UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Buntterfly: A Flexible Hardware Generator for the Number Theoretic Transform

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Jason Vranek

September 2020

The Thesis of Jason Vranek
is approved:

Scott Beamer, Chair

Heiner Litz

Owen Arden

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Buntterfly: A Flexible Hardware Generator for the Number Theoretic Transform

by

Jason Vranek

In the current era, many computations are being outsourced to third parties, whether to large cloud providers or to nodes in decentralized blockchains. As increasingly sensitive data is being operated on (e.g. financial and medical data), it is imperative that the data be protected and that the integrity of an outsourced computation is ensured.

The two cryptographic primitives Probabilistically Checkable Proofs and Fully Homomorphic Encryption enable verifiable and privacy preserving computations respectively, at the cost of high computational overhead.

Both primitives share common low level arithmetic operations, particularly math over polynomial rings. The number theoretic version of the traditional Fast Fourier Transform (FFT) for working over finite fields is called the Number Theoretic Transform (NTT), and is especially suited for FHE [34], [3], [33], [26], [41], [28], [37]. As far as we have seen in literature there has yet to be a paper demonstrating a hardware accelerator for ZKPs, particularly the variants of interest: ZK-STARKs [6] and ZK-SNARKs [10], however PCPs employ polynomial interpolation, evaluation, and Low-Degree Extensions, all of which are accelerated by NTTs.

By utilizing the recent hardware programming language Chisel [4], we created Buntterfly, an easily configurable open source NTT hardware generator that can be parameterized according to the workload (ZK-PCP or FHE) and target platform (Latency, Throughput, Area). Once the core modular arithmetic operators are in place, the flexibility of Chisel can allow different NTT architectures to be generated according to the target metric and target FPGA model. We demonstrate that instantiated NTT designs using Buntterfly on FPGAs can provide 64x speedup over optimized software NTTs. Buntterfly is flexible and can easily be extended to perform expensive operations such as Low-Degree Extensions for generating Reed-Solomon Codes [36] for PCPs, or large integer multiplication in FHE.

I'd like to dedicate this to my late grandfathers to whom I owe my passion for building and learning.

Acknowledgments

Thank you to my parents, the reason I've made it to where I am. Thank you to my advisor for all the guidance I have received, without which I would not have been able to complete this work.

Chapter 1

Introduction

The purpose of this thesis is to present my research exploring two very exciting and relevant developments in cryptography, Zero Knowledge Probabilistically Checkable Proofs (ZK-PCPs) and Fully Homomorphic Encryption (FHE), and then to demonstrate Buntterfly, a hardware generator designed to accelerate their computations.

ZK-PCPs and FHE are called cryptographic primitives, which are essentially tools or components used to construct cryptographic systems. Similarly to how arithmetic operations like addition and multiplication allow one to build up complex equations, ZK-PCPs and FHE can be used to build trustworthy and privacy preserving protocols.

ZK-PCPs are a pillar in the field verifiable computing, the goal being to verify that a computation done by a potentially malicious third party is guaranteed to be correct whilst keeping confidential data private [46]. This model removes the need for any element of trust in the computation by relying on the provable security of math and cryptography. In the current era of cloud computing, many expensive computations are outsourced to large data centers that provide massive compute power at low costs. Computations such as deep-learning or astronomical scientific calculations would be impractical to run on consumer hardware as they would potentially take years to complete. The extensive runtimes and non-negligible costs mean it is imperative that the results from the third party are of high integrity.

The question then is, how does one verify that a given output was correctly computed? Cloud infrastructures (CI's) are incentivized to behave honestly to preserve their reputation, and so one could trust their output. However, one could never rule out bad or compromised actors. There is always the possibility that an adversarial CI will intentionally produce an incorrect output. To counter this, a client could re-run the entire calculation and check for equivalent outputs, but this is problematic if the computation was already large enough to necessitate outsourcing.

Specialized hardware enclaves in the form of Trusted Execution Environments (TEE) are used to ensure there was no meddling involved during the computation. TEEs aim to provide a secure environment for running applications in isolation which protects against these applications corrupting others or vice-versa, ensuring correct program behavior. However, implementations in practice are susceptible to attacks [23], and hardware solutions inherently have chains of trust involved, from the often closed source designs all the way down the fabrication supply chain [46].

Even more subtle but nonetheless inevitable are incorrect computations due to minuscule bugs at the transistor level. Regardless of whether these outputs come from honest CI's or not, the end result is incorrect, and a method to verify correctness is necessary.

ZK-PCPs provide a means to verify the correctness of a computation in exponentially less time than it took to complete, regardless of whether the third party is trustworthy or not. Thus ZK-PCPs provide a solution to the computational integrity problem that does not rely on trust, but instead is rooted in math and cryptography.

Another pressing issue deals with data privacy. The implication of relying on CI's for the bulk of our everyday software services is the inherent security risk posed to our data, as ownership of the data must be relinquished to the CI for them to do anything with it. Of course one can always encrypt their data to prevent CI's from reading it, but without access to the decryption key, the CI is unable to perform useful work and acts as nothing more than external storage. FHE aims to remedy this scenario by allowing arbitrary computations to be done on encrypted data, without revealing anything about what the underlying data is. This will allow for CI's to perform the same software services, whilst keeping users' data private.

The end-all solution of verifiable computing would be to build machines capable of performing computations fully homomorphically, while simultaneously providing proofs attesting to the computation's correctness. The result of such a duo would eliminate any attempts at foul play when computations are outsourced as this setup will provide privacy and correctness by construction. However, ZK-PCPs and FHE each have massive computational overheads, meaning general purpose computations are currently impractical to implement. Fortunately, the gap between which computations are theoretically possible and which are actually practical has been narrowing each year.

Buntterfly is an attempt to help bridge this gap by providing a flexible hardware generator to produce circuits capable of outperforming current software libraries for core ZK-PCP and FHE operations. This is a first step towards practical verifiable computing as Buntterfly can easily be customized depending on the use case and then instantiated on commercial FPGAs to provide speedups at relatively low cost.

Buntterfly targets the number theoretic transform (NTT) as its generated hardware due to the NTT's ability to efficiently perform polynomial math. Large polynomial multiplication and Low-Degree Extensions constitute some of the most expensive operations in FHE and PCPs respectively, and are typically performed using NTTs. Thus Buntterfly can effectively be used to target either of these workloads.

The contributions from this paper can be summarized as follows:

- Created a flexible hardware generator that produces NTT circuits parameterized by a prime modulus, and the number of NTT points.

- Built upon the state-of-the-art modular multiplier algorithm [31] to be compatible with the addition and subtraction operations needed during the NTT, amortizing the cost of switching to the specialized redundant form.

- Yielded preliminary results showing potential speedups up to 64x when using Buntterfly over optimized software NTT libraries.

Chapter 2

Background

2.1 Chapter Overview

This chapter covers the history of PCPs and FHE and then provides an overview on how they work. Next, the number theoretic transform (NTT) is introduced, its involvement with PCPs and FHE is explained, and the involved math is explained in detail.

2.2 Blockchain Primer

It can be argued that in 2009, two parallel developments led to the revitalization of two previously stagnant fields of study that were once thought to be too computationally expensive for practical use: Fully Homomorphic Encryption (FHE) and Zero Knowledge Proofs (which will from now on be referred to as Probabilistically Checkable Proofs (PCPs), as these are the constructions used in practice and all PCPs can be made into *Zero Knowledge* PCPs).

First in 2009, the cryptocurrency Bitcoin [30]was released, and after a slow beginning, cryptocurrencies ramped up in the public's attention until its peak in

2017 and subsequent crash. This decade resulted in significant advances in the field of cryptography, which can be argued was somewhat of niche field before this point.

Cryptocurrencies ushered the term *Blockchain* into the public's vocabulary, which, in a nutshell, is a decentralized public database empowered through cryptography and game theory that provides a fault tolerant, *trustless* escrow system for transfers of digital assets, where such assets are anything from cryptocurrencies to crypto-kitties [43] to virtual real estate [24]. Notice that *trustless* is a carefully chosen term articulating that no single entity is to be trusted. Instead, the open source code running indisputable math is executed across millions of independent computers. This ensures that the code will execute regardless of the intentions of any individual actors. This is different than the centralized model that is customary in most applications. The classic example is a centralized bank that is trusted by the public, but has the ability to censor individual customers.

Blockchains like Ethereum [48] allow for more than just cryptocurrency transactions, and are *Turing complete*. This means Ethereum can be thought of like a global computer, where participants can run arbitrary code in the form of *Smart Contracts*, and all the changes to the computer's memory are permanently saved as blocks that form a blockchain.

Decentralization comes at the cost of throughput (measured in transactions per second), which has been one of the key bottlenecks preventing public adoption of blockchain for applications beyond cryptocurrencies. As a result many resources were devoted to develop scaling techniques, which led to an explosion of cryptographic research. The most promising of these developments in providing scaling for blockchain are the many PCP constructions discovered in the past decade.

2.3 PCP History

This spotlight on cryptography led to numerous breakthroughs over the years, particularly for PCPs which were previously considered to be too computationally expensive to be used in practice since their conception in 1986 [22].

Of particular importance is a PCP variant called non-interactive zero-knowledge (NIZK) that was popularized in the blockchain community. Of these proofs, ZK-SNARKs [10] (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) is one of key importance because it became the basis of the Zcash protocol [42]. Enabled by ZK-SNARKs, Zcash allows for monetary transactions between two parties to be publicly verified without revealing any information about the involved members, ensuring confidentiality and computational integrity through the use of ZK-PCPs.

While preserving privacy is a nice feature, one could argue that the *S* (succinct) in ZK-SNARKs is more important than the *ZK*. Succinctness is the characteristic that allows one to verify a proof in exponentially faster time than it takes to generate it, which will be longer than running the original computation. This is an extremely favorable characteristic in today's era of third party cloud computing. PCPs will allow for the integrity of a computation run by trusted or untrusted third party to be verified very cheaply.

For computations that only need to be produced once and verified many times, the extra proving overhead can be considered a fixed cost, and all subsequent verifiers will benefit from fast and cheap verifications, thereby saving compute cycles and electricity. Computations like updating account balances after transferring cryptocurrencies fit this bill. For this reason, PCPs are one of the most hopeful contenders for producing the scalability desired by blockchains. Instead of each node in the decentralized blockchain redundantly recomputing the account balances after a transaction to check for fraud, each node only needs to verify the succinct proof attesting to the transaction's correctness. Applications for succinct verification are plentiful across domains beyond blockchain, with the goal being to verify general purpose computations.

