ISTANBUL TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS FACULTY

INSTRUCTION SET EXTENSION FOR POST QUANTUM CRYPTOGRAPHY ALGORITHMS ON RISC-V CORES

SENIOR DESIGN PROJECT

Ali ÜSTÜN
Batuhan ATEŞ
Musa ANTİKE

ELECTRONICS AND COMMUNICATION ENGINEERING

DEPARTMENT

JULY 2020

<u>ISTANBUL TECHNICAL UNIVERSITY</u>

<u>ELECTRICAL-ELECTRONICS FACULTY</u>

INSTRUCTION SET EXTENSION FOR POST QUANTUM CRYPTOGRAPHY ALGORITHMS ON RISC-V CORES

SENIOR DESIGN PROJECT

Ali ÜSTÜN
(040160225)
Batuhan ATEŞ
(040160025)
Musa ANTİKE
(040160096)

ELECTRONICS AND COMMUNICATION ENGINEERING

DEPARTMENT

Project Advisor: Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın

JULY 2020

We are submitting the Senior Design Project entitled as “INSTRUCTION SET EXTENSION FOR POST QUANTUM CRYPTOGRAPHY ALGORITHMS ON RISC-V CORES”. The Senior Design Project 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 by ourselves, and we have abided by the ethical rules with respect to academic and professional integrity .

Ali ÜSTÜN ....
(040160225)

Batuhan ATEŞ ....
(040160025)

Musa ANTİKE ....
(040160096)

FOREWORD

First of all, we would like to thank our project advisor, Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın for guiding us through her advises and knowledge as well as sharing lots of her experiences with us.

Secondly, we would like to thank our Res. Assist. M.Sc. Latif Akçay for sharing his precious time and insights.

Also, we would like to thank all our friends for their support. Their company has always kept us entertained and motivated in times of struggle.

Finally, we want to express our endless gratitude and appreciation for our families, who have supported our decisions and guided us with their experiences.

JULY 2020

Ali ÜSTÜN
Batuhan ATEŞ
Musa ANTİKE

TABLE OF CONTENTS

Page

ABBREVIATIONS

PULP	: Parallel Ultra Low Power
RISC	: Reduced Instruction Set Computer
UART	: Universal Asynchronous Receiver Transmitter
GPIO	: General Purpose Input Output
ROM	: Read-Only Memory
NTRU	: Nth Degree Truncated Polynomial Ring Unit
IIS	: Integrated Systems Laboratory
EEES	: Energy-Efficient Embedded Systems Laboratory
NIST	: National Institute of Standards and Technology
ISA	: Instruction Set Architecture
GPR	: General Purpose Registers
FPR	: Floating Point Registers
PCR	: Privileged Control Registers
IoT	: Internet of Things
FPGA	: Field-Programmable Gate Array
PQC	: Post-Quantum Cryptography
NTT	: Number Theoretic Transformation
LWE	: Learning With Error
RLWE	: Ring Learning With Error
SVP	: Shortest Vector Problem
CVP	: Closest Vector Problem
RISC	: Reduced Instruction Set Computer
SoC	: System On a Chip
OS	: Operating System
SPI	: Serial Peripheral Interface
JTAG	: Joint Test Action Group
I2S	: Inter-IC Sound
DMA	: Direct Memory Access
VHDL	: Very High Speed Integrated Circuit Hardware Description Language

LIST OF TABLES

LIST OF FIGURES

INSTRUCTION SET EXTENSION FOR POST QUANTUM CRYPTOGRAPHY ALGORITHMS ON RISC-V CORES

SUMMARY

With the development of quantum computers, security gets more and more important. The commonly used cryptography algorithms such as RSA are less secure against quantum computers. Because of that, in the near future, cryptography algorithms resistant against quantum computers will be needed. With the aim of standardizing good post-quantum cryptography algorithms, NIST has started a project. Between the submissions, NewHope is one of the promising post-quantum cryptography algorithms. So we have decided to use NewHope algorithm in this project. Since post-quantum cryptography algorithms contain complex mathematical operations, they tend to be slow. With the rise of IoT, more data is getting digitalized which makes security even more important. Because of that, even small devices with RISC architectures may be required to use post-quantum cryptography algorithms. So, this project aims to extend RISC-V instruction set architecture to improve performance of NewHope post-quantum cryptography algorithm.

RISC-V is an open source instruction set architecture. It is easy to add custom extensions to RISC-V which makes it a suitable architecture for the aim of this project. Several RISC-V cores were analyzed: Potato core, PULPino, and Ibex. Ibex was chosen for the project since it is easy to work on, easy to understand and modify the source code, and it implements the standard multiplication extension of RISC-V. A small disadvantage of Ibex is that it does not come with a bus interface or any peripherals. In order to easily add and use peripherals, a modified project of Ibex, ibex_wb is used. This project includes a Wishbone bus interface module. With this bus, any Wishbone compatible peripheral module can be connected to the core easily. There is an example project top module for FPGA in the repository. This module is used as a base for preparing a new one for Nexys 4 DDR FPGA board. For debug purposes and getting used to editing the project, several peripherals are added. These peripherals are GPIO, timer and UART modules. GPIO and timer modules are written from scratch whereas the UART module is taken from an open source repository. A C library is written for each peripheral to easily use them on the software side.

At this point the project needs to be synthesized and implemented again after each modification on the software. In order to avoid this delay, a bootloader program is written, This bootloader takes the application image file via UART, copies it to the RAM and runs the program from RAM. This configuration significantly improves the development speed.

After making the environment ready for development, NewHope library is downloaded from the official website and tested on the device. With the use of static counters and GDB, the application is profiled to analyze which functions are used the most. Later with these analysis, custom instructions are decided considering the repeat amount and difficulty to implement.

In order to add a custom extension the ALU and the instruction decoder must be edited. Also, the compiler source code must be edited and rebuilt to let the compiler know about custom instructions. Later the custom instructions are used with inline assembly code inside the C code.

The first custom instruction is Hamming weight difference. This instruction takes the Hamming weights of two different values and takes the difference of them.
The second custom instructions is a part of coeff_freeze function which is a subtraction followed by a series of logic operations.
The third and final custom instruction is a part of flipabs function which is again a subtraction followed by a series of logic functions.

The instructions are used with inline assembly and the clock cycles are measured with performance counters of RISC-V. As a result, the algorithm takes 6.90% less clock cycles to complete with all the custom extensions while the area used is increased by 6.16%.

**RISC-V MİMARİLİ İŞLEMCİLER ÜZERİNDE
POST-KUANTUM KRİPTOPGRAFİ ALGORİTMALARI İÇİN
KOMUT SETİ GENİŞLETİLMESİ**

ÖZET

Günümüzde kuantum bilgisayarlar üzerine çalışmalar arttıkça, güvenlik gitgide daha da önem kazanmaktadır. Şu anda yaygın olarak kullanılan RSA gibi şifreleme algoritmaları kuantım bilgisayarlara karşı dayanıklı değildir. Kuantum bilgisayarların çalışma mantığından dolayı bu şifrelemeler çok daha hızlı bir şekilde kırılabilecektir. Bu sebeple yakın gelecekte kuantum bilgisayarlara dayanıklı (post-kuantum) şifreleme algoritmalarına ihtiyaç duyulacaktır. Post-kuantum şifreleme algoritmalarını standardize etme amacıyla NIST bir proje başlatmıştır. Bu proje yapılan katılımlar arasında, NewHope algoritması ümit vadedenler arasındadır. Bu sebeple bu projede NewHope algoritması kullanılması kararı verilmiştir. Post-kuantum şifreleme algoritmaları kompleks matematik işlemleri içerdiğinden genellikle yavaş çalışırlar. Nesnelerin interneti teknolojilerinin de gelişmesiyle kişisel veriler daha da dijitalleşmeye başlamıştır ve bu güvenliğe daha da önem kazandırır. Bu sebeple RISC mimamirili küçük cihazların bile post-kuantum şifreleme algoritmaları kullanması gerekebilir. Bu projenin amacı RISC-V mimarisine komut kümesi eklentisi yaparak NewHope post-kuantum şifreleme algoritmasının performansını arttırmaktır.

RISC-V açık kaynaklı bir komut kümesi mimarisidir. RISC-V mimarisine eklenti yapmak basit olduğundan bu projenin amacına uygundur. Potato core, PULPino ve Ibex olmak üzere birkaç RISC-V çekirdeği incelenmiştir. NewHope algoritmasına geçmeden önce tüm çekirdekler üzerinde NTRU algoritması ile performans testleri yapılmıştır. Potato hiçbir standart eklentiyi içermediğinden performans açısından çok geride kalmışıtr. PULPino ile iyi bir performans alınsada proje ve kaynak kodlar biraz karışıktır. Ibex ile de iyi bir performans alınmıştır. Kaynak kodlarının kolay anlaşılabilir ve değiştirilebilir olmasından ve standart çarpma eklentisini de içermesinden dolayı Ibex ile devam etme kararı verilmniştir. Ibex'in ufak bir dezavantajı çevresel modüllerin bağlanabileceği herhangi bir arayüz içermemesidir. Bu yüzden Ibex'in ekleme yapılmış bir versiyonu olan ibex_wb projesi kullanılmıştır. ibex_wb içerisinde Wishbone bus arayüzü bulundurur. Bu sayede herhangi bir Wishbone uyumlu çevresel modül, sisteme kolaylıkla eklenebilir.

Projenin içerisinde, FPGA üzerinde kullanılmak üzere yazılmış örnek bir top modül dosyası bulunmaktadır. Bu örnek kod baz alınarak, Nexys 4 DDR FPGA kartı üzerinde kullanmak amacıyla yeni bir top modül ve xdc dosyası yazılmıştır. Bu yeni top modül içerisinde çevresel modül eklemeyi kolaylaştırmak amacıyla modüller için enumlar tanımlanmıştır. Modüllerin adres, boyut ve sinyalleri dizilerde tutulup, bu diziler enum kullanılarak endekslenmektedir. Bu sayede modüllerin eklenmesi ve sinyal bağlantıları çok basit bir hal alır.

Proje üzerinde çalışmaya alışmak ve ileride debug amacıyla kullanmak üzere birkaç modül eklenmiştir. Bu modüller GPIO, zamanlayıcı ve UART modülleridir. Eklenen

bütün modülleri yazılım tarafında rahatlıkla kullanabilmek için, herbirine ayrı ayrı C kütüphaneleri yazılmıştır.

Wishbone uyumlu GPIO modülü sıfırdan yazılmıştır. Bu modül içerisinde 3 adet register bulunur: yön, giriş ve çıkış registerları. Yön registerı hangi pinlerin giriş hangi pinlerin çıkış olduğunu belirtir. Giriş registerı giriş bilgilerindeki sinyali okumak için, çıkış registerı ise çıkışları değiştirmek için kullanılır.

Aynı şekilde Wishbone uyumlu zamanlayıcı modülü de sıfırdan yazılmıştır. Zamanlayıcı modülünün 2 adet registerı vardır: kontrol ve sayaç. Kontrol registerında sayacın çalışması veya sıfırlanmasını belirten bayraklar bulunmaktadır. Sayaç registerı ise ne kadar zaman geçtiğini belirten, aktif durumdayken her saat darbesiyle değeri 1 arttırılan registerdır.

UART için açık kaynaklı, Wishbone uyumlu hazır bir modül kullanılmıştır. Yalnızca Wishbone sinyallerini ibex_wb projesinde varolan Wishbone arayüzüne uyarlamak amacıyla bir wrapper modül eklenmiştir.

Bu aşamada yazılım üzerinde herhangi bir değişiklik yapıldığında bütün proje baştan sentezlenmeli ve yeni bir bitstream dosyası üretilmelidir. Bu işlem çok zaman almaktadır. Bu gecikmeyi önlemek amacıyla bir bootloader programı yazılmıştır. Bu program başlangıçta hafızaya gömülü bir şekilde bulunup otomatik çalışır. UART üzerinden aldığı bir uygulama dosyasını hafızaya yazarak program akışını burdan devam ettirir. Bu sayede yazılımda bir değişiklik yapıldığında projeyi baştan sentezlemek yerine tek yapılması gereken işlemciyi resetleyip, ;UART üzerinden yeni uygulama dosyasının gönderilmesidir. Başlangıçta bu program hafızada boş kalan yer kadar bir dosya alacak şekilde tasarlanmıştır. Bu sebeple eğer uygulama dosyası küçükse, sonuna sıfırlar eklenerek gerekli boyuta büyütülür. UART üzerinden dosya aktarımı yavaş olduğundan bu tasarım da gecikmelere yol açmaktadır. Bunu önlemek için önce UART üzerinden dosya boyutunu alıp daha sonra bu boyut kadar veri okuyacak şekilde bootloader güncellenmiştir. Bu sayede program geliştirip test etmek çok daha hızlı bir hale gelir.

İşlemci, uygulama geliştirme ve test etmeye hazır bir hale geldikten sonra NewHope algoritması ile çalışmalara başlandı. NewHope algoritmasının C implementasyonu NewHope resmi sitesi üzerinden elde edilmiştir. Bu implementasyonun neredeyse tamamı standart C kütüphaneleri ile yazılmıştır. Yalnızca rastgele sayı üreteci için OpenSSL kütüphanesi kullanılmıştır. Bu projenin amacına bir etkisi olmayacağından, OpenSSL kütüphanesini RISC-V için derlemek yerine, rastgele sayı üreteci standard C fonksiyonu olan rand fonksiyonu ile değiştirilmiştir. Algoritma, Ibex işlemcisi üzerinde başarıyla çalıştırıldıktan sonra profil çıkarma işlemine başlanır. GDB debugger ve C içerisinde statik sayaçlar kullanılarak algoritma içerisinde hangi fonksiyonların ne kadar sık çağırıldığı incelenmiştir. Sık kullanılan ve donanım implementasyonu mümkün olduğunca basit olan fonksiyon ve fonksiyon parçaları, yeni komut eklentisi yapılmak üzere seçilmiştir.

