Efficiency of RNS in CSIDH

Performance Research In Post-Quantum Cryptography

Dorenbos, H.M.R.

Award date:
2019

Efficiency of RNS in CSIDH

Performance Research In Post-Quantum Cryptography

Author: H.M.R. Dorenbos
Supervisor: T. Lange
Co-Supervisor: L.S. Panny

Eindhoven University of Technology
Department of Mathematics and Computer Science

January 27, 2019

Abstract

The downfall of the internet approaches, because the strength of current asymmetric cryptographic algorithms will not last forever. Research to building quantum computers progresses, which will easily break all current asymmetric cryptographic algorithms. Therefore a new type of cryptographic algorithms is needed which do resist quantum computers but are still friendly to use by normal computer systems, such cryptographic algorithms are called: post-quantum cryptography. CSIDH is a post-quantum cryptographic algorithm that is not breakable by quantum computers but is still friendly to use by normal computers. In this report I will do research about increasing the performance of CSIDH which means minimizing the number of basic operations needed and minimizing the number of clock ticks for doing operations in CSIDH. Nevertheless security of CSIDH should never be at stake, which implies that the running time should not be dependent on the input. The approach in this report to optimize the performance of CSIDH will be to improve the performance of addition, subtraction and multiplication in CSIDH, since with only these operations we can do any operation in CSIDH. To optimize addition, subtraction and multiplication I will use RNS which is the abbreviation of the residue number system. RNS is an alternative method to represent numbers in which multiplication is very fast. However in RNS it is hard to do smaller/larger comparisons and modular reduction. Moreover there is no suitable residue number system for CSIDH. Therefore a different residue number system is taken in which we need to adjust addition, subtraction and multiplication to modular form. For modular addition and subtraction you can use the mixed radix system to do comparisons which translates addition and subtraction to modular form. And for modular multiplication we can use Montgomery multiplication in RNS. Implementation of these operations can be optimized by skipping the modular part of modular additions and subtractions, precomputation of values in the operations, selection of appropriate moduli and vector instructions which can do multiple basic computer operations at the same time. Unfortunately with these optimizations I did not manage to create a prototype RNS implementation that outperforms the previous implementation of the CSIDH operations. Nevertheless I do think that RNS can make these operations faster, because multiplication is very fast in RNS and RNS is very suitable for parallelization which can also be used to speed up mixed radix conversion and Montgomery multiplication.

Contents

1 Introduction **3**
- 1.1 Importance of Post-Quantum Cryptography . . . 3
- 1.2 Importance Efficient Cryptography . . . 3
- 1.3 Research in Post-Quantum Cryptography . . . 4

2 Definition of Performance **5**
- 2.1 Introduction to Computer Systems . . . 5
- 2.2 Efficiency of Computer Systems . . . 7
- 2.3 Definition . . . 8
- 2.4 Timing Attack . . . 9

3 Residue Number System **12**
- 3.1 Introduction to RNS . . . 12
- 3.2 Conversion to RNS . . . 14
- 3.3 Conversion from RNS . . . 15
- 3.4 Operations in RNS . . . 17
- 3.5 RNS in CSIDH . . . 18

4 RNS Modular Arithmetic **20**
- 4.1 Conversion Method . . . 20
- 4.2 Mixed Radix System for RNS . . . 22
- 4.3 Comparison in RNS . . . 24
- 4.4 Modular Addition in RNS . . . 25
- 4.5 Modular Subtraction in RNS . . . 26
- 4.6 Base Extension . . . 28
- 4.7 Exact Division . . . 29
- 4.8 Modular Multiplication in RNS . . . 33

5 Operation Implementation **35**
- 5.1 Efficient Implementation . . . 35
- 5.2 Combination of Modular Operations . . . 36
- 5.3 Precomputation . . . 37
- 5.4 Moduli Selection . . . 40
- 5.5 Efficient Usage of Computer Systems . . . 43

6 Performance of RNS **45**
6.1 Old Implementation of CSIDH 45
6.2 RNS Implementation of CSIDH 46

7 Conclusion **49**

Appendices **52**

A Basic Operations **52**

B RNS Moduli **53**

Chapter 1

Introduction

1.1 Importance of Post-Quantum Cryptography

The demise of the internet might be nearby, because the research to building quantum computers progresses. Therefore the internet becomes more at risk. The reason why the internet becomes more at risk is because current types of asymmetric cryptography will be easily cracked by quantum computers. For example asymmetric cryptographic algorithms like RSA and ECC are based on the hardness of prime factorization and the discrete logarithm problem, which are easier to solve by quantum computers. This means that every data that we have sent or will be send over the internet can be read by anyone having a quantum computer. Passwords, WhatsApp messages, e-mails, Dropbox files, etcetera will all be revealed to hackers. It might sound like a disaster, but there is a method to prevent further damage: post-quantum cryptography. Post-quantum cryptography are encryption algorithms designed to be friendly to use by normal computers, however are hard to crack by both normal computers and quantum computers. Hence encrypting data with such algorithms will prevent this data from being read by anyone having a quantum computer and therefore prevents the demise of the internet. Unfortunately data that was intercepted which is not encrypted by post-quantum cryptography is still readable, which is something that cannot be prevented. But communication in the future can still be secure.

1.2 Importance Efficient Cryptography

Security is not the only requirement of cryptographic algorithms. Efficiency is also a major requirement for cryptographic algorithms. The number of devices connected to the internet grows rapidly every year which must use cryptographic algorithms. New devices that will be connected to the internet includes servers, laptops and desktops, but also smaller devices like smartphones and even micro chips. The last group lacks the resources to do fast computation. Therefore

cryptographic operations should be efficient for these small devices. But efficient cryptography is also a must for large devices like servers that handle over a million requests every day. If decryption of messages takes too long then these servers will be overloaded which will cause high delays in internet communication. Even small delays can ruin applications completely, for example (live) video streaming and gaming applications. Performance is therefore a must for cryptographic algorithms, but can be a challenge for post-quantum cryptographic algorithms because these algorithms need more advanced mechanism unbreakable by quantum computers.

1.3 Research in Post-Quantum Cryptography

CSIDH is one of such post-quantum cryptographic algorithms [1]. In this report I will do research about improving the performance of CSIDH with RNS, which is an alternative notation of numbers. Therefore the main research question of this report is:

Will the usage of RNS increase the performance of CSIDH?

Giving directly an answer to this question is impossible. Therefore I will use a divide and conquer strategy to answer the main question which means that I will split the main question into multiple subquestions and answer these subquestions (divide). The answers of the subquestions will be used to solve the main question (conquer). In this report the main question will be split into the following subquestions which will be discussed in the following chapters:

1. How do you define performance of operations? (Chapter 2)
2. How does RNS works? (Chapter 3)
3. Which operations are needed for CSIDH? (Chapter 3)
4. How can you execute these operations in RNS theoretically? (Chapter 4)
5. How can you do these operations in a computer efficiently? (Chapter 5)
6. What is the performance of these operations in RNS? (Chapter 6)
7. What is the performance of these operations without RNS? (Chapter 6)

At the end of the report (Chapter 7) I will combine these answers and explain whether usage of RNS increases the performance of CSIDH or not.

Chapter 2

Definition of Performance

2.1 Introduction to Computer Systems

The definition of performance is an important aspect of this research. Therefore an entire chapter is dedicated to the definition of performance. Because this definition is used as measurement how good a certain implementation is. If such definition is not meaningful then the conclusion of this report is neither meaningful. Hence this definition should be meaningful. A meaningful definition of performance should take normal computer systems into account, since post-quantum cryptography is used by normal computer systems. This sound like an impossible job, because normal computer systems appear in a lot of machines that do not look similar, for example:

Laptops	Televisions	Modems
Desktops	Navigation Systems	Printers
Smartphones	Surveillance Cameras	Alarm Systems

All these computer systems have some form of communication. Nevertheless they all behave very different, thus a same definition for surveillance cameras and navigation systems might be a stupid idea. On the contrary, a same definition for all computer systems is an excellent idea, because all these computer systems have the same basic architecture. We need to understand this basic architecture to find an useful definition of performance. How computer systems work is also explained in the book 'The Architecture of Computer Hardware' [2], especially in chapter 7. A computer system stores data binary with transistors. The transistors are grouped which together creates a block. For most desktops/laptops this is either a block of 32 or 64 transistors, which can respectively store 32 or 64 bits of data. Any variable will be formatted in these blocks. Examples of such variables are a group of booleans, multiple characters, a string or an integer. In my research these variables are natural numbers (including

zero). If a natural number does not fit into a single block, then multiple blocks will be used to store this variable. This will happen when

$$a \geqslant 2^{\text{block-size}}$$

where a is the natural number. Unfortunately doing operations on these variables is more difficult than doing operations on variables that fit into a single block, because computer systems have an arithmetic logical unit (also abbreviated as ALU) and this ALU can only perform operations on single blocks. Operations that the ALU can do will use a fixed number of blocks as input and will produce a fixed number of blocks as output. Examples of ALU operations are:

Bitwise Exclusive Or Bitwise Exclusive Or, also abbreviated as XOR, is an operation which is supported by almost any arithmetic logical unit. The Bitwise Exlusive Or is an operation on two bits where a 1 is returned if both bits are different and 0 if both bits are the same. The Bitwise Exclusive Or will do this operation bitwise for 2 blocks as input. An example of this for block sizes of 8 bits is shown below:

$$\boxed{00100100} \oplus \boxed{01100010} \rightarrow \boxed{01000110} \tag{2.1}$$

Multiplication Multiplication will use two blocks as input and will output two blocks. One of these output blocks is the lower part of the result of multiplication. The other output block is the upper part of the result. An example of this for block sizes of 4 bits (which produces 2 blocks of 4 bits) is shown below:

$$\boxed{0101} \times \boxed{1011} \rightarrow \boxed{0011} \ \& \ \boxed{0111} \tag{2.2}$$

Logical Shift Left The Logical Shift Left operation will only use one input block and a constant c. This operation will shift all bits in this input block c positions to the left. The c leftmost bits will get discarded and the c rightmost bits will become 0. An example of this for a block size of 8 bits is shown below:

$$\boxed{11100101} << 2 \rightarrow \boxed{10010100} \tag{2.3}$$

As explained before, these operations are only possible for block size variables. However it does not mean that arithmetic for larger numbers is not possible. All operations an ALU can perform for block size variables can also be performed for variables larger than a block size. Although the ALU needs to do more operations. For example the Bitwise Exclusive Or can be executed on inputs of n blocks, by doing n Bitwise Exclusive Or operations on single blocks. In general an ALU which uses block sizes of 1 bit can compute everything an ALU can compute which uses block sizes of n bits. Nevertheless an ALU which uses block sizes of 1 bit is very inefficient, therefore most computer systems have larger block sizes and also have additional operations which can speed up computation.

2.2 Efficiency of Computer Systems

Vector instructions is an example of an additional operation that speed up computation, which is supported by most computer systems. Vector instructions execute the same operation on multiple blocks at once, which can be a huge performance optimization. Operations like addition, subtraction and multiplication have a vector instruction on almost any computer system. Some Intel processors for example can do addition for 16 blocks at once [3], which is 16 times faster than doing 16 seperate addition instructions. In general every normal operation also has a vector instruction in most computer systems. Unfortunately not every arithmetic operation is implemented on computer systems and therefore also do not have vector instruction. For example the division and modulo operation for integers do not have a vector instruction on most computer systems, because the division and modulo operation also do not have a normal instruction on most computer systems. Nevertheless you can still use vector instructions for addition, multiplication and subtraction. An example of performance gain by vector instructions is shown below:

1	2	3	4	5	6	7	8	9
$x \mathrel{+}= 1$	$y \mathrel{+}= 3$	$z \mathrel{+}= 4$	$x \mathrel{-}= x2$	$y \mathrel{-}= y2$	$z \mathrel{-}= z2$	$x \mathrel{*}= 5$	$y \mathrel{*}= 3$	$z \mathrel{*}= 2$

Table 2.1: Sequential Computation

1	2	3
$x \mathrel{+}= 1$	$x \mathrel{-}= x2$	$x \mathrel{*}= 5$
$y \mathrel{+}= 3$	$y \mathrel{-}= y2$	$y \mathrel{*}= 3$
$z \mathrel{+}= 4$	$z \mathrel{-}= z2$	$z \mathrel{*}= 2$

Table 2.2: Parallel Computation

9 actions are executed sequentially in table 2.1. This is not very optimal, since the operations on x, y and z can be executed as vector instructions instead. Which results in that only a third of the total number of operations is needed as shown in table 2.2. Hence the code will run 3 times faster. This is a huge performance gain, and this performance gain can make a code in the most optimal case 16 times faster. Therefore vector instructions are taken into account in this research. Other efficiencies of computer systems are the multiply accumulate instruction, which combines addition and multiplication. Unfortunately this is only supported for floating point numbers, not for integers. You can fix it by converting all integers to floating point numbers, however this might result in inaccurate calculations or a slowdown for vector operations, because floating point numbers often occupy multiple blocks. Hence the multiplicate accumulate is not taken into account in this research. Another efficiency of computer systems is that they often have multiple processors/cores that can do computations simultaneously. Also this is not taken into account, which is explained in section 2.4.

