Fully-Homomorphic Encryption for Real-Time Control

An FPGA Implementation

P. Stobbe

Technische Universiteit Delft

TU Delft

Fully-Homomorphic Encryption for Real-Time Control

An FPGA Implementation

by

P. Stobbe

Student number: 4601858
Project duration: August 31, 2020 – August 31, 2021
Thesis committee: Dr. R. Ferrari, TU Delft, supervisor
Ir. T. Keijzer. TU Delft, supervisor

TU Delft Delft University of Technology

Preface

In 2016 I started my bachelors in mechanical engineering, unaware of the gradual change of direction from everything physical, to the more theoretical. I always loved computers and software, but was hesitant to learn how to program, let alone try to understand how computers work on a physical level. The programming courses and mechatronics subjects gave me a nice and gentle introduction into the more abstract. My minor finally swayed me as I was taught about the fundamentals of IC-chips and electronics. It made me aware I wanted to understand how to build cyber-physical systems from the ground up.
When I started my masters in systems and control I wanted to specialise in real-time control systems. Throughout the varying courses I noticed that I particularly enjoyed the mathematics behind all the different ways of constructing control systems.
My thesis supervisor had conducted research into security and privacy preserving control as well as attacks on these schemes. One aspect that my supervisor and his research team had not yet researched is homomorphic encryption and how it could be applied in feedback control. I was only aware of some cursory aspects of encryption, but the mathematical problems that encryption poses intrigued me. Implementing this form of encryption would also likely be a programming challenge, which is right up my alley. As homomorphic encryption has only seen implementation in a few control schemes, it meant I would have to consider many aspects of control in a new setting. A very daunting but exciting prospect and so I decided to make it the research topic for my thesis.

As I suspected, to understand encryption I had to learn more about abstract algebra. This was mostly new to me, which made it a real challenge, but learning to understand new aspects of abstract algebra was always very rewarding. Even knowing the relevant mathematics, understanding how encryption schemes are constructed was quite a task. Underlying theory can differ wildly between schemes, making comparing them very tricky.

Safe to say, the research I have done has opened my eyes to a whole host of interesting and exciting engineering problems that I believe will help shape our future.

I would like to thank my daily supervisor Twan Keijzer and my professor Riccardo Ferrari for all the help and support and introducing me to a field of research I would not have sought out, but ended up fascinated by.

P. Stobbe
Delft, January 2021

Abstract

Currently, encryption is part of daily life, from commercial to industrial applications. Secure, long-distance communication is vital to the safe and reliable operation of industrial feedback control. Utilising public networks as a medium is often cost-effective at the risk of a security breach. Current industrial feedback control systems generally utilise end-to-end encryption to communicate control signals and gains securely. Data has to be decrypted for processing. Homomorphic encryption allows for manipulation of encrypted data. This eliminates the need for decryption to update controller states and calculating control effort. Partially Homomorphic Encryption supports either multiplication or addition of encrypted values, whereas Fully Homomorphic Encryption allows for both. Besides being flexible, Fully Homomorphic Encryption schemes are thought to be quantum safe. Unfortunately, Fully Homomorphic Encryption schemes are computationally expensive limiting practical applications. This thesis presents an enhanced version of the of the popular Fully Homomorphic Encryption scheme by Gentry. The encryption scheme is enhanced through the introduction of three alterations. New notation is introduced that streamlines its description. The main functions that compose the encryption scheme are all replaced with analytical equivalents. The so called reduced cipher is introduced. Rewriting the encryption scheme using the improved notation, analytical functions and reduced cipher leads to a more computationally and memory efficient implementation. The alterations make the encryption more suitable for implementation on Field Programmable Gate Arrays which decreases compute time. Such an implementation is presented and used to demonstrate the efficacy of the enhanced encryption scheme.

Contents

1 Introduction

The modern technology that powers our world requires control, ranging from simple on-off control to highly complex model driven control. To ensure correct operation and prevent sabotage or other interference, it important to prevent malicious attacks. Attacks can range from physical attacks to hacking. The type of security that a technology requires depends on factors such as functionality, location of deployment as well as how many times it is deployed. Take for example a vending machine. A vending machine is built to protect its contents and the machine itself requires very little security. They are usually left unsupervised and so they are built to withstand a certain amount of abuse. The contents are of relatively low cost. Even if some its contents are stolen, the profit still outweighs the risk. The machine itself is a lot more expensive and so if the machine would be broken or stolen, that would be much more of a problem. The likelihood of a vending machine being stolen or taken away is very low however, because there is not much incentive to steal the whole machine. There is unlikely to be a large market for stolen vending machines. Therefore the profit that vending machines produce outweighs the price of fixing or replacing them. When it comes to factory equipment, it is usually kept in an enclosed and secured space, where it is also feasible to supervise the equipment with security cameras and security staff. If different machines have to communicate, they are usually connected through a local network, wired or wirelessly. These networks can usually only be accessed by authorised computers that are on site. This means that it is impossible to perform a malicious attack using the internet. Such a security measure is also known as an air gap.

What if a system requires the remote operation across large distances? One such example is a pipeline. Pipelines can be tens of thousands of kilometres long [14] and so the majority of pipeline is usually unsupervised. To operate pipelines remotely, controllers on the pipeline are usually connected to the internet. Long distance cable networks cannot feasibly be monitored along the entire length and so it would be possible to physically connect to the network. It would be possible to connect a controller using a bespoke network of cables, however this would be very costly and it would suffer the same problem as a public internet connection. Without any security measures in place, any person could connect to the pipeline. To be able to send and retrieve control and sensor data at these distances securely, the industry utilises encryption. Encryption is the act of scrambling data, such that only those who have the corresponding key can unscramble the data. This means that an insecure network can be used to share data whilst being inaccessible to eavesdroppers or saboteurs.

Another way to secure such a system is by using security through obscurity. Security is derived from designing a system to be deliberately obtuse and keeping the inner workings secret, so that potential attackers will not understand the system well enough to attack it. An example of obscurity is Intel's Converged Security and Management Engine [48], CSME, which is part of many CPU's that Intel produces. IT experts such as Damian Zammit have pointed out that Intel's CSME has parts that are not secured in any way which poses a great security risk [55]. The technical specification of the CSME is kept secret, which is currently the only measure that is keeping the CSME secure. Of course if such a system is reverse engineered, security through obscurity is nullified. Hence encryption is the more robust security measure, making encryption the most popular solution.

Currently, the most widely used type of encryption scheme is end-to-end encryption. This means that when two devices want to exchange information, information is encrypted, sent and then decrypted. This type is

very effective, however if some saboteur manages to get access to either device, they can breach security. This is not usually a problem in situations such as that of two people sharing messages. When a control system is concerned however, there might be rogue agents with insider access, or authorised personnel may accidentally cause a security breach by leaking passwords. One such incident caused the Colonial Pipeline hack [50]. A password was leaked, which allowed hackers to shut down the entire 8850 kilometre long Colonial pipeline. A type of encryption that could help prevent such a hack is homomorphic encrytion, HME.
HME is a type of encyption that allows mathematical operations to be performed on encypted data. This makes it possible to design a system where only the device that performs the encryption is able to dencrypt any data. Take the following example. A section of pipeline equipped with a controllable valve and a flow sensor has to control its throughput, but to do so requires the coordination of a network of such sections. With HME it is possible to perform calculations on encrypted data. Hence multiple sections of pipe can encrypt data, send it of to a main controller which calculates how far the valves should be opened. The controller can do this without decrypting any of the incoming signals. Even if a rogue agent has direct access to the incoming and outgoing signals of controller, they would not be able to interfere because all data is encrypted. Only when instructions are received at the controllable valve of the pipe sections, are they decrypted and read and so the only way to gain access to the unencrypted data would be to physically breach the valve interface.
Encryption that secures against rogue agents with insider access is especially relevant currently. Applications such as cloud based services and distributed computation have become more prevalent. A company that stores data, or performs computations for clients will want to keep the data secure. If a client wants to alter their data remotely, that data has to be decrypted on the server that the data is stored on. If encryption keys have to be stored on a server, they could be stolen by a rogue agent with insider access. This has never been more relevant, as in 2010 a google employee used his access to information to stalk and threaten minors [8]. Other than large events like these, there are many smaller insider misuse. Verizon has analysed more than 100,000 security incidents of which more than 10% was perpetrated by an insider [42].

1.1. Recent work

There are currently two types of encryption schemes that researchers have used in feedback control. Partially homomorphic encryption (PHE) and fully homomorphic encryption (FHE). An HME scheme is considered partially homomorphic if the scheme supports only one homomorphic operation, either addition or multiplication [53]. If an HME scheme supports both operations, it is considered fully homomorphic.
PHE schemes are schemes that derive their security from difficulty of computing discrete logarithms. Encryption schemes use so called trapdoor functions. A trapdoor function can easily be applied to a value, however calculating the inverse is computationally hard. Trapdoor functions can be constructed such that it is easy to calculate the inverse given some key that is only available to authorised parties. This is traditionally how encryption schemes have functioned [18, 39, 41, 52]. Classically, encryption schemes have used modulus functions and products of primes to construct trapdoor functions. As such they all derive their security from difficulty of computing discrete logarithms. A relatively newer class of encryption schemes is that of lattice based encryption schemes. Lattice problems are linear algebra problems that can be used to construct encryption schemes that are fully homomorphic.
Currently the most prominent PHE schemes are the Rivest-Shamir-Adleman scheme known as RSA, the El Gamal scheme [20] and the Paillier scheme [38]. The RSA scheme features homomorphic addition and the El Gamal scheme features homomorphic multiplication. The Paillier scheme is partially homomorphic too, but it offers an extra operation that makes the scheme more practical than previous schemes. The Paillier scheme supports homomorphic addition and the multiplication of an encrypted value by an unencrypted value. Both RSA [41], the El Gamal scheme [20] and Paillier scheme [38] have been implemented in feedback control. The first implementation of homomorphic encryption in feedback control is that of Kiminao Kogiso in *Cyber-security enhancement of networked control systems using homomorphic encryption* [28]. The functionality of this control scheme was demonstrated with a simulation. The paper incorporates the El Gamal scheme, but in combination with the RSA scheme, because the El Gamal scheme's homomorphic addition is not enough to make a functional scheme. The extra homomorphic property of the Paillier scheme [38] allows for more streamlined designs. One such design is that from the paper *Implementing Homomorphic Encryption Based Secure Feedback Control* by Farhad Farokhi [49]. In this paper the authors stabilise an unstable plant in real-time. The authors are the first to have tested their system in practice. The scheme incorporates state feedback control and an discrete time observer which requires the multiplication of states and a state space matrices and gains. The Paillier supports homomorphic addition and the multiplication of an

encrypted value by some unencrypted value. Therefore authors have elected to keep the state space matrices and gain unencrypted so they can be used to update an observer of which the states, sensor data and control effort are encrypted.
There are two main ways of representing real numbers that are used in computation, fixed point- and floating point representation. Floating point numbers are by far the most popular representation due to its flexibility. Currently, some homomorphic encryption schemes can be used in combination with floating point numbers [34], however researchers commonly utilise fixed point representation due to its lower complexity. Section 2.6 will elaborate on Fixed Precision arithmetic. Homomorphic encryption imposes a limitation on fixed point representation. When multiplying two fixed point numbers, the decimal point of the representation shifts. Multiple multiplications would cause overflow. The exact cause will be explained in section 2.6.4. Normally truncation would prevent this, however most schemes do not allow for the truncation of encrypted numbers. Kiminao Kogiso [28] and Farhad Farokhi [49] each have a notable way of handling the issue of truncation. Kiminao Kogiso proposes to simply send states back to the plant, so they can be decrypted, truncated, re-encrypted and then sent back to the controller. Farhad Farokhi utilises a solution he has proposed in his earlier work [35] which introduces a periodic reset. The fractional bit representation of the observer states is allowed grow, until a certain point at which the observers are set to zero to prevent an overflow. This is functional, but expectedly, this does decrease performance. A newer solution is one that has been proposed by another recent paper by Junsoo Kim in *Dynamic controller that operates over homomorphically encrypted data for infinite time horizon* [26]. Junsoo Kim proposes the transformation of a control system such that it approximates the original system whist consisting of only integers, removing the need for fixed point numbers. This scheme too has an impact on performance, but much less than a periodic reset would. Authors Jung Hee Cheon and Damien Stehlé have also proposed their own FHE scheme that allows of the truncation of encrypted values, however this introduces noise to a degree that severely limits its practicality [10].
Besides the functionality FHE schemes offer, there is currently a consensus that lattice based schemes are likely to be quantum safe. Take for example a widely used lattice problem called learning with errors [40], LWE. In 2009 Odded Regev stated in his paper that one might conjecture that there is no quantum algorithm that can solve the LWE problem in polynomial time [40]. He bases this on the fact that up until the release of his paper, there had not yet been any quantum algorithms presented that could solve the LWE problem in polynomial time. This is despite the fact that the existence or non-existence of such an algorithm could mean lattice based encryption is the answer to quantum computers. Given its importance it would seem likely an algorithm would be put forward if such an algorithm could be constructed. As of yet, this still holds true in 2022 lending credence to Odded Regev's claim. This makes lattice based encyption attractive as compared to schemes such a RSA and other schemes that derive their security from difficulty of computing discrete logarithms. The RSA scheme can be cracked efficiently utilising a quantum algorithm [23]. Currently, PHE shemes utilise mechanisms similar to RSA, meaning they are likely to be insecure, or have already been confirmed to be insecure with respect to quantum computation.
There are two main downsides to FHE schemes. The first issue is that of multiplicative depth. To make FHE schemes secure, errors are injected into ciphers. This does not form a problem, unless homomorphic multiplication is applied. When multiplying two ciphers, the error grows. If this error grows too much, the message can no longer be retrieved correctly. The amount of homomorphic multiplications that can be performed without the message becoming irretrievable is called multiplicative depth. So far, Gentry's encryption scheme has improved multiplicative depth, however it remains an issue.
Secondly, FHE schemes are very computationally complex. There are currently no industrial or commercial applications that utilise FHE schemes. Currently there are simulations of FHE schemes [26] and the practical application of FHE on control systems that run at a slow update rate such as demonstrated in *Toward a secure drone system: Flying with real-time homomorphic authenticated encryption* [9] in which the authors provide a path for a drone to follow, which is updated at a rate of 10Hz.Though there are papers that aim to speed up the computations needed for FHE schemes such as Gentry's scheme [22], there is currently no research that implements the application of FHE in feedback control for the stabilisation of an unstable plant.

1.2. Research motivation

There is a clear lack in advancement in implementation of FHE in feedback control. One reason being the fact that lattice based encryption schemes are computationally expensive. Most controllers are implemented on conventional hardware, usually micro controllers equipped with a modern CPU. If operations can be performed in parallel, it may be advantageous to equip the target platform with a GPU. GPU's are set up to

perform many calculations in parallel. To use a GPU, a CPU must supply the GPU with instructions and data necessary for computation. The communication between CPU and GPU is commonly a bottleneck. For the purposes of implementing FHE in feedback control, an ad hoc harware design would be preferable, as FHE operations can be performed largely in parallel, but it also requires large data throughput, which would be bottle-necked using a CPU and GPU. Hence, implementation on an FPGA would be a good avenue of research. The acronym FPGA stands for Field Programmable Gate Array, which is a computer chip with logic gates and other components that can be connected to form circuits.
Given these factors and the need for advancement in the area of FHE in feedback control, the topic of this thesis will be: *An FPGA implementation of a fully-homomorphic encryption scheme for real-time control applications*.
Currently, the FHE scheme by Gentry [22] is still one the most versatile and robust FHE schemes available. All lattice based encryption schemes allow for homomorphic multiplication, but so far all of them have limited multiplicative depth. There are multiple schemes that have introduced methods to decrease this noise, but currently Gentry's scheme offers the least computationally expensive and simple way to decrease this noise. Another notable advancement is a new FHE scheme proposed by Jung Hee Cheon in *Homomorphic encryption for arithmetic of approximate numbers* [11]. This scheme is however designed for the purpose of secure machine learning, which is not the topic of this thesis. Therefore Gentry's FHE scheme will be the encryption scheme used to implement feedback control.
The optimal outcome of the research from this thesis would be to present an implementation and workflow that would allow for the implementation of a wide range of feedback controllers secured with FHE. Therefore the feedback controller that will be implemented for the purpose of the research will be discrete time state feedback control, which is a common controller topology that utilises memory which is a requirement for more complex feedback control. The plant that will be controlled for the purposes of demonstration will be a simulation of a double pendulum in an upright position. The upright position of such a mechanism is an unstable equilibrium and will require a sufficiently high sampling frequency to stabilise the plant around this equilibrium. Such a control scenario is a good use-case as it requires either the optimisation of the encryption scheme or the a computationally efficient hardware design, preferably both.

1.3. Research questions

The goal of this thesis is to adapt the Gentry scheme in tandem with an FPGA implementation to optimally perform the required operations. Due to time constraints, the functionality of the feedback system will be confirmed through a simulation of the hardware design of the system, of which the code will be provided in appendix section B. To demonstrate viability of the system, hardware designs of critical subsystems will be compiled and run in practice. The code of this demo can be found in the appendix section C.
To evaluate the performance of the resulting implementation, the following research question and sub-question will be answered: *given a novel adaptation and implementation of homomorphic feedback control, what level of performance can currently be reached?*

- *What are the challenges of using fully homomorphic encryption in feedback control?*
- *How can the underlying mathematics of Gentry's encryption scheme be rewritten to be suitable for digital computation?*
- *What is the hardware utilisation of an implementation of feedback control using Gentry's encryption scheme on an FPGA?*

1.4. Contributions

Gentry's encryption scheme relies on a set of black box functions. In this thesis I propose replacing these functions with analytical equivalents. This makes the scheme more intuitive and allows for easier manipulation of the constituent parts. I also propose the so called reduced cipher. Rewriting the scheme using the new notation and reduced cipher leads to more efficient computation of Homomorphic operations and reduced memory usage. Using the improved scheme I have designed an FPGA implementation of feedback control using state feedback control and Fully Homomorphic Encyrption. The improvements to Gentry's encryption scheme ensure that the sampling frequency can be high enough to allow for the stabilisation of an unstable plant which previous implementations of FHE in feedback control were unsuitable for. Fixed precision representation is the most performant numbering system that is compatible with FHE. Using fixed precision

representation representation and FHE poses two problems, decimal point shifts and limited multiplicative depth. Both problems are handled using the solution presented in [28], by sending states back from the controller to the plant. The states can then be truncated and re-encrypted. Preventing both overflow and resetting multiplicative depth every time step. The combination of solutions allows for the implementation of discrete-time feedback control with out requiring alteration to be compatible with FHE.

1.5. Structure

Finally, this section will conclude the introduction by giving an overview of the order of contents of this thesis. First chapter 3 will elaborate on the types of encryption mentioned in the introduction and will describe the Gentry scheme [22] in detail. Chapter 3 will introduce new notation and adaptations made to the Gentry scheme [22] as well as implications on performance. Chapter 4 will describe the controller topology in more detail followed by explaining how FPGA's function and how they can be deployed. This is followed by an overview of the implementation of the previously described feedback controller with FHE on an FPGA. Chapter 5 will describe the research setup followed by the results. Chapter 6 concludes the thesis and will give recommendations for further research.

2

Introduction to homomorphic encryption

To explain one of the main the contribution of thesis, it is important to cover the background of encryption itself as well as the encryption scheme used for implementation. Section 2.1 will introduce and explain the seminal Diffie-Hellman encryption scheme. Next section 2.2 discusses two categories that most encryption schemes fall into. Section 2.3 covers lattice problems and homomorphic encryption in more detail. Section 2.4 describes the LWE problem [40]. Section 2.5 discusses the Gentry encryption scheme, which is the encryption scheme that will be adapted and implemented for this thesis. Finally section 2.6 introduces the fixed point numbering system that can represent fractions and negative numbers, which is compatible with HME.

2.1. Diffie-Hellman: The introduction of modern cryptography

Cryptography has been a topic of research for thousands of years if not more. The earliest recorded use cryptography is by Egyptians 4000 years ago [39]. Encryption schemes from the past generally worked by changing the meaning of letters, by changing the meaning of letters used or by changing letters with other symbols. These methods were inventive, but flawed. The meaning of letters still leaves a pattern of words with fixed lengths and there are only so many ways letters can be swapped around, meaning cracking such a code by hand is still feasible. The advent of computer communication in the modern day changed this entirely however.
When using computers, transmission of information is many orders of magnitudes faster than sending a message on paper or via telephone.
This means that encryption can be more complex and secure, whilst still appearing instantaneous to its users. Computers work with binary code and as such a message is first converted to binary according to a preconceived standard and then encoded. All semblance of "human" structure (i.e. words and sentences) becomes unrecognisable. This makes it impossible to crack such encryption schemes by hand.
The first practical encryption scheme to do this was posed in "New Directions in Cryptography" and is called a Diffe-Hellman key exchange [18]. This work would become incredibly influential in the world of computer science, marking the beginning of modern cryptography. Almost any work in the field references this work and takes inspiration from it. Diffe-Hellman utilises the fact that given the right mathematical obfuscation, it would take a rogue agent (a party not privy to the message) with a conventional computer an infeasible amount of time to decode the message. Al the while, the two parties that have exchanged keys can decode the messages very easily. As there is no semblance of structure in the messages sent, there is no clever way to decode the messages and thus a rogue agent would be forced to decode through brute force. Hence it is possible to quantify the security of the method as the amount of mathematical operations needed to decode a message through brute force.

To explain the Diffe-Hellman key exchange, the concept of private keys and a public key must be introduced. The private key is a set of two numbers and is publicly agreed upon. Each party has their own private key, that they generate themselves and keep secret. The actual key exchange can be divided into 3 steps. Figure 2.1.1a shows a diagram of two computers, A and B, that wish to communicate through a public network, but computer C is secretly listening in and would like to know what A and B are talking about.

- Figure 2.1.1b: Computer A and B agree on a value g and p. p is a random prime number and g some integer smaller than p.
- Figure 2.1.1c: Computer A and B generate private keys a and b respectively. a and b are chosen at random, but smaller than p. Computer A sends $(g^a \bmod p)$ to computer B and computer B sends $(g^b \bmod p)$ to computer A.
- Figure 2.1.1d: Now both computer A and B can calculate $g^{a \cdot b} \bmod p$, using their private keys. They can do this due to the following mathematical property: $(g^b \bmod p)^a = (g^a \bmod p)^b = g^{a \cdot b} \bmod p$. Now computer A can send a message m using the shared key $g^{a \cdot b} \bmod p$ to encode the message. $\tilde{m}$ denotes the encoded message. Computer B has the shared key and can reverse the encoding. Computer C however, does not know what the a or b is and cannot generate $g^{a \cdot b} \bmod p$ and thus cannot read the message.

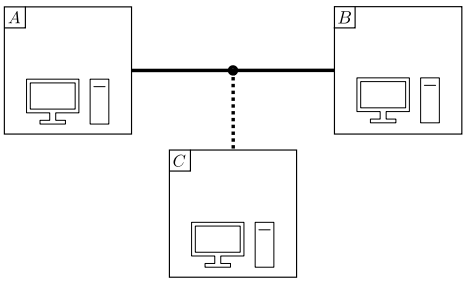
The encoding process is usually very simple. Take for example XOR-encoding. If the shared key $g^{a \cdot b} \bmod p$ would be the 4-bit long code $s = \{1,0,1,0\}$ in binary and the message to be encoded $\{1,0,1,1,0,0,1,0\}$, then the encoded message would be the result of the **XOR** (exclusive or) operation on each section of 4 bits of the message and the key as follows:

m is split up into $m = \{m_1, m_2\}$, each 4-bits long

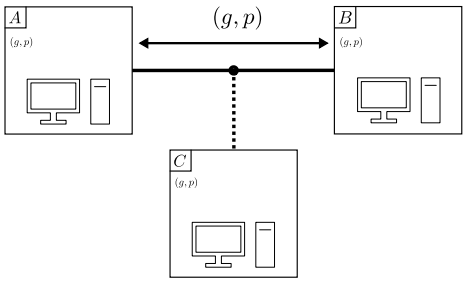
$$
\begin{aligned}
m_1 &= \{1,0,1,1\} \\
m_2 &= \{0,0,1,0\} \\
\tilde{m}_1 &= \mathbf{XOR}(m_1, s) = \mathbf{XOR}(\{1,0,1,1\},\{1,0,1,0\}) = \{0,0,0,1\} \\
\tilde{m}_2 &= \mathbf{XOR}(m_2, s) = \mathbf{XOR}(\{0,0,1,0\},\{1,0,1,0\}) = \{1,0,0,0\} \\
\tilde{m} &= \{m_1, m_2\} = \{0,0,0,1,1,0,0,0\}
\end{aligned}
$$

This can be reversed with the same process:

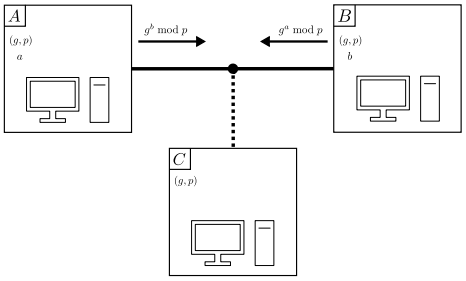
$$m = \{\mathbf{XOR}(\tilde{m}_1, s), \mathbf{XOR}(\tilde{m}_2, s)\}$$



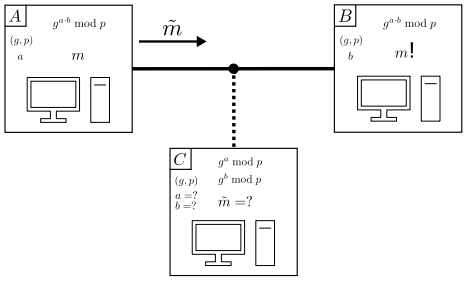
(a) Computer A and B want to communicate securely, computer C is able to intercept all the data A and B send to one another.



(b) Computer A and B agree on values for g and p



(c) Computer A and B can now send each other their private key mixed with the public key.



(d) Now computer A sends B a message encoded with the shared key. Computer C has all the information that A sends B have shared, but not their private keys and is unable to read the message.

Figure 2.1.1: This series of diagrams is a visualisation how the Diffie-Hellman key exchange is performed and subsequently used to send a message

Unfortunately the Diffe-Helman scheme is flawed. The scheme is sensitive to so called man in the middle attacks. The algorithm does not specify that any authentication needs to take place. Computer C could act like they are computer A or B and gather more information on the private keys by intercepting the key exchange and mixing in their own private key. There are ways to protect against these attacks by implementing authentication. Luckily there are other trapdoor function based schemes that use the same concept, but different trapdoor function. These alternatives are not at risk of a man in the middle attack. One of the most important (if not the most important) variation is so called Elliptic-curve cryptography. It is a kind of encryption that utilises a series of function evaluations on the curve $y^2 = x^3 + ax + b$ with a special rule of succession to generate keys [54].
These schemes are much more efficient, in the sense that it takes smaller keys than other schemes to reach the same security (i.e. it is much harder to guess private keys from the information that needs to be shared). This means that information can be shared at the same level of security, whilst encoding takes less time (due to the smaller key size). 384-bit Eliptic curve cryptography is actually the cryptography the American National Security Agency, NSA, has used for transmitting top secret information [36] (however, they are likely going to switch to a cryptographic scheme that is also safe against quantum computing [37]).

2.2. Symmetric-key and public-key encryption

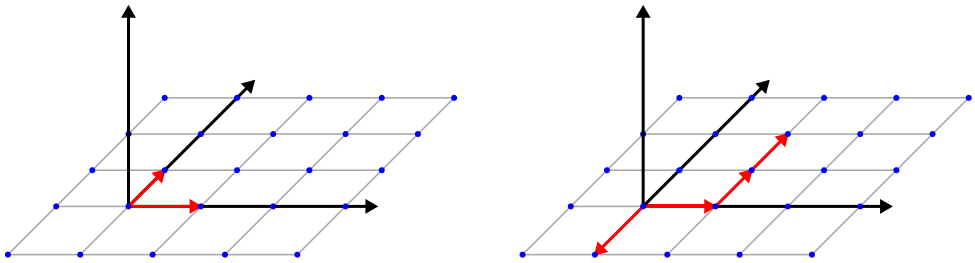
Diffie-Hellman requires both sender and recipient to generate a private key and share it with one another, this means that Diffie-Hellman is called **symmetric-key encryption**. In the case of symmetric-key-encryption, the public key is used to safely generate a shared key. There are schemes very similar to Diffie-Hellman, but where the public and private keys play a different role. Schemes that do not cooperation from other computers to generate keys.
These schemes are called **public-key encryption** schemes, which allow for encryption with only the public key. This means anyone can send an encrypted message to someone who makes their public key available. This decreases the amount of data that needs to be shared to share messages and eliminates the threat of man in the middle attacks. One of the most (probably the most) influential public key encryption schemes is the RSA [41]. RSA is mentioned in many papers regarding cryptography ([38, 49, 53, 54] to name a few). Currently, public-key encryption schemes rely on a mechanism where a private key is generated after which the public key is generated from the private key through a trapdoor function. This means that a public-key encryption scheme could be cracked by recovering the private key from the public key. Most public key encryption schemes that derive their security from difficulty of computing discrete logarithms over finite fields rely on the same mechanism as that of the RSA scheme. Generating a public-private key pair according to the RSA scheme requires the selection of two large prime numbers, which together form the private key. The result of the multiplication of these two prime numbers is then the public key. The public key is a factor of two primes. The process of recovering the two primes from such a number is called prime factorisation. If the primes are large enough, this process would take years if not decades to compute on conventional computers. According to the authors of *Comparing the Difficulty of Factorization and Discrete Logarithm: a 240-digit Experiment* [3], it would take a 1000 core-years worth of computation to crack 240-bit RSA which is a common key size. It would take an computer with an 8-core cpu (the same as used in [3]) roughly 250 years to crack the scheme.
One of the concerns with classical encryption schemes is that the introduction of quantum computers might pose a significant security threat. This is with good reason as in 1994 P.W. Shor [43] proposed a quantum algorithm that could be used to efficiently crack RSA and in 2019 Craig Gidney and Martin Ekerå presented their implementation of the algorithm [23]. Having run simulations of quantum computing, they estimate that it would take 8 hours to break 2048-bit RSA (2048 bit refers to the key size and is a widely used standard). The work is based on current knowledge of quantum computers with the assumption that it would be possible to construct a system consisting of 20 million qubits. The issue with that assumption is that one of the major issues with quantum computers is noise. The noise scales with the amount of qubits in a system. In the paper they simulate noise and implement appropriate error correction. Unfortunately, the largest quantum computer that is available right now only contains 65 qubits. It is not known yet if it is possible to construct a quantum computer with millions of qubits and so many are still sceptical of whether quantum computing is actually viable. IBM does promise to produce a quantum computer with 1000-qubits by the year 2023 [12].

2.3. Homomorphic encryption

After the introduction of the RSA scheme, researchers found that the scheme had a so called homomorphic property. When data is encrypted, this is called a cipher. For ease of legibility let $C = \mathbf{E}(\alpha)$ be the encryption of some unsigned integer α to produce a cipher C. Given two values α and β, encrypted using RSA, it is possible to use the two ciphers to generate a third which is the encrypted value of the multiplication the original two values:

$$\begin{aligned} &\alpha, \beta \in \mathbb{Z} \\ &\mathbf{E}(\alpha) \odot \mathbf{E}(\beta) = \mathbf{E}(\alpha \cdot \beta) \end{aligned} \tag{2.3.1}$$

$\odot$ denotes homomorphic multiplication. These operations can be performed without decryption at any stage. This is called a homomorphic property, an operation on ciphers which is the equivalence of some mathematical operation on the underlying numbers. In the case of the RSA scheme, this was a flaw, as it was an unintended property which might pose a security risk [2]. However researchers realised that homomorphisms could be useful for applications such as electronic vote counting [1] and dubbed in homomorphic encryption. Homomorphism refers to the homomorphic operations that such a scheme allows for. The two most important operations are homomorphic addition and multiplication. The homomorphic operations are operations that hold for encrypted values. As discussed earlier, homomorphic encryption is an incredibly useful type of encryption to feedback control, as it allows for the encryption of all the sensor data and control signals and its all around holistic approach to ensuring security. As discussed earlier in the introduction (section 1.1), besides using discrete logarithms problems as trapdoor functions (such as demonstrated in section 2.1), lattice problems too, can be used to construct fully homomorphic encryption schemes. A lattice is defined as follows: given a basis $\mathscr{B} \in \mathbb{R}^{n \times n}$ for some dimension $n \in \mathbb{N}$, a lattice constructed with $\mathscr{B}$ yields $\mathscr{L}(\mathscr{B}) = \mathscr{B}\mathbb{Z}^n$. In other words, a lattice is a type of grid (see figure 2.3.1). A lattice is the collection of all integer linear combinations of a given basis (see figure 2.3.1b)

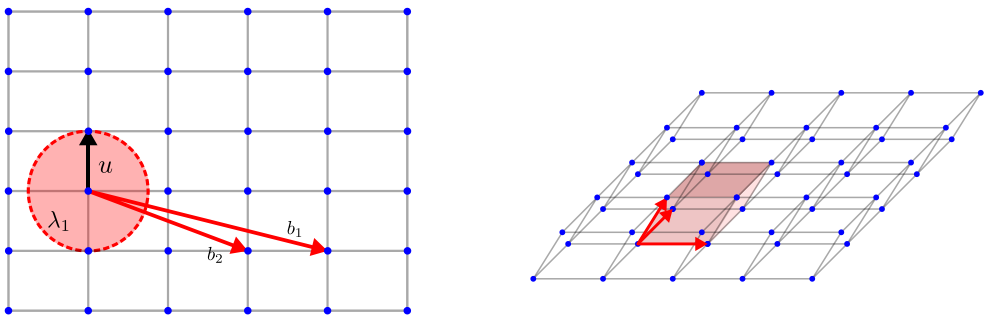


(a) Basis in red

(b) The set of all integer multiples of the basis vectors form a lattice

Figure 2.3.1: Visual representation of a lattice in 2D

With such a lattice, it is possible to encrypt a number by constructing a lattice problem, the solution of which is the value that one wants to encrypt. Take for instance the lattice problem that Miklós Ajtai has used in his paper, the shortest vector problem, SVP [51].
Given a basis $\mathscr{B} \in \mathbb{Z}^{n \times n}$, find $u \in \mathscr{L}(\mathscr{B}) \setminus \{0\}$ s.t. $||u|| = \lambda_1$. The goal is to find the find the smallest integer linear combination of a given bases. Or in other words, what is the lattice point that is closest to the origin (not including lattice point in the origin itself), find these lattice for every dimension of the lattice. Figure 2.3.2a shows a graphical representation of the problem. It is possible to find the graphically by drawing the lattice. It is however tricky to draw the lattice, because it requires drawing a random selection of lattice points, because the basis vectors are quite a bit longer than the lattice point that is closest. Although there are many situations with trivial solutions, when a basis is generated randomly it leads to cases that are non trivial. There is currently no efficient way of solving SVP problems efficiently



(a) Example of the SVP problem

(b) Example of a 3D-Lattice

Figure 2.3.2

Of course a two dimensional version of a lattice problem would be too easy to solve for a computer, since it could find a solution through brute force. Luckily all lattice problems can be extended to any dimension, generating higher dimensional lattice problems leads to better security (see figure 2.3.2b). Some encryption schemes would be proposed that used lattice problems, however they did not catch on. In 2009 however, Craig Gentry proposed an encryption scheme in his thesis [21] and subsequent paper *Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based* [22], which inspired a wide range of works and implementations [7, 10, 25–27, 35].

2.4. Learning with errors

The Learning With Errors problem is a generalisation of another lattice based problem, the Learning From Parity With Error problem, LPWE[40]. LPWE is defined as follows:

To generate an LPWE problem, randomly sample a set of vectors $a_1, a_2, \cdots \in \mathbb{Z}_2^n$. $\mathbb{Z}_2^n$ denotes a vector of dimension n, with modulus 2 (i.e. all entries are either 0 or 1). Next, randomly generate a vector $s \in \mathbb{Z}_2^n$. Let $\langle \cdot, \cdot \rangle$ be the inner product of two vectors. Set values b_i are set to equal $\langle s, a_i \rangle$ with probability $1-\epsilon$. In other words, the chance of $\langle s, a_i \rangle$ being equal to $b_i \bmod 2$ is $1-\epsilon$. Now discard s. The goal is to find a vector such that:

$$\begin{aligned} \langle s, a_1 \rangle &\approx_\epsilon b_1 \,(\mathrm{mod}\, 2) \\ \langle s, a_2 \rangle &\approx_\epsilon b_2 \,(\mathrm{mod}\, 2) \\ \langle s, a_3 \rangle &\approx_\epsilon b_3 \,(\mathrm{mod}\, 2) \\ &\vdots \\ \langle s, a_n \rangle &\approx_\epsilon b_n \,(\mathrm{mod}\, 2) \end{aligned} \tag{2.4.1}$$

Learning With Errors, LWE, is a natural extension of LPWE where $a_i, b_i \in \mathbb{Z}_q^n$, $s \in \mathbb{Z}_q^n$ and $q \in \mathbb{N}$. In other words, all vectors can contain integer values up to and including $q-1$. find a vector s such that [40]:

$$\begin{aligned} \langle s, a_1 \rangle &\approx_\chi b_1 \,(\mathrm{mod}\, q) \\ \langle s, a_2 \rangle &\approx_\chi b_2 \,(\mathrm{mod}\, q) \\ \langle s, a_3 \rangle &\approx_\chi b_3 \,(\mathrm{mod}\, q) \\ &\vdots \\ \langle s, a_n \rangle &\approx_\chi b_n \,(\mathrm{mod}\, q) \end{aligned} \tag{2.4.2}$$

where values $b_i = \langle s, a_i \rangle + e_i$ and e_1 is sampled from the χ distribution. The χ distribution is defined as $\chi : \mathbb{Z}_q \rightarrow \mathbb{R}^+$, the mapping of the Ψ_α distribution onto $\mathbb{Z}_q^n$. α and Ψ_α are defined as:

$$\begin{aligned} &\alpha \in (0,1) \text{s.t.} \alpha \leq \sqrt{\log n} \\ &\forall r [0,1), \Psi_\alpha(r) = \sum_{k=-\infty}^{\infty} \frac{1}{\alpha} \cdot \exp\left(-\pi \left(\frac{r-k}{\alpha}\right)^2\right) \end{aligned} \tag{2.4.3}$$

There are many ways of generating α. The way the author of [40] calculates α is through:

$$\alpha = \frac{1}{\sqrt{n} \cdot \ln n^2} \tag{2.4.4}$$

Next, Ψ_α is mapped to $\mathbb{Z}_p$ as:

$$\chi(i) = \int_{(i-\frac{1}{2})/p}^{(i+\frac{1}{2})/p} \Psi_\alpha(r)dr \tag{2.4.5}$$

To sample the χ distribution, one can sample a normal distribution $x \leftarrow \mathcal{N}(0, \frac{\alpha}{\sqrt{2\pi}})$, reducing the result mod 1, multiply it by modulo q and finally round the result (the derivation can be found in [40]). The mathematical discription of the sampling procedure is as follows:

$$\begin{aligned} x &\leftarrow \mathcal{N}(0, \frac{\alpha}{\sqrt{2\pi}}) \\ e &= \lceil q \cdot (x \bmod 1) \rfloor \end{aligned} \tag{2.4.6}$$

With equality $b_i = \langle s, a_i \rangle + e_i$, there is now a chance of $\chi(0)$ of $b_i = \langle s, a_i \rangle$ being correct.

2.5. Gentry's FHE scheme

This section will formally introduce Gentry's encryption scheme [22]. Section 2.5.1 introduces functions that are necessary to construct the scheme, followed by section 2.5.2 and 2.5.3 detailing how scheme functions. Finally section 2.5.4 discusses the homomorphic properties and details how the error injection inherent to the scheme limits multiplicative depth.

2.5.1. Gentry functions

Gentry's encryption scheme [22] requires the manipulation of individual bits that make up integers. This manipulation only applies to positive integers and will generally regard numbers contained in some multiplicative group. For some number $a \in \mathbf{Z}_q$ for some $q \in \mathbf{N}$, then $a^{[i]}$ denotes the i-th bit in the binary representation of number a. The convention that will be adhered to in this thesis is to let the 0th bit be the least significant bit. ℓ denotes the maximum amount of bits that are represented. Using this, in his paper [22], Gentry introduces four functions that are used to describe and prove the functionality of his Encryption scheme. The functions are defined as follows:

$$\begin{aligned} &a \in \mathbf{Z}_q^n, \ a = [a_0, a_1, \cdots, a_{n-1}] \\ &\textbf{BitDecomp}(a) &&= \left[a_0^{[0]}, a_0^{[1]}, \ldots, a_0^{[\ell-1]}, a_1^{[0]}, a_1^{[1]}, \ldots, a_1^{[\ell-1]}, \ldots, a_{n-1}^{[0]}, a_{n-1}^{[1]}, \ldots, a_{n-1}^{[\ell-1]}\right] = b \\ &\textbf{BitDecomp}^{-1}(b) &&= \left[a_0^{[0]} + 2 \cdot a_0^{[1]} + \cdots + 2^{\ell-1} \cdot a_0^{[\ell-1]}, \ldots, a_{n-1}^{[0]} + 2 \cdot a_{n-1}^{[1]} + \cdots + 2^{\ell-1} \cdot a_{n-1}^{[\ell-1]}\right] = a \\ &\textbf{Flatten}(b) &&= \textbf{BitDecomp}(\textbf{BitDecomp}^{-1}(b)) \\ &\textbf{Powersof2}^{-1}(a) &&= \left[a_0, 2 \cdot a_0, \ldots, 2^{\ell-1} \cdot a_0, \ldots, a_{n-1}, 2 \cdot a_{n-1}, \ldots, 2^{n-1} \cdot a_{n-1}\right] \end{aligned} \tag{2.5.1}$$

*Note: given $a =$ **BitDecomp**$^{-1}$(**BitDecomp**$(a))$ functions as the identity if the input a is smaller than q, however the opposite order is not the identity $a \neq$ **BitDecomp**(**BitDecomp**$^{-1}(a)$).*

Every function also applies to matrices, where the operations as described in (2.5.1) are performed per row. In Gentry's encryption scheme [22] plain text messages are stored as the solution to LWE problems. These LWE problems are represented by matrices. The functions mentioned above are utilised to construct the encryption scheme. Gentry points out the following properties of the previously introduced functions:

$$\langle \textbf{BitDecomp}(a), \textbf{Powersof2}(b) \rangle = \langle a, b \rangle \tag{2.5.2}$$

$$\langle a', \textbf{Powersof2}(b) \rangle = \langle \textbf{BitDecomp}^{-1}(a'), b \rangle = \langle \textbf{Flatten}(a'), \textbf{Powersof2}(b) \rangle \tag{2.5.3}$$

2.5.2. Encryption

In Gentry's encryption scheme, the LWE problem is formulated as a matrix vector problem. Vector s from equation 2.4.2 is renamed to t and vectors a_i from equation 2.4.2 are stored as a matrix that represents a

basis B (a_i form the span of B):

$$\begin{aligned} &t \in \mathbf{Z}^n, B \in \mathbf{Z}^{m \times n}, e \in \chi_q^m \\ &b = B \cdot t + e \end{aligned} \tag{2.5.4}$$

m and n positive integers that represent dimensions which influence security. The larger m and n are, the more equations have to be solved to find s. The public key is constructed from vector b and matrix B as:

$$A = [b, B] \tag{2.5.5}$$

It is possible to recover error vector e through:

$$A \cdot [1, -t^\top]^\top = [b, B] \cdot [1, -t^\top]^\top = b - B \cdot t = B \cdot t + e - B \cdot t = e \tag{2.5.6}$$

This property is what Gentry uses to encrypt messages. Take some value for n and set $m = n+1$, let C be:

$$C = I_{n+1} \cdot \mu + A \tag{2.5.7}$$

where μ is some number we would like to encrypt and I_{n+1} is an identity matrix of size $(n+1) \times (n+1)$. Matrix C has the following property, let $s = [1, -t^\top]^\top$, then:

$$C \cdot s = (I_{n+1} \cdot \mu + A) \cdot s = s \cdot \mu + A \cdot [1, -t^\top]^\top = s \cdot \mu + e \tag{2.5.8}$$

This is relationship is almost suitable as an encryption scheme, because e can be chosen such that it is large enough to make it computationally hard to recover secret key t from A, but small enough such that μ can be recovered from C. The actual way that numbers are encrypted in Gentry's scheme is as follows:

$$C = \textbf{Flatten}(I_N \cdot \mu + \textbf{BitDecomp}(R \cdot A)) \tag{2.5.9}$$

Randomly sample a uniform matrix $R \in \{0,1\}^{N \times m}$ to ensure that each message is unique, otherwise anyone who knows the public key could decrypt encrypted message. The Cipher becomes a matrix of size $N \times N$, where $N = (n+1) \cdot \ell$. Gentry has introduced the **Flattening** function to constrain the size of ciphers, which is more efficient than the method used in previous work called relinearisation [5]. Using the relationships from equations 2.5.2 and 2.5.3, the following demonstrates that message μ can still be recovered:

$$\begin{aligned} &C \cdot \textbf{Powersof2}(s) = \textbf{Flatten}(I_N \cdot \mu + \textbf{BitDecomp}(R \cdot A)) \cdot \textbf{Powersof2}(s) = \\ &I_N \cdot \mu \cdot \textbf{Powersof2}(s) + \textbf{BitDecomp}(R \cdot A) \cdot \textbf{Powersof2}(s) = \\ &I_N \cdot \mu \cdot \textbf{Powersof2}(s) + R \cdot A \cdot s = \\ &I_N \cdot \mu \cdot \textbf{Powersof2}(s) + R \cdot e \end{aligned} \tag{2.5.10}$$

2.5.3. Scheme summary

To summarise the Gentry encryption scheme [22], the encryption scheme can be divided into three algorithms. An algorithm for key generation, encryption and decryption.

Algorithm 1 Public and private key generation for the Gentry scheme

1: **procedure** KeyGen
2: draw a random column vector $t \in \mathbb{Z}_q^n$
3: calculate $s = \left[1, -t^\top\right]^\top$
4: calculate $v = \textbf{Powersof2}(s)$ ▷ Vector v is the private key
5: draw a random matrix $B \in \mathbb{Z}_q^{m \times n}$
6: draw a row vector e of m entries from the χ distribution
7: calculate $b = B \cdot t + e$
8: calculate $A = [b,\ B]$ ▷ Matrix A is the public key
9: **return** A, v
10: **end procedure**

Algorithm 2 Message encryption with the Gentry scheme

procedure Enc(μ, A)
  draw a random matrix $R \in \{0,1\}^{N\times m}$
  calculate cipher matrix $C = \textbf{Flatten}(\mu \cdot I_N + \textbf{BitDecomp}(R \cdot A))$
  discard R
  return C
end procedure

Algorithm 3 Cipher decryption Gentry scheme

procedure MPDec(C, A)
  calculate $\eta = C \cdot v$ ▷ Note that according to equation 2.5.10, $\eta = C \cdot v = \mu \cdot v + R \cdot e$
  set $\bar{\mu} = 0$
  for $i \leftarrow 0$ to $\ell - 1$ **do**
    $\texttt{check} = \eta_{\ell-i-1} - \texttt{shift\_left}(\mu, \ell - i - 1)$
    if $\texttt{check}^{\ell-1} \wedge \texttt{check}^{\ell-2} = 1$ **then**
      $\bar{\mu}^i \leftarrow 1$
    end if
  end for
  set $\mu \leftarrow \bar{\mu}$
  return μ
end procedure

Algorithm 3 is called **MPDec** as it was proposed by Micciancio and Peikert [32], which works for messages $\mu \in \mathbf{Z}_q$. Previous algorithms only allowed for the recovery of messages the size of a single bit.

