ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

HARDWARE IMPLEMENTATION OF THE POST-QUANTUM CRYPTOGRAPHY ALGORITHM FALCON

M.Sc. THESIS

Yasin YILMAZ

Department of Electronics and Communication Engineering

Electronics Engineering Programme

AUGUST 2025

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

HARDWARE IMPLEMENTATION OF THE POST-QUANTUM CRYPTOGRAPHY ALGORITHM FALCON

M.Sc. THESIS

Yasin YILMAZ
(504241211)

Department of Electronics and Communication Engineering

Electronics Engineering Programme

Thesis Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

AUGUST 2025

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

KUANTUM SONRASI KRİPTOGRAFİ ALGORİTMASI FALCON'UN DONANIM GERÇEKLEMESİ

YÜKSEK LİSANS TEZİ

Yasin YILMAZ
(504241211)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Sıddıka Berna ÖRS YALÇIN

AĞUSTOS 2025

Yasin YILMAZ, a M.Sc. student of ITU Graduate School student ID 504241211 successfully defended the thesis entitled "HARDWARE IMPLEMENTATION OF THE POST-QUANTUM CRYPTOGRAPHY ALGORITHM FALCON", which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. Sıddıka Berna ÖRS YALÇIN** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Erkay SAVAŞ** Sabancı University	.....
	Prof. Dr. Şerif BAHTİYAR Istanbul Technical University	.....
		.....

Date of Submission : **08 July 2025**
Date of Defense : **01 August 2025**

to my better half, Zeynep

FOREWORD

First of all, I would like to thank my project advisor, Prof. Dr. Berna Örs Yalçın for her guidance and support through all of my studies. Secondly, I want to express my gratitude to my family, Leyla, Erdem, Yahya, Hilal and Hale, for their never-ending trust and endorsement. Also, I want to thank my dear wife Zeynep and my newer family, Hülya, Bilgehan, Atilla and Zeliha, for their warm welcome and support. And finally, I would like to thank my friends at Procenne and the SIR Team for their help and companionship.

August 2025 Yasin YILMAZ

TABLE OF CONTENTS

ABBREVIATIONS

CVP	**:** Closest Vector Problem
DMA	**:** Direct Memory Address
DRBG	**:** Deterministic Random Bit Generator
ECC	**:** Elliptic Curve Cryptography
FFT	**:** Fast Fourier Transform
FIFO	**:** First-In-First-Out
GPV	**:** Gentry–Peikert–Vaikuntanathan
GSO	**:** Gram-Schmidt Orthogonalization
HLS	**:** High-Level Synthesis
iFFT	**:** Inverse-FFT
iNTT	**:** Inverse-NTT
ISA	**:** Instruction Set Architecture
KEMs	**:** Key Encapsulation Mechanisms
LWE	**:** Learning With Errors
NIST	**:** National Institute of Standards and Technology
NTT	**:** Number Theoretic Transform
PQC	**:** Post-Quantum Cryptography
RISC-V	**:** Reduced Instruction Set Computer-V
RSA	**:** Rivest-Shamir-Adleman
SIS	**:** Short Integer Solution
SoC	**:** System-on-Chip
SVP	**:** Shortest Vector Problem
TRNG	**:** True Random Number Generator

LIST OF TABLES

Page

LIST OF FIGURES

HARDWARE IMPLEMENTATION OF THE POST-QUANTUM CRYPTOGRAPHY ALGORITHM FALCON

SUMMARY

The emergence of quantum computing poses a significant threat to classical cryptographic systems, as it undermines the hardness assumptions on which widely used public-key schemes such as RSA and ECC are based. In response to this evolving threat landscape, the National Institute of Standards and Technology (NIST) launched a multi-year, open international process in 2016 to identify and standardize post-quantum cryptographic (PQC) algorithms that are secure against quantum adversaries. This initiative has involved extensive evaluation of submissions based on their security, efficiency, and suitability for practical deployment.

Among the algorithms submitted and assessed throughout NIST's multi-round selection process, FALCON (Fast Fourier Lattice-based Compact Signatures over NTRU) emerged as one of the selected candidates for digital signature standardization. FALCON is a lattice-based digital signature scheme, and leverages several techniques and methods for efficient polynomial arithmetic, offering significant performance improvements over conventional lattice-based schemes. The novel Fast Fourier Sampling method, that is developed by the creators of FALCON, is a central component of this efficiency. This method is employed alongside various other algorithms, including the Fast Fourier Transform (FFT) and floating-point arithmetic, to compose the complete algorithm in three parts: key generation, signature generation and verification.

This study aims to analyze FALCON in detail and identify key components for hardware acceleration. The goal is to design accelerator cores to enhance the speed of the major algorithmic parts, and to form a System-on-Chip (SoC) that includes the accelerator cores, a RISC-V processor, and a special Direct Memory Access (DMA) module, to improve the overall efficiency of the signature generation and verification processes.

The initial phase involved a comprehensive analysis of the FALCON algorithm, utilizing reference documentation and the official C implementation. A C profiler was used to measure the execution time distribution across various sub-functions. This profiling revealed the computational structure of the algorithm, identified performance bottlenecks, and highlighted operations most suitable for acceleration.

To enable hardware-software integration, the reference C code was modified for compatibility with a RISC-V environment. And to establish a baseline, compiled codes for the signature generation and verification of FALCON was employed on a actual FPGA implementation of a RISC-V core, VexRiscv, and the clock cycles required by each major sub-function were measured.

Following this analysis, a design methodology was adopted that focuses on implementing selected components of the algorithm in their entirety. Rather than developing generalized cores for basic operations—intended for reuse across multiple sub-functions—this approach targets the direct hardware realization of specific, high-impact sub-functions as dedicated accelerator cores. This methodology significantly reduces the communication overhead between the software and hardware domains, which is often a bottleneck in hybrid implementations. Although this approach may incur higher resource consumption, it enables maximum performance by minimizing latency and maximizing data locality, thereby achieving the highest possible execution speed.

Results from both profiling and baseline implementation were used in the selection process of the parts that will be realized on accelerator cores, following the explained design methodology. The design process began with a detailed examination of the selected functions. Following the functional analysis, a dedicated hardware circuit for the selected sub-functions was developed.

In the signature generation process, the majority of operations involve floating-point arithmetic, as the polynomials being processed reside in the FFT domain, where their elements are represented as floating-point complex numbers. Consequently, the accelerator cores for signature generation were designed using configurable floating-point IP cores provided by Xilinx, complemented by data flow control components such as internal memory blocks and First-In First-Out (FIFO) buffers.

On the other hand, the most time consuming part of the algorithm was determined to be polynomial multiplication for the verification part, where the Number Theoretic Transform (NTT) method is used in the reference implementation. Therefore, accelerator cores that carry out NTT and Inverse-NTT operations were constructed for improving the overall performance of the part.

Each core was then individually implemented and verified using test vectors generated from the reference C code, covering both the signature generation and verification components. The maximum achievable clock frequency for each circuit was measured, and necessary optimizations and modifications were applied to obtain the highest-performing versions of the cores.

With all of the components ready, the SoC was formed using the designed accelerators connected to the special DMA, together with the VexRiscv processor and a block RAM for data and instructions. The software responsible for managing the algorithmic flow and controlling the accelerator cores was developed in the C programming language. Core control is handled through dedicated driver code, which provides the accelerators with appropriate configuration parameters and input data sourced from memory via the DMA. Similarly, the drivers manage the transfer of computed outputs from the cores back to designated memory locations.

The system then was implemented on a Zynq-7000 FPGA, after the designed accelerator cores were prepared as IPs to be used in a block design. Implemented system was evaluated for its execution time and resource usage. Performance was prioritized over other considerations initially. Nonetheless, the design serves as both a proof of concept and a viable FPGA-based implementation.

The results demonstrate that the proposed implementation achieves an approximate 375× and 70× speed-up for the signature generation and verification, respectively, over the baseline RISC-V implementation. And compared to the related work, results from the system are remarkable, making the design methodology a viable solution for implementing FALCON.

The current implementation represents a first iteration of the system design. Simulation and performance analysis have revealed opportunities for further optimization for a number of accelerators, including parallelization and the design of custom floating-point units to improve area efficiency. Future work will aim to refine the design to balance speed, area, and power consumption.

This study not only provided a deep understanding of FALCON, particularly the signature generation and verification, but also highlighted the critical points for the both parts. The proposed hardware accelerators demonstrates a substantial improvement in execution speed. While this work prioritizes speed, future research may focus on achieving a more balanced trade-off between performance and hardware resource utilization.

KUANTUM SONRASI KRİPTOGRAFİ ALGORİTMASI FALCON'UN DONANIM GERÇEKLEMESİ

ÖZET

Kuantum bilgisayarların ortaya çıkışı, klasik kriptografik sistemler için ciddi bir tehdit oluşturmaktadır. RSA ve ECC gibi yaygın açık anahtarlı şemaların dayandığı matematiksel problemlerin kuantum algoritmaları ile çözülebilmesi, bu sistemleri güvensiz hâle getirmektedir. Bu doğrultuda, NIST (National Institute of Standards and Technology), 2016 yılında kuantuma dayanıklı Kuantum Sonrası Kriptografi (PQC) algoritmalarının belirlenmesi ve standardizasyonu amacıyla uluslararası açık bir çağrı başlatmış ve çok aşamalı bir değerlendirme süreci yürütmüştür.

Bu süreçte dijital imzalama yapısı olarak seçilen algoritmalardan biri olan FALCON (Fast-Fourier Lattice-based Compact Signatures over NTRU), kafes tabanlı bir dijital imza algoritmasıdır. Algoritmanın güvenliği, LWE (Learning With Errors) ve SIS (Short Integer Solution) gibi kafes problemlerinin zorluğuna dayanır. FALCON, Gentry-Peikert-Vaikuntanathan (GPV) çerçevesini temel almakta, yani kafeslerden kısa vektörler örnekleyerek imza üretimini gerçekleştirmektedir. Bu süreçte "trapdoor" adı verilen özel matematiksel yapılar kullanılmakta, böylece imza üretiminde etkinlik ve güvenlik birlikte sağlanmaktadır.

FALCON'un verimliliğinin merkezinde Fast Fourier Sampling algoritması yer alır. Bu yöntem, Babai tarafından ortaya konan en yakın düzlem (nearest-plane) algoritmasının bir türevi olup, kafeslerdeki en yakın vektör problemini çözmeye yönelik tasarlanmıştır. Bu yöntem, klasik kafes tabanlı imza algoritmalarına kıyasla hem hız hem de imza boyutu açısından önemli avantajlar sunmaktadır.

Bu çalışmada, FALCON algoritmasının ayrıntılı biçimde analiz edilip donanım hızlandırıcılarla optimize edilebilecek kısımların belirlenmesi ve ardından özel bir System-on-Chip (SoC) mimarisi içinde RISC-V işlemcisi ve DMA (Direct Memory Access) birimiyle birlikte gerçekleştirmektir. Böylelikle, hem imza üretimi hem de doğrulama süreçlerinde önemli performans kazanımları sağlanması hedeflenmektedir.

İlk adım olarak FALCON'un yazarları tarafından yayınlanan referans C kodu incelenmiş, bir profilleme aracı kullanılarak alt fonksiyonların harcadıkları süreler ölçülmüş ve darboğazlar tespit edilmiştir. Bu analizler, algoritmanın işlem yapısını anlamaya, darboğazları belirlemeye ve donanım hızlandırmasına en uygun noktaları seçmeye imkan vermiştir. Elde edilen bulgular, özellikle kayan nokta işlemlerinin (floating-point operations) toplam sürenin büyük çoğunluğunu oluşturduğunu göstermiştir.

FPGA üzerinde gerçeklenen RISC-V çekirdeği üzerinde uyarlanan C kodları çalıştırılmış ve belirli büyük alt fonksiyonların aldığı süreler ölçülüp temel oluşturmak

üzere kaydedilmiştir. Bu sonuçlar, donanım hızlandırıcılarının sağlayacağı performans kazanımlarını kıyaslamak için temel bir karşılaştırma noktası oluşturmuştur.

Analizler sonucunda, belirli alt fonksiyonların bütün halinde donanıma taşınmasına karar verilmiştir. Bu yaklaşımda, çok sayıda alt fonksiyonda tekrar kullanılabilecek temel işlem çekirdekleri tasarlamak yerine, belirli fonksiyonların doğrudan hızlandırıcı olarak gerçeklenmesi hedeflenmiştir. Böylelikle yazılım ve donanım arasındaki iletişimden doğan ek yük azaltılmıştır. Her ne kadar bu yöntem daha yüksek donanım kaynak tüketimine yol açsa da en yüksek hızın elde edilmesini mümkün kılmıştır.

Profil sonuçları ve FPGA üzerinde yapılan temel ölçümler, hangi alt fonksiyonların donanım hızlandırıcıları olarak tasarlanacağını belirlemede kullanılmıştır. İmza üretimi tarafında FFT (Fast Fourier Transform) uzayındaki polinom işlemleri ve Fsat Foirer Sampling metodunu gerçeklendiği ffSampling fonksiyonu öne çıkarken, doğrulama tarafında en çok zaman alan işlem NTT (Number Theoretic Transform) ile gerçekleştirilen polinom çarpımı olmuştur. Bu fonksiyonlar için özel hızlandırıcı çekirdekler tasarlanması planlanmıştır.

İmza üretiminde kullanılan polinom işlemleri FFT alanında gerçekleştiğinden elemanlar karmaşık sayılar olarak temsil edilmektedir ve yoğun biçimde kayan nokta hesaplamaları gerektirmektedir. Bu nedenle, çekirdeklerin ana işlem birinlerini oluşturmak için Xilinx tarafından sağlanan yapılandırılabilir floating-point IP çekirdekleri kullanılmıştır. Belirli işlemler için programlanmış IP çekirdekleri doğru şekillerde birleştirilerek her fonksiyon ve alt fonksiyonları için işlem birimleri oluşturulmuştur. Bu işlem birimleri kontrolcü, dahili bellekler ve FIFO'lar (First-in-First-Out) gibi kontrol komponentleri ile birleştirilerek hızlandırıcı çekirdekler tasarlanmıştır.

Doğrulama tarafında ise NTT ve ters NTT çekirdekleri, FFT ve ters FFT'nin tam sayı modüler aritmetik uyarlamaları olarak tasarlanmıştır. Bu çekirdeklerde toplama, çıkarma ve çarpma işlemleri $\bmod q$ uzayında gerçekleştirilmektedir, bu nedenle modüler indirgeme işlemleri de devreye eklenmiştir.

Her bir çekirdek ayrı ayrı tasarlanmış, test vektörleriyle doğrulanmış ve maksimum saat frekansları ölçülmüştür. Çekirdekler ereken optimizasyonlar uygulanarak en yüksek performansı sağlayan hallerine getirilmiştir.

Tüm çekirdekler hazırlandıktan sonra, tasarlanan hızlandırıcılar özel DMA birimi aracılığıyla RISC-V işlemciye bağlanmış ve blok RAM ile desteklenerek bir SoC mimarisi oluşturulmuştur. Yazılım tarafı C diliyle geliştirilmiş, hızlandırıcı çekirdeklerin doğru konfigürasyon ve giriş veri akışıyla beslenmesi için sürücü kodları yazılmıştır. Çekirdeklerden alınan sonuçlar yine sürücüler aracılığıyla bellek alanlarına yönlendirilmiştir. Tasarlanan sistem, Zynq-7000 FPGA üzerinde test edilmiş ve kaynak tüketimi, alan ve hız açısından değerlendirilmiştir.

