FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	FPGA Acceleration of the Baby Variant of the WTFHE Scheme
Student:	Bc. Pavel Chytrý
Supervisor:	Dr.-Ing. Martin Novotný
Study program:	Informatics
Branch / specialization:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	until the end of summer semester 2022/2023

Instructions

Get acquainted with the Torus Fully Homomorphic Encryption (TFHE) scheme and its extension WTFHE (neural netWork-ready TFHE), which enables an evaluation of a neural network on encrypted input data. Design a parameterizable digital circuit that implements fundamental arithmetic operations in (W)TFHE. Compose the WTFHE bootstrapping operation out of these arithmetic operations. Implement the circuit in ZedBoard SoC platform (equipped with Xilinx Zynq-7000 chip). Explore the parameters' boundaries of the baby variant of WTFHE, s.t. the corresponding circuit fits into the target platform. Write a program implementing a simple neural network; the designed circuit shall serve as a hardware accelerator of the bootstrapping operation.

Electronically approved by doc. Ing. Hana Kubátová, CSc. on 10 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

FPGA Acceleration of the Baby Variant of the WTFHE Scheme

Bc. Pavel Chytrý

Department of Digital Design
Supervisor: Dr.-Ing. Martin Novotný

June 27, 2021

Acknowledgements

I would like to thank my supervisor Dr.-Ing. Martin Novotný for a continued guidance and support during writing of this thesis. I would also like to thank Ing. Jakub Klemsa for his explanation of the WTFHE scheme. And lastly, to my family for giving me space to fully fucus on finishing this thesis.

Declaration

Citation of this thesis

Chytrý, Pavel. *FPGA Acceleration of the Baby Variant of the WTFHE Scheme*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

S nárůstem cloudových výpočeních služeb je soukromí osobních údajů často v otázce, jelikož k nim má poskytovatel služeb plný přístup. Tuto situaci dále zhoršují zařízení, které mají přístup k soukromým datům uživatelů, ale postrádají výpočení sílu k provedení vlastního výzkumu - například nemocnice.

Jedním z řešení tohoto problému by mohlo být takzvané plně homomorfní šifrování (FHE), které dokáže vyhodnotit libovolnou funkci na šifrovaných datech bez nutnosti dešifrování na straně poskytovatele cloudových služeb. Oblast tohoto výzkumu je aktuálně velmi aktivní, s průlomem Gentryho et al. v roce 2009 a následným představením šifry TFHE od Chilloti et al.

Ukázalo se, že TFHE schéma je zvláště vhodné pro zabezpečení strojového učení jako služby (MLaaS). TFHE ve své původní podobě pracuje pouze s jednobitovým prostorem, avšak několik vylepšení umožňuje využití více hodnot. Tuto aktuální verzi jsme pracovně nazvali netWork-ready TFHE (WTFHE).

Obecně platí, že (W)TFHE šifry jsou o několik řádů pomalejší, než běžná šifrovací schéma. Tato práce je případová studie k určení urychlení výpočtu WTFHE za použití FPGA zařízení. Náš přínos spočívá v návrhu FPGA akcelerátoru schopného vyhodnotit jednoduchou neuronovou síť, změření jeho výkonu ve srovnání s softwarovým řešením, zjištěním hardwarových požadavků a jeho potenciál ve škálovatelnosti.

Klíčová slova WTFHE, TFHE, FPGA, VHDL, Kryptografie, Číslicový návrh, Plně homomorfní šifrování, Neuronové sítě

Abstract

With the rise of cloud compute services, the privacy of user's data is often put into question, as the service provider has full access to it. This is further exacerbated by facilities that hold private data, but lack the computational power to run their own research – namely hospitals.

A Fully Homomorphic Encryption (FHE) could be a solution to this problem as it can evaluate arbitrary functions over encrypted data without the need for decryption on the Cloud service provider's side. Since the breakthrough by Gentry et al. in 2009, this field is very active with Chilloti et al. recently introducing the scheme called TFHE.

TFHE scheme has been shown to be suitable for securing Machine Learning as a Service (MLaaS). TFHE in its original form only works with one-bit plaintext space, however, several improvements allow the usage of multivalue plaintext space. This improved version was codenamed netWork-ready TFHE (WTHE).

In general, (W)TFHE Schemes implemented in software are several orders of magnitude slower than the commonly used encryption schemes. This thesis serves as a case study to determine the feasibility of accelerating the WTFHE Scheme with an FPGA. Our contributions consist of designing an FPGA accelerator capable of simple Neural Network evaluation, measuring its performance compared to the software setup, discovering resource requirements, and the potential of scalability.

Keywords WTFHE, TFHE, FPGA, VHDL, Cryptography, Digital Design, Fully Homomorphic Encryption, Neural Network

Contents

List of Figures

List of Tables

Introduction

Currently, there are many Cloud service providers offering Machine Learning as a Service (MLaaS) as a commercial service – Google [3], Microsoft [4], and IBM [5] to name a few.

However, concerns were raised towards data privacy in respect to MLaaS. MLaaS providers not only have access to user's data and their prediction models, but can even train their prediction models with the provided data without the users being aware of it.

Recently, few cryptographic schemes employing homomorphic encryption have emerged. Homomorphic schemes can evaluate an arbitrary function on the encrypted data without the need of decryption. Approaches by Graepel et al. [6] or Xie et al. [7] build on the so-called Leveled Homomorphic Encryption (LHE) – but those schemes limit the depth of non-linear operations, which would be detrimental to Deep Neural Network evaluation.

This issue was addressed by Bourse et al. [8] by improving upon the TFHE scheme by Chilloti et al. [2]. The TFHE scheme was chosen as a basis for a modified version codenamed the neural netWork-ready TFHE (WTFHE). WTFHE can evaluate addition and arbitrary function over a multivalue plaintext space.

In general, (W)TFHE Schemes implemented in software are several orders of magnitude slower than the commonly used encryption schemes. This thesis serves as a case study to determine the feasibility of accelerating the WTFHE Scheme on an FPGA. Our contributions consist of designing an FPGA accelerator capable of simple Neural Network evaluation. We will measure the performance of the implemented design compared to a similar software setup and explore the limits of scalability of FPGA acceleration.

This thesis is split into several chapters. In Chapter 1, I will introduce the WTFHE scheme and all relevant definitions and algorithms. In Chapter 2, I will discuss possible approaches to hardware acceleration of the WTFHE scheme. In Chapter 3, I will describe the implementation steps of the hardware accelerator on an FPGA, while Chapter 4 will cover demo application written in C language running a simple Neural Network evaluation using the FPGA to accelerate a Bootstrapping operation. In Chapter 5, I will explain the methodology used to verify the functionality of the design. In Chapter 6, I will discuss the measured performance and resource requirements of the hardware accelerator. In Chapter 7, I will describe the direction of possible future work improving the usability of the WTFHE scheme. And finally, in the last Chapter I will summarize this work.

Chapter **1**

Preliminaries

In this section, I will describe the building blocks of the WTFHE encryption scheme. The definitions, algorithms, and excerpts in this section were provided by Jakub Klemsa. For a more detailed explanation I will refer readers to his publication [9] on modifications done to the TFHE or to Chilloti et al.[2] for the original proposition of the TFHE. Note that I modified some parts to keep consistent notation across this thesis and reflect the actual form of algorithms used for implementation.

Symbols & Notation

As this thesis contains a lot of technical content, I adopted symbols and notation from Klemsa's paper[9].

- $\mathbb{B}$ the binary Galois Field GF_2,
- $\mathbb{Z}$ the set of Integers,
- $\mathbb{T}$ the set $\mathbb{R}/\mathbb{Z}$, also known as the Torus,
- $p(X) = p^{(N-1)}X^{N-1} + p^{(N-2)}X^{(N-2)} + \ldots + p^{(1)}X^1 + p^{(0)}$ a polynomial, $\text{coeffs}(p) = (p^{(0)}, p^{(1)}, \ldots, p^{(N-1)})$,
- $M^{(N)}[X]^k$ the set of polynomials modulo $X^N + 1$, for a set M of dimension k and polynomial order of N,
- $\mathbb{T}^{(n)}$ vector of Torus members of length n,
- For a vector(matrix) of polynomials $\mathbf{w} = (w_0^{(N-1)}X^{N-1} + w_0^{(N-2)}X^{N-2} + \ldots + w_0^{(0)}X^0, \ldots, w_{n-1}^{(N-1)}X^{N-1} + w_{n-1}^{(N-2)}X^{N-2} + \ldots + w_{n-1}^{(0)}X^0)$, I denote:
  - $w_i = w_i^{(N-1)}X^{N-1} + w_i^{(N-2)}X^{N-2} + \ldots + w_i^{(0)}X^0$, and
  - $w_i^{(k)} = w_i^{(k)}X^k$

- $a \xleftarrow{\$} M$ and $a \xleftarrow{\alpha} M$ the uniform and the zero-centered α-deviated draw, respectively, of a random variable a from M.

The Torus The set $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ (i.e. real numbers modulo 1) with standard operation addition is called the Torus. Torus is a module over $\mathbb{Z}$, meaning its elements can be multiplied by an integer. These operations can be extended to Integer-Torus polynomials.

Concentrated Distribution Unlike multiplication, the division of a Torus by an integer cannot be defined without ambiguity, the same holds for the expectation of a distribution over the Torus. However, this can be fixed for a *concentrated distribution* [2], which is a distribution with support limited to a ball of radius $1/4$, up to a negligible subset. For further details, we refer to [2].

1.1 Introduction to TFHE

TFHE by Chillotti et al. [2] is a recent fully homomorphic scheme building upon Learning with Errors (LWE) problem [10]. TFHE in the original form encrypts a single bit, but improvements by Carpov et al. [11] and Bourse et al. [8] allow usage of multivalue plaintext space and DNN evaluation on the encrypted data. This improved version of TFHE was used as a basis of this thesis. I will refer to this improved version as neural netWork-ready TFHE (WTFHE in short).

The homomorphic properties of (W)TFHE can be written as follows:

$$\mathsf{TFHE}(a) \oplus \mathsf{TFHE}(b) \approx \mathsf{TFHE}(a+b), \text{ and} \tag{1.1}$$

$$\mathsf{eval}_f(\mathsf{TFHE}(a)) \approx \mathsf{TFHE}(f(a)) \tag{1.2}$$

where $\approx$ means "encrypts the same", and $\oplus$ and eval_f stand for homomorphic addition and evaluation of arbitrary function. As the TFHE has a randomized encryption scheme, each encryption will result in a different cipher text.

TFHE uses noise to mask plaintext values and guarantee cryptographic safety. Each $\oplus$ operation increases the internal noise up to a point, where correct decryption may not be guaranteed.

1.2 (W)TFHE Samples

WTFHE is a fully homomorphic cipher employing two encryption schemes: T(R)LWE and TRGSW. TLWE is used to encrypt plaintext, while TRGSW and TRLWE are used internally within the Bootstrapping procedure. Note that WTFHE (and TFHE) is a subject of still ongoing research and the algorithms used can be subject to future changes.

1.2.1 T(R)LWE

Definition 1 ($\mathsf{T(R)LWE}$ Sample [9])**.** Let $n \in \mathbf{N}$ be the *dimension*, $N \in \mathbf{N}$, $N = 2^\nu$ for some $\nu \in \mathbb{N}_0$ be the *degree*, $\alpha \in \mathbf{R}_0^+$ *standard deviation* and let the plaintext space $\mathcal{P} = \mathbb{T}^{(N)}[X]$, the ciphertext (Sample) space $\mathcal{C} = \mathbb{T}^{(N)}[X]^{(n+1)}$ and the key space $\mathcal{K} = \mathbb{B}^{(N)}[X]^{(n)}$. For $m \in \mathcal{P}$, we call $\mathbf{c} = (\mathbf{a}, b) \in \mathcal{C}$ the *TRLWE Sample* of message $m \in$ with key $\mathbf{k} \in \mathcal{K}$ if

$$b = m + \mathbf{k} \cdot \mathbf{a} + e, \tag{1.3}$$

where $\mathbf{a} \xleftarrow{\$} \mathbb{T}^{(N)}[X]^{(n)}$ and $e \xleftarrow{\alpha} \mathbb{T}^{(N)}[X]$. Further, for $\mathbf{a} = \mathbf{0}$, we call the Sample *trivial*, for $m = \mathbf{0}$, we call the Sample *homogeneous*, and for $N = 1$, we call the Sample the *TLWE Sample*.

Note that TRLWE sampling is actually *encryption*. For *decryption*, we apply *TRLWE phase* function (followed by rounding if applicable), definition follows.

Definition 2 ((T(R)LWE) phase [9])**.** Let n, N and α be the TRLWE parameters as per Definition 1, and let $\mathbf{c} = (\mathbf{a}, b)$ be a T(R)LWE Sample of m under a T(R)LWE key $\mathbf{k}$. We call the function $\phi_\mathbf{k} \colon \mathbb{T}^{(N)}[X]^{(n)} \times \mathbb{T}^{(N)}[X] \to \mathbb{T}^{(N)}[X]$,

$$\phi_\mathbf{k}(\mathbf{a}, b) = b - \mathbf{k} \cdot \mathbf{a}, \tag{1.4}$$

the *$\mathsf{T(R)LWE}$ phase*. Next, we call the Sample $(\mathbf{a}, b)$ *valid* iff the distribution of $\phi_\mathbf{k}(\mathbf{a}, b)$ is concentrated.

In general, the TRLWE phase function returns $m + e$, i.e., the plaintext with a (zero-centered) noise. The noise e (if sufficiently small) can be removed by appropriate rounding.

1.2.2 TRGSW

Unlike Torus polynomials in TRLWE, TRGSW encrypts integer polynomials. For the purposes of Bootstrapping, it defines so called *External Product*, $\boxdot \colon \mathsf{TRGSW} \times \mathsf{TRLWE} \to \mathsf{TRLWE}$, which is multiplicatively homomorphic on $\mathsf{TRGSW} \times \mathsf{TRLWE}$ Samples. Definitions follow.

Definition 3 (Gadget Matrix [9, 2])**.** Let $B_g = 2^\gamma$ for some $\gamma \in \mathbb{N}$ and $l \in \mathbb{N}$ be decomposition parameters and let N and n be TRLWE degree and dimension, respectively. We call

$$\mathbf{H} = \left(\begin{array}{c|c|c} 1/B_g & \dots & 0 \\ \vdots & \ddots & \vdots \\ 1/B_g^l & \dots & 0 \\ \hline \vdots & \ddots & \vdots \\ \hline 0 & \dots & 1/B_g \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1/B_g^l \end{array}\right), \tag{1.5}$$

$\mathbf{H} \in \mathbb{T}^{(N)}[X]^{(n+1)l,n+1}$, the *gadget matrix*.

Next, we recall the *Gadget Decomposition Algorithm* as Algorithm 1, which is—in this particular form—entangled with the gadget matrix $\mathbf{H}$.

Algorithm 1 Gadget Decomposition of a TRLWE Sample [9, 2]

Input: TRLWE Sample $(\mathbf{a}, b) = (a_1(X), \dots, a_k(X), b = a_{n+1}(X)) \in \mathbb{T}^{(N)}[X]^{n+1}$,
Input: decomposition parameters B_g, l.
Output: Vector of integer polynomials $\mathbf{d} \in \mathbb{Z}^{(N)}[X]^{(n+1)l}$.

```
1: for all $a_i(X) = \sum_{j=0}^{N-1} a_i^{(j)} X^j$, $a_i^{(j)} \in \mathbb{T}$, do
2:     $\bar{a}_i^{(j)} \leftarrow \lfloor B_g^l \cdot a_i^{(j)} \rceil$
3:     let $[\bar{a}_{i,1}^{(j)}, \dots \bar{a}_{i,l}^{(j)}]$ be a $B_g$-ary representation of $\bar{a}_i^{(j)}$ s.t. $\bar{a}_i^{(j)} = \sum_{p=1}^{l} \bar{a}_{i,p}^{(j)} B_g^{l-p}$
4: for $i = 1 \dots n+1$ and $p = 1 \dots l$ do
5:     $d_{(i-1)l+p}(X) = \sum_{j=0}^{N-1} \bar{a}_{i,p}^{(j)} X^j$
6: return d
```

Note, for the gadget matrix $\mathbf{H}$, quality $\beta = B_g/2$ and precision $\epsilon = 1/2B_g^l$, we denote the gadget decomposition algorithm as $\mathsf{Dec}_{\mathbf{H},\beta,\epsilon}(\mathbf{a}, b)$.

Definition 4 (TRGSW Sample [9, 2])**.** Let n, N and α be the parameters of TRLWE with key $\mathbf{k}$. We call $\mathbf{C} = \mathbf{Z} + m \cdot \mathbf{H}$ the *TRGSW Sample* of $m \in \mathbb{Z}^{(N)}[X]$ if each row of $\mathbf{Z}$ is an independent homogeneous TRLWE Sample under key $\mathbf{k}$, and we call m the message of $\mathbf{C}$, denoted $\mathsf{msg}(\mathbf{C})$. The phase of $\mathbf{C}$ is defined as the vector of the $(n+1)l$ TRLWE phases, denoted $\phi_{\mathbf{k}}(\mathbf{C})$, and the error of $\mathbf{C}$ is defined as the vector of the $(n+1)l$ TRLWE errors, denoted $\mathsf{Err}(\mathbf{C})$. We call $\mathbf{C} \in \mathbb{T}^{(N)}[X]^{(n+1)l,n+1}$ a *valid* TRGSW Sample under key $\mathbf{k}$ iff there exists $m \in \mathbb{Z}^{(N)}[X]$ such that each row of $\mathbf{C} - m \cdot \mathbf{H}$ is a valid homogeneous TRLWE Sample under key $\mathbf{k}$.

Definition 5 (External Product [9, 2])**.** For decomposition parameters β and ϵ, we define the *External Product*, $\boxdot\colon \mathsf{TRGSW} \times \mathsf{TRLWE} \to \mathsf{TRLWE}$, as

$$\mathbf{A} \boxdot \mathbf{b} = \mathsf{Dec}_{\mathbf{H},\beta,\epsilon}(\mathbf{b})^T \cdot \mathbf{A}, \tag{1.6}$$

where TRLWE is the underlying cipher of TRGSW.

1.2.3 Bootstrapping

The procedure named *Bootstrapping* aims at reducing the internal noise of a TLWE Sample to a fixed level. As originally devised by Gentry[1] in 2009, *Bootstrapping* internally runs decryption procedure with encrypted key bits referred to as the *Bootstrapping keys.* In addition, it is capable of function evaluation at no extra cost.