In 2018, the successor of ZK-SNARKs, ZK-STARKs [6] were published which improved upon the security at the cost of an increased proof size. The main contributions are two-fold. First, ZK-STARKs eliminated a required trusted setup that was a single point of failure of ZK-SNARKs. Second, ZK-SNARKs depend on the use of public key cryptography, relying on elliptic curve pairings. The security behind this is rooted in the hardness of the discrete log problem (DLP), as are most of the public key encryption algorithms that we use in our daily lives. The idea is that as long as the DLP is hard to compute, meaning it is infeasible regardless of one's budget and compute power to try and crack it, the proof is secure.

One criticism is that, although impractical now, when quantum computers are more viable in the future, they can be used to break problems like the DLP that the classical computers of today cannot [44]. As a result, for not only ZK-SNARKs, but for much of the security underlying blockchain, it is important in the future to look at post-quantum secure cryptographic assumptions. ZK-STARKs are attractive because they rely only on the hardness of breaking hash functions, which are not made easier by quantum computers, thus making ZK-STARKs post-quantum secure.

2.3.1 Provers and Verifiers

PCPs are comprised of two parties, a prover and verifier. They are designed to allow the verifier to verify the correctness of a computation in exponentially

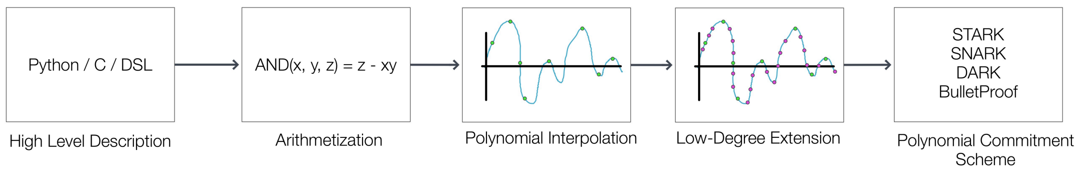


Figure 2.1: The high-level transformations from a problem into a PCP. Note that these graphs are shown over the familiar real numbers because the idea of evaluating over a larger domain is visually more intuitive, but in practice the polynomial values are modulo q. The Low-Degree Extension evaluates the interpolated polynomial on a much larger domain (i.e green vs purple dots).

less time than it takes to run the the original computation, the caveat being that the time for the prover to run the computation grows poly-logarithmically with the size of the computation. More concretely, if a computation originally took 100 days on a supercomputer, it would take approximately 700 days to generate a proof contesting to the computations correctness, and less than 7 days to verify (where the proof only ever needs to be generated once).

Because the security of these constructions are built solely on cryptography and math, there is no need to assume trusted hardware is used or that the prover is acting honestly. This means that even if a known bad actor is the prover, a verifier will accept the output if and only if the proof is correct, with only a negligible probability of them generating a fraudulent but passing proof. The verifier is able to achieve a sublinear verification time because the proofs are probabilistically checkable, implying that there is some probability of a fake proof passing, but this probability is a tunable system parameter chosen to be negligent, such as 2^{-128}.

2.3.2 How PCPs Work

Since 2018, an explosion of new PCP constructions has developed under different crypto-systems, but every construction depends on first converting the problem that is being proven into a problem based on polynomials. When the problem is described in the language of polynomials, one can then apply the rules of polynomials to construct efficient proofs.

Arithmetization

For example, to prove that after a certain number of steps the output of a computation is correct, first convert each instruction of code into its respective Boolean gates (AND, OR, NOT, etc). The behavior of a gate is a function of its inputs and can thus be written as an equation. For example an AND gate can be written as the function: $AND(z_1, z_2) = z_1 * z_2$ for $z_1, z_2 \in \{0, 1\}$. We can rewrite this function to encode a successful execution of the gate as whether the function evaluates to 0 or not: $F_{AND}(z_1, z_2, z_3) = z_3 - AND(z_1, z_2)$, where z_3 is the output [46]. $F_{AND}(0, 0, 1) = 1 - 0 = 1$, is an incorrect execution of the gate, while $F_{AND}(1, 1, 1) = 1 - 1 = 0$, is a correct execution.

If the function for *every* gate in the program evaluates to 0, then we know that the computation was run correctly. This process of converting from a gate to a function that evaluates to 0 if and only if it's a correct execution is called Arithmetization and the collection of gates is called a Boolean Circuit. In this scenario the verifier must check every arithmetized gate for correctness, which in practice may be more expensive than just running the original equation.

Error Correction Codes

In order to reduce the verifier's load, polynomials and Error-Correcting Codes (ECCs) are used. The gates are encoded in a polynomial, that will evaluate to 0 at all points only if the computation was correct. For example, the AND gate execution can be encoded as a line (which is a degree-1 polynomial), over some variable t: $L(t) = (z_3 - z_1 * z_2) \cdot t$. This line will evaluate to 0 for all t if and only if there is a correct execution of the AND gate. An incorrect execution will cause the line to only pass through the origin once (i.e. evaluate to 0 only once). This idea can be extended to polynomials of any degree and generalizes from individual gates to entire circuits, where the polynomial will evaluate to 0 for all t, if and only if the circuit is executed correctly [46].

To check a circuit's correctness it requires checking that the polynomial evaluates to 0 at *all* t. Checking every point is required because if the original polynomial evaluated to 0 at all but a single point, the verifier will accept the incorrect circuit execution if this point is not checked. However, testing every value of t is too costly, taking the verifier as much time as the original computation. To avoid this, the prover encodes the polynomial into a much larger polynomial. This operation is called Low-Degree Extension, and it involves evaluating the polynomial on a larger domain, such as 8-16x more evaluations of t.

The resulting extended polynomial is called a Reed-Solomon Codeword (RSC) [36]. The RSC, amplifies errors in the original polynomial (i.e. points that do not evaluate to 0). In exponentially less checks than all values of t, the RSC will reveal whether any of the gates were executed incorrectly.

Each PCP construction follows the above steps, although there are far more expressive languages than Boolean Circuits such as Arithmetic Circuits, R1CS, QSP, QAP, SSP, TinyRAM, and RAM which also are efficient for Zero Knowledge [8], [20], [7], [18], [9], [49].

Polynomial Commitments

The next step is polynomial commitment, and this is where each construction differs in the math and cryptography used. A commitment scheme is where one locks in a set of values so that they cannot be changed in the future, which helps ensure provers to do not cheat during the protocol. STARKs use hash-based commitments, SNARKs use pairing groups, DARKs [13] use groups of unknown order, and Bulletproofs [12] rely on the discrete log assumption (i.e. there is no known algorithm besides brute force to solve for x in $r^x \equiv a \mod m$). Each commitment scheme has different tradeoffs in a variety of metrics like the size of the proof, the cryptographic assumptions used, the transparency, post-quantum resistance, and whether a trusted setup is necessary. It is the verifier's job to query points on the extended polynomial for consistency, and if the prover were to know those points in advance, or be able to reply with consistent answers generated on the fly, then the prover would be able to deceive the verifier. Therefore, randomness is used to avoid the prover knowing the points in advance, and a commitment scheme is necessary to prevent the prover from changing their polynomial during the verification process.

PCP Summary

The above can be broadly summarized into three steps: problem encoding, RSC formation, and polynomial commitment. The problem encoding will be different depending on the expressiveness of what one is trying to prove, like whether the program can use memory or control flow or whether it is restricted to only arithmetic operations. And as mentioned previously, the polynomial commitment scheme will vary depending on what construction is used, whether STARKs, SNARKs, DARKs, or Bulletproofs, all of which have different tradeoffs. An important insight, however, is that the intermediate step of forming the RSC will be uniform across different problems and commitment schemes. The computations involved in this step lend extremely well to hardware and thus is the main target for Buntterfly.

2.4 History of FHE

Second in this, beginning in 2009, Craig Gentry published a paper [21] describing the first practical solution to the fully homomorphic encryption problem which was first proposed in 1978 [39]. FHE allows one to perform computations on encrypted ciphertext, which when decrypted, contains the same result as if the computations were applied to the underlying plaintext. Applications include genomics research on medical data that cannot be viewed due to privacy, and put more generally, data analysis on private data. If encrypted data could be analyzed, useful medical predictions can be extrapolated without the risks involved with compiling private unencrypted data where it can be hacked.

A key component to Gentry's work is that it focuses on lattice-based cryptography, and problems based on lattices are contenders to be post-quantum hard problems. Thus while the attention that cryptocurrencies brought to cryptography resulted in more pressure for post-quantum assurances, in parallel, work started by Gentry has propelled plausibly post-quantum lattice constructions into the spotlight.

2.5 How FHE Works

Lattice-based cryptography, particularly the Ring Learning with Errors (Ring-LWE) problem, involves multiplying together two large polynomials [27]. The data being encrypted is mixed with the secret key and hidden across the coefficients of a very large polynomial with large coefficients in the hundreds of bits. The sheer size of the polynomial combined with added noise, makes it impractical to extract the underlying data out of the polynomial without the secret key. The polynomial is closed under addition and multiplication, and with those two operations, arbitrary computations can be constructed. The party performing these additions and multiplications cannot distinguish between the noise and the plaintext, but can still perform meaningful changes. While addition of large polynomials is relatively cheap, multiplication does not scale well.

The naive way polynomial multiplication is taught in schools requires a complexity of $O(n^2)$ where n is the number of coefficients in a polynomial. This complexity is prohibitive at the scale that FHE must operate. Instead, to multiply the two polynomials, one can first perform the Number Theoretic Transform (NTT) on each input polynomial to get them in the evaluation domain, point-wise multiply each value, and then perform the inverse NTT to interpolate the result back into the coefficient domain, resulting in the same output polynomial as if traditional multiplication were used, but with a much better runtime of $O(nlog(n))$ [34]. The use of the NTT algorithm allows for FHE to be practical, as FHE computations may involve million-bit multiplications [19].

2.6 Prior FHE Work

While applications utilizing PCPs and FHE have started to become practical because of the developments in the past decade, there are still many opportunities to optimize the algorithms used in their construction. A key operation in both cryptographic primitives is large polynomial interpolation and evaluation, and thus the NTT is the algorithm of choice [34]. Prior work has focused on building NTT hardware accelerators for polynomial multiplication for the R-LWE problem with different target platforms in mind. For example in Poppelmann et al. [33], latency is their priority and they resort to precomputing twiddle factors and storing them in RAM. Conversely in Roy et al. [41], the goal was a low area design, and thus twiddle factors are computed on the fly. Most recently HEAX provides an NTT architecture to accelerate the CKKS FHE scheme, which has promising results for machine learning in a fully homomorphic environment [37].

2.7 Number Theoretic Transform

R-LWE and PCPs both work with polynomials defined over finite fields. Fast Fourier Transforms (FFT) have been around since 1965 and is used for converting signals between different domains with a low computational complexity of $O(nlog(n))$ [17]. Most of the prior work on building FFTs for hardware has been for FFTs in the complex plane.

