Slaven Vidaković, BSc

Design and Analysis of High Radix NTT and FFT Hardware

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisors

Florian Krieger, MSc

Sujoy Sinha Roy, Assoc.Prof. PhD

TU Graz

ISEC

Graz, March 2025

Abstract

Cryptographic methods such as Fully Homomorphic Encryption (FHE) and Post-Quantum Cryptography (PQC) are gaining traction in recent years. They provide unprecedented levels of privacy and will keep communications secure even as we enter the era of quantum computing. The fundamental operation in PQC and FHE is polynomial multiplication, which poses a significant performance bottleneck. The lowest-complexity algorithms to perform these multiplications are the Number Theoretic Transform and Fast Fourier Transform. Yet, even with these techniques, the viability of the aforementioned cryptographic methods often demands hardware acceleration. Many designs exist in this field, however, the issue with the existing solutions is the lack of flexibility regarding the degree of the polynomial, the bit-width of the modulo to the different variations of the algorithms used and above all, the radix of the transformation. This flexibility is in particular demand, as it would allow one generator tool to provide NTT or FFT modules to multiple cryptographic systems, as well as the ability to use higher radix designs. Such designs are desirable due to their lower latency than implementations with multiple lower-radix processing elements and their adaptability to memory bandwidth.
This work implements an over-encompassing solution in the form of a hardware module generator which is able to take a number of user-defined parameters, with a specific focus given to the radix, generate a design and implement it on an FPGA platform. To this end, we have overcome several challenges, with the most demanding being the memory access of the system. Not only is it vulnerable to hazards, but it also heavily depends on the degree, radix and the memory access scheme, all selectable by the user. In addition, the access pattern must change multiple times during the runtime of one operation. This problem is already challenging for a radix-2 system, nevertheless, this work proposes a solution which works for any radix and degree combination.
In addition to this, numerous designs are generated and implemented on the PYNQ-Z2 FPGA. Their evaluation shows that radix-2 implementations possess competitive resource utilization, being up to three times smaller than the state of the art, while the radix-8 designs have remarkably low latencies, up to two times less than other designs in the field.

Kurzfassung

Kryptographische Methoden wie homomorphe Verschlüsselung (HV) und Post-Quantum-Kryptographie (PQK) gewinnen in den letzten Jahren zunehmend an Bedeutung. Sie bieten ein beispielloses Maß an Datenschutz und sorgen dafür, dass die Kommunikation auch im Zeitalter des Quantencomputings sicher bleibt. Die grundlegende Operation in PQK und HV ist die Polynom-Multiplikation, die einen erheblichen Berechnungsbedarf verursacht. Die latenzoptimiertesten Methoden zur Durchführung dieser Multiplikationen sind die Zahlentheoretische Transformation (NTT) und die Schnelle Fourier-Transformation (FFT). Auch mit diesen Techniken setzt die Brauchbarkeit der vorgenannten kryptographischen Methoden oft die Nutzung einer Hardware-Beschleunigung voraus. Viele Designlösungen existieren in diesem Bereich, jedoch besteht das Problem bei existierenden Lösungen im Mangel an Flexibilität – sei es beim Grad des Polynoms, der Bit-Breite des Moduls, den verschiedenen Varianten des verwendeten Algorithmus oder vor allem bei Radix der Transformation. Diese Flexibilität ist besonders gefragt, da sie es einem Generator Tool ermöglichen würde, NTT- oder FFT-Module für mehrere kryptographische Designs bereitzustellen sowie höhere Radix-Designs zu nutzen, die aufgrund ihrer geringeren Latenz gegenüber Implementierungen mit mehreren Verarbeitungselementen niedrigerer Radices und ihrer Anpassungsfähigkeit an die Speicherbandbreite wünschenswert sind.

Diese Arbeit implementiert eine umfassende Lösung in Form eines Hardware-Modul-Generators, der eine Reihe von benutzerdefinierten Parametern verarbeiten kann – mit besonderem Fokus auf den Radix – um ein Design zu generieren und auf einer FPGA-Plattform zu implementieren. Zu diesem Zweck mussten wir mehrere Herausforderungen bewältigen, wobei der Speicherzugriff des Systems die größte darstellte. Dieser ist nicht nur anfällig für Konflikte, sondern hängt auch stark vom Grad, dem Radix und dem Speicherzugriffsschema ab – alle sind vom Benutzer auswählbar. Darüber hinaus muss sich das Zugriffsmuster während der Laufzeit einer einzelnen Operation mehrfach ändern. Dieses Problem ist bereits für ein Radix-2-System anspruchsvoll, doch diese Arbeit schlägt eine umfassande Lösung vor, die für jede Kombination aus Radix und Grad funktioniert.

Darüber hinaus wurden zahlreiche Designs auf dem PYNQ-Z2 FPGA generiert und implementiert. Diese Evaluierung zeigt, dass Radix-2-Implementierungen eine konkurrenzfähige Ressourcennutzung aufweisen und bis zu dreimal kleiner als der aktuelle Stand der Technik sind, während die Radix-8-Designs eine bemerkenswert niedrige Latenz besitzen – bis zu zweimal geringer als andere Designs in diesem Bereich.

Contents

List of Figures

List of Tables

1 Introduction

In the recent years various cryptographic techniques and methods are gaining more and more widespread use. These methods have many uses, including encrypting messages or sensitive and private data, which disallows anyone other than the owner and other authorized users access. They are also used for signing data, which guarantees its authenticity or integrity, and a host of other utilities. Even still, more and more malicious third parties try to breach these defences each day, therefore new methods are always being developed. The advent of quantum computers and everyday improvements in general computing performance, specifically in the field of integer factorizations have only exacerbated this issue. Two methods of particular interest emerged from these developments, Fully Homomorphic Encryption (FHE) and Post-Quantum Cryptography (PQC).

The fields of homomorphic encryption and post quantum cryptography are very relevant, vibrant and fast-developing. There are many works on the topic of HE, such as [YPB14], [OTD13], [Gen09] and [Cha+17], as well as on PQC, [MW17] [KP20], [Bav+23] and [BL17]. At the same time, PQC has gained sizeable fame through a competition [NIS17] ran by the National Institute of Standards and Technology (NIST) in an effort to keep future communications secure in the imminent world of quantum computing.

Not all parties that might access data not meant for them are necessarily malicious, with one example being the providers of cloud storage solutions (this of course assumes that attackers do not gain access to the server). The standard procedure of modifying data is to first decrypt it, perform the needed modifications and then re-encrypt it. This necessarily means that for any on-the-cloud operation performed on the user's data that data will spend some time in its plain-text state. Users of such services would enjoy higher levels of privacy and security if their data could be manipulated and have operations run on it without first necessitating the decryption. Homomorphic encryption can perform just this task. It is a type of encryption that allows for any operations to be carried out on the data without necessitating decryption.

On the other hand, the swift advancements made in the realm of quantum computing threaten standard encryption algorithms, with asymmetric encryption schemes being particularly affected. These algorithms rely on several mathematical principles, such as the large integer factorization or computation of a discrete logarithm.

Integer factorization is a process by which a number x is decomposed into its prime factors such that $x = y_0 \cdot y_1 \cdot \ldots y_n$. This operation is particularly problematic for standard computers seeing as no algorithm exists for quickly performing a decomposition of any number. [Mon94] states that "it is believed very hard to factor an arbitrary integer". Therefore basing a cryptographic standard on sufficiently large numbers was considered secure in the pre-quantum era. This fact made any cryptographic method

that relies on multiples of large prime numbers very secure, owing to the long time interval that such a factorization would take. One of these algorithms is the widely used RSA (Rivest–Shamir–Adleman) algorithm [RSA77]. It relies on the dichotomy of public and private keys. For a successful encryption and decryption one private and one public key (made up of two components) exist. These operations are equivalent to a chain of exponentiation and modular reduction, where exponentiating with one key would perform encryption, while exponentiating with the other decrypts. One of the exponents and the modulo would be released as a public key, while the other exponent is kept secret as a private key. This allowed for data encryption, certificate signing, or encrypting keys for symmetric cryptography, where both of the keys are identical. The generation of these keys is highly dependant on large and prime numbers, which get multiplied together, therefore the difficulty of finding these numbers, the factors of the multiplied result, kept the scheme very secure thus far. This may not be the case in the future since quantum computers could be able to perform these decompositions at unprecedented speeds, therefore rendering such methods pointless [BR20].
Similarly difficult for computers is the computation of discrete logarithms. It involves finding all solutions to $\log_x y \pmod q$. Yet again, no fast algorithm exists for this operation (as an example, [MOV96] calls the discrete logarithm problem "intractable" and gives some examples of its use), but quantum computers could speed it up significantly. [Sho94] lays out the original algorithm for the solution to this problem.
This is where Post-Quantum Cryptography proves decisive. It is a field within cryptography that aims to research and develop cryptographic functions able to stand up to the quantum computers of the future. One of the areas of development within this field is the so-called lattice-based cryptography, named after the geometric construction of the same name [Reg06].
Both of these approaches (HE and lattice-based cryptography) rely heavily on the base operations of polynomial multiplication. For any two polynomials with the degree of N, the time complexity of multiplication is in the order of $O(N^2)$. Considering the fact that these use-cases can have polynomials up to the degree of 2^{16} and bit-width of up to 64, the entire operation is quite resource intensive and cannot be performed on standard hardware within a reasonable time-frame, given that many of them have to be performed in a succession. As an example, [HMR23] states that an HE operation with such a setup takes between 65637 and 131225 μs and requires between 255 and 384 BRAM slices. As polynomial multiplication is a significant portion of the work and accounts for a significant share of the resources, it is clear that accelerating the operation, while keeping the memory usage to a minimum is of high concern.
Given this limit, a new paradigm was developed. Instead of directly multiplying two polynomials, they can be transformed using an approach called Discrete Fourier transform (DFT), into a form for which a multiplication of two polynomials
is an index-wise operation. Afterwards, the multiplications can be performed and, finally, the result can be inversely transformed. Unfortunately, the time complexity of DFT is also $O(N^2)$. This means that without significant speed-up
this method would be unviable. Thankfully, such methods do exist. In cases when the polynomial exists in an integer modular field, the approach of Number theoretic trans-

form (NTT) can be used, while if the polynomials coefficients are complex floating point numbers, Fast Fourier transform (FFT) is used. These two approaches produce the same outcome as the textbook DFT, yet take much less time to complete, thanks to the fact that they use the divide and conquer method. The time complexity of a transformation in this case is just $O(N \log N)$. This makes them a viable option for multiplying high-degree and large bit-width polynomials.

1.1 Related works

Many works in the field of both NTT and FFT exist, some of them with preset parameters, or limited customizations, others representing NTT/FFT generators which can take a handful of parameters and generate entire systems. From the point of a highly parametrizable system which can support different degrees, radices and algorithms, all the while using the minimum of memory resources and maximizing performance, each of these solutions has one or multiple drawbacks.
For example, on the NTT front: [HMR23], [Kri+24b], [Mer+20], [Liu+24], [FS19], [LPY22] and [Mu+22] support only radix-2 NTT implementations, while this work currently implements radix-2, radix-4 and radix-8 butterflies, with the implementation of radix-$2^k, k \in \mathbb{N}$ a fairly straight-forward procedure. On the other hand [Ngu+24] and [Che+21] support radix-2 and radix-4 implementations only.
Further, [Kri+24b] does not store pre-computed twiddle factors, opting instead to compute them at runtime. While this can reduce the memory requirements, it requires circuitry to perform such operations in parallel.
In addition: these solutions, by and large, tend to implement only one algorithm with only one memory access scheme, with some of the examples being [Mer+20], [FS19] and [LPY22]. This work will give the user the ability to create a variety of algorithm-memory access combinations, which will be explored in the rest of this work.
Another issue is represented by non-optimal memory resource usage, where systems implementing algorithms which sacrifice BRAM usage for higher performance end up with non-minimal memory utilization. An example would be [Liu+24].
Meanwhile, [Ngu+24] and [LPY22] represent the issue of having limitations on either the width of the polynomial coefficients, or degree, with [Ngu+24] having a modulo with a width of 23 bits and [LPY22] having the modulo bit width constrained to 32 bits or less and degree to 32k or less. On the other hand, this implementation can support the bit-width of 64 and any degree given as $R^k, k \in \mathbb{N}$.
Furthermore, some software solutions exist for computing NTT transformations, such as [AM+16], however, due to the fact that they are not hardware accelerated, the performance will, naturally, be lower.
In the domain of FFT solutions some of the same problems emerge, for example, [Che+18], [SFR09] and [Kri+24a] offer only radix-2 implementations, while [Yan+23] offers both radix-2 and radix-4 butterflies. This work will, in the same manner as in NTT provide implementations for radix-2, radix-4 and radix-8 butterflies and support

for any arbitrary radix-$2^k, k \in \mathbb{N}$ butterfly.
At the same time, the inflexibility of offering only one combination of DIF/DIT NR/RN algorithms is still prevalent, with [SFR09] and [Yan+23] being some of the examples. Again, analogue to the NTT, any of the four combinations of FFT algorithms are viable is this solution.
The last FFT specific downside of the explored solutions is their usage of lower-precision floating point formats. While [Che+18] offers only support for 32-bit floating point numbers and [SFR09] supports fixed-point numbers with 16 bits dedicated to the whole and 16 bits to the fractional part of the number, this solution implements double width floating point numbers (64 bits).
Finally, one downside in common to all of the mentioned designs is their inability to be parametrized into either FFT on NTT capable systems. This necessitates for the usage of multiple solutions in cases where both operations would be required. In contrast this design can be switched between both of these algorithms with just a single parameter. This will automatically instantiate and set up all of the required sub-systems as well as the system wrapper, thus requiring no further attention from the user (apart from making sure that the format of the delivered data is congruent with the selected transformation).

1.2 Contributions of this work

The main goal of this work is to offer a flexible and easy-to-use parameter-driven NTT and FFT hardware design generation tool. It takes a number of parameters whose values are defined by the user and generate a fully functional FFT/NTT transformation system, along with a wrapper which connects it to a BUS. Test benches for both the stand-alone system and the wrapper are also generated, along with a program meant for the master CPU. All of these parameters are synthesis-time flexible, meaning that they can only be changed before the synthesis step, as opposed to being modifiable during runtime, which is done to minimize resource and power usage. This work aims to implement a great number of changeable parameters in regards to performing NTT and FFT transformations on arbitrary polynomials. This includes the ability to switch between the two transformations, choose DIF or DIT algorithms, NR or RN memory access schemes, define the widths of all of the actors within the transformation, as well as the degrees of involved polynomials. The most important contribution of this work is the ability to instantiate and use any arbitrary transformation radix. This point is especially lacking in state-of-the-art solutions, thus the design choices within this work revolved around solving this issue. This goal will help the work stand out within the field as a solution where many of the usually predetermined factors can be freely changed by the user.
These goals posed a unique challenge in designing the system due to the fact that many of the static design considerations now had to be dynamic, and no section of the design can be inflexible. Every step in the transformation algorithm is informed by one or more parameters, and thus several challenges emerged in the process of design. The first of these is the problem of loading and storing the data in such a way as to prevent memory

access hazards, while preserving the functionality of the design. To handle the loading part, a new index generator was developed. It takes into account nearly all of the available parameters to generate a constant stream of indices to be fed to coefficient memory as addresses. It does this by utilizing several equations with most of their constant values being determined at elaboration time based off of provided parameters. This index generator permits the optimal memory resource utilization as well as a performance-optimized, pipelined system. In addition to this, it can generate indices for the loading of twiddle factors with any combination of permitted parameters.
To help with the complexity of such a module, the order of coefficients within memory has to be strict and predictable. To satisfy this requirement, a memory writing system was developed. It is referred to as the memory intercept, and it is capable of intercepting all outputs of a transformation system butterfly unit, calculating their appropriate address and storing them within RAM. Similarly to the index generator, it utilizes parameters as constant values in all of its calculations and can be used for any radix/degree combination in both NR and RN modes.
Finally, to accommodate variable radices, a flexible butterfly unit system is developed. It had to be able to serve any arbitrary radix and be able to do either NTT or FFT operations. This requirement was met with the development of a template for the generation of any radix butterfly core unit in both NTT and FFT configurations. Although only radices 2,4 and 8 are analysed in this paper, the template allows the user to create a butterfly unit for an arbitrary $2^k, k \in \mathbb{N}$ radix by following a set of simple steps. Afterwards, the appropriate butterfly unit will be instantiated in accordance with the user-defined parameters, by the elaboration time directives within the system.
With the solutions to these problems developed and implemented, the full system was made functional and was analysed and presented within this work.
The full extent of this work includes:

1. A highly parametrizable transformation system implemented in hardware
2. A wrapper for said system meant to attach it to a CPU to act as a hardware accelerator
3. Firmware for a CPU that makes use of the transformation accelerator
4. Test benches for both the stand-alone system, as well as the wrapper which can test the functionality of any set of parameters up to any number of run-iterations
5. Software model of the transformation system used for verification purposes
6. An entire host of supporting software scripts that assist in conducting verification
7. The generator, which sets up different parameters for synthesis and implementation
8. Software scripts which generate twiddle factors for use inside the system

The aforementioned generator is fast and easy to use. The workflow is demonstrated in Figure 1.1.

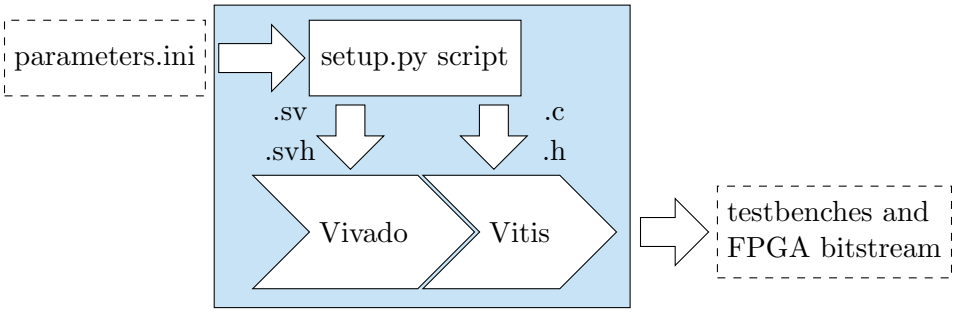

Figure 1.1: Work flow of the FFT/NTT generator

The user will define the values of all of the parameters inside the *parameters.ini* file. The *setup.py* is then ran, which configures all of the hardware description and software source code files. Afterwards, the test benches can be used, or the provided hardware modules, along with the software can be compiled and ran on an FPGA. In addition, provided software implementation files will also be configured. They can be ran as a demonstration of the transformation algorithm. By using this flow a user can generate and test a transformation system which can further be embedded into a larger project, or utilized on its own.

1.3 Structure of the work

This chapter will describe the makeup and purpose of each chapter within this paper in short lines. For more details, the chapter in question should be viewed.

1. Introduction - This chapter introduces the reader to the work and gives some motivation as to why the work was carried out and what the benefits of the work are.

2. Mathematical basis - This chapter introduces the reader to some mathematical concepts that will be used and referenced through the work.

3. Algorithms - This chapter is concerned with various optimization techniques, the concrete implemented algorithms, as well as some decisions made within those algorithms.

4. Hardware - This chapter goes over the hardware implementations of all of the concepts introduced in the previous chapter, as well as some hardware-specific optimizations.

5. Results - This chapter presents the results of synthesis and implementation on an FPGA platform, including resource utilization, power, latency, etc., as well as some graphs and result discussion.

6. State of the art - This chapter compares this work with similar works in the field, giving pros and cons of each and discussing why this work might be considered as an alternative.

7. Conclusion - This chapter aims to conclude the work and construct a closing statement.

2 Mathematical basis

An unavoidable operation in many cryptographic and other computational disciplines is polynomial multiplication. If for two polynomials $A(x)$ and $B(x)$ of degrees $N(a)$ and $N(b)$, respectively, given as:

$$A(x) = \sum_{i=0}^{N(a)} a_i \cdot x^i \tag{2.1}$$

$$B(x) = \sum_{i=0}^{N(b)} b_i \cdot x^i \tag{2.2}$$

the polynomial multiplication gives a polynomial $C(x)$, of a degree $N(c)$ given as:

$$A(x) \times B(x) = C(x) = \sum_{i=0}^{N(c)} c_i \cdot x^i \tag{2.3}$$

then the individual elements of the polynomial $C(x)$ are given as:

$$c_i = \sum_{\substack{0 \le j \le N(a) \\ 0 \le k \le N(b) \\ j+k=i}} a_j \cdot b_k \tag{2.4}$$

As can be observed, in order to multiply two polynomials of a degree N, a total of N^2 multiplication operations must be performed (in addition to a number of computationally trivial additions). In computational systems, multiplications are very costly operations and, as such should be avoided (or reduced) whenever possible. This is especially true of cryptographic applications, where polynomials can be as large as $N = 2^{16}$ and the individual elements can be as large as 64 or 128 bits wide.
To achieve this reduction, two mathematical operations named 'Number theoretic transform' and 'Fast Fourier transform' (NTT and FFT in further text, respectively) can be used. They transform polynomials in such a way that their multiplication is a coefficient-wise operation. Albeit the result of this multiplication is transformed in itself, so an additional inverse transformation is needed. This cuts the number of multiplications down from N^2 to N, while the transformation itself has the time complexity of $O(N \log N)$. The method of point-wise multiplying transformed polynomials is explored in [Für07].

2.1 Number theoretic transform

NTT can be used to transform integer numbers on a modular ring. The entire operation is described in [Für07].

The formula for achieving NTT transformations is as follows:

$$\hat{a_i} = \sum_{j=0}^{N-1} \omega^{i \cdot j} \cdot a_j \pmod{q} \tag{2.5}$$

where $\hat{a_i}$ is the $i-th$ coefficient of the transformed polynomial $\hat{A}(x)$, ω is a so-called $N-th$ primitive root of unity, $\omega^{i \cdot j}$ is a so-called 'twiddle factor', q is the ring modulus and $N-1$ is the degree of the polynomial $A(x)$.
Directly implementing this formula will yield an operation with a time complexity of $O(N^2)$, but this can be reduced with algorithms described in the 'Implementation' section.

