta_mask ← (1 ≪ (len ≪ 4)) - 1
            for $z$ ← 0 to $num\_zetas$ - 1 do
                tmp ← broadcast zeta[k- -]
                tmp ← tmp AND zeta_mask
                zetavec ← zetavec XOR tmp
                zeta_mask ← zeta_mask ≪ (len ≪ 5)
            idx ← i × 16
            Vec[$rjvec$] ← r[idx . . . idx+15]
            Vec[$nextvec$] ← r[idx+16 . . . idx+31]
            Vec[$rjlen\_vec$] ← (nextvec ⊕ rjvec) ≫ (len ≪ 4)
            Vec[$barrett\_arg$] ← rjvec + rjlen_vec
            Vec[$barrett\_arg\_vec\_low$] ← barrett_arg_vec AND [0x0000FFFF] * 8
            Vec[$barrett\_arg\_vec\_upp$] ← barrett_arg_vec ≫ 16
            Vec[$rjvec\_new\_low$] ← barrett_reduce_vec($barrett\_arg\_vec\_low$)
            Vec[$rjvec\_new\_upp$] ← barrett_reduce_vec($barrett\_arg\_vec\_upp$)
            Vec[$rjvec\_new$] ← rjlen_vec_low XOR rjlen_vec_upp
            Vec[$rjlen\_vec$] ← rjlen_vec - rjvec
            Vec[$rjlen\_vec\_low$] ← rjlen_vec AND [0x0000FFFF] * 8
            Vec[$rjlen\_vec\_upp$] ← rjlen_vec ≫ 16
            rjlen_vec_low ← fqmulvec($zetavec, rjlen\_vec\_low$)
            rjlen_vec_upp ← fqmulvec($zetavec, rjlen\_vec\_upp$)
            Vec[$rjlen\_vec\_new$] ← rjlen_vec_low XOR rjlen_vec_upp
            rjlen_vec_new ← rjlen_vec_new AND Masks[$mask\_len\{len\}$]
            rjlen_vec_new ← rjlen_vec_new ≪ (len ≪ 4)
            res ← rjvec_new XOR rjlen_vec_new
            store res to r[idx]
```

Algorithm 6: *Contd.* Vectorized INTT Implementation (Part 2)

```
for len in {16, 32, 64, 128} do
    for start ← 0 to 255 by 2×len do
        zetavec ← broadcast zeta[k- -]
        for i ← 0 to 15 do
            idx ← i×16 + start
            Vec[rj_vec] ← r[idx ... idx+15]
            Vec[rjlen_vec] ← r[idx+len ... idx+len+15]
            Vec[barrett_arg] ← rjvec + rjlen_vec
            Vec[barrett_arg_vec_low] ← barrett_arg_vec AND [0x0000FFFF] * 8
            Vec[barrett_arg_vec_upp] ← barrett_arg_vec ≫ 16
            Vec[rjvec_new_low] ← barrett_reduce_vec(barrett_arg_vec_low)
            Vec[rjvec_new_upp] ← barrett_reduce_vec(barrett_arg_vec_upp)
            Vec[rjvec_new] ← rjlen_vec_low XOR rjlen_vec_upp
            Vec[rjlen_vec] ← rjlen_vec - rjvec
            Vec[rjlen_vec_low] ← rjlen_vec AND [0x0000FFFF] * 8
            Vec[rjlen_vec_upp] ← rjlen_vec ≫ 16
            rjlen_vec_low ← fqmulvec(zetavec, rjlen_vec_low)
            rjlen_vec_upp ← fqmulvec(zetavec, rjlen_vec_upp)
            Vec[rjlen_vec_new] ← rjlen_vec_low XOR rjlen_vec_upp
            store rjlen_vec_new, rj_vec_new to r[idx+len], r[idx]
for idx = 0 to 255 by 16 do
    Vec[rj_vec] ← r[idx ... idx+15]
    Vec[rj_vec_low] ← rj_vec AND [0x0000FFFF] * 8
    Vec[rj_vec_upp] ← rj_vec ≫ 16
    rj_vec_new_low ← fqmulvec(zetavec, rjlen_vec_low)
    rj_vec_new_upp ← fqmulvec(zetavec, rjlen_vec_upp)
    rj_vec_new ← t_low XOR t_upp
    store rj_vec_new to r[idx]
```