When defined over finite fields, FFTs are called Number Theoretic Transforms (NTTs) because they exploit primitive roots of unity in finite fields instead of complex numbers to quickly perform number theoretic transforms. These terms will be described in the following sections. NTTs allow one to quickly interpolate data into a polynomial in the coefficient domain, or evaluate points on a polynomial back into data in the evaluation domain.

2.7.1 NTT Overview

NTTs have two operations: evaluation and interpolation. Given a polynomial, one could represent it as a list of coefficients $(4x^0 + 3x^1 + 2x^2 + 1x^3 = [4, 3, 2, 1])$. Performing an n-point NTT in the forward direction (which is just referred to as a NTT), one will evaluate the polynomial at n special points. Performing an n-point inverse NTT (INTT) on these evaluations will interpolate the values back into their original coefficients.

At first glance this may not seem very useful, but it has significant implications for many kinds of computing. For example, the multiplication of two integers can be performed using NTTs. By encoding the integers as two polynomials (i.e. 1234 can be encoded as $4 + 3x + 2x^2 + 1x^3$ in base-10), the NTT operations can be applied. Multiplication of two polynomials can be done by evaluating the polynomials at n special points, point-wise multiplying the resulting n evaluations, then interpolating the results. The result can then be decoded back to the integer and the result is the same as if the two integers had been multiplied together using typical multiplication. For very large integers this is faster than standard multiplication techniques, despite the high overhead. This same sequence of operations is how the NTT is used to perform polynomial multiplication for FHE in the R-LWE setting.

PCPs use Reed-Solomon codes, which can be formed using NTTs as follows:

- INTT to interpolate n-data points into a polynomial p(x).
- NTT to evaluate p(x) at $(n*m)$-points where m is referred to as the blowup factor.
- Now any n random points from (0, n*m) should be able to recover p(x)

2.8 NTT Math

All computations will be done in the realm of finite fields. We will be using modular arithmetic or clock arithmetic, where every value is bounded by a prime integer q, and values larger than q will wrap around. Let $q = 5$:

- $0 \mod 5 = 0$
- $1 \mod 5 = 1$

- $2 \mod 5 = 2$
- $3 \mod 5 = 3$
- $4 \mod 5 = 4$
- $5 \mod 5 = 0$
- $6 \mod 5 = 1$

2.8.1 Order of a Group

The modulus q forms a field with elements [0, q-1]. A multiplicative group excludes the *0* element because it is non-invertible, so the number of unique elements is denoted as the order of a group q: $|q| = q - 1$. NTTs rely on special primes that are composed of high powers of 2. For example, a Fermat Prime, which is a very special case, is of the form $2^m + 1$ [35].

An example is $q = 257 = 2^8 + 1$. Such a prime is chosen because $|q| = 256$, a power of 2. Lagrange's Theorem [40] tell us that for any random integer ω that we choose within the field, the order of the multiplicative subgroup |G| generated from ω will divide $|q|$. Since $|q| = 256$, $|\mathrm{G}| \in \{1, 2, 4, 16, 32, 64, 128, 256\}$, because all of these values divide 256.

A multiplicative subgroup is the set of unique elements that are generated from any random integer in the field, ω, called the generator. The way these elements are generated is by computing $\omega^i \mod q$ for $i \in [0, |G|]$.

For example: let $\omega = 193$. This generator will produce a multiplicative subgroup that cycles between 8 unique elements:

- $193^0 \mod 257 = 1$
- $193^1 \mod 257 = 193$

- $193^2 \mod 257 = 241$
- $193^3 \mod 257 = 253$
- $193^4 \mod 257 = 256$
- $193^5 \mod 257 = 64$
- $193^6 \mod 257 = 16$
- $193^7 \mod 257 = 4$
- $193^8 \mod 257 = 1$

Notice raising ω to the 8'th power causes the generated value to cycle back to 1. This will repeat for all subsequent powers of ω . In this example, $n = |\mathrm{G}| = 8$, and ω is a primitive n'th or 8'th root of unity because it generates 8 unique elements.

2.8.2 Picking Primes

NTTs require n to be a power of 2 due to the nice symmetries that it provides. This means q must be restricted to have certain qualities, namely high-2-adicity. The Fermat Primes, like 257, are ideal because they are composed only of powers of 2, but the highest known Fermat Prime is $2^{16} + 1$, which is too small for many problems. However, other composite primes are good candidates. For example $q = 3 * 2^{30} + 1$ is a prime such that following Lagrange's Theorem, will guarantee generators that produce multiplicative subgroups with orders $2^i : i \in \{0, 30\}$, and thus is suitable for NTTs up to $n = 2^{30}$.

2.8.3 Fourier Transform

Polynomial evaluation, in middle school algebra, solves for y by plugging in a value for x: $y(x) = 4x^0 + 3x^1 + 2x^2 + 1x^3 \Rightarrow y(2) = 4 + 6 + 8 + 8 = 26)$. Multi-point polynomial evaluation, the operation performed by the forward NTT, is the same operation, just performed over a sequence of x-values: $[y(0), y(1), y(2), y(3)] = [4, 10, 26, 58]$. The only caveat is that the x-values are the n elements in our multiplicative subgroup, and our math is all done modulo q:

For example, let $q = 257$ and $\omega = 241$. $\langle\omega\rangle$ = G $\in$ {1, 241, 256, 16} $\rightarrow$ |G| = $n = 4$.

Let y(x) = $4x^0 + 3x^1 + 2x^2 + 1x^3$

- $y(1) = 10$
- $y(241) = 4{*}241^0 + 3{*}241^1 + 3{*}241^2 + 1{*}241^3 = 4 + 723 + 116162 + 13997521 = 14114410 \mod 257 \equiv 227 \mod q$
- $y(256) \equiv 2 \mod q$
- $y(16) = 34 \mod q$

Thus the results in the evaluation domain are $[y(1), y(241), y(256), y(16)] = [10, 227, 2, 34]$

Notice the evaluation points are essentially random values in the field. This might not appear to be very useful, but it should be remembered that the goal of this operation is to transform from the polynomial domain into the evaluation domain, so operations like multiplication can easily be performed. Also, while seemingly random, these powers of ω have useful symmetry that allows for a more efficient algorithm to transform between domains.

The way of evaluating in this example requires multiplying all n coefficients by all n powers of ω. This $O(n^2)$ complexity is a hindrance for large values of n, and so a faster algorithm to achieve multipoint evaluation is required.

2.8.4 Speeding up the Fourier Transform

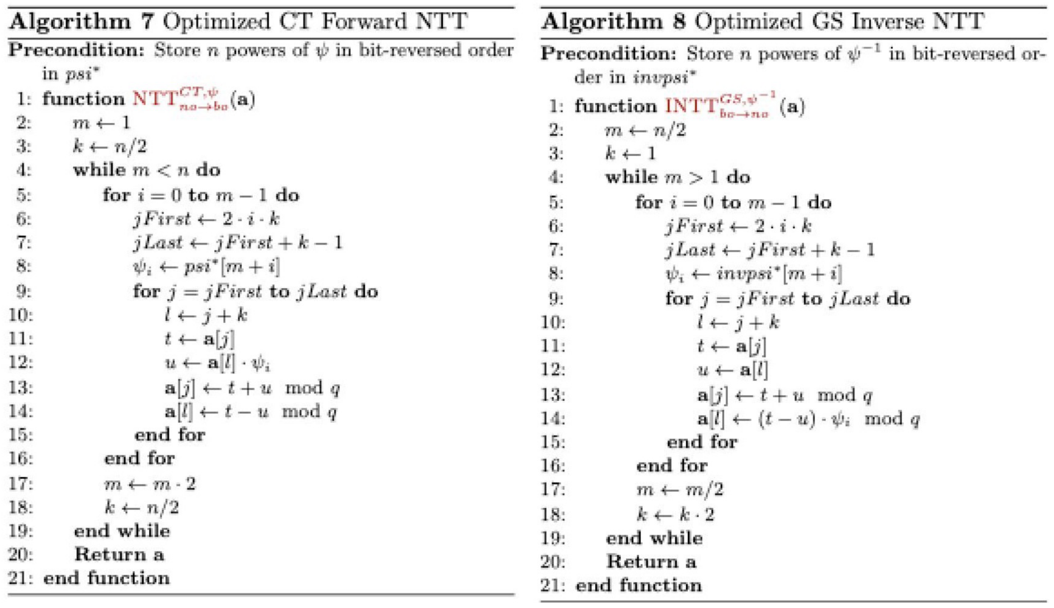
To borrow from the terminology from the original FFT paper [17], from now on the n powers of the n'th root of unity will be referred to as the twiddle factors. Because |G| is a power of 2, there is a symmetry between twiddle factors in the first half and second half, namely the i'th twiddle factor is the negative of the $i+n/2$'th twiddle factor. Borrowing the twiddle factors from the previous example, we can see $1 \equiv -1 * 256 \mod 257$, and $241 \equiv -1 * 16 \mod 257$.

One can exploit this symmetry to construct an algorithm with less than n^2 multiplications. A fact about polynomials is one can be evaluated by splitting it into even and odd coefficients and evaluating as follows [14]:

$y(x) = evens(x^2) + x * odds(x^2)$ and $y(-x) = evens(x^2) - x * odds(x^2)$

- Continuing the example: $y(x) = 4x^0 + 3x^1 + 2x^2 + 1x^3 \rightarrow evens(x) = 4x^0 + 2x^1$ and $odds(x) = 3x^0 + 1x^1$.
- In reals: $y(2) = evens(2^2) + 2 * odds(2^2) = (4 + 8) + 2 * (3 + 4) = 26$
- In finite fields: $y(241) = evens(241^2) + 241 * odds(241^2) = (4 + 116162) + 241 * (3 + 58081) = 14114410 \mod 257 = 227 \mod q$
- Exploiting the symmetry: $y(-241) = evens(-241^2) - 241 * odds(-241^2) = (4 + 116162) - 241 * (3 + 58081) = 34$

Notice evaluating $y(-241)$ yields the same value as $y(16)$ because these twiddle factors are negatives of each other. Also notice that $evens(241^2) = evens(-241^2)$,



Algorithm 7 Optimized CT Forward NTT

Precondition: Store n powers of ψ in bit-reversed order in psi^*

```
function NTT$^{CT,\psi}_{no \to bo}$(a)
    $m \leftarrow 1$
    $k \leftarrow n/2$
    while $m < n$ do
        for $i = 0$ to $m - 1$ do
            $jFirst \leftarrow 2 \cdot i \cdot k$
            $jLast \leftarrow jFirst + k - 1$
            $\psi_i \leftarrow psi^*[m + i]$
            for $j = jFirst$ to $jLast$ do
                $l \leftarrow j + k$
                $t \leftarrow \mathbf{a}[j]$
                $u \leftarrow \mathbf{a}[l] \cdot \psi_i$
                $\mathbf{a}[j] \leftarrow t + u \mod q$
                $\mathbf{a}[l] \leftarrow t - u \mod q$
            end for
        end for
        $m \leftarrow m \cdot 2$
        $k \leftarrow n/2$
    end while
    Return a
end function
```

Algorithm 8 Optimized GS Inverse NTT

Precondition: Store n powers of ψ^{-1} in bit-reversed order in $invpsi^*$

```
function INTT$^{GS,\psi^{-1}}_{bo \to no}$(a)
    $m \leftarrow n/2$
    $k \leftarrow 1$
    while $m > 1$ do
        for $i = 0$ to $m - 1$ do
            $jFirst \leftarrow 2 \cdot i \cdot k$
            $jLast \leftarrow jFirst + k - 1$
            $\psi_i \leftarrow invpsi^*[m + i]$
            for $j = jFirst$ to $jLast$ do
                $l \leftarrow j + k$
                $t \leftarrow \mathbf{a}[j]$
                $u \leftarrow \mathbf{a}[l]$
                $\mathbf{a}[j] \leftarrow t + u \mod q$
                $\mathbf{a}[l] \leftarrow (t - u) \cdot \psi_i \mod q$
            end for
        end for
        $m \leftarrow m/2$
        $k \leftarrow k \cdot 2$
    end while
    Return a
end function
```

Figure 2.2: NTT Algorithms used in Buntterfly from [33].

and similarly $odds(241^2) = odds(-241^2)$. This implies there are many redundant multiplications being performed, that only need to be performed once.