2.1.1 Modular ring arithmetic

NTT is an operation defined over polynomials whose coefficients are defined on a modular ring. This means that for a modulo q, every coefficient can have values between 0 and q-1, inclusive. This modulo has to be prime and satisfy the following:

$$q \equiv 1 \pmod{N} \tag{2.6}$$

The polynomials are also defined as:

$$R_q(x) = \frac{\mathbb{Z}_q[x]}{\Phi(x)} \tag{2.7}$$

where the $\Phi(x)$ is an irreducible polynomial. Note that all degrees used will be equal to $2^k - 1, k \in \mathbb{N}$.

2.1.2 Primitive N-th root of unity

For the chosen q and, if the degree of the polynomial is given as $N-1$, a so-called primitive $N-th$ root of unity is a number given as:

$$\omega^N \equiv 1 \pmod{q} \tag{2.8}$$

with the further restriction of:

$$\omega^k \not\equiv 1 \pmod{q}, \forall k | k < N \wedge k \neq 0 \tag{2.9}$$

2.1.3 Negative wrapped convolution

When using the naïve NTT approach, two polynomials of degrees N will first have to be zero-padded to $2N$ before being put through transformation and multiplication. However, there is a way to cut the required operations in half (i.e., skip the padding step) via an optimization known as 'Negative wrapped convolution' (NWC).
If the given polynomial is from the ring:

$$R_q(x) = \frac{\mathbb{Z}_q[x]}{x^N + 1} \tag{2.10}$$

and a $2N - th$ primitive root of unity exists, denoted as ψ, the requirement of which is:

$$q \equiv 1 \pmod{2N} \tag{2.11}$$

and provided that each coefficient a_i of both of the polynomials is multiplied with ψ^i, only two N-point transformations will be necessary. In that case, each coefficient a_i of the multiplication output will have to be multiplied by ψ^{-i}.
This work will not take these pre-processing (and post-processing) steps into account and the expectation is that every polynomial is pre-processed before being fed to the transformation system (and post-processed afterwards). There is, however, a modification to the NTT process known as the 'merged NTT' where all of these steps are merged into one operation, which has an effect on twiddle factors used. The NWC optimization is described in [HMR23].

2.2 Fast Fourier transform

Unlike the previously discussed NTT, Fast Fourier Transform functions on complex decimal numbers. This means that every coefficient, twiddle factor and intermediate product consists of two parts, real and imaginary. In addition to this, modular reduction operations are not needed anymore.
The Fast Fourier Transform (FFT in further text) is an implementation of the Discrete Fourier Transform (DFT) algorithm. The DFT algorithm can be seen below:

$$\hat{a_k} = \sum_{j=0}^{N-1} e^{\frac{-i2\pi}{N} \cdot kj} \cdot a_j \tag{2.12}$$

where $\hat{a_k}$ is the $k - th$ coefficient of the transformed polynomial $\hat{A}(x)$, and $e^{\frac{-i2\pi}{N} \cdot kj}$ acts like the twiddle factor.
This operation is further explored in [CLW69].

3 Algorithms

As the time complexity of performing the textbook DFT operation on a polynomial equals $O(N^2)$, optimizations must be introduced to make it a viable alternative to textbook multiplication of polynomials. The backbone of such optimizations is a so-called divide and conquer method. Two main variants of this method exist for both NTT and FFT applications, Gentleman-Sande (**GS**) [GS66] and Cooley-Tukey (**CT**) [CLW67]. The commonality between these methods is splitting the polynomial into multiple smaller polynomials recursively until the sub-polynomials are of a size equal to the radix. At that point the standard formulae are applied to them.

3.1 Gentleman-Sande method

The GS method, also called 'Decimation in Frequency' or DIF for short, splits the polynomial into successive parts of size $\frac{N}{R}$, where N is the polynomial degree and R is the transformation radix.
The formula for the complete DIF NTT transformation is:

$$\hat{a_i} = \sum_{p=0}^{R-1} \sum_{j=\frac{p \cdot N}{R}}^{(p+1) \cdot \frac{N}{R}-1} a_j \cdot \omega^{i \cdot j} \pmod{q} \tag{3.1}$$

The formula for the complete DIF FFT transformation is:

$$\hat{a_i} = \sum_{p=0}^{R-1} \sum_{j=\frac{p \cdot N}{R}}^{(p+1) \cdot \frac{N}{R}-1} a_j \cdot \omega^{i \cdot j} \tag{3.2}$$

where p represents the current part of the polynomial and q is the modulo of the NTT polynomial ring.
Both of these partitions will be recursively repeated until the subdivided polynomial is of size R. This is possible only if the original polynomial is of a size $N = R^k, k \in \mathbb{N}$.
The GS method is further explained in [GS66].

3.2 Cooley-Tukey method

The CT method, also called 'Decimation in Time' or DIT for short, splits the polynomial into parts of equal modularity with regards to the operation $N \pmod{R}$.

The formula for the complete DIT NTT transformation is:

$$\hat{a}_i = \sum_{p=0}^{R-1} \sum_{j=0}^{\frac{N}{R}-1} a_{Rj+p} \cdot \omega^{i\cdot(Rj+p)} \pmod{q} \tag{3.3}$$

The formula for the complete DIT FFT transformation is:

$$\hat{a}_i = \sum_{p=0}^{R-1} \sum_{j=0}^{\frac{N}{R}-1} a_{Rj+p} \cdot \omega^{i\cdot(Rj+p)} \tag{3.4}$$

where p represents the current part of the polynomial and q is the modulo of the NTT polynomial ring.
Same as with the previous method, these partitions continue until the subdivided polynomial is of size R, with the same restriction regarding the polynomial degree.
The CT method is further explained in [CLW67].

3.3 Memory addressing schemes

Regardless of the choice of method, the implemented algorithm can have one of two memory addressing schemes:

- Normal to reverse (NR) mode
- Reverse to normal (RN) mode

In NR schemes the input coefficient a_i will be in the $i-th$ location in memory, while the analogous transformation result will be in the location corresponding to the bit inverse of i. Conversely, in RN schemes the ordering is opposite, i.e. the input coefficient a_i will be in the location corresponding to the bit inverse of i, while the analogous transformation result will be in position i.
These properties are useful since they allow for a polynomial to be transformed using NR DIF (as an example) and inversely transposed using RN DIT, thus maintaining the original order of coefficients.

3.4 Avoiding RAW hazards

In a system with a radix R, each mathematical operation consumes R inputs and generates R outputs. Therefore, as one memory bank can only provide one element per clock cycle, the entire polynomial will be stored in R RAM banks of size $\frac{N}{R}$, where N is the total number of coefficients within the given polynomial.
To reduce the memory requirement of the design, this same RAM will be used to store the initial coefficients a_i, the intermediate values $\breve{a}_i$ and fully transformed coefficients $\hat{a}_i$. As R elements need to be read per clock cycle - one from each bank, a common technique to simplify the generation of their individual indices is to store them in the same addresses across the memory banks. This way a single index can be used for all reads.

This introduces RAW (read after write) hazards into the system. If an outcome of one operation is written to memory locations addressed by another operation, which takes place immediately afterwards, that second operation will get the wrong inputs and thus, produce wrong outputs.
One way of avoiding this problem is using a pipelined system, such that all operations that put each other at risk of RAW get their inputs loaded into the pipeline sequentially (and that the pipeline length allows all affected coefficients to be loaded at once). This necessitates the creation of a sub-system that can generate addresses of such operations as close to each other as possible, as well as a sub-system that can handle writing back to correct addresses post operation.

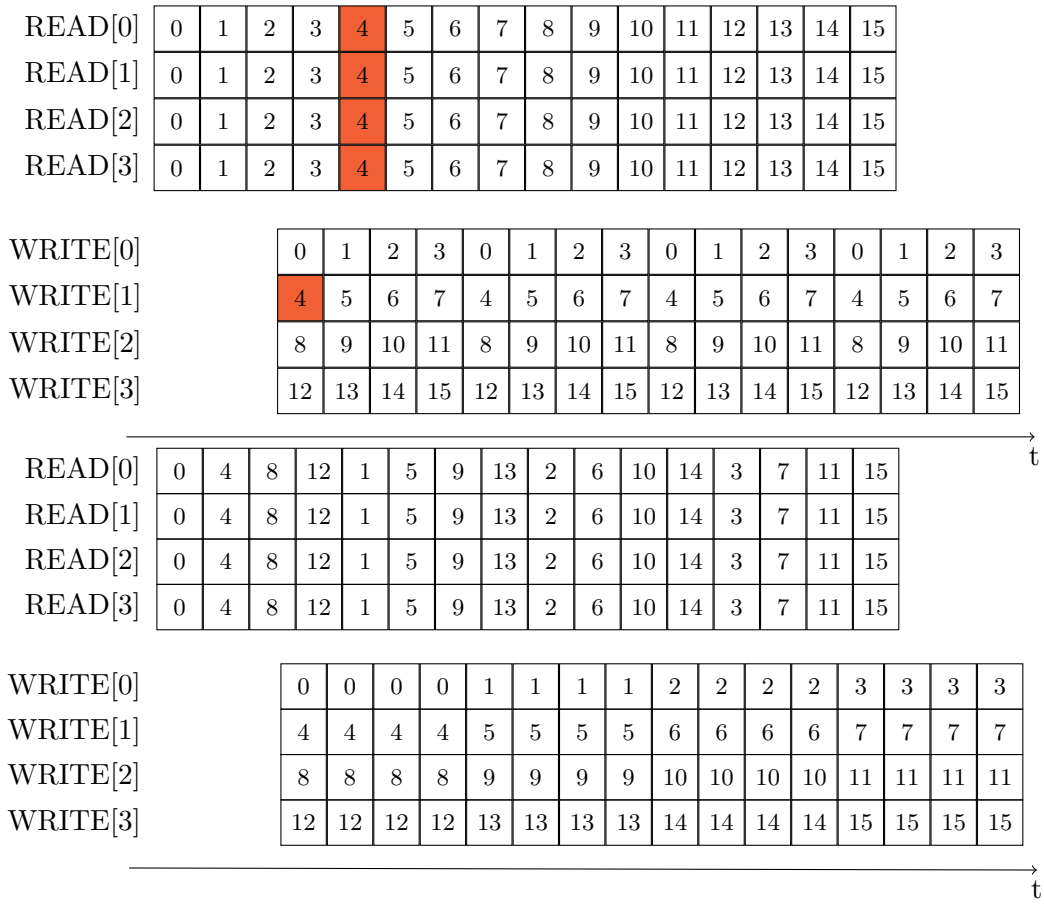

Figure 3.1: Naïve approach (top) and hazard avoidant approach (bottom) for index generation

Provided that some system with $R = 4$ and $N = 64$ was given. The system is ideal, which means that all operations take exactly 4 cycles, all reads and writes are instant, reads happen at the earliest point in time in a clock cycle in which they're shown and writes happen at the latest point it time in a clock cycle in which they're shown. For example,

in Figure 3.1 at the very beginning of clock cycle 0, a read occurs, then the operation lasts through the clock cycles 0, 1, 2 and 3 and in the very last instant of the clock cycle 3, the write occurs. Figure 3.1 showcases the difference for such a system in the naïve approach of incrementing the index by one per operation and the hazard-avoidant approach with a more intricate index generation algorithm. The marked elements show the first instance of a RAW hazard in the naïve system. The coefficient in position 4 is changed before being loaded, which will produce wrong results. The hazard-avoidant system has no such problems.
The logic behind the writing of coefficients to memory is explained in Algorithm 1 and Section 3.6.2.

3.5 Generation and efficient storage of twiddle factors

To avoid calculating the twiddle factors at runtime, it would be beneficial to pre-compute them and store them in memory. This presents a new problem of increased memory use, so a more optimized storage solution should be considered. To this end, two twiddle factor attributes, periodicity and symmetry, can be exploited.
The periodicity of twiddle factors is defined as the following relation:

$$\omega^{k+N} = \omega^k \tag{3.5}$$

The symmetry of twiddle factors is defined as the following relation:

$$\omega^{k+\frac{N}{2}} = -\omega^k \tag{3.6}$$

These twiddle factor properties can be seen in [Che+21].
From these two relations, the following conclusions are made: it is possible to skip storing every twiddle factor that can be described by either of the two left-hand side statements and modify the design of the system such that the right-hand side statement is used instead. This means reading the same twiddle factor for multiple operations to satisfy periodicity and reading the same twiddle factor for multiple operations and making one operation a subtraction, instead, to satisfy symmetry. These two conclusions could allow for a smaller memory footprint, but the required design complexity of twiddle factor index generation severely offsets any gains; on the other hand, the second conclusion will allow for an elimination of one multiplication per radix-2 core unit. Considering the fact that every unit of greater radix uses radix-2 units as submodules, this significantly reduces the number of instantiated multipliers.
Another optimization that can be carried out in NTT algorithms with regards to twiddle factor generation has to do with the functionality of modular integer reduction units present in the design. As they are Montgomery reduction units, rather than calculating the value of

$$x = y \pmod{z} \tag{3.7}$$

they calculate

$$x = y \cdot r^{-1} \pmod{z} \tag{3.8}$$

where r is a constant based on the attributes of a reduction unit. Since all reduction units are placed after multiplications with twiddle factors, this downside can be avoided by simply multiplying all twiddle factors by r during generation. For brevity, the factor r will not be mentioned alongside twiddle factors in the following text.
In addition, there are $\frac{R}{2} - 1$ internal twiddle factors, which are calculated as:

$$\omega_k = \omega_n^{\frac{(k+1)\cdot N}{R}} k = 0, ... \frac{R}{2} - 2 \tag{3.9}$$

3.6 Implemented algorithm

The limitations and developments discussed in the previous chapters heavily influenced the development of the main algorithm, which is presented in Algorithm 1. As mentioned in Sections 3.1 and 3.2, the polynomial is divided into sub-polynomials of size R, which then undergo FFT or NTT transformation. The resulting polynomials have to undergo the same operation recursively until the transformed polynomial is of the original size N.
As these divisions happen $\log_R N$ times, there need to be $\log_R N$ stages to the algorithm, which is accomplished via the for loop on line 2.
Furthermore, to exploit the useful property of NR and RN addressing schemes mentioned in Section 3.3, the final stage of every transformation addresses all coefficients sequentially and writes the results back in the origin addresses of the coefficients. Thus the distinction is made between standard stages, as seen in the if statement on line 8 and the final, special stage in the else statement on line 26.
The algorithm can, roughly, be divided into four parts:

1. Generation of indices (addresses for coefficients and twiddle factors) (lines 11 & 12).
2. Loading of coefficients and twiddle factors (lines 14 & 17).
3. Butterfly operations (line 19).
4. Result write-back (line 21).

While step two is self-explanatory and step three will have its own chapter, steps one and four warrant further explanation.

3.6.1 Generation of indices

As mentioned previously, in order to avoid RAW hazards, coefficients involved in operations that will impact each other must be addressed in groups. These groups consist of indices exactly O apart, where O is calculated in either step 4, or step 5 of the algorithm. This addressing is accomplished with the for loop on line 10.
On the other hand, individual groups of indices can be generated in any order, but to keep the complexity of the system down, they will simply be addressed sequentially. This is handled by the for loop on line 9.
The indices for the twiddle factors are generated by a separate algorithm, which will be explored in Section 3.7.

Algorithm 1 Transformation algorithm

Input: $Mem[R][\frac{N}{R}]$=(a_0,a_1...a_{N-1}), Polynomial coefficients
Input: $TMem[R-1][\frac{N}{R}]$=(ω^0,ω^1,...$\omega^{N-\frac{N}{R}-1}$), Twiddle factors
Input: $TInt[\frac{R}{2}-1]$ =(ω_0,ω_1,...$\omega_{\frac{R}{2}-1}$), BFU internal twiddle factors
Input: N, R, q
Output: $Mem\_r[R][\frac{N}{R}]$=($\hat{a_0}$,$\hat{a_1}$...$\hat{a}_{N-1}$), Transformed polynomial coefficients

$OpC \leftarrow 0$
for $s < \log_R N$ **do**
  if `Addressing in NR` **then**
    $O \leftarrow \frac{N}{R^{s+2}}$
  else
    $O \leftarrow R^s$
  end if
  if $s < (\log_R N - 1)$ **then** ▷ Standard transformation stages
    for $i < O$ **do**
      for j `from 0 by` O `to` $\frac{N}{R}$ **do**
        $Ad \leftarrow i + j$ ▷ Generate coefficient address
        $TAd \leftarrow TwidGen(Ad, s, R, N)$ ▷ Generate t. address
        for $b < R$ **do**
          $Cf[b] \leftarrow Mem[b][Ad]$ ▷ Load coefficients from memory
        end for
        for $b < R - 1$ **do**
          $Tw[b] \leftarrow TMem[b][TAd]$ ▷ Load t. factors from memory
        end for
        $Res \leftarrow BFU(Cf, Tw, TInt, q, R)$ ▷ Perform BFU operations
        for $b < R$ **do**
          $Mem[OpC][Ad + O(b - OpC)] = Res[b]$ ▷ Write back
        end for
        $OpC \leftarrow OpC + 1 \pmod R$
      end for
    end for
  else
    for $i < \frac{N}{R}$ **do** ▷ Final transformation stage
      for $b < R$ **do**
        $Cf[b] \leftarrow Mem[b][i]$ ▷ Load coefficients from memory
      end for
      $Res \leftarrow BFU(Cf, [1, ..1], q, R)$ ▷ All t. factors are 1
      for $b < R$ **do**
        $Mem[b][i] = Res[b]$ ▷ Write back
      end for
    end for
  end if
end for
$Mem\_r \leftarrow Mem$ ▷ Return transformed polynomial

3.6.2 Result write-back

To ensure that the coefficients are always ordered in such a way that every operation addresses the same index in every memory bank, results of operations cannot be stored back in the source indices of the previous operation. Instead, they are stored in the source indices for the next operation.

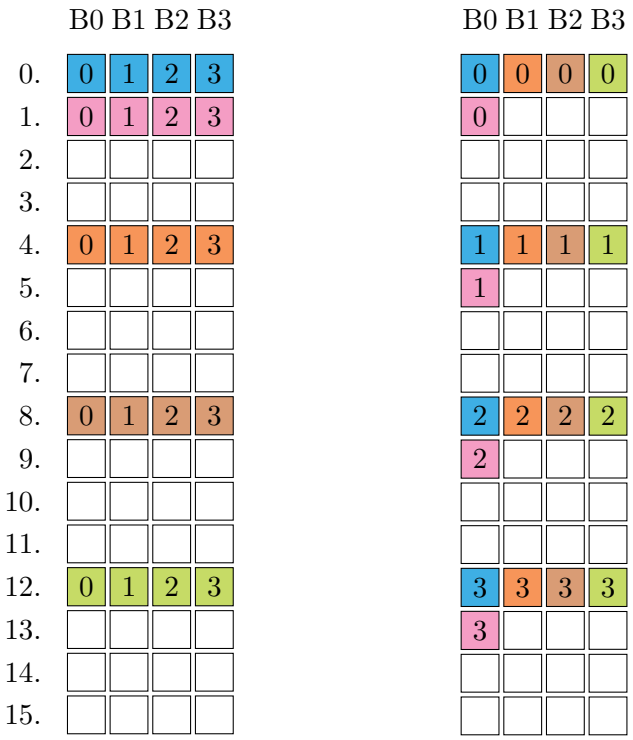

Figure 3.2: Memory locations of coefficients before (left) and after (right) the memory intercept

Figure 3.2 shows the memory layout of a system before coefficients are loaded and after they go through the memory intercept. It should be noted that while only five operations are shown, there are sixteen in total for this one stage and every subsequent operation follows the same access pattern.
While this does not represent any real system, it shows four memory banks, with each one holding sixteen coefficients. Each index points to four identically colored coefficients (arranged horizontally in the left image, one in each bank).
In this case, the order of loaded indices is (0, 4, 8, 12, 1...). Indices 0, 4, 8 and 12 represent one group.
Within one group, the coefficients from indices are written to the same locations in a memory bank, while that bank is determined by the position of the indices within the group (i.e., coefficients of an index in position i within any group will get written to the

bank i).
In the current example, the observed group is group 0, and the blue-colored index is at the address 0, therefore, this is the first index within group 0. This means that its coefficients will get written into the first memory bank (bank 0). The same process continues for the other indices in group 0, after which the second group begins being loaded (starting at address 1).
After determining the target bank, the algorithm will determine the addresses within that bank to distribute the coefficients from the current index.
Coefficient i within any index will be placed in the address of the i-th index within its own group.
This means that in the current example, with the first group being (0,4,8,12), the coefficients of index 0 get placed in the addresses (0,4,8,12) of bank 0.

3.7 Twiddle factor index generation

As mentioned previously, the twiddle factors are precomputed and written to memory in the initialization step. Afterwards, multiple transformations can take place with the same set of twiddle factors. Alternatively, they can be swapped out between each transformation, at the discretion of the user.
They will be stored in memory in $R-1$ banks and every multiplication operation will, in parallel, multiply $R-1$ partial BFU results with one twiddle factor from each of the banks. All of these twiddle factors, in a similar manner to coefficients, will be read from the same index.
Their order in memory will be as such:

$$\omega[i][j] = \omega_n^{(i+1)\cdot j} \tag{3.10}$$

where i represents the memory bank ($i = 0, ... \frac{R}{2} - 2$), j represents the address within that bank ($j = 0, ... \frac{N}{R} - 1$) and ω_n is the n-th primitive root of unity.
For a successful transformation, these twiddle factors must be loaded in a particular order. Algorithm 2 describes the process of generating twiddle factor indices for such an order of loads. The `BinaryReverseValue` algorithm reverses the binary representation of the input value.
To calculate the twiddle factor index, for DIF algorithms the coefficient index is multiplied by R for each stage of the operation, while for DIT algorithms the coefficient index is multiplied by $\frac{N}{R}$ to begin with, and then divided by R for each stage of the operation.
Precomputed twiddle factors addressed in such a way allow for efficient usage by the butterfly unit, whose algorithm is described in the following section.

