ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Fall Semester 2024

Design and Optimization of a PQC ISA Extension for OTBN

Semester Project

Pascal Etterli
petterli@ethz.ch

23. Dec. 2024, Version 1.0

Advisors: Dr. Andrea Caforio, andrea.caforio@lowrisc.org
Dr. Pascal Nasahl, nasahlpa@lowrisc.org
Dr. Pirmin Vogel, vogelpi@lowrisc.org
Navaneeth Kunhi Purayil, nkunhi@iis.ee.ethz.ch

Professor: Prof. Dr. L. Benini, lbenini@iis.ee.ethz.ch

Abstract

The rise of quantum computing poses significant threats to classical public key cryptographic systems, prompting the development of post quantum cryptography (PQC) schemes that remain secure against quantum attacks. While it is well understood how classical cryptographic schemes and their underlying big number problems can be executed efficiently, for PQC schemes, like ML-DSA (standardized in 2024 as FIPS204), it is not as straightforward as for classical schemes because these schemes are based on polynomial modulo arithmetics on small numbers (range of 32 bits). In this project, we focus on researching how ML-DSA can be efficiently implemented on OpenTitan Big-Number Accelerator (OTBN), a security-focused cryptographic co-processor of the OpenTitan root of trust project. The OTBN features 256-bit wide registers to facilitate efficient big number operations, but it lacks SIMD capabilities, which would be highly beneficial for an efficient ML-DSA execution. We therefore propose a generic low-cost SIMD instruction extension for the OTBN and implement it in hardware focusing on reusing existing resources. We show the benefits of the proposed instructions on an NTT benchmark, a salient computation in ML-DSA, resulting in a speed up of 3.46x. In addition, we synthesize the design on a TSMC65 technology node achieving a frequency of 125 MHz and area overhead of 23 %. The second part of the project focuses on optimizing the implementation. This results in an optimized design achieving a speed up of 3.27x whilst only leading to an area overhead of 19 %, increasing the area of the complete Earlgrey OpenTitan chip by less than 2 %.

Acknowledgments

First and foremost, I want to thank the lowRISC team and Professor Luca Benini for giving me the opportunity to do this semester project. Contributing to such a unique open source project was a superb experience and it allowed me to gain invaluable experience in digital design, computer architecture, and hardware-software co-design.

I am very grateful to Andrea Caforio, Pascal Nasahl and Pirmin Vogel, for their excellent supervision, close support, and detailed, helpful feedback on a regular basis and throughout all stages of this project. It surely is a pleasure to work with them and their exquisite OpenTitan platform.

I must thank Navaneeth Kunhi Purayil for helping me set up the synthesis flow and providing valuable feedback throughout all stages of this project.

Last but not least, I also want to thank Raphael Roth for having many fruitful technical discussions.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- (●) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Design and Optimization of a PQC ISA Extension for OTBN

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Etterli	Pascal

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Muri AG, 23. Dec. 2024	P. Etterli

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard

Contents

1. Introduction **1**

2. Background **3**
2.1. Lattice-based cryptography 3
2.2. Digital signatures 4
2.3. ML-DSA 5
2.4. Polynomial multiplication 6
2.4.1. Number Theoretic Transform 6
2.4.2. NTT implementation 9
2.5. Modular multiplication algorithms 11
2.6. OpenTitan 13
2.6.1. OpenTitan Big-Number Accelerator 14
2.6.2. Register blanking 17

3. Baseline architecture **20**
3.1. Literature review 20
3.2. ISA extension proposal 23
3.3. Benefits of proposed instructions 26

4. Baseline implementation **28**
4.1. Architectural decisions 28
4.2. Design description 29
4.2.1. Vectorized adder 29
4.2.2. Vectorized multiplier 29
4.2.3. Bignum ALU module 31
4.2.4. Bignum MAC module 37
4.3. Integration into pipeline 41
4.4. Verification 42

5. Baseline benchmark **44**
5.1. Non-vectorized NTT implementation 44
5.2. Vectorized NTT implementation 45

5.3. Results 46

6. Baseline synthesis **48**
6.1. Synthesis setup 48
6.2. Synthesis results 49
6.2.1. Critical path analysis 49

7. Design optimization **53**
7.1. Optimization proposal 53
7.2. Benchmark results 54
7.3. Synthesis results 55
7.3.1. Critical path analysis 57

8. Conclusion and Future Work **63**
8.1. Conclusion 63
8.2. Future work 63

A. Bignum MAC FSM control signals **66**

B. Synthesis **68**
B.1. Critical path of default OpenTitan Big-Number Accelerator (OTBN) . . . 68
B.2. Critical path comparison of all designs 68
B.3. Setup details 74

C. Task Description **75**

List of Acronyms **82**

List of Figures **84**

List of Tables **87**

Bibliography **88**

Chapter 1

Introduction

The today used public key cryptography (PKC) schemes are all based upon mathematical problems that are hard to solve on current computing platforms. Recent advances in quantum computing however pose a threat to these PKC schemes due to Shor's algorithm [1]. This algorithm allows to solve the PKC underlying problems i.e., integer factorization and discrete logarithms, in polynomial time on large-scale quantum computers. The existence of such systems seems to be realistic in the near future [2]. Due to these reasons, the National Institute of Standards and Technology (NIST) has started to standardize post quantum cryptography (PQC) schemes for digital signatures and key exchange mechanisms which are said to be resistant to quantum computer based attacks. One of these schemes for digital signatures is ML-DSA (Module-Lattice-Based Digital Signature Standard), previously known as DILITHIUM [3] and standardized in 2024 as FIPS 204 [4]. ML-DSA and most of the other selected schemes are lattice based algorithms operating over the ring of integers modulo a small prime which is typically smaller than 32 bits. In particular, one compute bottleneck is the Number Theoretic Transform (NTT), a variant of a discrete Fourier transform (DFT), can be efficiently computed using the Fast Fourier Transform (FFT) algorithm. The FFT algorithm operates in-place and requires to compute a specific function (named butterfly) repeatedly and therefore its required arithmetic operations could benefit strongly from Single Instruction Multiple Data (SIMD) vector instructions.

Designing efficient systems to execute lattice-based cryptography schemes is an active area of research and a lot of work has been published covering a broad range of strategies [5, 6, 7, 8, 9, 10, 11, 12]. These strategies range from full data flow implementations to SIMD instruction set architecture (ISA) extensions where some work targets the OpenTitan Big-Number Accelerator (OTBN), a security-focused cryptographic co-processor of the OpenTitan root of trust project.

The OpenTitan project is an open-source project that seeks to deliver a transparent, vendor-agnostic, and high-quality silicon root of trust (RoT) ecosystem. Such hardware RoT are the basic building blocks to enable a secure computing infrastructure. The OpenTitan project features a collection of hardware IPs as well as two top level designs for a

wide range of security applications. Because currently used cryptographic algorithms like RSA are often based upon big numbers, the OpenTitan project features a dedicated security hardened big number core named OpenTitan Big-Number Accelerator (OTBN). This accelerator has two instruction sets. One operates on regular 32-bit registers and is comparable to the RV32I instruction set. The second instruction set operates on special 256-bit wide registers which are beneficial for big-number arithmetics as found for example in RSA exponentiation. This allows the Ibex core to offload computations to improve the overall runtime.

The goal of this project was to summarize the existing work related to efficient implementations of Module-Lattice-Based Digital Signature Standard (ML-DSA) and then elaborate an OTBN extension proposal and finally implement and optimize the extension. This proposed extension should however not be fully specialized on lattice-based schemes but should have rather a generic character and lightweight area requirements. The reason is that the OTBN is intended to be used as a subpart of a heterogenous system on chip (SoC) to accelerate a broad range of PQC and current cryptographic schemes. A solution with relatively extreme area requirements but a very high performance is therefore not suitable. Also, keep in mind that the special security requirements (introduced in Section 2.6.2) add considerable overhead to any implementation and thus more generic and lightweight implementations are even more appealing.

As will be explained in Section 2.3, lattice-based cryptographic schemes heavily rely on Keccak functions and polynomial multiplications. In regard to Keccak computation, the OpenTitan ecosystem already features a dedicated IP block named KMAC which efficiently computes the required Keccak functions. Earlier work [11] already discussed the use of this IP block for PQC schemes and lowRISC is currently looking into this approach. This work therefore focused only on enabling efficient processing of polynomial arithmetics.

This report first discusses the required background about PQC schemes, the NTT, modulo reductions, and the OTBN in more detail in Chapter 2. Chapter 3 discusses related work and proposes an ISA extension hereinafter referred to as the baseline design. The implementation thereof is documented in Chapter 4 and its performance as well as the area impact is discussed in Chapter 5 and Chapter 6, respectively. A more in-depth analysis of the implementation is performed in Chapter 7 which finally results in a design optimization proposal. Chapter 7 documents the implementation of the optimization proposal and also features an in-depth analysis of the results, comparing the baseline and the optimized design with the original OTBN.

Chapter 2

Background

This chapter explains the required cryptographic background on post quantum cryptography schemes, the schemes themselves and implementation relevant aspects. This includes polynomial arithmetics, the Number Theoretic Transform (NTT), and modular reduction and multiplication algorithms. As last part, it introduces the OpenTitan project and the architecture of the OpenTitan Big-Number Accelerator (OTBN).

2.1. Lattice-based cryptography

Classical cryptographic methods, such as RSA and elliptic curve cryptography, rely on the computational hardness of problems like integer factorization and discrete logarithms. However, these methods are vulnerable to quantum computers, which can efficiently solve these problems using algorithms like Shor's algorithm [1]. This algorithm allows to solve the classical cryptographic methods' underlying problems in polynomial time on large-scale quantum computers which seems to be realistic in the near future [2].

In the last years, a lot of research has been conducted to design new quantum resistant cryptographic methods. One of the most promising types of quantum resistant cryptographic systems that emerged is based upon lattice-based problems. Unlike classical systems, lattice-based cryptographic schemes are based on hard problems in high-dimensional geometries called lattices. Simply put, a lattice is a grid-like arrangement of points in multidimensional space formed by linear combinations of basis vectors with integer coefficients. One of the best known problems is the shortest vector problem (SVP) where, given a lattice, the goal is to find the shortest possible non-zero vector inside the lattice. This problem is believed to be hard to solve in high-dimensional lattices. Most cryptographic systems however do not directly rely on such a search problem but are based on more elaborated variants like the Learning with Error (LWE) problem [13].

The Learning with Error (LWE) problem is a special kind of solving a system of linear equations over a lattice where these equations represent the information to

be encrypted [14]. These equations consist of polynomials from a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ defined by the cyclotomic polynomial $X^n + 1$. This ring includes polynomials that have coefficients from a ring over integers modulo a ring isomorphism q, denoted as $\mathbb{Z}_q$. The ring isomorphism q is mostly chosen to be a prime. When performing arithmetics like adding or multiplying polynomials from $\mathcal{R}_q$, the resulting coefficients therefore must be reduced by the prime q. This modulo reduction satisfying $r = x \mod q$ can be performed in either signed or unsigned congruence representation. For an odd prime q, the signed representation defines the elements of $\mathbb{Z}_q$ to be in the range of $\left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$ and its reduction is denoted as $r = x \mod^{\pm} q$. Elements in unsigned representation are within $[0, q[$ and the reduction is denoted as $r = x \mod q$. The inverse multiplicative element denoted as $x = a^{-1}$ of $a \in \mathbb{Z}_q$ is an element that satisfies $ax \equiv 1 \pmod q$ which can be written as $x \equiv a^{-1} \pmod q$. A reader searching for a more detailed introduction to algebraic number theory and especially lattice based encryption is referred to [13].

Returning back to the Learning with Error (LWE) problem, this problem can be thought of as the following: Given a random matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ and a small random error vector $\mathbf{e} \in \mathbb{Z}_q^m$, the system of equations is

$$\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \pmod q. \tag{2.1}$$

The problem now is to recover the secret vector $\mathbf{s}$ from only the information $\mathbf{A}$ and $\mathbf{b}$. Under special parameters, this turns out to be a hard problem due to the introduced error as regular methods like Gaussian elimination cannot be applied. It can even be shown that an LWE problem can be approximated to versions of known NP-hard lattice problems giving LWE problems strong worst-case hardness security proofs [13].

2.2. Digital signatures

Digital signatures can be used to ensure the integrity and authenticity of a message. A signer can create a pair of secret and public keys and then sign a message with the private key. This allows anyone with the corresponding public key to verify that the message originated from the sender and has not been altered whilst transmitting. As only the signer is in possession of the private key, any receiver can also verify the authenticity of the sender, i.e., whether the message originates from the sender and not an adversary [15]. In the context of a RoT system, such digital signatures can be used to verify firmware updates before installing them and thus ensure that only trusted firmware is running on the protected system. This requires that the public key is secretly distributed to the RoT system beforehand.

A common approach to creating a digital signature is defined in [16]. The message to be signed is first hashed and then the signature is generated using the private key. The resulting signature then can be verified by the receiver by again hashing the received message and then verifying the signature using the public key and hashed data. Another use of digital signatures is in the public key infrastructure context where certificates are signed by trusted certificate authority [4].

2.3. ML-DSA

The ML-DSA is a new digital signature algorithm which has been standardized in 2024 as FIPS 204 [4] and previously was known as Dilithium [3]. Its security is based on the Module-Learning with Error (MLWE) problem which is a generalization of the LWE problem. In the MLWE setting, the module $\mathbb{Z}_q$ is replaced by the polynomial ring $\mathcal{R}_q$ with $q = 2^{23} - 2^{13} + 1 = 8380417$ and $n = 256$. For the remainder of this work, q and n refer to these values if not stated otherwise. The ML-DSA offers three levels of security with the lowest level named ML-DSA-44 and the higher levels ML-DSA-65 and ML-DSA-87. The suffix of the levels (i.e. -44) relates to the dimensions of the used lattice space which, due to the nature of MLWE problems, scale the hardness of the problem. Note, that scaling the hardness directly scales the computational complexity of ML-DSA.

The effective algorithms for key generation, signing, and verifying are given in Algorithms 1 to 3 and in the following explained briefly. A detailed explanation and the definition of the helper functions can be found in the standard [4].

The key generation described in Algorithm 1 requires a random 32-bit seed as input and produces a public and private key. This seed is used to generate other seeds by expanding it using a special XOF (eXtendable-Output Function) which is based upon SHAKE256. Based on these created seeds, the public matrix $\mathbf{A}$ as well as the secret vectors $\mathbf{S}_1$ and $\mathbf{S}_2$ are pseudorandomly sampled from $\mathcal{R}_q$. Finally, the public value $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ is computed.

The message signing, described in Algorithm 2, takes the private key and the message as inputs and outputs a signature. First, the public matrix $\mathbf{A}$ is generated using the public seed ρ and the XOF. Within a rejection sampling loop, a possible signature is computed and checked whether it would leak information about the private key. This computation includes multiple polynomial multiplications whereas the sampling of the challenge is based also on the selected XOF. As this is a rejection sampling approach, the runtime is not deterministic. The average number of required rounds is around 5 [4]. An efficient computation is therefore of great importance.

The verifying process in Algorithm 3 takes the public key, the message and signature as inputs and returns whether the signature is valid or not. It again regenerates the public matrix $\mathbf{A}$, computes the challenge value, and finally performs the check using multiple polynomial multiplications.

To summarize, all three algorithms heavily rely on the chosen XOF and polynomial arithmetics, especially multiplication of polynomials. From a computational point of view, the ML-DSA was developed such that the polynomial multiplication can be improved by leveraging the Number Theoretic Transform (NTT) which is explained in Section 2.4. In the context of the OpenTitan ecosystem, the computation of the XOF can be outsourced to the dedicated Keccak module named kmac. However, the NTT computations remain a bottleneck for an ML-DSA implementation on the current Ibex or OTBN processors. Developing an ISA extension for OTBN is therefore of greater importance to enable an efficient ML-DSA implementation.

Algorithm 1: ML-DSA key generation [4].

Input : 32-bit Seed ξ.
Output: Public key pk and private key sk.

$(\rho, \rho', K) \leftarrow \mathrm{H}\left(\xi || \mathrm{IntegerToBytes}(k,1) || \mathrm{IntegerToBytes}(\ell,1), 128\right)$
$\hat{\mathbf{A}} \leftarrow \mathrm{ExandA}(\rho)$ `// A is generated and stored in NTT representation`
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \mathrm{ExpandS}(\rho')$
$\mathbf{t} \leftarrow \mathtt{NTT}\left(\hat{\mathbf{A}} \circ \mathrm{NTT}\left(\mathbf{s}_1\right)\right) + \mathbf{s}_2$ `// compute` $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_1$
$(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \mathrm{Power2Round}\left(t\right)$ `// compress` $\mathbf{t}$
$pk \leftarrow \mathrm{pkEncode}\left(\rho, \mathbf{t}_1\right)$
$tr \leftarrow \mathrm{H}\left(pk, 64\right)$
$sk \leftarrow \mathrm{skEncode}\left(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0\right)$
return $\left(pk, sk\right)$

2.4. Polynomial multiplication

A polynomial multiplication with polynomials from $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ is an essential operation for many lattice-based cryptographic schemes. The simplest approach to multiplying two polynomials of degree n requires $\mathcal{O}(n^2)$ multiplications as each coefficient is multiplied by every other coefficient. However, such a polynomial multiplication can be viewed as a negative-wrapped convolution which allows to compute it more efficiently by making use of the convolution theorem and the Number Theoretic Transform (NTT). The NTT is a variation of the DFT over finite fields which thus can be efficiently computed by leveraging the FFT algorithm. By using the FFT-like NTT, the required coefficient multiplications can be reduced to $\mathcal{O}(n\log(n))$. The following sections summarize the key concepts of the NTT based on the tutorial [17].

2.4.1. Number Theoretic Transform

A regular polynomial multiplication where the polynomials $G(x)$ and $H(x)$ of degree $n-1$ are from the ring $\mathbb{Z}_q[x]$ (note the missing cyclotomic polynomial $X^n + 1$) is defined as

$$Y(x) = G(x) \cdot H(x) = \sum_{k=0}^{2(n-1)} y_k x^k \tag{2.2}$$

$$\text{where} \qquad y_k = \sum_{i=0}^{k} g_i h_{k-i} \mod q. \tag{2.3}$$

This is equivalent to a linear convolution between the coefficients' vectors $\mathbf{g}$ and $\mathbf{h}$

$$\mathbf{y}[k] = (\mathbf{g} * \mathbf{h})[k] = \sum_{i=0}^{k} \mathbf{g}[i]\mathbf{h}[k-i]. \tag{2.4}$$

Algorithm 2: ML-DSA signing [4].

Input : Private key sk, formatted message M' and per message randomness or dummy variable rnd.

Output: Signature σ.

$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow \text{skDecode}(sk)$
$\hat{\mathbf{s}}_1 \leftarrow \text{NTT}(\mathbf{s}_1)$
$\hat{\mathbf{s}}_2 \leftarrow \text{NTT}(\mathbf{s}_2)$
$\hat{\mathbf{t}}_0 \leftarrow \text{NTT}(\mathbf{s}_0)$
$\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ `// A is generated and stored in NTT representation`
$\mu \leftarrow \text{H}(\text{BytesToBits}(tr)||M', 64)$
$\rho'' \leftarrow \text{H}(K||rnd||\mu, 64)$ `// compute private random seed`
$\kappa \leftarrow 0$ `// initialize counter` κ
$(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} \leftarrow \text{ExpandMask}(\rho'', \kappa)$
  $\mathbf{w} \leftarrow \text{INTT}\left(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{y})\right)$
  $\mathbf{w}_1 \leftarrow \text{HighBits}(\mathbf{w})$
  $\tilde{c} \leftarrow \text{H}(\mu||\text{w1Encode}(\mathbf{w}_1), \lambda/4)$
  $c \leftarrow \text{SampleInBall}(\tilde{c})$
  $\langle\langle c\mathbf{s}_1 \rangle\rangle \leftarrow \text{INTT}(\hat{c} \circ \hat{\mathbf{s}}_1)$
  $\langle\langle c\mathbf{s}_2 \rangle\rangle \leftarrow \text{INTT}(\hat{c} \circ \hat{\mathbf{s}}_2)$
  $\mathbf{z} \leftarrow \mathbf{y} + \langle\langle c\mathbf{s}_1 \rangle\rangle$
  $\mathbf{r}_0 \leftarrow \text{LowBits}(\mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle)$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ *or* $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
  else
    $\langle\langle c\mathbf{t}_0 \rangle\rangle \leftarrow \text{INTT}(\hat{c} \circ \hat{\mathbf{t}}_0)$
    $\mathbf{h} \leftarrow \text{MakeHint}(-\langle\langle c\mathbf{t}_0 \rangle\rangle, \mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle + \langle\langle c\mathbf{t}_0 \rangle\rangle)$
    if $\|\langle\langle c\mathbf{t}_0 \rangle\rangle\|_\infty \geq \gamma_2$ *or the number of 1s in* $\mathbf{h}$ *is greater than* ω **then**
      $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
    end
  end
  $\kappa \leftarrow \kappa + \ell$
end
$\sigma \leftarrow \text{sigEncode}(\tilde{c}, \mathbf{z} \mod q, \mathbf{h})$
return σ

Algorithm 3: ML-DSA verifying [4].

Input : Public key pk and message M'. Signature σ
Output: Boolean

$(\rho, \mathbf{t}_1) \leftarrow \text{pkDecode}(pk)$
$(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \text{sigDecode}(\sigma)$
if $mathbfh = \perp$ **then**
  return false
end
$\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ `// A is generated and stored in NTT representation`
$tr \leftarrow \text{H}(pk, 64)$
$\mu \leftarrow \text{H}(\text{BytesToBits}(tr)||M', 64)$
$c \leftarrow \text{SampleInBal}(\tilde{c})$
$\mathbf{w}'_{approx} \leftarrow \text{INTT}\left(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}\left(\mathbf{t}_1 \cdot 2^d\right)\right)$
`//` $\mathbf{w}'_{approx} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
$\mathbf{w}'_1 \leftarrow \text{UseHint}\left(\mathbf{h}, \mathbf{w}'_{approx}\right)$
$\tilde{c}' \leftarrow \text{H}\left(\mu||\text{w1Encode}\left(\mathbf{w}'_1\right), \lambda/4\right)$
return $\left[\|\mathbf{z}\|_\infty < \gamma_1 - \beta\right]$ and $\left[\tilde{c} = \tilde{c}'\right]$

For ML-DSA and similar schemes the polynomial are however in the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. The same trick can be applied when using a negative wrapped convolution (also named negacyclic convolution) which is defined as:

$$NWC(x) = \sum_{k=0}^{n-1} c_k x^k = Y(x) \mod (x^n + 1) \tag{2.5}$$

$$\text{where} \quad c_k = \sum_{i=0}^{k} g_i g_{k-i} - \sum_{i=k+1}^{n-1} g_i h_{k+n-i} \mod q \tag{2.6}$$

As a next step, the convolution is reduced to a pointwise multiplication in the NTT domain, similar to how one can convert a convolution into a pointwise multiplication in the Fourier domain. But for this, the polynomial coefficients must be first converted into the NTT domain. The NTT to compute a negative wrapped convolution requires the existence of a primitive $2n$-th root of unity ϕ (the root of unity for a DFT is $e^{i2\pi\frac{k}{N}n}$). A polynomial a can be transformed into the NTT domain with Definition 2.1 and back with Definition 2.2.

Definition 2.1 (NTT)**.** The Negative-Wrapped Number Theoretic Transform of a vector of polynomial coefficients $\mathbf{a}$ is defined as $\hat{\mathbf{a}} = \text{NTT}(\mathbf{a})$, where:

$$\hat{\mathbf{a}}_j = \sum_{i=0}^{n-1} \phi^{2ij+i} a_i \mod q \tag{2.7}$$

and $j = 0,1,2,\ldots,n-1$. The factors ϕ^{2ij+i} are called twiddle factors.

Definition 2.2 (INTT)**.** The Negative-Wrapped Inverse Number Theoretic Transform (INTT) of an NTT vector $\hat{\mathbf{a}}$ is defined as $\mathbf{a} = \text{INTT}(\hat{\mathbf{a}})$, where:

$$\mathbf{a}_i = n^{-1} \sum_{j=0}^{n-1} \phi^{-(2ij+j)} \hat{a}_j \mod q \tag{2.8}$$

and $i = 0,1,2,\ldots,n-1$.

The NTT and the INTT only differ in the scaling factor n^{-1} and the transposed ϕ elements.

A negative wrapped convolution c of polynomials $a, b \in \mathcal{R}_q$ can therefore be computed as a pointwise multiplication in the NTT domain as

$$\mathbf{c} = \text{INTT}\left(\text{NTT}(\mathbf{a}) \circ \text{NTT}(\mathbf{b})\right) \tag{2.9}$$

where $\circ$ is an elementwise vector multiplication in $\mathbb{Z}_q$.

These transforms still have a complexity of $\mathcal{O}(n^2)$. To reduce the computation complexity down to $\mathcal{O}(n \log n)$, the final trick is to apply the FFT algorithm to divide and conquer the computation from an NTT of size $2n$ into two NTT of size n and so forth. This is possible because the selected $2n$-th root of unity ϕ is periodic ($\phi^{k+2n} = \phi^k$) and symmetric ($\phi^{k+n} = -\phi^k$). An NTT or INTT for ML-DSA where $n = 256$ and $q = 8380417$ can thus be split into 8 so called *layers* of butterflies. Figure 2.1 shows the layers for an example where $n = 8$. For the NTT, the Cooley-Tukey (CT) butterfly (Figure 2.2 left) is used whereas for the INTT the similar Gentleman-Sande (GS) butterfly (Figure 2.2 right) is used. Using the FFT approach results that the transformed vector $\hat{\mathbf{a}}$ is ordered in bit-reversed order. For the cryptographic usage of the NTT, this does not matter as the transformed values do not have any meaning (compared to the frequency for a DFT) and the INTT expects the input in bit-reversed order.