2.3 Definition

Since we now have a clear understanding of computer systems, we can give reasonable definition of performance. This definition of performance should be universally applicable to all sorts of normal computer systems. Therefore it needs to take the architecture of a computer system into account. In section 2.1 we have seen that every computer system has an arithmetic logical unit (ALU) which can do operations on single blocks. These operations will be called basic operations which includes vector instructions, comparisons, arithmetic operations and bitwise operations[1]. The number of such basic actions needed by the ALU to execute the code is therefore a good measurement of performance, since this definition is independent of which computer system is being used.

Definition 1 (Main Performance Measurement)**.** *The number of basic operations needed by an algorithm is the main measurement of performance of that algorithm.*

Often asymptotic time complexity is used as main measurement of performance for algorithms. With asymptotic complexity you will explain how the amount of basic operations grows for large input. In general asymptotic time complexity is a good measurement of performance. Nevertheless I will not use this as main measurement of performance, because of two reasons:

1. Asymptotic time complexity only takes very large input into account. It cannot be used to say anything about small input. An algorithm with a better asymptotic time complexity can still perform worse on smaller input than an algorithm with a worse asymptotic time complexity. For example Karatsuba algorithm and Schönhage-Strassen are both multiplication algorithms. The asymptotic time complexity of Karatsuba algorithm is $\approx O(n^{1.585})$ [4, section 1.3.2, page 15] and the asymptotic time complexity of Schönhage-Strassen is $O(n \ln n \ln \ln n)$ [4, section 1.3.4, page 18]. However Schönhage-Strassen only becomes faster than Karatsuba algorithm for input larger than $2^{2^{15}}$ [5], which are numbers with at least 10000 digits. Numbers of this size are rarely used by computers. Also Karatsuba algorithm is faster for CSIDH than Schönhage-Strassen, because all numbers used by CSIDH are equal to 512 bits.

2. Constants are ignored for asymptotic time complexity, e.g. $O(10n) = O(n)$. Suppose both algorithm A and B have a worst case time complexity of $O(n)$ then it might be still the case that algorithm B is always 20 times faster than algorithm A. Vector instructions are therefore useless according to the asymptotic time complexity measurement, because they will not change the asymptotic time complexity. However for CSIDH it is definitely useful if operations become a constant factor c faster.

The number of basic operations on the other hand does not suffer from these problems. Therefore the number of basic operations is chosen as main per-

[1] A detailed list of all basic operations used in the final code can be found in appendix A

formance measurement instead of the asymptotic time complexity. Although I will sometimes discuss the asymptotic time complexity of an operation. Beside the main performance measurement I will also use a second performance measurement.

Definition 2 (Second Performance Measurement)**.** *The number of clock ticks used by an algorithm is the second measurement of performance of that algorithm.*

Clock ticks is also independent of the computer system used and is therefore a better measurement than running time[2]. Moreover it is also easier to measure than the main performance measurement. But the main performance measurement on the other hand will give a better indication how fast the algorithm actually can be. It might for example be the case that an algorithm is very badly implemented. Nevertheless both definitions have one main disadvantage: they both do not take security into account.

2.4 Timing Attack

Performance of CSIDH is indeed important, but security is even more important. The running time of a cryptosystem should be independent on the input used, otherwise the cryptosystem is vulnerable to a timing attack [7]. A hacker will then predict what the plaintext or key is based on the time encryption/decryption takes. Therefore when improving the performance of operations, it should be an improvement for all types of input. The running time of operations should never depend on the input size, divisors of the input or any other characteristic of the input. To satisfy this, there are three guidelines to follow when implementing operations. Of course it does not guarantee that the running time of an operation is input independent, but not following them will often cause an insecure operation.

Appending Zero Bits

When running operations for at most n bits input, you should always append zero bits to the left of the input until it has the size of n bits. And blocks of only 0 bits should be treated as any other blocks. For addition, subtraction and multiplication appending zero bits and treating blocks with only 0 bits as any other block will prevent a difference in running time between larger and smaller input sizes. For example suppose we have block sizes of 4 bits and in general use input of 8 bits. Then if want to multiply $[1011]_2 = 11$ by $[0101]_2 = 5$, we append 4 zero bits and do multiplication as shown in figure 2.1. If we do it this way, then multiplying 4 bit input will take the same amount of time as multiplying 8 bit input. Unfortunately 3 unnecessary multiplications are calculated in this case, but the running time of multiplication will be the same for every size of

[2]However make sure that hyperthreading and turbo boost are turned off, since this might influence the number of clock ticks [6]

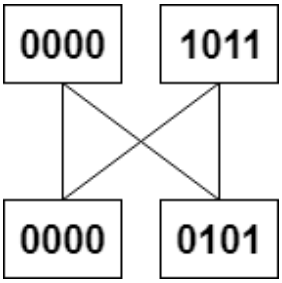

Figure 2.1: Appending bits for multiplication

input. Appending zero bits will sometimes also make the running time of more advanced operations input independent, however you should always check if this is indeed the case. For example computation of the greatest common divisor with the Euclidean algorithm has a different running time for different input, even if the input used is of size n bits. If for instance either one of the inputs is a small number actually or the difference between both inputs is small then the running time will also be smaller. Thus for more advanced operations you should always check if appending zero bits will make the running time input independent.

Do not use If/Else

Operations used should not contain if and else statements which depend on secret data, since the running time will most likely be different if the else part is executed instead of the if part. Based on this an eavesdropper might predict whether a statement was true or false and therefore can make suggestions about this secret data. Thus avoid if/else statements, unless absolutely sure that the if and else block will have the same running time. In the code below is an example of such bad cryptosystem which violates this principle.

Algorithm 1 Fussy Encryption

```
function FussyEncrypt(m[], k[])
 1: for i := 1; i ⩽ k.length; i++ do
 2:     if k[i] == 1 then
 3:         firstbit := m[i]                ▷ Swap all bits 1 position to the left
 4:         for j := 2; j ⩽ k.length; j++ do
 5:             m[j - 1] := m[j]
 6:         end for
 7:         m[k.length] := firstbit
 8:     else
 9:         Flip(m[i])          ▷ Flip changes a zero bit to one and vice-versa
10:     end if
11: end for
12: return m                                ▷ Return the encrypted message
```

m in this algorithm are the message bits and k are the key bits which are of the same size. In this code the if part will take a computer much more time to perform than the else part. Hence the number of one bits in the key can be predicted based on the running time of this encryption algorithm.

Multithreading

Multithreading is a different method of parallellism the vector instructions explained in section 2.2. With multithreading you can run parallel tasks in seperate threads. Computer systems will then schedule these threads such that computation power is equally divided over these threads. If a computer systems has multiple processors/cores then this means that the threads will be divided over these processors/cores. Which implies that multiple operations can be executed at the same time, since processors/cores have seperate ALU's and seperate memory. Most laptops/desktops nowadays have at least 4 processors/cores, hence scripts can become at least 4 times as fast with multithreading. Therefore multithreading sounds like a good idea. Unfortunately multithreading is a major security risk, because memory often get cached in case multithreading is used. And caching is insecure [8, section 2.2, page 3], because there are different level of caches. These different cache levels have differences in access time (time it takes to load data). A malicious program could manipulate the caches and measure time it takes for the cryptosystem to perform certain actions. With the measured time the malicious program can guess which cache levels were accessed and therefore can make a guess which data is used. Thus secret data might be revealed to malicious programs when multithreading is implemented. Therefore multithreading is not discussed any further in this report.

Chapter 3

Residue Number System

3.1 Introduction to RNS

The residue number system (abbreviated as RNS) is an alternative way to represent numbers. In RNS some operations are much easier than in a regular number system $\mathbb{Z}_s$, for example multiplication. On the other hand operations like modular reduction and checking which number is larger are harder in RNS than in a regular number system. Hence if you switch between a regular number system and RNS, it could make a lot of operations more efficient. This is the main assumption that RNS could improve the performance of CSIDH. Therefore in this chapter I will explain how you can do operations in the residue number system, how to switch a number to RNS and back from RNS. But before explaining this, we must first understand what the residue number system is. In RNS an integer k is written as a vector of integers $(k_1, k_2, \ldots, k_n)$ such that:

$$\begin{aligned} k &\equiv k_1 \pmod{m_1} \\ k &\equiv k_2 \pmod{m_2} \\ &\vdots \\ k &\equiv k_n \pmod{m_n} \end{aligned}$$

These k_i will be called residues and are variable numbers. The m_i will be called moduli and are fixed numbers on the other hand, which will be defined beforehand. You cannot select arbitrary integers for these moduli. The moduli m_i have to satisfy the following properties:

1. All moduli m_i must be natural numbers larger or equal to 2.
2. All moduli m_i must be coprime, i.e. $\gcd(m_i, m_j) = 1$ if $i \neq j$

We will see in section 3.2 that these properties are useful for conversion. Without these properties conversion to and from RNS will not be possible or will not make any sense. A common way to satisfy these properties is to pick different prime

numbers as moduli m_i. However other moduli are also possible. In section 3.5 and section 5.4 is given more information about how to select these moduli. In this research these moduli are selected such that they are friendly to use by computer systems. Likewise the residues should also be selected such that they are also friendly for computer systems. Therefore I use a slightly different definition of RNS. According the definition of RNS used in this report an integer k must be written as a vector of integers $(k_1, k_2, \ldots, k_n)$ such that:

$$\begin{aligned} k \bmod m_1 &= k_1 \\ k \bmod m_2 &= k_2 \\ &\vdots \\ k \bmod m_n &= k_n \end{aligned}$$

The rest of the definition will remain the same (including the properties of the moduli). The main difference between these definitions is therefore the difference between $a \equiv b \pmod{c}$ and $a \bmod c = b$ which is indeed different. You should not mix up these terms.

Important Note

- $a \equiv b \pmod{c}$ means $a = b + cd$ where $d \in \mathbb{Z}$
- $a \bmod c = b$ means $a = b + cd$ where $d \in \mathbb{Z}$ and $0 \leqslant b < c$

Both the statements $a \equiv b \pmod{c}$ and $a \bmod c = b$ will be used in this report, hence you should be aware of this, because the meaning of both notations is different. Below are examples in which the difference between both notations is made more clear:

Example 1. *The following statements are true:*

- $8 \equiv 13 \pmod 5$ - $8 \equiv 3 \pmod 5$ - $8 \not\equiv 4 \pmod 5$ - $8 \equiv -2 \pmod 5$
- $8 \bmod 5 \neq 13$ - $8 \bmod 5 = 3$ - $8 \bmod 5 \neq 4$ - $8 \bmod 5 \neq -2$

In general the b in $a \bmod c = b$ is unique given the values of a and c. This is the main reason why the last definition of RNS is used, since it is more convenient for computer systems to have an unique b. Moreover the result of $a \bmod c$ will always be in the range 0 (inclusive) up to c (exclusive). Thus you cannot get a number larger than c, which is also convenient for computer systems, because it is easier to do computations with smaller numbers in computer systems. Therefore we also convert large numbers to RNS, because for a large numbers the computation time needed to do operations might be much larger than the computation time needed to do operations for multiple smaller numbers. In the next section I will explain how to convert a large number to RNS and why this conversion makes sense.

3.2 Conversion to RNS

Conversion to RNS is very easy, because according to the definition of RNS a $k \in \mathbb{Z}_s$ can be converted to $(k_1, k_2, \ldots, k_n)$ in RNS with moduli $\{m_1, m_2, \ldots, m_n\}$ by using the following formula:

$$k_i = k \bmod m_i \quad \text{for all } i$$

Unfortunately this conversion is not meaningful if the conversion to RNS is not injective, since then you cannot convert it back. Luckily the conversion is injective if s is equal to the product of the moduli $(m_1, m_2, \ldots, m_n)$.

Lemma 1. *Conversion from $\mathbb{Z}_m$ to the residue number system with moduli $\{m_1, m_2, \ldots, m_n\}$ is injective with $m = \prod_{i=1}^{n} m_i$.*

Proof. According to the Chinese Remainder Theorem [9, section 4.3.2, page 286] if $l_1, l_2, \ldots, l_n$ are positive integers that are relatively prime and if $l = \prod_{i=1}^{n} l_i$ then for all integers $a, b_1, b_2, \ldots b_n$ there is exactly one integer b that satisfies:

$$a \leqslant b < a + l \qquad \text{and} \qquad b \equiv b_i \pmod{l_i} \quad \text{for all } i$$

It is therefore especially also true for $a = 0$ and $0 \leqslant b_i < l_i$. In RNS all m_i are positive integers and relatively prime, thus we can conclude that for all residues $b_i \in \mathbb{Z}_{m_i}$ there is exactly one $b \in \mathbb{Z}_m$ that satisfies:

$$b_i = b \bmod m_i \quad \text{for all } i$$

Hence the conversion from $\mathbb{Z}_m$ to the residue number system with moduli $\{m_1, m_2, \ldots, m_n\}$ is injective. □

And because the conversion is injective, we can translate from any $\mathbb{Z}_s$ to RNS, do the operations in RNS and convert it back to $\mathbb{Z}_s$. A summary of this conversion is given in pseudocode below.