3.8 Butterfly unit operations

All changes performed on the coefficients themselves happen in the butterfly unit (BFU). It is named after its characteristic criss-cross pattern of mathematical operations. The

Algorithm 2 Twiddle factor index generation

Input: Ad, Current coefficient index
Input: s, Current stage
Input: R, N
Output TAd, Current twiddle factor index

procedure TwidGen(Ad, s, R, N)
  if `DIF algorithm` **then**
    if `Addressing in RN` **then**
      $Ad \leftarrow$ `BinaryReverseValue`(Ad)
    end if
    $TAd \leftarrow Ad \cdot R^s \pmod{\frac{N}{R}}$
  else
    if `Addressing in NR` **then**
      $Ad \leftarrow$ `BinaryReverseValue`(Ad)
    end if
    $TAd \leftarrow Ad \cdot \frac{N}{R^{s+1}} \pmod{\frac{N}{R}}$
  end if
end procedure

mathematical operations performed by the butterfly unit will be called butterfly operations. The BFU takes a vector of R coefficients as an input and produces a vector of R modified coefficients as an output.
The DIF butterfly operations are given as:

$$\left(\begin{bmatrix} \omega_{0,0} & \cdot & \cdot & \cdot & \omega_{0,R-1} \\ \cdot & \cdot & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ \omega_{R-1,0} & & & & \omega_{R-1,R-1} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_{R-1} \end{bmatrix} \right) \odot \begin{bmatrix} 1 \\ \omega^1 \\ \cdot \\ \cdot \\ \cdot \\ \omega^{R-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_{R-1} \end{bmatrix} \tag{3.11}$$

where y_i are the outputs and x_i are the inputs of the butterfly unit, while ω_{ij} are the internal and ω^k external twiddle factors and $\odot$ denotes index-wise multiplication.
The DIT butterfly operations are given as:

$$\begin{bmatrix} \omega_{0,0} & \cdot & \cdot & \cdot & \omega_{0,R-1} \\ \cdot & \cdot & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ \omega_{R-1,0} & & & & \omega_{R-1,R-1} \end{bmatrix} \times \left(\begin{bmatrix} x_0 \\ x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_{R-1} \end{bmatrix} \odot \begin{bmatrix} 1 \\ \omega^1 \\ \cdot \\ \cdot \\ \cdot \\ \omega^{R-1} \end{bmatrix} \right) = \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_{R-1} \end{bmatrix} \tag{3.12}$$

where, again, y_i are the outputs and x_i are the inputs of the butterfly unit, while ω_{ij} are the internal and ω^k external twiddle factors and $\odot$ denotes index-wise multiplication.

The butterfly core unit performs the equivalent operation to a matrix-vector multiplication (inside the parentheses). This will be explored later in this chapter and can also be seen in Algorithm 2 within [CLW69] in its radix-8 configuration.
Butterfly operations are described by Algorithm 3, which is divided into the following sections:

1. Input reordering (lines 2-4).
   All inputs get reordered in RN algorithms.
   The reordering is an operation of inverting the binary representation of the index of the given element in the array of inputs. This way, if the index is four bits wide in binary, the element in input-position 1 decimal (0001 binary) will be placed in the input-position 8 (1000).
2. Pre-multiplication (lines 5-13).
   All inputs get pre-multiplied in DIT algorithms.
   The multiplication happens either with a combination of an integer multiplier and an integer modular reduction circuit (in NTT), or a complex floating point multiplier (in FFT). These operations are further explored in Section 4.2.4.
3. Core BFU unit (line 14).
   It will be explained further on.
   In short, it performs a matrix-vector portion of the butterfly operation, as given in Equations 3.11 and 3.12.
4. Post-multiplication (lines 5-23).
   All inputs get post-multiplied in DIF algorithms.
   Same as with point 3, the operations are explored in Section 4.2.4.
5. Output reordering (lines 24-28).
   All outputs get reordered in RN DIT algorithms.

The reordering at the input and the output of the butterfly unit adapts it to the functionality of the index generator and the memory intercept as well as the memory layout of the system. This particular combination of algorithm and memory access scheme is the only one to require this particular kind of adaptation.
The `BinaryReverseArray` algorithm reorders the elements of an array by inverting the binary representation of their index.
Another important attribute of the butterfly unit is its radix. The radix dictates how many coefficients are taken in as inputs and how many results are then produced as outputs. In addition to this, higher radix designs will need more twiddle factors per one operation, which necessitates more memory banks for their storage (and the storage of the coefficients themselves). The internal twiddle factors also rise in number with increasing radices (although they do this slower than the external twiddle factors, with the formula being $\frac{R}{2} - 1$. The rising radix also requires higher complexity in the control structures.

Algorithm 3 Butterfly unit operation

Input: $Cf[R]$=(c_0,c_1...c_{R-1}), Array of R coefficients
Input: $Tw[R-1]$=(ω_0,ω_1...ω_{R-2}), Array of $R-1$ twiddle factors
Input: $TInt[\frac{R}{2}-1]$=(ω_0,ω_1,...$\omega_{\frac{R}{2}-2}$), Array of $\frac{R}{2}-1$ BFU internal t. factors
Input: q, R
Output $Res[R]$=(r_0,r_1...r_{R-1}), Array of R results

1: **procedure** BFU($Cf, Tw, TInt, q, R$)
2: **if** `Addressing in RN` **then**
3: $Cf \leftarrow$ `BinaryReverseArray`(Cf)
4: **end if**
5: **if** `DIT algorithm` **then**
6: **for** i `from 1 by 1 to` R **do**
7: **if** `NTT algorithm` **then**
8: $Cf[i] \leftarrow Cf[i] \cdot Tw[i-1] \pmod q$
9: **else**
10: $Cf[i] \leftarrow Cf[i] \cdot Tw[i-1]$
11: **end if**
12: **end for**
13: **end if**
14: $Res \leftarrow$ `BFUCoreUnit`($Cf, Tint, q, R$)
15: **if** `DIF algorithm` **then**
16: **for** i `from 1 by 1 to` R **do**
17: **if** `NTT algorithm` **then**
18: $Cf[i] \leftarrow Cf[i] \cdot Tw[i-1] \pmod q$
19: **else**
20: $Cf[i] \leftarrow Cf[i] \cdot Tw[i-1]$
21: **end if**
22: **end for**
23: **else**
24: **if** `Addressing in NR` **then**
25: $Res \leftarrow$ `BinaryReverseArray`(Res)
26: **end if**
27: **end if**
28: **end procedure**

As stated before, the core of the butterfly unit performs a matrix-vector multiplication, however, by using optimizations with regard to the generation, storage and multiplication with twiddle factors, the number of multiplications is significantly reduced.The multiplication is given as:

$$y = W_R \times x \tag{3.13}$$

where y is the result, x is the vector of core unit inputs and W_R is the core unit's W matrix.

The W matrix is of size $R \times R$ and an individual element ω_{ij} is given as:

$$\omega_{ij} = \omega_n^{\frac{i \cdot j \cdot N}{R}} \; i, j = 0, 1...R-1 \tag{3.14}$$

where ω_n is the n-th primitive root of unity ([CLW69]).
This means that the smallest core unit's W matrix is given as:

$$W^2 = \begin{bmatrix} \omega_n^0 & \omega_n^0 \\ \omega_n^0 & \omega_n^{\frac{N}{2}} \end{bmatrix} = \begin{bmatrix} \omega_n^0 & \omega_n^0 \\ \omega_n^0 & \omega_n^{0+\frac{N}{2}} \end{bmatrix} = \begin{bmatrix} \omega_n^0 & \omega_n^0 \\ \omega_n^0 & -\omega_n^0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.15}$$

Therefore, by utilizing the symmetry of the twiddle factors, the multiplication by $\omega_n^{\frac{N}{2}}$ followed by addition was turned into a simple subtraction.
The calculations performed by the radix-2 core unit are now:

$$FFT : y_0 = x_0 + x_1 \wedge y_1 = y_0 - y_1 \tag{3.16}$$

$$NTT : y_0 = x_0 + x_1 \pmod q \wedge y_1 = y_0 - y_1 \pmod q \tag{3.17}$$

This set of equations would leave the NTT case with the need for two integer reduction units, however, due to the fact that every multiplication in the system is followed by a reduction and that all of the input coefficients and twiddle factors are already reduced by q, the bounds of the results are:

$$y_0 \in [0 : 2 \cdot (q-2)] \wedge y_1 \in [-q+1 : q-1] \tag{3.18}$$

This means that a reduction of the first equation would be equivalent to a subtraction by q and the reduction of the second equation would be equivalent to an addition with q.
To determine whether or not a reduced version of the result should be output, it is possible to generate both versions for both of the operations and then check the MSB of the reduced addition operation and the MSB of the non-reduced subtraction operation. If one of them is 1, the other version of the result should be used.
A butterfly unit implementing the described algorithm performs the required operations with the lowest amount of resources used and with high performance.

4 Hardware implementation

The following chapter will go into detail on the implementation of previously mentioned algorithms. It goes over the high-level block design with designated blocks that perform individual functions and implement individual algorithms from Section 3.

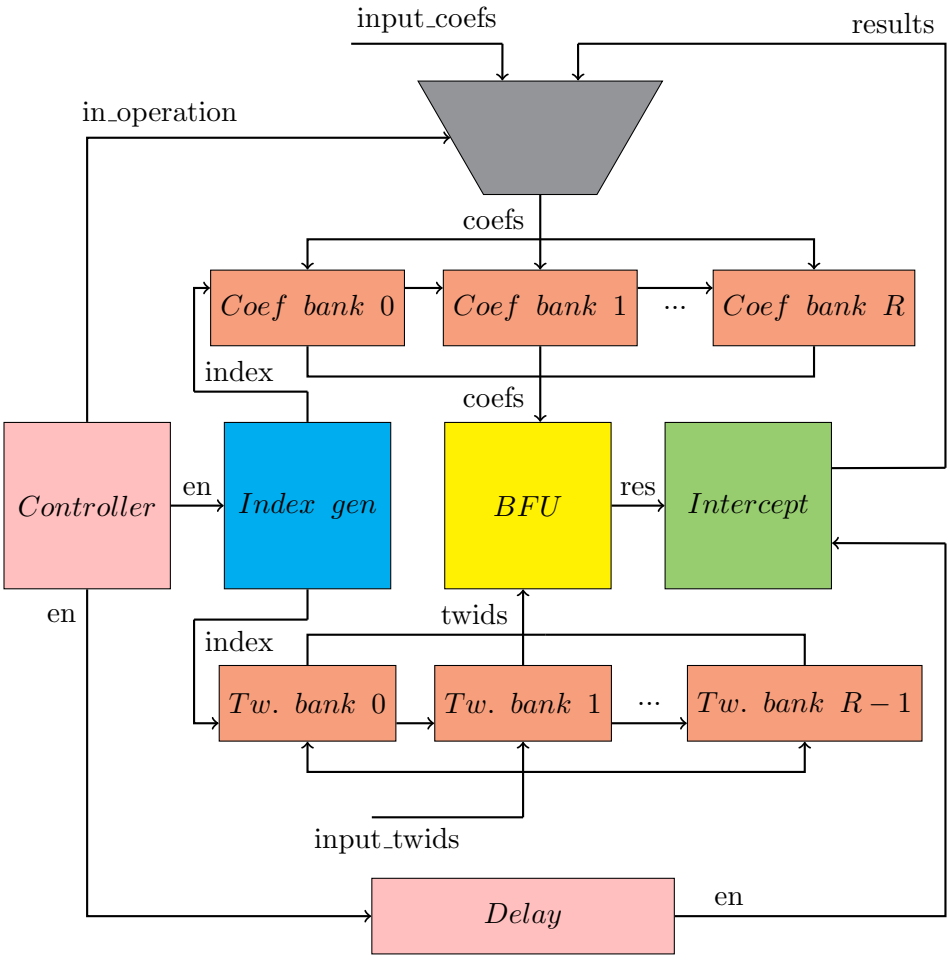

Figure 4.1: Block diagram of the system

4.1 Design block diagram

The aforementioned algorithms were implemented in the form of a hardware accelerator, meant to attach to a CPU and assist in performing the transformation operations. The block diagram of this design can be seen in Figure 4.1.
It is made up of the following elements:

1. Controller unit.
2. Index generator.
3. Butterfly unit.
4. Memory intercept.
5. Coefficient and twiddle factor memories.

The controller has the task of enabling all of the other sub-modules, performing IO operations and communicating with the outside world. The index generator generates indices which act as addresses for accessing coefficients and twiddle factors. The butterfly unit performs all mathematical operations pertaining to the transformations in the system. The memory intercept writes back all BFU results to their appropriate locations in memory. Lastly, the coefficient and twiddle factor memory banks will store input, intermediate and result coefficients and twiddle banks, respectively.
The design is highly parametrizable, and the parameters associated with it can change its operation significantly.
Some of these are **FFT_NTT**, which switches the design between performing FFT and NTT operations, **DIF_DIT**, which switches the system between using DIF and DIT algorithms, **NR_RN** switches the system between using NR and RN memory access schemes, **RADIX**, which sets the radix of the system, **DEGREE**, which sets the degree of the polynomial and **MOD**, which sets the default modulo for NTT operations.
While most parameter configurations can be freely selected, there are some restrictions in terms of the degree and radix of the system. Certain combinations of these two parameters cause 'write after read' (WAR) hazards. This can happen if the system tries to access a particular memory address before the result of a previous operation had the time to overwrite the contents. As a result, the next operation in line will receive incorrect inputs. To avoid this hazard, the delay between the read and write operations (in clock cycles) has to be longer than the shortest interval between reading from the same memory address twice (which is $\frac{N}{R^2}$ clock cycles).
From this, it can be concluded that each value of R has its own minimal value of N. They can be seen in Table 4.1.

Rad2 NTT	Rad2FFT	Rad4NTT	Rad4FFT	Rad8NTT	Rad8FFT
128	128	1024	1024	4096	32768

Table 4.1: Minimum allowed degrees for various radix values

Further text will go into more details on the implementation of individual sub-modules of the system, going over their place in the system, the function they perform, the connections they have with other sub-modules, any design decisions, or special considerations

in their construction and references to the algorithms they are based on.

4.2 System sub-modules

Each of the sub-modules will be described in one chapter, with an additional chapter dedicated to the explanation of the wrapper meant to handle communications with a master CPU.

4.2.1 Controller

The controller is the central sub-module of the system; it is connected to all other sub-modules and manages their operation to some extent.
It handles all memory operations, enabling of other modules, storage of internal twiddle factors and reduction modulo, as well as all communication with the outside world.
At its heart is a finite state machine with 4 states, as seen in Figure 4.2.

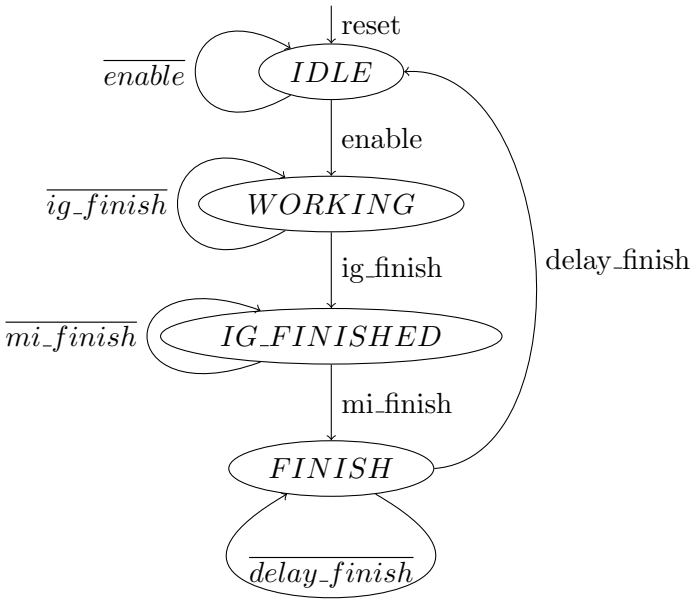

Figure 4.2: Finite state machine of the controller

The system will start in the **IDLE** state and stay there until the enable signal is given from the outside.
At that point it will switch to **WORKING**. Here, the index generator is enabled instantly and the memory intercept after a delay. This delay serves to allow the butterfly unit to perform the first set of transformation operations, while at the same time delaying the end of the overall transformation in order to allow the final operations to be performed. The delay length is always equal to the number of clock cycles needed

for one BFU operation. To exit the **WORKING** state, the signal that the index generator has finished its work must be received, after which point the system enters the **IG_FINISHED** state. This state is meant to serve as a buffer in order to allow the memory intercept to finish its work as well.
Here, all enable signals are released, and the controller will wait for the confirmation that the memory intercept is finished. This brings the system into the **FINISH** state.
In this state a counter is used to time the FIFO queues inside of the memory intercept emptying. Once that happens, the system returns to the **IDLE** state.
For the one clock cycle in which the last change occurs, the outside world is notified of the operation being completed.
All in/out operations are disabled in every state besides **IDLE**, and in these cases the controller acts as a middleman between the memory banks and the BFU.
Another one of the controller's duties is to manage the inter-module communications. This includes memory read/write operations as well as system control signals. All of the relevant signals can be seen in the Appendix Tables 7.1 and 7.2.
The control signals will be generated by the wrapper (or more broadly, by any actor in the outside world) and the controller will act on them. They encompass the already mentioned enable signal, which will start the transformation operations and the finish signal, which is meant to notify the wrapper (or any actor in the outside world) of the conclusion of the transformation operation.
The memory IO signals are either generated directly by some outside agent, or generated by that outside agent and then modified and translated by the wrapper. These signals entail enabling or disabling the memory IO operations entirely, targeting one of the memory bank groups (coefficients or twiddle factors), targeting a particular bank within that group or targeting an address within that bank. In addition to that, both writes and reads are available to the coefficient banks, while the twiddle banks, internal twiddle registers and the reduction modulo can only be written to. The memory IO communications also implement two lines of communications, one for transactions originating in the system (treated as reads by the wrapper) and ones originating in the outside world (treated as writes by the wrapper).
The controller, implemented in such a way, will uphold the necessary timings for the correct execution of transformation operations, while at the same time allowing for easy and fast communications with other modules (namely, a master CPU).

4.2.2 Index generator

The index generator is the first submodule (besides the controller) to be activated during an operation. It is connected directly to the controller and the twiddle factor memory banks as well as indirectly to the coefficient memory banks (through the controller).
It handles the address generation for coefficient and twiddle factor memory and implements the algorithm given in Section 3.6.1. From the functional perspective, its main constituents are two counters, named 'degree counter' and 'offset counter' respectively, an index adder and a twiddle index generator, which is a sub-sub-module.
The degree counter increments by the value of the offset each clock cycle and keeps track

of the next index within the current group to be generated. It counts to the maximum of the highest index value, which is $\frac{N}{R}$. On the other hand, the offset counter increments by one each clock cycle and determines the group to which the index should belong. It counts to the current value of offset.
The generation of indices is a simple addition of these two counters.
As can be seen in Algorithm 1, the offset is dependent on one or more constant values and the current stage. In accordance with this and to avoid using multiple registers to store every possible offset value, it will be updated every time the stage changes. Additionally, considering that the multiplication and division operations are much slower than simple bit manipulations, right or left shifts will be used with operands determined by all of the values on the right-hand side of those operations.
The twiddle index generator is a very simple module that handles the generation of all twiddle factor indices when provided with the coefficient index value and the current stage. The multiplications and divisions are again handled via left and right shifts. On the other hand, since the modulo in both of the reductions in Algorithm 2 is a power of two, the reduction can be performed by simply making the bit-width of the result register one smaller than it would take to capture the modulo.
This index generator will facilitate the reading of coefficients and twiddle factors in such a way that will allow the butterfly units to perform the transformation operations and forward the results to the memory intercept, which is the next sub-module to be discussed.

4.2.3 Memory intercept

The memory intercept is the last submodule to be activated during an operation. It is enabled by the controller, via the system delay, and it is connected directly to the BFU and indirectly to the coefficient memory banks through the controller.
It handles the writing on all BFU results back into memory and implements the algorithm given in Section 3.6.2.
At its heart are two counters that act in the same way as the ones in the index generator. However the offset counter increments only every radix clock cycles. This is due to the fact that the write addresses only change once every memory bank has been written to. The 'operations counter' that handles this behavior also acts as the pointer that dictates which bank should be written to next.
It should be noted is that the value of the offset is always radix times greater than in the index generator; this compensates for the radix many coefficients written per operation.
Analogous to the index generator, the base address of the write operation is generated via an addition of the two counters. This base address represents the address that the very first result of the operation gets written to. All of the other addresses are generated by adding it to a so-called 'offset array'. The values in the offset array are:

$$OA[i] = \frac{O \cdot i}{R}, i = 0, 1, ..., R \tag{4.1}$$

They correspond to the distance between index 0 and indices $1, ...R$, therefore adding them to the base address produces addresses for each write in one BFU operation.

This multiplication is handled by the second sub-sub-module of the design, the 'array_generator'. It makes use of the fact that every offset value is given as $2^k, k \in \mathbb{N}$. Thus, a multiplication with i is a simple series of left shifts and logical OR operations. This avoids the need for an integer multiplier.
The final stage of the transformation will, unlike previous ones, simply write coefficients to their address of origin. This places them in an advantageous position for future inverse transformations. However, to accomplish this, the offset is set to 1 for this stage, regardless of any other calculations.
Further, since the memory can only accept one write operation per clock cycle, a FIFO system will be used to pace these operations out over a longer time. Since each operation that targets a particular bank occurs once every radix clock cycles and each operation creates radix results to be written, there is exactly enough time to write every coefficient during standard stages.
The first operation of the special stage will overlap with the last operation of the final standard stage, which targets the last bank, therefore, the special stage writes will have to be delayed by exactly radix clock cycles.
This is accomplished with a standard FIFO with one input and one output and radix registers. On the other hand, standard operations make use of a modified FIFO with radix inputs, one output and radix registers. This allows them to begin outputting at the earliest possible time.
The memory intercept allows for the correct implementation of the transformation algorithm and keeps the memory footprint of the design minimal.