Yeni bir komut eklemek için hem ALU hem de decoder üzerinde değişiklik yapılmalıdır. ALU operatör enum'ına yeni komut eklenir. ALU içerisine komutu gerçekleyen donanım eklenip, çıkış multiplexer'ına bağlanır. Decoder içerisinde de yeni komut için belirlenen funct3/funct7 kombinasyonu geldiğinde komuta karşılık düşen ALU operatörünün seçilmesi eklenir. Zaten varolan register-register opcode'una bir ekleme olduğundan bu bir brownfield eklentidir. Bu komutların aynı zamanda derleyiciye de tanıtılması gerekir. Bunun için derleyicinin kaynak kodunda değişiklikler yapılarak, derleyici baştan build edilir. Daha sonra eklenen komutlar C içerisinde inline assembly şeklinde kullanılabilir.

Eklenen ilk komut, Hamming ağırlık farkıdır. Hamming ağırlığı bir sayı içerisinde değeri 1 olan bitlerin sayısıdır. Bu eklenen komut iki sayısnın Hamming ağırlıklarını bulup bu değerlerin farkını alır. Hamming ağılığının hesaplanması için 4-bit look-up table kullanılmıştır. Operandlar 4-bit dilimlere ayrılıp, hepsinin Hamming ağırlıkları bulunup sırayla toplanır ve en son iki ağırlığın farkı alınır.

İkinci komut coeff_freeze adlı bir fonksiyonun içinden bir parçadır. Başta bir çıkarma işlemi yapılıp daha sonra bir dizi lojik işlem yapılmaktadır. Başlangıçtaki çıkarma işlemi için zaten ALU içerisinde bulunan toplama-çıkarma modülü kullanılarak alandan tasarruf edilir.

Üçüncü ve son komut ise flipabs fonksiyonunun bir parçasıdır. Yine ikinci komut gibi buradada bir çıkarma işlemi ve ardından bir dizi lojik işlem yapılmaktadır.

Algoritma, eklenen bütün komutlar için ayrı ayrı ve en son hepsi beraber test edilmiştir. RISC-V'in performans sayaçları kullanılarak algoritmanın kaç saat darbesinde tamamlandığı ölçülmüştür. Yapılan testler sonucu, bütün eklentiler kullanıldığında algoritmanın %6.90 daha kısa sürede tamamlandığı ve kullanılan alanın %6.16 arttığı gözlenmiştir.

1. INTRODUCTION

In this graduation project, instruction set extension on RISC-V cores for post-quantum cryptography algorithms is implemented. In order to create an instruction set extension for RISC-V cores, three different RISC-V cores have been studied on. These cores are Potato RISC-V, PULPino and Ibex. Results of post-quantum cryptography codes are compared between the cores with their runtimes and memory usages and then, an extension is implemented for the suitable core.

Since protecting personal and private datas are so important in today's world, encrypting information is an important part of life. But with the quantum computing technology according to the Shor's and Grover's algorithms it is seen that classical cryptography algorithms such as Rivest, Shamir, Adleman algorithm (RSA) etc. will be easily cracked and became vulnerable [7] [8].

Therefore, design of an algorithm that is strong enough to encrypt data that would not be easily cracked with both post-quantum computing and classical computing methods. That's why National Institute of Standards and Technology (NIST), has opened a project and tries to standardize the encryption algorithm for post-quantum computational system [9].

In this project, a post-quantum cryptography algorithm is analyzed and have been implemented on a RISC-V core. In order to implement this algorithm, Xilinx Vivado tools and Nexys 4 DDR Field-Programmable Gate Arrays (FPGA) are used.

2. POST-QUANTUM CRYPTOGRAPHY

Today's computers simply represent the data as 1s and 0s and all the information need to be converted and represented as with these two bits as a result. The theory of being able to become two state at the same time (superposition and entanglement) of quantum mechanics started the researches on quantum computers which works with qubits. "Superposition can transfer the complexity of the problem from a large number of sequential steps to a large number of coherently superposed quantum states. Entanglement is used to create complicated correlations that permit interference between the parallel “computations” performed by the machine." [10]. Qubits can represent 1s, 0s and also the superposition state which is the possibility of being both at the same time. By being able to process third state of unclarity, in theory, computations which are done with classical computers can be calculated parallel with this stochastic approach and as a result basically computational loops can be transformed into a single computation. “Taking benefit of the superposition principle, it could process simultaneously all the possible inputs. This “ massive quantum parallelism” enables the quantum computer to perform in a single run 2^n calculations on an n qubits input.” [11].

Since two of the mathematical problems that today's cryptography methods predicated on, are integer factorization and discrete logarithm problems and also hardnesses of these problems are based on computational loops, this new computational power can be threatening for these cryptography methods. According to Peter Shor's and Lov Grover's algorithms, quantum computers can search possible permutations faster and can find prime factors of integers easier, so especially the crypto systems which are based on these mathematical equations are in danger against quantum computers. [7] [8].

There are very common cryptography systems which are used to encrypt our data based on the integer factorization problem, elliptic curve discrete logarithm problem and discrete logarithm problem such as Rivest–Shamir–Adleman (RSA). RSA uses computational workload of the reverse engineering for prime factors of a very large numbers which are the multiplication of large prime numbers and commonly used at cryptography world such as at TLS. According to Bernstein, “Shor’s algorithm and its generalizations will then completely break RSA, DSA, ECDSA, and many other popular cryptographic systems: for example, a quantum computer will find an RSA user’s secret key at essentially the same speed that the user can apply the key”. [12]. Therefore idea of ending up in an unsecure environment after the realization of quantum computational powers has increased the importance of Post Quantum Cryptography.

Post Quantum Cryptography is simply the research of secure algorithms designed to be run on classical machines and counted as secure even against the quantum computational power. This algorithms needs to be run on non-quantum machines since quantum computing is still at its early stages, and needs to be based on new mathematical problems which are secure against Shor’s and Grover’s algorithms. “The goal of post-quantum cryptography (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and classical computers, and can interoperate with existing communications protocols and networks” [13].

So, National Institute of Standards and Technology (NIST) has started a project to discover Post Quantum Cryptography algorithms which are more secure against quantum computer attacks and trying to standardize Post Quantum Cryptography. Right now project has reached its second phase. Some of the algorithms proposed to the project have already been started to be used by technology companies as a testing purposes. “The company has now successfully demonstrated the first PQC implementation on a commercially available contactless security chip, as used for electronic ID documents.” [14].

We have also chosen one of these Post Quantum Cryptography algorithms from phase two for our thesis project. Algorithm’s name is NewHope and it is under the group of lattice-based Post Quantum Cryptography methods [15].

2.1 Lattice Based Cryptography

"A lattice is a set of points in n-dimensional space with a periodic structure.... More formally, given n-linearly independent vectors b1,...,bn Rn, the lattice generated by them is the set of vectors." [16]. Example of a basic lattice can be seen from the Figure 2.1.

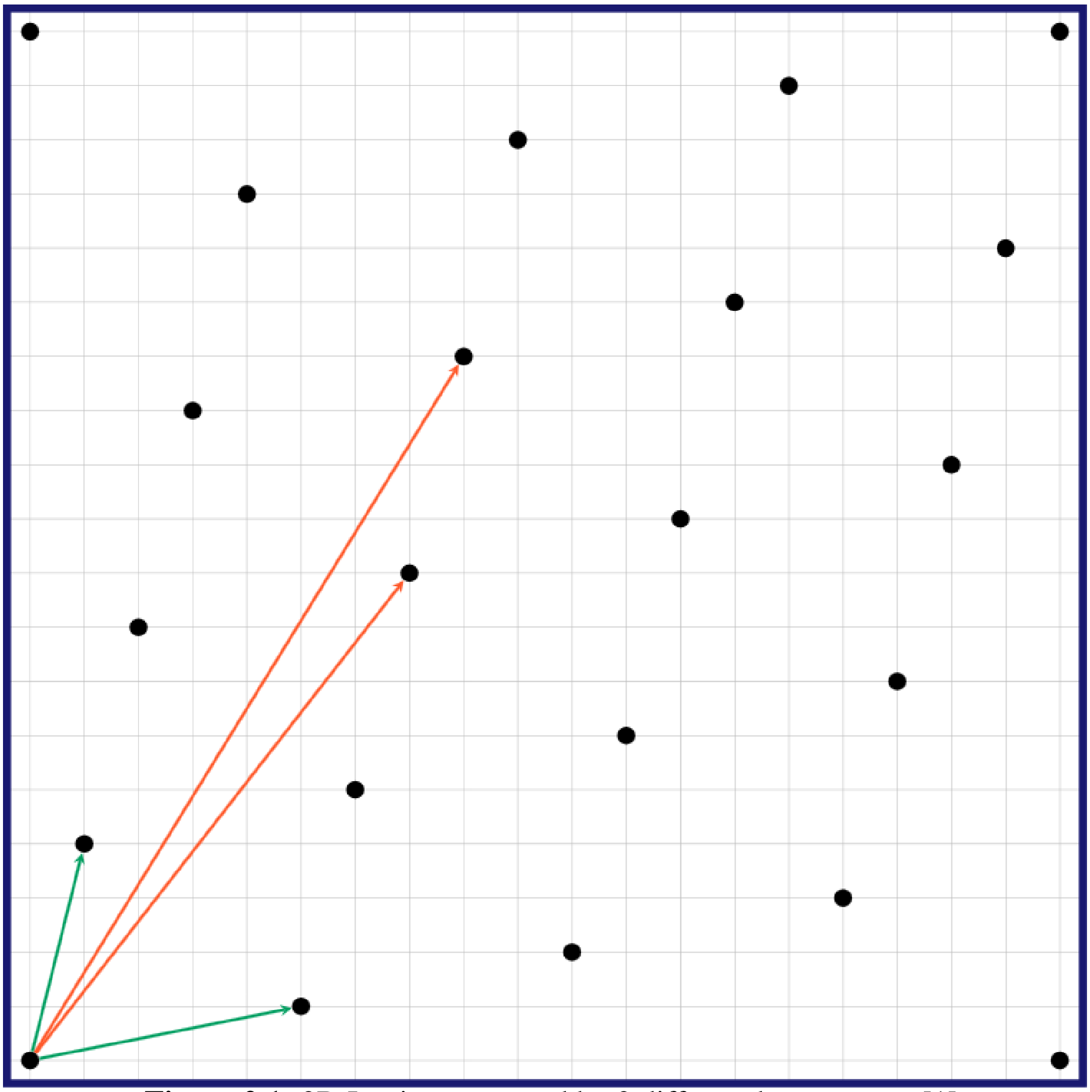

Figure 2.1: 2D Lattice generated by 3 different base vectors [1]

Most of the lattice based methods which are counting on computational lattice problems are counted as secure and used often for Post Quantum Cryptography algorithms. "... Micciancio and Regev conclude that "there is no polynomial time algorithm that approximates lattice problems to within polynomial factors" [16]. Even for the NIST Post Quantum Cryptography project has 12 lattice based candidates out of 26 applicants. [9]. There are two different computational lattice problems that are based on for cryptography algorithms; Shortest Vector Problem and Closest Vector Problem.

Shortest Vector Problem is searching of a shortest lattice vector (or a point) when a basis of a lattice is given as it can be seen from the Figure 2.2.

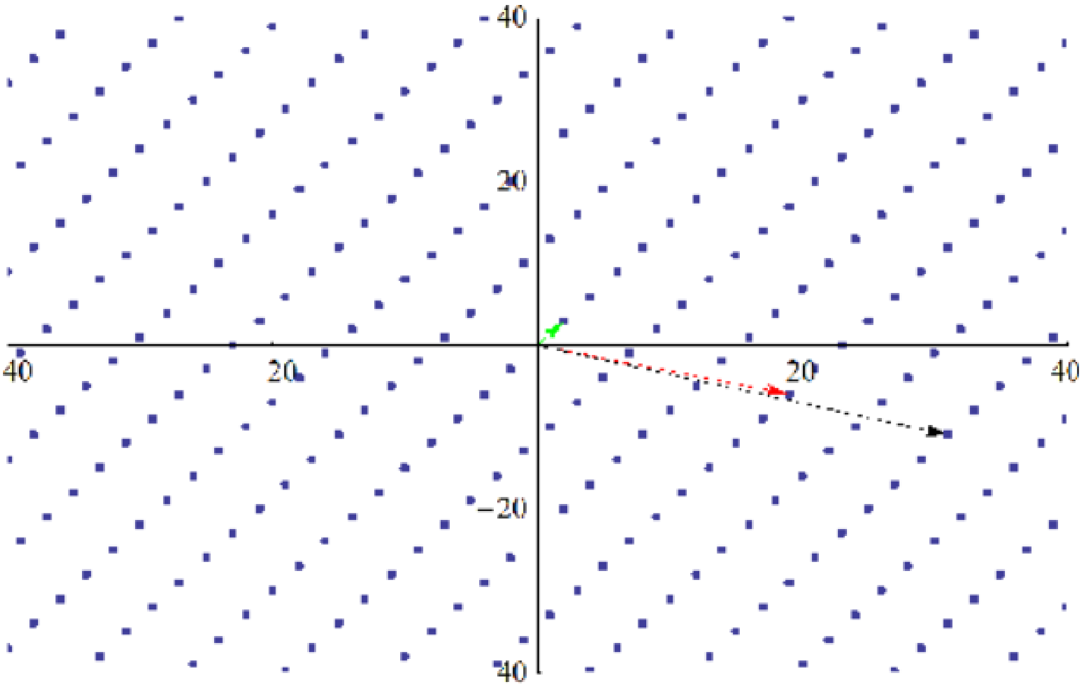

Figure 2.2: Shortest Vector Problem example. [2]

Closest Vector Problem is similar to the SVP, when given a target vector and a basis of a lattice, it tries to find the closest lattice point to this target vector. At the Figure 2.3 closest point to the given vector can be seen from the right picture.