Algorithm 2 Conversion to RNS

```
function toRNS(b, m[])
1: k := new array(m.length)                    ▷ Create an empty array
2: for i := 1; i ⩽ k.length; i++ do
3:     k[i] := b % m[i]                        ▷ Do the modulo operation
4: end for
5: return k                                    ▷ Return the RNS notation
```

The input b is the integer in $\mathbb{Z}_m$ that will be converted to RNS form.
m[] is the array of moduli. The pseudocode is not very long, because as written before the conversion to RNS form is easy. Conversion from RNS notation to $\mathbb{Z}_m$ on the other hand is a little bit harder.

3.3 Conversion from RNS

A conversion algorithm from the residue number system to $\mathbb{Z}_m$ is explained in 'The Art Of Computer Programming' [9, section 4.3.2, page 286-287]. In the residue number system you have a standard basis:

$$\begin{aligned} s_1 &= (1, 0, \ldots, 0, 0) \\ s_2 &= (0, 1, \ldots, 0, 0) \\ &\vdots \\ s_{n-1} &= (0, 0, \ldots, 1, 0) \\ s_n &= (0, 0, \ldots, 0, 1) \end{aligned}$$

The conversion start by finding the integers a_i in $\mathbb{Z}_m$ that are mapped to these basis vectors s_i of the residue number system, i.e.

$$a_i \bmod m_i = 1 \qquad \text{and} \qquad a_i \bmod m_j = 0 \quad \text{for } j \neq i$$

After finding these a_i it will be easy to convert $(b_1, b_2, \ldots, b_n)$ to the corresponding b in $\mathbb{Z}_m$, since vector $(b_1, b_2, \ldots, b_n)$ can be expressed as linear combination of the basis vectors s_i:

$$(b_1, b_2, \ldots, b_n) = b_1 \cdot s_1 + b_2 \cdot s_2 + \cdots + b_n \cdot s_n$$

and therefore we can determine b, since a_i corresponds to s_i:

$$b = (b_1 \cdot a_1 + b_2 \cdot a_2 + \cdots + b_n \cdot a_n) \bmod m$$

The only information missing is how to determine these a_i. We know that these a_i exists and are unique in $\mathbb{Z}_m$, which is true because of lemma 1. But how can you find a_i? Luckily a_i can be determined with the following formula:

$$a_i = M_i \cdot (M_i)^{-1}$$

where M_i is defined as the product of all moduli except m_i, i.e.

$$M_i = \frac{m}{m_i}$$

and M_i^{-1} is the multiplicative inverse of M_i modulo m_i, which can be determined by the extended Euclidean algorithm[1]. The product of both numbers is equal to a_i and the product is in interval $[0, m)$, which will be proven on the next page.

Lemma 2. *If* $a_i = M_i \cdot (M_i)^{-1}$ *then* $a_i \equiv 1 \pmod{m_i}$, $a_i \equiv 0 \pmod{m_j}$ *for all* $j \neq i$ *and* $0 \leqslant a_i < m$.

[1] Usage of the Extended Euclidean Algorithm as explained in section 2.4 might reveal the input used, however since the moduli used for RNS are public information this is not a security concern.

Proof. The proof can be divided in 4 seperate parts, which together prove this lemma:

1. $(M_i)^{-1}$ is well defined, i.e. there exist an integer g such that: $g \cdot M_i \equiv 1 \pmod{m_i}$
2. $a_i \equiv 1 \pmod{m_i}$
3. $a_i \equiv 0 \pmod{m_j}$ for all $j \neq i$
4. $0 \leqslant a_i < m$

1. According to the definition of the residue number system all moduli must be coprime, i.e. $\gcd(m_j, m_k) = 1$ if $j \neq k$. Also if $\gcd(d, e) = 1$ then [10]:

   $$\gcd(d \cdot e, f) = \gcd(d, f) \cdot \gcd(e, f)$$

   Hence we can deduce that:

   $$\gcd(M_i, m_i) = \prod_{j \neq i} \gcd(m_j, m_i) = \prod_{j \neq i} 1 = 1$$

   And since $\gcd(M_i, m_i) = 1$ it means that M_i is coprime to m_i, thus the inverse of M_i modulo m_i exists.

2. By the definition of the modular multiplicative inverse this is true, i.e. for all invertible h it holds that $h \cdot h^{-1} \equiv 1 \pmod{m_i}$.

3. We can prove that a_i is divisible by m_j with $j \neq i$, since:

   $$\begin{aligned} a_i &= M_i \cdot (M_i)^{-1} \\ &= \left(\prod_{k \neq i} m_k \right) \cdot (M_i)^{-1} \\ &= m_j \cdot \left(\left(\prod_{k \neq i,j} m_k \right) \cdot (M_i)^{-1} \right) \end{aligned}$$

   Therefore a_i is divisible by m_j, which means that $a_i \equiv 0 \pmod{m_j}$.

4. $M_i^{-1} < m_i$ because the modular inverse is an integer in interval $[0, m_i)$. Therefore $a_i = M_i \cdot M_i^{-1} < M_i \cdot m_i = m$. Furthermore $M_i \geqslant 0$ and $M_i^{-1} \geqslant 0$. Thus $a_i = M_i \cdot M_i^{-1} \geqslant 0$.

Because all these separate parts are proven, we can conclude that if $a_i = M_i \cdot (M_i)^{-1}$ then $a_i \equiv 1 \pmod{m_i}$, $a_i \equiv 0 \pmod{m_j}$ for all $j \neq i$ and $0 \leqslant a_i < m$. □

The method to convert back from RNS to $\mathbb{Z}_m$ is now clear. This method is also written in the pseudocode below:

Algorithm 3 Conversion from RNS

```
function fromRNS(b[], m[])
 1: prod := 1                                   ▷ The product of the residues
 2: for i := 1; i ⩽ m.length; i++ do
 3:     prod := prod * m[i]
 4: end for
 5: a := new array(m.length)                    ▷ Integers mapped to standard basis
 6: for i := 1; i ⩽ a.length; i++ do
 7:     prod2 := prod / m[i]                    ▷ Product of all moduli except m[i]
 8:     a[i] := inverse(prod2, m[i])            ▷ Do extended Euclidean algorithm
 9:     a[i] *= prod2
10: end for
11: r := 0
12: for i := 1; i ⩽ a.length; i++ do
13:     r := (r + b[i] * a[i]) % prod
14: end for
15: return r                                    ▷ Return regular notation
```

b[] in this algorithm is an array of the RNS notation and m[] is an array of residues with $b \bmod m[i] =$ b[i]. Since we now know how to convert to RNS and convert back from RNS, we can do the operations in RNS instead of the regular number system.

3.4 Operations in RNS

In RNS the rules of modular arithmetic can be applied for addition, subtraction and multiplication. For addition this means that if $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ are elements of the same residue number system with residues $(m_1, m_2, \ldots, m_n)$ then $a + b$ is defined as:

$$((a_1 + b_1) \bmod m_1, (a_2 + b_2) \bmod m_2, \ldots, (a_n + b_n) \bmod m_n)$$

which is just addition of the corresponding entries. This is the correct representation of $a + b$, because for a and b we know by the definition of the residue number system that:

$$a \equiv a_i \pmod{m_i} \qquad \text{and} \qquad b \equiv b_i \pmod{m_i} \quad \text{for all } i$$

Hence:

$$a + b \equiv a_i + b_i \pmod{m_i} \quad \text{for all } i$$

Subtraction $a - b$ is likewise defined as:

$$((a_1 - b_1) \bmod m_1, (a_2 - b_2) \bmod m_2, \ldots, (a_n - b_n) \bmod m_n)$$

And multiplication $a \cdot b$ is defined in the same way:

$$((a_1 \cdot b_1) \bmod m_1, (a_2 \cdot b_2) \bmod m_2, \ldots, (a_n \cdot b_n) \bmod m_n)$$

This means that only n multiplications are needed to calculate $a \cdot b$ in RNS. If m_i are all small numbers, which is the case when all m_i are smaller than 1 block size, then it will have an outstanding result in the performance of multiplication, because it will take $O(n)$ number of basic operations. On the other hand for a k-bit number it will have a much worse performance. Toom-Cook 3 way, a very fast multiplication algorithm, takes about $O(k^{1.465})$ basic operations for multiplying an arbitrary k-bit number [4, section 1.3.3]. Hence multiplication is much faster in RNS than in $\mathbb{Z}_m$. Unfortunately these operations (including multiplication) need to be adjusted for CSIDH, thus the fast performance of multiplication will decline.

3.5 RNS in CSIDH

CSIDH is a post-quantum cryptography algorithm which is introduced in 'CSIDH: An Efficient Post-Quantum Commutative Group Action' [1]. In CSIDH integers are represented in $\mathbb{Z}_p$. This p is an approximately 512 bit number, which is defined by:

$$p = 4 \left(\prod_{i=1}^{74} p_i \right) - 1$$

The first 73 p_i are the first 73 primes numbers starting from 3 and the 74th number is 587 which is also a prime number. The result of this calculation will give p which is also a prime number. This means that the only residue number system that is isomorphic to $\mathbb{Z}_p$ is the residue number system with only 1 modulus, which is equal to p. Using this residue number system will of course not increase the performance of CSIDH, since the modulus that is used is still very large. To improve the performance of RNS it is better to use small moduli. The moduli used in this research will therefore be a grouping of the small prime numbers p_i used for p with some additional factors. These additional factors will be the same small prime numbers p_i and 4 such that the product of all moduli m_i is large enough to do operations and such that all moduli are still coprime. An example of such moduli set might be:

$$\begin{aligned} m_1 &= (p_1)^3 \cdot p_3 \cdot (p_7)^2 \cdot p_8 \\ m_2 &= (p_{65})^5 \\ &\vdots \\ m_{18} &= p_5 \cdot p_6 \cdot p_9 \cdot p_{11} \cdot (p_{12})^2 \cdot p_{69} \\ m_{19} &= p_4 \cdot p_{55} \end{aligned}$$

Since such residue number system is not isomorphic to $\mathbb{Z}_p$ we need to deal with modular operations in RNS, because the operations in CSIDH are still

executed modulo p. Therefore we need to determine how to do the following basic operations of $\mathbb{Z}_p$ in RNS:

1. Addition: $(x + y) \bmod p$ with $x, y \in \mathbb{Z}_p$
2. Subtraction: $(x - y) \bmod p$ with $x, y \in \mathbb{Z}_p$
3. Multiplication: $(x \cdot y) \bmod p$ with $x, y \in \mathbb{Z}_p$

When we know how to do these basic operations, we can also do any advanced operation executed in CSIDH. Since every advanced operation can be performed with only these basic operations. Examples of such advanced operations are:

Exponentiation Exponentiation is defined as $x^c \bmod p$ with $x \in \mathbb{Z}_p$ and c a constant. This advanced operation $x^c \bmod p$ can be calculated with $\log c$ multiplications, by efficiently squaring numbers and multiplying them by x, which is shown in the pseudocode below:

Algorithm 4 Modular Exponentiation

```
function ModularExponentiation(x, c, p)
1: if c == 0 then
2:     return 1
3: else if c is odd then
4:     v := ModularExponentiation(x, c - 1, p)
5:     return (x * v) % p
6: else
7:     v := ModularExponentiation(x, c / 2, p)
8:     return (v * v) % p
9: end if
```

Inversion Inversion is defined as finding the modular multiplicative inverse of a number: $x^{-1} \bmod p$ with $x \in \mathbb{Z}_p$. As explained in section 2.4, you cannot do this with the extended Euclidean algorithm, since x is sensitive information. Fortunately you can also calculate the inverse with Euler's theorem [11, section 10.3, page 133], because:

$$x^{-1} \bmod p = x^{\phi(p)-1} \bmod p = x^{p-2} \bmod p$$

Therefore inversion can be calculated with exponentiation.

Division Division $\frac{x}{y} \bmod p$ can be calculated by multiplying x by the inverse of y. Hence division can be calculated with inversion and multiplication.

So if we know how to do modular addition, modular subtraction and modular multiplication in RNS, we can do any operation of CSIDH in RNS.

Chapter 4

RNS Modular Arithmetic

4.1 Conversion Method

There are multiple methods for doing modular arithmetic in RNS. One way to deal with the problem of modular arithmetic in RNS is to do normal addition and multiplication in RNS and execute the main modulo operation $\bmod z$ in $\mathbb{Z}_m$. This procedure is very simple to understand and to use. Moreover it also does not have many requirements which have to be satisfied before using this procedure. The only requirement is that we have to select the moduli m_i for the residue number system such that the product of moduli m is large enough to do addition and multiplication modulo z. This means that a certain lowerbound on m is required for the following operations:

Addition The largest number that can occur after addition of two numbers in $\mathbb{Z} \cap [0, z)$ is $2z - 2$ by adding $z - 1$ to itself. The smallest number that can occur after addition is 0 by adding 0 to itself. Therefore the residue number system should handle integers in the range $[0, 2z - 2]$. According to lemma 1 the conversion from $\mathbb{Z}_m$ to RNS is injective with m the product of the moduli of that residue number system. Thus if the product of the moduli $m > 2z - 2$ then $\mathbb{Z} \cap [0, 2z - 2]$ is a subset of $\mathbb{Z}_m$ which has an injective conversion to that residue number system. This means that if $m > 2z - 2$ then every integer in $\mathbb{Z} \cap [0, 2z - 2]$ will have a unique representation in RNS. Hence addition of two numbers modulo z will never go wrong if for the product of moduli it holds that $m > 2z - 2$.

Multiplication The largest number that can occur after multiplication of two numbers in $\mathbb{Z} \cap [0, z)$ is $(z - 1)^2$ by multiplying $z - 1$ by itself. The smallest number that can occur after multiplication is 0 by multiplying 0 by itself. Hence the residue number system should handle integers in the range $[0, (z - 1)^2]$. Similarly to addition these numbers will have a unique representation in RNS if the product of the moduli $m > (z - 1)^2$. Therefore multiplication of two numbers modulo z will never fail if for the product of moduli it holds that $m > (z - 1)^2$.

We know that the product of moduli m used is definitely larger than 3, which means that $(z-1)^2 \geqslant 2z-2$ hence we only have to assure that the product of the moduli is larger than $(z-1)^2$. When this is true, we can do both addition and multiplication in RNS and do the modular reduction in $\mathbb{Z}_m$ to retrieve the result of modular addition/multiplication. The procedure of this method is summarized below:

1. Do normal addition/multiplication in RNS
2. Convert the result back to $\mathbb{Z}_m$
3. Do the main modulo operation $\bmod z$ in $\mathbb{Z}_m$
4. Convert the result of the modulo operation back to RNS

An example of this for modular multiplication is shown below:

Example 2. *Suppose we have a residue number system with moduli* $(4, 5, 7)$ *and we want to do modular multiplication in* $\bmod 11$ *then this is possible because* $4 \cdot 5 \cdot 7 = 140 > 100 = (11-1)^2$*. Let a be a RNS number with* $(3, 2, 0)$*, which is equal to* 7*. And b also be a RNS number with* $(2, 1, 6)$*, which is equal to* 6*. Then both RNS numbers are valid to use, because their representation is in* $\mathbb{Z} \cap [0, 11)$*. So we can follow the procedure:*

1. $(3, 2, 0) \cdot (2, 1, 6) = ((3 \cdot 2) \bmod 4, (2 \cdot 1) \bmod 5, (0 \cdot 6) \bmod 7) = (2, 2, 0)$ *Thus* $(2, 2, 0)$ *is the result after normal multiplication in RNS.*
2. *To convert the result back to we need to compute* $M_i \cdot M_i^{-1}$*:*

   $M_1 \cdot M_1^{-1} = 35 \cdot 3 = 105 \qquad M_2 \cdot M_2^{-1} = 28 \cdot 2 = 56 \qquad M_3 \cdot M_3^{-1} = 20 \cdot 6 = 120$

   The representation of $(2, 2, 0)$ *in* $\mathbb{Z}_{140}$ *is*

   $$(2 \cdot 105 + 2 \cdot 56 + 0 \cdot 120) \bmod 140 = 42$$

3. $42 \bmod 11 = 9$ *is the result after the main modulo operation.*
4. 9 *is equal to the RNS number* $(9 \bmod 4, 9 \bmod 5, 9 \bmod 7) = (1, 4, 2)$*. This is therefore the final result after modular multiplication in RNS.*

It looks like a very efficient method, since addition and multiplication in RNS is very efficient and the modulo operation is more easier in $\mathbb{Z}_m$. Nevertheless it will be faster to do the addition and multiplication directly in $\mathbb{Z}_m$ and then do the main modulo operation $\bmod z$ in $\mathbb{Z}_m$. The difference is that steps 1, 2 and 4 are replaced by a multiplication in $\mathbb{Z}_m$. And multiplication in $\mathbb{Z}_m$ with Toom-Cook 3 way [4, section 1.3.3] is much faster than executing step 2, because multiple big integers are added and multiplied by small integers. Hence with this method CSIDH will not become faster with RNS. Therefore in the rest of this chapter we will try a different approach. In chapter 6 we will see whether this approach is indeed successful or not.

4.2 Mixed Radix System for RNS

For this new approach to handle modular operations we use the mixed radix system. The mixed radix system is another system in which you can represent numbers of $\mathbb{Z}_m$. To understand the mixed radix system we should first understand a normal radix system. In a normal radix system a number a is written as:

$$a = a_1 + a_2\beta + a_3\beta^2 + \cdots + a_{n-1}\beta^{n-1} + a_n\beta^n$$

where β is the base and each a_i is an integer such that $0 \leqslant a_i < \beta$. Well known examples of these systems are decimal systems with $\beta = 10$ and binary systems with $\beta = 2$. The mixed radix system is pretty similar, however it can use different bases $(\beta_1, \beta_2, \ldots, \beta_n)$. In the mixed radix system, a number a is written as:

$$a = a_1 + a_2\beta_1 + a_3\beta_1\beta_2 + \cdots + a_{n-1}\prod_{i=1}^{n-2}\beta_i + a_n\prod_{i=1}^{n-1}\beta_i$$

where each a_i is an integer such that $0 \leqslant a_i < \beta_i$. The mixed radix system is very useful, since it has the nice properties of a normal radix system. For example comparison of numbers is easier and modulo operations are also possible. But how do we convert a residue number system to a mixed radix system? In Knuth [9, section 4.3.2, page 290] is explained how you can determine digits $(d_1, d_2, \ldots, d_n)$ for the mixed radix system with bases $(m_1, m_2, \ldots, m_n)$. For this you need the residues $(r_1, r_2, \ldots, r_n)$ of a residue number system with moduli $(m_1, m_2, \ldots, m_n)$. Then we can determine the digits with the following equations where $|m_i^{-1}|_{m_j}$ is the inverse of m_i modulo m_j:

$$\begin{aligned}
d_1 &= r_1 \bmod m_1 \\
d_2 &= (r_2 - d_1) \cdot |m_1^{-1}|_{m_2} \bmod m_2 \\
d_3 &= ((r_3 - d_1) \cdot |m_1^{-1}|_{m_3} - d_2) \cdot |m_2^{-1}|_{m_3} \bmod m_3 \\
&\vdots \\
d_n &= (\ldots((r_n - d_1) \cdot |m_1^{-1}|_{m_n} - d_2) \cdot |m_2^{-1}|_{m_n} - \cdots - d_{n-1}) \cdot |m_{n-1}^{-1}|_{m_n} \bmod m_n
\end{aligned}$$

These equations are correct, which will be shown in the proof below.

Proof. To prove that the equations are correct, we have to show that:

$$\left(d_1 + d_2m_1 + \cdots + d_{n-1}\prod_{j=1}^{n-2} m_j + d_n\prod_{j=1}^{n-1} m_j\right) \bmod m_i = r_i$$

On the left side we can remove the terms that are a multiple of m_i:

$$\left(d_1 + d_2m_1 + \cdots + d_{i-1}\prod_{j=1}^{i-2} m_j + d_i\prod_{j=1}^{i-1} m_j\right) \bmod m_i$$

Furthermore d_i can be rewritten as:

$$d_i = \left(r_i \prod_{j=1}^{i-1} |m_j^{-1}|_{m_n} - \sum_{k=1}^{i} d_k \prod_{j=k}^{i-1} |m_j^{-1}|_{m_n} \right) \bmod m_i$$

So the last term of the left side is equal to:

$$\begin{aligned} \left(d_i \prod_{j=1}^{i-1} m_j \right) \bmod m_i &= \left(r_i - \sum_{k=1}^{i} d_k \prod_{j=1}^{k-1} m_j \right) \bmod m_i \\ &= \left(r_i - d_{i-1} \prod_{j=1}^{i-2} m_j - \cdots - d_2 m_1 - d_1 \right) \bmod m_i \end{aligned}$$

Therefore:

$$\left(d_1 + d_2 m_1 + \cdots + d_{i-1} \prod_{j=1}^{i-2} m_j + d_i \prod_{j=1}^{i-1} m_j \right) \bmod m_i =$$

$$\left(d_1 + d_2 m_1 + \cdots + d_{i-1} \prod_{j=1}^{i-2} m_j + r_i - d_{i-1} \prod_{j=1}^{i-2} m_j - \cdots - d_2 m_1 - d_1 \right) \bmod m_i =$$

$r_i \bmod m_i = r_i$

Which means that these equations are indeed correct. □

Moreover every equation is explicit, so we can determine all digits for the mixed radix system. A pseudocode for this conversion can be found below.

Algorithm 5 RNS to MR

```
function RNSToMR(r[], m[])
 1: d := new array(r.length)                    ▷ The digits for mixed radix
 2: for i := 1; i ⩽ d.length; i++ do
 3:     d[i] := r[i]
 4:     for j := 1; j < i; j++ do
 5:         d[i] = (d[i] + m[i] - d[j]) % m[i]
 6:         d[i] *= inverse(m[j], m[i])     ▷ Multiply by inverse of m[j] in m[i]
 7:         d[i] %= m[i]
 8:     end for
 9: end for
10: return d
```

The input r in this algorithm is an array with residues for the residue number system using the moduli in array m. The output d is an array of digits for mixed radix sorted by significance (d[1] is the least significant digit). With this algorithm we can convert a residue number system (RNS) to mixed radix (MR) and use the nice properties of MR for RNS, such as easy comparison.

4.3 Comparison in RNS

Comparison in mixed radix systems is easier than in residue number systems. Nevertheless to check if two numbers are equal is already possible in RNS by checking whether all residues are the same. But determining whether a number is smaller/larger than some number is much harder in RNS than in MR. If you want to check whether a number in mixed radix is smaller/larger than another number you take a look at the most significant digits and determine which one is larger. If these digits are equal you will compare lesser significant digits until you find a case where the digits are not equal. This method is also proposed in "Algorithms for comparison in residue number systems" [12]. However the running time of this procedure is input dependent, because if a number is much larger than another number the algorithm will stop more earlier. Fortunately a constant running time method is also possible. The pseudocode below is an example of a constant running time method for smaller than comparison in the residue number system:

Algorithm 6 Smaller than for RNS

function isSmallerRNS(r1[], r2[], m[])

```
1: d1 := RNSToMR(r1, m)
2: d2 := RNSToMR(r2, m)
3: smaller := (d1[1] < d2[1])
4: for i := 2; i ⩽ m.length; i++ do
5:     smaller := (d1[i] == d2[i]) & smaller
6:     smaller := (d1[i] < d2[i]) | smaller
7: end for
8: return smaller
```

The input r1 is an array of residues of the first number and r2 is an array of residues of the second number with moduli m. In this algorithm a boolean variable named "smaller" is used. This boolean variable will keep track whether the lesser significant digits are already smaller or not. "smaller" at line 6 will be set to true if the lesser significant digits are already smaller and the new digits are equal. It will also be set to true if the new digit of the first number is smaller than the new digit of the second number. Therefore this algorithm will return true if the first number is smaller than the second number and false otherwise. The advantage of this algorithm is that it is running time independent. It will start with comparison of the least significant digits, until we reach the most significant digits. Therefore we loop over all digits and we know only at the end whether a number is smaller than some other number. Hence the pseudocode has a constant running time[1]. And therefore it can be used in CSIDH operations, for example modular addition in CSIDH.

[1] The & and | operators should however be implemented as bitwise operators not as logical operators, otherwise a short-circuit evaluation is possible which is not running time independent on the expression used [13].

4.4 Modular Addition in RNS

Modular addition in RNS with z as main modulus can now be computed, because modular addition can be performed by doing normal additions/subtractions and comparisons in RNS. When you add two integers in range $[0, z-1]$ the smallest result will be obtained when you add 0 to itself, which results in 0. The largest result will be obtained when you add $z-1$ to itself, which results in $2z-2$. Therefore the result will be in range $[0, 2z-2]$. Thus when you do modular addition $(a+b)$ mod z the result is either $a+b$ or $a+b-z$. To determine which one it is can be done with a comparison in RNS. If $a+b \geqslant z$ then $a+b-z$ is the result of the modular addition. Otherwise $a+b$ is the result of the modular addition. This method is summarized as pseudocode below:

Algorithm 7 Modular Addition for RNS

```
function modularAdditionRNS(r1[], r2[], z[], m[])
 1: r3 := new array(m.length)              ▷ The RNS number after addition
 2: for i := 1; i ⩽ m.length; i++ do
 3:     r3[i] := (r1[i] + r2[i]) % m[i]
 4: end for
 5: r4 := new array(m.length)                    ▷ Subtraction of result by z
 6: for i := 1; i ⩽ m.length; i++ do
 7:     r4[i] := (r3[i] - z[i]) % m[i]
 8: end for
 9: if isSmallerRNS(r3, z, m) then
10:     return r3
11: else
12:     return r4
13: end if
```

r1 and r2 are the residues of the numbers in RNS that will be modular added. z is the main modulus for this operation in RNS representation. m is the array with moduli used for the residue number system. A nice property of this algorithm is that the running time is independent of the inputs used, because the subtraction of the main modulus is always computed, even if the result is already smaller than the main modulus. The only requirement that needs to be satisfied before using this algorithm, is that the product of moduli m for the residue number system is larger than $2z-2$. When this is not the case an overflow might happen. An example of this is given below:

Example 3. *Suppose we have a residue number system with moduli* 4 *and* 5. *In this residue number system we want to add* 9 *and* 12 *in mod* 13. 9 *is equal to* $(1, 4)$, 12 *is equal to* $(0, 2)$ *and* 13 *is equal to* $(1, 3)$. *If we add both numbers* $(1, 4)$ *and* $(0, 2)$ *we get*

$$((1+0) \bmod 4, (4+2) \bmod 5) = (1, 1)$$

To check whether we should subtract the main modulus, we convert both $(1,1)$ *and* $(1,3)$ *to mixed radix and get respectively:*

$$(1,1)_{RNS} = (1,0)_{MR} \qquad (1,3)_{RNS} = (1,3)_{MR}$$

The most significant digit of $(1,0)$ *is* 0 *which is smaller than the most significant digit of* $(1,3) = 3$*. Therefore* $(1,1)$ *is the result of the modular addition. But*

$$(9 + 12) \bmod 13 = 8$$

And the RNS representation of 8 *is equal to* $(0,3)$ *which is not equal to* $(1,1)$*. On the other hand if we have a residue number system with moduli* 4 *and* 7 *then there is no risk of overflow. Because the product of* 4 *and* 7 *is* 28 *which is larger than* $2 \cdot 13 - 2 = 24$*. In this residue number system* 9 *is equal to* $(1,2)$*,* 12 *is equal to* $(0,5)$ *and* 13 *is equal to* $(1,6)$*. If we add both numbers* $(1,2)$ *and* $(0,5)$ *we get*

$$((1 + 0) \bmod 4, (2 + 5) \bmod 7) = (1,0)$$

To check whether we should subtract the main modulus, we convert both $(1,0)$ *and* $(1,6)$ *to mixed radix and get respectively:*

$$(1,0)_{RNS} = (1,5)_{MR} \qquad (1,6)_{RNS} = (1,3)_{MR}$$

The most significant digit of $(1,5) = 5$ *is larger than the most significant digit of* $(1,3) = 3$*. Hence we have to subtract the main modulus and get*

$$((1 - 1) \bmod 4, (0 - 6) \bmod 7) = (0,1)$$

This RNS number is indeed equal to 8*, because* $8 \bmod 4 = 0$ *and* $8 \bmod 7 = 1$

So we found a method to do modular addition for RNS. A similar method can also be applied for modular subtraction.

4.5 Modular Subtraction in RNS

Modular subtraction can also be solved with comparison in RNS. However the difference between modular addition and modular subtraction is that modular subtraction can result in an underflow instead of an overflow. An underflow happens when the result of normal subtraction in RNS gets smaller than 0, which happens when you subtract a larger number from a smaller number. There are two ways to deal with this underflow for modular subtraction $a - b \pmod z$:

1. Do normal RNS subtraction of a and b. After RNS subtraction you check whether the result of this subtraction is smaller than z. If the result is smaller than z no underflow has happened. If it is larger or equal to z an underflow has happened. In this case we add z to the result of the subtraction. This method works when the product of moduli used is larger than $2z - 2$.

2. Do normal RNS addition of a and z. After this subtract b from the result in RNS. Then check if the final result is smaller than z. If it is smaller than z, you can just return the final result. If not, you have to subtract z from the final result.

The first method need fewer normal additions and subtractions for RNS than the second method. However we still prefer the second method, because when we use the second method we can sometimes skip the comparison part which will be explained in section 5.2. The lower bound before comparison for this method is 1 which can be obtained when the smallest number 0 in mod z is subtracted from the highest number $z-1$ in mod z. The result of the method in this case is:

$$0 + z - (z - 1) = 1$$

The upper bound before comparison occurs when you subtract the largest number $z-1$ from the lowest number 0, which results in:

$$(z - 1) + z - 0 = 2z - 1$$

Therefore this method can be used when the product of moduli m for the residue number system is larger than $2z-1$. The pseudocode for this modular subtraction is summarized below:

Algorithm 8 Modular Subtraction for RNS

```
function modularSubtractionRNS(r1[], r2[], z[], m[])
 1: r3 := new array(m.length)          ▷ The RNS number after subtraction
 2: for i := 1; i ⩽ m.length; i++ do
 3:     r3[i] := (r1[i] + z[i] - r2[i]) % m[i]
 4: end for
 5: r4 := new array(m.length)                    ▷ Subtracted by z result
 6: for i := 1; i ⩽ m.length; i++ do
 7:     r4[i] := (r3[i] - z[i]) % m[i]
 8: end for
 9: if isSmallerRNS(r3, z, m) then
10:     return r3
11: else
12:     return r4
13: end if
```

r2 are the residues of the number that is subtracted from the RNS number with the residues of array r1. z are the residues of the main modulus and m are the moduli used for the residue number system. With this algorithm we have figured out how to do modular subtraction in RNS. And we already have figured out how to do modular addition in RNS. Modular multiplication in RNS is much harder to understand. For this we first need to understand how base extension and division by a modulus of the residue number system works.

4.6 Base Extension

Base extension in the residue number system is a method to include an additional modulus w in a residue number system with the corresponding residue v. This modulus w must be coprime to the other moduli m_i of the residue number system according to the requirements defined in section 3.1. Base extension can be very useful, since it can make some computations easier with an auxiliary base. Moreover determining this corresponding residue of this auxiliary base is easy in mixed radix representation, because:

$$
\begin{aligned}
v &= \left(a_1 + a_2 m_1 + \cdots + a_n \prod_{i=1}^{n-1} m_i \right) \bmod w \\
&= \left(a_1 \bmod w + (a_2 m_1) \bmod w + \cdots + \left(a_n \prod_{i=1}^{n-1} m_i \right) \bmod w \right) \bmod w \\
&= \left(a_1 \bmod w + (a_2 \bmod w) \cdot (m_1 \bmod w) + \cdots + (a_n \bmod w) \cdot \left(\prod_{i=1}^{n-1} m_i \right) \bmod w \right) \bmod w
\end{aligned}
$$

Therefore the main method to do a base extension for RNS would be to convert the residue number system to mixed radix, do the modular reduction on the bases and the digits in mixed radix and take the modular sum of these values. The modular sum of these values will be the corresponding residue for the new base w. The pseudocode for this procedure can be found below:

Algorithm 9 Base Extension for RNS

```
function baseExtensionRNS(r[], m[], w)
 1: d := RNSToMR(r, m)
 2: m2[] := array(m.length)              ▷ The mixed radix bases modulo w
 3: m2[1] := 1
 4: for i := 2; i ⩽ m.length; i++ do
 5:     m2[i] := m[i - 1] * m2[i - 1]
 6:     m2[i] %= w
 7: end for
 8: v := 0
 9: for i := 1; i ⩽ m.length; i++ do
10:     v += d[i] * m2[i]
11:     v %= w
12: end for
13: return v
```

r is an array with the residues of the residue number system, m is the array of moduli of the residue number system and w is the new base. This pseudocode will return the residue of the new base w. A new array of residues can be written as $(r_1, r_2, \ldots, r_n, v)$ and the new array of moduli can be written as $(m_1, m_2, \ldots, m_n, w)$.

4.7 Exact Division

The definition of exact division and how to do exact division in RNS is also explained in "An RNS Montgomery Modular Multiplication Algorithm". We use the same definition of 'exact division', however we use a slightly modified version of this exact division algorithm. 'Exact division' in RNS is defined as division by one of the moduli used in the residue number system. Exact division of the RNS number $(a_1, a_2, \ldots, a_n)$ by modulus m_i from the set of RNS moduli $(m_1, m_2, \ldots, m_n)$ is only possible if $a_i = 0$. Otherwise the number is not a multiple of m_i and in that case exact division not possible. Furthermore the product of moduli m also needs to be larger or equal to $\max_i m_i^2$ for this particular method, i.e.

$$m \geqslant \max_i m_i^2$$

where $\max_i m_i$ is the largest residue used. If these requirements are fulfilled then the residues of the result after division $(b_1, b_2, \ldots, b_n)$ can be easily determined for all residues except b_i, since

$$b_j = \left(a_j \cdot |m_i^{-1}|_{m_j}\right) \mod m_j \qquad \text{for } j \neq i$$

where $|m_i^{-1}|_{m_j}$ is the inverse of m_i in m_j. Computing b_i is a lot harder, since only a_i does not give us any information about b_i. If you multiply x mod y by y you always get 0 mod y independent of the value of x. So to find b_i we need the other residues. With base extension we can recover this lost residue from the other divided residues b_i with $i \neq j$. However we use a different method of base extension: we convert the residue number system back to $\mathbb{Z}_m$ and do the modular reduction in $\mathbb{Z}_m$. This will result in the following formulas for b_i:

$$b_i = \left| \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} \right|_{M_i} \right|_{m_i} = \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - \alpha M_i \right|_{m_i}$$

We prefer to use the right formula, since α can be computed with:

$$\alpha = \left| |M_i|_t^{-1} \left(\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - u \right) \right|_t$$

where t is an extra modulus for the residue number system, such that t is coprime to the other moduli of the residue number system and $t \geqslant n$ where n is the number of moduli in the residue number system. u is the residue of this modulus t after division by m_i, which can be found with the base conversion algorithm which is explained in section 4.6. M_i is the product of all moduli except m_i, i.e. $M_i = \prod_{j \neq i} m_i$ and $M_{i,j}$ is the product of all moduli except m_i and m_j, i.e.

$$M_{i,j} = \prod_{\substack{k \neq i \\ k \neq j}} m_k$$

We also still use $|x|_y^{-1}$ as definition for the inverse of x modulo y. And $|x|_y$ is defined as $x \bmod y$. More details why these formulas are correct can be found below in the form of a proof.

Claim.

1. *All new residues except the ith residue is correct, i.e.*

$$b_j = \left(a_j \cdot |m_i|_{m_j}^{-1}\right) \bmod m_j \qquad \text{for } j \neq i$$

2. *The ith residue is correct, i.e.*

$$b_i = \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - \alpha M_i \right|_{m_i}$$

$$\alpha = \left| |M_i|_t^{-1} \left(\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - u \right) \right|_t$$

Proof.

1. $|m_i|_{m_j}^{-1}$ is well defined since m_i and m_j are coprime, thus we can take the inverse of m_i modulo m_j. Moreover when we multiply the residue b_j by m_i we get:

$$b_j \cdot m_i = \left(a_j \cdot |m_i|_{m_j}^{-1} \bmod m_j\right) \cdot m_i = a_j$$

   The only number that results in a_j when being multiplied by m_i is $\left(a_j \cdot |m_i|_{m_j}^{-1}\right) \bmod m_j$, because m_i is not a zero divisor in $\mathbb{Z}_{m_j}$. Hence all b_j are correct residues after exact division.

2. Part of this proof is found due to the master thesis "RNS support for RSA cryptography" [14, section 4.1.1, page 28] and due to section 3.3 of this report. As explained in section 3.3 if we want to convert an RNS number with residues $(b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n)$ and moduli $(m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n)$ to $\mathbb{Z}_{M_i}$ we use the following formula:

$$\left| \sum_{j \neq i} b_j M_{i,j} |M_{i,j}|_{m_j}^{-1} \right|_{M_i}$$

   this formula can be rewritten as:

$$\left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} \right|_{M_i}$$

   With this formula we get the corresponding number in $\mathbb{Z}_{M_i}$. Therefore if we reduce this result modulo m_i, we receive b_i, i.e.

$$b_i = \left| \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} \right|_{M_i} \right|_{m_i}$$

Since $m \geqslant \max_i m_i^2$ we know that $M_i \geqslant \max_i m_i$, therefore this formula will result in b_i. The middle modular reduction can also be written as α times M_i subtracted from this number, where α is an integer:

$$b_i = \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - \alpha M_i \right|_{m_i}$$

According to the Shenoy and Kumaresan theorem [14, section 4.1.1, page 28] this α can be determined with an extra modulus t coprime to the other moduli in the residue number system and corresponding residue u, because for u likewise b_i we know that:

$$\begin{aligned}
u = \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - \alpha M_i \right|_t \implies \\
|\alpha M_i|_t = \left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - u \right|_t \implies \\
|\alpha|_t = \left| |M_i|_t^{-1} \left(\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - u \right) \right|_t
\end{aligned}$$

Moreover $|\alpha|_t = \alpha$, because:

$$\begin{aligned}
\left| \sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} \right|_{M_i} \geqslant 0 \implies \\
\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - \alpha M_i \geqslant 0 \implies \\
\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} \geqslant \alpha M_i \implies \\
\alpha \leqslant \frac{\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j}}{M_i}
\end{aligned}$$

We can moreover derive that:

$$\alpha \leqslant \frac{\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j}}{M_i} \leqslant \frac{\sum_{j \neq i} M_{i,j} m_j}{M_i} \leqslant \frac{\sum_{j \neq i} M_i}{M_i} \leqslant n - 1$$

Because t is selected such that it is greater or equal to n, we can finally conclude that $\alpha < t$. Hence $|\alpha|_t = \alpha$. From this we can derive that:

$$\alpha = \left| |M_i|_t^{-1} \left(\sum_{j \neq i} M_{i,j} \left| b_j |M_{i,j}|_{m_j}^{-1} \right|_{m_j} - u \right) \right|_t$$

We have proven now that claim 2 is true, because we have proven that both formulas are correct. □

A summary of this exact division algorithm in pseudocode form can be found below. Part of this code can be precomputed which is explained in section 5.3.

Algorithm 10 Exact Division for RNS

```
function exactDivisionRNS(r[], m[], d, u, t)
 1: r2 = array(r.length)                          ▷ The result of exact division
 2: for j in {1, 2, ..., r.length} - {d} do
 3:     r2[j] := (r[j] * inverse(m[d], m[j])) % m[j]
 4: end for
 5: u := (u * inverse(m[d], t)) % t
 6: prod = 1;                                     ▷ Determine the values of M_i
 7: for j in {1, 2, ..., r.length} - {d} do
 8:     prod *= m[j]
 9: end for
10: prods = array(r.length)                       ▷ Determine the values of M_{i,j}
11: for j in {1, 2, ..., r.length} - {d} do
12:     prods[j] := prod / m[j]
13: end for
14: alpha := 0                                    ▷ Determine the value of alpha
15: for j in {1, 2, ..., r.length} - {d} do
16:     sumpart := (r2[j] * inverse(prods[j], m[j])) % m[j]
17:     alpha := (alpha + sumpart * prods[j]) % t
18: end for
19: alpha := (inverse(prod, t) * (alpha - u)) % t
20: r2[d] := 0                                    ▷ Determine the last residue
21: for j in {1, 2, ..., r.length} - {d} do
22:     sumpart := (r2[j] * inverse(prods[j], m[j])) % m[j]
23:     r2[d] += (sumpart * prods[j]) % m[d]
24: end for
25: r2[d] := (r2[d] - alpha * prod) % m[d]
26: return r2, u
```

r in this pseudocode is an array of residues which gets divided, m is the corresponding array of moduli and d is the index of the modulus m[d] which is the divisor. t is the additional modulus and u is the corresponding residue for modulus t before division of m[d]. The running time of this algorithm is dependent on the input used, because the running time of the Euclidean inverse operation is dependent on the input used. Nevertheless it does not violate the requirement in section 2.4, since only moduli of the residue number system and the additional modulus are used for the Euclidean inverse operation and these numbers are public information. Therefore this procedure is suitable for CSIDH operations, especially modular multiplication in RNS.

4.8 Modular Multiplication in RNS

For modular multiplication in RNS, we use the Montgomery modular multiplication algorithm for RNS. This algorithm is also described in the article "An RNS Montgomery Modular Multiplication Algorithm" [15]. It is a fast method to use for modular multiplication in RNS, however there are a few requirements for this method:

1. The product of the moduli m should be larger than $3z \cdot \max_i m_i$, i.e.

   $$m > 3z \cdot \max_i m_i$$

   where $\max_i m_i$ is the maximum individual modulus used for the residue number system and z is the main modulus used for modular multiplication.

2. The main modulus z should be coprime to the product of moduli m. Since the modulus p used for CSIDH is a prime, we know that this requirement is always satisfied.

3. The product of the moduli m should be larger or equal to the square of the largest modulus, i.e.

   $$m \geqslant \max_i m_i^2$$

   This requirement is always satisfied if you use multiple moduli which are about the same size.

If we meet all these requirements then we can do Montgomery multiplication for RNS. Montgomery multiplication is a function $f(a, b, z)$ which uses three RNS numbers a, b and z in the same residue number system. And it will compute the following value in RNS:

$$f(a, b, z) = (a \cdot b \cdot m^{-1}) \bmod z$$

z is the main modulus in this function and m^{-1} is the multiplicative inverse of the product of moduli modulo z. With this function we can compute the modular multiplication of two RNS numbers a and b by using this function twice, because:

$$(a \cdot b) \bmod z = f(f(a, b, z), m^2 \bmod z, z)$$

$m^2 \bmod z$ used in this function is a precomputed value. If we use this function twice we indeed obtain the modular multiplication of a and b modulo z, because

$$\begin{gathered} f(f(a, b, z), m^2 \bmod z, z) = \\ (a \cdot b \cdot m^{-1} \cdot m^2 \cdot m^{-1}) \bmod z = \\ (a \cdot b) \bmod z \end{gathered}$$

But how can we execute this Montgomery multiplication function? The algorithm that executes this Montgomery multiplication function is described on the next page.

Algorithm 11 Montgomery Multiplication for RNS

function MONTGOMERYMULTIPLICATIONRNS(mr[], r[], m[], z[], t, u, v)

```
1:  r3 = array(r.length)                    ▷ Initialize an array filled with zeros
2:  r4 = 0
3:  for i := 1; i ⩽ r.length; i++ do
4:      q := (r3[i] + mr[i] * r[i]) * inverse(m[i] - z[i], m[i]) % m[i]
5:      for j := 1; j ⩽ r.length; j++ do
6:          r3[j] += (mr[i] * r[j] + q * z[j]) % m[i]
7:      end for
8:      r4 += (mr[i] * u + q * v) % t
9:      r3, r4 = exactDivisionRNS(r3, m, i, r4, t)
10: end for
11: return r3
```

This Montgomery multiplication algorithm will use input mr[] which is the mixed radix representation of the first number used for multiplication. Moreover r[] is used as input, which are the RNS residues of the second number. m[] is the array of moduli for this residue number system. z[] are the RNS residues of the main modulus. t is the additional base, with u the additional residue for this base for the second number and v the additional residue for this base for the main modulus. Proof of this algorithm can be found in 'An RNS Montgomery Modular Multiplication Algorithm' [15]. When you use this Montgomery multiplication algorithm for modular multiplication in RNS, you get the following algorithm:

Algorithm 12 Modular Multiplication for RNS

function MODULARMULTIPLICATIONRNS(r[], r2[], m[], z[], t, msq[])

```
1:  mr = RNStoMR(r, m)                      ▷ Convert first number to mixed radix
2:  u = baseExtensionRNS(r2, m, t)          ▷ Extra residue for second number
3:  v = baseExtensionRNS(z, m, t)           ▷ Extra residue for main modulus
4:  c = MontgomeryMultiplicationRNS(mr, r2, m, z, t, u, v)
5:  mc = RNStoMR(c, m)                      ▷ Convert the result to mixed radix
6:  m2 = baseExtensionRNS(msq, m, t)        ▷ Extra residue for m^2 (mod z)
7:  return MontgomeryMultiplicationRNS(mc, msq, m, z, t, m2, v)
```

r[] are the residues of the first number, r2[] are the residues of the second number and m[] are the moduli of this residue number system. z[] are the residues of the main modulus in this residue number system. t is an additional modulo that is coprime to the moduli in m[]. The last input msq[] are the residues of m^2 mod z in the residue number system with moduli m[]. When we use this input we get $(a \cdot b)$ mod z in RNS. Hence we have finally solved how to do modular operations in RNS. But how can you do these operations efficiently?

Chapter 5

Operation Implementation

5.1 Efficient Implementation

In chapter 4 we have seen how to do modular arithmetic for RNS. Unfortunately this is a theoretical approach to do modular arithmetic, not an efficient approach. An efficient approach to do modular arithmetic will be discussed in this chapter by showing how certain performance optimizations can be implemented and used. For this optimization we use the architecture of a computer system from chapter 2 including the efficiencies of the computer system architecture. Moreover we use the possible performance gain of doing operations multiple times. This will result in the following improvements of performance which are discussed in the following sections of this chapter:

Skipping Modular Reduction The modular reduction of addition and subtraction can be skipped sometimes when doing a combination of additions, subtractions and multiplications. Why this is possible and how to do this is explained in section 5.2.

Precomputation Some values can be computated in advance. Therefore these values do not have to be determined during every operation. Which values can be precomputed will be discussed in section 5.3.

Moduli Selection If every number fits in a block then the number of operations needed decreases. How I have determined which moduli are used is explained in section 5.4.

Vector Instructions The vector instructions are likewise explained in section 2.2. These instructions can also be used for some operations. Where these instructions can be used is also explained in section 5.5.

5.2 Combination of Modular Operations

To execute different modular operations in RNS, one can simply do all modular operations separately. However this is not very efficient. A more efficient method would be to skip the modulo part of the modular additions and modular subtractions. This means that you directly return $r3$ after line 4 in algorithm 7 for modular addition and directly return $r3$ after line 4 in algorithm 7 for modular subtraction. The reason behind this is that if a modular multiplication is executed then the modular reduction still happens. And if the result of the combination of additions/subtractions did not become too high then the Montgomery multiplication algorithm will still return the correct result. But when is the result too high to use Montgomery multiplication? As mentioned in section 4.8, the requirement for the Montgomery multiplication algorithm is that:

$$m > 3z \cdot \max_i m_i$$

where m is the product of moduli used by the residue number system, $\max_i m_i$ is the maximum modulus used by the residue number system and z is the modulus in which the modular arithmetic happens. In general Montgomery multiplication can deal with numbers up to $y > z$ if

$$m > 3y \cdot \max_i m_i$$

The lowerbound of r modular additions operations of section 4.4 is 0 which is obtained by adding $r + 1$ times the lowest number 0. The upperbound after r modular additions is $(r + 1) \cdot (z - 1)$ which is obtained by adding $r + 1$ times the highest number $z - 1$. The bounds of modular subtractions of to the second method in section 4.5 are different. The lowerbound of r modular subtractions is r which is obtained when the highest number $z - 1$ get subtracted r times from the lowest number 0, i.e.

$$0 + \underbrace{(z - (z - 1)) + \cdots + (z - (z - 1))}_{r} = r$$

and the upperbound is $(r + 1) \cdot z - 1$ which is obtained when the lowest number 0 get subtracted r times from the highest number $z - 1$, i.e.

$$z - 1 + \underbrace{(z - 0) + \cdots + (z - 0)}_{r} = (r + 1) \cdot z - 1$$

Hence we can conclude that the lowerbound for a combination of r modular additions and modular subtractions is 0 and the upperbound for this combination is $(r + 1) \cdot z - 1$ which means that:

Lemma 3. *If x is the result of a combination of r modular additions and modular subtractions in* mod z *then Montgomery multiplication will return the right output for x if*

$$m > 3 \cdot ((r + 1) \cdot z - 1) \cdot \max_i m_i$$

This theorem is the main reason why I prefered to use the second method in section 4.5 instead of the first method for modular subtraction. Otherwise the lowerbound would be lower than 0 which is very problematic for modular operations, since after an underflow it is hard to recover the real result. Therefore the second method is used for modular subtraction, because with the second method you can postpone the modulo operation till the next modular multiplication. However what should you do when there is no next modular multiplication? Which is the case when you algorithm ends with modular additions and modular subtractions. There are two options to deal with this:

1. Do the modular additions and modular subtractions separately, which means you do not skip the modulo operation.

2. Do a Montgomery multiplication of the final result a with $m \bmod z$ at the end where $m \bmod z$ is the product of moduli of the residue number system in $\bmod z$. This is the same as a modulo operation, because

$$(a \cdot m \cdot m^{-1}) \bmod z = a \bmod z$$

Of course the choice will not result in a significant change of performance, since this is only applied for the last modular additions/subtractions and if there are too many modular additions/subtractions before a modular multiplication which does not occur very often. Nevertheless the procedure for these cases should be well defined. In this research I will try both options, since the first option is good if only a few modular additions/subtractions are executed at the end. The second option on the other hand is a better choice if there are a lot of modular additions/subtractions and if you do not know in advance whether a modular multiplication will happen or not.

5.3 Precomputation

The moduli in the residue number system which are used for CSIDH are constants and public information. Likewise the prime p used for CSIDH is a constant and public information, which is used as main modulus. Moreover the same holds for the primes used to compute this p:

$$p = 4 \left(\prod_{i=1}^{74} p_i \right) - 1$$

Every number that is derived from only these constants is likewise a constant value. Therefore it is useless to calculate the derived constants during every operation. You can determine them in advance instead which will reduce the running time of the operations. These precomputed values will then be stored in memory and loaded from memory during a operation when needed. This should be possible, since there are not many precomputed values and the precomputed values are very small. The values that can be calculated of course includes

the moduli of the used residue number system $\{m_1, m_2, \ldots, m_n\}$. Moreover the residues of p in this residue number system can be calculated $\{p_1, p_2, \ldots, p_n\}$. However there are still a lot more values that can be determined in advance. The following values in the following operations can be precomputed:

Mixed Radix Conversion

In mixed radix the inverse of m_i in m_j is used. These inverses $|m_i|_{m_j}^{-1}$ can of course be precomputed for all $1 \leqslant i < j \leqslant n$. Which means that for algorithm 5 line 6 is changed. The variable will get multiplied by a predefined constant.

Base Extension

The additional modulus t used must be coprime to the moduli m_i of the residue number system. This t will therefore be equal to the 75th prime (379), because this is a prime number which is not used in the moduli of the residue number system[1]. Hence t is definitely coprime to all these numbers, because t does not share a prime factor with one of these m_i. Furthermore the 75th prime (379) is larger than the number of moduli used, which is a requirement in section 4.7 for exact division. So the 75th prime is a suitable choice as additional base. With this additional base t we can also determine the mixed radix bases in $\mod t$ in advance, i.e.

$$\begin{gathered} m_1 \bmod t \\ (m_1 \cdot m_2) \bmod t \\ \vdots \\ \left(\prod_{i=1}^{n-1} m_i\right) \bmod t \end{gathered}$$

This will make base extension faster, because line 2 up to 7 can be skipped of algorithm 9. Since m2 is an array that can already be determined in advance.

Exact Division

Precomputation will have a massive influence on the performance of exact division, since a lot of the values used in exact division can be determined in advance. The inverses $|m_i|_{m_j}^{-1}$ for example which are used to determine the new residue r_j after division by m_i with $i \neq j$ can be precomputed. This is pretty similar to mixed radix conversion. However for exact division we need to determine more of these inverses $|m_i|_{m_j}^{-1}$, because we need to calculate all these inverses with $i \neq j$. For mixed radix conversion we only had to precompute these inverses for $i < j$. Moreover the inverses $|m_i|_t^{-1}$ for all i can also be determined in advance where t is the additional modulus. As a result the start of algorithm 10 will become more efficient. However these are not the only precomputations for this

[1] More information how these moduli should be chosen can be found back in section 3.5

algorithm. $M_i \bmod m_i$ for all i can be calculated where M_i is the product of all moduli in the residue number system except m_i. Similarly $M_{i,j} \bmod m_i$ can be determined in advance for all $i \neq j$, where

$$M_{i,j} = \prod_{\substack{k \neq i \\ k \neq j}} m_k$$

These precomputations will have a massive influence on the performance, since a lot of lines can be skipped if this is precomputed. Moreover $M_{i,j}^{-1} \bmod m_j$ can be precomputed for all $i \neq j$. Likewise $M_i^{-1} \bmod t$ and $M_{i,j} \bmod t$ can be determined where t is the additional base. With all these determined values, we can skip line 6 up to 13 in algorithm 10. And line 3, 5, 19, 20, 22, 25, 26 and 28 are changed where numbers will be multiplied by these precomputed constants instead.

Montgomery Multiplication

Also a few values for Montgomery multiplication can be precomputed. The RNS number $m^2 \bmod p$ in the residue number system with moduli $(m_1, m_2, \ldots, m_n)$ is precomputed as mentioned in section 4.8. It does not make sense to compute this value at runtime, because you cannot use the same modular multiplication method in section 4.8, since you need the value $m^2 \bmod p$ for this. Another value that can be precomputed is the inverse of $m_i - p_i$ modulo m_i, since both m_i and p_i are constants. This means that at line 4 of algorithm 11 you will multiply the left value by a constant instead. Moreover the residue of $m^2 \bmod p$ in the additional base t can be precomputed. And last of all the residue of the main modulus p in the additional base t can be precomputed, i.e. $p \bmod t$

Other Operations

The modular addition and subtraction operations which are respectively algorithms 7 and 8 do not directly have values which can be precomputed. Likewise the RNS comparison operation in algorithm 6 does not directly have any value that can be determined in advance. Nevertheless these operations include the mixed radix conversion operation which has precomputed values. Therefore also these operations will become more efficient.

Other Remark

As mentioned earlier in this section every variable is indeed small if we pick all the moduli $(m_1, m_2, \ldots, m_n)$ small. Because a variable is either an inverse or remainder modulo $(m_1, m_2, \ldots, m_n, t)$. For t we know that it is pretty small, since it is the 75th prime number (379). However we have not yet determined the moduli $(m_1, m_2, \ldots, m_n)$ for the residue number system. In the next section I will explain how to choose the right moduli for the residue number system. With these moduli we can precompute the value of all constants in this section.

5.4 Moduli Selection

We have seen the general approach to select the moduli for the residue number system in section 3.5. According to this requirement a modulus m_i must be written as:

$$m_i = 4^{a_{i,0}} \cdot \prod_{l=1}^{74} p_l^{a_{i,l}}$$

where for every $m_j \neq m_i$ it holds for every l that $a_{j,l} = 0 \vee a_{i,l} = 0$ and there should be a l for every i such that $a_{i,l} \neq 0$ otherwise a modulus is equal to 1. Moreover for every l including 0 there should be a j such that $a_{j,l} \neq 0$, because every prime p_l and 4 must be contained in one modulus. With these constraints we have still a lot of possible options left. In this section I will try to find the best option in such way that it is the most friendly set of moduli to use for computer systems. For this we take 32 bit computer systems into account, which means that it uses 32 bit blocks. Therefore moduli should be smaller than 2^{31}, because then every modulus fits into 1 block and addition will not result in an overflow which makes computations a lot easier. Moreover there should be at least 17 moduli if they are 31 bit size. Since at least 17 moduli of 31 bits are needed to make the product m of moduli larger than the main modulus p which is 512 bits large. But to do Montgomery multiplication the product of moduli should be larger than:

$$m > 3p \cdot \max_i m_i$$

which means that at least 18 moduli of 31 bits should be used, because the product of m with 17 moduli can be at most $p \cdot \max_i m_i$ which is smaller than $3p \cdot \max_i m_i$. Therefore we should have at least 18 moduli. But is it possible to use at most 18 moduli? My approach to find a computer friendly set of moduli will therefore consists of the following steps:

1. Check what the smallest possible number of moduli is such that the product of the moduli m is larger than $3p \cdot \max_i m_i$. We define the length of the smallest possible moduli set that satisfies this condition as n.

2. Try to find a set of n moduli such that the product of the moduli is maximized, where p_i and 4 may be used multiple times in the same modulus.

The second question can be solved with linear programming, even though it does not look as a linear problem. However we can use logarithms to make this problem linear. The only thing that is left is to find the smallest possible number of moduli n. This number can be guessed. We start with $n = 18$ and try to find a valid solution to the second question. If such valid solution cannot be found then we increment n until we find a valid solution.

Linear Programming

In this linear programming problem we define $p_0 = 4$ to have a more convenient notation. Now every modulus m_i can be written as:

$$m_i = \prod_{j=0}^{74} p_j^{a_{i,j}}$$

The goal is to maximize the product of moduli m, which is:

$$m = \prod_{i=1}^{n} m_i = \prod_{i=1}^{n} \prod_{j=0}^{74} p_j^{a_{i,j}}$$

This is the same as maximizing the following goal function:

$$\log_2 \left(\prod_{i=1}^{n} \prod_{j=0}^{74} p_j^{a_{i,j}} \right)$$

because the base 2 logarithm is a strict increasing function. Therefore a larger number is also larger in base 2 logarithm. But this goal property has the nice property that we can change it to a linear objective function, because we can rewrite above goal function as:

$$\sum_{i=1}^{n} \sum_{j=0}^{74} a_{i,j} \cdot \log_2(p_j)$$

This is a linear goal function since $\log_2(p_j)$ is a constant. In the linear program we define the following variables:

1. $a_{i,j}$ which represents how often p_j is contained in modulus m_i. This is an integer variable.

2. $c_{i,j}$ which represent whether p_j is contained in modulus m_i. This is a boolean variable (integer variable in range $[0, 1]$).

We have the following constraints for this problem:

Variable domains The number $a_{i,j}$ must be larger or equal to 0. Which means that $a_{i,j} \geqslant 0$ for all i and j. The number $c_{i,j}$ must be larger or equal to 0 and smaller or equal to 1. Which means that $c_{i,j} \geqslant 0$ and $c_{i,j} \leqslant 1$

Meaning of contains $a_{i,j}$ may only be bigger than 0 if $c_{i,j}$ is 1. This result in the following requirement $a_{i,j} \leqslant 31 \cdot c_{i,j}$. If $c_{i,j}$ is 0 then $a_{i,j}$ is indeed 0. If $c_{i,j}$ is 1 then this requirement cannot be violated, because the order of p_i in a number can never be larger than 31, since then it is larger than $p_i^{31} > 2^{31}$ which is the maximum size of a valid modulus. This constraint can be rewritten as $31 \cdot c_{i,j} - a_{i,j} \geqslant 0$. Moreover if contains is equal to 1 then the $a_{i,j}$ should be at least 1. This constraint can be written as $a_{i,j} \geqslant c_{i,j}$ which is the same as $a_{i,j} - c_{i,j} \geqslant 0$

Coprime Condition Every p_j is contained in exactly 1 modulus, which means that for every j:

$$\sum_{i=1}^{n} c_{i,j} = 1$$

Not 1 Condition No modulus can be equal to 1. This means that for every i:

$$\sum_{j=0}^{74} c_{i,j} \geqslant 1$$

Modulus Limit The most important condition is that every modulus is smaller than 31 bits, which means that for every i:

$$m_i = \prod_{j=0}^{74} p_j^{a_{i,j}} \leqslant 2^{31} - 1$$

This condition is satisfied when:

$$\log_2 \left(\prod_{j=0}^{74} p_j^{a_{i,j}} \right) \leqslant \log_2(2^{31} - 1)$$

Hence we can rewrite this as a linear condition:

$$\sum_{j=0}^{74} a_{i,j} \log_2(p_j) \leqslant \log_2(2^{31} - 1)$$

We have now found a linear optimization problem, but we have not yet obtained an answer to this problem. Solving this problem with a computer will take a very long time, because unfortunately linear programming with integers is an NP-complete problem and this formulation of the problem contains 2700 integer variables. Therefore it is very hard to solve this linear program with a computer. Hence in the computer model we need to optimize certain things. First of all the second coprime condition is not required, because an optimal solution will definitely not have a modulus equal to 1, since then additional prime factors could be added to that modulus to make it larger (for a small set of moduli). Also the right hand side of the modulus limit constraint can be adjusted to 31 instead of $\log_2(2^{31} - 1)$, because the product of moduli will never be equal to 2^{31}. Since the prime factorization of every number including 2^{31} is unique. Hence we can only use the factor 4 to get 2^{31}. However if we only use 4 as factor then we get 2^k where k is an even exponent. Thus the product of moduli cannot be equal to 2^{31}. Therefore the right hand size of the modulus limit constraint can be changed to 31. Furthermore we put a time limit of 1 hour on this computer model, because approximated optimal solutions are still useful. The approximated optimal moduli can be found in appendix B. Unfortunately this set of moduli is not appropriate in practice, therefore I used the largest

47 primes below 2^{14}. Since 47 moduli and the additional moduli then exactly fit into 3 vectors of size 16 (this makes it more easier to do vector operations which will be explained in the next section)[2]. Now we have obtained a suitable set of moduli for computer systems which improves the performance of RNS by taking the architecture of computer systems into account. But if we take the architecture of computer systems into account then we can also find other ways to improve the performance of RNS.

5.5 Efficient Usage of Computer Systems

We have seen in section 2.2 that computer systems are designed to be efficient. So why should you not make use of this? Vector instructions for example are very useful, because you can execute the same operation multiple times at the same moment. This type of parrallelism can be used for RNS, because addition, subtraction and multiplication is just a pairwise operation for all residues. The result of these operations for a pair of residues does not dependent on the other residues. Therefore these operations can be executed in parallel for all pairwise residues. Without vector instructions it would require 48 additions/subtractions/multiplications to add/subtract/multiply two RNS elements, because we have a residue number system with 48 moduli. With vector instructions that can do 16 operations at the same time, it will only require 3 operations to add/subtract/multiply two RNS elements. Vector instructions will therefore make addition, subtraction and multiplication of two RNS elements 16 times faster in this case. So vector instructions are a huge benefit for RNS which will therefore also be used in this research to optimize RNS. For example you can use vector instructions in the modular addition algorithm and the modular subtraction algorithm at line 2 up to 4 and line 6 upto 8 of both algorithms (respectively algorithm 7 and algorithm 8). But also other algorithms can be optimized with vector instructions:

Mixed Radix Conversion

The algorithm for Mixed Radix Conversion is entirely changed when using vector instructions. The formulas to determine the mixed radix representation are:

$$
\begin{aligned}
d_1 &= r_1 \bmod m_1 \\
d_2 &= (r_2 - d_1) \cdot |m_1^{-1}|_{m_2} \bmod m_2 \\
d_3 &= ((r_3 - d_1) \cdot |m_1^{-1}|_{m_3} - d_2) \cdot |m_2^{-1}|_{m_3} \bmod m_3 \\
&\vdots \\
d_n &= (\dots((r_n - d_1) \cdot |m_1^{-1}|_{m_n} - d_2) \cdot |m_2^{-1}|_{m_n} - \dots - d_{n-1}) \cdot |m_{n-1}^{-1}|_{m_n} \bmod m_n
\end{aligned}
$$

[2]The approach in section 3.5 is not mandatory to use. The only additional requirement for the moduli is that they should not contain p or the 75th prime number (379) as prime factor. These requirements are satisfied if we use the largest 47 primes below 2^{14} as moduli.

If you read these formula's from left to right then the structure is very clear. First there will be a subtraction then a multiplication then a subtraction etcetera. So vector instructions might be suitable. However there are two problems:

1. To compute lower digits of the mixed radix representation you need less subtractions and multiplications.

2. Higher digits of the mixed radix representation contain lower digits in their formula.

The first problem can be solved with so called mask vector instructions. In masked vector instructions you use an additional vector as parameter of the same size as the other vectors. This vector contains bits and at each position with a 1 bit the vector operation is executed and at the positions with a 0 bit no vector operation is executed. So for earlier digits we change the corresponding bit in the mask vector to 0 at some time when the computations are done for that digit. The second problem can be solved by a vector instruction that creates a vector with the same value at all indices in the vector. When you need the value of a lower digit for the computation of the higher digits then you just create a vector with this value and do the operation on that vector. Hence we use vector instructions for Mixed Radix Conversion and determine all digits by going through the formulas from left to right.

Other Operations

- For smaller than comparison you can compute (d1[i] == d2[i]) for all i with vector instructions before starting the for loop at line 5 in algorithm 6. Likewise you can compute (d1[i] < d2[i]) for all i with vector instructions at line 6 in algorithm 6.

- For base extensions you can compute d[i] * m2[i] for all i with vector instructions in advance at line 10 of algorithm 9.

- The sumpart at line 16 and 22 can be computed in advance for all j with vector instructions in the algorithm 10.

- In algorithm 11 you can parrallelize line 5 up to 7 with vector instructions for all j.

Chapter 6

Performance of RNS

6.1 Old Implementation of CSIDH

The old implementation of CSIDH represents numbers in $\mathbb{Z}_p$ and similarly to the new implementation the old implementation also uses Montgomery multiplication to do modular multiplication. Moreover for modular addition and modular subtraction the old implementation also uses comparisons similar to the new implementation. I measured the number of clock ticks needed to do the same operation a number of times. The upper cells in the columns indicate how many times the same operation was executed and the left cells in the rows indicate which operation was executed.

	1	2	5	10	20	50	100	200	500	1000
Modular Addition	0	0	0	0	0	0	0	0	0	1
Modular Subtraction	0	0	0	0	0	0	0	1	0	0
Modular Multiplication	0	0	0	0	1	0	1	1	2	4

	2000	5000	10000	20000	50000	100000
Modular Addition	1	0	1	3	6	12
Modular Subtraction	0	1	1	2	4	11
Modular Multiplication	7	17	38	80	166	347

Table 6.1: Old Implementation Results

As you can see, the old implementation for CSIDH is very efficient. The duration of 10000 modular additions/subtractions is negligible. And for modular multiplication the running time is negligible below 200 operations. It is therefore very hard for the new implementation to outperform this performance.

6.2 RNS Implementation of CSIDH

For the CSIDH operations I have created an RNS implementation prototype. This implementation might still contain some errors and is not entirely optimal. Nevertheless it gives an indication about how RNS performs. In this implementation I used vector operations and precomputed constants. The number of clock ticks needed for modular addition and modular subtraction can be found below where a reduction takes place after every addition and subtraction:

	1	2	5	10	20	50	100	200	500	1000
Modular Addition	0	1	1	4	4	10	20	43	121	258
Modular Subtraction	1	1	1	3	4	12	17	51	104	191

Table 6.2: RNS Implementation: Clock Ticks (Multiple Reductions)

More than 1000 modular additions and modular subtractions take a lot of time. Hence these results are not measured. Nevertheless for these modular additions and modular subtractions I also have measured how many basic operations are needed to perform these modular additions and subtractions:

	1	2	5	10	20	50
Modular Addition	11076	22152	55380	110760	221520	553800
Modular Subtraction	11094	22188	55470	110940	221880	554700

	100	200	500	1000
Modular Addition	1107600	2215200	5538000	11076000
Modular Subtraction	1109400	2218800	5547000	11094000

Table 6.3: RNS Implementation: Basic Operations (Multiple Reductions)

The performance of these modular additions and subtractions is much worse than the old implementation. However there is also a second method to do modular additions and subtractions, namely skipping the modular reductions and do a Montgomery multiplication after all additions and subtractions. But then you should not exceed the upperbound on the number of additions and subtractions which can be found by the lemma in section 5.2. Fortunately we do not have to worry about this upperbound, because with our set of moduli we can do up to $8,5 \cdot 10^{41}$ additions and subtractions. The number of clock ticks required to do modular additions and subtractions using this method is:

	1	2	5	10	20	50	100	200	500	1000
Modular Addition	1	1	1	1	1	1	1	1	3	1
Modular Subtraction	0	1	1	0	1	0	0	1	1	2

Table 6.4: RNS Implementation: Clock Ticks (Single Reduction)

We can also measure the number of clock ticks for a larger number of operation, since these modular additions and subtractions are quite fast:

	2000	5000	10000	20000	50000	100000
Modular Addition	1	5	6	12	29	65
Modular Subtraction	3	5	8	15	34	49

Table 6.5: RNS Implementation: Clock Ticks (Single Reduction)

The number of clock ticks for modular additions and subtractions is quite low. Unfortunately it is still higher than the number of clock ticks for the old implementation, but not much higher. Also the number of basic operations for modular addition and subtractions is quite low which can be found in the tables below:

	1	2	5	10	20	50
Modular Addition	36137	36152	36197	36272	36422	36872
Modular Subtraction	36155	36188	36287	36452	36782	37772

	100	200	500	1000	2000	5000
Modular Addition	37622	39122	43622	51122	66122	111122
Modular Subtraction	39422	42722	52622	69122	102122	201122

	10000	20000	50000	100000
Modular Addition	186122	336122	786122	1536122
Modular Subtraction	366122	696122	1686122	3336122

Table 6.6: RNS Implementation: Basic Operations (Single Reduction)

As you also might notice, if you do more than 5 modular additions, subtractions then a single Montgomery multiplication at the end is faster than a modular reduction after every addition and subtraction. So if we use the RNS implementation of CSIDH then we prefer this method for modular additions and subtractions. Of course we only do this modular reduction at the end of the code if the code does not end with a modular multiplication. If the code ends with a modular multiplication then we do not have to modular reduce the result, because modular multiplication will automatically do a modular reduction. In this case modular addition and subtraction is even faster. Unfortunately postponing modular reduction is not possible for modular multiplication. Therefore there is only one version of modular multiplication in the RNS implementation. The number of clock ticks for this version are:

	1	2	5	10	20	50	100	200	500	1000
Modular Multiplication	1	2	4	11	16	40	81	206	431	916

Table 6.7: RNS Implementation: Clock Ticks

These results show that the prototype implementation for RNS multiplication is not very efficient. It takes almost a second to do 1000 modular multiplications, which is definitely too long and less efficient than the old implementation. And the number of basic operations needed to do modular multiplication is also large which is shown in the tables below:

	1	2	5	10	20	50
Modular Multiplication	77719	155438	388595	777190	1554380	3885950

	100	200	500	1000
Modular Multiplication	7771900	15543800	38859500	77719000

Table 6.8: RNS Implementation: Basic Operations

Chapter 7

Conclusion

In chapter 6 we have seen that the prototype RNS implementation of the CSIDH operations was not very efficient. My most efficient implementation of modular addition and subtraction operations in RNS were slower than the modular addition and subtraction operation in the old implementation. Moreover the implementation of modular multiplication in RNS took much more time than the old implementation for modular multiplication. Hence my prototype implementation was definitely not efficient. Nevertheless I do not conclude that RNS cannot make CSIDH operations faster. My prototype RNS implementation was not very efficiently designed, because I did not manage to:

1. Use vector operations for 32/64 bit integers
2. Do 32 operations for 16 bits at once instead of 16 operations for 16 bits at once
3. Efficiently check for overflow/underflow after addition/subtraction/multiplication
4. Optimize the for loops and memory usage
5. Use vector operations at more places in the code

I believe that if you optimize the prototype RNS implementation that it can actually become faster than the old implementation, because RNS is very suitable for parallelization and multiplication in RNS is really fast. And the slowdowns of RNS: mixed radix conversion and Montgomery multiplication can also be parallelized which makes these operations less of a trouble. Therefore RNS has a lot of potential of making CSIDH a lot more efficient. I did not manage to make CSIDH more efficient, but others who have more knowledge about computer systems will definitely be able to change this prototype in a more efficient one.

Bibliography

[1] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: An Efficient Post-Quantum Commutative Group Action." Cryptology ePrint Archive, Report 2018/383, 2018. https://eprint.iacr.org/2018/383.

[2] I. Englander, *The Architecture Of Computer Hardware, Systems Software, & Networking*. John Wiley & Sons, 5 ed., February 2013.

[3] "_mm512_add_epi32/_mm512_mask_add_epi32." https://software.intel.com/en-us/node/523404, August 2015. Retrieved 23 October 2018.

[4] P. Z. Richard P. Brent, *Modern Computer Arithmetic*. Richard P. Brent, Paul Zimmermann, 0.2 ed., June 2018.

[5] "Arithmetic." https://web.archive.org/web/20060820053803/http://magma.maths.usyd.edu.au/magma/Features/node86.html, August 2006. Retrieved 27 October 2018.

[6] "SUPERCOP." http://bench.cr.yp.to/supercop.html, August 2018. Retrieved 11 October 2018.

[7] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113, Springer, 1996. https://www.paulkocher.com/doc/TimingAttacks.pdf.

[8] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*. https://eprint.iacr.org/2013/448.pdf.

[9] D. E. Knuth, *The Art Of Computer Programming*, vol. 2. Addison Wesley Longman, 3 ed., 1998.

[10] A. Bogomolny, "Properties of GCD and LCM." http://www.cut-the-knot.org/arithmetic/GcdLcmProperties.shtml. Retrieved 27 September 2018.

[11] A. Cohen, H. Cuypers, and H. Sterk, "Sets, Logic and Algebra," 2012. Dictation of the course: Verzamelingenleer en algebra (2WF40).

[12] H. Xiao, Y. Ye, G. Xiao, and Q. Kang, "Algorithms for comparison in residue number systems," https://ieeexplore.ieee.org/document/7820790.

[13] "Logical AND Operator: &&." https://msdn.microsoft.com/en-us/library/c6s3h5a7.aspx. Retrieved 24 November 2018.

[14] N. Cucu Laurenciu, "RNS support for RSA cryptography," Master's thesis, Delft University of Technology, Delft, The Netherlands, August 2010. https://repository.tudelft.nl/islandora/object/uuid%3A7dd09f0f-254b-42f1-a852-b61cb9f2bed3.

[15] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "An RNS Montgomery Modular Multiplication Algorithm," *IEEE Transactions On Computers*, vol. 47, pp. 766–776, July 1998. https://pdfs.semanticscholar.org/3f90/d53f2c7154212f3b12314cdc6c1b5fecc08b.pdf.

[16] "Free Stalen Sleutel Stock Photo." https://nl.freeimages.com/photo/steel-key-1417279, August 2007. Retrieved 3 September 2018.

[17] D. J. Guan, "Montgomery Algorithm for Modular Multiplication." https://guan.cse.nsysu.edu.tw/note/montg.pdf, August 2003. Retrieved 2 January 2019.

[18] "Intel Intrinsics Guide." https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Retrieved 2 January 2019.

Appendix A

Basic Operations

Below is a list of all operations in the prototype RNS code which are considered as basic operations:

Normal Operations

Operations on 2 blocks of 32 bits integers (or smaller)

- Addition: $c = a + b$
- Subtraction: $c = a - b$
- Lower Multiplication: $c = a \cdot b$
- Modular Reduction: $c = a \,\%\, b$ [1]
- Bitwise And: $c = a \,\&\, b$
- Bitwise Or: $c = a \mid b$
- Equality Check: $a == b$
- Lesser Check: $a < b$

Vector Operations

Operations on vectors with 16 times 16 bit integers:

- Addition: $\vec{c} = \vec{a} + \vec{b}$
- Subtraction: $\vec{c} = \vec{a} - \vec{b}$
- Lower Multiplication: $\vec{c} = \vec{a} \cdot \vec{b}$
- Higher Multiplication: $\vec{c} = \vec{a} \cdot \vec{b}$
- Assignment: $\vec{c} = \vec{a}$
- Bitwise Right Shift: $\vec{c} = \vec{a} >> b$
- Bitwise And Not: $c = \neg a \,\&\, b$
- Bitwise Or: $c = a \mid b$

[1] This is considered as basic operation, although most computer systems use multiple instructions to execute this operation

Appendix B

RNS Moduli

	Value	Factorization
m_1	2105958464	$4^3 \cdot 31^2 \cdot 97 \cdot 353$
m_2	2120612113	$17 \times 61 \times 73 \times 109 \times 257$
m_3	2041520893	$89 \times 269^2 \times 317$
m_4	2121299863	$79 \times 193 \times 373^2$
m_5	2074775855	$5 \times 29 \times 139 \times 311 \times 331$
m_6	2075005411	$179 \times 223 \times 227 \times 229$
m_7	2074856643	$3^2 \times 71 \times 137^2 \times 173$
m_8	2122650059	$23^2 \times 103 \times 163 \times 239$
m_9	2129516219	$19 \times 43 \times 101 \times 131 \times 197$
m_{10}	1880194477	$7 \times 59 \times 107 \times 157 \times 271$
m_{11}	2138929007	$83 \times 281 \times 293 \times 313$
m_{12}	2047565311	$11 \times 37 \times 113 \times 211^2$
m_{13}	2118174911	$149 \times 191 \times 263 \times 283$
m_{14}	2072095579	$53 \times 307 \times 347 \times 367$
m_{15}	2099295377	$167 \times 181 \times 199 \times 349$
m_{16}	2095440997	$13 \times 41 \times 47 \times 233 \times 359$
m_{17}	2128012889	$127 \times 241 \times 251 \times 277$
m_{18}	2001334823	$67 \times 151 \times 337 \times 587$

Table B.1: Previous RNS Moduli

The actual RNS Moduli are the largest 48 primes below 2^{14}