TFHE Bootstrapping consists of three algorithms: $\mathsf{BlindRotate}$, $\mathsf{SampleExtract}$ and $\mathsf{KeySwitch}$. I will list only their variants relevant to the implementation part of this thesis. As such, the TRGSW dimension (n) was fixed to a permanent value of 1.

The general evaluation of *Bootstrapping* is as follows. First, $\mathsf{BlindRotate}$ takes the Bootstrapped TLWE Sample and runs homomorphically a decryption-like procedure with encrypted key bits (referred to as the *Bootstrapping keys*). This way, it “blindly rotates” the second input – a TRLWE Sample, which encodes (and encrypts) the Bootstrapping function in a form of a Torus polynomial.

Next, $\mathsf{SampleExtract}$ is used to extract the constant term of TRLWE-encrypted polynomial back into a TLWE Sample. At this point, the Sample is usually encrypted with a different key, thus $\mathsf{KeySwitch}$ procedure should be called. The keys in this proof-of-concept demo were chosen such that the procedure was not needed, and so $\mathsf{KeySwitch}$ was omitted.

1.2.3.1 Blind Rotate

$\mathsf{BlindRotate}$ is the cornerstone of Bootstrapping since this is where homomorphic decryption takes place, i.e., where the noise is refreshed. It inputs the Bootstrapped TLWE Sample $(\mathbf{a}, b)$ in a scaled and rounded integer form $(\bar{\mathbf{a}}, \bar{b}) \in \mathbb{Z}^{n+1}$ (details to come later). In accordance with TLWE decryption (phase function, cf. Definition 2), $\mathsf{BlindRotate}$ internally calculates

$$-\bar{m} = -\bar{b} + \sum k_i \cdot \bar{a}_i, \tag{1.7}$$

where k_i's are TRGSW-encrypted under key $k'(X)$, referred to as the *Bootstrapping keys* and denoted by BK_i or $\mathsf{BK}_{\mathbf{k}\to k'}$. In $\mathsf{BlindRotate}$, the (hidden) value $-\bar{m}$ emerges as a power of X, by which the other input—a TRLWE Sample $(u, v) \in \mathbb{T}^{(N)}[X]^2$—is multiplied. The multiplicative homomorphic property is applied, hence (u, v) gets *blindly rotated.*

The (possibly trivial) TRLWE Sample (u, v) encrypts a Torus polynomial $tv(X)$, referred to as the *test vector*. Its Torus coefficients encode the (rescaled) Bootstrapping function $f\colon \mathbb{Z}_N \to \mathbb{T}$, for now and for simplicity, as

$$tv^{(k)} = f(k), \quad k \in [0, N-1]. \tag{1.8}$$

Enhancement of BlindRotate Zhou et al. [12] suggest to unfold the original BlindRotate loop, which multiplies the TRLWE Sample (u, v) one by one by $X^{k_i \bar{a}_i}$, cf. (1.7), and group the terms by two. Bourse et al. [8] further improve the technique by Zhou et al. by reducing the number of required encryptions of Bootstrapping keys from 4 to 3 (per pair of key bits).

For pairs (k, k') and (a, a') of consecutive elements of vectors $\mathbf{k}$ and $\mathbf{a}$, respectively, they write

$$X^{ka+k'a'} = kk'(X^{a+a'} - 1) + k(1-k')(X^a - 1) + (1-k)k'(X^{a'} - 1) + 1. \tag{1.9}$$

I.e., their Bootstrapping keys consist of TRGSW encryptions of kk', $k(1-k')$ and $(1-k)k'$ for each pair of bits of the global TLWE key $\mathbf{k}$. The improved algorithm is listed as Algorithm 2.

Algorithm 2 Blind Rotate Improved [2], improved by [9, 8, 12]

```
Input: TLWE Sample $(\mathbf{a}, b) \in \mathbb{T}^{(n+1)}$,
Input: TRLWE Sample $\mathrm{P} \in \mathbb{T}^{(N)}[X]^2$,
Input: TRGSW Samples $\mathrm{BK}_i \in \mathbb{T}^{(N)}[X]^{(2\cdot l),2}, i \in [1, 3/2 \cdot n]$,
Output: TRLWE Sample $\mathrm{ACC} \in \mathbb{T}^{(N)}[X]^2$.
1: $\bar{a}_i \leftarrow \lfloor 2Na_i \rceil$ for $i \in [1, n]$, $\bar{b} \leftarrow \lfloor 2Nb \rceil$
2: $\mathrm{ACC} \leftarrow X^{-\bar{b}} \cdot \mathrm{P}$
3: for $i \in [1, n/2]$ do
4:     $\mathrm{ACC} \leftarrow ((X^{\bar{a}_{2i-1}+\bar{a}_{2i}} - 1)\mathrm{BK}_{3i-2} + (X^{\bar{a}_{2i-1}} - 1)\mathrm{BK}_{3i-1} + (X^{\bar{a}_{2i}} - 1)\mathrm{BK}_{3i}) \boxdot \mathrm{ACC} + \mathrm{ACC}$
5: return ACC
```

1.2.3.2 Sample Extract

SampleExtract algorithm inputs the output of BlindRotate, which is a TRLWE Sample – let us denote it (r, s) (previously ACC). Recall that (r, s) encrypts the desired value at the constant term of its message under the key $k'(X) \in \mathbb{B}^{(N)}[X]$. As outlined, the goal of SampleExtract it to extract the constant term in a form of a TLWE Sample.

First, let us write down the constant term of the message of (r, s). After some rearrangements we get

$$m^{(0)} = s^{(0)} - \underbrace{(k^{(0)}, k^{(1)}, \dots, k^{(N-1)})}_{\text{new TLWE key } \mathbf{k}' = \mathrm{coeffs}(k')} \cdot (r^{(0)}, -r^{(N-1)}, \dots, -r^{(1)}). \tag{1.10}$$

It follows that $\big((r^{(0)}, -r^{(n-1)}, \ldots, -r^{(1)}), s^{(0)}\big)$ is a TLWE Sample, which encrypts $m^{(0)}$ under the key $\mathbf{k}' = \text{coeffs}(k')$. SampleExtract algorithm follows as Algorithm 3 (a slightly modified version of [2]).

Algorithm 3 Sample Extract [9]

```
Input: TRLWE Sample $(r, s) \in \mathbb{T}^{(N)}[X]^2$,
Output: TLWE Sample $(a_0, \ldots, a_{n-1}, b) \in \mathbb{T}^{(n+1)}$.
1: return $(\mathbf{a}, b) = \big((r^{(0)}, -r^{(n-1)}, \ldots, -r^{(1)}), s^{(0)}\big)$
```

1.2.3.3 WTFHE Bootstrapping

Previously described procedures allow us to properly define *Bootstrapping* in a form relevant to WTFHE scheme as Algorithm 4.

Algorithm 4 WTFHE Bootstrapping [9]

```
Input: TLWE Sample $(\mathbf{a}, b) \in \mathbb{T}^{(n+1)}$,
Input: TRLWE Sample $\mathrm{P\_IN} \in \mathbb{T}^{(N)}[X]^2$,
Input: TRGSW Samples $\mathrm{BK}_i \in \mathbb{T}^{(N)}[X]^{(2\cdot L),2}, i \in [1, 3/2 \cdot n]$,
Output: TLWE Sample $(\mathbf{a}', b') \in \mathbb{T}^{(n+1)}$.
1: $\mathrm{P\_OUT} \leftarrow \mathrm{BlindRotateIm}((\mathbf{a}, b), \mathrm{P\_IN}, \mathrm{BK})$
2: return SampleExtract(P_OUT)
```

1.2.4 DNN in WTFHE

In this section, I will outline the evaluation of Neural Networks in the WTFHE scheme. An *(Artificial) Neural Network* (NN) is a series of elementary building blocks referred to as *perceptrons* organized in a net structure. For this thesis, only the simplest structure of NN's organized in layers is considered.

A perceptron P on certain layer inputs k_P values from perceptrons on the preceding layer (or NN inputs) and outputs a single value. A *weight* $w_i^{(P)}$ is assigned to each (i-th) input of the perceptron P. These weights, together with the structure, define the Neural Network. The perceptron P evaluates its k_P input values $(v_i)_{i=1}^{k_P}$ as follows:

$$\mathsf{eval}_P(v_i)_{i=1}^{k_P} = f\Big(\sum_{i=1}^{k_P} w_i^{(P)} v_i\Big), \tag{1.11}$$

where f is referred to as the *activation function.*

In WTFHE, the activation function f represents *Bootstrapping* with signum function encrypted in a TRLWE Sample.

Chapter 2

Analysis and Design

In this chapter I will describe chosen sets of WTFHE parameters. I will also analyze possible approaches to hardware acceleration of the WTFHE scheme – mainly the hardware/software split in regards to volume of transmitted data.

2.1 Parameters of WTFHE

To evaluate the feasibility of WTFHE acceleration on an FPGA, a few sets of parameters devised by Ing. Jakub Klemsa were provided to me (see Table 2.1). The "Infant" set was used for verification only, the "Baby" set was created afterwards for a proof-of-concept demo (see section 4) evaluating a set of neural networks using the FPGA as an accelerator. "PI=4 Test" set was created at the end of development to test scalability of the implemented design. Finally, the set "Full" represents values that could be used in the real-world deployment of WTFHE with sufficient cryptographic security.

Note that these values are subject to the current understanding of the WTFHE scheme and are expected to change with breakthroughs in research.

2.2 Target Platform

Due to the expected amount of data processed, we decided to use the biggest FPGA accessible to us – the Zynq-7000 SOC (part identification number is xc7z020clg484-3, [13]). Another benefit of using Zynq SOC is in-build ARM Core and 1GB of DDR3 memory.

Both of these features could be used in the WTFHE scheme – the ARM core for procedures that are not worth to accelerate on hardware and the DDR3 memory to store TRGSW Samples, which are expected to be quite large. The FPGA part also boasts 220 DPS48 modules beneficial to hardware acceleration as multiplication is a base operation within WTFHE.

Parameters					
Name	"Infant"	"Baby"	"PI=4 Test"	"Full"	Description
K	1	1	1	1	Dimension of TRLWE/TRGSW
L	4	5	17	2	Depth of decomposition of TRLWE/TRGSW
$GAMMA$	3	3	1	9	Width of decompositon of TRLWE/TRGSW
TAU	20	22	24	28	Torus floating point precision
PI	2	3	4	4	Plain text bit size
DIM	4	4	4	860	Cipher key bit size
Derived parameters					
$Bg = 2^{GAMMA}$	8	8	2	512	Basis of decomposition in TRLWE/TRGSW
$IOTA = GAMMA \cdot L$	12	15	17	18	Rounded Torus floating point precision
$N \geq \sqrt{(6 \cdot (n+1))} \cdot 2^{PI-1}$	16	32	64	1024	Order of polynomials
$NU = log2(N)$	4	5	6	10	Number of bits of N
Calculated values					
$K + 1$	2	2	2	2	Number of columns of TRGSW
$(K+1) \cdot L$	8	10	34	4	Number of rows of TRGSW
$NU \cdot TAU$	80	110	144	280	TLWE size in bits
$(K+1) \cdot N \cdot TAU$	640	1 408	3 072	~57 Kb	TRLWE size in bits
$\frac{3}{2} \cdot DIM \cdot 2 \cdot (K+1) \cdot L \cdot N \cdot TAU$	~30 Kb	~83 Kb	~612 Kb	~282 Mb	TRGSW size in bits

Table 2.1: List of WTFHE Parameters

2.3 HW/SW Split

One of the main considerations of implementing WTFHE on the FPGA is hardware/software split – i.e. for which parts of the scheme it is more beneficial to be implemented on the FPGA (parallelizable operations) or in software on ARM/external PC (sequential operations or parts, which do not have a static form).

Figure 2.1 visualizes the hierarchy of operations based on their complexity and continuity (i.e. the superior operation contains the whole inferior operation and adds additional computation). Operations listed in Figure 2.1 create complete functional units and signify major steps in Bootstrapping evaluation.

There are two factors when deciding where to split the design. The main one is the bandwidth requirements on the interconnect between FPGA and ARM/CPU – AXI bus and UART/USB connection – as those are expected to be in the order of magnitude slower than cache–core communication and slice LUT–register–BlockRAM connection. The second factor is parallelizability as any sequential load is expected to run faster in ARM/CPU due to the low frequency of FPGA core.

The Table 2.2 shows the volume of data required to send over AXI/USB connection in a hypothetical situation, that the listed operation would be accelerated on FPGA, while everything hierarchically above (in the Figure 3.9) would run in software. The first column lists the combined volume of inputs and outputs required to run one evaluation of the listed operation. The second column shows how many times the operation would be run in one Bootstrapping operation, and the last column is the total amount of data sent. All figures are calculated for the "Baby" set of parameters.

Note "Bootstrapping without TRGSW transmission" means that TRGSW Samples would be saved in a local memory easily accessible from FPGA, thus the only transmission needed is TLWE and TRLWE Samples.

Operation	Volume of function data [bits]	Number of function calls	Total volume for "Baby" set [bits]
Polynomial Mult.	1 504	40	60 160
Vector-Matrix Mult.	16 448	2	32 896
External Product	16 896	2	33 792
Bootstrapping	86 108	1	86 108
Bootstrapping without TRGSW transmission	1 430	1	1 430

Table 2.2: Volumes of Transmitted Data

Based on the volume of transmitted data, accelerating External Product,

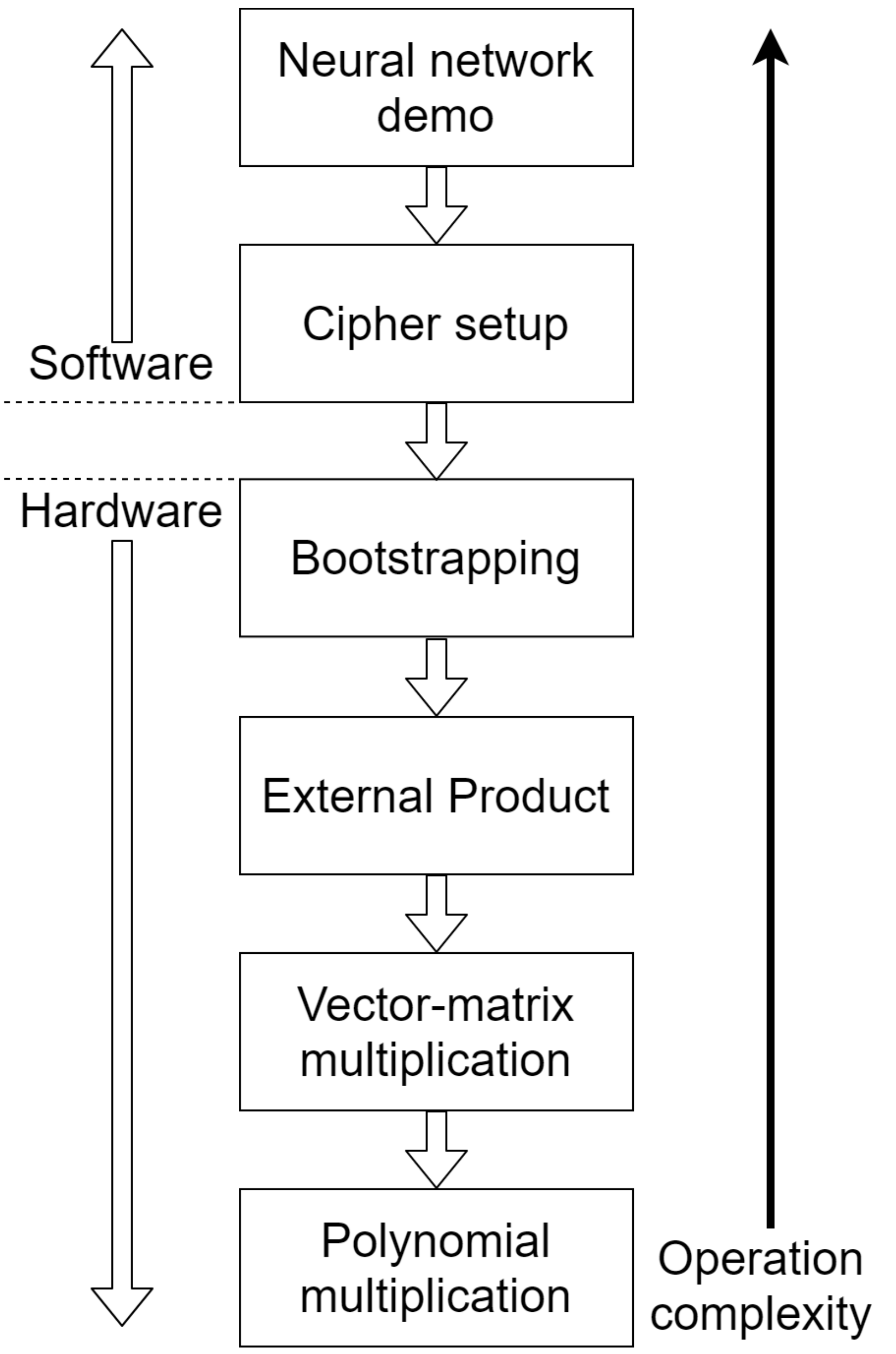

Figure 2.1: Hierarchy of Operations

or Bootstrapping (without TRGSW transmission) seems like the best option. Do note that these assumptions do not take into consideration the actual computational time of each operation as I have no suitable method to predict the duration of each operation.

For reference, the “hardware” and “software” denominated arrows in Figure 2.1 show the split used in final implemented version.

2.3.1 Detailed Breakdown of HW/SW Splits

The following list discusses the pros and cons of the hardware/software split at different levels. The first term in the heading signifies everything below with the listed entity included would be implemented in hardware, while the second term (and everything above) would be done in software.

Polynomial Multiplication / Vector-Matrix Multiplication Polynomial Multiplication (of integer and Torus polynomials, see Algorithm 5 for more details) is an elementary operation within WTFHE scheme and as such would be called very frequently. This highlights the advantage of FPGA as several multiplications of coefficients can be done in parallel with the usage of DPS48 blocks.

As in each Vector-Matrix Multiplication, there are $(K + 1) \cdot (K + 1) \cdot L$ Polynomial Multiplications, which also can be done in parallel, there are no benefits to split Vector-Matrix and Polynomial multiplication onto different platforms as the latency of data transmission is expected to overshadow the increase in performance gained by accelerating Polynomial multiplication only.