2.4.2. NTT implementation

From an implementation viewpoint, there exist multiple algorithms to perform the FFT-based NTT and INTT. Whereas some apply a recursive approach for each element there exist iterative algorithms which compute each layer sequentially. The benefit of such iterative algorithms is that they can operate in-place, reducing the memory footprint significantly. In addition, all butterflies of a layer are fully independent, and therefore its computation can easily be vectorized. For executing vectorized computations SIMD instructions are strongly beneficial and thus are the main interest for this work. Another optimization is to precompute the twiddles factors and load the values from memory during computation which trades increased code size for a reduced execution time. Two of such implementations can be found in [4] or in [12]. The NTT and INTT algorithms of [12] are Algorithm 4 and Algorithm 5, respectively. These are used to benchmark the implementation of this work (see Chapter 5).

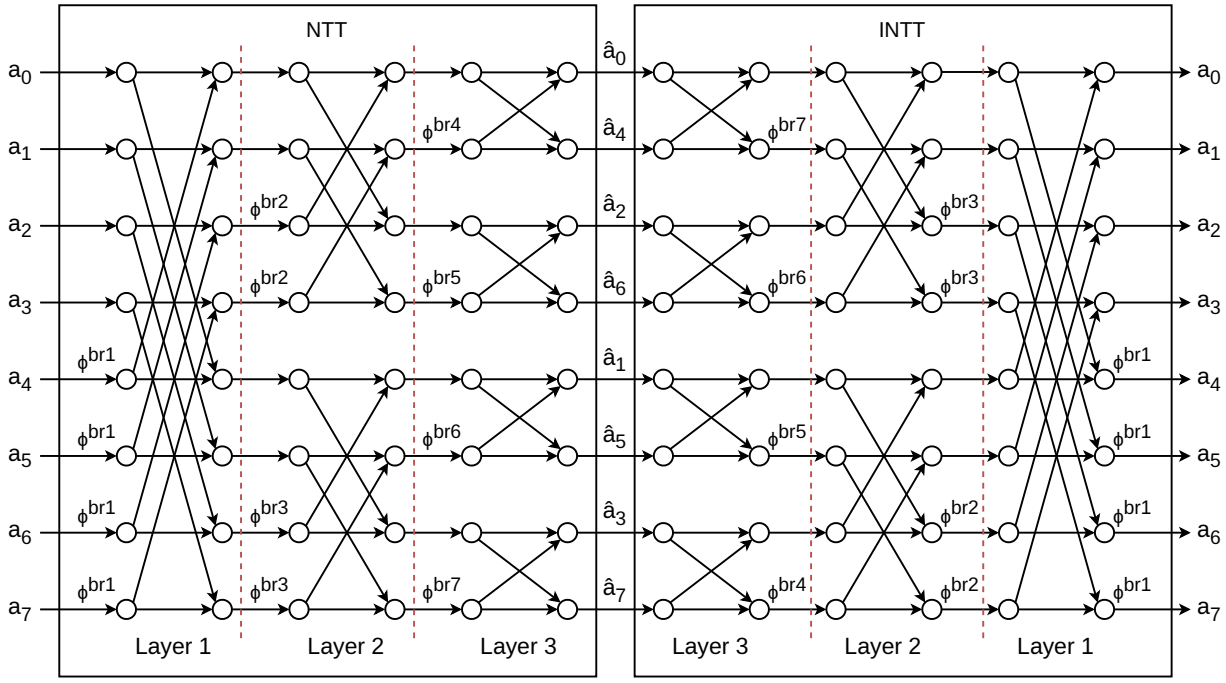

Figure 2.1.: A NTT and INTT with the butterflies for $n = 8$. The twiddle factors ϕ are indexed with the bit-reversed order. The scaling factor at the end of the INTT of n^{-1} is omitted.

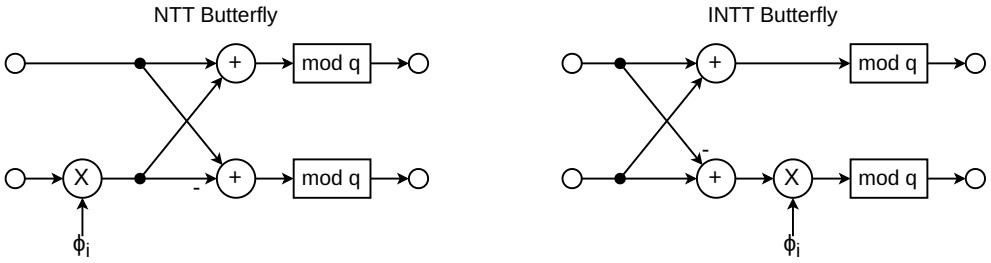

Figure 2.2.: Left: Cooley-Tukey butterfly. Right: Gentleman-Sande butterfly.

Algorithm 4: NTT algorithm from [12].

Input : $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ with q prime and $q \equiv 1 \mod 2n$, $n = 2^k$ for $k \in \mathbb{N}$, precomputed table Φ build from powers of ϕ in bit-reversed order, for ϕ a $2n$-th root of unity

Output: $\hat{\mathbf{a}} \leftarrow \mathrm{NTT}(\mathbf{a})$ inplace and in bit-reversed order

```
t ← n for m = 1; m < n; m = 2n do
    t ← t/2
    for i = 0; i < m; i++ do
        j1 ← 2it
        j2 ← j2 + t − 1
        S ← Φ[m + i]
        for j = j1; j ≤ j2; j++ do
            U ← a_j
            V ← a_{j+t} · S  mod q
            a_j ← U + V  mod q
            a_{j+t} ← U − V  mod q
        end
    end
end
return a
```

2.5. Modular multiplication algorithms

Most of the discussed operations for ML-DSA operate over a finite field and therefore theoretically require modular reduction after each operation. To compute a modular multiplication efficiently without long trial divisions, well-known algorithms are the Montgomery multiplication [18] or Plantard multiplication [19]. There also exists an optimized version that works with signed integers [20]. This is used in the official ML-DSA reference implementation. However, in the context of OTBN, unsigned values are much better supported because the ISA lacks support for sign extension. Due to this reason, only the unsigned Montgomery and Plantard algorithms are considered for this work. Both of these methods require the multiplication operands to be transformed in a special representation prior to computing the multiplication. This transform is an expensive computation and thus these methods are only useful if values must be multiplied several times before transforming the value back which is exactly the case when computing an NTT. The additions and subtractions of a butterfly can be directly performed in this special representation and therefore the NTT coefficients and the final results must only be transformed once.

The Montgomery multiplication described in Algorithm 6 actually computes

$$r = a \cdot b(2^{-d}) \mod q \tag{2.10}$$

where d is the bit width of the operands a and b. For a correct result, the multiplication

Algorithm 5: INTT algorithm from [12].

Input : $\hat{\mathbf{a}} = (\hat{a}_0, \ldots, \hat{a}_{n-1}) \in \mathbb{Z}_q^n$ in bit-reversed order with q prime and $q \equiv 1 \mod 2n$, $n = 2^k$ for $k \in \mathbb{N}$, precomputed table Φ^{-1} build from powers of ϕ^{-1} in bit-reversed order, for ϕ a $2n$-th root of unity

Output: $\mathbf{a} \leftarrow \mathrm{NTT}(\hat{\mathbf{a}})$ inplace and in bit-reversed order

$t \leftarrow 1$ **for** $m = n;\ m > 1;\ m = m/2$ **do**
  $j_1 \leftarrow 0$
  $h \leftarrow m/2$
  for $i = 0;\ i < h;\ i++$ **do**
    $j_2 \leftarrow j_1 + t - 1$
    $S \leftarrow \Phi^{-1}[h + i]$
    for $j = j_1;\ j \leq j_2;\ j++$ **do**
      $U \leftarrow a_j$
      $V \leftarrow a_{j+t}$
      $a_j \leftarrow U + V \mod q$
      $a_{j+t} \leftarrow (U - V) \cdot S \mod q$
    end
    $j_1 \leftarrow j_1 + 2t$
  end
  $t \leftarrow 2t$
end
for $j = 0;\ j < n;\ j++$ **do**
  $a_j = a_j \cdot n^{-1} \mod q$
end
return a

operands must be converted in a special representation called the q-residue by computing

$$c_{\text{Montgomery}} = a_{\text{Normal}} \cdot 2^d \mod q. \tag{2.11}$$

As this special representation is based upon a power of two the modulo and divisions operations required can be implemented efficiently with bit masking and shifting. The result r is in the q-residue representation and can be transformed back by applying the Montgomery multiplication with the second input being 1. An alternative approach to obtain a correct result directly in the normal representation is to transform only one of the inputs into the Montgomery representation and then perform the Montgomery multiplication. The result then is already in the normal representation as the factor 2^d multiplied to the one operand is canceled with the 2^{-d} during the computation. This approach is especially useful when computing an NTT because the twiddle factors can be stored in the Montgomery representation and thus no transformations at all are required at runtime. A more comprehensive tutorial can be found in [21].

Algorithm 6: Unsigned Montgomery multiplication [18].

Input : $a, b \in [0, q]$, $q \in]0, 2^d[$, $R = (-q^{-1}) \mod 2^d$
Output: $r = ab(2^{-d}) \mod q$ and $r \in [0, q]$

1 $c = ab$
2 $r = \left[c + \left[[c]_d R\right]_d q\right]^d$
3 **if** $r \geq q$ **then**
4 | **return** $r - q$
5 **end**
6 **return** r

The Plantard multiplication [19] in Algorithm 7 is similar to the Montgomery multiplication except it uses a different residue representation for the operands and result. This representation sets the conversion factor to -2^{-2d} and thus a value in the Plantard representation requires $2d$ bits. The benefit of using the Plantard representation is that the computation can be altered to have no conditional subtraction and, additionally, if one input, e.g. b, is constant the multiplication with R can be pre-computed. This saves one multiplication compared to the Montgomery algorithm in cost of increased memory requirements when storing the precomputed value. The trick of transforming only one input also applies for Plantard and for an NTT computation is preferably applied to the twiddle factors.

2.6. OpenTitan

The OpenTitan project represents a pioneering initiative, stewarded by lowRISC, in the realm of open-source hardware with a primary focus on the design and implementation of a transparent, ven, and high-quality silicon root of trust (RoT) ecosystem. Verifiable

Algorithm 7: Plantard multiplication [19].

Input : $a, b \in [0, q]$, $q < \frac{2^2}{\phi}$, $\phi = \frac{1+\sqrt{5}}{2}$, $R = q^{-1} \mod 2^{2d}$

Output: $r = ab(-2^{-2d}) \mod q$ and $r \in [0, q]$

1 $r = \left[\left(\left[[abR]_{2d} \right]^d + 1 \right) q \right]^d$

2 **return** r

and robust RoT systems are a critical component in modern computing systems responsible for ensuring the integrity and confidentiality of sensitive operations which includes essential functions such as secure boot, cryptographic key management, and protection of sensitive data. OpenTitan's open-source approach ensures that its design can be independently inspected and audited leading to security through transparency [22].

The OpenTitan project provides not only complete solutions like the standalone microcontroller *EarlGrey* or an integrated execution environment *Darjeeling* but it also bundles a variety of hardware IPs. These blocks cover the range from cryptographic operations like Advanced Encryption Standard (AES) or Keccak computations, secure random number generators, and a big-number co-processor OTBN designed to accelerate asymmetric cryptography like RSA or elliptic curve cryptography [22]. The main processor of the *EarlGrey* chip is the Ibex core, an open-source 32-bit RISC-V implementing the Integer (I), Integer Multiplication-Division (M), Compressed (C) and Bit Manipulation (B) instruction sets. However, currently used cryptographic algorithms like RSA are often based upon big numbers and the Ibex core with its 32-bit architecture is rather inefficient in executing these arithmetic operations. To improve the execution performance of these algorithms, the OpenTitan project features a dedicated security hardened big number core named OpenTitan Big-Number Accelerator (OTBN). This allows the Ibex core to offload computations to improve the overall runtime.

2.6.1. OpenTitan Big-Number Accelerator

The OTBN is a self-contained co-processor specialized for executing security-sensitive asymmetric cryptography code where big numbers are prevalent. Such big-number arithmetic operations are supported by 256-bit wide registers and a 256-bit data path which can be operated on with specialized instructions. To reduce the possible data leakage the control flow is isolated with separate 32-bit registers and data path [23].

In the OpenTitan context, the main processor can offload a computation to the OTBN by writing the desired application and data into OTBN's instruction and data memory, respectively. It then can start the execution on the OTBN and can retrieve the results afterward by reading from the data memory. During the execution, the OTBN cannot be interrupted except if it encounters an error [23].

The OTBN design is kept simplistic to reduce the attack surface and includes many security measures against side-channel analysis (SCA) and fault injection (FI) attacks. For example, all branches and conditional jumps execute in constant time and there is no

data caching. The instruction and data memory are protected by scrambling the content and most of the data path implements integrity protection codes which can detect at least three bits per 32-bit word. On RTL level, register blanking is applied which is described in more detail in Section 2.6.2 [23].

The following sections summarize and highlight technical details which are relevant to this work. A complete documentation can be found in [23].

Architecture

The OTBN is based on the Harvard architecture and is built around a 32-bit wide instruction (8 KiB) and a 256-bit wide data memory (4 KiB). Figure 2.3 depicts the hardware block level architecture where two data paths (32-bit and 256-bit) are visible. The purple part is the control flow which is based on 32 32-bit registers named GPR (general purpose register) and the computations are implemented in the arithmetic logic unit (ALU) named Base ALU. The green part is the 256-bit wide data path based on 32 256-bit registers named WDR (wide data registers) which are fed into the two special ALU modules Bignum ALU (BN ALU) and Bignum MAC (BN MAC) (multiply and accumulate) to compute big-number arithmetics. In addition to the GPRs and WDRs, there exist so called CSRs (control and status register) and WSRs (wide special purpose register) which can be accessed by both data paths. The BN ALU implements basic arithmetic functions like addition, shifting, and logic operations (AND, XOR, etc.) as well as specialized pseudo-modulo instructions (see instruction set description below). To implement these pseudo-modulo instructions the BN ALU contains the MOD WSR, which is used to store the modulus value, and two CSRs storing the arithmetic flags. Multiplication is handled in the BN MAC which contains a 64-bit multiplier. A 256-bit multiplication therefore must be executed using the schoolbook algorithm and accumulating the partial products. The accumulation is enabled by the ACC WSR inside the BN MAC. A third WSR is in the interface to the entropy distribution network (EDN) which provides the OTBN with either true random values or values from a XoShiRo256++ PRNG (pseudo random number generator).

Due to security hardness, the instructions for both data paths are designed as single cycle execution and the instruction pipeline is thus kept very simplistic and consists only of an instruction fetch and a combined decode-execute stage. The controller is responsible for steering the pipeline and coordinating the different modules. To improve the performance the OTBN prefetches instructions but never speculatively to enforce constant time execution.

Instruction set

The OTBN features two different instruction sets, one for each data path. The base instruction set operates on the 32-bit path and is similar to the RISC-V RV32I instruction set. It includes a hardware loop instruction and is primarly used to manage the control flow of an application and its hardware implementation is not security hardened. The big-number instruction set operates on the WDRs and does not include any control flow

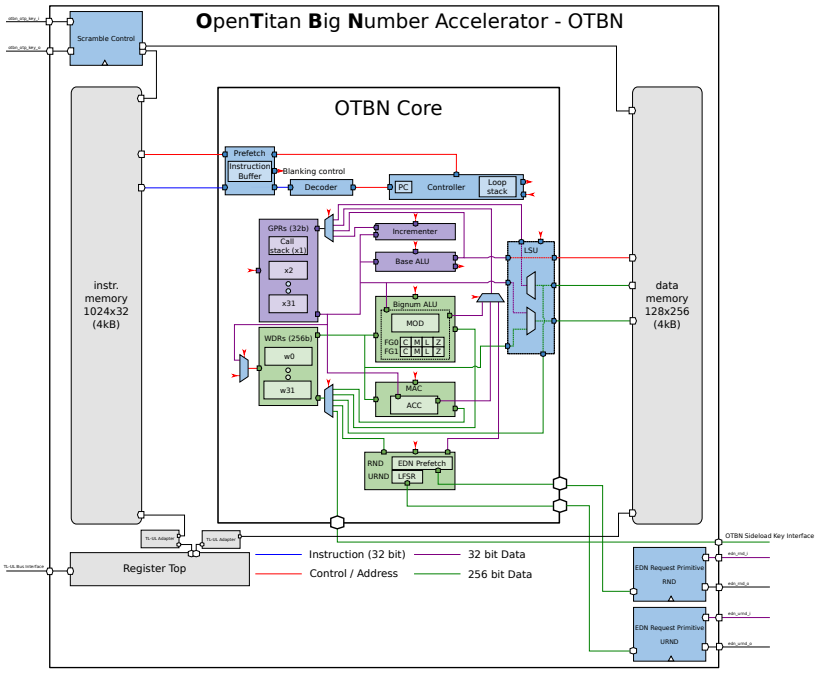

Figure 2.3.: The OTBN architecture at hardware block level [23].

instructions, but its implementation includes many security hardening features. Note, that the big-number instructions do not support signed numbers natively as there is no sign extension capability. As such, the primary use lies in computations with unsigned integers. This instruction set contains some rather uncommon instructions specifically designed for cryptographic workloads which are explained in the following. Also, note that there are no SIMD instructions available. For a complete description see [23].

- `bn.addm`: The "pseudo-modulo" addition sums two WDRs and if the sum is greater than the value in the MOD WSR then MOD is subtracted before writing it back to the destination WDR.
- `bn.subm`: Same as bn.addm but with subtraction and MOD is added if the sum is less than zero.
- `bn.mulqacc[.so, .wo]`: Multiplies two quarter WDRs (selectable 64-bit part of a WDR) and accumulates the product in the ACC WSR. The product can be shifted prior to the accumulation. Optionally the ACC WSR can be zeroed. There exists also the variants bn.mulqacc.wo and bn.mulqacc.so which write back the full (.wo) or half (.so) final value of ACC into a destination WDR.
- `bn.rshi`: Concatenates two WDRs and then right shifts the 512-bit value. The result is truncated to 256 bits and written back into a destination WDR.

Instruction set simulator and register transfer level (RTL) simulation

For development and testing there exists a cycle accurate OTBN Python simulator and a complete Verilator model for the RTL implementation. These two simulations are used to check the RTL implementation by running them in parallel (co-simulation) and comparing their execution traces and register updates. The Python simulator acts as golden model and can report also statistics such as cycle count, stalls, and a histogram of used instructions and function calls. In this work, the simulator is used to check parts of the RTL implementation and also to perform software benchmarks to analyze the design changes.

2.6.2. Register blanking

Security is essential for the OTBN, and thus, the implementation contains many security hardening features. Besides the already mentioned mitigations like memory scrambling, the data path integrity checks, no caches, no branch prediction, and some others (see [23]), the main feature of the microarchitecture implementation is register blanking. Register blanking forces data and control paths that are unused by the current instruction to zero to reduce the power signature generated by secret data bits. In the following, the concept as it is implemented in the OTBN is explained on behalf of the register file of the Ibex core as is covered in [24]. The Ibex pipeline and register file are very similar to the OTBN and the explanation can be directly related.

A register file in a simple processor like OTBN or Ibex is a great source of cryptographic leakage if no blanking mechanism would be implemented as explained in [24]. The left part of Figure 2.4 shows the Ibex register file read and write ports without blanking mechanisms. To read a register, a MUX tree selects the desired register which is controlled with 1-bit signals representing the register address. According to [24] this implementation however raises the following problems:

- Switching wires in MUX tree: When reading registers in consecutive cycles the wire between two MUX levels can change its value. For example, reading the register `x3` in the first cycle and `x4` in the second will change the 5th bit of the address signals. The wire between the MUX of `x1` and `x2` and the L1 MUX will change from the data in `x1` to `x2`. If `x1` and `x2` contain parts of a secret this switching generates power leakage which can be used to reconstruct the content of `x1` and `x2`.

- Glitchy address signals: When decoding a register read or write instruction the address signals may glitch for a short amount of time. This glitching results in the same transition of data as explained in the previous point.

Solving these problems in software is difficult or even impossible depending on the exact hardware implementation. As a solution, the authors in [24] propose a hardware gating mechanism as shown in the right of Figure 2.4. The MUX structure is replaced by AND gates for each register which are controlled by a onehot signal. The desired register

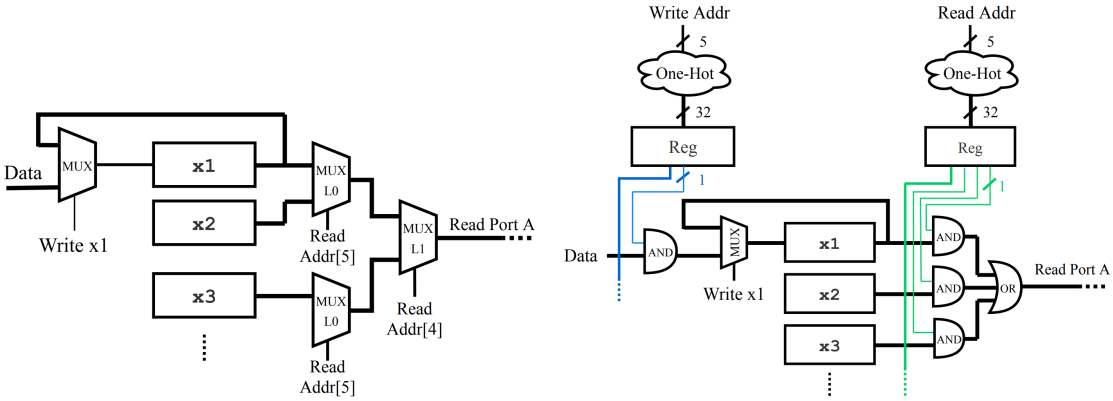

Figure 2.4.: *Left*: The original register file of the Ibex core. A multiplexer tree is used to read registers based on the 5-bit read address. Writing is done via a multiplexer, controlled by a 1-bit write-enable signal, which is derived from the write address.
Right: Secured register file. The register output is additionally gated and the multiplexer tree is replaced by a tree of OR gates. The writing mechanism remains unchanged, except that it is extended by an additional AND gate for the write data. From [24], modified.

value is then fed through a cascade of OR gates to the read port. With this structure, hereinafter referred to as a onehot MUX, at most one bit is set due to the onehot control signal and therefore at most one register is read. This solves the first problem but the glitching would still occur as the naive implementation generates the MUX control signals in the decode-execute stage of the pipeline. To solve this problem, the onehot control signal is flopped such that it is already stable at the beginning of the clock cycle / when the instruction enters the decode-execute stage which in turn requires that the control signals are generated in the preceding pipeline stage.

The described problem does not only apply to the register file. It also applies to any data path that is unused by the current instruction. For example, for an add instruction the unused shifter inside the OTBN's BN ALU may not receive the values of the source registers as the shifting logic then would generate leakage. To prevent this leakage all unused data paths must be tied to zero which can be implemented by adding a so called *blanker* into the path. A blanker is similar to the register gating mechanism but only includes the ANDing part and no multiplexing. Each bit of the to be blanked path is connected to a 2-input AND gate where the second input is a predecoded control signal. Now the data can only propagate if the control signal is set and otherwise the path is blanked, i.e., no data can propagate and no leakage is generated [24].

For the OTBN only the WDR register file, WSRs, the BN ALU, and BN MAC modules implement such blanking mitigations. The base registers and ALU is not protected. For additional redundancy against FI attacks, the decoder generates the control signals for the register files and blankers in parallel to the predecoder. These signals are then

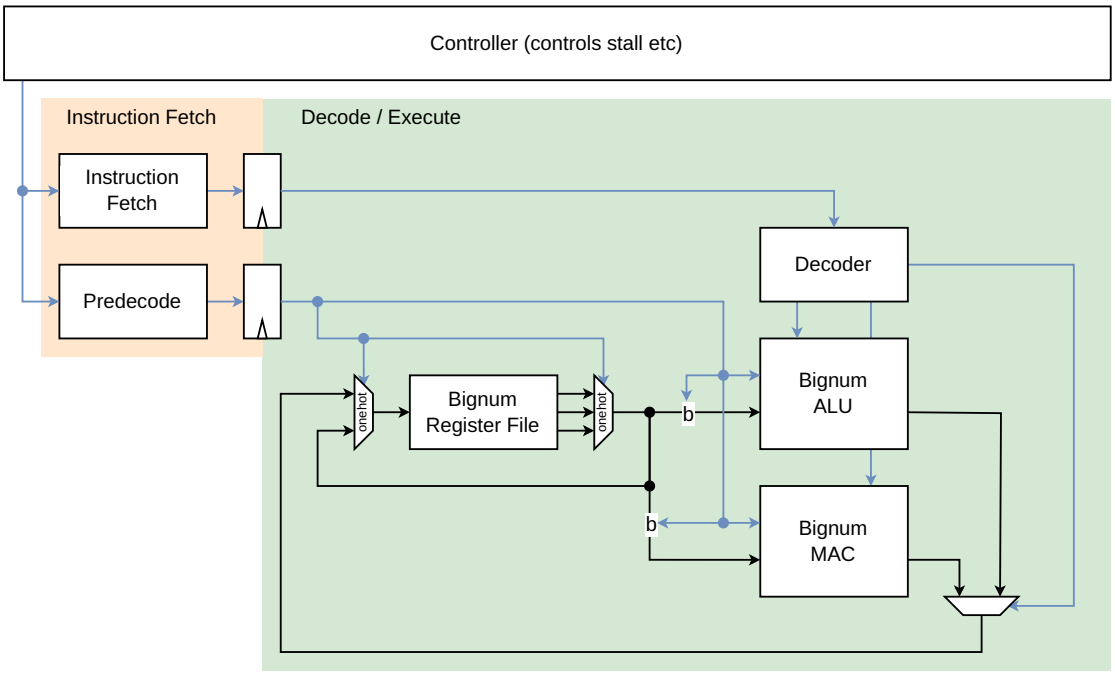

Figure 2.5.: Simplified pipeline of OTBN to highlight the blanking implementation. The predecoded signals generated in the predecoder are flopped between the instruction fetch and decode-execute stage. The register file uses onehot controlled AND-OR structures and the lowercase 'b' represents blankers on the data paths.

compared to the predecoded values and a fatal error is raised if these do not match [23]. With the predecoding, the OTBN pipeline can be thought of as depicted in Figure 2.5. It is a drastic simplification but captures the relevant parts. Namely, the predecoded control signals are flopped between the instruction fetch and decode-execute stage and the WDR register file is accessed with a onehot structure. The lowercase 'b' represents blankers in the unused data paths which there are many more inside the BN ALU and BN MAC module.