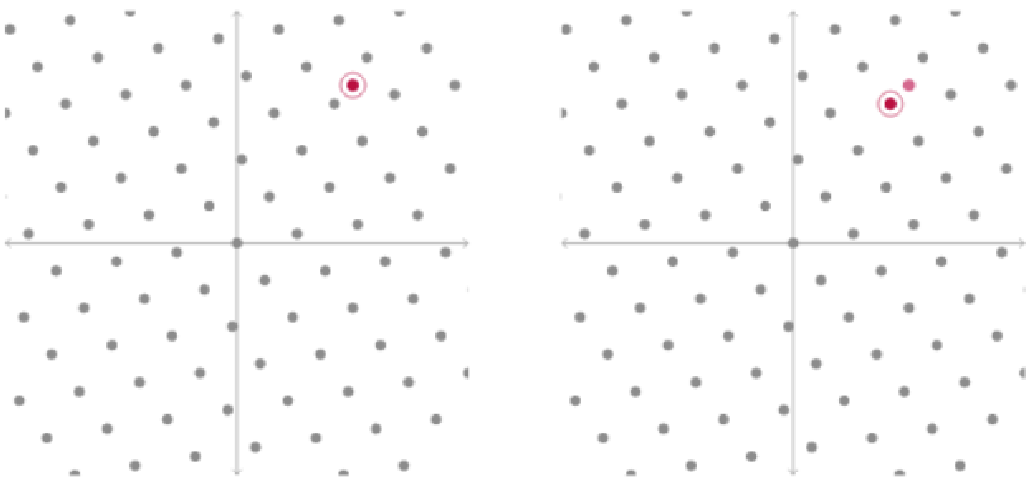

Figure 2.3: Closest Vector Problem example. [3]

2.2 Learning With Error

Given m samples of (a, b = <s,a> + e mod q) data with “e” as a small noise factor, learning with error (LWE) method is simply tries to estimate the secret “s” when “a” and “b” are known. When this equation is simplified to the (a, b = <s,a>), it can be solved by Gaussian Elimination but with an extra small noise “e” added to the equation, it transforms the equation into a machine learning problem.

When the coefficients become polynomials and the number of samples increases it turns into a computational lattice problem since the attacker needs to guess the closest vector to the “b” as it is “<s,a>”. For Post Quantum Cryptography this approach is used with relatively huge polynomials on a relatively larger lattices to increase security.

In pure LWE algorithms huge coefficient matrices are needed, so to be able to minimize the size and the efficiency other implementations of LWE are derived such as Ring Learning With Error.

For pure LWE, coefficients of the matrices need to be preserved after they are generated, so as a result they occupy large space at memory and as the dimension increases, used memory size will also increases. Ring Learning With Error prevents this situation. Even though there are other implementations of R-LWE, simply by sending the first row of a matrice with a predetermined rule, such as each row can be 2 times cyclic right shifted version of the previous one with mod x for a wrapping rule, sender doesn’t need to create or preserve the other coefficients. Receiver will generate rest of the matrice with the provided rule if it is needed so while it decreases the memory usage, it also speeds up the process.

2.3 NewHope PQC Algorithm

NewHope is a lattice based Post Quantum Cryptography algorithm working with Ring Learning With Error approach on its core. The version we have implemented is NEWHOPE-512-Chosen Ciphertext Attacks-Key Encapsulation Mechanism which is retrieved from the official website of NewHope [17].

At the upper layer of abstraction, algorithm consists of three steps: Key Generation, Encapsulation, Decapsulation.

Inside the NewHope-CCA-KEM algorithm there exists a PKE implementation of a previous NewHope Simple project but since it is transformed into a Key Encapsulation

Mechanism, it could not handle message encyrption with different lengths and just used as a transformation step from Public Key Encryption to the Key Encapsulation Mechanism. Because of this, the version of the algorithm we have implemented, contains PKE functions also.

Algorithm 19 NewHope-CCA-KEM Key Generation

function NewHope-CCA-KEM.Gen()
  $(pk, sk) \xleftarrow{\$}$ NewHope-CPA-PKE.Gen()
  $s \xleftarrow{\$} \{0, \ldots, 255\}^{32}$
  return $(pk, \overline{sk} = sk \| pk \| \mathsf{SHAKE256}(32, pk) \| s)$

Figure 2.4: NewHope Key Generation. [4]

Algorithm 20 NewHope-CCA-KEM Encapsulation

function NewHope-CCA-KEM.Encaps(pk)
  $coin \xleftarrow{\$} \{0, \ldots, 255\}^{32}$
  $\mu \leftarrow \mathsf{SHAKE256}(32, \mathtt{0x04} \| coin) \in \{0, \ldots, 255\}^{32}$
  $K \| coin' \| d \leftarrow \mathsf{SHAKE256}(96, \mathtt{0x08} \| \mu \| \mathsf{SHAKE256}(32, pk)) \in \{0, \ldots, 255\}^{32+32+32}$
  $c \leftarrow$ NewHope-CPA-PKE.Encrypt($pk, \mu; coin'$)
  $ss \leftarrow \mathsf{SHAKE256}(32, K \| \mathsf{SHAKE256}(32, c \| d))$
  return $(\bar{c} = c \| d, ss)$

Figure 2.5: NewHope Encapsulation. [4]

Algorithm 21 NewHope-CCA-KEM Decapsulation

function NewHope-CCA-KEM.Decaps($\bar{c}, \overline{sk}$)
  $c \| d \leftarrow \bar{c} \in \{0, \ldots, 255\}^{3n/8+7n/4+32}$
  $sk \| pk \| h \| s \leftarrow \overline{sk} \in \{0, \ldots, 255\}^{7n/4+7n/4+32+32+32}$
  $\mu' \leftarrow$ NewHope-CPA-PKE.Decrypt(c, sk)
  $K' \| coin'' \| d' \leftarrow \mathsf{SHAKE256}(96, \mathtt{0x08} \| \mu' \| h) \in \{0, \ldots, 255\}^{32+32+32}$
  if $c =$ NewHope-CPA-PKE.Encrypt($pk, \mu'; coin''$) **and** $d = d'$ **then**
    $fail \leftarrow 0$
  else
    $fail \leftarrow 1$
  $K_0 \leftarrow K'$
  $K_1 \leftarrow s$
  return $ss = \mathsf{SHAKE256}(32, K_{fail} \| \mathsf{SHAKE256}(32, c \| d))$

Figure 2.6: NewHope Decapsulation. [4]

2.3.1 Randomness and sampling

NewHope-CCA-KEM uses byte arrays as a data structure for both sampled datas, and preserved coefficients. SHAKE256 function is used for hashing, squezing and expanding the byte arrays according to the given seed. [18]. It takes two arguments, one for input data byte array d and one for number of output bytes. Since its output is also a byte array, output values are in between 0,,,,255.

V <- SHAKE256(64, seed) : by using 32 byte random seed it fills V with a byte array of 64 elements

For the generation of noise factor “e”, binomial sampling is used. It is preferred instead of Gaussian distribution since it is easier to implement and does not require large tables.

Algorithm 4 Deterministic sampling of polynomials in $\mathcal{R}_q$ from ψ_8^n

function SAMPLE($seed \in \{0, \ldots, 255\}^{32}$, positive integer $nonce$)
  $\mathbf{r} \leftarrow \mathcal{R}_q$
  $extseed \leftarrow \{0, \ldots, 255\}^{34}$
  $extseed[0{:}31] \leftarrow seed[0{:}31]$
  $extseed[32] \leftarrow nonce$
  for i **from** 0 **to** $(n/64) - 1$ **do**
    $extseed[33] \leftarrow i$
    $buf \leftarrow \mathsf{SHAKE256}(128, extseed)$
    for j **from** 0 **to** 63 **do**
      $a \leftarrow buf[2 * j]$
      $b \leftarrow buf[2 * j + 1]$
      $r_{64*i+j} = \mathsf{HW}(a) + q - \mathsf{HW}(b) \bmod q$
  return $\mathbf{r} \in \mathcal{R}_q$

Figure 2.7: Sampling from Binomial Distribution. [4]

2.3.2 Number Theoretic Transformation (NTT)

Since NewHope algorithm is based on lattice-based R-LWE approach, polynomial computations are often calculated throughout the implementation. Subtraction and addition of polynomials can be carried out as coefficient-wisely but multiplication of polynomials is a challenging task when timing and resource constraints are considered. [4]

"The Number Theoretic Transform (NTT) provides efficient algorithms for cyclic and nega-cyclic convolutions, which have many applications in computer arithmetic, e.g., for multiplying large integers and large degree polynomials." [19]. NewHope algorithm uses NTT library for this challenging task to surpass these limitations of polynomial multiplications. What this task do is, it simply transforms the polynomials to the NTT domain with fourier transform operations and does the multiplications at that domain since its fourier transform is taken now. At NTT domain these multiplications can be carried out coefficient-wisely instead of multiplying the whole polynomials. And the result of this operations after coefficient-wise multiplication is inverse transformed from NTT domain to the regular domain.

2.3.3 Encryption Scheme

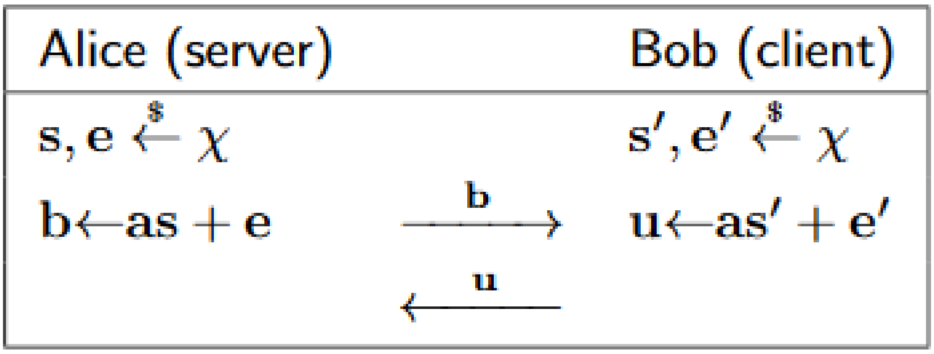

Figure 2.8: R-LWE Based KEM. [4]

Alice has v = us = ass' + e's

Bob has v'= bs' = ass' + es'

Since noise and secret polynomials s', s, e', e are small enough, v' and v are more or less the same values. So, after this transaction, each side will have the same key even though they have different noise and secret values at the beginning which are sampled from the same distribution.

2.3.4 NewHope Real Life Implementations

Although the algorithm is still in the project phase and quantum computers are still far away, especially the NewHope algorithm is started to be tested at various platforms and for some use cases it has already been deployed as a product. Thanks to its R-LWE implementations, its size is relatively small and since there are still no algorithms which exposes the quantum weaknesses of lattice-based problems such as Shor's and Grover's algorithm, NewHope is seen as a promising candidate for future implementations. According to Google's security blog, "We're indebted to Erdem Alkim, Léo Ducas, Thomas Pöppelmann and Peter Schwabe, the researchers who developed “New Hope”, the post-quantum algorithm that we selected for this experiment." [20].

“None the less, if the need arose, it would be practical to quickly deploy NewHope in TLS 1.2”. [21].

3. RISC-V

Instruction set architectures [5] (ISA) are models of computer which differs with the instruction's complexity. ISA defines supported data types, registers and also input/output model of implementation. RISC-V is a Reduced Instruction Set Computer (RISC) which is designed to have a high performance and power efficiency.

This architecture is open source and has support for 32-, 64- and 128-bits systems. Project began in 2010 by University of California, Berkeley. Its RV32I and RV64I base instruction sets are frozen and also, there are 6 other frozen extensions. RISC-V ISA has fixed 32-bit instructions in base instructions but ISA supports also 16-bit instructions which is in compressed instruction-set extension called "C". There are four types of base instructions which are R-, I-, S- and U-Type and its structures can be seen in Figure 3.1. And based on the immediate, two further instruction types are available, B- and J-Type.

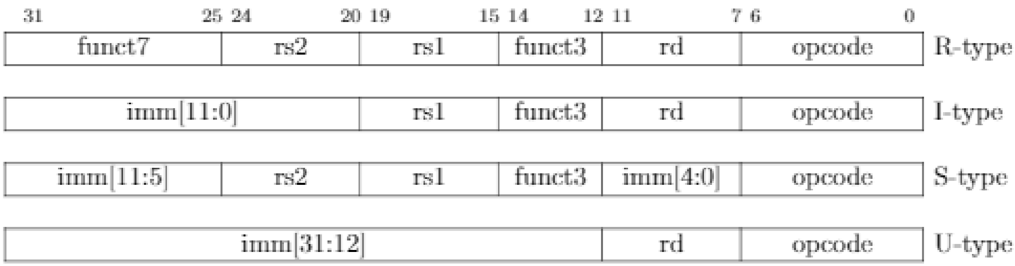

Figure 3.1: RISC-V Base Instruction Formats [5].

RISC-V architecture enables to have 32 general purpose registers (GPR), 32 floating-point registers (FPR) and 32 privileged control registers (PCR). Width of these registers may vary with its purpose and system definition. With the design specifications that are decided by an committee of RISC-V Foundation, RISC-V architecture enables the developers to design extension to the instruction sets [5].

3.1 RISC-V Applications

RISC-V specifications and further developments currently maintained by RISC-V Foundation which is a group of members around the world focused on developing a free and highly efficient ISA. There is a list available on the official site of RISC-V Foundation and it lists the both software and hardware projects contributed to the RISC-V community [22].

Hardware projects listed on RISC-V Foundation site [22] includes both cores and System on a Chip (SoC) platforms. The cores only have the instruction extension that are available on the RISC-V specifications and the SoC platforms also includes the peripherals of the system which enables us to use them with an interface.

Software projects listed on the site includes simulators, toolchains, bootloaders and operating systems (OS). Simulators that are available focused on simulating the behaviour and the outputs of the implemented RISC-V instruction set. The toolchain projects are focused on compiling and optimizing C and the higher level software languages into RISC-V specific programs and therefore, enables us to create and run programs on RISC-V. After that both bootloader and OS projects are designed to improve user experiences on RISC-V systems and also, to create a better research platforms. These projects improves the usability of the RISC-V cores and as seen in the RISC-V Foundation website most of these hardware and software projects are listed as free and open source [22].

3.2 RISC-V GNU Toolchain

RISC-V GNU Toolchain is included in GNU Compiler Collection (GCC), and it includes frontends for C, C++ and, also it is a free project. This software project enables us to compile and optimize C and C++ based software projects and create a RISC-V supported executables. Therefore, it provides flexibility and ease when developing a RISC-V software.

In the software development step, RISC-V GNU Toolchain will be used to compile C software and create their memory and executable files. Also, since new instructions will be added into the RISC-V SoC in the project, RISC-V GNU Toolchain will be modificated and used with these new added instructions.

3.2.1 Setup

In order to install RISC-V GNU Toolchain, prerequisites follows as shown in Table 3.1.

Table 3.1: RISC-V GNU Toolchain prequisities

CMake Version	≥2.6
GCC Version	≥5.2
Python Version	≥2.7

These prequisities can be controlled with the commands shown below for Linux systems.

$ cmake −−version

$ gcc −−version

$ python −−version

After that, these prequisities are confirmed, following code must be run.

$ 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

After these steps, RISC-V GNU Toolchain repository must be downloaded from the official RISC-V repositories. Then, the instructions written in the repository must be followed and RISC-V GNU Toolchain must be compiled and installed for your specific instruction set requirements which might be differ for other cores with respect to both instruction set extensions and hardware implementations of different units in the core and these steps can be found in the repositories of your RISC-V core. For example newlib installation can be made with the following lines,

$ ./configure −−prefix=/opt/riscv

$ make

Installation path is determined with the "**−−prefix**" in the codes above and can be changed to another path. These lines might take a long while to complete.

With the steps above completed, RISC-V GNU Toolchain can be used from the bash. But, if installation path is added to the "**.bashrc**" in the "**/home/user**" directory, "**riscv-gcc**" can be used without specifying the directory every time when RISC-V GNU Toolchain needed to be used. It can be added as shown in Figure 3.2.

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

PATH=$PATH:/tools/riscv/bin

```

Figure 3.2: Adding Toolchain path to the .bashrc

After that, RISC-V GNU Toolchain can be used with commands like "**riscv-unknown-elf-gcc**".

3.2.2 Toolchain Modifications

After modifications were made on RISC-V cores, the toolchain needs to be edited in order to run tests with the software so that it recognizes the custom instruction and creates the executable with this new instruction. The compiler will not be able to optimize the code with the newly added instruction but this modification is required in order to create C software for RISC-V with the custom extensions.

In order to add new instruction to RISC-V GNU Toolchain, modifications inside the "**riscv-bintuils**" must be made. Documents that needs to be modified can be seen in the Table 3.2 and these files can be found under the "**riscv-gnu-toolchain**" project folder.

Table 3.2: Modified documents

riscv-opc.c
riscv-opc.h

In the "**riscv-opc.c**" document which can be found under the "**./riscv-binutils/opcodes/**" directory, the new instruction must be added to the riscv_opcodes structure. New defined instructions can be added to the end of the structure. An example of this can be seen in the Figure 3.3.

```
{"cust0",         0, {"I", 0},   "d,s,t",  MATCH_CUST0, MASK_CUST0, match_opcode, 0 },
{"cust1",         0, {"I", 0},   "d,s,t",  MATCH_CUST1, MASK_CUST1, match_opcode, 0 },
{"cust2",         0, {"I", 0},   "d,s,t",  MATCH_CUST2, MASK_CUST2, match_opcode, 0 },
{"cust3",         0, {"I", 0},   "d,s,t",  MATCH_CUST3, MASK_CUST3, match_opcode, 0 },
/* Terminate the list.  */
{0, 0, {0}, 0, 0, 0, 0, 0}
};
```

Figure 3.3: riscv-opc.c Modification

Elements in the structure specifies the following:

First element shows the name of the instruction and can be named arbitrarily. Second one shows the "**Xlen-bits**", and should be selected accordingly to the added instruction in the core. Third one shows the which instruction set extension it is included in. In this step if a "brownfield extension" is aimed then, an instruction set extension that exists in the core can be selected but if a "greenfield extension" is expected than you can add your instruction accordingly to that instruction specification which you have created [5]. Forth one in the structure lists are the operands that your instruction needs and also, it should be defined with the requirements defined in your instruction. Fifth, sixth and the seventh ones are about the structure of your instruction and it masks and unmasks your operands with these elements added into the structure. For the greenfield extensions there are much less free space than the brownfield extension in the RISC-V's encoding space.

After that modification made in the "**riscv-opc.c**", also modifications must be made in the "**riscv-opc.h**" document which is located under the "**./riscv-binutils/include/opcode/**" directory. In that file, "opcode mask" and "opcode match" should be defined for new instructions. An example of modifications can be seen in Figure 3.4 and 3.5.

```
#define MATCH_CUST0 0x30000033
#define MASK_CUST0  0xfe00707f
#define MATCH_CUST1 0x50000033
#define MASK_CUST1  0xfe00707f
#define MATCH_CUST2 0x60000033
#define MASK_CUST2  0xfe00707f
#define MATCH_CUST3 0x70000033
#define MASK_CUST3  0xfe00707f
```

Figure 3.4: riscv-opc.h match and mask definitions

```
DECLARE_INSN(cust0, MATCH_CUST0, MASK_CUST0)
DECLARE_INSN(cust1, MATCH_CUST1, MASK_CUST1)
DECLARE_INSN(cust2, MATCH_CUST2, MASK_CUST2)
DECLARE_INSN(cust3, MATCH_CUST3, MASK_CUST3)
```

Figure 3.5: Instruction declaration in riscv-opc.h

As seen in the Figure 3.4 and 3.5, "match" and "mask" values are defined for each new added instruction. Given figures in this chapter is an example of greenfield extension and therefore it uses the existing opcodes and instruction types. Some of the instruction types can be seen in Figure 3.1 and also, with the bits allocation as seen in the figure mask and match values can be obtained.

After that, these modifications are complete on both documents for each instruction than RISC-V GNU Toolchain must be install again with these modified documents. Following two commands must be used for your new toolchain,

$ ./configure −−prefix=/opt/riscv

$ make

This step will take a long while to complete. In order to reduce the time spent on this step, a set of custom instructions can be defined with the instructions types that are most likely to be used in the project. Therefore, there will be no need to install the toolchain for every new instruction that are created during the project.

After that the installation complete, new instructions can be used in the softwares but as mentioned before since RISC-V GNU Toolchain will not be able to optimize the code with the new added instructions, these specific instructions must be added as their Assembly Language forms. So, the inline assembly method should be used when writing a C code with using the new custom instruction. New instruction usage can be seen in Figure 3.6.

```c
int main(){

    int result, a, b;

    asm("cust0 %[result1], %[value1], %[value2]\n\t" :
        [result1] "=r" (result) : [value1] "r" (a) , [value2] "r" (b));

}
```

Figure 3.6: Inline assembly method

4. PULPino

PULPino is a RISC-V project which is implemented by ETH Zürich and it uses a RISC-V core named RI5CY. Also, there are several companies around the world that support the PULPino project such as NVDIA, Google and Microsemi. Project purpose is to create a highly efficient ultra low power RISC-V system [6].

4.1 PULPino Architecture

PULPino project is a SoC platform which includes several peripherals around the RI5CY [23] core. These peripherals help the users to create a system for their test and improvements. Peripherals that are included in the PULPino project are Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), Joint Test Action Group (JTAG), Inter-IC Sound (I2S), Direct Memory Access (DMA) and General Purpose Input Outputs (GPIO). PULPino project architecture can be seen in the Figure 4.1.

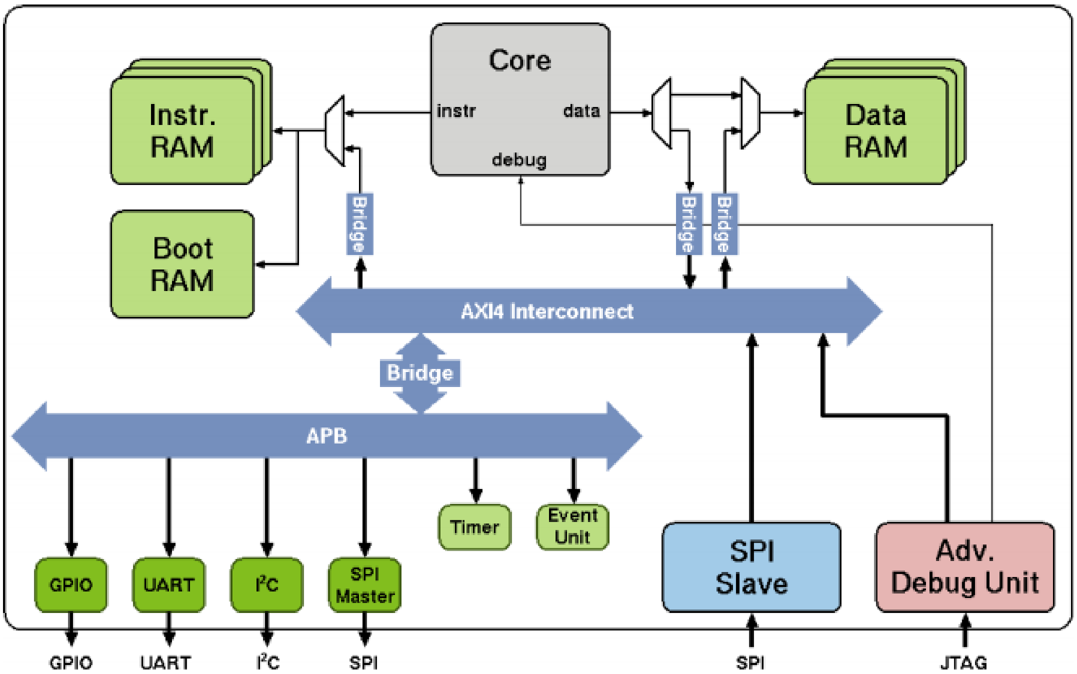

Figure 4.1: PULPino architecture [6]

As seen in the Figure 4.1, PULPino project has been designed as a single core micro-controller. Project has a RISC-V core called RI5CY which has a several different features besides an ordinary RISC-V core such as "hardware loops", "post-incrementing ld/st", "multiply-accumulate" and some Arithmetic Logic Unit (ALU) extensions like min, max and absolute value. With these extensions added, project aimed to save from unnecessary instructions and branches. An overview of the RI5CY core can be seen in Figure 4.2.

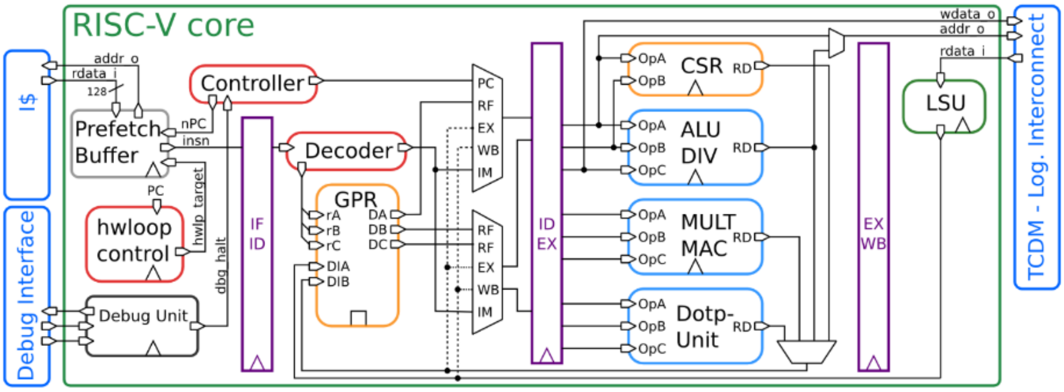

Figure 4.2: RI5CY core overview [6]

4.2 Setup

In order to create a test system with PULPino, the necessary requirements are as follows in Table 4.1,

Table 4.1: PULPino prequisities

CMake Version	≥2.8.0
GCC Version	≥5.2
RISC-V GNU Toolchain	riscv32-unknown-elf-gcc
Xilinx Vivado	2015.1
ModelSim	≥10.2c

For this project to be implemented on an FPGA and get tested, Xilinx Vivado 2015.1 must be used and RISC-V GNU Toolchain must be configured correctly.

4.2.1 RISC-V GNU Toolchain

In order to install a toolchain with extensions of RI5CY core, there is a project called "**pulp-riscv-gnu-toolchain**" on "pulp-platform" repository. Installation steps of this toolchain is similar to the official RISC-V GNU Toolchain with the differences on configuration step and steps can be seen below,

$ ./configure −−prefix=/opt/riscv −−with-arch=rv32imc −−with-cmodel=medlow −−enable-multilib
$ make

After the installation step of toolchain, C codes can be analyzed and their memory files can be created with using "**riscv32-unknown-elf-**" from the toolchain. Some example of usage of toolchain can be seen below,

$ riscv32-unknown-elf-gcc –o program.elf program.c
 # creates .elf file of the program which is its assembly language version before
 # turned into the machine language.
$ riscv32-unknown-elf-objcopy -O binary program.elf program.bin
 # creates binary and .elf file of the program.

In the Figure 4.3, an example of .dis disassembly can be seen. With this disassembly of the C code, code can be analysed and modification can be made in order to create an efficient algorithm.

4.2.2 Implementation on FPGA

There are two different board that PULP platform has configured the project on and these two can be selected with "**setenv BOARD**" command as "zybo" and "zedboard". After that the board is selected, "**make all**" command should be used in the directory "**./fpga**" under the project repository. This command will run Vivado 2015.1 and it will create the necessary bitstream files for the FPGA.

In our implementation, Nexys 4 DDR FPGA board [24] is used and therefore changes made on the PULPino project files and .xdc constraints file has been modified for the Nexys 4 DDR FPGA board.

4.3 Simulation Environment

ModelSim 10.2c is recommended in the repository of the PULPino for the simulations of the project but since RI5CY core has hardware loops inside, behavioral simulations of the project fails with both Vivado and ModelSim. Therefore, post-implementation