2.5.4. Homomorphic properties

Gentry has pointed out 4 homomorphic properties. The homomorphic properties relevant to this thesis are the following three:

- **MultConst**(C, α)
  Given a cipher $C = \textbf{Enc}(\mu)$ and a number α, where $\mu, \alpha \in \mathbb{Z}_q$, then

$$\textbf{MPDec}(\textbf{MultConst}(C, \alpha)) = \mu \cdot \alpha + \cancelto{0}{R \cdot e} \tag{2.5.11}$$

  To perform the **MultConst**(C, α) operation, set $M_\alpha = \textbf{Flatten}(I \cdot \alpha)$, then **MultConst**$(C, \alpha) = \textbf{Flatten}(M_\alpha \cdot C)$. This relationship can be confirmed as follows:

$$\textbf{MultConst}(C, \alpha) = \textbf{Flatten}(M_\alpha \cdot C) = \textbf{Flatten}(\textbf{Flatten}(I \cdot \alpha) \cdot \textbf{Flatten}(I \cdot \mu + \textbf{BitDecomp}(R \cdot A))) =$$
$$\textbf{Flatten}(I \cdot \mu \cdot \alpha + M_\alpha \cdot \textbf{BitDecomp}(R \cdot A)) \cdot v = \alpha \cdot \mu \cdot v + M_\alpha \cdot R \cdot e$$

  Note that the error increases by at most $N \cdot m$. Both matrices M_α and R contain only binary elements and $M_\alpha \cdot R \in \mathbb{Z}_N^{(n+1)\times m}$, therefore e is at most scaled by N times by m elements.

- **Add**(C_1, C_2)
  Given two ciphers C_1, C_2 corresponding to the encryption of μ_1 and μ_2 respectively where $\mu_1, \mu_2 \in \mathbb{Z}_q$, if **Add**$(C_1, C_2) = \textbf{Flatten}(C_1 + C_2)$, then **MPDec**(**Add**$(C_1, C_2)) = \mu_1 + \mu_2$. Gentry states that this relationship is obvious, but for the sake of completeness, the following proves the relationship:

$$\textbf{Add}(C_1, C_2) = \textbf{Flatten}(C_1 + C_2) =$$
$$\textbf{Flatten}(\textbf{Flatten}(I \cdot \mu_1 + \textbf{BitDecomp}(R_1 \cdot A)) + \textbf{Flatten}(I \cdot \mu_2 + \textbf{BitDecomp}(R_2 \cdot A))) =$$
$$\textbf{Flatten}(I \cdot (\mu_1 + \mu_2) + \textbf{BitDecomp}(R_1 \cdot A) + \textbf{BitDecomp}(R_2 \cdot A)) =$$
$$\textbf{Flatten}(I \cdot (\mu_1 + \mu_2) + \textbf{BitDecomp}(R_3 \cdot A))$$
$$\textbf{Flatten}(I \cdot (\mu_1 + \mu_2) + \textbf{BitDecomp}(R_3 \cdot A)) \cdot v = (\mu_1 + \mu_2) \cdot v + R_3 \cdot e$$

$R_3 \in \mathbb{Z}_2^{N\times m}$ is equivalent to the addition of matrices R_1 and R_2 after flattening. When performing homomorphic addition, the error does not grow.

- **Mult**(C_1, C_2)
  Given two ciphers C_1, C_2 corresponding to the encryption of μ_1 and μ_2 respectively where $\mu_1, \mu_2 \in \mathbb{Z}_q$, take **Mult**$(C_1, C_2)$ = **Flatten**$(C_1 \cdot C_2)$, then **MPDec**(**Mult**(C_1, C_2)) = $\mu_1 \cdot \mu_2$. This relationship can be proven through:

$$\begin{aligned}&\textbf{Mult}(C_1, C_2) \cdot v = \textbf{Flatten}(C_1 \cdot C_2) \cdot v = C_1 \cdot C_2 \cdot v = C_1 \cdot (\mu_2 \cdot v + R_2 \cdot e) = \\ &C_1 \cdot \mu_2 \cdot v + C_1 \cdot R_2 \cdot e = \mu_2 \cdot (C_1 \cdot v) + C_1 \cdot R_2 \cdot e = \mu_2 \cdot (\mu_1 \cdot v + R_1 \cdot e) + C_1 \cdot R_2 \cdot e = \\ &\mu_1 \cdot \mu_2 \cdot v + \mu_2 \cdot R_1 \cdot e + C_1 \cdot R_2 \cdot e\end{aligned}$$

  Though matrix multiplication is not commutative, homomorphic multiplication is commutative with respect to decryption ($C_1 \cdot C_2 \neq C_2 \cdot C_1$, but **MPDec**(**Mult**$(C_1, C_2)$) = **Mult**$(C_2, C_1)$ if the error vector remains small). Observe that the error of this operation scales with both N as well as the size of μ_2 (or μ_1 depending on the order of the matrix multiplication of C_1 and C_2). This means that with successive multiplications, the error will grow exponentially. This property is defined as multiplicative depth which limits the scheme and requires a solution or workaround. The solution to address multiplicative depth selected in this thesis will be discussed in section 4.2.

2.5.5. Security

Gentry's encryption scheme derives it security from the computational complexity of the LWE problem. Hence security of the Gentry scheme follows from an implicit lemma from the paper by Odded Regev which introduces the LWE problem *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography* [40]:

Lemma 1. *Let params* $= (n, q, \chi, m)$ *be such that the* $LWE_{n,q,\chi}$ *hardness assumption holds [40]. Then, for* $m = O(n \log q)$ *and* A, R *as generated according to algorithms 1 and 2, the joint distribution* $(A, R \cdot A)$ *is computationally indistinguishable from uniform over* $\mathbb{Z}_q^{m\times(n+1)} \times \mathbb{Z}_q^{N\times(n+1)}$

Odded Regev clarifies that the condition $m > 2n \log q$ is sufficient to ensure security [40].

2.6. Numbering system

Currently, it is only possible to store unsigned integers using current homomorphic encryption schemes. It is possible to store numbers in floating point format, but doing so would require the implementation of some logic scheme to perform necessary steps such re-normalisation [34]. Luckily, there is Q-notation, also known as fixed point representation. With this format it is possible to store negative numbers and fractions. The notation consists of two components, 2's complement and fractional bit representation.

2.6.1. Binary addition and multiplication

Modern computation is currently dominated by digital computation which utilises binary representation of numbers and states. The standard way of representing numbers in binary is called 1-complement (pronounced one's-complement). 1-complement is easy to understand as counting works much the same as in base-10. Addition and multiplication are also easy to perform. Addition of two numbers x and y can be performed by counting up from x up to each 1 digit in y (see table 2.6.1a). Multiplication is slightly more involved. Multiplication is performed through the following procedure: take the first digit in y, if it is 1, then add x to the result. For the next digit in y add $x \cdot 2$, then $x \cdot 2^2$, $x \cdot 2^3$ and so forth (see table 2.6.1b). Computers are limited to a fixed number of digits that can be represented. If the result of an addition or multiplication consists of more digits than a computer can represent, this is called overflow (see figure2.6.1c). Sections of binary that exceed the computers capacity, these are usually discarded and ignored (though most computers are able to detect if overflow occurs).

0011	0101	1100
0010 +	0011 ×	0100+
0101	0101	1000
	1010 +	1000+
	1111	10000

(a) Addition: $3+2$ (b) Multiplication 5×3 (c) $12+4$ causing Overflow

Table 2.6.1: Binary addition and multiplication

2.6.2. Negative numbers

When introducing negative numbers, 1-complement alone is not enough to perform all operations we might want to carry out. Of course, on paper we can simply use negative sign notation we would with decimal. The goal is to run these calculations on a computer and a computer is not as flexible as the human mind.
We could simply add an extra bit to indicate sign and change the way addition is performed. However, there is a smarter solution to this.
Another way of representing binary numbers is using so called 2-complement (pronounced Two's complement). The most significant bit of a number is reserved for sign. When the sign bit is zero, the other bits represent positive numbers the same as in 1-complement. When the sign bit is 1, the other bits represent $2^{\ell}-|x|$, where ℓ is the number of bits reserved to represent decimal number x (see table 2.6.2).

2-complement					decimal
0	1	1	1	1	= 15
0	0	0	1	0	= 2
0	0	0	0	1	= 1
0	0	0	0	0	= 0
1	1	1	1	1	= -1
1	1	1	1	0	= -2
1	0	0	0	0	= -16

Table 2.6.2: Two's complement for numbers stored in 5-bit large container

This way of representing numbers has the benefit that addition and multiplication logic used for 1-complement can be used for both positive and negative numbers and any combination (see 2.6.3). Bits that carry out side of the container are simply ignored, therefore the overflow condition is now slightly different. Overflow is now when an illegal sign change occurs (see table 2.6.3c). In other words the following must hold:

- negative + negative = negative
- positive + positive = positive

Otherwise overflow has occurred.

1101	1110	0111
0100 +	1110 ×	0010+
0001	0100	1001

(a) Addition: $-3+4$ (b) Multiplication -2×-2 (c) $7+2$ causing Overflow

Table 2.6.3: Binary addition and multiplication

2.6.3. Q-notation

Lastly, we can adapt the representation even further to represent fixed precision fractions. Define ℓ the total length of a binary number, m the amount of bits designated to represent the integer portion of a number and n the fractional portion of a number. Then we know $\ell = m+n+1$. A fraction is the result of a division of two integers, which extends to binary. The most common format to store binary fractions is by using Q-notation, which relates to base-10 through the relation:

$$d = -b_\ell 2^{\ell-n-1} + \sum_{i=1}^{\ell-1} 2^{i-n-1} b_i$$

where d is a decimal number, b binary and b_i refers to the $i-$th digit in b ordered from least to most significant. This notation is also called the Q-notation (see table 2.6.4 for examples). Fixed point number representation will be denoted as $Q(m,n)$, where m denotes the amount of integer bits and n denotes the amount of fractional bits.
Fractions in decimal can be converted to Q-notation by multiplying the number by 2^n rounding it and converting the resulting number to 2-complement. These binary numbers can be converted back to base-10 by dividing the integer value by 2^n.

A more intuitive interpretation of Q-notation is that where normally each bit represents a power of two, Q-notation extends to negative exponents. Equation 2.6.1 illustrates this with an example.

$$\begin{array}{ccccccc} 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & \\ 0 & 1 & 1 & 0 & 1_2 & = & 13_{10} \end{array}$$
$$\begin{array}{ccccccc} 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & & \\ 0 & 1 & 1 & 0 & 1_2 & = & 3.25_{10} \end{array} \tag{2.6.1}$$

Q-notation					decimal
0	1	1	1	1	= $\frac{15}{4}$
0	0	0	1	0	= $\frac{1}{2}$
0	0	0	0	1	= $\frac{1}{4}$
0	0	0	0	0	= 0
1	1	1	1	1	= -$\frac{1}{4}$
1	1	1	1	0	= -$\frac{1}{2}$
1	0	0	0	0	= -4

Table 2.6.4: Q-notation stored in 5-bit large container with $m = 2$

The useful property of Q-notation is that arithmetic is fully compatible with for 1-complement (aside from overflow handling of course).

2.6.4. Fixed precision addition and multiplication

Adding two fixed precision values is similar to the addition of two 2's complement values, in fact, when two values with the same amount of fractional bits are added, the binary values can be added as two one's complement numbers. It is also possible to add two fractional numbers with different amounts of fractional bits, but this will not be elaborated on as it is not necessary in the context of this thesis.
When multiplying two fractions in decimal, the denominator grows. Something similar happens to fixed precision numbers. When multiplying two fixed precision numbers, the decimal point shifts by the number of fractional bits of each number combined. To demonstrate this property, take the following theorem:

Theorem 2. *When multiplying two fixed precision numbers representing $Q(m_1,n_1)$ and $Q(m_2,n_2)$ where $m_1+n_1 = m_2+n_2 = \ell$ (i.e. ℓ is the register size), if the output register is the same size as the input registers and the operation does not cause overflow, the result represents $Q(\ell-(n_1+n_2), n_1+n_2)$.*

Proof. Take two fixed point numbers, $a_{\text{fp}} = a' \cdot 2^{-n_1}$ and $b_{\text{fp}} = b' \cdot 2^{-n_2}$, where $a' = \lfloor a \cdot 2^{n_1} \rceil$ and $b' = \lfloor b \cdot 2^{n_2} \rceil$. If $|a'| \cdot |b'| < 2^\ell$, then the following will compute without overflow:

$$a_{\text{fp}} \cdot b_{\text{fp}} = a' \cdot 2^{-n_1} \cdot b' \cdot 2^{-n_1} = (a' \cdot b') \cdot 2^{-(n_1+n_2)} \tag{2.6.2}$$

□

To ensure that a multiplication is performed successfully, ensure that for $Q(m_1, n_1)$ and $Q(m_2, n_2)$, n_1 and n_2 are chosen based on necessary precision. Set the integer portion to what is left of register size ℓ, then $m_1 = \ell - n_1$ and $m_2 = \ell - n_2$. So long as the integer portion of the result is smaller than $2^{\ell-(n_1+n_2)}$ an output of representation $Q(\ell - (n_1 + n_2), n_1 + n_2) = Q(m_3, n_3)$ will always fit the output register.
Of course, after multiple computations, the size of the Q-notation will exceed available register size. In other words, overflow occurs. To prevent this from happening, numbers are right shifted to shift the decimal point to a desired position, which is called truncation. Unfortunately, this is not yet possible to do with encrypted numbers, thus alternate solutions are needed. The chosen solution will be discussed in section 4.7 and chapter 5 will elaborate with a numerical example.

3

Gentry's Encryption scheme revisited

As it stands, the encryption using Gentry's scheme as well as the homomorphic operations are computationally expensive. Ciphers consist of binary elements. Hence when multiplying or adding two ciphers, such an operation consists of many bit-wise operations. A naive approach to an implementation leads to large inefficiencies. This chapter will introduce notation that improves legibility and simplifies manipulation of Gentry's encryption scheme. All functions are replaced with analytical equivalents utilising the new notation. These improvements lay bare avenues of potential simplification which will be followed to their natural conclusion in this chapter.
Section 3.1 starts off the chapter by introducing new notation and how it can be used to replace functions proposed by Gentry's with analytical equivalents. Section 3.2 introduces the concept of reduced ciphers and how they can be applied. Finally section 3.3 discusses the benefits of using reduced ciphers.

3.1. New notation

Legibility of the description and proofs that will follow in later sections can be significantly simplified by introducing the following notation: Take some positive integer a. The relationship between its binary representation and representation in decimal can be characterised through:

$$a = \sum_{i=0}^{\infty} 2^i a^{[i]} \tag{3.1.1}$$

In Gentry's scheme [22] cipher space has to be constrained and on top of that, using physical hardware means representation of integers will be limited in precision. Usually, this means that following mathematical operations, if the result exceeds the size which the available registers can represent, the bits that exceed this size, or overflow, are ignored. This behaviour will be made explicit through the following notation:

$$(a)^{\ell} = \sum_{i=0}^{\ell-1} 2^i a^{[i]} \tag{3.1.2}$$

This means that if $a \leq q$ where $q = 2^{\ell} - 1$, then $(a)^{\ell} = a$, if $a > q$, then $(a)^{\ell} \neq a$. This operation has a useful property, given two numbers $a, b \in \mathbb{N}$, then

$$(b + (a)^{\ell})^{\ell} = ((b)^{\ell} + (a)^{\ell})^{\ell} = ((b)^{\ell} + a)^{\ell} = (b + a)^{\ell} \tag{3.1.3}$$

Although this property is not used explicitly in Gentry's encryption scheme, it is integral to its functionality as many derived properties depend on this property.
To further improve legibility, for some positive integer $a \in \mathbb{N}$, let

$$[a]^{\ell} = \textbf{BitDecomp}(a) \tag{3.1.4}$$

This notation makes the dependency on parameter ℓ more explicit. Finally, note that given a vector, containing the ℓ bits of some number a, number a can be recovered through:

$$\mathbf{BitDecomp}^{-1}([a]^{\ell}) = \left[a^{[0]}, a^{[1]}, a^{[2]}, \cdots, a^{[\ell-1]}\right] \begin{bmatrix} 1 \\ 2 \\ 4 \\ \vdots \\ 2^{\ell-1} \end{bmatrix} = [a]^{\ell} \cdot g^{\top} = a \tag{3.1.5}$$

Vector $g = \left[1, 2, 4, \cdots 2^{\ell-1}\right]$ is commonly called the gadget vector. The $\mathbf{BitDecomp}^{-1}$ function can be generalised for matrices. Let $\otimes$ be the Kronecker product, then:

$$\begin{aligned} &A \in \mathbb{Z}_q^{n_1 \times n_2 \cdot \ell} \\ &\mathbf{BitDecomp}^{-1}([A]^{\ell}) = [A]^{\ell} \otimes g^{\top} = [A]^{\ell} \cdot I_{n_2} \otimes g^{\top} = [A]^{\ell} \cdot G_{n_2} \end{aligned} \tag{3.1.6}$$

where I_{n_2} is the identity matrix of size $n_2 \times n_2$ and G_{n_2} is a matrix called the gadget matrix. This matrix will be used in presenting proofs, but will not actually be used in practical application.
Using the new notation and the gadget matrix, the functions proposed by Gentry can reconstructed:

Definition 1. *For any matrix* $A \in \mathbb{Z}_q^{n_1 \times n_2}$

$$\begin{aligned} \textbf{\textit{BitDecomp}}(A) &= [A]^{\ell} \\ \textbf{\textit{BitDecomp}}^{-1}([A]^{\ell}) &= [A]^{\ell} \cdot G_{n_2} = A \\ \textbf{\textit{Flatten}}(A) &= [A \cdot G_{n_2}]^{\ell} \\ \textbf{\textit{Powersof2}}^{-1}(A) &= A \cdot G_{n_2}^{\top} \end{aligned} \tag{3.1.7}$$

Note: recall that if any entries in A are larger than $q-1$, *i.e.* $A \notin \mathbb{Z}_q^{n_1 \times n_2}$, *then* $[A]^{\ell} \cdot G_{n_2} \neq A$.

With the newly introduced notation, the relationships from equations (2.5.2) and (2.5.3) can be rewritten. From the original description it is not immediately clear why these relationships hold. Utilising the new notation it can easily be confirmed algebraically:

$$\langle \mathbf{BitDecomp}(a), \mathbf{Powersof2}(b) \rangle = \langle [a]^{\ell}, b \cdot G^{\top} \rangle = [a]^{\ell} \cdot (b \cdot G^{\top})^{\top} = [a]^{\ell} \cdot G \cdot b^{\top} = a \cdot b^{\top} = \langle a, b \rangle \tag{3.1.8}$$

Note that the last step in the derivation only holds in case $a \leq q$. If the size of a is unknown, the following relationship holds for any a:

$$\langle [a]^{\ell}, b \cdot G^{\top} \rangle = \langle (a)^{\ell}, b \rangle \tag{3.1.9}$$

Next, take $\langle a', \mathbf{Powersof2}(b) \rangle$. Substitute a' for $[a]^{\ell}$ and rewrite the equation using the new notation, then the relationship from 2.5.3 follows naturally:

$$\langle a', \mathbf{Powersof2}(b) \rangle = \langle [a]^{\ell}, b \cdot G^{\top} \rangle = [a]^{\ell} \cdot G \cdot b^{\top} = \langle [a]^{\ell} \cdot G, b \rangle = \langle \mathbf{BitDecomp}(a'), b \rangle \tag{3.1.10}$$

$$\langle [a]^{\ell}, b \cdot G^{\top} \rangle = \langle [[a]^{\ell} \cdot G]^{\ell}, b \cdot G^{\top} \rangle = \langle \mathbf{Flatten}(a'), \mathbf{Powersof2}(b) \rangle \tag{3.1.11}$$

Next, we can rewrite the homomorphic properties using the new notation.

$$\begin{aligned} \mathbf{MultConst}(C, \alpha) &= \mathbf{Flatten}(\mathbf{Flatten}(I \cdot \alpha) \cdot C) = \left[[\alpha \cdot I_N \cdot G_{n+1}]^{\ell} \cdot C \cdot G_{n+1} \right]^{\ell} \\ \mathbf{Add}(C_1, C_2) &= \mathbf{Flatten}(C_1 + C_2) = [(C_1 + C_2) G_{n+1}]^{\ell} \\ \mathbf{Mult}(C_1, C_2) &= \mathbf{Flatten}(C_1 \cdot C_2) = [(C_1 \cdot C_2) G_{n+1}]^{\ell} \end{aligned} \tag{3.1.12}$$

3.2. Reduced cipher

Having introduced the new notation, I now propose the so called Reduced cipher:

$$\begin{aligned} &\tilde{C} \in \mathbb{Z}_q^{(n+1) \cdot \ell \times (n+1)} \\ &\tilde{C} = C \cdot G \end{aligned} \tag{3.2.1}$$

The reduced cipher removes many calculation steps and simplifies all aspects of Gentry's encryption scheme. Reduced ciphers largely reduces the computational complexity and memory footprint of encryption and all homomorphic operations. Note the pattern of how the message μ is incorporated in a cipher:

$$C = \textbf{Flatten}(I_N \cdot \mu + \textbf{BitDecomp}(R \cdot A))$$

Before the **Flatten** function is applied, message μ is added to the diagonal of matrix **BitDecomp**$(R \cdot A)$. Matrix **BitDecomp**$(R \cdot A)$ is already a matrix with values between 0 and 1. Large parts of the message is then discarded when **Flatten** is applied. This regularity can be exploited. Rewriting the generation of cipher C with the new notation yields:

$$C = [(I_N \cdot \mu + [R \cdot A]^\ell) \cdot G]^\ell$$

Note the following property that follows from the new notation:

Lemma 3. *For any matrix $\Lambda \in \mathbb{N}^{n_1 \times n_2}$, we have $[\Lambda]^\ell G_{n_2} = (\Lambda)^\ell$.*

Proof. First consider $\alpha \in \mathbb{N}$. for any α it holds

$$(\alpha)^\ell = \sum_{i=0}^{\ell-1} 2^i \alpha^{[i]} = \left[\alpha^{[0]}, \ldots, \alpha^{[\ell-1]}\right] \cdot g = [\alpha]^\ell \cdot g\,.$$

Then apply this relation on each element of Λ, giving

$$(\Lambda)^\ell = [\Lambda]^\ell \cdot I_{n_2} \otimes g = [\Lambda]^\ell \cdot G_{n_2}$$

□

Using lemma 3, the generation of a reduced cipher can be performed as follows:

$$\begin{aligned}
\tilde{C} =& [(I_N \cdot \mu + [R \cdot A]^\ell) \cdot G]^\ell \cdot G = \\
& ((I_N \cdot \mu + [R \cdot A]^\ell) \cdot G)^\ell = \\
& (G \cdot \mu + [R \cdot A]^\ell \cdot G)^\ell = \\
& (G \cdot \mu + (R \cdot A)^\ell)^\ell = \\
& (G \cdot \mu + R \cdot A)^\ell
\end{aligned}$$

Furthermore, the gadget matrix G has a very simple structure and can be implemented without performing a matrix multiplication. Take some matrix $\Lambda \in \mathbb{Z}_2^{N \times N}$ and some number α, then let $\gamma(\alpha, \Lambda)$ be:

$$\Gamma = \gamma(\alpha, \Lambda) = \begin{cases} \Gamma_{i,j} = (\tilde{\Lambda}_{i,j} + (\alpha \ll i \bmod \ell)^\ell)^\ell & \text{if } j = \lfloor i/\ell \rfloor\,, \\ \Gamma_{i,j} = \tilde{\Lambda}_{i,j} & \text{otherwise}\,. \end{cases} \tag{3.2.2}$$

A reduced cipher $\tilde{C}$ can then be efficiently generated through $\tilde{C} = \gamma(\mu, R \cdot A)$. Given the definition of reduced ciphers, take the following theorem:

Theorem 4. *Given ciphers $C_1, C_2 \in \mathbb{Z}_2^{N \times N}$ and scalar $\alpha \in \mathbb{Z}_q$ the existing homomorphic operations can equivalently be written using the reduced ciphers as*

$$C_3 = [(C_1 + C_2)G_{n+1}]^\ell \Longleftrightarrow \tilde{C}_3 = \left(\tilde{C}_1 + \tilde{C}_2\right)^\ell \tag{3.2.3}$$

$$C_4 = [(C_1 \cdot C_2)G_{n+1}]^\ell \Longleftrightarrow \tilde{C}_4 = \left(C_1 \cdot \tilde{C}_2\right)^\ell \tag{3.2.4}$$

$$C_5 = [[\alpha G_{n+1}]^\ell \cdot C_1 G_{n+1}]^\ell \Longleftrightarrow \tilde{C}_5 = ([\alpha G_{n+1}]^\ell \, \tilde{C}_1)^\ell \tag{3.2.5}$$

Proof. Each equivalence will be proven separately below. To do so the notation from Definition 1, as well as the result from Lemma 3 will be used.

$$\begin{aligned}
\tilde{C}_3 &= [(C_1 + C_2)G_{n+1}]^\ell \, G_{n+1} \\
&= (C_1 G_{n+1} + C_2 G_{n+1})^\ell = \left(\tilde{C}_1 + \tilde{C}_2\right)^\ell \\
\tilde{C}_4 &= [(C_1 \cdot C_2)G_{n+1}]^\ell \, G_{n+1} = \left(C_1 \cdot \tilde{C}_2\right)^\ell \\
\tilde{C}_5 &= \left[[\alpha I_N G_{n+1}]^\ell \cdot C_1 G_{n+1}\right]^\ell G_{n+1} \\
&= \left([\alpha G_{n+1}]^\ell \cdot \tilde{C}_1\right)^\ell
\end{aligned}$$

□

Finally, decryption can be rewritten using the novel notation as follows:

$$\mu = \mathbf{MPDec}((CG_{n+1}s)^{\ell}) = \mathbf{MPDec}((\tilde{C}s)^{\ell}) \tag{3.2.6}$$

3.3. Computational complexity

Performance of the Gentry scheme is improved using the new notation in combination with the reduced cipher. This will be called the reduced cipher approach. This section will highlight the differences between the naive approach and the reduced cipher approach in terms of computational complexity and memory usage. The naive approach to computation of homomorphic multiplications can be broken up into the following three stages:

Given some plain text message $\mu \in \mathbb{Z}_q$ where $q \in \mathbb{N}$, which requires ℓ bits of storage and some security parameter n and $N = (n+1)\cdot\ell$. Homomorphic multiplication then becomes

$$\begin{aligned} C_{1+2} &= C_1 + C_2 & C_{1+2} &\in \mathbb{Z}_3^{N\times N} \\ \tilde{C}_3 &= (C_{1+2}\cdot G_{(n+1)})^{\ell} & \tilde{C}_3 &\in \mathbb{Z}_q^{N\times(n+1)} \\ C_3 &= \left[\tilde{C}_3\right]^{\ell} & C_3 &\in \mathbb{Z}_2^{N\times N} \end{aligned}$$

Addition of two ciphers would require the generation and storage of an intermediate matrix which can contain values 0,1 and 2 which would require unconventional storage conventions. Producing the final result requires the flattening operation. Multiplication of two ciphers is even worse as its intermediate product can contain values between 0 and q even though the result contains only binary elements:

$$\begin{aligned} C_{1\times2} &= C_1 \cdot C_2 & C_{1\times2} &\in \mathbb{Z}_{(N+1)}^{N\times N} \\ \tilde{C}_4 &= (C_{1\times2}\cdot G_{(n+1)})^{\ell} & \tilde{C}_3 &\in \mathbb{Z}_{(N+1)\cdot q}^{N\times(n+1)} \\ C_4 &= \left[\tilde{C}_4\right]^{\ell} & C_3 &\in \mathbb{Z}_2^{N\times N} \end{aligned}$$

Using reduced ciphers, addition and multiplication can be performed in one step (see equations 3.3.1 and 3.3.2). The result can be produced directly, as the Flattening operation is incorporated through discarding overflow. Given that any target platform will have limited precision, the $(\cdot)^{\ell}$ operation is an inherent quality of the addition process.

$$\tilde{C}_3 = \left(\tilde{C}_1 + \tilde{C}_2\right)^{\ell} \tag{3.3.1}$$

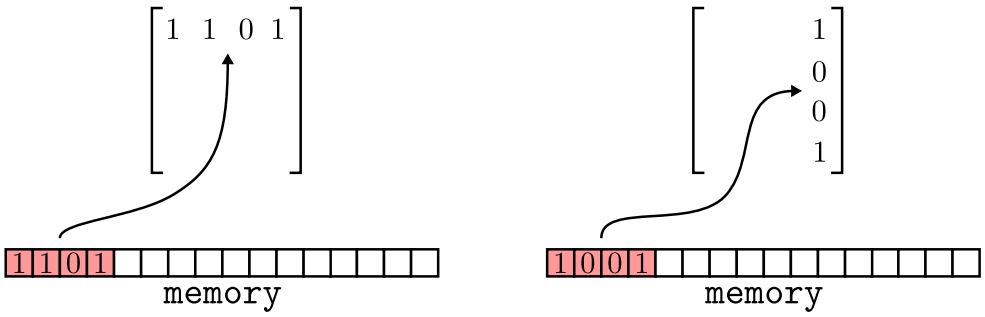
$$\tilde{C}_4 = \left(C_1 \cdot \tilde{C}_2\right)^{\ell} = \left(\tilde{C}_1^{\top} \cdot C_2\right)^{\ell} \tag{3.3.2}$$

Cipher multiplication as described in equation 3.3.2 can directly be implemented on an FPGA, however on a CPU it is important which form the reduced cipher equivalent takes. Take note of how in equation 3.3.2, only one of the ciphers can be in reduced form. Take the operation

$$C_1 \cdot \tilde{C}_2 \tag{3.3.3}$$

Two ensure optimal performance, it important to take into account how data is loaded from memory into the CPU. To perform the matrix multiplication from equation 3.3.3, individual bits of the reduced cipher $\tilde{C}_1$ have to accessed. In conventional computing, data is stored one dimensionally, the only dimension being a single address at which a 64 bit chunk of memory is located. To store matrices, they have to be stored either one row at a time, or one column at a time. These conventions are called row-major order and column major order respectively. Which convention is used depends on the programming language used (see figures 3.3.1). Conventional CPU's have different types of memory, mass storage memory, work memory and cache memory. Mass storage is relatively cheap, but slow. Work memory is faster, but more expensive and smaller. Cache is memory that is located within the CPU, this is the fastest memory. Currently most CPU's have a memory interface that support either single channel or dual channel mode, which utilise a 64 bit and 128 bit

bus width respectively [29]. Therefore data can be moved between different memory devices in 64 bit or 128 bit chunks.



(a) Row major order storage of matrix

(b) Column major order storage of matrix

Figure 3.3.1: Matrix storage in conventional computing

The loading operation necessary for evaluating equation 3.3.3 will be efficient so long as $\tilde{C}_1$ stored in row major order.

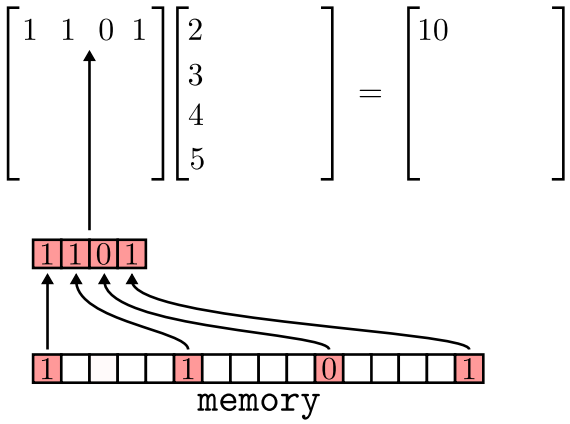


Figure 3.3.2: Due to the matrix being stored in column-major order, more memory has to be loaded in to perform the matrix multiplication

If the matrix is stored with the column major order convention, for one multiplication of a row and column, only the first bit of each element loaded would be used. Figure 3.3.2 illustrates storage convention can lead to data being loaded into cache unnecessarily (with 4 bits instead of 64 for clarity). Conventional CPU's only have a limited amount of cache memory, which is the memory that can directly be used in computation. Ciphers tend to be rather large, in which case large parts of memory have to be loaded in and out of cache memory which takes time. If matrices are stored in column major order the alternative form should be used:

$$\tilde{C}_1^{\top} \cdot C_2 \tag{3.3.4}$$

The benefits of the adaptations can be demonstrated by comparing the amount of elements that have to be evaluated to perform each operation using the naive approach as compared to the adapted notation. The result of such a comparison can be seen in table 3.3.1.

Table 3.3.1: Number of operations required to perform homomorphic operations.

Cipher&Cipher	Homomorphic addition		Homomorphic multiplication	
	Cipher	Red. Cipher	Cipher	Red. Cipher
Bit Operation	0	0	$\mathcal{O}(n^3\ell^3)$	$\mathcal{O}(n^3\ell^2)$
Addition	$\mathcal{O}(n^3\ell^2)$	$\mathcal{O}(n^2\ell)$	$\mathcal{O}(n^3\ell^3)$	$\mathcal{O}(n^3\ell^2)$
Multiplication	$\mathcal{O}(n^3\ell^2)$	0	$\mathcal{O}(n^3\ell^2)$	0
Memory	$\mathcal{O}(n^2\ell^2)$	$\mathcal{O}(n^2\ell^2)$	$\mathcal{O}(n^2\ell^2\log(n\ell))$	$\mathcal{O}(n^2\ell^2)$
Cipher&Known				
Bit Operation	$\mathcal{O}(n^3\ell^2)$	$\mathcal{O}(\ell)$	$\mathcal{O}(n^2\ell^3)$	$\mathcal{O}(n^2\ell^2)$
Addition	$\mathcal{O}(n^3\ell^2)$	$\mathcal{O}(n\ell)$	$\mathcal{O}(n^3\ell^3)$	$\mathcal{O}(n^2\ell^2)$
Multiplication	$\mathcal{O}(n^2\ell)$	0	$\mathcal{O}(n^3\ell^2)$	0
Memory	$\mathcal{O}(n^2\ell^2)$	$\mathcal{O}(n^2\ell^2)$	$\mathcal{O}(n^2\ell^2\log(\ell))$	$\mathcal{O}(n^2\ell^2)$

Table 3.3.1 consists of two parts: the top portion "Cipher&Cipher" which are the homomorphic operations between two ciphers and the second "Cipher&Known" which are the homomorphic operations between a Cipher and an unencrypted value. The two main columns "Homomorphic addition" and "Homomorphic multiplication" are both split in two columns for the cipher and reduced cipher equivalent of each type of homomorphic operation. Each homomorphic operation can be broken down into bit operations, additions and multiplication of individual elements. As a final note, the memory footprint is also given. The amount of memory that has to be handled is significantly reduced. Also, note that a need multiplication of individual elements is entirely removed. Homomorphic addition and multiplication can be computed all or mostly with addition and bit operations. With respect to an FPGA design, this makes homomorphic multiplication, the most computationally expensive operation, much more space efficient, reducing compute time. The importance of space efficiency will be elaborated on in the next chapter, section 4.3.2.

4

Hardware implementation

The practical implementation of a feedback controller requires the design and application on a hardware platform. This chapter will start with the introduction of the controller topology that will be implemented with FHE in section 4.1. Section 4.2 will broadly discus how the controller will be integrated with a physical plant. Section 4.3 will go into detail as to what an FPGA is, its functionality and how it is deployed. The generation of random numbers is non-trivial, especially given the tight constraints on both time and quality that encryption requires. Hence, section 4.4 will describe in detail, the decisions that have been made in the design of the random number generation. Sections 4.5 discusses the subsystems that performs arithmetic, followed by section 4.6 which covers the timing mechanisms used for coordination of the various subsystems. The chapter ends section 4.7 with a detailed explanation of how the plant interface and controller are designed, which summarises how all the subsystems are connected to make up larger subsystem that combine to form system as a whole.

4.1. controller topology

As mentioned in section 1.2, one of the goals of this research is to facilitate the implementation of a wide range of feedback controllers. A discrete time state feedback controller is a common controller topology that utilises memory which is a requirement for more complex feedback control. Such a controller could be expanded with disturbance observers or Kalman filtering and more.
The observer that will be implemented for the purposes of this research will be a Luenberg observer (see [45] for more information). The controller topology then becomes:

$$\begin{cases} \hat{x}(k+1) = A_d\hat{x}(k) + B_d u(k) + L(y(k) - C\hat{x}(k)) \\ u(k+1) = -K\hat{x}(k+1) \end{cases} \tag{4.1.1}$$

Note that the future control effort u is calculated. The control effort of the next time step is calculated ahead of time. This is because the computation of the control effort takes time, which would introduce a delay. In some systems this delay might be negligible, but in the case of most real time control systems this is not the case. This is especially the case for the system that is the goal of this thesis. The transfer of data alone is likely to introduce large delays, hence the choice of this design. The control effort for the next time step is calculated and held for application at the correct time. Figure 4.1.2 illustrates the mechanism (see figure 4.1.1 for the convention of the diagram). The dimensionless actuator state has to be switched from 0.0 to 0.56, followed by 0.94. If the current control effort is calculated, the calculation time causes the update of the actuation state to be delayed (see figure 4.1.2a). If the calculation of the control effort is non-deterministic, delay would also vary in length, exacerbating the issue. Calculating the future control effort allows for the controller to actuate at regular intervals (see figure 4.1.2b).

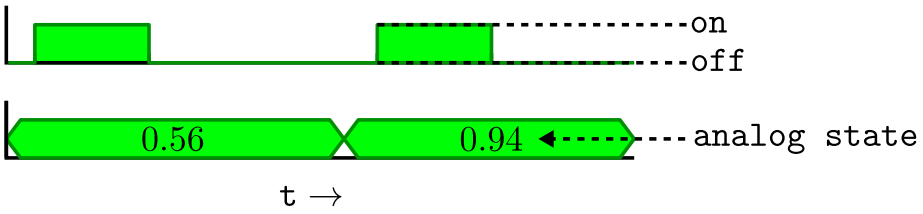
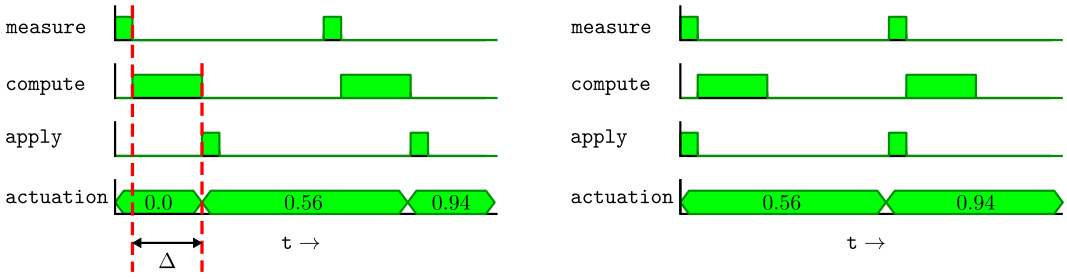


Figure 4.1.1: Timing diagram convention



(a) Delay in control effort application

(b) Control effort applied at the moment of sensor measurement

Figure 4.1.2: Comparison of the calculation of control effort of current time step vs calculation of the control effort one step ahead

To flip the sign of a value would normally happen negligibly fast. This is not the case when using FHE. Negation would mean a multiplication by -1, because bit operations are not available. Luckily this can be circumvented by making a slight change to the controller topology. Take matrices $\bar{L} = -L$ and $\bar{K} = -K$ and let $\bar{y}(k) = -y(k)$. Matrices L and K can easily be negated apriori and the sensor data can be negated before encryption, which yields the following controller:

$$\begin{cases} \hat{x}(k+1) = A_d \odot \hat{x}(k) + B_d u(k) + \bar{L}(\bar{y}(k) + C\hat{x}(k)) \\ u(k+1) = \bar{K}\hat{x}(k+1) \end{cases} \tag{4.1.2}$$

Finally, with encryption, the controller becomes:

$$\begin{cases} \hat{\mathbf{x}}(k+1) = \mathbf{A_d}\hat{\mathbf{x}}(k) \oplus \mathbf{B_d} \odot \mathbf{u}(k) + \bar{\mathbf{L}} \odot (\bar{\mathbf{y}}(k) \oplus \mathbf{C} \odot \hat{\mathbf{x}}(k)) \\ \mathbf{u}(k+1) = \bar{\mathbf{K}} \odot \hat{\mathbf{x}}(k+1) \end{cases} \tag{4.1.3}$$

Here, the bold print denotes that the signal or matrix is encrypted and $\odot$ and $\oplus$ denote homomorphic multiplication and addition respectively. The exact plant definition and modelling as well as the tuning of the control parameters will be discussed in chapter 5.

4.2. System overview

To design the feedback system, there are of course two problems have to be addressed as described in sections 2.5 and 1.1, FHE schemes have limited multiplicative depth. The other issue is that of truncation as discussed in section 2.6.4. The main goal of this thesis is to improve the performance of FHE in feedback control. Therefore the solution propose by [28] be used. The plant interface takes sensor data and encrypts it to be sent to the controller. The controller updates its observer states and computes the control effort. When the observer is updated, the decimal point of the observer states shifts. When the controller has finished computation, it sends the control effort as well as the observer states back to the plant interface. This is so that the plant interface can decrypt the observer states and control effort. This resets the multaplicative depth. Control effort is stored to be applied at the right time. Next, the plant interface can then truncate the observer states and control effort after which it re-encrypts the signals to be sent to the controller along with new sensor data for the calculation of the next observer states and control effort.

Physically, the entire system consists of two Field Programmable Gate Arrays, FPGA's. In a broad sense, an FPGA is a platform that allows for the programming of circuits. Section 4.3 will elaborate on how FPGA's function. Figure 4.2.1 shows an overview of the entire system.

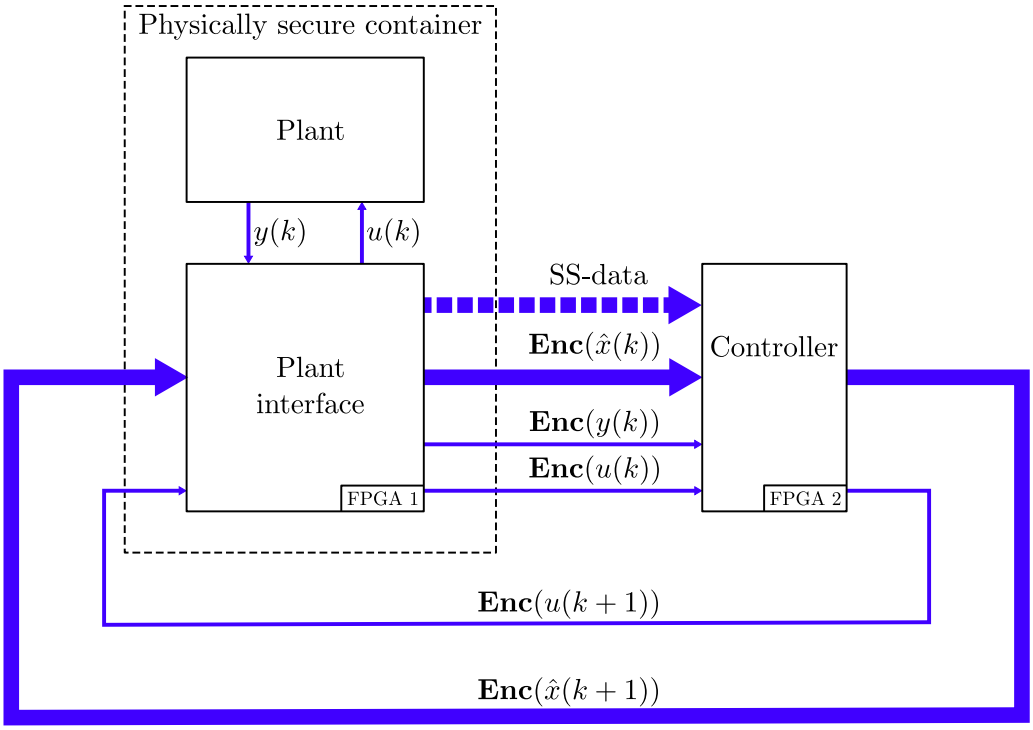


Figure 4.2.1: Diagram showing an overview of the control loop consisting of the plant, plant interface and controller

FPGA 1 functions as the plant interface and FPGA 2 functions as a controller. The interface and plant are connected physically in a shared container that prevents physical tampering. At boot up, the plant interface generates a public-private key pair and encrypts the state space data (state space matrices as well as controller- and observer gains), SS-data. After the state space data has been set to the controller, the regular control loop is started. This loop is then repeated to stabilise the plant.

4.3. FPGA design

A micro-controller is designed to run machine code, whereas an FPGA is collection of electronics that can be rewired. This means an FPGA is not programmed with programming language, instead a hardware design can be flashed to the chip. FPGA's and conventional CPU's are able to perform all the same computations, however a CPU requires large amount of abstraction. An FPGA is much more flexible and so they can be used to design bespoke solutions. Conventional CPU's also perform instruction largely sequentially, whereas FPGA designs can be made to perform most or all instructions in parallel.

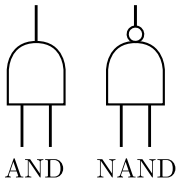
4.3.1. Logic Gates

At the basis of every computer are logic gates. The most simple logic gate is the NOT-gate. It takes one input that can either be true or false. If the input is true, the NOT-gate returns false and if the input it false, the gate returns true. Another important gate is the is the AND-gate. An AND-gate takes two inputs and only if both inputs are true, does the and-gate return true. Connecting an AND gate to a NOT-gate, is called a NAND-gate 4.3.1b. Table 4.3.1a shows the truth table of a NAND-gate where a and b denote the two inputs, c denotes the output and 0 denotes false and 1 denotes true. The NAND-gate is the logic gate that can be built as a circuit with the least amount of transistors. A NAND-gate is on its own completely Turing-complete, which means one can build a computer solely using NAND-gates (a machine that can solve any computational problem). For a NAND-gate to be Turing complete, one needs to be able to build an entire Turing complete operation

set. A Turing complete set includes a read, write and comparison function [15]. A NAND-gate can be used to construct any type of logic-gate (there are other gates such as OR and XOR, see figure 4.3.2a), but that leaves the question of how to build memory to write and read values.

input		output
a	**b**	**c**
0	0	1
1	0	1
0	1	1
1	1	0

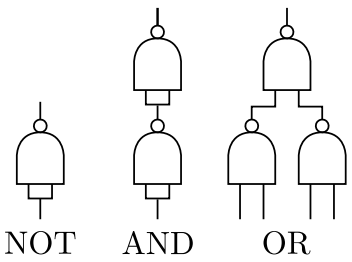
(a) Truth table of a NAND gate



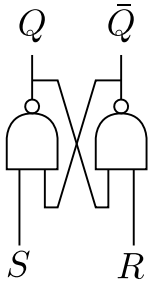
(b) AND gate and NAND gate

Figure 4.3.1: NAND gate truth table and depiction

Not only can NAND gates be used to construct all possible logic gates, they can also be constructed a so called SR-flipflop (see figure 4.3.2b).



(a) Other gates constructed with NAND gates



(b) SR-flipflop

Figure 4.3.2: NAND gate configurations

As can be seen from the truth table 4.3.1, an SR-flipflop functions as a form of memory, as when the inputs S and R are false, its output will be their previous values ($\bar{Q}$ is defined as NOT Q, so if Q is true, $\bar{Q}$ is false and vice versa). It must be noted that S and R should never both be set to true, because then the flipflop will behave unexpectedly (as can be seen from the illegal output).