One could minimize the number of multiplication operations by efficiently scheduling the order in which twiddle factors are multiplied by these coefficients. In practice this is done using recursion, but for hardware NTTs, iterative algorithms are required. The result is that multipoint evaluation can be done in $O(nlog(n))$ multiplications (likewise for the inverse operation, polynomial interpolation).

2.8.5 Chosen NTT Algorithms

Figure 2.2 shows the NTT and INTT algorithms used in Buntterfly. The core operation being performed is referred to as the butterfly operation in the original FFT paper [17] because of the resemblance to a butterfly as shown in Figure 2.3. Figure 2.4 demonstrates the scheduling of an 8-point forward NTT using the

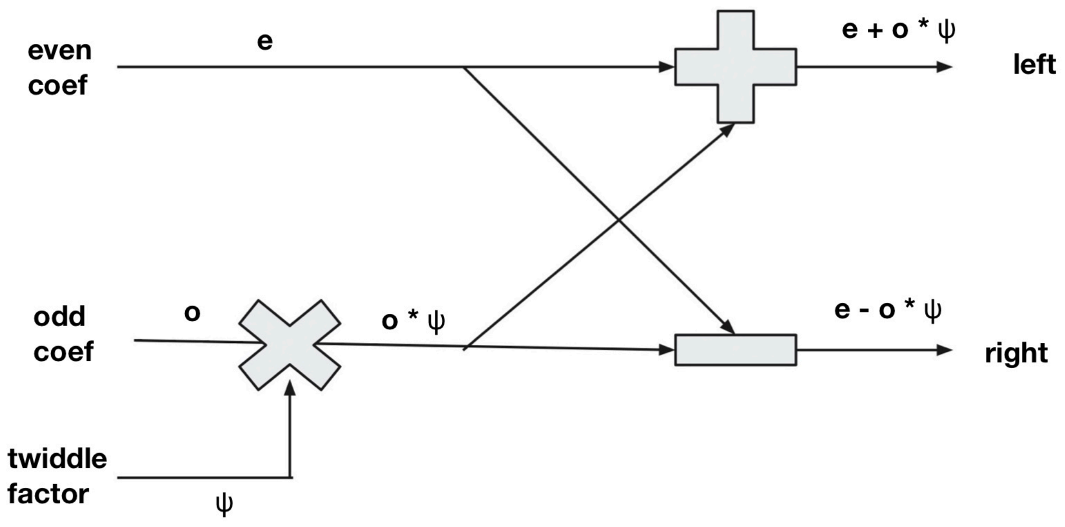


Figure 2.3: The classic *butterfly operation*. The ordering of the operations in this figure are for the forward NTT, corresponding to lines 13 and 14 in Figure 2.2.

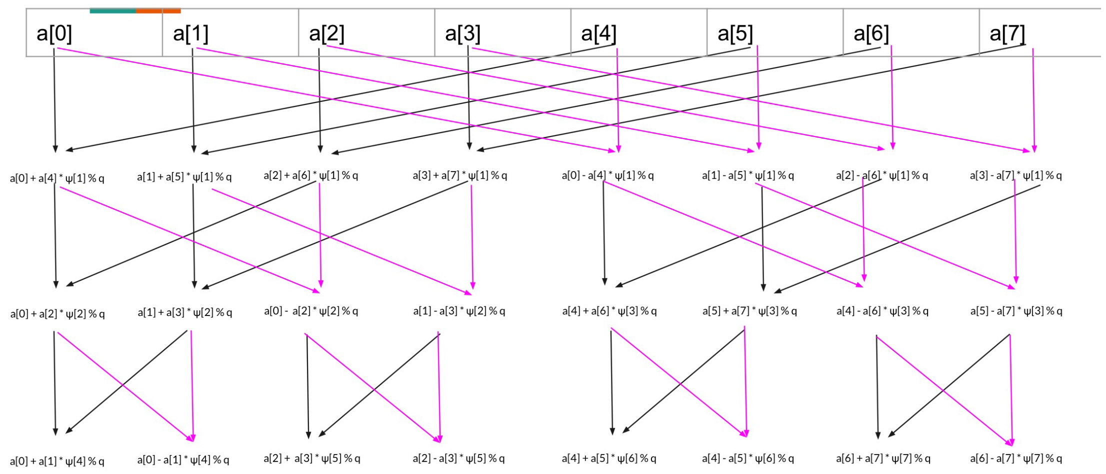


Figure 2.4: An 8-point NTT has $log_2(8) = 3$ levels. Each pair of two arrows denotes a butterfly operation, the black arrows being the even plus the odd times twiddle factor, and the purple arrows being the even minus the odd times twiddle factor.

algorithm shown in 2.2.

One caveat of these algorithms is that instead of ω, a new symbol is used for the twiddle factors, ψ. The difference is that instead of ψ being an n'th root of unity, it is a $2n$'th root of unity, meaning the multiplicative subgroup generated from ψ contains twice as many elements as ω. This is an important optimization for applications like FHE, where the intended use of the NTT is for multiplication. Using a $2n$'th root of unity employs negative-wrapped convolution, which prevents the number of coefficients from doubling during multiplication [47].

Chapter 3

Buntterfly

3.1 System Overview

Buntterfly is a flexible circuit generator capable of creating specialized hardware optimized for accelerating the NTT algorithm, which can be deployed to an ASIC or FPGA for use. Users can specify the prime modulus, the number of NTT points, and the desired parallelism, and Buntterfly will generate a pipelined NTT circuit that can compile down to synthesizable Verilog. The high-level features of the hardware construction language Chisel allow for Buntterfly to easily be extended to suit certain problems. Alternatively the reusability that hardware generators provide means future projects can treat Buntterfly as a blackbox and use it as a library component.

3.2 Chisel

Chisel is a relatively new hardware construction language for designing hardware [4]. Embedded in the programming language Scala (a functional JVM language), Chisel allows hardware engineers to leverage the high-level features that allow software engineers to rapidly develop scalable code. Such features include object-oriented programming, static-time compilation, strong typing, and access to the variety of mature Java libraries that already exist. All of this lends well to code reuse and Agile development, which allowed a graduate student to develop an out of order processor using a fraction of the time and resources as an industrial team in just three years [50].

Hardware description languages like Verilog describe the behavior of a circuit over time. Chisel works differently by instantiating hardware components through the use of Scala. The wiring between the components is then described using Chisel. Thus at a high level, a hardware generator is simply a Scala function that leverages Chisel internals to generate hardware components. These generators provide an interface to instantiate different designs based on how the generator is parameterized.

For example, if one were to design an adder in Chisel, one would first describe the behavior of the design in terms of wires and gates. This behavior can be wrapped in a Scala function that parameterizes the Adder based on the bit-widths of the adder inputs. This function can now generate any sized Adder depending on the intended use. Perhaps multiple different Adder designs can be included in this module, each with its own tradeoffs in performance, energy, and area. Depending on the target platform, one could instantiate an adder by parameterizing according to their target metric. All of these decisions happen during elaboration time. Once the design is transformed to synthesizable Verilog to be placed on an FPGA or ASIC, only the target adder is used, and none of the other adder designs are generated in hardware.

Generators allow for more code reuse and smaller code bases. By providing clean interfaces, components can easily be swapped, providing flexible designs and is a step towards open source hardware designs.

When designing the NTT accelerator, Chisel made it easy to generate different circuits by reparameterizing the modulus, bit-widths, and number of NTT points.

3.3 Implementation Decisions

The following are important decisions made while implementing Buntterfly in Chisel and will be detailed throughout this chapter:

- Using Poppelmann's algorithms for the NTT [33].
- Using the Ozturk Modular Multiplier as the core modular multiplier unit [31].
- Keeping all of the intermediate values in k1d1 form.
- Pre-computing the coefficient addresses.
- Keeping negative numbers when in the inverse direction.
- Choosing a centralized control approach as opposed to a distributed approach.

3.4 NTT Algorithm Prior Work

Prior works on NTT hardware accelerators have implemented designs based on a variety of algorithms. Aysu et al. [3] and Roy et al. [41] propose designs with area in mind, choosing to compute the twiddle factors on the fly. The major contribution from Roy et al. is this is the first mention of optimizing negative wrapped convolution in the NTT setting as opposed to FFT. The insight is that by using ψ as a 2n'th root of unity, one can remove the coefficient growth that arises during polynomial multiplication, keeping the polynomial fixed at n-points. The cost, however, is that the resulting coefficients will be scaled by ψ, requiring n multiplications by ψ^{-1} to recover the correct coefficients. Roy et al. modified the NTT algorithm in the forward direction by scaling each coefficient by ψ with zero extra cost (replaced the round where each coefficient is multiplied by 1 with ψ).

Poppelmann et al. built on the previous algorithm, allowing for a similar technique but in the inverse direction, requiring a decimation-in-frequency approach as opposed to decimation-in-time [33]. Due to the access patterns of the algorithm, pre-computed powers of ψ and ψ^{-1} are stored in hardware.

Longa et al. used the same algorithms from Poppelmann but contributed their own modular reduction algorithms [26]. Similarly, in this work, Buntterfly uses the same algorithms from Poppelmann et al., but uses the algorithm from Ozturk et al. [31] to build the system's core-multiplier, accelerating the modular multiplication and reduction operations.

3.5 Architectural Overview

Figure 3.1 illustrates the full system integration for Buntterfly. The sequential steps can be broken down as follows (assume all enable signals are high):

1. The AddrGenUnit produces address triplets: *even addr*, *odd addr*, and *twid addr*.

2. *odd coef* and *even coef* are fetched from *CoefMem* and *twiddle factor* is fetched from *TwiddleFactorMem*.