Vector-Matrix Multiplication / External Product Vector-Matrix Multiplication is a major subset of External Product. External Product only adds Gadget Decomposition to convert TRLWE Sample into a vector of integer polynomials (see Algorithm 1). Gadget decomposition consists of reordering data arrays and rounding (trivial both in hardware and software) and so both Vector-Matrix Multiplication and External Product should be done on the same platform.

External Product / Bootstrapping During each Bootstrapping operation, a total of $\frac{DIM}{2}$ evaluations of External Product is done. For each of those evaluations, TRGSW and TRLWE Samples are needed as inputs.

When split in such a way, that External Product is accelerated in hardware and the rest of Bootstrapping is done in software, each call of External Product requires to send a new TRGSW Sample as it cannot be precalculated due to dependency on values of the current TRWE Sample (see Algorithm 2).

If both External Product and Bootstrapping are accelerated in hardware, $\frac{3 \cdot DIM}{2}$ TRGSW Samples are needed to be sent over to FPGA and stored locally

– they cannot be generated locally on FPGA side since they encrypt the Bootstrapping key, which is the identical to the private key due to KeySwitch procedure being omitted in this thesis. For one run of Bootstrapping this would be slower than sending one TRGSW Sample for each External Product. But the TRGSW Samples are static during the whole session of one user (due to KeySwitch procedure being omitted), so as long as all $\frac{3 \cdot DIM}{2}$TRGSW Samples can fit into the FPGA-accessible memory and the evaluation of the expression on line 4 in Algorithm 2 (multiplication and summation of TRGSW Samples) can be done with reasonable usage of FPGA resources, it is beneficial to run Bootstrapping in hardware.

The huge decrease in transmitted data of subsequent Bootstrapping runs while sharing TRGSW Samples can be observed in the last two rows of Table 2.2.

Bootstrapping / Cipher Setup and Neural Network Evaluation Any hierarchically superior operations should not be evaluated on FPGA as these operations are private to the end-user. This consists of the encryption of plaintext and evaluated function over said plaintext, and definitions of the Neural Networks.

Chapter 3

FPGA Accelerator

In this chapter, I will describe steps taken during the implementation of the WTFHE scheme. The whole process was iterative, starting from elementary blocks with verification after each major version and any change due to updates in specification and improvements to underlying cipher algorithms. In this chapter, I focus solely on implementation of the hardware part, i.e., Bootstrapping accelerator on an FPGA.

I have chosen VHDL language over Verilog simply by a matter of preference. I decided to start implementation in ISE 14.7 with Nexys3 board due to previous on-hand experience with those development tools. Nexys3 was later exchanged for ZedBoard FPGA when it was apparent that the design will have considerable resource requirements and also the onboard ARM chip could be employed with encryption and process control. In addition with the FPGA swap, I had to port the project to Vivado 2020.1 since ZedBoard is not supported in ISE.

3.1 Parameters and Datatypes

Due to the fact, that VHDL does not have any complex data types I had to define my own variables to represent used structures. Values of parameters can be seen in Table 3.1 and are current as of the latest deployed version (versions are elaborated upon in the following section).

Fundamental data types	
Torus decimal with a fixed-point presicion	TAU bit wide variable of type UNSIGNED
Integer	$GAMMA$ bit wide variable of type UNSIGNED

Parameters			
Name	"Infant" values	"Baby" values	Description
NU	4	5	Number of bits of N
N	16	32	Order of polynomials
$GAMMA$	3	3	Width of decompositon of TRLWE/TRGSW
K	1	1	Dimension of TRLWE/TRGSW, $K+1$ is the number of columns of Bootstrapping matrices
L	4	5	Depth of decomposition of TRLWE/TRGSW, also number of rows of Bootstrapping matrices
DIM	4	4	Cipher key bit size, n in the TFHE paper
TAU	20	22	Torus floating point precision
$IOTA$	12	15	Rounded Torus floating point precision

Variables		
Name	Implementation	Description
TORUS_POLY	array[N] of Torus coefficients	Torus polynomial
INT_POLY	array[N] of integer coefficients	Integer polynomial
INT_POLY_SPLIT	array[L] of INT_POLY	Vector of integer polynomials
TORUS_POLYS	array[$(K+1)$] of TORUS_POLY	TRLWE Sample - two Torus polynomials
TORUS_VECTOR	array[$((K+1)\cdot L)$] of TORUS_POLY	Vector of Torus polynomials
TORUS_MATRIX	$K+1$ long array of TORUS_VECTOR	TRGSW Sample - Bootstrapping matrix
TLWE	array[N] of TAU bit UNSIGNED	TLWE Sample
TLWE_R	array[N] of $NU+1$ bit UNSIGNED	Rounded down version of TLWE

Table 3.1: Set Values of Parameters

3.2 Version Timeline

For ease of orientation, I decided to codename each step with a version number. Below follows a brief summary of what was added/changed in each version.

v0.8 : Polynomial Multiplication

Experiments with polynomial multiplication and DSPs on Nexys3. RS232 was chosen and tested as the communication interface.

v0.9 : Vector-Matrix Multiplication

The project was ported to Vivado and ZedBoard FPGA. Vector-Matrix multiplication was successfully synthesized and tested, although with wrong parameters due to a misunderstanding of the underlying algorithm.

v1.0 : Register based Bootstrapping

First major milestone. The Bootstrapping operation was successfully simulated against reference Ruby code, but could not be synthesized due to the high LUT/FF requirements of the design. Values of Parameters correspond to the ”Infant” version of Bootstrapping.

v1.1 : BRAM based Bootstrapping

Previous space violations were fixed by saving input matrices into BlockRAM and optimizing matrix storage in registers.

v1.2 : Added DSPs to Bootstrapping

Forced DSP48 instantiation which resulted in lowering resource requirements as DSP Blocks implement multiplication internally. Fixed previously incorrect value of TAU from 18 to 20.

v1.3 : Optimized usage of DSPs in Bootstrapping

Started using internal accumulator inside DSP48 blocks (now employing Multiply-and-Accumulate instruction). As external accumulators are omitted, resource requirements are further reduced.

v1.4 : Design scaled to new set of “Baby” parameters

Bug-fixing and rewriting code to be scaleable with different parameters.

v1.5 : Neural Network Demo

Fixed BOOTSTRAPPING entity output. Implemented C code counterpart which used the FPGA as an accelerator for Bootstrapping function. First Neural Network demo.

v1.6 : Communication data flow improvements

Improvements to the communication protocol, which hugely lessened the amount of data sent over RS232 interface.

v1.7 : Pipelining

Pipelining of some datapaths to meet timing requirements for 100 MHz synthesis.

3.3 Polynomial Multiplication

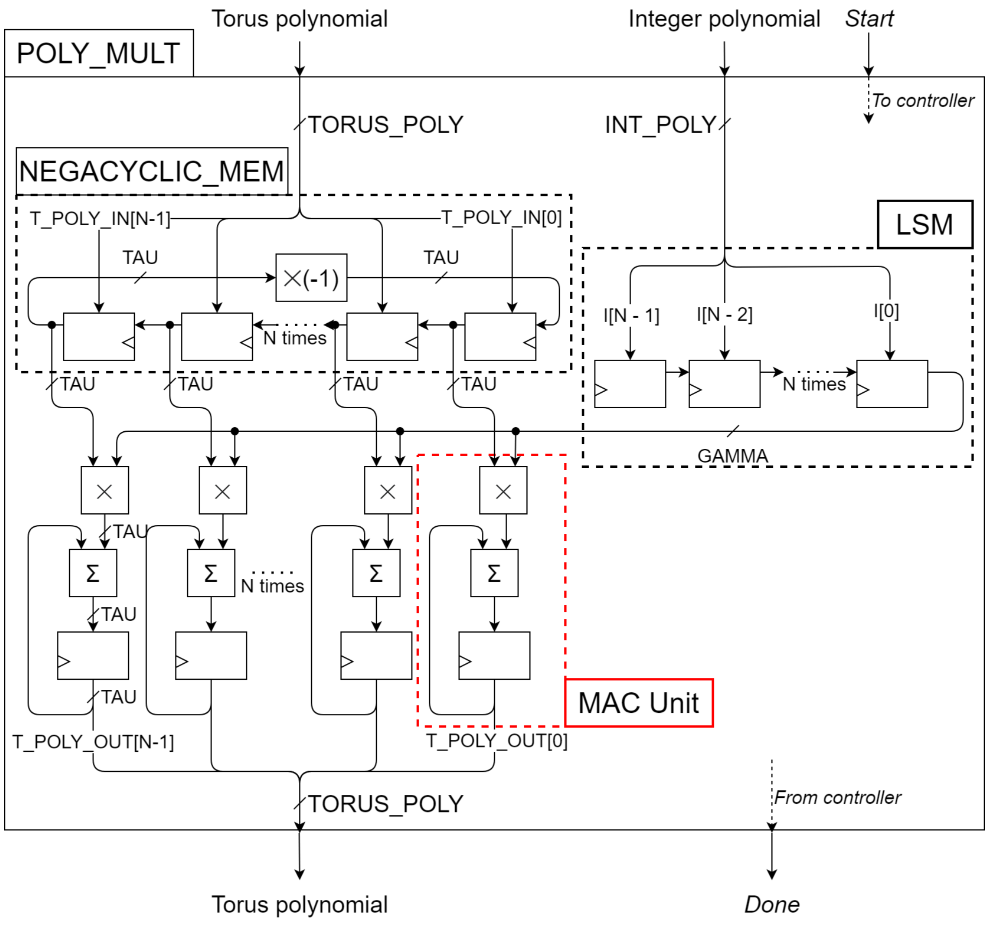

Figure 3.1: Polynomial multiplication

Polynomial multiplication (Algorithm 5, entity name POLY_MULT) is the baseline operation in Bootstrapping. Its inputs are two polynomials – the first polynomial has coefficients from the Torus ($\mathbb{T}$) and the second polynomial from integers ($\mathbb{Z}$). Both polynomials are of the order of $N = 32$. Torus coefficients of the first polynomial are $TAU = 22$ bits wide and the integer coefficients of the second polynomial are $GAMMA = 3$ bits wide. The output is a Torus polynomial of the same order (32) and again TAU bit wide coefficients.

The implemented design is based on the least-significant-bit binary multiplier. Figure 3.1 illustrates datapath of the implementation. As opposed to the binary multiplier, this design has a few, but significant modifications. Primarily, every operation is done over arrays of bits as opposed to single

Algorithm 5 POLY_MULT.vhd

```
Input: Torus polynomial $T = (T^{(N-1)}, \dots, T^{(0)}) \in \mathbb{T}^{(N)}[X]$,
Input: Integer polynomial $I = (I^{(N-1)}, \dots, I^{(0)}) \in \mathbb{Z}^{(N)}[X]$,
Input: Control signals RST and START,
Output: Torus polynomial $Y = (Y^{(N-1)}, \dots, Y^{(0)}) \in \mathbb{T}^{(N)}[X]$.
1: for $i \in [0, N-1]$ do
2:     for $j \in [0, N-1]$ do
3:         $Y^{(j)} \leftarrow Y^{(j)} + (T^{(j)} \cdot I^{(i)})$
4:     $T \leftarrow$ NegacyclicRotate($T$)
5: return $Y$
```

bits. Secondly, the bit-shifter on primary input is replaced with a negacyclic register field. Outlined in the dashed red line is the Multiply-and-Accumulate unit; while in case of binary multiplier a single AND gate is sufficient, in our case we need $TAU \times GAMMA$ multiplier. MAC unit was redesigned several times during development and I will describe it more accurately in the following subsection.

When START signal is sent, inputs are loaded into their respective register fields, the accumulator is reset and the internal counter is set to $N-1=31$. In each step, Torus polynomial inside NEGACYCLIC_MEM is multiplied by a constant coefficient of the integer polynomial (output of LSM) and the result is added to the partial result (stored in accumulator). At the end of each step, Torus polynomial is negacyclicaly rotated (Algorithm 6) and integer polynomial shifted to the right (Algorithm 7). Carry bit during any operation are discarded as the output is from the Torus field – the result is reduced modulo 1.

In the latest implemented version, negacyclic rotation, multiplication, and addition is calculated in one cycle (lines 3 and 4 in Algorithm 5), thus the delay between START and DONE signals is $N+2=34$ clock cycles – N cycles for the multiplication, and 2 cycles of setup delay caused by the controller.

When synthesizing this design with a clock frequency exceeding beyond 100 MHz, negacyclic rotation, multiplication, and addition should be split to separate pipeline stages to meet timing requirements.

This design is usable in this proof-of-concept demo while using relatively small parameters of the “Baby” variant, but would quickly use up all of FPGA resources when N is scaled beyond 64 (ZedBoard has 220 DSPs and the whole accelerator is using $2 \cdot N$ of them – assuming one multiplication still uses only one DSP).

Algorithm 6 NEGACYCLIC_MEM.vhd

Input: Torus polynomial $T = \big(T^{(N-1)}, \dots, T^{(0)}\big) \in \mathbb{T}^{(N)}[X]$,
Input: Control signals RST, LOAD and PULSE, optional DIR,
Output: Torus polynomial $Y = \big(Y^{(N-1)}, \dots, Y^{(0)}\big) \in \mathbb{T}^{(N)}[X]$.

```
1: if LOAD = 1 then
2:     $Y \leftarrow T$
3: else
4:     if PULSE = 1 then
5:         $\big(Y^{(N-1)}, \dots, Y^{(0)}\big) \leftarrow \big(Y^{(N-2)}, \dots, Y^{(0)}, -Y^{(N-1)}\big)$
6: return $Y$
```

3.3.1 Negacyclic Memory

Negacyclic Memory (Algorithm 6, entity NEGACYCLIC_MEM) holds Torus polynomial, i.e. the first operand of polynomial multiplication. It is a series of cyclically linked registers; each register holds one coefficient. On the clock rising edge, the polynomial is cyclically shifted to the left. The uppermost coefficient (x^{N-1}) which overflows to the position of constant coefficient (x^0) is multiplied by -1 (`not(x)+1` in VHDL code) – and vice versa when rotating in the reverse direction.

This operation is equal to multiplying the whole polynomial by x^1 in negacyclic set $\mathbb{T}^{(N)}[X]$. The implemented entity has an optional direction of the shift (thus allowing to multiply by x^{-1}), which was used during early versions of Bootstrapping, but was deemed unnecessary later on.

3.3.2 Linear Shift Memory

Linear Shift Memory (LSM – Algorithm 7) holds the second operand – integer polynomial. When PULSE signal is set to high, the polynomial is logically shifted to the right (the highest coefficients are padded with zeros).

Algorithm 7 LSM.vhd

Input: Integer polynomial $I = \big(I^{(N-1)}, \dots, I^{(0)}\big) \in \mathbb{Z}^{(N)}[X]$,
Input: Control signals RST, LOAD and PULSE,
Output: Integer coefficient $Y \in \mathbb{Z}$.

```
1: if LOAD = 1 then
2:     $Y \leftarrow I$
3: else
4:     if PULSE = 1 then
5:         $\big(Y^{(N-1)}, \dots, Y^{(0)}\big) \leftarrow \big(0, Y^{(N-2)}, \dots, Y^{(1)}\big)$
6: return $Y^{(0)}$
```

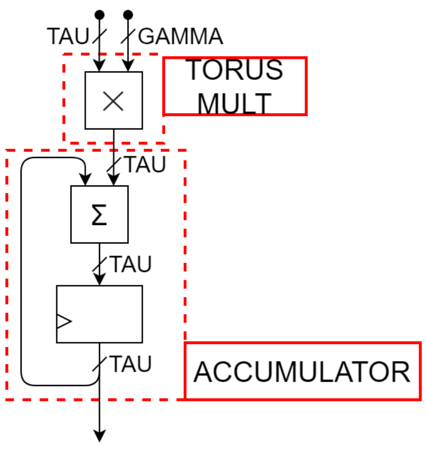

Figure 3.2: Initial implementation

3.3.3 MAC Unit

MAC Unit inside POLY_MULT entity calculates and stores the result of line 3 in Algorithm 5. Each coefficient Y^j is evaluated separately using the parallelism of the FPGA.

3.3.3.1 Version Evolution

The design of the MAC unit saw a few changes over its lifetime. Those changes were facilitated by a need to reduce the usage of FPGA logic cells and did not change the functionality of the design.

Versions 0.8 – 1.1

In these versions (see Figure 3.2) I implemented the multiplication with a multiplication sign ($*$) inside TORUS_MULT entity. For $GAMMA = 12$ and $TAU = 18$ (preliminary values), the synthesis tool inferred DSP block, but when I fixed $GAMMA$ to correct value of 3, the synthesis tool inferred LUTs instead. ACCUMULATOR entity was used to store the output (see Algorithm 8).

Version 1.2

In this version (see Figure 3.3) I replaced multiplication sign ($*$) inside TORUS_MULT with forced instantiation of DSP48 block. The most significant bit of each input had to be padded with zero bit since DSP uses signed multiplication on its operands. This solution also limits parameters to $TAU \leq 24$ and $GAMMA \leq 17$, due to DSP input size.

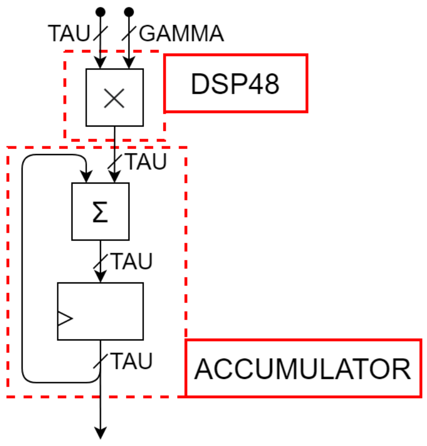

Figure 3.3: Optimization with DSP48 block

Version 1.3 onwards

Since version 1.3 (see Figure 3.4) I replaced both TORUS_MULT and ACCUMULATOR entities with an instance of DSP48 IP Core running multiply-and-accumulate instruction. This is functionally equal to version 1.2 but saves additional resources as the accumulator is already present inside the DSP48 unit.

3.4 Vector-Matrix Multiplication

Vector-matrix multiplication (Algorithm 9) is the next major step towards Bootstrapping. Its inputs are a matrix of Torus polynomials (TORUS_MATRIX – dimensions of $(K+1)\cdot L \times (K+1) = 20$) and vector of integer polynomials (INT_VECTOR – dimensions of $(K+1)\cdot L = 10$). The vector-matrix multiplicator outputs $K+1$ Torus polynomials (TORUS_POLYS).

