

Sparse and Scalable Modular Arithmetic

by

Benjamin Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Benjamin Chen 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modular arithmetic is a crucial concept in computer algebra and finds extensive use in applications such as cryptography, polynomial GCD computations, and linear system solving. This thesis investigates methods to enhance the efficiency of modular arithmetic by focusing on choosing sparse and scalable moduli. A contribution of this work is the exploration of a balanced binary representation, which provides the sparsest way to represent integers.

Techniques that convert to and from RNS (Residue Number System) using special form moduli (e.g. Mersenne type, Fermat type, and “trinomial” type) are also studied, demonstrating significant speedups over conventional division methods. The thesis also explores different schemes to generate scalable moduli sets. One important result is the discovery of a close relation between the inverses of the moduli in sparse balanced binary form and the inverses under a polynomial setting. This relation allows for the generation of scalable moduli sets with Fermat type numbers.

We test the proposed improvements on modular arithmetic on a two-layer modular arithmetic scheme that leverages scalable moduli to improve the efficiency of RNS (Residue Number System) conversion and modular inverses to show the effectiveness of modular arithmetic with sparse and scalable moduli. Benchmark results demonstrate significant computational advantages achieved through these methods, offering scalable solutions for large integer operations.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Eugene Zima for his continuous support and guidance throughout my academic journey. His patience, encouragement, and immense knowledge have been invaluable to me. Under his supervision, I identified my weaknesses and practiced meticulous academic writing. The past two years have been a struggle for me, but Professor Zima's guidance has been a beacon of light that has helped me navigate through the darkness. I want to thank him for his mentorship to my development as a researcher.

I would also like to thank my other supervisor, Professor George Labahn. Before I started my master's studies, I had the opportunity to work with Professor Labahn on the MathBrush project. His guidance has been instrumental in my academic growth. I am grateful for his support and encouragement. I would also like to thank everyone in the Symbolic Computation Group for their support and friendship.

I would like to thank Professor Stephen New for his academic advice and support. My interests in academia have been greatly influenced by his teachings. Professor New guided me through the time when I was unsure about my future path. The way he teaches has always been inspiring to me. If I had not met Professor New, I probably would not have chosen to pursue further education.

I gratefully appreciate my thesis committee members, Professor Arne Storjohann and Professor Éric Schost, for their time reading my thesis and their constructive feedback.

On a more personal note, I would like to thank my long-time friend Francis Sun since high school. Francis is one of the many friends I first made when I came to Canada. I am fortunate enough to have kept in touch with him over the years. It is rewarding to see how both of us have grown and matured.

Dedication

I dedicate this thesis to my dearest parents, who supported me throughout my academic journey.

I would like to share a famous quote by the great Chinese scholar Zhang Zai, which has supported me through my academic journey:

为天地立心，为生民立命，为往圣继绝学，为万世开太平。

To ordain conscience for Heaven and Earth.

To secure life and fortune for the people.

To continue lost teachings for past sages.

To establish peace for all future generations.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Preliminaries	6
2.1 Sparse Balanced Binary Representation	6
2.2 Scaling	8
3 Conversion to and from RNS	10
3.1 Conversion to RNS	10
3.2 Conversion from RNS	13

4	Searching for Scalable Moduli Sets	16
4.1	Generating Scalable Moduli Sets	16
4.1.1	“Shift” Scheme	17
4.1.2	“Block” Schemes	17
4.1.3	“Trinomial” Scheme	18
4.2	Searching for Inverses	22
4.2.1	Fermat Type Inverses	22
4.2.2	Number of Set Bits of the Inverses	28
5	Implementation	31
5.1	Helper Classes	31
5.1.1	SparseBits	31
5.1.2	MultiplicationByShifting	32
5.2	Modulus	33
5.3	Reconstruction	33
5.4	Bits Extraction	35
5.5	Benchmark	35
5.5.1	RNS Conversion Comparison	36
5.5.2	Two-level Modular Scheme	37
6	Conclusions	41
	References	41
	APPENDICES	45
A	Appendix	46
A.1	Co-primality Results	46
A.1.1	Co-primality of $x^n + 1$ and $x^m - 1$	46
A.1.2	Co-primality of $x^n + 1$ and $x^m + 1$	47
A.2	Justifications of Basic Facts	48