Chapter 3

Baseline architecture

This chapter presents different implementation strategies for PQC schemes found in literature and discusses their advantages and disadvantages as well as their applicability to OTBN. Finally, an ISA extension is proposed which serves as a baseline for further optimizations. The implementation thereof is documented in Chapter 4.

3.1. Literature review

All of the reviewed literature [5, 6, 7, 8, 9, 10, 11, 12] can be assigned to one of the following strategies:

- Dataflow implementation: Proposals of implementations of ML-DSA and similar algorithms in a streaming and pipelined manner. These designs target field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC) solutions and are very specialized for one cryptographic algorithm. This approach does not fit well into the OTBN context as a generic extension is targeted. However, some optimization tricks may be useful.
- Co-processor: These propose to design a processor similar to OTBN. Most of them are however very specialized but simpler processors. These may contain ideas for new instructions.
- ISA extension: Proposals of new instructions for platforms RISC-V like processors or directly the OTBN. These extensions range from very generic SIMD to highly specialized butterfly instructions. These are the most interesting strategies.

In the following, each work is summarized and its key points are highlighted. The reviews are ordered in increasing applicability starting with full dataflow implementations and ending with the ISA extension proposals. A more comprehensive overview of the literature including performance and area metrics can be found in [25].[1]

[1] See `artifacts-for-report/literature-research.ods`

A Compact and High-Performance Hardware Architecture for CRYSTALS Dilithium [5]

This work proposes a fully specialized, segmented, and pipelined dataflow for FPGAs. Key functions of ML-DSA are implemented in modules that are pipelined to reduce intermediate result storage requirements. It includes a special NTT module which is pipelined and based on 4 parallel butterfly units which allows to compute 8 butterflies in a time sliced fashion. The rejection sampling module computes the matrix A on-the-fly to reduce the memory footprint. Due to its dataflow architecture and high specialization, the proposed ideas are not directly applicable to the OTBN.

Polynomial Multiplication for Post-Quantum Cryptography [6]

This PhD thesis provides a solid introduction to modular reduction and multiplication algorithms and then expands on different NTT types (on NTT-friendly and unfriendly rings) and shows how to implement these in the most efficient way on Cortex-M4 and M3 processors. These processors are all 32-bit based and many proposed implementations make use of the inherited wrap around. The described techniques are therefore only of partial use.

Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium [7]

Describes a hardware accelerator for ML-DSA for either standalone use or integration as a co-processor. Internally it contains modules each implementing a subfunction of ML-DSA which are controlled by a global finite state machine (FSM). These modules all communicate through a central main memory to compute the ML-DSA in a streamed and pipelined manner. Modular reduction is performed by converting the reduction into a series of 2 to 12-bit additions, bit shiftings, and one multiplication. This is possible due to the special property of the ML-DSA modulus which can be split recursively. Similar to the previously discussed works this is a fully specialized accelerator and therefore is of less interest. The proposed modulo reduction module might be of interest for a possible optimization but probably results in a very long critical path.

A Highly-efficient Lattice-based Post-Quantum Cryptography Processor for IoT Applications [9]

This work describes a custom processor with SIMD registers, a NTT ALU module called PALU and a custom instruction set. It is based on the CV32E40P processor (RISC-V, in-order 4 stage pipeline) and has 5 special SIMD capable registers of 16 64-bit values each where the data elements are interpreted as 32-bit values. Next to the specialized NTT ALU instructions, the processors feature specialized instructions for sampling of centered binomial distributions, rejection sampling, and Keccak computation (xor and rotate combinations) which operate on the SIMD registers. The NTT focused PALU consists of 8 processing units each of these features a multiplier, two adders and modular

reduction circuitry perfectly matching the butterfly operation. It includes additional functionality to perform vector element shuffling to optimize the data alignment inside the SIMD registers. This enables efficient NTT layer computations as the stride between inputs to the butterfly operation shrinks with each layer. The implementation of ML-DSA is reported to result in about 40 KiB of code and requires up to 64 KiB for the highest level of ML-DSA. Despite its promising performance, the proposed ideas are actually only of partial use because the PALU and other instructions are very specific to ML-DSA and some of the functionality can be solved using the OpenTitan KMAC IP. However, an NTT module may be of interest.

ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V [8]

The main contribution is a RISC-V ISA extension proposal targeting two other PQC schemes namely Kyber [26] (standardized in 2024 as ML-KEM [27]) and NewHope [28]. ML-DSA is not directly considered but is very similar in computational aspects. The proposed instructions provide arithmetic operations (addition, subtraction, multiplication, and reduction) directly on finite fields that operate on the 32-bit registers, i.e. these are Single Instruction Single Data (SISD) instructions. The finite field reduction is based upon the Barrett reduction [29] and is implemented in the memory and writeback stage of the RISC-V pipeline. In addition to the ISA extension, an on-the-fly NTT twiddle factor computation is explained. This technique may be of interest if the instruction memory footprint of an application must be reduced but comes with a high performance penalty. Regarding OTBN, the described ideas are less applicable because the execution of the instructions is spread over the pipeline stages whereas the OTBN does only have a combined decode-execution stage. In addition, only the OTBN's 256-bit data path includes security features (extending the 32-bit path is infeasible due to the inferred costs in area and verification) and possible SIMD opportunities cannot be exploited with the presented idea.

PQ.V.ALU.E: Post-Quantum RISC-V Custom ALU Extensions on Dilithium and Kyber [10]

This work presents finite field operations for a RISC-V core similar to [8] which are implemented in a custom ALU module. Its difference is that all instructions are implemented in the execution pipeline stage and features a single cycle butterfly operation. The proposed butterfly implementation could easily be adapted to support SIMD functionality operating on OTBN's WDR. However, to support single cycle execution, three read and two write ports to the Bignum register file are required (reading two inputs and a twiddle factor, writing back the two results). The current OTBN only features two read and one write port. Extending the register file seems infeasible due to the area increase driven by extending the register file and the additionally required blanking functionality, see Section 2.6.2. On the other hand, the gained speedup is estimated to be around a factor of 3 because a butterfly requires at least 3 instructions (one multiplication, one

addition and one subtraction).

Towards ML-KEM & ML-DSA on OpenTitan [11]

This work describes an implementation of ML-DSA on the OTBN and is based on a master thesis [12]. First, a native implementation using the default OTBN is described and profiled. As explained in Chapter 2, this work also identifies the bottlenecks as Keccak computation and the NTT. To improve the performance, the authors present the following two optimizations. For the Keccak computation, they propose to interface the KMAC IP block whereas for the NTT a simple and generic SIMD ISA extension is proposed. These new instructions operate on the WDRs and include vectorized element wise addition, subtraction, multiplication, bit shifting, and element shuffling. All arithmetic instructions come also with a modulo variant where addition and subtraction is only a pseudo reduction similar to the existing `bn.add / bn.sub` instructions (see Section 2.6.1) and the multiplication is a direct hardware implementation of the Montgomery multiplication. In regard of the implementation, the approach is to reuse as much of the existing OTBN resources to keep the area overhead to a minimum. Reusing resources like adders for all instructions is possible with minor modifications to the existing logic except for the multiplication. The solution for multiplication is more involved and two approaches are described. The simpler to implement approach describes a new ALU module that only handles the vectorized (modulo) multiplication and co-exists with the Bignum ALU and Bignum MAC modules. Whilst this approach requires only simple control logic and provides single cycle multiplications the required area is huge as it doubles the OTBN area as the new module includes three multipliers. Alternatively, it is proposed to reuse the existing multiplier of the Bignum MAC module and convert the multiplication to a multi-cycle instruction. This way only smaller additions to the Bignum MAC module are required and the area overhead reduces to about 12% of the OTBN, but the control logic becomes more sophisticated. Similar to [9] the ML-DSA results in a code size of approximately 32 KiB and requires up to 124 KiB of DMEM. Overall, this work proposes a reasonable ISA extension that can fulfill the initial requirements of a generic and relatively lightweight extension.

3.2. ISA extension proposal

As discussed in the previous section, most of the reviewed literature focuses on either a very specific solution or suggests a dataflow implementation. Others describe ISA extensions which are legitimate but target an instruction pipeline with more stages than OTBN's pipeline or are only useful for a 32-bit register architecture. The work of [9] describes an interesting approach with its 5 SIMD registers and specialized PALU module. However, the suggested implementation with 8 separate cores (including 8 multipliers) probably results in a similar infeasible area increase as the single cycle approach described in [11]. Lastly, the idea in [11] matches nicely with the project's goals as it provides instructions of generic character. The less area intensive but less performant

approach of reusing the existing multiplier presents a suitable area to performance tradeoff as achieving the best possible performance is not the goal of this work.

In regard to instruction memory and data memory, all processor related papers report high requirements with up to 124 KiB of data memory and 32 KiB of instruction memory for the highest ML-DSA security level. These values certainly exceed the current memory size of 8 KiB and 4 KiB for instructions and data, respectively. However, this problem is out of scope for this work as there are plenty of possible solutions. The most basic solution would be to increase the memory sizes but other ideas like using other memory regions already present in an OpenTitan system or only offloading parts of the ML-DSA like the NTT to the OTBN are also possible approaches. Also, the solution depends on the capabilities of refactoring the implementation by making use of the new extension and thus should be investigated in a follow up work.

Given the rationale just elaborated, the ISA extension with the multiplier reuse strategy described in [11] is chosen as a baseline design with extended SIMD element length support. This design introduces five new types of instruction to the OTBN. All these instructions operate on the WDRs and interpret the 256-bits as a vector of unsigned elements of configurable bit widths. In the following, these instructions are described in more detail and the element length i.e., bit width, is specified with the parameter `<elen>` which either can be `.2Q` for 128-bit elements, `.4D` for 64-bit and `.8S` or `.16H` representing 32-bits or 16-bits elements, respectively. The only exception is the multiplication instruction `bn.mulv(m)` which only supports the 32-bit or 16-bit (`.8S` or `.16H`) variants due to constraints given by reusing the existing multiplier (see explanation given in Section 4.2.4). Arithmetic operations also include a variant where a (pseudo) modulo reduction is performed. These variants are differentiated by adding a `m` as suffix. In this case, the modulus value is read from the MOD WSR similar to the already existing `bn.addm` and `bn.subm` instructions expect it (see Section 2.6.1).

- `bn.addv(m).<elen> <wdr>, <wrs1>, <wrs2>`: Add the vector elements in WDRs `<wrs1>` and `<wrs2>` element wise and store the result in the WDR `<wdr>`. The results are truncated in case of an overflow. If the modulo variant is selected a pseudo reduction is performed, meaning if an individual result is equal to or larger than MOD, MOD is subtracted from it.

- `bn.subv(m).<elen> <wdr>, <wrs1>, <wrs2>`: Subtract the vector elements in WDR register `<wrs2>` from `<wrs1>` element wise and store the result in the WDR `<wdr>`. The results are truncated to the element length. If the modulo variant is selected a pseudo reduction is performed meaning if an individual result is negative, MOD is added to it.

- `bn.mulv(m)(l).<elen> <wdr>, <wrs1>, <wrs2>[, <lane>]`: Multiply elements in WDRs `<wrs1>` and `<wrs2>` element wise and store the result in the WDR `<wdr>`. The results are truncated to the element length. This instruction supports only element lengths of type `.8S` or `.16H`. The suffix `l` specifies a lane wise operation where all elements of `<wrs1>` are multiplied with a fixed element in `<wrs2>` at the index specified by `<lane>`. This applies to both the regular and modulo multiplication. If

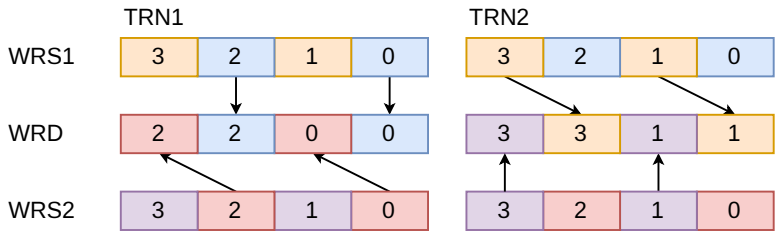

Figure 3.1.: An illustration of the proposed instructions `bn.trn1` and `bn.trn2` for vectors with four elements.
Left: The `bn.trn1` places even-indexed vector elements from `<wrs1>` into even-indexed elements of `<wrd>` and even-indexed vector elements from `<wrs2>` are placed into odd-indexed elements of `<wrd>`.
Right: For `bn.trn2` odd-indexed vector elements from `<wrs1>` are placed into even-indexed elements of `<wrd>` and odd-indexed vector elements from `<wrs2>` are placed into odd-indexed elements of `<wrd>`

the modulo variant is selected instead of a regular multiplication a Montgomery multiplication is performed for all elements. This requires the modulus value and the corresponding element length's Montgomery constant to be placed in the MOD WSR. The input operands must be transformed into the Montgomery representation accordingly before executing this instruction. Note, that due to the chosen strategy of reusing the existing multiplier, this instruction executes over multiple cycles. See Section 2.5 and Section 4.2.4 for a detailed explanation.

- `bn.trn1/bn.trn2.<elen> <wdr>, <wrs1>, <wrs2>`: Interleaves the vectors in `<wrs1>` and `<wrs2>` as illustrated in Figure 3.1. The `bn.trn1` places even-indexed vector elements from `<wrs1>` into even-indexed elements of `<wrd>` and even-indexed vector elements from `<wrs2>` are placed into odd-indexed elements of `<wrd>`. For `bn.trn2` it is vice versa. Odd-indexed vector elements from `wrs1` are placed into even-indexed elements of `wrd` and odd-indexed vector elements from `wrs2` are placed into odd-indexed elements of `wrd`.

- `bn.shv.<elen> <wdr>, <wsr> <shift_type> <shift_bits>`: Logically shifts each element of vector `<wrs>` by `<shift_bits>` bits in `<shift_type>` direction. The options for `<shift_type>` are `<<` or `>>` for left or right shift, respectively.

The instruction encoding is presented in Figure 3.2. To avoid any clash with already existing instructions all new instructions are placed in the opcode group 1011011_2 (bits 0 to 6) which corresponds to the RISC-V opcode group custom-2. For the `bn.shv` the parameter `<shift_type>` is abbreviated as ST and `<shift_bits>` is split over SB1 and SB0. The encoding is chosen to allow certain future expansion. For example, the `bn.addv(m)` and `bn.subv(m)` contain a bit to select a vectorized flag group (FG) such that flag functionality similar to the non vectorized instructions is possible. The

Instruction	Bit																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bn.addv(m)	FG	0	ELEN		M	0	x	WRS2					WRS1					0	0	0	WRD					1	0	1	1	0	1	1
bn.subv(m)	FG	1	ELEN		M	0	x	WRS2					WRS1					0	0	0	WRD					1	0	1	1	0	1	1
bn.addvc	FG	0	ELEN		M	1	x	WRS2					WRS1					0	0	0	WRD					1	0	1	1	0	1	1
bn.subvc	FG	1	ELEN		M	1	x	WRS2					WRS1					0	0	0	WRD					1	0	1	1	0	1	1
bn.addvi	FG	0	ELEN		IMM1			WRS					IMM0					0	0	1	WRD					1	0	1	1	0	1	1
bn.subvi	FG	1	ELEN		IMM1			WRS					IMM0					0	0	1	WRD					1	0	1	1	0	1	1
reserved																		0	1	0						1	0	1	1	0	1	1
bn.mulv(l)	LANE				ELEN		L	WRS2					WRS1					0	1	1	WRD					1	0	1	1	0	1	1
bn.mulvm(l)	LANE				ELEN		L	WRS2					WRS1					1	0	0	WRD					1	0	1	1	0	1	1
bn.trn1	x	0	ELEN		x	x	x	WRS2					WRS1					1	0	1	WRD					1	0	1	1	0	1	1
bn.trn2	x	1	ELEN		x	x	x	WRS2					WRS1					1	0	1	WRD					1	0	1	1	0	1	1
reserved																		1	1	0						1	0	1	1	0	1	1
bn.shv	x	ST	ELEN		x	SB1		WRS					SB0					1	1	1	WRD					1	0	1	1	0	1	1

Figure 3.2.: Encoding of proposed instructions. The gray marked instructions are not proposed but space is reserved in the encoding for future expansion. An 'x' represents a 'don't care' value.

corresponding vectorized add or subtract with carry instructions as well as add or subtract an immediate are also reserved and highlighted grey in Figure 3.2.

3.3. Benefits of proposed instructions

Introducing the new instructions leads to great benefits when computing functions like the butterfly on the OTBN. An implementation for 32-bit values using the default OTBN instructions as discussed in [11] and replicated in Listing 3.1 is quite cumbersome. In Listing 3.1, eight butterfly coefficients and the required twiddle factor are already loaded into the WDRs, hereinafter named as `coeffsa`, `coeffsb` and `twiddle`. The WDR `consts` contains the required Plantard constants and other masking helpers. To compute the butterfly, first, each 32-bit value must be extracted from a WDR, then a Plantard multiplication must be implemented step by step followed by the actual butterfly computation. Finally, the result must be stored back into the WDR at the correct location. This requires 10 instructions to compute one butterfly, which must be looped 8 times to fully process a WDR, resulting in an execution time of 80 cycles (each instruction is single cycle).

```
/* coeffsa and coeffsb contain 8 32-bit coefficients.
 * consts contains the Plantard constants and masking helpers. */

/* Mask out coefficients from buffer */
bn.and coeffa, coeffsa, consts >> 192
bn.and coeffb, coeffsb, consts >> 192

/* Plantard multiplication: Twiddle * coeffb */
bn.mulqacc.wo.z coeffb, coeffb.0, twiddle.0, 192 /* (coeffb*R) mod 2^2d */
bn.add          coeffb, consts,   coeffb >> 160  /* +1 */
bn.mulqacc.wo.z coeffb, coeffb.1, consts.2, 0    /* *q */
bn.rshi         wtmp,   consts,   coeffb >> 32   /* >> d */
/* Butterfly */
bn.subm coeffb, coeffa, wtmp
bn.addm coeffa, coeffa, wtmp

/* Shift results back to buffer and shift out used coefficients */
bn.rshi coeffsa, coeffa, coeffsa >> 32
bn.rshi coeffsb, coeffb, coeffsb >> 32
```

Listing 3.1: Butterfly implementation using the default OTBN instructions.

Using the new instructions, this implementation can be easily vectorized and simplified as shown in Listing 3.2. The unwrapping and storing of the coefficients back in the WDRs is no longer required as all 8 butterflies can be computed at once. Also, the Plantard multiplication is replaced with the built in Montgomery multiplication which frees the `consts` WDR for other purposes. In total, the computation of 8 butterflies now takes only 3 instructions. However, the Montgomery multiplication is implemented as a multi-cycle operation and requires 12 cycles (see Section 4.2.4). This results in an execution time of 14 cycles for processing 8 butterflies improving the performance theoretically by 5.7x.

```
/* coeffsa and coeffsb contain 8 32-bit coefficients.
 * twiddle contains 8 twiddle factors. */
bn.mulvml.8S wtmp,    coeffsb, twiddle, 0
bn.subvm.8S  coeffsb, coeffsa, wtmp
bn.addvm.8S  coeffsa, coeffsa, wtmp
```

Listing 3.2: Butterfly implementation using the proposed vectorized instructions.

The `bn.trn1` and `bn.trn2` are useful for rearranging the order of coefficients in the WDRs which is required when the stride between the butterfly coefficients becomes less than 8 i.e., the required coefficients are loaded into the same WDR from the memory. A full display of the benefits and the achieved speed ups can be found in Chapter 5.

Chapter 4

Baseline implementation

This chapter describes how the proposed instructions are implemented and integrated into the OTBN. It first discusses architectural decisions and the actual RTL implementation of the circuitry computing the instructions. At last, the changes to the OTBN pipeline to integrate the adapted ALU modules and the chosen verification strategy are explained.

4.1. Architectural decisions

There are multiple possibilities for how to integrate the proposed instructions into the OTBN design. However, as already discussed in Chapter 3 the main focus lies on creating an area efficient implementation that requires to reuse existing hardware. From an architectural viewpoint, the task is now to decide which instruction can reuse which existing hardware most efficiently. The chosen decisions are described below, whereas the implementation is documented in Section 4.2.

As introduced in Section 2.6.1, the OTBN features two modules for big number computations namely the BN ALU and BN MAC. The BN ALU main components are two 256-bit adders, the MOD WDR, and circuitry for logic operations. These components are ideal for realizing the instructions `bn.addv(m)`, `bn.subv(m)`, `bn.trn1/2` and `bn.shv` because these instructions are of a similar kind as the already implemented instructions and the main change required is to vectorize the two adders.

To realize the `bn.mulv(m)(l)` instructions the 64-bit multiplier from the BN MAC is reused as elaborated in Section 3.2 and [11]. However, the integration of `bn.mulv(m)(l)` into the BN MAC is challenging as the OTBN pipeline technically only supports single cycle instructions and the proposed instruction implements a multi-cycle Montgomery algorithm in hardware. To enable a multi-cycle computation a stall control logic must be implemented inside the BN MAC. This logic stalls the main pipeline and controls the data path inside the BN MAC to compute the Montgomery multiplication. In addition, the chosen Montgomery computation approach requires storing temporary values which requires two additional registers next to the existing ACC WSR. Nonetheless,

integrating the `bn.mulv(m)(l)` into the BN MAC presents still the best tradeoff between area, performance, and complexity.

4.2. Design description

This section describes the actual RTL implementation of the proposed instructions and all required security measures to harden the implementation. First, it is explained how the existing 256-bit adders and 64-bit multiplier are adapted to support vectorized computations which resulted in the creation of two new modules, a configurable vectorized adder and a vectorized multiplier. Finally, the actual implementation of the new instructions based on these new building blocks is presented.

4.2.1. Vectorized adder

To vectorize a 256-bit adder, the main trick is based upon the fact that a 256-bit addition can be split into 16 adders each computing 16 bits and propagating the carry accordingly. By controlling whether a carry bit is propagated or not, it is possible to create multiple adders of different bit widths. Figure 4.1 shows the designed vectorized adder which supports vectors with 6-bit, 32-bit, 64-bit, 128-bit or 256-bit elements. It is constructed by chaining 16 17-bit adders where the lowest bit is used for the incoming carry. Between each adder, a multiplexer (MUX) selects either the carry out from the previous adder or an external carry. For the original 256-bit addition all carries are propagated to the next adder. If the input WDRs represent 16-bit vectors then each 17-bit adder computes an independent addition based on the provided carries. In the case of 32-bit vectors, two adders are connected resulting in eight 32-bit additions. This approach easily allows to compute also 64-bit and 128-bit additions by correctly setting the MUX control signals. Subtraction is also supported by using the trick that $a - b = a + \overline{b} + 1$ and setting the input carries accordingly.

In regard to leakage, this design is secure as long as the MUX control signals are predecoded. Imagine the two 256-bit inputs representing eight 32-bit values each which are shares of a secret. If the MUX control signals were unstable, e.g., set to 64-bit, then the 32-bit values would be interpreted as 64-bit values which would result in leakage because secret shares were mixed.