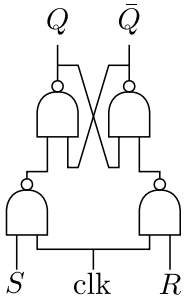
input		output	
S	R	Q	$\bar{Q}$
0	0	Q	$\bar{Q}$
0	1	0	1
1	0	1	0
1	1	×	×

Table 4.3.1: Truth table of an SR-flipflop

By adding two additional NAND-gates, a so called SR-flipflop can be created. This is shown in figure 4.3.3b. The truth table of an SR-flipflop can be seen in figure 4.3.3a. The input clk denotes the clock, an internal oscillator that switches between true and false at a set period. A flipflop only accepts input when the clk is true (and again all inputs at true would yields an invalid state). In reality the flipflop is further changed to ensure the flipflop can only change its value when a rising edge of the clock signal is detected too ensure only one operation per clock tick can be made. This ensures the behaviour of the flipflop is entirely predictable (the underlying electronics could otherwise behave unexpectedly).

input			output	
clk	S	R	Q	$\bar{Q}$
0	Na	Na	Q	$\bar{Q}$
1	0	0	Q	$\bar{Q}$
1	0	1	0	1
1	1	0	1	0
1	1	1	×	×

(a) Truth table of a gated SR flipflop, (Na = 'not applicable', i.e. the input is ignored)



(b) Gated SR-flipflop

Figure 4.3.3: Gated SR-flipflop thruth table and depiction

With a way of storing singular bits and all logic gates to perform read, write and compare bit, we have a Turing complete operation set and as such it is possible to build any type of classic computer with NAND-gates.

4.3.2. FPGA components

Rather than individual gates an memory cells, FPGA's contain a collection of generic digital components. The majority of components on an FPGA is usually the Adaptive Logic Module, ALM. ALM's commonly consist of a look up table, LUT, Full Adders, multiplexers and Flip-Flops. Full adders are circuits that can add up two bits. A Full adder has two outputs, the result and a carry out. If both inputs are 0, then the result is 0. If one of the two inputs is 1, then the result is 1. If both inputs are 1 then the the result is zero, because the result is carried on to the next digit and so the carry out is 1. A Multiplexer is a circuit which has multiple regular inputs, selection inputs and one output. A multiplexer has one or more selection inputs that can be used to select which of the regular inputs to pass through to the output. Take the ALM design from intel's Aria 10 FPGA [47] in figure 4.3.4.

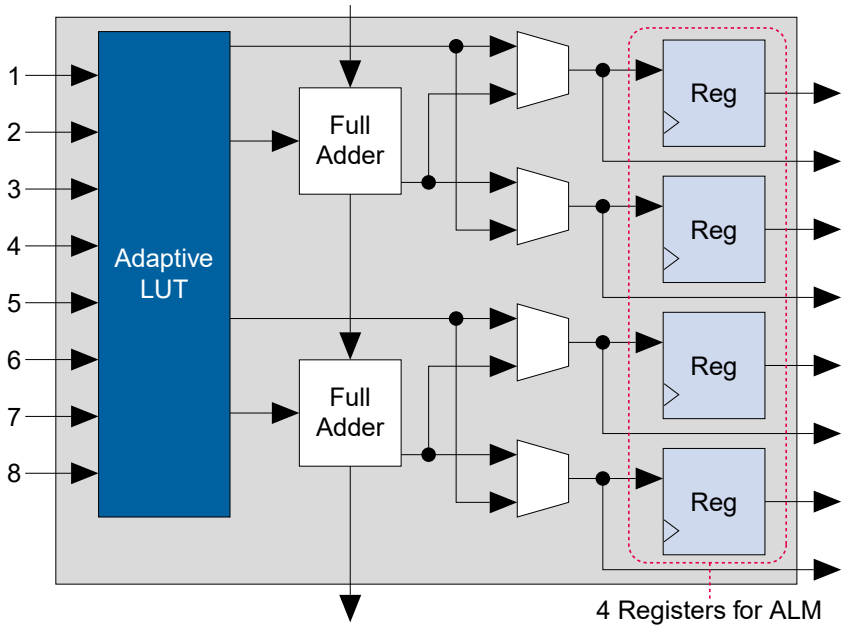


Figure 4.3.4: ALM design by Intel [47]

The LUT (dark blue, on the left) in this ALM can be pre-loaded with data. If the external input of the top Full Adder is unconnected, both Full Adders simply pass through the signal from the LUT. The Full Adders can be connected to other ALM's or they can be connected to the output of the ALM itself to produce counter and other such logic. The Multiplexers can be configured to manipulate the LUT output before the signal

reaches the Flip-Flops where output is captured (Reg, in light blue, to the right). This combination of logic units allows ALM's to be connected to one another to form a wide range of logical circuits. FPGA's also contain larger lookup tables, large registers of Flip-Flops and slower but more space efficient Block Random Access Memory, BRAM.
Finally, whilst ALM's can be used to perform addition, performing multiplication with these modules would be incredibly space inefficient. Therefore most FPGA's are equiped with Digital Signal Processing units, DSP's. These are circuits that can perform multiplication relatively efficiently. These modules are usually in short supply compared to other components on FPGA's. FPGA's usually have ALM's in the order of thousands to tens of thousands, whereas there are usually only tens to hundreds of DSP's available. To improve performance of any FPGA design, it is important to minimise or eliminate the use for DSP's. Note in table 3.3.1, that after adaptation of the Gentry scheme [22], using reduced ciphers, to perform homomorphic multiplication, the need for multiplication of individual components has been eliminated. This means that DSP's are not necessary for the generation of circuits to perform homomorphic multiplication (however they can of course still be used as to not waste them). This should allow for more efficient utilisation of hardware resources available on an FPGA, which should improve performance.

4.3.3. FPGA design language

An FPGA can be configured by an external computer using software. The international standard that is used to describe a hardware design (gate/flipflip routing) is called Very High Speed Integrated Circuit Hardware Description Language, VHSICHDL, which is shortened to VHDL. VHDL is a design language, not a programming language. VHDL can be interpreted to produce an actual physical computer chip. VHDL only describes what kind of binary logic to perform and needs to be interpreted as transistor networks to be implemented.
To write and test VHDL there are programs such as Vivado and Quaternion. They allow to test the logic as well as generate a physical design and a bitstream that can be used to flash an FPGA.
The modern VHDL language standard allows for fundamental bit operations as well as defining signed or unsigned values, that will be interpreted as a register of the size that is specified. The size of the container can be non-standard (such as 7 or 13). VHDL allows for **if**-statements as well as switch-case statements (state machines). These will be interpreted as a state registers with accompanying bit logic. FPGA's offer great freedom as well as speed, as the logic does not run in software but directly on the hardware. To illustrate, micro controllers usually allow for the calculation with variables of some fixed size in bits (usually 16, 32, or 64), whereas with an FPGA one could design circuit to perform calculations on variables consisting of 13 bits along side of variables of 20 bits. An FPGA also allows for complete concurrency, as all processes can run in parallel as long as timing constraints are taken into consideration. Some processes will be faster that others depending on how many logic gates are used, which needs to be accounted for.

4.4. Random number generation for encryption

To implement any modern encryption scheme, random number generation is vital to ensuring adequate security. There are two kinds of random number generation that are required for a functional design, uniform number generation and normal number generation. In many stages of Gentry's encryption scheme, some form of uniform number generation is utilised, only the generation of numbers from the χ-distribution to generate an error vector requires normally distributed random numbers. This is a key component in ensuring security and should therefore be handled with care. Section 4.4.1 will discuss the generation of uniform numbers and section 4.4.3 will discuss the generation of normally distributed random numbers. However, it is important to first discuss the generation of random numbers in practice.
Generating truly random numbers at high speed is generally infeasible. Hence, in cryptography, a common solution is to generate a smaller selection of truly random bits, which are then used to generate a seed that is fed to a pseudo-random number generator [19]. A psuedo-random number generator is a chaotic system that has some initial state from which it can generate non-repeating patterns for a long time before it returns to the initial state. This creates a seemingly random stream of uniformly distributed numbers. Psuedo-random number generators are generally very efficient and therefore very useful for testing. Luckily, the initial state, which is called the seed, can be periodically updated using truly random bits. This provides an efficient way of generating uniformly distributed numbers that are sufficiently random for the use of encryption.
Randomness is measured by the amount of entropy that sets of numbers exhibit. If a set of generated numbers exhibit high entropy, numbers are highly random, if entropy is low, there is a discernible pattern. Pseudo-random numbers are numbers that are statistically random, but that have been generated utilising some logic

or function and some initial state. Of course solely using pseudo-random numbers for encryption would be a security risk as it would be potentially possible to predict the generated numbers. To handle this issue, the inclusion of truly random bits solves this issue, assuming the logic or functions used are sufficiently chaotic. Such a generator is called a cryptographically secure pseudo-random number generator, CSPNRG. For more information on the topic, consult the book *Research Methods for Cyber Security* [19].
For the purposes of the research question of this thesis, psuedo-random number generation will suffice, as it will be enough to demonstrate the functionality of a control setup. Further research could delve into including additional logic to produce functionally true random number generation.

4.4.1. Uniform number generation

To determine the best way to generate pseudo-random numbers, it is important to consider the target platform. An FPGA has an abundance of simple logic and registers. This makes the platform well suited for the use of linear feedback shift registers, LFSR's [13]. LFSR's consist of a register, and some simple logic gates. In general, an LFSR has a register that is loaded with some begin state, which is called the seed. Then a few of the most significant bits are used to generate a new bit using some simple logic gates, which is called the feedback. This bit will be used as the output. Next the register is shifted to left and the first bit is set to the feedback bit. This concludes one cycle. By running an LFSR for multiple cycles an entire sequence of random bits can be generated, with which one could for instance construct random integers. The architecture that will be used is a simple XNOR-LFSR as specified by Xilinx [46]. The LFSR will follow the following procedure:

Algorithm 4 Basic LFSR

```
1: procedure LFSR(clk, enable)
2:     if rising_edge(clk) AND enable = 1 then
3:         feedback ← register[ℓ − 1] XOR register[ℓ − 2] XOR register[ℓ − 3] XOR register[ℓ − 4]
4:         register[ℓ − 2 : 1] ← register[ℓ − 2 : 0]
5:         register[0] ← feedback
6:         return feedback
7:     end if
8: end procedure
```

At initialisation, `register` will be set to some preprogrammed random binary string (seed). Variables `clk` and `enable` are signal lines that can either be high or low. `clk` will be connected to the internal clock of the hardware platform. By ensuring the LFSR can only be updated on a rising edge of the hardware clock ensures correct timing and thus synchronisation with the other parts of the hardware design. The enable line is used to turn the LFSR on only when needed. As in previous chapter ℓ is defined as the size of the internal register `register`. Due to the chaotic nature of the algorithm, there is no repetition for 2^ℓ cycles. This means that an LFSR with a 64-bits register, operating continuously at 400Mhz, there would be no repetition of the contents of the register for at least 1500 years.
Of course, to generate larger random sequences would require either concatenation of multiple LFSR subcircuits or running a singe LFSR circuit for multiple cycles. The final design utilises a hybrid solution. A circuit combines 256 LFSR's, with 64-bit internal registers, each with their own seed for a random 256-bit output. This output is then read for multiple cycles. Such a solution allows for high speed generation of random data whilst remaining small in footprint. Assuming the generation of random integers is allowed 1ms, at 400Mhz, the circuit can generate approximately 1.3 gigabytes of random data. If the random data is used to encrypt some message to be sent utilising an internet connection, this speed should be more adequate, given that the highest available internet download speeds are advertised at 10,000Mbps, which is 1.25 gigabytes per second [24]. The highest average speed in the world is more around 0.0375 gigabytes per second. Given these speeds, transmission is more likely to form a bottleneck rather than random number generation. As such, the aforementioned solution should offer adequate performance.

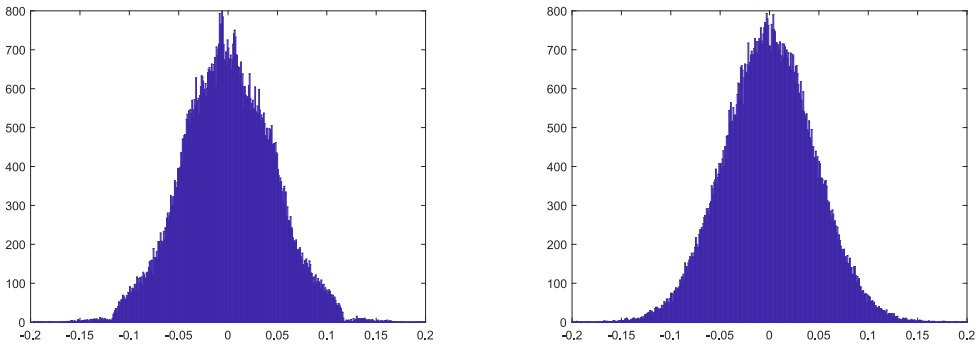
4.4.2. Random normally distributed numbers

The generation of random normally generated numbers is a more complex matter. There are a few popular solutions. They differ considerably, however, in essence they are all a mapping of uniformly generated numbers to normally distributed numbers.
The most simple but straight forward method is to use the Box-Muller transform [4]. Given two uniformly

random numbers $U_1, U_2 \in (0,1)$, two normally distributed numbers can be generated through:

$$\begin{aligned} Z_1 &= \sqrt{-2\ln U_2}\cos(2\pi U_1) \\ Z_2 &= \sqrt{-2\ln U_1}\sin(2\pi U_2) \end{aligned} \tag{4.4.1}$$

Two issues should be immediately clear. The previous section discussed how to generate randomly generated integers from random bits. This means that the generated integers will always range from $(0, 2^{\ell-1}-1)$ in case output is interpreted as unsigned integers or $(1-2^{\ell-1}, 2^{\ell-1}-2,)$ if signed. Therefore the output will have to be interpreted either as floating point numbers or fixed point numbers. Not only that, this algorithm would also require the implementation of a sine and cosine function, natural logarithm and a square root function. These functions could be implemented using either polynomials approximations, series expansions or by implementing more involved algorithms such as the CORDIC algorithm. The CORDIC algorithm requires iteration, which would complicate timing, as it would require analysis of execution time. The other options do not fair much better as in the case of utilising fixed point representation, such implementation would introduce distortion on top of the distortion that the Box-Muller transformation introduces inherently. On top of that, polynomial approximation requires multiplication. To get an approximation that is somewhat serviceable requires a polynomial of an order of at least 3 or 4, if not more which is already require many DSP-slices. Take the following example: The natural log can be split up into two polynomials and the cosine can be approximated with one. Choosing two 3rd order polynomials for the natural log and a single 4th order polynomial for the cosine yields disastrous results (see figure 4.4.1a). Picking orders of 5th, 9th and 6th respectively yields good results, but the order is far to high to be practical (see figure 4.4.1b).



(a) Box-muller method, utilising low order polynomial approximation

(b) Box-muller method, utilising polynomial approximation of appropriate order

Figure 4.4.1: Box-muller method example

Another more complex algorithm is the Ziggurat algorithm [30]. A newer more efficient version of the algorithm as proposed by Christopher McFarland [31], offers good performance, but still does not have a deterministic execution time.

The best solution for hardware implementation is the algorithm as proposed in *A New Hardware Efficient Inversion Based Random Number Generator for Non-uniform Distributions* [16]. Note that the authors have since published a version of their algorithm in [17] which allows for arbitrary precision. The precision that their previous algorithm offers is more than high enough, given that the normally distributed random numbers are scaled and rounded down (see equation 2.4.6). The algorithm from [16] is most well suited to hardware implementation, because instead of using only a single or a few high order polynomials, the algorithm instead elects to chop into many first or second order polynomials. On conventional hardware, such an approach would require the use of many **if**-statements to check which polynomial to use, which would make for a slow algorithm on conventional hardware. However, the simple nature of the type of check, an FPGA can perform all these checks instantly at a low hardware cost.

4.4.3. Normal distribution generation

The method described in [16] makes clever use of the properties of binary representation to construct a piecewise approximation of the inverse cumulative density function. Take normal distribution Z and uniform

distribution U and a value x, we want to find a transformation such that:

$$T(U) = Z \tag{4.4.2}$$

If such a transformation exists, it is possible to transform values from U to values in Z. Take the probability that value $Z \leq x$ and substitute $T(U)$ for Z.

$$\mathbf{Pr}(Z \leq x) = \mathbf{Pr}(T(U) \leq x) = \mathbf{Pr}(U \leq T^{-1}(x)) \tag{4.4.3}$$

Assume distribution U is uniform on the interval $(0,1)$, then we can define the so called cumulative probability function $\mathbf{Pr}(U \leq x)$ (i.e the chance that a number from U is equal to or smaller than x) as:

$$\mathbf{Pr}(U \leq x) = \begin{cases} x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } 1 < x \\ 0 & \text{otherwise .} \end{cases} \tag{4.4.4}$$

This means that if $0 \leq x \leq 1$, then we can write equation 4.4.3 as:

$$\mathbf{Pr}(Z \leq x) = \mathbf{Pr}(U \leq T^{-1}(x)) = \mathbf{Pr}(U \leq T^{-1}(x)) = T^{-1}(x) \tag{4.4.5}$$

Equation 4.4.5 shows that the probability density function of a normal distribution is the inverse of the transformation we seek. Thus if we can find the inverse of the probability function of Z, we can generate normally generated numbers from uniformly distributed numbers.

To find this function, we need to take the probability density function of a normal distribution:

$$\mathbf{Pr}(Z = x) = f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{4.4.6}$$

Here, μ and σ are real valued numbers defined as the mean and standard deviation respectively.
The cumulative density function can be found by integrating the probability density function on the domain $(-\infty, x)$

$$\mathbf{Pr}(Z \leq x) = \int_{-\infty}^{x} f(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{4.4.7}$$

There is no (known) analytical solution for the integral, however the target is to find and approximation, hence this should pose no problem. Finally, the inverse is the following:

$$\begin{aligned} \mathbf{Pr}(Z \leq x)^{-1} &= \sigma \cdot \sqrt{2}\, erf^{-1}(x - \mu) \\ erf(x) &= \frac{2}{\sqrt{\pi}} \int_{\infty}^{x} e^{-t^2}\, dt \end{aligned} \tag{4.4.8}$$

The $erf^{-1}(x)$ function too cannot be evaluated analytically. Numerical evaluation of $\mathbf{Pr}(Z \leq x)^{-1}$ yields the following plot:

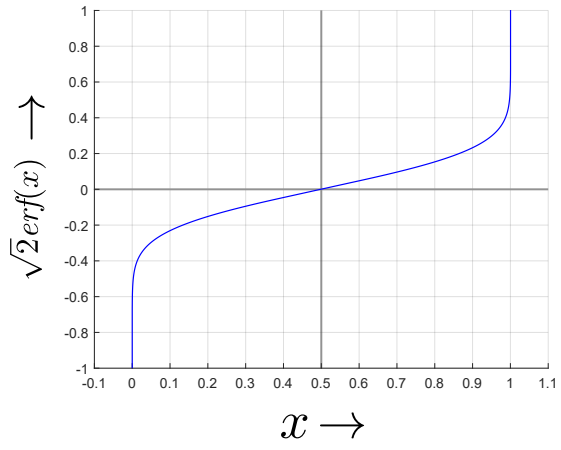


Figure 4.4.2: Inverse CDF

This function is an inverse cumulative density function, ICDF. By applying the ICDF to uniformly distributed numbers on the domain $(0,1)$ yields normally distributed numbers.
Note the symmetry around $x = 0.5$. To approximate the ICDF on the domain $(0,1)$, one can use an approximation on $(0,0.5)$ or $(0.5,1)$ and change the sign of the output based on the sign of the input. The next step is to choose a numbering system. The system that requires the least amount of computation would be to simply use fixed point numbers. Besides this, it is also a good choice since this makes it straight forward to use binary data supplied by LFSR's. Producing random uniformly distributed numbers on the domain $(0,1)$, is a simple question of adjusting the representation accordingly. Finally, fixed point numbers have an inherent property that makes them suitable to piece-wise approximation. Note that as x tends to 0, $\sqrt{2}erf^{-1}(x)$ tends to minus infinity (or to minus infinity as x tends to 1). Trying to fit polynomials that tend to infinity would fail to be accurate enough even when using higher order polynomials. This would lead to significant distortion around the tail ends of the distribution. The authors of [16] propose to approximate the ICDF on $(0,0.5)$ and split the function into sections logarithmically, each with their own polynomial, such that the approximation becomes more detailed for input closer to 0. Fixed point numbers are perfect to this end. Each bit in a fixed point number represents a power of 2. Given a 64-bit input, assuming unsigned fixed point numbers with representation $Q(0,64)$, the ICDF can be split into 64 sections. The first domain spans $[0,2^{-64}\rangle$, followed by domains spanning $[2^{-i},2^{1-i}\rangle$ where $i = 1,2,\cdots 64$.

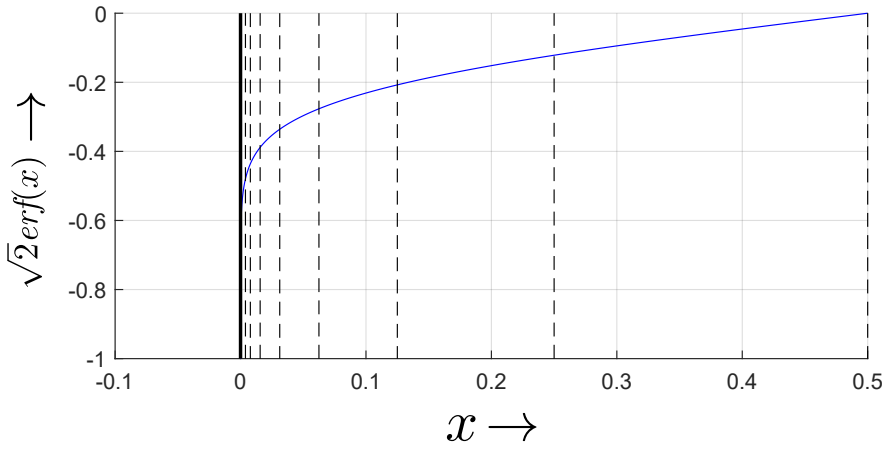


Figure 4.4.3: Inverse CDF slicing

When using fixed precision numbers, to find in which part of the interval an input falls is very simple. Take some value x where the leading bit is in the i-th position, then we know that $2^i \leq x < 2^{1+i}$ and so we know which polynomial to pick, by counting the number of leading zeros (see 4.4.9). There are efficient ways of utilising hardware to perform the count of leading zeros, such as proposed in *Modular Design Of Fast Leading Zeros Counting Circuit* [33]. However, this aspect is unlikely to need optimal implementation given the available hardware, thus the use of a simple lookup table should suffice.

$$\begin{aligned} &Q(0,64) \\ &x = 000000000000\,10110101010111101\ldots \\ &12 \text{ leading zeros} \\ &2^{-11} \leq x < 2^{-12} \end{aligned} \tag{4.4.9}$$

Finally, the authors also add that the sections closer to $x = 0.5$ might be too low resolution. To improve resolution in these sections, they can be split up into sections evenly. Set the amount of sections N to be a power of 2, $N = 2^n$, then selection of the evenly spaced sections can be done by simply reading the n bits, following the leading bit, as an index for selection. To implement this in hardware design, one can connect a multiplexer to the input register, using the output of the leading zero count as a selector, such that the first n-bits that follow the leading bit are piped to the input of a lookup table which select the corresponding polynomial. The selection procedure, can be summed up in algorithm 5.

Algorithm 5 Selection of a polynomial

```
1: procedure PolySel(clk, enable, x)
2:     if rising_edge(clk) AND enable = 1 then
3:         N_lz ← number of leading zeros of x
4:         β ← β ≪ (N_lz + 1)                    ▷ Removing the leading bit
5:         N_lin ← β ≫ (ℓ − n)                   ▷ Isolating bits for linear segment indexing
6:         θ ← LUT_pl(N_lz, N_lin)               ▷ Reading lookup table to select polynomial coefficients
7:         return θ
8:     end if
9: end procedure
```

Here θ is an array of length p which contains the polynomial coefficients according to the convention:

$$y = \theta_p x^p + \theta_{p-1} x^{p-1} + \cdots \theta_1 x^1 + \theta_0 \tag{4.4.10}$$

The following diagram shows the entire circuit:

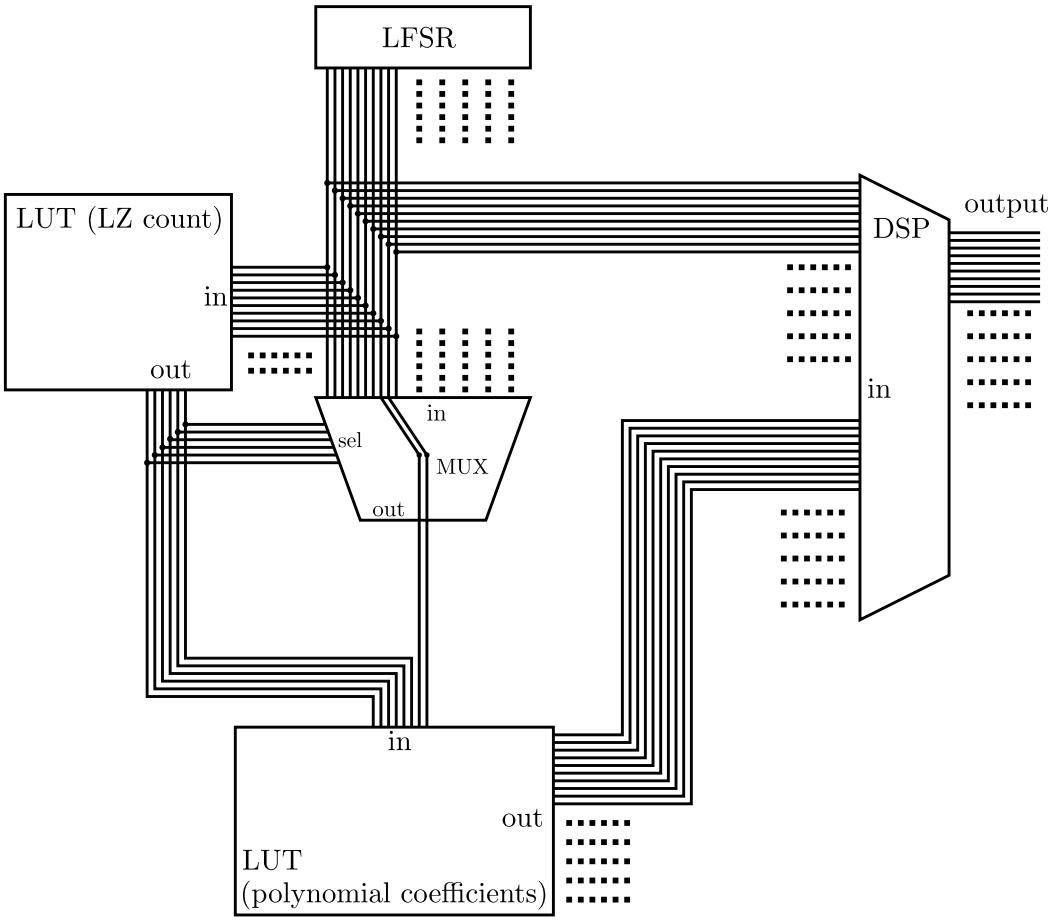


Figure 4.4.4: Circuit diagram for the generation of normally distributed random numbers

This diagram shows an example where entire range of the input of 64-bits is split up into 64 parts logarithmically (powers of two). It takes 6-bits to represent 64 leading zeros. Each logarithmic section is divided into 4 linear sections and so it takes 2 bits to represent these sections. This is reflected in diagram 4.4.4, the leading zeros counter LUT feeds the result of the count to a multiplexer which selects the two bits following the leading one. These bits as well as the amount of leading zeros is interpreted using LUT which outputs the corresponding polynomial. Finally a DSP or a set of DSP-slices evaluates the polynomial using the polynomial

coefficients and the value generated by the LFSR. In case the value of the x is larger than 0.5, x is set to $1-x$ and a flag is set to turn the output of the DSP negative.

4.4.4. LWE error vector generation

To produce an error vector requires the sampling of the χ distribution, which can be done by generating normally distributed numbers and mapping those to $\mathbb{Z}_q$. To generate a single entry, take equation 2.4.6 from section 2.4:

$$e = \lceil q \cdot (x \bmod 1) \rfloor$$

Where x is a normally distributed random number. Using an actual modulo function would be problematic as it wouldn't have a deterministic execution time. Luckily, the second operand is a power of 2. This means the operation can be simplified. Take a first operand x which is either an integer of fixed point number and second operand $y = 2^n$. Then if $z = x \bmod y$, z can be determined by taking the first $(n-1)$-bits and interpret the value as unsigned.

$$\begin{aligned} -13_{10} &= 11110011_2 \\ -13_{10} \bmod 2 = 1_{10} &= 00000001_2 \rightarrow 11110011_2 \\ -13_{10} \bmod 4 = 3_{10} &= 00000011_2 \rightarrow 11110011_2 \end{aligned} \tag{4.4.11}$$

Note that $a/2^n$ can be rewritten as $a \gg n$, given:

$$a/2^n = \left(\sum_{i=0}^{\infty} 2^i a^{[i]}\right)/2^n = \sum_{i=0}^{\infty} 2^{-n} \cdot 2^i a^{[i]} = \sum_{i=0}^{\infty} 2^{i-n} a^{[i]} \tag{4.4.12}$$

Now take an integer $a \in \mathbb{Z}$ and $b = 2^n$ where $n \in \mathbb{N}$ and $a \geq 0$, then

$$a \bmod b = a - \left(b \cdot \left\lfloor \frac{a}{b} \right\rfloor\right) = a - \left(2^n \cdot \left\lfloor \frac{a}{2^b} \right\rfloor\right) = a - (2^n \cdot (a \ll n)) = a - ((a \ll n) \gg n) \tag{4.4.13}$$

The operation $((a \ll n) \gg n)$ leaves the number a', which is a where the first n bits are set to zero. Subtracting a' from a leaves the first n-bits of a. This means that $a \bmod b$ can be performed by selecting the first n-bits of a and setting the rest to zero.
Of course, 2's complement will be used to represent negative numbers. To prove the modulus operator can be reduced to a selection even in the case $a < 0$, take $\bar{a} = -a$:

$$\begin{aligned} a - \left(b \cdot \left\lfloor \frac{a}{b} \right\rfloor\right) = -\bar{a} - \left(b \cdot \left\lfloor \frac{-\bar{a}}{b} \right\rfloor\right) = -\bar{a} + b \cdot \left(1 + \left\lfloor \frac{\bar{a}}{b} \right\rfloor\right) = \\ -\bar{a} + b + b \cdot \left\lfloor \frac{\bar{a}}{b} \right\rfloor = b - b \cdot \left(\bar{a} - \left\lfloor \frac{\bar{a}}{b} \right\rfloor\right) = b - \bar{a} \bmod b \end{aligned} \tag{4.4.14}$$

Given the absolute value of a negative number, $a \bmod b$ can be equivalently calculated of the selection of the first $(n-1)$-bits, and taking the $(n-1)$-bit 2's complement equivalent of the result. This is the equivalent of the selection of the first $(n-1)$-bits of a negative number stored in ℓ-bit 2's complement format.
The calculation of the error vector requires $x \bmod 1$. To figure out how exactly to perform this operation, a choice of representation must be made. Firstly, we need $-1 \leq x \leq 1$. Secondly, the DSP-slices that will be necessary for the evaluation will have output registers that are double the size of the input registers.
Take an LFSR size of 64-bits, then we can pose that the output represents $Q(2,63)$. The integer part is one bit larger than than in reality, this last bit is implied as it would be the sign bit, but the input of the LFSR will always be interpreted as a positive number since the input of the ICDF transformation must be between 0 and 1. Given enough segments, the approximation of the ICDF will be precise enough when the polynomials are chosen to be linear.
To get an idea of the accuracy, take the following example (performed using MATLAB): Given a linear approximation of the ICDF, split into 64 logarithmic segments (base 2) and each of those into 4 linear segments, using 64-bit floating point numbers yields an average error of $1.0642 \cdot 10^{-5}$ when compared with the built in functions from MATLAB.

The largest coefficient is around 2600 and the coefficients can also be negative. Keeping to within 64-bits of storage per coefficient, reserving one bit to represent sign, the coefficients should be stored in $Q(13,51)$ format. The largest positive and negative values that can be stored are $2^{12} - 1 = 4095$ and $-2^{12} = -4096$

respectively. Evaluating a polynomial results in a number representing $Q(13,114)$. The result will be between -1 and 1 and the error vector should contain values between $-q$ and q. Though the chance of any value reaching this bound should be exceedingly small, therefore we can convert the ICDF result to $Q(0,64)$ by selecting only the relevant portion.
The final step is to multiply the ICDF result by q and rounding the result. Value q is an integer of size $2^{64}-1$, therefore the result of the multiplication represents $Q(64,64)$. The rounding step can be performed by checking the first fractional bit. If it it 1, the the first digit is $2^{-1} = 0.5$, thus following the tie to even convention, add 1_{10} to the result and then trim it to $Q(64,0)$. Otherwise trim the result to $Q(64,0)$ with no alteration.

4.5. Matrix and vector arithmetic

Matrix-Matrix multiplication and matrix vector multiplication is relatively straight forward. Theoretically, all matrix vector operations could be performed in parallel, in practice however, a mid range FPGA would not have enough DSP's to do this. A fair amount of parallelisation is possible. It is possible to design parametricly. The design chosen for matrix and vector arithmetic has two parameters that can be changed to suit the target platform, the amount of row and partitions that are processed at a time. The parameters can be set such that multiple rows and/or columns are processed at a time (see eq. 4.5.1), or a single row can be split into partitions that are handled one each at a time (see eq. 4.5.2).

$$\begin{bmatrix} 0 & 9 & 3 & 3 \\ 8 & 4 & 0 & 8 \\ 10 & 2 & 4 & 8 \\ 1 & 6 & 0 & 8 \end{bmatrix} \begin{bmatrix} 5 & 10 & 8 & 4 \\ 3 & 7 & 10 & 4 \\ 8 & 3 & 9 & 8 \\ 9 & 3 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 78 & 81 \\ 124 & 132 \\ & \\ & \end{bmatrix} \tag{4.5.1}$$

4.1: Illustration of matrix multiplication where multiple rows and columns are evaluated per clock cycle

$$\begin{bmatrix} 0 & 9 & 3 & 3 \\ 8 & 4 & 0 & 8 \\ 10 & 2 & 4 & 8 \\ 1 & 6 & 0 & 8 \end{bmatrix} \begin{bmatrix} 5 & 10 & 8 & 4 \\ 3 & 7 & 10 & 4 \\ 8 & 3 & 9 & 8 \\ 9 & 3 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 27 \\ \\ \\ \end{bmatrix}$$

$$\downarrow \tag{4.5.2}$$

$$\begin{bmatrix} 0 & 9 & 3 & 3 \\ 8 & 4 & 0 & 8 \\ 10 & 2 & 4 & 8 \\ 1 & 6 & 0 & 8 \end{bmatrix} \begin{bmatrix} 5 & 10 & 8 & 4 \\ 3 & 7 & 10 & 4 \\ 8 & 3 & 9 & 8 \\ 9 & 3 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 78 \\ \\ \\ \end{bmatrix}$$

4.2: Illustration of matrix multiplication where half of a row is evaluated per clock cycle

When given the start signal, the circuit will start incrementing counters for each row, column and partition, performing the corresponding multiplications and additions corresponding to the current count. When the final count is reached for each counter, the circuit will set its "done flag" to high for a single clock cycle to signal it is done and the result has been prepared to be read out. At any time, when the start signal is set to high the counters will be set to 0 and the process will start again. Even when already in operation, the start signal will reset the process. Only when the final count is reached will the output of the arithmetic unit be updated.

4.6. Timing

All the different circuits that make up the entire system have to communicate when to start and when they are done. Some systems only take one clock cycle to complete, which mean they do not need to signal when they are done. However circuits such as the matrix/vector arithmetic circuits require multiple cycles to complete. To ensure proper timing, all circuits that require multiple cycles have an internal counter and an operation flag which is wired into a comparator type circuit. This circuit is called the state debounce comparitor.

The debounce comparator takes an input and sets the output to high only if the input has been low previously.

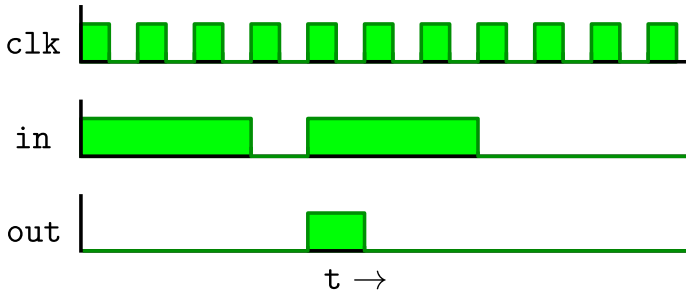


Figure 4.6.1: Timing diagram showing the input-output relation of a comparator type circuit

The operation flag should be set to low when in operation and high when it is not. The debounce comparator output is used to operate the done flag. The done flag will only be high for a single clock cycle ensuring predictable behaviour. As stated in the previous section, if the start signal of the matrix-vector arithmetic is set to high, circuit will reset. If the start signal is set to high for more than a single clock cycle, the arithmetic circuit will reset until the signal is set to low. The circuit supplying the input might switch to another task when it is done with the previous. This may mean that the input can only be read for a few cycles or just one before changing. In that case the input will be read incorrectly, which will lead to unpredictable behaviour if there is some timing mismatch.
A circuit such as matrix-vector a arithmetic circuit keeps track of its state with counters. An alternative way to track progress might be a state machine. In that case the mode of operation can be anything. So long as it has some final state where a done signal is set to high which is read by a debounce comparitor, correct timing can be ensured.

4.7. Plant interface and controller implementation

Having established the implementation of all fundamental elements of the final system, the plant interface and controller implementation as shown in figure 4.2.1 can now be explored in more detail.
Firstly, to prevent the diagrams from becoming cluttered some functionality table 4.7.1.

symbol	name	description
	task manager port	Every circuit that requires careful timing is connected to the task manager through this port. Instead of a line, this symbol indicates that the circuit is connected to the task manager.
	right hand bit shift	These circuits bit shift the incoming signal by a pre-programmed amount.
	Hold circuit	This circuit receives the control effort for the next time step and holds the control effort to be released at the right time.

Table 4.7.1: Table of all symbols used in the plant interface and controller diagrams

During operation, the plant interface utilises counters to time when to measure sensor data and when to push control effort. As stated earlier in section 4.6, calculation of the control effort takes time and thus the future control effort is calculated at each current time step. The control effort is applied at the same clock tick that

the sensor data is read, encrypted and then sent to the controller. The controller calculates the next observer states and control effort utilising the encrypted control effort and observer state from previous time steps. The calculated control effort and observer states are then sent back to the plant interface for decryption, application and re-encryption.
The inner workings of the pant interface is shown in detail in figure 4.7.1.

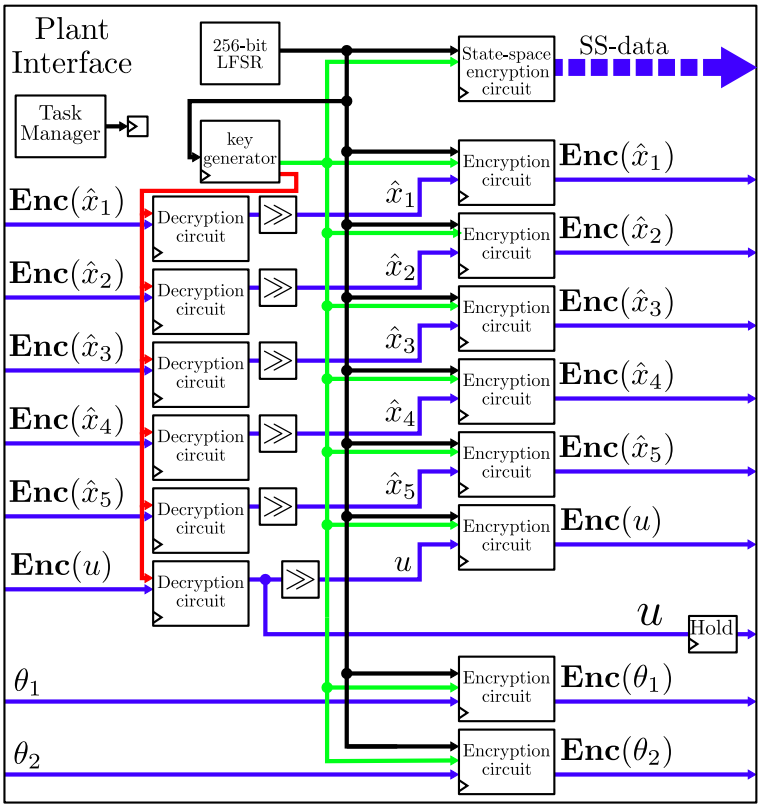


Figure 4.7.1: Detailed view of the plant interface circuit. The blue lines represent data lines. The green line and red line represent the public-key and private-key signal lines respectively. The black line is the 256-bit line which transmit uniformly distributed numbers. The SS-data line is the bus connection transmitting all state-space matrices, observer gains and controller gains.

The task manager turns on the key generator at boot up. The done flag signal line is connected to the State-space encryption circuit. The circuit encrypts the state-space matrices, controller gains and observer gains. These signals are summarised by thick blue dashed arrow with the label "SS-data". When the SS-data has been encrypted and transmitted, the Task manager starts the timer for nominal operation. At a set interval the decryption circuits are triggered. Note that the controller assumes that the new observer states and control effort are supplied before the end of one controller time-step. The decryption and encryption circuits are connected such that the first decryption triggers a cascade. First one decryption circuit is given the start signal by the task manager. The done flag of this decryption unit is connected to the start signal input of the encryption circuit that follows. When an Encryption circuit is encrypting the input it requests random numbers from the LFSR-circuit. When enough random numbers have been supplied, the encryption circuit signals the next decryption unit, to start decrypting and encrypting the next oberver state or control effort. This is repeated until all inputs have been decrypted and encrypted again. When all re-encryption has been performed, the data is sent.
As discussed earlier in section 4.2, multiplication using fixed point numbers causes the number of fractional bits to grow and on top of that, FHE has limited multiplicative depth. To deal with this issue, a solution is to send the observer state and control effort back to the plant, where they can be decrypted, truncated and then re-encrypted, after which they are sent to the controller along with the new measurements. Besides this, it is necessary to perform sign extension of the state space and gain matrices to ensure validity of all computations.
When the controller is done calculating, it sends the plant interface the control effort of the next time step

and so the control effort is held until the current control cycle ends (see the ⟦Hold⟧-symbol).
Finally, figure 4.7.2 shows the controller circuit. When the controller unit has received the SS-data at boot up, the controller is ready for operation. When it receives the encrypted observer states and control-effort the calculation of the following observer state and control effort commences. Not all calculations can be performed in parallel, therefore the controller has two arithmetic circuits available to for matrix vector arithmetic. Each multiplication or addition of two values is performed through matrix multiplication and addition of the corresponding ciphers. A memory unit contains all SS-data and the calculation results. The memory unit supplies the inputs to the addition and multiplication unit to cycle through all the necessary calculations to perform the matrix vector multiplications of the underlying values. When the memory unit receives a start signal, the matrix multiplication unit start too for the first calculation. The initial instruction is performed as the counter is set to 0. When a calculation is done, the counter is incremented and the flag signal is carried to either the multiplication circuit or the addition unit depending on the counter state. Depending on the counter state, the memory unit will pass the signal to either arithmetic circuit. After the final value of the counter, the counter will reset to zero and the new observer states and control effort are sent back to the plant interface.

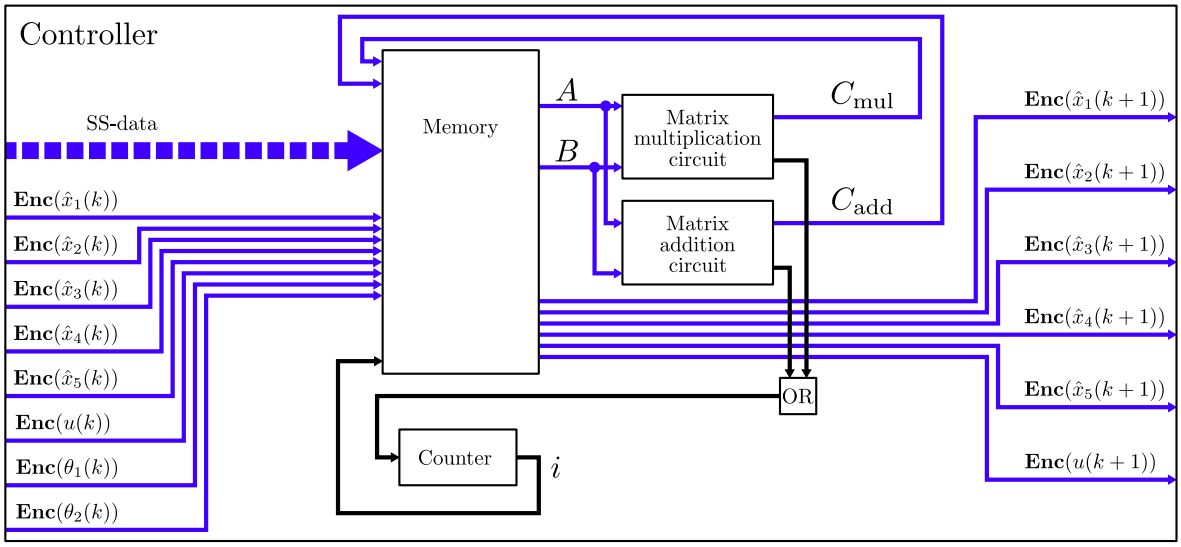


Figure 4.7.2: Detailed view of the controller circuit. Blue lines represent data lines. The black lines coming from the matrix multiplication and addition circuits are flags that signal that a matrix multiplication or addition has completed. The OR-block drives a counter so that when either matrix multiplication or addition has completed, the counter is incremented. The count is read by the memory unit so that it passes the correct matrices to the matrix multiplication and addition units.

5

Results

This chapter will present an implementation of the control system as discussed in previous examples, providing all design parameters. This chapter will start of with section 5.1 which will go into detail on the model that will be used as a use-case as well as the configuration of the controller. Section 5.1.1 describes the physical model of a double pendulum followed by section 5.1.2 which demonstrates tests the observability and controllability of the system. Section 5.1.3 discusses the selection of a suitable sampling frequency for control. Section 5.1.4 describes the requirements on the fixed point representation derived from the controller topology as introduced earlier in section 4.1. Section 5.1.5 covers the tuning of the controller parameters. Section 5.2 presents plots to show the performance of the system for a couple different sampling times. The final section of the chapter, section 5.3 will discuss the hardware requirements for given security parameters, sampling times and the amount mathematical operations necessary for the operation of a controller. This will enable the reader to determine the hardware requirements for their controller of choice. Section 5.3 will also provide the reader with an example, specifying the hardware requirements for a given implementation of the controller setup as discussed in section 5.1.

5.1. Pendulum

The system that will be used to implement the control system on will be an inverted double pendulum. A double pendulum is a nonlinear system that requires a controller that operates at a high sampling rate. This makes it a good system to analyse whether the encryption is fast enough to enable real-time control. The following section will describe the model of the double pendulum.

5.1.1. Model

The pendulum consists of two links, each a length of ℓ_1 and ℓ_2. The position of each link is described by two angles, θ_1 and θ_2, see figure 5.1.1.