3. These values are concurrently pipelined through the *DestAddrFIFO*, *ButterflyUnit*, and *ButterflyInverseUnit* respectively.

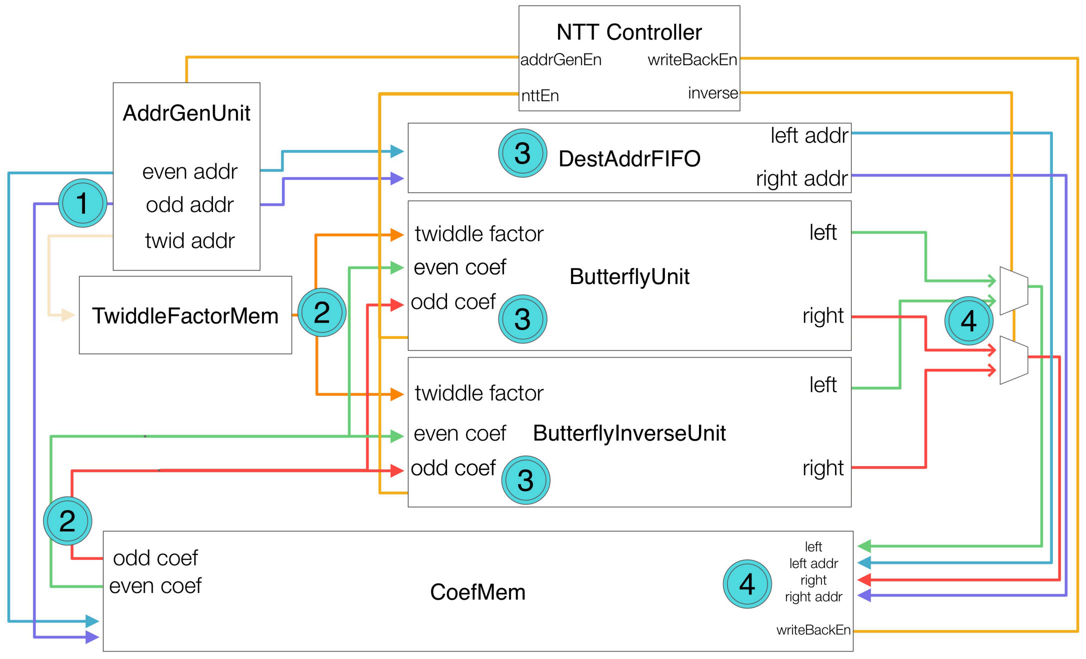


Figure 3.1: Buntterfly data and control paths. (Buntterfly is capable of instantiating multiple parallel ButterflyUnits and ButterflyInverseUnits, but this is omitted for succinctness).

4. The results *left* and *right* are mulitiplexed depending on whether the NTT or INTT is being performed, and then written back to *CoefMem* at *left addr* and *right addr*.

3.6 Modular Multiplier

Modular multiplication finds the product of two numbers and returns the remainder after dividing by the modulus. An example being $3 * 4 \mod 7 = 5$ since $12/7 = 1$ with remainder 5. Previously we saw our twiddle factors in the form $\omega^i \mod q$, which is example of what is called modular exponentiation. This operation is simply many repeated instances of modular multiplication, and is a common operation in cryptography.

Modular multiplication is difficult for two reasons. First, there is the problem of bit-growth. Any multiplication between two integers doubles the width of the larger integer in the worst case. Since we must design around the worst case, this is considered a rule of thumb when designing hardware. This means that repeated multiplications will cause an exponential blowup in the number of bits, which is impractical to consider in hardware. The solution to this is to perform the modular reduction (modulus operation) after each step, which will truncate each intermediate product to a maximum of the modulus width. The second problem, however, is that division is hard in computers and should be avoided when possible. Unfortunately this is unavoidable, but algorithms have been developed to address this.

Prior techniques for modular multiplication classically include using Montgomery Modular multipliers or Barrett reductions [29], [5]. Montgomery multipliers employ a special Montgomery form where division is circumvented by repeated additions and few subtractions. Thus Montgomery multipliers excel for modular exponentiation because the cost of converting to and from the Montgomery form is amortized across many intermediate modular multiplications. The main application for these works has been RSA crypto-systems [38], where modular exponentiation is the most common operation.

3.6.1 Ozturk Modular Multiplier

In 2019 Ozturk et al, designed a low-latency modular multiplication algorithm [31]. Intended applications for this multiplier are verifiable delay functions (VDF), which are a promising cryptographic primitive with interesting implications for blockchain [11]. The security behind VDFs involves performing modular exponentiation as fast as possible, and thus low-latency modular multiplication algorithms are necessary.

Ozturk's design was used to break the LCS35 time-lock puzzle in only 60 days. This timelock was intended to take 35 years, even taking into account Moore's law; however, vast improvements in low-latency modular multipliers were not taken into account [16]. While it is more expensive in terms of area, the multiplier's low-latency and ability to be pipelined made it a suitable choice to use as the core modular multiplier in Buntterfly.

k1d1 Form

Just as the Montgomery multiplier uses a special form, the Ozturk multiplier employs a redundant form during intermediate multiplications. In this paper, we introduce the terminology "k1d1 form" to refer to this redundant form. The k in k1d1 is the number of coefficients in a polynomial, and d is the bit-width of each coefficient. Any n-bit integer can be encoded as a polynomial with k coefficients that are each d-bits wide by first selecting a value for d. The radix r, such that $r = 2^d$ is the maximal value that any single coefficient can be. Since the modulus is the largest integer in the system, $n = numBits(modulus)$. The number of coefficients needed to encode the integer can be found as $k = \lceil n/d \rceil$. Binary and Hexadecimal encodings are everyday examples of such representations.

For example representing an integer as a base-16 polynomial would work as follows: given $a = 1532$ and $n = numBits(1532) = 11b$. In Hex, each term is 4-b wide, so let $d = 4$. The resulting radix $r = 2^4 = 16$ (each term is from [0,15]). The number of coefficients $k = \lceil 11/4 \rceil = 3$ coefficients. Thus a can be encoded as a degree 2 polynomial with 4-bit coefficients: $p_a(x) = 5x^2 + 15x + 12$. One can see this is equivalent to the hex encoding of a: $hex(a) = 0x5fc$, when one substitutes the radix: $p_a(radix = 16) = 5(16^2) + 15(16^1) + 12(16^0) = 1532$.

Selecting different values for d allows for arbitrarily based polynomials besides base-2 and base-16. When in this form, coefficients can accumulate numbers even greater than the radix whilst still representing the same number. For example the following are redundant representations of 1532: $1532 = 5(16^2) + 15(16) + 12 = 0(16^2) + 95(16^1) + 12(16^0) = 0(16^2) + 0(16^1) + 1532(16^0)$.

The k1d1 form deviates very slightly from this. The n-bit integers are still broken down into k, d-bit coefficients; however, one additional coefficient and one additional bit per coefficient are added for redundancy. The resulting polynomial will still have radix $r = 2^d$, but will contain $(k+1)$ coefficients, each $(d+1)$-bits wide, hence k1d1. A valid k1d1 representation for a would be: $p_a(radix = 16) = 5(16^2) + 14(16^1) + 28(16^0) = 1532$. Notice the coefficients are $\leq d + 1 = 5$-bits, but the radix is still $r = 2^d$. The largest k1d1 polynomial with $k = 3$ and $d = 4$ would be: $p(radix = 16) = 31(16^3) + 31(16^2) + 31(16^1) + 31(16^0) = 135439$

The reason for choosing such an encoding is further explained in [31], but can be summarized at a high-level: With the goal being modular multiplication of two integers, one can first encode the integers as k1d1 polynomials. Multiplying the two polynomials can be done using a variety of techniques such as (Schoolbook, Comba, Karatsuba, Strassen, NTTs, and hybrids of these [15], [25], [45],[34]). The Ozturk multiplier employs standard $O(n^2)$ schoolbook multiplication because all $(k+1)^2$ partial products can be computed in parallel (although exploring different multiplication algorithms with better asymptotic runtimes combined with the k1d1 form is something we wish to try in the future). Accumulating all $(k+1)^2$ partial products results in bit-growth for the resulting polynomial, but by carrying, reducing via Look Up Tables (LUTs), accumulating, and finally carrying again, the resulting product is returned to the same k1d1 form and can be used for another round of modular multiplication without any expensive reductions using division. Without this k1d1 form, intermediate polynomial products would grow

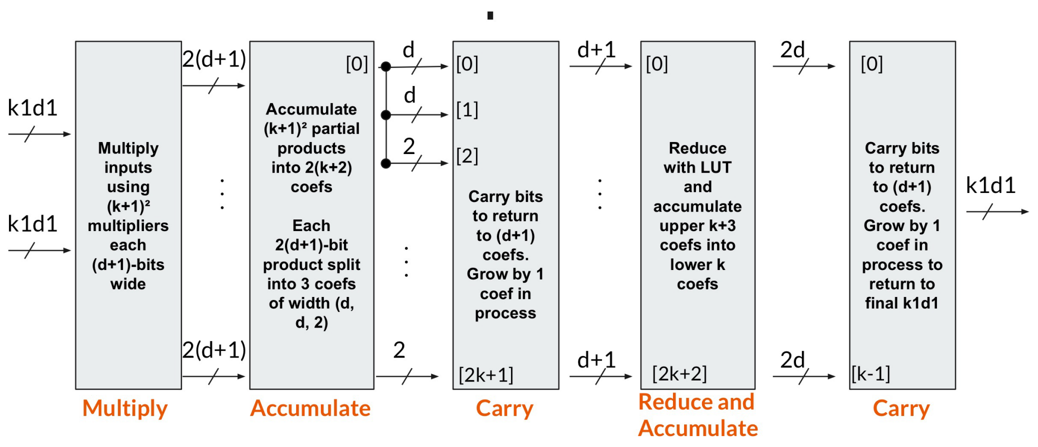

k1d1
k1d1
Multiply inputs using (k+1)² multipliers each (d+1)-bits wide
2(d+1)
2(d+1)
[0]
Accumulate (k+1)² partial products into 2(k+2) coefs
Each 2(d+1)-bit product split into 3 coefs of width (d, d, 2)
d
d
2
2
[0]
[1]
[2]
[2k+1]
Carry bits to return to (d+1) coefs. Grow by 1 coef in process
d+1
d+1
[0]
[2k+2]
Reduce with LUT and accumulate upper k+3 coefs into lower k coefs
2d
2d
[0]
[k-1]
Carry bits to return to (d+1) coefs. Grow by 1 coef in process to return to final k1d1
k1d1
Multiply
Accumulate
Carry
Reduce and Accumulate
Carry

Figure 3.2: The steps involved to multiply two k1d1 polynomials modulo a prime number and return the product to k1d1 form.

between rounds or need more reductions, thereby lengthening the critical path of the design.