4.2.2. Vectorized multiplier

The designed vectorized multiplier can compute either one 64-bit, two 32-bit or four 16-bit multiplications at once. This is possible by splitting a 64-bit multiplication into its 16-bit partial products and then shift and sum these according to the selected element width. Consider a 64-bit multiplication of the operands a and b and split them into four 16-bit chunks like $a = [a_3, a_2, a_1, a_0]$ and $b = [b_3, b_2, b_1, b_0]$. The multiplication can be

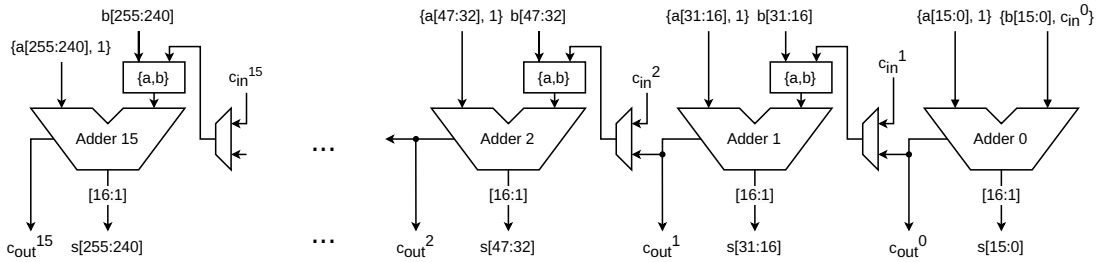

Figure 4.1.: A vectorized 256-bit adder supporting vectors with 16-bit, 32-bit, 64-bit, 128-bit or 256-bits elements. It is constructed by chaining 16 17-bit adders. The carry is either an input or originate from the result of the lower adder depending on the vector element length.

rewritten with $R = 2^{16}$ as in Equation (4.1).

$$\begin{aligned} a \cdot b = \quad & (a_0 b_0) \\ & + R \cdot (a_0 b_1 + a_1 b_0) \\ & + R^2 \cdot (a_0 b_2 + a_1 b_1 + a_2 b_0) \\ & + R^3 \cdot (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) \\ & + R^4 \cdot (a_1 b_3 + a_2 b_2 + a_3 b_1) \\ & + R^5 \cdot (a_2 b_3 + a_3 b_2) \\ & + R^6 \cdot (a_3 b_3) \end{aligned} \tag{4.1}$$

Splitting the partial product computation from the summation allows to compute vectorized 16-bit or 32-bit multiplications. In the case of 16-bit multiplications, we simply pick the four partial products $a_i b_i$ as results. For 32-bit elements, the two results c_0 and c_1 can be computed with Equation (4.2). For a 64-bit multiplication, the 32-bit results can directly be used to compute the 64-bit result by summing up the remaining partial products.

$$\begin{aligned} c_0 = \{a_1, a_0\} \cdot \{b_1, b_0\} = \quad & (a_0 b_0) \\ & + R \cdot (a_0 b_1 + a_1 b_0) \\ & + R^2 \cdot (a_1 b_1) \end{aligned} \tag{4.2}$$

$$\begin{aligned} c_1 = \{a_3, a_2\} \cdot \{b_3, b_2\} = \quad & (a_2 b_2) \\ & + R \cdot (a_2 b_3 + a_3 b_2) \\ & + R^2 \cdot (a_3 b_3) \end{aligned} \tag{4.3}$$

When implementing this in RTL, caution has to be taken regarding leakage. Depending on the configured element length, not all 16-bit partial products may be computed. For example, if the element length is set to 16-bits, only the $a_i b_i$ (i is the same for a and b) partial products must be computed. Computing any other partial product generates leakage because this operation then entangles two 16-bit shares which are not supposed

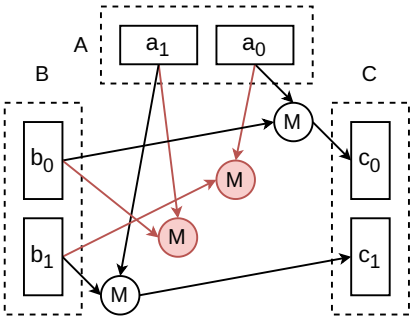

Figure 4.2.: For the partial product generation, inputs to the unused partial product multipliers (marked red) must be blanked as otherwise leakage is generated.

to be combined. The same applies to 32-bit elements and only for the 64-bit case, all partial products can be computed. This selective partial product computation can be achieved by blanking the corresponding 16-bit shares before the multiplier circuit. A simplified case is represented in Figure 4.2 which presents the partial product computation of a vectorized multiplier. This multiplier is capable of computing $C = A \cdot B$ and can interpret A and B as either two full word values or as vectors with two elements each. For the first interpretation, all partial products must be computed and then summed up with a logic not shown. This is equivalent to the 64-bit case. For the second case, where the inputs are interpreted as vectors, only the two partial products a_0b_0 and a_1b_1 may be computed. The two multipliers marked red may not receive the values as this would mix up, for example, a_0 with b_1 and therefore generate leakage. As a consequence, the red marked paths must be blanked in this case.

The second step, summing the partial products, requires also some attention. Figure 4.3 depicts the summation circuitry. Despite the partial products are only computed on demand, two blankers are still essential. The first blanker ensures that in the 16-bit case the partial products a_ib_i are not propagated into the 32-bit summation (these are used for both cases) and only follow the 16-bit path. The second blanker handles the 32-bit result forwarding and is only active in the case of a 64-bit multiplication. For the result selection, a regular MUX is sufficient because one of the 32-bit and 64-bit inputs either contains the result or is all zero. The 16-bit input can always be propagated into the MUX as in all three cases the data is already mixed up in the summations anyway. In addition, the MUX control signal is nonetheless predecoded as it is helpful to reduce the critical path.

4.2.3. Bignum ALU module

The dataflow of the vectorized BN ALU is depicted in Figure 4.4. It consists of one vectorized shifter, two vectorized adders (X and Y), a modulo result selector, the MOD WSR, a modulo value replicator, logic operators for regular 256-bit operands (AND, OR, XOR, NOT) and a vector transposer. In the following, the implementation of the new instructions is explained starting with the vectorization of addition and subtraction,

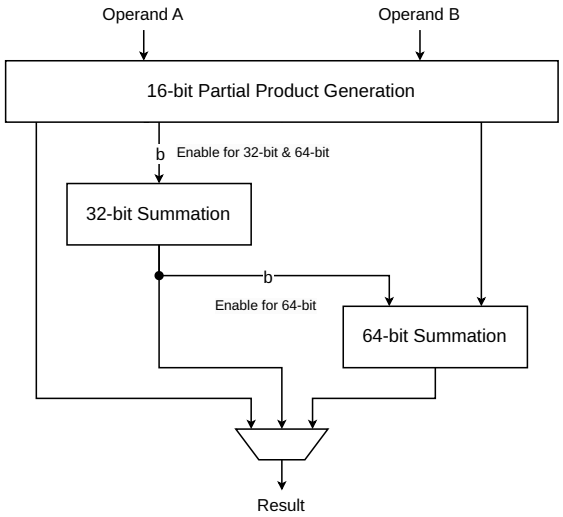

Figure 4.3.: The partial product summation circuit of the vectorized multiplier requires two blankers which are depicted with a lowercase 'b'.

vectorized shifting, vector transposing and concluding with the required blanking mitigations. The already existing logic circuitry is not vectorized and, thus no modifications are made.

Vectorized (pseudo modulo) addition and subtraction

The instructions `bn.add` and `bn.addv` as well as their subtraction counterparts are directly implemented using the adder Y. The second operand B is first passed through the shifter to support the pre-shifting functionality of `bn.add`.

For the pseudo modulo variants (`bn.addm` and `bn.addvm` as well as the subtraction variants) adder X computes the initial addition $X = A + B$ and adder Y computes the pseudo modulo result $Y = X - \text{MOD}$ where MOD is the value stored in the MOD WSR. In the case of a pseudo modulo addition, the result is selected depending on the carry bits according to the following rules:

- If adder X generates a carry:
  $A + B >$ MOD (because MOD is 256-bit) $\rightarrow$ Select Y.
- If adder Y generates a carry:
  $X - \text{MOD} = A + B - \text{MOD} >= 0 \rightarrow A + B >= \text{MOD} \rightarrow$ Select Y.
  Note, this is only valid if adder X does not generate the carry.
- If neither adder generates a carry $\rightarrow$ Select X.

For the pseudo modulo subtraction, the result is selected based on

- If adder X generates a carry: $A - B >= 0 \rightarrow$ Select X.

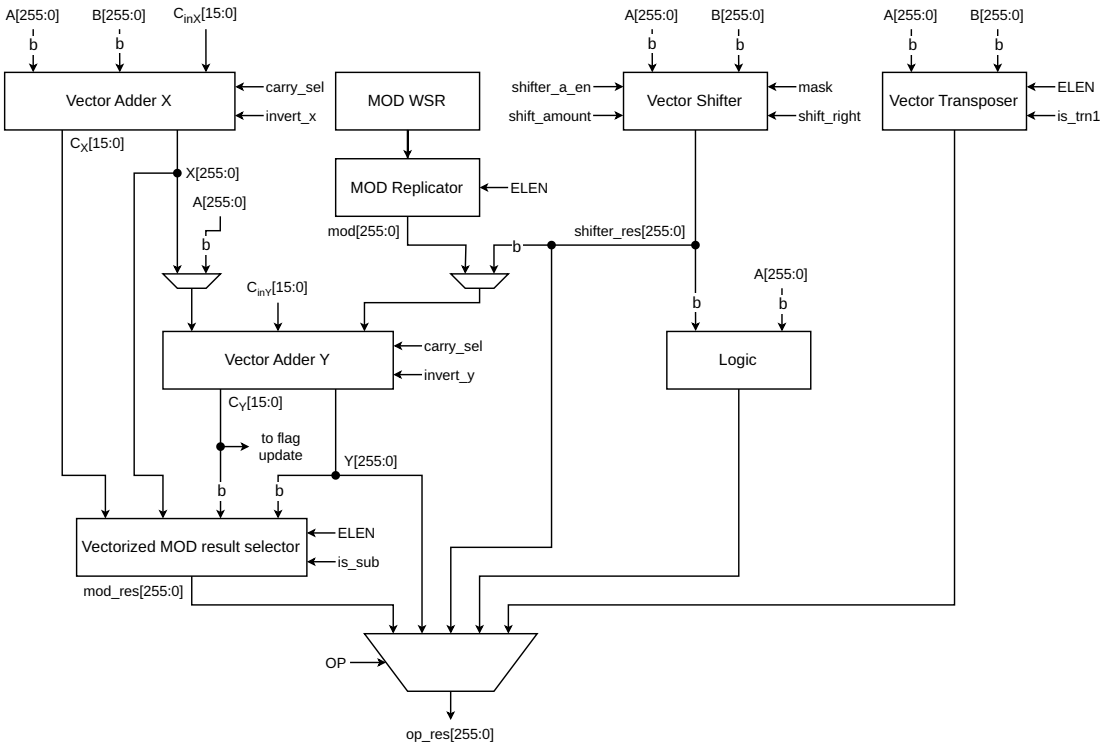

Figure 4.4.: The vectorized BN ALU structure. The two adders X and Y are used to compute the pseudo modulo reduction. A lowercase 'b' represents a blanker.

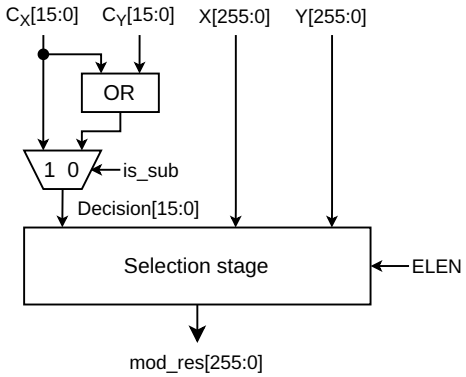

Figure 4.5.: The complete modulo result selector circuit. The result selection decision is computed for each 16-bit chunk independently and the forwarded to the selection stage.

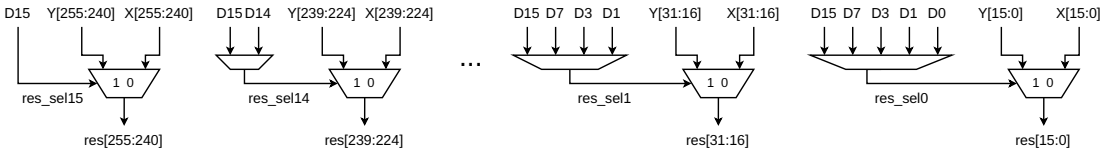

Figure 4.6.: The modulo result selection stage. Depending on the element length the appropriate decision bit D_i is forwarded to the result selection MUX for each 16-bit chunk independently.

- Otherwise select Y.

This decision logic is easily extended to vectorized computations by considering the appropriate carry bits for each vector element individually and replicating the MOD value for each vector element which is implemented with the MOD value replicator block.

The result selection described before is implemented in the modulo result selector module, shown in Figure 4.5, and is split into two stages. The first stage computes a decision bit indicating which result to take assuming a 16-bit operation was performed, i.e., the decision is based on the two carry bits from the corresponding 17-bit adders inside the vectorized adders (see Section 4.2.1). These decision bits are then propagated into the selection stage (Figure 4.6) which selects for each 16-bit chunk the appropriate decision bit based on the desired element length. This selected decision bit (res_sel_i) then controls the result selection between X and Y. For example, for a 32-bit pseudo modulo subtraction, the result bits $[15:0]$ and $[31:16]$ are controlled by the decision bit D_1 which is generated by the carries of the 2nd 16-bit chunk ($C_X[1], C_Y[1]$).

Vectorized shifter

The shifter shown in Figure 4.7 (left) operates on a 512-bit value to implement the already existing `bn.rshi` (concatenate and right shift) instruction and is based on a barrel shifter supporting a right shift up to 256 bits. The lower (256-bit) half of the input and output can be reversed to allow left shift implementation. There is no concatenate and left shift instruction so reversing is not required over the full width.

In regular mode (i.e., no concatenation) only the operand B is relevant and the upper 256 bits are set to zero. Vectorizing this shifter can be achieved by adding a simple masking step after the shift step. Figure 4.7 (right) shows a simplified example of this masking step. Assume the input is a vector with four elements which should be right shifted by two bits (i.e., by the shift_amount value). To compute this, the vector is first shifted equally as a non vectorized shift resulting in a vector where the lowest two bits of each element are wrongly shifted into the element right to it. These wrong bits (located where the red marked zeros are in the mask) are then set to zero by ANDing with a bit-mask for the desired element length. The resulting 256-bit value is a vector where each element is individually shifted right by two bits. This implementation natively supports the default 256-bit case by setting all bits of the mask to one.

Vectorized transposer

The vectorized transposer implements the functionality of `bn.trn1` and `bn.trn2` by directly shuffling the bits for each configurable element length separately and then selecting the desired result with a onehot MUX to avoid any leakage. For an explanation of the wiring consult Section 3.2 and Figure 3.1.

Blanking

In regard to blanking, the BN ALU is rather simple. The operands A and B are blanked before they propagate into a computation circuitry and each blanker is only activated if the current instruction actually requires the circuitry. The two MUXs in front of the adder Y can be regular MUXs with predecoded control signals as their inputs are already blanked. Inside the modulo result selector the MUX for selecting the decision bit depending on whether it is an addition or subtraction also only requires a regular MUX but its control signal is_sub must be predecoded. However, all MUXs inside the selection stage must be onehot MUX as otherwise the decision bits over multiple 16-bit chunks and thus vector elements are combined. Note that adder X is never used in isolation, it is always combined with adder Y so there is no need for blanking between adder X and adder Y. For the vectorized shifter regular MUXs with predecoded control signals are sufficient. The mask theoretically does not have to be predecoded but it still is to optimize the critical path.

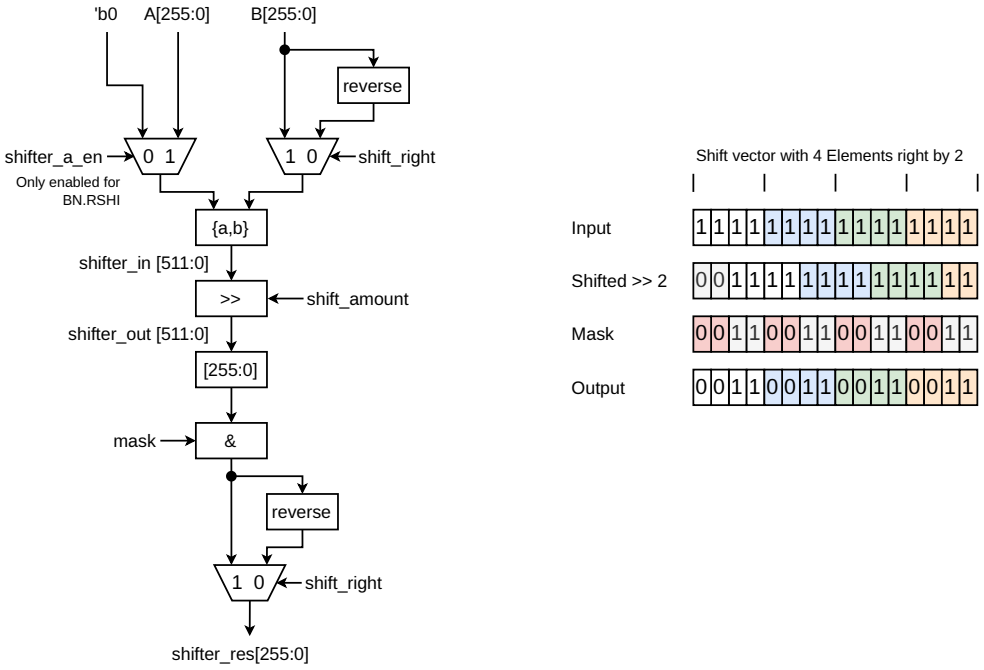

Figure 4.7.: *Left*: The RTL implementation of the vectorized shifter. The shift itself is always a right shift. Left shifting is achieved by reversing the input and output.
Right: An example where a vector consisting of four 4-bit elements is shifted by two bits to the right. First the raw input is shifted to the right. Then a mask is generated and ANDed to delete the bits which were shifted into the neighboring vector element.

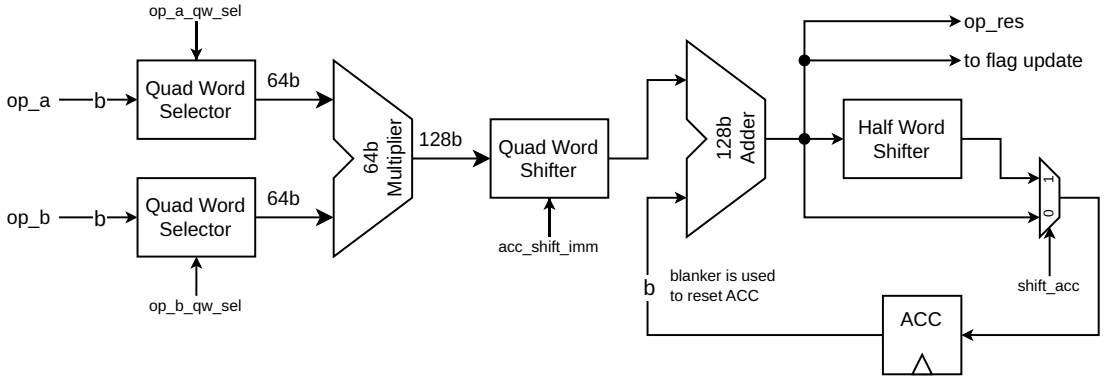

Figure 4.8.: The original unmodified BN MAC featuring a 64-bit multiplier, 256-bit adder and a 256-bit WSR named ACC.

4.2.4. Bignum MAC module

Vectorizing the BN MAC and reusing the existing hardware poses some challenges due to the complexity of the instructions and the chosen multi-cycle approach. To better understand the implemented changes this section is split in three parts. First, the original BN MAC, supporting the `bn.mulqacc[.so, .wo]` instruction (see Section 2.6.1), is explained followed by how the Montgomery algorithm (see Algorithm 6) can be computed using only one multiplier by splitting its execution over multiple cycles. Lastly, it is described how the elaborated concepts are integrated into the BN MAC to enable vectorized (Montgomery) multiplications and what problems arise in regard to security hardening the implementation.

Original BN MAC functionality

The original BN MAC is shown in Figure 4.8. To compute a multiplication, first, the desired quad word (64 bits) of each operand (WDRs, 256 bits) is extracted and forwarded into the 64-bit multiplier. Next, the multiplication result can be shifted in multiple of 64 bits to achieve an efficient partial product summation with the value stored in the ACC WSR. After summing the shifted result with the content of the ACC WSR the summation result can again be shifted by 128 bits before writing it into the ACC WSR. In case the ACC WSR should be set to zero before the addition the path to the adder can be blanked. The final WDR update for the `.so` and `.wo` options is handled in the OTBN controller block and not relevant to the BN MAC.

Splitting Montgomery over multiple cycles

The Montgomery algorithm (see Algorithm 6) requires a total of three multiplications. As it was decided to only reuse the existing 64-bit multiplier the computation therefore has to be split over multiple cycles. This splitting requires to store temporary values and therefore two new registers named TMP and C are introduced. With these new registers

the Montgomery algorithm can be computed in 3 cycles as described below where a and b are the operands, d the bit-width of the operands, q the selected prime and R the corresponding Montgomery constant for the chosen d and q.

- Cycle 1:
  - $\text{TMP} = [a \cdot b]_d$
  - $C = a \cdot b$
- Cycle 2:
  - $\text{TMP} = [\text{TMP} \cdot R]_d$
  - $C = a \cdot b$ (unchanged)
- Cycle 3:
  - $result = \left[C + \text{TMP} \cdot q\right]^d (-q)$ where $(-q)$ represents the conditional subtraction.

A simplified implementation of this splitting is presented in Figure 4.9 where orange highlighted components replace existing hardware with its vectorized components as introduced previously (see Section 4.2.1 and Section 4.2.2). Therefore, this data flow is already in vectorized fashion, i.e., it is assumed that the operands a and b are 64-bit vectors containing either one 64-bit element, two 32-bit elements or 16 16-bit elements and the final result is 64-bit wide and represents vector elements. The steps for each cycle described above naturally applies for each vector element separately, i.e., the bit selection steps $[.]_d$ and $[.]^d$ are also applied in a vectorized fashion.

In addition, this circuitry does natively support regular 64-bit multiplications as well as its vectorizied version in a single cycle. as the first cycle computes $a \cdot b$. The result in this case is simply the output of the vectorized multiplier after the first cycle.

Integrating the approach

As mentioned, the instructions `bn.mulv(m)(l)` operate on whole WDRs (256 bits) whereas the circuitry described previously can only handle 64 bit chunks at once. Therefore, to multiply two WDRs they must be split into four 64 bit chunks which in turn requires temporarily storing three 64 bit results. To temporarily store these values the existing ACC WSR can be used. By processing a 64-bit chunk and then storing the result at the corresponding quad word inside the ACC WSR this problem can be solved. As a consequence, the execution of `bn.mulv(l)` requires 4 cycles to complete (4x1 cycle) and the `bn.mulvm(l)` a total of 12 cycles (4x3 cycles). To handle this multi cycle execution a BN MAC internal stateful logic handling the control signals is required. The following section first discusses the actual RTL design of the data flow and only afterwards the state handling is described.

Figure 4.10 shows the complete structure of the BN MAC where the existing functionality is merged with the previously elaborated design. The original functionality

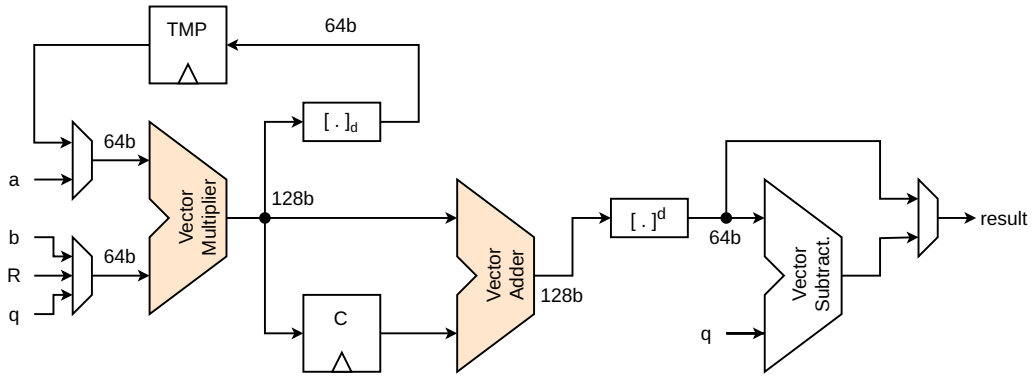

Figure 4.9.: A vectorized multi cycle data flow implementation of a Montgomery multiplication. If compared to the original BN MAC the orange highlighted components could replace existing hardware. Note, this is not the actually implemented design.

is possible if the control signals 'is_mod' and 'is_vec' are set to zero. In this case, the operands flow through the 'Quad Word (& Lane) Selectors' to select the desired 64-bit chunk from the WDRs followed by the vectorized multiplier which is configured to compute a 64-bit multiplication. The result then goes into the 'Quad Word Shifter' and is summed up with the ACC value. Finally, the addition result is output as result and optionally shifted before it is written back into the ACC WSR.

For a regular vectorized multiplication again the desired 64-bit chunks are selected and propagated into the multiplier. Then, the result is directly forwarded into the 'ACC Merger' which is responsible for merging it at the correct location in the ACC register.

To compute a Montgomery multiplication the new registers TMP and C come into action. The first cycle is similar to the other multiplication modes except that the result is stored in register C and the modulo d result is stored in TMP (taking only the lower d bits is equivalent to computing a value modulo d). In the second cycle, the value of TMP is multiplied by the Montgomery constant R and the result stored in TMP. This constant is expected to be loaded, prior to the execution, into the MOD WSR at bits $[61:32]$ which is then extracted by the 'MOD Extractor' block. The 'MOD extractor' also provides the prime q from the bits $[31:0]$ for the last cycle. Finally, the last multiplication, the addition with C and the conditional subtraction are performed and the result is then merged into the ACC WSR. The conditional subtraction is implemented with the described basic vectorized adder where all carries in are set to one and one input is inverted. Depending on the carries out either the subtracted or original values are forwarded to the 'ACC merger'.

Regarding the security hardening, the BN MAC implementation is quite sophisticated due to its complex MUX structures and many data paths. Blanking data paths that are unused by the current execution cycle is implemented with blankers. These are represented by a lowercase 'b' in Figure 4.10. Their control signal is noted next to them and is generated by the predecoder and also the state handling logic as described

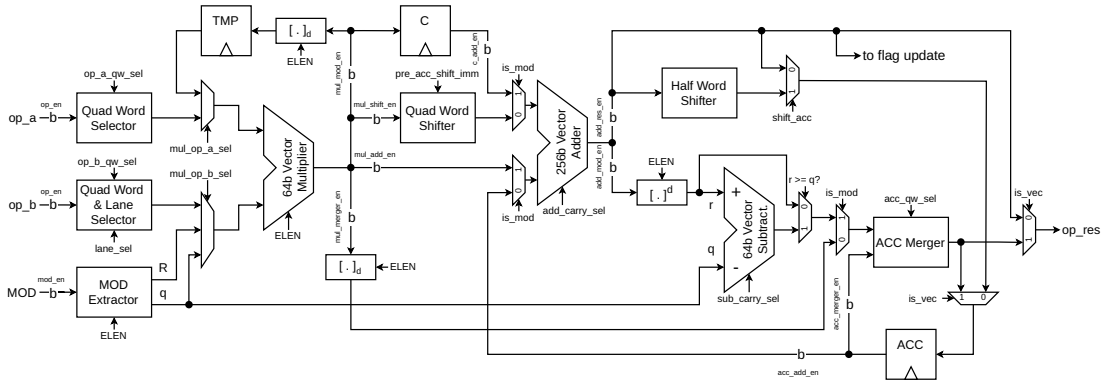

Figure 4.10.: The BN MAC structure supporting the regular functionality as well as vectorized (Montgomery) multiplication. A lowercase 'b' represents a blanker where its control signal is listed next to it.

in the next section. The more complex part is designing the various MUX structures correctly such that no vector elements are mixed up. This report does not further elaborate on these details as the RTL code contains detailed explanations. Repeating these explanations here would result in overly complicated explanations as the context of the actual RTL codebase[1] is required to understand the problematic cases. Please note that the code submitted with this work does not contain a completely leakage free BN MAC implementation. In particular, the 'Quad Word (& Lane) Selectors' need to be reworked and there was no time left to fix the implementation. There are also issues with the stability of the blanker control signals as explained in the next section.