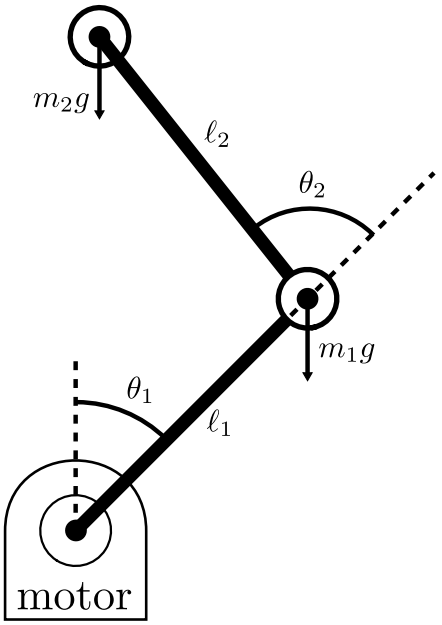


Figure 5.1.1: Drawing of pendulum model

Angles θ_1 and θ_2 can be measured with sensors in the joints. The dynamics of the system are the following:

$$
\begin{aligned}
&\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \\
&\begin{cases} M(\theta)\ddot{\theta} + C(\theta,\dot{\theta})\dot{\theta} + G(\theta) = [T \quad 0]^\top, \\ T + \tau_e \dot{T} = k_m u \end{cases} \\
&M(\theta) = \begin{bmatrix} P_1 + P_2 + 2P_3\cos\theta_2 & P_2 + P_3\cos\theta_2 \\ P_2 + P_3\cos\theta_2 & P_2 \end{bmatrix} \\
&C(\theta,\dot{\theta}) = \begin{bmatrix} b_1 - P_3\dot{\theta}_2\sin\theta_2 & -P_3(\dot{\theta}_1 + \dot{\theta}_2)\sin\theta_2 \\ P_3\dot{\theta}_1\sin\theta_2 & b_2 \end{bmatrix} \\
&G(\theta) = \begin{bmatrix} -g_1\sin\theta_1 - g_2\sin(\theta_1+\theta_2) \\ -g_2\sin(\theta_1+\theta_2) \end{bmatrix} \\
&P_1 = m_1c_1^2 + m_2l_1^2 + I_1 \quad P_2 = m_2c_2^2 + I_2 \\
&P_3 = m_2l_1c_2 \\
&g_1 = (m_1c_1 + m_2l_1)g \qquad g_2 = m_2c_2g
\end{aligned}
\tag{5.1.1}
$$

In this model, g is the gravitational constant on earth, c is the centre of mass of a link, I is the inertia of a link and b the damping coefficients in the two joints. Control effort u can range between -1 and 1 and the maximum torque of the motor is k_m. Finally, τ_e is the time constant of the motor. See the following table for the values of each parameter that reflect a real setup (double pendulums found in the TU Delft, 3me, DCSC lab):

Parameter	Value	Parameter	Value
m_1	$0.125\ kg$	m_2	$0.05\ kg$
l_1	$0.1\ m$	l_2	$0.1\ m$
c_1	$-0.04\ m$	c_2	$0.06\ m$
I_1	$0.074\ kgm^2$	I_2	$0.00012\ kgm^2$
b_1	$4.8\ kgs^{-1}$	b_2	$0.0002\ kgs^{-1}$
k_m	$50\ Nm$	τ_e	$0.03\ s$
g	$9.81ms^{-2}$		

Table 5.1.1: Model and encryption parameters

5.1.2. Stability

The upright position of the pendulum is unstable, but is stabilisable with appropriate control. A viable approach is discrete time control. To do so, the system must first be linearised around the target equilibrium, which is at $\theta_1 = 0$, $\theta_2 = 0$. First, isolate $\ddot{\theta}_1$ and $\ddot{\theta}_2$ from equation 5.1.1:

$$\begin{aligned}\ddot{\theta}_1 = -\frac{1}{P_3^2\cos(\theta_2)^2 - P_1P_2}(&\cos(\theta_2)\sin(\theta_2)P_3^2\dot{\theta}_1^2 + P_2 * \sin(\theta_2)P_3\dot{\theta}_1^2 + 2P_2\sin(\theta_2)P_3 * \dot{\theta}_1\dot{\theta}_2 + \\ &P_2 * \sin(\theta_2)P_3\dot{\theta}_2^2 + b_2\cos(\theta_2)P_3\dot{\theta}_2 - g_2\sin(\theta_1+\theta_2)\cos(\theta_2)P_3 - \\ &P_2b_1\dot{\theta}_1 + P_2b_2\dot{\theta}_2 + P_2T + P_2g_1\sin(\theta_1))\end{aligned} \tag{5.1.2}$$

$$\begin{aligned}\ddot{\theta}_2 = \frac{1}{P_3^2\cos(\theta_2)^2 - P_1 * P_2}(&P_2T - P_2b_1\dot{\theta}_1 + P_1b_2\dot{\theta}_2 + P_2b_2\dot{\theta}_2 - P_1g_2\sin(\theta_1+\theta_2) + \\ &P_3T\cos(\theta_2) + P_2g_1\sin(\theta_1) - P_3b_1\dot{\theta}_1\cos(\theta_2) + 2P_3b_2\dot{\theta}_2\cos(\theta_2) + \\ &2P_3^2\dot{\theta}_1^2\cos(\theta_2)\sin(\theta_2) + P_3^2\dot{\theta}_2^2\cos(\theta_2)\sin(\theta_2) - P_3g_2\sin(\theta_1+\theta_2)\cos(\theta_2) + \\ &P_1P_3\dot{\theta}_1^2\sin(\theta_2) + P_2P_3\dot{\theta}_1^2\sin(\theta_2) + P_2P_3\dot{\theta}_2^2\sin(\theta_2) + P_3g_1\cos(\theta_2)\sin(\theta_1) + \\ &2P_3^2\dot{\theta}_1\dot{\theta}_2\cos(\theta_2)\sin(\theta_2) + 2P_2P_3\dot{\theta}_1\dot{\theta}2\sin(\theta_2))\end{aligned} \tag{5.1.3}$$

Define a state vector x and a nonlinear function which calculates the derivative of x:

$$x = \begin{bmatrix}\theta_1\\ \dot{\theta}_1\\ \theta_2\\ \dot{\theta}_2\\ T\end{bmatrix} \qquad \dot{x} = f(\theta_1,\dot{\theta}_1,\theta_2,\dot{\theta}_2,T,u) = f(x,u) = \begin{bmatrix}\dot{\theta}_1\\ \ddot{\theta}_1\\ \dot{\theta}_2\\ \ddot{\theta}_2\\ \dot{T}\end{bmatrix} \tag{5.1.4}$$

We can then derive state space matrices by determining the jacobian of function $f(x)$ with respect to x and u:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{5.1.5}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad D = 0$$

$$\frac{\partial f(x,u)}{\partial x} = A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ a_1 & a_2 & a_3 & a_4 & a_5 \\ 0 & 0 & 0 & 1 & 0 \\ a_6 & a_7 & a_8 & a_9 & a_{10} \\ 0 & 0 & 0 & 0 & -\frac{1}{\tau_2} \end{bmatrix} \quad \frac{\partial f(x,u)}{\partial u} = B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{k_m}{\tau_e} \end{bmatrix} \tag{5.1.6}$$

$$\begin{aligned} a_1 &= \frac{P_3 g_2 - P_2 g_1}{(P_3^2 - P_1 P_2} \qquad a_2 = \frac{P_2 b_1}{P_3^2 - P_1 P_2} \quad a_3 = \frac{P_3 g_2}{P_3^2 - P_1 P_2} \\ a_4 &= -\frac{P_2 b_2 + P_3 b_2}{P_3^2 - P_1 P_2} \qquad a_5 = \frac{P_2}{P_3^2 - P_1 P_2} \quad a_6 = -\frac{P_1 g_2 - P_2 g_1 - P_3 g_1 + P_3 g_2}{P_3^2 - P_1 P_2} \\ a_7 &= -\frac{P_2 b_1 + P_3 b_1}{P_3^2 - P_1 P_2} \quad a_8 = -\frac{P_1 g_2 + P_3 g_2}{P_3^2 - P_1 P_2} \quad a_9 = \frac{P_1 b_2 + P_2 b_2 + 2P_3 b_2}{P_3^2 - P_1 P_2} \\ a_{10} &= \frac{P_2 + P_3}{P_3^2 - P_1 P_2} \end{aligned} \tag{5.1.7}$$

Now we must investigate controlability and observability of the system. The linearisation of the sytem is LTI, therefore controlability and observability can be tested using the Hautus tests (see *Feedback systems: An introduction for scientists and Engineers*[45] for more information). The linearised has five unique eigenvalues and eigenvetors. Two modes of the system are unstable (two positive eigenvalues). Now take matrices:

$$\begin{aligned} &[\lambda I - A, B] \\ &[\lambda I - A; C] \end{aligned} \tag{5.1.8}$$

Both matrices are full rank for each eigen value λ of A, therefore the system is both controllable an observable (See section A.1 in the appendix for the numerical calculations). Given that the nonlinear is continuously differentiable and time independent, the nonlinear system is locally controllable and locally observable. We know now that the system can be stabilised given the control input and sensor data that is available. However there is one last complication. Discrete control will be used, which means that the criteria mentioned above may not hold if sampling times are not small enough. given infinite sampling time.
Firstly, take the discretisation of matrices A and B (see *Computer controlled systems* [44] for more information):

$$\begin{aligned} A_d &= e^{Ah} \\ B_d &= \int_0^h e^{As} ds \cdot B \end{aligned} \tag{5.1.9}$$

where h denotes sampling time. Matrix C does not have to be discretised. To determine the controlability of the system take the controllability matrix (W_c) and observability matrix (W_o) [44]:

$$\begin{aligned} W_c &= \begin{bmatrix} B_d & A_d B_d & {A_d}^2 B_d & {A_d}^3 B_d & {A_d}^4 B_d \end{bmatrix} \\ W_o &= \begin{bmatrix} C \\ CA_d \\ CA_d{}^2 \\ CA_d{}^3 \\ CA_d{}^4 \end{bmatrix} \end{aligned} \tag{5.1.10}$$

Both are full rank, which means that the discrete system too, is controllable and observable.

5.1.3. Sampling time

Before moving on to the controller design, sampling time has to be selected to ensure that the controller is able to stabilise the system successfully. To reconstruct a continuous time signal correctly, according to Nyquist's sampling theorem, the sampling frequency has to be at least twice the highest frequency that the system exhibits [44]. This is necessary, but not sufficient condition. A rule of thumb that is commonly used is to set the sampling time to be at least 30 times the bandwidth of the target system [6].

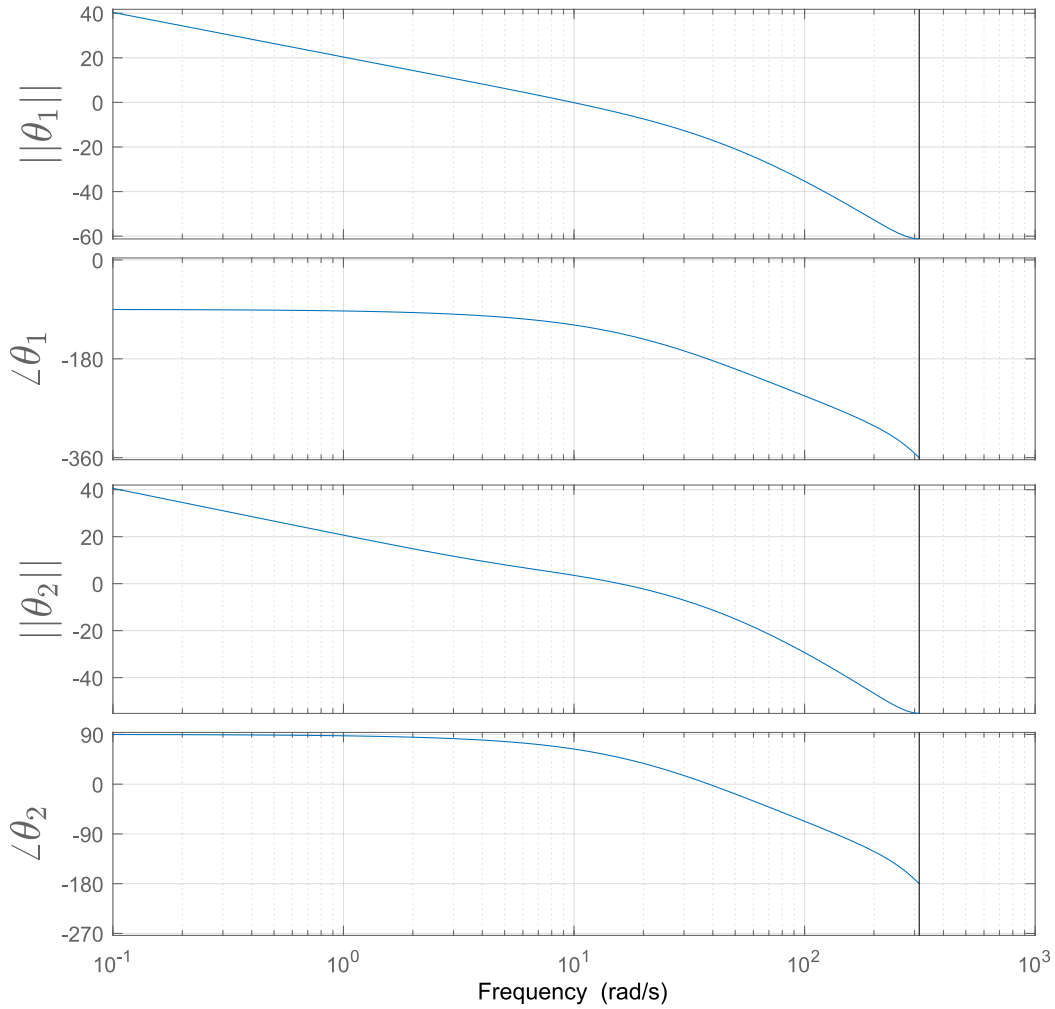


Figure 5.1.2: Bode diagram of θ_1 and θ_2

The bode diagram in figure 5.1.2 shows that the fastest frequency in the measurement data (crossover frequency) is that of θ_2 at around 16 *rad/s* which is a frequency of 2.55 Hz. According to the rule of thumb as stated earlier, the sampling frequency should be at least $2.55 \cdot 30 \approx 76$ Hz. For good measure, the sampling frequency is chosen to be 100 Hz.

5.1.4. Encrypted state space computation

This section will demonstrate how to determine what fractional bit representation will have to be chosen for each signal, gain and state-space matrix. Take the controller from 4.1.3. It is optimised to require the least amount of multiplications.

$$\begin{cases} \hat{\mathbf{x}}(k+1) = \mathbf{A_d} \odot \hat{\mathbf{x}}(k) \oplus \mathbf{B_d} \odot \mathbf{u}(k) + \bar{\mathbf{L}} \odot (\bar{\mathbf{y}}(k) \oplus \mathbf{C} \odot \hat{\mathbf{x}}(k)) \\ \mathbf{u}(k+1) = \bar{\mathbf{K}} \odot \hat{\mathbf{x}}(k+1) \end{cases}$$

To provide consistency, all state space constants are set to represent the same amount of fractional bits ε and integer bits δ. In short, the state-state matrices and gains are represented as $Q(\delta,\varepsilon)$. As an exception, only state-space matrix $\mathbf{C}$ should be set to $Q(\delta,0)$. The reason for this will become clear shortly.
The observer requires are two homomorphic multiplications to evaluate

$$\bar{\mathbf{L}} \odot (\bar{\mathbf{y}}(k) \oplus \mathbf{C} \odot \hat{\mathbf{x}}(k)) \tag{5.1.11}$$

And one homomorphic multiplication to evaluate each partial product

$$\mathbf{A_d} \odot \hat{\mathbf{x}}(k) \tag{5.1.12}$$

and

$$\mathbf{B_d} \odot \mathbf{u}(k) \tag{5.1.13}$$

If partial products $\bar{\mathbf{y}}(k)$ and $\mathbf{C} \odot \hat{\mathbf{x}}(k)$ were to have different amounts of fractional bits, it would not be possible to correctly add them up without manipulating them first. Matrix $\mathbf{C}$ contains no fractional bits and only contains ones and zeros and so $\mathbf{C} \odot \hat{\mathbf{x}}(k)$ does not change the representation of $\hat{\mathbf{x}}(k)$. If signals $\hat{\mathbf{x}}(k)$ and $\bar{\mathbf{y}}(k)$ are represented as $Q(\delta,\varepsilon)$, then partial product 5.1.11 is represented as $Q(\delta, 2\cdot\varepsilon)$. Signal $\mathbf{u}(k)$ should also be set to $Q(\delta,\varepsilon)$ during re-encryption so that 5.1.13 is represented as $Q(\delta, 2\cdot\varepsilon)$. The addition of products 5.1.11, 5.1.12 and 5.1.13 will then be represented as $Q(\delta, 2\cdot\varepsilon)$ (addition does not induce a shift of the decimal point). The next control effort $\mathbf{u}(k+1)$ will then be represented as $Q(\delta, 3\cdot\varepsilon)$. To ensure that $Q(\delta, 3\cdot\varepsilon)$ is supported by the Gentry scheme [22], the register size should be set to

$$\ell = \delta + 3\cdot\varepsilon \tag{5.1.14}$$

Where δ and ε should be set such that the observer is precise enough whilst allowing for large enough integers. Note that only signal $\mathbf{u}(k+1)$ has $3\cdot\varepsilon$ fractional bits. If the control effort is smaller in size than any of the other signals, one could opt for a smaller register size. In the case of the system presented in 5.1, the control effort remains between 0 and 1, whereas the observer states can take on values at around 3/4. From testing, it appears the control effort requires around 22 fractional bits to for the system to run smoothly. The control effort remains between 0 and 1 and so requires only one integer bit. Therefore the register size can be set to be $\ell = 2 + 3\cdot\varepsilon = 2 + 3\cdot 22 = 68$ bits. This means that $\hat{\mathbf{x}}(k+1)$ will be represented as $Q(\ell - 2\cdot\varepsilon, 2\cdot\varepsilon) = Q(24,44)$ which should provide more than enough precision and space for the integer part. Finally, fractional bit representation utilises 2's complement convention to represent sign. After multiplication, the decimal point shifts and therefore the sign bit does too. To ensure correct sign changes, sign extension of the state-space matrices and gains is required. To perform sign extension, simply take the sign bit of a given value with representation $Q(\delta,\varepsilon)$ and convert it to $Q(\ell,\varepsilon)$ by setting the remaining most significant bit to the value of the sign bit.

5.1.5. Tuning

To place the poles of the closed loop system, the gains can be generated using discrete LQ-optimal control (see *Computer controlled systems* [44] for more information). The poles of the observer can be hand tuned, as it is less sensitive.
Given the state space discription of the linear system:

$$x(k+1) = A_d x(k) + B_d u(k) u(k) = -Kx(k)$$

Take the cost function:

$$J = \sum_{k=0}^{\infty} (x(k)^\top x(k) + u(k)^\top R u(k) + 2x(k)^\top N u(k)) \tag{5.1.15}$$

Using the control law $u(k) = -Kx(k)$, the optimal trajectory can be reached by setting gain matrix K to:

$$K = (R + B_d^\top P B_d)^{-1} (B_d^\top P A_d + N^\top) \tag{5.1.16}$$

where P is the solution to the algebraic Riccati equation:

$$P = A_d^\top P A_d - (A_d^\top B_d + N)(R + B_d^\top P B_d)^{-1} (B_d^\top P A_d + N^\top) + Q \tag{5.1.17}$$

To find the optimal gain, matrix Q has been set to:

$$Q = \frac{1}{10}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.1.18}$$

and $R = 1$ and $N = 0$. Matrix Q only affects θ_1 and θ_2 as those are the only state variables that should be prioritised in terms of returning to zero. Variable R has been made larger in proportion to ensure that the pendulum is operated smoothly. Variable N has been set to zero as control effort, as there is no desire for two way-coupling.

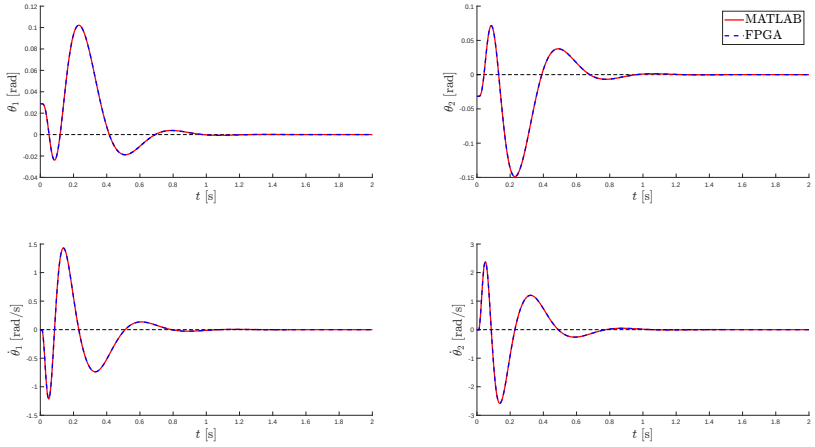
The observer gains have been tuned by hand. The poles of the observer gain have to be within the unit circle, but have been chosen to be close to one, to ensure that the observer is fast enough, but such that it converges gradually. The error dynamics $e(k)$ of the observer can be derived as follows:

$$\begin{aligned} \hat{x}(k+1) =& A_d\hat{x}(k) + B_d u(k) + L(y(k) - C\hat{x}(k)) \\ e(k+1) =& x(k+1) - \hat{x}(k+1) = \\ & A_d x(k) + B_d u(k) - (A_d\hat{x}(k) + B_d u(k) + L(y(k) - C\hat{x}(k))) = \\ & A_d(x(k) - \hat{x}(k)) - L(Cx(k) - C\hat{x}(k)) = \\ & A_d e(k) - LC(x(k) - \hat{x}(k)) = A_d e(k) - LCe(k) = \\ & (A_d - LC)e(k) \end{aligned} \tag{5.1.19}$$

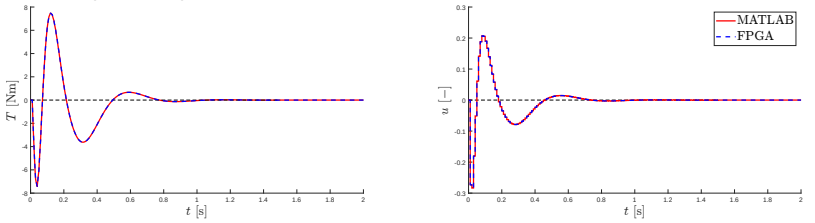
Matrix L has been set such that the poles of $(A_d - LC)$ are $[0.8, -0.8, 0.6, -0.6, 0.5]$. These gains have been tuned by hand (by comparing results) results in an observer that reacts quickly. The poles have been placed as such to ensure that the observer operates smoothly. If the poles would be chosen to be too fast, the controller would respond erratically. The performance of the tuning will be discussed in the next section.

5.2. Performance

A major benefit of using a fully homomorphic encrytion scheme is that it can be used to implement discrete time feedback controllers without the need for alteration. The resulting hardware simulation then yields exactly the same results as compared with the MATLAB simulation which does not apply any encryption. Figure 5.2.1 shows both simulation results in one set of plots (MATLAB simulation in red and FPGA hardware simulation in blue).



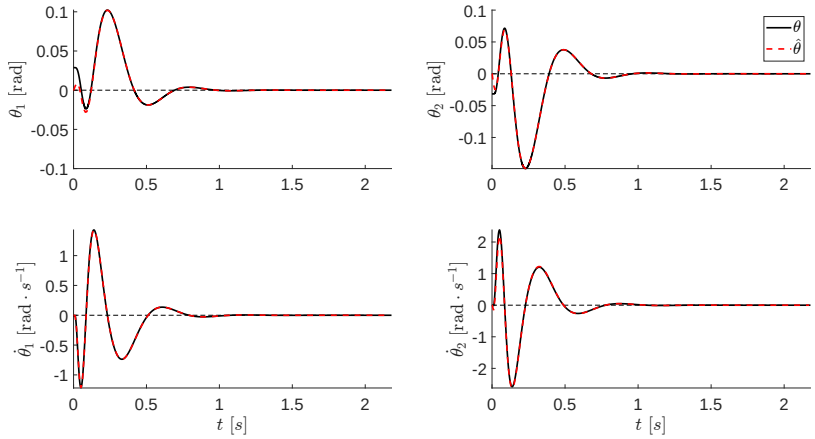
(a) Pendulum angles and angular velocities



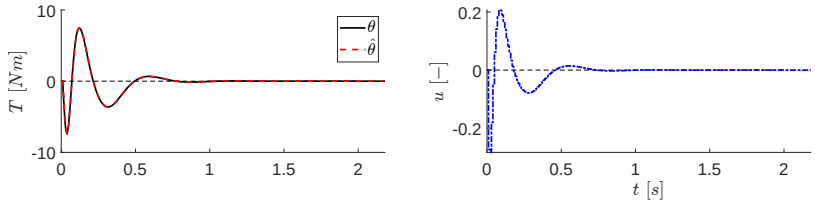
(b) Motor torque and control effort

Figure 5.2.1: Simulation results of both the MATLAB simulation (no encryption) and FPGA hardware simulation

The controller gains and observer gains from section 5.1.5 result in a smoothly operating feedback controller. Figure 5.2.2 shows the FPGA hardware simulation. Both the system states and observer states are visible. Note that the observer quickly and smoothly track the actual states.



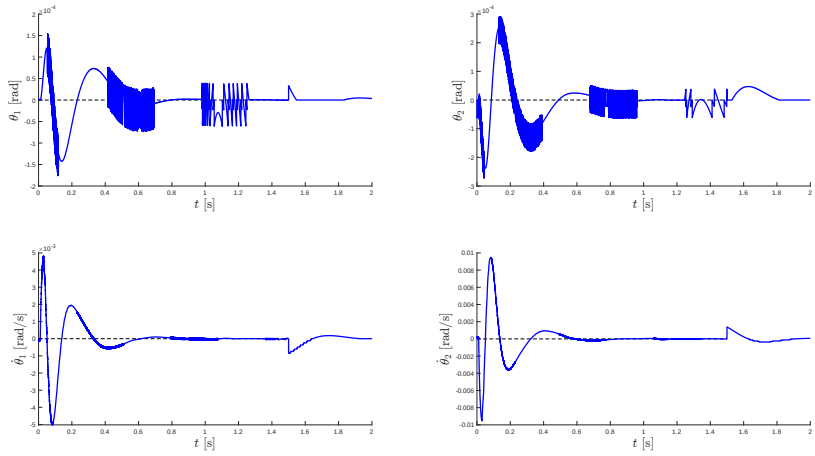
(a) Pendulum angles and angular velocities



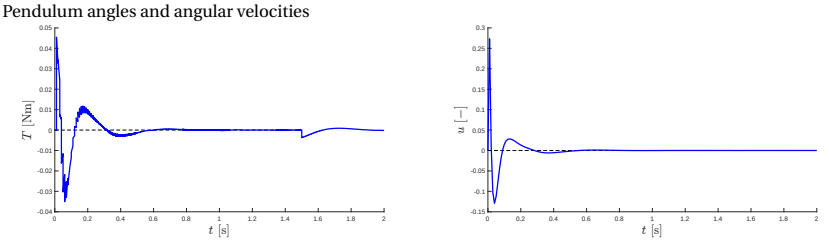
(b) Motor torque and control effort

Figure 5.2.2: Plot FPGA hardware simulation showing the system states and observer states

The reliable observer allows for swift operation of the controller, bringing the pendulum in equilibrium position with only a few direction changes. Finally, figure 5.2.3 shows a difference plot of all states, the L_1 norm of the difference of the states from the FPGA hardware simulation and the MATLAB simulation.



(a) Pendulum angles and angular velocities



(b) Motor torque and control effort

Figure 5.2.3: Plots that show the difference between the MATLAB simulation and FPGA harware simulation

The difference plots closely track scaled versions of the respective derivatives. This demonstrates that the two

simulations are almost exactly the same. Note that at some time steps there is some high frequency noise, this is likely due to rounding errors which are expected when using limited precision (which is inherent to digital computation).

5.3. Hardware utilisation

To understand the limitations of the feedback system (as illustrated in chapter 4) it is important to record the hardware utilisation of a given implementation on an FPGA. Unfortunately, recording the hardware utilisation of the entire system (both FPGA's) would be infeasible given the amount of parameters that can be tuned and the amount of modules that make up the system. To narrow down the scope, this section will consider the most critical part of the system. The plant interface is considerably less demanding with respect to hardware utilisation than the controller. The plant interface has to encrypt and decrypt the sensor data, observer states and control effort. The controller has to take the same data and perform an amount of computations that is many orders of magnitude larger than the computations performed by plant interface. Furthermore, DSP units are not needed due to the alterations made to the Gentry scheme. The hardware resource that is the limiting factor in the hardware design are Slice registers. Slice registers can store ciphers and intermediate results. Slice registers are the resource that is depleted first when synthesising a design and so slice register utilisation will be the measure used to measure the size of a given design. The utilisation of a cipher multiplication circuit and a cipher addition circuit will be explored in the following subsections.

5.3.1. Homomorphic multiplication and addition circuits

As discussed earlier in section 4.5, the calculation of the matrix multiplication of two matrices can be divided up in multiple ways (see figure 5.3.1). The user can specify how many elements in the output cipher should be calculated at a time (per clock tick).

$$\begin{bmatrix} 0 & 9 & 3 & 3 \\ 8 & 4 & 0 & 8 \\ 10 & 2 & 4 & 8 \\ 1 & 6 & 0 & 8 \end{bmatrix} \begin{bmatrix} 5 & 10 & 8 & 4 \\ 3 & 7 & 10 & 4 \\ 8 & 3 & 9 & 8 \\ 9 & 3 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 78 & 81 & & \\ 124 & 132 & & \\ & & & \\ & & & \end{bmatrix} \tag{5.3.1}$$

5.1: Illustration of a matrix multiplication where multiple elements of the result are calculated at a time.

It stands to reason that calculating more elements per clock tick requires more hardware. As discussed in section 3.2, a reduced cipher consists of $(n+1)\cdot\ell \times (n+1)$ elements that are stored in ℓ sized containers. Hence the total amount of bits that a cipher contains is $(n+1)\cdot\ell\cdot(n+1)\cdot\ell = ((n+1)\ell)^2$. To illustrate the hardware utilisation for different configurations of parallelism, figures 5.3.1 and 5.3.2 compile data from synthesised designs in two plots, one of a homomorphic multiplication circuit and the other of a homomorphic addition circuit. The blue lines represent the hardware utilisation for a given size of a cipher in bits. A cipher contains $((n+1)\ell)^2$ bits, multiple configurations of security parameter n and register size ℓ. For example, a security parameter $n = 3$ and $\ell = 16$ results in a cipher containing $((n+1)\ell)^2 = 4096$ bits. Choosing $n = 7$ and $\ell = 8$ also results in a cipher consisting $((n+1)\ell)^2 = 4096$ bits. These two configurations have the same utilisation and so figures 5.3.1 and 5.3.2 can be used to determine the utilisation of a configuration of choice. These diagrams have been generated by synthesising hardware designs on a Nexys 4 FPGA for a selection of parameter. This data has then been used to fit curves. Exact hardware utilisation will be different on different FPGA's, because the exact lay-out of hardware components will influence how a hardware design is synthesised. The green line that follows the green vertical axis on the right, is a plot of the amount of clock ticks it takes to completely compute a cipher multiplication.

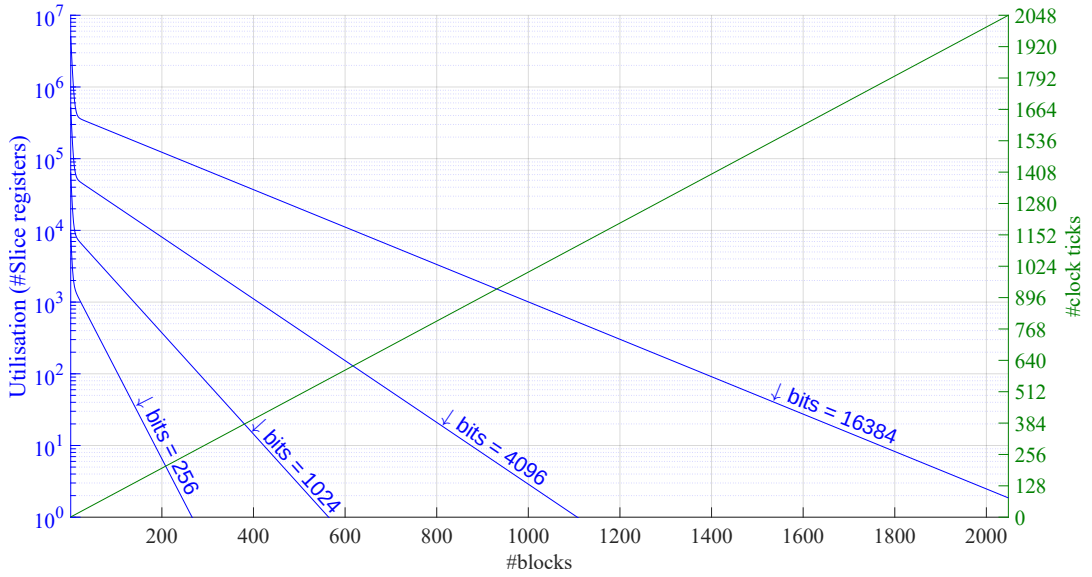


Figure 5.3.1: Utilisation plot of multiplication unit

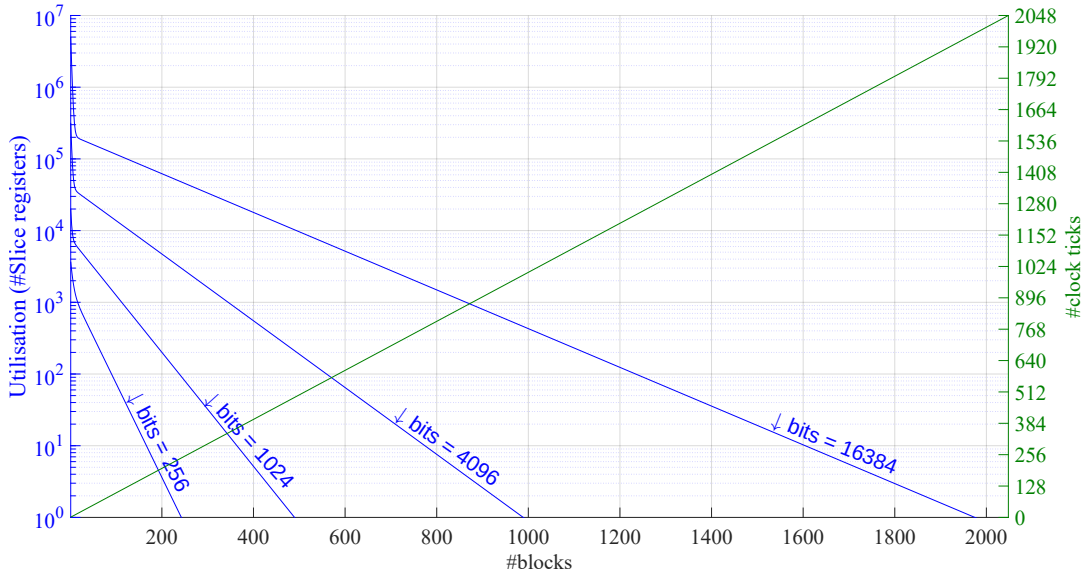


Figure 5.3.2: Utilisation plot of multiplication unit

There are many cipher sizes that could be considered that are not shown in the diagram. It is however possible to use figure 5.3.1 by interpolating between two existing utilisation lines. How to do this will be described in more detail in the next section.

5.3.2. Using utilisation plots

There are a couple ways that figures 5.3.1 and 5.3.2 can be used. A design use case could be that the size of the cipher is known. If an FPGA is already selected and the maximum allowable utilisation is known, one could use the plot to determine how long it would take to perform homomorphic multiplications.
Take figure 5.3.3. Given is that there a maximum of 10^4 slice registers are available for the multiplication circuit and security parameter is set to $n = 7$ and register size is set to $\ell = 8$ and so the cipher will consist of 4096 bits. To find the highest possible performance level, first find the intersection with the utilisation line (read the utilisation on the blue axis on the left) that matches 4096 bits (step 1). Then move down along a vertical line to find the height of the green line at that horizontal position (step 2). Now read the amount of

clock ticks it would take from the green axis on the right. Rounding up, computing the cipher multiplication would take 200 clock ticks (the cipher would be broken up in 200 blocks).

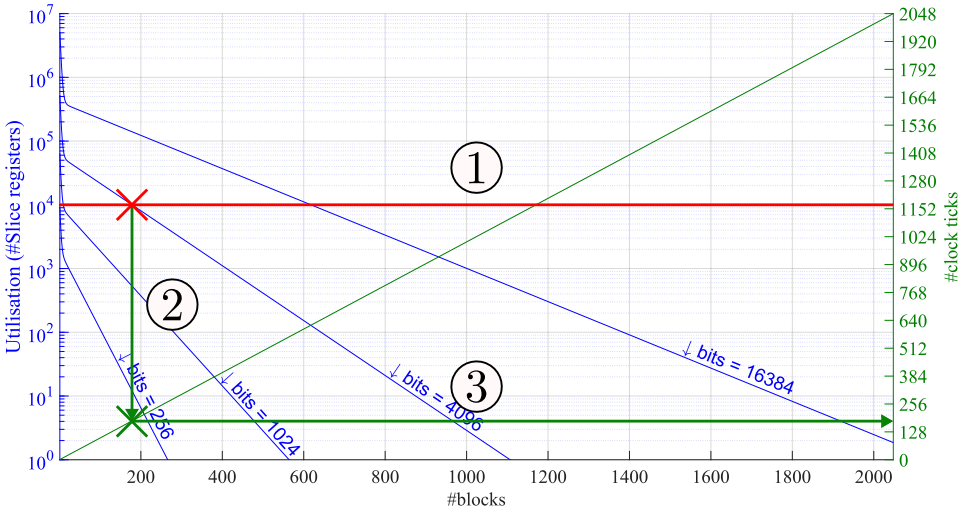


Figure 5.3.3: Utilisation plot of multiplication unit

An alternative scenario could be that the size of the ciphers is known and the necessary level of performance is known and one would like to decide what platform would be needed to meet the requirements. It is given that the cipher will be 16384 bits in size. The target sampling frequency of the system is 100 Hz and the amount of cipher multiplications that have to be computed is 1217. To reach the target performance, an FPGA would have $1/(100\,\text{Hz}\cdot 6000) = 1.6667\,\mu$s to compute one matrix multiplication. Assuming that the FPGA a clock rate of 400 MHz (a common clock rate) then the FPGA should not need more than $1/(100\,\text{Hz}\cdot 6000)\cdot 400\,\text{MHz} = 666.667$ clock ticks to calculate the result. Figure 5.3.4 illustrates how to find the minimum amount of slice registers needed for a hardware design that meets the requirements. To make the search easier, take the maximum amount of clock tick to be 640. To find the minimum utilisation in figure 5.3.4, find the 640 clock tick point on the green line (step 1). Next draw a vertical line and find the intersection of this line with the blue 16384-bits line (step 2). Finally read the utilisation at the point of intersection on the blue axis on the left hand side (step 3). A hardware design would require around 10^4 slice registers. This can then be used to determine what FPGA would have the required amount of slice registers.

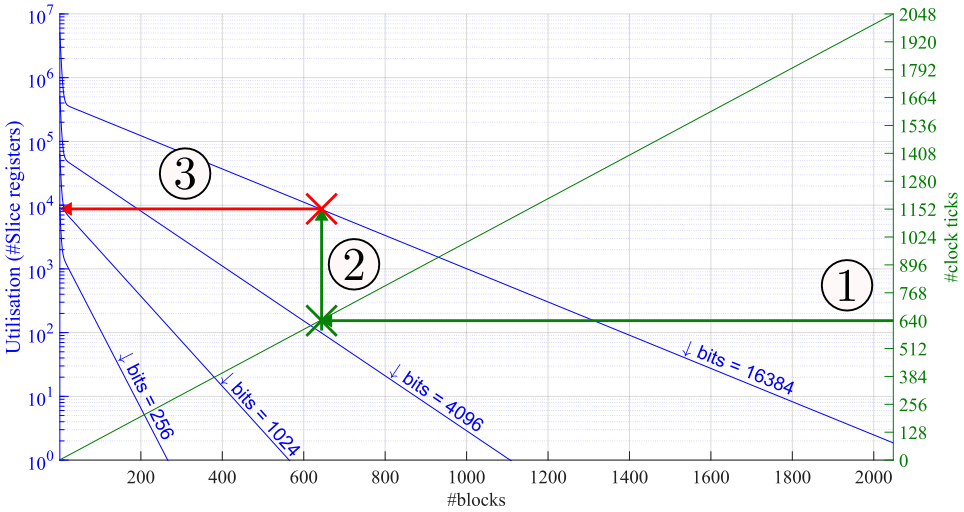


Figure 5.3.4: Utilisation plot of multiplication unit

6

Conclusion

The goal of this thesis has been to answer the research question *given a novel adaptation and implementation of homomorphic feedback control, what level of performance can currently be reached?* To answer this question, the following research sub-questions have to be answered:

- *What are the challenges of using fully homomorphic encryption in feedback control?*
- *How can the underlying mathematics of Gentry's encryption scheme be rewritten to be suitable for digital computation?*
- *What is the hardware utilisation of an implementation of feedback control using Gentry's encryption scheme on an FPGA?*

In the search of answering these research questions it has become clear that using fully homomorphic encryption in feedback control is very challenging. Both in its implementation and it computational complexity. Large problems to be overcome are limited multiplicative depth and truncation of encrypted values. Sending observer states back to the plant for re-encryption solves both problems, which requires no alteration of the controller topology. Even so, implementing feedback control using FHE is very challenging due to the many considerations. Every multiplication must be taken into account due limitations of fixed point representation. Other solutions to circumvent the problem of truncation require large alterations to the controller topology. FHE also complicates the programming of an implementation of discrete-time feedback control. Encrypted values become cipher matrices and so any single matrix or vector multiplication becomes a series of matrix-vector operations when handling encrypted data. Computational complexity of FHE, the complexity of describing systems using FHE, truncation of encrypted data and limited multiplicative depth are the main cahllenges in implementing fully homomorphic encryption in feedback control. This answers the first research sub-question

What are the challenges of using fully homomorphic encryption in feedback control?

Rewriting the Gentry encryption scheme utilising the novel notation, analytical function descriptions and the introduction of reduced ciphers has made it possible to strip encryption and homomorphic operation to their core. The rewritten encryption scheme makes it clear how the scheme can be implemented such that it suits the target hardware platform. This answers the second research sub-question.

How can the underlying mathematics of Gentry's encryption scheme be rewritten to be suitable for digital computation?

The last research sub-question to be answered is

What is the hardware utilisation of an implementation of feedback control using Gentry's encryption scheme on an FPGA?

The plots presented in section 5.3 provide an estimate for a range of parameter choices. Unfortunately synthesising hardware designs takes a long time (multiple hours to a full day for larger designs). Synthesis of a practical implementation would likely require more powerful hardware than was available during the run of

this thesis.
The adaptation of Gentry's scheme by introducing new notation and reduced ciphers simplifies the description and proofs. Analytical descriptions of the functions at the basis of the scheme paved the way towards implementing each operation using a minimal amount of computation and memory. Due to the computational complexity of FHE, previous work on FHE in feedback control has focused on systems that only require control at lower sampling rates, usually in the order of 10 Hz. If the feedback control system as proposed in this thesis is practically implementable, the resulting system would operate at 100 Hz. Unfortunately due to time constraints it has not been possible to confirm that the system can be synthesised. The software that was used to synthesise hardware, Vivado, has exhibited many bugs, which further slowed down the development. Given the data that has been collected it is possible to estimate whether a full scale hardware design can be realised using currently available harware. The Nexys 4, the FPGA used for the purpose this thesis, can run at a clock speed of 450 MHz. The controller design as proposed in section 5.1.4 requires 55 homomorphic multiplications and 12 homomorphic additions. This means that the Nexys 4, would have $1/(450\,\text{MHz}\cdot 55) = 81818$ clock ticks to finish multiplication and $1/(450\,\text{MHz}\cdot 12) = 375000$ for multiplication. This means that figures 5.3.2 and5.3.2 suggest the operations could be computed in very small blocks keeping utilisation low, whilst still finishing computation in time. This can however only be concluded for the smaller ciphers represented in these graphs.
Unfortunately this means that the final research sub-question and main research question cannot yet be answered conclusively. My main recommendation for followup research is therefore the synthesis of a full scale design. The hardware designs that have been generated for this thesis, have been synthesised automatically. Manually invoking hardware components for the some of the circuits would likely speed up synthesis and would allow the design to more fully utilise the capabilities of an FPGA.
An other avenue of research could be a more thorough exploration of the rewritten gentry scheme. This might reveal further optimisations that would decrease hardware utilisation, which would speed up compute time. Also, the rewritten scheme has been tested on an FPGA, however the implementation on a conventional computer may also feasible given the improvements.
Another important issue that requires more research is that of truncation of ciphers. Though there is an FHE scheme that allows for truncation of ciphers, it decreases the chance of successful decryption of truncated ciphers. The novel notation could potentially be used to find a more effective way of truncating ciphers. If truncation on ciphers is feasible, that would still leave the issue of multiplicative depth. Works such as [26] have suggested modelling the error introduced by FHE and incorporating it into the controller topology. Incorporating such solutions remove the need for truncation at the plant which would lower the computational burden on the plant interface.
In conclusion, the improvements made to the Gentry encryption scheme have improved performance and the implementation on a feedback controller have laid the foundation for practical implementation. This is however only the beginning. Hopefully this work will help to take a step toward the design of a viable type homomorphic encryption in feedback control.