```
00010114 <_start>:
  la x26, _bss_start
   10114: 00001d17            auipc s10,0x1
   10118: e38d0d13            addi  s10,s10,-456 # 10f4c <_edata>
  la x27, _bss_end
   1011c: 00001d97            auipc s11,0x1
   10120: e3cd8d93            addi  s11,s11,-452 # 10f58 <_bss_end>
  bge x26, x27, zero_loop_end
   10124: 01bd5863            bge s10,s11,10134 <main_entry>

00010128 <zero_loop>:
  sw x0, 0(x26)
   10128: 000d2023            sw  zero,0(s10)
  addi x26, x26, 4
   1012c: 004d0d13            addi  s10,s10,4
  ble x26, x27, zero_loop
   10130: ffaddce3            bge s11,s10,10128 <zero_loop>

00010134 <main_entry>:
  addi x10, x0, 0
   10134: 00000513            li  a0,0
  addi x11, x0, 0
   10138: 00000593            li  a1,0
  jal x1, main
   1013c: 18c000ef            jal ra,102c8 <main>

00010140 <memcpy>:
   10140: 00a5c7b3            xor a5,a1,a0
   10144: 8b8d                  andi  a5,a5,3
   10146: 00c508b3            add a7,a0,a2
   1014a: e7a1                  bnez  a5,10192 <memcpy+0x52>
   1014c: 478d                  li  a5,3
   1014e: 04c7f263            bgeu  a5,a2,10192 <memcpy+0x52>
   10152: 00357793            andi  a5,a0,3
   10156: 872a                  mv  a4,a0
   10158: eba9                  bnez  a5,101aa <memcpy+0x6a>
```

Figure 4.3: Disassembly file example

simulations must be made in order to get reliable results from the simulation but with the cell delays and post-implementation considered this simulation works so slow.

Since its hard to get simulation result for PULPino project, verification tools can be used to get results of applications.

4.4 Applications

PULPino project has example codes in order to test the peripherals of the system and also, project has a base libraries of the PULPino for the first time configurations of the core. Project needs a boot code contained in the "**boot_code.sv**" file under the project hierarchy.

For the first tests of the project since there is no OS running on the core, applications can be ran with changing the boot code for every applications to test. For every boot code, project bitstream must be generated again.

Since PULPino project has several peripherals, UART codes can be written and checked with USB interface with using "**minicom**" program. As seen in the Figure 4.4, a C code is compile and uploaded to PULPino with changing "boot_code.sv" file in the hierarchy, tested and "Hello World!!!!!" outputs has been observed in the terminal window.

```c
#include <stdio.h>

int main()
{
  printf("Hello World!!!!!\n");
  return 0;
}
```

Figure 4.4: UART helloworld! example

4.5 Tests

In order to test the performance of the PULPino platform, a PQC algorithm which is tested in different platforms is tested on PULPino. The PQC algorithm is called NTRU. So, in order to obtain a comparison data, Nth Degree Truncated Polynomial Ring Unit (NTRU) [25] algorithm is tested with different key lengths and program runtimes are recorded. Comparison of algorithm runtime with two different cores can be seen in Table 5.1.

As seen in Table 5.1, PULPino has a 15 times better result than Potato RISC-V core but it seems that it is unable to work with algorithm that has longer key length.

5. Potato RISC-V

Potato is a simple RISC-V project written in Very High Speed Integrated Circuit Description Language(VHDL) [26]. Potato implements RV32I instruction set [5] and it also has a wishbone bus in the project.

5.1 Potato RISC-V Architecture

In the Potato RISC-V project, RV32I instruction set is implemented. Project includes peripherals such as UART, GPIO, Timer, ROM and RAM. These peripherals are interconnected with each other with Wishbone B4 Bus Interface [27].

5.2 Setup

A Vivado [28] project is created by following the tutorial in the github repository of the project [27]. The top module of potato has two UART, a GPIO and two timer modules. To create the project, documents inside the src/, soc/ and example/ directories must be added to a Vivado project. Then also, a clock generator and PAEE ROM IPs must be added to project hierarchy. Clock generator outputs should be set as 10 MHz "timer_clk" and 50 MHz "system_clk". Also, reset and locked signals should be enabled and reset should be selected as "active low". After that, PAEE ROM should be added as Block Memory IP and should be configured as Single-Port ROM and named as "aee_rom". Port A should be configured with "width : 32" and "depth : 4096". Then, always enabled should be selected.

"aee_rom" includes the boot code of the core and a .coe memory file should be added into it. This .coe file can be obtained from "potato-master/software/bootloader" directory with the Makefile. "make all" command should be ran in this directory to create the .coe file. The bootloader code that is added as first time example in the project, sends opening message and waits for a 128 KB which will be your input program.

With using a Serial Communication terminal, a .bin file should be sent to the core with using 115200 baud, 8N1 configurations. The .bin document can be sent to the core via an USB connection.

The code that is uploaded to the core can be seen in Figure 5.1.

```
// hello/main.c
#include <stdint.h>
#include "platform.h"
#include "uart.h"

static struct uart uart0;

void exception_handler(uint32_t cause, void * epc, void * regbase)
{
     // Not used in this application
}

int main(void)
{
     uart_initialize(&uart0, (volatile void *) PLATFORM_UART0_BASE);
     uart_set_divisor(&uart0, uart_baud2divisor(115200,
PLATFORM_SYSCLK_FREQ));
     uart_tx_string(&uart0, "Hello world\n\r");
     return 0;
}
/////////////////////////////////////////////////////////////////////
```

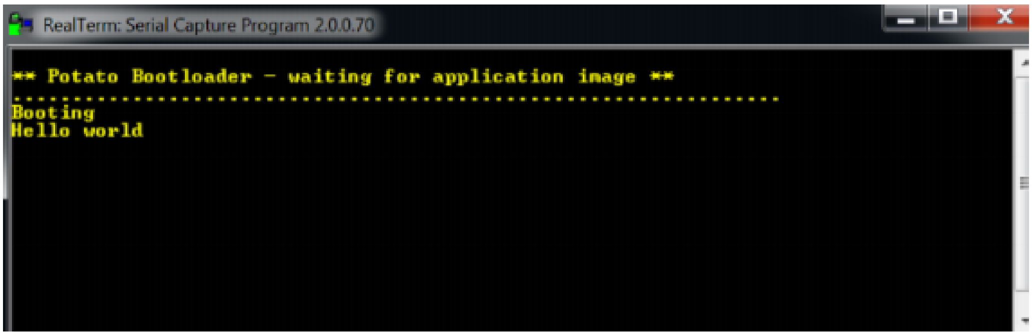

Figure 5.1: Potato UART Test code

The bootloader program simply reads 128KB of data from the UART and fills the main memory with this data then jump there. It is used to load the core with an application. Since the application code is usually smaller than 128KB, the file is extended with zeros until it is 128KB and following code can be used to complete your .bin file to 128 KB. **$ cat hello.bin /dev/zero | head -c131072 > hello_new.bin** .

The instruction cache of potato core is disabled due to a bug. After testing that the core works correctly with the hello world example program, we tested the core performance with NTRU post-quantum crypto algorithm [25]. The timer module is used to count the amount of clock cycles it takes to run the algorithm. Algorithm runtime cycles can be seen in Table 5.1.

Table 5.1: NTRU PQC algorithm runs on different platforms

	32-bit Algorithm Runtime	48-bit Algorithm Runtime
Potato Core	788,173,585	2,415,247,120
PULPino Core	52,489,786	-

Since Potato core includes only the base integer instruction set, it is not great performance wise. NTRU algorithms with longer key length do not work properly and get stuck during execution. This may be caused by a stack overflow.

6. Ibex

6.1 Introduction

After inspecting various RISC-V cores, we have decided to continue with Ibex since the source code of Ibex is easy to understand and edit. Ibex implements the RV32IMC instruction set. This includes 32-bit base integer set (I), standard multiplication extension (M), and compressed instructions extension (C). In this project, compressed instructions are not used. Ibex does not have any bus interface with peripherals by default. It only has connections to data memory and instruction memory. In order to easily add peripherals and run tests on FPGA, a version of Ibex with wishbone bus called ibex_wb is used [29]. ibex_wb comes with an example implementation for Xilinx Arty A7-100: Artix-7 FPGA [30] development board so, that implementation is used as a basis.

6.2 Vivado Project

A Vivado project is created for ibex_wb by simply adding the required modules from the repository. Unfortunately, ibex_wb does not include a detailed manual so, to find required files, first the top modules is added, then rest of the missing files are added according to hierarchy. The following is a list of all of the required files' paths relative to the project repository:

The example top module included in the project has a 25MHz clock generator and as peripheral there is only a wishbone LED module. The constraints in xdc file is updated for Nexys 4 DDR board [24]. The single port RAM module is edited to take memory initialization file name as a parameter to easily initialize the memory with program instructions.

For the software part, the RISC-V GNU Toolchain provided by lowRISC is used to compile and link the code written in C language. The machine code is generated with

Table 6.1: ibex_wb source file list

pulpino/rtl/components/pulp_clock_mux2.sv	common_cells/src/cdc_2phase.sv
riscv-dbg/debug_rom/debug_rom.sv	common_cells/src/deprecated/fifo_v2.sv
riscv-dbg/src/dm_top.sv	common_cells/src/fifo_v3.sv
riscv-dbg/src/dm_csrs.sv	ibex/rtl/ibex_core.sv
riscv-dbg/src/dm_sba.sv	ibex/rtl/ibex_if_stage.sv
riscv-dbg/src/dm_mem.sv	ibex/rtl/ibex_prefetch_buffer.sv
riscv-dbg/src/dmi_cdc.sv	ibex/rtl/ibex_fetch_fifo.sv
riscv-dbg/src/dmi_jtag.sv	ibex/rtl/ibex_compressed_decoder.sv
riscv-dbg/src/dmi_jtag_tap.sv	ibex/rtl/ibex_id_stage.sv
rtl/wb_ibex_core.sv	ibex/rtl/ibex_register_file_ff.sv
rtl/core2wb.sv	ibex/rtl/ibex_decoder.sv
rtl/slave2wb.sv	ibex/rtl/ibex_controller.sv
rtl/wb_dm_top.sv	ibex/rtl/ibex_ex_block.sv
soc/common/rtl/wb_interconnect_sharedbus.sv	ibex/rtl/ibex_alu.sv
soc/fpga/arty-a7-100/rtl/ibex_soc.sv	ibex/rtl/ibex_multdiv_fast.sv
soc/fpga/arty-a7-100/rtl/crg.sv	ibex/rtl/ibex_multdiv_slow.sv
soc/fpga/arty-a7-100/rtl/sync_reset.sv	ibex/rtl/ibex_load_store_unit.sv
soc/fpga/arty-a7-100/rtl/wb_spramx32.sv	ibex/rtl/ibex_cs_registers.sv
soc/fpga/arty-a7-100/rtl/spramx32.sv	ibex/rtl/ibex_pmp.sv
soc/fpga/arty-a7-100/rtl/wb_led.sv	ibex/shared/rtl/fpga/xilinx/prim_clock_gating.sv
pulpino/rtl/components/cluster_clock_inverter.sv	

the makefile provided in the project and it is then converted to memory file (text file with each instruction written as hexadecimal) with a script. This memory file is used for initializing the ram. It is added to the Vivado project and then bitstream for the project is generated in order to test it on the FPGA board. The example code simply makes the LEDs blink with a delay.

6.3 Peripherals

After confirming that the core works fine on the board, we have started to edit the project and add peripherals. In the top module, size and addresses of the peripherals are changed into an enum style to modify them in an easier way as seen in Figure 6.1. To put it simple, there is an enum for masters and another enum for slaves. The size and address of a slave are stored in separate arrays at the index of enum corresponding to that slave. The peripherals are accessed by either reading from or writing to the registers of the peripheral so, each peripheral should be given enough size (address space) to address all the registers contained in that peripheral.

```
typedef enum {
  DM_M,
  COREI_M,
  CORED_M
} wb_master_e;

typedef enum {
  DM_S,
  RAM_S,
  GPIO0_S,
  GPIO1_S,
  UART_S,
  TIMER_S
} wb_slave_e;

localparam NrMaster = 3;
localparam NrSlave = 6;

localparam [31:0] wb_base_addr [NrSlave] = {
  'h1A110000, //DMS
  'h00000000, //RAM
  'h10000000, //GPIO0
  'h10000010, //GPIO1
  'h10010000, //UART
  'h10020000  //TIMER
};

localparam [31:0] wb_size [NrSlave] = {
  'h10000, //DMS
  'h80000, //RAM
  'h00010, //GPIO0
  'h00010, //GPIO1
  'h00010, //UART
  'h00010  //TIMER
};

wb_if wbm[NrMaster](.*);
wb_if wbs[NrSlave](.*);
```

Figure 6.1: HDL code for peripheral enums.

Wishbone interfaces for each master and slave are also stored in arrays which can also be indexed by using enums. This makes adding, removing, and editing peripherals much simpler. Module instantiations can be seen in Figure 6.2.

The size of the RAM is increased from the default, since it might be required for later applications. This has caused some problems about the clock signal so, the 25MHz clock generator (soc/fpgs/arty-a7-100/rtl/crg.sv) that comes with ibex_wb is changed with the 50MHz clock generator (ibex/shared/rtl/fpga/xilinx/clkgen_xil7series.sv) which used in original Ibex. ibex_wb project also includes a debug module. Since

```
wb_interconnect_sharedbus #(
  .numm      (NrMaster),
  .nums      (NrSlave),
  .base_addr (wb_base_addr),
  .size      (wb_size)
) wb_intercon (.*);

wb_spramx32 #(
  .size(wb_size[RAM_S]),
  .init_file("bootloader.mem")
) wb_spram (
  .wb(wbs[RAM_S]));

wb_gpio #(
  .size (32)
) wb_gpio0 (
  .gpio (gpio0),
  .wb  (wbs[GPIO0_S]));

wb_gpio #(
  .size (27)
) wb_gpio1 (
  .gpio (gpio1),
  .wb  (wbs[GPIO1_S]));

wb_wbuart_wrap #(
  .HARDWARE_FLOW_CONTROL_PRESENT  (1'b0),
  .INITIAL_SETUP                  (31'd217),
  .LGFLEN                         (4'd6)
) wb_uart (
  .wb  (wbs[UART_S]),
  .i_uart_rx (uart_rx),
  .o_uart_tx (uart_tx),
  .i_cts_n   (1'b0));

wb_timer (
  .wb  (wbs[TIMER_S]));
```

Figure 6.2: HDL code for peripheral module instantiations.

it is not used in this project, the debug module is conditionally removed. It can be readded simply by defining the macro DEBUG_MODULE_ACTIVE.

6.3.1 GPIO

A wishbone GPIO module, whose HDL code can be seen in Figure 6.3, is added to the project in order to use LEDs, switches and buttons on the board . These are mostly used for debug purposes. GPIO module contains three registers: One for setting the pin directions (which pins are input and which pins are output), one for reading the inputs and one for setting the outputs.