One contribution from Bunterfly was noticing that the Ozturk multiplier can be used to perform not just the core modular multiplications in the NTT's butterfly operation, but all the arithmetic operations by keeping all intermediate values in the k1d1 form. Thus the k1d1 form can be used for the entire NTT computation, amortizing the cost of converting to and from this form.

Ozturk Multiplier Overview

Figure 3.2 demonstrates the bit-width and polynomial degree transformations during each step of the Ozturk modular multiplication algorithm. The steps are summarized briefly below and in more detail in the Bunterfly Unit section.

- Multiply - Compute $(k+1)^2$ partial products by multiplying the two k1d1 input polynomials.

- Accumulate - Accumulate each $2(d+1)$-bit partial product amongst three coefficients by splitting partial product into [2:d:d] bits.

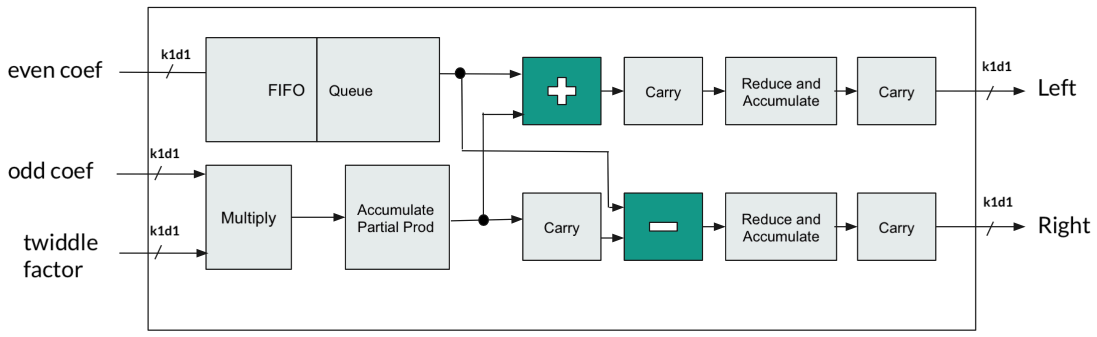


Figure 3.3: *ButtterflyUnit* Chisel module. The Ozturk multiplier has been adapted to include the addition and subtraction operations required in the forward butterfly operation.

- Carry - Carry the accumulated result, reducing bit-widths to $(d + 1)$ but growing one coefficient to $(2k + 3)$.
- Reduce - Use lookup tables (LUTs) to reduce each of the upper $(k+3)$ coefficients. Finds each kd polynomial corresponding to an evaluated coefficient modulo q.
- Accumulate - Accumulate each of the looked up kd polynomials with the lower k coefficients not looked up in the previous step.
- Carry - Carry to return the result to k1d1 form. It has now undergone a modular multiplication without the need for reduction via the modulus operator.

3.7 Buntterfly Unit

The Ozturk multiplier is integrated into a ButterflyUnit and ButterflyInverse-Unit Chisel modules. Figure 3.3 is a picture showing the pipeline stages for the forward direction.

PolyMAC

The first step in a forward butterfly operation is to multiply an odd input by a twiddle factor (power of ψ). All inputs and twiddle factors are stored in k1d1 form. Using $O(n^2)$ schoolbook multiplication, $(k+1)^2$ partial products (PP) are computed in parallel. Since multiplication doubles bit-width, each PP will have grown to $2(d+1)$-bits. The PP's need to be accumulated back into a single polynomial, and the Ozturk algorithm does this in a unique way.

Instead of accumulating $2(d+1)$-bit PPs to their respective destination coefficient, each PP is broken into three pieces: the lower d-bits, the next d-bits, and the upper 2-bits, and then is accumulated into three destination coefficients. This is where the redundancy from polynomial representations is useful. The accumulation of PPs could just as well have been done as learned in school, but by splitting it this way, an equivalent product is achieved, but one with smaller coefficient bit-widths that is more suitable to be reduced.

The PolyMAC Chisel module performs the above steps by first multiplying and saving PPs to registers. Chisel bit-manipulations are used to extract out the lower, middle, and upper bits. Filtering, mapping, and folding are all functional Scala operators that at compile-time are used to coordinate which PP-bits are added together. The end result is that during runtime, the $(k+1)^2$ PPs are accumulated into $2(k+1)$ coefficients, each $(2*d)$-bits wide (The accumulated partial products will be referred to as APP).

Addition then Carry

One may notice from Figure 3.3, that the even coefficient does nothing during both cycles of the PolyMAC stage. Instead, a FIFO pipelines the even input for the two cycles so that it can be added with the APP. In its current form, the

APP contains twice as many coefficients as the even input. One can safely add the lower $(k+1)$ coefficients of the APP with the $(k+1)$ coefficients of the even input. The result is equivalent to the unreduced product of an odd coefficient and twiddle factor added with the even input, which is the desired left output of the butterfly operation. However, all NTT operations are done modulo q, and this output is not in the desired k1d1 form to be used for the next round of the NTT.

The CarryChain Chisel module is the first step in reducing back to k1d1. This module brings each coefficient down to $(d+1)$-bits, but grows the polynomial by one coefficient to a total of $2(k+1)+1$. The result, even plus odd times twiddle, will be referred to as EPOTT.

Carry then Subtraction

The butterfly operation requires the APP from the PolyMAC to be subtracted from the even coefficient. The $2(k+1)$, $(2*d)$-bit coefficients are carried to form a $(2(k+1)+1)$ coefficient polynomial with $(d+1)$-bit coefficients. Since the even input will have the same width coefficients of the same sign, when subtracting there is no bit-growth. The result (even minus odd times twiddle - EMOTT) will be in the same form as the EPOTT.

Reduce and Accumulate

Both the EMOTT and EPOTT work in lockstep from this stage on. Ozturk was able to achieve low-latency because of the techniques used for modular reduction. Instead of performing division or the modulus operator on the EMOTT and EPOTT, precomputed values are stored in tables called lookup tables (LUTs), and fetched from to perform the reduction.

The goal is to return the polynomials back into k1d1 form, but the polynomials contain $2(k+1)+1$ coefficients. The upper $(k+3)$ need to be truncated to bring the polynomial back to k coefficients, so one more carry can bring the result to k1d1.

Each of the upper $(k+3)$ coefficients precomputes its own LUT. This requires computing 2^{d+1} possible values, each weighted by the LUT's corresponding coefficient's degree.

For example, if $d = 4$, and this LUT is for the $(k+1)$'th coefficient where $k = 3$, the LUT precomputes $radix = 2^d = 16$ values. Each value is scaled by the radix raised to the coefficient's position: 16^4, and then reduced $\mod q$ using the standard remainder operation. This reduced value is then saved as a polynomial with k coefficients, each d-bits wide. The overall operation is summarized as: $LUT_{coefPos} = i * radix^{coefPos} \mod q : i \in [0, radix)$

One reduces each of the upper $(k+3)$ coefficients in the EMOTT and EPOTT by querying the respective LUTs with the contained coefficient values (each from $[0, radix)$), and then accumulating the kd LUT outputs with the lower k coefficients in the EMOTT and EPOTT. The results of the reduction and accumulation stages are k coefficients, each $2 * d$-bits wide.

Final Carry

To get the two reduced and accumulated polynomials into the final k1d1 form, a final carry grows the polynomials by one coefficient, and reduces each coefficient to $(d+1)$-bits. At this point, the left and right outputs will be suitable for another round of the NTT. One should note however, that the final result will need to be reduced one more time after the completion of the NTT. The purpose of using the k1d1 form is not complete modular reduction, but to keep intermediate results from blowing up (solving the bit-explosion problem) whilst avoiding costly

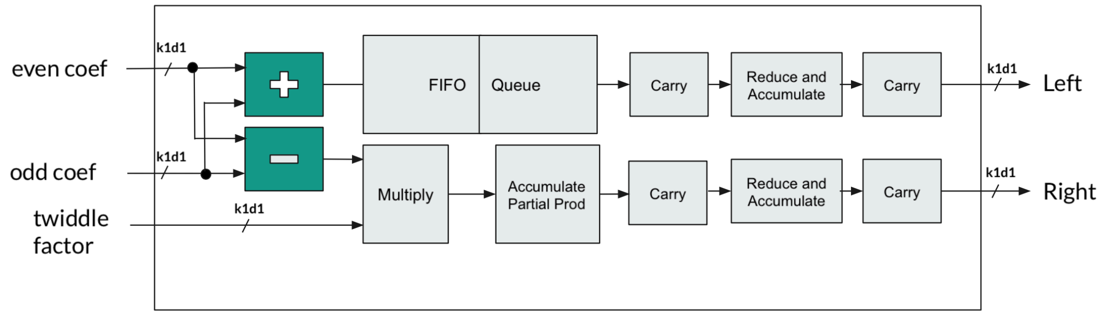


Figure 3.4: *BuntterflyInverseUnit* Chisel module. The inverse butterfly requires different operations than the forward direction shown in Figure 2.2.

division.

3.8 Inverse Buntterfly Unit

Figure 3.4 shows the pipeline stages for the butterfly operation in the inverse direction. Many of the stages are identical to the forward Buntterfly Unit. However, in Poppelmann et al. [33], in order to merge powers of ψ in the inverse direction, a decimation-in-frequency approach is taken. This results in a slightly different ordering of the arithmetic operations. It is required to first subtract the odd input from the even input before multiplying by the twiddle factor.

A lesson learned while implementing this is to keep any negative differences in negative form. It is tempting to convert to a positive number by adding the modulus. For example: $-1 \mod 257 = -1 + 257 \mod 257 = 256$. The problem with this example is the result grew to the bit-width of the modulus. If all $(k+1)$ coefficients are negative after subtracting the odd input from the even input, the resulting polynomial can potentially grow to $(k1n1)$ which is impractical for large n.

The even minus odd (EMO) input to the PolyMAC will contain a mix of positive and negative coefficients. Fortunately this is not a problem. The CarryChain

Module can handle both positive and negative coefficients. The only other difference between the forward and inverse butterfly units is during the reduction phase. A positive and negative LUT are multiplexed depending on the sign of the coefficient. This step corrects all coefficients to positive integers for the next round of the INTT.

3.8.1 Address Generator

The AddrGenUnit is a necessary component that could easily be decoupled from the rest of the NTT computation. There are $log(n)$ sequential levels in a NTT or INTT. Each level requires $n/2$ butterfly operations, where each operation requires an even, odd, and twiddle factor input. The AddrGenUnit is responsible for generating the addresses for each of these inputs to be fetched from memory.