Appendices

A

Numerical tests of feedback controller and system

A.1. Hautus tests

Using the data from tabel 5.1.1, the eigenvalues and corresponding eigenvectors of A are

$$\lambda = \begin{bmatrix} 9.58 & 0 & -10.24 & -64.53 & -33.33 \end{bmatrix}$$
$$V = \left(\begin{bmatrix} 5.03E-05 \\ 4.82E-4 \\ -1.04E-1 \\ -9.95E-1 \\ 0 \end{bmatrix}, \begin{bmatrix} 7.07E-1 \\ 0 \\ -7.07E-1 \\ -3.65E-15 \\ 0 \end{bmatrix}, \begin{bmatrix} -7.89E-05 \\ 8.08E-4 \\ -9.72E-2 \\ 9.95E-1 \\ 0 \end{bmatrix}, \begin{bmatrix} -6.80E-3 \\ 4.39E-1 \\ 1.39E-1 \\ -8.98E-1 \\ 0 \end{bmatrix}, \begin{bmatrix} -9.05E-3 \\ 3.02E-1 \\ 1.94E-2 \\ -6.47E-1 \\ 7.00E-1 \end{bmatrix} \right) \tag{A.1.1}$$

λ_1 to λ_5 represent the eigenvalues from high to low and V_1 to V_5 are the corresponding eigenvectors. The system has 3 stable modes and 2 unstable modes. Now take $[I\lambda_i - A, B]$ for all eigenvalues:

$$
\begin{aligned}
[\lambda_1 I - A, B] &= \begin{bmatrix} 9.58 & -1 & 0 & 0 & 0 & 0 \\ 0.40 & 74.10 & 0.40 & -0.0054 & -13.44 & 0 \\ 0 & 0 & 9.58 & -1 & 0 & 0 \\ -98.89 & -129.03 & -98.89 & 10.26 & 26.88 & 0 \\ 0 & 0 & 0 & 0 & 42.91 & 1666.67 \end{bmatrix} \\
[\lambda_2 I - A, B] &= \begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 \\ 0.40 & 64.52 & 0.40 & -0.0054 & -13.44 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ -98.89 & -129.03 & -98.89 & 0.68 & 26.880 & \\ 0 & 0 & 0 & 0 & 33.33 & 1666.67 \end{bmatrix} \\
[\lambda_3 I - A, B] &= \begin{bmatrix} -10.24 & -1 & 0 & 0 & 0 & 0 \\ 0.40 & 54.28 & 0.40 & -0.0054 & -13.44 & 0 \\ 0 & 0 & -10.24 & -1 & 0 & 0 \\ -98.89 & -129.03 & -98.89 & -9.56 & 26.880 & \\ 0 & 0 & 0 & 0 & 23.09 & 1666.67 \end{bmatrix} \\
[\lambda_4 I - A, B] &= \begin{bmatrix} -64.53 & -1 & 0 & 0 & 0 & 0 \\ 0.40 & -0.017 & 0.40 & -0.0054 & -13.44 & 0 \\ 0 & 0 & -64.53 & -1 & 0 & 0 \\ -98.89 & -129.03 & -98.89 & -63.85 & 26.880 & \\ 0 & 0 & 0 & 0 & -31.20 & 1666.67 \end{bmatrix} \\
[\lambda_4 I - A, B] &= \begin{bmatrix} -33.33 & -1 & 0 & 0 & 0 & 0 \\ 0.40 & 31.18 & 0.40 & -0.0054 & -13.44 & 0 \\ 0 & 0 & -33.33 & -1 & 0 & 0 \\ -98.89 & -129.03 & -98.89 & -32.65 & 26.880 & \\ 0 & 0 & 0 & 0 & 0 & 1666.67 \end{bmatrix}
\end{aligned} \tag{A.1.2}
$$

They can be row reduced to the following reduced echelon forms:

$$
\begin{aligned}
[\lambda_1 I - A, B] \rightarrow & \begin{bmatrix} 1 & 0 & 0 & 5.06E-5 & 0 & 0 \\ 0 & 1 & 0 & 4.8E-4 & 0 & 0 \\ 0 & 0 & 1 & -0.10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
[\lambda_2 I - A, B] \rightarrow & \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
[\lambda_3 I - A, B] \rightarrow & \begin{bmatrix} 1 & 0 & 0 & 7.92E-5 & 0 & 0 \\ 0 & 1 & 0 & 8.11E-4 & 0 & 0 \\ 0 & 0 & 1 & 0.10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
[\lambda_4 I - A, B] \rightarrow & \begin{bmatrix} 1 & 0 & 0 & 7.57E-3 & 0 & 0 \\ 0 & 1 & 0 & 0.489 & 0 & 0 \\ 0 & 0 & 1 & 0.015 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\
[\lambda_4 I - A, B] \rightarrow & \begin{bmatrix} 1 & 0 & 0 & 0 & 0.013 & 0 \\ 0 & 1 & 0 & 0 & -0.43 & 0 \\ 0 & 0 & 1 & 0 & -0.028 & 0 \\ 0 & 0 & 0 & 1 & 0.92 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{A.1.3}
$$

All matrices are full rank, thus the linearised system is controllable. Next investigate observability, take $[I\lambda_i - A, C]$ for all eigenvalues:

$$
\begin{aligned}
[\lambda_1 I - A; C] &= \begin{bmatrix} 9.58 & -1 & 0 & 0 & 0 \\ 0.40 & 74.10 & 0.40 & -0.0054 & -13.44 \\ 0 & 0 & 9.58 & -1 & 0 \\ -98.89 & -129.03 & -98.89 & 10.26 & 26.88 \\ 0 & 0 & 0 & 0 & 42.91 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\
[\lambda_2 I - A; C] &= \begin{bmatrix} 0 & -1 & 0 & 0 & 0 \\ 0.40 & 64.52 & 0.40 & -0.0054 & -13.44 \\ 0 & 0 & 0 & -1 & 0 \\ -98.89 & -129.03 & -98.89 & 0.68 & 26.88 \\ 0 & 0 & 0 & 0 & 33.33 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\
[\lambda_3 I - A; C] &= \begin{bmatrix} -10.24 & -1 & 0 & 0 & 0 \\ 0.40 & 54.28 & 0.40 & -0.0054 & -13.44 \\ 0 & 0 & -10.24 & -1 & 0 \\ -98.89 & -129.03 & -98.89 & -9.56 & 26.88 \\ 0 & 0 & 0 & 0 & 23.09 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\
[\lambda_4 I - A; C] &= \begin{bmatrix} -64.53 & -1 & 0 & 0 & 0 \\ 0.40 & -0.017 & 0.40 & -0.0054 & -13.44 \\ 0 & 0 & -64.53 & -1 & 0 \\ -98.89 & -129.03 & -98.89 & -63.85 & 26.88 \\ 0 & 0 & 0 & 0 & -31.20 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\
[\lambda_4 I - A; C] &= \begin{bmatrix} -33.33 & -1 & 0 & 0 & 0 \\ 0.40 & 31.18 & 0.40 & -0.0054 & -13.44 \\ 0 & 0 & -33.33 & -1 & 0 \\ -98.89 & -129.03 & -98.89 & -32.65 & 26.88 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}
\end{aligned} \tag{A.1.4}
$$

They can be row reduced to the same reduced echelon form:

$$
[\lambda I - A; C] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.1.5}
$$

All matrices $[I\lambda - A; C]$ are full rank, hence the linearised system is observable.

A.2. Discrete system controllability and observability tests

The controllabilty matrix of the system is:

$$W_c = \begin{bmatrix} 0.0029 & 0.0159 & 0.0326 & 0.0486 & 0.0621 \\ 0.8154 & 1.6003 & 1.6797 & 1.4832 & 1.2095 \\ -0.0059 & -0.03169 & -0.0651 & -0.0969 & -0.1241 \\ -1.6276 & -3.1889 & -3.3489 & -2.9745 & -2.4633 \\ 14.1734 & 10.1557 & 7.2769 & 5.2141 & 3.7361 \end{bmatrix} \tag{A.2.1}$$

The controllability matrix can be row reduced to:

$$W_c \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.2.2}$$

The observability matrix of the system is:

$$W_o = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1.0000 & 0.0074 & -1.6070E-5 & 1.6246E-7 & 0.0005 \\ 0.0049 & 0.0053 & 1.0049 & 0.0010 & -0.0010 \\ 1.0000 & 0.0112 & -5.34213e-05 & 3.3693e-070.0014 & \\ 0.0197 & 0.0176 & 1.0197 & 0.0199 & -0.0028 \\ 1.0000 & 0.0133 & -0.0001 & 2.2883E-07 & 0.0025 \\ 0.0444 & 0.0338 & 1.0444 & 0.0301 & -0.0049 \\ 1.0 & 0.0143 & -0.0002 & -3.1643E-07 & 0.0033 \\ 0.0791 & 0.0523 & 1.0791 & 0.0405 & -0.0067 \end{bmatrix} \tag{A.2.3}$$

The observability matrix can be row reduced to:

$$W_o \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.2.4}$$

Both matrices W_c and W_o are full rank, which means the discrete time system is controllable and observable.

B

Feedback system code

Below, the code used to test the feedback system can be found. Note that large parts of "LFSR256.vhd" have been omitted, because most of the code is repeated with slight only variation (different seeds and corresponding module inference).

B.1. CipherAdd_core.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

entity CipherAdd_core is
    port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_matA:     in  matrix_array(0 to FHE_CN-1, 0 to FHE_n);
            input_matB:     in  matrix_array(0 to FHE_CN-1, 0 to FHE_n);

            output_mat:     out matrix_array(0 to FHE_CN-1, 0 to FHE_n);
            calc_done:      out std_logic
         );
end CipherAdd_core;

architecture Behavioral of CipherAdd_core is

component debounce_comparator is
generic(init : std_logic := '0');
port(
        clk:        in std_logic;
        db_in:      in std_logic;
        pulse_out:  out std_logic
    );
end component debounce_comparator;


-------------------------------------------------------------------------------------------------------------
constant paralel_rows_cnt: natural := 1; --FHE_CN;
constant paralel_cols_cnt: natural := 1; --(FHE_n+1);

signal row, new_row: natural range 0 to FHE_CN-1 := FHE_CN-1;
signal col, new_col: natural range 0 to FHE_n := FHE_n;

signal done: std_logic := '1';
signal new_done: std_logic := '0';
signal mat_pipe: matrix_array(0 to FHE_CN-1, 0 to FHE_n) := (others => (others => (others => '0')));

signal calc_done_pipe: std_logic := '0';

-------------------------------------------------------------------------------------------------------------------------------------------
begin

debounce_comparator_unit: debounce_comparator
generic map(init => '1')
port map(
            clk=>clk,
            db_in=>done,
            pulse_out=>calc_done_pipe
        );

row_counting: process(clk, calc, row, col)

procedure MatMul_fun(input_matA, input_matB: matrix_array(0 to FHE_CN-1, 0 to FHE_n); row, col: natural) is --matrix_array is
    variable col_bit_cnt : natural range 0 to FHE_CN;

    begin
    if row <= FHE_CN-1 and col <= FHE_n then
        for row_sub in 0 to paralel_rows_cnt-1 loop
            for col_sub in 0 to paralel_cols_cnt-1 loop
```

```
                mat_pipe(row+row_sub, col+col_sub) <= input_matA(row+row_sub, col+col_sub) + input_matB(row+row_sub, col+col_sub);
            end loop;
        end loop;
    end if;
end procedure;

begin
if rising_edge(clk) then
    calc_done <= calc_done_pipe;

    done <= new_done;

    row <= new_row;
    col <= new_col;

    if done = '0' then
        MatMul_fun(input_matA, input_matB, row, col);
    end if;
end if;

if calc = '1' then
    new_row <= 0;
    new_col <= 0;

    new_done <= '0';
else
    if row >= FHE_CN-1 then
        if col >= FHE_n then
            new_row <= row;
            new_col <= col;

            new_done <= '1';
        else
            new_row <= 0;
            new_col <= col + paralel_cols_cnt;

            new_done <= '0';
        end if;
    else
        new_row <= row + paralel_rows_cnt;
        new_col <= col;

        new_done <= '0';
    end if;
end if;

end process;

output_print: process(done)
begin
if rising_edge(done) then
    output_mat <= mat_pipe;
end if;
end process;
end Behavioral;
```

B.2. CipherMult_core.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

entity CipherMult_core is
    port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_matA:     in  matrix_array(0 to FHE_CN-1, 0 to FHE_n);
            input_matB:     in  matrix_array(0 to FHE_CN-1, 0 to FHE_n);

            output_mat:     out matrix_array(0 to FHE_CN-1, 0 to FHE_n);
            calc_done:      out std_logic
          );
end CipherMult_core;

architecture Behavioral of CipherMult_core is

component debounce_comparator is
generic(init : std_logic := '0');
port(
        clk:        in std_logic;
        db_in:      in std_logic;
        pulse_out:  out std_logic
    );
end component debounce_comparator;


-------------------------------------------------------------------------------------------------------------------
constant paralel_rows_cnt: natural := FHE_CN/4;
constant paralel_cols_cnt: natural := (FHE_n+1)/4;

signal row, new_row: natural range 0 to FHE_CN-1 := FHE_CN-1;
signal col, new_col: natural range 0 to FHE_n := FHE_n;

signal done: std_logic := '1';
signal new_done: std_logic := '0';
signal mat_pipe: matrix_array(0 to FHE_CN-1, 0 to FHE_n) := (others => (others => (others => '0')));

signal calc_done_pipe: std_logic := '0';

-------------------------------------------------------------------------------------------------------------------------------
begin
```

```
debounce_comparator_unit: debounce_comparator
generic map(init => '1')
port map(
            clk=>clk,
            db_in=>done,
            pulse_out=>calc_done_pipe
        );

row_counting: process(clk, calc, row, col)

procedure MatMul_fun(input_matA, input_matB: matrix_array(0 to FHE_CN-1, 0 to FHE_n); row, col: natural) is --matrix_array is
    variable carry : signed(FHE_L-1 downto 0) := (others => '0');
    variable col_bit_cnt : natural range 0 to FHE_CN;

    begin
    if row <= FHE_CN-1 and col <= FHE_n then
        for row_sub in 0 to paralel_rows_cnt-1 loop
            for col_sub in 0 to paralel_cols_cnt-1 loop
                carry := (others => '0');
                col_bit_cnt := 0;
                for i in 0 to FHE_n loop
                    for k in 0 to FHE_L-1 loop
                        dbg0 := row_sub;
                        dbg1 := col_sub;
                        dbg2 := i;
                        dbg3 := k;
                        if (input_matA(row+row_sub, i)(k) = '1') then
                            carry := carry + input_matB(col_bit_cnt, col+col_sub);
                        else
                            carry := carry;
                        end if;
                        col_bit_cnt := col_bit_cnt+1;
                    end loop;
                end loop;
                mat_pipe(row+row_sub, col+col_sub) <= carry;
            end loop;
        end loop;
    end if;
end procedure;

begin
if rising_edge(clk) then
    calc_done <= calc_done_pipe;

    done <= new_done;

    row <= new_row;
    col <= new_col;

    if done = '0' then
        MatMul_fun(input_matA, input_matB, row, col);
    end if;
end if;

if calc = '1' then
    new_row <= 0;
    new_col <= 0;

    new_done <= '0';
else
    if row >= FHE_CN-1 then
        if col >= FHE_n then
            new_row <= row;
            new_col <= col;

            new_done <= '1';
        else
            new_row <= 0;
            new_col <= col + paralel_cols_cnt;

            new_done <= '0';
        end if;
    else
        new_row <= row + paralel_rows_cnt;
        new_col <= col;

        new_done <= '0';
    end if;
end if;

end process;

output_print: process(done)
begin
if rising_edge(done) then
    output_mat <= mat_pipe;
end if;
end process;
end Behavioral;
```

B.3. Controller.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

entity Controller is
    port (
            clk             :   in std_logic;
            calc            :   in std_logic;
            reset           :   in std_logic;
```

```
            theta_0         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            theta_1         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            upd_u           :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh0         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh1         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh2         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh3         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh4         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            out_u           :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh0         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh1         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh2         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh3         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh4         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_00            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_10            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_20            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_30            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_40            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_01            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_11            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_21            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_31            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_41            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_02            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_12            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_22            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_32            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_42            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_03            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_13            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_23            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_33            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_43            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_04            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_14            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_24            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_34            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_44            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            B_0             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_1             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_2             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_3             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_4             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            K_0             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_1             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_2             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_3             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_4             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            L_00            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_10            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_20            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_30            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_40            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            L_01            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_11            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_21            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_31            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_41            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            neg_one         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            done_flag       :   out std_logic
          );
end Controller;

-------------------------------------------------------------------------------------------------

architecture Behavioral of Controller is

component debounce_comparator is
generic(init : std_logic := '0');
port(
        clk:        in std_logic;
        db_in:      in std_logic;
        pulse_out:  out std_logic
    );
end component debounce_comparator;

component CipherAdd_core is
    port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_matA:     in  matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            input_matB:     in  matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            output_mat:     out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            calc_done:      out std_logic
          );
end component;

component CipherMult_core is
    port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_matA:     in  matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            input_matB:     in  matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
```

```
            output_mat:     out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            calc_done:      out std_logic
          );
end component;

-------------------------------------------------------------------------------------------------
constant steps: natural := 85+1;
signal i, new_i: natural range 0 to steps := steps;

signal mult, mult_done, add, add_done, mult_flag, add_flag: std_logic;
signal done: std_logic := '1';
signal done_db, boot, boot_db: std_logic := '0';

signal xh_0, xh_1, xh_2, xh_3, xh_4, new_u, cur_u : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1) := (others => (others => '0'));
signal new_xh_0, new_xh_1, new_xh_2, new_xh_3, new_xh_4 : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal new_xh_tmp_0, new_xh_tmp_1, new_xh_tmp_2, new_xh_tmp_3, new_xh_tmp_4 : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal carry_0, carry_1, carry_2, carry_3, carry_4 : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

signal sum_of_elements : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

signal cur_mat1, cur_mat2, cur_mult_out, cur_add_out : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

signal start_delay: std_logic;

begin
-------------------------------------------------------------------------------------------------
debounce_comparator_unit: debounce_comparator
generic map(init => '1')
port map(
            clk=>clk,
            db_in=>done,
            pulse_out=>done_db
        );

debounce_comparator_unit_boot: debounce_comparator
generic map(init => '1')
port map(
            clk=>clk,
            db_in=>boot,
            pulse_out=>boot_db
        );

done_flag <= done_db;

CipherMult_core_unit: CipherMult_core
port map(
            clk=>clk,
            calc=>mult,

            input_matA=>cur_mat1,
            input_matB=>cur_mat2,

            output_mat=>cur_mult_out,
            calc_done=>mult_done
        );

CipherAdd_core_unit: CipherAdd_core
port map(
            clk=>clk,
            calc=>add,

            input_matA=>cur_mat1,
            input_matB=>cur_mat2,

            output_mat=>cur_add_out,
            calc_done=>add_done
        );

calculation_flag_mux: process(clk)--process(mult_flag, add_flag, mult_done, add_done, done, boot_db, i)
begin
    if rising_edge(clk) then
        start_delay <= mult_done or add_done;
--        if (mult_flag = '1' and done = '0') or calc = '1' then
        if mult_flag = '1' and done = '0' then
            mult <= start_delay or calc;
        else
            mult <= '0';
        end if;

--      if add_flag = '1' and done = '0' then
        if (add_flag = '1' and done = '0') or calc = '1' then
            add <= start_delay or calc;
        else
            add <= '0';
        end if;
    end if;
end process;


increment_logic: process(calc, i)
begin
    if calc = '1' then
        new_i <= 0;
        done <= '0';
--        boot <= '0';
    else
--        if i = 0 then
--            new_i <= i + 1;
--            done <= '0';
--            boot <= '1';
        if i = steps then
            new_i <= i;
            done <= '1';
--            boot <= '0';
        else
            new_i <= i + 1;
            done <= '0';
--            boot <= '0';
        end if;
    end if;
end process;
```

```
sync: process(clk, reset)
begin
if rising_edge(clk) then
--    if mult = '1' or add = '1' then
--        i <= new_i;
--    end if;

    if (mult_done = '1' or add_done = '1') and calc = '0' then
        i <= new_i;
    elsif calc = '1' then
        i <= 0;
    end if;

--    if done_db = '1' then
--        crtl_effort <= u;
--    end if;

end if;
end process;

-------------------------------------------------------------------------------------------------------------------------------
input_proc_0: process(i)
begin
--        if i = 0 then
--                cur_mat1  <= neg_one;
--                cur_mat2  <= upd_xh0;
--                mult_flag <= '1';
--                add_flag  <= '0';
--        end if;
--        if i = 1 then
--                cur_mat1  <= neg_one;
--                cur_mat2  <= upd_xh2;
--                mult_flag <= '1';
--                add_flag  <= '0';
--        end if;
        if i = 0 then
                cur_mat1  <= upd_xh0;
                cur_mat2  <= theta_0;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 1 then
                cur_mat1  <= upd_xh2;
                cur_mat2  <= theta_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 2 then
                cur_mat1  <= L_00;
                cur_mat2  <= new_xh_0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 3 then
                cur_mat1  <= L_01;
                cur_mat2  <= new_xh_1;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 4 then
                cur_mat1  <= carry_0;
                cur_mat2  <= carry_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 5 then
                cur_mat1  <= L_10;
                cur_mat2  <= new_xh_0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 6 then
                cur_mat1  <= L_11;
                cur_mat2  <= new_xh_1;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 7 then
                cur_mat1  <= carry_0;
                cur_mat2  <= carry_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 8 then
                cur_mat1  <= L_20;
                cur_mat2  <= new_xh_0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 9 then
                cur_mat1  <= L_21;
                cur_mat2  <= new_xh_1;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 10 then
                cur_mat1  <= carry_0;
                cur_mat2  <= carry_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 11 then
                cur_mat1  <= L_30;
                cur_mat2  <= new_xh_0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 12 then
                cur_mat1  <= L_31;
                cur_mat2  <= new_xh_1;
                mult_flag <= '1';
                add_flag  <= '0';
```

```
        end if;
        if i = 13 then
                cur_mat1  <= carry_0;
                cur_mat2  <= carry_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 14 then
                cur_mat1  <= L_40;
                cur_mat2  <= new_xh_0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 15 then
                cur_mat1  <= L_41;
                cur_mat2  <= new_xh_1;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 16 then
                cur_mat1  <= carry_0;
                cur_mat2  <= carry_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 17 then
                cur_mat1  <= B_0;
                cur_mat2  <= upd_u;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 18 then
                cur_mat1  <= new_xh_0; --!!!!!!!!!!!!!!!!!!! tm B4
                cur_mat2  <= new_xh_tmp_0;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 19 then
                cur_mat1  <= B_1;
                cur_mat2  <= upd_u;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 20 then
                cur_mat1  <= new_xh_1;
                cur_mat2  <= new_xh_tmp_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 21 then
                cur_mat1  <= B_2;
                cur_mat2  <= upd_u;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 22 then
                cur_mat1  <= new_xh_2;
                cur_mat2  <= new_xh_tmp_2;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 23 then
                cur_mat1  <= B_3;
                cur_mat2  <= upd_u;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 24 then
                cur_mat1  <= new_xh_3;
                cur_mat2  <= new_xh_tmp_3;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 25 then
                cur_mat1  <= B_4;
                cur_mat2  <= upd_u;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 26 then
                cur_mat1  <= new_xh_4;
                cur_mat2  <= new_xh_tmp_4;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 27 then
                cur_mat1  <= A_00;
                cur_mat2  <= xh_0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 28 then
                cur_mat1  <= A_01;
                cur_mat2  <= xh_1;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 29 then
                cur_mat1  <= A_02;
                cur_mat2  <= xh_2;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 30 then
                cur_mat1  <= A_03;
                cur_mat2  <= xh_3;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 31 then
                cur_mat1  <= A_04;
                cur_mat2  <= xh_4;
```

```
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 32 then
            cur_mat1  <= carry_0;
            cur_mat2  <= carry_1;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 33 then
            cur_mat1  <= carry_1;
            cur_mat2  <= carry_2;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 34 then
            cur_mat1  <= carry_2;
            cur_mat2  <= carry_3;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 35 then
            cur_mat1  <= carry_3;
            cur_mat2  <= carry_4;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 36 then
            cur_mat1  <= A_10;
            cur_mat2  <= xh_0;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 37 then
            cur_mat1  <= A_11;
            cur_mat2  <= xh_1;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 38 then
            cur_mat1  <= A_12;
            cur_mat2  <= xh_2;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 39 then
            cur_mat1  <= A_13;
            cur_mat2  <= xh_3;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 40 then
            cur_mat1  <= A_14;
            cur_mat2  <= xh_4;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 41 then
            cur_mat1  <= carry_0;
            cur_mat2  <= carry_1;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 42 then
            cur_mat1  <= carry_1;
            cur_mat2  <= carry_2;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 43 then
            cur_mat1  <= carry_2;
            cur_mat2  <= carry_3;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 44 then
            cur_mat1  <= carry_3;
            cur_mat2  <= carry_4;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 45 then
            cur_mat1  <= A_20;
            cur_mat2  <= xh_0;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 46 then
            cur_mat1  <= A_21;
            cur_mat2  <= xh_1;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 47 then
            cur_mat1  <= A_22;
            cur_mat2  <= xh_2;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 48 then
            cur_mat1  <= A_23;
            cur_mat2  <= xh_3;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 49 then
            cur_mat1  <= A_24;
            cur_mat2  <= xh_4;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 50 then
```

```
            cur_mat1  <= carry_0;
            cur_mat2  <= carry_1;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 51 then
            cur_mat1  <= carry_1;
            cur_mat2  <= carry_2;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 52 then
            cur_mat1  <= carry_2;
            cur_mat2  <= carry_3;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 53 then
            cur_mat1  <= carry_3;
            cur_mat2  <= carry_4;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 54 then
            cur_mat1  <= A_30;
            cur_mat2  <= xh_0;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 55 then
            cur_mat1  <= A_31;
            cur_mat2  <= xh_1;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 56 then
            cur_mat1  <= A_32;
            cur_mat2  <= xh_2;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 57 then
            cur_mat1  <= A_33;
            cur_mat2  <= xh_3;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 58 then
            cur_mat1  <= A_34;
            cur_mat2  <= xh_4;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 59 then
            cur_mat1  <= carry_0;
            cur_mat2  <= carry_1;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 60 then
            cur_mat1  <= carry_1;
            cur_mat2  <= carry_2;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 61 then
            cur_mat1  <= carry_2;
            cur_mat2  <= carry_3;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 62 then
            cur_mat1  <= carry_3;
            cur_mat2  <= carry_4;
            mult_flag <= '0';
            add_flag  <= '1';
        end if;
        if i = 63 then
            cur_mat1  <= A_40;
            cur_mat2  <= xh_0;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 64 then
            cur_mat1  <= A_41;
            cur_mat2  <= xh_1;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 65 then
            cur_mat1  <= A_42;
            cur_mat2  <= xh_2;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 66 then
            cur_mat1  <= A_43;
            cur_mat2  <= xh_3;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 67 then
            cur_mat1  <= A_44;
            cur_mat2  <= xh_4;
            mult_flag <= '1';
            add_flag  <= '0';
        end if;
        if i = 68 then
            cur_mat1  <= carry_0;
            cur_mat2  <= carry_1;
            mult_flag <= '0';
            add_flag  <= '1';
```

```
        end if;
        if i = 69 then
                cur_mat1  <= carry_1;
                cur_mat2  <= carry_2;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 70 then
                cur_mat1  <= carry_2;
                cur_mat2  <= carry_3;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 71 then
                cur_mat1  <= carry_3;
                cur_mat2  <= carry_4;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 72 then
                cur_mat1  <= new_xh_0;  --!!!!!!!!!!!!!!!!!!!! tm end
                cur_mat2  <= new_xh_tmp_0;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 73 then
                cur_mat1  <= new_xh_1;
                cur_mat2  <= new_xh_tmp_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 74 then
                cur_mat1  <= new_xh_2;
                cur_mat2  <= new_xh_tmp_2;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 75 then
                cur_mat1  <= new_xh_3;
                cur_mat2  <= new_xh_tmp_3;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 76 then
                cur_mat1  <= new_xh_4;
                cur_mat2  <= new_xh_tmp_4;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 77 then
                cur_mat1  <= K_0;
                cur_mat2  <= xh_0; --upd_xh0;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 78 then
                cur_mat1  <= K_1;
                cur_mat2  <= xh_1; --upd_xh1;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 79 then
                cur_mat1  <= K_2;
                cur_mat2  <= xh_2; --upd_xh2;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 80 then
                cur_mat1  <= K_3;
                cur_mat2  <= xh_3; --upd_xh3;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 81 then
                cur_mat1  <= K_4;
                cur_mat2  <= xh_4; --upd_xh4;
                mult_flag <= '1';
                add_flag  <= '0';
        end if;
        if i = 82 then
                cur_mat1  <= carry_0;
                cur_mat2  <= carry_1;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 83 then
                cur_mat1  <= carry_1;
                cur_mat2  <= carry_2;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 84 then
                cur_mat1  <= carry_2;
                cur_mat2  <= carry_3;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
        if i = 85 then
                cur_mat1  <= carry_3;
                cur_mat2  <= carry_4;
                mult_flag <= '0';
                add_flag  <= '1';
        end if;
end process;

output_proc_1: process(clk)
begin
        if rising_edge(clk) then
--                if i = 0 and mult_done = '1' then
--                        new_xh_0 <= cur_mult_out;

--                        xh_0 <= upd_xh0;
```

```
--                      xh_1 <= upd_xh1;
--                      xh_2 <= upd_xh2;
--                      xh_3 <= upd_xh3;
--                      xh_4 <= upd_xh4;

--                end if;
--                if i = 1 and mult_done = '1' then
--                        new_xh_1 <= cur_mult_out;
--                end if;
                  if i = 0 and add_done = '1' then
                          new_xh_0 <= cur_add_out;

--                      cur_u <= new_u;
--                      new_u <= upd_u;

                        xh_0 <= upd_xh0;
              xh_1 <= upd_xh1;
              xh_2 <= upd_xh2;
              xh_3 <= upd_xh3;
              xh_4 <= upd_xh4;
                  end if;
                  if i = 1 and add_done = '1' then
                          new_xh_1 <= cur_add_out;
                  end if;
                  if i = 2 and mult_done = '1' then
                          carry_0 <= cur_mult_out;
                  end if;
                  if i = 3 and mult_done = '1' then
                          carry_1 <= cur_mult_out;
                  end if;
                  if i = 4 and add_done = '1' then
                          new_xh_tmp_0 <= cur_add_out;
                  end if;
                  if i = 5 and mult_done = '1' then
                          carry_0 <= cur_mult_out;
                  end if;
                  if i = 6 and mult_done = '1' then
                          carry_1 <= cur_mult_out;
                  end if;
                  if i = 7 and add_done = '1' then
                          new_xh_tmp_1 <= cur_add_out;
                  end if;
                  if i = 8 and mult_done = '1' then
                          carry_0 <= cur_mult_out;
                  end if;
                  if i = 9 and mult_done = '1' then
                          carry_1 <= cur_mult_out;
                  end if;
                  if i = 10 and add_done = '1' then
                          new_xh_tmp_2 <= cur_add_out;
                  end if;
                  if i = 11 and mult_done = '1' then
                          carry_0 <= cur_mult_out;
                  end if;
                  if i = 12 and mult_done = '1' then
                          carry_1 <= cur_mult_out;
                  end if;
                  if i = 13 and add_done = '1' then
                          new_xh_tmp_3 <= cur_add_out;
                  end if;
                  if i = 14 and mult_done = '1' then
                          carry_0 <= cur_mult_out;
                  end if;
                  if i = 15 and mult_done = '1' then
                          carry_1 <= cur_mult_out;
                  end if;
                  if i = 16 and add_done = '1' then
                          new_xh_tmp_4 <= cur_add_out;
                  end if;
                  if i = 17 and mult_done = '1' then
                          new_xh_0 <= cur_mult_out;
                  end if;
                  if i = 18 and add_done = '1' then
                          new_xh_0 <= cur_add_out;
                  end if;
                  if i = 19 and mult_done = '1' then
                          new_xh_1 <= cur_mult_out;
                  end if;
                  if i = 20 and add_done = '1' then
                          new_xh_1 <= cur_add_out;
                  end if;
                  if i = 21 and mult_done = '1' then
                          new_xh_2 <= cur_mult_out;
                  end if;
                  if i = 22 and add_done = '1' then
                          new_xh_2 <= cur_add_out;
                  end if;
                  if i = 23 and mult_done = '1' then
                          new_xh_3 <= cur_mult_out;
                  end if;
                  if i = 24 and add_done = '1' then
                          new_xh_3 <= cur_add_out;
                  end if;
                  if i = 25 and mult_done = '1' then
                          new_xh_4 <= cur_mult_out;
                  end if;
                  if i = 26 and add_done = '1' then
                          new_xh_4 <= cur_add_out;
                  end if;
                  if i = 27 and mult_done = '1' then
                          carry_0 <= cur_mult_out;
                  end if;
                  if i = 28 and mult_done = '1' then
                          carry_1 <= cur_mult_out;
                  end if;
                  if i = 29 and mult_done = '1' then
                          carry_2 <= cur_mult_out;
                  end if;
                  if i = 30 and mult_done = '1' then
                          carry_3 <= cur_mult_out;
                  end if;
                  if i = 31 and mult_done = '1' then
```

```
        carry_4 <= cur_mult_out;
end if;
if i = 32 and add_done = '1' then
        carry_1 <= cur_add_out;
end if;
if i = 33 and add_done = '1' then
        carry_2 <= cur_add_out;
end if;
if i = 34 and add_done = '1' then
        carry_3 <= cur_add_out;
end if;
if i = 35 and add_done = '1' then
        new_xh_tmp_0 <= cur_add_out;
end if;
if i = 36 and mult_done = '1' then
        carry_0 <= cur_mult_out;
end if;
if i = 37 and mult_done = '1' then
        carry_1 <= cur_mult_out;
end if;
if i = 38 and mult_done = '1' then
        carry_2 <= cur_mult_out;
end if;
if i = 39 and mult_done = '1' then
        carry_3 <= cur_mult_out;
end if;
if i = 40 and mult_done = '1' then
        carry_4 <= cur_mult_out;
end if;
if i = 41 and add_done = '1' then
        carry_1 <= cur_add_out;
end if;
if i = 42 and add_done = '1' then
        carry_2 <= cur_add_out;
end if;
if i = 43 and add_done = '1' then
        carry_3 <= cur_add_out;
end if;
if i = 44 and add_done = '1' then
        new_xh_tmp_1 <= cur_add_out;
end if;
if i = 45 and mult_done = '1' then
        carry_0 <= cur_mult_out;
end if;
if i = 46 and mult_done = '1' then
        carry_1 <= cur_mult_out;
end if;
if i = 47 and mult_done = '1' then
        carry_2 <= cur_mult_out;
end if;
if i = 48 and mult_done = '1' then
        carry_3 <= cur_mult_out;
end if;
if i = 49 and mult_done = '1' then
        carry_4 <= cur_mult_out;
end if;
if i = 50 and add_done = '1' then
        carry_1 <= cur_add_out;
end if;
if i = 51 and add_done = '1' then
        carry_2 <= cur_add_out;
end if;
if i = 52 and add_done = '1' then
        carry_3 <= cur_add_out;
end if;
if i = 53 and add_done = '1' then
        new_xh_tmp_2 <= cur_add_out;
end if;
if i = 54 and mult_done = '1' then
        carry_0 <= cur_mult_out;
end if;
if i = 55 and mult_done = '1' then
        carry_1 <= cur_mult_out;
end if;
if i = 56 and mult_done = '1' then
        carry_2 <= cur_mult_out;
end if;
if i = 57 and mult_done = '1' then
        carry_3 <= cur_mult_out;
end if;
if i = 58 and mult_done = '1' then
        carry_4 <= cur_mult_out;
end if;
if i = 59 and add_done = '1' then
        carry_1 <= cur_add_out;
end if;
if i = 60 and add_done = '1' then
        carry_2 <= cur_add_out;
end if;
if i = 61 and add_done = '1' then
        carry_3 <= cur_add_out;
end if;
if i = 62 and add_done = '1' then
        new_xh_tmp_3 <= cur_add_out;
end if;
if i = 63 and mult_done = '1' then
        carry_0 <= cur_mult_out;
end if;
if i = 64 and mult_done = '1' then
        carry_1 <= cur_mult_out;
end if;
if i = 65 and mult_done = '1' then
        carry_2 <= cur_mult_out;
end if;
if i = 66 and mult_done = '1' then
        carry_3 <= cur_mult_out;
end if;
if i = 67 and mult_done = '1' then
        carry_4 <= cur_mult_out;
end if;
if i = 68 and add_done = '1' then
        carry_1 <= cur_add_out;
```

```
                end if;
                if i = 69 and add_done = '1' then
                        carry_2 <= cur_add_out;
                end if;
                if i = 70 and add_done = '1' then
                        carry_3 <= cur_add_out;
                end if;
                if i = 71 and add_done = '1' then
                        new_xh_tmp_4 <= cur_add_out;
                end if;
                if i = 72 and add_done = '1' then
                        xh_0 <= cur_add_out;
                end if;
                if i = 73 and add_done = '1' then
                        xh_1 <= cur_add_out;
                end if;
                if i = 74 and add_done = '1' then
                        xh_2 <= cur_add_out;
                end if;
                if i = 75 and add_done = '1' then
                        xh_3 <= cur_add_out;
                end if;
                if i = 76 and add_done = '1' then
                        xh_4 <= cur_add_out;
                end if;
                if i = 77 and mult_done = '1' then
                        carry_0 <= cur_mult_out;
                end if;
                if i = 78 and mult_done = '1' then
                        carry_1 <= cur_mult_out;
                end if;
                if i = 79 and mult_done = '1' then
                        carry_2 <= cur_mult_out;
                end if;
                if i = 80 and mult_done = '1' then
                        carry_3 <= cur_mult_out;
                end if;
                if i = 81 and mult_done = '1' then
                        carry_4 <= cur_mult_out;
                end if;
                if i = 82 and add_done = '1' then
                        carry_1 <= cur_add_out;
                end if;
                if i = 83 and add_done = '1' then
                        carry_2 <= cur_add_out;
                end if;
                if i = 84 and add_done = '1' then
                        carry_3 <= cur_add_out;
                end if;
                if i = 85 and add_done = '1' then
--                          new_u <= cur_add_out;
                        out_u <= cur_add_out;

                        out_xh0 <= xh_0;
                        out_xh1 <= xh_1;
                        out_xh2 <= xh_2;
                        out_xh3 <= xh_3;
                        out_xh4 <= xh_4;

                end if;
        end if;
end process;

end Behavioral;
```

B.4. debounce_comparator.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural design of a circuit that generates a pulse when the input
--changes from low to high

--Every clock tick the input is stored. If the previous input was low and
--the new input is high, then the output is set to high for one clock tick.
--This adapter is a useful tool to ensure that when another circuit
--outputs a high signal, it can be converted to a single pulse (one clock tick).
--This can be necessary, as some circuitry accepts start signals that
--are expected to only take a single clock tick before returning to low.

library IEEE;
use IEEE.std_logic_1164.all;

--entity declaration--------------------------------------------------------------------
entity debounce_comparator is
generic(init: std_logic := '0');
port(
        clk         :   in std_logic;
        db_in       :   in std_logic;
        pulse_out   :   out std_logic
    );
end;

---------------------------------------------------------------------------------------
architecture Behavioural of debounce_comparator is

--signal declaration--------------------------------------------------------------------
signal db_out   :   std_logic := init;

begin
button_press_pacing: process(clk, db_out, db_in)
    begin
        if rising_edge(clk) then
            db_out <= db_in;
        end if;
```

```
        if db_in = '1' and db_in /= db_out then
            pulse_out <= '1';
        else
            pulse_out <= '0';
        end if;
end process;

end architecture Behavioural;
```

B.5. Decoder.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity Decoder is
port(
    clk             :   in std_logic;
    calc            :   in std_logic;
    reset           :   in std_logic;
    cipher          :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
    private_key     :   in vector_array(0 to FHE_n-1);
    decoded_mu      :   out signed(FHE_L-1 downto 0);

    done_flag       :   out std_logic
    );
end Decoder;

architecture Behavioral of Decoder is

component BitMatVectMul_core is
generic(row_countA, col_countA, row_countB : natural);
    Port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_mat:      in matrix_bit_array(0 to row_countA-1, 0 to col_countA-1);
            input_vect:     in vector_array(0 to row_countB-1);

            output_vect:    out vector_array(0 to row_countB-1) := (others => (others => '0'));
            calc_done:      out std_logic
          );
end component;

-------------------------------------------------------------------------------------------------------------------------
signal po2_sk, mug      : vector_array(0 to FHE_CN-1);
signal calc_done        : std_logic;
signal mu_pipe          : signed(FHE_L-1 downto 0) := (others=>'0');

-------------------------------------------------------------------------------------------------------------------------

function PowersOf2(t: vector_array(0 to FHE_n-1)) return vector_array is
variable sk: vector_array(0 to FHE_n);
variable po2: vector_array(0 to FHE_CN-1);
begin
sk(0) := to_signed(1, FHE_L);
for i in 1 to FHE_n loop
    sk(i) := t(i-1);
end loop;

for i in 0 to FHE_n loop
    for j in 0 to FHE_L-1 loop
        if (i = 0) then
            po2(i*FHE_L+j) := shift_left(sk(i),j);
        else
            po2(i*FHE_L+j) := -shift_left(sk(i),j);
        end if;
    end loop;
end loop;
return po2;
end function;

--function MPDec(mug: vector_array(0 to FHE_CN-1)) return signed is
--variable mu, tmp: signed(FHE_L-1 downto 0) := (others => '0');
--begin
--for i in 0 to FHE_L-1 loop
--    if (mu(i) = '1') then
--        tmp := mug(FHE_L - i - 1) + 1;
--        tmp := shift_right(tmp,FHE_L-i-1);
--        mu(i) := tmp(i);
--    else
--        tmp := mug(FHE_L - i - 1);
--        tmp := shift_right(tmp,FHE_L-i-1);
--        mu(i) := tmp(i);
--    end if;
--end loop;
--mu(L-1) := '0';

--return mu;
--end function;

function MPDec(mug: vector_array(0 to FHE_CN-1)) return signed is
variable mu, tmp: signed(FHE_L-1 downto 0) := (others => '0');
variable xor_check: std_logic;
constant bit_one: signed(FHE_L-1 downto 0) := (0 => '1', others => '0');
```

```
begin

mu(0) := mug(FHE_L - 1)(FHE_L - 1);

for i in 1 to FHE_L-1 loop
    tmp := mug(FHE_L - i - 1) - shift_left(mu, FHE_L - i - 1);

    xor_check := tmp(FHE_L-1) xor tmp(FHE_L-2);
    if xor_check = '1' then
        mu := mu + shift_left(bit_one, i);
    end if;
end loop;

return mu;
end function;

begin

BitMatVectMul_unit: BitMatVectMul_core
generic map(row_countA=>(FHE_n+1)*FHE_L, col_countA=>(FHE_n+1)*FHE_L, row_countB=>(FHE_n+1)*FHE_L)
port map(
        clk=>clk,
        calc=>calc,

        input_mat=>cipher,
        input_vect=>po2_sk,

        output_vect=>mug,
        calc_done=>calc_done
        );

operation: process(clk, calc, calc_done)
begin

done_flag <= calc_done;

if rising_edge(clk) then
    if (calc = '1') then
        po2_sk <= PowersOf2(private_key);
    end if;

    if (calc_done = '1') then
        mu_pipe <= MPDec(mug);
    end if;
end if;
end process;

decoded_mu <= mu_pipe;

end Behavioral;
```

B.6. error_vect_gen.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity error_vect_gen is
port(
    clk             :   in std_logic;
    input_vect      :   in vector_array_e(0 to FHE_m-1);
    calc            :   in std_logic;

    output_vect     :   out vector_array(0 to FHE_m-1);
    done_flag       :   out std_logic := '0'
    );
end error_vect_gen;

architecture Behavioral of error_vect_gen is

-------------------------------------------------------------------

type LZ_list_type is array (FHE_m-1 downto 0) of natural range 0 to L;
constant one_sub: signed(L-1 downto 0) := ('1', others => '0');
constant padding: signed(FHE_L-L-1 downto 0) := (others => '0');
-------------------------------------------------------------------

--This modulo operator works only when the second operand is a power of 2
function mod_powerof2(a: signed(L-1 downto 0); b: integer) return signed is
variable c : signed(L-1 downto 0) := (others => '0');
begin
    c(b-2 downto 0) := a(b-2 downto 0);
return c;
end function;


function LZ_count(vect: vector_array_e(0 to FHE_m-1)) return LZ_list_type is
variable tmp        : signed(L-1 downto 0)  := (others => '0');
variable LZp_list   : LZ_list_type          := (others => 0);
variable done       : std_logic             := '0';
variable check      : std_logic             := '0';
begin
    for i in 0 to FHE_m-1 loop
        tmp := vect(i);
        done := '0';
        for j in 1 to L loop
            check := tmp(L-j);
```

```
            if (tmp(L-j) = '0') then
                if (done = '0') then
                    LZp_list(i) := LZp_list(i) + 1;
                else
                    LZp_list(i) := LZp_list(i);
                end if;
            else
                done := '1';
            end if;
        end loop;
    end loop;
return LZp_list;
end function;

--Paper: A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision
function polyfit(input_vect: vector_array_e(0 to FHE_m-1)) return vector_array_e is

variable output_vect    : vector_array_e(0 to FHE_m-1);
variable x              : vector_array_e(0 to FHE_m-1);
variable x_tmp          : signed(L-1 downto 0)              := (others => '0');
variable x_out_tmp      : signed(L-1 downto 0)                  := (others => '0');
variable level_1        : signed(L-1 downto 0)                  := (others => '0');
variable level_2        : natural range 0 to lin_segs_pow       := 0;
variable level_tmp      : natural range 0 to lin_segs*log_segs  := 0;
variable level_sel      : natural range 0 to lin_segs*log_segs  := 0;
variable LZ_list        : LZ_list_type                          := (others => 0);
variable half_way_flag  : std_logic_vector(FHE_m-1 downto 0)    := (others => '0');
variable mult_tmp0      : signed(2*L-1 downto 0)                := (others => '0');
variable mult_tmp1      : signed(2*L-1 downto 0)                := (others => '0');
variable mult_tmp2      : signed(L-1 downto 0)                  := (others => '0');

begin
    for i in 0 to FHE_m-1 loop
        half_way_flag(i) := input_vect(i)(L-1);
        if (half_way_flag(i) = '1') then
            x_tmp := input_vect(i)(L-1 downto 0);
            x_tmp(L-1) := '0';
            x_out_tmp := one_sub - x_tmp;
            x(i) := padding & x_out_tmp;
        else
            x(i) := input_vect(i);
        end if;
    end loop;

    LZ_list := LZ_count(x);
    for i in 0 to FHE_m-1 loop
        level_1 := shift_left(x(i)(L-1 downto 0),(LZ_list(i) + 1));
        level_2 := to_integer(unsigned(level_1(L-1 downto L-lin_segs_pow)));

        level_tmp := p_vect_size - LZ_list(i)*log_segs + level_2;
        if level_tmp < 0 then
            level_sel := 0;
        elsif level_tmp > p_vect_size-1 then
            level_sel := p_vect_size-1;
        else
            level_sel := level_tmp;
        end if;

        mult_tmp0 := p0(level_sel)*x(i)(L-1 downto 0);
        mult_tmp1 := shift_right(mult_tmp0, L); --See documentation for elaboration for the use of the Qmn format on this line
        mult_tmp2 := mult_tmp1(L-1 downto 0) + p1(level_sel);

        if (half_way_flag(i) = '1') then
            output_vect(i) := padding & (-mult_tmp2);
        else
            output_vect(i) := padding & mult_tmp2;
        end if;
    end loop;
return output_vect;
end function;


function psi_transform(input_vect: vector_array_e(0 to FHE_m-1)) return vector_array is

variable output_vect    : vector_array(0 to FHE_m-1);
variable tmp0           : signed(2*L - 1 downto 0)      := (others => '0');
variable tmp1           : signed(2*L - 1 downto 0)      := (others => '0');
variable tmp2           : signed(L - 1 downto 0)        := (others => '0');
variable tmp3           : signed(L - 1 downto 0)        := (others => '0');

variable debug: signed(L - 1 downto 0) := (others => '0');
begin
    for i in 0 to FHE_m-1 loop
        debug := mod_powerof2(input_vect(i)(L-1 downto 0), Qmn_norm_one);
        tmp0 := FHE_q_bitvect * (mod_powerof2(input_vect(i)(L-1 downto 0), Qmn_norm_one));
        tmp1 := shift_right(tmp0, Qn_norm);
        tmp2 := tmp1(L-1 downto 0);

        --the next procedure rounds the result
        --rule: round to the nearest -> ties to even
        if (tmp2(L-1) = '1') then
            if (tmp2(Qn_norm-1) = '0') then
                tmp3 := shift_right(tmp2, Qn_norm) - 1;
            else
                tmp3 := shift_right(tmp2, Qn_norm);
            end if;
        else
            if (tmp2(Qn_norm-1) = '1') then
                tmp3 := shift_right(tmp2, Qn_norm) + 1;
            else
                tmp3 := shift_right(tmp2, Qn_norm);
            end if;
        end if;

        output_vect(i) := resize(tmp3, FHE_L);
    end loop;
return output_vect;
end function;
-------------------------------------------------------------------------------------------------

begin
```

```
rand_gen: process(clk, calc, input_vect)
begin
if rising_edge(clk) then
    done_flag <= calc;

    if calc = '1' then
        output_vect <= psi_transform(polyfit(input_vect));
    end if;
end if;
end process;

end architecture Behavioral;
```

B.7. FHE_controller_types.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
    use IEEE.MATH_REAL.ALL;
package types_package is