Multi cycle state handling

The second main challenge of integrating the new functionality into the BN MAC consists of generating the control signals over multiple cycles and still ensuring single cycle execution for the regular `bn.mulqacc` instruction. To address this problem, a FSM is designed whose states are depicted in Figure 4.11. This FSM has a default state named 'Regular64b' and is the entry point for any multiplication instruction. Per default, the control signals are set such that the regular 64-bit multiplication is executed. This is required to support single cycle execution of the original instructions (`bn.mulqacc`). However, this state also includes two sub-states 'StartQW0' and 'StartModQW0C0' which are the starting point for all multi cycle instructions and both states include their respective lane modes. For a regular vectorized multiplication the FSM switches into the 'StartQW0' state (within the starting cycle) and progresses on the 'QW1', 'QW2' and 'DoneQW3' path for the remaining 3 cycles and finally returns into the default state. These states then set all required MUX, register write enable and blanker signals. The modulo multiplication, i.e., Montgomery multiplication, is handled similarly except that there are a total of 12 states. For example, the state 'ModQW1C1' represents the

[1]The code is open source, see [25]

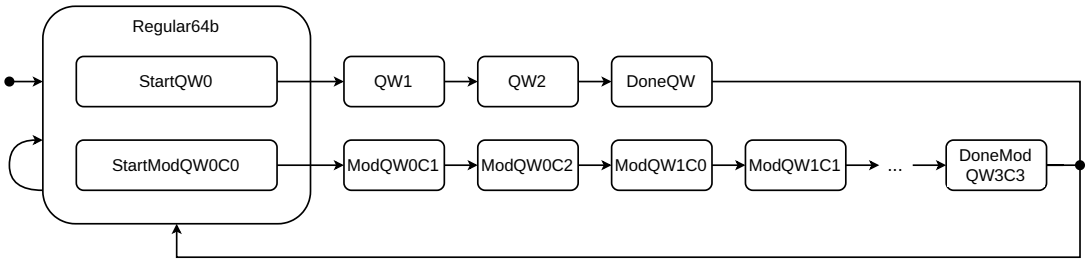

Figure 4.11.: The state diagram of the FSM handling the BN MAC control signals. The initial state 'Regular64b' handles the single cycle multiplication and contains two sub-states which serve as starting point for both the vectorized regular and Montgomery multiplication.

cycle where the 2nd cycle of the Montgomery multiplication for the 2nd 64-bit chunk is performed.

How the control signals must be set for each state is documented in Figure A.1. This figure also states which component / logic generates a control signal value which is relevant for the blanker control signals. As described in Section 2.6.2, a control signal for a blanker must be predecoded, i.e., must come directly from a register. To reduce the complexity of a first implementation, the designed FSM does generate some blanker control signals combinatorially and therefore these blankers are rendered ineffective from a security aspect. However, their functionality is still relevant for the circuit to work properly. Generating all blanker signals for the multi cycle execution in the predecoder is not possible as the OTBN pipeline is stalled during the multi cycle execution (this is explained in Section 4.3). Therefore, the predecoder is also stalled, i.e., the generated signals are static over the duration of the instruction.

Implementing a solution to this problem did not fit into the timeline of this project but a possible approach is to adapt the FSM to also predecode these signals. This means the FSM sets the signal for the next state and has flops to stabilize the signals such that in the next cycle proper blanker control signals are present. The problem of this approach is how to know what type of multiplication should be performed in advance (one cycle earlier). A solution would be to transfer the flops for registering all the control signals, that originate from the FSM or the predecoder, into the FSM. Then the predecoder forwards the combinatorial / unregistered signal to the BN MAC. These signals are only valid in the next clock cycle. When the FSM is in the default state ('Regular64b'), those flops get the value from the predecoder, otherwise from the FSM.

4.3. Integration into pipeline

Integrating the modified BN ALU and the new instruction into the OTBN pipeline is relatively straightforward as all of these instructions execute in a single cycle. The new instructions are simply added to the decoder and predecoder logic in the same fashion

as all the existing instructions are decoded. On the other hand, integrating the new BN MAC and its multi cycle instructions gives rise to multiple challenges.

The first problem is that the BN MAC requires the value of the MOD WSR which is part of the BN ALU and thus not directly accessible. This problem is solved by extending the BN ALU interface to expose the MOD WSR via an integrity protected signal which then is routed through the controller into the BN MAC.

The second challenge lies in enabling a multi cycle execution as the OTBN has no actual support for multi cycle instructions. However, the OTBN features indirect WDR load and store instructions (`bn.lid` and `bn.sid`) where the source / destination WDR is defined by the value in a GPR. This indirect load from the GPR cannot be done in the same cycle as accessing the WDR and thus requires a stalling mechanism that stalls the pipeline until the indirect load has finished. When the indirect load has finished the actual `bn.lid` execution is continued. The same mechanism is used to stall the pipeline while a multi cycle multiplication is executed. It is implemented by asserting the stall flag inside the OTBN controller until the BN MAC asserts its valid flag.

During these cycles, the FSM steps through the required states as described in Section 4.2.4 and performs multiple reads from the source WDRs and should only update the destination WDR in the last cycle. However, the current BN MAC integration has one unpleasant limitation namely, *a source WDR to any multi cycle multiplication may not be the destination WDR for the same instruction execution*. The reason is that in the implemented integration, the destination WDR write enable signal is set over the whole duration instead of only for the last cycle. This design decision was required because the predecoder generates the write enable signal and as it is stalled cannot change it dynamically as otherwise a predecode error would arise. The indirect loads and writes also suffer from this problem, but there a special predecoding handling is implemented. However, for this project, it was infeasible to implement this due to the complexity and the limited project duration.

4.4. Verification

Designing a comprehensive verification suite to test the implementation of the proposed instructions was an essential part of the work. Whereas the OpenTitan project relies on a fully fledged UVM (Standard Universal Verification Methodology) and formal property verification approach, for this project a simplified approach was chosen. This approach differentiates between the new building blocks like the vectorized multiplier and the integration / modifications to the BN ALU, BN MAC and OTBN pipeline.

Testing the low level RTL implementations of building blocks is supported by individual SystemVerilog testbenches running on QuestaSim 2023.4. These testbenches instantiate the module and execute the defined testcases in complete isolation from any other component which simplifies testing as well as debugging. Detailed instructions on how to perform the module tests can be found under `hw/ip/otbn/pre_dv` in [25].

To test the modifications and integration into the ALUs such a testbenche approach is infeasible as the ALUs depend on too many external interactions. Therefore, testing the

new instructions, and thus the integration, was done by using a co-simulation of the RTL code with the OTBN simulator. This required to extend the existing OTBN simulator with a cycle accurate implementation of the proposed instructions, which in turn also required to write test cases for the simulator. These simulator test cases were integrated into the existing simulator testing framework present in `hw/ip/otbn/dv/otbnsim`. The proposed instructions finally were tested by running simple OTBN assembly programs in a co-simulation based on Verilator and the extended OTBN simulator. All these test programs can be found in `sw/otbn/vectorized/tests`. To quickly get a fail/pass test feedback during development, the OTBN project also features a smoke test program that executes all available instructions at least once. This smoke test was extended with the proposed instructions.

Chapter 5

Baseline benchmark

This chapter describes benchmarks performed to investigate the benefits of the implemented ISA extension. These benchmarks include NTT computations for 32-bit elements which were implemented with either the default OTBN instructions or the proposed instructions. Such implementations of the NTT and INTT on the OTBN have already been published by e.g., [12, 11]. The benchmarked implementations follow these implementations closely.

Note that benchmarking the NTT computation only illustrates a limited aspect of the advantages provided by the proposed extension. The vectorized instructions are also highly beneficial when computing elementwise multiplications in the NTT domain and other computations related to ML-DSA. However, these applications were not benchmarked in this work due to time constraints and as there exists a comprehensive analysis in [11]. Additionally, the benefits of the proposed instructions are discussed only for 32-bit elements, but the benefits are even more pronounced for 16-bit elements. For 16-bit elements, one instruction operates on twice the number of elements therefore approximately reducing the required instructions.

5.1. Non-vectorized NTT implementation

As explained in Section 2.4.1, a NTT requires to compute butterflies over multiple layers. How to compute such a butterfly on the default OTBN has already been shown in Listing 3.1. This section therefore focuses on optimizations regarding the data loading and possible pre-computations.

For example, for a 8 layer NTT ($n = 256$) a naive approach would be to compute the first layer by loading the coefficients $a_0, a_1, a_{128}, a_{129}$ into the WDRs, compute the butterflies and store the results back into memory. This would be repeated until the first layer is fully processed and then the other layers are computed in a similar fashion. However, this approach requires loading and storing all coefficients once per layer which is ineffective and can be optimized by applying a technique called layer merging. For example, to merge two layers, instead of the previously mentioned coefficients the

coefficients $a_0, a_{64}, a_{128}, a_{192}$ are loaded into WDRs. This allows to compute the butterflies (a_0, a_{128}), (a_{64}, a_{192}) and immediately afterward the computation of the second layer butterflies (a_0, a_{64}) and (a_{128}, a_{192}) (Figure 2.1 may help to understand this optimization). With a 2-layer merge, the number of load and store instructions can be reduced by a factor of 2. Theoretically, all layers of a NTT could be merged. However, the limiting factor is the number of available registers as merging l layers requires 2^l coefficients to be loaded simultaneously. Following the implementation of [12] the best approach for OTBN is to create a 4-4 layer merge which can directly be applied to Algorithm 4 and also Algorithm 5.

As this work focuses on a NTT for ML-DSA the parameters of the NTT are known at compile time. This allows to precompute the twiddle factors and save multiplications and exponentiations at the cost of program size. By using the butterfly implementation described in Listing 3.1 these precomputed twiddle factors must be stored in the Plantard representation (see Algorithm 7). Another optimization for the INTT is to multiply half of the last layer twiddle factors with the factor n^{-1}. This saves $n/2$ multiplications for the final scaling.

The described NTT approach is implemented in the functions `ntt_base_dilithium.s` and `intt_base_dilithium.s`. Test programs invoking these functions can be found at `sw/otbn/ntt/tests` in [25].[1] For a more in-depth description of the implementation see [12], the source code is publicly available.[2]

5.2. Vectorized NTT implementation

For the vectorized NTT implementation the same optimizations apply as for the non-vectorized version. The difference lies in the computation of the butterflies as the approach in Listing 3.2 is implemented. Due to this efficient computation less WDR loads are required and therefore fewer loops.

When implementing the NTT layer merging, one peculiarity arises for the second layer merge (layers 5 to 8). The coefficients are stored in memory in a linear fashion, and they are also loaded in this order into the WDRs. This does not present any issues when computing layers 1 to 5, as the required coefficients are loaded into distinct WDRs. However, for layers 6 to 8, the stride between the coefficients is 4, 2, and 1, respectively. This necessitates the computation between coefficients inside the same register, which is not feasible to compute with the vectorized instructions. To resolve this issue, the WDRs must be transposed such that the stride between two WDR elements is 8. Listing 5.1 shows this transposition for 8 WDRs. The remaining layers can then be computed efficiently with the use of the vectorized instructions. Prior to storing the results back in memory, the transposition must be reversed by applying Listing 5.1 a second time.

[1] The benchmarks are available on the 'benchmark' branch.
Commit `94bdc0a069d3eb3a26dd579350844315fd66e0f1`

[2] https://github.com/dop-amin/dilithium-on-opentitan-thesis

```
/* Transpose w0 - w7 via w24 - w31 */
bn.trn1.8S w24, w0, w1
bn.trn2.8S w25, w0, w1
bn.trn1.8S w26, w2, w3
bn.trn2.8S w27, w2, w3
bn.trn1.8S w28, w4, w5
bn.trn2.8S w29, w4, w5
bn.trn1.8S w30, w6, w7
bn.trn2.8S w31, w6, w7
bn.trn1.4D w4,  w24, w26
bn.trn2.4D w24, w24, w26
bn.trn1.4D w26, w25, w27
bn.trn2.4D w25, w25, w27
bn.trn1.4D w27, w28, w30
bn.trn2.4D w28, w28, w30
bn.trn1.4D w30, w29, w31
bn.trn2.4D w29, w29, w31
bn.trn1.2Q w0, w4,  w27
bn.trn2.2Q w4, w4,  w27
bn.trn1.2Q w1, w26, w30
bn.trn2.2Q w5, w26, w30
bn.trn1.2Q w2, w24, w28
bn.trn2.2Q w6, w24, w28
bn.trn1.2Q w3, w25, w29
bn.trn2.2Q w7, w25, w29
```

Listing 5.1: Transposing the WDRs by using the proposed `bn.trn1/2` instructions.

The vectorized NTT is implemented in the functions `ntt_mldsa.s` and `intt_mldsa.s`. The benchmark programs are also at `sw/otbn/ntt/tests` in [25].

5.3. Results

The results of the performed benchmarks are presented in Table 5.1. As expected, the proposed ISA enables a more efficient NTT computation. With the vectorized instructions the 32-bit element NTT computation is about 3.46x faster and 3.40x for the INTT, respectively. The effect of the multi cycle instructions is clearly visible when comparing the number of instructions with the total required cycles. For vectorized implementations, the ratio of stall cycles to instructions reaches up to 200% for the vectorized NTT implementation.

In regard to the code size, the vectorized approach has a clearly smaller instruction memory footprint. Whereas the default NTT implementation requires 2296 B the vectorized implementation only requires 1660 B (27 % less) and for the INTT the savings are 30 % (2408 B to 1676 B). Again, this is because one instruction can operate on multiple elements at once. The DMEM requirements are also reduced significantly by 44 % and 45 % for the NTT and INTT, respectively. The reason here is, that the Twiddle factors can be stored in Montgomery representation instead of Plantard representation which

Benchmark	**ISA**	**Cycles**	**Instr.**	**DMEM** (B)	**IMEM** (B)
NTT	Default	8156 (x1.00)	7711 (x1.00)	3744 (x1.00)	2296 (x1.00)
	Baseline	2360 (x3.46)	785 (x0.10)	2080 (x0.56)	1660 (x0.73)
INTT	Default	8649 (x1.00)	8204 (x1.00)	3752 (x1.00)	2408 (x1.00)
	Baseline	2545 (x3.40)	794 (x0.10)	2080 (x0.55)	1676 (x0.70)

Table 5.1.: Benchmark results for NTT and INTT implementations based on the default and proposed ISA.

reduces the size of the twiddle factor array by half (32 bits vs 64 bits).

Chapter 6

Baseline synthesis

After exploring the computational benefits of the proposed vectorized ISA, this chapter focuses on its area requirements. In order to evaluate the proposed extension, the elaborated design and the default OTBN were synthesized to perform an area-performance analysis based on the TSMC65 technology, using Synopsis 2019.03. This chapter first describes the synthesis setup which requires some attention to stop the tool from optimizing away security related redundant signals, such as blankers and onehot MUX structures. Finally, the achieved results are presented and the critical path is explored. The default, unmodified OTBN design used as the starting point is hereinafter referred to as the default design. The design implementing the proposed instructions is referred to as the baseline design.

6.1. Synthesis setup

The synthesis was set up based on an established setup created by ETHZ's Integrated Systems Laboratory (IIS). However, there were some modifications required regarding the optimization process. When synthesizing an RTL design, the synthesis tool tries to optimize any combinatorial logic as well as to remove redundant signals. However, this poses a problem for a security hardened design as it may modifies structures that must be synthesized exactly as designed as otherwise their security functionality is lost. In the OTBN for example, this applies, among other things, to the blankers and onehot MUXs. These constructs with their special structure (see Section 2.6.2) could be optimized heavily in regard to their area but this would break their security purpose. E.g., a onehot MUX would possibly be optimized to a regular MUX which would lead to leakage.

To avoid this problematic optimization, the OpenTitan ecosystem features generic primitive modules for such circuitry. These modules must be used wherever such special hardware is required. This allows to specify extra constraints for the synthesis, such that these blocks are not optimized. In this work, the implementation of these primitives was directly implemented with TSMC65 library cells, and all modules were given a

'set_size_only' constraint. This prevents the synthesis tool from removing these cells and therefore ensures that the expected hardware is generated. For more detail about the synthesis setup see Appendix B.3.

6.2. Synthesis results

The results of the area-performance analysis are shown in Figure 6.1 where for both designs the clock period was swept from 4 ns to 25 ns. [1] The dashed lines in Figure 6.1 show the absolute minimal area requirements which are estimated by setting the clock to 1000 ns. For this case, the baseline requires about 11.5 % more area and for the realistic clock cycles above 15 ns the increase is between 12.9 % to 17.4 %. With clock periods below 12 ns the area requirements begin to rise gradually for the default design until the hard border at around 5.5 ns is reached. In comparison, the baseline design curve steepens earlier and the minimal clock period is around 8 ns. At this point, the area increase is about 22.6 %. A more in-depth analysis of the area result is presented in Section 7.3. This part contains area results for the designed vectorized components and an analysis of the optimized baseline design. For a critical path analysis of the default OTBN design see Appendix B.1.

6.2.1. Critical path analysis

Regarding any possible baseline optimizations, it is of interest to know what part of the design limits the performance. In the following, the baseline's critical path is investigated for a clock period of 8 ns. This clock period is presented as it is the fastest the baseline can be run at and it is the default clock period for the OTBN in OpenTitan's Earlgrey microcontroller.

Figure 6.2 depicts the critical path which takes 7.7532 ns and has a slack of 0.0001 ns. The path originates from the flops holding the predecoded signals for addressing the Bignum register file and then propagates through the BN MAC. Inside the BN MAC, the path leads, as expected, through the vectorized multiplier and vectorized adder to the conditional subtraction for the Montgomery multiplication. Finally, the path arrives at the write port of the ACC WSR. This described path is the same also for the 2nd most critical path (7.7524 ns, 0.0003 ns). The 3rd path is almost identical but branches after the ACC Merger and ends at the Bignum register file (7.7514 ns, 0.0003 ns). An exception is the 4th path (7.7311 ns, 0.0004 ns), which starts also at the predecoding flop, traverses the Bignum register file and then enters the BN ALU. There it passes the Adder X, Adder Y and the modulo result selection. It then propagates through the integrity check for the MOD WSR and thus into the secure wipe handling. From there, as the final step, it affects the reset value path of the ACC WSR. These presented critical paths are also the most common critical paths for the other synthesized clock periods.

[1] Designs can be found in [25].
Default: `a332afe31fce61780385c9dd045ff3aa2c540fe4`
Baseline: `53e3ebe35e76bcdef12c005ab5aeb35e326badf0`

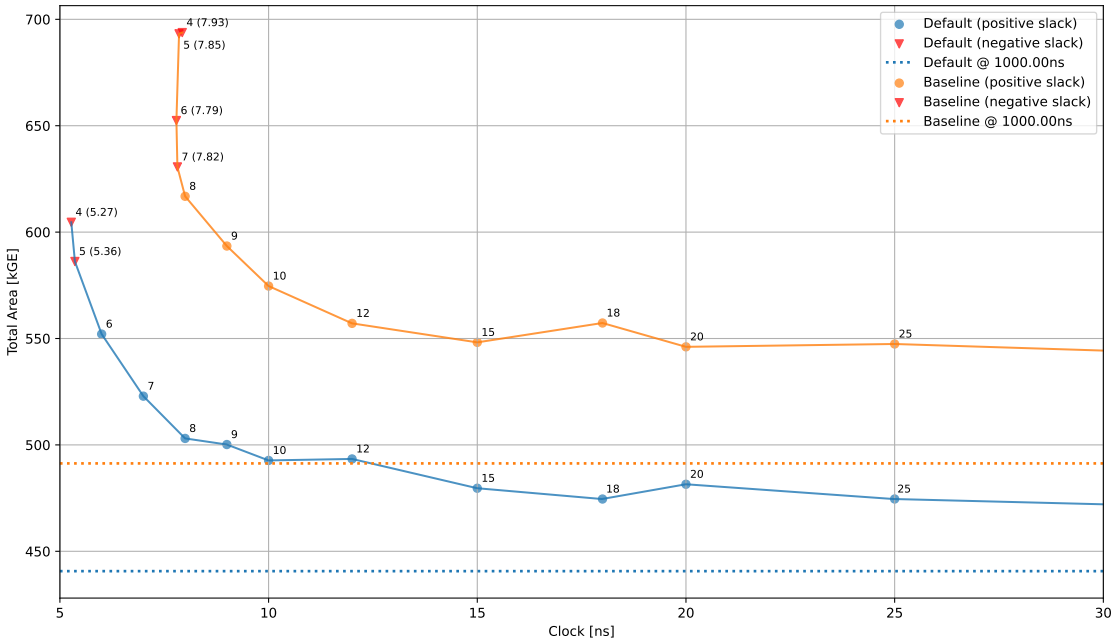

Figure 6.1.: AT plot of the default OTBN (Default) and the OTBN with the proposed instructions (Baseline) for a clock sweep. Red markers designate a timing violation, i.e., the slack is negative. The target clock is listed next to the points. For cases with timing violations, the achievable clock is stated in parentheses.

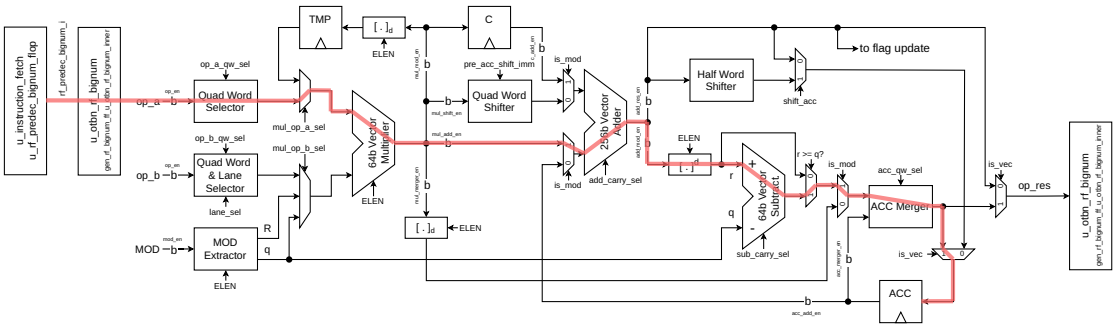

Figure 6.2.: The critical path, marked in red, through the BN MAC of the baseline OTBN design for a target clock of 8 ns.

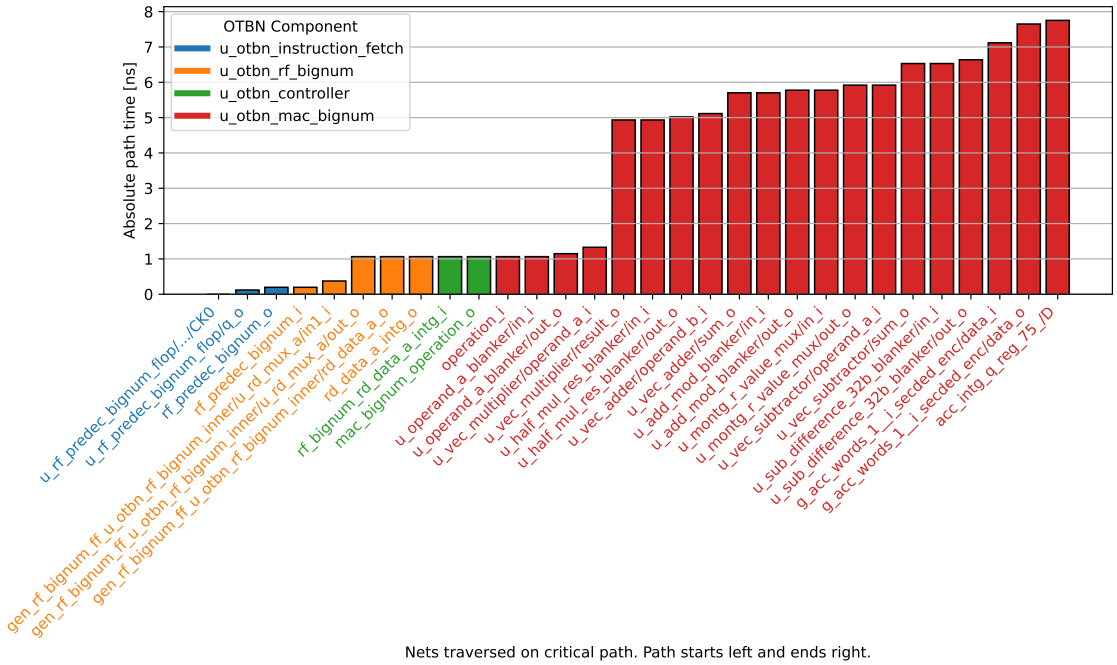

Figure 6.3.: This figure shows where time is lost on the critical path for the baseline OTBN at 8 ns. Each bar represents the absolute time required for the signal to reach this net.