As shown in 2.2, the triple nested for-loop is not conducive to being done in hardware. This is where Chisel excels. Since the access patterns are known in advance, all of the addresses are generated using Scala at compile-time. These addresses are then saved to hardware read-only-memory (ROM) in the order they are accessed.

Because the addresses are stored in the order they are consumed, no complex control needs to be implemented to fetch them from ROM. Instead just two counters are used, one indexes through the ROM to return address triplets, and another tracks the current level.

The interface for the AddrGenUnit allows it to be parameterized based on the number of parallel NTT units that will be used in the overall system. Every clock cycle it returns *numNTTUnits* number of address triplets to be fetched from memory as input to the *numNTTUnits* number of parallel BuntterflyUnits and BuntterflyInverseUnits.

3.8.2 Twiddle Factor Generator

The access pattern in 2.2 requires the input coefficients to be in standard order and the twiddle factors in bit-reversed order. Bit-reversed implies reversing the bit-value of the twiddle factor's index.

For example, if $n = 16$, requiring 4-bits to index, the twiddle factor stored at index 1: $0b0001$ becomes $0b1000$, or the 8'th element. If $n = 32$, requiring 5-bits to index, $0b00011$ becomes $0b11000$, and certain indices can also stay the same: $0b00100$ remains $0b00100$.

Requiring bit-reversed twiddle factors makes calculating them on the fly difficult. Typically if one want's to do that, they will use different NTT algorithms with bit-reversed input coefficients and standard ordered twiddle factors, which is more conducive to straightforward modular exponentiation. These designs are flexible and more area efficient as vastly less storage is required for the pre-computed twiddle factors, at the cost of energy and latency. For future work, we wish to add this as an option when parameterizing the NTT to support area-constrained, embedded devices. An Ozturk multiplier can easily be pipelined to deliver twiddle factors in k1d1 form each cycle to be consumed by the butterfly units.

Currently, Buntterfly requires all twiddle factors to be pre-computed and stored in ROM. The n twiddle factors are calculated in Scala at compile time and store in bit-reversed order to a Chisel Vec of Wire. They are accessed via the address from the AddrGenUnit.

3.8.3 NTT Controller

Everything is coordinated via a finite-state-machine (FSM) controller. Figure 3.5 visually describes the transitions between the five FSM states: **init**, **fill**,

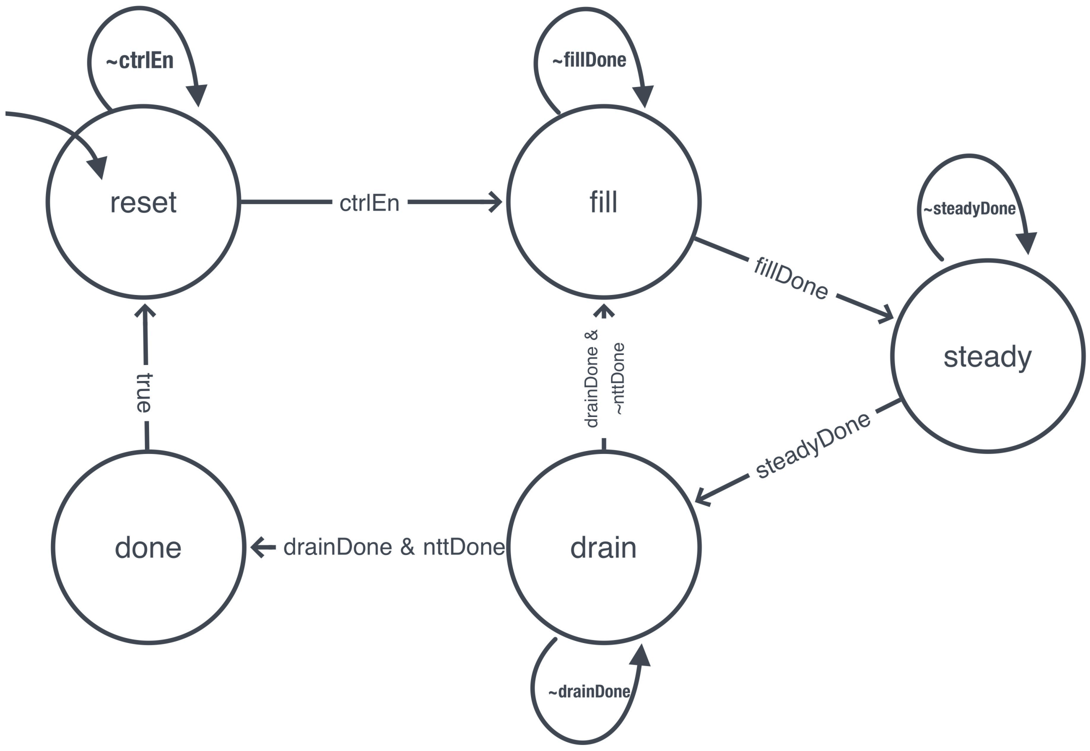


Figure 3.5: The states and state transitions to control the modules involved in a NTT/INTT computation.

state	addrGenEn	nttEn	writeBackEn	fillCntEn	steadyCntEn	drainCntEn
reset	0	0	0	0	0	0
fill	1	1	0	1	0	0
steady	1	1	1	0	1	0
drain	0	1	1	0	0	1
done	0	0	0	0	0	0

Table 3.1: Six control signals are generated by the FSM. *nttEn*, *addrGenEn*, and *writeBackEn* control top level modules, and *fillCntEn*, *steadyCntEn*, and *drainCntEn* control internal FSM counters for determining state transitions.

steady, **drain**, and **done**. It uses three internal counters and an enable signal to drive the FSM, producing enable signals for the NTT units, AddrGenUnit, and memory write-back. The primary goal for the FSM was to deal with the edge cases when filling and draining the pipeline.

- **init**: initialization, reset all signals, read in input coefficients to memory.
- **fill**: begin filling the pipeline without enabling write-back.
- **steady**: pipeline is fully utilized, NTT outputs are now valid, enable write-back.
- **drain**: stop entering values into pipeline, disable AddrGenUnit, and wait until it is full drained.
- **done**: finished the computation, disable all output control signals, enable **outReady** signal.

Chapter 4

Results

4.1 Evaluation Baseline: libfqfft

In order to get a sense of the relative impact that Buntterfly would have over a software based NTT library, the libfqfft [1] library was chosen as a baseline. This software NTT library is a component in libsnark [2], a popular open-source ZK-SNARK library developed by SCIPR Lab, an academic collaboration responsible for many cutting edge PCP developments. Written in C++, libfqfft is designed with performance in mind, to accelerate one of the most computationally expensive operations in PCPs, polynomial evaluation and interpolation.

Figure 4.1 contains benchmarks for libfqfft. The x-axis contains the number of NTT points, n from 2^{15} to 2^{20}, and the y-axis is the NTT runtime in seconds. This data was collected using a 2.40GHz, Intel Core i7-3630QM quad-core with 16GB RAM. The tests did not exceed 4 threads as performance decreased when the number of threads exceeded the number of physical cores, requiring hyper-threading.

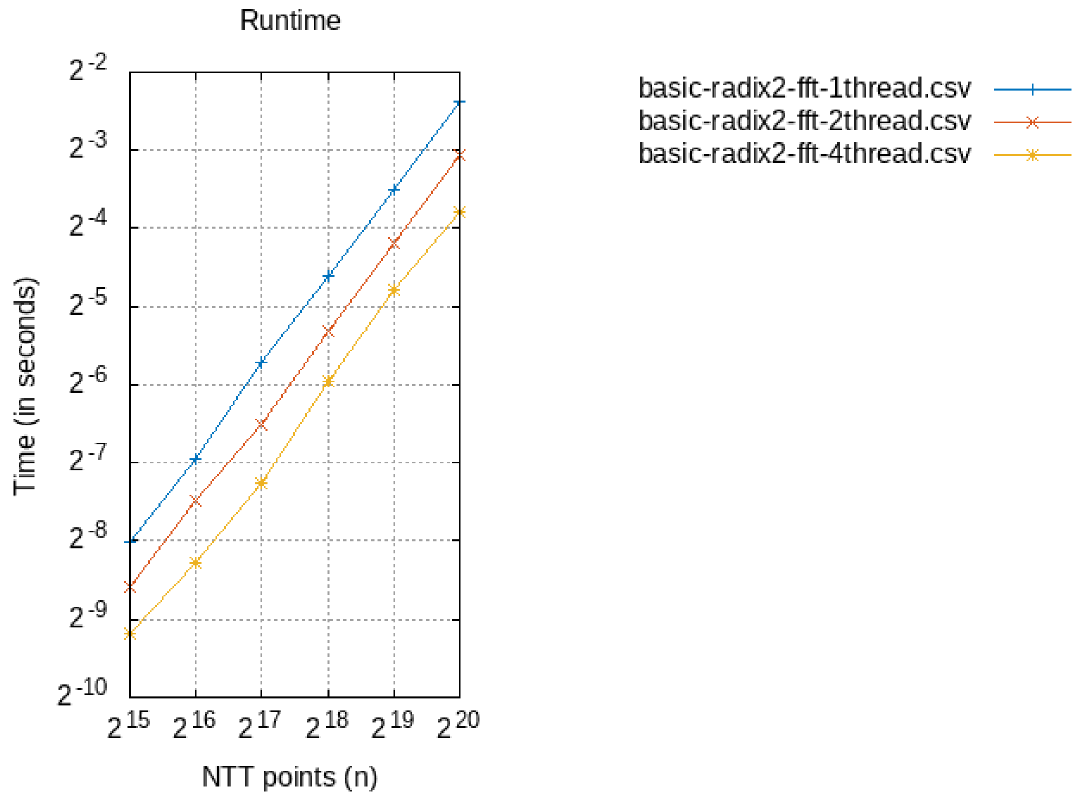


Figure 4.1: libfqfft NTT Benchmarks

4.2 Evaluation Method

Evaluating the correctness of Buntterfly is done using Chisel's built in tester: PeekPokeTester. The hardware module is instantiated and values written in software are used as input to the module. One can then step through in time, simulating clock cycles, and test that the module's outputs match expected software values. We directly compare the module's output with a Scala NTT implementation that we wrote first.

While Buntterfly successfully executes software simulations of the generated designs, these results should only be used to validate the circuit's correctness and to judge performance in cycles. At this stage of development, Buntterfly has not been run through the CAD toolchain and placed onto a FPGA, and so empirical data for clock frequency and area is not currently available.

However, it is possible to determine a theoretical runtime by determining the cycle count for a n-point NTT, and then multiplying it by a theoretical clock period. Each NTT requires $n/2$ butterfly operations per level with $log_2(n)$ total levels. The butterfly operations can be parallelized, so the number of operations per level is $(n/2)/m$, where m is the number of parallel butterfly units.