Current design is based on standard vector–matrix multiplication imple-

Algorithm 8 ACCUMULATOR.vhd

```
Input: Torus polynomial $A = (A^{(N-1)}, \dots, A^{(0)}) \in \mathbb{T}^{(N)}[X]$,
Input: Control signals RST and PULSE,
Output: Torus polynomial $Y = (Y^{(N-1)}, \dots, Y^{(0)}) \in \mathbb{T}^{(N)}[X]$.
1: if PULSE $= 1$ then
2:     $Y \leftarrow Y + A$
3: return $Y$
```

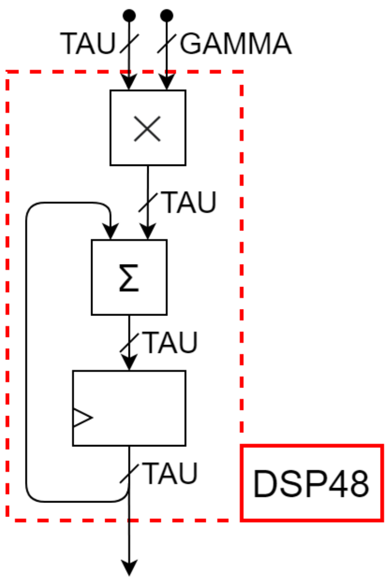

Figure 3.4: Final usage of DSP48 block

Algorithm 9 Vector-Matrix multiplication

Input: Vector of integer polynomials $\mathbf{I} = (I_{(K+1)\cdot L-1}, \dots, I_0) \in \mathbb{Z}^{(N)}[X]^{(K+1)L}$,
Input: Matrix of Torus polynomials $\mathbf{T} = (T_{((K+1)\cdot L-1,K)}, \dots, T_{(0,0)}) \in \mathbb{T}^{(N)}[X]^{((K+1)\cdot L),(K+1)}$,
Input: Control signals RST and START,
Output: Vector of Torus polynomials $\mathbf{Y} = (Y_K, \dots, Y_0) \in \mathbb{T}^{(N)}[X]^{(K+1)}$.

```
1: for $i \in [0, K]$ do
2:     for $j \in [0, (K+1) \cdot L - 1]$ do
3:         $T\_OUT_i \leftarrow I_j \cdot T_{(i,j)}$
4:     $Y_i \leftarrow Y_i + T\_OUT_i$
5: return $\mathbf{Y}$
```

menting the equation 3.1 (also lines 3 and 4 in Algorithm 9):

$$\forall j \in \{0, K\} : Y_j = \sum_{k=0}^{(K+1)\cdot L-1} I_k \cdot T_{k,j} \tag{3.1}$$

Since parameter K was permanently set to 1, thus the matrix has always two columns and the output is always two Torus polynomials which are calculated independently, I decided to use innate parallelism of the FPGA to split the design into two halves, each processing one column of the matrix and outputting one Torus polynomial (see Figure 3.5). Both halves have their own

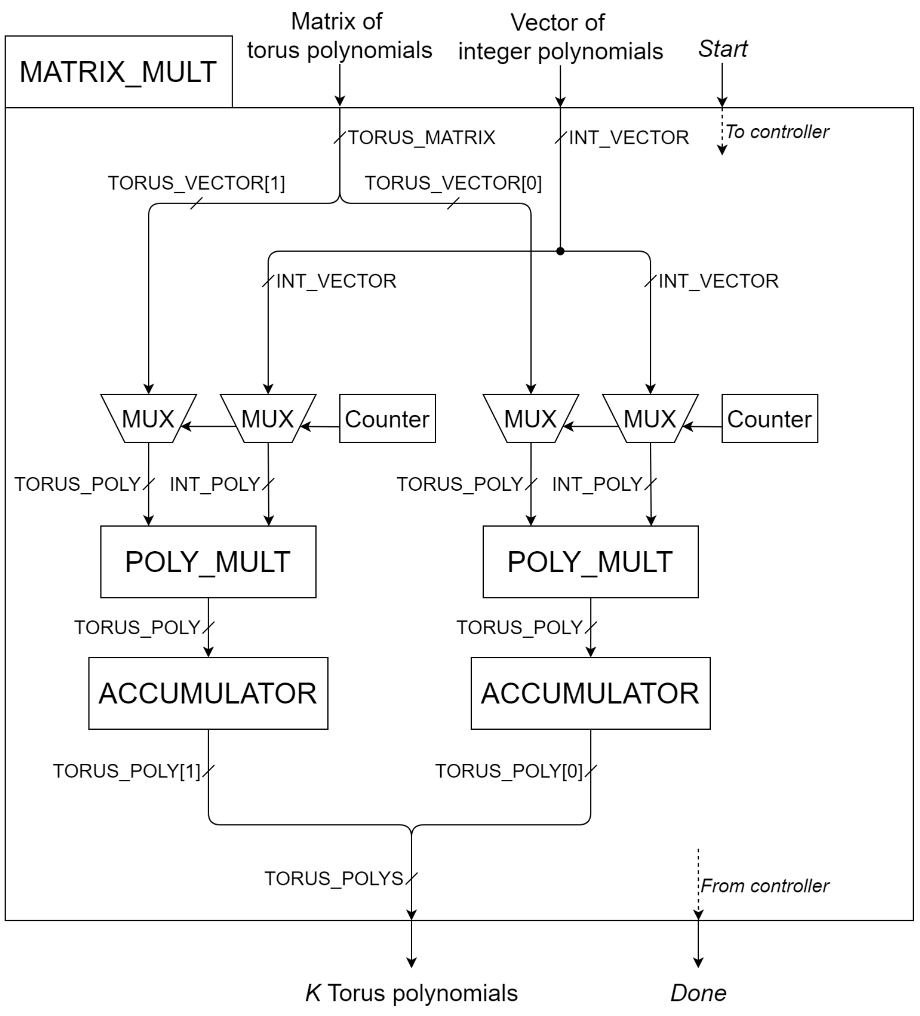

Figure 3.5: Vector–matrix multiplication

independent counter to multiplex the inputs into single polynomials, which are fed into an instance of POLY_MULT entity (described in subsection 3.3]) and this result is accumulated (line 4 in Algorithm 9) in a register accumulator (I reused the same entity from Algorithm 8).

As a precaution, I added barrier synchronization consisting of simple synchronous flag registers waiting for both sides to complete. This was found to be superfluous later on as polynomial multiplication always takes the same amount of clock cycles independent of input data. For the “Infant” set of parameters the minimal processing time of vector-matrix multiplication is $(K+1)\cdot L\cdot 34 = 340$ clock cycles. Due to setup delays in controller, the overall latency between START and DONE control signals is 352 clock cycles.

Vector-Matrix multiplication has not seen any significant implementation changes throughout its development.

3.5 External Product

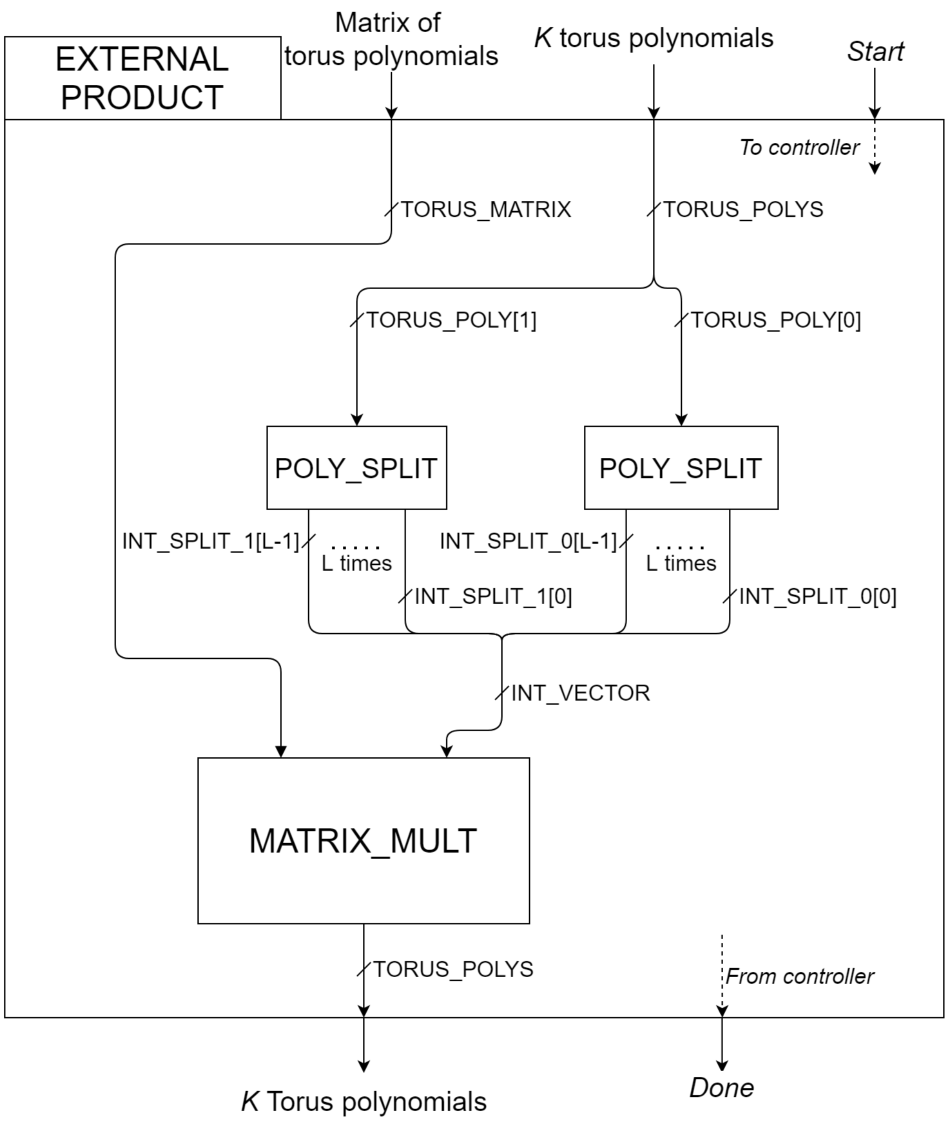

Figure 3.6: External product

External product (entity EXT_PROD) is an extension to the vector–matrix multiplication. It serves as an interface between MATRIX_MULT (see Algorithm 9, see also Figure 3.5) and variables calculated in Bootstrapping (see Algorithm 11) where it is denoted by a $\boxdot$ symbol.

External Product has two inputs: TRGSW Sample (a matrix) and TRLWE Sample (two polynomials) Sample. Both are simply hooked up to the MATRIX_MULT entity. The second input, TRLWE Sample is split inside POLY_SPLIT (Algorithm 10) into a vector of smaller integer polynomials. Implemented datapath can be seen in Figure 3.6.

3.5.1 Polynomial Split

Polynomial splitter (POLY_SPLIT) splits input Torus polynomial into a vector of L integer polynomials. Both input and output polynomials have an order of N. Torus coefficients are $TAU = 22$ bit wide, while integer coefficients are being only $GAMMA = 3$ bits wide.

Algorithm 10 Polynomial split

Input: Torus polynomial $T = (T^{(N-1)}, \dots, T^{(0)}) \in \mathbb{T}^{(N)}[X]$,
Output: Vector of integer polynomials $\mathbf{I} = (Y_{L-1}, \dots, Y_0) \in \mathbb{Z}^{(N)}[X]^L$.

```
1: $\bar{T}^{(i)} \leftarrow \lfloor 2^{IOTA} \cdot T^{(i)} \rceil$ for $i \in [0, N-1]$
2: for $i \in [0, N-1]$ do
3:     for $j \in [0, L-1]$ do
4:         $I^{(i)}_{L-1-j} \leftarrow \bar{T}^{(i)}[(GAMMA+1) \cdot j - 1$ downto $GAMMA \cdot j]$
5: return I
```

Firstly, every coefficient of the Torus polynomial is converted to a $IOTA$ wide integer. Conversion is done by multiplying the coefficient by 2^{IOTA} and rounding the decimals to the nearest integer (line 1 in Algorithm 10). In VHDL, I simply extracted $IOTA$ most significant bits and the result summed with $(IOTA+1)^{th}$ bit.

Next, each $IOTA$ bit integer was split into L parts, each having $GAMMA$ consecutive bits. The first part held $GAMMA$ most significant bits, the second part held next $GAMMA$ bits etc.

Those parts were then rearranged into the output vector. J^{th} part of i^{th} coefficient of the integer polynomial was saved into i^{th} coefficient of $(L-1-j)^{th}$ polynomial in the vector (line 4 in Algorithm 10).

So, the 0^{th} polynomial in the vector now holds $GAMMA$ most significant bits of the integer polynomial (which are original Torus coefficients modulo $\frac{1}{8}$), the 1^{st} polynomial holds next $GAMMA$ most significant bits (so modulo $\frac{1}{8^2}$ on the remainder of the previous operation) and so on.

Since each entity POLY_SPLIT takes only one polynomial, I used two parallel instances of them for both polynomials in EXT_PROD and merged their outputs to a single INT_VECTOR datatype.

3.6 Bootstrapping

The aim of this work was to implement operation Bootstrapping on an FPGA. I denoted the first implementation, which was successfully verified in simulation, as version 1.0. The generalized block schema of the implementation can be seen in Figure 3.7. Version 1.0 could not be synthesized into real hardware due to high resource usage and had to be improved with subsequent versions.

I will explain the implementation of each step of the Bootstrapping Algorithm in the following subsections. For simplification and clarification I have rewritten the Algorithm 4 into the version depicted as Algorithm 11.

Algorithm 11 Bootstrapping

```
Input: TLWE Sample $(a_0, \ldots, a_{DIM-1}, b) \in \mathbb{T}^{(DIM+1)}$,
Input: TRLWE Sample $P\_IN \in \mathbb{T}^{(N)}[X]^{(K+1)}$,
Input: $DIM \cdot 3/2$ TRGSW Samples $BK \in \mathbb{T}^{(N)}[X]^{((K+1)\cdot L),(K+1)}$,
Input: Control signals RST, START and BRAM Interface,
Output: TLWE Sample $(a_0, \ldots, a_{DIM-1}, b) \in \mathbb{T}^{(DIM+1)}$.
1: $\bar{a}_i \leftarrow \lfloor 2^{NU+1} \cdot a_i \rceil$ for $i \in [0, DIM-1]$, $\bar{b} \leftarrow \lfloor 2^{NU+1} \cdot b \rceil$
2: $ACC \leftarrow X^{-\bar{b}} \cdot P\_IN$
3: for $i \in [0, (\frac{DIM}{2}) - 1]$ do
4:     $MAT \leftarrow (((X^{\bar{a}_{2i-1}+\bar{a}_{2i}} - 1)BK_{3i-2} + (X^{\bar{a}_{2i-1}} - 1)BK_{3i-1} + (X^{\bar{a}_{2i}} - 1)\ BK_{3i})$
5:     $EXT\_PROD \leftarrow MAT \boxdot ACC$
6:     $ACC \leftarrow EXT\_PROD + ACC$
7: return SampleExtract($ACC$)
```

In version, 1.1, I redesigned BOOTSTRAPPING entity to employ Block-RAM IP Core to store input TRGSW and TRLWE Samples. I also renamed the entity to BOOTSTRAPPING_RAM to signify significant changes done to the design. Updated schema can be seen on Figure 3.8.

In the following text, I will describe every sub-unit of the Bootstrapping entity.

3.6.1 TLWE Rounding

Entity TLWE_ROUNDING evaluates the line 1 of Algorithm 11:

$$\bar{a}_i \leftarrow \lfloor 2^{NU+1} \cdot a_i \rceil \text{ for } i \in [0, DIM-1], \bar{b} \leftarrow \lfloor 2^{NU+1} \cdot b \rceil \tag{3.2}$$

For simplification, the datatype of TLWE (input) and TLWE_R (output) both hold all a_i and b variables in a single array. Similarly as the multiplication and rounding in POLY_SPLIT (Algorithm 10), I implemented this operation by extracting $NU + 1$ most significant bits of each operand and summed with $(NU + 2)^{th}$ bit. This rounding gives an upper bound on the value of each operand of $2^{NU+1} - 1$. This is beneficial, as rotating the polynomial by 2^{NU+1} steps would result in an identity.

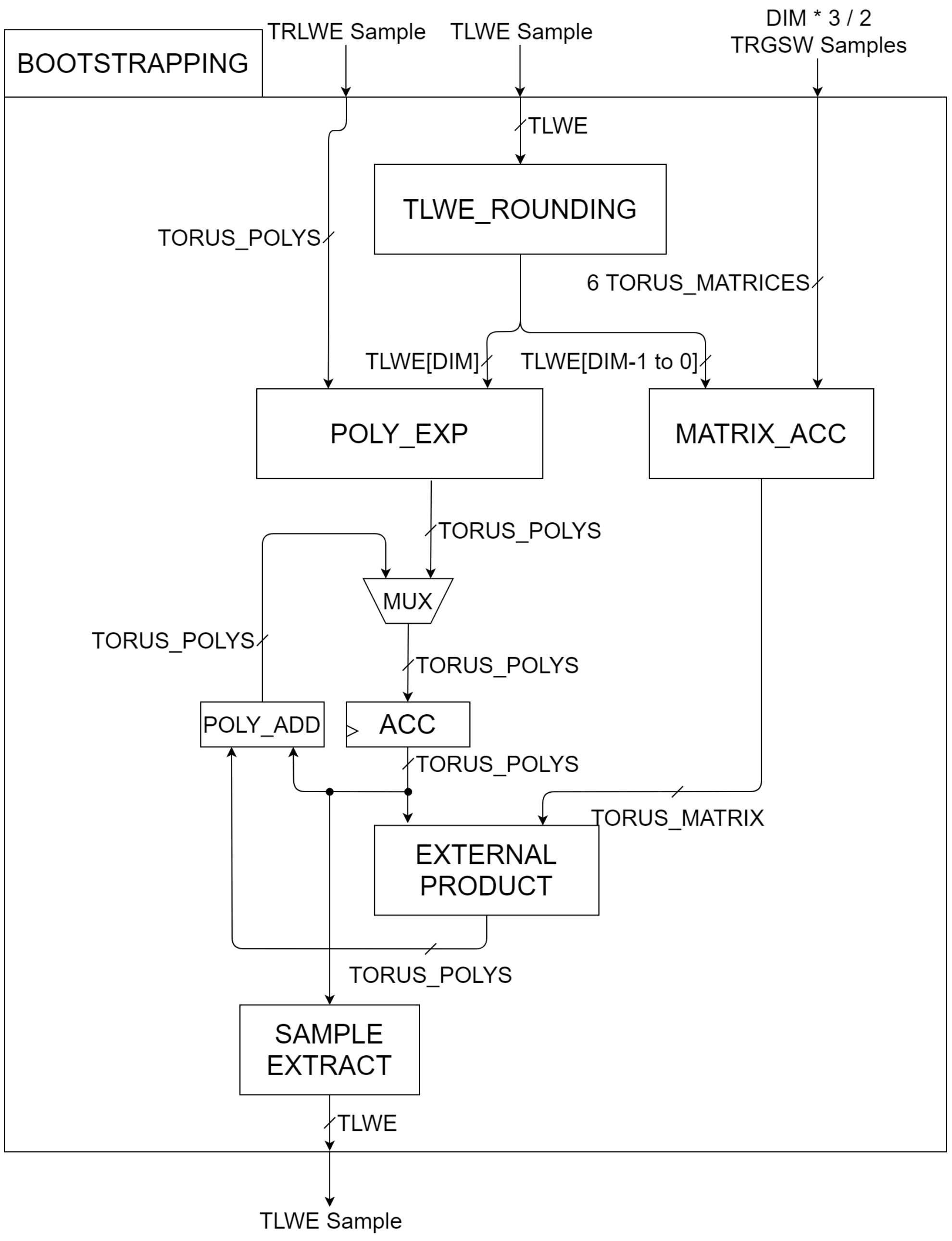

Figure 3.7: Bootstrapping datapath – Version 1.0

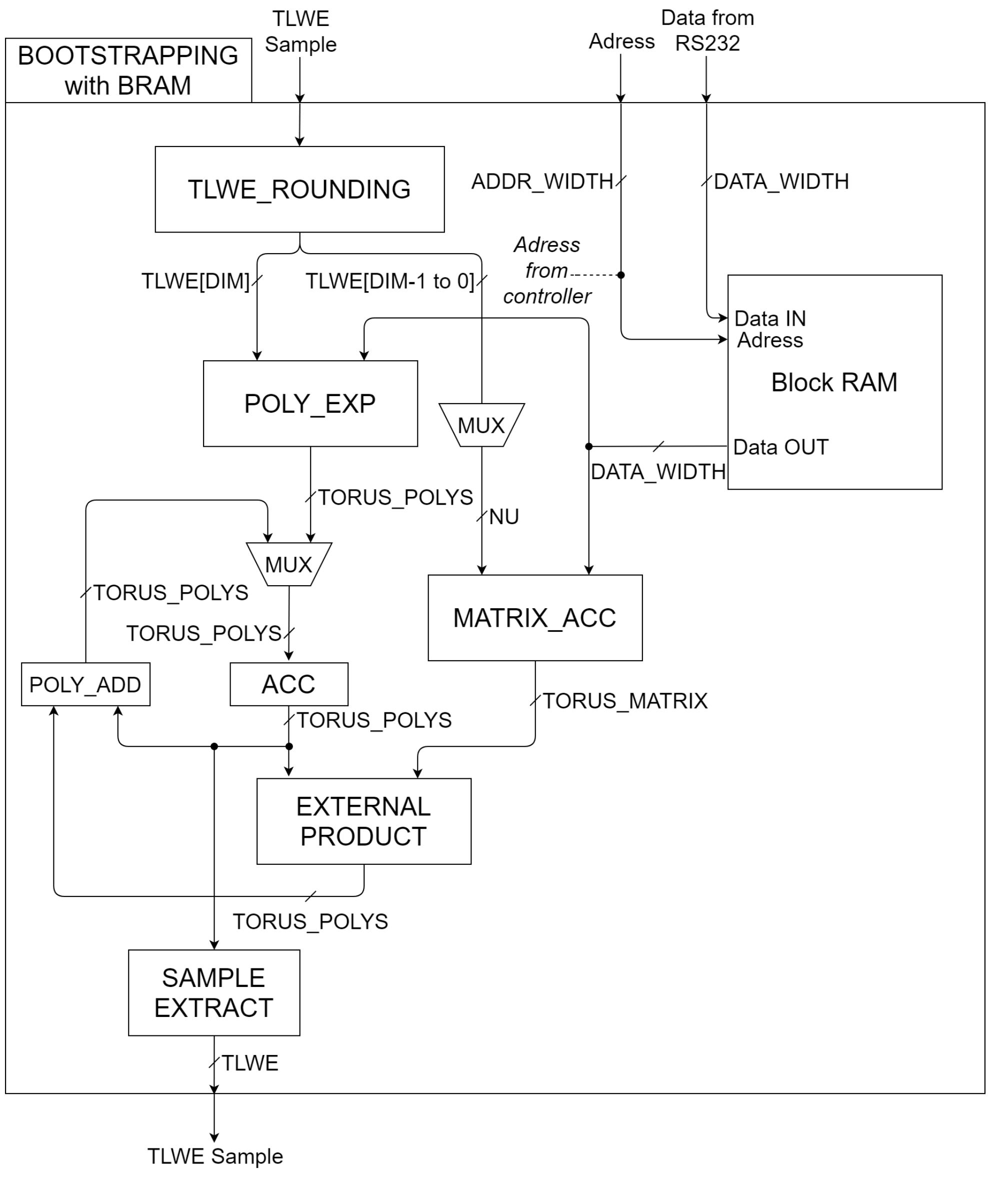

Figure 3.8: Bootstrapping with integrated BlockRAM – Version 1.1 onwards

3.6.2 Polynomial Multiplication by a Monomial (POLY_EXP)

Polynomial multiplication by a monomial (entity name POLY_EXP) serves the purpose of storing TRLWE Sample (two Torus polynomials) multiplied by a monomial with exponent given as a parameter – operation on line 2 in

Algorithm 11:

$$ACC \leftarrow X^{-\bar{b}} \cdot P\_IN \tag{3.3}$$

Since we are working with negacyclic polynomials, multiplying the polynomial by $X^{-\bar{b}}$ is done by negacyclic rotation of the polynomial by $\bar{b}$ positions to the right (when the coefficient of the highest order is on the left).

Version 1.0

In the first version, I reused entity NEGACYCLIC_MEM (see Algorithm 6) to rotate both polynomials at once. The number of steps was controlled by a counter which initialized to a value of b. The computational time was then $b+1$ clock cycles. Since $-\bar{b}$ is negative in this step, I used the optional control signal DIR in NEGACYCLIC_MEM to rotate to the right.

This optional direction control was primarily for simplification as the same result could be calculated by rotating the array by $2 \cdot N - \bar{b}$ positions to the left instead.

Version 1.1 onwards

In version 1.1 I added BlockRAM to store TRLWE Sample P_IN (Figure 3.8). With the BlockRAM I couldn't access the whole polynomial at once and a better approach was to sequentially process all coefficients and save them in a correct "spot". I came up with the following solution:

$$ACC_i^{(k-\bar{b})(\bmod\ N)} \leftarrow DATA \tag{3.4}$$

Where ACC is a register array of TORUS_POLYS data-type, i is the index of a polynomial in TRLWE, k is the degree of coefficient, $\bar{b}$ is the value of the exponent of a monomial and DATA is the current output from BlockRAM.

Coefficients of TRLWE Samples are read from addresses specified by a parameter POLY_START (0xF3F) decreasing down to MATRIX_HIGH+1 (0xF00) – $((K+1) \cdot N) = 10$ values in total (see subsection 3.6.7 for more info on RAM organization).

Each input is passed along with the current value of counter decrementing from $(K+1) \cdot N - 1 = 63$ down to zero.

The calculated exponent ($(k - \bar{b})$ in the equation 3.4) is used to index a location within the output array (ACC) and is calculated as follows:

$$\text{IDX} \equiv (\text{CNTR}(\bmod\ N) - \bar{b})(\bmod\ 2 \cdot N), \tag{3.5}$$

where $CNTR$ is the current value of the counter, and b is the exponent given as a parameter.

$(NU+1)^{th}$ least significant bits of IDX and b are used to detect overflow/underflow of coefficients during negacyclic rotation. When FLAG $= 1$, DATA is multiplied by -1 (`not(DATA)+1` in VHDL) before storing.

$$\text{FLAG} = \text{BitXOR}(IDX[NU], \bar{b}[NU]) \tag{3.6}$$

The final location within ACC array is determined by a bit extraction:

- `D = IDX[NU-1 downto 0]` : Degree of a coefficient
- `P = CNTR[NU]` : Index of the polynomial (one of two so one bit is sufficient)
- `ACC[P][D] = DATA` : Storage of the input

The change from register fields in version 1.0 to sequential reading from BlockRAM in version 1.1 prolonged computational time from $b+1$ to $(K+1)\cdot N = 64$ clock cycles.

3.6.3 Matrix Accumulator

Matrix accumulator (MATRIX_ACC) is an entity which accumulates the result of the equation 3.7 on the line 4 in Algorithm 11:

$$MAT \leftarrow ((X^{a_{2i-1}+a_{2i}} - 1)BK_{3i-2} + (X^{a_{2i-1}} - 1)BK_{3i-1} + (X^{a_{2i}} - 1)BK_{3i}) \tag{3.7}$$

Matrix accumulator was during development very similar to the POLY_EXP described in subsection 3.6.2.

Version 1.0

In the first implementation in version 1.0, MATRIX_ACC was split between several entities as shown in Figure 3.9). I used the same approach to calculate the result of $X^{a_i} \cdot BK_i$ by reusing NEGACYCLIC_MEM in a similar fashion as in POLY_EXP – rotating each polynomial inside the matrix by a_i steps to the left. This operation was put into a separate entity called MAT_EXP ($\times(X^{a_i})$ blocks in Figure 3.9).

Entities MAT_ADD (Σ blocks in Figure 3.9) and MAT_SUB($-$ blocks in Figure 3.9) calculated the addition and subtraction of the whole matrix data structure. I used embedded parallel loops to iterate over every coefficient, with addition and subtraction written in simple $+$ and $-$ signs respectively. Since operands are again from the Torus field, overflow bits were discarded. The output of those two entities was synchronous with the rising edge of the clock. Detailed block schema of the datapath is visualized in Figure 3.9.

This version was quite fast, the total duration of the calculation was only $max(a_{2i-1}+a_{2i}, a_{2i-1}, a_{2i}) + 3$ cycles, but was extremely demanding on slice usage in the FPGA as every matrix was stored in a separate register array.

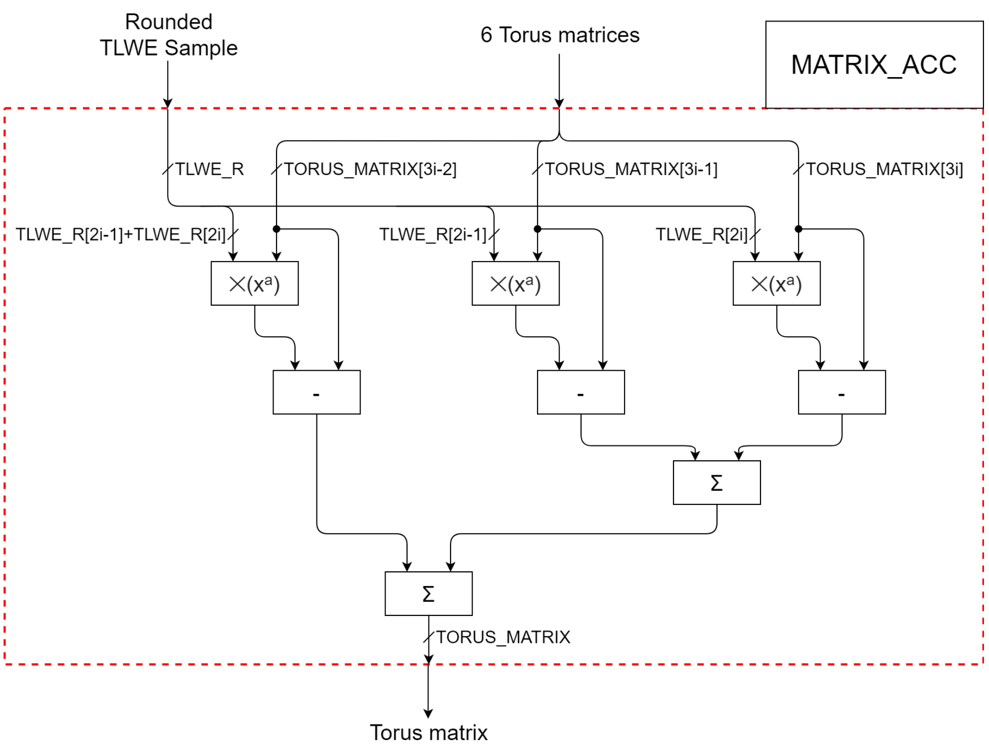

Figure 3.9: Detail of MATRIX_ACC in version 1.0

Version 1.1 onwards

In version 1.1 I have removed previous entities MAT_EXP, MAT_ADD, and MAT_SUB and merged them into one entity named MAT_ACC, which combines functionality of the previous blocks.

MAT_ACC is based on the same principle as POLY_EXP in version 1.1 (see subsection 3.6.2). Since BlockRAM has two clock cycles read delay between inputting an address and reading out the data, I decided to process three matrices sequentially in each pass.

The coefficients of the three matrices used in each loop are saved sequentially on adresses from 0x000 to 0x77F (MATRIX_LOW) (BK_i for $i \in [0, 2]$) and from 0x800 (MATRIX_LOW+1) to 0xEFF (MATRIX_HIGH) (BK_i for $i \in [3, 5]$) for a total of six TRGSW Samples (see subsection 3.6.7 for more info on RAM organization).

The equation on line 4 in Algorithm 11 was split into two following phases. The phases are controlled by EN_EXP (Phase 1) and EN_SUB (Phase 2) control signals.

Phase 1 : Polynomial Multiplication by a Monomial

In the first phase I accumulated the result of the equation 3.8:

$$MAT \leftarrow \big((X^{a_{2i-1}+a_{2i}})BK_{3i-2} + (X^{a_{2i-1}})BK_{3i-1} + (X^{a_{2i}})BK_{3i}\big) \tag{3.8}$$

The equation is somewhat identical to the previous one used in POLY_EXP (see subsection 3.6.2), only in this case the operands are matrices of polynomials (TRGSW) and coefficients are summed instead of overwritten. Due to this reason, this section of implementation is largely identical to version 1.1 implementation of POLY_EXP (see subsection 3.6.2).

The implementation of my design which implements the equation 3.8 can be simplified as follows:

$$MAT_{i,j}^{(k+a)(\text{mod } N)} += DATA, \tag{3.9}$$

where MAT is a register array of TORUS_MATRIX data-type, i and j are indices of rows and columns respectively, k is the degree of coefficient, a the current exponent of a monomial and DATA is the current output from BlockRAM.

Total of $3\cdot(K+1)\cdot(K+1)\cdot L\cdot N-1 = 1919$ inputs are read from BlockRAM representing all coefficients of the three BK matrices in each loop iteration. The location of the last coefficient in BlockRAM is specified by parameters MATRIX_HIGH and MATRIX_LOW (see subsection 3.6.7 for more info on RAM organization).

I reused the procedure from 1.1 version of POLY_EXP (see subsection 3.6.2) to calculate the correct exponent:

$$\text{IDX} \equiv (\text{CNT\_COEFF}(\text{mod } N) + a)(\text{mod } 2\cdot N) \tag{3.10}$$

and to detect underflow/overflow in negacyclic rotation:

$$\text{FLAG} = \text{BitXOR}(IDX[NU], b[NU]) \tag{3.11}$$

Since the value of parameter L is usually not a power of 2, I had to use multiple additional counters to make the design scaleable.

- `CNT_MAIN` : Counting from $3\cdot(K+1)\cdot(K+1)\cdot L\cdot N-1 = 1919$ down to 0; counting number of coefficients read from BlockRAM,
- `CNT_DIM` : Counting from $\frac{DIM}{2} - 1 = 1$ down to 0; number of blind rotate loops,
- `CNT_MAT` : Counting from 2 down to 0; number of matrices processed in each blind rotate loop,

- `CNT_COEFF` : Counting from $(K+1)\cdot(K+1)\cdot L\cdot N-1=639$ down to 0; indexing coefficients of currently processed matrix.

The final location within MAT array is determined by a bit extraction:

- `D = IDX[NU-1 downto 0]` : The degree of a coefficient
- `R = CNT_COEFF[NU+ROW_WIDTH downto NU+1]` : The row index in the matrix, $2^{\text{ROW\_WIDTH}} \geq (K+1)\cdot L$
- `C = CNT_COEFF[NU]` : The column index in the matrix
- `MAT[C][R][D] += DATA` : Storage of the input

Phase 2 : Matrix Subtraction

In the second phase I calculated the remainder of the expression 3.7:

$$MAT \leftarrow MAT - BK_{3i-2} - BK_{3i-1} - BK_{3i} \tag{3.12}$$

The algorithm of my design is as follows:

$$MAT^{k}_{i,j} -= DATA \tag{3.13}$$

This equation is a simplified form of Phase 1 without evaluation of a coefficient's final degree. As such, there is no need to calculate any special index and possibly invert DATA.

Thus I used only bit extraction to subtract the input in a given location within MAT array:

- `D = CNT_COEFF[NU-1 downto 0]` : Degree of a coefficient
- `R = CNT_COEFF[NU+ROW_WIDTH downto NU+1]` : Row index in the matrix, $2^{\text{ROW\_WIDTH}} \geq (K+1)\cdot L(=6)$
- `C = CNT_COEFF[NU]` : Column index in the matrix
- `MAT[C][R][D] -= DATA` : Subtraction of the input

3.6.4 External Product

The TRGSW Sample calculated in the previous step MATRIX_ACC (see subsection 3.6.3) and TRLWE Sample held in the ACC block was passed along to an instance of EXT_PROD entity to calculate line 5 in Algorithm 11. The output is again TRLWE Sample.

$$EXT\_PROD \leftarrow MAT \boxdot ACC \tag{3.14}$$

Note that operation External product is denominated by the symbol $\boxdot$.

3.6.5 Poly ADD

The last operation of one for the cycle is line 6 in Algorithm 11.

$$ACC \leftarrow EXT\_PROD + ACC \tag{3.15}$$

The sum is calculated inside entity POLY_ADD (marked with Σ symbol on Figure 3.7). Both inputs and output are TRLWE Samples.

Parallel for loop iterates over all coefficients. Adders are again synchronous with the rising edge of the clock.

3.6.6 Sample Extract

The last operation to obtain desired result from Bootstrapping is called Sample Extract (line 7 in Algorithm 11). The Algorithm 12 was quite straightforward to implement in VHDL as it consists only of extraction of several coefficients. The whole entity consists of one generate loop extracting needed coefficients and reorganizing them into the TLWE data structure.

Algorithm 12 SampleExtract

```
Input: TRLWE Sample $T = (r, s) \in \mathbb{T}^{(N)}[X]^{(K+1)}$,
Output: TLWE Sample $(a_0, \dots, a_{DIM-1}, b) \in \mathbb{T}^{(DIM+1)}$.
1: return $(\mathbf{a}, b) = ((r^{(0)}, -r^{(DIM-1)}, \dots, -r^{(1)}), s^{(0)})$
```

When I first implemented this Algorithm in version 1.0 I made en error and returned $(N+1)$ total coefficients - $(\mathbf{a}', b') = ((-r^{(N-1)}, \dots, -r^{(1)}), s^{(0)})$. I discovered and fixed this error in version 1.5 which lowered LUT and FF usage by almost 2%.

3.6.7 BlockRAM Organization

BlockRAM was added in version 1.1 to store majority of input data. To simplify the usage of BlockRAM I choose to organize the data in following arrangement:
0x000 to 0x01F: N coefficients of the polynomial in the first column and first row of the first TRGSW Sample – coefficients with higher degree are saved to a higher adress
0x000 to 0x03F: first row of polynomials of the first TRGSW Sample
0x000 to 0x2F7: first TRGSW Sample
0x000 to 0x77F (MATRIX_LOW): lower three TRGSW Samples
0x800 to 0xEFF (MATRIX_HIGH): higher three TRGSW Samples
0xF00 to 0xF1F: N coefficients of the first polynomial of a TRLWE Sample
0xF00 to 0xF3F (POLY_START) : TRLWE Sample

All the data is stored sequentially without any padding, i.e. the last coefficient of the first polynomial on address 0x01F, the constant coefficient of the following polynomial is saved on address 0x020, etc.

The total amount of memory used is:

$$(DIM \cdot \frac{3}{2} \cdot (K+1) \cdot L \cdot (K+1) \cdot N + (K+1) \cdot N) \cdot TAU = 85888 \text{ bits} \quad (3.16)$$

In BlockRAM IP customizer I used a read/write width of 32 bits ($\geq TAU$) and depth of 4092 (thus address has 12 bits). This gives memory size of $32 \cdot 4096 = 131072$ bits. Expected BlockRAM usage was four 36K BRAM blocks, but during synthesis, the synthesis tool reduced the usage down to three 36K BRAM blocks (as the real used read/write width is only 22 bits).

In the entity BOOTSTRAPPING, I brought out BlockRAM interface as a set of ports so I could write into the memory from outside of the entity.

3.6.8 Pipelining

In version 1.7 I attempted to optimize the design to meet the timing requirements of 100 MHz clock cycle frequency. In entity BOOTSTRAPPING I added additional pipeline stages to two locations:
P_REG_DELAY: Added additional registers to store TRLWE Sample (TORUS_POLYS) between the output of ACC registers and input to EXTERNAL_PRODUCT entity. EXTERNAL_PRODUCT now starts one cycle later.
MATRIX_ACC Delay: Input data, all control signals, and index calculations inside MATRIX_ACC entity were delayed by one clock cycle.

This improvement added about 3% usage of slice registers, but decreased LUT usage by around 1.5% compared to version 1.6 (see Table 6.1).

3.7 Main

Entity MAIN encompasses this whole project and is the top-level module that is synthesized into a bitstream. The entity provides interface between external UART connection and inner Bootstrapping (see Algorithm 11) accelerator. Block schema of the implementation is shown in Figure 3.10.

3.7.1 RS232

RS232 standard was chosen as the communication protocol mainly for its simplicity. This standard is implemented in the entity RS232, which was provided to me by my supervisor.

Implemented RS232 uses two data wires RxD and TxD for external connection. Internal input and output are 8 bit wide and parity bits were not used. The interface runs at a baud rate of 115200 (parametrizable).

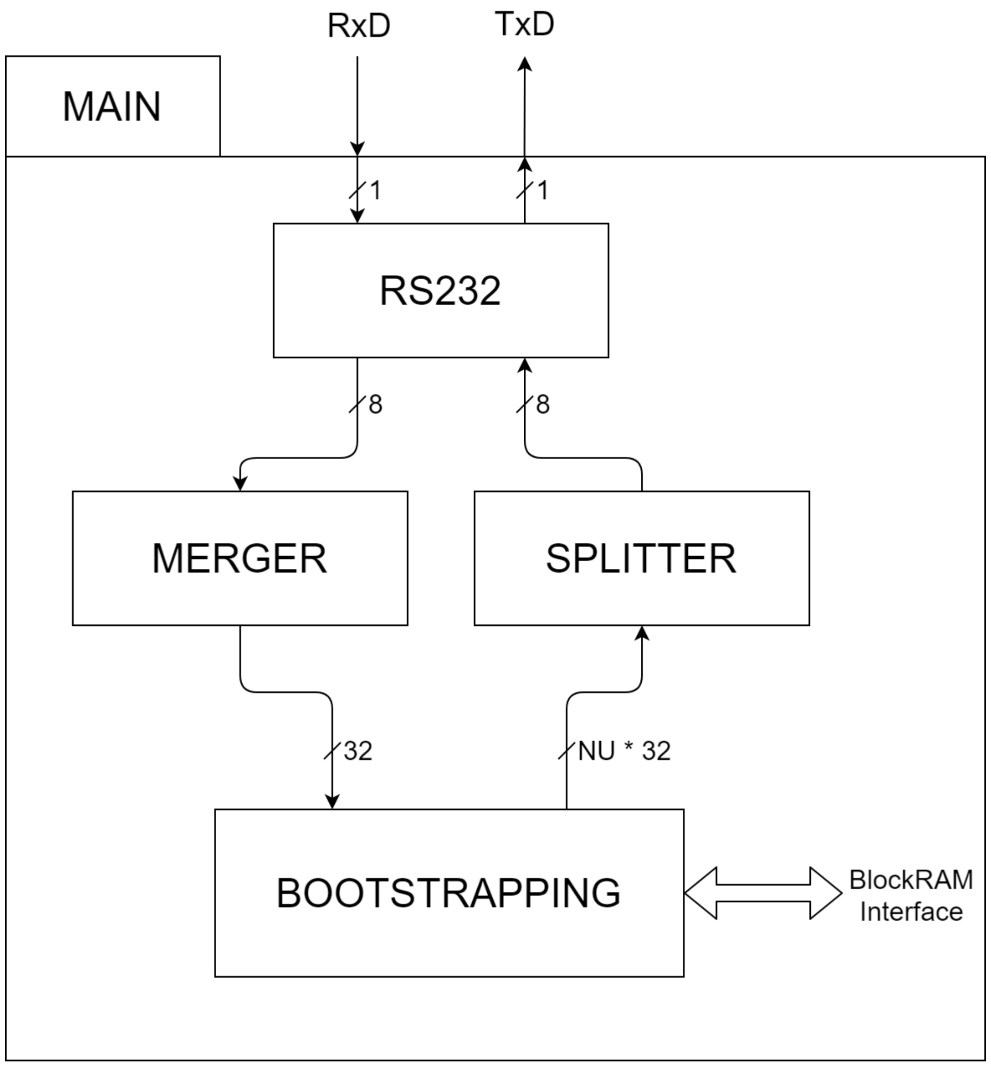

Figure 3.10: Top level MAIN entity

With Nexys3 FPGA, TxD, and RxD pins from RS232 were directly connected to an onboard micro-B USB port. The ZedBoard FPGA has an built-in UART controller, which is accessible from the ARM programming side. As the ARM core was not used in this project, I extended TxD and RxD ports to the Pmod connector, which was then wired to an external USB UART-to-RS232 converter.

3.7.2 Splitter and Merger

SPLITTER and MERGER are two entities that I created to simplify the connection between RS232 and the currently verified entity.

On every data arrival from the receiver side in RS232, entity MERGER concatenated 8 received bits into an array of 32 (NO_OF_INPUT_BITS) bits – a value of parameter NO_OF_INPUT_BITS has to always be a multiple of RS232 data width. Signal DONE is set to high every time a total of 32 bits are received. Formally output data are valid only for one cycle as the data are rewritten each time receiver receives additional input. I set NO_OF_INPUT_BITS to 32 bits as the input can now be easily represented with an integer data type and its hexadecimal representation is human-readable.

Entity SPLITTER works the opposite way. When the control signal SEND is high, $NU*32$ (NO_OF_OUTPUT_BITS) bits are loaded into the internal array and subsequently sent in 8-bit chunks to the transmitter in RS232. Value $NU*32$ is equal to the size of TLWE Sample with each coefficient padded to 32 bits. I did not bother to pre-parse the TLWE Sample to single coefficients in MAIN as they would be split into 8-bit chunks in SPLITTER either way and the functionality to split the array of bits was already built-in in the entity.

3.7.3 Communication Protocol

In this section, I will describe my communication protocol to control the operation of the FPGA. I implemented this protocol in version 1.6.

I have created 3 control codes:

0x4D4F4F4E: set FPGA to receive all 6 TRGSW Samples and save them into BlockRAM

0x594F4C4F: set FPGA to receive TRLWE and TLWE Samples, evaluate operation Bootstrapping and return TLWE Sample

0x484F4C44: abort current operation and reset FPGA to default state (does not clear contents of BlockRAM)

The values of those 3 control codes are completely arbitrary. Control codes differentiate from normal input data by setting the most significant byte to a non-zero value as every other input is only $TAU = 22$ bits wide and padded with zeros to 32 bits.

Status LEDs indicate the current state of the embedded controller on the lower 4 LEDs and the 4 least significant bits of counter on the 4 upper LEDs. On startup, the device initialized to a STARTUP state (indicated by only LED0 being on) and will not react to any input until switch SW0 is toggled on and off (LED1 turns on, LED0 turns off).

3.7.4 PLL

In version 1.5 timing requirements of the clock frequency of 100 MHz could no longer be met. I solved this by routing the CLK signal generated from the onboard crystal oscillator into the PLL circuit. I have chosen to halve the clock frequency down to 50 MHz as I was sure this would create enough leeway for synthesis without excessively slowing down the circuit. In version 1.7 I improved the circuit by adding pipeline stages to strategic places and raised the clock frequency back to 100 MHz. The increase in clock frequency didn't have much of an impact on the overall speed of the accelerator (see Table 6.4).

Chapter **4**

Demo Application

From the first version in ISE until Vivado version 1.4 I used the terminal.exe application on my Windows 10 operating system to send and receive data from the FPGA. Due to some fault (I believe some kind of buffering) I had to manually paste input data into the application instead of reading the data from an input file, as the FPGA would not receive all information and would return unusable results. As expected, this approach was quite prone to errors.

In version 1.5 I have written an application in C code to handle the transmission of data in and out of the FPGA (see enclosed file `./c_code/wtfhe.c`).

The C language was chosen since I aimed to run the code related to the WTFHE scheme on the built-in ARM core inside the ZedBoard SOC. The transfer to the ARM core was not carried out due to time constraints. Note that I renamed the names of variables in the snippets of the C code shown below for clarity sake.

4.1 Neural Network Demo

A small-scale Artificial Neural Network using the WTFHE scheme was chosen as a proof-of-concept demo. The flowchart on Figure 4.1 visualizes the sequence of implemented operations. I will elaborate upon each method below. Note the method `send_TRGSW()` was called at different points during major versions of the software, see section 4.1.4.

Evaluation of NN Matrices

I will explain how the WTFHE scheme is incorporated into the Neural Network evaluation with the help of Figure 4.2.

I used identical Neural Networks as in Jakub Klemsa's neural-baby-demo.rb (see enclosed file in `./Ruby/app/demo/` directory), which also served for verification. The Neural Network has three layers, the first layer with five perceptrons, the second layer with two and the last third layer with three perceptrons. Each

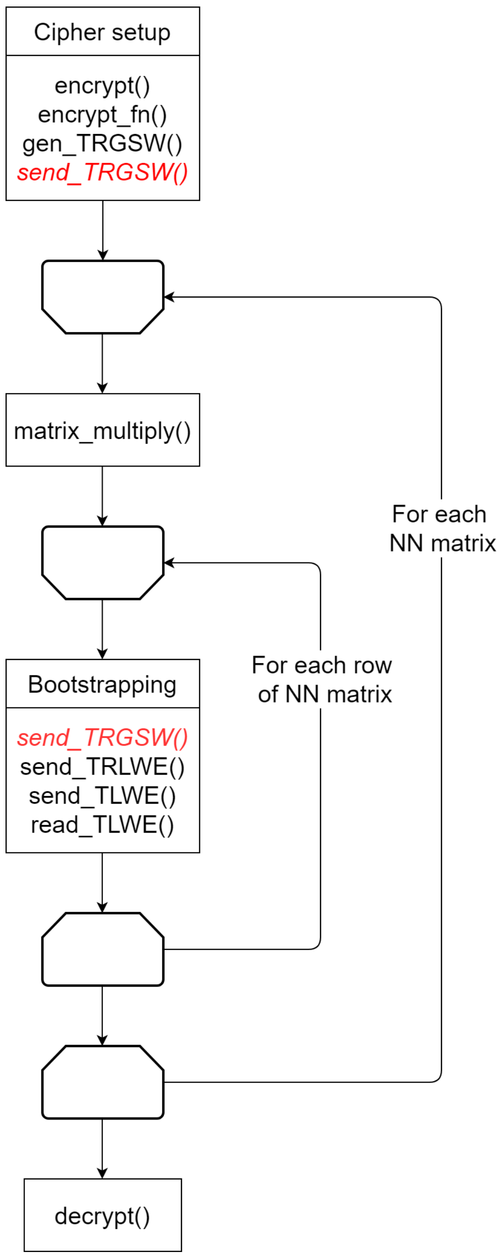

Figure 4.1: Demo flowchart

perceptron uses Bootstrapping as its activation function, so this gives ten runs of Bootstrapping operation on FPGA.

Weights of perceptrons are integers in range of $[0, 2^{PI} - 1]$ with the same values as in neural-baby-demo.rb. The weights are stored in matrices with dimensions given by a number of perceptrons in each layer and the number

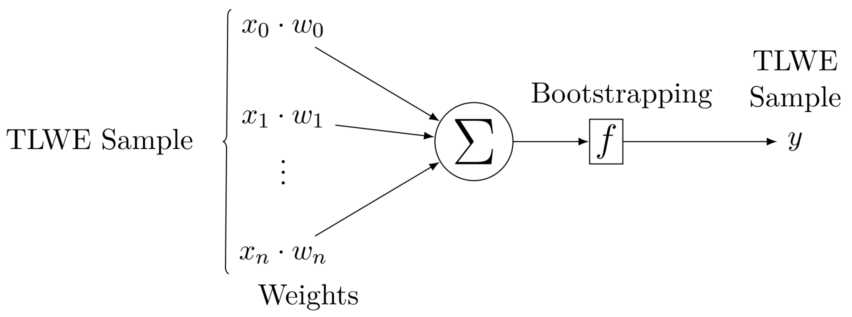

Figure 4.2: Perceptron in WTFHE scheme

of TLWE samples evaluated. The weights are arbitrary and only serve for demonstration purposes.

Plain text has a length of three. After encryption into TLWE samples the coefficients are represented as a matrix of Torus coefficients. This matrix is multiplied by a corresponding weight matrix to evaluate the output of the propagation function of every perceptron in a given layer at once. Since the elements of the resulting matrix are Torus coefficients I modulate them back into the 2^{TAU} range.

A final step is evaluation of a activation function. For encrypted data this is call of the Bootstrapping function on the FPGA, and for unencrypted it is a signum function.

4.1.1 Encryption and Decryption

Functions `encrypt()` and `decrypt()` mediate the encryption (decryption) of plain text into (from) TLWE Sample.

Encryption

Encryption is defined as follows:

$$b = m + \mathbf{k} \cdot \mathbf{a} + e \tag{4.1}$$

where m is plain text, $\mathbf{k}$ is a binary masking key, $\mathbf{c} = (\mathbf{a}, b)$ is a TLWE sample (i.e. TRLWE sample with polynomial degree $N = 1$) and e is random noise.

Plain text is an integer in the range of $[0, 2^{PI} - 1]$. To convert a plain text value into a Torus representation I shifted it to the left by $TAU - PI$ bits.

Binary masking key k is a vector of length $DIM = 4$ generated (currently static values for testing purposes) at the cipher setup. a is a vector of Torus members of the same length (DIM) and is randomly generated with each encryption.

Values of k and a are multiplied in a loop and summed to a shifted plaintext value to calculate b – the result of the encryption. Note that the result is again a Torus member, thus it is clamped back into range of $[0, 2^{TAU} - 1]$

The final operation is adding the noise e. The noise is random value in range $[-1, 1]$ (Torus value represented as an integer – i.e. decimal value of $\pm\frac{1}{2^{TAU}}$).

A snippet of my implementation in C is shown in Listing 4.1.

Listing 4.1: Encryption in C

```c
// Init a_i with random values
for (int i = 0; i < DIM; i++) {
  tlwe[i] = rand() % (1 << TAU);
}
// Init b to plain text value
tlwe[DIM] = pt << (TAU - PI);
// b += k_i * a_i
for (int i = 0; i < DIM; i++) {
  tlwe[DIM] += key[i] * tlwe[i];
}
tlwe[DIM] += (rand() % 3) - 1; // Add noise <-1,0,1> (e)
tlwe[DIM] = tlwe[DIM] % (1 << TAU); // Clamp to <0, 2^TAU>
```

Decryption

Decryption is the reverse operation to encryption and is defined as follows:

$$m = b - \mathbf{k} \cdot \mathbf{a} \tag{4.2}$$

where m is plain text, $\mathbf{k}$ is a binary masking key and $\mathbf{c} = (\mathbf{a}, b)$ is a TLWE sample.

The order of operations is in reverse compared to encryption. After the decryption, we are left with plain text with a non-zero amount of noise (i.e. $m+e$). Thus, before converting decrypted plain text from its Torus representation back into an integer, I extracted $(PI+1)^{th}$ bit and used it for rounding to the nearest integer.