--Normal distribution generation parameters-----------------------------------------------------------------------
constant L: natural := 64;
constant Qm_norm: natural:= 13;
constant Qn_norm: natural:= 51;

type p_vect_type is array (240-1 downto 0) of signed(L-1 downto 0);

constant p0_0: signed(L-1 downto 0):= "0101000111000110011001111010010100010100001101000000000000000000";
constant p0_1: signed(L-1 downto 0):= "0100110101010011101000100100111010000111000001110000000000000000";
constant p0_2: signed(L-1 downto 0):= "0100100101011001011100100100110001101101100100000000000000000000";
constant p0_3: signed(L-1 downto 0):= "0100001011000101100001001011010111010101111010000000000000000000";
constant p0_4: signed(L-1 downto 0):= "0100001010001001000100101000101110010011011111000000000000000000";
constant p0_5: signed(L-1 downto 0):= "0011111110011000000011101001100111000010110011000000000000000000";
constant p0_6: signed(L-1 downto 0):= "0011110011101000100010111010111010000100011100000000000000000000";
constant p0_7: signed(L-1 downto 0):= "0011101001110010010010000001001000110011010110000000000000000000";
constant p0_8: signed(L-1 downto 0):= "0011100000101110010110011100100011010111110100000000000000000000";
constant p0_9: signed(L-1 downto 0):= "0011011000010110111010001100111111010111111010000000000000000000";
constant p0_10: signed(L-1 downto 0):= "0011010000100110111111010010000011111000101011000000000000000000";
constant p0_11: signed(L-1 downto 0):= "0011001001011010010101011111111100100010010110000000000000000000";
constant p0_12: signed(L-1 downto 0):= "0011000010101010100101000000100010111101110100000000000000000000";
constant p0_13: signed(L-1 downto 0):= "0010111100011100101011011010001111111001000000000000000000000000";
constant p0_14: signed(L-1 downto 0):= "0010110110010110111111011010110011101100000100000000000000000000";
constant p0_15: signed(L-1 downto 0):= "0010110001000110000101000101111011101100100111010000000000000000";
constant p0_16: signed(L-1 downto 0):= "0010101001011111111100011100101010100110001000000000000000000000";
constant p0_17: signed(L-1 downto 0):= "0010100000010011111111111001110010110010001010010000000000000000";
constant p0_18: signed(L-1 downto 0):= "0010011000000110001110001001100110000000101000000000000000000000";
constant p0_19: signed(L-1 downto 0):= "0010010000101101001011010110111010100011110010100000000000000000";
constant p0_20: signed(L-1 downto 0):= "0010001010000001010000001000100101001011000000000000000000000000";
constant p0_21: signed(L-1 downto 0):= "0010000011111100001111000000110010001101100010000000000000000000";
constant p0_22: signed(L-1 downto 0):= "0001111110011001000000100010001001100111110100100000000000000000";
constant p0_23: signed(L-1 downto 0):= "0001110010100110101000111001001100111001100100000000000000000000";
constant p0_24: signed(L-1 downto 0):= "0001110100100111100110100001011110010111010101000000000000000000";
constant p0_25: signed(L-1 downto 0):= "0001110000001001011010111111100000011011001100100000000000000000";
constant p0_26: signed(L-1 downto 0):= "0001101100010010011101101110101011000100111110000000000000000000";
constant p0_27: signed(L-1 downto 0):= "0001101000100100010100110011101100101010101101000000000000000000";
constant p0_28: signed(L-1 downto 0):= "0001100101000110011101011101010101110110000011000000000000000000";
constant p0_29: signed(L-1 downto 0):= "0001100001110111010100101011100101011011111011000000000000000000";
constant p0_30: signed(L-1 downto 0):= "0001011110110101011010111010111100111011111101000000000000000000";
constant p0_31: signed(L-1 downto 0):= "0001010111111111110001000010001000110001100101100000000000000000";
constant p0_32: signed(L-1 downto 0):= "0001011000000100000011111011111001001000010100100000000000000000";
constant p0_33: signed(L-1 downto 0):= "0001010011010011111000000111110011000101001100000000000000000000";
constant p0_34: signed(L-1 downto 0):= "0001001111000011110011101111001010000100011011010000000000000000";
constant p0_35: signed(L-1 downto 0):= "0001001011001110111101111011111111000100011100000000000000000000";
constant p0_36: signed(L-1 downto 0):= "0001000111110001011100101101000011101110000000000000000000000000";
constant p0_37: signed(L-1 downto 0):= "0001000100101000000100100111111100001000100001000000000000000000";
constant p0_38: signed(L-1 downto 0):= "0001000001110000001000101110111011010011100111000000000000000000";
constant p0_39: signed(L-1 downto 0):= "0000111111000111011110000101101111010001001100100000000000000000";
constant p0_40: signed(L-1 downto 0):= "0000111100101100001110110100101010101000101001110000000000000000";
constant p0_41: signed(L-1 downto 0):= "0000111010011100101111010110111110101101011100100000000000000000";
constant p0_42: signed(L-1 downto 0):= "0000111000011000000010111111101110100000010110000000000000000000";
constant p0_43: signed(L-1 downto 0):= "0000110100111001010010011100000101011110101000100000000000000000";
constant p0_44: signed(L-1 downto 0):= "0000101001001001101011100101011000111010111010100000000000000000";
constant p0_45: signed(L-1 downto 0):= "0000110010111110010100000001001110101100011000100000000000000000";
constant p0_46: signed(L-1 downto 0):= "0000110001011001110011011101111000001110001101010000000000000000";
constant p0_47: signed(L-1 downto 0):= "0000101111111011000001100011101100100101101001000000000000000000";
constant p0_48: signed(L-1 downto 0):= "0000101101111001001000010001000001001111000100010000000000000000";
constant p0_49: signed(L-1 downto 0):= "0000101011011011011000110000001000111101010111000000000000000000";
constant p0_50: signed(L-1 downto 0):= "0000101001001110010001010001101110000010000011010000000000000000";
constant p0_51: signed(L-1 downto 0):= "0000100111001111010000100101111010110101010000010000000000000000";
constant p0_52: signed(L-1 downto 0):= "0000100101011100010100100110010110001110101111111000000000000000";
constant p0_53: signed(L-1 downto 0):= "0000100011100111100110010001100001001010111111110000000000000000";
constant p0_54: signed(L-1 downto 0):= "0000100010010100010100110000110100000100010001111000000000000000";
constant p0_55: signed(L-1 downto 0):= "0000100000111001100001011000011111100100001111000000000000000000";
constant p0_56: signed(L-1 downto 0):= "0000011111101100001001110111010000100111101110100000000000000000";
constant p0_57: signed(L-1 downto 0):= "0000011101000011011001111000101110111011100010110000000000000000";
constant p0_58: signed(L-1 downto 0):= "0000011101011100101101110000010001101001010010000000000000000000";
constant p0_59: signed(L-1 downto 0):= "0000011100011100100111001100001001110000101111011000000000000000";
constant p0_60: signed(L-1 downto 0):= "0000011011100000111000100010111101010111111110001000000000000000";
constant p0_61: signed(L-1 downto 0):= "0000011010101001000101111100011011111101110101111000000000000000";
constant p0_62: signed(L-1 downto 0):= "0000011001110100111011000000101110101111011111000000000000000000";
constant p0_63: signed(L-1 downto 0):= "0000011001000011110110011011101110111111111010000000000000000000";
constant p0_64: signed(L-1 downto 0):= "0000011000000000000100100100101000101001000010111000000000000000";
constant p0_65: signed(L-1 downto 0):= "0000010110110111000001101101000101001000011101111000000000000000";
constant p0_66: signed(L-1 downto 0):= "0000010101110010010101001100010000100100001111110100000000000000";
constant p0_67: signed(L-1 downto 0):= "0000010100010001010010111101100011000000101011010000000000000000";
constant p0_68: signed(L-1 downto 0):= "0000010011100110110010100101011010011100001110000000000000000000";
constant p0_69: signed(L-1 downto 0):= "0000010010110000011001010101000010001001110010111000000000000000";
constant p0_70: signed(L-1 downto 0):= "0000010001111110101100110010100010111100001100011000000000000000";
constant p0_71: signed(L-1 downto 0):= "0000010001010001001110011101111101111010111000010101100000000000";
constant p0_72: signed(L-1 downto 0):= "0000010000100111001001000000111111001010000010001000000000000000";
constant p0_73: signed(L-1 downto 0):= "0000010000000000001011101100100011001111100010101000000000000000";
constant p0_74: signed(L-1 downto 0):= "0000001111011100011011100111000011100011011001001000000000000000";
constant p0_75: signed(L-1 downto 0):= "0000001110111011000010001011110100000111110001001100000000000000";
constant p0_76: signed(L-1 downto 0):= "0000001110011011111010010101001011000010111011100000000000000000";
```

```
constant p0_77: signed(L-1 downto 0):= "0000001101111110110101100000110001000010001100011000000000000000";
constant p0_78: signed(L-1 downto 0):= "0000001101100011100111000100101100110001001010000000000000000000";
constant p0_79: signed(L-1 downto 0):= "0000001101001010000011111100100111011110010100000100000000000000";
constant p0_80: signed(L-1 downto 0):= "0000001100100110101110001001010101101100101101011110000000000000";
constant p0_81: signed(L-1 downto 0):= "0000001011111101111110010001100000110110100011000010000000000000";
constant p0_82: signed(L-1 downto 0):= "0000001011010101101010001111100100010011001111100100000000000000";
constant p0_83: signed(L-1 downto 0):= "0000001010110011001011110011100110011101010111001000000000000000";
constant p0_84: signed(L-1 downto 0):= "0000001010010011111110001011100111111100101011001100000000000000";
constant p0_85: signed(L-1 downto 0):= "0000001001110111100100110000011101001000100111011010000000000000";
constant p0_86: signed(L-1 downto 0):= "0000001001011101100111111100011010010111010101101000000000000000";
constant p0_87: signed(L-1 downto 0):= "0000001001000101110100000111100101111111111001110100000000000000";
constant p0_88: signed(L-1 downto 0):= "0000001000101111111000110100011101000110000001100010000000000000";
constant p0_89: signed(L-1 downto 0):= "0000001000011011101000001000100111011001011100001100000000000000";
constant p0_90: signed(L-1 downto 0):= "0000001000001000110110001110010100100100010100011101000000000000";
constant p0_91: signed(L-1 downto 0):= "0000000111101111011000111100100100101101111011001100000000000000";
constant p0_92: signed(L-1 downto 0):= "0000000111001110001111001000011101101011001000011000000000000000";
constant p0_93: signed(L-1 downto 0):= "0000000111010111111010100000111110101100000001001100000000000000";
constant p0_94: signed(L-1 downto 0):= "0000000111001100101110011010001100101000100110001000000000000000";
constant p0_95: signed(L-1 downto 0):= "0000000110111100010011101111010111111001111110001100000000000000";
constant p0_96: signed(L-1 downto 0):= "0000000110101001110100011101011100110011010111011010000000000000";
constant p0_97: signed(L-1 downto 0):= "0000000110010011011011101100001111001110000000111000000000000000";
constant p0_98: signed(L-1 downto 0):= "0000000101111111011000111100000001101101001101111010000000000000";
constant p0_99: signed(L-1 downto 0):= "0000000101101101010101011101010101100011000100111000000000000000";
constant p0_100: signed(L-1 downto 0):= "0000000101011100111111100010010000100101111010110001000000000000";
constant p0_101: signed(L-1 downto 0):= "0000000101001110000110110010111000100000110111010000000000000000";
constant p0_102: signed(L-1 downto 0):= "0000000101000001000000101100001000000001011111010110000000000000";
constant p0_103: signed(L-1 downto 0):= "0000000100110100000001011111001111111000001111010111000000000000";
constant p0_104: signed(L-1 downto 0):= "0000000100101000100001101000101000101111010101011111000000000000";
constant p0_105: signed(L-1 downto 0):= "0000000100011101111001100001100100010001010000011110000000000000";
constant p0_106: signed(L-1 downto 0):= "0000000100010100000010111101000001010010100101101100000000000000";
constant p0_107: signed(L-1 downto 0):= "0000000100001010111000101100101001001011011100000111000000000000";
constant p0_108: signed(L-1 downto 0):= "0000000100000010010110001000000000011000110110100110000000000000";
constant p0_109: signed(L-1 downto 0):= "0000000011111010010111010011101001100110101000100010100000000000";
constant p0_110: signed(L-1 downto 0):= "0000000011110010111000110011010010010000010110101001000000000000";
constant p0_111: signed(L-1 downto 0):= "0000000011101011110111100110001101100010000100110101000000000000";
constant p0_112: signed(L-1 downto 0):= "0000000011100010001001111011101111000010110001001010000000000000";
constant p0_113: signed(L-1 downto 0):= "0000000011010110011001000011110001100111101000100100000000000000";
constant p0_114: signed(L-1 downto 0):= "0000000011001011110110101111010100101101011000101101000000000000";
constant p0_115: signed(L-1 downto 0):= "0000000011000010011100100011101100101101101001110100000000000000";
constant p0_116: signed(L-1 downto 0):= "0000000010111001110000101100110101010100001010100101000000000000";
constant p0_117: signed(L-1 downto 0):= "0000000010110001111011100111011110100011111111101000000000000000";
constant p0_118: signed(L-1 downto 0):= "0000000010101010110001011100110010111110101110110000000000000000";
constant p0_119: signed(L-1 downto 0):= "0000000010100100001100110101110100101011000110011111000000000000";
constant p0_120: signed(L-1 downto 0):= "0000000010011110001001010010101111001010100110101010100000000000";
constant p0_121: signed(L-1 downto 0):= "0000000010011000100011000000001011001100111011111111000000000000";
constant p0_122: signed(L-1 downto 0):= "0000000010010011010110111011110000001001010100101100000000000000";
constant p0_123: signed(L-1 downto 0):= "0000000010001110100001101101011111011111010110011001000000000000";
constant p0_124: signed(L-1 downto 0):= "0000000010001010000011000101101001100100010100101010000000000000";
constant p0_125: signed(L-1 downto 0):= "0000000010000101110100001010010101101101000110111110000000000000";
constant p0_126: signed(L-1 downto 0):= "0000000010000001110111110000100010111111000001100000000000000000";
constant p0_127: signed(L-1 downto 0):= "0000000001111110001010110000011000101001101010001010000000000000";
constant p0_128: signed(L-1 downto 0):= "0000000001110010000101010101000110001100101101110100000000000000";
constant p0_129: signed(L-1 downto 0):= "0000000001100101101010010111010010100001111110010010000000000000";
constant p0_130: signed(L-1 downto 0):= "0000000001101101010001000001011001111010001011010000000000000000";
constant p0_131: signed(L-1 downto 0):= "0000000001101000001111111110010101100000001101100100000000000000";
constant p0_132: signed(L-1 downto 0):= "0000000001100011101100001000010111011000110011001000000000000000";
constant p0_133: signed(L-1 downto 0):= "0000000001011111100100000011001001111010010001111101000000000000";
constant p0_134: signed(L-1 downto 0):= "0000000001011011110001101100001000111101011101100010000000000000";
constant p0_135: signed(L-1 downto 0):= "0000000001011000010011000111101110100111011001000000000000000000";
constant p0_136: signed(L-1 downto 0):= "0000000001010101000101111110100110100011100100100000000000000000";
constant p0_137: signed(L-1 downto 0):= "0000000001010010001000010000110001111001111001111100000000000000";
constant p0_138: signed(L-1 downto 0):= "0000000001001111011000010001100010100100110111001000000000000000";
constant p0_139: signed(L-1 downto 0):= "0000000001001100110100100010100011100010111100010011000000000000";
constant p0_140: signed(L-1 downto 0):= "0000000001001010010111110100001110111101011110111010000000000000";
constant p0_141: signed(L-1 downto 0):= "0000000001001000001101000000011100111010010000011110000000000000";
constant p0_142: signed(L-1 downto 0):= "0000000001000110000111001010011001010110100010010100100000000000";
constant p0_143: signed(L-1 downto 0):= "0000000001000100001001011100110101011010000001001111111000000000";
constant p0_144: signed(L-1 downto 0):= "0000000001000001011011010111001111111101011001000000000000000000";
constant p0_145: signed(L-1 downto 0):= "0000000000111110001000101111001101111101001000101111111111111110";
constant p0_146: signed(L-1 downto 0):= "0000000000111011001011001110000011001100111001111110111111111110";
constant p0_147: signed(L-1 downto 0):= "0000000000111000100000101000110011100011001111100010111111111110";
constant p0_148: signed(L-1 downto 0):= "0000000000110110000101111110011100110101101110111011111111111110";
constant p0_149: signed(L-1 downto 0):= "0000000000110011111001000100100010100011100100000011111111111110";
constant p0_150: signed(L-1 downto 0):= "0000000000110001111000001000101100001000011001010111111111111110";
constant p0_151: signed(L-1 downto 0):= "0000000000110000000001101011101110000011011011101011111111111110";
constant p0_152: signed(L-1 downto 0):= "0000000000101110010100011101101100000011001001000001011111111110";
constant p0_153: signed(L-1 downto 0):= "0000000000101100101111010101111010011100101000110101111111111110";
constant p0_154: signed(L-1 downto 0):= "0000000000101011010001101001111010000010000101011010111111111110";
constant p0_155: signed(L-1 downto 0):= "0000000000101001111010011001001001000100110000100101111111111110";
constant p0_156: signed(L-1 downto 0):= "0000000000101000101000111110000010000100100000000010011111111110";
constant p0_157: signed(L-1 downto 0):= "0000000000100111011001100111011111000001110110101011111111111110";
constant p0_158: signed(L-1 downto 0):= "0000000000100110010101011001111111010100011100101101111111111110";
constant p0_159: signed(L-1 downto 0):= "0000000000100101010010010100100001011111101101111111111111110";
constant p0_160: signed(L-1 downto 0):= "0000000000100011110101011000111000011000111011111110011111111110";
constant p0_161: signed(L-1 downto 0):= "0000000000100010000101001011001110100101111101010100111111111110";
constant p0_162: signed(L-1 downto 0):= "0000000000100000011111101000001101110111001001111011111111111110";
constant p0_163: signed(L-1 downto 0):= "0000000000011111000100011011010000011100101001100111111111111110";
constant p0_164: signed(L-1 downto 0):= "0000000000011101110001101101010000101110110110001111111111111110";
constant p0_165: signed(L-1 downto 0):= "0000000000011100100110011011010000011000111010000011111111111110";
constant p0_166: signed(L-1 downto 0):= "0000000000011011000010100100000110010001010010100011111111111110";
constant p0_167: signed(L-1 downto 0):= "0000000000011010100001110101111101110111111011111010111111111110";
constant p0_168: signed(L-1 downto 0):= "0000000000011001100111010100111010010101100000010111111111111110";
constant p0_169: signed(L-1 downto 0):= "0000000000011000110001001010111011111000100110110110111111111110";
constant p0_170: signed(L-1 downto 0):= "0000000000010111111111011001011010001010011000010000111111111110";
constant p0_171: signed(L-1 downto 0):= "0000000000010111010000000110000011110011001010111001111111111110";
constant p0_172: signed(L-1 downto 0):= "0000000000010110100100011010001111001101100101000011011111111110";
constant p0_173: signed(L-1 downto 0):= "0000000000010101111011100010010000000110011101111100111111111110";
constant p0_174: signed(L-1 downto 0):= "0000000000010101010100111010000000100100010001010101111111111110";
constant p0_175: signed(L-1 downto 0):= "0000000000010100110001001011011101010000101111011110111111111110";
constant p0_176: signed(L-1 downto 0):= "0000000000010011111111101000001100111100001000010001111111111110";
constant p0_177: signed(L-1 downto 0):= "0000000000010011000010101100100100111010010110100111111111111110";
constant p0_178: signed(L-1 downto 0):= "0000000000010010001100010110100011011011101000001011111111111110";
constant p0_179: signed(L-1 downto 0):= "0000000000010001011011010011000010010101100010110000111111111110";
constant p0_180: signed(L-1 downto 0):= "0000000000010001011101100100000110101110001001111101111111111110";
constant p0_181: signed(L-1 downto 0):= "0000000000010000000110001100010111001001110100010110111111111110";
constant p0_182: signed(L-1 downto 0):= "0000000000001111100001000001100011000101000111010110111111111110";
constant p0_183: signed(L-1 downto 0):= "0000000000001110111101101001010011001111010010101011111111111110";
constant p0_184: signed(L-1 downto 0):= "0000000000001110011111010100110001011110000010101100011111111110";
constant p0_185: signed(L-1 downto 0):= "0000000000001110000010001000110111111011110000011101111111111110";
constant p0_186: signed(L-1 downto 0):= "0000000000001101100111000010011111110100001100011000111111111110";
constant p0_187: signed(L-1 downto 0):= "0000000000001101001101110011100100111100111010100101111111111110";
constant p0_188: signed(L-1 downto 0):= "0000000000001100110110001111111111010000110110111001111111111110";
```

```
constant p0_189: signed(L-1 downto 0):= "000000000000011001000000011010010101000100011111111010111111111110";
constant p0_190: signed(L-1 downto 0):= "000000000000110000101110000111110100111011010011110011111111111110";
constant p0_191: signed(L-1 downto 0):= "000000000000101111100000011001001110101111111000011011111111111110";
constant p0_192: signed(L-1 downto 0):= "000000000000101101101001010101010011111100010000111111111111111110";
constant p0_193: signed(L-1 downto 0):= "000000000000101011110001111110100110101111001111110001111111111110";
constant p0_194: signed(L-1 downto 0):= "000000000000101001111001011010000010010011111111001011111111111110";
constant p0_195: signed(L-1 downto 0):= "000000000000101000010010110110011110000001100010001001111111111110";
constant p0_196: signed(L-1 downto 0):= "000000000000100111010010110011110011101000011110001111111111111110";
constant p0_197: signed(L-1 downto 0):= "000000000000100101011011010000100111101101001100101011111111111110";
constant p0_198: signed(L-1 downto 0):= "000000000000100100001011000111000111111011000100001111111111111110";
constant p0_199: signed(L-1 downto 0):= "000000000000100011000001011010001111111111101100011111111111111110";
constant p0_200: signed(L-1 downto 0):= "000000000000100001111011000011011000100011001100001111111111111110";
constant p0_201: signed(L-1 downto 0):= "000000000000100000111101010110101111011011111010111111111111111110";
constant p0_202: signed(L-1 downto 0):= "000000000000100000000100010110110100000000010110101011111111111110";
constant p0_203: signed(L-1 downto 0):= "000000000000011111001110000101101010100011100111101111111111111110";
constant p0_204: signed(L-1 downto 0):= "000000000000011110011011101101110110011001111110000111111111111110";
constant p0_205: signed(L-1 downto 0):= "000000000000011101101100001000010001010111111000111111111111111110";
constant p0_206: signed(L-1 downto 0):= "000000000000011100111111110001011001011010000011111111111111111110";
constant p0_207: signed(L-1 downto 0):= "000000000000011100010110000111111000000111001101110111111111111110";
constant p0_208: signed(L-1 downto 0):= "000000000000011011011100011101010011011100010111000011111111111110";
constant p0_209: signed(L-1 downto 0):= "000000000000011010010110101011101111111110011000100111111111111110";
constant p0_210: signed(L-1 downto 0):= "000000000000011001011000001101110001000111011001011111111111111110";
constant p0_211: signed(L-1 downto 0):= "000000000000011000011111111111111000000110100110110011111111111110";
constant p0_212: signed(L-1 downto 0):= "000000000000010111101101001001110111100010110010110001111111111110";
constant p0_213: signed(L-1 downto 0):= "000000000000010110111101111001100101010110001100110111111111110";
constant p0_214: signed(L-1 downto 0):= "000000000000010110010100110111100010101000110010100110111111111110";
constant p0_215: signed(L-1 downto 0):= "000000000000010101101110010101110001101110101011100111111111110";
constant p0_216: signed(L-1 downto 0):= "000000000000010100101011111010010111111100110010011111111111110";
constant p0_217: signed(L-1 downto 0):= "000000000000010100101010011011100010010000100100011111111111110";
constant p0_218: signed(L-1 downto 0):= "000000000000010100001100011001011101000000110010011101111111110";
constant p0_219: signed(L-1 downto 0):= "000000000000010011100001001111101010111001000101111111111111110";
constant p0_220: signed(L-1 downto 0):= "000000000000010011010110111000110000000111011011101111111111110";
constant p0_221: signed(L-1 downto 0):= "000000000000010010111101111111010100000010101111101111111111110";
constant p0_222: signed(L-1 downto 0):= "000000000000010010100100011000111101010110110100011011111111110";
constant p0_223: signed(L-1 downto 0):= "000000000000010010010100001110001001101100011001110011111111110";
constant p0_224: signed(L-1 downto 0):= "000000000000010001110111110100010100010011001000101111111111110";
constant p0_225: signed(L-1 downto 0):= "000000000000010001010110001010010010111001001010110011111111110";
constant p0_226: signed(L-1 downto 0):= "000000000000010000111000101111110011101100100100111111111111110";
constant p0_227: signed(L-1 downto 0):= "000000000000010000011110000000001001010010010100100111111111110";
constant p0_228: signed(L-1 downto 0):= "000000000000010000001000011101110110110011000110101111111111110";
constant p0_229: signed(L-1 downto 0):= "000000000000001111110100110001000100011110100001000111111111110";
constant p0_230: signed(L-1 downto 0):= "000000000000001111100011100110010111001011011010001111111111110";
constant p0_231: signed(L-1 downto 0):= "000000000000001110101001011011101100100000010011001111111111110";
constant p0_232: signed(L-1 downto 0):= "000000000000001110001111101010000110011000000000100111111111110";
constant p0_233: signed(L-1 downto 0):= "000000000000001110110001111010000111000000000010101001111111111110";
constant p0_234: signed(L-1 downto 0):= "000000000000001110110011111010110111010011010011111111111111110";
constant p0_235: signed(L-1 downto 0):= "000000000000001110101100011110111010010111100100011111111111110";
constant p0_236: signed(L-1 downto 0):= "000000000000001110100110101000010111010101000100111111111111110";
constant p0_237: signed(L-1 downto 0):= "000000000000001110100010010010111111111111100101001111111111110";
constant p0_238: signed(L-1 downto 0):= "000000000000001110011110101111000000000111000100010111111111110";
constant p0_239: signed(L-1 downto 0):= "000000000000001110011100000001001110001000100010000111111111110";
constant p1_0: signed(L-1 downto 0):= "111111111111100110101010010001110011101010011101001100000000000000";
constant p1_1: signed(L-1 downto 0):= "111111111111001101011100000001001010111101010011100000000000000";
constant p1_2: signed(L-1 downto 0):= "111111111111001101100110111010101000011101011001110000000000000";
constant p1_3: signed(L-1 downto 0):= "111111111111001101101111011010001100000011100000100000000000000";
constant p1_4: signed(L-1 downto 0):= "111111111111001101101110001011010110100011011000000000000000000";
constant p1_5: signed(L-1 downto 0):= "111111111111001101111111010001000101111111100001000000000000000";
constant p1_6: signed(L-1 downto 0):= "111111111111001100001101010011100001000110001111000000000000000";
constant p1_7: signed(L-1 downto 0):= "111111111111001100011010111011000010100011110101000000000000000";
constant p1_8: signed(L-1 downto 0):= "111111111111001100101001000011011010101000110101011000000000000";
constant p1_9: signed(L-1 downto 0):= "111111111111001100110110001000000010110001001001101000000000000";
constant p1_10: signed(L-1 downto 0):= "111111111111001101000010101111010100000111010001000000000000000";
constant p1_11: signed(L-1 downto 0):= "111111111111001101001110110111010001000011001001000000000000000";
constant p1_12: signed(L-1 downto 0):= "111111111111001101101010011000011001000011100111100000000000000";
constant p1_13: signed(L-1 downto 0):= "111111111111001101100101111000011010001100011110010000000000000";
constant p1_14: signed(L-1 downto 0):= "111111111111001101110000111011001101000001100010010000000000000";
constant p1_15: signed(L-1 downto 0):= "111111111111001110111011100100100011101000101001100000000000000";
constant p1_16: signed(L-1 downto 0):= "111111111111001110001010110000110011101101011011010000000000000";
constant p1_17: signed(L-1 downto 0):= "111111111111001110011100100100011011100100101110010000000000000";
constant p1_18: signed(L-1 downto 0):= "111111111111001110100001100010011011000001101100010000000000000";
constant p1_19: signed(L-1 downto 0):= "111111111111001111000010010100110000100100011010110000000000000";
constant p1_20: signed(L-1 downto 0):= "111111111111001111010011000010100000001101001001010000000000000";
constant p1_21: signed(L-1 downto 0):= "111111111111001111100010111111010110011000000100000000000000000";
constant p1_22: signed(L-1 downto 0):= "111111111111001111111001001000010001011110101111110000000000000";
constant p1_23: signed(L-1 downto 0):= "111111111111010000000000011100011100110110110110100000000000000";
constant p1_24: signed(L-1 downto 0):= "111111111111010000000111011110000001101111000100100000000000000";
constant p1_25: signed(L-1 downto 0):= "111111111111010000011100011100111011001010110001100000000000000";
constant p1_26: signed(L-1 downto 0):= "111111111111010000010100101111000011001001111011000000000000000";
constant p1_27: signed(L-1 downto 0):= "111111111111010000011011000000111100001000000000010000000000000";
constant p1_28: signed(L-1 downto 0):= "111111111111010000100001000101001010011011000000100000000000000";
constant p1_29: signed(L-1 downto 0):= "111111111111010000100110111001010001111110100111101000000000000";
constant p1_30: signed(L-1 downto 0):= "111111111111010000101100101000001101010010011111100000000000000";
constant p1_31: signed(L-1 downto 0):= "111111111111010000110010001000100111011100011011101000000000000";
constant p1_32: signed(L-1 downto 0):= "111111111111010000111001111111000111011010011001010000000000000";
constant p1_33: signed(L-1 downto 0):= "111111111111010001000100000101111100111011111001100000000000000";
constant p1_34: signed(L-1 downto 0):= "111111111111010001001101101010000110101011001111000000000000000";
constant p1_35: signed(L-1 downto 0):= "111111111111010001010110101111100011111100100010100100000000000";
constant p1_36: signed(L-1 downto 0):= "111111111111010001011110011001010110001011110101110000000000000";
constant p1_37: signed(L-1 downto 0):= "111111111111010001100111010100000010001010011010110100000000000";
constant p1_38: signed(L-1 downto 0):= "111111111111010001101111000111010111001000001011110000000000000";
constant p1_39: signed(L-1 downto 0):= "111111111111010001110110011000011000001100001110110000000000000";
constant p1_40: signed(L-1 downto 0):= "111111111111010001111100110100111100010111010110111000000000000";
constant p1_41: signed(L-1 downto 0):= "111111111111010010000101011010011111100000011100101100000000000";
constant p1_42: signed(L-1 downto 0):= "111111111111010010001100001010001001101100000001011000000000000";
constant p1_43: signed(L-1 downto 0):= "111111111111010010010010101010100010101110110100010000000000000";
constant p1_44: signed(L-1 downto 0):= "111111111111010010011000111101000000011001111111011100000000000";
constant p1_45: signed(L-1 downto 0):= "111111111111010010011110000100011011010010001001000000000000000";
constant p1_46: signed(L-1 downto 0):= "111111111111010010100100111011000111111001101000100100000000000";
constant p1_47: signed(L-1 downto 0):= "111111111111010010101010101000100000010100010000010000000000000";
constant p1_48: signed(L-1 downto 0):= "111111111111010010110010110010000010011001100110111000000000000";
constant p1_49: signed(L-1 downto 0):= "111111111111010010111101010000011100010011001011011100000000000";
constant p1_50: signed(L-1 downto 0):= "111111111111010011000111001011011011111000001001011000000000000";
constant p1_51: signed(L-1 downto 0):= "111111111111010011010000100110110001001100000111111000000000000";
constant p1_52: signed(L-1 downto 0):= "111111111111010011011001100101011101000100111110111000000000000";
constant p1_53: signed(L-1 downto 0):= "111111111111010011100010001010001100110001010101011100000000000";
constant p1_54: signed(L-1 downto 0):= "111111111111010011101010010111010011111001001101100000000000000";
constant p1_55: signed(L-1 downto 0):= "111111111111010011100100011101100110100110111100010000000000000";
constant p1_56: signed(L-1 downto 0):= "111111111111010011111001100100111011110110011000000000000000000";
constant p1_57: signed(L-1 downto 0):= "111111111111010100000010000111100010000101001000100000000000000";
constant p1_58: signed(L-1 downto 0):= "111111111111010100001000000100001010111100000110111000000000000";
constant p1_59: signed(L-1 downto 0):= "111111111111010100001110110100111000000110111010010000000000000";
constant p1_60: signed(L-1 downto 0):= "111111111111010100010101010111011110100111010011110100000000000";
```

```
constant p1_61: signed(L-1 downto 0):= "1111111111111010100011011010110111010111010011000010000000000000";
constant p1_62: signed(L-1 downto 0):= "1111111111111010100100001110011001101011110011000010000000000000";
constant p1_63: signed(L-1 downto 0):= "1111111111111010100100111011111000001111101010000000000000000000";
constant p1_64: signed(L-1 downto 0):= "1111111111111010100110000001101010000110101111001110100000000000";
constant p1_65: signed(L-1 downto 0):= "1111111111111010100111011000110011010101110111100001000000000000";
constant p1_66: signed(L-1 downto 0):= "1111111111111010101000101101011011111101101110001000000000000000";
constant p1_67: signed(L-1 downto 0):= "1111111111111010101001110011101001100101010011111110000000000000";
constant p1_68: signed(L-1 downto 0):= "1111111111111010101011000100100100111011100000110101000000000000";
constant p1_69: signed(L-1 downto 0):= "1111111111111010101100001011111110000111010000010101000000000000";
constant p1_70: signed(L-1 downto 0):= "1111111111111010101101010000001001101011010110011010100000000000";
constant p1_71: signed(L-1 downto 0):= "1111111111111010101110010001110101010101110011000111000000000000";
constant p1_72: signed(L-1 downto 0):= "1111111111111010101111010000110010101010110010100110000000000000";
constant p1_73: signed(L-1 downto 0):= "1111111111111010110000001101011000001101000111001110000000000000";
constant p1_74: signed(L-1 downto 0):= "1111111111111010110001000111110001010110011011100111000000000000";
constant p1_75: signed(L-1 downto 0):= "1111111111111010110010000000001000010000011010000010100000000000";
constant p1_76: signed(L-1 downto 0):= "1111111111111010110010110110100110000000000001111011000000000000";
constant p1_77: signed(L-1 downto 0):= "1111111111111010110011101011010010101111000001000100100000000000";
constant p1_78: signed(L-1 downto 0):= "1111111111111010110100011110010101110011010010001000000000000000";
constant p1_79: signed(L-1 downto 0):= "1111111111111010110101001111110101111010100010110000000000000000";
constant p1_80: signed(L-1 downto 0):= "1111111111111010110110010110100001011011101101101100000000000000";
constant p1_81: signed(L-1 downto 0):= "1111111111111010110111100010110101101110100010101010000000000000";
constant p1_82: signed(L-1 downto 0):= "1111111111111010111001000111001000000110000110111100000000000000";
constant p1_83: signed(L-1 downto 0):= "1111111111111010111010011001011100010101011100101110000000000000";
constant p1_84: signed(L-1 downto 0):= "1111111111111010111011100111011100100101010010000100000000000000";
constant p1_85: signed(L-1 downto 0):= "1111111111111010111100110010000001001000101111000001100000000000";
constant p1_86: signed(L-1 downto 0):= "1111111111111010111101011100101100001011110100011001000000000000";
constant p1_87: signed(L-1 downto 0):= "1111111111111010111110111101101010101111010101001100000000000000";
constant p1_88: signed(L-1 downto 0):= "1111111111111010111111111111110011101000100100000000010000000000";
constant p1_89: signed(L-1 downto 0):= "1111111111111011000000111101110101101000011110100000000000000000";
constant p1_90: signed(L-1 downto 0):= "1111111111111011000001110111111011001111001010000010000000000000";
constant p1_91: signed(L-1 downto 0):= "1111111111111011000010110101110000011010111010000000000000000000";
constant p1_92: signed(L-1 downto 0):= "1111111111111011000011011111101010100001010010000110000000000000";
constant p1_93: signed(L-1 downto 0):= "1111111111111011000100100111011110010010001110110100100000000000";
constant p1_94: signed(L-1 downto 0):= "1111111111111011000101011100010101111110000010001011000000000000";
constant p1_95: signed(L-1 downto 0):= "1111111111111011000110010000001000011001110111100000000000000000";
constant p1_96: signed(L-1 downto 0):= "1111111111111011000111011010000110000101111000001010000000000000";
constant p1_97: signed(L-1 downto 0):= "1111111111111011001000111001001110011000001100010110000000000000";
constant p1_98: signed(L-1 downto 0):= "1111111111111011001010010011011101110110000010111100000000000000";
constant p1_99: signed(L-1 downto 0):= "1111111111111011001011010010011000010001111010101010000000000000";
constant p1_100: signed(L-1 downto 0):= "1111111111111011001100111011010111100001011101101100111000000000000";
constant p1_101: signed(L-1 downto 0):= "1111111111111011001110001001000011101000111010011100000000000000";
constant p1_102: signed(L-1 downto 0):= "1111111111111011001111010011110111000110110010000001100000000000";
constant p1_103: signed(L-1 downto 0):= "1111111111111011010000011011101000100001111101000001000000000000";
constant p1_104: signed(L-1 downto 0):= "1111111111111011010001100000100111101001101000100110000000000000";
constant p1_105: signed(L-1 downto 0):= "1111111111111011010010100011000010010101110100100010000000000000";
constant p1_106: signed(L-1 downto 0):= "1111111111111011010011100011000100111001101100000101100000000000";
constant p1_107: signed(L-1 downto 0):= "1111111111111011010100100000111010010010010011111000000000000000";
constant p1_108: signed(L-1 downto 0):= "1111111111111011010101011100101100010010110000000001100000000000";
constant p1_109: signed(L-1 downto 0):= "1111111111111011010110010110100011101100011000000101100000000000";
constant p1_110: signed(L-1 downto 0):= "1111111111111011010111001110101000100001000100011100000000000000";
constant p1_111: signed(L-1 downto 0):= "1111111111111011011000000101000001110110011000110101000000000000";
constant p1_112: signed(L-1 downto 0):= "1111111111111011011001010010101111001010001100101110100000000000";
constant p1_113: signed(L-1 downto 0):= "1111111111111011011010110110101110100111011011011000000000000000";
constant p1_114: signed(L-1 downto 0):= "1111111111111011011100010101100011011101111011000010000000000000";
constant p1_115: signed(L-1 downto 0):= "1111111111111011011101110111111011010101101100100100000000000000";
constant p1_116: signed(L-1 downto 0):= "1111111111111011011110001011100000000111010000011011000000000000";
constant p1_117: signed(L-1 downto 0):= "1111111111111011011000000101111111010110111011010001100000000000";
constant p1_118: signed(L-1 downto 0):= "1111111111111011100001100110101101010001010110100000100000000000";
constant p1_119: signed(L-1 downto 0):= "1111111111111011100010110010010010001100011000010100000000000000";
constant p1_120: signed(L-1 downto 0):= "1111111111111011100011110101111001101010010010001000100000000000";
constant p1_121: signed(L-1 downto 0):= "1111111111111011100101000000111011011110100110101011000000000000";
constant p1_122: signed(L-1 downto 0):= "1111111111111011100110000100011010111110101011110101000000000000";
constant p1_123: signed(L-1 downto 0):= "1111111111111011100111000101100101101001000110100110100000000000";
constant p1_124: signed(L-1 downto 0):= "1111111111111011101000000100101001000111100011000101000000000000";
constant p1_125: signed(L-1 downto 0):= "1111111111111011101001000001101011001010010101101111000000000000";
constant p1_126: signed(L-1 downto 0):= "1111111111111011101001111001101010011010111100110110000000000000";
constant p1_127: signed(L-1 downto 0):= "1111111111111011101010110110001110101111111110101110000000000000";
constant p1_128: signed(L-1 downto 0):= "1111111111111011101100001000010000001101010111100010100000000000";
constant p1_129: signed(L-1 downto 0):= "1111111111111011101101100011101010110101011010110111000000000000";
constant p1_130: signed(L-1 downto 0):= "1111111111111011101111010110000000010010001100010100000000000000";
constant p1_131: signed(L-1 downto 0):= "1111111111111011110001101010101000011010110011110001000000000000";
constant p1_132: signed(L-1 downto 0):= "1111111111111011110010010000001111010111001101011111000000000000";
constant p1_133: signed(L-1 downto 0):= "1111111111111011110011001110010111010101110101100000000000000000";
constant p1_134: signed(L-1 downto 0):= "1111111111111011110100111010011111100101001111001110100000000000";
constant p1_135: signed(L-1 downto 0):= "1111111111111011111010001010011110101010100010111100000000000000";
constant p1_136: signed(L-1 downto 0):= "1111111111111011111011010111010100001010111100101100000000000000";
constant p1_137: signed(L-1 downto 0):= "1111111111111011111000100001100000111111101011010110000000000000";
constant p1_138: signed(L-1 downto 0):= "1111111111111011111001101001000000110011100111100111000000000000";
constant p1_139: signed(L-1 downto 0):= "1111111111111011111010101110000101011111110010100001100000000000";
constant p1_140: signed(L-1 downto 0):= "1111111111111011111011110000111001110000110010010111100000000000";
constant p1_141: signed(L-1 downto 0):= "1111111111111011111100110001100111001110011100000000000000000000";
constant p1_142: signed(L-1 downto 0):= "1111111111111011111101110000010110100100001001001000100000000000";
constant p1_143: signed(L-1 downto 0):= "1111111111111011111111010110100111101000011001010100000000000000";
constant p1_144: signed(L-1 downto 0):= "1111111111111100000000000100010010011011010010000000000000000000";
constant p1_145: signed(L-1 downto 0):= "1111111111111100000001110100011000001110000101111110000000000000";
constant p1_146: signed(L-1 downto 0):= "1111111111111100000011011101100011010000001001111011000000000000";
constant p1_147: signed(L-1 downto 0):= "1111111111111100000101000100000011101110011010110111100000000000";
constant p1_148: signed(L-1 downto 0):= "1111111111111100000110100100101110001011010000001101000000000000";
constant p1_149: signed(L-1 downto 0):= "1111111111111100001000000001001100001100111010010000000000000000";
constant p1_150: signed(L-1 downto 0):= "1111111111111100001001100111010101011001010100010100000000000000";
constant p1_151: signed(L-1 downto 0):= "1111111111111100001010101110111111000101011101000101110000000000";
constant p1_152: signed(L-1 downto 0):= "1111111111111100001100000000111000101100011100111110100000000000";
constant p1_153: signed(L-1 downto 0):= "1111111111111100001101001111110100110101000010001111100000000000";
constant p1_154: signed(L-1 downto 0):= "1111111111111100001110011100000000101011101000001011100000000000";
constant p1_155: signed(L-1 downto 0):= "1111111111111100001111001011010001101001110111101000000000000000";
constant p1_156: signed(L-1 downto 0):= "1111111111111100010000101100111000010100000011011000000000000000";
constant p1_157: signed(L-1 downto 0):= "1111111111111100010001110001111001110111111001000100000000000000";
constant p1_158: signed(L-1 downto 0):= "1111111111111100010010110100110110000001000010110100000000000000";
constant p1_159: signed(L-1 downto 0):= "1111111111111100010011110101110101001101010011000111000000000000";
constant p1_160: signed(L-1 downto 0):= "1111111111111100010101001011000011101010110001010000000000000000";
constant p1_161: signed(L-1 downto 0):= "1111111111111100010111001010011111101011001111010100000000000000";
constant p1_162: signed(L-1 downto 0):= "1111111111111100011000111100001100111010100111001111000000000000";
constant p1_163: signed(L-1 downto 0):= "1111111111111100011010101000011111111001111111111001100000000000";
constant p1_164: signed(L-1 downto 0):= "1111111111111100011100001111111001110110100000000010000000000000";
constant p1_165: signed(L-1 downto 0):= "1111111111111100011101110010110111100000100101001001100000000000";
constant p1_166: signed(L-1 downto 0):= "1111111111111100011111010001110001101011100010010111000000000000";
constant p1_167: signed(L-1 downto 0):= "1111111111111100100000101100111110000011000110010100000000000000";
constant p1_168: signed(L-1 downto 0):= "1111111111111100100010000100101111010000111100010111000000000000";
constant p1_169: signed(L-1 downto 0):= "1111111111111100100011011001010111001101111110000010000000000000";
constant p1_170: signed(L-1 downto 0):= "1111111111111100100100101011000011101100110010001001100000000000";
constant p1_171: signed(L-1 downto 0):= "1111111111111100100101111010000010010110100001010100000000000000";
constant p1_172: signed(L-1 downto 0):= "1111111111111100100111000110011111000011111011010000100000000000";
```

```
    constant p1_173: signed(L-1 downto 0):= "1111111111111100101000010000100100100000111000110110000000000000";
    constant p1_174: signed(L-1 downto 0):= "1111111111111100101001011000011100010110100010101110000000000000";
    constant p1_175: signed(L-1 downto 0):= "1111111111111100101010011110001111010011101111111001000000000000";
    constant p1_176: signed(L-1 downto 0):= "1111111111111100101100000010000101011101010000000111100000000000";
    constant p1_177: signed(L-1 downto 0):= "1111111111111100101110000010110001100101110011101010100000000000";
    constant p1_178: signed(L-1 downto 0):= "1111111111111100101111111101000011001001001000011000000000000000";
    constant p1_179: signed(L-1 downto 0):= "1111111111111100110001110001100011011111101111010001100000000000";
    constant p1_180: signed(L-1 downto 0):= "1111111111111100110011100000110101111010010110111000000000000000";
    constant p1_181: signed(L-1 downto 0):= "1111111111111100110101001011011000111001001110001111100000000000";
    constant p1_182: signed(L-1 downto 0):= "1111111111111100110110110001100110101000100010001101100000000000";
    constant p1_183: signed(L-1 downto 0):= "1111111111111100111000010011110110001001000111011011100000000000";
    constant p1_184: signed(L-1 downto 0):= "1111111111111100111001110010011011100101100100011001100000000000";
    constant p1_185: signed(L-1 downto 0):= "1111111111111100111011001101101000110001010111100010100000000000";
    constant p1_186: signed(L-1 downto 0):= "1111111111111100111100100101101101011111110000000111100000000000";
    constant p1_187: signed(L-1 downto 0):= "1111111111111100111101111010110111110110011010101001100000000000";
    constant p1_188: signed(L-1 downto 0):= "1111111111111100111111001101010100011100111001111000100000000000";
    constant p1_189: signed(L-1 downto 0):= "1111111111111101000000011101001110101001010111111100100000000000";
    constant p1_190: signed(L-1 downto 0):= "1111111111111101000001101010110000101011010000110010000000000000";
    constant p1_191: signed(L-1 downto 0):= "1111111111111101000010110110000011110100001111110110100000000000";
    constant p1_192: signed(L-1 downto 0):= "1111111111111101000100100001110010011001000001100110100000000000";
    constant p1_193: signed(L-1 downto 0):= "1111111111111101000110101100101001001100011110001110000000000000";
    constant p1_194: signed(L-1 downto 0):= "1111111111111101001000110000100100111110101111010010000000000000";
    constant p1_195: signed(L-1 downto 0):= "1111111111111101001010101011100100011100000111010101010000000000";
    constant p1_196: signed(L-1 downto 0):= "1111111111111101001100100110010101000101001101111110010000000000";
    constant p1_197: signed(L-1 downto 0):= "1111111111111101001110011001001111010001000111001100000000000000";
    constant p1_198: signed(L-1 downto 0):= "1111111111111101010000000111011100010100110100000111100000000000";
    constant p1_199: signed(L-1 downto 0):= "1111111111111101010001110001010100101100101101001111100000000000";
    constant p1_200: signed(L-1 downto 0):= "1111111111111101010011010111001101110110010010101000000000000000";
    constant p1_201: signed(L-1 downto 0):= "1111111111111101010100111001011010101110001100001101000000000000";
    constant p1_202: signed(L-1 downto 0):= "1111111111111101010110011000001100001000011100111110100000000000";
    constant p1_203: signed(L-1 downto 0):= "1111111111111101010111110011110001000100011001011101100000000000";
    constant p1_204: signed(L-1 downto 0):= "1111111111111101011001001100010110111001111100010000000000000000";
    constant p1_205: signed(L-1 downto 0):= "1111111111111101011010100010001001110110010011100101100000000000";
    constant p1_206: signed(L-1 downto 0):= "1111111111111101011011110101010100101001000100000100000000000000";
    constant p1_207: signed(L-1 downto 0):= "1111111111111101011101000110000001001011111101010101000000000000";
    constant p1_208: signed(L-1 downto 0):= "1111111111111101011110111001010011101000101111011010000000000000";
    constant p1_209: signed(L-1 downto 0):= "1111111111111101100001001101100111110011011100111110000000000000";
    constant p1_210: signed(L-1 downto 0):= "1111111111111101100011011010001011010000111000101100000000000000";
    constant p1_211: signed(L-1 downto 0):= "1111111111111101100101011111101100100011000010111000100000000000";
    constant p1_212: signed(L-1 downto 0):= "1111111111111101100111011110110011010000101100111011100000000000";
    constant p1_213: signed(L-1 downto 0):= "1111111111111101101001011000000001010111111111011001000000000000";
    constant p1_214: signed(L-1 downto 0):= "1111111111111101101011001011110100001110011010110000000000000000";
    constant p1_215: signed(L-1 downto 0):= "1111111111111101101100111010100101010011000001011001000000000000";
    constant p1_216: signed(L-1 downto 0):= "1111111111111101101110100100101010110110010000010100100000000000";
    constant p1_217: signed(L-1 downto 0):= "1111111111111101110000001010011000011001111010000001100000000000";
    constant p1_218: signed(L-1 downto 0):= "1111111111111101110001101011111111001010111101010100100000000000";
    constant p1_219: signed(L-1 downto 0):= "1111111111111101110011001001101110010110101011110011100000000000";
    constant p1_220: signed(L-1 downto 0):= "1111111111111101110100100011110011011100000000110000100000000000";
    constant p1_221: signed(L-1 downto 0):= "1111111111111101110101111010011010011001111001010000000000000000";
    constant p1_222: signed(L-1 downto 0):= "1111111111111101110111001101101101111011010011010000100000000000";
    constant p1_223: signed(L-1 downto 0):= "1111111111111101111000011101110111100001010000110101000000000000";
    constant p1_224: signed(L-1 downto 0):= "1111111111111101111010001110111110110110101110011111100000000000";
    constant p1_225: signed(L-1 downto 0):= "1111111111111101111100011110000000101110011001111011100000000000";
    constant p1_226: signed(L-1 downto 0):= "1111111111111101111110100010011000101001100100010101000000000000";
    constant p1_227: signed(L-1 downto 0):= "1111111111111110000000001110010101100000011101101101110000000000";
    constant p1_228: signed(L-1 downto 0):= "1111111111111110000010001101010110011110111001110110000000000000";
    constant p1_229: signed(L-1 downto 0):= "1111111111111110000011110100110001100111010110100101000000000000";
    constant p1_230: signed(L-1 downto 0):= "1111111111111110000101010011001100100000011111101011000000000000";
    constant p1_231: signed(L-1 downto 0):= "1111111111111110000110101000110001011101000110110110100000000000";
    constant p1_232: signed(L-1 downto 0):= "1111111111111110000111110101100101011001110011010010100000000000";
    constant p1_233: signed(L-1 downto 0):= "1111111111111110001000111001101000001111101100101010000000000000";
    constant p1_234: signed(L-1 downto 0):= "1111111111111110001001110100110100111110000010001101000000000000";
    constant p1_235: signed(L-1 downto 0):= "1111111111111110001010100111000001101011101010110110000000000000";
    constant p1_236: signed(L-1 downto 0):= "1111111111111110001011001111111111100000111101110010000000000000";
    constant p1_237: signed(L-1 downto 0):= "1111111111111110001011101111011010011010001001101000000000000000";
    constant p1_238: signed(L-1 downto 0):= "1111111111111110001100000100111000110001111001001111100000000000";
    constant p1_239: signed(L-1 downto 0):= "1111111111111110001100001111110110000110111011101111100000000000";

    constant p0: p_vect_type := (p0_239,p0_238,p0_237,p0_236,p0_235,p0_234,p0_233,p0_232,p0_231,p0_230,p0_229,p0_228,p0_227,p0_226,p0_225,p0_224,p0_223,p0_222,p0_221,p0_220,p0_219,p0_218,p0_217,p0_216,p0
    constant p1: p_vect_type := (p1_239,p1_238,p1_237,p1_236,p1_235,p1_234,p1_233,p1_232,p1_231,p1_230,p1_229,p1_228,p1_227,p1_226,p1_225,p1_224,p1_223,p1_222,p1_221,p1_220,p1_219,p1_218,p1_217,p1_216,p1

    constant p_vect_size: integer := 240;
    constant log_segs: integer := 15;
    constant lin_segs: integer := 16;
    constant lin_segs_pow: integer := 4;
    constant Qmn_norm_one: integer := 52;

    constant FHE_q_bitvect: signed(L-1 downto 0) := "0000001000000000000000000000000000000000000000000000000000000000"; --Qm_norm, Qn_norm