The total pipeline has nine stages, one for fetching the addresses, one for fetching data, six for the butterfly operation, and one for write-back. Thus the total cycles per level is: $cycles_{level} = cycles_{fill} + cycles_{steady} + cycles_{drain} = 9 + ((n/2)/m - 9) + 9$. The total cycles per NTT is: $cycles_{total} = log_2(n) * cycles_{level} = log_2(n) * (9 + (n/2)/m)$.

	$cycles_{total}$			
n	$m = 1$	$m = 2$	$m = 4$	$m = 8$
2^{16}	524432	262288	131216	65680
2^{17}	1114265	557209	278681	139417
2^{18}	2359458	1179810	589986	295074
2^{19}	4980907	2490539	1245355	622763
2^{20}	10485940	5243060	2621620	1310900
2^{21}	22020285	11010237	5505213	2752701
2^{22}	46137542	23068870	11534534	5767366
2^{23}	96469199	48234703	24117455	12058831

Table 4.1: The number of cycles required for Buntterfly to compute an n-point NTT given m-parallel butterfly units.

4.3 Theoretical Results

The theoretical runtime can be computed as $Time = cycles_{total} * T_{clk}$. Figure 4.2 compares the extrapolated runtimes for Buntterfly at two frequencies, 100MHz (i.e. FPGA) and 1000MHz (i.e. ASIC), compared to the baseline libfqfft results.

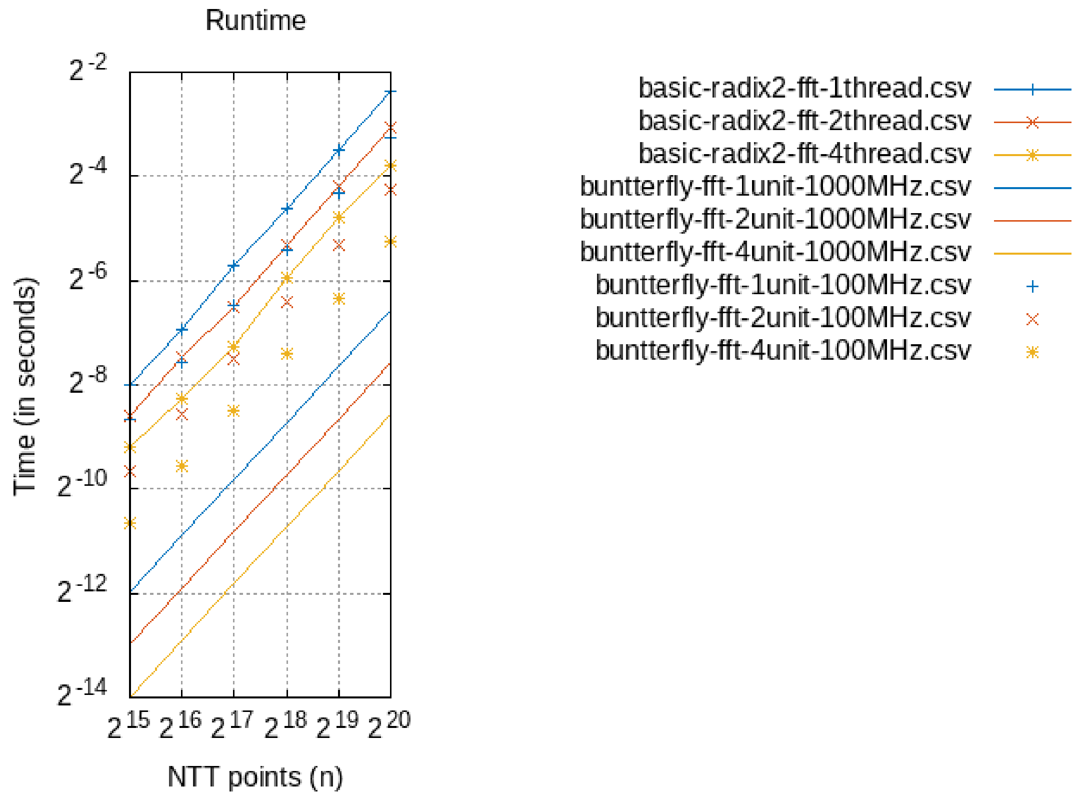


Figure 4.2: Theoretical Buntterfly Runtimes vs libfqfft Benchmarks

These results are promising for two reasons: the practicality of these runtimes, and the ease of adding more parallelism. If the theoretical clock period for the FPGA were at 100MHz, a single Buntterfly unit performs approximately 1.5x faster than the single threaded libfqfft NTT. When parallelism is considered, Buntterfly with four parallel butterfly units outperforms the four-thread libfqfft runtime by approximately 2.5x at 100MHz.

On the other extreme, at 1000MHz with no parallelism, Buntterfly outperforms the benchmark by 16x. When four parallel units are used at 1000MHz, Buntterfly outperforms the benchmark by 64x. These preliminary results are promising, as achieving a clock period in this frequency range is practical, although the limiting factor for the period will be when routing large designs with multiple parallel units.

The second reason why these results are promising is due to the differences in implementing parallelism in software and hardware. Adding additional butterfly units in hardware is just a matter of changing the $numNTTUnits$ parameter, and letting the Chisel generator handle the rest. More area and power are required, but in terms of added complexity, the additional parallelism comes for free. Although impractical due to area and clock frequency constraints, one can keep scaling the number of parallel units until $n/2$ units, resulting in maximum throughput.

For software NTT implementations, there may be diminishing returns when using more threads. This was the reality when evaluating the libfqfft benchmarks. The eight-threaded benchmark performed worse than the four-threaded benchmark because OpenMP performs optimally when threaded across different physical cores. Since this laptop only has four physical cores, OpenMP will run two hyperthreads per core which can cause adversarial behavior due to thread contention, hence the worse performance. Squeezing out more performance via parallel programming techniques is a much harder challenge than using hardware that is by nature parallel.

Chapter 5

Conclusion

5.1 Contributions

We made the following contributions:

- Created Buntterfly, a flexible hardware generator that produces NTT circuits parameterized by a prime modulus, and the number of NTT points.
- Built upon the state-of-the-art modular multiplier algorithm [22] to be compatible with the addition and subtraction operations needed during the NTT, amortizing the cost of switching to the specialized redundant form.
- Yielded preliminary results showing potential speedups up to 64x when using Buntterfly over optimized software NTT libraries.

The code for Buntterfly will be made open source and can be found at `https://github.com/ucsc-vama/fft-gen`.

5.2 Future Work

Future work for Buntterfly falls under two categories: internal optimizations and fully integrated designs. Potential changes to optimize the internal NTT computation include:

- Leveraging the object-oriented features of Scala to make designs more modular so one can substitute core components for ones with tradeoffs better suited for the target application. For example, one could replace the Ozturk multiplier with a different one (i.e Montgomery multiplier) as long as there are clean interfaces between the components.
- Exploring different underlying NTT algorithms besides Poppelmann's (i.e. generate twiddle factors on the fly for more area constrained devices).
- Use more sophisticated memory access schemes (i.e. store coefficients as pairs in memory to reduce the number of memory accesss).
- More hardware reuse with forward and inverse butterfly units (i.e. unified module that changes NTT direction based on runtime signal).
- More hardware reuse with LUTs, so one LUT can be used for many parallel butterfly units.

Some *glue* logic is needed to extend Buntterfly to be used for operations like polynomial multiplication. Potential additions to Buntterfly include:

- More control flow and arithmetic operations (i.e. scaling by n^{-1} and pointwise multiplication) to build a polynomial multiplier out of a NTT and INTT for use in FHE.

- Addition of modules besides polynomial multiplication to support a complete hardware implementation of FHE.
- More complex twiddle factor generation, memory management, and hardware reuse to perform Low-Degree Extensions for PCPs.
- Addition of modules for supporting different polynomial commitment schemes (i.e. a *hash* module for accelerating Merkle Trees during ZK-STARK proving).

Bibliography

[1] libfqfft: https://github.com/scipr-lab/libfqfft.

[2] libsnark: https://github.com/scipr-lab/libsnark.

[3] Aydin Aysu, Cameron Patterson, and Patrick Schaumont. Low-cost and area-efficient fpga implementations of lattice-based cryptography. In *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*, pages 81–86. IEEE, 2013.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[5] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in Cryptology—CRYPTO 86*, page 311323, Berlin, Heidelberg, 1987. Springer-Verlag.

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. 2018.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108. Springer, 2013.

[8] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for r1cs. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 103–128. Springer, 2019.

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 781–796, 2014.

[10] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography Conference*, pages 315–333. Springer, 2013.

[11] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[12] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[13] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 677–706. Springer, 2020.

[14] Vitalik Buterin. Fast fourier transforms, May 2019.

[15] Paul G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Syst. J.*, 29(4):526–538, 1990.

[16] Adam Conner-Simons. Programmers solve mit's 20-year-old cryptographic puzzle.

[17] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[18] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct nizk arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 532–550. Springer, 2014.

[19] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. Evaluating the hardware performance of a million-bit multiplier. In *2013 Euromicro Conference on Digital System Design*, pages 955–962. IEEE, 2013.

[20] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[21] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[22] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[24] Ioannis Karamitsos, Maria Papadaki, and Nedaa Baker Al Barghuthi. Design of the blockchain smart contract: A use case for real estate. *Journal of Information Security*, 9(3):177–190, 2018.

[25] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[26] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.

[27] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[28] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 253–260. IEEE, 2019.

[29] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[30] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[31] Erdinç Öztürk. Modular multiplication algorithm suitable for low-latency circuit implementations. *IACR Cryptol. ePrint Arch.*, 2019:826, 2019.

[32] Maksym Petkus. Why and how zk-snark works. *arXiv preprint arXiv:1906.07221*, 2019.

[33] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015.

[34] Ciara Rafferty, Máire ONeill, and Neil Hanley. Evaluation of large integer multiplication methods on hardware. *IEEE Transactions on Computers*, 66(8):1369–1382, 2017.

[35] I Reed, R Scholtz, Treiu-Kien Truong, and L Welch. The fast decoding of reed-solomon codes using fermat theoretic transforms and continued fractions. *IEEE Transactions on Information Theory*, 24(1):100–106, 1978.

[36] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[37] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. Heax: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1295–1309, 2020.

[38] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120126, February 1978.

[39] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[40] Richard L. Roth. A history of lagrange's theorem on groups. *Mathematics Magazine*, 74(2):99–108, 2001.

[41] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 371–391. Springer, 2014.

[42] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[43] Alesja Serada, Tanja Sihvonen, and J Tuomas Harviainen. Cryptokitties and the new ludic economy: how blockchain introduces value, ownership, and scarcity in digital gaming. *Games and Culture*, page 1555412019898305, 2020.

[44] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[45] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[46] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

[47] Franz Winkler. *Texts and Monographs in Symbolic Computation.* 01 1996.

[48] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[49] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vram: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 908–925. IEEE, 2018.

[50] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine.