ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR NTRU ALGORITHM

SENIOR DESIGN PROJECT

Elif Nur İŞMAN
Canberk TOPAL

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JULY 2020

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR NTRU ALGORITHM

SENIOR DESIGN PROJECT

Elif Nur İŞMAN
040150214

Canberk TOPAL
040160057

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Assoc Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JULY 2020

<u>İSTANBUL TEKNİK ÜNİVERSİTESİ</u>
<u>ELEKTRİK-ELEKTRONİK FAKÜLTESİ</u>

RISC-V İŞLEMCİSİNİN KOMUT SETİNİN NTRU ALGORİTMASI İÇİN GENİŞLETİLMESİ

<u>LİSANS BİTİRME TASARIM PROJESİ</u>

Elif Nur İŞMAN
040150214

Canberk TOPAL
040160057

Proje Danışmanı: Doç. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

TEMMUZ, 2020

We are submitting the Senior Design Project Report entitled as “EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR NTRU ALGORITHM”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Canberk TOPAL ....
040160057

Elif Nur İŞMAN ....
040150214

FOREWORD

We would like to thank to our mentor Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın who helped us to find this project and who supported and guided us in all of our mistakes. Secondly, we would like to offer our gratitude to our mentor Res. Assist. M.Sc. Latif Akçay, who gave his attention, remarks and part of his own work to the project. Without them, we would spend lots of unnecessary time in order to finish our project. Finally, we would like to emphasize that we are grateful to our friends in İTÜ and our families who has the biggest role on our successes for our entire life.

June 2020

Canberk TOPAL
Elif Nur İŞMAN

TABLE OF CONTENTS

Page

FOREWORD .... **v**
TABLE OF CONTENTS .... **vii**
ABBREVIATIONS .... **ix**
SYMBOLS .... **x**
LIST OF TABLES .... **xi**
LIST OF FIGURES .... **xii**
SUMMARY .... **xiii**
ÖZET .... **xiv**
1.INTRODUCTION **15**
1.1 Basic Consepts 15
1.1.1 What is open source processors and why do we use it ? 15
1.1.2 What is instruction set architecture? 16
1.1.3 What is RISC-V processor and why etending ISA of it? 16
1.1.4 How to extend ISA? 16
1.2 Mathematical Background for NTRU Algorithm 16
1.2.1 NTRU keys and parameters: 17
1.2.2 Key generation 17
1.2.3 Encryption 18
1.2.4 Decryption 18
1.3 Preparing Work Environment 19
1.3.1 Installing Ubuntu 16.04 and required programs 19
1.3.2 Installing RISC-V GNU toolchain from GitHub 21
1.3.3 Preparing the software development environment 23
2. IMPLEMENTING AN OPEN SOURCE RISC-V PROCESSOR ON FPGA **24**
2.1 Implementing LowRISC Chip With Rocket Core 25
2.2 Implementing Ibex Core 25
3. RUNNING C PROGRAMS ON RISC-V CORE AND OPTIMIZATION OF THE NTRU C IMPLEMENTATION **27**
3.1 Compiling C Codes And The Structure Of Generated Memory 27
3.2 Running C Program in RISC-V Core 28
3.3 Implementation and Optimization of the NTRU Algorithm in C Programming Language 29
3.3.1 Optimization of the c program 30
3.4 Profiling the NTRU C Implementation 31
4. INSTRUCTION SET EXTENSION OF RISC-V PROCESSOR **32**
4.1 Software Part 33
4.1.1 Opcode space 33
4.1.2 Inline Assembly method 33
4.1.3 Chosen instructions and their types 34
4.1.4 Implementing the custom instructions to the C programs 34
4.1.5 Developing and testing the instructions using simple C programs 35

4.1.6 Adding custom instructions to the optimized C code 37
4.2 Hardware Part 37
4.2.1 Custom Module Design 38
4.2.2 Changes for multi-clock cycle operations 40
4.2.3 Connections with RAM and other modules 40
5. PERFORMANCE & TIMING ANALYSIS 42
5.1 Benchmark C Program 43
5.2 Behavioral Simulation to Check the Results 44
5.3 Using 7-Segment Display and LEDs to See the Results on Board 45
5.4 Usage of ILA IP 45
5.5 Comparing Selected Operation Implementations on Core with and without Custom Module 46
6. REALISTIC CONSTRAINTS AND CONCLUSIONS 47
6.1 Practical Application of this Project 48
6.2 Realistic Constraints 48
6.2.1 Social, environmental and economic impact 48
6.2.2 Cost analysis 48
6.2.3 Standards 49
6.2.4 Health and safety concerns 49
6.3 Future Work and Recommendations 49
REFERENCES 50
APPENDICES 54
CURRICULUM VITAE 70

ABBREVIATIONS

ALU : Arithmetic Logic Unit
CPU : Central Processing Unit
FF : Flip Flop
FPGA : Field Programmable Gate Array
GCC : GNU Compiler Collection
ILA : Integrated Logic Analyzer
IP : Intellectual Property
ISA : Instruction Set Architecture
LED : Light Emitting Diode
LUT : Look Up Table
NTRU : N-Truncated Polynomial Ring
NIST : National Institute of Science and Techonology
RAM : Random Access Memory
RISC : Reduced Instruction Set Computer

SYMBOLS

- ❖ : Terminal Commands

LIST OF TABLES

Page

Table 5.1 : Performance Results **47**
Table 5.2 : Area Usage Results... **47**

LIST OF FIGURES

Page

Figure 1.1: CLion Change of Kernel Variables Prompt **244**
Figure 2.1: List of SoC and Cores That Uses RISC-V ISA **244**
Figure 2.2: Modified Architecture of Ibex **266**
Figure 3.1: Simple Summation C Program for 32-bit RISC-V Core.....................**299**
Figure 3.2: Example of Optimization Process in the Implementation...................**300**
Figure 3.3: Speed Difference Between Optimized and Unoptimized Version **311**
Figure 3.4: Part of the Profiling Script for C Program...**322**
Figure 4.1: Instruction Format of R-type Instructions [18]...................................**333**
Figure 4.2: *instr_equ* Function with the Inline Assembly Method **344**
Figure 4.3: *array_equ* Function within the C Code **355**
Figure 4.4: Parts of the C Code for Testing Functionality of the Added Instructions **366**
Figure 4.5: Behavioral Simulation of Test C Code..**377**
Figure 4.6: Diagram of Execution Module **39**
Figure 4.7: State Flowchart for *Custom_Module* **39**
Figure 4.8: Input and Output Ports of Custom Module **411**
Figure 5.1: Main Part of the Benchmarking C Code **444**
Figure 5.2: Number of Clock Cycles Comparison Between the Custom Instructions and Basic C Operations...**444**
Figure 5.3: Dashboard Screen **466**

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR NTRU ALGORITHM

SUMMARY

The technological progress of humanity has reached a point that even science fiction writers cannot predict. With the ease of access to information and invention of social media, cyber security has become increasingly important in our lives. Especially the concept of the quantum computers, whose feasibility has ceased to be an issue of debate and the discussions moved towards the question of when it will start to affect daily life, threatens the security standards created by hundreds of engineers in the past decades. Quantum computing technology has taken priority in the field of information security, where companies and governments constantly compete, and algorithms that can resist the computing power of post-quantum computers gain importance in this field. NTRU algorithm, which is one of the few candidates approaching to be the standard for post quantum public key cryptography was discussed in this project.

The main problem in the field of electronic engineering; balance between area and performance has been the main issue that determined the limits in this project. A cryptography algorithm is needed in almost all electronic devices. Since the public key encryption method is especially used for the secure communication of the two parties, this algorithm is expected to work with high performance in internet of things devices and unmanned aerial vehicles. One of the prominent methods to achieve this high performance in small and relatively weak processors is to expand the instruction sets of these processors for this algorithm. We have created a project plan accordingly. One of the cheapest and most convenient methods for performing the instruction set expansion that forms the building block of our project was to modify an already designed open source processor. After achieving this goal, the FPGA card, one of the most important assistants of engineers working in this field, was used to test real-time results. In the project, the open source RISC-V processor has been implemented and developed on the Nexys 4 DDR FPGA card produced by Digilent company.

As a result, it was observed that the three candidate array arithmetic operation commands we added had an accelerating effect on the NTRU algorithm performance. This effect has the potential of especially to speeding up the implementation of secure communication protocols of small processors. In this study, it is envisaged that the processors to be produced may not be doomed to existing patterns and can be developed for use, and this study will increase the performance and added value of the product in that area.

RISC-V İŞLEMCİSİNİN KOMUT SETİNİN NTRU ALGORİTMASI İÇİN GENİŞLETİLMESİ

ÖZET

İnsanlığın teknolojik ilerleyişi bilimkurgu yazarlarının bile tahmin edemeyeceği bir noktaya gelmiş durumda. Bilgiye ulaşımın kolaylaşmasıyla ve sosyal medyanın icadıyla beraber, hayatımızda özellikle siber güvenlik giderek daha önemli bir yer alıyor. Özelikle son dönemde yapılabilirliği bir tartışma olmaktan çıkıp, ne zaman gündelik hayatı etkilemeye başlayacağı konuşulmaya başlanan kuantum sonrası bilgisayarlar geçtiğimiz on yıllarda yüzlerce mühendisin işbirliğiyle oluşturmuş olduğu güvenlik standartlarını tehdit ediyor. Şirketlerin ve devletlerin sürekli yarıştığı bilgi güvenliği sahasında önceliği kuantum hesaplama teknolojisi almış durumda ve bu alanda özellikle kuantum sonrası bilgisayarların işlem gücüne karşı durabilen algoritmalar önem kazanmakta. Bu algoritmalardan açık anahtar kuantum sonrası şifreleme algoritması standardı olmaya yaklaşan birkaç adaydan biri olan NTRU algoritması bu projede ele alındı.

Elektronik mühendisliği alanında baştan beri temel sorun olan alan ve performans dengesi bu projede sınırları belirleyen ana konu olmuştur. Bir kriptografi algoritmasına neredeyse bütün elektronik cihazlarda hitiyaç vardır. Açık anahtarlı şifreleme yöntemi özellikle iki tarafın birbiriyle güvenli bir şekilde haberleşmesi için kullanıldığından bu algoritmanın internet nesneleri cihazlarında ve insansız hava araçlarında yüksek performans ile çalışması beklenmektedir. Bu yüksek performansı küçük ve görece güçsüz işlemcilerde gerçekleştirmek için öne çıkan yöntemlerden biri, bu işlemcilerin komut setlerini bu algoritmaya yönelik şekilde genişletmektir. Biz de buna göre bir proje planı oluşturduk. Projemizin temel katmanını oluşturan komut seti genişletmesini gerçekleştirmek için en ucuz ve elverişli yöntemlerden biri, halihazırda tasarlanmış olan açık kaynak kodlu bir işlemciyi değiştirmekti. Bu amacı gerçekleştirdikten sonra gerçek zamanlı sonuçları test etmek için, bu alanda çalışan mühendislerin en önemli yardımcılarından olan FPGA kartı kullanıldı. Projede açık kaynak kodlu RISC-V işlemcisinin Digilent firması tarafından üretilen Nexys 4 DDR FPGA kartı üzerinde gerçeklenmesi ve geliştirilmesi yapılmıştır.

Sonuç olarak, eklemiş olduğumuz üç adet dizi aritmetiği komutunun NTRU algortiması üzerinde hızlandırıcı etki yaptığı görüldü. Bu etki, özellikle küçük işlemcilerin güvenli haberleşme protokollerini uygulamasını oldukça hızlandıracak potansiyele sahip. Bu çalışmada, üretilecek olan işlemcilerin varolan kalıplara

mahkum olmayıp kullanıma yönelik geliştirilebileceği ve bu çalışmanın ürünün o alandaki performansını ve katma değerini oldukça arttıracağı öngörülmüştür.

1. INTRODUCTION

With the developing technology, interest and investments in developing quantum computers are increasing rapidly[1, 2]. However, this poses a threat to cryptography algorithms used in every system where information security is needed today. The quantum era requires fundamental changes in information security. New cryptography algorithms that can resist post-quantum computers are being developed in order to maintain information security in banking, military and many other areas. In order to be usable and practical in daily life, low area usage and low performance are prioritized in the algorithms created. N-Truncated Polynomial Ring Units(NTRU)[3] is one of the most promising postquantum cryptography algorithms as they are among 28 standardization candidates in the National Institute of Science and Techonology(NIST) competition for public key cryptography[4].

Based on the Reduced Instruction Set Computer (RISC) architecture[5], RISC-V[6] is an open source alternative to a world of proprietary instruction set architectures. Our project aims to increase the performance of a NTRU cryptosystem application on an open source, low-power RISC-V processor. The plan is to increase the performance by extending the instruction set with most commonly used operations in the application.

1.1 Basic Consepts

1.1.1 What is open source processors and why do we use it ?

A processor, or "microprocessor," is a small chip that positioned in computers and other electronic devices. Its basic job is to receive input and provide the appropriate output, according to the structures that embedded on it. This may seem like a simple task at the first glance, but

processors of today’s world can handle trillions of calculations per second. The most basic processor will include a register file, an ALU, system memory, and a control unit that allows the processor to make decisions based on the instruction it's executing.

1.1.2 What is instruction set architecture?

The Instruction Set Architecture (ISA)[5] design is one of the most critical structures for a processor. Designing it properly and correctly at the beginning is very important. It is accessible by the programmer or compiler writer. It defines the relationship and boundaries between software and hardware. User can have knowledge about supported data types, registers, interrupts, the hardware support for managing main memory features and the input/output model of a bunch of implementations by examining it.

1.1.3 What is RISC-V processor and why etending ISA of it?

There are various popular instruction sets that are used in the industry and each one of them has its own unique usage and advantages. Reduced Instruction Set Computer (RISC) is one of them. It has fewer cycles per instruction. Instructions are simple, fewer, more general and usually fixed-length. Registers are also fixed-length, generally. This type of ISA is easy to develop control logic on, requires lower area, lower power. But besides its advantages, it has low performance. RISC-V is an open standard ISA based on established RISC principles. To be able to accelerate the NTRU implementation, some complex instructions would be useful. We are planning to create custom instructions that consume less clock cycles to execute operations in the NTRU algorithm.

1.1.4 How to extend ISA?

The decision for the custom instructions will be done by detecting operations that are repetitive and spend many clock cycles in the NTRU C code. Since simple instruction architectures will not be suitable to execute them, we will extend the execution block in the core by creating a custom module that can be execute the instructions we will create.

1.2 Mathematical Background for NTRU Algorithm

NTRU differs from the previously found public key cryptosystems by the foundations it is based on which is the shortest vector problem in a lattice[3]. NTRU is shown as an alternative to Rivest-Shamir-Adelman (RSA)[7] and Elliptic Curve Cryptography (ECC)[8] by using a lattice-based approach to cryptography.

A truncated polynomial ring $R = Z[X]/(X^{N-1})$ that is created based on the determined parameters form the backbone of the steps in NTRU cryptosystem. During the process, different polynomials are created by using the R and all of them have to have integer coefficients and degree at most $N-1$.

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1} \tag{1.1}$$

1.2.1 NTRU keys and parameters:

N - the polynomials in the ring R have degree $N-1$. (Non-secret)

q - the large modulus to which each coefficient is reduced. (Non-secret)

p - the small modulus to which each coefficient is reduced. (Non-secret)

f - a polynomial that is the private key.

g - a polynomial that is used to generate the public key h from f (Secret but discarded after initial use)

h - the public key, also a polynomial

r - the random ``blinding'' polynomial (Secret but discarded after initial

use)

d - coefficient

1.2.2 Key generation

If two person named Alice and Bob are communicating through a secure channel, sending a secret message from Alice to Bob requires the generation of a public and a

private key. While the public key is known by both sides, private key should only be known by the receiver[9].

As the first step, two polynomials named f and g in the R are selected randomly. Chosen polynomials with degree at most $N-1$ with coefficients $[-1,0,1]$ must be invertible. Then, the inverse of f according to modulo q (f_q) and modulo p (f_p) should be calculated. Operations, especially in the decryption part, will be depend on the f_q and f_p satisfying the equations:

$$f * f_q = 1 \, mod \, q \tag{1.2}$$

and

$$f * f_p = 1 \, mod \, p \tag{1.3}$$

In the third step, public key h will be calculated with the equation $h = p * (f_q * g) \, mod \, q$. f and f_p are used to create a longer and protected private key.

1.2.3 Encryption

As the beginning, message to be transmit will put in the form of polynomial and represented with m, with coefficients $[-1,0,1]$.

$$m = 1 - X^2 + X^5 - X^7 + X^{10} \tag{1.4}$$

Than, a 'blinding value' is chosen randomly to obscure the message. Blinding value is a small polynomial that represented with r.

$$r = 1 + X^1 + X^2 - X^3 - X^9 \tag{1.5}$$

Last step of the encryption is to calculating the encrypted message by the equation:

$$e = r * h + m \, mod \, q \tag{1.6}$$

1.2.4 Decryption

Private key that Bob have is the combination of f and f_p, as mentioned before, Private key is the only information Bob has aside from the encrypted message. He can try to solve the message by using f. First, he multiplies the e and f, represent the result with a polynomial a.

$$a = f * e \, mod \, q \tag{1.7}$$

If equation is rearranged with the equality of e:

$$a = f * (r * h + m) mod q \quad (1.8)$$

$$a = f * \left(r * p f_q * g + m\right) mod q \quad (1.9)$$

$$a = pr * g + f * m \ mod\ q \quad (1.10)$$

Instead of choosing the coefficients of a between 0 and $q - 1$, they are chosen in the interval $[-q/2 , q/2]$. Aim of this is to prevent that the original message may not be properly recovered since Alice chooses the coordinates of her message m in the interval $[-p/2 , p/2]$

Next step will be calculating a modulo p, result will be represented with polynomial b:

$$b = a \ mod\ p \quad (1.11)$$

Since modulo of $pr * g$ equals to 0,

$$b = f * m \ mod\ p \quad (1.12)$$

Now, Bob can use the f_p to recapture m, by multiplication of b and f_p.

$$c = f_p * b = f_p * f * m mod p \quad (1.13)$$

$$c = m \ mod\ p \quad (1.14)$$

1.3 Preparing Work Environment

In the final version of this project a modified RV32IMC RISC-V core is implemented on the Digilent FPGA card Nexys 4 DDR[10]. However, before this selection, the first candidate was a RV64GC core. In the following sections, preparation for the implementation of these candidates are given. The implementation of both candidates are done using Xilinx Vivado 2018.1[11] and the reason behind it is to automatizing the synthesis, implementation, place and route processes and using the generate bit-stream feature to program the FPGA card. In order to focus the main effort on design and not dealing with side problems such as driver failures and tool bugs, a Linux based system is installed on the computers from the beginning.

1.3.1 Installing Ubuntu 16.04 and required programs

In order to run the project, Ubuntu 16.04.5 LTS [12] is chosen as the default operating system. Linux based operating system is chosen because complex open source projects like processors needs to have a building scripts and their own tools in order to work properly. This situation requires certain packages and tools like CMake[13] and RISC-V GNU Toolchain [14]. Installing these requirements is a lot more easier in Linux based systems than Windows based ones.

After installing Ubuntu, Vivado 2018.1 should be installed from the Xilinx website. After its installation, in order to run the program effortlessly, one should edit her .bashrc file in the /home directory.

- source /opt/Xilinx/Vivado/2018.1/settings64.sh

This ensures when the terminal is called in the system, settings64.sh script would always be entered in the background. In addition to that, Vivado 2018.1 requires external drivers to be installed in order to recognise the FPGA cards from Digilent. For Nexys 4 DDR, following drivers need to be installed into the system.

- Adept 2.16.1 Runtime, X64 DEB
- Adept 2.2.1 Utilities, X64 DEB

After installing Vivado 2018.1 and Ubuntu 16.04 with the required packages, one could begin to implement an open source processor as a Vivado project. As it is mentioned in the previous section, there are many open source RISC-V processors in the internet. Initial aim of this project was to implement a 64-bit RISC-V processor because of the large number of bits in the cryptographic algorithms would make use of 64 bits fully. So, the first candidate to implement was lowRISC[15] chip with Rocket core. The core is RV64GC which means it is 64 bits and includes G and C

standard extensions. In the next sections, implementation steps for its FPGA project and the reason behind the switching of the processor is explained.

CMake must be installed to compile and install complex projects and programs to the system. To install CMake Linux version 3.13.2, it can be downloaded from https://cmake.org/download/ [13] . After downloading the files, the commands below must be entered inside the download folder to the terminal

- ./bootstrap
- make
- make install

1.3.2 Installing RISC-V GNU toolchain from GitHub

Any open source RISC-V processor repository needs its own development tools like RISC-V compiler, ISA simulator et cetera to be installed. In order to install these tools, system should have some necessary packages. The command below does the installation of the necessary packages:

- sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libncurses5-dev libusb-1.0-0 libboost-dev

Also, in order to use GitHub effectively in Linux operating system, git tool needs to be installed. So, in the first step one should open a terminal and enter this command:

- sudo apt-get install git

When every necessary package is built on the system, one could start to install the RISC-V tools. This project includes C programs to be run in the digital system itself. That means the compiled version of the C code should not include any libraries in the Linux operating system. In order to ensure there is no such error in the tools, one should install the tools with their cross-compilation bare-metal version. Normally, computers compile and run their programs for their own system. Cross-compilation means the compilation process is done in another system (in this case Linux operating system) for the processor.

In order to install the compiler, one should build the RISC-V GNU Toolchain using the commands below:

- git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

- sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev

- ./configure --prefix=/opt/riscv --with-arch=rv64gc

- make

This processes takes 20-30 minutes total depending on the performance of the computer. The last command installs the RV64GC compiler. However, when the target processor changed to its final version RV32IMC a slight modification needed to be done. The modified command can be seen below:

- ./configure --prefix=/opt/riscv --with-arch=rv32imc
- make

After installing the toolchain, now by calling:

- ❖ which riscv32-unknown-elf-gcc

One could make sure the compiler is installed correctly.

After the installation of all the necessary programs for the hardware part of this project, the preparation of the software development environment may begin.

1.3.3 Preparing the software development environment

NTRU has many open source C code implementations ranging from key generation, encryption and decryption functions to the whole cryptosystem. In this project aim was to increase the performance of NTRU in the encryption, decryption and key generation functions.

In order to work and test different C codes an Integrated Development Environment needs to be installed to the system. CLion 2019.3.1 was te choice for that role because of its ease of use and its own profiling tools. In order to install CLion to Ubuntu 16.04LTS one should download the compressed file and enter the following commands:

- ❖ sudo tar xvzf CLion-2019.3.1.tar.gz -C /opt/
- ❖ sh /opt/clion- 2019.3.1/bin/clion.sh

After installing CLion to the system, it is now possible to create and edit C projects with it. However, one small installation needs to be made in order to use the CPU Profiler tool of the program.

- ❖ uname -r

This command returns the exact version of the operating system, which will be used when installing the dependencies of the Profiler tool. In the systems that this project is done, the version was 4.15.0-106-generic

- ❖ sudo apt-get install linux-tools-4.15.0-106-generic

After that upon the first launch, CLion asks for the kernel variable changes. The reason behind that is to record the changes in the kernel while being not the root user. Since the profiler essentially analyzes the kernel it is important to record the logs of it.

Figure 1.1: CLion Change of Kernel Variables Prompt

2. IMPLEMENTING AN OPEN SOURCE RISC-V PROCESSOR ON FPGA

With the developing technology After the preparation of the work environment regarding hardware and software design, the implementation process for the open source RISC-V processors begun. In this aspect, several candidate processors and system on chips have been considered. These are Ariane core, Ibex (formerly Zero-riscy) core and LowRISC SoC which uses rocket core.

Cores

Name	Supplier	Links	Priv. spec	User spec	Primary Language	License
Ibex (formerly Zero-riscy)	lowRISC	GitHub	1.11	RV32I[M]C/RV32E[M]C	SystemVerilog	Apache 2.0
Ariane	ETH Zurich, Università di Bologna	Website,GitHub	1.11-draft	RV64GC	SystemVerilog	Solderpad Hardware License v. 0.51

SoC platforms

Name	Supplier	Links	Core	License
Rocket Chip	SiFive, UCB BAR	GitHub,Simulator	Rocket	BSD
LowRISC	lowRISC	GitHub	RV32IM	BSD

Figure 2.1: List of SoC and Cores That Uses RISC-V ISA

Ariane is a 64-bit RISC-V core which implements six stage pipeline structure with single issue, in-order architecture. LowRisc SoC also uses a 64-bit RISC-V core however implemented core on the system, rocket core, has not a six stage but a five stage pipeline structure. There are more than a dozen of different cores from different

companies with different hardware description languages. So, better documented and more comprehensible cores would be the bigger candidates in this project. It is decided to use more simple core because it is thought that the modification of the core will be complex enough.

2.1 Implementing LowRISC Chip With Rocket Core

In this project, first candidate system for the modification of the processor was the system on chip designed by lowRISC organisation. The name of this SoC is lowRISC-Chip and it contains a core,

After that, in order to use the open source chip it is needed to clone the project files to the computer. The command below does the cloning:

- ❖ git clone -b refresh-v0.6 --recursive https://github.com/lowrisc/lowrisc-chip.git

After installing necessary tools, packages and Vivado 2018.1 finally the project files for the SoC can be built. Running the commands below in the /fpga/board/nexys4 path results with the verilog files and the vivado project of the system as a whole.

- ❖ make vivado
- ❖ make project

Since the output Verilog files of the whole project is automatically generated from Chisel it is assumed that the editing of the Verilog codes would not be difficult. After closer inspections in the Rocket Core, it is seen that the assumption is simply wrong. The reason behind that is the nature of the connections in the Verilog files are highly complex because the main idea is to write them in a higher level hardware description language, Chisel. Since learning a new hardware description language from scratch and editing the processor using that language is beyond the scope of this project it is decided to move on to another open source RISC-V processor named Ibex.

2.2 Implementing Ibex Core

32-bit Ibex[15] core is chosen as a suitable RV32IMC core because of its hardware design language, SystemVerilog[16], and detailed documentation. Architecture of Ibex can be seen in Fig.2.2

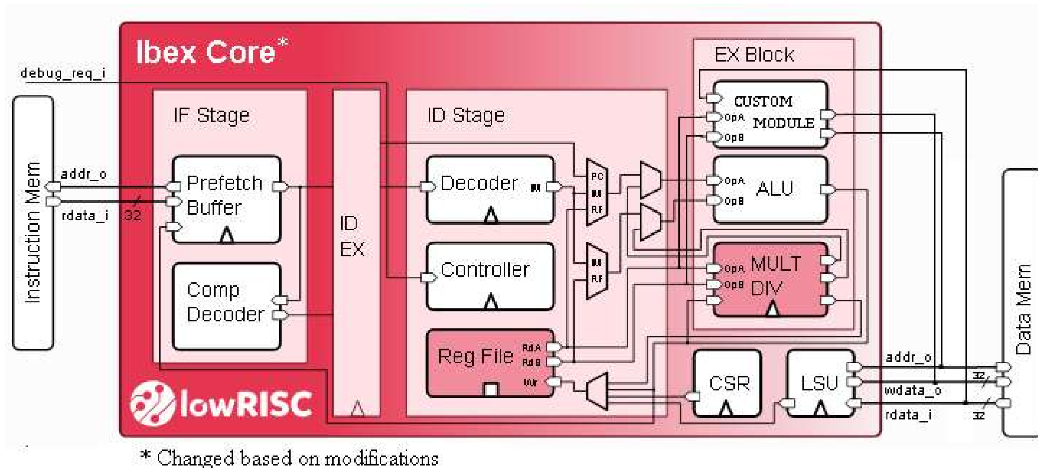


Figure 2.2: Modified Architecture of Ibex

Core simply takes instructions from an instruction memory and acts like a two-stage pipeline processor. Which means it fetches the instructions and buffers the inputs of the decoder. However, there is no buffer for the execution stage of the processor which makes the design two stage. Ibex has its own dependencies and in this section all the steps that are executed will be explained. However, the modifications on the core and how its done will be explained in the next sections.

The Ibex project needs srecord, fusesoc and pip to be built. In order to install these dependencies one should enter the commands:

- ❖ sudo apt install python-pip
- ❖ sudo apt-get install srecord
- ❖ sudo pip install fusesoc

After installing the dependencies, first thing to do is to clone Ibex repository to the desired path using the following command:

- ❖ git clone --recursive https://github.com/lowRISC/ibex

After cloning the repository by going into ibex/examples/fpga/artya7/ directory and entering the following command would ensures the vivado project to be built with an example of generated bitstream with the altering led lights in the Nexys 4 DDR FPGA card.

- ❖ make build-arty-100 program-arty

After this command the generated Vivado project will include synthesis, implementation and bitstream of the example described above. The whole project includes a memory which acts like both instruction and data memory. By changing the memory path in the SystemVerilog file of the memory, core could execute any compiled program at this point. How to run a program with this method will be explained in detail in the following sections.

3. RUNNING C PROGRAMS ON RISC-V CORE AND OPTIMIZATION OF THE NTRU C IMPLEMENTATION

After building the the RISC-V core project. The processor fetches the instructions from the memory and executes them in order while writing up to the register file and the memory itself. In order to see this process, behavioral simulation of the system needs to be done. The behavioral simulation is chosen for analyzing the inner works of processor and the intricacies of the whole process. Since behavioral simulation allows for the inspection of the intermediate signals in the design it is used for the verification of the both modified and unmodified versions of the processor. First task to do after the implementation was to compile some example C codes and verify the inner workings of the processor. In the next section how to generate a memory file from a C program will be explained.

3.1 Compiling C Codes And The Structure Of Generated Memory

Generation of the compilation files including executable linked file (.elf), disassembly file (.dis), binary file(.bin) and memory file (.mem) is done by a script which is in the Makefile format. The script needs three inputs in the same directory with itself. First a C program, in the example makefile script, the C program is named "2d_array". After that, in order to show where to put the instructions into the memory and setting up the general flow of the memory structure a linker script and a C runtime file is needed. The example makefile script can be seen in the appendix. In this project, the linker script and C runtime file from the demonstration example named link.ld and crt0.S is used.

Using this Makefile, one can generate the memory file of any C program by simply entering the command below:

- ❖ make

Likewise, simply entering the command below would delete the generated outputs of the script:

- ❖ make distclean

This script mainly uses riscv32-unknown-gcc compiler that is built in the previous section. This compiler simply turns C program to an Assembly program and then generates binary instruction codes from each instruction one by one. After that process it simply divides the binary file into 32-bit hexadecimal parts.

When the generating binary codes of the instructions, compiler does a simple comparison using opcode mask and opcode match method. In this method fetched instruction is XORed by all the opcode matches. After that it is masked by applying opcode mask bits. If it is equal to zero, the generated opcode would be correct. Opcode matches are unique codes for each opcode spaces. The term opcode space and the opcode space of custom instructions will be explained in the later sections. The same method also applied for recognizing the individual instructions. That means every instruction has its own mask and match code for generating the binary files.

More details about the instruction generation and manipulation of the compiler in order to generate binary codes of custom instructions will be explained in the "Instruction Set Extension" section.

3.2 Running C Program in RISC-V Core

After generating the necessary memory file for the C programs, one should include the destination of the memory file as a variable in the Vivado project settings. SRAM_INIT_FILE variable is created for that reason and it simply initializes the memory structure with the generated instructions. After setting the variable as desired, core would perform all the instructions given. In order to analyze the core, it is decided to work with the behavioral simulation. Specifically, input and output signals of the external memory, execution stage of the core and the outputs of the decoder and register file are inspected with the waveform. In one of the first example C program, a

simple addition of two 32-bit variables is tested. The result of these two variables are stored in a distinct RAM address.

```
1   #include <stdint.h>
2
3   int main(int argc, char **argv) {
4
5
6   uint64_t    a = 3994674403;
7   uint64_t    b = 3132440345;
8
9   volatile unsigned int  *var = (volatile unsigned int  *) 0x0000c010;
10
11  *var = a+b;  //Summation
12
13  return 0;
14  }
```

Figure 3.1: Simple Summation C Program for 32-bit RISC-V Core

Reason for trying this kind of simple C programs is simply for trying to understand the overall process of the core by following most of the signals from fetch to write back stages. After successfully tracing necessary signals, it is decided to move onto finding the implementation of NTRU algorithm in C programming language.

3.3 Implementation and Optimization of the NTRU Algorithm in C Programming Language

As its mathematical background and history explained before, NTRU is a candidate for post quantum cryptography public key standard. This means it needs a lot of implementation which are secure, optimized and open source. Since the complex mathematical background of the algorithm with its need for array like implementation is out of the scope of this project, it is decided to implement a ready to use NTRU C code. In this effort two main candidates are considered. First one is named "libntru"[17] and it is a highly optimized version of NTRU algorithm for both Java and C programming languages. However, compatible compilation of this implementation was not successful. The reason for that was the high instruction memory and external library usage of the implementation. The reserved memory for the instruction memory is 48kB and for the stack is 16kB in this project and because it is a hardcoded limitation, libntru implementation is eliminated as acandidate. The second option is to use the custom implementation of NTRU algorithm with 48bit message length and parameters N = 53, p = 3 and q = 101. Since this implementation

is written by our faculty member Res. Assist. M. Sc. Latif Akçay, it was fairly simple to troubleshoot and integrate it to the system. However, it was unoptimal since it is mainly designed for functional correctness.

3.3.1 Optimization of the c program

The optimization of the code started with identifying the flow of the code. More than dozen of sequential for loops have been found in the code itself. Since the implementation is fairly modular with fundamental functions like polynomial multiplication and polynomial division, optimizing with a simple principle would yield significant improvement in the performance. The whole optimization step consists of first, reducing the number of for loops by combining unnecessary ones into one loop. Second, removing the printing statements which are there for debugging purposes. At last, removing residual variables for better memory usage is done. An example figure for the optimization process involving before and after of a snippet of the code for modulus operation in arrays is given in Figure 3.2

```c
// make mod calculations
for (i = 0; i < size_a; ++i){
        while (pola[i] < 0){
                pola[i] = pola[i] + mod;
        }
}

for (i = 0; i < size_a; ++i){
        pola[i] = pola[i] % mod;
}

for (i = 0; i < size_b; ++i){
        while (polb[i] < 0){
                polb[i] = polb[i] + mod;
        }
}

for(i = 0; i < size_b; ++i){
        polb[i] = polb[i] % mod;
}
```

BEFORE (UNOPTIMIZED)

```c
// make mod calculations
for (i = 0; i < size_a; ++i){
    pola[i] = a[i];
    while (pola[i] < 0){
        pola[i] = pola[i] + mod;
    }
    pola[i] = pola[i] % mod;
}

for (i = 0; i < size_b; ++i){
    polb[i] = b[i];
    while (polb[i] < 0){
        polb[i] = polb[i] + mod;
    }
    polb[i] = polb[i] % mod;
}
```

AFTER(OPTIMIZED)

Figure 3.2: Example of Optimization Process in the Implementation

Both the optimized and unoptimized version of the program is run in the computer before running it in the processor. The reason for that is the ineffective speed of the behavioral simulation. In detail, the whole program is so complex that in order to simulate the full operation with behavioral simulation, one may use a very high performance computer. As a result, the optimized and unoptimized versions of the program is compared with the designing computer itself. The result shows that optimization process caused a performance gain of 14.48 percent.

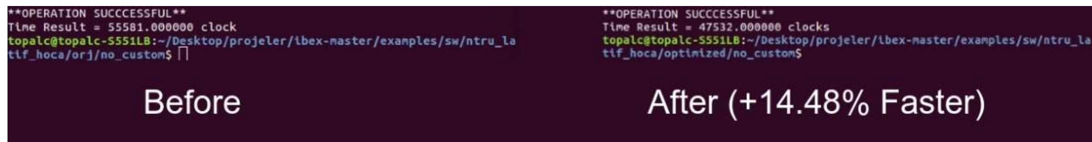


Figure 3.3: Speed Difference Between Optimized and Unoptimized Version

3.4 Profiling the NTRU C Implementation

In order to continue the next step of the project which is implementing custom instructions to the processor, fundamental step to take is choosing the custom instructions. Since this project uses the implementation with the C programming language and there was no realistic way was found for this project to edit the compiled executable linkable format, the profiling step of the project took place in the C program too. The editing of the executable linkable format (.elf) file was necessary because the binary code of the whole program is generated in that file. In other words, for changing the content of the instruction memory and adding custom instruction codes to it one should try to edit the source of that generated memory. However, it is found a more simple and high-level way to approach this problem. By editing the C program and use the compiler in such a way that generates the desired custom instruction codes there was no need for being involved in complex problems. The details about this solution will be explained in the next section with the subsection of inline assembly method.

In order to find the best suiting instruction for this application, it is thought that finding the most frequent function and trying to reduce it to an instruction would be an ideal scenario. So, the profiling of the optimized code is done by a script that uses a counter to record how many times a particular part of the program is called. The script is used with the commands:

- riscv32-unknown-elf-gcc -g -O3 <name of the c program> -o <desired name of the object file>
- riscv32-unknown-elf-gdb --command= <name of the gdb script>

```
 1 set pagination off
 2 set logging file gdb.txt
 3 set logging on
 4 file NTRU48
 5 target sim
 6 load NTRU48
 7
 8 start
 9 set $ctr13 = 0
10 set $ctr50  = 0
11 set $ctr57 = 0
12 set $ctr88 = 0
13 set $ctr116 = 0
14 set $ctr123 = 0
15 set $ctr143 = 0
16 set $ctr206 = 0
17 set $ctr274 = 0
18 set $ctr306 = 0
19 set $ctr424 = 0
20 set $ctr512 = 0
21 set $ctr546 = 0
22 set $ctr569 = 0
23 set $ctr571 = 0
24
25
26 break NTRU_48_bitM.c:13
27 commands
28         set $ctr13 = $ctr13 + 1
29         c
30 end
```

Figure 3.4: Part of the Profiling Script for C Program

The result shows that the polynomial multiplication function in the code is called 535 times while the polynomial division function is called 136 times. These functions mainly include basic array arithmetic operations like element-wise addition, element-wise subtraction and element-wise equalization. After confirming the counter results again with the CLion profiler tool, the process for implementing these array operations into the instruction set begun

4. INSTRUCTION SET EXTENSION OF RISC-V PROCESSOR

After profiling the optimized NTRU code and finding the candidate instructions to add to the ISA itself, different methods for implementing is discussed. In order to get a broader view for this problem, in the first subsections some important terms is needed to be explained. The first thing to choose was the type of the custom instructions in order to generate their machine codes. R-type instructions, as shown in the Figure 4.1 is chosen due to their availability for two operands per instruction.

Figure 4.1: Instruction Format of R-type Instructions [18]

4.1 Software Part

4.1.1 Opcode space

Opcode Space[18] is a term for a group of instructions that have the same instruction type and enable the same sort of response in the core to some degree. For example, all 32-bits ALU instructions are in the same opcode space, OP_32[18]. There is also three different opcode spaces for customized instructions. In this project, opcode space *CUSTOM_0* is used.

4.1.2 Inline Assembly method

Modifying the compiler to include the custom function and adding a *.insn* directive to the source code is considered as two different options for changing the machine code of the project. Between these two options, using *.insn* directive is picked for its ease of use. The directive is an assembly directive; in order to combine it with the C code, inline assembly method[19] is used.

Generally the inline term is used to instruct the compiler to insert the code of a function into the code of its caller at the point where the actual call is made. The benefit of inlining is that it reduces function-call overhead. As can be understood from the definition, inline assembly is a set of assembly instructions written as inline functions.

Inline assembly structure is created and used for each instruction with the name *instr_*[instruction]. *instr_equ* can be seen in Fig. 4.2 as an example. *asm* stands for 'assembly' and volatile indicates that the variables can be modified from outside of the C program. *".insn r CUSTOM_0, 0x7, 5, %0, %1, %2 \n"* is the assembly code that define a R-type custom0_rd_rs1_rs2 instruction. Registers can be appointed automatically by % symbol. 0x7 is the funct3 value which specify that instruction will take 2 registers (rs1 and rs2) and returns to a destination register (rd). 5 is the funct7 value. funct7 is used for specifying the individual instructions in the same opcode space. In the project, 0x03 implies array addition, 0x05 implies array equalization and 0x06 implies element-wise modulus operation. First of the two lines below the

assembly code indicates the output operands and second indicates the input operands. Result of the a1 and a2 arrays will be written to the a1 in this case.

```
void instr_equ(unsigned int *a1, unsigned int *a2){

    asm volatile(
    ".insn r CUSTOM_0, 0x7, 5, %0, %1, %2  \n"
    : "=r"(&a1[0])
    : "r"(&a1[0]), "r"(&a2[0])

    );
    return;
}
```

Figure 4.2: *instr_equ* Function with the Inline Assembly Method

4.1.3 Chosen instructions and their types

There are different instruction formats for different needs and for a custom instruction, R-type instruction format is chosen due to its capability of using two different source registers. Other types are in lack of this property or are not in a proper structure. The third register in R-type, destination register, is simply equals to the first operand in our structure since, all of the added custom instructions were designed to be self returning instructions. However, using the methods mentioned above on any type of instruction could be added to the instruction set simply by changing their parameters in inline assembly code.

4.1.4 Implementing the custom instructions to the C programs

Parallelization is a powerful application for increasing the timing performance. To be able to execute parallelization, we imagined of a structure that processes every element of the source array at the same time. But it was not possible due to limitations of the instruction structure. We can send maximum two source operands(rs1 and rs2) and receive one result (rd) by using R-type instruction structure. Therefore it was not possible to design a digital system to calculate all the elements, both for the huge area cost and the insufficiency of the number of operands. For example for a simple array operation one might need three operands: array1, array2 and length of the both arrays. Since the last operand, length cannot be fit into the instruction itself we decided to implement its value in the hardware by hard-coding it to the module. The detailed explanation for this implementation can be found in the next chapters. There are many parts in the optimized C code with "for loops" where basic operations done for

elements of arrays. The solution we found is to using addresses of the source arrays and lengths of them in a wrapper function since the instruction itself could not contain all three information. The detailed explanation of the hardware implementation of this part is in the hardware section.

As can be seen from Figure 4.2, custom instruction with the inline assembly method contains, first element addresses of the arrays a1 and a2. These are assigned to the source registers. However, there is a difference between the *instr_mod* and the other two. A second array is not needed in array modulation. A second variable called *mod* will be used as the second input and is not a pointer. Based on this *mod* value, the modulus of each array element will be calculated. In the hardware part which will be explained later on, three elements of each array are processed by calling *instr_*[instruction] functions.

4.1.5 Developing and testing the instructions using simple C programs

In the previous subsections, we said that *instr_equ* is helpful to process three elements of each array at the same time. But in the C code of the NTRU algorithm, there are many elements with different array lengths use the custom operations. To be able to execute operations on arrays that have more than 3 elements and with different capacities, we create a new function structure. *array_equ* shown in the Fig. 4.3 created to execute equilization and is one of the three functions that the structure is used.

```
void array_equ(int *a1,int *a2,int length) {

    int i = 0;
    switch(length%3) {

        case 0:
            for (i = 0; i < (length / 3); i++) {
                instr_equ((unsigned int*)&a1[3 * i],(unsigned int*) &a2[3*i]);
            }
        case 1:
            for (i = 0; i < ((length-1) / 3); i++) {
                instr_equ((unsigned int*)&a1[3 * i],(unsigned int*) &a2[3*i]);
            }
            a1[length-1] = a2[length-1];
        case 2:
            for (i = 0; i < ((length-2) / 3); i++) {
                instr_equ((unsigned int*)&a1[3 * i],(unsigned int*) &a2[3*i]);
            }
            a1[length-1] = a2[length-1];
            a1[length-2] = a2[length-2];
    } //end of switch case
} //end of function
```

Figure 4.3: *array_equ* Function within the C Code

In *array_equ*, there is a switch structure that helps to decide how many times to call the *instr_equ*. If *length* is divisible by 3 without remainder, *instr_equ* will called *length/3* times, since *instr_equ* is handling 3 array elements from each array at a time.

If *length* is divided by 3 with remainder 1, *instr_equ* will called $(length - 1)/3$ times and last elements will be equilize by operand '='.

If *length* is divided by 3 with remainder 2, *instr_equ* will called $(length - 2)/3$ times and last two elements will be equilize by operand '='.

This function differs for *array_mod*. Like the *instr_mod*, *mod* value will be used instead of the a2 pointer.

After building the structure for array operations, tested three functions (*array_mod, array_add, array_equ*) on simple C codes to see if they are giving the wanted results correctly. Global array definitions and main part of the C code written for testing, without instruction definitions for simplicity, is given in the Figure 4.4

```
int array1[17] = {-1,-2,-3,1,2,3,4,5,6,7,8,9,0xa,0xb,16,1,6};
int array2[17] = {0xa,0xb,0xc,0xd,0xe,0xf,0xa1,0xa2,0xa3,0xa4,0xa5,0xa6,1,2,3,2,8};
int main() {

    array_add(array1,array2,17);

    array_mod(array1,7,17);

    array_equ(array1,array2,17);

    return 0;
}
```

Figure 4.4: Parts of the C Code for Testing Functionality of the Added Instructions

2 global arrays were defined at the beginning; array1 and array2. They defined as having 17 elements and random values are assigned to them. At the main part, custom instructions were tried one by one.

To see the outcomes, created .mem file for the code and run it on the core in Vivado environment. By debugging, we made sure that the correct commands were entered in custom_module written in hardware. By examining behavioral simulation, we made sure that added instructions are working correctly. Results of the test can be seen in Figure 4.5.

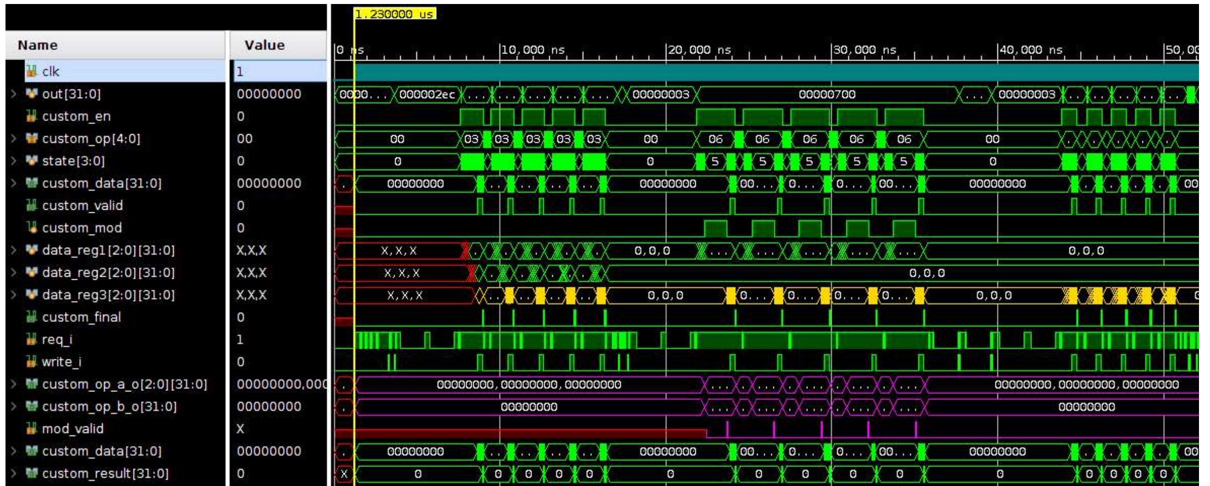


Figure 4.5: Behavioral Simulation of Test C Code

4.1.6 Adding custom instructions to the optimized C code

Optimized NTRU code has many for loops which are aimed to be replaced with the functions *array_equ, array_mod and array_add*. Changed the loops with the functions array_mod(array1, mod, [array_length]), array_add(array1, array2, [array_length]) and array_equ(array1, array2, [array_length]). Final version of the NTRU C code is given in appendix.

4.2 Hardware Part

After the candidate instructions for extension is chosen, modification in the core are mainly done to the execution stage of the structure. In order to not get the illegal opcode error in the core itself, first thing to do is introducing the custom instructions to the instruction decoding stage of the core. This is done by adding a new state in the decoder hardware of the core for the specific opcode space *CUSTOM_0*. Whenever that opcode space is decoded in the decoder, an enable signal and the specific opcode of the instruction is sent to the added module in the execution stage.

In the execution stage of the core, there are two modules present in the vanilla version. First one is Arithmetic Logic Unit(ALU) and another module is for the multiplication and division module MUL/ DIV. ALU is used for the single cycle operations while MUL/ DIV is used for multi cycle operations, the detailed explanation for the multi cycle instructions and their connections will be given in the next subsection. In this project, a new custom module and a new remainder module are added to the execution stage for enabling the array operations in the core.

4.2.1 Custom Module Design

In the first iteration of the project, all the new instruction is designed in a way that the all operations would be done in a single custom module. However, in the future iterations it would be found that the design is mostly inefficient and not using the full features of hardware design. By simply writing a driver module to pull the data from RAM and then dispersing the data into a parallel network of modules would be seen most efficient way to implement the custom instructions.

In the second iteration of the design, making use of the already existing MUL\DIV module would seem appropriate because of its advantage for the area problem. However, even adding two new modules of MUL\DIV would create a lot bigger designs. In order to solve that issue, the search for the specific part that is responsible for the remainder instruction begun. Upon finding that part of the module, creating a new module that uses the specific part was seem to be enough. After doing the testing of that design it is found that the inner architecture of interconnected MUL\DIV module and ALU module was the main problem for the area. Since the remainder instruction part uses ALU to do its basic calculations, in order to do the full parallelization of the process, one has to multiply the number of ALU's as well. Since this is a pretty big addition to the execution stage and it would complicate the system too much the project moved to its next and final iteration.

In the final version of the project, two new modules are added in the execution stage of the core. Input signals of the execution stage is divided between the original modules and the custom module. Outputs of the execution stage is chosen in respect to the enable signal that comes from the decoder. The diagram that summarizes the connection between the sub-modules of execution stage module is shown in the Figure 4.6. Remainder module is a module that is used for doing the modulo operation using the non-restoring division algorithm implementation[20]. The reason for not using the inherent REM instruction of RISC-V is that the core itself use both ALU and MUL\DIV modules to execute REM instruction. Thus, parallelization of this structure is costly in terms of area usage. In order to increase the performance by doing the same operation in the same time, three instances of this module is generated in the execution stage.

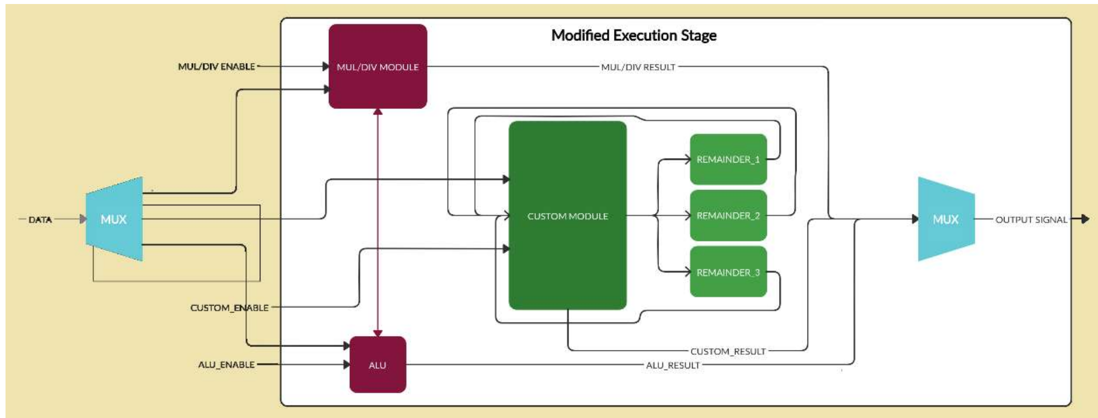


Figure 4.6: Diagram of Execution Module

Another module named *custom_module* acts like a driver between the memory of the system and the remainder module. Also, it does simple algebraic computations like additions and equalization. Main structure of the module consists of eight states, which are shown in Figure 4.7.

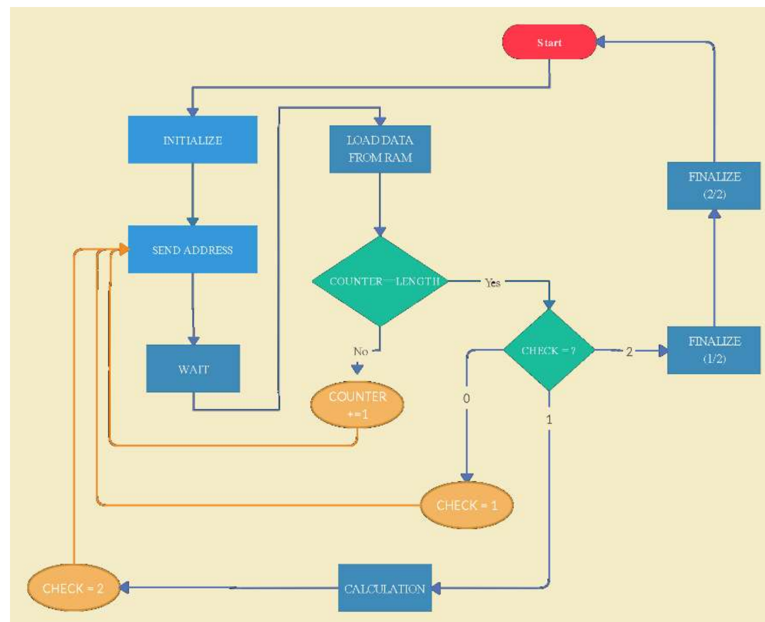


Figure 4.7: State Flowchart for *Custom_Module*

State zero initializes the counter, temporary registers etc. First state is for sending the address information to the memory and retrieve the corresponding data. In order to read from the memory, one clock cycle has to be passed. So, second state is used for that delay. Third state is for loading the incoming data into local registers *datareg_1, datareg_2*. This loading sequence is repeated until the capacity of the local registers are full. In this project, it is decided to use three data at a time because of the trade-off between area and increase in performance. Fourth state is for sending the modulus

operands to remainder instances or doing the addition. Action in the fifth state is determined by the unique opcode of the instruction. In the fifth state results are stored in another local register called *datareg_3*. Contents of this register is sent to the memory of the system by changing the control signal *check* and return the module to state 1. After *valid* signal is set to high, the module goes into the end routine, from which it sends the control signals to the core to end the multi cycle waiting process. Also, it gives out the final result to the destination register as intended.

4.2.2 Changes for multi-clock cycle operations

Any process that reaches the memory structure and pulls multiple data then executes the same operation has to be multi clock cycle. That is because of the nature of the memory structure would allow only one data to be read in one clock cycle. So, there is a need for implementing multi clock cycle instructions to the core.

The core already has multi clock cycle instructions from MUL\DIV module. An unfinished instruction should send a signal to the instruction fetch stage of the core, the reason behind it is that if the fetch stage continues to work after one clock cycle, decoder receives another instruction and the core moves on to the next instruction to fetch. That will conclude with erroneous results. The control signals inside of instruction fetch stage are modified to include the custom module enabling signals and another signal that indicates the validity of the custom module outputs. Whenever the custom module is enabled, core would enter a waiting stage just like if it receives a MUL\DIV enabling instruction. Moreover, parallel to MUL\DIV module when the *custom_valid* signal is high the core would start to fetch and decode instruction from where it left off.

4.2.3 Connections with RAM and other modules

In this section, all the connections of custom module will be listed with their brief explanations.

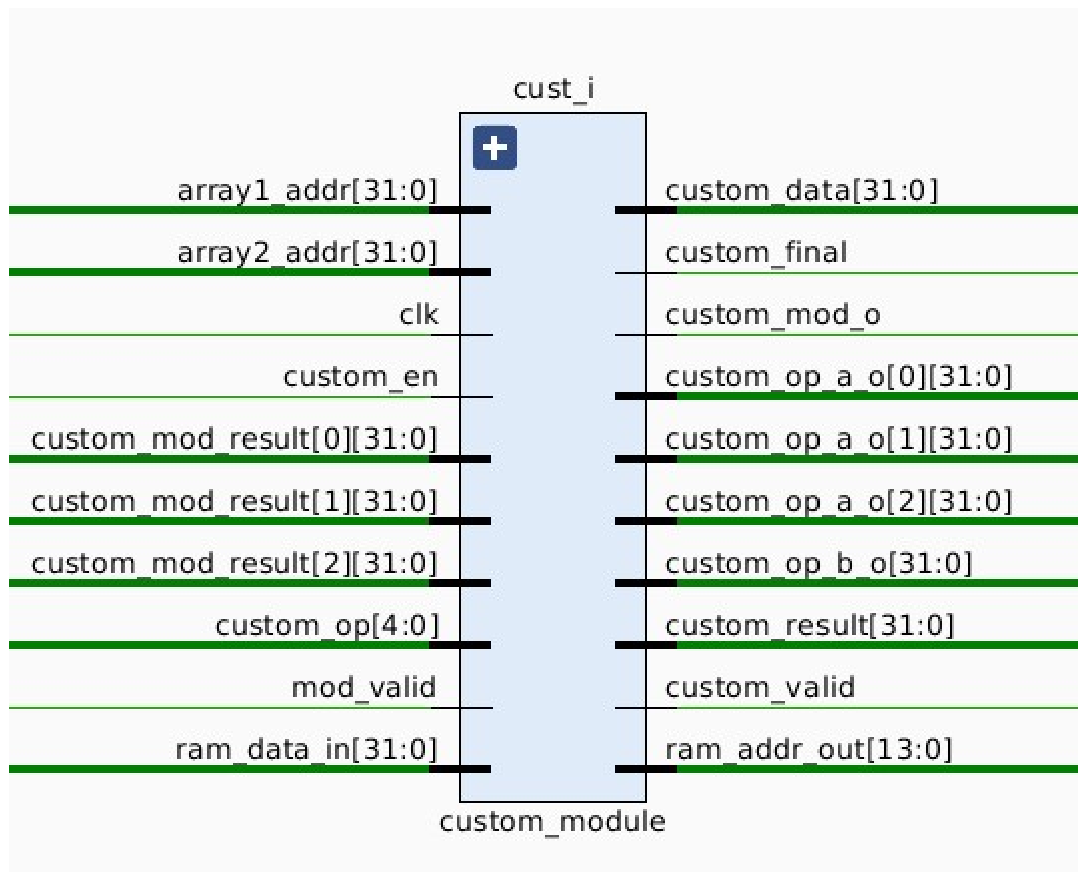


Figure 4.8: Input and Output Ports of Custom Module

- As it can be deducted from their names array1_addr and array2_addr ports of the module corresponds to the two operands of the instruction. For the modulation instruction array2_addr port acts like it is the divisor.
- clk port takes the same clock signal as the execution stage input clock.
- custom_en is the active high enable port that ensures the module only activates when the relevant instruction is decoded.
- custom_mod_result ports are the inputs to custom module that comes from the custom written three remainder modules. Remainder modules would take their inputs from the custom module, calculates the remainders and send it back to custom module as explained in the hardware part section before.
- custom_op is the port for specifying the individual instructions. The individual opcodes for instruction would be decoded in the decoder. So, this port gets its input from the decoder.

- mod_valid is the port that checks if the remainder modules provide the valid results at the same time. In order to check that, this port takes its input as the result of three input and gate which includes valid signals from all three modules.
- ram_data_in is the port for pulling the data from RAM module.
- custom_data is the output port for pushing resulting data to RAM module.
- custom_final is the output port that implies the multiclock instruction is finished. Hence, it is connected to the instruction fetching module and related to enabling the stall signals of the core.
- custom_mod_o is the output port that enables the all three remainder modules when the opcode for modulus instruction comes.
- custom_op_a_o and customopbo output ports are the inputs to the remainder module as described above.
- custom_result is the output port that sends the address of the first array since all the instructions are self returning type.
- custom_valid is the output port for indicating the results needed was calculated thus controlling the write enable port of the RAM module.
- ram_addr_out is the output port for controlling the RAM module address input.

5. PERFORMANCE & TIMING ANALYSIS

We have tested the custom instruction's functionality as mentioned previously. And after the structural improvements on both software and hardware, we had the implementation of the NTRU C code with custom instructions on the modified core.

In this section, we will explain how time analysis is done on the instructions and whole structure. Also, the results of these analyzes will be shown.

5.1 Benchmark C Program

Benchmarking[21] can be defined simply as measuring relative performance of an object by using a computer program or a set of programs. To be able to do timing analysis, we have used this method.

The C code, written for testing the functionality of the custom instructions, is explained before in the 'Developing and Testing the Instructions Using Basic C Programs' subsection. This code is updated to measure how fast each specific operation was compared to using basic C operations such as '+', '%' and '='. There were two arrays defined globally: array1 and array2. Another global array named *resultkon* is defined.

resultkon defined as a global array because, we wanted the data to be saved in RAM, such as array1 and array2. In this way, we can see the test signals at the output. The resultkon, which contains 1 element, was used for a slightly more specific job than others.

In the main part of the code, the custom operations were called in order for testing. However, when written in this way, only the result of the last operation could be observed; It was not possible to measure how long each process took. Therefore, by making use of the sequential working feature of the C programming language, some specific values are assigned to the resultkon array after each custom instruction called. Thus, resultkon served as a control signal here. As seen in the Figure 5.1 the *array_add* command is called between the assignments of the 0xdebdebde and 0xdcdcdcdc signals to the resultkon. It was assumed that the difference between the times when two specified control signals were seen at the output, was equal to the time it took to sum two 17-element arrays. Similarly, how many clock cycles the modulation and equalization processes took by using two 17-element arrays were measured.

```
int main() {
    result_kon[0] = 0xdcdcdcdc;
    array_add(array1,array2,17);
    result_kon[0] = 0xdebdebde;
    array_mod(array1,7,17);
    result_kon[0] = 0xdabadaba;
    array_equ(array1,array2,17);
    result_kon[0] = 0xAAbabaaa;
    return 0;
}
```

Figure 5.1: Main Part of the Benchmarking C Code

A comparison of the implementation of these processes with the custom operations and the basic C library is given in Table

İdeal Koşullarda..(17 elemanlı array)	add	mod	equ
cust	279	454	199
norm	335	894	314
improvement(-%)	0.1671641791	0.4921700224	0.3662420382

Figure 5.2: Number of Clock Cycles Comparison Between the Custom Instructions and Basic C Operations

5.2 Behavioral Simulation to Check the Results

After the behavioral simulations on the trivial C code examples, it's time to test the final version of the NTRU C code. When we used behavioral simulation, see that heavy computing need of it would make simulating the whole operation practically impossible.

In order to solve this problem, NTRU implementation has to be tested in the real-world conditions. An FPGA card is used to implement the whole project and run the C code on the core. The card we used was Nexys 4 DDR which is in Xilinx's Artix-7 FPGA family.

5.3 Using 7-Segment Display and LEDs to See the Results on Board

There were some changes needed in RAM(ram_1p) and clock generator (clkgen_xil7series.sv) module to be able to create bitstream and run it on the board. Assignment to the CLKIN1_PERIOD in the clock generator is needed only when behavioral simulation is running.10 assigned to it because of the clock period determined in the testbench module. As for RAM, created .mem file is read by the *$display* and *$readmemh* commands in *[initial begin .... end]* block for the behavioral simulation. This part replaced with a *['ifdef ... 'endif]* block that helps reading the SRAM_INIT_FILE for creating bitstream. SRAM_INIT_FILE which shows the .mem file, can be changed from the Tools/Settings/General/Verilog options/Defines.

The memory structure of the project is quite simple, it has the same memory for both data and instructions. All global arrays that are in the C code is saved in a constant address in the memory. Data output of the core which changes rapidly is also saved in the memory and we wanted to observe changes on it during the NTRU algorithm. We had two main ideas to observe the output that named *'data_wdata_o'*: Using LEDs on the FPGA board and observe the output on the 8-digit 7-segment LED display. But, data changes so fast that human eye could not catch and distinguish the changing values on both of them. Observation problem is solved by using Integrated Logic Analyzer (ILA) IP.

5.4 Usage of ILA IP

ILA[22] is a module that was used for measuring and reading the values from the project. In order to see the memory input and core output of the project, one probe of the ILA is connected to the wire that connects both of them in the top module.

When FPGA board programmed with the bitstream, a dashboard is opening in the hardware manager. We used trigger setup and waveform in the dashboard options list.

As mentioned before, main function of the C code includes both key generation, encryption and decryption algorithms for NTRU. In an effort to measure the performance improvements that are achieved for individual parts of the cryptosystem, a specific data assigned to our control signal *resultkon* at the beginning and the end of the each function.

By simply checking the number of clock cycles or time between the two uniquely designated signals, an accurate measurement is concluded.

This check is done using trigger setup and waveform. Trigger setup used to detect a specific value in the probe of the ILA. Our control signal connected to the probe with the name *leds* and it is what we are looking for to see *resultkon* values assigned in the C code. In the waveform, looking for the *leds* and a clock counter to detect the each values time to come.

Dashboard screen can be seen in the Figure 5.3. Different control signals are looked for by changing the leds value in the trigger setup code.

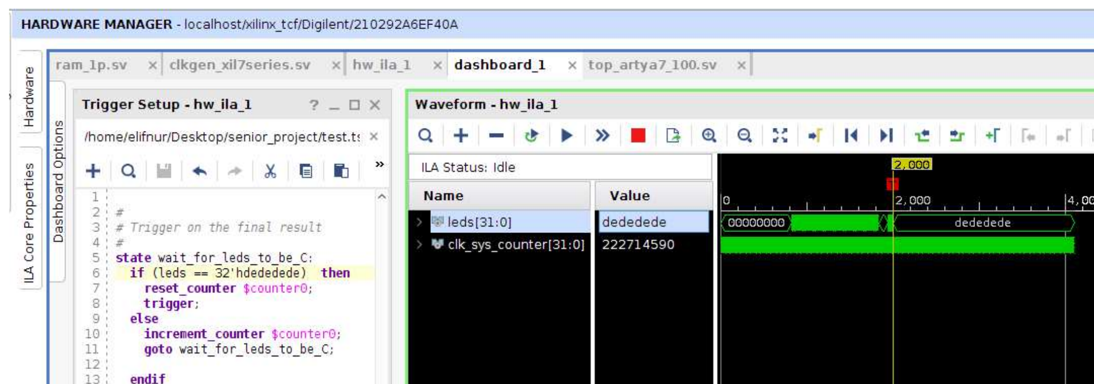


Figure 5.3: Dashboard Screen

There is a reason why we insert the control signal into the functions. When we used it between calling the functions in the main part, measurements were not logical. So, we measure a function's time consumption by detecting beginning and end of it.

5.5 Comparing Selected Operation Implementations on Core with and without Custom Module

Performance measurements are made with the previously mentioned method. First measurement was the unmodified NTRU code with the modified core. Array equalization instruction made an unexpected increase in the clock cycles. It is possible that it is because of the disruption of the optimization process of the compiler, caused by the new instruction. However, the overall effect of that instruction in the last configuration shows that it can be a helpful in decreasing the clock cycles in some combinations. It is also worth mentioning that there was no change in the working frequency of the core in the case of modified version.

Table 5.1 : Performance Results

Added Instruction	Clock Cycles	Improvement
Vanilla	329,281,688	-
MOD	255,087,137	- 22.53%
ADD	310,181,046	- 5.80%
EQU	345,059,480	+4.79%
MOD + ADD	224,556,529	−31.80%
MOD + EQU	321,277,395	−2.43%
ADD + EQU	312,095,182	−5.22%
MOD + ADD + EQU	222,705,262	−32.37%

The area measurements are made by implementing unmodified Ibex core and comparing the utilization report of it with the modified core. To be able to obtain more ideal results, from the Tools/Settings/Synthesis/Strategy, we choosed 'Flow_AreaOptimized_high'. This property helps Vivado tool to synthesis the project with high area optimization.

Increase in the usage of Look-up tables (LUT) and flip-flops (FF) are shown in the Table II.

Table 5.2 : Area Usage Results.

Status	LUT	Increase	FF	Increase
VANILLA	2991	-	1923	-
MODIFIED	37.44%	25.18%	2929	+52.31%

While the number of clock cycles are decreasing, the critical path of the whole system does not change significantly. Results of this project shows that with a directly connected data memory and a custom driver for the data handling in execution stage would improve the overall performance of the core for NTRU cryptosystem implementation.

6. REALISTIC CONSTRAINTS AND CONCLUSIONS

Post-Quantum cryptography is one of the issues that should be studied in all aspects for information security both today and in the coming years. Although NTRU is the oldest of the candidate algorithms, it stands out with its security level and processing speeds. In this study, new instructions are designed and implemented on an open-

source RISC-V processor to speed up NTRU Crypyosystem operations effectively. For this purpose, NTRU Cryptosystem is firstly designed in C language as a software application. Then, profiling is applied at a functional level with a classical method to determine the most frequently used blocks. New instructions that implement the operations of the detected blocks are designed and integrated into the processor core. The designs are tested on FPGA and compared with others for all versions. According to the results obtained, even if the resource utilization increases the design requirement slightly, it has provided a serious improvement in terms of performance.

6.1 Practical Application of this Project

This project proves that by adding carefully planned, custom instructions to the ISA, great performance improvements could be delivered with the small changes in the core. Most of the complex and computing intensive operations could be simplified this way.

6.2 Realistic Constraints

Most important impact of this project is that it takes advantage of open source hardware designs. Big open source projects like RISC-V processors give a great reduction in time spent dealing with the implementation part of the project. This extra time would allow designers to think more creatively and design products faster.

6.2.1 Social, environmental and economic impact

The society we live in today is called the information society. Military and state secrets are shared interchangeably, also personal and social information are shared too. The possibility of leakage of information that have an importance at any level, raises the need to protect and encrypt said information. It is of utmost importance to design equipment that will ensure this healthy and secure sharing environment and implement crypto protocols. Fast and efficient design and performance of these equipment will make information sharing safer and healthier.

6.2.2 Cost analysis

A FPGA evaluation board that faculty management meets was the essential cost factor of this project. In addition to the FPGA evaluation board, Vivado development

environment is also used for implementing the whole project to the board and debugging it using ILA. Since the project mainly uses open source sources for its operating system and other necessities, there are no other cost factor.

6.2.3 Standards

The studies to be carried out in the project will be in accordance with several different standards. The modified core itself will be in accordance to RISC-V ISA standards. Hardware implementation will be in accordance with IEEE(Institute of Electrical and Electronics Engineers) while cryptographic algorithm NTRU will be in accordance with NIST. Also, it is aimed to modify and optimize the NTRU C implementation in accordance with C programming language standards. Finally, the engineering code of conduct is followed throughout the project.

6.2.4 Health and safety concerns

Since FPGA itself is sort of a black box mentality, there is no actual danger to any human. Also, by the nature of this project the design product is not risky by any means.

6.3 Future Work and Recommendations

The aim for a future project would be to implement a simple communication protocol that utilizes a lattice-based cryptographic algorithm. Then testing it with two modified cores and analyzing the performance improvements over the unmodified ones. Also, another project about porting GCC for any custom instruction would benefit the designing team greatly. First, the pure software part of the project would not need any modification unlike the current version of the project that includes inline assembly directives. Furthermore, it would enable assembly level optimizations in the code. Because of the inline assembly method and the complexity of the NTRU algorithm, low level optimizations are difficult to implement manually. Porting the compiler would help in that case enormously.

REFERENCES

Alagic G., Alperin-Sheriff J., Apon D., Cooper D., Dang Q., Liu Y., Miller C., Moody D., Peralta R., Perlner R., Robinson A. and Smith-Tone D., "Status Report on the First Round of the NIST Post-quantum Cryptography Standardization Process," National Institute of Standards and Technology, Tech. Rep. 8240, January 2019.

Buktu T., Gueron S., S.F.: libntru github Repository, https://github.com/tbuktu/libntru

CMake, "Cross Platform Make," https://cmake.org/cmake/help/v3.3/index.html.

Furber S. B., *VLSI RISC Architecture and Organization*. Routledge, 19 Sep 2017.

Hoffstein J., Pipher J., and Silverman J. H., "NTRU: A Ring-based Public Key Cryptosystem," *in International Algorithmic Number Theory Symposium.*Springer, 1998, pp. 267–288.

Ibex Documentation, lowRISC, April 22 2020, https://ibexcore.readthedocs.io/ /downloads/en/latest/pdf/.

Integrated Logic Analyzer v6.2, Xilinx, Inc., October 5 2016, https://www.xilinx.com/support/documentation/ip_documentation/ila/ v6_2/pg172-ila.pdf.

Kanoun K. and Spainhower L., *Dependability Benchmarking for Computer Systems*, January 7 2008, doi:10.1002/9780470370506.

Mohseni M., Read P., Neven H., Boixo S., Denchev V., Babbush R., Fowler A., Smelyanskiy V. and Martinis J., "Commercialize Quantum Technologies in Five Years," Nature, vol. 543, no. 7644, pp. 171–174, 2017.

Nexys4 DDR FPGA Board Reference Manual, Digilent, Inc., April 11 2016, https://reference.digilentinc.com/ media/reference/programmablelogic/ nexys4ddr/nexys4ddr rm.pdf.

Paar C., "Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems," Presented at the 3rd workshop on Elliptic Curve Cryptography (ECC 1999), November 1-3 1999, http://www.cacr.math.uwaterloo.ca/conferences/1999/ecc99/slides.ht ml.

Patterson D. A. and Hennessy J. L., *Computer Organization and Design RISCV Edition,* 1st ed. Morgan Kaufmann, 12 May 2017.

RISC-V, "GNU COMPILER TOOLCHAIN" https://github.com/riscv/riscv-gnu-toolchain.

Rivest R. L., Shamir A. and Adleman L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978.

Sako K., Public Key Cryptography. Boston, MA: Springer US, 2011, pp. 996–997. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5 22.

SystemVerilog, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language - Redline*," IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline*, pp. 1–1346, 2009.

Ubuntu, "Ubuntu 16.04 Documentation," https://help.ubuntu.com/16.04/ubuntu-help/index.html.

Vivado Design Suite User Guide, Xilinx, Inc., April 4 2018, https://www.xilinx.com/support/documentation/sw manuals/xilinx2018 1/ug910-vivado-getting-started.pdf.

Waterman A., Lee Y., Patterson D. and Asanovic K., "The RISC-V Instruction Set Manual," 2016.

Yanofsky N. S. and Mannucci M. A., *Quantum Computing for Computer Scientists,* 1st ed. Cambridge University Press, 11 Aug 2008.

Yusmardiah Y., Mohd D., Karimi A., Abdul A. and Kamsani A., "Translation of Division Algorithm Into Verilog HDL," ARPN Journal of Engineering and Applied Sciences, vol. 12, pp. 3214–3217, 05 2017.

Url-1 <*https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html*.>

[1] **Yanofsky N. S. and Mannucci M. A.,** *Quantum Computing for Computer Scientists,* 1st ed. Cambridge University Press, 11 Aug 2008.

[2] **Mohseni M., Read P., Neven H., Boixo S., Denchev V., Babbush R., Fowler A., Smelyanskiy V. and Martinis J.,** "Commercialize Quantum Technologies in Five Years," Nature, vol. 543, no. 7644, pp. 171–174, 2017.

[3] **Hoffstein J., Pipher J., and Silverman J. H.,** "NTRU: A Ring-based Public Key Cryptosystem," *in International Algorithmic Number Theory Symposium.*Springer, 1998, pp. 267–288.

[4] **Alagic G., Alperin-Sheriff J., Apon D., Cooper D., Dang Q., Liu Y., Miller C., Moody D., Peralta R., Perlner R., Robinson A. and Smith-Tone D.,** "Status Report on the First Round of the NIST Post-quantum Cryptography Standardization Process," National Institute of Standards and Technology, Tech. Rep. 8240, January 2019.

[5] **Furber S. B.**, *VLSI RISC Architecture and Organization*. Routledge, 19 Sep 2017.

[6] **Patterson D. A. and Hennessy J. L.,** *Computer Organization and Design RISCV Edition,* 1st ed. Morgan Kaufmann, 12 May 2017.

[7] **Rivest R. L., Shamir A. and Adleman L.,** "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978.

[8] **Paar C.,** "Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems," Presented at the 3rd workshop on Elliptic Curve Cryptography (ECC 1999), November 1-3 1999, http://www.cacr.math.uwaterloo.ca/conferences/1999/ecc99/slides.html.

[9**] Sako K.,** Public Key Cryptography. Boston, MA: Springer US, 2011, pp. 996–997. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5 22

[10] *Nexys4 DDR FPGA Board Reference Manual,* Digilent, Inc., April 11 2016, https://reference.digilentinc.com/ media/reference/programmablelogic/nexys4ddr/nexys4ddr rm.pdf.

[11]*Vivado Design Suite User Guide*, Xilinx, Inc., April 4 2018, https://www.xilinx.com/support/documentation/sw manuals/xilinx2018 1/ug910-vivado-getting-started.pdf.

[12] Ubuntu, "Ubuntu 16.04 Documentation," https://help.ubuntu.com/16.04/ubuntu-help/index.html.

[13] CMake, "Cross Platform Make," https://cmake.org/cmake/help/v3.3/index.html.

[14] RISC-V, "GNU COMPILER TOOLCHAIN" https://github.com/riscv/riscv-gnu-toolchain.

[15] Ibex Documentation, lowRISC, April 22 2020, https://ibexcore.readthedocs.io/ /downloads/en/latest/pdf/.

[16] SystemVerilog, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language - Redline*," IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline*, pp. 1–1346, 2009.

[17] **Buktu T., Gueron S., S.F.**: libntru github Repository, <https://github.com/tbuktu/libntru>

[18] **Waterman A., Lee Y., Patterson D. and Asanovic K.**, “The RISC-V Instruction Set Manual,” 2016.

[19]**Url-1**<*https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html*.>

[20] **Yusmardiah Y., Mohd D., Karimi A., Abdul A. and Kamsani A.**, “Translation of Division Algorithm Into Verilog HDL,” ARPN Journal of Engineering and Applied Sciences, vol. 12, pp. 3214–3217, 05 2017.

[21] **Kanoun K. and Spainhower L.**, *Dependability Benchmarking for Computer Systems*, January 7 2008, doi:10.1002/9780470370506.

[22]*Integrated Logic Analyzer v6.2,* Xilinx, Inc., October 5 2016, https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.

APPENDICES

APPENDIX A: The Example makefile Script
APPENDIX B: C Code of the NTRU Algorithm
APPENDIX C: SystemVerilog Codes of the Custom Module and Remainder Module
APPENDIX D: RTL Schematics

APPENDIX A

```
1   # Copyright lowRISC contributors.
2   # Licensed under the Apache License, Version 2.0, see LICENSE for details
3   # SPDX-License-Identifier: Apache-2.0
4   #
5   # Generates a baremetal application
6
7   PROGRAM ?= 2d_array
8   PROGRAM_CFLAGS = -Wall -g -Os
9   ARCH = rv32imc
10
11  SRCS = $(PROGRAM).c
12
13  CC = /opt/riscv/bin/riscv32-unknown-elf-gcc
14
15  OBJCOPY ?= $(subst gcc,objcopy,$(wordlist 1,1,$(CC)))
16  OBJDUMP ?= $(subst gcc,objdump,$(wordlist 1,1,$(CC)))
17
18  LINKER_SCRIPT ?= link.ld
19  CRT ?= crt0.S
20  CFLAGS ?= -march=$(ARCH) -mabi=ilp32 -static -mcmodel=medany \
21      -fvisibility=hidden -nostdlib -nostartfiles $(PROGRAM_CFLAGS)
22
23  OBJS := ${SRCS:.c=.o} ${CRT:.S=.o}
24  DEPS = $(OBJS:%.o=%.d)
25
26  OUTFILES = $(PROGRAM).elf $(PROGRAM).vmem $(PROGRAM).bin $(PROGRAM).dis
27
28  all: $(OUTFILES)
29
30  $(PROGRAM).elf: $(OBJS) $(LINKER_SCRIPT)
31      $(CC) $(CFLAGS) -T $(LINKER_SCRIPT) $(OBJS) -o $@ $(LIBS)
32
33  %.dis: %.elf
34      $(OBJDUMP) -SD $^ > $@
35
36  # Note: this target requires the srecord package to be installed.
37  # XXX: This could be replaced by objcopy once
38  # https://sourceware.org/bugzilla/show_bug.cgi?id=19921
39  # is widely available.
40  # XXX: Currently the start address 0x00000000 is hardcoded. It could/should be
41  # read from the elf file, but is lost in the bin file.
42  # Switching to objcopy will resolve that as well.
43  %.vmem: %.bin
44      srec_cat $^ -binary -offset 0x0000 -byte-swap 4 -o $@ -vmem
45
46  %.bin: %.elf
47      $(OBJCOPY) -O binary $^ $@
48
49  %.o: %.c
50      $(CC) $(CFLAGS) -MMD -c $(INCS) -o $@ $<
51
52  %.o: %.S
53      $(CC) $(CFLAGS) -MMD -c $(INCS) -o $@ $<
54
55  clean:
56      $(RM) -f *.o *.d
57
58  distclean: clean
59      $(RM) -f $(OUTFILES)
```

Figure A.1: The Example makefile Script

APPENDIX B

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   static int product[150];
5   static int product2[150];
6   static int result[150];
7   static int ti_2[100];
8   static int random_keys[318];
9   volatile int resultkon[1]={0XBEBEBEBE};
10
11  ////////////////CUSTOM INSTRUCTION SET///////////////
12
13  void instr_add(unsigned int *a1, unsigned int *a2){
14
15      asm volatile(
16      ".insn r CUSTOM_0, 0x7, 3, %0, %1, %2  \n"
17      : "=r"(&a1[0])
18      : "r"(&a1[0]), "r"(&a2[0])
19
20      );
21      return;
22  }
23
24  void array_add(int *a1, int *a2, int length) {
25  int i =0;
26      switch(length%3) {
27
28          case 0:
29              for (i = 0; i < (length / 3); i++) {
30                  instr_add((unsigned int*)&a1[3 * i], (unsigned int*)&a2[3*i]);
31              }
32              break;
33          case 1:
34              for (i = 0; i < ((length-1) / 3); i++) {
35                  instr_add((unsigned int*)&a1[3 * i], (unsigned int*)&a2[3*i]);
36              }
37              a1[length-1] = a1[length-1] + a2[length-1];
38              break;

39          case 2:
40              for (i = 0; i < ((length-2) / 3); i++) {
41                  instr_add((unsigned int*)&a1[3 * i], (unsigned int*)&a2[3*i]);
42              }
43              a1[length-1] = a1[length-1] + a2[length-1];
44              a1[length-2] = a1[length-2] + a2[length-2];
45              break;
46      } //end of switch case
47  } //end of function
48
49
50  void instr_equ(unsigned int *a1, unsigned int *a2){
51
52      asm volatile(
53      ".insn r CUSTOM_0, 0x7, 5, %0, %1, %2  \n"
54      : "=r"(&a1[0])
55      : "r"(&a1[0]), "r"(&a2[0])
56
57      );
58      return;
59  }
60
61
62  void array_equ(int *a1,int *a2,int length) {
63  int i = 0;
64      switch(length%3) {
65
66          case 0:
67              for (i = 0; i < (length / 3); i++) {
68                  instr_equ((unsigned int*)&a1[3 * i],(unsigned int*) &a2[3*i]);
69              }
70              break;
71          case 1:
72              for (i = 0; i < ((length-1) / 3); i++) {
73                  instr_equ((unsigned int*)&a1[3 * i],(unsigned int*) &a2[3*i]);
74              }
75              a1[length-1] = a2[length-1];
76              break;
```

```
77          case 2:
78              for (i = 0; i < ((length-2) / 3); i++) {
79                  instr_equ((unsigned int*)&a1[3 * i],(unsigned int*) &a2[3*i]);
80              }
81              a1[length-1] = a2[length-1];
82              a1[length-2] = a2[length-2];
83              break;
84      } //end of switch case
85  } //end of function
86
87
88
89  void instr_mod(unsigned int *a1, unsigned int mod) {
90
91      asm volatile(
92      ".insn r CUSTOM_0, 0x7, 6, %0, %1, %2  \n"
93      : "=r"(&a1[0])
94      : "r"(&a1[0]), "r"(mod)
95
96      );
97      return;
98  }
99  void array_mod(int *a1, int mod,int length){
100         int i = 0;
101
102         switch(length%3) {
103
104         case 0:
105             for (i = 0; i < (length / 3); i++) {
106                 instr_mod((unsigned int *) &a1[3 * i], (unsigned int) mod);
107             }
108             break;
109         case 1:
110             for (i = 0; i < ((length-1) / 3); i++) {
111                 instr_mod((unsigned int *) &a1[3 * i], (unsigned int) mod);
112             }
113             a1[length-1] = a1[length-1] % mod;
114             break;

115         case 2:
116             for (i = 0; i < ((length-2) / 3); i++) {
117                 instr_mod((unsigned int *) &a1[3 * i], (unsigned int) mod);
118             }
119             a1[length-1] = a1[length-1] % mod;
120             a1[length-2] = a1[length-2] % mod;
121             break;
122         } //end of switch case
123 } //end of function
124
125
126 ////////////////////////////////////////////////
127
128
129
130 int *polymult(int *a, int size_a, int *b, int size_b, int mod, int star_mult){
131
132     int line[size_b][size_a + size_b];
133     int i,j,k;
134     int *return_address1;
135     int pola[size_a];
136     int polb[size_b];
137
138     //make all line vectors zero
139     for (i = 0; i < size_b; ++i){
140         for(j = 0; j < size_a + size_b -1; ++j){
141             line[i][j] = 0;
142             product[j] = 0;
143         }
144     }
145
146     // make mod calculations
147     array_equ(pola,a,size_a);
148     array_mod(pola,mod,size_a);
149
150     array_equ(polb,b,size_b);
151     array_mod(polb,mod,size_b);
152
153
```

```
154     // make calculations for partial products, if need add mod calculations
155     if(star_mult == 1){
156         for (j = size_b -1; j >= 0; j = j-1){
157             for (k = size_a-1; k >= 0; k = k-1 ){
158                 line[size_b-1-j][k] = pola[k]*polb[j];
159             }
160         }
161     }
162     else{
163         for (j = size_b -1; j >= 0; j = j-1){
164             for (k = size_a-1; k >= 0; k = k-1 ){
165                 line[size_b-1-j][k + j] = pola[k]*polb[j];
166             }
167         }
168     }
169 
170     //mod calculations
171     for (i = 0; i < size_b; ++i){
172         array_mod(line[i],mod,size_a+size_b-1);
173     }
174 
175 
176     // construct product
177     for(i = 0; i < size_b; ++i){
178         array_add(product,line[i],size_a+size_b-1);
179     }
180 
181     //mod calculations
182     array_mod(product,mod,(size_a+size_b-1));
183 
184     return_address1 = &product[0];
185 
186     return return_address1;
187 }
188 
189 
```

```
190 int *polymult2(int *a, int size_a, int *b, int size_b, int star_mult){
191 
192     int line[size_b][size_a + size_b];
193     int i,j,k;
194     int *return_address2;
195     int pola[size_a];
196     int polb[size_b];
197 
198     //make all line vectors zero
199     for (i = 0; i < size_b; ++i){
200         for(j = 0; j < size_a + size_b -1; ++j){
201             line[i][j] = 0;
202             product2[j] = 0;
203         }
204     }
205 
206     array_equ(pola,a,size_a);
207 
208     array_equ(polb,b,size_b);
209 
210     // make calculations for partial products, if need add mod calculations
211     if(star_mult == 1){
212         for (j = size_b -1; j >= 0; j = j-1){
213             for (k = size_a-1; k >= 0; k = k-1 ){
214                 line[size_b-1-j][k] = pola[k]*polb[j];
215             }
216         }
217     }
218     else{
219         for (j = size_b -1; j >= 0; j = j-1){
220             for (k = size_a-1; k >= 0; k = k-1 ){
221                 line[size_b-1-j][k + j] = pola[k]*polb[j];
222             }
223         }
224     }
225 
226     // construct product
227     for(i=0;i<size_b;++i){
228          array_add(product2,line[i],size_a+size_b-1);
229     }
```

```
230
231     return_address2 = &product2[0];
232
233     return return_address2;
234 }
235
236 int *polydiv(int *num, int size_N, int*denum, int size_D, int mod){
237
238     int u,d,d2,i,b_N,r_d;
239     int *return_address3;
240
241     int v[size_N];
242     int q[size_N];
243     int *product;
244     int num_temp[size_N];
245     int denum_temp[size_D];
246
247     array_equ(num_temp,num,size_N);
248
249
250
251     array_mod(num_temp, mod, size_N);
252
253
254
255     // make mod calculation for coefficents
256     array_equ(denum_temp,denum,size_D);
257     array_mod(denum_temp,mod,size_D);
258
259     for (i = 0; i < size_N; ++i){
260         q[i] = 0;
261         v[i] = 0;
262     }
263
264     //find b_N (denum) and degree denum
265     for (i = size_D-1; i >= 0; i = i-1){
266         if( denum_temp[i] != 0 )
267             break;
268     }
269     d2 = i; //degree of f
270     b_N = denum_temp[i];
271
272     // Set u := (b_N)^-1 mod p (denum) //
273     for (u = 0; u < mod; ++u){
274         if ( (b_N*u)%mod == 1 )
275             break;
276     }
277
278     // find degree num and r_d
279     for (i = size_N-1; i >= 0; i = i-1){
280         if( num_temp[i] != 0 )
281             break;
282     }
283     d = i;
284
285     r_d = num_temp[d];
286
287     // While-1 deg num >= deg denum do
288     while (d >= d2){
289
290         // Set v := u × r_d × X^(d-N)
291         v[(d-d2)] = u*r_d;
292
293         array_mod(v, mod, size_N);
294
295         // v*b
296         product = polymult(denum_temp,size_D,v,size_D,mod,0);
297
298         // make mod calculation for coefficents
299         array_mod(product, mod, size_N);
300
301
302         //r = r- v*b
303         for (i = 0; i < size_N; ++i){
304             num_temp[i] = num_temp[i] - product[i];
305         }
306
307
308         // make mod calculation for coefficents
309         array_mod(num_temp, mod, size_N);
310
```

```
311
312         // q = q + v;
313         array_add(q, v, size_N);
314
315         array_mod(q, mod, size_N);
316
317
318         // Set d := deg r(X) (num)
319         for (i = size_N-1; i >= 0; i = i-1){
320             if( num_temp[i] != 0 )
321                 break;
322         }
323         d = i;
324         r_d = num_temp[d];
325
326         // make zero for next calculations
327         for (i = 0; i < size_N; ++i){
328             v[i] = 0;
329         }
330
331
332     } //End While-1
333
334     array_equ(result,q,size_N);
335
336
337
338     for(i = size_N; i < (2*size_N); ++i){
339         result[i] = num_temp[i-size_N];
340     }
341
342     return_address3 = &result[0];
343
344     return return_address3;
345 }
346
347

348 int* ext_euclid(int* polyR, int* polyf, int size, int mod) {
349
350     int *return_address4;
351     int N, i, j;
352     int ri_2[size]; // MX-ring poly, a
353     int ri_1[size]; // f-random poly, f[N] = 0, b
354     int ri[size];
355
356     int ti_1[size]; // ti_1[0] = 1
357     int ti[size];
358     int qi_1[size];
359     int temp[size];
360     int *res;
361     int *res1;
362     int *res2;
363     int controlR = 1;
364
365     N = size - 1;
366     for (i = 0; i < size; ++i) {
367         ti_2[i] = 0;
368         ti_1[i] = 0;
369     }
370     array_equ(ri_2,polyR,size);
371     array_equ(ri_1,polyf,size);
372
373     ti_1[0] = 1;
374
375     while (controlR != 0) {
376
377         controlR = 0;
378         // make mod for ri_2
379         array_mod(ri_2,mod,size);
380
381         // make mod for ri_1
382         array_mod(ri_1,mod,size);
383
384         res = polydiv(ri_2, N + 1, ri_1, N + 1, mod);
385         for (i = N + 1; i < (2 * N + 2); ++i) {
386             ri[i - (N + 1)] = res[i];
387         }
388
```

```
/// qi_1 = (ri_2 - ri)/ri_1; ///

        for (i = 0; i < N + 1; ++i) {
            temp[i] = ri_2[i] - ri[i];
        }


        //make mod calc
        array_mod(temp,mod,size);

        res1 = polydiv(temp, N + 1, ri_1, N + 1, mod);
        array_equ(qi_1,res1,size);

        //make mod calc
        array_mod(qi_1,mod,size);

/// ti = ti_2 - qi_1*ti_1; ///

        res2 = polymult(qi_1, N + 1, ti_1, N + 1, mod, 0);
        array_equ(temp,res2,size);

        // make mod calculations
        array_mod(temp,mod,size);

        for (i = 0; i < N + 1; ++i) {
            ti[i] = ti_2[i] - temp[i];
        }

        array_mod(ti,mod,size);

/// ri_2 = ri_1; ri_1 = ri; ti_2 = ti_1; ti_1 = ti; ///

array_equ(ri_2,ri_1,size);
array_equ(ri_1,ri,size);
array_equ(ti_2,ti_1,size);
array_equ(ti_1,ti,size);

```

```
        for (i = 0; i < N + 1; ++i) {
             qi_1[i] = 0;
            controlR += ri[i];
        }
    }

        for (j = 0; j < mod; ++j) {
            if (((ri_2[0] * j) % mod) == 1)
                break;
        }


        for (i = 1; i < N; ++i) {
            if (ri_2[i] != 0)
                break;
        }

        for (i = 0; i < N + 1; ++i) {
            ti_2[i] = (ti_2[i] * j) % mod;
        }


        return_address4 = &ti_2[0];
        return return_address4;
    }



int* generate_keys(int N, int p, int q){
    resultkon[0] = 0xaaaaaaa0;
    int f[55] = {-1, 1, 1, 0, -1, 0, 1, 0, 0, 1, -1, -1, 0, 1, 0, -1, 0, 1, 1, 0, 0,  -1, 0, 1, 0, 0, 1, 1, 1,  -1, 0, 1, 0, 0, 1, 1, 1,    -1, 0, 1, 0, 0, 1, 1, 1};
    int g[55] = {-1, 0, 1, 1, 0, 1, 0, 0, -1, 0, -1, -1, 0, 1, 0, -1, 0, 1, 0, 0, 1, -1, 0, 1, 0, 0, 1, -1, 0, 1, 0, 0, 1, -1, 0, 1, 0, 1,   1, 0, 0, 1, -1, 0, 1, 0, 1};

    int *fp;
    int *fq;
    int *fg;
    int *pk;
    int polyR[N+1];
    int i;
    int* return_address5;
```

```
467
468     polyR[N] = 1;
469     polyR[0] = -1;
470
471     for(i = 1; i < N; ++i){
472         polyR[i] = 0;
473     }
474
475
476
477     for (i = N; i < 55; ++i ){
478         f[i] = 0;
479         g[i] = 0;
480     }
481
482     for(i = 0; i < N; ++i){
483         random_keys[i] = f[i];
484     }
485
486     for(i = N; i < (2*N); ++i){
487         random_keys[i] = g[i-N];
488     }
489
490     fp = ext_euclid(polyR, f, (N+1), p);
491
492     for(i = (2*N); i < (3*N); ++i){
493         random_keys[i] = fp[i-(2*N)];
494     }
495
496     fq = ext_euclid(polyR, f, (N+1), q);
497
498     for(i = (3*N); i < (4*N); ++i){
499         random_keys[i] = fq[i-(3*N)];
500     }
501
502     fg = polymult2(fq, N, g, N, 0);
503
504     for(i = 0; i < (2*N-1); ++i){
505         fg[i] = fg[i]*p;
506     }
507
508     pk = polydiv(fg, (2*N-1), polyR, (N+1), q);
509
510     for(i = (4*N); i < (5*N); ++i){
511         random_keys[i] = pk[i-(4*N)+(2*N-1)];
512     }
513
514     for(i = (5*N); i < (6*N+1); ++i){
515         random_keys[i] = polyR[i-(5*N)];
516     }
517
518     return_address5 = &random_keys[0];
519     resultkon[0] = 0xaaaaaaa1;
520     return return_address5;
521 }
522
523
524 int* ntru_encyrpt(int N, int q, int* message, int* public_key, int* polyR){
525     resultkon[0] = 0xbbbbbbb0;
526     int *return_address6;
527     int *CT;
528
529     int random_val[48] = { 1, -1, 0, 1, -1, 1, -1, 0,1, 1, 0, -1, 1, -1, 1, 0,1, 1, 0, -1, 1, -1, 1, 0,1, 1, 0, 1, -1, 1, -1, 0, 1, -1, 0, 1, -1, 1, -1, 0,1, 1, 0, -1, 1, -1,
530     int *temp;
531     int i;
532
533
534     temp = polymult(public_key,N,random_val,48,q,0);
535
536     /////// PUBLIC_KEY*RANDOM_VALUE ///////
537
538     array_add(temp,message,48);
539     ////// PUBLIC_KEY*RANDOM_VALUE + MESSAGE //////
540
541     CT = polydiv(temp,(2*N),polyR,(N+1),q);
542
543     for(i = (2*N); i < (3*N); ++i){
544         CT[i-(2*N)] = CT[i];
545     }
```

```
546
547     return_address6 = &CT[0];
548     resultkon[0] = 0xbbbbbbb1;
549     return return_address6;
550 }
551
552
553 int* ntru_deceypt(int N, int p, int q, int* secret_key_f, int* secret_key_fp, int* Enc_Message, int* polyR){
554     resultkon[0] = 0xccccccc0;
555     int* return_address7;
556     int* a;
557     int* a2;
558     int* c;
559     int* c2;
560     int i;
561
562     // a = f.e mod q
563     a = polymult2(secret_key_f,N,Enc_Message,N,0);
564
565     a2 = polydiv(a,(2*N),polyR,(N+1),q);
566 /////////// Vector a2 (f.e modq) /////////
567
568     for(i = (2*N); i < ((2*N)+N); ++i){
569         a2[i-(2*N)] = a2[i];
570     }
571
572     //centerlifting a2
573     for (i = 0; i < N; ++i){
574         if(a2[i] <= q/2)
575             a2[i] = a2[i];
576         else
577             a2[i] = (-1)*(q-a2[i]);
578     }
579
580     c = polymult2(secret_key_fp,N,a2,N,0);
581 /////////"Vector c(fp*a2)//////////
582
583     c2 = polydiv(c,(2*N),polyR,(N+1),p);
584     for(i = (2*N); i < ((2*N)+N); ++i){
585         c2[i-(2*N)] = c2[i];
586     }

587     ////////"Vector c2 (decrypted message)/////////
588
589     return_address7 = &c2[0];
590     resultkon[0] = 0xccccccc1;
591     return return_address7;
592 }
593
594
595 int main(){
596     int* Enc_Message;
597     int* Dec_Message;
598     int* keys;
599     int N = 53;
600     int p = 3;
601     int q = 101;
602
603     int message[48] = {1, 1, 2, 1, 1, 1, 1, 2,1, 1, 0, 1, 1, 2, 1, 0,1, 2, 0, 1, 1, 1, 1, 0,1, 1, 0, 2, 1, 2, 1, 0, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 2};
604     int public_key[N];
605     int secret_key_f[N];
606     int secret_key_g[N];
607     int secret_key_fp[N];
608     int secret_key_fq[N];
609     int ring_poly[N+1];
610     int i;
611
612     // Alice generates public key from her randomly created secret keys.
613
614     keys = generate_keys(N, p, q);
615
616     for(i = 0; i < N; ++i){
617         secret_key_f[i] = keys[i];
618     }
619
620     for(i = N; i < (2*N); ++i){
621         secret_key_g[i-N] = keys[i];
622     }
623
624     for(i = (2*N); i < (3*N); ++i){
625         secret_key_fp[i-(2*N)] = keys[i];
626     }
```

```c
627
628     for(i = (3*N); i < (4*N); ++i){
629         secret_key_fq[i-(3*N)] = keys[i];
630     }
631
632     for(i = (4*N); i < (5*N); ++i){
633         public_key[i-(4*N)] = keys[i];
634     }
635
636     for(i = (5*N); i < (6*N+1); ++i){
637         ring_poly[i-(5*N)] = keys[i];
638     }
639
640     // Bob encrypts message using Alice's public key and sends it to Alice.
641
642     Enc_Message = ntru_encyrpt(N, q, message, public_key, ring_poly);
643
644     // Alice decrypts the message using her secret keys.
645
646     Dec_Message = ntru_deceypt(N, p, q, secret_key_f, secret_key_fp, Enc_Message, ring_poly);
647
648
649     // Comparison and Proof
650     for(i = 0; i < 32; ++i){
651         if(!(message[i] == Dec_Message[i])){
652             resultkon[0] = 0xBABABABA;
653
654             break;
655         }
656         else{
657             resultkon[0] = 0xDEDEDEDE;
658         }
659     }
660
661
662     return 0;
663 }
```

Figure B.1: C Code of the NTRU Algorithm

APPENDIX C

```
1   `timescale 1ns / 1ps
2
3
4   module custom_module(
5       input logic custom_en,
6       input logic [31:0] array1_addr,     //operand_a
7       input logic [31:0] array2_addr,     //operand_b
8       output logic [31:0] custom_result,  //adress of the result array
9       output logic [31:0] custom_data,    //ram'in data girisine bagli olan sey bu
10      input logic clk,
11      input logic [31:0] ram_data_in,     //data comes from RAM
12      output logic [13:0] ram_addr_out,   //data send to RAM
13      output logic custom_valid,
14      output logic custom_final,
15      input logic [4:0] custom_op,                // instruction operands
16      output logic custom_mod_o,                  //enable signal remainder module
17      output logic [31:0] custom_op_a_o [2:0],    //datas that send to the remainder module
18      output logic [31:0] custom_op_b_o,          //mod value send to remainder module
19      input logic [31:0] custom_mod_result [2:0], //results taken from remainder module
20      input logic mod_valid                       //valid signal for remainder module
21      );
22
23      logic [2:0] addr_check = 0,array_length = 3;
24
25      reg [1:0] k,i=3;
26      reg [4:0] custom_op_temp;
27      reg [13:0] array1_addr_temp;
28      reg [13:0] array2_addr_temp;
29      reg [31:0] data_reg1 [2:0];
30      reg [31:0] data_reg2 [2:0];
31      reg [31:0] data_reg3 [2:0];
32
33      logic c;
34      logic [3:0] state=0;
35      logic [31:0] mod;
36      logic custom_mod;
37
38      assign custom_mod_o = custom_mod;
39      assign c = (i==array_length-1) ? 1 : 0;
40
41      always@(posedge clk)
42
43      begin case (state)
44
45      1 :begin  //Adres Yolla
46          case(custom_op_temp)
47          4'b0110 : begin //mod
48              if(addr_check[0]) begin
49                  for(int k=0;k<array_length;k++) begin
50                      if(data_reg1[k][31]) begin  //check for negativity
51                         data_reg1[k] +=  mod <<5 ;
52                      end
53
54                  state = 5;
55                  end
56                end
57                else begin
58                  ram_addr_out = ((array1_addr_temp)+i);
59                  state = state+1;
60                end
61          end
62
63          4'b0101 : begin //equ
64              if (addr_check[1]) begin
65                  ram_addr_out = ((array1_addr_temp)+i);
66                  state = state+1;
67              end
68              else begin
69                  addr_check <= 1;
70                  ram_addr_out = ((array2_addr_temp)+i);
71                  state = state+1;
72              end
73          end
74
75          4'b0011 : begin //add
76              ram_addr_out = addr_check[0] ? ((array2_addr_temp)+i) : ((array1_addr_temp)+i) ;
77              state = state+1;
78          end
79
80          endcase
81      end
83
84      2 :begin  //Wait_1
85          if (!i) begin
86          state =state+1;
87          end
88          else begin
89          state = state+2;
90          end
91      end
92
93      3 :begin  //Wait_2
94          state = state+1;
95      end
96
97
```

```
98      4 :begin  //Input data
99          i = c ? 0 : i+1;
100
101         case(custom_op_temp)
102         4'b0101 : begin //equ
103             if(!addr_check) begin
104                 data_reg1[i] = ram_data_in;
105                 addr_check = c ? 1 : 0;
106                 state = 1;
107             end
108             else if (addr_check[0]) begin
109                 data_reg3[i] = ram_data_in;
110                 addr_check = c ? 2 : 1;
111                 state = 1;
112             end
113             else if (addr_check[1]) begin
114                 custom_valid = 1;
115                 custom_data  = data_reg3[i];
116                 state = c ? state+2 : 1;
117             end
118         end    //end of equ
119
120         default: begin
121             if(!addr_check) begin
122                 data_reg1[i] = ram_data_in;
123                 addr_check = c ? 1 : 0;
124                 state = 1;
125             end
126             else if (addr_check[0]) begin
127               data_reg2[i] = ram_data_in;
128                 state = c ? state+1 : 1;
129
130             end
131             else if (addr_check[1]) begin
132                 custom_valid = 1;
133                 custom_data  = data_reg3[i];
134                 state = c ? state+2 : 1;
135             end //elseif end
136         end //default end
137         endcase //case end
138     end //state end
139
140     5 :begin    //Calculate
141        case(custom_op_temp)
142
143         4'b0011: begin   //add
144             for(int j=0;j<array_length;j++) begin
145                 data_reg3[j] = data_reg1[j]+data_reg2[j];
146             end
147             state = 1;
148         end
149
150         4'b0110 : begin  //mod
151             custom_mod = 1'b1;
152             custom_op_b_o = mod;
153             custom_op_a_o = data_reg1;
154
155             if (mod_valid) begin
156                 data_reg3 = custom_mod_result;
157                 custom_mod = 0;
158                 state = 1;
159             end
160             else begin
161                 state = 5;
162             end
163         end
164         endcase   //end of instruction cases
165
166         addr_check = 2;
167     end //end of state5
168
169     6  :begin  // Finalize_1
170         custom_final = 1;
171         custom_result = array1_addr_temp<<2;
172         i = 0;
173         custom_valid = 0;
174         state = state +1;
175     end
176
177      7  :begin  //Finalize_2
178         for(int i=0;i<array_length;i++) begin
179         data_reg3[i] = 32'b0;
180         data_reg1[i] = 32'b0;
181         data_reg2[i] = 32'b0;
182         end
183         state = 0;
184      end
185
186
187      default: begin
188          array1_addr_temp = array1_addr>>2;
189          array2_addr_temp = array2_addr>>2;
190          mod = array2_addr;
191          custom_op_temp = custom_op;
192
193          custom_op_a_o[0] = 32'b0;
194          custom_op_a_o[1] = 32'b0;
195          custom_op_a_o[2] = 32'b0;
196          custom_op_b_o = 32'b0;
197          custom_mod = 1'b0;
198
199          ram_addr_out = 14'b0;
200          custom_valid = 1'b0;
201          custom_data  = 32'b0;
202          custom_final = 0;
203
204          i = 0;
205          addr_check = 0;
206          custom_result = 32'b0;
207          custom_data = 32'b0;
208
209          if(custom_en)   state = 1;
210      end
211      endcase
212      end
213 endmodule
```

Figure C.1: SystemVerilog Code of the Custom Module

```
1  `timescale 1ns / 1ps
2
3
4  module remainder(clk,enable,dividend,divisor,result,valid);
5
6  input clk;
7  input enable;
8  input [31:0] dividend,divisor;
9  output [31:0] result;
10 output valid;
11
12 logic [63:0] dividend_copy,divisor_copy,diff;
13 logic [6:0]  bits = 64;
14 logic valid=0;
15
16 always@(posedge clk)   begin
17     if (enable) begin
18         if(bits[6])   begin
19             bits = 32;
20             dividend_copy = {32'd0,dividend};
21             divisor_copy = {1'd0,divisor,31'd0};
22         end
23         else   begin
24             valid = (~bits[6] && ~bits[5] && ~bits[4] && ~bits[3] && ~bits[2] && ~bits[1] && bits[0]) && enable;
25             diff = dividend_copy - divisor_copy;
26             if( !diff[63] )   begin
27                     dividend_copy = diff;
28             end
29
30             divisor_copy = divisor_copy >> 1;
31             bits = (!bits) ? 64 :  bits - 1;
32         end
33     end
34
35 end
36
37 assign result = dividend_copy [31:0];
38 endmodule
```

Figure C.2: SystemVerilog Code of the Remainder Module

APPENDIX D

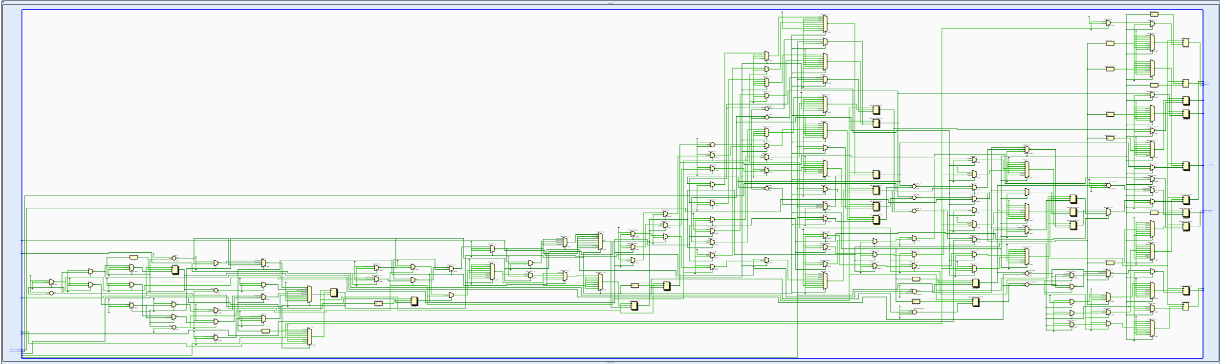


Figure D.1: RTL Schematic of Custom Module

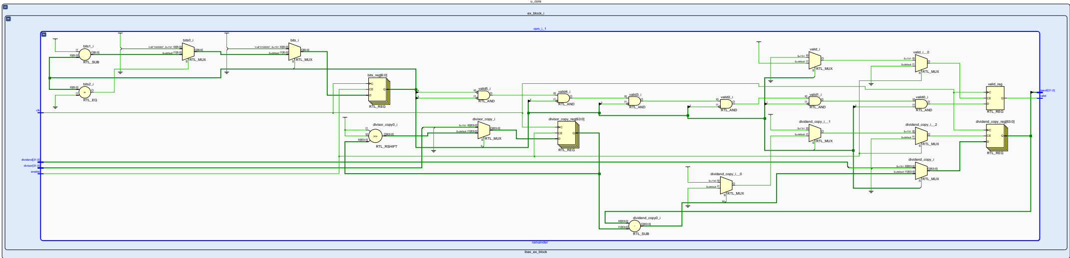


Figure D.2: RTL Schematic of Remainder Module

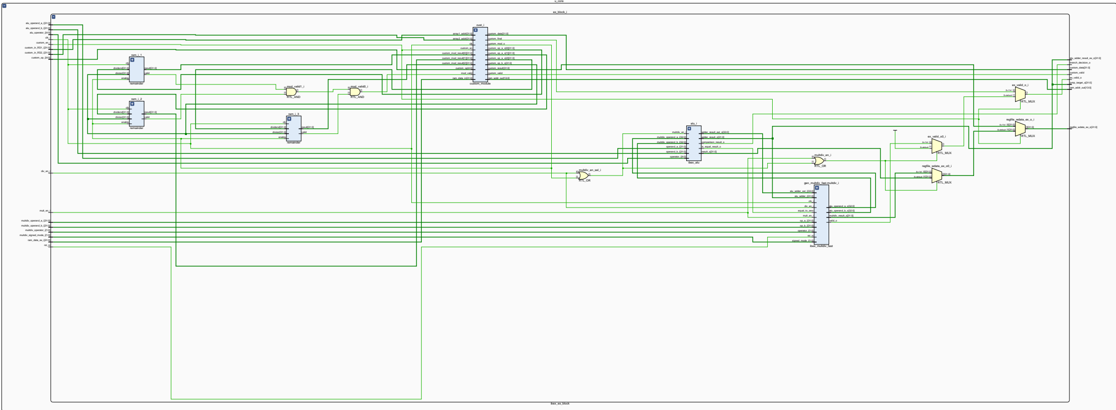


Figure D.3: RTL Schematic of Execution Block

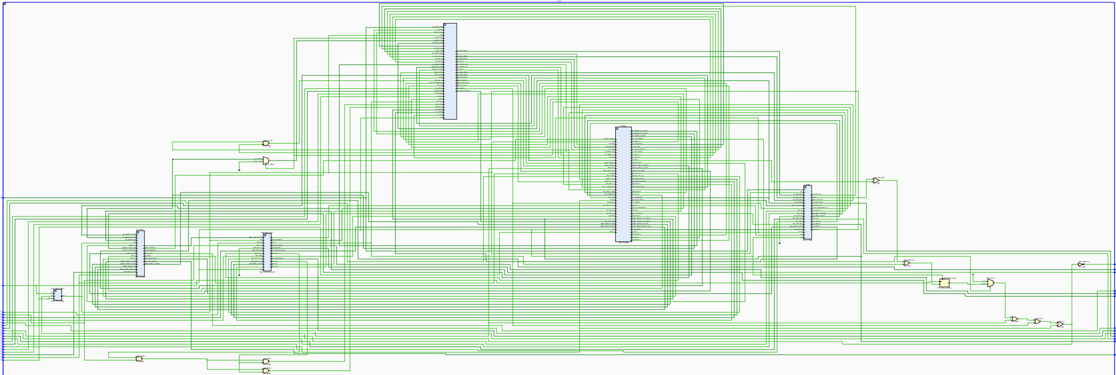


Figure D.4: RTL Schematic of Ibex Core

Figure D.5: RTL Schematic of Top Module

CURRICULUM VITAE

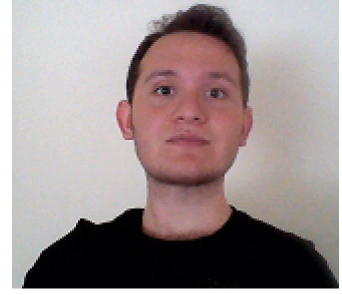
Name Surname : Elif Nur İşman

Place and Date of Birth : İstanbul / 10.11.1997

E-Mail : elifnurisman@hotmail.com

Elif Nur İşman finished primary and high school in İstanbul. She is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University Electrical-Electronics Faculty. She completed her internships in İTÜ GSTL Laboratory, Türk Telekom A.Ş. and ASELSAN A.Ş., also completed a voluntary internship in TÜBİTAK.

CURRICULUM VITAE



Name Surname : Canberk Topal

Place and Date of Birth : Bafra / 25.10.1998

E-Mail : topalc16@itu.edu.tr

Canberk Topal is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University Electrical-Electronics Faculty. He completed her internships in ASELSAN A.Ş. and HAVELSAN A.Ş. His primary areas of interest include digital system design, cryptography and machine learning.