4.2.4 Butterfly unit

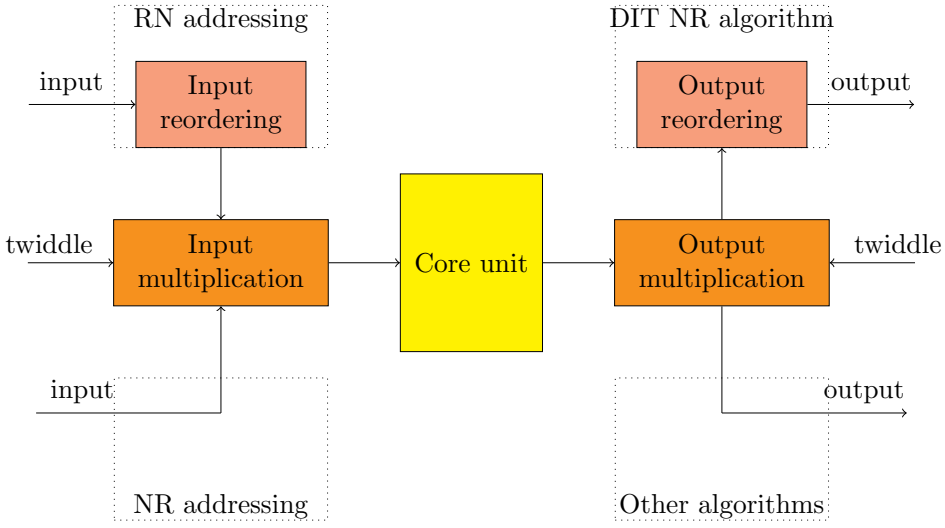

Figure 4.3: Block diagram of the butterfly unit

The butterfly unit sits between the memory banks and the memory intercept. It is directly connected to all of them and is the second submodule to be activated during runtime. It performs all of the mathematical operations on the coefficients in the system and implements the algorithm given in Section 3.8.

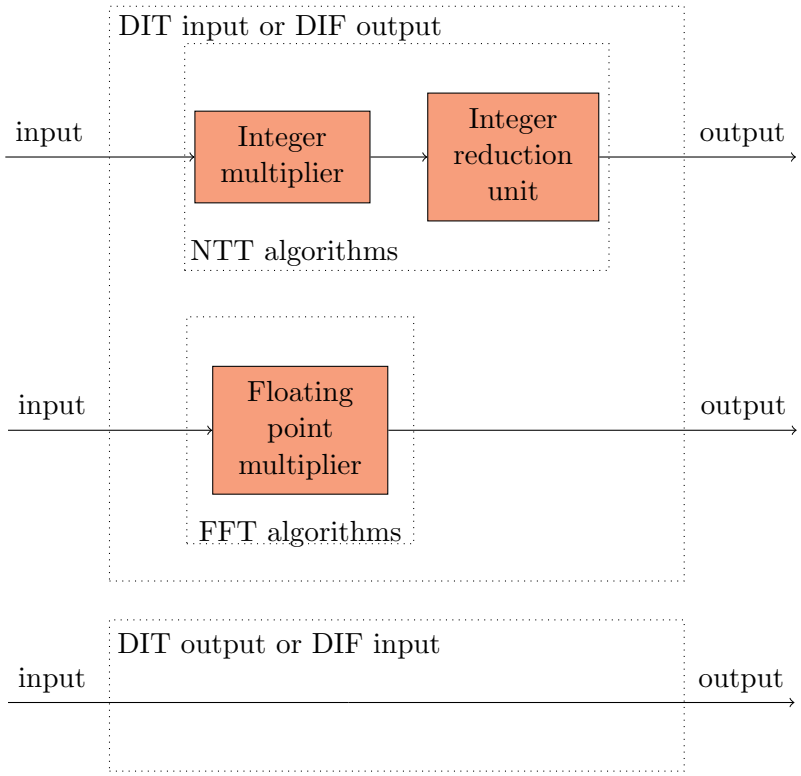

Figure 4.4: Block diagram of the multipliers

The butterfly unit can be seen in Figure 4.3. It is made up of one core unit and additional multipliers and reordering units. The core unit is responsible for handling the matrix-vector multiplication, which, using several previously mentioned optimizations gets severely cut down into a shorter sequence of additions, subtractions and multiplications. Next, the multipliers handle the index-wise multiplications seen in Equations 3.11 and 3.12. The reordering units serve to align the operations of the butterfly unit with the operations of the index generator and the memory intercept.
The multipliers and reordering units will be touched on in this chapter (and can be seen in Figure 4.4), while the core butterfly unit is explored in more detail in Section 4.2.5.
For DIF algorithms, only the output multiplier is instantiated, while for DIT algorithms the same is true of the input multiplier. At the same time, in NTT algorithms, every integer multiplier is followed by a modular integer reduction unit, while in FFT algorithms, only the complex floating-point multiplier is used.
All modular integer reduction units in the design use the Montgomery reduction algo-

rithm with a modulo bit-width of 64 bits. The Montgomery reduction goes through a number of steps and reduces the given input by a certain word size, until the remainder has the same size as the modulus. Then, a simple check is performed to determine if an additional subtraction operation is necessary. The algorithm and implementation of the Montgomery reduction algorithm can be seen in [HMR23].
On the other hand, the complex floating-point multiplier is made up of four standard floating-point multipliers and two floating-point adders.
The reordering units take an array as input and return it with elements reordered via binary reversal of their indices. This is the equivalent of the `BinaryReverseArray` algorithm.
RN memory access schemes make use only of the input reordering unit, while NR access schemes make use only of the output reordering unit and only if the DIT algorithm is in effect.
As stated earlier, the core unit performs an equivalent to a matrix-vector multiplication. However, by using optimizations with regard to the generation, storage and multiplication with twiddle factors, the number of multiplications is significantly reduced.

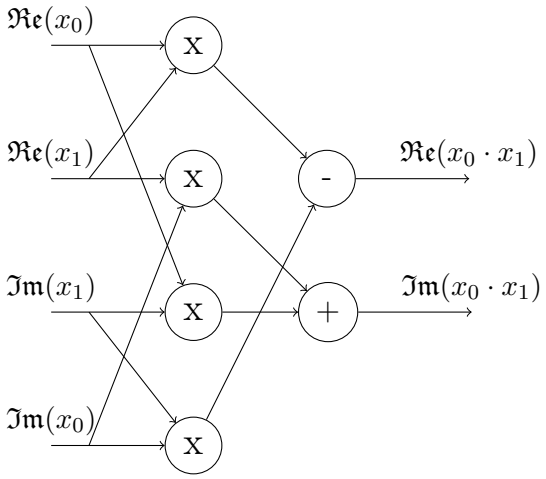

Figure 4.5: Block diagram of the complex floating point multiplier

4.2.5 Core of the butterfly unit

At the core of the butterfly unit is a sub-module dedicated to performing the BFU matrix-vector multiplication. This submodule is also called the radix-R NTT/FFT core unit. Its structure is highly dependent on the radix parameter and the different settings produce drastically different units.
The building block of any butterfly core unit is the radix-2 unit. The NTT version can be seen in Figure 4.6. It showcases the optimization, which eliminates two modular reduction units by adding and subtracting q from the two results. The FFT version of the core unit does not need such an optimization, since it only uses one complex floating-point adder and one complex floating-point subtraction unit, and it can be seen

in Figure 4.7.

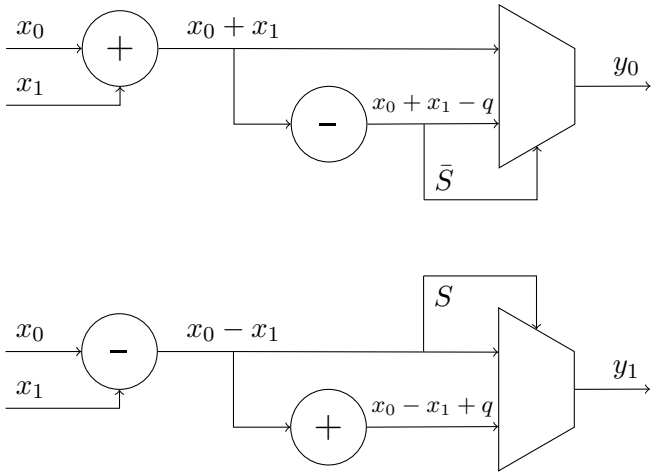

Figure 4.6: Block diagram of the radix-2 NTT core unit

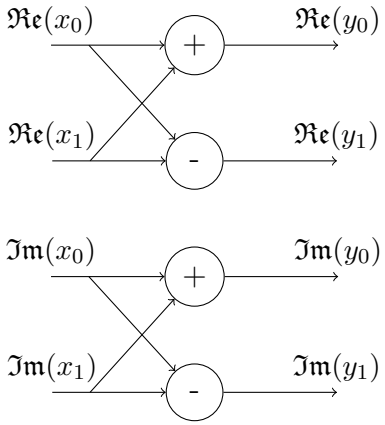

Figure 4.7: Block diagram of the radix-2 FFT core unit

To construct higher radix butterfly core units (where $R = 2^k, k \in \mathbb{N}$) the following steps are taken. First two core radix $\frac{R}{2}$ units are instantiated. The inputs for the first unit are all $x_i, i = 2k$ and the inputs for the second unit are all $x_i, i = 2k + 1$. For every output of the second unit $y_i, i > 0$ a multiplier is attached, which calculates:

$$NTT : \breve{y}_i = y_i \cdot \omega_{i-1} \pmod q \vee FFT : \breve{y}_i = y_i \cdot \omega_{i-1} \tag{4.2}$$

where ω_i are internal twiddle factors.
For every other output:

$$\breve{y}_i = y_i \tag{4.3}$$

Next $\frac{R}{2}$ radix-2 core units are instantiated, and the inputs of the i-th radix-2 unit are:

$$x_{i0} = \breve{y}_{0i} \wedge x_{i1} = \breve{y}_{1i} \tag{4.4}$$

The outputs are arranged in such a way that the first $\frac{R}{2}$ are y_0 and the last $\frac{R}{2}$ are y_1 outputs of the radix-2 units in order.
Finally, if the first two core radix $\frac{R}{2}$ units require internal twiddle factors of their own, they are chosen such that the matrix-vector multiplication formula is maintained (also keeping in mind periodicity and symmetry of twiddle factors). The easiest way to do this is to connect, in order, all odd-indexed internal twiddle factors from the radix-R core unit to all internal twiddle factors of the radix-$\frac{R}{2}$ core unit.

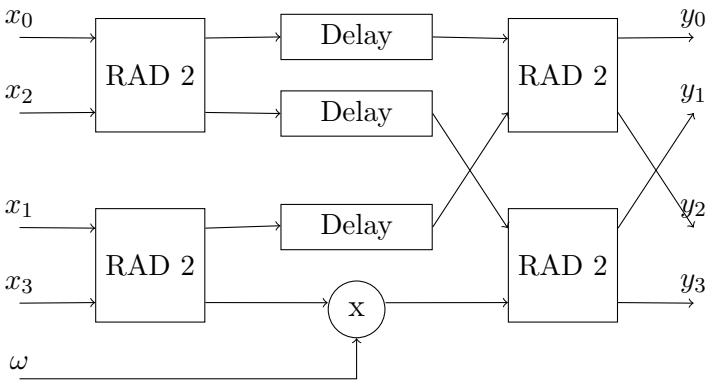

Figure 4.8: Example of constructing a radix-4 core unit

In the example in Figure 4.8 the multiplier is either an integer multiplier and integer modular reduction unit combination, or a complex floating point multiplier, depending on whether the design is NTT of FFT, respectively. The delay modules serve to synchronize the intermediate products that do not need multiplication with the ones that do. Their delay is equal in clock cycles to the complete delay of the multiplication unit in clock cycles.
Any higher radix unit, where $R = 2^k, k \in \mathbb{N}$, can be constructed recursively by repeating this procedure one step at a time, where $R_{new} = R_{old} \cdot 2$.
For FFT designs, the internal twiddle factors should have their real and imaginary parts reversed, which can be done at runtime, or at twiddle factor generation, currently the latter solution is implemented.

4.2.6 Recursive core unit

An alternative to the already proposed core butterfly unit is the so called 'Recursive core unit'. It is enabled by setting the **REC_BFU** parameter to 1.
Its name is derived from the fact that it recursively instantiates itself in order to construct a core unit of any radix. This radix is given as a parameter.

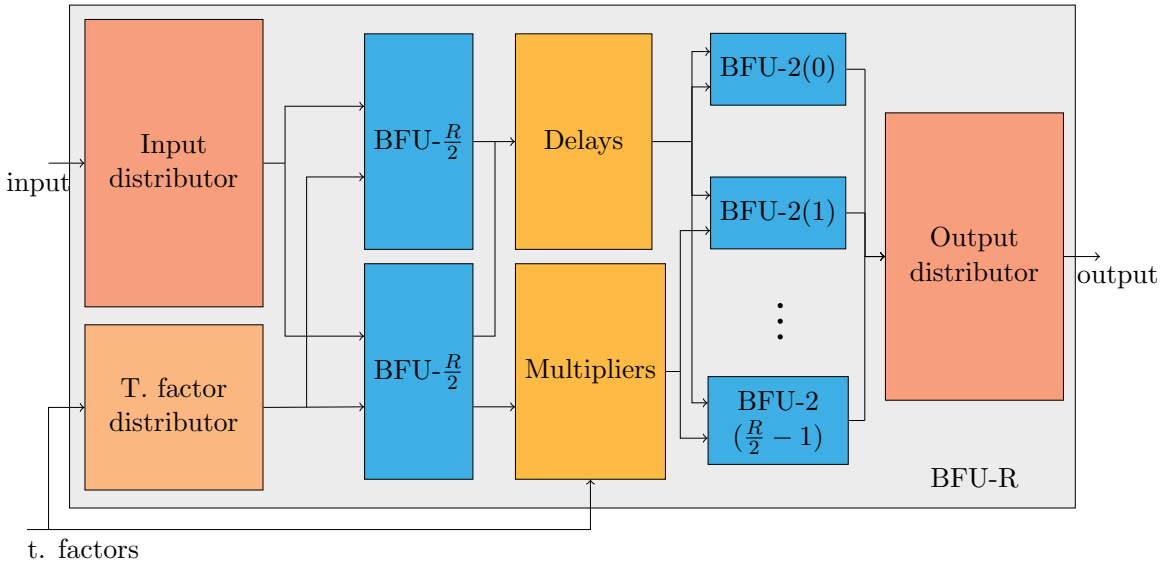

Figure 4.9: Block diagram of the recursive core unit

The construction of this module can be seen in Figure 4.9. It is made up of the input, twiddle factor and output distributors, delays, multipliers and two types of lower-radix core units.
To accomplish the task of processing R coefficients per operation, this sub-module instantiates itself (hence the name), with the first set of lower-radix core sub-units being just that - this very unit, with the radix parameter set to $\frac{R}{2}$. They are marked with 'BFU-$\frac{R}{2}$'.
These units need coefficient and twiddle factor inputs. To this end, two distributors were designed. The input distributor will provide the first lower-radix sub-module with inputs of the original module, with indices corresponding to $i = 2 \cdot k, k = 0, 1...$ and the second one with indices $i = 2 \cdot k + 1, k = 0, 1....$ This way the first radix-$\frac{R}{2}$ unit operates on even and the second one on odd index inputs. The twiddle factor distributors will provide these sub-modules with appropriate internal twiddle factors. These twiddle factors will, for every radix, consist of all of the odd-indexed twiddle factors of the core unit one radix above.
As the radix-2 unit does not use internal twiddle factors, a slight optimization can be made, where if the radix of the recursive unit is 4, then the lower-radix unit is not instantiated, but instead the original radix-2 unit is used. Alongside this, the internal twiddle factor arrays are not created and the corresponding wires are not instantiated. Alternatively, this slight optimization can be omitted and a special case made when the radix is equal to 2 which generates the standard butterfly unit operations as described in Section 4.2.5.
All of the outputs of the second lower-radix sub-unit, bar the one with index 0 are then multiplied with internal twiddle factors as described previously, while the one with index 0 and every output of the first lower-radix sub-unit are delayed, so that they will

be available at the same time as the multiplied values. All of these values are called post-multiplication values.
Next, $\frac{R}{2}$ radix-2 sub-units are used, here marked as BFU-2(0) to BFU-2($\frac{R}{2}$-1). The numbers in the parentheses can be considered their indices. Inputs 0 and 1 of the radix-2 sub-unit of index i are the post-multiplication values corresponding with the outputs i of the first and second radix-$\frac{R}{2}$ unit, respectively.
Lastly, the output distributor will arrange the outputs of these radix-2 sub-units such that the first $\frac{R}{2}$ are the outputs index-0 of the radix-2 units and the next $\frac{R}{2}$ are their outputs index-1.

4.2.7 System wrapper and memory

The final submodules of the system are the memory banks. They are connected to the controller, the index generator and the BFU. Their purpose is to store the polynomial coefficients and the twiddle factors.
The system contains $2 \cdot R - 1$ memory banks, where the first R contain coefficients, while the last $R - 1$ contain twiddle factors. The depth of these banks is $\frac{N}{R}$, while the width is parametrizable, though set to 64 as a default. In cases when the FFT algorithm is selected, the width of every bank is doubled to accommodate both the real and imaginary components of every number. In such a case, the lower half of all of the bits stores the real component, while the higher half stores the imaginary component.
In order to connect the system to a main CPU, a wrapper is necessary. Such a wrapper extends around the entire system, while also containing multiple submodules of its own. Its purpose is to translate the memory IO and control signals coming from the CPU and oversee the functioning of the transformation system.
The makeup and position of the system wrapper can be seen in Figure 4.10. It is made up of the transformation system, as well as memory IO and control systems.
It is expected to be connected to a main CPU via some form of a bus, in this case an AXI bus with the corresponding interconnect. To convert from AXI to a more general form of communication, a translation layer is expected, as well as some form of memory to hold commands (program memory).
The memory IO system handles all communications with the outside world. It will translate any address into signals corresponding to Table 7.2. Therefore, 'mem_addr' is equal to the lowest $\log_2 \frac{N}{R}$ bits, 'mem_bank' is the next lowest $\log_2 R$ and 'mem_select' is one bit after that. Further, 'mem_mode' is either the read/write signal, or set to high if one of the valid twiddle factor addresses is targeted. The 'mem_enable' is set to high if the system is not in operation and a legal write operation is attempted, or whenever the coefficients are targeted (this permits reading as well as writing, while twiddle factors have no such concern, as they will never be read). In cases when the system is running FFT algorithms, it will have to receive both the real and the imaginary parts of every coefficient and twiddle factor. Therefore, in those cases, an additional check is performed. All reads proceed as normal, but every write only happens in pairs (i.e., one real component and the corresponding imaginary component), which means that every write to an even address is saved, only to be written whenever a write to

an odd address is detected. The written data is then the concatenation of these two values and the 'mem_enable' signal can go high, while the write address is the address of the real part. This means that any programmer should make sure that all numbers are written as a complete package (real component, then immediately the imaginary component).
The control system drives the control signals in Table 7.1 and generates the signal, which tells the memory IO system whether the transformation system is in operation or not. It is connected to the program memory and receives commands from it.
If the command fetched from the program memory is the predefined 'start' command and the system is not in operation, then the 'start' signal from Table 7.1 will be driven high for one clock cycle. When the system is finished with the transformation, the 'finished' signal is driven high for one clock cycle. It is suggested to dispatch all commands in pairs, where one 'start' command is accompanied with any other command. This is due to the fact that the memory will continuously return the latest command if all of them are executed, therefore, the system will restart the transformation before the results could be collected. The second command will act as a 'stop' to the first ones 'start' instruction.

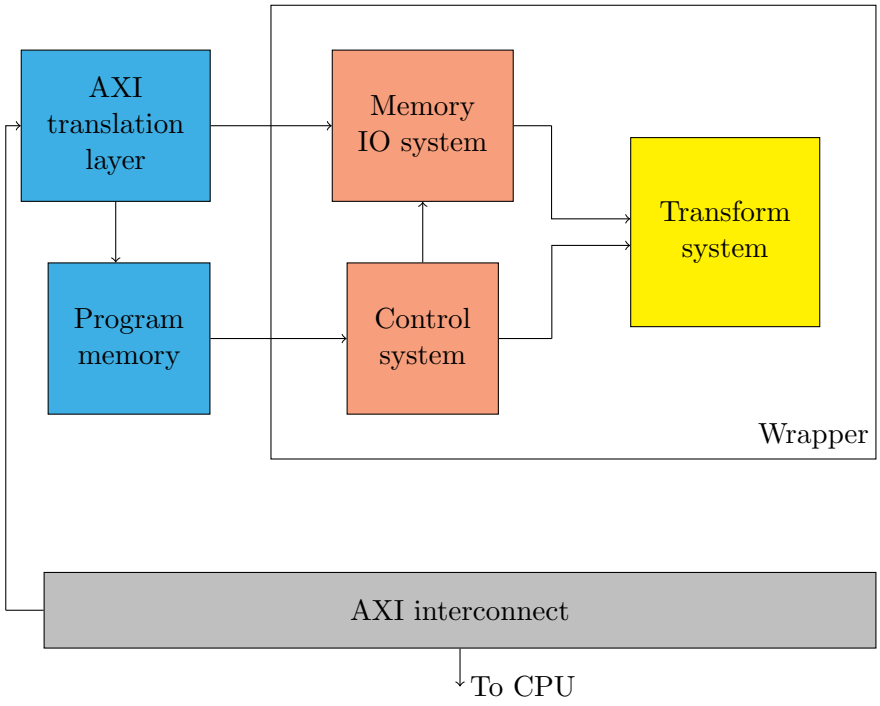

Figure 4.10: System wrapper and its position within the greater system

5 Results

Mem Access	Radix	Degree	LUT/FF/ DSP/BRAM	Power (W)	CC to finish	Latency (μs)	ATP (k)
NR	2	128	1844/2173/ 20/3	0.071	471	4.71	22.4
RN	2	128	1842/2181/ 20/3	0.07	471	4.71	22.3
NR	4	1024	7230/9147/ 80/7	0.295	1323	13.23	229.4
RN	4	1024	7223/9141/ 80/7	0.294	1323	13.23	229.3
NR	2	4096	2018/2253/ 20/12	0.089	24599	245.99	1872.7
RN	2	4096	2023/2256/ 20/12	0.088	24599	245.99	1874
NR	4	4096	7169/9187/ 80/14	0.307	6187	61.87	1198.1
RN	4	4096	7172/9186/ 80/14	0.311	6187	61.87	1198.4
NR	8	4096	28355/28352/ 216/15	1.085	2113	21.13	1149.6
RN	8	4096	28402/28319/ 216/15	1.098	2113	21.13	1150.6
NR	2	32768	3366/2251/ 20/65	0.2	245783	2457.83	61450
RN	2	32768	3230/2254/ 20/65	0.202	245783	2457.83	61116
NR	8	32768	20950/27969/ 162/61.5	0.731	20545	205.45	11423
RN	8	32768	20542/27969/ 162/61.5	0.728	20545	205.45	11339

Table 5.1: Implementation results for NTT DIF

All implementations are done on the PYNQ-Z2 platform, unless stated otherwise. The frequency of the clock is set to 100 MHz. The selected degree values represent the minimal values for various radices and several values compatible with multiple radices

in order to showcase any differences between them.
Table 5.1 presents the implementation results for several parameter combinations of the NTT DIF algorithm. The results are quite expected, with NR and RN variation of every setup having identical latencies and nearly identical resource utilization and power consumption.