Elde edilen sistem referans RISC-V implementasyonuna kıyasla imza üretiminde yaklaşık 375 kat, doğrulama kısmında ise 70 kat hızlanma sağlamıştır. Literatürdeki benzer çalışmalarla kıyaslandığında da hız ve alan açısından kayda değer bir noktaya yerleşebilmektedir. Bu sonuçlar, önerilen tasarım metodolojisinin etkinliğini ve FALCON için güçlü bir çözüm sunduğunu ortaya koymaktadır.

Mevcut tasarımın ilk sürümü, daha ileri optimizasyonlara imkan tanımaktadır. Özellikle bazı alt fonksiyonların paralelleştirilmesi, özel kayan nokta devrelerinin tasarlanması ve alan verimliliğinin artırılması gelecek çalışmalarda hedeflenmektedir. Böylelikle, hız, alan ve güç tüketimi arasında daha dengeli bir tasarım elde edilebilecektir.

Çalışma sonucunda, PQC algoritmaları temellerini ve özellikle FALCON algoritması derinlemesine öğrenilmiştir. Çeşitli tasarım metodolojisi ve teknikleri incelenmiş ve uygulanmış, karşılaşılan problemlerin çözülmesiyle bu alanda da kazanımlar sağlanmıştır. Tasarlanan hızlandırıcı çekirdekler, algoritmanın performansını büyük ölçüde artırmıştır. Gelecekte yapılacak iyileştirmeler ile FALCON'un donanım tabanlı uygulanabilirliği daha da güçlenecek ve PQC standartlarının pratik uygulamaları için önemli katkılar sunacaktır.

1. INTRODUCTION

In the current digital era, public-key cryptographic algorithms underpin the security of communication, authentication, and data integrity across nearly all sectors-from banking and government infrastructure to personal digital devices. However, with the theoretical and practical development of quantum computing, the foundations of these classical schemes are under existential threat. Quantum algorithms such as one described by Shor (1997) can efficiently solve the integer factorization and discrete logarithm problems-the mathematical basis for the algorithms Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC), respectively. If a sufficiently powerful quantum computer is realized, most widely used public-key cryptographic systems will become obsolete overnight.

This emerging threat necessitates the development and implementation of post-quantum cryptographic algorithms. One such algorithm, FALCON (Fast Fourier Lattice-based Compact Signatures over NTRU), has been selected by the National Institute of Standards and Technology (NIST) (2023) for standardization. FALCON is a lattice-based digital signature scheme that follows the Gentry–Peikert–Vaikuntanathan (GPV) framework, which generates signatures by sampling short vectors from a lattice using a trapdoor mechanism (Gentry, Peikert, and Vaikuntanathan, 2008).

This thesis study aims to construct a hardware implementation of FALCON. Through detailed analysis and inspection, key aspects of FALCON that can be investigated for efficient implementation are deducted to be used in the design process. Construction of accelerators for the selected parts from the algorithms constitutes the design processes. And finally, a System-on-Chip(SoC) that includes the accelerator cores alongside a processor that effectively implements FALCON is proposed.

In the next chapter, background information is included, followed by the literature review of similar works that involve implementation of FALCON. Here, different studies across different platforms is featured for a wider point of view. After that,

detailed analysis of FALCON that enables deeper understanding of the algorithm is given. Next, design process is presented, including methodology, design decisions and details about the circuitry. And finally, results from the FPGA implementation is provided, and the paper is concluded.

2. BACKGROUND INFORMATION

2.1 Post-Quantum Cryptography

Post-Quantum Cryptography (PQC) refers to cryptographic algorithms that are designed to be secure against both classical and quantum adversaries. PQC is intended for classical devices, but is constructed upon hard problems proven to remain secure even with access to quantum computational power. Among the promising mathematical foundations of PQC are code-based, multivariate, hash-based, isogeny-based, and lattice-based cryptographic schemes (Chen, Jordan, Liu, Moody, Peralta, Perlner, and Smith-Tone, 2016). The urgency surrounding PQC led the NIST to initiate an international effort in 2016 to evaluate and standardize post-quantum cryptographic algorithms (National Institute of Standards and Technology (NIST), 2023). The multi-round competition attracted a wide range of proposals, and in its third round, a select group of finalist and alternate candidates were identified. FALCON, based on lattice problems, was chosen as one of the primary digital signature schemes for standardization.

2.2 Lattice-Based Cryptography

Lattice-based cryptography is a class of cryptographic constructions that leverage the complexity of lattice problems. A lattice in R^n is a discrete set of points generated as all integer combinations of linearly independent vectors. This set of vectors is referred to as a basis of the lattice, and the structure can be visualized as a regular, infinitely extending grid in multiple dimensions. A 2-dimensional lattice and two possible bases can be seen in the Figure 2.1.

What makes lattices appealing for cryptography is the computational difficulty of certain problems defined over them, such as Shortest Vector Problem (SVP), Closest Vector Problem (CVP) and Learning With Errors (LWE). Unlike factoring or discrete logarithm problems, lattice problems admit no known sub-exponential

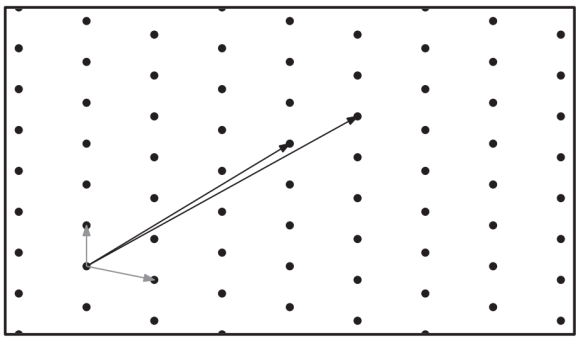

Figure 2.1: 2-Dimentional Lattice

solution on either classical or quantum computers (Micciancio and Regev, 2009). This durability makes them an attractive basis for constructing digital signatures, Key Encapsulation Mechanisms (KEMs), and more. Another appealing property of lattice-based cryptography is efficiency. Lattice operations primarily consist of matrix and vector arithmetic over rings and fields, often allowing for high-speed and parallelizable implementations.

Several algorithms submitted to the NIST PQC standardization process are based on lattice problems. Notably, among the four algorithms selected by NIST for standardization, three are lattice-based. Kyber is a KEM, constructed upon the Module Learning With Errors (Module-LWE) problem, an efficient variant of the standard LWE problem. Dilithium, another algorithm selected for digital signatures, is based on the Short Integer Solution (SIS) and LWE problems. FALCON is the third lattice-based algorithm selected by NIST and is founded on the SVP over structured lattices.

A key motivation behind FALCON's selection was its ability to offer compact signatures and efficient implementation while maintaining rigorous security proofs grounded in worst-case lattice assumptions. Its strong theoretical underpinnings and practical applicability make it a focal point for future-proof cryptographic systems (Fouque, Hoffstein, Kirchner, Lindner, Nguyen, Prest, & Whyte, 2018).

2.3 FALCON

FALCON is a digital signature algorithm, built upon the GPV framework, with incorporation of the mathematical properties of NTRU lattices and the introduction

of a novel Fast Fourier Sampling technique to optimize computational efficiency. Its key innovation lies in this advanced sampling method, which serves as a trapdoor mechanism. FALCON is designed to operate in two modes: FALCON-512 and FALCON-1024, corresponding to NIST security levels I and V, respectively. The numerical values 512 and 1024 denote the degrees of the lattice polynomials utilized within the algorithm; as expected, higher-degree polynomials provide enhanced security (Fouque, Hoffstein, Kirchner, Lindner, Nguyen, Prest, & Whyte, 2018).

The algorithm is structured into three distinct parts: key generation, signature generation, and verification. The aforementioned techniques are integrated into one or more of these parts to realize the complete FALCON algorithm. A more detailed examination of these components and methods will be presented in the following sections.

FALCON's advantage to its competitors is its compact signature and key-pair sizes, making it suitable for systems where memory is constrained. Moreover, its verification performance stands out, fastest among the selected digital signature algorithms. However, usage of Fast Fourier Sampling methods brings the Fourier Transform into the algorithm, which makes the operands floating-point numbers. This might limit its applicability, because of floating-point arithmetic being more complex and demanding for implementing. A comparison with selected algorithms and a popular classical cryptography algorithm RSA is given in the Figure 2.2, retrieved from PQShield (2025).

2.3.1 The GPV framework

The GPV framework is a lattice-based signature scheme, which uses mathematical properties of lattices proven to be secure against quantum computing, introduced by Gentry, Peikert, and Vaikuntanathan (2008). In the framework, the public key is a full-rank *nxm* matrix $A \in \mathbb{Z}_q^{n \times m}$ $(with\ m > n)$, with integer elements in the interval $[0,\ q)$. This public key is used as a basis to generate the lattice $\mathscr{L}$. Similarly, the secret key is a matrix $B \in \mathbb{Z}_q^{m \times m}$ that generates the counterpart lattice $\mathscr{L}_q^{\perp}$. These lattices are constructed to be orthogonal to each other, ensured by selecting the base matrices to be orthogonal as in (2.1):

Scheme	Parameterset	NIST level	Pk bytes	Sig bytes	pk+sig	Sign (cycles)	Verify (cycles)
Falcon	1024	5	1,793	1,280	3,073	2,053,080	160,596
Falcon	512	1	897	666	1,563	1,009,764	81,036
ML-DSA	ML-DSA-87	5	2,592	4,627	7,219	642,192	279,936
ML-DSA	ML-DSA-65	3	1,952	3,309	5,261	529,106	179,424
ML-DSA	ML-DSA-44	2	1,312	2,420	3,732	333,013	118,412
RSA	2048	Pre-Q	272	256	528	27,000,000	45,000
SLH-DSA	SHAKE-192s	3	48	16,224	16,272	8,091,419,556	6,465,506
SLH-DSA	SHAKE-256s	5	64	29,792	29,856	7,085,272,100	10,216,560
SLH-DSA	SHAKE-128s	1	32	7,856	7,888	4,682,570,992	4,764,084
SLH-DSA	SHAKE-256f	5	64	49,856	49,920	763,942,250	19,886,032
SLH-DSA	SHAKE-192f	3	48	35,664	35,712	386,861,992	19,876,926
SLH-DSA	SHAKE-128f	1	32	17,088	17,120	239,793,806	12,909,924

Figure 2.2: Comparioson of FALCON with Other Algorithms

$$B \cdot A^T \equiv 0 \mod q \tag{2.1}$$

Given a message m, a valid signature of this message $s \in \mathbb{Z}_q^m$ must satisfy (2.2):

$$A \cdot s \equiv H(m) \mod q, \quad \text{and} \quad \|s\| < \beta \tag{2.2}$$

where $H(m)$ is the hash of the message, $\|s\|$ is the norm of the signature, and β is the predefined maximum allowed norm of a signature.

Finding such s with small enough norm is computationally infeasible, while a verifier can easily compute a vector's norm with efficient algorithms corresponding to the SVP problem in lattices. However, knowledge of the secret matrix B allows the signer to efficiently compute a short vector to attain a valid signature. This is achieved by:

1. Solving $A \cdot s' = H(m) \mod q$ for some initial s', which doesn't need to have a small norm
2. Sampling a vector v in the lattice $\mathscr{L}_q^{\perp}$ close to the vector s'
3. Computing the final signature as $s = s' - v$

The signature $s = s' - v$ satisfies the GPV conditions, since $A \cdot s = A \cdot s' - A \cdot v = H(m) - 0$. This is ensured by v being orthogonal to the rows of A, resulting in $A \cdot v = 0$.

2.3.2 NTRU lattices

NTRU is a lattice-based public key cryptosystem introduced by Hoffstein, Pipher, and Silverman (1998), which operates over polynomial rings. A ring, donated by R is an algebraic structure that allows addition and multiplication, and a polynomial ring includes polynomials with coefficients in the the corresponding ring, supporting operations like polynomial addition and multiplication.

In FALCON, public and private keys are constructed from NTRU polynomials, distinguishes it from other GPV-based schemes and makes arithmetic operations both fast and compact.

NTRU lattices are defined over the ring defined in (2.3):

$$R_q = \mathbb{Z}_q[X]/(X^n + 1) \tag{2.3}$$

Here, the $\mathbb{Z}_q[X]$ denotes a ring with integer elements in $\mod q$ domain, and division by $(X^n + 1)$ determines the maximum degree of the polynomials to be n.

The key generation involves preparing polynomials $f, g, F, G \in R_q$, as in (2.4):

$$fG - gF = q \mod X^n + 1 \tag{2.4}$$

The public key is then computed as (2.5):

$$h = gf^{-1} \mod q \tag{2.5}$$

These polynomials are then used to construct public and secret keys, as matrices described in (2.6) and (2.7):

$$PublicKey : A = \begin{pmatrix} 1 & h \end{pmatrix} \tag{2.6}$$

$$SecretKey : B = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix} \tag{2.7}$$

These matrices satisfy the GPV rules in the equation (2.8):

$$\begin{aligned} B \cdot A^T &= \begin{pmatrix} g - hf \\ G - hF \end{pmatrix} = \begin{pmatrix} g - (gf^{-1})f \\ G - (gf^{-1})F \end{pmatrix} = \begin{pmatrix} g - g \\ f^{-1}f(G - (gf^{-1})F) \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ f^{-1}(fG - gF) \end{pmatrix} = \begin{pmatrix} 0 \\ f^{-1}q \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \mod q \end{aligned} \tag{2.8}$$

The equalities $h = gf^{-1}$ and $fG - gF = q$ are used for deriving this result.

Another important property of NTRU polynomials is that computation f^{-1} and g from the public key h is being computationally infeasible. This property ensures that the public key does not reveal any information about the corresponding secret key (Richter, Bertram, Seidensticker, & Tschache, 2022). Due to these suitable properties, NTRU polynomials are used alongside the GPV framework as bases for dual lattices of FALCON.

2.3.3 Fast Fourier Sampling

GPV framework's second step relies on a method for sampling a short vector, which must be both secure and efficient. Typically, nearest plane algorithms can be used to generate close lattice points to a target point.

FALCON uses a highly efficient fast Fourier sampling developed by Ducas and Prest (2015), to generate the short vector v. In this method, Gram-Schmidt Orthogonalization (GSO) process, which is a critical step in nearest plane methods is optimized. By exploiting the structure of circulant matrices and applying Fast Fourier Transform (FFT) technique, the complexity is reduced from quadratic to quasi-linear time, specifically $\theta(d\ logd)$ for a matrix of dimension d. This improvement significantly enhances the overall efficiency. Fast Fourier sampling method is adapted as the sampler of GPV framework, and together with NTRU lattices constitutes key components of the algorithm.

2.3.4 FALCON's key generation

The key generation process of FALCON involves the construction of NTRU polynomials f, g, F and G, each composed of small-valued coefficients. This procedure begins with the sampling of f and g, followed by the computation of F and G. Subsequently, the secret and public keys are derived in accordance with the GPV framework, described in (2.6) and (2.7).

The secret key is retained by the key generator and is utilized for signing messages. Conversely, the public key is distributed to enable third parties to verify messages that have been signed by the corresponding signer.

2.3.5 FALCON's signature generation

In the signature generation process, the message is first concatenated with a randomly generated 40-byte salt r. This combined input is then processed by a function employing the SHAKE256 hash algorithm to produce the output c, which is subsequently used in the formation of s' within the GPV framework.

$$s' = (c, 0)^T \tag{2.9}$$

After that, using the private key and fast Fourier sampling method, a close vector $v \in \mathscr{L}_q^{\perp}$ to s' is computed. The final signature s is formed as in (2.10):

$$s = s' - v = (s_1, s_2)^T = (c - v_1, -v_2)^T \tag{2.10}$$

The signature is only accepted if its norm is within a predefined bound. If the signature is out-of-bounds, a new one is generated using a different r until a valid signature is acquired. The signature component s_2 is transmitted alongside the salt r as the signature.