```
module wb_gpio #(
    parameter size = 32
  )(
    inout wire [size-1:0] gpio,
    wb_if.slave           wb
  );

  logic [size-1:0] input_reg;
  logic [size-1:0] output_reg    = 'b0;
  logic [size-1:0] direction_reg = 'b0;

  for (genvar i = 0; i < size; i = i + 1) begin
    assign gpio[i] = direction_reg[i] ? output_reg[i] : 1'bz;
    assign input_reg[i] = direction_reg[i] ? 1'b0 : gpio[i];
  end

  logic valid;
  assign valid = wb.cyc && wb.stb;

  assign wb.stall = 1'b0;
  assign wb.err = 1'b0;

  always @(posedge wb.clk or posedge wb.rst)
    if (wb.rst) begin
      direction_reg <= '0;
      output_reg <= '0;
    end
    else if (valid)
      if (wb.we)
        case (wb.adr[3:0])
          4'h4 : output_reg    <= wb.dat_i[size-1:0];
          4'h8 : direction_reg <= wb.dat_i[size-1:0];
          default              : ;
        endcase
      else
        case (wb.adr[3:0])
          4'h0 : wb.dat_o <= input_reg;
          4'h4 : wb.dat_o <= output_reg;
          4'h8 : wb.dat_o <= direction_reg;
          default              : ;
        endcase

  always_ff @(posedge wb.clk or posedge wb.rst)
    if (wb.rst)
      wb.ack <= 1'b0;
    else
      wb.ack <= valid & ~wb.stall;
endmodule
```

Figure 6.3: HDL code for GPIO module.

A C library is written to use the GPIO module easily. The library is simple to use. First a gpio struct is initialized with the base address of GPIO module using gpio_init function, then the pin directions are set using gpio_set_direction function. Then the inputs and outputs can be controlled with gpio_get_input and gpio_set_output functions. To modify only a single pin, gpio_set_pin and gpio_clear_pin functions can be used.

6.3.2 Timer

Second, a wishbone timer module, whose HDL code can be seen in Figure 6.4, is added to keep track of performance in future tests. The timer module contains two registers: One holds the control bits while the other holds the count. Control register holds run and clear flags. When run flag is set, count register is incremented with every clock cycle. When clear flag is set, count register is cleared and the clear flag is reset. Clear flag has higher priority to run flag so when they are both set at the same time, first the count register is cleared, then it is continued to be incremented with the next clock.

```
module wb_timer(
    wb_if.slave wb
    );

    logic [31:0] control_reg = 'b0;
    logic [31:0] counter_reg = 'b0;

    logic valid;
    assign valid = wb.cyc && wb.stb;

    assign wb.stall = 1'b0;
    assign wb.err = 1'b0;

    always @(posedge wb.clk or posedge wb.rst)
        if (wb.rst) begin
            control_reg <= '0;
            counter_reg <= '0;
        end
        else begin
            if (valid)
                if (wb.we)
                    case (wb.adr[2:0])
                        3'h0 : control_reg <= wb.dat_i;
                        3'h4 : counter_reg <= wb.dat_i;
                        default : ;
                    endcase
                else
                    case (wb.adr[2:0])
                        3'h0 : wb.dat_o <= control_reg;
                        3'h4 : wb.dat_o <= counter_reg;
                        default : ;
                    endcase

            if (~(wb.we & valid))
                if (control_reg[1]) begin
                    counter_reg <= 'b0;
                    control_reg[1] <= 'b0;
                end
                else if (control_reg[0])
                    counter_reg <= counter_reg + 1;
        end

    always_ff @(posedge wb.clk or posedge wb.rst)
        if (wb.rst)
            wb.ack <= 1'b0;
        else
            wb.ack <= valid & ~wb.stall;

endmodule
```

Figure 6.4: HDL code for timer module.

Again, a C library is written for timer module as well. First a timer struct is initialized with the base address of timer module using timer_init function. timer_start starts the timer to increment. timer_stop stops the timer from incrementing and holds the current value in count register. timer_clear clears the count register of timer. timer_reset both stops and clears the timer. timer_get_count returns the current value in count register while time_set_count sets a desired value to the count register.

Later we have realized that ibex implements performance counters of RISC-V (control and status registers). So, instead of the external timer module, we decided to use the internal performance counters to keep track of clock cycles.

6.3.3 UART

Lastly a wishbone UART module is added to establish communication between computer and ibex core. The wishbone UART module is taken from wbuart32 [31] project. A wrapper module which can be seen in Figure 6.5 is written to make port connections and addresses fit with the current ibex_wb project. With the UART module it is possible to get text output or input with a serial port communication program. Minicom [32] is used for that purpose. This UART module contains four registers: SETUP, FIFO, RX_DATA, and TX_DATA. SETUP register holds the baudrate and some other configuration flags which are unused in this project. FIFO register is a read-only register which holds status flags for both RX and TX FIFOs. RX register is used for reading the received data and TX register is used for transmitting data.

A C library is written to easily send and receive data via UART communication. A uart struct instance is initialized with the base address of UART module by uart_init function. Before actually using the UART module, the baudrate must be set using uart_set_baudrate function. For more detailed configurations using the other flags in SETUP register, uart_configure function can be used. This is not required in this case.

There are several functions to use for transmitting data. uart_tx function transmits a single byte, but uart_tx_ready function must be used to check if there is any space TX FIFO. In order to send an entire string, uart_tx_string is used which internally checks if the TX FIFO is available. If the FIFO is full, then the function waits until a space is cleared. There is also a uart_printf function which transmits a formatted string. Since

```
module wb_wbuart_wrap #(
    parameter [30:0]  INITIAL_SETUP = 31'd25,
    parameter [3:0]   LGFLEN = 4,
    parameter [0:0]   HARDWARE_FLOW_CONTROL_PRESENT = 1'b1
  )(
    wb_if.slave wb,

    input  wire i_uart_rx,
    output wire o_uart_tx,

    input  wire i_cts_n,
    output wire o_rts_n,
    output wire o_uart_rx_int,
    output wire o_uart_tx_int,
    output wire o_uart_rxfifo_int,
    output wire o_uart_txfifo_int
  );

  logic [1:0] addr;

  always_comb
    case (wb.adr[3:0])
      4'h0: addr = 2'b00;
      4'h4: addr = 2'b01;
      4'h8: addr = 2'b10;
      4'hc: addr = 2'b11;
      default: addr = 2'b01;
    endcase

  wbuart #(
    .INITIAL_SETUP                  (INITIAL_SETUP),
    .LGFLEN                         (LGFLEN),
    .HARDWARE_FLOW_CONTROL_PRESENT  (HARDWARE_FLOW_CONTROL_PRESENT)
  ) wbuart (
    .i_clk              (wb.clk),
    .i_rst              (wb.rst),
    .i_wb_cyc           (wb.cyc),
    .i_wb_stb           (wb.stb & wb.cyc),
    .i_wb_we            (wb.we),
    .i_wb_addr          (addr),
    .i_wb_data          (wb.dat_i),
    .o_wb_ack           (wb.ack),
    .o_wb_stall         (wb.stall),
    .o_wb_data          (wb.dat_o),

    .i_uart_rx          (i_uart_rx),
    .o_uart_tx          (o_uart_tx),

    .i_cts_n            (i_cts_n),
    .o_rts_n            (o_rts_n),
    .o_uart_tx_int      (o_uart_tx_int),
    .o_uart_rx_int      (o_uart_rx_int),
    .o_uart_txfifo_int  (o_uart_txfifo_int),
    .o_uart_rxfifo_int  (o_uart_rxfifo_int)
  );

  assign wb.err = 1'b0;

endmodule
```

Figure 6.5: HDL code for UART wrapper module.

the standard printf function is too big for small systems like this, a smaller, lightweight version of printf is taken from PULPino project and integrated with the UART module.

Similar to transmit functions, there are also several functions for receiving data. uart_rx reads and returns a single byte. uart_rx_ready must be used to check if there is any available data in the RX FIFO. uart_rx_line keeps reading data until a newline character is read. It internally checks if there is any data in the FIFO and waits if there is not. The received data is copied to a string and the newline at the end is replaced with a null character. The function returns the amount of bytes received including the newline character.

6.4 Utilities

A small library called "utils" is written for general utility functions. This library includes a sleep function, several functions to access performance counters using inline assembly code, and a function to convert clock cycles to microseconds.

6.5 Bootloader

A problem for the development of this project was that it takes too much time to generate the bitstream file and it has to be generated again, using a new memory file, every time there is a simple change in the software. In order to eliminate this delay and make the development faster, a simple bootloader program is written inspired by the bootloader from Potato project. Then the memory file of bootloader is generated which is used for initializing the RAM. When the core starts working, bootloader awaits data from UART. The machine code of the program is then sent via UART as a binary file. Bootloader reads the machine code from UART and copies it to the RAM. Once all the machine code is copied, program jumps to the address with the new code. This configuration makes development much faster since there is no need to generate bitstream again and again unless there is a change in the hardware.

The bootloader was first designed to read a fixed amount of data from UART to fill all of the remaining space in the RAM. Since most of the time the binary file generated from the code is much smaller than the size of remaining RAM, the binary file is padded with zeros to the RAM size. At this point, the RAM size to be used by the program was set to 128KB. In order to pad the binary file with zeros, the following recipe is added to the makefile.

```
%_128k.bin: %.bin
	cat $(PROGRAM).bin /dev/zero | head -c128k > $(PROGRAM)_128k.bin
```

Then the padded binary file can be sent to the device via UART with the following command in terminal.

```
cat $(PROGRAM)_128k.bin > /dev/ttyUSB1
```

Here ttyUSB1 may be different depending on other USB serial port adapters connected to the computer and $(PROGRAM) is the name of the C file with the main function.

The problem with this approach is that the binary code is usually much smaller than the RAM but the RAM needs to stay larger since the remaining memory is going to be used as stack by the program. It still takes a long time to send a big binary file each time a code needs to be loaded to the device. In order to eliminate that delay, bootloader is modified to first get the size of the binary file from UART, then read that many bytes and copy them to the RAM. Once all the binary code is copied to the RAM, the remaining space is filled with zeros by the bootloader. This significantly improves the time required to load the device with new code. To automate this process a small script is written called sendapp.sh. This script first sends the file size as a string, then sends the file itself.

```
#!/bin/sh
# Script for sending image to Ibex bootloader.
filename=$1
imagesize=$(wc -c < $filename)
echo $imagesize > /dev/ttyUSB1
cat $filename > /dev/ttyUSB1
```

The script is then added to the makefile.