Mem Access	Radix	Degree	LUT/FF/ DSP/BRAM	Power (W)	CC to finish	Latency (μs)	ATP (k)
NR	2	128	1873/2182/ 20/3	0.077	471	4.71	22.5
RN	2	128	1878/2182/ 20/3	0.081	471	4.71	22.5
NR	4	1024	7049/8950/ 80/7	0.319	1323	13.23	226.9
RN	4	1024	7043/8948/ 80/7	0.327	1323	13.23	226.8
NR	2	4096	2067/2294/ 20/12	0.093	24599	245.99	1886.0
RN	2	4096	2017/2257/ 20/12	0.092	24599	245.99	1873.7
NR	4	4096	7429/8997/ 80/14	0.327	6187	61.87	1214.4
RN	4	4096	7456/9036/ 80/14	0.322	6187	61.78	1214.3
NR	8	4096	27583/29707/ 168/15	1.05	2113	21.13	1032.9
RN	8	4096	27646/29707/ 168/15	1.06	2113	21.13	1034.2
NR	2	32768	2289/2233/ 20/65	0.205	245783	2457.83	58469
RN	2	32768	2350/2236/ 20/65	0.205	245783	2457.83	58619
NR	8	32768	21907/28013/ 162/61.5	0.799	20545	205.45	11620
RN	8	32768	21931/28021/ 162/61.5	0.797	20545	205.45	11625

Table 5.2: Implementation results for NTT DIT

Table 5.2 presents the implementation results for several parameter combinations of the NTT DIT algorithm. It can be seen that the results are identical
to NTT DIF (for latency), or very close to them (for resource utilization and power).
One standout variation for both of these tables is the radix-8 degree 4096, with noticeably higher resource utilization and power consumption. This is due to the fact that different

implementation settings were used for it. The Vivado implementation engine failed to close the timing with default settings; therefore, the 'performance explore' preset was used. This could have affected the results.

Mem Access	Radix	Degree	LUT/FF/ DSP/BRAM	Power (W)	CC to finish	Latency (μs)	ATP (k)
NR	2	128	7265/6583/ 44/5	0.156	475	4.75	62.5
RN	2	128	7208/6586/ 44/5	0.163	475	4.75	62.3
NR	4	1024	27826/26269/ 176/10	0.775	1333	13.33	645.5
RN	4	1024	27821/26269/ 176/10	0.777	1333	13.33	645.5
NR	2	4096	7305/6640/ 44/16.5	0.26	24603	246.03	4097.6
RN	2	4096	7307/6640/ 44/16.5	0.264	24603	246.03	4098.1
NR	4	4096	27665/26373/ 176/18	0.834	6197	61.97	3139.7
RN	4	4096	27661/26372/ 176/18	0.832	6197	61.97	3139.5
NR	2	32768	7451/6653/ 44/129	0.497	245787	2457.87	124247
RN	2	32768	7422/6653/ 44/129	0.494	245787	2457.87	124176
NR	8	32768	86492/80628/ 528/217.5	2.063	20560	205.6	42053
RN	8	32768	86503/80641/ 528/217.5	2.034	20560	205.6	42056

Table 5.3: Implementation results for FFT DIF

Table 5.3 presents the implementation results for several parameter combinations of the FFT DIF algorithm. The results are, expectedly, quite different from the NTT implementations when it comes to resource utilization and power consumption. For example, the radix-2 DIF NR combination uses ≈ 4 times as many LUT elements and ≈ 3 times as many FF elements. The power consumption is also ≈ 2 times greater. This discrepancy is mostly down to the fact that the floating-point multiplication and addition/subtraction units are much more demanding with regard to both of these benchmarks than their integer equivalents. This holds, even with the consideration of every integer multiplication circuit being followed by a modular reduction unit. Additionally, the FFT implementations make use of complex numbers, which are not only twice as wide, therefore causing twice as much switching across all internal communication channels than in the NTT

system, but also require two separate floating-point addition/subtraction units per addition/subtraction operation, respectively, and four floating-point multipliers, with two additional adders per multiplication operation.

Mem Access	Radix	Degree	LUT/FF/ DSP/BRAM	Power (W)	CC to finish	Latency (μs)	ATP (k)
NR	2	128	7183/6471/ 44/5	0.172	475	4.75	62.1
RN	2	128	7178/6474/ 44/5	0.172	475	4.75	62.1
NR	4	1024	27629/26221/ 176/10	0.812	1333	13.33	642.9
RN	4	1024	27631/26221/ 176/10	0.812	1333	13.33	642.9
NR	2	4096	7331/6600/ 44/16.5	0.268	24603	246.03	4104.0
RN	2	4096	7331/6600/ 44/16.5	0.27	24603	246.03	4104.0
NR	4	4096	27364/26261/ 176/18	0.859	6197	61.97	3121.0
RN	4	4096	27367/26260/ 176/18	0.854	6197	61.97	3121.2
NR	2	32768	7554/6750/ 44/129	0.512	245787	2457.87	124500
RN	2	32768	7568/6750/ 44/129	0.505	245787	2457.87	124535
NR	8	32768	84270/78822/ 528/217.5	1.737	20560	205.6	41596
RN	8	32768	84272/78835/ 528/217.5	1.796	20560	205.6	41597

Table 5.4: Implementation results for FFT DIT

Table 5.4 presents the implementation results for several parameter combinations of the FFT DIT algorithm. Once more, the demands are higher than in the corresponding NTT table, while they are comparable between the two FFT tables.
The radix-8 implementation represents an outlier in these two tables due to the fact that the design itself is so big that it could not fit on the PYNQ-Z2 platform; therefore, Alveo U250 was used. This had an effect on resource and power utilization.
Another notable thing is the lack of radix-8 degree 4096 implementations for FFT algorithms. This is due to the limitations described in Section 4.1. In short, if the butterfly unit latency (in clock cycles), in conjunction with memory read time is longer than the shortest time between reading the same memory address twice (once in one stage and another time in another stage), then the system encounters a write after read (WAR)

hazard. The value was supposed to be overwritten before being read the second time, but if the result bound for that address is still in the pipeline, this cannot happen and the results will be wrong. The shortest distance between reading the same address twice is equal to $\frac{N}{R^2}$ and the memory takes 2 clock cycles to read from; therefore, the total butterfly latency should be less than $\frac{N}{R^2} - 2$. The more that the latency goes over this value, the more result coefficients will be wrong. While the NTT radix-8 butterfly manages to be fast enough for 4096 coefficients; the FFT one is over that latency value.

5.1 Resource utilization analysis

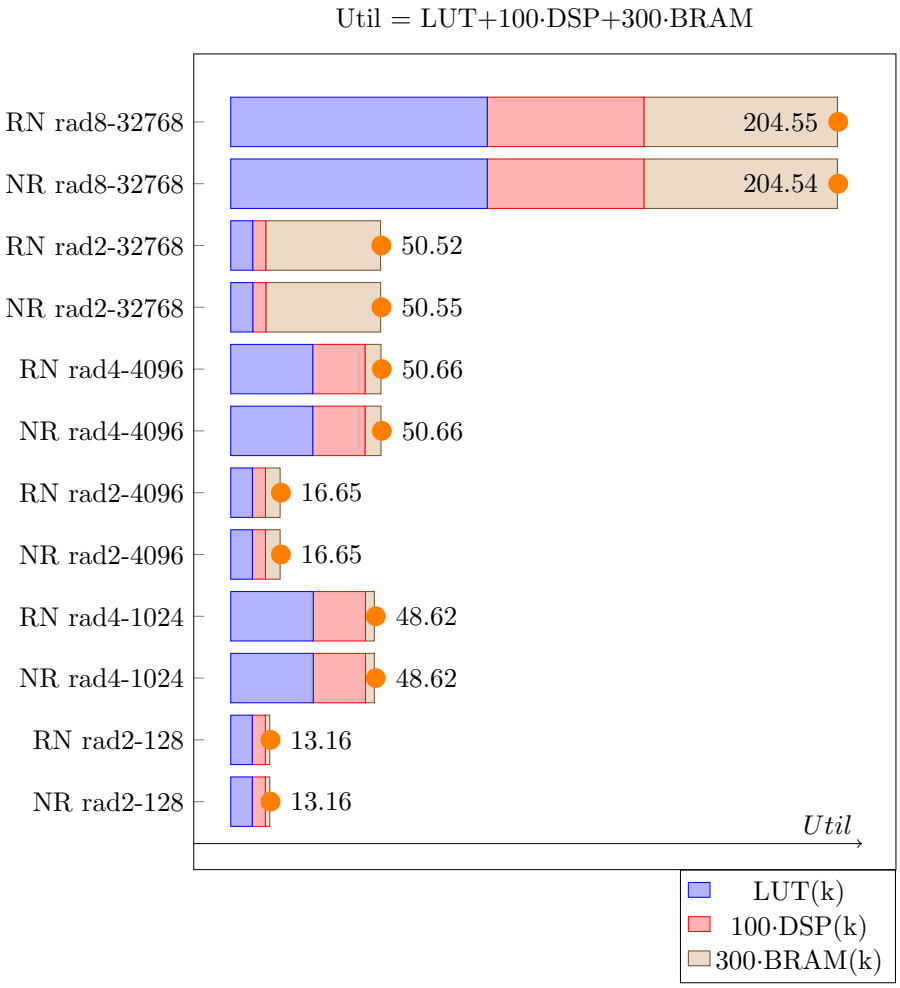

Figure 5.1: Resource utilization for FFT DIF

Figure 5.2 showcases the resource utilization of the NTT DIF algorithm in thousands of units. The number of units is equal to the number of LUT elements, together with the

number of DSP elements multiplied by 100 and number of BRAM elements multiplied by 300.
The NTT DIT algorithm produced a chart that was very similar to this one; therefore, its inclusion is redundant. In addition, the bars representing NR and RN memory access schemes are nearly identical. This shows that regardless of the selection of either the DIF or the DIT algorithm and NR or RN memory access schemes, the results are very comparable; therefore, the selection should not be made with resource utilization, power consumption or latency in mind, but whichever combination of parameters is the most convenient to embed in the user's overall system.

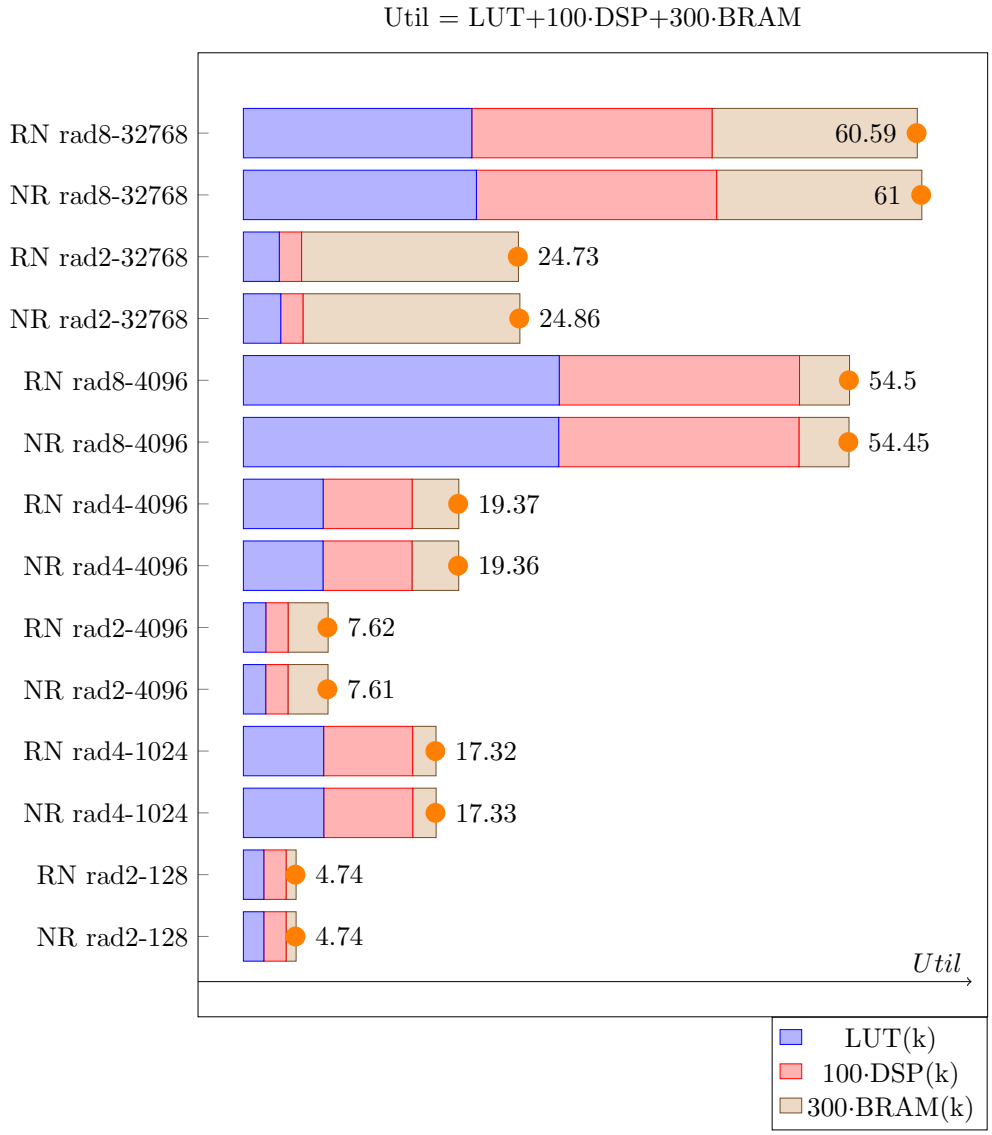

Figure 5.2: Resource utilization for NTT DIF

Figure 5.1 showcases the resource utilization of the FFT DIF algorithm in thousands of units. Again, the DIT version of the FFT algorithm produces very similar results, therefore it was omitted. As can be seen, the FFT algorithms have a significant increase in resource utilization over the NTT algorithms. As an example, the radix-8 degree 32768 setup uses nearly 3 and a half times the resources in the FFT algorithms.
Another interesting observation to make is the fact that the main contributor to the usage of LUT and DSP elements is the radix of the system, with there being especially significant jumps between radices 4 and 8. On the other hand, the main contributor to the usage of BRAM elements is the degree of the system. This can be seen in the fact that diverse system setups with the same degree use nearly identical amounts of memory. This would indicate that in order to keep the resource usage symmetrical higher radix values should be paired with higher degree values, and once more, the other parameters play an insignificant role in this comparison.

5.2 Latency analysis

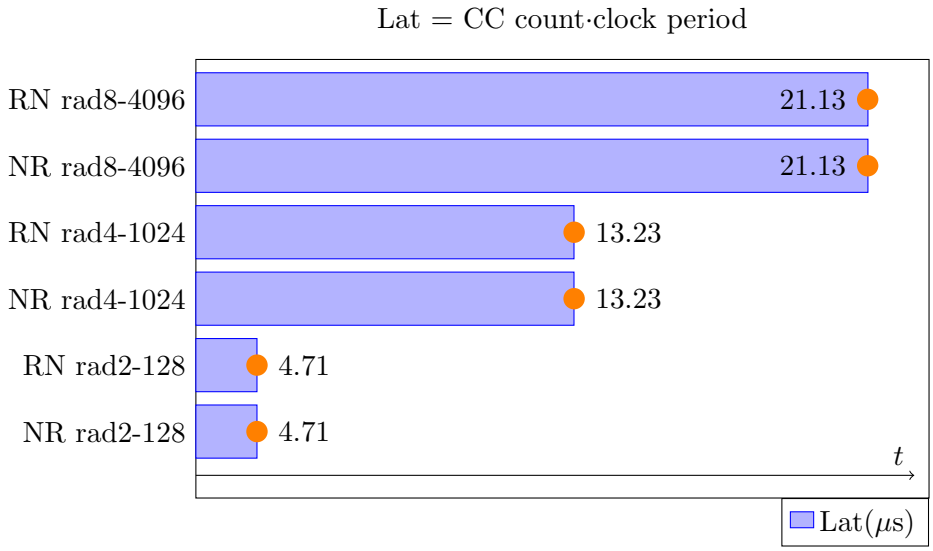

Figure 5.3: Latency for NTT (minimal degrees)

Figure 5.3 and Figure 5.6 present the time in μs required for the minimal degree values for each radix to complete one transformation. As can be expected, higher radices only work with higher degrees; therefore, they will take longer to complete. Additionally, the effect is exaggerated for the FFT algorithms due to the fact that the FFT radix-8 algorithms have much higher minimal degree values, as established previously.
Another conclusion that can be drawn is the very small difference between latency for NTT and FFT designs in cases where the degrees are identical. This is despite the fact that the FFT butterfly can take significantly more clock cycles per one operation. The explanation for this phenomenon is the full pipelining of the system, i.e., it does

not matter how long one set of coefficients takes to be processed, since each clock cycle sees the same number of coefficient sets processed as is the depth of the pipeline, and the depth of the pipeline is the same as the time it takes to process one coefficient set. Therefore, using a butterfly system, which takes more clock cycles to perform the same operation, would have minimal impacts on the performance, assuming that the new, slower butterfly unit is pipelined to the same degree, which is to say, that it can accept one set of inputs per clock cycle. The few clock cycles of difference between the same setups in NTT and FFT algorithms can be seen as either the difference in the time it takes the pipeline to fill up, or the time it takes the pipeline to completely empty. In other words, a butterfly unit that takes x cycles more per operation will only slow down the overall system by x clock cycles, instead of $x \cdot \#OfOperations$.

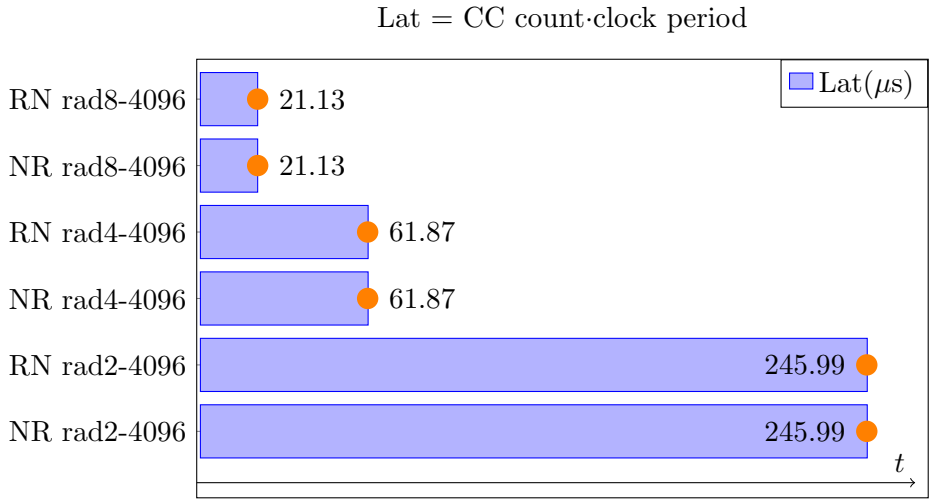

Figure 5.4: Latency for NTT (4096 coefficients)

Figure 5.4 and Figure 5.5 present the time in μs required for each radix with the degree values of 4096 and 32768, respectively, to complete one transformation. As can be seen, higher radix values perform equivalent operations much quicker. This is due to the fact, that not only do they process $\frac{R_{larger}}{R_{smaller}}$ times of coefficients more per one operation, but they also require $log_{R_{smaller}} R_{larger}$ times fewer stages to complete the full transformation. In addition, despite the fact that higher radix butterfly units have much longer latencies, due to the pipelined design, as explained above, the impact is limited to just the difference in the latency of one operation. These facts make the higher radix designs much more time-efficient at transformation operations.

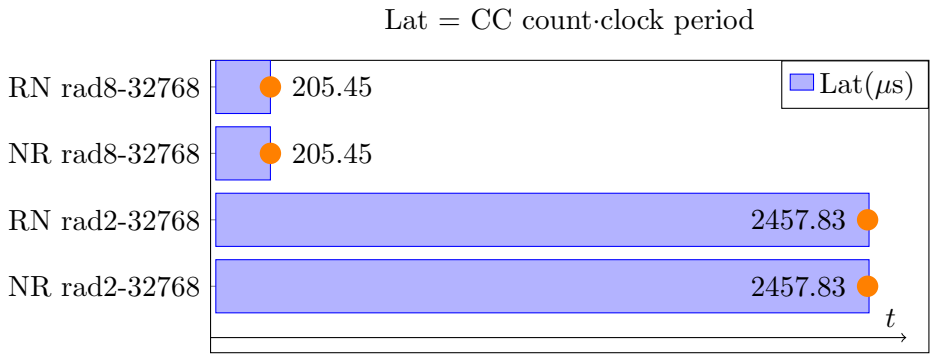

Figure 5.5: Latency for NTT (32678 coefficients)

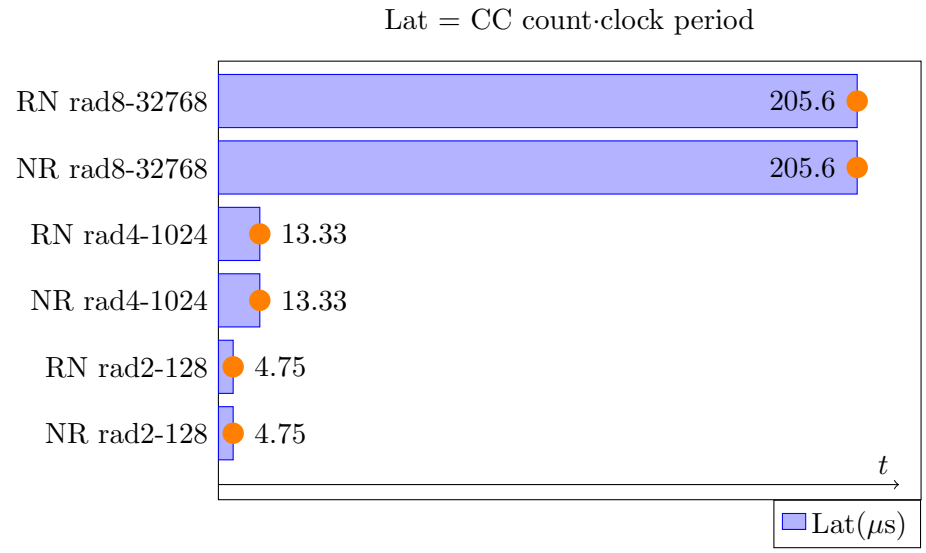

Figure 5.6: Latency for FFT (minimal degrees)

Figure 5.7 and Figure 5.8 showcase the differences in time taken for different radix values to transform a degree 4096 and a degree 32768 polynomial, respectively. The proportions of time needed are very similar to the same case with the NTT algorithm. This further solidifies the insights about the performance of different setups of parameters and their selection based on presented benchmarks.
The conclusion that can be drawn is that, assuming that the degree and radix are constant, the performance of any transformation system will be very consistent as well.

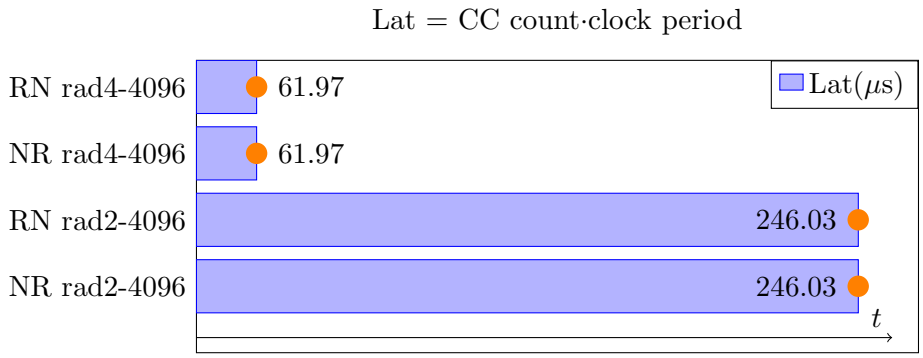

Figure 5.7: Latency for FFT (4096 coefficients)

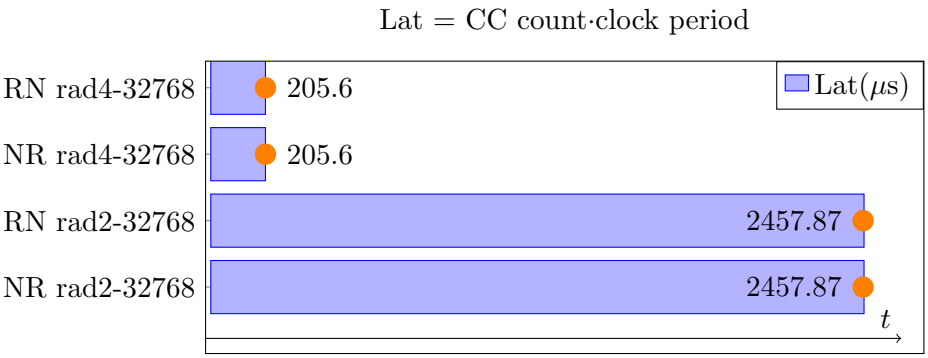

Figure 5.8: Latency for FFT (32768 coefficients)

5.3 Area time product analysis

Figure 5.9 and Figure 5.10 showcase the area time product of the two systems with FFT algorithms. The area time product (ATP) demonstrates the efficiency of a system and is calculated by multiplying its resource utilization, as given in Tables 5.1, 5.2, 5.3 and 5.4, with the corresponding latencies in μs. The values are too big for all of them to be shown in a single plot; therefore, vertical breaks are used.
As can be seen, the ATP values for a system with the same algorithm and degree, but different memory access scheme will be very similar. At the same time, the ATP of higher radix systems for all observed degree values is lower; therefore, the conclusion that using the higher radix system is more efficient is reached.
This does not mean that lower radix systems have no place, however, since they are capable of handling lower degree polynomials and they consume fewer resources and less power. Also, as the degree lowers, the gap in ATP will also lower, as the non BRAM-related portion of the utilization is fairly static as the degree changes and the BRAM-related utilization rises at a similar pace regardless of radix. This only leaves the latency, which will, naturally, get lower with the degree. Therefore, the non BRAM-

related portion of the utilization will constitute a higher and higher share of the ATP calculation, thus making the lower radix designs more efficient.

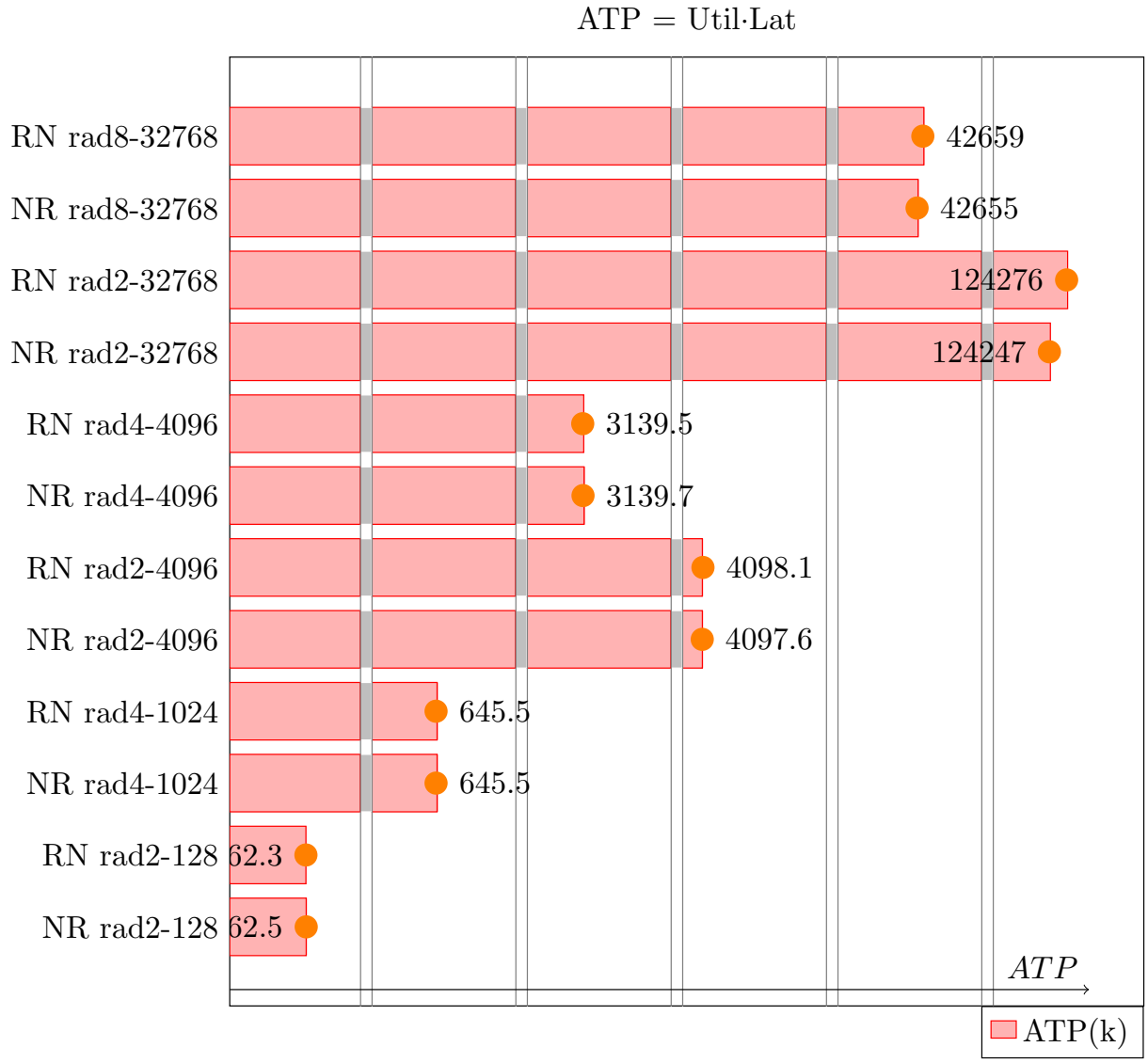

Figure 5.9: Area time product for FFT DIF

Figure 5.11 showcases the ATP values of the NTT DIF algorithm. They are many times lower than the ones in FFT-related figures. This is once more mostly due to the demands of the floating-point circuitry and the double width of all elements in memory.
The differences between different memory access schemes are more pronounced in this figure. This is most likely due to the subtle differences in implementation solutions found by the Vivado implementation engine, which are then greatly magnified by being multiplied with large latencies (such as the one for radix-2 degree 32768).

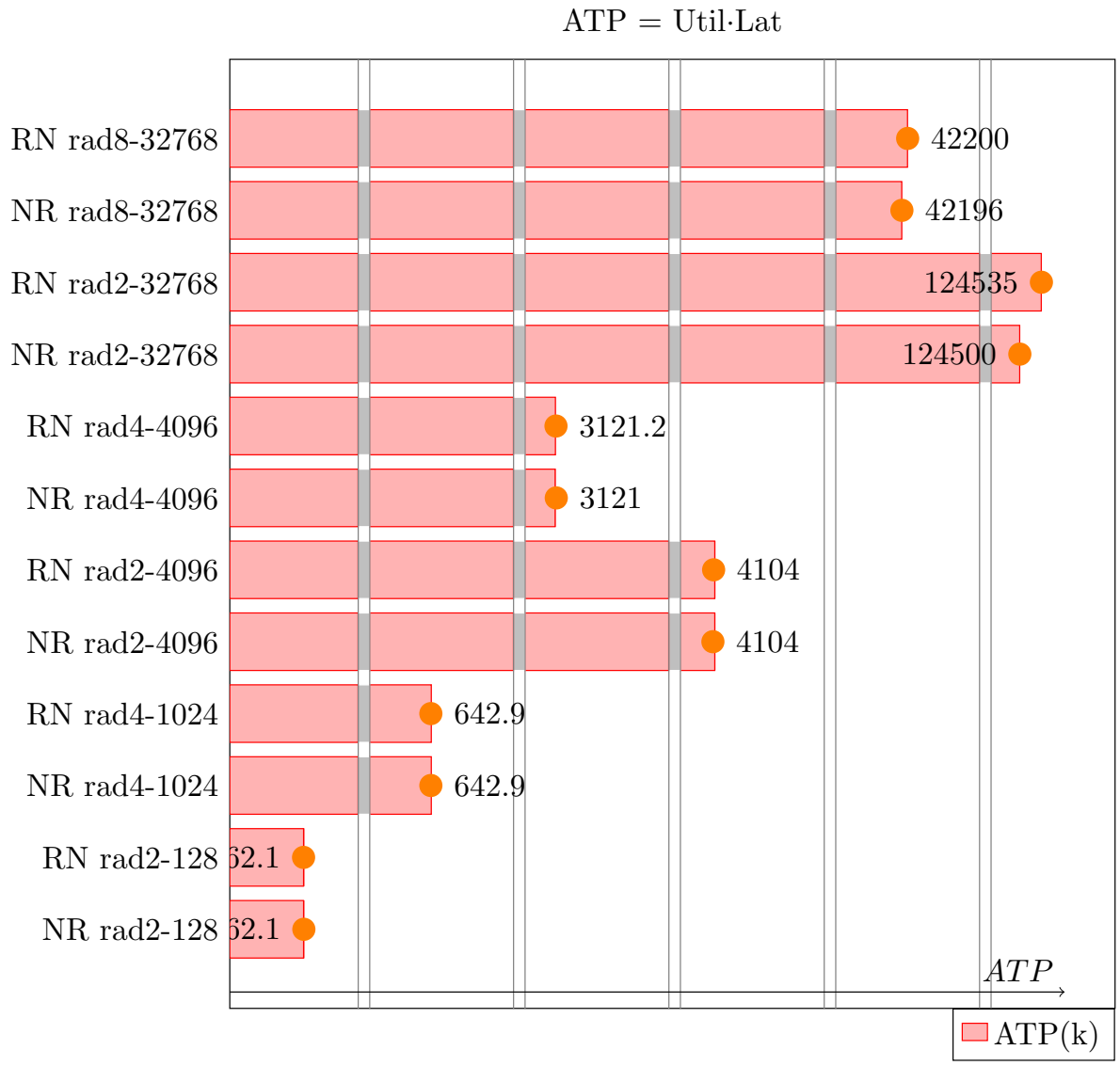

Figure 5.10: Area time product for FFT DIT

The ATP values of the NTT DIT algorithm can be seen in Figure 5.12. Here, the same trend is continued; the values are much lower than in FFT, but comparable to the other NTT variation.
Once again, differences between different memory access schemes within the same algorithm are pronounced, especially, as before, with the radix-2 degree 32768 design.

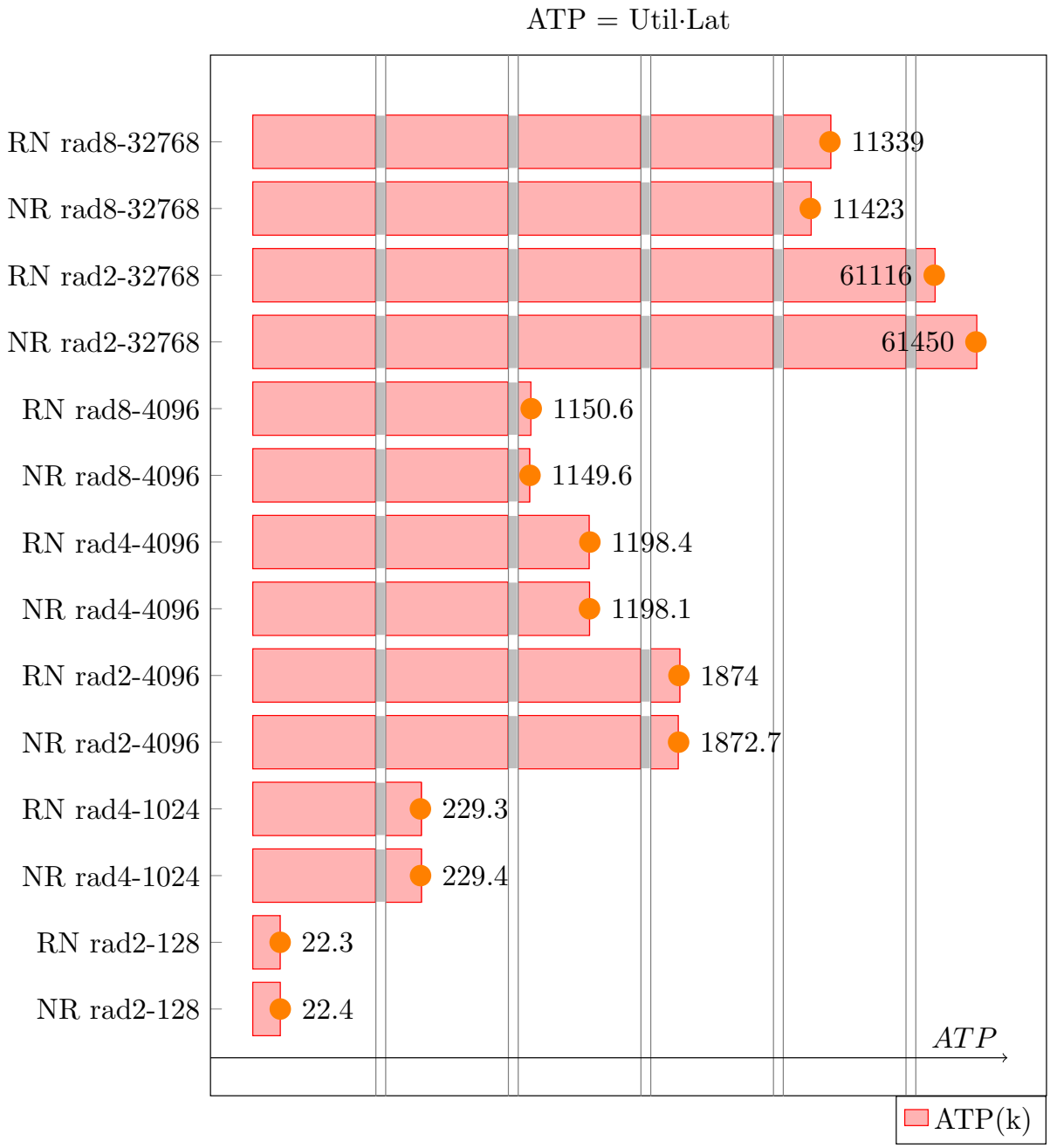

Figure 5.11: Area time product for NTT DIF

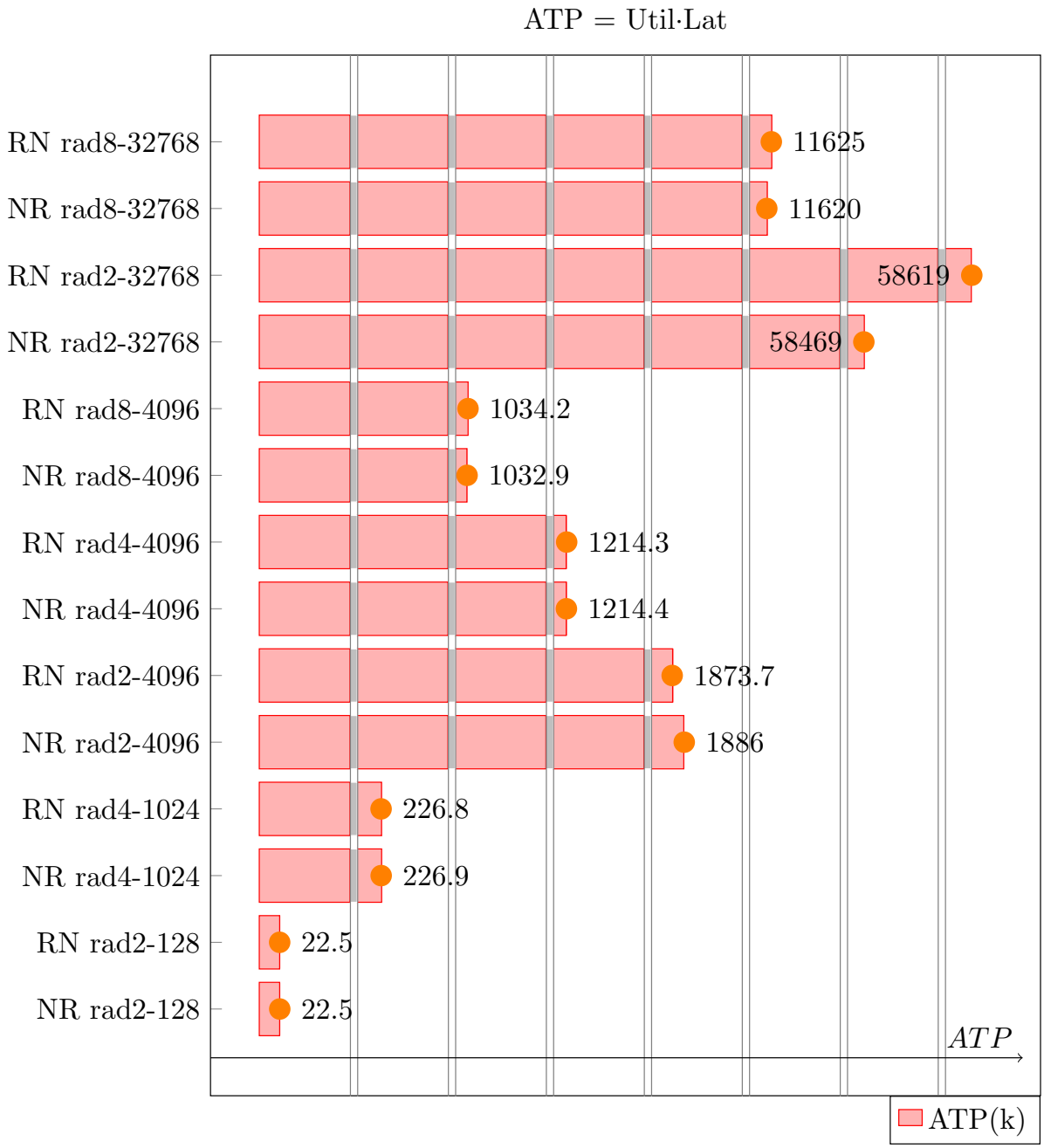

Figure 5.12: Area time product for NTT DIT

5.4 Energy consumption analysis

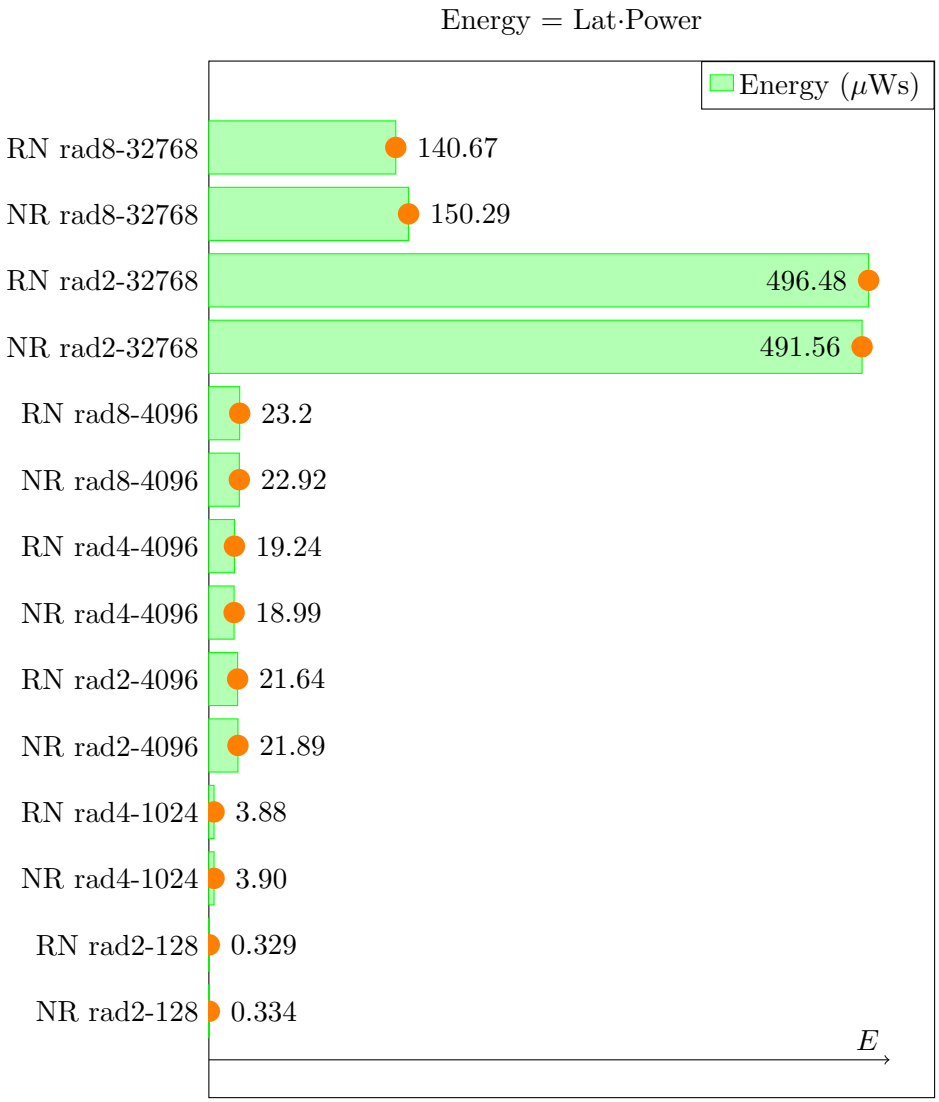

Figure 5.13: Energy consumption per one transformation for NTT DIF algorithms

Figure 5.13 demonstrates the energy usage per one transformation while using the NTT DIF algorithm. It is a product of power usage and the latency of one operation. NTT DIT algorithms have very similar energy demands, so that plot was omitted.
The results show that for large degrees, higher radix values are more energy efficient, however, at degrees of 4096, all three queried designs achieved similar results, with radix-4 winning out, by a slim margin. Therefore, at such degrees, the radix choice is up to other design decisions. At the same time, very low degree values have a negligible energy use.