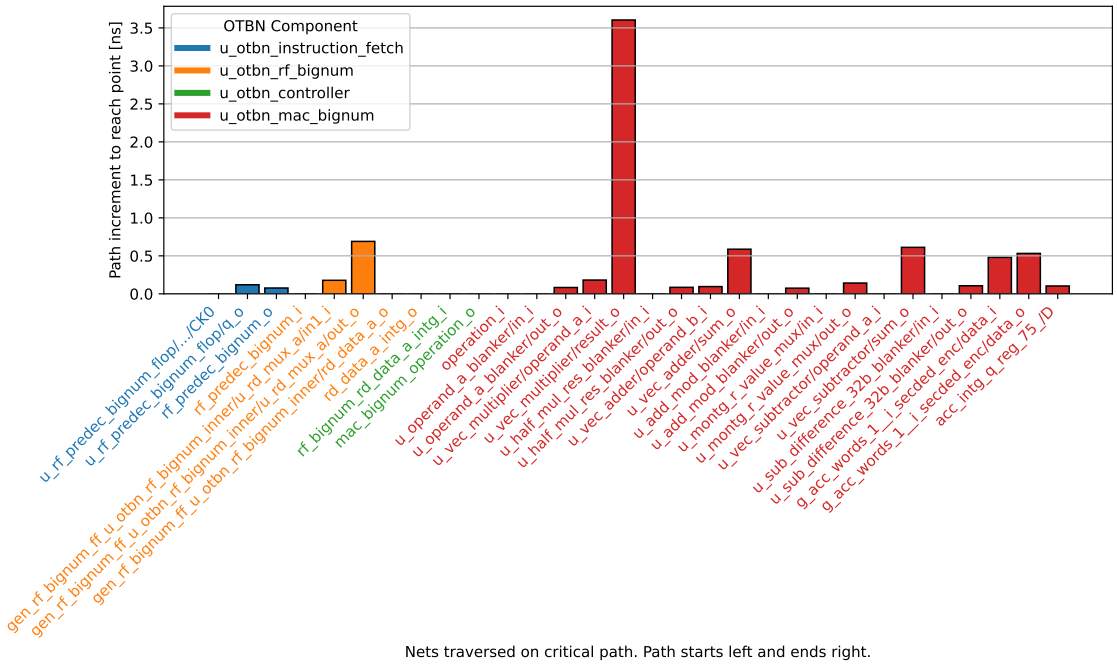

Figure 6.4.: This figure shows where time is lost on the critical path for the baseline OTBN at 8 ns. Each bar represents the increment in delay when the signal propagates from one net to the next net.

In addition to knowing where the critical path runs along, it is important to understand which parts contribute to the delay and to what extent. This information is presented in Figure 6.3. The horizontal axis represents the critical path by listing nets in the order in which the signal propagates, i.e., the signal propagates from left to right. The bar for each net corresponds to the absolute time required until the signal arrives at this net. A jump therefore represents that the circuitry to this point takes a long time. An alternative representation is shown in Figure 6.4 which shows the increment in delay from one net to the next net. From this representation, it is clearly evident which components contribute to the signal propagation delay. The most contributing component is the vectorized multiplier, accounting for 3.6041 ns. Other bottlenecks are the Bignum register file read port (the orange part, 0.8683 ns) and the redundancy encoding of the ACC write data (the last step, 0.5321 ns). In regard to a design optimization, both of these are of lesser interest because these cannot be changed easily. The most interesting part is the vectorized multiplier. However, due to the chosen 'reuse' architecture and the required blankers, the vectorized multiplier does not allow for significant optimizations. Therefore, three other decent contributors remain: the vectorized adder and subtractor as well as the ACC merger part. These add 0.5875 ns, 0.6119 ns and 0.4801 ns, respectively. The possible design optimizations based on these findings are explored in Chapter 7.

Chapter 7

Design optimization

The next step of this project was to optimize the design based on ideas gathered during the implementation and analysis of the baseline. These ideas range from algorithmic changes to RTL detail changes. Due to time constraints, only one idea was explored by implementing and benchmarking it. This chapter first explores the selected idea in detail whereas the other ideas are described in Section 8.2. The chapter then concludes with a comprehensive analysis of the benchmark and synthesis results of all implemented designs.

7.1. Optimization proposal

The analysis in Chapter 5 and Chapter 6 showed that for the gained performance the resulting area overhead is decent but the timing of the baseline design is rather deficient. In partial, the critical path is along the hardware implementing the Montgomery multiplication and one major delay contributor is the circuitry for the conditional subtraction. This circuitry performs the last step of the Montgomery multiplication (see Algorithm 6 and Section 4.2.4). However, this conditional subtraction functionality already exists in the implementation of `bn.addvm` (vectorized addition with pseudo modulo reduction). The computation of a vectorized modulo multiplication over a whole WDR can therefore be changed in the following way:

- Compute the Montgomery multiplication without the conditional subtraction for the whole vector (i.e., WDR) and store the result back into a WDR. This requires still 12 clock cycles.
- Perform a `bn.addvm` on the previously computed result where the other operand is a zero vector. This computes the conditional subtraction and requires one clock cycle.

The final result is the correct Montgomery multiplication result but computing takes 13 instead of 12 cycles compared to the baseline. This adds a theoretical performance

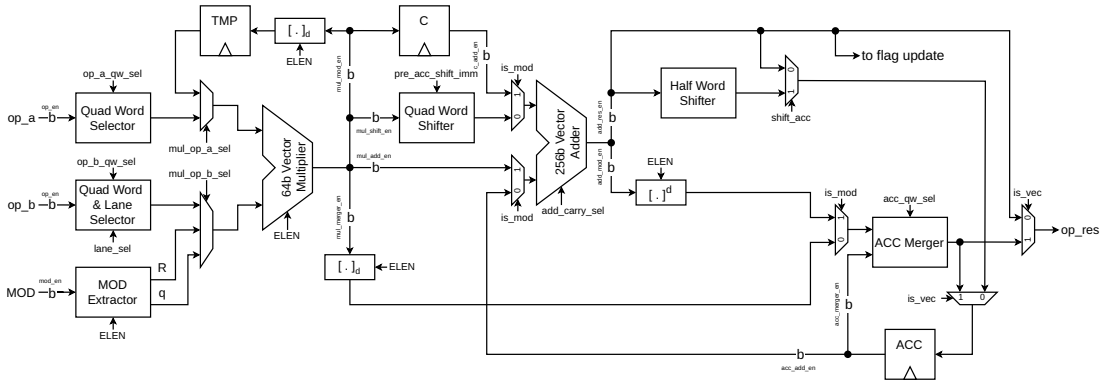

Figure 7.1.: The optimized BN MAC structure without the conditional subtraction. A lowercase 'b' represents a blanker where its control signal is listed next to it.

overhead of about 8.3 % to a vectorized Montgomery multiplication. However, it allows to remove the complete conditional subtraction circuitry resulting in a shorter critical path and smaller area requirements. These savings are more valuable as these apply at all times and not only to the computation of a vectorized Montgomery multiplication. Figure 7.1 shows the modified BN MAC architecture without the conditional subtraction components. Compared to the baseline design, the subtractor and the required MUXs, for selecting the correct results, are removed and the vectorized adder result is directly forwarded to the ACC Merger.

7.2. Benchmark results

The software performance of the optimized design was evaluated based on the same benchmarks as used in Chapter 5 for the baseline. The only modification was to integrate the conditional subtraction which requires to add a `bn.addvm` after each Montgomery multiplication. These NTT and INTT computations are implemented in the functions `ntt_mldsa_exp_reduction` and `intt_mldsa_exp_reduction` and the corresponding test programs at `sw/otbn/ntt/tests` in [25][1].

Table 7.1 lists the achieved metrics next to the results already presented in Chapter 5. With this design optimization, the achieved performance is slightly reduced as it only achieves a speed up of 3.27x whereas the baseline achieves 3.46x (−5.4 %). Still, this is a valuable improvement. The performance loss is due to the extra `bn.addvm` instructions which also appear as an increase in instruction count and code size. The optimized design requires 919 instructions resulting in a code size of 1928 B. This is 16 % worse than the baseline but still yields a 16 % improvement compared to the default design. Regarding the data memory, as expected, the optimized design requires the same memory as the baseline design.

For the INTT, these observations apply equally but the performance loss is slightly

[1] On the branch 'benchmark'. Commit `94bdc0a069d3eb3a26dd579350844315fd66e0f1`

Benchmark	ISA	Cycles	Instr.	DMEM (B)	IMEM (B)
NTT	Default	8156 (x1.00)	7711 (x1.00)	3744 (x1.00)	2296 (x1.00)
	Baseline	2360 (x3.46)	785 (x0.10)	2080 (x0.56)	1660 (x0.73)
	Optimized	2494 (x3.27)	919 (x0.12)	2080 (x0.56)	1928 (x0.84)
INTT	Default	8649 (x1.00)	8204 (x1.00)	3752 (x1.00)	2408 (x1.00)
	Baseline	2545 (x3.40)	794 (x0.10)	2080 (x0.55)	1676 (x0.70)
	Optimized	2695 (x3.21)	944 (x0.12)	2080 (x0.55)	1976 (x0.82)

Table 7.1.: Benchmark results for NTT and INTT implementations for all three designs.

more pronounced. It is only possible to achieve a speed up of 3.21x instead of 3.40x. The reason is, that the INTT contains more multiplications than the NTT due to the scaling with n^{-1}, and thus requires more additional `bn.addvm` instructions. This again is represented in the increase of code size from 1676 B to 1976 B. However, compared to the native implementation it still presents a reduction of 18 % in code size.

7.3. Synthesis results

Using the same synthesis setup as described in Section 6.1, the optimized design was synthesized for clock periods of 4 ns to 25 ns. [2] The result is plotted together with the results of the default and baseline design in Figure 7.2 and listed in Table 7.2. In the following, these results are analyzed in detail. Further analysis of the critical path for the default design can be found in Appendix B.1.

For slower clocks down to 12 ns the optimized design is only about 1 % smaller than the baseline design except at 18 ns and 20 ns where the savings are 3 % or even none. At 10 ns the designs coincide but for faster clocks, a clear separation is visible until both designs hit their timing boundary at around 7.8 ns and 7.3 ns for the baseline and the optimized design, respectively. The savings at 9 ns is 2 % and at 8 ns it amounts to 4 % which is a considerable saving. Compared to the default design at 8 ns, the optimization reduces the area overhead of the extension from 23 % to 19 %.

Interestingly to know, despite the total required area, is to understand which components contribute to the area overhead the most. In the following, the designs are analyzed in detail in order to understand the area overhead. This analysis was performed for the results of the 8 ns clock period with the same reasoning as in Chapter 6. The critical path of the optimized design is analyzed in a separate discussion, see Section 7.3.1.

Starting at the OTBN component level, the area requirements of the modified components are presented in Table 7.3 and Figure 7.3. Whereas in the default design, the

[2] Designs can be found in [25]. Default and Baseline are the same as in Chapter 6.
Optimized: `1c10283e77adcb1f384f08d6ca64eee5a19c14be`

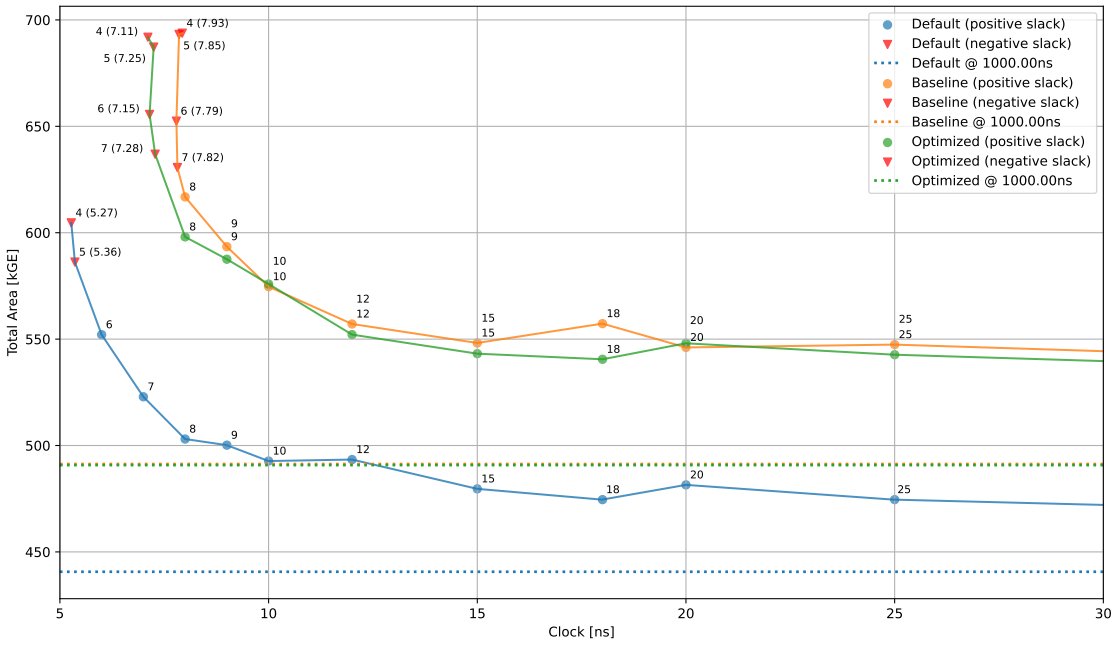

Figure 7.2.: AT plot of the default OTBN, the baseline and the optimized design without the conditional subtraction. Red markers designate a timing violation, i.e., the slack is negative. The target clock is listed next to the points. For the cases with timing violations, the achievable clock is stated in parentheses.

BN MAC is smaller than the BN ALU (48.508 kGE vs. 60.153 kGE), in both, the baseline and optimized design, the BN MAC is significantly larger. The BN MAC requires up to 113.206 kGE and 101.671 kGE for the baseline and optimized design, which is an overhead of 133 % and 110 %, respectively. Breaking down the BN MAC into its components, as shown in Table 7.4, reveals that the multiplier is the major contributor to the area overhead for both designs. It requires about 56 % of the total BN MAC area where most of it is actual combinatorial logic as the internal blankers contribute only about 4 % to the total multiplier area. A comparison to the default multiplier is not possible as the synthesis report does not explicitly state the multiplier area for the default design. The component 'Untraceable' includes all the area which is not explicitly named in the synthesis report and thus it is not possible to break it down further.

Returning the focus back to the OTBN components, the BN ALU changes only moderately from 60.153 kGE to 74.848 kGE (24 %) and 76.668 kGE (28 %). The breakdown into the components of the BN ALU is shown in Table 7.5. The biggest contributors are the untraceable elements with 34.102 kGE and 34.311 kGE followed by the vectorized shifter with 14.603 kGE and 15.446 kGE for the baseline and optimized design, respectively. For unknown reasons, the optimized BN ALU is slightly larger than the baseline's despite there being no differences to the BN ALU. One assumption is that this results from how the tool optimizes certain paths.

Of the remaining OTBN components, the controller stays approximately the same with 33.735 kGE and 32.842 kGE for the baseline and optimized design, respectively. However, the decoder and predecode more than triple in size. The decoder increases from 0.743 kGE to 2.684 kGE and 2.759 kGE. The predecode increases from 0.943 kGE to 2.850 kGE and 2.839 kGE. Breaking down the exact structure is not possible as the synthesis report does not provide this information. However, it is assumed that the increase is due to the additional control signals required for the new instructions and the extended decoder structure.

7.3.1. Critical path analysis

The critical path of the optimized design, shown in Figure 7.4, takes 7.7525 ns (slack is 0.0 ns) and follows a similar path as for the baseline design (7.7532 ns, 0.0001 ns). It originates at the flop which predecodes the Bignum register file address and propagates through the Bignum register file and controller into the BN MAC. There it follows, the same path as in the baseline design, through the 'operand a quad word selector', vectorized multiplier and the vectorized adder. Here the path diverges and does not end at the ACC WSR. Instead, it continues to the BN MAC output and ends in the write port of the Bignum register file. The 2nd most critical path is similar but ends at the flag registers in BN ALU (7.7020 ns, 0.0001 ns). It starts at the same point, but when entering the BN MAC via the operand b, it traverses the lane MUX and then follows the same path until after the vectorized adder. From there, it diverges into the flag update logic and therefore ends at the flag registers in the BN ALU. The 3rd most critical path is similar to the 1st but ends into the ACC instead of the Bignum register file (7.7524 ns, 0.0002 ns). The 4th and some following paths are again the same as the

	Total Area (kGE)			**Slack** (ns)		
Clock (ns)	Default	Baseline	Optimized	Default	Baseline	Optimized
4	604.620 (x1.00)	693.770 (x1.15)	691.868 (x1.14)	−1.273	−3.932	−3.109
5	586.196 (x1.00)	693.185 (x1.18)	687.218 (x1.17)	−0.359	−2.855	−2.246
6	552.083 (x1.00)	652.428 (x1.18)	655.577 (x1.19)	0.000	−1.791	−1.148
7	522.902 (x1.00)	630.631 (x1.21)	636.900 (x1.22)	0.000	−0.815	−0.282
8	503.044 (x1.00)	616.833 (x1.23)	598.051 (x1.19)	0.000	0.000	0.000
9	500.171 (x1.00)	593.459 (x1.19)	587.549 (x1.17)	0.001	0.000	0.000
10	492.702 (x1.00)	574.656 (x1.17)	575.900 (x1.17)	0.000	0.000	0.000
12	493.400 (x1.00)	557.115 (x1.13)	552.117 (x1.12)	0.000	0.000	0.000
15	479.635 (x1.00)	548.188 (x1.14)	543.161 (x1.13)	0.001	0.000	0.000
18	474.565 (x1.00)	557.287 (x1.17)	540.527 (x1.14)	0.013	0.001	0.000
20	481.501 (x1.00)	546.086 (x1.13)	548.039 (x1.14)	0.001	0.000	0.004
25	474.578 (x1.00)	547.423 (x1.15)	542.673 (x1.14)	0.035	0.002	0.000
1000	440.669 (x1.00)	491.302 (x1.11)	490.791 (x1.11)	906.594	884.401	888.516

Table 7.2.: Synthesis results for all three OTBN design variants.

	Total Area (kGE)		
Component	Default	Baseline	Optimized
Bignum MAC	48.508 (x1.00)	113.206 (x2.33)	101.671 (x2.10)
Bignum ALU	60.153 (x1.00)	74.848 (x1.24)	76.668 (x1.28)
Decoder	0.743 (x1.00)	2.684 (x3.61)	2.759 (x3.71)
Predecode	0.943 (x1.00)	2.850 (x3.02)	2.839 (x3.01)
Controller	32.161 (x1.00)	33.735 (x1.05)	32.842 (x1.02)
OTBN	503.044 (x1.00)	616.833 (x1.23)	598.051 (x1.19)

Table 7.3.: Area requirements of the modified OTBN components for all three designs at 8 ns.

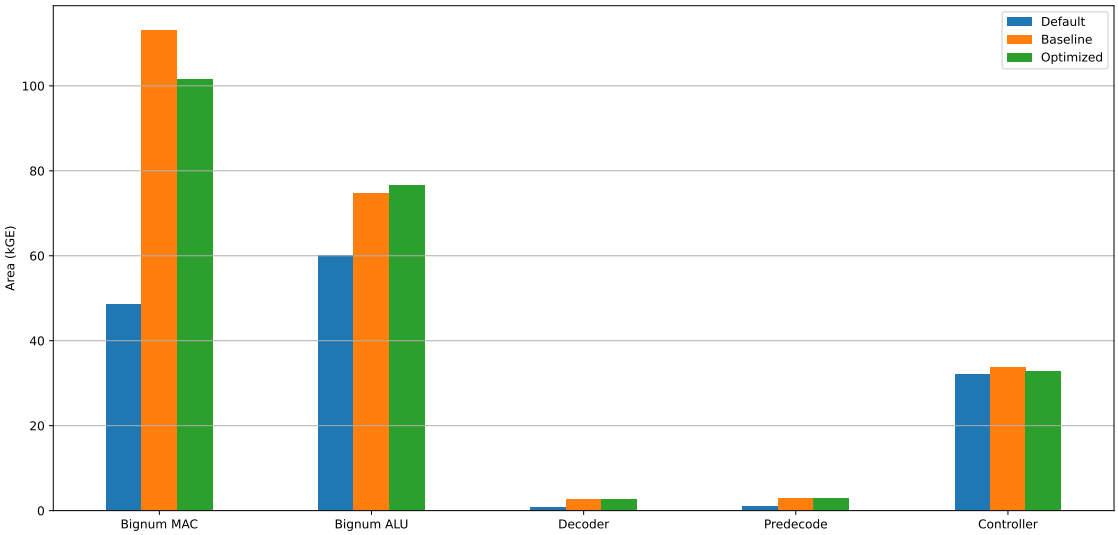

Figure 7.3.: Area requirements of OTBN components in kGE for all three designs at 8 ns.

	Total Area (kGE)	
Component	Baseline	Optimized
Vectorized Multiplier	63.887	57.908
Blankers	(2.532)	(2.171)
Untraceable	(61.355)	(55.737)
Vectorized Adder	5.667	5.938
Vectorized Subtractor	1.847	0.0
Blankers	5.867	4.862
Untraceable	35.938	32.964
Total	113.206	101.671

Table 7.4.: Area breakdown of the BN MAC at 8 ns. Blankers does not include the blankers inside the other components.

Component	**Total Area** (kGE)	
	Baseline	Optimized
Vectorized Modulo Replicator	3.004	2.997
Vectorized Adder X	5.756	5.996
Vectorized Adder Y	5.761	6.019
Vectorized Result Selector	1.359	1.630
Vectorized Shifter	14.630	15.446
Vectorized Transposer	6.432	6.435
Blankers	3.803	3.833
Untraceable	34.102	34.311
Total	74.848	76.668

Table 7.5.: Area breakdown of the BN ALU at 8 ns. Blankers does not include the blankers inside the other components.

first (4th: 7.7529 ns, 0.0002 ns).

Figure 7.5 shows the distribution of the delay along the most critical path. On the horizontal axis the critical path is represented by listing nets in the order in which the signal propagates, i.e., the signal propagates from left to right. The bar for each net corresponds to the absolute time required until the signal arrives at this net. A jump therefore represents that the circuitry to this point takes a long time. An alternative representation is shown in Figure 7.6 which shows the increment in delay from one net to the next net. As the path is almost the same as for the baseline design, the delay is also distributed similarly. The first major contributor is the Bignum register file, where the operands are read from, which adds 0.8922 ns. Inside the BN MAC, the vectorized

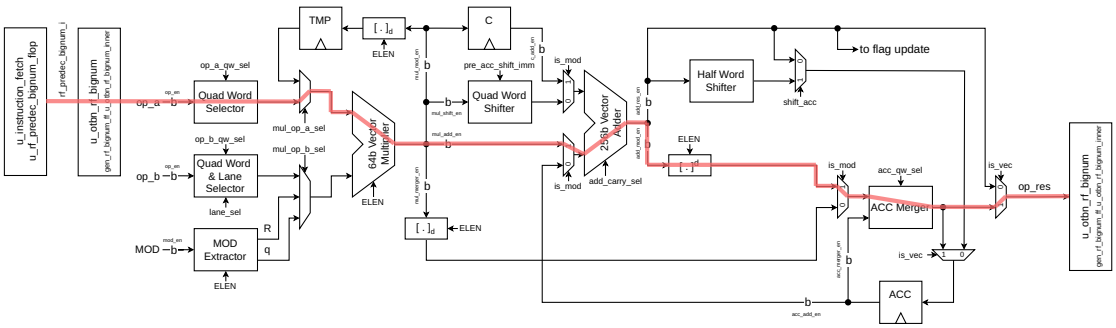

Figure 7.4.: The critical path, marked in red, through the BN MAC of the optimized OTBN design for a target clock of 8 ns.

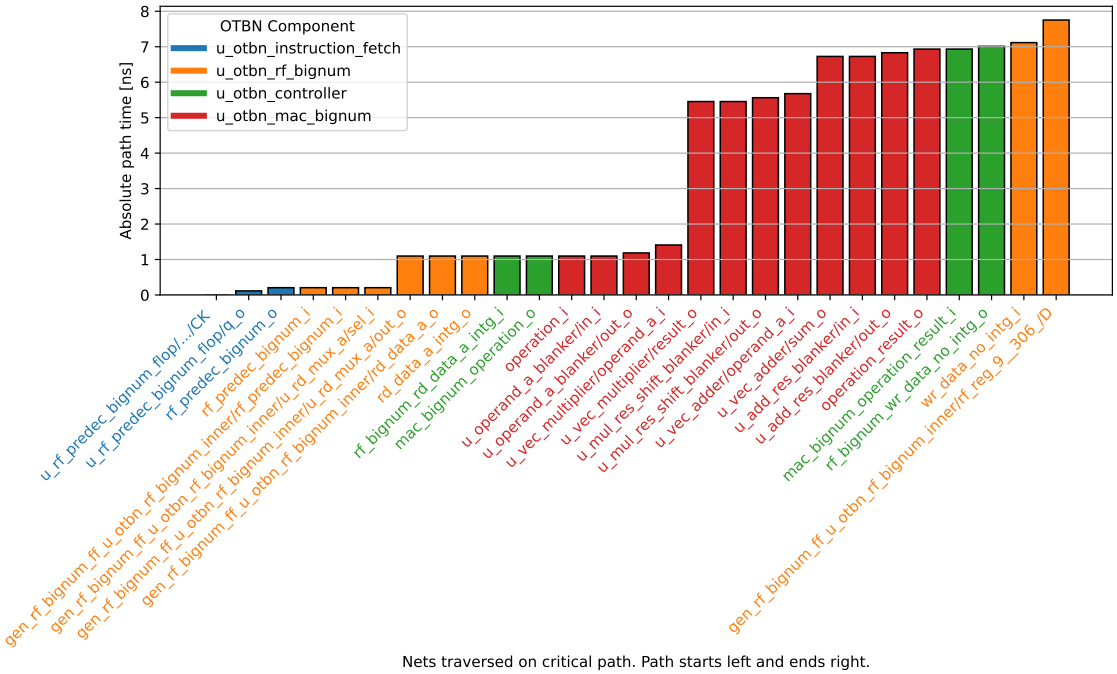

Figure 7.5.: This figure shows where time is lost on the critical path for the optimized design at 8 ns. Each bar represents the time required to reach this net in the design.

multiplier requires 4.0450 ns, slightly more than in the baseline design (3.6041 ns). Next, the vectorized adder adds 1.0496 ns (0.5875 ns for baseline). The final considerable delay is the actual write back into the Bignum register file which requires 1.8714 ns. Compared to the baseline, the delays of the vectorized multiplier and adder are higher allowing the synthesis to generate more optimized paths and this is probably where the area savings originate from. In case the critical paths of the different designs are to be compared, normalized figures are given in Appendix B.2.