    --FHE parameters----------------------------------------------------------------------------------------------------

    constant FHE_m: natural := 3;
    constant FHE_n: natural := 3;
    constant FHE_L: natural := 112;--80;

    constant FHE_CN: natural := (FHE_n+1)*FHE_L;

    constant FHE_Qm: natural := 16;
    constant FHE_Qn: natural := 32;

    type vector_array is array (natural range <>) of signed(FHE_L-1 downto 0);
    type matrix_array is array (natural range <>, natural range <>) of signed(FHE_L-1 downto 0);

    type vector_array_e is array (natural range <>) of signed(FHE_L-1 downto 0);
    type matrix_array_e is array (natural range <>, natural range <>) of signed(FHE_L-1 downto 0);

    type vector_bit_array is array (natural range <>) of std_logic;
    type matrix_bit_array is array (natural range <>, natural range <>) of std_logic;


    constant A00_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111101111001001100110";
    constant A01_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111000101111000001011000";
    constant A02_coef: signed(FHE_L-1 downto 0) := "1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111101111000000000000";
    constant A03_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001010111010";
    constant A04_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001000000000111110110110";
    constant A10_coef: signed(FHE_L-1 downto 0) := "1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111010000100000100000000000";
    constant A11_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000010000110010010101111100111000101";
    constant A12_coef: signed(FHE_L-1 downto 0) := "1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111010000100000100000000000";
    constant A13_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011000100100111110";
    constant A14_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000010101001011100001110101111101";
    constant A20_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000001010000110001101011111011";
    constant A21_coef: signed(FHE_L-1 downto 0) := "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000001010110010100001010101011";
```

```
constant A22_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000001000000010100001100011010111111011";
constant A23_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010100011100011101001000000";
constant A24_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111000000000000000000000000";
constant A30_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000011111100001100001011101000010011";
constant A31_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000111101000000000001011001000001110";
constant A32_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000011111100001100001011101000010011";
constant A33_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111100010011100011001110110";
constant A34_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111111111010101110000110010000000000000";
constant A40_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
constant A41_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
constant A42_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
constant A43_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
constant A44_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000010110111011011101001100010010001";

constant B0_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000110000001111001001101101";
constant B1_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000000011010000101110111001101101100001";
constant B2_coef: signed(FHE_L-1 downto 0)  := "111111111111111111111111111111111111111111111111111111111111111111111111111111111111111110011111010110000000000000";
constant B3_coef: signed(FHE_L-1 downto 0)  := "111111111111111111111111111111111111111111111111111111111111111111111111111111111100101111101011000001100000000000";
constant B4_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000111000101100011001100011001110010110";

constant K0_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000010011010001111100001001101000111110";
constant K1_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000000011010000110100111110101000010010";
constant K2_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000010001101100011011011001011111001111";
constant K3_coef: signed(FHE_L-1 downto 0)  := "000000000000000000000000000000000000000000000000000000000000000000000000000000000001101110010001000110100010010001";
constant K4_coef: signed(FHE_L-1 downto 0)  := "111111111111111111111111111111111111111111111111111111111111111111111111111111111111111001000001111111000000000000";

constant L00_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111111110011000111111111010010000000000";
constant L01_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000010001101111101100001011000000";
constant L10_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111111101001010011100010110000000000000";
constant L11_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000001101110011101101111001111010111110";
constant L20_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000000000011001001101001110101001011111";
constant L21_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111110111011101011110110101000000000000";
constant L30_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000111100001111000101011111001010001010";
constant L31_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111100001101110011100101100101100000000000";
constant L40_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111101001111010000001000001000000000000";
constant L41_coef: signed(FHE_L-1 downto 0) := "000000000000000000000000000000000000000000000000000000000000000000000000000000011100001010111111010111001001100100";

constant neg_one_coef: signed(FHE_L-1 downto 0) := "111111111111111111111111111111111111111111111111111111111111111111111111111111111000000000000000000000000000000000";

end package;

--package body-------------------------------------------------------------------------------------------------------
package body types_package is


end package body;
```

B.8. key_generator.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;


--entity declaration-----------------------------------------------------------------
entity key_generator is
generic (LFSR_reg_size: natural);
port (
        clk             :   in std_logic;
        reset           :   in std_logic;
        gen             :   in std_logic;
        LFSR_reg        :   in std_logic_vector(LFSR_reg_size-1 downto 0);

        public_key      :   out matrix_array(0 to FHE_m-1, 0 to FHE_n);
        private_key     :   out vector_array(0 to FHE_n-1);
        gen_done        :   out std_logic
      );
end key_generator;

architecture Behavioral of key_generator is

--component declaration--------------------------------------------------------------
component debounce_comparator is
generic(init: std_logic := '0');
port(
        clk         :   in std_logic;
        db_in       :   in std_logic;
        pulse_out   :   out std_logic
    );
end component;

component error_vect_gen is
port(
        clk             :   in std_logic;
        input_vect      :   in vector_array_e(0 to FHE_m-1);
        calc            :   in std_logic;

        output_vect     :   out vector_array(0 to FHE_m-1);
        done_flag       :   out std_logic := '0'
    );
end component;

component MVa_core is
generic(row_count, col_count : natural);
    Port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_mat:      in matrix_array(0 to row_count-1, 0 to col_count-1);
            input_vect0:    in vector_array(0 to col_count-1);
            input_vect1:    in vector_array(0 to row_count-1);
```

```
            output_vect:    out vector_array(0 to row_count-1);
            calc_done:      out std_logic
          );
end component;

--signal declaration--------------------------------------------------------------------
constant cycle_count_e  : natural := integer(ceil(real(FHE_m)*real(FHE_L)/real(LFSR_reg_size))); --amount of cycles needed to fill an array for errorvector generation
constant final_cycle_e  : natural := FHE_m*FHE_L-(cycle_count_e-1)*LFSR_reg_size; --in case the target (FHE_m) is not evenly devisible in blocks of 256, a special subroutine handles the final allocat

constant cycle_count_t  : natural := integer(ceil(real(FHE_n)*real(FHE_L)/real(LFSR_reg_size))); --amount of cycles needed to fill an array for errorvector generation
constant final_cycle_t  : natural := FHE_n*FHE_L-(cycle_count_t-1)*LFSR_reg_size;

constant cycle_count_B  : natural := integer(ceil(real(FHE_m)*real(FHE_n)*real(FHE_L)/real(LFSR_reg_size))); --amount of cycles needed to fill an array for errorvector generation
constant final_cycle_B  : natural := FHE_m*FHE_n*FHE_L-(cycle_count_B-1)*LFSR_reg_size;

signal t_vect           : vector_array(0 to FHE_n-1);

signal b_vect           : vector_array(0 to FHE_m-1);
signal LFSR_vect_pipe   : vector_array_e(0 to FHE_m-1);

signal B_mat            : matrix_array(0 to FHE_m-1, 0 to FHE_n-1);

signal e_empt           : vector_array(0 to FHE_m-1) := (others => (others => '0'));
signal e_vect           : vector_array(0 to FHE_m-1);
signal e_i, new_e_i     : natural range 0 to cycle_count_e-1    := cycle_count_e-1;
signal t_i, new_t_i     : natural range 0 to cycle_count_t-1    := cycle_count_t-1;
signal B_i, new_B_i     : natural range 0 to cycle_count_B-1    := cycle_count_B-1;

signal e_in_done_db, t_done_db, B_done_db, error_gen_done, MMult_done: std_logic := '0';
signal e_in_done, t_done, B_done : std_logic := '1';
signal e_in_done_delay, t_done_delay, B_done_delay : std_logic := '1';

constant debug_const: natural := 0;

--function declaration-------------------------------------------------------------------
function reg2vect_e_in(input_reg: std_logic_vector(LFSR_reg_size-1 downto 0); input_vect: vector_array_e(0 to FHE_m-1); i: natural) return vector_array_e is
variable output_vect: vector_array_e(0 to FHE_m-1);
variable sel_i: natural range 0 to FHE_m-1;
variable sel_j: natural range 0 to FHE_L-1;

begin
    output_vect := input_vect;

    if i = cycle_count_e-1 then
        for j in 0 to final_cycle_e-1 loop
            sel_i := (i*LFSR_reg_size+j)/FHE_L;
            sel_j := (i*LFSR_reg_size+j)-FHE_L*sel_i;

            output_vect(sel_i)(sel_j) := input_reg(j);
        end loop;
    else
        for j in 0 to LFSR_reg_size-1 loop
            sel_i := (i*LFSR_reg_size+j)/FHE_L;
            sel_j := (i*LFSR_reg_size+j)-FHE_L*sel_i;

            output_vect(sel_i)(sel_j) := input_reg(j);
        end loop;
    end if;
return output_vect;
end function;

function reg2vect_t(input_reg: std_logic_vector(LFSR_reg_size-1 downto 0); input_vect: vector_array(0 to FHE_n-1); i: natural) return vector_array is
variable output_vect: vector_array(0 to FHE_n-1);
variable sel_i: natural range 0 to FHE_n-1;
variable sel_j: natural range 0 to FHE_L-1;

begin
    output_vect := input_vect;

    if i = cycle_count_t-1 then
        for j in 0 to final_cycle_t-1 loop
            sel_i := (i*LFSR_reg_size+j)/FHE_L;
            sel_j := (i*LFSR_reg_size+j)-FHE_L*sel_i;

            if sel_j < debug_const then
                output_vect(sel_i)(sel_j) := input_reg(j);
            else
                output_vect(sel_i)(sel_j) := '0';
            end if;
        end loop;
    else
        for j in 0 to LFSR_reg_size-1 loop
            sel_i := (i*LFSR_reg_size+j)/FHE_L;
            sel_j := (i*LFSR_reg_size+j)-FHE_L*sel_i;

            if sel_j < debug_const then
                output_vect(sel_i)(sel_j) := input_reg(j);
            else
                output_vect(sel_i)(sel_j) := '0';
            end if;
        end loop;
    end if;
return output_vect;
end function;

function reg2mat(input_reg: std_logic_vector(LFSR_reg_size-1 downto 0); input_mat: matrix_array(0 to FHE_m-1,0 to FHE_n-1); i: natural) return matrix_array is
variable output_mat: matrix_array(0 to FHE_m-1,0 to FHE_n-1);
variable sel_i: natural range 0 to FHE_m-1;
variable sel_j: natural range 0 to FHE_n-1;
variable sel_k: natural range 0 to FHE_L-1;

begin
    output_mat := input_mat;

    if i = cycle_count_B-1 then
        for j in 0 to final_cycle_B-1 loop
            sel_i := (i*LFSR_reg_size+j)/(FHE_L*FHE_n);
            sel_j := (i*LFSR_reg_size+j)/FHE_L - FHE_n*sel_i;
            sel_k := (sel_j*FHE_L+j+i*LFSR_reg_size)-(((sel_j*FHE_L+j+i*LFSR_reg_size)/FHE_L))*FHE_L;

            if sel_k < debug_const then
                output_mat(sel_i, sel_j)(sel_k) := input_reg(j);
```

```
            else
                output_mat(sel_i, sel_j)(sel_k) := '0';
            end if;
        end loop;
    else
        for j in 0 to LFSR_reg_size-1 loop
            sel_i := (i*LFSR_reg_size+j)/(FHE_L*FHE_n);
            sel_j := (i*LFSR_reg_size+j)/FHE_L - FHE_n*sel_i;
            sel_k := (sel_j*FHE_L+j+i*LFSR_reg_size)-(((sel_j*FHE_L+j+i*LFSR_reg_size)/FHE_L))*FHE_L;

            if sel_k < debug_const then
                output_mat(sel_i, sel_j)(sel_k) := input_reg(j);
            else
                output_mat(sel_i, sel_j)(sel_k) := '0';
            end if;
        end loop;
    end if;
return output_mat;
end function;

function pk_assembler(b_vect: vector_array(0 to FHE_m-1); B_mat: matrix_array(0 to FHE_m-1,0 to FHE_n-1)) return matrix_array is
variable pk: matrix_array(0 to FHE_m-1,0 to FHE_n);
variable tmp: signed(FHE_L-1 downto 0);

begin
    for i in 0 to FHE_m-1 loop
        pk(i,0) := b_vect(i);
    end loop;

    for i in 0 to FHE_m-1 loop
        for j in 1 to FHE_n loop
            pk(i,j) := B_mat(i,j-1);
        end loop;
    end loop;
return pk;
end function;

begin

debounce_comparator_unit_LFSR: debounce_comparator
generic map(init=>'1')
port map(
            clk=>clk,
            db_in=>e_in_done_delay,
            pulse_out=>e_in_done_db
        );

debounce_comparator_unit_t_vect: debounce_comparator
generic map(init=>'1')
port map(
            clk=>clk,
            db_in=>t_done,
            pulse_out=>t_done_db
        );

debounce_comparator_unit_B_mat: debounce_comparator
generic map(init=>'1')
port map(
            clk=>clk,
            db_in=>B_done_delay,
            pulse_out=>B_done_db
        );

error_vect_gen_unit: error_vect_gen
port map(
            clk=>clk,
            input_vect=>LFSR_vect_pipe,
            calc=>e_in_done_db,

            output_vect=>e_vect,
            done_flag=>error_gen_done
        );

MVa_core_unit: MVa_core
generic map(row_count=>FHE_m, col_count=>FHE_n)
port map(
            clk=>clk,
            calc=>B_done_db,

            input_mat=>B_mat,
            input_vect0=>t_vect,
            input_vect1=>e_empt, --e_empt for debug!!!!!! normally must be e_vect

            output_vect=>b_vect,
            calc_done=>MMult_done
        );

gen_done <= MMult_done;

sync_read: process(clk)
begin
if rising_edge(clk) then
    e_i <= new_e_i;
    t_i <= new_t_i;
    B_i <= new_B_i;
end if;
end process;

cycle_increment_logic_e: process(gen, e_i)
begin
if gen = '1' then
    new_e_i <= 0;
    e_in_done <= '0';
else
    if e_i = cycle_count_e-1 then
        new_e_i <= e_i;
        e_in_done <= '1';
    else
        new_e_i <= e_i + 1;
        e_in_done <= '0';
    end if;
end if;
```

```
end process;

cycle_increment_logic_t_mat: process(e_in_done_db, t_i)
begin
if e_in_done_db = '1' then
    new_t_i <= 0;
    t_done <= '0';
else
    if t_i = cycle_count_t-1 then
        new_t_i <= t_i;
        t_done <= '1';
    else
        new_t_i <= t_i + 1;
        t_done <= '0';
    end if;
end if;
end process;

cycle_increment_logic_B_mat: process(t_done_db, B_i)
begin
if t_done_db = '1' then
    new_B_i <= 0;
    B_done <= '0';
else
    if B_i = cycle_count_B-1 then
        new_B_i <= B_i;
        B_done <= '1';
    else
        new_B_i <= B_i + 1;
        B_done <= '0';
    end if;
end if;
end process;

LFSR_read: process(clk)
begin
if rising_edge(clk) then
    e_in_done_delay <= e_in_done;
    t_done_delay <= t_done;
    B_done_delay <= B_done;

    if e_in_done_delay = '0' then
        LFSR_vect_pipe <= reg2vect_e_in(LFSR_reg, LFSR_vect_pipe, e_i);
    end if;

    if t_done_delay = '0' then
        t_vect <= reg2vect_t(LFSR_reg, t_vect, t_i);
    end if;

    if B_done_delay = '0' then
        B_mat <= reg2mat(LFSR_reg, B_mat, B_i);
    end if;

    if MMult_done = '1' then
        public_key <= pk_assembler(b_vect, B_mat);
        private_key <= t_vect;
    end if;
end if;
end process;

end Behavioral;
```

B.9. LFSR_sub.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

--entity declaration--------------------------------------------------------------------
entity lfsr_sub is
generic(seed: std_logic_vector(63 downto 0));
port(
    clk             :   in std_logic;
    reset           :   in std_logic;
    enable          :   in std_logic;
    output_bit      :   out std_logic
    );
end lfsr_sub;

architecture Behavioral of lfsr_sub is

signal sequence_reg : std_logic_vector (63 downto 0) := seed;
signal feedback : std_logic;

begin
feedback <= not(sequence_reg(63) xor sequence_reg(62) xor sequence_reg(60) xor sequence_reg(59));

count_gen: process (reset, enable, clk)
begin
if rising_edge(clk) then
    if (reset = '1') then
        sequence_reg <= (others => '0');
    elsif enable = '1' then
        sequence_reg <= sequence_reg(62 downto 0) & feedback;
    end if;
end if;
end process;
output_bit <= sequence_reg(0);
```

```
end architecture Behavioral;
```

B.10. LFSR256.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity lfsr256 is
port(
        clk         :   in std_logic;
        enable      :   in std_logic;
        reset       :   in std_logic;
        output_reg  :   out std_logic_vector(255 downto 0)
    );
end lfsr256;

architecture Behavioral of lfsr256 is

component lfsr_sub is
generic(seed: std_logic_vector(63 downto 0));
port(
        clk             :   in std_logic;
        enable          :   in std_logic;
        reset           :   in std_logic;
        output_bit      :   out std_logic := '0'
    );
end component;

--signal declaration-------------------------------------------------------------------

signal sequence_reg : std_logic_vector(255 downto 0);

constant seed0:  std_logic_vector(63 downto 0):= "1100101011110101000100101101110100101101001111010101001010011101";
constant seed1:  std_logic_vector(63 downto 0):= "1110010011101111010110111110000110110000000000011101110110010101";

--The full list of seeds has been removed for brevity, the entire code is 20k lines-------

constant seed255:  std_logic_vector(63 downto 0):= "0100010100110110111001110111001001011101001000100000000011000001";

--componants wiring--------------------------------------------------------------------
begin

rand_gen0: lfsr_sub
generic map(seed => seed0)
port map(
            clk => clk,
            reset => reset,
            enable => enable,
            output_bit => sequence_reg(0)
        );

rand_gen1: lfsr_sub
generic map(seed => seed1)
port map(
            clk => clk,
            reset => reset,
            enable => enable,
            output_bit => sequence_reg(1)
        );

--The full list of seeds has been removed for brevity, the entire code is 20k lines-------

rand_gen255: lfsr_sub
generic map(seed => seed255)
port map(
            clk => clk,
            reset => reset,
            enable => enable,
            output_bit => sequence_reg(255)
        );

output_reg <= sequence_reg;

end architecture Behavioral;
```

B.11. main_controller.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity main_controller is
port(
        clk             :   in std_logic;
        setup           :   in std_logic;
        crtl            :   in std_logic;
```

```
        reset           :   in std_logic;

        public_key      :   in matrix_array(0 to FHE_m-1, 0 to FHE_n);
        theta_0_enc     :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        theta_1_enc     :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        upd_u           :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        upd_xh0         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        upd_xh1         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        upd_xh2         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        upd_xh3         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        upd_xh4         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        new_u           :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        new_xh0         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        new_xh1         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        new_xh2         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        new_xh3         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        new_xh4         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        crtl_done       :   out std_logic;
        setup_done      :   out std_logic
    );
end main_controller;

architecture Behavioral of main_controller is

component System_gen is
generic (LFSR_reg_size: natural);
port(
        clk             :   in std_logic;
        calc            :   in std_logic;

        LFSR_reg        :   in std_logic_vector(LFSR_reg_size-1 downto 0);
        public_key      :   in matrix_array(0 to FHE_m-1, 0 to FHE_n);

        A_00            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_10            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_20            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_30            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_40            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_01            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_11            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_21            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_31            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_41            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_02            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_12            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_22            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_32            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_42            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_03            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_13            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_23            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_33            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_43            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_04            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_14            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_24            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_34            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_44            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        B_0             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_1             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_2             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_3             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_4             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        K_0             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_1             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_2             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_3             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_4             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        L_00            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_10            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_20            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_30            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_40            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        L_01            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_11            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_21            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_31            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_41            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        neg_one         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        sys_done        :   out std_logic
    );
end component;

component Controller is
    port (
            clk             :   in std_logic;
            calc            :   in std_logic;
            reset           :   in std_logic;

            theta_0         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            theta_1         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            upd_u           :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh0         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh1         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh2         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh3         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            upd_xh4         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
```

```
            out_u           :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh0         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh1         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh2         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh3         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            out_xh4         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_00            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_10            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_20            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_30            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_40            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_01            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_11            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_21            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_31            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_41            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_02            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_12            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_22            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_32            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_42            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_03            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_13            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_23            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_33            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_43            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            A_04            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_14            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_24            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_34            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            A_44            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            B_0             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_1             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_2             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_3             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            B_4             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            K_0             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_1             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_2             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_3             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            K_4             :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            L_00            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_10            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_20            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_30            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_40            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            L_01            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_11            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_21            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_31            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
            L_41            :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            neg_one         :   in matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

            done_flag       :   out std_logic
          );
end component;

component lfsr256 is
port(
        clk         :   in std_logic;
        enable      :   in std_logic;
        reset       :   in std_logic;
        output_reg  :   out std_logic_vector(255 downto 0)
    );
end component;
-------------------------------------------------------------------------------------------------------------------------

signal A_00, A_10, A_20, A_30, A_40: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal A_01, A_11, A_21, A_31, A_41: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal A_02, A_12, A_22, A_32, A_42: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal A_03, A_13, A_23, A_33, A_43: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal A_04, A_14, A_24, A_34, A_44: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

signal B_0, B_1, B_2, B_3, B_4: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal K_0, K_1, K_2, K_3, K_4: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

signal L_00, L_10, L_20, L_30, L_40: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal L_01, L_11, L_21, L_31, L_41: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

signal neg_one: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

constant LFSR_reg_size: natural := 256;
signal LFSR_reg: std_logic_vector(LFSR_reg_size-1 downto 0);
signal LFSR_enable : std_logic := '0';
signal setup_pipe  : std_logic;

-------------------------------------------------------------------------------------------------------------------------
--constant trunc_amount : natural := 30;
-------------------------------------------------------------------------------------------------------------------------

--function circshift(input_mat: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1)) return matrix_bit_array is
--variable output_mat: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
--begin
--for i in trunc_amount to FHE_CN-1 loop
--    for j in 0 to FHE_CN-1 loop
--        output_mat(i-trunc_amount,j) := input_mat(i,j);
--    end loop;
--end loop;

--for i in 0 to trunc_amount-1 loop
--    for j in 0 to FHE_CN-1 loop
--        output_mat(i+(FHE_CN-trunc_amount)-1,j) := input_mat(i,j);
```

```
--    end loop;
--end loop;
--return output_mat;
--end function;

-------------------------------------------------------------------------------------------------------------------
begin

lfsr256_unit: lfsr256
port map(
        clk => clk,
        enable => LFSR_enable,
        reset => reset,
        output_reg => LFSR_reg
    );

System_gen_unit: System_gen
    generic map (LFSR_reg_size => LFSR_reg_size)
    port map(
                clk => clk,
                calc => setup,

                LFSR_reg => LFSR_reg,
                public_key => public_key,

                A_00 => A_00,
                A_10 => A_10,
                A_20 => A_20,
                A_30 => A_30,
                A_40 => A_40,

                A_01 => A_01,
                A_11 => A_11,
                A_21 => A_21,
                A_31 => A_31,
                A_41 => A_41,

                A_02 => A_02,
                A_12 => A_12,
                A_22 => A_22,
                A_32 => A_32,
                A_42 => A_42,

                A_03 => A_03,
                A_13 => A_13,
                A_23 => A_23,
                A_33 => A_33,
                A_43 => A_43,

                A_04 => A_04,
                A_14 => A_14,
                A_24 => A_24,
                A_34 => A_34,
                A_44 => A_44,

                B_0  => B_0 ,
                B_1  => B_1 ,
                B_2  => B_2 ,
                B_3  => B_3 ,
                B_4  => B_4 ,

                K_0  => K_0 ,
                K_1  => K_1 ,
                K_2  => K_2 ,
                K_3  => K_3 ,
                K_4  => K_4 ,

                L_00 => L_00,
                L_10 => L_10,
                L_20 => L_20,
                L_30 => L_30,
                L_40 => L_40,

                L_01 => L_01,
                L_11 => L_11,
                L_21 => L_21,
                L_31 => L_31,
                L_41 => L_41,

                neg_one => neg_one,

                sys_done => setup_pipe
        );

setup_done <= setup_pipe;

Controller_unit: Controller
port map(
            clk => clk,
            calc => crtl,
            reset => reset,

            theta_0 => theta_0_enc,
            theta_1 => theta_1_enc,

            upd_u  =>upd_u,
            upd_xh0=>upd_xh0,
            upd_xh1=>upd_xh1,
            upd_xh2=>upd_xh2,
            upd_xh3=>upd_xh3,
            upd_xh4=>upd_xh4,

            out_u => new_u,
            out_xh0=>new_xh0,
            out_xh1=>new_xh1,
            out_xh2=>new_xh2,
            out_xh3=>new_xh3,
            out_xh4=>new_xh4,


            A_00 => A_00,
            A_10 => A_10,
```

```
            A_20 => A_20,
            A_30 => A_30,
            A_40 => A_40,

            A_01 => A_01,
            A_11 => A_11,
            A_21 => A_21,
            A_31 => A_31,
            A_41 => A_41,

            A_02 => A_02,
            A_12 => A_12,
            A_22 => A_22,
            A_32 => A_32,
            A_42 => A_42,

            A_03 => A_03,
            A_13 => A_13,
            A_23 => A_23,
            A_33 => A_33,
            A_43 => A_43,

            A_04 => A_04,
            A_14 => A_14,
            A_24 => A_24,
            A_34 => A_34,
            A_44 => A_44,

            B_0  => B_0 ,
            B_1  => B_1 ,
            B_2  => B_2 ,
            B_3  => B_3 ,
            B_4  => B_4 ,

            K_0  => K_0 ,
            K_1  => K_1 ,
            K_2  => K_2 ,
            K_3  => K_3 ,
            K_4  => K_4 ,

            L_00 => L_00,
            L_10 => L_10,
            L_20 => L_20,
            L_30 => L_30,
            L_40 => L_40,

            L_01 => L_01,
            L_11 => L_11,
            L_21 => L_21,
            L_31 => L_31,
            L_41 => L_41,

            neg_one => neg_one,

            done_flag => crtl_done
    );

LFSR_enable_logic: process(clk)
begin
    if rising_edge(clk) then
        if setup = '1' then
            LFSR_enable <= '1';
        end if;
        if setup_pipe = '1' then
            LFSR_enable <= '0';
        end if;
    end if;
end process;

end architecture Behavioral;
```

B.12. Setup_Encoder.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity Setup_Encoder is
generic (LFSR_reg_size: natural);
port(
        clk             :   in std_logic;
        calc            :   in std_logic;

        LFSR_reg        :   in std_logic_vector(LFSR_reg_size-1 downto 0);
        mu              :   in signed(FHE_L-1 downto 0);
        public_key      :   in matrix_array(0 to FHE_m-1, 0 to FHE_n);

        output_cipher   :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        enc_done_flag   :   out std_logic := '0'
    );
end Setup_Encoder ;

architecture Behavioral of Setup_Encoder is

component debounce_comparator is
generic(init: std_logic := '0');
```

```
port(
        clk:        in std_logic;
        db_in:      in std_logic;
        pulse_out:  out std_logic
    );
end component debounce_comparator;

component BitMatMul_core is
generic(row_countA, col_countA, row_countB, col_countB : natural);
    Port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_matA:     in matrix_bit_array(0 to row_countA-1, 0 to col_countA-1);
            input_matB:     in matrix_array(0 to row_countB-1, 0 to col_countB-1);

            output_mat:     out matrix_array(0 to row_countA-1, 0 to col_countB-1);
            calc_done:      out std_logic
          );
end component;

-------------------------------------------------------------------------------------------------------------------------------------------------------
signal R_mat                    :   matrix_bit_array(0 to FHE_CN-1, 0 to FHE_m-1);
signal RA_mat                   :   matrix_array(0 to FHE_CN-1, 0 to FHE_n);
signal RA_mat_pmu               :   matrix_array(0 to FHE_CN-1, 0 to FHE_n);
signal done_flag, done_delay    :   std_logic := '0';

constant cycle_count    : natural := integer(ceil(real(FHE_CN)*real(FHE_m)/real(LFSR_reg_size))); --amount of cycles needed to fill an array for errorvector generation
constant final_cycle    : natural := FHE_CN*FHE_m-(cycle_count-1)*LFSR_reg_size; --in case the target (FHE_m) is not evenly devisible in blocks of 256, a special subroutine handles the final allocati

signal R_fill_done      :   std_logic := '1';
signal R_fill_done_db   :   std_logic := '0';
signal i, new_i: natural range 0 to cycle_count-1 := cycle_count-1;

signal output_cipher_pipe : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1) := (others => (others => '0'));
-------------------------------------------------------------------------------------------------------------------------------------------------------

function reg2mat_R(input_reg: std_logic_vector(LFSR_reg_size-1 downto 0); input_mat: matrix_bit_array(0 to FHE_CN-1,0 to FHE_m-1); i: natural) return matrix_bit_array is
variable output_mat: matrix_bit_array(0 to FHE_CN-1,0 to FHE_m-1);
variable sel_i: natural range 0 to FHE_CN-1;
variable sel_j: natural range 0 to FHE_m-1;

begin
    output_mat := input_mat;

    if i = cycle_count-1 then
        for j in 0 to final_cycle-1 loop
            sel_i := (i*LFSR_reg_size+j)/FHE_m;
            sel_j := (i*LFSR_reg_size+j) - FHE_m*sel_i;

            output_mat(sel_i, sel_j) := input_reg(j);
        end loop;
    else
        for j in 0 to LFSR_reg_size-1 loop
            sel_i := (i*LFSR_reg_size+j)/FHE_m;
            sel_j := (i*LFSR_reg_size+j) - FHE_m*sel_i;

            output_mat(sel_i, sel_j) := input_reg(j);
        end loop;
    end if;
return output_mat;
end function;

--The following functions generate the cipher. To reduce the amount of calculations required, the generation is heavily altered
--i.e. replaced with equivalent but more efficient operations. See documentation on the alterations.
function Imu_add(mu: signed(FHE_L-1 downto 0); RA_mat: matrix_array(0 to FHE_CN-1, 0 to FHE_n)) return matrix_array is
variable RA_plus_Imu: matrix_array(0 to FHE_CN-1, 0 to FHE_n);
variable mu_shifted: signed(FHE_L-1 downto 0);
variable k: natural range 0 to FHE_L-1;
variable container: natural range 0 to FHE_n+1;

begin
    RA_plus_Imu := RA_mat;
    container := 0;
    k := 0;
    for i in 0 to FHE_CN-1 loop
        mu_shifted := shift_left(mu,k);
        RA_plus_Imu(i,container) := RA_mat(i,container) + mu_shifted;
        if (k = FHE_L-1) then
            container := container + 1;
            k := 0;
        else
            k := k + 1;
        end if;
    end loop;
return RA_plus_Imu;
end Imu_add;

--The Flattening operation is integrated in the next two functions. Binary representation is used, hence BitDecomp and inverse-BitDecomp
--have to be interpreted and integrated
function cipher_gen(RA_plus_Imu: matrix_array(0 to FHE_CN-1, 0 to FHE_n)) return matrix_bit_array is
variable cipher : matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

begin
    for i in 0 to FHE_CN-1 loop
            for j in 0 to FHE_n loop
                for k in 0 to FHE_L-1 loop
                    cipher(i,k+j*FHE_L) := RA_plus_Imu(i,j)(k);
                end loop;
            end loop;
        end loop;
return cipher;
end function;


--signal dfun_out: integer;
--function dfun(R_mat: matrix_bit_array(0 to FHE_CN-1,0 to FHE_m-1)) return integer is
--variable vect: vector_bit_array(0 to FHE_m-1);
--variable val: natural;

--begin
--for i in 0 to FHE_CN-1 loop
```

```
--    vect := (others => '0');
--    for j in 0 to FHE_m-1 loop
--        vect(j) := R_mat(i,j);
--    end loop;
--    val := to_integer(unsigned(vect));
--    report natural'image(val);
--end loop;
--return 0;
--end function;

-------------------------------------------------------------------------------------------------------------------------------------------------------
begin

debounce_comparator_unit: debounce_comparator
generic map(init=>'1')
port map(
            clk=>clk,
            db_in=>R_fill_done,
            pulse_out=>R_fill_done_db
        );

BitMatMul_core_unit: BitMatMul_core
generic map(row_countA=>FHE_CN, col_countA=>FHE_m, row_countB=>FHE_m, col_countB=>FHE_n+1)
port map(
        clk=>clk,
        calc=>R_fill_done_db,

        input_matA=>R_mat,
        input_matB=>public_key,

        output_mat=>RA_mat,
        calc_done=>done_flag
        );

cycle_increment_logic: process(calc, i)
begin
if calc = '1' then
    new_i <= 0;
    R_fill_done <= '0';
else
    if i = cycle_count-1 then
        new_i <= i;
        R_fill_done <= '1';
    else
        new_i <= i + 1;
        R_fill_done <= '0';
    end if;
end if;
end process;

mu_capture: process(clk)
begin
if rising_edge(clk) then
i <= new_i;
done_delay <= done_flag;
enc_done_flag <= done_delay;

    if R_fill_done = '0' then
        R_mat <= reg2mat_R(LFSR_reg, R_mat, i);
    end if;

    if done_flag = '1' then
        RA_mat_pmu <= Imu_add(mu, RA_mat);
    end if;

    if done_delay = '1' then
        output_cipher_pipe <= cipher_gen(RA_mat_pmu);
    end if;
end if;
end process;

output_cipher <= output_cipher_pipe;

end architecture Behavioral;
```

B.13. System_gen.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.MATH_REAL.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity System_gen-----------------------------------------------------------------------
entity System_gen is
generic (LFSR_reg_size: natural);
port(
        clk             :   in std_logic;
        calc            :   in std_logic;

        LFSR_reg        :   in std_logic_vector(LFSR_reg_size-1 downto 0);
        public_key      :   in matrix_array(0 to FHE_m-1, 0 to FHE_n);

        A_00            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_10            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_20            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_30            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_40            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
```

```
        A_01            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_11            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_21            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_31            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_41            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_02            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_12            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_22            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_32            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_42            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_03            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_13            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_23            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_33            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_43            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        A_04            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_14            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_24            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_34            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        A_44            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        B_0             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_1             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_2             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_3             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        B_4             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        K_0             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_1             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_2             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_3             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        K_4             :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        L_00            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_10            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_20            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_30            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_40            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        L_01            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_11            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_21            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_31            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        L_41            :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        neg_one         :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);

        sys_done        :   out std_logic
    );
end System_gen;

architecture Behavioral of System_gen is

component debounce_pulse is
generic(init: std_logic := '0');
port(
        clk:        in std_logic;
        db_in:      in std_logic;
        pulse_out:  out std_logic
    );
end component debounce_pulse;

component lfsr256 is
port(
        clk         :   in std_logic;
        enable      :   in std_logic;
        reset       :   in std_logic;
        output_reg  :   out std_logic_vector(255 downto 0)
    );
end component;

component Setup_Encoder is
generic (LFSR_reg_size: natural);
port(
        clk             :   in std_logic;
        calc            :   in std_logic;

        LFSR_reg        :   in std_logic_vector(LFSR_reg_size-1 downto 0);
        mu              :   in signed(FHE_L-1 downto 0);
        public_key      :   in matrix_array(0 to FHE_m-1, 0 to FHE_n);

        output_cipher   :   out matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
        enc_done_flag   :   out std_logic := '0'
    );
end component;


signal coef: signed(FHE_L-1 downto 0);
-------------------------------------------------------------------------------------------------------------------------------------------------------

constant A_dim: natural := 5; --square
constant operation_cnt: natural := A_dim*A_dim + 4*A_dim + 1;

signal i, new_i: natural range 0 to operation_cnt-1;
signal done: std_logic := '1';
--signal done_db: std_logic := '0';

signal calc_cipher: std_logic := '0';

constant LFSR_size: natural := 256;

signal cur_cipher: matrix_bit_array(0 to FHE_CN-1, 0 to FHE_CN-1);
signal enc_done_flag: std_logic;

-------------------------------------------------------------------------------------------------------------------------------------------------------
begin

--debounce_pulse_unit: debounce_pulse
--generic map( init=>'1' )
```

```
--port map(
--            clk=>clk,
--            db_in=>done,
--            pulse_out=>done_db
--        );

Setup_Encoder_unit: Setup_Encoder
generic map(LFSR_reg_size=>LFSR_reg_size)
port map(
            clk=>clk,
            calc=>calc_cipher,

            LFSR_reg=>LFSR_reg,
            mu=>coef,
            public_key=>public_key,

            output_cipher=>cur_cipher,
            enc_done_flag=>enc_done_flag
        );

sync: process(clk)
begin
if rising_edge(clk) then
    if done = '1' then
        sys_done <= enc_done_flag;
    else
        sys_done <= '0';
    end if;

    if (enc_done_flag = '1' and done = '0') or calc = '1' then
        i <= new_i;
    end if;
end if;
end process;

cycle_increment_logic: process(calc, enc_done_flag, i)
begin
if calc = '1' then
    new_i <= 0;
    done <= '0';

    calc_cipher <= '1';
else
    if i = operation_cnt-1 then
        new_i <= i;
        done <= '1';

        calc_cipher <= '0';
    else
        new_i <= i + 1;
        done <= '0';

        calc_cipher <= enc_done_flag;
    end if;
end if;
end process;

input_piping: process(clk)
begin
if i = 0 then
    coef <= A00_coef;
end if;
if i = 1 then
    coef <= A10_coef;
end if;
if i = 2 then
    coef <= A20_coef;
end if;
if i = 3 then
    coef <= A30_coef;
end if;
if i = 4 then
    coef <= A40_coef;
end if;

if i = 5 then
    coef <= A01_coef;
end if;
if i = 6 then
    coef <= A11_coef;
end if;
if i = 7 then
    coef <= A21_coef;
end if;
if i = 8 then
    coef <= A31_coef;
end if;
if i = 9 then
    coef <= A41_coef;
end if;

if i = 10 then
    coef <= A02_coef;
end if;
if i = 11 then
    coef <= A12_coef;
end if;
if i = 12 then
    coef <= A22_coef;
end if;
if i = 13 then
    coef <= A32_coef;
end if;
if i = 14 then
    coef <= A42_coef;
end if;

if i = 15 then
    coef <= A03_coef;
end if;
if i = 16 then
    coef <= A13_coef;
```

```
end if;
if i = 17 then
    coef <= A23_coef;
end if;
if i = 18 then
    coef <= A33_coef;
end if;
if i = 19 then
    coef <= A43_coef;
end if;

if i = 20 then
    coef <= A04_coef;
end if;
if i = 21 then
    coef <= A14_coef;
end if;
if i = 22 then
    coef <= A24_coef;
end if;
if i = 23 then
    coef <= A34_coef;
end if;
if i = 24 then
    coef <= A44_coef;
end if;

if i = 25 then
    coef <= B0_coef;
end if;
if i = 26 then
    coef <= B1_coef;
end if;
if i = 27 then
    coef <= B2_coef;
end if;
if i = 28 then
    coef <= B3_coef;
end if;
if i = 29 then
    coef <= B4_coef;
end if;

if i = 30 then
    coef <= K0_coef;
end if;
if i = 31 then
    coef <= K1_coef;
end if;
if i = 32 then
    coef <= K2_coef;
end if;
if i = 33 then
    coef <= K3_coef;
end if;
if i = 34 then
    coef <= K4_coef;
end if;

if i = 35 then
    coef <= L00_coef;
end if;
if i = 36 then
    coef <= L10_coef;
end if;
if i = 37 then
    coef <= L20_coef;
end if;
if i = 38 then
    coef <= L30_coef;
end if;
if i = 39 then
    coef <= L40_coef;
end if;

if i = 40 then
    coef <= L01_coef;
end if;
if i = 41 then
    coef <= L11_coef;
end if;
if i = 42 then
    coef <= L21_coef;
end if;
if i = 43 then
    coef <= L31_coef;
end if;
if i = 44 then
    coef <= L41_coef;
end if;


if i = 45 then
    coef <= neg_one_coef;
end if;

end process;

output_piping: process(clk)
begin
if rising_edge(clk) then
    if enc_done_flag = '1' then
        if i = 0 then
            A_00 <= cur_cipher;
        end if;
        if i = 1 then
            A_10 <= cur_cipher;
        end if;
        if i = 2 then
            A_20 <= cur_cipher;
        end if;
        if i = 3 then
```

```
            A_30 <= cur_cipher;
        end if;
        if i = 4 then
            A_40 <= cur_cipher;
        end if;

        if i = 5 then
            A_01 <= cur_cipher;
        end if;
        if i = 6 then
            A_11 <= cur_cipher;
        end if;
        if i = 7 then
            A_21 <= cur_cipher;
        end if;
        if i = 8 then
            A_31 <= cur_cipher;
        end if;
        if i = 9 then
            A_41 <= cur_cipher;
        end if;

        if i = 10 then
            A_02 <= cur_cipher;
        end if;
        if i = 11 then
            A_12 <= cur_cipher;
        end if;
        if i = 12 then
            A_22 <= cur_cipher;
        end if;
        if i = 13 then
            A_32 <= cur_cipher;
        end if;
        if i = 14 then
            A_42 <= cur_cipher;
        end if;

        if i = 15 then
            A_03 <= cur_cipher;
        end if;
        if i = 16 then
            A_13 <= cur_cipher;
        end if;
        if i = 17 then
            A_23 <= cur_cipher;
        end if;
        if i = 18 then
            A_33 <= cur_cipher;
        end if;
        if i = 19 then
            A_43 <= cur_cipher;
        end if;

        if i = 20 then
            A_04 <= cur_cipher;
        end if;
        if i = 21 then
            A_14 <= cur_cipher;
        end if;
        if i = 22 then
            A_24 <= cur_cipher;
        end if;
        if i = 23 then
            A_34 <= cur_cipher;
        end if;
        if i = 24 then
            A_44 <= cur_cipher;
        end if;

        if i = 25 then
            B_0 <= cur_cipher;
        end if;
        if i = 26 then
            B_1 <= cur_cipher;
        end if;
        if i = 27 then
            B_2 <= cur_cipher;
        end if;
        if i = 28 then
            B_3 <= cur_cipher;
        end if;
        if i = 29 then
            B_4 <= cur_cipher;
        end if;

        if i = 30 then
            K_0 <= cur_cipher;
        end if;
        if i = 31 then
            K_1 <= cur_cipher;
        end if;
        if i = 32 then
            K_2 <= cur_cipher;
        end if;
        if i = 33 then
            K_3 <= cur_cipher;
        end if;
        if i = 34 then
            K_4 <= cur_cipher;
        end if;

        if i = 35 then
            L_00 <= cur_cipher;
        end if;
        if i = 36 then
            L_10 <= cur_cipher;
        end if;
        if i = 37 then
            L_20 <= cur_cipher;
        end if;
        if i = 38 then
```

```
            L_30 <= cur_cipher;
        end if;
        if i = 39 then
            L_40 <= cur_cipher;
        end if;

        if i = 40 then
            L_01 <= cur_cipher;
        end if;
        if i = 41 then
            L_11 <= cur_cipher;
        end if;
        if i = 42 then
            L_21 <= cur_cipher;
        end if;
        if i = 43 then
            L_31 <= cur_cipher;
        end if;
        if i = 44 then
            L_41 <= cur_cipher;
        end if;

        if i = 45 then
            neg_one <= cur_cipher;
        end if;
    end if;
end if;
end process;

end architecture Behavioral;
```

C

Demo code

The FPGA can be interfaced with utilsing the matlab code below. The files "nexys4_demo_interface.m" and "Gentry_HME_MM.m" are followed by the vhdl code that is used in the demo. To prevent repetition, the modules that have already been presented in appendix B have been omitted.

C.1. nexys4_demo_interface.vhd

```
close;
clear;
clc;

%% prep
%-------------------------
transmit_queue = [];
for i = 1:4:4*64
    transmit_queue = [transmit_queue, uint64(i)+uint64(i+1)*uint64(2^16)+uint64(i+2)*uint64(2^(2*16))+uint64(i+3)*uint64(2^(48))];
%     transmit_queue = [transmit_queue, uint64(i+3)+uint64(i+2)*uint64(2^16)+uint64(i+1)*uint64(2^(2*16))+uint64(i)*uint64(2^(48))];
end
transmit_queue = [transmit_queue, transmit_queue];
%-------------------------