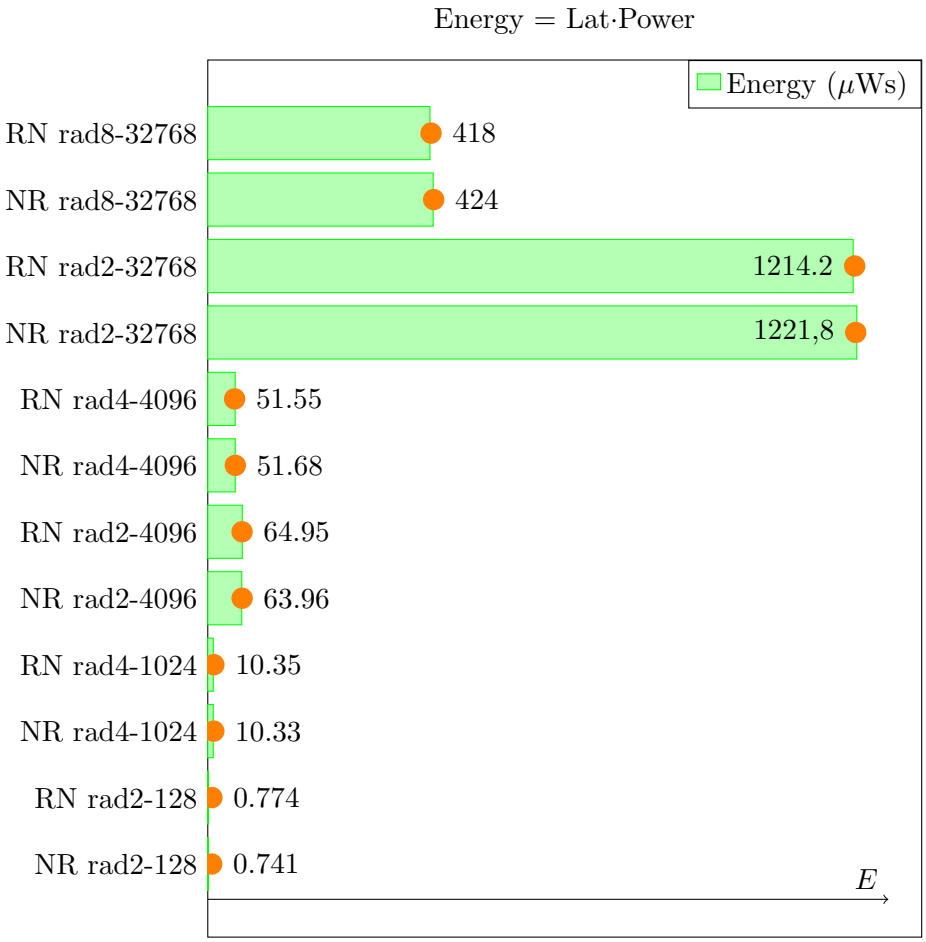

Figure 5.14: Energy consumption per one transformation for FFT DIF algorithms

The energy consumption per one transformation for FFT DIF systems can be seen in Firgure 5.14.
The results align with previous findings, with the exception of the radix-8 designs, which likely use more energy due to being implemented on a different board. Despite this, they are still much more efficient than the radix-2 designs of an equivalent degree. Among the degree 4096 designs, radix-4 is still the most efficient and the lower degree designs still use negligible amounts of energy.

6 Comparison with the state of the art

6.1 NTT comparisons

Work	PC	Platform	Freq. (MHz)	LUT/FF/ DSP/BRAM	Clock cycles	Latency (μs)	Power (W)
This work	T1	PYNQ-Z2	100	2018/2253/ 20/12	24599	245.99	0.089
This work	T2	PYNQ-Z2	100	7169/9187/ 80/14	6187	61.87	0.307
This work	T3	PYNQ-Z2	100	28355/28352/ 216/15	2113	21.13	1.085
This work	T4	PYNQ-Z2	100	7230/9147/ 80/7	1323	13.23	0.295
[HMR23]	C1	Virtex-7	150	18.9k/16.8k/ 220/16	8293	55.28	∅
[HMR23]	C2	Virtex-7	150	25.0k/19.6k/ 240/24	8310	55.39	∅
[HMR23]	C3	Virtex-7	150	22.6k/18.0k/ 220/16	4185	27.89	∅
[HMR23]	C4	Virtex-7	150	27.6k/22.5k/ 240/24	4214	28.09	∅
[HMR23]	C5	Virtex-7	150	15.7k/14.0k/ 180/4	2133	14.21	∅
[HMR23]	C6	Virtex-7	150	21.0k/16.4k/ 200/6	2138	14.25	∅
[HMR23]	C7	Virtex-7	150	18.6k/14.9k/ 180/4	1110	7.39	∅
[HMR23]	C8	Virtex-7	140	23.3k/18.8k/ 200/6	1134	8.09	∅
[Kri+24b]	C1	Virtex-7	250	6003/9600/ 44/16	24650	98.60	∅
[Kri+24b]	C2	Virtex-7	240	35262/44107/ 352/18	3146	13.10	∅
[Mer+20]	C1	Virtex-7	142	2.7k/2.6k/ 31/12	24780	173.00	∅
[Mer+20]	C2	Virtex-7	125	22k/17k/ 248/96	3276	26.00	∅

Table 6.1: NTT implementation comparisons, part one

Work	PC	Platform	Freq. (MHz)	LUT/FF/ DSP/BRAM	Clock cycles	Latency (μs)	Power (W)
[Mer+20]	C3	Virtex-7	125	338k/138k/ 1984/768	972	7.00	∅
[Liu+24]	C1	Virtex-7	154	1.9k/1.8k/ 42/17	24585	159.60	∅
[Liu+24]	C2	Virtex-7	150	14.1k/12.5k/ 336/41	3081	20.50	∅
[Liu+24]	C3	Virtex-7	133	52k/47k/ 1.3k/160	777	5.80	∅
[Liu+24]	C4	Virtex-7	130	108k/95.2k/ 2.7k/320	393	3.00	∅
[FS19]	C1	ASIC (65 nm)	25	∅	∅	∅	0.004
[FS19]	C2	ASIC (65 nm)	25	∅	∅	∅	0.003
[Ngu+24]	C1	Artix-7	234	3669/3544/ ∅/∅	∅	∅	∅
[Ngu+24]	C2	Artix-7	180	7451/5275/ ∅/∅	∅	∅	∅
[Ngu+24]	C3	Artix-7	208	7388/7151/ ∅/∅	∅	∅	∅
[Ngu+24]	C4	Artix-7	163	13804/11019/ ∅/∅	∅	∅	∅
[Mu+22]	C1	Virtex-7	185	802/525/ 4/7	24583	132.90	∅
[Mu+22]	C2	Virtex-7	157	5665/3118/ 33/8.5	3079	19.70	∅
[Mu+22]	C3	Virtex-7	121	14591/6468/ 80/12	1543	12.80	∅
[Che+21]	C1	Artix-7	213	1161/967/ 12/3	1308	6.10	∅
[Che+21]	C2	Virtex-7	270	1196/969/ 12/3	1308	4.80	∅
[Che+21]	C3	Virtex-7	250	2953/1875/ 27/5.5	668	2.70	∅
[Che+21]	C4	Virtex-7	278	475/307/ 3/1.5	5134	18.50	∅
[Che+21]	C5	Virtex-7	270	863/561/ 6/2.5	2574	9.50	∅
[Che+21]	C6	Virtex-7	263	2011/1070/ 12/5	1294	4.90	∅
[Che+21]	C7	Virtex-7	227	6300/2124/ 27/10	654	2.90	∅
[AM+16]	C1	Software	∅	∅	∅	13.90	∅
[AM+16]	C2	Software	∅	∅	∅	15.30	∅

Table 6.2: NTT implementation comparisons, part two

As stated before, many works exist in the realms of NTT and FFT transformation, be they fixed attribute hardware modules, or full-fledged hardware generators. This chapter aims to provide comparisons between those works and this one, looking at resource utilization, energy consumption and latencies, as well as flexibilities of each work (i.e., how many parameters are freely changeable by the user and to what degree do they impact the functionality of the design). With this, the goal is to present the pros and cons of each solution as well as to argue why this work can be a suitable alternative given certain design decisions.
The parameter combination of the work being analyzed will be given in the second column of Tables 6.1 and 6.3.
The utilization formula used for all comparisons going forward will be $Util = LUT + 100 \cdot DSP + 300 \cdot BRAM$.
The NTT parameter combinations for this work are as follow:
T1 - radix-2, degree 2^{12}, modulo bit width 64, DIF NR algorithm
T2 - radix-4, otherwise the same as T1
T3 - radix-8, otherwise the same as T1
T4 - radix-4, degree 2^{10}, modulo bit width 64, DIF NR algorithm

I: Work [HMR23], titled 'PROTEUS: A Tool to generate pipelined Number Theoretic Transform Architectures for FHE and ZKP applications' proposes a highly parameterizable tool for the purposes of generating hardware-based NTT transformation solutions. One of the parameter combinations shown in that work is degree of 2^{12} and modulo bit width of 64. This setup was explored in Section 5 of this work, albeit in both NR and RN memory access schemes, while [HMR23] only explores NR variants.
One major advantage of this work over [HMR23] is its ability to run algorithms of an arbitrary radix, while [HMR23] uses only radix-2 implementations. With this in mind, all radix variants implemented at the time of writing will be compared against the radix-2 implementation of [HMR23].
On the other hand, [HMR23] implements two different transformation architectures, Single-path Delay Feedback (SDF) and Multi-path Delay Commutator (MDC). The difference between these architectures is the fact that SDF consumes one coefficient per clock cycle and produces one result per clock cycle (given that the pipeline stages have filled up), while MDF consumes two coefficients per clock cycle and produces two results per clock cycle (again, given that the pipeline stages have filled up). The commonality between these two operations, however, is the fact that they make use of $\log_2 N$ processing elements. Each processing element consists of one butterfly unit and supporting circuitry. Each additional processing element lowers system latency, as more coefficients can be operated on in parallel; however, the resource use and complexity is also significantly increased. For these reasons, this work only makes use of one processing element.
Another advantage that [HMR23] possesses is the fact that it is runtime flexible with regards to the NTT algorithm it runs, i.e., it can freely switch between DIF and DIT algorithms. While this feature gives it an edge in runtime flexibility, it was deemed not significant enough to warrant the increased complexity of the control circuitry in this

design.
The final advantage that this work has over [HMR23] in the area of flexibility is its ability to be freely switched between NTT and FFT algorithms at synthesis time and with the change of a single parameter.
The parameter combinations are as follow:
C1 - SDF architecture, degree 2^{12}, modulo bit width of 64, no NWC technique
C2 - same as C1, but with NWC (thus, making this algorithm MNTT, or merged NTT)
C3 - MDC architecture, otherwise identical to C1
C4 - same as C3, but with NWC
C5 - SDF architecture, degree 2^{10}, modulo bit width of 64, no NWC
C6 - same as C5, but with NWC
C7 - MDC architecture, otherwise identical to C5
C8 - Same as C7, but with NWC.
For the parameter combination of C1, this work is more resource efficient with its own T1 and T2 and less resource efficient with the T3 design, while C2 is almost on par with T3. At the same time, C3 is slightly more resource efficient than the T3 implementation in this work, while C4 of [HMR23] is slightly less. For the smaller degrees, T4 is more efficient than C5, C6, C7 and C8.
The true advantage of [HMR23] lies in the domain of latency, owing to its multi-processing element design. Every combination for [HMR23] has much less latency than the T1 implementation, and less latency than T2, although the gap is narrower, while T3 has less latency than all of the combinations C1 - C4.
For the smaller degree combinations, latency puts the T4 implementation in front of C5 and C6, but behind the C7 and C8 combinations.
It should be noted that all designs in [HMR23] run at 150 MHz, apart from C8, which runs at 140 MHz, meaning that for the same clock cycle count, the actual latency in μs will be lower.
With all of these details, it can be concluded that [HMR23] should be used if multiple processing elements, with the choice of either SDF or MDC and DIF or DIT at the expense of resource utilization are desired, while if a flexibility in radices is wanted, then this work has the edge. When it comes to resources, this work in the T1 configuration should be considered for minimal usage, but for lower latency, either one of the implementations in [HMR23] or the T3 design in this work should be used.

II: Work [Kri+24b], titled 'OpenNTT: An Automated Toolchain for Compiling High-Performance NTT Accelerators in FHE' proposes another framework for compiling NTT hardware implementations with various parameter setups. One major feature of this work is the on-the-fly twiddle factor generation, which calculates the needed twiddle factors at the time of each operation instead of storing them in memory and merely reading them when necessary. This does save memory, but it increases complexity and necessitates dedicated hardware to achieve these calculations. For these reasons, this work opted for pre-calculated and stored twiddle factors.
As with the previous example, the advantage of this work is the flexible radix setting

as well as the ability to switch between FFT and NTT, but [Kri+24b] does enable the user to select the number of processing elements to instantiate, therefore lowering the gap in flexibility. This work is locked to one processing element in order to reduce the complexity of the control system and to make the design less resource demanding.
The parameter combinations are as follow:
C1 - degree 2^{12}, modulo bit width of 60, one processing element
C2 - same as C1, but with eight processing elements.
When compared against C1, this work is more resource efficient for the T1, slightly more efficient for the T2 and less efficient for the T3 parameter combination, while for C2, all three higher radix combinations (T1, T2 and T3) are more efficient. It should be noted that with a modulo four bits narrower, the design in [Kri+24b] has a slight edge when it comes to efficiency.
As for the latency, this work takes more time per operation with T1 and less with T2 and T3, provided that [Kri+24b] uses parameters C1 and if [Kri+24b] uses parameters C2, then it takes less time than any of the combinations in this work at that degree. It should be noted that [Kri+24b] runs at 250 MHz and therefore has a significant advantage when it comes to latency in μs. However, when observing clock cycles per one operation for the case when the combination C1 is used in [Kri+24b], the T1 design is nearly identical to it, while both T2 and T3 take significantly fewer clock cycles. If the C2 combination is used, then [Kri+24b] is significantly faster than the T1 twice as fast as the T2, yet slower than the T3 design.
In terms of deciding between the usage of this work and [Kri+24b], when it comes to flexibility, this work offers multiple radices, but [Kri+24b] offers the choice of multiple processing elements. On the other hand, if on-the-fly twiddle factor generation is desirable [Kri+24b] should be used, but if twiddle factor pre-calculation fits better into the overall design, then this work should be chosen. For the resource utilization, the highest efficiency is achieved by the T1 implementation in this work, but for latency, either the T3 implementation in this work, or the C2 design (with eight processing elements) in [Kri+24b] provide the best performance.

III: Work [Mer+20], titled 'A Flexible and Scalable NTT Hardware: Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography' proposes yet another NTT hardware generator driven by user-defined parameters. It is capable of generating radix-2 NTT implementations with an arbitrary number of processing elements.
In terms of flexibility, this work features the ability to switch between different radices and NTT/FFT algorithms as well as the ability to switch between DIF and DIT algorithms and NR and RN memory access schemes, while [Mer+20] can only use the radix-2 DIT RN NTT algorithm. Work [Mer+20] does, however, allow the user to instantiate multiple processing elements, thus bridging the flexibility gap somewhat.
The parameter combinations are as follow:
C1 - degree 2^{12}, modulo bit width of 60, implemented with the 'Area opt.' strategy
C2 - same as C1, but implemented with the 'Balanced' strategy

C3 - same as C1, but implemented with the 'Perf. opt.' strategy
In terms of resource utilization, the C1 combination is ahead of the T2 and T3 implementations, but behind T1. On the other hand, both C2 and C3 are behind all three of the higher degree combinations in this work.
As for the latency, the C1 design takes longer than all three higher degree parameter combinations in this work, although the difference between it and T1 is very slight. C2 is ahead of T1 and T2, but behind the T3 design. Lastly, C3 is the most performant implementation of all six currently under investigation.
It should be noted that all implementations in [Mer+20] run at 125 MHz, thus making them take less time with the same clock cycle count.
In conclusion, in cases where arbitrary radices, or a slightly wider modulo are more important, this work comes out ahead, but if the number of processing elements is of greater concern [Mer+20] should be considered. If the resource utilization must be minimized, then the T1 implementation in this work should be used, but if it is the processing time instead, then, the C3 implementation in [Mer+20] offers the best results.

IV: Work [Liu+24], titled 'An Area-Efficient, Conflict-Free, and Configurable Architecture for Accelerating NTT/INTT' aims to implement an efficient NTT and inverse NTT (INTT) hardware implementation with a so-called Constant-Geometry (CG) algorithm, which combines pre- and post-processing steps into the NTT operation. It also purports to allow for wide degree support even after compilation, as well as being able to be parallelized with multiple processing elements.
While the changeable number of processing elements and the run-time flexible degree in [Liu+24] lend it a great deal of flexibility, this work, again, allows arbitrary radix implementations, switching between NTT and FFT, between DIF and DIT algorithms and between NR and RN memory access schemes. This makes the algorithm in this work more flexible than [Liu+24] which only does the radix-2 NTT DIF NR algorithm.
The parameter combinations are as follow:
C1 - degree 2^{12}, modulo bit width of 60, one processing element
C2 - same as C1, but with eight processing elements.
C3 - same as C1, but with 32 processing elements.
C4 - same as C1, but with 64 processing elements.
The implementation in [Liu+24] with the parameter combination of C1 is less resource efficient than the T1, but more efficient than the T2 and T3 implementations in this work, while C2, C3 and C4 are less resource efficient than all three of the higher degree parameter combinations in this work.
In terms of clock cycles to complete one transformation, [Liu+24] with the C1 parameters is slower than T2 and T3 and faster than T1, while C2 is behind T3, but ahead of T1 and T2. The C3 and C4 combinations take fewer clock cycles than any parameter combination in this work for the same degree.
In conclusion, the work in [Liu+24] offers a new CG algorithm, a higher level of polynomial degree flexibility and a choice in the number of processing elements, but this work still offers more flexibility in the terms of radix choice and the choice of algorithms

and memory access schemes. For the lowest resource usage, the T1 combination in this work should be used, but for the lowest latency, the design in [Liu+24] with sixty-four processing elements (C4) should be chosen instead.

V: Work [FS19], titled 'Efficient and Flexible Low-Power NTT for Lattice-Based Cryptography' aims to create an ASIC-based NTT design focused on minimizing the power consumption.
Unfortunately, while [FS19] gives power results for a degree of 2^{10}, the modulo bit width is set to 16, which means that it is four times narrower than the modulo in this work. In addition, the table on the results of utilization does not clearly state the degree of the implemented design. The power results will still be presented for the sake of analysis.
The parameter combinations are as follow:
C1 - non-optimized design
C2 - optimized design
Both the C1 and C2 designs consume far less power than the C4 parameter combination in this work, but it should be kept in mind that this work implements all of the designs on an FPGA, while [FS19] does it on ASIC (with a fairly low frequency); therefore, that explains some of the power usage discrepancy.
Unfortunately, no further comparisons can be made with such scant details in [FS19] and the large differences in design, but in terms of flexibility this work wins out with its ability to use greater degrees, different radices, both NTT and FFT, DIF and DIT algorithms as well as NR and RN memory access schemes. For designs where a wider modulo is needed, this work will, again, be more suitable.

VI: Work [Ngu+24], titled 'High-Speed NTT Accelerator for CRYSTAL-Kyber and CRYSTAL-Dilithium' presents a hardware implementation of NTT in radix-2 and radix-4 configurations, while also putting forth a fully fledged polynomial multiplier based on that NTT system. Unfortunately, the transformation system in [6] is designed around the needs of dilithium and kyber cryptography schemes, which employ modular bit widths of 23 and 14 bits and degrees of either 2^8, or 2^9. Regardless, the results will be presented for analysis.
The parameter combinations are as follow:
C1 - degree 2^9, modulo bit width of 14, radix-2
C2 - degree 2^8, modulo bit width of 23, radix-2
C3 - same as C1, but in radix-4
C4 - same as C2, but in radix-4
The C1 combination is twice as efficient as this work's T4 implementation, while the C2 combination is on par with it. Meanwhile, the C3 combination is about on par with the T4 design from this work, while the C4 combination is about twice as demanding.
This means that, for the same radix and for higher degrees and greater modulo bit widths, this work comes out on top in terms of resource efficiency. At the same time, this work wins out in terms of flexibility, as [Ngu+24] only presents the choice of two radices, degrees and modulo bit widths, while this work can freely change all of them

and additional parameters on top. The work in [Ngu+24] does have the slight edge in the fact that it allows for multiple processing elements to be instantiated at once.