Listing 4.2: Decryption in C

```c
// Init m to b
unsigned int pt = tlwe[DIM];
// m -= k_i * a_i
for (int i = 0; i < DIM; i++) {
  pt -= key[i] * tlwe[i];
}
pt = pt % (1 << TAU); // Clamp to <0, 2^TAU>
// Extract first decimal bit
unsigned int decimal = (pt >> (TAU - PI - 1)) & 1;
pt = pt >> (TAU - PI); // Convert from Torus to int
pt += decimal; // Round to a nearest integer
return pt % (1 << PI); // Clamp plain text to <0, 2^PI-1>
```

A snippet of my implementation in C is shown in Listing 4.2.

4.1.2 Function Encryption

Method `encrypt_fn()` encrypts function selected for evaluation into a TRLWE sample. Chosen function is encoded in a 2^{PI-1} long array of unsigned integers in range $[0, 2^{PI} - 1]$. Elements of this array hold result of function evaluation at points $0, 1, \ldots$ (i.e. $f[0], f[1], \ldots$). In this demo application I use signum function with values $(0, 1, 1, 1)$, which is one of the possible Neural Network activation functions.

Before usage in Bootstrapping the function, values have to be converted into a TRLWE sample. Since the function is known at the start, the first polynomial is zero-ed out. The second polynomial in TRLWE sample will hold unencrypted values of the provided function.

Each element of the original array is duplicated $\frac{N}{2^{PI-1}}$ times behind each other to get $N = 32$ values. Those values are then negacyclically rotated to the left by 2^{PI-1} positions. This operation is equal to centering the duplicated values over the original set point.

Snippet of my implementation in C is shown in Listing 4.3.

Listing 4.3: Function encryption in C

```c
// Extend the array
for (int i = 0; i < pow(2, PI - 1); i++) {
  for (int j = 0; j < N / pow(2, PI - 1); j++) {
    tmp[(i * N) / (int)pow(2, PI) + j] = signum[i] % (1 << PI);
  }
}
// Center the array
for (int i = 0; i < N; i++) {
  trlwe_inst[1][i] = // Line split
  = tmp[(i + N / (int)pow(2, PI)) % N] << (TAU - PI);
  trlwe_inst[0][i] = 0;
}
```

4.1.3 Generation of Bootstrapping Matrices

For security reasons, a new set of TRGSW samples (Bootstrapping matrices) should be generated for every user session. For clarity reasons I decided to split `gen_TRGSW` method into three parts.

The instantiation starts by preparing the keys. The private binary key of length DIM is extended to a polynomial of the order of N. This is done by padding the first $N - DIM = 28$ coefficients with a zero. The same private key is also used to calculate Bootstrapping keys – by multiplying pairs of the key to create triplets. A snippet of my implementation in C is shown in Listing 4.4.

Listing 4.4: Initialization of TRGSW keys

```c
// Extend the key
polynomial ext_key;
for (int i = 0; i < N - DIM; i++) {
  ext_key.coeff[i] = 0;
}
for (int i = 0; i < DIM; i++) {
  ext_key.coeff[N - DIM + i] = key[i];
}

// Create key tripplets
unsigned int b_keys[DIM * 3 / 2];
for (int i = 0; i < DIM / 2; i++) {
  b_keys[i * 3] = key[i * 2] * key[i * 2 + 1];
  b_keys[i * 3 + 1] = key[i * 2] * (1 - key[i * 2 + 1]);
  b_keys[i * 3 + 2] = (1 - key[i * 2]) * key[i * 2 + 1];
}
```

After all the keys are initialized, the first column of the matrix is filled with random noise. The second column is then filled row by row by multiplying the polynomial in the first column by the extended key. The multiplication is negacyclic and all coefficients are modulo 2^{TAU}.

Again, a snippet of my implementation can be seen in Listing 4.5.

Listing 4.5: Initialization of TRGSW Samples

```c
// Init first column to random noise
for (int il = 0; il < (DIM * 3 / 2); il++) {
  for (int j = 0; j < (K + 1) * L; j++) {
    for (int i = 0; i < N; i++) {
      matrices[il].poly[0][j].coeff[i] = rand() % (1 << TAU);
      matrices[il].poly[1][j].coeff[i] = 0;
}}}

// Calculate second column
for (int il = 0; il < (DIM * 3 / 2); il++) {
  for (int j = 0; j < (K + 1) * L; j++) {
  // Multiply 1st column with extended key
    poly_multiply(matrices[il].poly[0][j],  // Line split
    ext_key, &matrices[il].poly[1][j]);
    for (int i = 0; i < N; i++) {
      // Add noise <-1,0,1>
      matrices[il].poly[1][j].coeff[i] += (rand() % 3) - 1;
    }
}}
```

The last step is adding the Gadget Matrix (see Algorithm 1) to the TRGSW Samples. Since the key is binary, the Gadget matrix is added only to the TRGSW Samples which have a non-zero Bootstrapping key. Snippets of the code follows in Listing 4.6.

Listing 4.6: Adding the Gadget Matrix

```
// Add gadget matrix
unsigned int gadget = 1 << (TAU - GAMMA * L);

for (int i = 0; i < L; i++) {
  for (int j = 0; j < DIM * 3 / 2; j++) {
    if (b_keys[j] != 0) {
      for (int ii = 0; ii < K + 1; ii++) {
        matrices[j].poly[ii][L-i-1+ii*L].coeff[0]
        += gadget * b_keys[j];
      }
    }
  }
  gadget = gadget << GAMMA;
}
```

4.1.4 Communication Protocol

For the most part, communication with the FPGA consists of sending and receiving compute data. I haven't used any status messages and FPGA does not send any feedback except for the result of Bootstrapping operation. I added three control signals to manage what data were sent in version 1.6. to lessen communication overhead.

I have written three separate methods in the C code to send the data over (`send_TRGSW()`, `send_TRLWE()`, `send_TLWE()`) and one to receive the result (`read_TLWE()`). All these four methods iterate over every member of its given data class. Torus members of $TAU = 22$ bits are represented in 32-bit unsigned integers and since I use 8-bit data width in UART I split them into four parts of 8 bits before sending (and receiving) them over.

Technically I needed to send/receive only 24 bits (the lowest whole multiple of 8) instead of all 32. I choose to keep 32 bits simply because their hexadecimal representation is much more human-readable. Lowering the data size to 24 bits would speed up transaction speed, but would require a new set of control codes discussed in section 4.1.4.

Version 1.5

In the version 1.5 I called all three methods (`send_TRGSW()`, `send_TRLWE()`, `send_TLWE()`) to send compute data over to FPGA to initialize the computation of each Bootstrapping operation.

This resulted in a huge communication overhead as the majority of the data is in the TRGSW samples ($\sim$ 15kB ouf of $\sim$ 16kB), which are static during the evaluation of the whole Neural Network. I had to send TRGSW samples, since FPGA expected TRGSW, TRLWE and TLWE samples in this order to begin Bootstrapping.

Version 1.6 Onwards

To control dataflow over RS232 I created 3 control codes:

0x4D4F4F4E: set FPGA to receive all 6 TRGSW samples and save them into BlockRAM

0x594F4C4F: set FPGA to receive TRLWE and TLWE samples, evaluate operation Bootstrapping and return TLWE sample

0x484F4C44: abort current operation and reset FPGA to default state (does not clear contents of BlockRAM)

The codes 0x4D4F4F4E and 0x594F4C4F are sent first in `send_TRGSW()` and `send_TRLWE()` respectively. As the FPGA expects TRLWE and TLWE samples right after each other I did not add any control signal into `send_TLWE()` method.

Due to this change `send_TRGSW()` is called only once during cipher setup and I need to call `send_TRLWE()`, `send_TLWE()` and `read_TLWE()` only once for each Bootstrapping operation. This resulted in a significant speed-up (see Table 6.4).

Chapter 5

Simulation and Verification

An integral part of the implementation was the verification of the design. I used testbench files to check the validity of newly created entities and also redesign changes in those entities in between each major version.

5.1 Testbench Flowchart

The verification was done against a reference implementation in Ruby provided by Ing. Jakub Klemsa (see enclosed files in `./ruby/app/`). I used his ruby code was used to generate test data and the results were compared against the output of my circuit. I used either assert instructions in simulation or visual comparison to verify the correctness of the calculation.

All testbench files share one overarching structure visualized in Figure 5.1. Firstly, all inputs are loaded from their respective files and brought to the input of the currently verified entity. START signal is sent afterward. Blocking wait until follows for signal DONE. When signal DONE is received, series of assert signals compare the result against expected values.

5.2 Version 0.8 with Nexys3 and ISE

I started writing testbench files in the ISE project directly after the creation of few first entities. Note that these files were not maintained, the original files from the old ISE project were discarded and are no longer available.

TB_NEGA: Testbench file for NEGACYCLIC_MEM (see Algorithm 6).

Simple testbench where I checked the functionality of the NEGACYCLIC_MEM entity. I created manual input and verified the functionality only visually as it was sufficient.

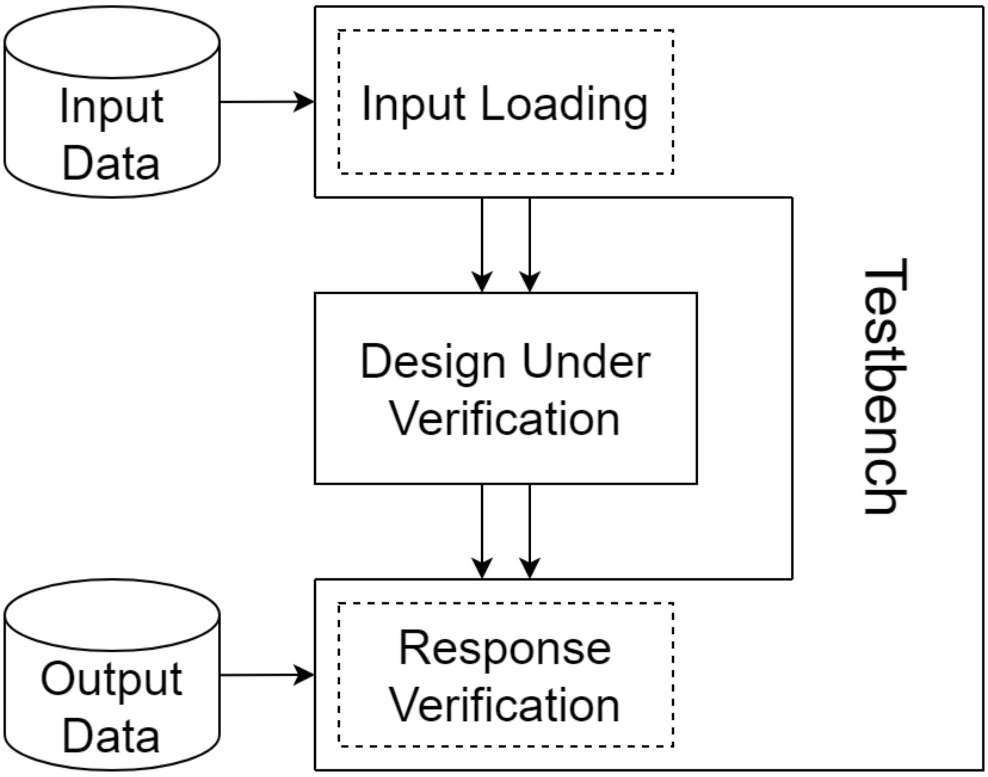

Figure 5.1: Testbench flowchart

TB_POLY: Testbench file for POLY_MULT (see Algorithm 5).

First testbench where I used input files to feed the data into the testbench. I used Ruby code (see Listing 5.1) to generate custom inputs. At first, I created basic polynomials with coefficients either being 1 or 0. For final verification, I used randomly generated polynomials.

Shown below are snippets of the Ruby code I used to create the test vectors (see enclosed file `./ruby/app/demo/polynomials-demo.rb`). Note that for clarity's sake I have shortened the list of coefficients with three dots and the instantiation of most of the polynomials is removed, as it was only repeating code. In this version, I was using a "Baby" set of parameters. The number of coefficients is $N = 16$, I also used $N = 4$ for small-scale tests. Number of polynomials used in both vectors is equal to $(K + 1) \cdot L = 8$. Finally, the binary precision of Torus coefficients is $TAU = 18$ and Integer coefficients $GAMMA = 12$ (This value was wrong and fixed later in version 1.0, see Table [3.1] for correct values).

Listing 5.1: Generating test vectors in Ruby for POLY_MULT and MATRIX_MULT

```
zz1 = PolyN.coeffs([0, ... ,0,1])
zz2 = ... # Integer polynomials
zzv = Vector[zz1, zz2, ... , zz8] # Vector of polynomials

t01 = PolyN.coeffs([TorusInt[1,TAU], ... ,TorusInt[1,TAU]])
t02 = ...  # Torus polynomials
ttv0 = Vector[t01, t02, ... , t08] # Vector of polynomials
ttv1 = Vector[t11, ... , t18]

r = zz1 * t01 # POLY_MULT calculation
r0 = zzv.dot ttv0 # one half of MATRIX_MULT calculation
r1 = zzv.dot ttv1 # second half of MATRIX_MULT calculation
// result is one Torus polynomial

# Methods to generate random coefficients
zz1 = PolyN.rand(N, Integer, GAMMA)
t01 = PolyN.rand(N, TorusInt, TAU)
```

TB_MATRIX_MULT: Testbench file for MATRIX_MULT (see Algorithm 9).

This testbench is similar to previous TB_POLY as the functionality of verified entities is identical, only the data types at input/output are different. I reused the ruby code from the previous section (Listing 5.1) to generate the test vectors.

TB_MAIN: Testbench file for MAIN (subsection 3.7)

The main purpose of this testbench was to verify the functionality of SPLITTER and MERGER(subsection 3.7.2) entities before syncretization of the whole project. As the one-bit period is 8 680 ns the overall real-time simulation time is quite long and I used this testbench only for a few small-scale tests.

5.3 Version 0.9 with Vivado and ZedBoard FPGA

In version 0.9 I ported the project over to Vivado. As the code remained unchanged, I only re-ran previous test benches when converting std_logic_vectors into UNSIGNED.

5.4 Version 1.0 onwards

TB_BOOTSTRAP: In version 1.0 I implemented operation Bootstrapping (Algorithm 11) and added a corresponding testbench TB_BOOTSTRAP, which remains as the key testbench, since TB_MAIN is unusable due to its excessive real-time simulation time.

I used Ruby code (see Listing 5.2, enclosed file `./Ruby/app/demo/ext-product.rb`) to generate test vectors. The code is an unrolled version of the Bootstrapping procedure. I exported values of TRGSW, TRLWE, and TLWE samples into a file and also saved the value of acc after the last operation (acc is a TLWE sample). Note that I skipped creating testbench for External Product (see subsection 3.5) as the operation is only an extension of matrix multiplication and a part of the blind rotate loop, thus I could visually check values of variables before and after against expected values.

Listing 5.2: Generating test vectors in Ruby for BOOTSTRAPPING

```ruby
# Cipher initialization
tg = TRGSW.new(tau, deg, k, gamma, l, alpha, key)
tr = tg.TRLWE

# Plaintexts for Integer and Torus Polynomial
p_int = 3
p_tpoly = PolyN.coeffs((1..deg).to_a.map{|p|TorusInt[p,prec]})

# TRLWE sample
p = tr.sample(p_tpoly)

# TRGSW samples
m1 = tg.sample(p_int)
m2 = tg.sample(p_int)
m3 = tg.sample(p_int)
m4 = tg.sample(p_int)
m5 = tg.sample(p_int)
m6 = tg.sample(p_int)

# TLWE sample (not random values)
xb1 = PolyN.x_to(3,deg)
xb2 = PolyN.x_to(6,deg)
xb12 = PolyN.x_to(3 + 6,deg)
xb3 = PolyN.x_to(5,deg)
xb4 = PolyN.x_to(8,deg)
xb34 = PolyN.x_to(5 + 8,deg)
xb5 = PolyN.x_to(-8,deg)

# Improved blind rotate
acc = xb5 * p
acc = (((xb12 - 1) * m1) + ((xb1 - 1) * m2) + ((xb2 - 1) * m3))
  .ext(acc, gamma, l) + acc
acc = (((xb34 - 1) * m4) + ((xb3 - 1) * m5) + ((xb4 - 1) * m6))
  .ext(acc, gamma, l) + acc
```

During versions 1.0 through 1.3 I used test vectors generated with parameters corresponding to “Infant” set (see Table 3.1) – although with a wrong value of $TAU = 18$ – saved in files named matrix_in_18 (for input) and tlwe_out_18 (for output). From version 1.4 onwards I re-generated test vectors with the “Baby” set of parameters and saved them in files matrix_in_22 and tlwe_out_22.

5.5 Demo Verification

Listing 5.3: Neural Network demo in Ruby

```ruby
# Cipher Initialization
w = WTFHE.init_baby

# Input Vector
p_in = Vector[1,1,0]

# NN Initiazation (omitted for clarity)
nn = [ Matrix[ [4, 6, 3], [ ...

# Encryption
c_in = p_in.map{|p| w.encrypt(p) }

# Activation function - signum
def act_fn(w, v)
  case v
    when TrlweSample
      w.bootstrap(v, [0,1,1,1])
    when Numeric
      [0,1,1,1,0,7,7,7][v % 8]
    else
      raise
  end
end


# Evaluate one layer of NN
def eval_layer(w, input, synapses)
  (synapses * input).map{|v| act_fn(w, v) }
end

nn_p = p_in.clone
nn_c = c_in.clone
# Evaluate NN
nn.each do |l_syn|
  nn_p = eval_layer(w, nn_p, l_syn)
  nn_c = eval_layer(w, nn_c, l_syn)
end

# Check Homomorphism
puts "NN#{p_in.to_a} = #{nn_p.to_a}"
puts "FHE^-1{ NN[ FHE(#{p_in.to_a.join('), FHE(')}) ] } =
  #{nn_c.map{|c| w.decrypt(c) }.to_a}"
```

In version 1.5 I added a demo application, which also needed to be verified. Verification of my application was done against Ing. Jakub Klemsa’s Ruby code shown in Listing 5.3 (see enclosed file `./Ruby/app/demo/neural-baby-demo.rb`). The Ruby code runs dual calculations, both for unencrypted and encrypted data to check the validity of homomorphism.