2.3.6 FALCON's verification

Verification involves checking if the signature maps correctly under the public key and that the norm is within bounds. First, the c is calculated as in the signature generation using the message and salt r. The equivalency (2.11) of GPV framework

$$A \cdot s = \begin{pmatrix} 1 & h \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} = s_1 + h\, s_2 = c \tag{2.11}$$

is used to reconstruct the missing signature component as (2.12):

$$s_1 = c - s_2 \cdot h \tag{2.12}$$

Then the norm is checked as in (2.13):

$$\|(s_1, s_2)^T\| < \beta \tag{2.13}$$

A norm smaller than β can be achieved only if the other party transmits a valid s_2, enabling the correct reconstruction of s_1 using the public key h. If the computed norm exceeds the allowed bound, the signature is deemed invalid and subsequently rejected.

FALCON's verification process is computationally lightweight and highly efficient, representing one of the algorithm's key strengths.

2.4 RISC-V

The Reduced Instruction Set Computer-V (RISC-V) is an open-standard Instruction Set Architecture (ISA) that originated from a research project at the University of California, Berkeley. Initiated by Professor Krste Asanović and his graduate students Yunsup Lee and Andrew Waterman, the project aimed to create a clean, extensible,

and efficient ISA for both academic research and industrial application (RISC-V International, 2021). What began as a five-year academic effort has since evolved into a global open-source movement. RISC-V's openness distinguishes it from most of the ISAs, allowing unrestricted access for developers, educators, and hardware designers to use and modify the architecture freely.

RISC-V defines a base set of instructions that all compliant processors must support, while also offering modular extensions for more advanced functionality such as floating-point arithmetic, atomic operations, and vector processing. The ISA does not mandate implementation strategies or specific subsets, granting designers the flexibility to tailor systems to a wide variety of performance, power, and area requirements. This modular and implementation-agnostic nature makes RISC-V particularly suitable for diverse domains—from embedded systems to high-performance computing. Its well-structured and extensible design has made it a prominent choice for open hardware initiatives and next-generation processor development.

2.5 Special DMA: ODBEM

Direct Memory Address (DMA) modules are important components on SoC designs providing and controlling memory-processor-peripheral connections. ODBEM is a scalable and adaptable DMA, eliminating the need for the processor to have a dedicated DMA interface (Esen, 2023). Instead, communication between the processor and the DMA module occurs through predefined memory addresses, ensuring compatibility with various processor architectures, including open-source RISC-V cores. This design choice not only reduces the processor's workload but also enhances the overall system performance by allowing concurrent data transfers and processing.

ODBEM utilizes First-In-First-Out (FIFO) buffers for seamless data flow between modules. These buffers hold inputted data up to their specified capacity, and output the first inputted data when commanded from their output port. By integrating the ODBEM into an SoC design data throughput and processing efficiency can be improved, making it particularly beneficial for applications requiring high-speed data

transfers between accelerator cores and processor. Usage of ODBEM is detailed in later chapters.

3. LITERATURE REVIEW

Since the introduction of FALCON, numerous implementations have been proposed to accelerate or simply enable its practical realization. These implementations vary in nature: some are purely software-based and target different computing architectures, while others are entirely hardware-oriented, focusing on the realization of specific components or sub-functions of FALCON across various platforms. Additionally, several hardware/software co-design approaches have emerged, integrating both hardware and software elements to implement FALCON. Given that the objective of this study is to develop a co-design implementation of FALCON, this section will review relevant hardware and hardware/software implementations, examining their design methodologies, key considerations, and performance outcomes.

FALCON's first hardware implementation effort was carried out by Schmid, Amiet, Wendler, Zbinden, and Wei (2023), in which signature and key generation algorithms are realized using High-Level Synthesis (HLS) techniques. Reference C code, which is made available by the creator of the FALCON, is modified to be compatible with HLS, and used as a basis. Authors point out the design challenges, like floating-point arithmetic and mathematical complexity of algorithm. This study is one of the few that focuses on key generation alongside signature generation. But lacks in analysis of the algorithm and hardware optimizations, because of the chosen design methodology, and also due to being an early effort.

Karabulut and Aysu (2023) address one of the most computationally intensive components of the FALCON digital signature scheme, discrete Gaussian sampling in the fast Fourier sampling part, by proposing a hardware-software co-design tailored for efficient and secure implementation. The authors propose an architecture that leverages a tightly integrated hardware accelerator that executes the core operations like floating-point arithmetic and Gaussian-sampling, while a software control sets variable parameters. Their results show that the sampling is accelerated up to

9.83× and all of the FALCON signature generation scheme by 2.7×, compared to the reference software implementation. This design maintains compatibility with constrained embedded systems. The design is a modular one that can serve as a building block for full FALCON hardware accelerators, but only realizes parts of it at this state.

Alsuhli, Saleh, Al-Qutayri, Mohammad, and Stouraitis (2024) propose a compact processor tailored for FFT and inverse-FFT (iFFT) operations in the FALCON, specifically aimed for low-power systems. Methods like twiddle factor compression, conflict-free scheduling and many others are used alongside a memory-based structure, where inputs and outputs of FFT/iFFT computing unit read from and written from memories. Overall performance of FALCON is increased 3.8× compared to an existing pure-software implementation. In this study, specific components of FALCON's signature generation process, FFT and iFFT, were selected, analyzed, and implemented independently. While overall performance could potentially be enhanced by identifying and accelerating a critical bottleneck within the algorithm, this work does not constitute a full implementation of the FALCON scheme.

Ouyang, Zhu, Zhu, Yang, Zhang, Wang, Tao, Zhu, Wei, and Liu (2025) introduce FALCONSign, a high-performance, configurable crypto-processor designed to accelerate FALCON signature generation on FPGA/ASIC platforms. The architecture includes parallel processing units, like Fast-Fourier Sampling and floating-point units. FALCON's sign algorithm is split into small tasks that will be queued by the controller and carried out by the correct processing unit. Outputs of a processor unit is used by the next one to put together the whole sign algorithm. The FPGA implementation achieves a throughput improvement of approximately 5.1× over state-of-the-art designs, with significant reductions in area-time product metrics, demonstrating the potential for efficient fully hardware implementations. This study is one of the most complete implementation of FALCON's signature generation, but key generation and verification isn't worked on. Moreover, integration with an existing system could be challenging, due to possible compatibility problems.

In their study, Lee, Youn, Nam, Jung, Cho, Na, Park, Jeon, Kang, Oh, and Paek (2024) propose an efficient hardware/software co-design for implementing the FALCON digital signature algorithm on low-end embedded systems. The authors address the challenges of limited computational resources and memory constraints by optimizing key modules such as the FFT, Gaussian sampling, and floating-point operations through hardware acceleration, while retaining flexibility via software control. They implemented their design with Samsung's 28 nm process at 300 MHz of clock frequency, and achieved 3.58× improvement over an existing hardware implementation. This study highlights the potential of software/hardware co-design approaches, completely realizes FALCON's signature generation, but lacks in modularity and configurability.

In this thesis work, the objective was to construct a modular and flexible system architecture incorporating a RISC-V processor, a dedicated DMA controller, and specialized accelerator cores for various sub-functions of the FALCON algorithm. The two operation modes of FALCON are implemented in a parameterizable fashion, allowing for the realization of either the 512 version alone or both modes, differently from some of the existing work which only supports FALCON-512. Depending on specific resource constraints such as chip area and power consumption, certain cores may be omitted, with their corresponding functionality instead executed via software routines. Conversely, a full-featured implementation can integrate all accelerator cores to maximize performance, thereby promoting design flexibility and scalability. This modularity is unique to proposed design, emerging as one of the most flexible in the literature.

Furthermore, the signature generation and verification are implemented in a single system supporting both 512 and 1024 modes for both, with their respective accelerator cores interfaced to the processor through a unified architecture. This design approach enables efficient and seamless execution of both processes without requiring separate communication interfaces and components, distinct from most of the current studies where mostly the key generation or part of it is focused on. Also, for future work, accelerator cores for the other PQC algorithms can be designed and integrate to the system with ease. Mentioned points form the key ideas and motivation of this work.

4. ANALYSIS OF FALCON

In this chapter, the FALCON's signature generation and verification algorithms will be examined in detail, initially focusing on explaining its core functions using the reference documentation. The authors of FALCON also published a reference C code, which can be utilized for tasks like simulating FALCON, inspecting specific functions, and generating test vectors. Profiling tools will be used on this implementation to determine the execution time distribution across individual functions. This analysis will help identify computational bottlenecks and highlight potential candidates for hardware acceleration. Subsequently, the reference C code will be adapted for compatibility with the RISC-V architecture. This implementation will then be analyzed, forming the foundation for the proposed design in which accelerator cores will be integrated to undertake and optimize the execution of critical operations.

4.1 Reference Documentation

The signature generation and verification processes of the FALCON algorithm will be examined using the pseudocode provided in the reference documentation. The primary functions, `Sign` and `Verify`, which respectively implement the signature generation and verification procedures, will first be briefly described. Subsequently, their corresponding sub-functions will be analyzed, and the specific tasks performed by each will be identified. This initial examination serves to clarify the algorithmic structure and control flow of the primary functions, thereby facilitating a smoother transition to the analysis of the reference C implementation, which follows a similar naming convention for its functions. First, for the `Sign` algorithm, pseudocode is presented in the Figure 4.1.

Algorithm 10 Sign (m, sk, $\lfloor \beta^2 \rfloor$)

Require: A message m, a secret key sk, a bound $\lfloor \beta^2 \rfloor$
Ensure: A signature sig of m

1: $\mathsf{r} \leftarrow \{0,1\}^{320}$ uniformly
2: $c \leftarrow \mathsf{HashToPoint}(\mathsf{r} \| \mathsf{m}, q, n)$
3: $\mathbf{t} \leftarrow \left(-\frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(f)\right)$ ▷ $\mathbf{t} = (\mathsf{FFT}(c), \mathsf{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$
4: do
5: do
6: $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$
7: $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$ ▷ At this point, s follows a Gaussian distribution: $\mathbf{s} \sim D_{(c,0)+\Lambda(\mathbf{B}),\sigma,0}$
8: while $\|\mathbf{s}\|^2 > \lfloor \beta^2 \rfloor$ ▷ Since s is in FFT representation, one may use (3.8) to compute $\|\mathbf{s}\|^2$
9: $(s_1, s_2) \leftarrow \mathsf{invFFT}(\mathbf{s})$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
10: $\mathsf{s} \leftarrow \mathsf{Compress}(s_2, 8 \cdot \mathsf{sbytelen} - 328)$ ▷ Remove 1 byte for the header, and 40 bytes for r
11: while $(\mathsf{s} = \perp)$
12: return sig $= (\mathsf{r}, \mathsf{s})$

Figure 4.1: `Sign` Pseudocode

The `Sign` proceeds through the following steps:

- A 40-byte random number r is generated to be combined with the message to be signed.
- The number r is concatenated with the message m and given as the input to the `HashToPoint` function, which generates the polynomial c.
- `FFT` function is used on the polynomial c and the secret key components f and F, and a vector t in the FFT domain is calculated.
- Within a loop between lines 5 and 8, the vector t and the secret key component T are given as input to the `ffSampling` function. The output is the vector z
- In the line 7, the signature vector s is generated using the vectors t, z, and matrix $\hat{B}$, which is FFT of the matrix:

$$B = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix} \tag{4.1}$$

- In the line 8, the norm of the vector s is calculated, and it is checked whether the square of this value is smaller than the bound β^2. The loop continues until an s with a sufficiently small norm is produced.

- If a generated signature passes all of the checks, its iFFT is taken with the function `invFFt`, and finally, the compressed signature is combined with the value r to form the final signature.

It is observed that the function `Sign` has a number of main sub-functions. After preliminary operations that includes `FFT`, the function `ffSampling`-which is the realization of the aforementioned Fast Fourier Sampling algorithm-is called in a loop until a valid signature is generated. After that, `invFFT` is called to bring the signature back to integer domain.

With the brief examination of the signature generation done, similar inspection is done on the verification. Pseudocode for `Verify` is in the Figure 4.2

Algorithm 16 Verify (m, sig, pk, $\lfloor\beta^2\rfloor$)

Require: A message m, a signature sig $=$ (r, s), a public key pk $= h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor\beta^2\rfloor$
Ensure: Accept or reject

1: $c \leftarrow$ HashToPoint(r$\|$m, q, n)
2: $s_2 \leftarrow$ Decompress(s, $8 \cdot$ sbytelen $- 328$)
3: if $(s_2 = \perp)$ then
4: | reject ▷ Reject invalid encodings
5: $s_1 \leftarrow c - s_2 h \bmod q$ ▷ s_1 should be normalized between $\lceil -\frac{q}{2} \rceil$ and $\lfloor \frac{q}{2} \rfloor$
6: if $\|(s_1, s_2)\|^2 \leq \lfloor\beta^2\rfloor$ then
7: | accept
8: else
9: | reject ▷ Reject signatures that are too long

Figure 4.2: `Verify` PseudoCode

Algorithmic steps of the `Verify` are:

- Similarly to the `Sign`, generation of vector c with the `HashtoPoint` is carried out.
- Signature is decompressed, and the signature component s_2 is obtained. If that step fails, signature is rejected.
- The other signature component s_1 is calculated with polynomial addition and multiplication in mod q domain.
- Norm of the components are checked against β^2, again similarly to `Sign`, and if norm is smaller the signature is accepted as valid, rejected if the norm is not small enough.

As it can be seen, `Verify` is much simpler, and has similarities to the `Sign` that can be realized likewise.

4.2 Reference Implementation

Reference C codes are packaged as a multi-file structure, source codes that defines functions are contained in files with proper naming, and main functions are defined together in a separate file. Files for testing are also included. Test files are joint while source codes are divided for 512 and 1024 variants.

FALCON is written in three major functions, `crypto_keypair`, `crypto_sign` and `crypto_open` for key generation, signature generation and verification, respectively. Sub-functions that carry out various operations like basic floating-point addition and multiplication, or more complete ones like FFT and sampling are called in those functions to complete required tasks. In testing files, random messages are generated to be signed and verified by corresponding function, using generated public and private keys. Similar setting for profiling was built using provided C codes and a newly written main function that only calls `crypto_sign` or `crypto_open` with correct inputs, to analyze each one independently.

Profiling process includes compiling of the codes with proper configuration, and then running it with correct commands to generate a report. Commands used for generating an executable file for signature generation of FALCON-512 can be seen below.

```
> gcc -W -Wall -O2 -pg sign.c -/build/kat512int
> gprof ./build/kat512int gmon.out > sign_500.txt
> gprof ./build/kat512int gmon.out
| gprof2dot
| dot -Tpng -o callgraph_sign_5000.png
```

That report includes several result like calling hierarchy of functions, number of calls and execution time percentage for each function. To ensure greater accuracy, profiling was conducted multiple times, with the signature generation and verification function invoked number of times during each profiling session. Profiling result of signature

generation and verification when they are called 5000 times in succession can be seen in the Figure 4.3 and Figure 4.4.