VII: Work [Mu+22], titled 'Scalable and Conflict-Free NTT Hardware Accelerator Design: Methodology, Proof, and Implementation' implements an NTT hardware generator with a particular focus on supporting different degrees, modulo bit widths, numbers of processing elements and, uniquely to all the works analyzed so far, the number of processing element layers. The processing elements in one layer feed their results to the processing elements next layer over and get their inputs from the elements one layer before.
The parameter combinations are as follow:
C1 - degree 2^{12}, modulo bit width of 24, with one processing element
C2 - same as C1, with four by two processing elements
C3 - same as C1, with eight by two processing elements
The C1 combination in [Mu+22] is more resource efficient than any implementations in this work (disregarding the discrepancy in bit widths). The combination C2 is less efficient than the T1 implementation in this work, but more efficient than the T2 and T3 implementations. Lastly, combination C3 is less efficient than the T1 and T2 implementations, but more efficient than T3.
In terms of clock cycles per one operation, the C1 combination has less latency than T1, but more than T2 and T3. Combination C2 has less latency than the T1 and T2 implementations in this work, but more than T3 and the third combination (C3) is the least latent of all six designs currently being analyzed.
In conclusion, while [Mu+22] offers the option of instantiating a wide number of processing units both in width and depth, this work permits the choice between FFT and NTT, as well as DIF and DIT algorithms and NR and RN memory access schemes. This means that this work offers more parameters which significantly impact the functionality of the system. Further, if the bit width of 24 is desired, then the work in [Mu+22] should be used, since it offers both the lowest utilization and the lowest clock cycle counts per operation, but if bit widths of 64 are needed, then this work should be used.

VIII: Work [Che+21], titled 'CFNTT: Scalable Radix-2/4 NTT Multiplication Architecture with an Efficient Conflict-free Memory Mapping Scheme' aims to provide radix-2 and radix-4 hardware implementations with a focus on minimizing required mathematical operations in order to save the maximal amount of hardware resources. It also provides the option of instantiating multiple processing elements in one system.
The parameter combinations are as follow:
C1 - degree 2^{10}, modulo bit width of 14, 2x2 processing element setup (radix-4)
C2 - same as C1, different FPGA
C3 - same as C2, but with a 4x2 processing element setup
C4 - same as C2, but with 1 processing element (radix-2)
C5 - same as C4, but with 2 processing elements
C6 - same as C4, but with 4 processing elements

C7 - same as C4, but with 8 processing elements
These combinations all work with polynomial degrees of 2^{10}, therefore, they will be compared to T4.
In terms of resource utilization, all of the mentioned parameter combinations are more efficient than T4, with the biggest discrepancy being with C4 (which is around 14 times less demanding) and the closest comparison being with C7 (which is only 1.4 times less demanding). It should be noted that [Che+21] uses more than four times narrower modulo widths, which explains the difference in resource demands. Another interesting point is the similarity between C1 and C2, showing that the change in the FPGA platform had minimal effect on utilization.
When it comes to clock cycle count per one operation, the T4 combination comes out ahead of C4 and C5, but behind all of the other implementations in [Che+21], although the results for C1, C2 and C6 are very close. The designs in [Che+21] run at more than twice the frequency than this work; therefore, the latency in μs will be lower, but even still, T4 beats out C4 by around 5 μs.
In conclusion, if multiple processing elements, or a narrower modulo bit width are desired, then [Che+21] should be used, but if a wider modulo, or higher levels of flexibility are needed, then this work would fit the design better. In terms of resource usage, the most efficient combination is the one given by C4, but it is also, by far, the most latent. The combination with the least latency is C7, but, relative to its narrow modulo, the resource utilization is quite high.

IX: Work [AM+16], titled 'NFLlib: NTT-Based Fast Lattice Library', which is within the 'Topics in Cryptology – CT-RSA 2016' publication implements a software-based library for performing NTT operations. The aim of the work is to use various optimization techniques to speed up the transformations as much as possible.
The work is software-based; therefore, it is difficult to compare against this work's hardware implementation in terms of flexibility, but it does have an edge in terms of ease-of-use, as all that a prospective user needs is a compiler to set it up.
The parameter combinations are as follow:
C1 - degree 2^{10} and modulo bit width of 62, gcc compiler, c4.2xlarge on AWS
C2 - same as C1, clang compiler, macbookair
The results within [AM+16] are scant, but the latency of one operation in C1 is similar to this work's T4, although T4 is faster by about 0.6 μs, while C2 is slower and takes about 2 μs more than T4.

6.2 FFT comparisons

Work	PC	Platform	Freq. (MHz)	LUT/FF/ DSP/BRAM	Clock cycles	Latency (μs)	Power (W)
This work	T1	PYNQ-Z2	100	7265/6583/ 44/5	475	4.75	0.156
This work	T2	PYNQ-Z2	100	27826/26269/ 176/10	1333	13.33	0.775
This work	T3	PYNQ-Z2	100	7305/6640/ 44/16.5	24603	246.03	0.26
This work	T4	PYNQ-Z2	100	27665/26373/ 176/18	6197	61.97	0.834
This work	T5	PYNQ-Z2	100	7632/6912/ 44/43.5	53275	532.75	0.31
[Che+18]	C1	ASIC (45nm)	1k	∅	1380	1.38	91.3e-3
[Che+18]	C2	ASIC (45nm)	1k	∅	21600	21.6	∅
[Yan+23]	C1	ASIC (28nm)	∅	∅	∅	4.56e-3	15.99e-3
[Yan+23]	C2	ASIC (28nm)	∅	∅	∅	1.94e-3	5.11e-3
[SFR09]	C1	Altera Cyclone III	∅	∅	4864	∅	∅
[SFR09]	C2	Altera Cyclone III	∅	∅	8459	∅	∅
[Kri+24a]	C1	ZYNQ-7000	130	19274/15163/ 56/97	∅	410	∅
[Kri+24a]	C2	ZYNQ-7000	130	20253/18152/ 100/82.5	∅	410	∅
[Kri+24a]	C2	Kintex-7	200	17680/14431/ 56/97	∅	267	∅
[Kri+24a]	C2	Kintex-7	200	20728/17647/ 100/82.5	∅	0.267	∅

Table 6.3: FFT implementation comparisons

The FFT parameter combinations for this work are as follow:
T1 - radix-2, degree 2^7, DIF NR algorithm, double precision FP
T2 - radix-4, degree 2^{10}, DIF NR algorithm, double precision FP
T3 - radix-2, degree 2^{12}, DIF NR algorithm, double precision FP
T4 - radix-4, otherwise identical to T3
T5 - degree 2^{13}, otherwise identical to T1

XI: Work [Che+18], titled 'A Variable-Size FFT Hardware Accelerator Based on Matrix Transposition' aims to implement a degree-flexible FFT hardware design meant for digital signal processing applications, while operating on single precision floating point numbers.
The design in [Che+18] does offer a wide range of degrees, from 2 to 2^{20}, which is wider than this work, since the lowest supported degree is 2^7. However, this work still has many more parameters which significantly impact its operation, therefore making it more flexible.
The parameter combinations are as follow:
C1 - degree 2^{10}, single precision FP
C2 - same as C1, with degree 2^{12}
With regards to latency, C1 takes a comparable number (though slightly more) of clock cycles as this work's T2; however, due to its higher frequency, the latency in μs is much lower. On the other hand, C2 takes fewer clock cycles than this work's T3, but nearly four times more than T4. Yet again, the greater frequency gives it an edge and in terms of real time, C2 is almost three times faster than T4.
When it comes to power consumption, the combination given in [Che+18] uses much less than any combination in this work, owing to the ASIC implementation.
In conclusion, this work should be used if flexibility, or the higher precision floating-point numbers are of concern, but if either latency, or power consumption, are more important, then [Che+18] should be considered.

XII: Work [Yan+23], titled 'Design of High Hardware Efficiency Approximate Floating-Point FFT Processor' implements an FFT hardware design with a particular focus on overall efficiency. This is achieved by implementing approximation to the floating-point numbers in use. Therefore, the results won't be as accurate as with exact numbers, but the latency, power use and area demand will all be lower.
It has a significant degree of flexibility when it comes to the approximations, allowing for many algorithms to be selected. It also supports degrees of 16 and 64. At the same time, this work supports many more operationally defining parameters, still making it more flexible overall.
The parameter combinations are as follow:
C1 - degree 2^6, no approximation
C2 - degree 2^6, approximation given as [5 5 5] (each number represents the stage mantissa width)
Both of the combinations C1 and C2 have much lower latencies and power use than any of the combinations in this work, owing to the much lower degrees and the ASIC implementation.
In conclusion, in cases where low-degree approximated results are all that is needed, work [Yan+23] should be chosen, but in all other cases this work would be more fitting.

XIII: Work [SFR09], titled 'OPTIMIZED HARDWARE IMPLEMENTATION OF FFT PROCESSOR' aims to implement an optimized hardware design for performing FFT

operations with radix-2 butterfly units.
It implements both twiddle factor pre-calculations with storage and on-the-fly twiddle factor computation. While this work is more flexible, this fact does give [SFR09] a degree of flexibility of its own.
The parameter combinations are as follow:
C1 - degree 2^9, twiddle factor pre-calculation
C2 - degree 2^9, on-the-fly twiddle factor calculation
Clock cycle counts per one operation are given in [SFR09] and both C1 and C2 are slower than this work's T2 combination, C1 about three times and C2 about six times. It should be noted that T2 has twice as many coefficients as C1 and C2.
In conclusion, in cases when on-the-fly twiddle factor calculations are needed, [SFR09] should be considered, but in all other cases, this work has better performance metrics.

XIV: Work [Kri+24a], titled 'Aloha-HE: A Low-Area Hardware Accelerator for Client-Side Operations in Homomorphic Encryption' first and foremost implements a hardware design of an accelerator for client-side HE operations, however, as the kind of HE algorithm employed in [Kri+24a] uses floating-point numbers, the transformation operation needed is FFT, as opposed to NTT had module ring integers been used. As such, [Kri+24a] also offers an implementation of an FFT design, along with some results.
The transformation is locked to a DIF algorithm during encoding and DIT during decoding, but [Kri+24a] does offer the choice of either twiddle factor pre-calculation or on-the-fly calculation, similar to [SFR09].
The parameter combinations are as follow:
C1 - degree 2^{13}, twiddle factor pre-calculation
C2 - degree 2^{13}, on-the-fly twiddle factor calculation
C3 - same as C1, different FPGA
C4 - same as C2, different FPGA
When it comes to resource utilization, C1, C2, C3 and C4 are quite comparable, with only one to three thousand units between them; however, they are all around two times more demanding than this work's T5 combination.
In terms of latency, C1 and C2 have the same results, and they are about 30% less latent than this work's T5 implementation. It should be kept in mind that [Kri+24a] runs at a 30% higher frequency, meaning that with the same clock, the two works would be comparable when it comes to latency. On the other hand, C3 and C4 are, again, identical to each other, all the while being twice as fast as T5, albeit with two times higher frequency.
In conclusion, while the FFT system is not the main point of [Kri+24a], it is still quite latency efficient. Despite this, when minimizing latency, either C1, C2, C3, C4 or T5 can be considered, but when minimizing resource utilization, T5 wins out. In terms of flexibility, [Kri+24a] does offer two choices in regards to handling twiddle factors, but this work still offers more parameters with significant effects on transformations.

7 Conclusion

This work introduced a tool for generating NTT and FFT hardware accelerator designs with support for both DIF and DIT algorithms as well as NR and RN memory access schemes and the ability to implement and use a butterfly unit of an arbitrary radix given as $2^k, k \in \mathbb{N}$. In addition, it also supports a host of other parameters, such as a choice of polynomial degrees, and more.
The most substantial contribution to this end was the development of a comprehensive control system, which could adapt to all of the user-defined parameters. Two key components in its construction were the index generator and the memory intercept. The index generator carried out the vital function of loading coefficients from memory in order for the rest of the system to operate on them. The biggest challenge in creating a memory load module in this environment was the vulnerability of the system to hazards and conflicts. In order to avoid such issues, a flexible module was created, which can take into account any combination of parameters and generate a series of indices, which both allow the transformation to be carried out and avoid any conflicts in RAM. The second part of this control structure is the memory intercept. It had to prepare the memory structure in such a way that it would allow the index generator to accomplish its own function within one clock cycle and with the least amount of complexity. To accomplish this, it, too, takes in any combination of parameters and generates a stream of new addresses for the results of every operation carried out within the system. In this way, these two modules perform similar tasks and influence each-others functionality.
Further, to perform the butterfly operations, a template was developed in order to assemble butterfly units of various radices from a handful of pre-constructed submodules. Finally, to connect the constructed accelerator to a suitable CPU, a wrapper was designed, meant to support AXI communications.
A selection of designs was then created with the mentioned generator and implemented on the PYNQ-Z2 FPGA platform as a means of proving their correctness. The results obtained were then analyzed and compared against existing works in the field. These comparisons showed that, for comparable degrees, this work provides competitive metrics in the fields of resource utilization, latency and power usage. At the same time, the flexibility provided by this generator is unmatched as of yet.
These factors make this work suitable for a wide variety of needs, from relatively low degree, low resource demand systems, to very fast systems meant to transform large polynomial, all while having a large throughput. At the same time, the generator is simple to use and intuitive, allowing for a low-effort generation of transformation systems, which are meant to be just a part of a greater work. In this way, the design process of cryptographic systems, which make use of NTT or FFT transformations should be significantly streamlined, as one of the most common operations needed is now easily

accessible. With this simplicity of use, this system could see a large number of more complex designs spring up, using NTT and FFT systems generated by it. This would significantly accelerate the adoption of PQC and HE schemes.

7.1 Further opportunities

There are, several opportunities for further research and improvements. One area of development, which is quite common in state-of-the-art, but as of yet unsupported in this work is the paradigm of having multiple processing elements. In order to achieve this, an architectural change will have to be undertaken on the index generator to allow it to generate multiple indices at once, along with modifying the memory intercept such that it can handle multiple sets of writes per operation. Alternatively, a new 'super-index generator' and 'super-memory intercept' can be developed, which instantiate multiple smaller index generators and memory intercepts, which all handle individual portions of a polynomial. This area of study is, however, of limited utility, as multiple processing elements would increase resource demands and lower latency - an outcome which is handily achieved by utilizing a higher radix design.
In addition to this, a 'pause switch' can be implemented in the index generator and the memory intercept, which would allow them to skip a number of clock cycles in order to allow the butterfly unit to catch up, in cases where the implemented unit is too slow to handle the low number of coefficients. This, while increasing resource demands and the latency per operation, would allow the system to handle lower polynomials, thus increasing its flexibility even further.
Another improvement worth considering would be the ability to support multiple primes during runtime. This would require either a flexible integer reduction unit, or, provided that only a handful of primes are to be supported, a number of reduction units, with the appropriate one being in effect any any one time.
Lastly, a larger end-to-end polynomial multiplication unit can be developed using the existing system. It would consist of the forward and inverse transformation modules accompanied by either an array of multipliers, or a single one, depending on constraints. Further, this number can be left up to the user, with appropriate FIFO structures being generated on the fly.

Acknowledgement

The author used ChatGPT for the final grammar check as well as refining the reading flow of the document. No content was generated by AI - all of it was written by the author.

Bibliography

[AM+16] Carlos Aguilar-Melchor et al. "NFLlib: NTT-Based Fast Lattice Library". In: *Topics in Cryptology – CT-RSA 2016*. Switzerland: Springer International Publishing, 2016.

[Bav+23] Ritik Bavdekar et al. "Post Quantum Cryptography: A Review of Techniques, Challenges and Standardizations". In: *International Conference on Information Networking (ICOIN)* (2023).

[BL17] Daniel J. Bernstein and Tanja Lange. "Post-quantum cryptography". In: *NATURE* 549 (2017).

[BR20] Vaishali Bhatia and K.R. Ramkumar. "An Efficient Quantum Computing technique for cracking RSA using Shor's Algorithm". In: *IEEE 5th International Conference on Computing Communication and Automation (ICCCA)*. IEEE, 2020.

[Cha+17] Melissa Chase et al. "SECURITY OF HOMOMORPHIC ENCRYPTION". In: *Homomorphic Encryption Standardization Workshop* (2017).

[Che+18] Xiaowen Chen et al. "A Variable-Size FFT Hardware Accelerator Based on Matrix Transposition". In: *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS* 26.10 (2018).

[Che+21] Xiangren Chen et al. "CFNTT: Scalable Radix-2/4 NTT Multiplication Architecture with an Efficient Conflict-free Memory Mapping Scheme". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2021).

[CLW67] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. "Historical Notes on the Fast Fourier Transfor". In: *PROCEEDINGS OF THE IEEE* 55.10 (1967).

[CLW69] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. "The Fast Fourier Transform and Its Applications". In: *IEEE TRANSACTIONS ON EDUCATION* 12.1 (1969).

[FS19] Tim Fritzmann and Johanna Sepúlveda. "Efficient and Flexible Low-Power NTT for Lattice-Based Cryptography". In: (2019).

[Für07] Martin Fürer. "FASTER INTEGER MULTIPLICATION". In: (2007).

[Gen09] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *STOC'09 Proceedings of the 2009 ACM International Symposium on Theory of Computing*. 2 Penn Plaza, Suite 701 New York, New York 10121-0701: The Association for Computing Machinery, 2009.

[GS66] W. M. Gentleman and G. Sande. "FAST FOURIER TRANSFORMS—FOR FUN AND PROFIT". In: *AFIPS '66 (Fall): Proceedings of the November 7-10, 1966, fall joint computer conference*. New York NY United States: Association for Computing Machinery, 1966.

[HMR23] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. "PROTEUS: A Tool to generate pipelined Number Theoretic Transform Architectures for FHE and ZKP applications". In: (2023).

[KP20] Manoj Kumar and Pratap Pattnaik. "Post Quantum Cryptography(PQC) - An overview". In: (2020).

[Kri+24a] Florian Krieger et al. "Aloha-HE: A Low-Area Hardware Accelerator for Client-Side Operations in Homomorphic Encryption". In: *Design, Automation & Test in Europe Conference (DATE 2024)*. EDAA, 2024.

[Kri+24b] Florian Krieger et al. *OpenNTT: An Automated Toolchain for Compiling High-Performance NTT Accelerators in FHE*. Cryptology ePrint Archive, Paper 2024/1740. 2024. URL: https://eprint.iacr.org/2024/1740.

[Liu+24] Si-Huang Liu et al. "An Area-Efficient, Conflict-Free, and Configurable Architecture for Accelerating NTT/INTT". In: *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS* 32.3 (2024).

[LPY22] Dai Li, Akhil Pakala, and Kaiyuan Yang. "MeNTT: A Compact and Efficient Processing-in-Memory Number Theoretic Transform (NTT) Accelerator". In: *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS* 30.5 (2022).

[Mer+20] Ahmet Can Mert et al. "A Flexible and Scalable NTT Hardware: Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography". In: (2020).

[Mon94] Peter L. Montgomery. "A Survey of Modern Integer Factorization Algorithms". In: (1994).

[MOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[Mu+22] Jianan Mu et al. "Scalable and Conflict-Free NTT Hardware Accelerator Design: Methodology, Proof, and Implementation". In: *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS* 42.5 (2022).

[MW17] Ruben Niederhagen amd Michael Waidner. *Practical Post-Quantum Cryptography*. Rheinstraße 75, 64295 Darmstadt: Fraunhofer SIT, 2017.

[Ngu+24] Trong-Hung Nguyen et al. "High-Speed NTT Accelerator for CRYSTAL-Kyber and CRYSTAL-Dilithium". In: 12 (2024).

[NIS17] NIST. *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization. Accessed: 2025-03-22. 2017.

[OTD13] Monique Ogburn, Claude Turner, and Pushkar Dahal. "Homomorphic Encryption". In: *Complex Adaptive Systems, Publication 3*. Elsevier B.V, 2013.

[Reg06] Oded Regev. "Lattice-Based Cryptography". In: (2006).

[RSA77] R.L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: (1977).

[SFR09] Ahmad A. Al Sallab, Hossam Fahmy, and Mohsen Rashwan. "OPTIMIZED HARDWARE IMPLEMENTATION OF FFT PROCESSOR". In: (2009).

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: (1994).

[Yan+23] Chenggang Yan et al. "Design of High Hardware Efficiency Approximate Floating-Point FFT Processor". In: *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS* 70.11 (2023).

[YPB14] Xun Yi, Russell Paulet, and Elisa Bertino. *Homomorphic Encryption and Applications*. Springer, 2014.

Appendix

Internal name	External name	Source	Destination	Purpose
enable_i	start	Outside	Controller	Start operations
finished_o	finished	Controller	Outside	Operation end notification

Table 7.1: Controller control signals

Internal name	External name	Source	Destination	Purpose
mem_enable	mem_enable	Outside	Controller	Enable memory operations
mem_select	mem_select	Outside	Controller	Select bank group to access
mem_mode	mem_mode	Outside	Controller	Select mode of access
mem_bank	mem_bank	Outside	Controller	Select bank to access
mem_addr	mem_addr	Outside	Controller	Select address to access
mem_in_data	mem_in_data	Outside	Controller	Data input
mem_out_data	mem_out_data	Controller	Outside	Data output

Table 7.2: Controller memory IO signals

The *mem_select* signal selects between accessing coefficient banks and accessing twiddle factor banks, internal twiddle factor storage registers and the reduction modulo.
The *mem_mode* signal toggles between read and write when accessing coefficients, or twiddle factor memory and internal twiddle factor storage alongside the reduction modulo otherwise.
In addition, in order to change the modulo, **FFT_NTT** must be set to 1 and **MOD_SAFEGUARD** must be set to 0.