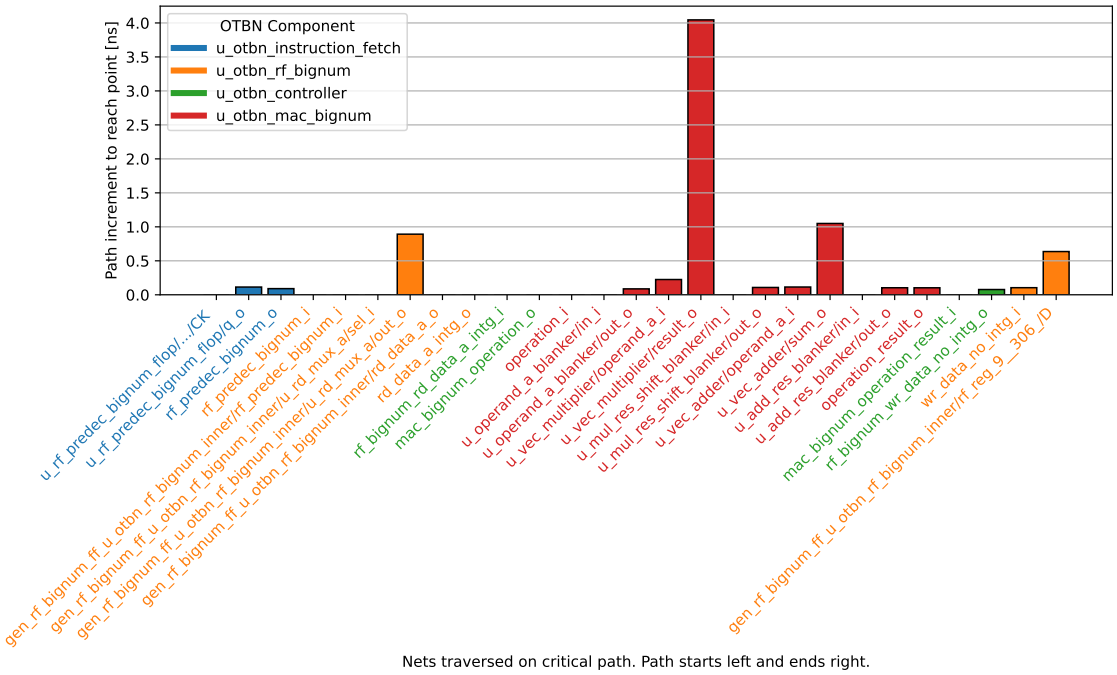

Figure 7.6.: This figure shows where time is lost on the critical path for the optimized design at 8 ns. Each bar represents the increment in delay when the signal propagates from one net to the next net.

Chapter 8

Conclusion and Future Work

8.1. Conclusion

In this project, we designed an optimized SIMD extension for OTBN to enable efficient computations of efficient polynomial arithmetic, as observed in emerging post quantum cryptography schemes. We analyzed the ML-DSA algorithm and defined a generic and lightweight ISA extension to accelerate the crucial NTT computation. We implemented the extensions in hardware, focusing on reusing existing resources and including the necessary security features such as register blanking. The performance benefit of the new instructions is demonstrated by benchmarking the NTT computation and synthesizing the design resulting in a speed up of 3.46x for an area overhead of 23 % compared to an NTT implementation based upon the default OTBN. As a next step, we analyzed the benchmark results in a broad spectrum and proposed a hardware-software co-optimization. We showed that this optimized design results in a slightly lower NTT speed up of 3.27x whilst only leading to an area overhead of 19 %. In conclusion, the elaborated extension provides a solid foundation to enable efficient implementations of PQC schemes like ML-DSA on the OTBN.

8.2. Future work

For a full and secure implementation of ML-DSA, future work is definitively required on the following topics:

- The instructions `bn.mulvm(l)` have the limitation that the source and destination WDRs may not be the same. This is due to the complexity of predecoding the register write signals and could not be addressed within the project time frame.
- The control signals for the BN MAC blankers, generated by the FSM, are unstable and therefore render certain security measures ineffective. In addition, the RTL implementation of the lane selection generates leakage across vector elements. A

solution to both problems was presented but due to time constraints, the solutions could not be implemented.

- The reviewed literature reported instruction and data memory requirements up to 64 KiB for ML-DSA whereas the current OTBN only provides 4 KiB and 8 KiB of data and instruction memory, respectively. This requires further work to investigate whether more memory is required or if there is a smart solution.

Due to time constraints, this work only explored one of many ideas on how to optimize the design. The following part presents ideas that might be worth to explore further. These ideas cover optimizations regarding performance as well as reducing the area and improving the timing.

Parallel multiplier

This idea is about increasing the performance by extending the BN MAC such that it can process 128 bits instead of 64 bit chunks. With this design, a Montgomery multiplication over a complete WDR could be performed in 6 cycles instead of 12 cycles which presents a huge performance boost. The resulting design would definitively result in a large area overhead as the multiplier is the most costly element. Furthermore, the whole BN MAC data path would have to be extended (adders, register and MUX). An idea to amortize the additional area is by simultaneously introducing a variant of the `bn.mulqacc` instruction which directly computes a 128-bit multiplication. This could also benefit big number tasks.

Support only 32-bit elements in BN MAC

When limiting the supported element width to 32 bits, the BN MAC design could be simplified. For example, the lane selection would be simpler and the vectorized multiplier could be implemented with fewer partial product generations (fewer but larger multipliers) and thus less additions and blanking is required. In regard to the BN ALU, this limitation is not as beneficial. It would only remove some small MUXs inside the vectorized adders and the modulo result selection logic. However, these components are quite small in terms of size and are not on the critical path.

Opcode complexity tradeoff

Another idea is to split the complex multiplication instructions into multiple instructions. This way, a `bn.mulvm` instruction would require the programmer to write multiple instructions in series. With this, some of the FSM logic can be transferred into the software code and thus allows a simpler hardware implementation of the control logic, especially in regards to the control signal predecoding. The drawback here is, that the OTBN has a RISC opcode architecture and thus only limited numbers of opcodes. Wasting these precious opcodes and increasing programming complexity as well as code size is probably not worth the area savings.

Alternative modulo reduction

Instead of computing a Montgomery reduction an alternative approach using recursive additions is maybe of interest. Such an approach is described in [7]. There, the prime is split into powers of two and with some algebraic tricks the reduction is converted into a small multiplication and a series of additions. However, this would require a rather drastic change in architecture and may leads to a long critical path.

Lazy reduction

It would be worth checking whether the OTBN ISA allows to perform lazy reduction on some parts of the ML-DSA computations. A lazy reduction means, that when executing a series of operations on finite field elements, the modulo reduction is only performed after the last operations. This is possible if a full reduction can be computed and the intermediate results do not overflow the word size. If possible, some expensive `bn.mulvm` instructions (12 cycles) could be replaced with 4x faster `bn.mulv` instructions.

Appendix A

Bignum MAC FSM control signals

Figure A.1 lists all BN MAC control signals for all possible multiplication types and for each FSM state.

Generated in	Signal	Multiplication type	Regular 64b	Vectorized 16b/32b				Vectorized 16b/32b lane mode				Vectorized 16b/32b Montgomery. For lane operation rename states to ModLaneQWx_x and take value right of //											
		FSM State	Regular64b	Start_QW0	QW1	QW2	DoneQW	StartLaneQW0	LaneQW1	LaneQW2	DoneLane	StartModQW0_0	ModQW0_1	ModQW0_2	ModQW1_0	ModQW1_1	ModQW1_2	ModQW2_0	ModQW2_1	ModQW2_2	ModQW3_0	ModQW3_1	DoneMod
		Comment																					
Predecoder	ELEN	Onehot encoded	64b	predecoded				predecoded				predecoded											
Predecoder	vec_adder_carry_sel		n/a	predecoded				predecoded				predecoded											
Predecoder	vec_sub_carry_sel		n/a	predecoded				predecoded				predecoded											
Decoder	lane_sel		n/a	n/a				0..15				n/a // 0..15											
Decoder / FSM	op_a_qw_sel		decoded	QW0	QW1	QW2	QW3	QW0	QW1	QW2	QW3	QW0			QW1			QW2			QW3		
Decoder / FSM	op_b_qw_sel		decoded	QW0	QW1	QW2	QW3	QWLane (decoded)				QW0 // QWLane			QW1 // QWLane			QW2 // QWLane			QW3 // QWLane		
FSM	mul_op_a_tmp_sel		a	a				a				a	tmp	tmp	a	tmp	tmp	a	tmp	tmp	a	tmp	tmp
FSM	mul_op_b_sel		b	b				b				b	R	q	b	R	q	b	R	q	b	R	q
FSM	acc_qw_sel		n/a	QW0	QW1	QW2	QW3	QW0	QW1	QW2	QW3	QW0			QW1			QW2			QW3		
Decoder	shift_acc		decoded	n/a				n/a				n/a											
Decoder	pre_acc_shift_imm		decoded	n/a				n/a				n/a											
Predecoder	is_mod	Used for Blankers and MUXs	0	0				0				1											
Predecoder	is_vec	Used for MUXs	0	1				1				1											
Predecoder	mul_type		Regular	Vectorized				VectorizedLane				VectorizedMod // VectorizedModLane											
Controller	n/a	Continue to next state	0	mac_en_i & mac_commit_i (This condition also applies for all register wr signals)																			
FSM	tmp_wr_en		0	0				0				1	1	0	1	1	0	1	1	0	1	1	0
FSM	c_wr_en		0	0				0				1	0	0	1	0	0	1	0	0	1	0	0
FSM	acc_wr_en		mac_en_i & mac_commit_i	mac_en_i & mac_commit_i				mac_en_i & mac_commit_i				0	0	1	0	0	1	0	0	1	0	0	1
Predecoder	op_en		1	1				1				1											
Predecoder	mul_shift_en	Can't be replaced with ~is_vec	1	0				0				0											
FSM	mul_add_en	Should be predecoded	0	0				0				0	0	1	0	0	1	0	0	1	0	0	1
FSM	mul_mod_en	Should be predecoded	0	0				0				1	1	0	1	1	0	1	1	0	1	1	0
Predecoder	mul_merger_en		0	1				1				0											
FSM	c_add_en	Should be predecoded	0	0				0				0	0	1	0	0	1	0	0	1	0	0	1
Predecoder	add_res_en	Can't be replaced with ~is_vec	1	0				0				0											
FSM	add_mod_en	Should be predecoded	0	0				0				0	0	1	0	0	1	0	0	1	0	0	1
Predecoder	acc_add_en	Use to reset acc	predecoded (.Z)	0				0				0											
FSM	acc_merger_en	Should be predecoded	0	0	1			0	1			0	0	1	0	0	1	0	0	1	0	0	1
Decoder	zero_acc	See "acc_add_en"	decoded	n/a				n/a				n/a											
FSM	valid_o		1	0			1	0			1	0											1

Figure A.1.: The BN MAC FSM control signal values for each state.

Appendix B

Synthesis

This appendix contains additional results as well as details about the synthesis setup used. The synthesis is performed using the TSMC65 technology.

B.1. Critical path of default OTBN

Contrary to the baseline and optimized designs, the critical path of the default OTBN design, at 8 ns, is through the BN ALU. This path is shown in Figure B.1 and requires 7.6179 ns. It starts also at the flops holding the predecoded control signals for the Bignum register file. Then passages the Bignum register file read port 'rd_data_b_intg_o' and enters the BN ALU as operand b for the shifter. After the shifter the path leads through the blanker and from there the actual path it is not exactly clear as the report only states library cells. Presumably the path goes through the Adder Y computation and then into the result MUX. The next reported net is the result leaving the BN ALU. From there it propagates straight through integrity encoding blocks and finally into the write port of the Bignum register file.

The delay distribution is shown in Figure B.2 and Figure B.3. There are two main contributors to the critical path. Firstly, the shifter adds a delay of 1.9615 ns and secondly, the part between the blanker and the result MUX adds 2.6459 ns. The second part most probably includes the Adder Y computation.

B.2. Critical path comparison of all designs

In the following, the critical paths of the default, baseline and optimized designs are compared. For each design there is a figure where the critical path is normalized. With these plots it is easier to compare the effects of the contributing elements between two designs. Figure B.4 shows the default design, Figure B.5 the baseline design and Figure B.6 the optimized design.

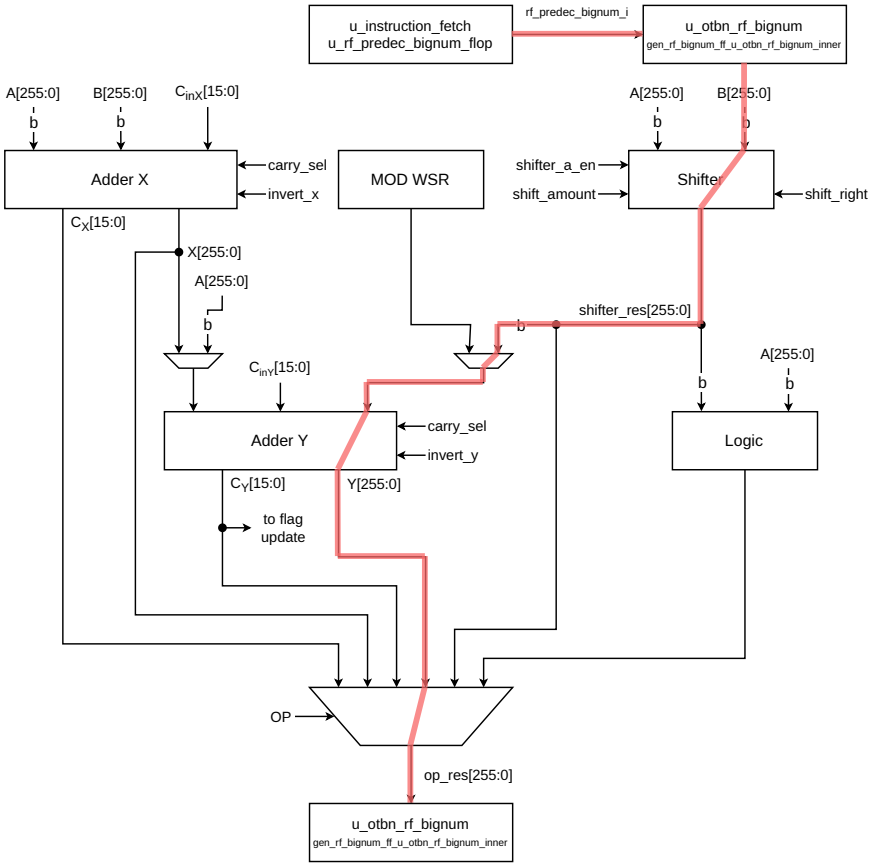

Figure B.1.: The red paths marks the critical path through the BN ALU of the default OTBN design for a target clock of 8 ns.

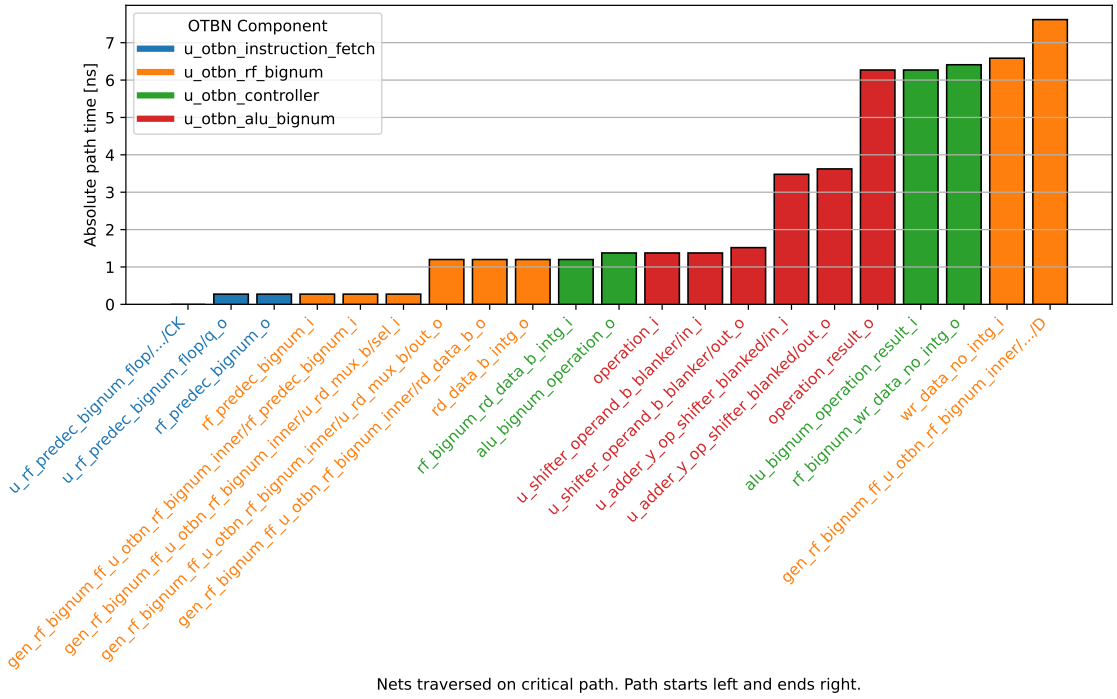

Figure B.2.: This figure shows where time is lost on the critical path for the default design at 8 ns. Each bar represents the time required to reach this net in the design.

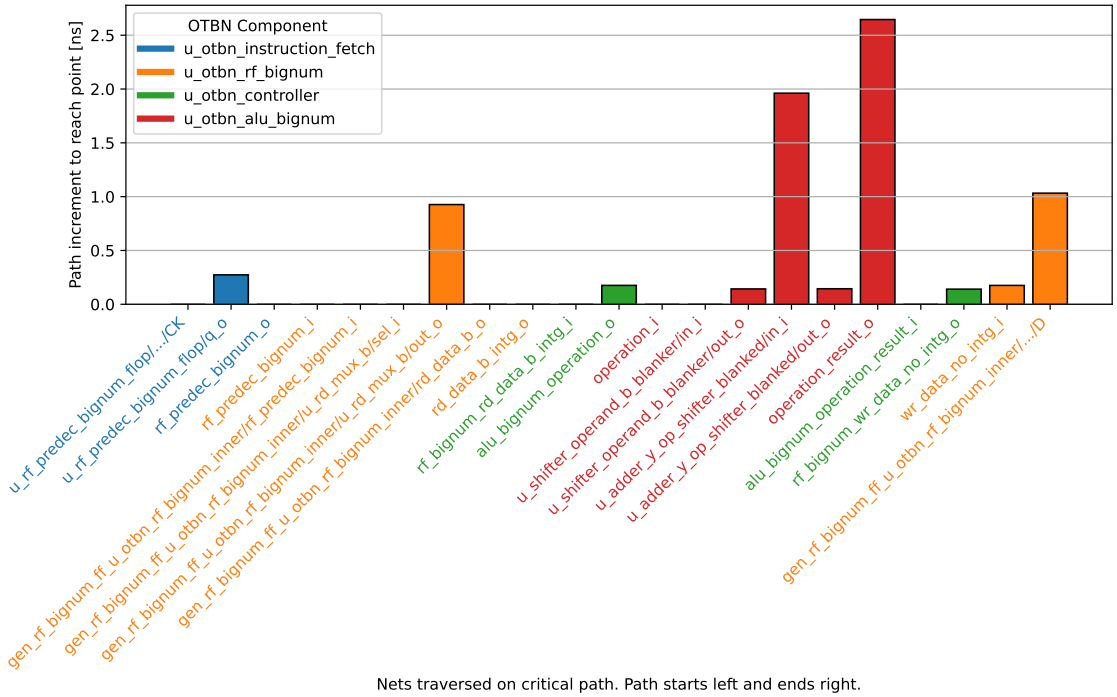

Figure B.3.: This figure shows where time is lost on the critical path for the default design at 8 ns. Each bar represents the increment in delay when the signal propagates from one net to the next net.

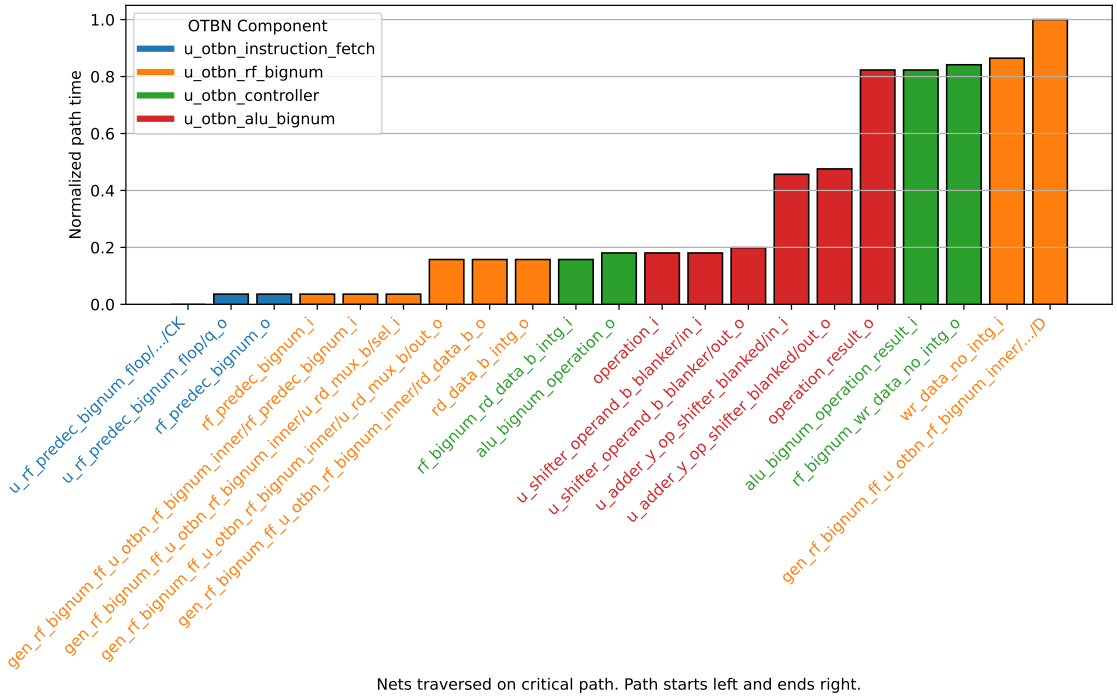

Figure B.4.: Default design at 8 ns. Each bar represents the normalized time required to reach this net in the design.

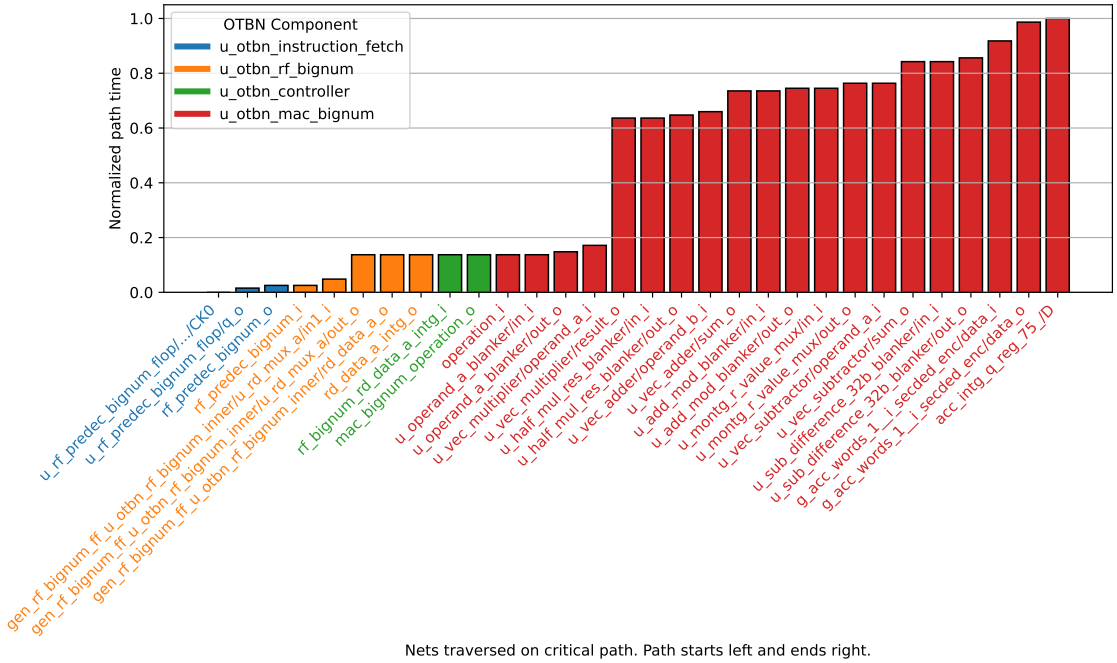

Figure B.5.: Baseline design at 8 ns. Each bar represents the normalized time required to reach this net in the design.

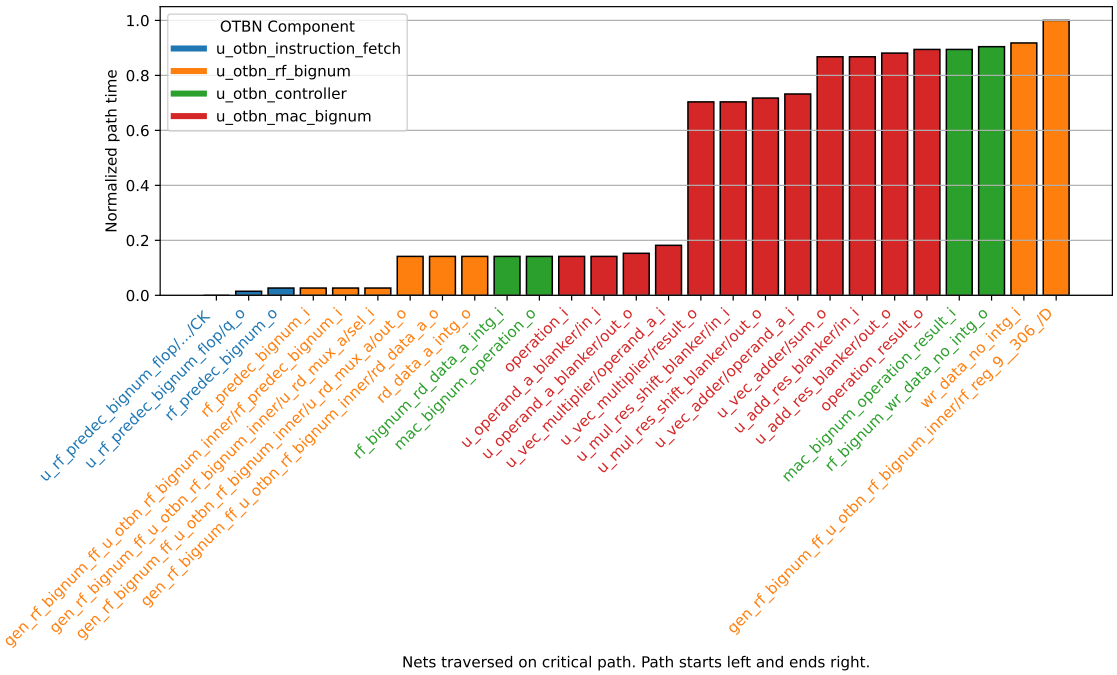

Figure B.6.: Optimized design at 8 ns. Each bar represents the normalized time required to reach this net in the design.

Parameter	Value
Clock uncertainty (setup)	0.2
Clock uncertainty (hold)	0.1
Maximal transition (Clock and Data)	0.1
Input delay	0.3 · clock period
Output delay	0.3 · clock period

Table B.1.: Synthesis constraints used for all designs.

B.3. Setup details

The synthesis was performed for TSMC65 with Synopsis 2019.03 with the constraints given in Table B.1. To set the special constraint for the security related modules, a 'set_size_only' constraint was applied to all instances. This was done by adding a special name tag to the module and then searching through all design components for this tag. The synthesis was run with an established setup created by the IIS. This setup uses the `compile_ultra` command with the option `-no_autoungroup` set.

Appendix C

Task Description

Institut für Integrierte Systeme
Integrated Systems Laboratory

TASK ASSIGNMENT FOR A
SEMESTER THESIS AT THE DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

FALL SEMESTER 2024

Pascal Etterli

Design and Optimization of a PQC ISA Extension for OTBN

16.09.2024

Advisors: Dr. Pirmin Vogel, vogelpi@lowrisc.org
Dr. Pascal Nasahl, nasahlpa@lowrisc.org
Navaneeth Kunhi Purayil, nkunhi@iis.ee.ethz.ch

Professor: Prof. Dr. Luca Benini

Handout: September 16, 2024
Due: December 20, 2024

1 Introduction

OpenTitan is a collaborative, open-source hardware ecosystem that bundles an array of individual IPs for deployment in a wide range of applications alongside the top-level designs *Earl Grey*, a standalone micro-controller, and *Darjeeling*, an integrated execution environment. The principal module of the project is the Ibex core, a 32-bit general-purpose RISC-V processor implementing the Integer (I), Integer Multiplication/-Division (M), Compressed (C) and Bit Manipulation (B) instruction sets. Although the performance of the Ibex core is adequate for most programs it lacks the capabilities for the efficient execution of big-number arithmetic that is often found in cryptographic algorithms. As a remedy, OpenTitan features a second core termed the OpenTitan Big-Number Accelerator (OTBN) whose base instruction set, reminiscent of RV32I, is extended with big-number instructions that operate on 256-bit registers. This means that, in the presence of the OTBN, the Ibex core can offload certain computations, e.g., RSA exponentiation, which can significantly improve the overall runtime of a program.

Generally speaking, invoking the accelerator is only sensible if the cost of moving data between Ibex and the big-number accelerator plus the runtime of the loaded OTBN program improves on an Ibex-only execution, which would not be the case for the computation of AES block cipher (a dedicated IP block already exists), but might be for lattice-based algorithms as standardized in the NIST post-quantum cryptography portfolio. In fact, the most salient computation in lattice cryptography is the Number Theoretic Transform (NTT) which is a discrete variant of the Fast Fourier Transform. More specifically, the NTT is an in-place algorithm composed of a repeated computation of a specific function (called butterfly) over the ring of integers modulo a small prime number (usually smaller than 32 bits), which would benefit from Single Instruction Multiple Data (SIMD) vector instructions that compute multiple butterflies in parallel. Note that the current iteration of the OTBN does not feature any kind of vectorization in the big-number instruction set.

2 Project Description

Accelerating the NTT with SIMD instructions is not a novel idea and has been investigated in the literature before in the context of RISV-V/OTBN ISA extensions leading to several competing designs [1, 2, 3, 4, 5, 6, 7] exhibiting various trade-offs. The goal of this project lies in the consolidation of those schemes and an ultimate proposal proposal of an efficient OTBN SIMD ISA extension that is suitable for lattice-based cryptography but also useful for other applications that can benefit from vectorized computations. The project proceeds in three phases that build on each other (a rough time frame is assigned to each phase):

2.1 Phase 1 (Fundamentals, 3 Weeks)

The first phase of the project is dedicated to literature review and becoming comfortable with the OpenTitan/OTBN toolchain. More specifically, the student should attain sufficient knowledge of the state of the art SIMD RISC-V ISA extensions in order to derive a detailed comparison and identify the relevant optimization strategies. Concurrently, the student should be able to write OTBN programs and test them both in the Python simulator and the Verilator test rig.

Milestones:

- Refresh knowledge of programming in C and RISC-V assembly. Use that knowledge to dive into the OpenTitan/OTBN toolchain.
- Write some OTBN programs, e.g., a straightforward NTT implementation, and run them in the Python simulator or simulate them via Verilator.
- Create a thorough comparison of all the optimizations from the linked literature and tabulate their efficiency gains in terms of runtime, memory, hardware footprint and identify the most relevant techniques.

2.2 Phase 2 (Baseline, 5 Weeks)

In the second phase, the student implements a baseline OTBN SIMD ISA extension for a predetermined set of functions, for instance ADD/MUL etc., and creates a benchmarking framework to extract its metrics. The ISA extension can either stem directly from a paper or be a cherry-picked selection from multiple sources.

Milestones:

- Refresh knowledge of SystemVerilog and RTL programming in general. Dive into the OTBN source code and implement the chosen ISA extension.
- Using the available tools at ETH, prepare a synthesis flow to compile RTL designs into netlists to extract circuit area and timing figures. Having a working synthesis flow will save time when it comes to the implementation in the second phase. The supervisors from ETH and lowRISC will be assisting the student in this task.
- Thoroughly test the implementation with a comprehensive test suite. Verify the efficiency gains by running a program against it, for example the NTT (for small and large prime numbers), and report the runtime (cycles) and memory (bytes) footprint.
- Synthesize the new OTBN with the available synthesis tools at ETH and tabulate the induced hardware overhead both for circuit area and timing (critical path).

2.3 Phase 3 (Optimization, 6 Weeks)

Lastly, the third phase is concerned with optimisation and is thus open-ended. The student is expected to research the potential improvements to the implemented ISA extension from the previous phase. The goal is to reach an efficient low-overhead implementation that both accelerates the computation of the NTT in lattice-based algorithms but is simultaneously general enough to be of use for other applications as well that could benefit from vectorized OTBN instructions.

Milestones:

- Research potential optimisation in the field of SIMD RISC-V instructions and apply them to the baseline implementation.
- Report efficiency improvements and corresponding overheads.
- Start writing the final report.

3 Project Realization

The progress of the project is tracked with weekly write-ups by the student and discussed in a weekly meeting. In these write-ups (length at most 1 page), the student summarizes all the tasks he tackled in the previous week and describes the plan for the current week. If the student wishes, an intermediate report can be submitted after the completion of the second phase, which can serve as a partial draft of the final report.

3.1 Report

Documentation is an important and often overlooked aspect of engineering. A final report has to be completed within this project.

The common language of engineering is de facto English. Therefore, the final report of the work is preferred to be written in English.

Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Inkscape or Tgif[1] or any other vector drawing software (for block diagrams) is strongly encouraged by the IIS staff. If you write the report in LaTeX, we offer an instructive, ready-to-use template, which can be downloaded at `https://iis-people.ee.ethz.ch/~vlsi1/templates/report.tar.gz`.

[1] Tgif is a simple vector drawing software, quite useful for drawing block diagrams. For further information about Tgif we refer to `http://bourbon.usc.edu/tgif/vector.html` and `http://eda.ee.ethz.ch/index.php/Tgif`.

Final Report The final report has to be presented at the end of the project and a digital as well as one printed copy need to be handed in and remain property of the IIS. Note that this task description is part of your report and has to be attached to your final report.

3.2 Presentation

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

3.3 HDL Guidelines

Naming Conventions: Adapting a consistent naming scheme is one of the most important steps in order to make your code easy to understand. If signals, processes, and entities are always named the same way, any inconsistency can be detected easier. Moreover, if a design group shares the same naming convention, all members would immediately *feel at home* with each others code. The naming conventions we follow in the PULP project are at `https://github.com/pulp-platform/style-guidelines` for reference.

4 Deliverables

In order to complete the project successfully, the following deliverables have to be submitted at the end of the work:

- Project plan
- Final report incl. presentation slides
- Source code and documentation for all developed software and hardware
- Testsuites (software) and testbenches (hardware)
- Synthesis and implementation scripts
- FPGA bitstreams

References

[1] A. Abdulrahman, F. Oberhansl, H. N. H. Pham, J. Philipoom, P. Schwabe, T. Stelzer, and A. Zankl, "Towards ML-KEM ML-DSA on OpenTitan,"

Cryptology ePrint Archive, Paper 2024/1192, 2024. [Online]. Available: https://eprint.iacr.org/2024/1192

[2] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic - accelerating kyber and NewHope on RISC-v," Cryptology ePrint Archive, Paper 2020/049, 2020. [Online]. Available: https://eprint.iacr.org/2020/049

[3] N. Gupta, A. Jati, A. Chattopadhyay, and G. Jha, "Lightweight hardware accelerator for post-quantum digital signature CRYSTALS-dilithium," Cryptology ePrint Archive, Paper 2022/496, 2022. [Online]. Available: https://eprint.iacr.org/2022/496

[4] C. Zhao, N. Zhang, H. Wang, B. Yang, W. Zhu, Z. Li, M. Zhu, S. Yin, S. Wei, and L. Liu, "A compact and high-performance hardware architecture for crystals-dilithium," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, Issue 1, pp. 270–295, 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9297

[5] K. Miteloudi, J. Bos, O. Bronchain, B. Fay, and J. Renes, "PQ.v.ALU.e: Post-quantum RISC-v custom ALU extensions on dilithium and kyber," Cryptology ePrint Archive, Paper 2023/1505, 2023. [Online]. Available: https://eprint.iacr.org/2023/1505

[6] Z. Ye, R. Song, H. Zhang, D. Chen, R. C. Cheung, and K. Huang, "A highly-efficient lattice-based post-quantum cryptography processor for iot applications," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2024, no. 2, pp. 130–153, 2024. [Online]. Available: https://doi.org/10.46586/tches.v2024.i2.130-153

[7] M. J. Kannwischer, "Polynomial Multiplication for Post-Quantum Cryptography," Ph.D. dissertation, Nijmegen U., 2022.

Zurich, 16.09.2024 Prof. Dr. Luca Benini

List of Acronyms

AES . . . . . Advanced Encryption Standard

ALU . . . . . arithmetic logic unit

ASIC . . . . . application-specific integrated circuit

BN ALU . . . . Bignum ALU

BN MAC . . . . Bignum MAC

DFT . . . . . discrete Fourier transform

FFT . . . . . Fast Fourier Transform

FI . . . . . fault injection

FPGA . . . . . field-programmable gate array

FSM . . . . . finite state machine

IIS . . . . . Integrated Systems Laboratory

INTT . . . . . Inverse Number Theoretic Transform

ISA . . . . . instruction set architecture

LWE . . . . . Learning with Error

LWE . . . . . Ring-Learning with Error

ML-DSA . . . . Module-Lattice-Based Digital Signature Standard

MLWE . . . . . Module-Learning with Error

MUX . . . . . .multiplexer

NIST . . . . . .National Institute of Standards and Technology

NTT . . . . . .Number Theoretic Transform

OTBN . . . . . .OpenTitan Big-Number Accelerator

PKC . . . . . .public key cryptography

PQC . . . . . .post quantum cryptography

RoT . . . . . .root of trust

RTL . . . . . .register transfer level

SCA . . . . . .side-channel analysis

SIMD . . . . . .Single Instruction Multiple Data

SISD . . . . . .Single Instruction Single Data

SoC . . . . . .system on chip

SVP . . . . . .shortest vector problem

XOF . . . . . .eXtendable-Output Function

List of Figures

2.1. A NTT and INTT with the butterflies for $n = 8$. The twiddle factors ϕ are indexed with the bit-reversed order. The scaling factor at the end of the Inverse Number Theoretic Transform (INTT) of n^{-1} is omitted. 10
2.2. Left: Cooley-Tukey butterfly. Right: Gentleman-Sande butterfly. 10
2.3. The OTBN architecture at hardware block level [23]. 16
2.4. *Left*: The original register file of the Ibex core. A multiplexer tree is used to read registers based on the 5-bit read address. Writing is done via a multiplexer, controlled by a 1-bit write-enable signal, which is derived from the write address.
Right: Secured register file. The register output is additionally gated and the multiplexer tree is replaced by a tree of OR gates. The writing mechanism remains unchanged, except that it is extended by an additional AND gate for the write data. From [24], modified. 18
2.5. Simplified pipeline of OpenTitan Big-Number Accelerator (OTBN) to highlight the blanking implementation. The predecoded signals generated in the predecoder are flopped between the instruction fetch and decode-execute stage. The register file uses onehot controlled AND-OR structures and the lowercase 'b' represents blankers on the data paths. . 19

3.1. An illustration of the proposed instructions `bn.trn1` and `bn.trn2` for vectors with four elements.
Left: The `bn.trn1` places even-indexed vector elements from `<wrs1>` into even-indexed elements of `<wrd>` and even-indexed vector elements from `<wrs2>` are placed into odd-indexed elements of `<wrd>`.
Right: For `bn.trn2` odd-indexed vector elements from `<wrs1>` are placed into even-indexed elements of `<wrd>` and odd-indexed vector elements from `<wrs2>` are placed into odd-indexed elements of `<wrd>` 25
3.2. Encoding of proposed instructions. The gray marked instructions are not proposed but space is reserved in the encoding for future expansion. An 'x' represents a 'don't care' value. 26

4.1. A vectorized 256-bit adder supporting vectors with 16-bit, 32-bit, 64-bit, 128-bit or 256-bits elements. It is constructed by chaining 16 17-bit adders. The carry is either an input or originate from the result of the lower adder depending on the vector element length. 30
4.2. For the partial product generation, inputs to the unused partial product multipliers (marked red) must be blanked as otherwise leakage is generated. 31
4.3. The partial product summation circuit of the vectorized multiplier requires two blankers which are depicted with a lowercase 'b'. 32
4.4. The vectorized Bignum ALU (BN ALU) structure. The two adders X and Y are used to compute the pseudo modulo reduction. A lowercase 'b' represents a blanker. 33
4.5. The complete modulo result selector circuit. The result selection decision is computed for each 16-bit chunk independently and the forwarded to the selection stage. 34
4.6. The modulo result selection stage. Depending on the element length the appropriate decision bit D_i is forwarded to the result selection multiplexer (MUX) for each 16-bit chunk independently. 34
4.7. *Left*: The register transfer level (RTL) implementation of the vectorized shifter. The shift itself is always a right shift. Left shifting is achieved by reversing the input and output.
Right: An example where a vector consisting of four 4-bit elements is shifted by two bits to the right. First the raw input is shifted to the right. Then a mask is generated and ANDed to delete the bits which were shifted into the neighboring vector element. 36
4.8. The original unmodified Bignum MAC (BN MAC) featuring a 64-bit multiplier, 256-bit adder and a 256-bit WSR named ACC. 37
4.9. A vectorized multi cycle data flow implementation of a Montgomery multiplication. If compared to the original BN MAC the orange highlighted components could replace existing hardware. Note, this is not the actually implemented design. 39
4.10. The BN MAC structure supporting the regular functionality as well as vectorized (Montgomery) multiplication. A lowercase 'b' represents a blanker where its control signal is listed next to it. 40
4.11. The state diagram of the finite state machine (FSM) handling the BN MAC control signals. The initial state 'Regular64b' handles the single cycle multiplication and contains two sub-states which serve as starting point for both the vectorized regular and Montgomery multiplication. 41

6.1. AT plot of the default OTBN (Default) and the OTBN with the proposed instructions (Baseline) for a clock sweep. Red markers designate a timing violation, i.e., the slack is negative. The target clock is listed next to the points. For cases with timing violations, the achievable clock is stated in parentheses. 50

6.2. The critical path, marked in red, through the BN MAC of the baseline OTBN design for a target clock of 8 ns. 50
6.3. This figure shows where time is lost on the critical path for the baseline OTBN at 8 ns. Each bar represents the absolute time required for the signal to reach this net. 51
6.4. This figure shows where time is lost on the critical path for the baseline OTBN at 8 ns. Each bar represents the increment in delay when the signal propagates from one net to the next net. 52

7.1. The optimized BN MAC structure without the conditional subtraction. A lowercase 'b' represents a blanker where its control signal is listed next to it. 54
7.2. AT plot of the default OTBN, the baseline and the optimized design without the conditional subtraction. Red markers designate a timing violation, i.e., the slack is negative. The target clock is listed next to the points. For the cases with timing violations, the achievable clock is stated in parentheses. 56
7.3. Area requirements of OTBN components in kGE for all three designs at 8 ns. 59
7.4. The critical path, marked in red, through the BN MAC of the optimized OTBN design for a target clock of 8 ns. 60
7.5. This figure shows where time is lost on the critical path for the optimized design at 8 ns. Each bar represents the time required to reach this net in the design. 61
7.6. This figure shows where time is lost on the critical path for the optimized design at 8 ns. Each bar represents the increment in delay when the signal propagates from one net to the next net. 62

A.1. The BN MAC FSM control signal values for each state. 67

B.1. The red paths marks the critical path through the BN ALU of the default OTBN design for a target clock of 8 ns. 69
B.2. This figure shows where time is lost on the critical path for the default design at 8 ns. Each bar represents the time required to reach this net in the design. 70
B.3. This figure shows where time is lost on the critical path for the default design at 8 ns. Each bar represents the increment in delay when the signal propagates from one net to the next net. 71
B.4. Default design at 8 ns. Each bar represents the normalized time required to reach this net in the design. 72
B.5. Baseline design at 8 ns. Each bar represents the normalized time required to reach this net in the design. 73
B.6. Optimized design at 8 ns. Each bar represents the normalized time required to reach this net in the design. 73

List of Tables

5.1. Benchmark results for Number Theoretic Transform (NTT) and INTT implementations based on the default and proposed instruction set architecture (ISA). 47

7.1. Benchmark results for NTT and INTT implementations for all three designs. 55
7.2. Synthesis results for all three OTBN design variants. 58
7.3. Area requirements of the modified OTBN components for all three designs at 8 ns. 58
7.4. Area breakdown of the BN MAC at 8 ns. Blankers does not include the blankers inside the other components. 59
7.5. Area breakdown of the BN ALU at 8 ns. Blankers does not include the blankers inside the other components. 60

B.1. Synthesis constraints used for all designs. 74

Bibliography

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134. [Online]. Available: https://doi.org/10.1109/SFCS.1994.365700

[2] M. Mosca and M. Piani, "2022 quantum threat timeline report," Global Risc Institute, Tech. Rep., 2022. [Online]. Available: https://globalriskinstitute.org/publication/2022-quantum-threat-timeline-report/

[3] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 1, pp. 238–268, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i1.238-268

[4] NIST, *Module-Lattice-Based Digital Signature Standard (ML-DSA) (FIPS PUB 204)*, National Institute of Standards and Technology, Aug. 2024.

[5] C. Zhao, N. Zhang, H. Wang, B. Yang, W. Zhu, Z. Li, M. Zhu, S. Yin, S. Wei, and L. Liu, "A compact and high-performance hardware architecture for crystals-dilithium," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 1, pp. 270–295, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i1.270-295

[6] M. J. Kannwischer, "Polynomial multiplication for post-quantum cryptography," Ph.D. dissertation, Nijmegen University, 2022.

[7] N. Gupta, A. Jati, A. Chattopadhyay, and G. Jha, "Lightweight hardware accelerator for post-quantum digital signature CRYSTALS-dilithium," Cryptology ePrint Archive, Paper 2022/496, 2022. [Online]. Available: https://eprint.iacr.org/2022/496

[8] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic - accelerating kyber and NewHope on RISC-v," Cryptology ePrint Archive, Paper 2020/049, 2020. [Online]. Available: https://eprint.iacr.org/2020/049

[9] Z. Ye, R. Song, H. Zhang, D. Chen, R. C. Cheung, and K. Huang, "A highly-efficient lattice-based post-quantum cryptography processor for iot applications," *IACR*

Trans. Cryptogr. Hardw. Embed. Syst., vol. 2024, no. 2, pp. 130–153, 2024. [Online]. Available: https://doi.org/10.46586/tches.v2024.i2.130-153

[10] K. Miteloudi, J. W. Bos, O. Bronchain, B. Fay, and J. Renes, "PQ.V.ALU.E: post-quantum RISC-V custom ALU extensions on dilithium and kyber," *IACR Cryptol. ePrint Arch.*, p. 1505, 2023. [Online]. Available: https://eprint.iacr.org/2023/1505

[11] A. Abdulrahman, F. Oberhansl, H. N. H. Pham, J. Philipoom, P. Schwabe, T. Stelzer, and A. Zankl, "Towards ML-KEM i& ML-DSA on OpenTitan," Cryptology ePrint Archive, Paper 2024/1192, 2024. [Online]. Available: https://eprint.iacr.org/2024/1192

[12] A. Abdulrahman, "Dilithium on opentitan," Master's thesis, Ruhr University Bochum, 2023.

[13] Y. Li, K. S. Ng, and M. Purcell, "A tutorial introduction to lattice-based cryptography and homomorphic encryption," *CoRR*, vol. abs/2208.08125, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.08125

[14] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, pp. 34:1–34:40, 2009. [Online]. Available: https://doi.org/10.1145/1568318.1568324

[15] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. [Online]. Available: https://www.crcpress.com/Introduction-to-Modern-Cryptography-Second-Edition/Katz-Lindell/p/book/9781466570269

[16] NIST, *Digital Signature Standard (DSS) (FIPS PUB 186)*, National Institute of Standards and Technology, Feb. 2023.

[17] A. Satriawan and R. Mareta, "A complete beginner guide to the number theoretic transform (NTT)," *IACR Cryptol. ePrint Arch.*, p. 585, 2024. [Online]. Available: https://eprint.iacr.org/2024/585

[18] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985. [Online]. Available: https://api.semanticscholar.org/CorpusID:119574413

[19] T. Plantard, "Efficient word size modular arithmetic," *IEEE Trans. Emerg. Top. Comput.*, vol. 9, no. 3, pp. 1506–1518, 2021. [Online]. Available: https://doi.org/10.1109/TETC.2021.3073475

[20] G. Seiler, "Faster AVX2 optimized NTT multiplication for ring-lwe lattice cryptography," *IACR Cryptol. ePrint Arch.*, p. 39, 2018. [Online]. Available: http://eprint.iacr.org/2018/039

[21] J. W. Bos and P. L. Montgomery, "Montgomery arithmetic from a software perspective," *IACR Cryptol. ePrint Arch.*, p. 1057, 2017. [Online]. Available: http://eprint.iacr.org/2017/1057

[22] OpenTitan Contributors, "OpenTitan documentation," https://opentitan.org/book/doc/introduction.html, 2024, [Online; accessed 9-December-2024].

[23] ——, "Opentitan big-number accelerator documentation," https://opentitan.org/book/hw/ip/otbn/index.html, 2024, [Online; accessed 9-December-2024].

[24] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco: Co-design and co-verification of masked software implementations on cpus," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 1469–1468. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl

[25] P. Etterli. Github repository of this work. [Online]. Available: https://github.com/etterli/opentitan-otbn-pqc-isa

[26] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - kyber: A cca-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 353–367. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00032

[27] NIST, *Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) (FIPS PUB 203)*, National Institute of Standards and Technology, Aug. 2024.

[28] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A new hope," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 327–343. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim

[29] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, ser. Lecture Notes in Computer Science, A. M. Odlyzko, Ed., vol. 263. Springer, 1986, pp. 311–323. [Online]. Available: https://doi.org/10.1007/3-540-47721-7_24