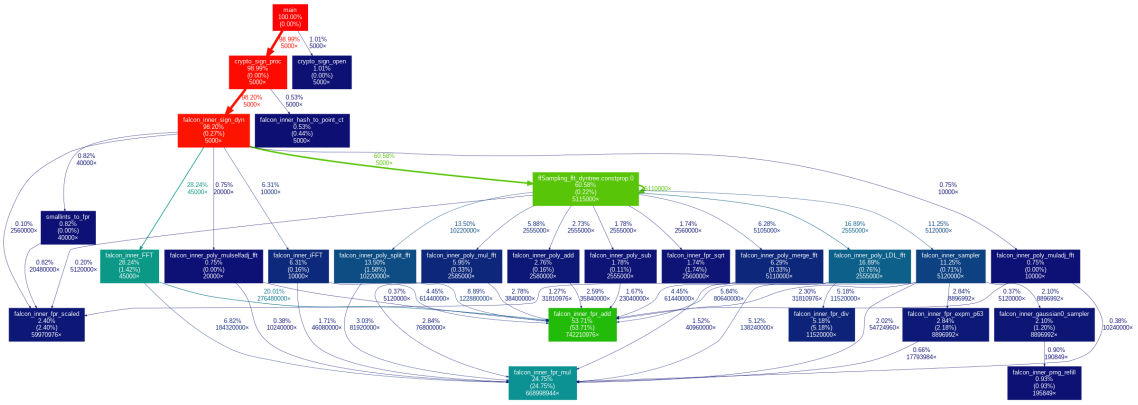

Figure 4.3: Callgraph of Signature Generation and Verification

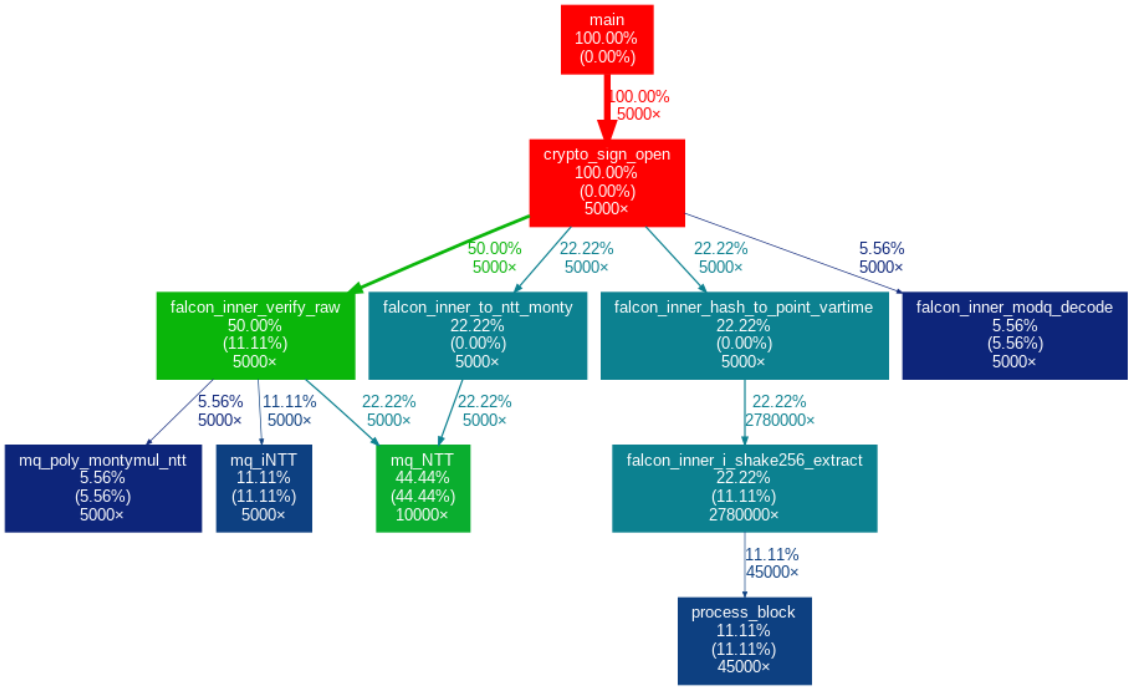

Figure 4.4: Callgraph of Signature Verification

As illustrated in the figures, signature generation is notably more time-consuming and computationally complex compared to verification. The frequency and duration of function calls highlight the significant role of floating-point operations—such as multiplication and addition—in the signature generation process, as these operations are invoked extensively. Key functions contributing to the overall execution time include `ffSampling`, `FFT`, and `iFFT`, which correspond to the Fast Fourier

Sampling, FFT, and iFFT algorithms, respectively. These functions collectively account for the majority of the computational load during signature generation.

On the other hand, for the verification process, the `NTT` and `iNTT` functions are most prominent. These functions implement the Number Theoretic Transform (NTT) and its inverse (iNTT), which are specialized forms of FFT and iFFT that operate over integer rings. Their efficient integer-based computation makes the verification process comparatively lightweight.

Our results are coherent with a different study that also analyzed FALCON's reference implementation, and deducted execution time consumption for sub-functions (Howe & Westerbaan, 2022). Their study covers FALCON's each mode with different compilation styles and modifications on several platforms, but mentioned major functions again shows up on their findings.

4.3 RISC-V Implementation

For compatibility with RISC-V compiler, structure of the reference C implementation is slightly modified. This modifications are made on the parts that compiler gave errors when codes are compiled without alterations, mostly on the data allocation and usage of memory movement functions like `memcpy`. With corrected codes, compiler generates an output file with `.elf` extension that can be used in several platforms to simulate the compiled program, or used to program an actual processor.

Reference C codes are then adjusted to the testing environment of an actual 32-bit RISC-V core, VexRiscv, that will also be used on the implementation of FALCON (SpinalHDL, 2025). For testing FALCON with VexRiscv, `.elf` output of the modified reference codes is used to generate instruction memory content. Data, instruction and stack memory spaces are set accordingly to be compatible with the processor. A block design that includes VexRiscv, ODBEM, a binary counter and a block memory that contains both instructions and data is formed for this experiment and implemented on the Zynq-7000 series FPGA, that is employed by Procenne on its devices on the market (AMD, n.d.).

Using the counter, cycle count of signature generation and some of the main functions of them is measured, and listed in the Table 4.1. As it can be seen the total execution times, signature generation is much more time consuming compared to the verification. Most demanding functions from both are also listed below the part.

Table 4.1: Execution times of FALCON's functions.

Algorithm	Execution Time (ms)
Sign	748.3
→ ffSampling	418.1
→ FFT	23.5
→ iFFT	24.2
Verify	12.9
→ NTT	1.03
→ iNTT	1.23

Analysis results from different methods and experiments showed that signature generation is much more complex and time-draining, and the `ffSampling` is the function with the highest execution time percentage. Importance of floating-point operations in the FALCON's overall performance is also highlighted. However these possible focus points exist, there are several major and minor functions for both the signature generation and verification that can be studied on and implemented on hardware to achieve the highest speed. With those points in mind, next step is to decide on design methodology and choosing parts from the algorithm to be accelerated.

5. DESIGN AND IMPLEMENTATION

5.1 Design Methodology

The system that FALCON will be implemented on is an SoC that consists of a DMA and a processor, namely ODBEM and 32-bit VexRiscv core. Moreover, a block memory that acts as both data and instruction memory for the processor while also providing the information transmission between the ODBEM and VexRiscv through reserved memory addresses is featured. Processor can run any compiled RISC-V program, and peripherals and additional modules can be controlled using driver software through ODBEM. This system framework, visualized in Figure 5.1, enables partial implementation of an algorithm in software, while selected components can be executed on dedicated hardware circuits under software control.

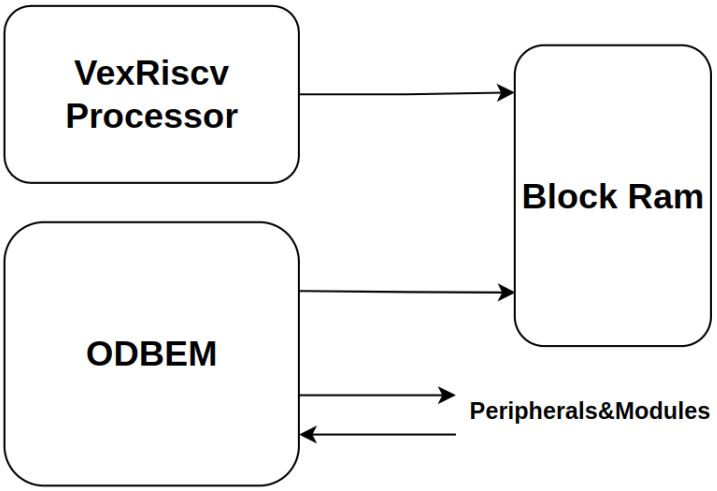

Figure 5.1: A system with ODBEM, RISC-V Processor, and Block Memory

For this study, an inclusive approach will be conducted. Instead of realizing basic functions-like floating-point arithmetic or polynomial operations-in hardware and then controlling the execution flow from software, circuits of main functions will be designed in their entirety in hardware, incorporating control logic and required memories. Hereby, required input data and configuration information will be sent

to the accelerator cores, required computations will be carried out by internal control and computation logic, then the output will be sent to the software side to be sent to the next core. Software will also be tasked with simple jobs, as communication between hardware and software also brings an overhead. This approach minimalizes the communication between hardware and software side, reducing the mentioned overhead. However, realizing major functions in hardware requires more complex and large accelerator designs, resulting in higher area usage and brings more challenges in design process. This more complex implementation approach was deliberately chosen despite the associated challenges, with performance improvements prioritized over simplicity and compactness. Also, all of the accelerators cores will be designed to realize both of the FALCON's modes, 512 and 1024, for a even more complete implementation.

In the FALCON, IEEE-754 standard is used for representing floating-point numbers and floating-point operations, using 64 bits precision (IEEE Computer Society, 2019). Therefore, floating-point units must be coherent with this standard. Additionally, a variety of arithmetic operations are required, ranging from basic addition and subtraction to more complex operations such as inversion and square root. Furthermore, interconnection between units responsible for different operations is essential to enable the construction of complex number operations and, ultimately, to facilitate efficient polynomial computations. Considering this points, it is decided to employ Xilinx's Floating-Point IPs due to their configurability for a variety of operations and between performance and area, stream-based input and output ports that also enables building of pipelined structures, and finally convenient utilization on the targeted FPGA platform. Ab unconfigured IP is given in the Figure 5.2

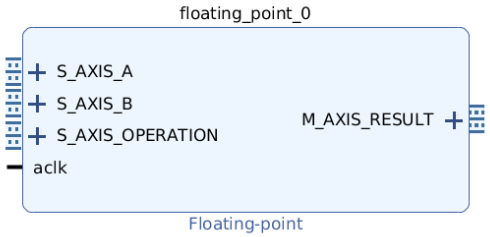

Figure 5.2: Xilinx's Floating-Point IP, Not Configured

In the next sub-sections, design process of the accelerator cores will be presented. First, a sub-section on selection of parts from signature generation and verification to be designed as accelerators will be delivered. This process was carried out using the results from thorough analysis of the mentioned algorithms. Also, explained design methodology was administrated in this stage.

Next, design of an interface and data circuitry that enables efficient communication between RISC-V core and accelerators will be explained. All of the accelerators are intended to work connected to ODBEM, so they need similar interfaces. In addition, control circuitry, internal memory blocks, and FIFOs are required to manage, store, and manipulate data according to the needs of each operation. These shared components were designed as reusable modules and integrated into each accelerator according to necessities. After that, design of the kernel part of each accelerator that executes the operations in the selected algorithm will be introduced. And finally, design of the complete SoC with its software side will be explained.

5.2 Accelerator Cores

5.2.1 Selection of accelerator cores

For a software-hardware co-design implementation, parts from the targeted algorithm to be realized in the hardware side must be selected. As explained, instead of basic functions, it was decided to design cores for more complex algorithms. It is also anticipated that shifting more algorithmic components to hardware would result in greater overall speed improvements. However, such design decisions must also take into account area and power constraints. As previously explained, performance was prioritized above other considerations at this stage. In accordance to those points and conducted analysis of FALCON, parts from the signature generation and verification was selected and a preliminary plan for accelerator cores was formed.

Relatively small algorithms like `HashToPoint`, `Compress`, and some C code sections are also realized as accelerators, and can be added to a design for the maximum performance. These basic functions correspond to minor portion of the execution time, and are called mostly once per signature generation and verification. Some of them are

used in both parts of the FALCON. Implementation of basic functions will be explained in another subsection, and initially major functions and corresponding accelerators will be explained.

For the signature generation designing accelerator cores for complex algorithms `FFT`, `iFFT` and `ffSampling` was decided initially. These cores will be the main components of the hardware side, which are also expected to improve the overall speed the most. Next, some of the polynomial operations that are outside of those complex algorithms-like polynomial multiplication and addition-was planned to be implemented as an special Arithmetic and Logic Unit (ALU). A list of the planned cores of the major functions and their task is presented below:

- $ffSampling\_core$: Fast-Fourier Sampling operation
- fft_core : FFT conversion of polynomials-vectors
- $ifft\_core$: Inverse-FFT conversion of polynomials-vectors
- *poly_ALU* : Multiplication and addition of polynomials recurred outside of other major functions

For the verification process, the selection was more straightforward due to the relative simplicity of this part. Without the simpler algorithms, rest of the verification involves `NTT`, `Inverse-NTT` and montgomery multiplication. It was decided to realize each of these algorithms with an accelerator. List of the cores that was arranged for the verification are listed below:

- *ntt_core* : NTT conversion of polynomials-vectors
- *intt_core* : Inverse-NTT conversion of polynomials-vectors

Following the decisions regarding the accelerator cores, the design phase was initiated.

5.2.2 Design of the core interface and auxiliary components

Plotted cores need to operate connected to the ODBEM, executing required operations on the given input and prepare the output, where the required configuration information and input data sent by the RISC-V processor. Draft of the system is given in the Figure 5.3.

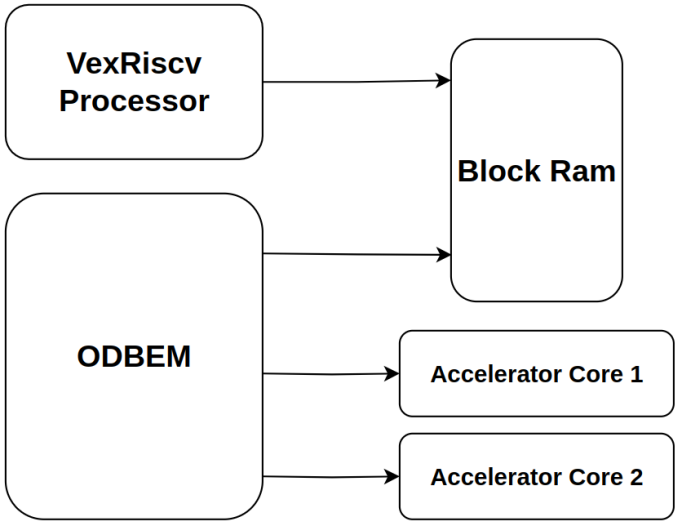

Figure 5.3: SoC with the Accelerator Cores

For compatibility with this operation scheme, every core will incorporate circuitry for input and output transmission, operation control and if necessary, storage units for intermediate data. Initial draft of an accelerator core addressing these requirements is presented in the Figure 5.4.