```
SENDAPP = ../../../scripts/sendapp.sh
run: $(SENDAPP) $(PROGRAM).bin
```

Now the device can be loaded with a new binary image by simply typing **make run** at the terminal.

6.6 Makefile For Software

Several modifications are made on the makefile supplied with with the project. Since the libraries can be used commonly by separate applications, they are stored in a different directory then the main source code of the application. In order to compile and link the libraries, source file paths and include directories must be added to the makefile.

SRCS = $(PROGRAM).c $(wildcard ../../libs/soc/*.c)
INCS = -I../../libs/soc

The memory file extension is changed from .vmem to .mem since this is the default memory file extension used in Vivado.

CC, OBJCOPY, and OBJDUMP variables are written in a more compact way. This is not mandatory, but makes the file more readable.

PREFIX ?= riscv32-unknown-elf
CC := $(PREFIX)-gcc
OBJCOPY := $(PREFIX)-objcopy
OBJDUMP := $(PREFIX)-objdump

In order to remove unused library functions from the application image, following flags are added. // Compile flags (CFLAGS): -ffunctions-sections -fdata-sections // Linker flags (LDFLAGS): -Wl,–gc-sections

-Wl,–Map,$(PROGRAM).map linker flag is added to output the .map file which shows how the memory is mapped.

6.7 Linker Scripts

The default linker script must be modified for the bootloader configuration and increased RAM size. For linking the bootloader application, two memory sections are defined: rom and stack, 16KB each. Those sections will take up the first 32KB of the RAM. For the rest of the application, which will be loaded to the device via bootloader, only one memory section is defined: ram. This sections starts at 64K and has a length of 448KB which fills the rest of the RAM. This section will be used for both storing the application image and as stack. The memory between 32K and 64K is left unused intentionally to create a bumper between bootloader and application in case of any unexpected behaviours.

6.8 NTRU

After all of the modules and bootloader are tested, development and testing for the post quantum crypto algorithms has started. First NTRU algorithm is tested on Ibex as well. In order to measure the performance, the mtime register is used. The performance counts are reset and started, NTRU algorithm runs, then mtime is stopped and the clock cycle count is read. Which is then transmitted to the computer via UART and printed on terminal using Minicom.

32-bit NTRU algorithm took 105,855,607 clock cycles to complete.

48-bit NTRU algorithm took 320,417,348 clock cycles to complete.

64-bit NTRU algorithm took 365,122,489 clock cycles to complete.

It is seen that the performance is much better compared to potato core.

6.9 NewHope

NewHope algorithm NIST submission package is obtained from NewHope official website [15]. From the optimized implementations, newhope512cca [**?**] is used for this project. Almost all of the algorithm is implemented using standard C libraries except the random number generation. Since we are not concerned with random number generation method, in order to simplify things, OpenSSL library [33] is removed and rand function from stdlib.h is used instead. This is made by conditional compiling using the preprocessor commands. The rand function is used when SIMPLE_RNG macro is defined. It can be defined by adding -DSIMPLE_RNG flag to PROGRAM_CFLAGS in the makefile. No changes are made to the code other than this. The source file paths and include directories should be added to the makefile with the following code:

SRCS = $(PROGRAM).c $(wildcard ../../libs/soc/*.c) $(wildcard ../../libs/newhope512cca/*.c)

INCS = -I../../libs/soc -I../../libs/newhope512cca

There are three important functions to use NewHope algorithm for key encapsulation mechanism (KEM): crypto_kem_keypair, crypto_kem_enc, crypto_kem_dec. These functions are provided by the NewHope library. crypto_kem_keypair generates a public and private key pair, crypto_kem_enc generates cypher text and shared secret for the given public key, and crypto_kem_dec generates shared secret for given cipher text and private key. Once the key is safely excahnged, the communication can be established by any method using the previously exchanged key.

For the test in this project, all three functions (keypair, enc, dec) are called and the amount of clock cycles passed is read from performance counters while also checking whether the functions work correctly. To verify that the functions work correctly, returns values are checked, and shared secrets generated by encapsulation and decapsulation methods are compared. If all the return values are zero and the shared secrets match, it indicates that the functions work correctly. Once the performance counters are read, the results are then transmitted to computer via UART and printed on terminal using Minicom.

At the beginning the codes were compiled with -Os optimization flag. This was the default in the makefile provided by ibex_wb project. Upon trying other optimization flags, we have noticed that -O1 flag gives the best performance result in this case. The clock counts and code sizes with all optimization flags are given on Table 6.2.

Table 6.2: Compiling results with different GCC flags

Optimization	-Os	-O0	-O1	-O2	-O3
Clock Count	8,664,162	29,628,211	7,373,240	11,902,160	11,283,484
Code Size	20,580	36,124	21,460	25,500	44,356

Upon these results, we have decided to use -O1 optimization for further tests.

6.9.1 Extending the ISA

6.9.1.1 Profiling

In order to further analyze the code and do the profiling of the function calls, we have used both GNU Debugger (GDB) and static counters at C code. With GDB we have used breakpoints on every function and tracked the number of function calls through these breakpoints and with also internal static counters we have confirmed our profiling results.

GNU Compiler Collection (GCC) project has already a support for RISC-V instructions but to be able to compile the new custom instructions with GCC, adding new instruction definitions to the GCC source code were needed.To do this, source code of GCC has been inspected and opcode related codes have been discovered.

6.9.1.2 Adding Custom Instruction Hardware

There are two kinds of extensions in RISC-V architecture: greenfield and brownfield extensions. Greenfield extensions use a new encoding space which means they should have new opcodes. Brownfield extensions on the other hand, use existing encoding spaces. For example the opcode for register-register operations have a lot of available slots on the funct3/funct7 spaces. This does not have any time performance effect. The difference is about conflicts with different extensions. Since this is not a priority for this project, brownfield extensions are used.

For the extensions both ALU and the instruction decoder must be modified. In Ibex, ALU result is selected using an enum defined in ibex_pkg.sv for the ALU operator. This enum must be extended with the new instruction. The circuit which calculates the instruction result is added in ALU and the result is added to the result multiplexer as seen in Figure 6.6.

```
////////////////
// Result mux //
////////////////

always_comb begin
  result_o   = '0;

  unique case (operator_i)
    // Standard Operations
    ALU_AND:  result_o = operand_a_i & operand_b_i;
    ALU_OR:   result_o = operand_a_i | operand_b_i;
    ALU_XOR:  result_o = operand_a_i ^ operand_b_i;

    // Adder Operations
    ALU_ADD, ALU_SUB: result_o = adder_result;

    // Shift Operations
    ALU_SLL,
    ALU_SRL, ALU_SRA: result_o = shift_result;

    // Comparison Operations
    ALU_EQ,   ALU_NE,
    ALU_GE,   ALU_GEU,
    ALU_LT,   ALU_LTU,
    ALU_SLT,  ALU_SLTU: result_o = {31'h0,cmp_result};

    // Custom Operations
    ALU_CUST0: result_o = cust0_result;
    ALU_CUST1: result_o = cust1_result;
    ALU_CUST2: result_o = cust2_result;

    default:;
  endcase
end
```

Figure 6.6: HDL code of result multiplexer in ALU module.

The extensions must be added to decoder too. We have decided to use brownfield extensions to register-register opcode. There is no need to edit any control signals except ALU operator. Under the case for funct3 and funct7, custom instructions are added with corresponding operators as seen in Figure 6.7. Five custom extensions are added in advance in case it is needed for future use.

```
OPCODE_OP: begin  // Register-Register ALU operation
  alu_op_a_mux_sel_o = OP_A_REG_A;
  alu_op_b_mux_sel_o = OP_B_REG_B;
  regfile_we         = 1'b1;

  if (instr[31]) begin
    illegal_insn = 1'b1;
  end else begin
    unique case ({instr[30:25], instr[14:12]})
      // RV32I ALU operations
      {6'b00_0000, 3'b000}: alu_operator_o = ALU_ADD;   // Add
      {6'b10_0000, 3'b000}: alu_operator_o = ALU_SUB;   // Sub
      {6'b00_0000, 3'b010}: alu_operator_o = ALU_SLT;   // Set Lower Than
      {6'b00_0000, 3'b011}: alu_operator_o = ALU_SLTU;  // Set Lower Than Unsigned
      {6'b00_0000, 3'b100}: alu_operator_o = ALU_XOR;   // Xor
      {6'b00_0000, 3'b110}: alu_operator_o = ALU_OR;    // Or
      {6'b00_0000, 3'b111}: alu_operator_o = ALU_AND;   // And
      {6'b00_0000, 3'b001}: alu_operator_o = ALU_SLL;   // Shift Left Logical
      {6'b00_0000, 3'b101}: alu_operator_o = ALU_SRL;   // Shift Right Logical
      {6'b10_0000, 3'b101}: alu_operator_o = ALU_SRA;   // Shift Right Arithmetic

      // Custom instructions
      {6'b1100_00, 3'b000}: alu_operator_o = ALU_CUST0; // Custom 0
      {6'b1100_01, 3'b000}: alu_operator_o = ALU_CUST1; // Custom 1
      {6'b1100_10, 3'b000}: alu_operator_o = ALU_CUST2; // Custom 2
      {6'b1100_11, 3'b000}: alu_operator_o = ALU_CUST3; // Custom 3
      {6'b1101_00, 3'b000}: alu_operator_o = ALU_CUST4; // Custom 4
```

Figure 6.7: HDL code for added custom extension in instruction decoder.

6.9.1.3 Custom Instruction 0: Hamming Weight Difference

The first custom instruction added is Hamming weight difference which is used in poly.c. Inside this library there is a function called hw which takes the Hamming weight of a value. Hamming weight is the number of set bits in a binary number. This hw function is called twice in poly_sample function with different values and then the difference of weights is calculated.

The added instruction takes two values from registers, counts the amount of set bits in each of them and takes the difference of those amounts then writes the result to a register. This is used in polynomial sampling. The hardware is implemented with look-up tables and adders. First a look-up table with 4-bit control input is designed to count the number of set bits in a 4-bit input as seen in Figure 6.8.

```
module hw_lut(
    input  logic [3:0] data,
    output logic [2:0] count
  );

  always_comb
    case (data)
      4'b0000 : count = 3'b000;
      4'b0001 : count = 3'b001;
      4'b0010 : count = 3'b001;
      4'b0011 : count = 3'b010;
      4'b0100 : count = 3'b001;
      4'b0101 : count = 3'b010;
      4'b0110 : count = 3'b010;
      4'b0111 : count = 3'b011;
      4'b1000 : count = 3'b001;
      4'b1001 : count = 3'b010;
      4'b1010 : count = 3'b010;
      4'b1011 : count = 3'b011;
      4'b1100 : count = 3'b010;
      4'b1101 : count = 3'b011;
      4'b1110 : count = 3'b011;
      4'b1111 : count = 3'b100;
      default : ;
    endcase
endmodule
```

Figure 6.8: HDL code of look-up table for Hamming weight.

The 32-bit values from registers are separated to 4-bit slices and each slice is connected to a look-up table to count the number of set bits. Then the results are added in pairs until the total number of set bits in each 32-bit value is obtained which can be seen in Figure 6.9. Finally, the difference of those amounts is calculated with a subtractor. In order to use less space, each adder/subtractor has minimum width input/output. Normally newhope512cca library takes the Hamming weight of only 8-bit values, but in order to make it a more general instruction, it is designed for 32-bit values in hardware.

```
//////////////
// Custom 0 //
//////////////

logic [31:0] cust0_result;
logic [32:0] cust0_result_ext;
logic [5:0] cust0_count_a;
logic [5:0] cust0_count_b;
logic [32:0] cust0_count_a_ext;
logic [32:0] cust0_count_b_ext;

logic [7:0][2:0] cust0_lut_counts_a;
logic [7:0][2:0] cust0_lut_counts_b;
logic [3:0][3:0] cust0_lut_add1_a;
logic [3:0][3:0] cust0_lut_add1_b;
logic [1:0][4:0] cust0_lut_add2_a;
logic [1:0][4:0] cust0_lut_add2_b;

generate
  for (genvar i = 0; i < 8; i = i + 1) begin : hw_luts
    hw_lut (
      .data (operand_a_i[4*i+3:4*i]),
      .count (cust0_lut_counts_a[i])
    );

    hw_lut (
      .data (operand_b_i[4*i+3:4*i]),
      .count (cust0_lut_counts_b[i])
    );
  end

  for (genvar i = 0; i < 8; i = i + 2) begin : hw_luts_add1
    assign cust0_lut_add1_a[i/2] = {1'b0, cust0_lut_counts_a[i]} + {1'b0, cust0_lut_counts_a[i+1]};
    assign cust0_lut_add1_b[i/2] = {1'b0, cust0_lut_counts_b[i]} + {1'b0, cust0_lut_counts_b[i+1]};
  end

  for (genvar i = 0; i < 4; i = i + 2) begin : hw_luts_add2
    assign cust0_lut_add2_a[i/2] = {1'b0, cust0_lut_add1_a[i]} + {1'b0, cust0_lut_add1_a[i+1]};
    assign cust0_lut_add2_b[i/2] = {1'b0, cust0_lut_add1_b[i]} + {1'b0, cust0_lut_add1_b[i+1]};
  end
endgenerate

assign cust0_count_a = {1'b0, cust0_lut_add2_a[0]} + {1'b0, cust0_lut_add2_a[1]};
assign cust0_count_b = {1'b0, cust0_lut_add2_b[0]} + {1'b0, cust0_lut_add2_b[1]};

assign cust0_count_a_ext[32:0] = {26'b0, cust0_count_a, 1'b1};
assign cust0_count_b_ext[32:0] = ~{26'b0, cust0_count_b, 1'b0};
assign cust0_result_ext = $unsigned(cust0_count_a_ext) + $unsigned(cust0_count_b_ext);
assign cust0_result = cust0_result_ext[32:1];
```

Figure 6.9: HDL code of custom instruction 0 in ALU module.

The custom instruction is used in the C code with inline assembly code. It is added with conditional compiling as seen in Figure 6.10. If the macro BAM_CUST0 is defined, then the custom instruction is used, otherwise the standard instructions are used. This definition is added to the makefile with the flag -DBAM_CUST0 to PROGRAM_CFLAGS.

6.9.1.4 Custom Instruction 1: coeff_freeze

The second custom instruction is a part of the coeff_freeze function in poly.c. It consists of a subtraction followed by a series of logic operations as seen in the Figure 6.11. For the subtraction the adder/subtractor that already exists in the ALU is used. In the software, right shift is used to fill the value with sign bit. In hardware instead of shifting, the sign bit is directly copied.

```c
void poly_sample(poly *r, const unsigned char *seed, unsigned char nonce)
{
#if NEWHOPE_K != 8
#error "poly_sample in poly.c only supports k=8"
#endif
  unsigned char buf[128], a, b;
//  uint32_t t, d, a, b, c;
  int i,j;

  unsigned char extseed[NEWHOPE_SYMBYTES+2];

  for(i=0;i<NEWHOPE_SYMBYTES;i++)
    extseed[i] = seed[i];
  extseed[NEWHOPE_SYMBYTES] = nonce;

  for(i=0;i<NEWHOPE_N/64;i++) /* Generate noise in blocks of 64 coefficients */
  {
    extseed[NEWHOPE_SYMBYTES+1] = i;
    shake256(buf,128,extseed,NEWHOPE_SYMBYTES+2);
    for(j=0;j<64;j++)
    {
      a = buf[2*j];
      b = buf[2*j+1];

#ifdef BAM_CUST0
      asm volatile (
                "cust0 %[result1], %[value1], %[value2]\n\t"
                : [result1] "=r" (r->coeffs[64 * i + j])
                : [value1] "r" (a), [value2] "r" (b)
                );

      r->coeffs[64 * i + j] += NEWHOPE_Q;
#else
      r->coeffs[64*i+j] = hw(a) + NEWHOPE_Q - hw(b);
#endif
    }
  }
}
```

Figure 6.10: C code of usage of custom instruction 0.

```systemverilog
//////////////
// Custom 1 //
//////////////

logic [31:0] cust1_m;
logic [31:0] cust1_c;
logic [31:0] cust1_result;

assign cust1_m = adder_result;
assign cust1_c = {32{cust1_m[31]}};
assign cust1_result = cust1_m ^ ((operand_a_i ^ cust1_m) & cust1_c);
```

Figure 6.11: HDL code of custom instruction 1 in ALU module.

Again, as seen in Figure 6.12, the custom instruction is added as inline assembly code to the software with conditional compiling using he flag -DBAM_CUST1.

```c
static uint16_t coeff_freeze(uint16_t x)
{
  uint16_t m, r;
  int16_t c;
  r = x % NEWHOPE_Q;

#ifdef BAM_CUST1
  asm volatile (
          "cust1 %[result1], %[value1], %[value2]\n\t"
          : [result1] "=r" (r)
          : [value1] "r" (r), [value2] "r" (NEWHOPE_Q)
          );
#else
  m = r - NEWHOPE_Q;
  c = m;
  c >>= 15;
  r = m ^ ((r^m) & c);
#endif

  return r;
}
```

Figure 6.12: C code of usage of custom instruction 1.

6.9.1.5 Custom Instruction 2: flipabs

The third custom instruction is a part of flipabs function in poly.c. Similar to previous one, it consists of a subtraction, copying the sign bit, addition and XOR operation as seen in Figure 6.13. The subtractor in ALU is used for the initial subtraction.

```systemverilog
//////////////
// Custom 2 //
//////////////

logic [31:0] cust2_result;
logic [31:0] cust2_r;
logic [31:0] cust2_m;

assign cust2_r = adder_result;
assign cust2_m = {32{cust2_r[31]}};
assign cust2_result = (cust2_r + cust2_m) ^ cust2_m;
```

Figure 6.13: HDL code of custom instruction 2 in ALU module.

Again, as seen in Figure 6.14, the custom instruction is added as inline assembly code to the software with conditional compiling using he flag -DBAM_CUST2.

```c
static uint16_t flipabs(uint16_t x)
{
  int16_t r, m;
  r = coeff_freeze(x);

#ifdef BAM_CUST2
  asm volatile (
            "cust2 %[result1], %[value1], %[value2]\n\t"
            : [result1] "=r" (r)
            : [value1] "r" (r), [value2] "r" (NEWHOPE_Q / 2)
            );
#else
  r = r - NEWHOPE_Q / 2;
  m = r >> 15;
  r = (r + m) ^ m;
#endif
  return r;
}
```

Figure 6.14: C code of usage of custom instruction 2.

6.9.1.6 Performance Improvements

Table 6.3 shows the performance and area changes with the added custom instructions.

Table 6.3: Performance and area changes with different custom instructions

	Without extensions	With Cust0	With Cust1	With Cust2	All together
Clock Cycles	7,374,191	6,882,431	7,360,469	7,371,121	6,865,637
Clock reduced	-	491,760	13,722	3,070	508,554
Clock reduced (%)	-	6.67%	0.18%	0.04%	6.90%
LUTs used	5,586	5,799	5,756	5,713	5,930
LUTs increased	-	213	170	127	344
LUTs increased (%)	-	3.81%	3.04%	2.27%	6.16%

7. REALISTIC CONSTRAINTS AND CONCLUSIONS

7.1 Practical Application of this Project

7.2 Realistic Constraints

7.2.1 Social, environmental and economic impact

Post-quantum cryptography algorithms has not been standardized yet. Therefore, its impact on social, environmental and economic figures has not been analyzed. But instruction extension for these algorithms increases the reliability of the quantum computing technology.

7.2.2 Cost analysis

In the project FPGA board and personal computers are used. In the Form 3, we have mentioned that the Nexys 4 DDR FPGA board is $265 and we have obtained that board from "Embedded Systems Design Laboratory".

7.2.3 Standards

In this project, Institute of Electrical and Electronics Engineers (IEEE) standard have been followed. Also, post-quantum cryptography algorithms are under development of standardization.

7.2.4 Health and safety concerns

This project is not classified as an risky project and behaviours of the project is mostly simulated.

7.3 Future Work and Recommendations

In the project, a post-quantum cryptography algorithm has been analyzed and an instruction extension have been created for it to work faster. New set of instructions was able to reduce the runtime by 10% but other than analyzing the C code of the algorithm with a full post-quantum cryptography block, runtime can be reduced more and it might became much faster. Thus, a new crypto algorithm block might be better to be considered. Also, with a much detailed analyze of the assembly code of the algorithm instruction extension might be expanded.

REFERENCES

[1] **PQSHIELD**, Lattices, `https://pqshield.com/site/wp-content/themes/PQShield/images/blog/lattice_bases.svg`.

[2] **Chuang, Y.Y.**, **Fan, C.I. and Tseng, Y.F.**, An Efficient Algorithm for the Shortest Vector Problem, `https://www.semanticscholar.org/paper/An-Efficient-Algorithm-for-the-Shortest-Vector-Chuang-Fan/cae8e359cc1b1abc7081d89d2e9224ad2847e41a/figure/0`.

[3] **Wong, D.**, Lattices and Tikz, `https://www.cryptologie.net/article/284/lattices-and-tikz/`.

[4] **Alkim, E.**, **Avanzi, R.**, **Bos, J.**, **Ducas, L.**, **de la Piedra, A.**, **Pöppelmann, T.**, **Schwabe, P.**, **Stebila, D.**, **Albrecht, M.R.**, **Orsini, E.**, **Osheter, V.**, **Paterson, K.G.**, **Peer, G. and Smart, N.P.**, (2020). NewHope Algorithm Specifications and Support Documentation V 1.1.

[5] **RISC-V Foundation**, (2019). The RISC-V Instruction Set Manual, 20190608-Priv-MSU-Ratified.

[6] **ETH Zurich**, PULPino, `https://github.com/pulp-platform/pulpino`.

[7] **Shor, P.W.** (1999). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, *SIAM Review*, *41*(2), 303–332, `https://doi.org/10.1137/S0036144598347011`, `https://doi.org/10.1137/S0036144598347011`.

[8] **Grover, L.K.** (1996). A fast quantum mechanical algorithm for database search, *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, `http://dx.doi.org/10.1145/237814.237866`.

[9] **National Institute of Standards and Technology**, NIST, `https://www.nist.gov/`.

[10] **Bertoni, A.**, **Mereghetti, C. and Palano, B.** (2003). Quantum Computing: 1-Way Quantum Automata, ***Z. Ésik and Z. Fülöp**, editors, Developments in Language Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp.1–20.

[11] **Ansari, R.**, **Tran Thanh Van, J. and Giraud-Heraud, Y.**, editors, (1996). Dark matter in cosmology, quantum measurements, experimental gravitation. Proceedings, 31st Rencontres de Moriond, 16th Moriond Workshop, Les Arcs, France, January 2-27, 1996, Ed. Frontieres, Gif-Sur-Yvette.

[12] **Bernstein, D.J.** (2010). Grover vs. McEliece, ***N. Sendrier**, editor, Post-Quantum Cryptography*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp.73–80.

[13] **National Institute of Standards and Technology**, NIST, `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`.

[14] **Infineon**, Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip, `https://www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html`.

[15] **Alkim, E.**, **Ducas, L.**, **Pöppelmann, T. and Schwabe, P.** (2015). Post-quantum Key Exchange - A New Hope, *IACR Cryptol. ePrint Arch.*, *2015*, 1092.

[16] **Micciancio, D. and Regev, O.**, (2009). Lattice-based Cryptography, Springer Berlin Heidelberg, Berlin, Heidelberg, pp.147–191, `https://doi.org/10.1007/978-3-540-88702-7_5`.

[17] **Alkim, E.**, **Avanzi, R.**, **Bos, J.**, **Ducas, L.**, **de la Piedra, A.**, **Pöppelmann, T.**, **Schwabe, P.**, **Stebila, D.**, **Albrecht, M.R.**, **Orsini, E.**, **Osheter, V.**, **Paterson, K.G.**, **Peer, G. and Smart, N.P.**, NewHope Official Website, `https://newhopecrypto.org/`.

[18] **Peeters, M.**, **Bertoni, G.**, **Daemen, J.**, **Keer, R.V.**, **Assche, G.V. and Hoffert, S.**, Keccak Specifications Summary, `https://keccak.team/keccak_specs_summary.html`.

[19] **Longa, P. and Naehrig, M.** (2016). Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography, ***S. Foresti and G. Persiano**, editors, Cryptology and Network Security*, Springer International Publishing, Cham, pp.124–139.

[20] **Braithwaite, M.**, (2016), Experimenting With Post Quantum, `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html`.

[21] **Langley, A.**, (2016), CECPQ1 Results, `https://www.imperialviolet.org/2016/11/28/cecpq1.html`.

[22] **RISC-V FOUNDATION**, `https://riscv.org/`.

[23] **Traber, A.**, **Gautschi, M. and Schiavone, P.D.**, (2019). RI5CY: User Manual, revision 4.0.

[24] **Digilent**, Nexys 4 DDR, `https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual`.

[25] **Chen, C.**, **Danba, O.**, **Hoffstein, J.**, **Hülsing, A.**, **Rijneveld, J.**, **Schanck, J.M.**, **Schwabe, P.**, **Whyte, W. and Zhang, Z.** (2019). Algorithm Specications And Supporting Documentation.

[26] **Perry, D.L.** (2002). *VHDL Programming By Example*, McGraw-Hill Professional.

[27] **Kristian Klomsten Skordal**, A simple RISC-V processor for use in FPGA designs, https://github.com/skordal/potato.

[28] **Xilinx**, (2020). Vivado Design Suite User Guide.

[29] **pbing**, ibex_wb, https://github.com/pbing/ibex_wb.

[30] **Digilent**, Arty A7, https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual.

[31] **Dan Gisselquist**, wbuart32, https://github.com/ZipCPU/wbuart32.

[32] **die.net**, minicom(1) - Linux man page, https://linux.die.net/man/1/minicom.

[33] **OpenSSL Software Foundation**, OpenSSL, https://www.openssl.org/.

APPENDICES

APPENDIX A.1 : Makefiles

APPENDIX A.1

```makefile
# Makefile to generate baremetal app

PROGRAM ?= newhope
PROGRAM_CFLAGS = -Wall -g -O1 -DSIMPLE_RNG -DBAM_CUST0 -DBAM_CUST1 -DBAM_CUST2
#ARCH = rv32imc
ARCH = rv32im
SRCS = $(PROGRAM).c $(wildcard ../../libs/soc/*.c) $(wildcard ../../libs/newhope512cca/*.c)
INCS = -I../../libs/soc -I../../libs/newhope512cca

PREFIX ?= riscv32-unknown-elf
CC := $(PREFIX)-gcc
OBJCOPY := $(PREFIX)-objcopy
OBJDUMP := $(PREFIX)-objdump

LINKER_SCRIPT ?= link.ld
CRT ?= crt0.S
CFLAGS ?= -march=$(ARCH) -mabi=ilp32 -static -mcmodel=medany \
	-fvisibility=hidden -ffunction-sections -fdata-sections \
	-nostdlib -nostartfiles $(PROGRAM_CFLAGS)
LDFLAGS ?= $(CFLAGS) -T $(LINKER_SCRIPT) -Wl,--gc-sections -Wl,--Map,$(PROGRAM).map

OBJS := ${SRCS:.c=.o} ${CRT:.S=.o}
DEPS = $(OBJS:%.o=%.d)

OUTFILES = $(PROGRAM).elf $(PROGRAM).mem $(PROGRAM).bin $(PROGRAM).dis

HEX2VMEM = ../../../scripts/hex2vmem.pl
SENDAPP = ../../../scripts/sendapp.sh

all: $(OUTFILES)

$(PROGRAM).elf: $(OBJS) $(LINKER_SCRIPT)
	$(CC) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)

%.dis: %.elf
	$(OBJDUMP) -SD $^ > $@

%.mem: %.hex
	$(HEX2VMEM) $< > $@

%.hex: %.elf
	$(OBJCOPY) -O verilog --interleave-width=4 --interleave=4 --byte=0 $< $@

%.bin: %.elf
	$(OBJCOPY) -O binary $< $@

%.o: %.c
	$(CC) $(CFLAGS) -c $(INCS) -o $@ $<

%.o: %.S
	$(CC) $(CFLAGS) -c $(INCS) -o $@ $<

clean:
	$(RM) -f $(OBJS) $(DEPS) *.hex

distclean: clean
	$(RM) -f $(OUTFILES) *.map

run:
	$(SENDAPP) $(PROGRAM).bin
```

Figure A.1: NewHope software makefile.

CURRICULUM VITAE

Name Surname: Ali Üstün

Place and Date of Birth: İstanbul, 28.08.1998

E-Mail: ustuna16@itu.edu.tr

EDUCATION:

- **B.Sc.:** Senior student, Istanbul Technical University, Electrical-Electronics Faculty, Department of Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2020 TUBITAK-BILGEM Part-time Researcher
- 2020 Embedded Systems Design Laboratory Internship
- 2019 TUBITAK-BILGEM Internsip
- 2017 Istanbul Technical Universty Solar Car Team - Embedded Systems
- 2016 Turkish National Physics Olympics

CURRICULUM VITAE

Name Surname: Batuhan ATEŞ

Place and Date of Birth: Istanbul, 17.10.1998

E-Mail: atesb16@itu.edu.tr

EDUCATION:

- **B.Sc.:** Senior student, Istanbul Technical University, Electrical-Electronics Faculty, Department of Electronics and Communication Engineering
- **B.Sc.:** Double major program student, Istanbul Technical University, Electrical-Electronics Faculty, Department of Control and Automation Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2020-present Internship at Borda Technology as Embedded Software Developer
- 2019 Internship at Istanbul Technical University Embedded System Design Laboratory

CURRICULUM VITAE

Name Surname: Musa Antike

Place and Date of Birth: Ankara 26.07.1995

E-Mail: antike16@itu.edu.tr

EDUCATION:

- **B.Sc.:** Senior Student, Istanbul Technical University, Electrical - Electronics Faculty, Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2017-2019 Istanbul Technical University - Solar Car Team - Embedded Systems.
- 2019-2020 Borda Technology - Part Time Embedded Software Developer
- 2020-Present Borda Technology - Embedded Software Developer