% row_1 = uint64(1)+uint64(2)*uint64(2^16)+uint64(3)*uint64(2^(2*16))+uint64(4)*uint64(2^(48));
% fact = 1;
% row_1 = uint64(fact*1)+uint64(fact*1)*uint64(2^16)+uint64(fact*1)*uint64(2^(2*16))+uint64(fact*1)*uint64(2^(48));

% transmit_queue = [row_1, zeros(1,63), row_1, zeros(1,63)];
% transmit_queue = repmat(row_1,1,2*64);

% transmit_queue = repmat(intmax('uint64'),1,2*64);
% transmit_queue = uint64(randi(intmax('uint32'),1,2*64)+randi(intmax('uint32'),1,2*64));

%%
% transmit_queue = [];
% for i = 0:63
%     transmit_queue = [transmit_queue, uint64(2^i)];
% end
% transmit_queue = uint64([zeros(1,16),ones(1,16),zeros(1,32)]);

%%
val1 = uint64(7);
% val1t = val1;
val1t = val1 + val1*uint64(2^48);

val2 = uint64(1);
% val2t = val2 + val2*uint64(2^48);
val2t = val2;

transmit_queue(1:64) = val1t;
transmit_queue(65:end) = val2t;

%%
FHE_n = 3;
FHE_L = 16;
FHE_CN = (FHE_n+1)*FHE_L;

% C1 = reshape(uint64(randi(intmax('uint8'),1,(FHE_n+1)*FHE_CN)),FHE_CN,FHE_n+1);
% C2 = reshape(uint64(randi(intmax('uint8'),1,(FHE_n+1)*FHE_CN)),FHE_CN,FHE_n+1);

% cnt = 1;
% for i = 1:FHE_CN
%     C1(i,:) = uint64(cnt:(cnt+FHE_n));
%     cnt = cnt + FHE_n+1;
% end
% C2 = C1;

C1 = repmat(uint64([val1,0,0,val1]),FHE_CN,1);
C2 = repmat(uint64([val2,0,0,0]),FHE_CN,1);

% C1 = repmat(uint64([val1,0,0,0]),FHE_CN,1);
% C2 = repmat(uint64([val2,0,0,val2]),FHE_CN,1);

C_out = Gentry_HME_MM(C1,C2,FHE_n,FHE_L);
```

```
C_out(1,:)

%% contact
device = serialport("COM3",1E6);
% write(device, "flp", 'char')
% write(device, "mode", 'char')

%% send
write(device, "clc", 'char')

for i = 1:length(transmit_queue)
    write(device, transmit_queue(i),  'uint64')
end

write(device, "calc", 'char')
data = uint64(read(device,4,'uint64'));

%%
ciph_out = zeros(1,4*length(data));
for i = 1:4:4*length(data)
    tmp = de2bi(data(floor(i/4)+1),64);
    ciph_out(i:i+3) = [bi2de(tmp(1:16)),bi2de(tmp(17:32)),bi2de(tmp(33:48)),bi2de(tmp(49:end))];
end

ciph_out_old = ciph_out;
ciph_out = zeros(length(ciph_out_old)/4,4);
for i = 1:length(ciph_out_old)
    ciph_out(floor((i-1)/4)+1,mod(i-1,4)+1) = ciph_out_old(i);
end

ciph_out
```

C.2. Gentry_HME_MM.vhd

```
function C_out = Gentry_HME_MM(C1,C2,FHE_n,FHE_L)
    if size(C1) ~= size(C1)
        error('C1 and C2 must be the same size')
    end
    FHE_CN = (FHE_n+1)*FHE_L;

    for col = 1:(FHE_n+1)
        for i = 1:FHE_CN
            row_cnt = 1;
            carry = 0;
            for j = 1:(FHE_n+1)
                cur_el = C1(i,j);
                for k = 1:FHE_L
                    sel = uint64(2^(k-1));
                    pre_sel = uint64(2^k);

                    step1 = idivide(cur_el,pre_sel);
                    step2 = cur_el - step1*pre_sel;
                    bit = idivide(step2,sel);

                    carry = carry + bit*C2(row_cnt,col);

                    row_cnt = row_cnt+1;
                end
            end
            C_out(i,col) = carry;
        end
    end
end
```

C.3. APU.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural and structural design of a simple APU
--This APU is hard wired for performing a cipher multiplication and addition

--The matrix and vector data are loaded continuously from the input register
--every time the circuit receives a read-ready signal (UART receiver has
--received 8 bytes of information) in the order: matrix in row major order,
--and then the vector. To ensure correct loading of the values, sending
--a "clc" command to the UART port is recommended as this resets the loading
--process.
--To perform the calculation, send a "calc" command to the UART port

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
    use IEEE.MATH_REAL.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity APU is
port(
    clk         :   in std_logic;
    reset       :   in std_logic;
    input_reg   :   in std_logic_vector(UART_msg_len-1 downto 0);
    read_ready  :   in std_logic;
    calc        :   in std_logic;

    result_mat  :   out matrix_array(0 to FHE_CN-1, 0 to FHE_n);
    TXD_start   :   out std_logic
    );
```

```
end APU;

--------------------------------------------------------------------------------
architecture Behavioral of APU is

--component declaration------------------------------------------------------------
component CipherAdd_core is
    port (
            clk:            in std_logic;
            calc:           in std_logic;

            input_matA:     in  matrix_array(0 to FHE_CN-1, 0 to FHE_n);
            input_matB:     in  matrix_array(0 to FHE_CN-1, 0 to FHE_n);

            output_mat:     out matrix_array(0 to FHE_CN-1, 0 to FHE_n);
            calc_done:      out std_logic
         );
end component CipherAdd_core;

--signal declaration----------------------------------------------------------------

constant cycle_count  : natural := integer(ceil(real(FHE_CN)*real(FHE_CN)/real(UART_msg_len))); --amount of cycles needed to fill an array for errorvector generation
constant final_cycle  : natural := FHE_CN*FHE_CN-(cycle_count-1)*UART_msg_len;

constant mat_count: natural := 2;

signal input_matA, input_matB, output_mat: matrix_array(0 to FHE_CN-1, 0 to FHE_n) := (others => (others => (others => '0')));

signal i, new_i: natural range 0 to cycle_count-1;
signal mat_cntr, new_mat_cntr: natural range 0 to mat_count-1;
signal calc_done, switch_input: std_logic := '0';
signal load, new_load: std_logic := '1';

signal output_pipe: std_logic_vector(L-1 downto 0);

--function declaration--------------------------------------------------------------
function reg2mat(input_reg: std_logic_vector(UART_msg_len-1 downto 0); input_mat: matrix_array(0 to FHE_CN-1,0 to FHE_n); i: natural) return matrix_array is
variable output_mat: matrix_array(0 to FHE_CN-1,0 to FHE_n);
variable sel_i: natural range 0 to FHE_L-1;
variable sel_j: natural range 0 to FHE_n+1;
variable sel_k: natural range 0 to FHE_L-1;
variable loop_cnt: natural range 0 to UART_msg_len;

begin
    output_mat := input_mat;

    if i = cycle_count-1 then
        loop_cnt := final_cycle;
    else
        loop_cnt := UART_msg_len;
    end if;

    for j in 0 to loop_cnt-1 loop
        sel_i := (i*UART_msg_len+j)/FHE_CN;
        sel_j := (i*UART_msg_len+j)/FHE_L - (FHE_n+1)*sel_i;
        sel_k := (i*UART_msg_len+j) - ((i*UART_msg_len+j)/FHE_L)*FHE_L;--(sel_j*FHE_L+j+i*UART_msg_len)-(((sel_j*FHE_L+j+i*UART_msg_len)/FHE_L))*FHE_L;

        output_mat(sel_i, sel_j)(sel_k) := input_reg(j);
    end loop;

return output_mat;
end function;

--component wiring------------------------------------------------------------------
begin

CipherAdd_core_unit :   CipherAdd_core
port map(
            clk => clk,
            calc => calc,

            input_matA => input_matA,
            input_matB => input_matB,

            output_mat => output_mat,
            calc_done => calc_done
        );

--State changes are registered on the clock tick
sync: process(clk)
begin
    if rising_edge(clk) then
        load <= new_load;
        i <= new_i;
        mat_cntr <= new_mat_cntr;
    end if;
end process;

--Register loading into buffer
register_read: process(i, read_ready, reset, input_reg, load, output_pipe, mat_cntr)
begin
if rising_edge(read_ready) then
    if load = '1' then
        if mat_cntr = 0 then
            input_matA <= reg2mat(input_reg, input_matA, i);
        else
            input_matB <= reg2mat(input_reg, input_matB, i);
        end if;
    end if;
end if;

if reset = '1' then
    new_i <= 0;
    new_mat_cntr <= 0;
    new_load <= '1';
else
    if read_ready = '1' then
        if i = cycle_count-1 then
            if mat_cntr = mat_count-1 then
                new_i <= i;
                new_mat_cntr <= mat_cntr;
                new_load <= '0';
```

```
            else
                new_i <= 0;
                new_mat_cntr <= mat_cntr + 1;
                new_load <= '1';
            end if;
        else
            new_i <= i + 1;
            new_mat_cntr <= mat_cntr;
            new_load <= '1';
        end if;
    else
        new_i <= i;
        new_mat_cntr <= mat_cntr;
        new_load <= load;
    end if;
end if;
end process;

--Matrix vector multiplication
calculation: process(clk, calc_done)
begin
    if rising_edge(clk) then
        result_mat <= output_mat;
    end if;
end process;

--Sending out signal when result is ready
TXD_start <= calc_done;

end architecture Behavioral;
```

C.4. counter.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural design of a counter
--Every clock tick the counter counts, but only when the enable signal is high

--When the enable signal is low, the count is retained.
--When the reset signal is high the count is reset

--Both enable and reset signals are acted upon synchronously

--When the counter reaches the maximum counter value, the output signal is set to high.
--Only when the reset signal is sent will the count be reset and the output signal set to low.
--If the user wishes for a clock that resets itself, see cyc_counter.
--Linking the output signal to the reset of this unit will likely cause unexpected behaviour
--and hence not advised (use cyc_counter).

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--entity declaration--------------------------------------------------------------------
entity counter is
generic(system_speed, max_count : natural);
port(
    clk         : in    std_logic;
    enable      : in    std_logic;
    reset       : in    std_logic;
    output      : out   std_logic
    );
end counter;

------------------------------------------------------------------------------------------
architecture Behavioral of counter is

--signal declaration--------------------------------------------------------------------
type state_type is (s0, counting, finish);
signal state, new_state : state_type;
signal count, new_count : natural range 0 to max_count:= 0;

--State changes are registered on the clock tick
begin
sync  :  process (clk)
    begin
    if (rising_edge(clk)) then
        state <= new_state;
        count <= new_count;
    end if;
end process;

--State change logic
logic : process(clk, state, reset, count)
    begin
    case state is
        when s0=>
        if (reset = '1') then
            new_state <= s0;
        else
            new_state <= counting;
        end if;

        when counting =>
        if (reset = '1') then
            new_state <= s0;
        elsif (count > max_count-2) and (reset = '0') then
            new_state <= finish;
        else
            new_state <= counting;
        end if;

        when finish =>
        if reset = '1' then
```

```
            new_state <= s0;
        else
            new_state <= finish;
        end if;

        when others =>
            new_state <= finish;
    end case;
end process;

--Behaviour attributed to each state
operation : process(state, count, enable)
    begin
    case state is
        when s0 =>
        new_count <= 0;
        output <= '0';

        when counting =>
        if enable = '1' then
            new_count <= count + 1;
        else
            new_count <= count;
        end if;
        output <= '0';

        when finish =>
        new_count <= 0;
        output <= '1';
    end case;
    end process;
end Behavioral;
```

C.5. cyc_counter.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural design of a cyc_counter
--Every clock tick the counter counts, but only when the enable signal is high
--When the maximum count has been reached, the count is reset the next clock tick.

--When the enable signal is low, the count is retained.
--When the reset signal is high the count is reset

--Both enable and reset signals are acted upon synchronously

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--entity declaration----------------------------------------------------------------
entity cyc_counter is
generic(system_speed, max_count : integer);
port(
    clk         : in    std_logic;
    enable      : in    std_logic;
    reset       : in    std_logic;
    output      : out   std_logic := '0'
    );
end cyc_counter;

-------------------------------------------------------------------------------------
architecture Behavioral of cyc_counter is

--signal declaration----------------------------------------------------------------
type state_type is (s0, counting);
signal state, new_state : state_type;
signal count, new_count : natural range 0 to max_count:= 0;

--State changes are registered on the clock tick
begin
sync  :  process (clk)
    begin
    if (rising_edge(clk)) then
        state <= new_state;
        count <= new_count;

        if new_count = max_count and enable = '1' then
            output <= '1';
        else
            output <= '0';
        end if;
    end if;
end process;

--State change logic
logic : process(clk, state, reset, count)
    begin
    case state is
        when s0=>
        if (reset = '1') then
            new_state <= s0;
        else
            new_state <= counting;
        end if;

        when counting =>
        if (reset = '1') then
            new_state <= s0;
        elsif (count > max_count-2) and (reset = '0') then
            new_state <= s0;
        else
            new_state <= counting;
        end if;
```

```
        when others =>
            new_state <= s0;
    end case;
end process;

--Behaviour attributed to each state
operation : process(state, count, enable)
    begin
    case state is
        when s0 =>
        new_count <= 0;

        when counting =>
        if enable = '1' then
            new_count <= count + 1;
        else
            new_count <= count;
        end if;

    end case;
    end process;
end Behavioral;
```

C.6. instruction_launcher.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural and structural design of a circuit that receives reads the input register
--and checks if a command has been received, then performs the instruction pertaining
--to the given command.
--This design utilises a cyc_counter unit, debounce_comparator and pulse_to_switch unit.

--Whenever a command has been received, the circuit switches from the default idle state
--to the busy state. When in the busy state, the counter is enabled. Whenever the counter
--reaches its maximum count is reset and the circuit switches to the idle state. For
--the duration of the busy state, no new command can be received.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

--entity declaration-------------------------------------------------------------------
entity instruction_handler is
generic(system_speed, baud_ticks: integer);
port(
    clk             :   in std_logic;
    input_reg       :   in std_logic_vector(63 downto 0);
    ready           :   out std_logic;

--instruction switches:
    switch          :   out std_logic;
    clear           :   out std_logic;
    calc            :   out std_logic
--    mode            :   out std_logic
    );
end instruction_handler;


architecture Behavioral of instruction_handler is
--signal declaration-------------------------------------------------------------------
type state_type is (idle, busy);
signal state, new_state : state_type;
signal switch_command, enable, reset, command_done, calc_command: std_logic := '0';  --mode_command

constant zeros5b: std_logic_vector(39 downto 0) := (others => '0');
constant zeros4b: std_logic_vector(31 downto 0) := (others => '0');

--component declaration----------------------------------------------------------------
component pulse_to_switch is
port(
        clk:        in std_logic;
        pulse_in:   in std_logic;
        output:     out std_logic
    );
end component pulse_to_switch;

component cyc_counter is
generic(system_speed, max_count : integer);
port(
    clk         : in    std_logic;
    enable      : in    std_logic;
    reset       : in    std_logic;
    output      : out   std_logic
    );
end component cyc_counter;

component debounce_comparator is
port(
        clk         :   in std_logic;
        db_in       :   in std_logic;
        pulse_out   :   out std_logic
    );
end component;
begin
--component wiring---------------------------------------------------------------------
pulse_to_switch_unit_cmd_state: pulse_to_switch
port map(clk=>clk, pulse_in=>switch_command, output=>switch);
reset <= not enable;
ready <= command_done;

--pulse_to_switch_unit_mode_state: pulse_to_switch
--port map(clk=>clk, pulse_in=>mode_command, output=>mode);
```

```
counter_unit: cyc_counter
generic map(system_speed=>system_speed, max_count=>90*baud_ticks)
port map(clk=>clk, enable=>enable, reset=>reset, output=>command_done);

debounce_comparator_unit    :   debounce_comparator
port map(clk=>clk, db_in=>calc_command, pulse_out=>calc);

--State changes are registered on the clock tick
sync:   process(clk)
begin
    if rising_edge(clk) then
        state <= new_state;
    end if;
end process;

--Behaviour attributed to each state
state_logic: process(state, input_reg, command_done)
begin
    case state is
        when idle =>
            if      input_reg = zeros5b & "01110000" & "01101100" & "01100110"  --flp
                or  input_reg = zeros5b & "01110000" & "01101101" & "01100011"  --cmp
                or  input_reg = zeros5b & "01100011" & "01101100" & "01100011"  --clc
                or  input_reg = zeros4b & "01100011" & "01101100" & "01100001" & "01100011"  --calc
--                or  input_reg = zeros4b & "01100101" & "01100100" & "01101111" & "01101101"  --mode
                  then

                new_state <= busy;
            else
                new_state <= idle;
            end if;

        when busy =>
            if command_done = '1' then
                new_state <= idle;
            else
                new_state <= busy;
            end if;
    end case;
end process;

--State change logic
operation: process(state)
begin
    case state is
        when idle =>
            enable <= '0';

        when busy =>
            enable <= '1';
    end case;
end process;

--Behaviour attributed to each state
command_operation: process(input_reg, switch_command)
begin
    if input_reg = zeros5b & "01110000" & "01101100" & "01100110" then   --flp
        switch_command <= '1';
    else
        switch_command <= '0';
    end if;

    if input_reg = zeros5b & "01100011" & "01101100" & "01100011" then   --clc
        clear <= '1';
    else
        clear <= '0';
    end if;

    if input_reg = zeros4b & "01100011" & "01101100" & "01100001" & "01100011" then --calc
        calc_command <= '1';
    else
        calc_command <= '0';
    end if;

--    if input_reg = zeros4b & "01100101" & "01100100" & "01101111" & "01101101" then --mode
--        mode_command <= '1';
--    else
--        mode_command <= '0';
--    end if;
end process;

end architecture Behavioral;
```

C.7. main.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Structural design of the main module

--This circuit contains all the modules that make up the functionality
--of the design. The main module is setup with constraints for the
--physical design.

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------------------------
entity main is
port(
    clk                 : in  std_logic;
    RXD                 : in std_logic;
```

```
    TXD                 : out std_logic;

--    MV_switch           : in std_logic;
    button              : in std_logic;
    calc_button         : in std_logic;

    green               : out std_logic;
    blue                : out std_logic;
    red                 : out std_logic;

    switch_led          : out std_logic
    );
end main;

---------------------------------------------------------------------------------------
architecture Structural of main is

--component declaration-----------------------------------------------------------
component UART_receiver is
generic(system_speed, baud_ticks: integer);
port(
    clk         :   in std_logic;
    RXD         :   in std_logic;
    clear       :   in std_logic;
    finished    :   out std_logic;
    output_reg  :   out std_logic_vector(UART_msg_len-1 downto 0)
    );
end component UART_receiver;

component UART_cipher_transmitter is
generic(system_speed, baud_ticks: integer);
port(
        clk         :   in std_logic;
        input_mat   :   in matrix_array(0 to FHE_CN-1, 0 to FHE_n);
        start       :   in std_logic;
        TXD         :   out std_logic := '1';
        finished    :   out std_logic := '1'
     );
end component UART_cipher_transmitter;

component button_switch is
generic(system_speed, db_interval: integer);
port(
        clk:    in std_logic;
        button: in std_logic;
        output: out std_logic
    );
end component button_switch;

component debounce_comparator is
port(
        clk         :   in std_logic;
        db_in       :   in std_logic;
        pulse_out   :   out std_logic
    );
end component;

component instruction_handler is
generic(system_speed, baud_ticks: integer);
port(
    clk             :   in std_logic;
    input_reg       :   in std_logic_vector(UART_msg_len-1 downto 0);
    ready           :   out std_logic;

--instruction switches:
    switch          :   out std_logic;
    clear           :   out std_logic;
    calc            :   out std_logic
    );
end component instruction_handler;

component APU is
port(
    clk         :   in std_logic;
    reset       :   in std_logic;
    input_reg   :   in std_logic_vector(UART_msg_len-1 downto 0);
    read_ready  :   in std_logic;
    calc        :   in std_logic;

    result_mat  :   out matrix_array(0 to FHE_CN-1, 0 to FHE_n);
    TXD_start   :   out std_logic
--    debug       :   out std_logic
    );
end component APU;

--signal declaration---------------------------------------------------------------
constant baud_ticks:    integer := 100;     --sets baudrate (125 -> 1E6 baud at system clock = 20E6)
constant db_interval:   integer := 4e5;     --(db_interval/system_speed = debounce time)
constant system_speed:  integer := 20e6;    --Hz
signal UART_out, receive_flag:  std_logic;
signal receive_reg:             std_logic_vector(L-1 downto 0);
signal calculation_result_mat:  matrix_array(0 to FHE_CN-1, 0 to FHE_n);
signal switch_out, debug:       std_logic;

signal APU_calc, calc, MV_mode, command_clear, dummy_ready, clear, calc_done, result_start: std_logic;
signal UART_finish_flag, calc_UART_out: std_logic;
signal UART_start: std_logic;

begin

--component wiring-----------------------------------------------------------------
UART_receiver_unit  :   UART_receiver
generic map(system_speed => system_speed, baud_ticks => baud_ticks)
port map(clk=>clk, RXD=>RXD, clear=>'0', finished=>receive_flag, output_reg=>receive_reg);

instruction_handler_unit  :   instruction_handler
generic map(system_speed=>system_speed, baud_ticks=>baud_ticks)
port map(clk=>clk, input_reg=>receive_reg, ready=>dummy_ready, switch=>switch_out, clear=>clear, calc=>calc);

APU_unit  :   APU
port map(clk=>clk, reset=>clear, input_reg=>receive_reg, read_ready=>receive_flag, calc=>APU_calc, result_mat=>calculation_result_mat, TXD_start=>calc_done);--, debug=>debug);
```

```
UART_calc_result_unit   :   UART_cipher_transmitter
generic map(system_speed => system_speed, baud_ticks => baud_ticks)
port map(clk=>clk, input_mat=>calculation_result_mat, start=>UART_start, txd=>calc_UART_out, finished=>UART_finish_flag);

--LED wiring-----------------------------------------------------------------------------------------------------------------------------------
red <= calc;
blue <= clear;
green <= receive_flag;

switch_led <= switch_out;
TXD <= calc_UART_out;

UART_start <= calc_done or button;

APU_calc <= calc or calc_button;

end Structural;
```

C.8. MatVect_types.vhd

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;

package types_package is

--Vector and matrix definitions----------------------------------------------------------------------------------------------------------------
type intmat is array (m-1 downto 0, n-1 downto 0) of signed(L-1 downto 0);
type intvect is array (n-1 downto 0) of signed(L-1 downto 0);

constant zeros_vect: intvect := (others => (others => '0'));
constant zeros_mat:  intmat  := (others => (others => (others => '0')));
constant UART_msg_len : natural := 64;

--FHE parameters--------------------------------------------------------------------------------------------------------------------------------

constant FHE_m: natural := 3;
constant FHE_n: natural := 3;
constant FHE_L: natural := 32; --80;

constant FHE_CN: natural := (FHE_n+1)*FHE_L;

constant FHE_Qm: natural := 16;
constant FHE_Qn: natural := 32;

type vector_array is array (natural range <>) of signed(FHE_L-1 downto 0);
type matrix_array is array (natural range <>, natural range <>) of signed(FHE_L-1 downto 0);

type vector_array_e is array (natural range <>) of signed(FHE_L-1 downto 0);
type matrix_array_e is array (natural range <>, natural range <>) of signed(FHE_L-1 downto 0);

type vector_bit_array is array (natural range <>) of std_logic;
type matrix_bit_array is array (natural range <>, natural range <>) of std_logic;

end package;

--package body----------------------------------------------------------------------------------------------------------------------------------
package body types_package is

end package body;
```

C.9. pulse_to_switch.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural and structural design of a circuit that switches between a low
--and high state when input changes from low to high (input is required to
--be a pulse (single clock tick)).

--The circuit has two states, high and low. The initial state is low. When
--the input (pulse_in) pulses high, the circuit switches between high and low.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

--entity declaration----------------------------------------------------------------------
entity pulse_to_switch is
port(
        clk:        in std_logic;
        pulse_in:   in std_logic;
        output:     out std_logic := '0'
    );
end pulse_to_switch;

--------------------------------------------------------------------------------------------
architecture Behavioural of pulse_to_switch is

--signal declaration----------------------------------------------------------------------
type state_type is (low, high);
signal state, new_state : state_type;
signal prev_pulse: std_logic;

begin
--behaviour-------------------------------------------------------------------------------
sync: process(clk)
begin
    if rising_edge(clk) then
```

```
        state <= new_state;
        prev_pulse <= pulse_in;
    end if;
end process;

logic: process(clk, state, prev_pulse, pulse_in)
begin
    case state is
        when high =>
        if pulse_in = '1' and prev_pulse = pulse_in then
            new_state <= low;
        else
            new_state <= high;
        end if;

        when low =>
        if pulse_in = '1' and prev_pulse = not pulse_in then
            new_state <= high;
        else
            new_state <= low;
        end if;

        when others =>
        new_state <= low;

    end case;
end process;

operation: process(clk, state)
begin
    case state is
        when high =>
        output <= '1';

        when low =>
        output <= '0';

        when others =>
        output <= '0';

    end case;
end process;

end Behavioural;
```

C.10. UART_cipher_transmit.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural and structural design of a circuit to send a UART signal.
--This design utilises two cyc_counter units.
--This circuit sends 8 bytes of data via UART when given a start signal.

--The circuit has two states, idle and sending. When given a start signal, the circuit
--starts sending out a UART signal containing the bytes supplied to the input register.
--When in the sending state, the baud counter and transmision counter are activated.
--The baud counter switches to high after one symbol duration (1/baud rate) and the
--transmission counter switches the done signal to high, prompting the circuit to
--switch to the idle state.

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.NUMERIC_STD.ALL;
    use IEEE.MATH_REAL.ALL;
library Work;
    use Work.types_package.all;

--entity declaration-------------------------------------------------
entity UART_cipher_transmitter is
generic(system_speed, baud_ticks: integer);
port(
        clk         :   in std_logic;
        input_mat   :   in matrix_array(0 to FHE_CN-1, 0 to FHE_n);
        start       :   in std_logic;
        TXD         :   out std_logic := '1';
        finished    :   out std_logic := '1'
    );
end UART_cipher_transmitter;

architecture Behavioral of UART_cipher_transmitter is
--signal declaration-------------------------------------------------------
constant bit_total  : natural := FHE_CN*FHE_CN + 2*integer(ceil(real(FHE_CN)*real(FHE_CN)/real(UART_msg_len)));

signal baudrate_clock, reset, enable: std_logic := '0';
signal done, new_done: std_logic := '1';
signal done_db: std_logic := '0';

signal bit_counter, new_bit_counter: natural range 0 to bit_total-1:= bit_total-1;

type state_type is (idle, sending);
signal state, new_state : state_type;

--function declaration-----------------------------------------------------------------

function shift_register_fun(input_mat: matrix_array(0 to FHE_CN-1, 0 to FHE_n); bit_counter: natural) return std_logic is
    variable st_dv, st_md, cnt, sel_i, sel_j, sel_k: natural;
    variable TXD_output: std_logic;
begin
    st_dv := bit_counter/(UART_msg_len+2);
    st_md := bit_counter - st_dv*(UART_msg_len+2);
        if st_md = 0 then
            TXD_output := '0';
        elsif st_md = UART_msg_len+1 then
            TXD_output := '1';
```

```
        else
            cnt   := bit_counter - 2*st_dv - 1;
            sel_i := cnt/FHE_CN;
            sel_j := cnt/FHE_L - sel_i*(FHE_n+1);
            sel_k := cnt - sel_i*FHE_CN - sel_j*FHE_L;
            if sel_i < FHE_CN then
                TXD_output := input_mat(sel_i, sel_j)(sel_k);
            else
                TXD_output := '0';
            end if;
        end if;
    return TXD_output;
end shift_register_fun;

--component declaration-------------------------------------------------------------------
component cyc_counter is
generic(system_speed, max_count : integer);
port(
    clk         : in    std_logic;
    enable      : in    std_logic;
    reset       : in    std_logic;
    output      : out   std_logic
    );
end component cyc_counter;

component debounce_comparator is
generic(init : std_logic := '0');
port(
        clk:        in std_logic;
        db_in:      in std_logic;
        pulse_out:  out std_logic
    );
end component debounce_comparator;


begin
--component wiring------------------------------------------------------------------------
baud_clock: cyc_counter
generic map(system_speed => system_speed, max_count=> baud_ticks-1)
port map(
            clk => clk,
            enable => enable,
            reset => reset,
            output => baudrate_clock
        );

debounce_comparator_unit: debounce_comparator
generic map(init => '1')
port map(
            clk=>clk,
            db_in=>done,
            pulse_out=>done_db
        );

finished <= not enable;

--state behaviour-------------------------------------------------------------------------
sync: process(clk)
begin
    if rising_edge(clk) then
        state <= new_state;
    end if;
end process;

state_logic: process(clk, state, start, done_db, baudrate_clock)
begin
    case state is
        when idle =>
            if start = '1' then
                new_state <= sending;
            else
                new_state <= idle;
            end if;

        when sending =>
            if done_db = '1' then
                new_state <= idle;
            else
                new_state <= sending;
            end if;

        when others =>
            new_state <= idle;
    end case;
end process;

operation: process(clk, state, bit_counter, input_mat)
begin
    case state is
        when idle =>
            enable <= '0';
            reset <= '1';
            TXD <= '1';

        when sending =>
            enable <= '1';
            reset <= '0';
            TXD <= shift_register_fun(input_mat, bit_counter);

        when others=>
            enable <= '0';
            reset <= '1';
            TXD <= '1';
    end case;
end process;
--output generation-----------------------------------------------------------------------

bit_send_sync: process(clk, done, baudrate_clock, new_bit_counter)
begin
    if rising_edge(clk) then
        if baudrate_clock = '1' or new_bit_counter = 0 then
            bit_counter <= new_bit_counter;
```

```
            done <= new_done;
        end if;
    end if;
end process;

bit_send_logic: process(clk, bit_counter, start)
begin
    if start = '1' then
        new_bit_counter <= 0;
        new_done <= '0';
    else
        if bit_counter = bit_total-1 then
            new_bit_counter <= bit_counter;
            new_done <= '1';
        else
            new_bit_counter <= bit_counter + 1;
            new_done <= '0';
        end if;
    end if;

    end process;

end Behavioral;
```

C.11. UART_receiver.vhd

```
--Author: Pieter Stobbe
--Date latest update:

--Behavioural and structural design of a circuit to receive a UART signal.
--This design utilises one counter unit, three cyc_counter units and a debounce unit.
--This circuit detects an incoming UART signal and polls the signal state at the
--midpoints of each bit (if a bit starts at t = 0s and has a duration of t = 1s
--then the state of the incomming bit is polled at t = 0.5s).
--The UART circuit has a 64-bit or 8-byte buffer in which input is stored.

--When there is no incomming signal (i.e. UART signal is high), the circuit is in
--the idle state. In this state, the input is connected to a debounce unit. When the
--input signal switches to low for half a symbol duration (1/baud-rate) an incomming
--UART signal is detected and so the circuit switches to the hold state.
--When in the hold state, the hold and reset counters are turned on. The reset counter
--ensures the input buffer is flushed after the time required to send a message of 8 bytes.
--In case a command of less that 8 bytes is received, the reset counter ensures that
--this command is not concatenated to the next message (See example at the bottom of the
--discription). The hold counter serves as a clock to introduce half a symbol duration of
--off-set. When the hold counter is done, the circuit switches to the receiving state,
--where the baud clock and transmision clock are turned on and input is polled everytime
--the baud clock is high. The transmission clock resets the state to the idle state when
--a byte of information has been received.

--The reset counter is a regular counter that has to be reset externaly, this prevents
--circuit from rapidly switching between enabling and disabling the reset counter when
--input buffer is filled and a new transmission starts immediatly after flushing the input buffer.

--Example of concatination issue:
--Take a command "clac". The FPGA expects "calc" and so the command "clac" is stored, but
--is not registered as a command. If the command correct command "calc" is sent next, the
--buffer now stores "claccalc", which is also wrong.
--The reset counter flushes the buffer, which means that sending "clac" and then "calc",
--the command is received without issue.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

--entity declaration-------------------------------------------------------------------
entity UART_receiver is
generic(system_speed, baud_ticks: integer);
port(
clk         :   in std_logic;
RXD         :   in std_logic;
clear       :   in std_logic;
finished    :   out std_logic;
output_reg  :   out std_logic_vector(63 downto 0) := (others => '0')
    );
end UART_receiver;

---------------------------------------------------------------------------------------
architecture Behavioral of UART_receiver is

--signal declaration-------------------------------------------------------------------
constant transmission_length: integer   := 63;
constant zeros: std_logic_vector(transmission_length downto 0) := (others => '0');

signal baudrate_clock, reset, enable, done: std_logic := '0';
signal hold_enable, hold_reset, hold_done: std_logic := '0';
signal db_RXD, not_RXD:  std_logic := '0';
signal bus_enable, not_bus_enable, bus_done: std_logic := '0';

signal shift_register : std_logic_vector(transmission_length downto 0) := (others => '0');
signal bit_counter, new_bit_counter: natural range 0 to transmission_length:= 0;

type state_type is (idle, hold, receiving);
signal state, new_state : state_type;

--component declaration----------------------------------------------------------------
component cyc_counter is
generic(system_speed, max_count : integer);
port(
    clk         : in    std_logic;
    enable      : in    std_logic;
    reset       : in    std_logic;
    output      : out   std_logic
    );
end component cyc_counter;
```

```
component counter is
generic(system_speed, max_count : integer);
port(
    clk          : in    std_logic;
    enable       : in    std_logic;
    reset        : in    std_logic;
    output       : out   std_logic
    );
end component counter;

component debounce is
generic(db_interval, system_speed: integer);
port(
        clk:    in std_logic;
        button: in std_logic;
        output: out std_logic
    );
end component debounce;

-------------------------------------------------------------------------------------
begin
--component wiring-------------------------------------------------------------------
baud_clock: cyc_counter
generic map(system_speed => system_speed, max_count=> baud_ticks-1)
port map(
            clk => clk,
            enable => enable,
            reset => reset,
            output => baudrate_clock
        );

transmision_clock: cyc_counter
generic map(system_speed => system_speed, max_count=> 8*baud_ticks - 1)
port map(
            clk => clk,
            enable => enable,
            reset => reset,
            output => done
        );

reset_clock: counter
generic map(system_speed => system_speed, max_count=> (10*8 + 1)*baud_ticks)
port map(
            clk => clk,
            enable => bus_enable,
            reset => not_bus_enable,
            output => bus_done
        );

phase_offset_clock: cyc_counter
generic map(system_speed => system_speed, max_count=> baud_ticks - 2) --4 clock ticks less, as it takes 3 clock ticks for the state machine to start receiving
port map(
            clk => clk,
            enable => hold_enable,
            reset => hold_reset,
            output => hold_done
        );

debounce_unit: debounce
generic map(baud_ticks/2-4, system_speed)
port map(clk=>clk, button=>not_RXD, output=>db_RXD);

finished <= bus_done;
reset <= not enable;
hold_reset <= not hold_enable;
not_bus_enable <= not bus_enable;

--State changes are registered on the clock tick
sync: process(clk)
begin
    if rising_edge(clk) then
        state <= new_state;
    end if;
end process;

--State change logic
state_logic: process(clk, state, done, bus_done, hold_done, db_RXD)
begin
    case state is
        when idle =>
            if db_RXD = '1' then
                new_state <= hold;
            else
                new_state <= idle;
            end if;

        when hold =>
            if hold_done = '1' then
                new_state <= receiving;
            else
                new_state <= hold;
            end if;

        when receiving =>
            if done = '1' then
                new_state <= idle;
            else
                new_state <= receiving;
            end if;

        when others =>
            new_state <= idle;
    end case;
end process;

--Behaviour attributed to each state
operation: process(clk, state, RXD)
begin
    case state is
        when idle =>
            enable <= '0';
```

```
            hold_enable <= '0';
            not_RXD <= not RXD;

        when hold =>
            enable <= '0';
            hold_enable <= '1';
            not_RXD <= '0';

        when receiving =>
            enable <= '1';
            hold_enable <= '0';
            not_RXD <= '0';

        when others=>
            enable <= '0';
            hold_enable <= '0';
            not_RXD <= '0';
    end case;
end process;

--output generation-------------------------------------------------
bit_receive_sync: process(clk, baudrate_clock, bit_counter, new_bit_counter, RXD, clear)
begin
if rising_edge(clk) then
    if (baudrate_clock = '1' or hold_done = '1') and clear = '0' and done = '0' and bus_done = '0' then
        bit_counter <= new_bit_counter;
        shift_register(bit_counter) <= RXD;
    elsif clear = '1' or bus_done = '1' then
        bit_counter <= 0;
        shift_register <= zeros;
    end if;
end if;
end process;

bit_counter_assignment: process(bit_counter, new_bit_counter)
begin
    if bit_counter = transmission_length then
        new_bit_counter <= 0;
    else
        new_bit_counter <= bit_counter + 1;
    end if;

end process;

cutting_transmisson: process(db_RXD, bus_done)
begin
    if rising_edge(db_RXD) then
        bus_enable <= '1';
    end if;

    if bus_done = '1' then
        bus_enable <= '0';
    end if;
end process;

--Register is updated everytime the input buffer is updated
output_register_assignment: process(shift_register, baudrate_clock)
begin
if rising_edge(baudrate_clock) then
    output_reg <= shift_register;
end if;
end process;

end Behavioral;
```

Bibliography

[1] Josh Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, September 1987. URL https://www.microsoft.com/en-us/research/publication/verifiable-secret-ballot-elections/. Accessed: 02/01/2021.

[2] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. Why textbook elgamal and rsa encryption are insecure. In *Advances in Cryptology — ASIACRYPT 2000*, pages 30–43, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44448-0.

[3] Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé, and Paul Zimmermann. Comparing the difficulty of factorization and discrete logarithm: A 240-digit experiment. *Advances in Cryptology – CRYPTO 2020*, page 62–91, 2020. doi: 10.1007/978-3-030-56880-1_3.

[4] G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610 – 611, 1958. doi: 10.1214/aoms/1177706645. URL https://doi.org/10.1214/aoms/1177706645.

[5] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 505–524, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22792-9.

[6] Mark Cannon. Discrete systems, 2014. http://www.researchgate.net/file.PostFileLoader.html?id=553cb110ef97137f6d8b45fe&key=4cabe738-7427-452b-a353-56d6c2397624&assetKey=.

[7] Mariano Perez Chaher, Bayu Jayawardhana, and Junsoo Kim. Homomorphic Encryption-Enabled Distance-Based Distributed Formation Control with Distance Mismatch Estimators. In *60th IEEE Conference on Decision and Control*, pages 4915–4922, 2021. ISBN 9781665436588.

[8] Adrian Chen. Google engineer allegedly fired for accessing private user information to stalk teens, Sep 2010. URL https://www.businessinsider.com/google-engineer-stalked-teens-spied-on-chats-2010-9?international=true. Accessed: 11/11/2020.

[9] J.H. Cheon, K. Han, S.-M. Hong, H.J. Kim, J. Kim, S. Kim, H. Seo, H. Shim, and Y. Song. Toward a secure drone system: Flying with real-time homomorphic authenticated encryption. *IEEE Access*, 6:24325–24339, 2018. doi: 10.1109/ACCESS.2018.2819189.

[10] Jung Hee Cheon and Damien Stehlé. Fully homomophic encryption over the integers revisited. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 513–536, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46800-5.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. *Advances in Cryptology – ASIACRYPT 2017*, page 409–437, 2017. doi: 10.1007/978-3-319-70694-8_15.

[12] Adrian Cho and Lucy Hicks. Ibm promises 1000-qubit quantum computer-a milestone-by 2023, Sep 2020. URL https://web.archive.org/web/20201203065646/https://www.sciencemag.org/news/2020/09/ibm-promises-1000-qubit-quantum-computer-milestone-2023. Accessed: 24/11/2020.

[13] Thomas W. Cusick and Pantelimon Stănică. *Cryptographic Boolean functions and applications*. Elsevier/Academic Press, 2017.

[14] Joe Danson and Ravindra Mittal. Oil and gas pipelines, 2022. https://www.globaldata.com/wp-content/uploads/2018/11/Oil-Gas-Brochure.pdf.

[15] Liesbeth De Mol. Turing machines, Sep 2018. URL https://plato.stanford.edu/entries/turing-machine/. Accessed: 01/02/2021.

[16] Christian de Schryver, Daniel Schmidt, Norbert Wehn, Elke Korn, Henning Marxen, and Ralf Korn. A new hardware efficient inversion based random number generator for non-uniform distributions. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 190–195, 2010. doi: 10.1109/ReConFig.2010.20.

[17] Christian de Schryver, Daniel Schmidt, Norbert Wehn, Elke Korn, Henning Marxen, Anton Kostiuk, and Ralf Korn. A hardware efficient random number generator for nonuniform distributions with arbitrary precision. *International Journal of Reconfigurable Computing*, 2012:1–11, 2012. doi: 10.1155/2012/675130.

[18] Whitfield Diffie and M Martin E. Hellman. New directions in cryptography. *IEEE Transactions on information theory*, November 1976.

[19] Thomas W. Edgar and David O. Manz. *Research Methods for Cyber Security*. Syngress Publishing, 1st edition, 2017. ISBN 0128053496.

[20] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 10–18, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39568-3.

[21] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[22] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *Advances in Cryptology – CRYPTO 2013 Lecture Notes in Computer Science*, page 75–92, 2013. doi: 10.1007/978-3-642-40041-4_5.

[23] Craig Gidney and Martin Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits, 2019.

[24] Peter Holslin. Fastest internet providers 2022, December 2021. https://www.highspeedinternet.com/resources/fastest-internet-providers#:~:text=Fiber%20is%20currently%20the%20fastest,and%20reliable%20over%20long%20distances.

[25] Junsoo Kim, Chanhwa Lee, Hyungbo Shim, Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Encrypting controller using fully homomorphic encryption for security of cyber-physical systems. *IFAC-PapersOnLine*, 49(22):175–180, 2016. ISSN 2405-8963. doi: 10.1016/j.ifacol.2016.10.392. 6th IFAC Workshop on Distributed Estimation and Control in Networked Systems NECSYS 2016.

[26] Junsoo Kim, Hyungbo Shim, and Kyoohyung Han. Dynamic controller that operates over homomorphically encrypted data for infinite time horizon, 2019.

[27] Junsoo Kim, Hyungbo Shim, Henrik Sandberg, and Karl H Johansson. Method for Running Dynamic Systems over Encrypted Data for Infinite Time Horizon without Bootstrapping and Re-encryption. In *60th IEEE Conference on Decision and Control*, pages 5614–5619, 2021. ISBN 9781665436588.

[28] Kiminao Kogiso and Takahiro Fujita. Cyber-security enhancement of networked control systems using homomorphic encryption. In *Conference on Decision and Control*, 12 2015. doi: 10.1109/CDC.2015.7403296.

[29] Jim Ledin. *Modern computer architecture and organization: Learn x86, ARM, and RISC-V architectures and the design of smartphones, pcs, and cloud servers*. Packt Publishing, 2022.

[30] George Marsaglia and Wai Wan Tsang. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journal on Scientific and Statistical Computing*, 5(2):349–359, 1984. doi: 10.1137/0905026. URL https://doi.org/10.1137/0905026.

[31] Christopher D. McFarland. A modified ziggurat algorithm for generating exponentially- and normally-distributed pseudorandom numbers. *CoRR*, abs/1403.6870, 2014. URL http://arxiv.org/abs/1403.6870.

[32] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. Cryptology ePrint Archive, Report 2011/501, 2011. https://ia.cr/2011/501.

[33] Nebojša Milenković, Vladimir Stankovic, and Miljana Milić. Modular design of fast leading zeros counting circuit. *Journal of Electrical Engineering*, 66:329–333, 11 2015. doi: 10.2478/jee-2015-0054.

[34] Subin Moon and Younho Lee. An efficient encrypted floating-point representation using heaan and tfhe. *Security and Communication Networks*, 2020:1–18, 03 2020. doi: 10.1155/2020/1250295.

[35] C. Murguia, F. Farokhi, and I. Shames. Secure and private implementation of dynamic controllers using semihomomorphic encryption. *IEEE Transactions on Automatic Control*, 65(9):3950–3957, 2020. doi: 10.1109/TAC.2020.2992445.

[36] NSA, 2009. URL https://web.archive.org/web/20090207005135/http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml. Accessed: 01/09/2020.

[37] NSA, 2015. URL https://apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm. Accessed: 01/09/2020.

[38] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. *Advances in Cryptology-EUROCRYPT'99*, page 223–238, 1999.

[39] Roger A. Prichard. History of encryption. Technical report, Global Information Assurance Certification Paper, January 2002. Accessed: 10/08/2020.

[40] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, May 2009. Accessed: 10/08/2020.

[41] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. doi: 10.1145/359340.359342. Accessed: 05/11/2020.

[42] Ben Rossi. How common is insider misuse?, May 2018. URL https://www.information-age.com/common-insider-misuse-123462235/. Accessed: 01/09/2020.

[43] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700. Accessed: 15/12/2020.

[44] Åström Karl J. and Wittenmark Björn. *Computer Controlled Systems*. Prentice-Hall, 1984.

[45] Åström Karl J. and Richard M. Murray. *Feedback systems: An introduction for scientists and Engineers*. Princeton University Press, 2021.

[46] technical document. Efficient shift registers, lfsr counters, and long pseudorandom sequence generators. Technical report, Xilinx, Inc., San Jose, California, July 1996. https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf.

[47] technical document. Arria 10 device overview. Technical report, Intel Corporation, September 2013. https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/hb/arria-10/arria-10-aib.pdf.

[48] technical document. Intel converged security and management engine (intel csme). Technical report, Intel Corporation, November 2020. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf.

[49] Julian Tran, Farhad Farokhi, Michael Cantoni, and Iman Shames. Implementing homomorphic encryption based secure feedback control. *Control Engineering Practice*, 97:104350, 2020. ISSN 0967-0661. doi: https://doi.org/10.1016/j.conengprac.2020.104350. URL http://www.sciencedirect.com/science/article/pii/S0967066120300319. Accessed: 25/08/2020.

[50] William Turton and Kartikay Mehrotra. Hackers breached colonial pipeline using compromised password, June 2021. https://www.bloomberg.com/news/articles/2021-06-04/hackers-breached-colonial-pipeline-using-compromised-password.

[51] Vinod Vaikuntanathan. Advanced topics in cryptography: Lattices, September 2015.

[52] Mark A. Will and Ryan K.L. Ko. Chapter 5 - a guide to homomorphic encryption. In Ryan Ko and Kim-Kwang Raymond Choo, editors, *The Cloud Security Ecosystem*, pages 101 – 127. Syngress, Boston, 2015. ISBN 978-0-12-801595-7. doi: https://doi.org/10.1016/B978-0-12-801595-7.00005-7. URL http://www.sciencedirect.com/science/article/pii/B9780128015957000057. Accessed: 15/08/2020.

[53] Mark A. Will and Ryan K.L. Ko. Chapter 5 - a guide to homomorphic encryption. In Ryan Ko and Kim-Kwang Raymond Choo, editors, *The Cloud Security Ecosystem*, pages 101 – 127. Syngress, Boston, 2015. ISBN 978-0-12-801595-7. doi: https://doi.org/10.1016/B978-0-12-801595-7.00005-7. URL http://www.sciencedirect.com/science/article/pii/B9780128015957000057. Accessed: 10/08/2020.

[54] Jeremy Wohlwend. Elliptic curve cryptographu: pre and post quantum. Technical report, MIT, 2016.

[55] Damien Zammit. Intel x86s hide another cpu that can take over your machine (you can't audit it), Mar 2020. URL https://boingboing.net/2016/06/15/intel-x86-processors-ship-with.html.