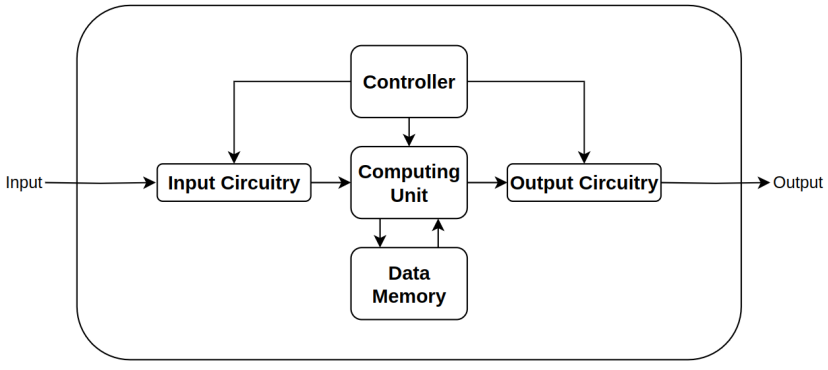

Figure 5.4: Top View of the Accelerators

ODBEM has two FIFO interfaces allocated for each processing unit, one for sending input and one for reading the output. To match with this interface, every core includes an input and output FIFO. These FIFOs hold the data in their slots when written, and can output the data in the inputting order with a read signal. First input data from the interface to the input FIFO serves as a header, carrying configuration information like selection of FALCON-512 and 1024 modes, or operation for `poly_ALU`. Data words that are to be operated follows after the header. Ordering and the size of the input will be known by tho both software and accelerator side. The input circuitry is given in the Figure 5.5

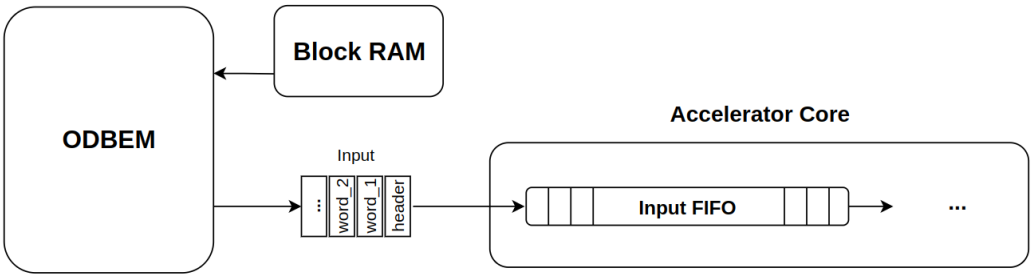

Figure 5.5: Input Circuitry of the Accelerators

The output data is written to the output FIFO, and read by the ODBEM to be written back the data memory of the system, which also can be reach by the processor. This data may also contain information about the valid output data size, or indication about successful/unsuccessful operation. The circuitry is visualized in the Figure 5.6. Ordering and the size of the output will also be known by both parties. These input and output FIFO structure is used for all of the cores, with varying FIFO capacities according to each accelerators needs.

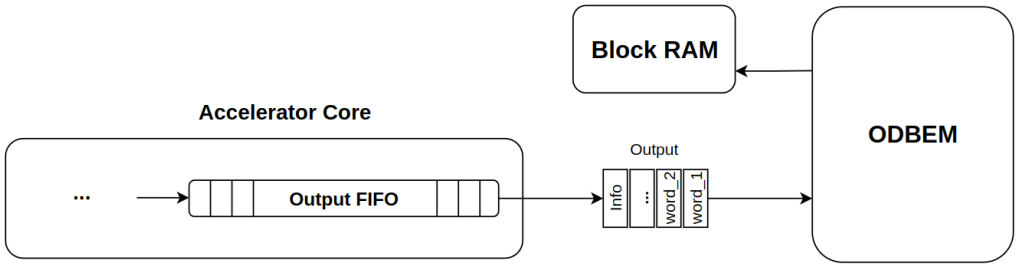

Figure 5.6: Output Circuitry of the Accelerators

Operation control circuitry, specifically designed for each core, manage and coordinate the operation of their respective cores. Configuration information in the header is used by a control unit to carry out tasks specified by a state machine. This tasks involve data movement from FIFOs to computing units and from computing units to FIFOs for all of the cores, and depending on the operation and the design of the computing units, managing of memories by generating address and write enable signals may need to be performed. Control unit, generated control signals and memories make up the control circuitry. Complex algorithms require more comprehensive control, therefore bigger control circuitry, whereas simpler cores are controlled with smaller controller and fewer control signals. Rough draft of a control unit is given in the Figure 5.7

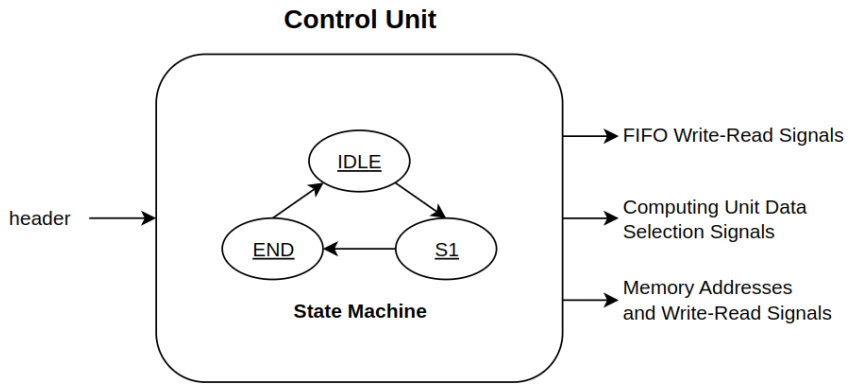

Figure 5.7: Control Unit

Operation that the accelerator is tasked with is realized with computing units, fed and controlled by the mentioned components. In the next two subsections, mentioned computing units of the cores of signature generation and verification are explained, excluding details about the input-output transmission and control mechanisms if they are not out of ordinary.

5.2.3 Design and testing of the major signature generation cores

For the signature generation, design process is started with the more complex `ffSampling`, `FFT` and `iFFT` algorithms. After that, simpler cores are attended. Simpler cores are explained briefly in a later section together with the ones from verification.

5.2.3.1 ffSampling

The function `ffSampling` is the primary component of FALCON, in terms of both complexity and execution time. This function is the realization of the novel algorithm Fast Fourier Sampling, which is also built by the creators of FALCON. Therefore, efficient implementation of `ffSampling` is projected to impact the overall performance of FALCON's signature generation the most.

Task of the `ffSampling`, as explained in the analysis chapter, is to generate a vector close to a given target vector in the lattice generated by the secret key basis, a job deemed unfeasible without possession of the secret key. Pseudocode from the reference documentation is given in the Figure 5.8.

Algorithm 11 ffSampling $_n(\mathbf{t}, \mathsf{T})$

Require: $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^n+1)\right)^2$, a FALCON tree T
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^n+1)\right)^2$
Format: All polynomials are in FFT representation.

1: if $n = 1$ then
2: $\quad \sigma' \leftarrow \mathsf{T.value}$ ▷ It is always the case that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
3: $\quad z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma')$ ▷ Since $n = 1$, $t_i = \mathsf{invFFT}(t_i) \in \mathbb{Q}$ and $z_i = \mathsf{invFFT}(z_i) \in \mathbb{Z}$
4: $\quad z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma')$
5: $\quad$ return $\mathbf{z} = (z_0, z_1)$
6: $(\ell, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
7: $\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$ ▷ $\mathbf{t}_0, \mathbf{t}_1 \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^{n/2}+1)\right)^2$
8: $\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$ ▷ First recursive call to ffSampling $_{n/2}$
9: $z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$ ▷ $\mathbf{z}_0, \mathbf{z}_1 \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^{n/2}+1)\right)^2$
10: $t_0' \leftarrow t_0 + (t_1 - z_1) \odot \ell$
11: $\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t_0')$
12: $\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$ ▷ Second recursive call to ffSampling $_{n/2}$
13: $z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
14: return $\mathbf{z} = (z_0, z_1)$

Figure 5.8: `ffSampling` Pseudocode

The target vector is formed with the output of `HashToPoint` function, which uses the message concatenated with a 40-byte random number and SHAKE256 hash algorithm to obtain the vector. In the reference documentation, the function uses a FALCON Tree, which is generated from secret key components using LDL decomposition, which is the method of rewriting a given matrix as multiplication of three matrices as $\boldsymbol{A} = \boldsymbol{LDL}^*$. But in reference C implementation, an alternative method that generates the FALCON Tree jointly with the `ffSampling` is constructed. The second method

was adopted in our design, as the pre-constructed tree structure required by the first approach is in an excessive size, and consequently introduces significant transmission latency.

As it can be seen, `ffSampling` is structured recursively, with each invocation relying on the results produced by the previous call to compute new outputs. As a result, its sub-functions are executed in a defined sequential order, processing relevant data at each level of recursion to ultimately construct the desired short lattice vector. To implement `ffSampling`, sub-modules that realize sub-functions are designed first. After that, algorithmic flow of the function is analyzed in depth, and then control circuitry that uses sub-modules and imitates the flow is built.

Sub-functions of the algorithm can be categorized as:

- *split*: Splitting of polynomials of degree n to two polynomials with the degree n/2
- *merge*: Merging of polynomials of degree n/2 to two polynomials with the degree n
- *LDL*: LDL decomposition on given polynomials as matrix components
- *samplerz*: Sample a number z near a given floating-point using Gaussian-sampling
- *tb*0: Calculation of polynomial tb0 with a set of polynomial operations, named after the variable name on the reference C implementation, and visible in the line 10 of the pseudocode

As it can be seen, sub-functions consists of polynomial operations, except sampling. So, it was initially focused on the polynomial operation related majority. Namely, target polynomials $t0$ and $t1$, and secret key polynomials $g00$, $g01$ and $g11$ are operated on. These polynomials reside in the FFT domain, meaning they are represented as complex numbers with floating-point values. As a result, polynomial operations translate into complex arithmetic, which can be reduced to series of floating-point computations. These reductions are visualized in the Figure 5.9.

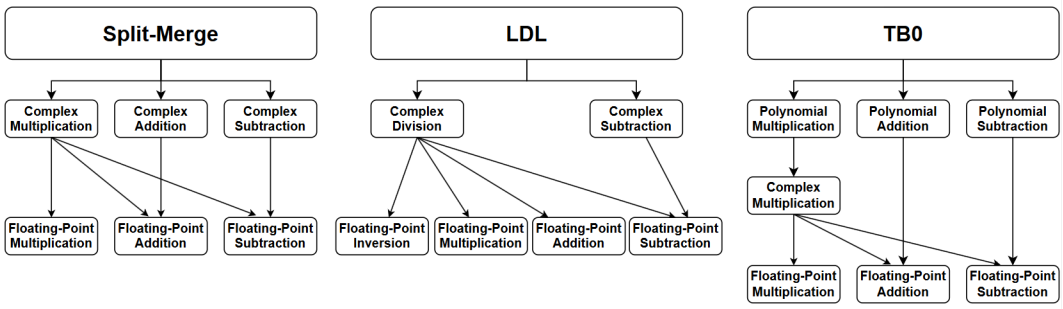

Figure 5.9: Reduction of the Sub-Functions

Polynomial sub-functions were implemented using Xilinx floating-point IPs configured for the required operations. Each sub-module accepts real and imaginary inputs through two ports, routed to floating-point units cascaded into a pipelined computation path that yields the final output.

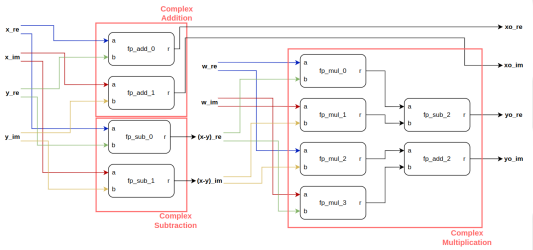

Figure 5.10a: *split*

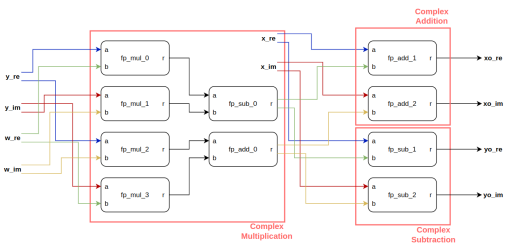

Figure 5.10b: *merge*

Figure 5.10: *split* and *merge* Sub-Modules

In the Figure 5.10, sub-modules of the *split* and *merge* are given. Complex number operations are highlighted, thus the reduction to floating-point operations can be observed. Similarly constructed sub-modules for the *LDL* and *tb*0 are given in the Figure 5.11 and Figure 5.12.

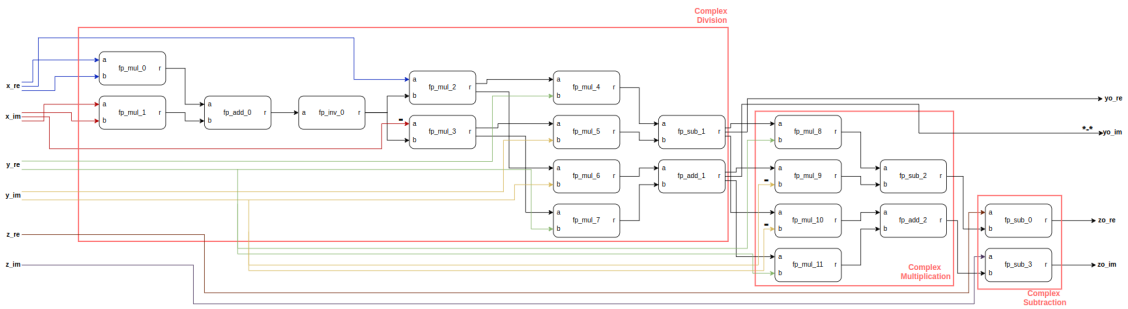

Figure 5.11: *LDL* Sub-Module

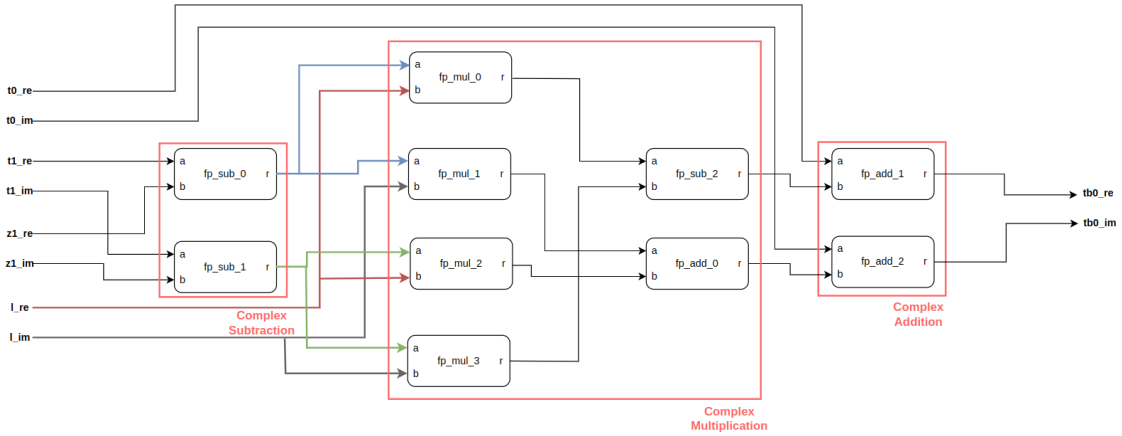

Figure 5.12: *tb*0 Sub-Module

Moving on the the `SamplerZ`, it was observed using the pseudocode given in the Figure 5.13 and reference C implementation that the function consists of loops, Gaussian sampling, random number generation and floating-point operations including square-root and exponentiation. Therefore, a sub-module with its own control and computing units are designed for the *samplerz*.

Algorithm 15 SamplerZ(μ, σ')

Require: Floating-point values $\mu, \sigma' \in \mathcal{R}$ such that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$

Ensure: An integer $z \in \mathbb{Z}$ sampled from a distribution very close to $D_{\mathbb{Z},\mu,\sigma'}$

1: $r \leftarrow \mu - \lfloor \mu \rfloor$ ▷ r must be in $[0, 1)$

2: $ccs \leftarrow \sigma_{\min}/\sigma'$ ▷ ccs helps to make the algorithm running time independent of σ'

3: while (1) do

4: $\quad z_0 \leftarrow$ BaseSampler()

5: $\quad b \leftarrow$ UniformBits(8) & 0x1

6: $\quad z \leftarrow b + (2 \cdot b - 1) z_0$

7: $\quad x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2}$

8: $\quad$ if (BerExp$(x, ccs) = 1$) then

9: $\quad\quad$ return $z + \lfloor \mu \rfloor$

Figure 5.13: `SamplerZ` Pseudocode

The control unit that regulates the operation follows a state machine, derived from the reference C function of the sampler. Differently from the other sub-modules, computing units are not pipelined, instead, each one is inputted with proper data when they are required for that state. And if the conditions oblige, state machine forms a loop. Registers are used to hold intermediate data. Simplified state machine of the *samplerz* is presented in the Figure 5.14. The state "Floating-Point Operations" consists of multiple states in the actual implementation, but basically, lines 6-7 of the pseudocode that involves many floating-point operations are realized.

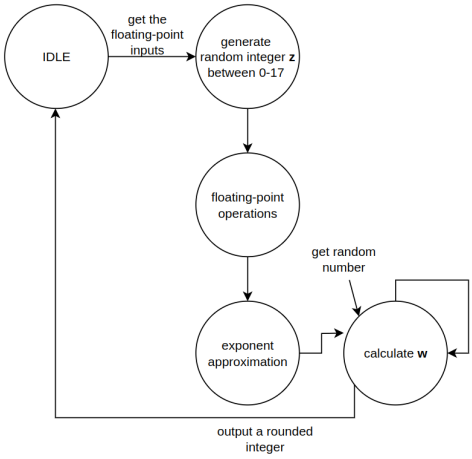

Figure 5.14: *samplerz* State Machine

Required mathematical operations for the *samplerz* are formed as inner sub-modules. Xilinx IPs are used for square-root and exponentiation. While for the random number generation, an existing True Random Number Generator (TRNG) core is utilized. This core generates random bits, and when accumulated to a specified length, they are stored in a FIFO. When required, controller of the *samplerz* gets a random number from this FIFO.

Gaussian sampling follows the reference implementation, where TRNG-generated random numbers are compared with a precomputed constant list. This list is designed so that the count of elements smaller than the random number follows a Gaussian distribution. The Gaussian sampler, along with the other components, forms the *samplerz* sub-module shown in Figure 5.15.

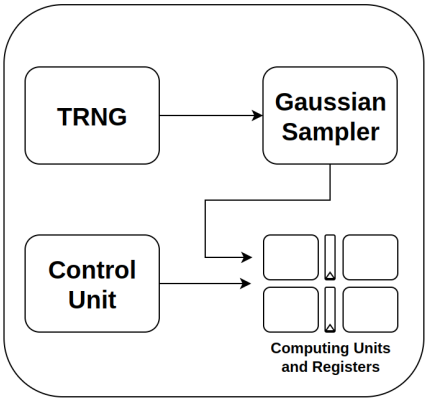

Figure 5.15: *samplerz* Sub-Module

With all of the sub-modules of $ffsampling\_core$ ready, a control circuitry that uses them to realize `ffSampling` is constructed. First, algorithmic flow of the function is analyzed using reference C codes and debugging tools. Calling convention of the sub-functions and the pattern of the recursive structure is resolved, and translated into a state machine that a designed control unit can realize. This state machine is visualized in the Figure 5.16.

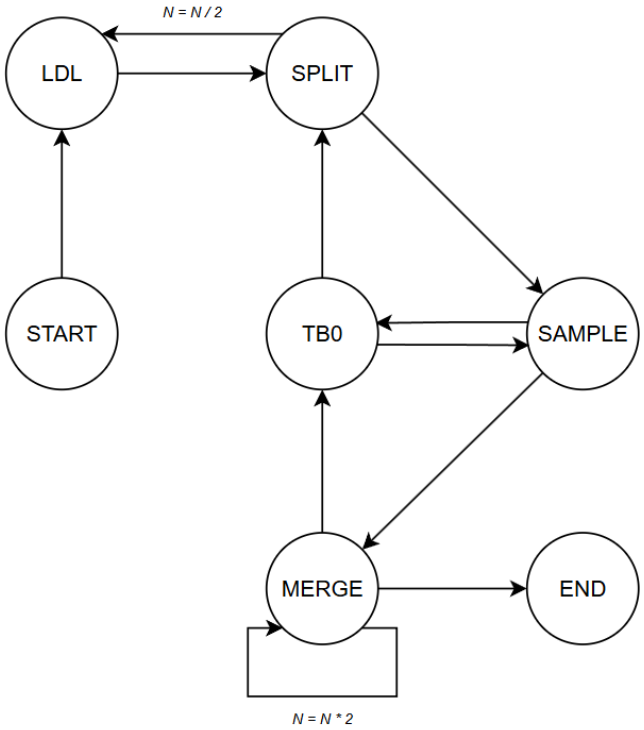

Figure 5.16: Scheduler State Machine

The control circuitry, responsible for ensuring that the entire system adheres to the defined state machine, is designed to include two data memories for the real and imaginary parts of each polynomial $t0$, $t1$, $g00$, $g01$ and $g11$, and a *scheduler* module that generates address and write enable signals for memories and input-output selection signals for computing units. Memories are dual-port, enabling simultaneous reading and writing from separate ports.

Mentioned scheduler module realizes the state machine in itself, generates control signals for each step of the algorithmic flow, and outputs them to FIFOs to be stored. Outside of the scheduler, state machines for each inputting and outputting process read those signals from FIFOs, and use them to feed the sub-modules with correct input and forward their output to correct memory addresses. This multi-state machine

topology simplifies the implementation of the otherwise complex algorithmic flow of the `ffSampling`, visualized in the Figure 5.17.

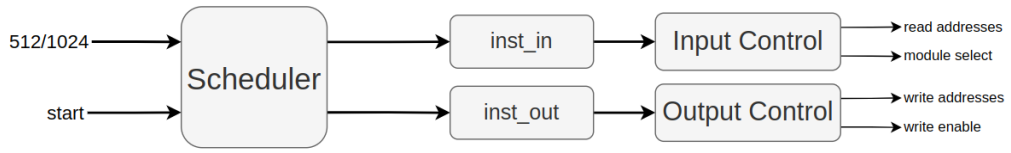

Figure 5.17: Scheduler

Scheduler, sub-modules and memories are put together on a top module given in the Figure 5.18 to form *ffSampling_core*, finally the full implementation of the `ffSampling` algorithm. An outer wrapper module that includes input and output FIFOs is wrapped around the *ffSampling_core*, and is responsible for writing input data from the input FIFO to memories inside *ffSampling_core*, initiating the operation with a start signal, and forwarding the generated output from memories to the output FIFO after the operation is finished, indicated by a done signal.

Before moving on to the testing, a second version where two *samplerz* sub-modules are utilized instead of one is constructed. As it can be seen in the lines 3 and 4 of the psuedo-code of the `ffSampling`, there are two `SamplerZ` operations that can be realized in parallel. This opportunity is utilized to enhance the performance for the price of area usage. Comparison of the versions will be discussed in the results chapter.

The core is then tested with generated test vectors from the reference C implementation. For circuit to be able to generate exactly the same output, a simulation mode is integrated to the random number generator in the *samplerz* sub-module, outputting the same numbers as the reference C implementation read from a memory. After debugging and correction phase, the design of the accelerator core is finished.

5.2.3.2 FFT and iFFT

Functions `FFT` and `iFFT` computes the FFT and Inverse-FFT transformations of given polynomial or vector, respectively. These transformations are inverse of each other, as it is obvious from their naming. Therefore, corresponding accelerators are designed similarly. Moreover, it was initially intended to realize both function in one core, but a control circuitry that is capable of conducting both will be large and slow, limiting

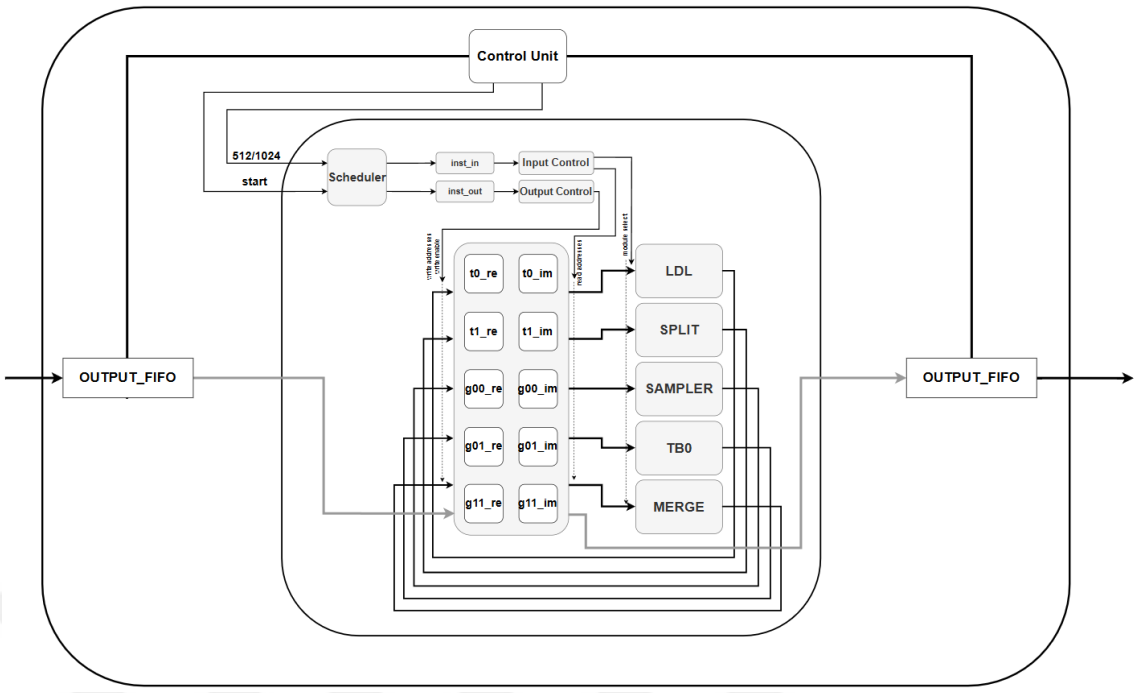

Figure 5.18: *ffsampling_core*

the maximum clock frequency and therefore performance. So, following previously explained design methodology, separate circuits are designed that target highest speed.

FFT and Inverse-FFT algorithms consists of stages, in which arithmetic operations are performed on the particular pairs of elements in a structural way to calculate new results to be used in the next stage. These paired operations are named butterfly operation, as visualization of its crosswise flow resembles a butterfly. There are several mathematical methods for the butterfly operation types, nevertheless, butterflies of the cores are selected as it is in the reference implementation. Butterflies of `FFT` and `iFFT` is given in the Figure 5.19.

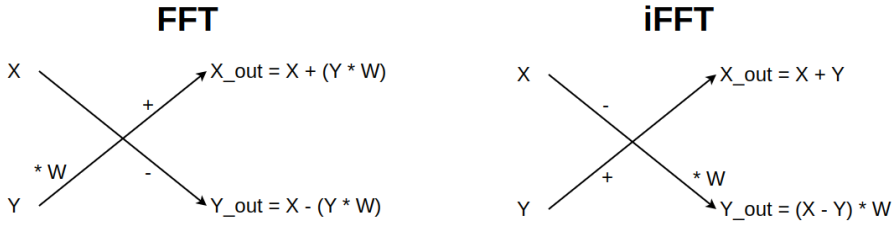

Figure 5.19: Butterfly Operations of the FFT and Inverse-FFT

Here, x and y are the paired elements, and w is a constant named twiddle factor, that is precomputed for FALCONs each 512 and 1024 mode, and stored in a memory to be used. These constants are derived from the Fourier's equation, and for each butterfly operations across all of the stages the proper twiddle factor is used.

It was noticed that the butterfly operations resemble the *split* and *merge* modules, the complex number operations are the same for both. So, modified version of those sub-modules are used as butterfly units. Using the Figure 5.20, resemblance between the sub-operations of the FFT butterfly and the *merge* sub-module in the Figure 5.10b can be seen.

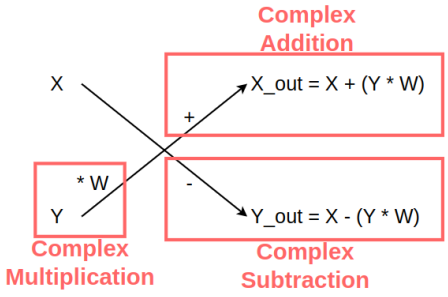

Figure 5.20: FFT Butterfly's Sub-Operations

The number of stages is determined by the number of elements, i. e. degree of the polynomial. For modes of FALCON with degrees 512 and 1024, normally, 9 and 10 stages would be needed. But because the polynomials are treated as pairs of imaginary and real parts, polynomials are treated as with degree 256 and 512 as a result. Therefore, one less stage is needed, resulting in 8 and 9 stages. And each stage, four elements of the polynomial will be inputted to a butterfly unit, meaning there are 128 and 256 operations in each stage for 512 and 1024 modes, respectively.

After the preliminary analysis and decisions, design process is started. The circuit is designed as a memory-based structure. Instead of realizing all the stages with all of the operations occurring in parallel, one butterfly unit is utilized. Similar to the $ffsampling\_core$, four memories are employed for the four inputs and outputs of the butterfly unit. And for each stage, inputs are read from correct memory addresses and inputted to the butterfly. Again as it is in $ffsampling\_core$, butterfly unit is pipelined,

enabling streamlined inputting of the data. After butterfly operation is done for a input set, generated output is then written to the proper memory address.

Using this memory-based structure, memory dependencies can occur when required data pairs reside in the same memory. This is is prevented using the method of interchanging write and read memories of pairs, similar to an existing design (Xiao, Oruklu, and Saniie, 2008).

A control unit that controls stages, input and output movement, and the mentioned interchange operation is built. Control of the modules fft and $ifft$ differs, as their stages progress backwards, therefore memory addresses and interchanging signals are generated inversely. Also, a final stage that rounds the outputs of the last stage to nearest integer is required for the inverse-FFT operation, as now the polynomial is in the integer domain. Control units are designed considering mentioned points, and then integrated together with the rest of the components to finally form the final fft_core and $ifft\_core$ accelerators, drafted in the Figure 5.21 and Figure 5.22, respectively.

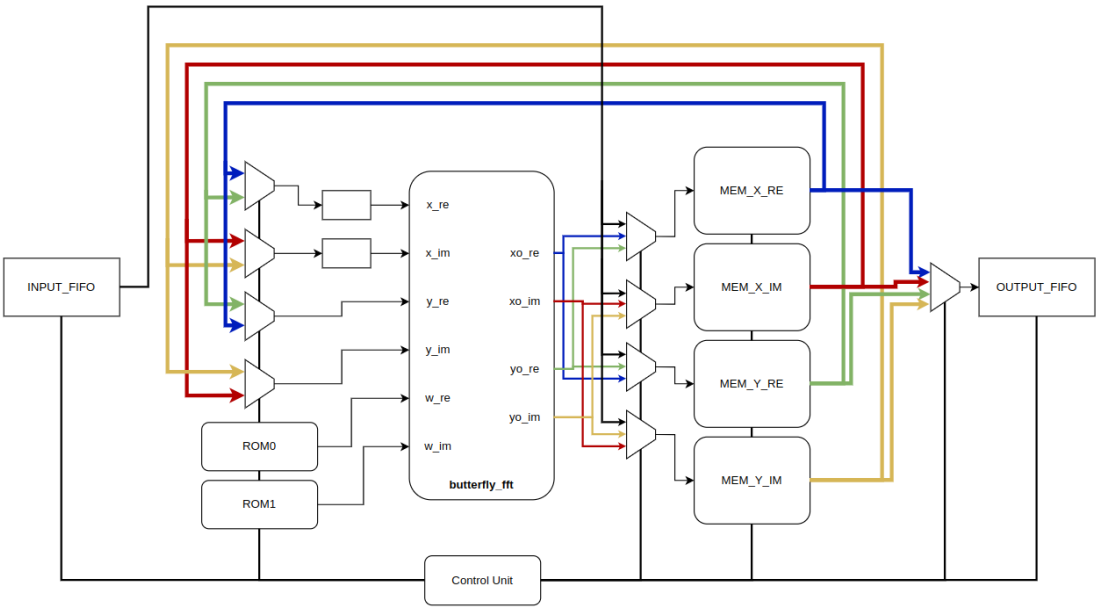

Figure 5.21: fft_core

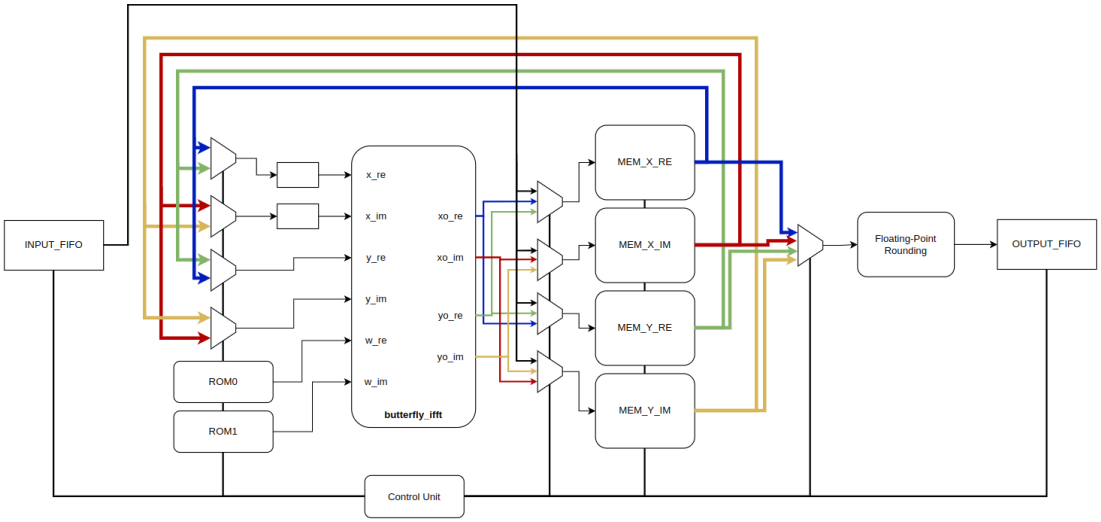

Figure 5.22: *ifft_core*

5.2.3.3 poy_ALU

Last of the major cores for the signature generation is the *poly_ALU*, which will be used for number of polynomial operations that are needed outside the other operations. The core supports complex multiplication and pair-wise polynomial addition and multiplication. Since complex multiplication is already implemented for *split* and *merge*, it was adapted to include polynomial operations, as shown in Figure 5.23

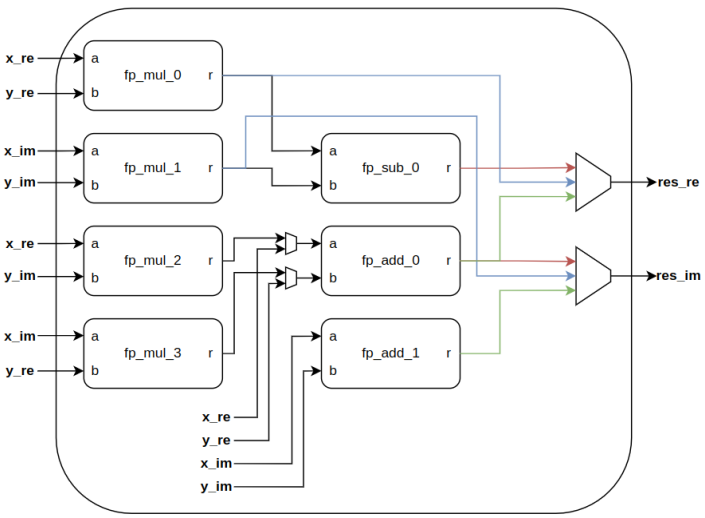

Figure 5.23: *ploy_ALU* Sub-Module

This core is then wrapped together with a control unit that regulates the operation defined in the header, to finally form the *poly_ALU_core*, given in the Figure 5.24. Differently from the other cores, *poly_ALU_core* doesn't include memories, as the operation isn't multi-staged one. Instead, four FIFOs for the four inputs of the core isutilized, as one of them also acting as the input FIFO. Input data that is directed from ODBEM is written to that FIFOs properly for each operation and inputted to the core. The calculated result is stored by two output FIFOs again for the two outputs of the core, and finally sent outside in succession

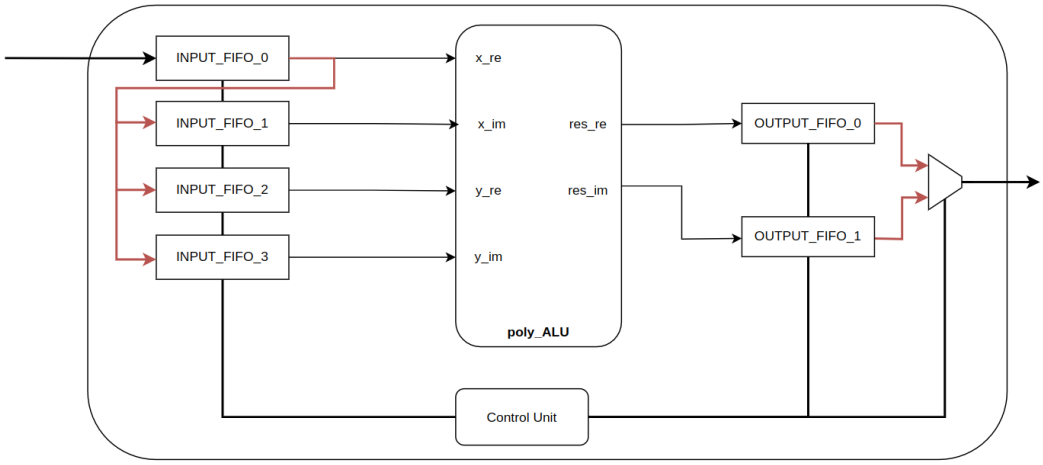

Figure 5.24: *poly_ALU_core*

5.2.4 Design and testing of the verification cores

5.2.4.1 NTT and iNTT

As previously noted, the NTT and iNTT operations are specialized forms of the FFT and iFFT, in which the inputs, outputs, and twiddle factors are integers. Consequently, a similar design methodology was employed for the development of the corresponding cores. A key distinction, however, lies in the use of arithmetic in the $\bmod q$ domain, necessitating modular reduction for addition, subtraction, and multiplication operations. Operations are visualized in the Figure 5.25

For the modulo reduction operation, a circuit realizing the K2RED algorithm tailored to the FALCON's $q = 12289$ is designed. K2RED is an efficient modulo reduction

algorithm designed to reduce the computational cost of performing modular arithmetic, a popular choice for cryptographic algorithms.

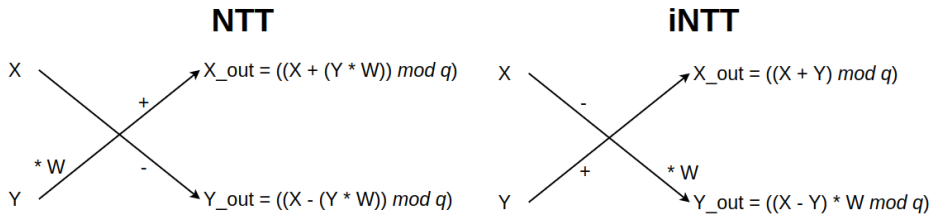

Figure 5.25: Butterfily Operations of the NTT and Inverse-NTT

Analogous to the fft_core and $ifft\_core$, the ntt_core and $intt\_core$ are implemented using a memory-based architecture. In this structure, the computation unit receives data from memory modules, with data flow and stages managed by a dedicated control unit. Differently, a computing unit that executes 4 $\bmod q$ butterfly operations is utilized, as the data coming from the input FIFO can hold 4 input data due to integer's size being smaller than the floating-point numbers. This occurrence is leveraged to enhance the overall system performance.

With their 4-input computing unit, top modules of the ntt_core and $intt\_core$ are realized again resembling their complex counterpart, fft_core and $ifft\_core$. However, unlike fft_core and $ifft\_core$, ntt_core and $intt\_core$ are almost identical, only the butterfly is different. The circuit diagram for the two can be seen in the Figure 5.26.

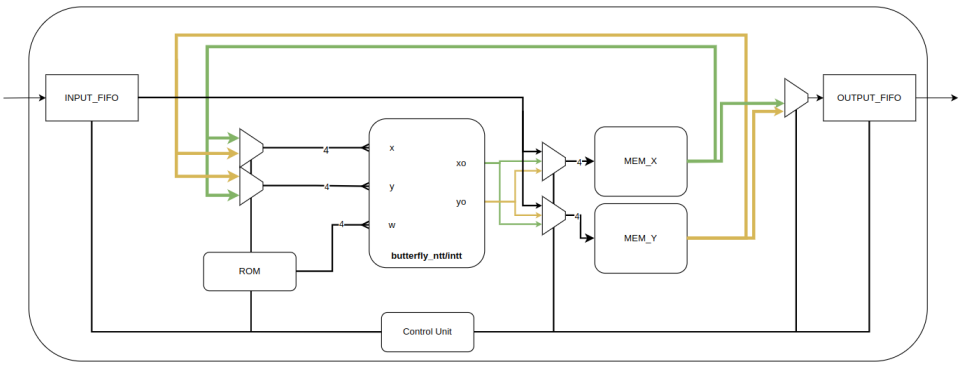

Figure 5.26: ntt_core and $intt\_core$

5.2.5 Simpler cores

Apart from the major functions and parts that corresponds to the large portion of the execution time, basic functions are also implemented and can be included in a system if the maximum performance is desired. Most of these functions involve integer and binary operations, while few incorporates floating-point operations. List of the designed accelerators and their tasks are presented below:

- *encode_decode_core* : Decoding of secret and public key
- *compress_core* : Encoding of the signature
- *decompress_core* : Decoding of the signature
- *htps_core* : Mapping message to a lattice point using existing SHAKE256 hash algorithm core
- *fpr_of_core* : Floating-point conversion of signed and unsigned integers
- *find_lp_core* : Finding corresponding lattice point of the *ffSampling_core* output
- *is_short_core* : Checking if the generated signature is sufficiently short

These accelerators are designed with the same methodology. The cores include input and output FIFOs and control components together with the required computing units. Key points from the design process of the simpler cores will briefly explained in this section.

The *encode_decode_core*, *compress_core*, and *decompress_core* realizes the encoding and decoding functions that are called on secret and public keys, and generated signature. Encoding of the keys are done in the key generation part of the FALCON, so the they are excluded. In the signature generation, first the secret key is decoded. And after a signature is generated successfully, it is decoded to be sent to a verifier. And at the verification, public key and the signature is encoded as the first step. Encoding of the keys and signature is not mandatory, but is described by the creator of

the FALCON, and is implemented in the reference C code. At the software side, the elements are stored in 8 bits, or 16 for the large valued public key. But their designated value range requires less bits. So for smaller key and signature, this encoding and decoding operations are carried out. These functions involve checking the key and signature elements for overflow from the designated value range, and bit-compress and decompress of the on those elements. The computing unit of the accelerators for these functions are realized with required control logic, bit compressors and comparators for the boundary checking. Floating-point operations are not needed, all of the elements are in the integer domain. Overviews of these cores are given in the Figure 5.27.

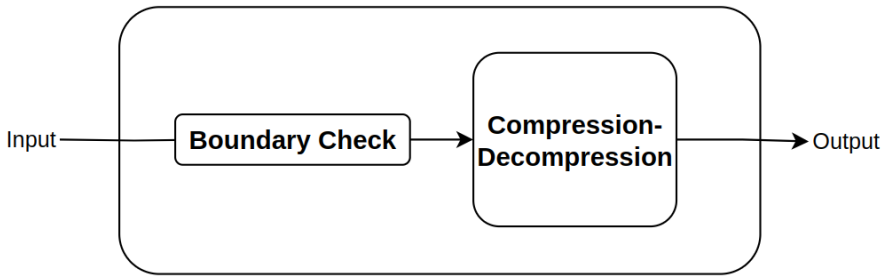

Figure 5.27: Overview of *encode_decode_core*, *compress_core*, and *decompress_core*

fpr_of_core is used for the integer to floating-point conversion that is needed before some of the major functions, like `FFT`. For the computing part of this core, a Xilinx IP configured to the conversion is used, and inputted by the input FIFO, similar to the major accelerators.

With the *htps_core*, the function `HashToPoint` which is used by both the signature generation and verification is realized. The core uses a hash string that is generated by an existing SHAKE256 core using the message and a random nonce value as input. This nonce is generated by the Deterministic Random Bit Generator(DRBG) core, again that already exists. *htps_core* uses the chunks from the hash to generate polynomial elements, effectively mapping the message and nonce input to a lattice point.

find_lp_core is used for finding the corresponding lattice point of the *ffsampling_core* output. Output of the *ffsampling_core* consists of floating-point numbers that are close to integer some integer. In the core, configured Xilinx IP is used

for rounding the element to a integer. Also, the difference between $htps\_core$ output and the rounded $ffsampling\_core$ output is calculated using integer subtraction. The $find\_lp\_core$ outputs the difference polynomial, which has integer values.

Lastly, the is_short_core uses the $find\_lp\_core$ output, and checks its norm against a pre-defined bound. Norm calculation is done in the integer domain, using multiplication and addition.

5.3 Firmware

In order for the designed hardware cores to operate correctly, they must be supported by dedicated driver software that provides appropriate configuration parameters and supplies the required input data streams. These drivers form the interface layer between the hardware and the higher-level software, ensuring proper initialization and data transmission with the computational units. In addition to the drivers, a supporting software functions are employed to prepare the input data, manage the collection and formatting of the hardware outputs, and execute the complementary algorithmic steps that remain in software. Together, these elements constitute the system firmware, which serves as the essential control and coordination layer of the overall implementation. The firmware not only enables the interaction between hardware and software components but also plays a decisive role in realizing the complete functionality of the FALCON post-quantum signature scheme in a practical, integrated manner.

5.4 System-on-Chip Design and FPGA Implementation

A top design that includes designed cores together with VexRiscv, ODBEM and a Block Memory forms the complete SoC capable of executing FALCON's signature generation and verification, given in the Figure 5.28. The modular design methodology allows for flexible selection of cores to be included in the system, based on specific application requirements. Signature generation and verification functionalities can be implemented either jointly or independently, depending on the desired use case.

The SoC design is then implemented on an FPGA using a block design. Designed accelerator cores are prepared as IPs, and connected to the ODBEM through FIFO interfaces. VexRiscv, ODBEM and Blcok memory is then connected as in the SoC design, completing the FPGA system design of the FALCON.

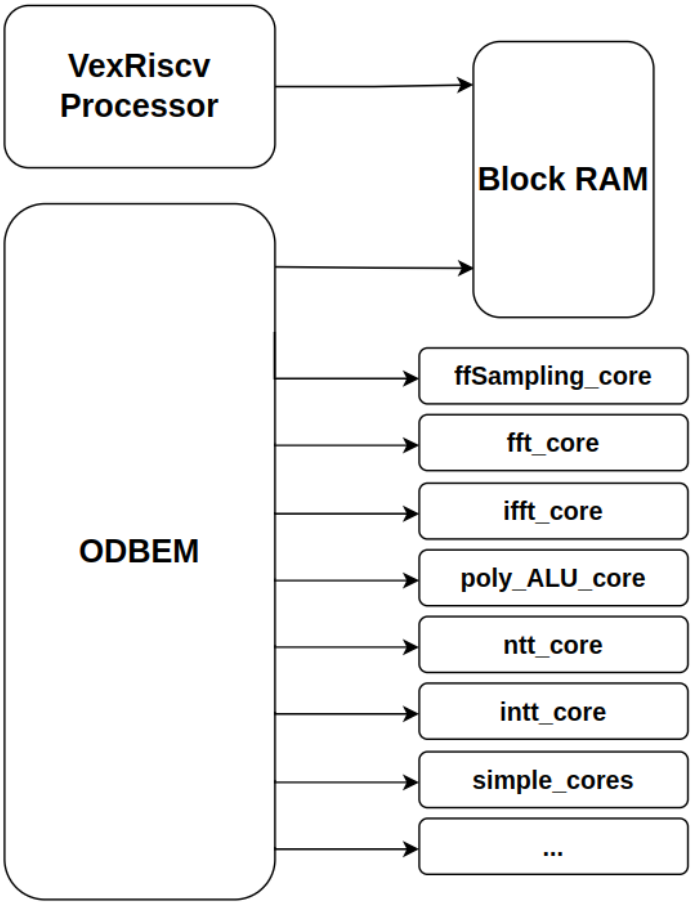

Figure 5.28: The Final SoC

Designed system first underwent testing, using the Vivado Simulator. After validation of both signature generation and verification with both the 512 and 1024 modes. After that, the system is synthesized and implemented on a real FPGA, to be tested in real time. Simulation and implementation results will be discussed in the next chapter.

6. RESULTS

In this chapter, performance results including, speed, resource consumption and area will be presented for the designed system.

The speed of the accelerator cores was the top priority, as explained earlier. Thus, the speed measurements were initially carried out. Maximum clock frequencies achieved for each of core are given in the Table 6.1.

Table 6.1: Maximum clock frequency of the accelerator cores.

	Clock Frequency (MHz)
ffSampling	200
FFT	250
iFFT	250
NTT	145
iNTT	125

The execution time of the task associated with each core was measured at the core's maximum clock speed, and is presented in the Table 6.2.

Table 6.2: Execution times of accelerator cores.

	FALCON-512	FALCON-1024
ffSampling-1	2.076 ms	4,390 ms
ffSampling-2	1.697 ms	3.395 ms
FFT	8.192 μs	17.41 μs
iFFT	8.264 μs	17.48 μs
NTT	20.765 μs	47.696 μs
iNTT	25.176 μs	44.448 μs

As compared to a standalone RISC-V core, speed improvements are very promising. These individual improvements are projected to increase the overall speed of the signature generation and verification when utilized together connected to the ODBEM and VexRiscv processor.

Although performance was prioritized, area and power consumption remain important considerations and were measured individually for each core and listed in the Table 6.3.

Table 6.3: Resource consumption and power usage of the accelerator cores.

	LUT	Flip-Flop	BRAM	DSP	Total On-Chip Power
ffSampling-1	37756	59226	42.5	289	2.830 W
ffSampling-2	49265	66335	45	324	3.219 W
FFT	15114	18019	10	0	0.742 W
iFFT	16391	20898	10	0	0.801 W
NTT	3099	1693	3	4	0.291 W
iNTT	3535	1986	3	12	0.302 W

As it can be observed, for some of the modules that uses Xilinx IPs, DSP blocks are configured to either to be used or not used in consideration of the targeted area and frequency. thr $ffsampling_core$ is unexpectedly the main consumer of resources and power, with high DSP usages. When utilized together, all of the cores and other SoC components fit to the used FPGA part.

Table 6.4: Speed comparison of this study with related work.

	Area (LUT/FF/DSP/BRAM or mm^2)	Frequency (MHz)	Sign (ms)	Verify (μs)
ARM Base	-	168	200.65	2828.73
	-	168	434.02	5823.8
RISC-V Base	-	200	748.3	12.9
	-	200	-	-
This Study (Zynq7000)	100795/121823/353/156	200	2.666	183.7
		200	5.68	363.5
This Study-2 (Zynq7000)	112304/128932/358/158	200	2.084	183.7
		200	4.685	363.5
Ouyang et al. (Zynq UltraScale+)	80497/46768/220/45	185	0.864	-
	80496/46495/220/58	185	1.728	-
Lee et al. (Samsung 28nm)	0.038	300-168	37.82	-
		300-168	80.86	-
Karabulut et al. (Artix-7)	3683/2107/-/2	121	416.563	-
	-	-	-	-
Schmid et al. (Zynq UltraScale+)	46971/44249/182/32	187.5	4.2	-
	45223/41370/182/37	187.5	8.73	-
Beckwith et al. (Artix-7)	14500/7287/4/2	142	-	16.8
	13956/6737/4/2	142	-	32.8

SoC that includes the accelerator cores, ODBEM and VexRiscv, making up the full implementation that this thesis study outputs, is simulated and signature generation and verification operation's execution time is measured. The Table 6.4 includes execution time measurements for FALCON's both modes, 512 and 1024 in separate lines, and includes different implementations across several platforms. These implementations are chosen to enable a valid and meaningful comparison of this study with existing work. Two versions of our design, one where the $ffsampling\_core$ core has one sampler and the other with dual-sampler, is indicated by 1 and 2, respectively. As expected, version 2 has better speed.

Compared to the base software implementation on ARM (Kannwischer, Rijneveld, Schwabe, & Stoffelen, 2019), that is the recommended platform by NIST for the implementers, proposed design achieves $\times 100$ improvement for signature generation, and achieves $\times 15$ improvement for the verification, for both of the operating modes. And against the bare software implementation on the RISC-V core that already exists on the system that the accelerator cores are added, $\times 280$ speed-up is achieved for signature generation and verification, respectively. This high increase from base implementations are promising, making the system a viable solution if area and power requirements are suitable.

There are very few implementations of the verification of FALCON. In fact, signature generation is prioritized in this study as well, as it is more time consuming and complex. When compared to an existing study (Beckwith, Nguyen, & Gaj, 2022), our results are $\times 11$ slower. This result is mostly due to communication overhead that comes naturally with our system design, whereas the design that is used as comparison realizes the whole verification in one circuit. Our design has the advantage of flexibility and ease of improvement, as parts of the algorithm are separate and can be worked on individually, or new accelerators can be added. Moreover, our system is capable of executing both the signature generation and verification, while the compared design only implements the verification.

When compared against the existing signature generation implementations, our system exhibits promising results. As it can bee seen from the Table 6.4, measurements

from our system with both the versions outperforms the most. Compared to a fully hardware implementations, our system achieves $\times 200$ improvement against the study of Karabulut et al. (2023), and $\times 2$ against the design of Schmid et al. (2023). Similar design that utilizes hardware/software co-design topology is outperformed by our system by $\times 17$ (Lee et al., 2024). One study that displays better results is the design of Ouyang et al. (2025) with $\times 2.6$ speed-up, that is also a hardware/software co-design system. The authors built several circuits for parts of the signature generation of the FALCON, and combined them together in a core with data buses and software interface, making it a single accelerator for the signature generation. Similarly, higher speed is due to minimum communication between software and hardware, and also due to their novel execution ordering and vectorized computing units. Similar design strategies are also used in our system like the parallelization and emphasis on floating-point operations.

The detailed comparison with existing work indicates that our system is a qualified competitor. Even though area is not targeted for optimization, it is in the acceptable region, and can be worked on at the new versions of the system. The design also has one of the best speed and the most flexibility and completeness in comparison.

7. CONCLUSION

Through the analysis of post-quantum cryptographic algorithms—particularly FALCON—a solid understanding of the mathematical foundations of lattice problems and lattice-based cryptographic schemes was developed. The in-depth examination of Falcon led to a comprehensive understanding of the algorithm, with particular emphasis on its signature generation and verification processes. The key components of these algorithms were identified and examined in greater detail with the aim of enabling hardware implementation. This process facilitated the acquisition of knowledge and experience in translating algorithmic descriptions into hardware circuit designs.

The proposed hardware implementation of Falcon's signature generation and verification, which consist of accelerator cores, a RISC-V processor and ODBEM the special DMA has resulted in a promising performance increase. Using this system, advantages of integrating custom accelerator cores to enhance the efficiency of Falcon are revealed. Throughout the design process, various methods and methodologies were studied and applied to address the encountered design challenges. A wide range of circuits was developed for tasks ranging from floating-point arithmetic to sampling operations, contributing to valuable experience across diverse types of digital circuit design. While the current system design emphasizes computational speed over resource efficiency, future research could investigate further optimizations to attain a more balanced trade-off between performance and hardware resource utilization.

REFERENCES

Alsuhli, **G.**, **Saleh**, **H.**, **Al-Qutayri**, **M.**, **Mohammad**, **B.**, & **Stouraitis**, **T.** (2024). Area and power efficient fft/ifft processor for falcon post-quantum cryptography. *IEEE Transactions on Emerging Topics in Computing*. https://doi.org/10.1109/TETC.2024.3407124

AMD. (n.d.). Amd zynq™ 7000 socs [Accessed: 2025-07-12].

Beckwith, **L.**, **Nguyen**, **D. T.**, & **Gaj**, **K.** (2022). High-performance hardware implementation of lattice-based digital signatures. https://eprint.iacr.org/2022/217

Chen, **L.**, **Jordan**, **S.**, **Liu**, **Y.-K.**, **Moody**, **D.**, **Peralta**, **R.**, **Perlner**, **R.**, & **Smith-Tone**, **D.** (2016). *Report on post-quantum cryptography* (tech. rep. No. NISTIR 8105). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8105

Ducas, **L.**, & **Prest**, **T.** (2015). Fast fourier orthogonalization [Presented at ISSAC 2016]. https://eprint.iacr.org/2015/1014

Esen, **M. M.** (2023). *Özel doğrudan bellek erisim modülü* [Master's thesis]. Istanbul Technical University. https://polen.itu.edu.tr/items/b650fb8d-fd1c-48aa-82c0-ea531dd39a49

Fouque, **P.-A.**, **Hoffstein**, **J.**, **Kirchner**, **P.**, **Lindner**, **R.**, **Nguyen**, **P. Q.**, **Prest**, **T.**, & **Whyte**, **W.** (2018). Falcon: Fast-fourier lattice-based compact signatures over ntru (version 1.2) [NIST Post-Quantum Cryptography Project].

Gentry, **C.**, **Peikert**, **C.**, & **Vaikuntanathan**, **V.** (2008). Trapdoors for hard lattices and new cryptographic constructions. *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC 2008)*, 197–206. https://doi.org/10.1145/1374376.1374407

Hoffstein, **J.**, **Pipher**, **J.**, & **Silverman**, **J. H.** (1998). NTRU: A ring-based public key cryptosystem. In **J. P. Buhler** (Ed.), *Algorithmic number theory (ants iii)* (pp. 267–288, Vol. 1423). Springer. https://doi.org/10.1007/BFb0054868

Howe, **J.**, & **Westerbaan**, **B.** (2022). Benchmarking and analysing the nist pqc lattice-based signature schemes standards on the arm cortex m7. https://eprint.iacr.org/2022/405

IEEE Computer Society. (2019, July). Ieee standard for floating-point arithmetic. https://doi.org/10.1109/IEEESTD.2019.8766229

Kannwischer, **M. J.**, **Rijneveld**, **J.**, **Schwabe**, **P.**, & **Stoffelen**, **K.** (2019). Pqm4: Testing and benchmarking NIST PQC on ARM cortex-m4. https://eprint.iacr.org/2019/844

Karabulut, **E.**, & **Aysu**, **A.** (2023). A hardware-software co-design for the discrete gaussian sampling of falcon digital signature. https://eprint.iacr.org/2023/908

Lee, **Y.**, **Youn**, **J.**, **Nam**, **K.**, **Jung**, **H. H.**, **Cho**, **M.**, **Na**, **J.**, **Park**, **J.-Y.**, **Jeon**, **S.**, **Kang**, **B.-G.**, **Oh**, **H.**, & **Paek**, **Y.** (2024). An efficient hardware/software co-design for falcon on low-end embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. https://doi.org/10.1109/ACCESS.2024.3387489

Micciancio, **D.**, & **Regev**, **O.** (2009). Lattice-based cryptography. In **D. J. Bernstein**, **J. Buchmann**, & **E. Dahmen** (Eds.), *Post-quantum cryptography* (pp. 147–191). Springer. https://doi.org/10.1007/978-3-540-88702-7_5

National Institute of Standards and Technology (NIST). (2023). Post-quantum cryptography standardization.

Ouyang, **Y.**, **Zhu**, **Y.**, **Zhu**, **W.**, **Yang**, **B.**, **Zhang**, **Z.**, **Wang**, **H.**, **Tao**, **Q.**, **Zhu**, **M.**, **Wei**, **S.**, & **Liu**, **L.** (2025). Falconsign: An efficient and high-throughput hardware architecture for falcon signature generation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2025*(1), 203–226. https://doi.org/10.46586/tches.v2025.i1.203-226

PQShield. (2025). Post-quantum signatures zoo [Accessed: 2025-07-12].

Richter, **M.**, **Bertram**, **M.**, **Seidensticker**, **J.**, & **Tschache**, **A.** (2022). A mathematical perspective on post-quantum cryptography. *Mathematics*, *10*(15), 2579. https://doi.org/10.3390/math10152579

RISC-V International. (2021). *The risc-v instruction set manual, volume i: User-level isa, document version 20191213*. https://riscv.org/technical/specifications/

Schmid, **M.**, **Amiet**, **D.**, **Wendler**, **J.**, **Zbinden**, **P.**, & **Wei**, **T.** (2023). Falcon takes off - a hardware implementation of the falcon signature scheme. https://eprint.iacr.org/2023/1885

Shor, **P. W.** (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, *26*(5), 1484–1509. https://doi.org/10.1137/S0097539795293172

SpinalHDL. (2025). Vexriscv: A fpga friendly 32-bit risc-v cpu implementation [Accessed: 2025-07-12].

Xiao, **X.**, **Oruklu**, **E.**, & **Saniie**, **J.** (2008). An efficient fft engine with reduced addressing logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *55*(11), 1149–1153. https://doi.org/10.1109/TCSII.2008.2004540

CURRICULUM VITAE

Yasin YILMAZ

EDUCATION:

- **M.Sc.:** Senior student, Istanbul Technical University, Faculty of Electrical and Electronics, Electronics Engineering Department of Electronics and Communication Engineering
- **B.Sc.:** 2022, Istanbul Technical University, Faculty of Electrical and Electronics, Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2022 - 2025 PROCENNE Digital Design Engineer
- 2021 - 2022 TUBITAK-BILGEM Part-time Researcher
- 2019 TUBITAK-BILGEM Internship