Line `w = WTFHE.init_baby` initializes cipher with “Baby” set of parameters (see Table 3.1). This line also creates TRGSW Samples, which are then constant for the rest of the run.

Variable `p_in` holds plain text, and its TLWE encrypted form is saved in variable `c_in`. Neural Networks stored in variable `nn` have arbitrary values and have dimensions of $5 \times 3, 2 \times 5$ and 3×2. Dimensions were chosen so that three TLWE Samples, each with a length of five $(DIM + 1)$, on the input would at the end result in three TLWE Samples again.

Method `act_fn` defines activation function signum for both encrypted (Bootstrapping operation) and unencrypted inputs (negacyclically extended array of images of the signum function). This method is called for each TLWE Sample of input in method `eval_layer` (i.e. each row after matrix multiplication of “synapses · input”).

Method `eval_layer` is called in a loop for each defined neural matrix both on encrypted and unencrypted data. The results of both evaluations are printed at the end in a decrypted form to verify the correctness of the calculation.

I saved variables of one run of Ruby code into a file to have static values to validate my code against. The saved variables are TRGSW Samples (`w.BK[k][l].sample[j,i].send(:coeffs).map{|t| t.numer}`) prints one polynomial of one TRGSW Sample), input TLWE Samples (`c_in`) and output TLWE Samples (`nn_c` after evaluation).

Chapter 6

Measurements and Results

In this chapter, I will describe the performance and resource requirements of the FPGA Bootstrapping accelerator.

6.1 Resource Requirements

The Table 6.1 lists usage of FPGA resources for each major version after the implementation step in Vivado. All these measurements represent resource usage on ZedBoard FPGA with Zynq-7000 SOC (part identification number is xc7z020clg484-3, [13]). The previously used Nexys3 board was not benchmarked.

Versions 0.9 through 1.3 represent FPGA usage corresponding to a "Infant" set (see Table 3.1) of parameters. The initial implementation of Bootstrapping (see subsection 3.6) in version 1.0 could not be synthesized due to excessive requirements. In version 1.1 I added BlockRAM to store input operands, saw a significant drop in slice usage, and the design was successfully implemented.

Versions 1.1 through 1.3 saw a steady decline in slice usage as more logic was offloaded into DSP48 blocks. Due to the optimizations with DSP the usage lessened even though TAU was raised to a correct value of 20 in version 1.2.

Versions 1.4 through 1.7 represent usage for "Baby" set (see Table 3.1) of parameters. Compared to the "Infant" set, the "Baby" set uses considerably more resources. Also, timing requirements started to be an issue with versions 1.5 and 1.6 unable to meet timings for 100 MHz clock frequency. The reduction of usage between 1.4 and 1.5 versions is caused by a fix in the Sample Extract algorithm (see Algorithm 3). As expected, pipelining added in version 1.7 (see subsection 3.6.8) increased usage of slice registers, but reduced usage of slice LUTs with final clock frequency back at 100 MHz.

	v0.9	v1.0	v1.1	v1.2	v1.3	v1.4	v1.5	v1.6	v1.7	v1.7
Slice LUTs [%]	7.8	109.2	15.5	14.4	13.2	31.8	29.9	30.0	28.6	148.8
Slice Registers [%]	7.9	51.9	8.8	9.1	8.5	21.1	20.5	20.5	23.7	120.4
DSP48s [%]	14.6	N/A	0	14.6	14.6	29.1	29.1	29.1	29.1	58.2
BlockRAM [%]	0	N/A	0.7	1.4	1.4	2.1	2.1	2.1	2.1	20.7
Min clock period [ns]	6.32	N/A	8.97	8.87	9.16	9.77	14.76	16.51	9.73	N/A
TAU	18	18	18	20	20	22	22	22	22	24
$GAMMA$	12	3	3	3	3	3	3	3	3	1
N	16	16	16	16	16	32	32	32	32	64
L	N/A	4	4	4	4	5	5	5	5	17
NU	N/A	4	4	4	4	5	5	5	5	6

Table 6.1: Zynq-7000 FPGA Resource Usage

Version	v1.0	v1.1-v1.3	v1.4	v1.5-v1.6	v1.7
Parameter set	“Infant”	“Infant”	“Baby”	“Baby”	“Baby”
Number of clock cycles	389	3 433	8 477	8 477	8 479
FPGA Clock frequency [MHz]	100	100	100	50	100
Real-time duration [ns]	3 890	34 330	84 770	169 540	84 790

Table 6.2: Simulated Bootstrapping Duration in FPGA

The record in the last column is for the ”PI=4” set of parameters. Those parameters are experimental aiming to assess the scalability of the design. The requirements proved to be out of bounds for used FPGA and devised design. As the BlockRAM usage is still fairly low, there is a possibility to store more internal variables in RAM instead of slice registers. The second approach is to reduce implemented parallelism and convert selected parts to a more sequential approach.

6.1.1 Detailed Breakdown

The Table 6.3 shows a detailed breakdown of resource usage of the most significant entities. As expected, entity Bootstrapping takes the vast majority of resources and the UART connection is comparably insignificant. One of the more demanding entities in Bootstrapping is Mat_ACC, which stores one whole TRGSW Sample. The TRGSW Sample is stored in a register field and from a slice usage standpoint, it would be beneficial to store the temporary variable in a BlockRAM. The register usage of Mat_Acc closely follows size of one TRGSW Sample ($14202 > 14080 = (K+1)\cdot(K+1)\cdot L\cdot N\cdot TAU$).The second biggest entity is Ext_Prod, the main compute entity. Mat_Mult inside Ext_Prod occupies most of LUTs as logic, while takes only about two-thirds of registers, as the TRLWE Sample rounded down before passed to Mat_Mult. Interestingly, Poly_Mult (multiplication of integer and Torus polynomial) entity is lighter on resources than Poly_Add (sum of two Torus polynomials), assumed due to integer polynomials being smaller and the usage of DSPs to store the result. Register usage of Poly_Add and Poly_Reg directly correlates to the size of polynomials ($1408 = (K+1)\cdot N\cdot TAU$).

Entity Name	LUTs	Registers	DSP48s	BRAMs
Available	53 200	106 400	220	140
Used Total	15 208	25 256	64	3
Bootstrapping	14 963	24 923	64	3
Ext_Prod	7 118	4 980	64	0
Mat_Mult	7 114	3 058	64	0
Poly_Mult [x2]	1 464	1 618	64	0
Mat_Acc	5 316	14 202	0	0
Poly_Add	1 408	1 408	0	0
Poly_Reg	0	1 408	0	0

Table 6.3: Entity Resource Usage

6.2 Time Measurements

In this section, I will elaborate upon time measurements either in a simulation or real-world deployment.

6.2.1 Bootstrapping Time on FPGA

The Table 6.2 shows the number of clock cycles and calculated real-time duration of Bootstrapping operation. The cycles measured are the difference between raising control signal START and receiving control signal DONE in the entity BOOTSTRAPPING (respective BOOTSTRAPPING_RAM for version 1.1 onwards). In versions that contain BlockRAM, the loading times are not considered, since the duration is heavily dependant on UART transmission speed, which was unreasonable to simulate.

Register-based version 1.0 (see subsection 3.6) proved to be a very fast theoretical solution taking just under 400 cycles to compute. This speed was achieved by a parallel design at the cost of excessive resource requirements.

Designs using BlockRAM (v1.1 onwards) were discovered to be about ten times slower than version 1.0 using the same set of parameters. The second big duration increase was observed with raising parameters to the “Infant” set of parameters (values of N and L having the greatest impact, see Table 6.1). Improvements in any version after 1.1 did not considerably affect compute time.

6.2.2 Deployed Neural Network Demo Measurements

The Table 6.4 shows measured times for either Bootstrapping or whole Neural Network demo throughout different versions and systems. Due to issues with the original Unix virtual guest OS (USB connectivity issues and disconnects), I swapped to Ubuntu Mate virtual guest OS. The specific versions of both systems were Ubuntu 18.04.5 LTS (”Ubuntu”) and Ubuntu Mate 20.04.5 LTS (”Mate”).

Version	v1.5	v1.5	v1.6	v1.7	Ruby
System	Ubuntu	Mate	Mate	Mate	Mate
Bootstrapping average time [ms]	19 577.4	1 372.5	35.63	35.56	N/A
NN evaluation average time [ms]	195 879.3	13 724.8	1 692.1	1 692.9	4 314
Host system config	Windows 10	CPU: R5 1600		Ruby v2.7.0	

Table 6.4: Real-world Time Measurements

Bootstrapping average time is the measured time it took for one operation of Bootstrapping on FPGA along with the transmission of relevant data. This includes transmission of input data – methods `send_TRGSW()`, `send_TRLWE()`, `send_TLWE()` (or `send_TRLWE()`, `send_TLWE()` methods for versions 1.6 and 1.7) – and receiving data with `read_TRLWE()`.

Measured Neural Network evaluation time spans over matrix multiplication of all three Neural Networks and one Bootstrapping operation for every row in every matrix (i.e. ten Bootstrapping operations in total). Thus, method `send_TRGSW()` is called ten times in version 1.5 and only once in 1.6 onwards for each Neural Network evaluation.

The results are an average of 100 completed Neural Network runs (50 for v1.5 on Ubuntu due to constant crashing) and 1000 Boostrapping runs.

The connectivity issues were so severe on the ”Ubuntu” system, that the same version 1.5 runs more than ten times slower than on the ”Mate” system. The change of communication protocol in version 1.6 (see subsection 4.1.4) considerably improved the average time of Bootstrapping operation and the overall run-time was reduced below the run-time of reference ruby implementation.

As the FPGA compute time is in order of hundreds of μs (see Table 6.4), measured results in any version show a huge communication overhead.

Since the average times of the whole Neural Network, evaluation is similar in both 1.6 and 1.7 versions, the observed speed-up of Bootstrapping operation can not be attributed to the increase of FPGA frequency from 50 to 100 MHz.

Chapter 7

Future Work

In this chapter, I will describe steps that could be taken in the future to improve the performance of the FPGA accelerator.

7.1 Optimization

The first steps taken could be a few simple optimizations.

Pre-calculate partial step of BlindRotate: The loop inside Blind Rotate Algorithm 2 needs to calculate following expression 7.1 before the result is sent to External Product entity:

$$MAT \leftarrow ((X^{a_{2i-1}+a_{2i}} - 1)BK_{3i-2} + (X^{a_{2i-1}} - 1)BK_{3i-1} + (X^{a_{2i}} - 1)BK_{3i}) \tag{7.1}$$

In my implementation of Mat_ACC, which calculates the expression 7.1, I decided to split the calculations into two phases (see subsection 3.6.3). The second phase is relevant for this potential optimization:

$$MAT \leftarrow MAT - BK_{3i-2} - BK_{3i-1} - BK_{3i} \tag{7.2}$$

The expression 7.2 does not rely on the value of TLWE Sample, thus can be pre-calculated at the start of the session and the result would be saved in the FPGA BlockRAM. This optimization would reduce the usage of LUTs by a few thousand (for the “Baby” version) and several hundreds of clock cycles. The register usage of Mat_ACC is still directly linked to a size of one TRGSW Sample, thus no savings would be done there.

Removing one instance of POLY_MULT : Currently, as TRGSW and TRLWE Samples have two columns and two polynomial respectively, there are two instances of POLY_MULT working in parallel. This puts the upper bound on N to 64, as the design uses $2 \cdot N$ DSP48 blocks out of 220 available.

Converting parallel designs to sequential: The biggest issue of scaling the design is LUT/Register usage (see the last column of Table 6.1). There are two approaches that could be implemented simultaneously. One is removing parallelism and converting designs to a more sequential type, and the second is offloading more temporary results into BlockRAM and even into onboard DDR3. Those improvements are expected to be much more extensive and would require a major overhaul of implemented entities.

7.2 ARM Integration

Originally, the idea of ARM core integration was to generate TRGSW Samples locally and load them directly into BlockRAM as this would save a significant part of data needed to be transmitted over a slow UART connection. This was later rejected because the Bootstrapping keys are private and should not be shared with the accelerator.

The current idea is to use FPGA as an accelerator of External Product only. The External Product uses roughly half of LUTs and a fifth of Registers compared to the whole Bootstrapping entity (see Table 6.3). As long as the ARM core can compute the expression 7.1 and save the result into BlockRAM in a comparable time to the Bootstrapping entity – which takes roughly $2 \cdot 3 \cdot$ TRGSW Sample size in clock cycles – the overall speed would be comparable, but with much lower resource usage.

A side benefit of routing the data through the ARM core is an increased bandwidth of USB connection over UART. As one of the main problems is communication overhead, this change would increase the performance of the accelerator without any major redesign.

7.3 GPU Acceleration

Overall, the current version of the WTFHE scheme in its full version is still very resource-heavy. The degree of the polynomials is expected to be 1024 at minimum. The sheer size might make it prohibitive to FPGA-based acceleration, but discrete GPUs have massive amounts of cores which can also do multiplication in parallel. The amounts of VRAM are also usually higher compared to what is available to FPGA, further benefiting the proposal.

Conclusion

In this thesis I presented the current version of the WTFHE scheme and the possibility of using an FPGA as a hardware accelerator to increase the performance of the cipher.

I implemented the hardware accelerator in VHDL language, running on ZedBoard Zynq-7000 SOC. I connected the FPGA with UART over a USB connection with a custom demo application written in C language running a simple Neural Network Evaluation. The demo application runs the same algorithm, but with no encryption, thus serving as a validator to the FPGA computation.

During the implementation process, I created a number of testbench files validating the functionality of the FPGA. A reference implementation in Ruby provided by Ing. Jakub Klemsa served as a test vector generator for the testbench files.

I tested several sets of WTFHE parameters intending to find the performance of the design, possible scalability, and resource requirements. A test set codenamed "Baby" with $3 = PI$-bit plaintext, thus range of $[0, 2^3]$, was successfully implemented. The accelerator then evaluated a simple Neural Network faster than the reference solution in Ruby. However, the design suffers from a significant communication overhead due to the amount of data sent over a slow UART connection. Routing the data through the built-in ARM core and using a standard USB connection to PC could increase the performance of the accelerator.

A test set of parameters codenamed "$PI = 4$" with 4-bit plaintext size exceeded the available resources in FPGA and could not be synthesized. I anticipate, that with a redesign of some components of the accelerator, the "$PI = 4$" version could fit into the ZedBoard SOC. However, as the real-world parameters are still significantly higher to guarantee encryption security, I expect a GPU assisted acceleration might be a more promising direction.

Bibliography

[1] Craig Gentry and Dan Boneh, *A fully homomorphic encryption scheme, volume 20*, Standford University, 2009.

[2] Ilaria Chilloti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène, *Tfhe: Fast fully homomorphic encryption over the torus*, Journal of Cryptology, 33(1):34-91, 2020.

[3] Google Cloud and Vertex AI. [Online]. Available: `https://cloud.google.com/vertex-ai`.

[4] Microsoft and Azure Machine Learning. [Online]. Available: `https://azure.microsoft.com/en-us/services/machine-learning/`.

[5] IBM and Watson OpenScale. [Online]. Available: `https://www.ibm.com/cloud/watson-openscale`.

[6] Thore Graepel, Kristin Lauter, and Michael Naehrig, *Machine learning on encrypted data*, International Conference on Information Security and Cryptology, Springer, 2012.

[7] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig, *Crypto-nets: Neural networks over encrypted data*, arXiv preprint, arXiv:1412.6181, 2014.

[8] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier, *Fast homomorphic evaluation of deep discretized neural networks*, Annual International Cryptology Conference, Springer, 2018.

[9] Jakub Klemsa, *Setting up efficient tfhe parameters for multivalue plaintexts and multiple additions*, Cryptology ePrint Archive, Report 2021/634, `https://eprint.iacr.org/2021/634`, 2021.

[10] Oded Regev, *On lattices, learning with errors, random linear codes, and cryptography*, Journal of the ACM (JACM), 56(6):1-40, 2009.

[11] Sergiu Carpov, Malika Izabachène, and Victor Mollimard, *New techniques for multi-value input homomorphic evaluation and applications*, Cryptographer's Track at the RSA Conference, Springer, 2009.

[12] Tanping Zhou, Xiaoyuan Yang, Longfei Liu, Wei Zhang, and Ningbo Li, *Faster bootstrapping with multiple addends*, IEEE Access, 6:49868-49876, 2018.

[13] Xilinx, *Zynq-7000 soc data sheet: Overview.* [Online]. Available: `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`.

Appendix **A**

Acronyms

ANN	Artificial Neural Network
ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CPU	Central Processing Unit
DDR	Double Data Rate
DSP	Digital Signal Processor
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
LHE	Leveled Homomorphic Encryption
LWE	Learning With Errors
LUT	Look-Up Table
MAC	Multiply-and-Accumulate
MLaaS	Machine Learning as a Service
PC	Personal Computer
PLL	Phase-Locked Loop
RAM	Random Access Memory

RISC Reduced Instruction Set Computer

SoC System on a Chip

TFHE Fast Fully Homomorphic Encryption over the Torus

TLWE Torus TLWE

TRGSW Torus-Ring Gentry-Sahai-Waters

TRLWE Torus-Ring LWE

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

WTFHE netWork-ready TFHE

Appendix B

Contents of enclosed storage media

c_code	Directory with C code
└ wtfhe.c	Demo application in C
data	Input/Output data for testbenches
ruby	reference Ruby implementation by Ing. Jakub Klemsa
└ app	Ruby subfolder
└ bin	Directory with Ruby main file
└ demo	Directory with Ruby test files
thesis	Thesis tex source files
v0_9_matrix_multiplication	Vivado project for v0.9
v1_0_boots_registers	Vivado project for v1.0
v1_1_boots_bram_no_dsp_tau_18	Vivado project for v1.1
v1_2_boots_bram_dsp_tau_20	Vivado project for v1.2
v1_3_boots_bram_better_dsp	Vivado project for v1.3
v1_4_boots_pi_3_scalable	Vivado project for v1.4
v1_5_neural_demo_50_mhz	Vivado project for v1.5
v1_6_better_neural	Vivado project for v1.6
v1_7_final	Final Vivado project - v1.7
v1_7_pi_4_test	Scalability test - v1.7
master_thesis_chytry.pdf	the thesis text in PDF format
master_thesis_chytry.zip	the thesis text in zip file
WTFHE_v1_7.bit	Synthetized v1.7 bistream for Zynq 7000 SoC
readme.mb	read-me with file/directory description