List of Figures

1.1	The general scheme of modular arithmetic using residue number system (RNS)	2
5.1	The class hierarchy diagram	35

List of Tables

3.1	Time complexities of different conversion schemes for converting a to RNS.	13
3.2	Benchmark results of reducing 16 square matrices of dimension 16 where each entry contains a number with n random bits by different moduli. . . .	13
5.1	Timing (in seconds) for the input 32 by 32 matrix conversion to RNS and immediate reconstruction from RNS.	37
5.2	Timing (in seconds) of square integer matrix multiplication benchmark . .	39

Chapter 1

Introduction

Modular arithmetic is a fundamental idea in computer algebra with many useful applications. For example, it is used in factoring, computing greatest common divisors of polynomials, solving systems of linear equations, and more. It is also essential in cryptography, where it is used in RSA, elliptic curve cryptography (ECC), and many other cryptographic schemes.

The general idea of an algorithm using modular arithmetic is to reduce a problem in a ring R to several problems in $R/\langle m_i \rangle$. Given a set of moduli $\{m_1, m_2, \dots, m_n\}$, we can represent any integer in R as a set of remainders $\{r_1, r_2, \dots, r_n\}$. This is also commonly known as the residue number system (RNS). The problems are solved in the domain of $R/\langle m_i \rangle$, and the solutions are reconstructed back to R . This is one of the benefits of the algorithms that use modular arithmetic. For example, intermediate expression swell often occurs when computing the greatest common divisors of polynomials or solving systems of linear equations. By using modular arithmetic, numbers in $R/\langle m_i \rangle$ are bounded by a manageable size.

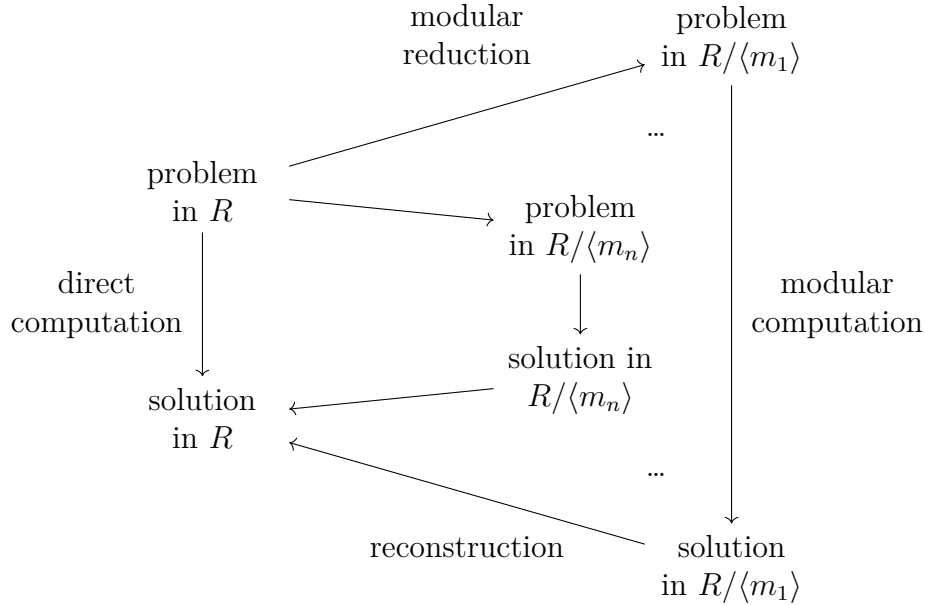


Figure 1.1: The general scheme of modular arithmetic using residue number system (RNS)

The general steps of performing modular arithmetic are as follows. First, modular reduction reduces the size of both a and b . Assume we reduce both a and b to a_1, a_2, \dots, a_n and b_1, b_2, \dots, b_n respectively. Then, the computation is carried out in the domain of $R/\langle m_i \rangle$. Suppose the computation is f . We compute $f(a_i, b_i)$ for all i . Finally, we reconstruct the solution back to R . Denote the cost of reduction and reconstruction to be $R_1(\cdot)$ and $R_2(\cdot)$, and the cost of the computation to be $F(\cdot, \cdot)$. The overall cost of the algorithm using modular arithmetic is given by $R_1(a) + R_1(b) + nF(a_i, b_i) + R_2(a) + R_2(b)$.

Since the cost of the computation, F , is fixed, we aim to further improve the speed of modular arithmetic by minimizing the overhead, $R_1(a) + R_1(b) + R_2(a) + R_2(b)$. In other words, the focus is on minimizing the cost of reduction and reconstruction.

There are two main strategies to achieve the goal of minimizing the overhead. One strategy is to perform the calculations within the machine words and utilize hardware arithmetic [2, 5, 13]. The other strategy is to select special moduli set to reduce the cost of reduction and reconstruction [12, 14, 17]. In this thesis, we focus on the latter strategy.

Consider pairwise co-prime natural numbers m_0, m_1, \dots, m_k . Assume we work with non-negative representatives in each class of residues modulo m_i . The modular reconstruction problem is: given non-negative x_0, x_1, \dots, x_k ($x_i < m_i$), find non-negative $X <$

$m_0 m_1 \dots m_k$ such that $X = x_i \bmod m_i$ for $i = 0, 1, \dots, k$. Standard modular reconstruction algorithms [8] precompute products of moduli $M_i = \prod_{j=0}^{i-1} m_j$ and inverses $M_i^{-1} \bmod m_i, i = 1, \dots, k$. Reconstruction involves several multiplications by these quantities which provide a significant contribution to the total complexity. When selecting a particular approach for choosing moduli m_i , one can try to satisfy the following natural requirements:

1. $\gcd(m_i, m_j) = 1$ for $i \neq j$;
2. reduction modulo m_i is “simpler” than division with remainder;
3. products of moduli and their inverses have a bit pattern (preferably scalable) that allows accelerated multiplication by those quantities;
4. the bit length of moduli m_i is balanced.

The idea of choosing special moduli to accelerate computation is not new. Schönhage [15] showed how to generate scalable relatively prime moduli of Mersenne type. In his scheme requirements 1, 2 and 4 are satisfied, but requirement 3 is not addressed. The idea of using Mersenne numbers as moduli on binary computers to simplify addition, subtraction, and multiplication operations is described again later by Knuth [14]. Fraenkel [1] stated that in a multiplicative group corresponding to a Mersenne prime, multiplication by 2 is equivalent to a cyclic left shift of the binary representation. Hiasat [11] proposed a family of moduli of the form $2^n - (2^p \pm 1)$, which we will call it “trinomial” type moduli. His focus is multiplication within the RNS, restricting the size of the multiplicand to be at most n bits. He proposed a fast residue multiplier for this family of moduli, $2n + p - 2$ partial products were introduced, where the magnitude of each partial product is less than 2^n . The result can be calculated by summing all the partial products. The value of p controls how fast we can compute the partial products. A large p value leads to a slower speed due to the number of iterations needed to reduce the size of the product, and vice versa. In his analysis, the assumption $p \leq n/2$ is made to ensure only two iterations are needed to reduce the sum of the partial products. The simple shape of the moduli mentioned allows for efficient reduction by avoiding divisions when converting to RNS and efficient computation in RNS. That is, requirement 2 is satisfied. The “trinomial” type moduli also satisfy requirement 4, as we can vary the p value to generate balanced moduli in bit length.

However, all of the work leaves requirement 3, the reconstruction part, unattended. An attempt to also address the reconstruction part was made by Zima and Steward [17]. Zima and Steward expanded the choice of modulus to Cunningham numbers of the forms $2^n + 1$ and $2^n - 1$. They proposed two schemes for generating the moduli set, “shift” scheme and

“block” scheme. Due to the simple shape of the moduli, the merits of using Mersenne numbers are retained. In addition to that, they presented a closed-form inverse of the product of moduli in the moduli set. They all improve modular reduction time (replacing division with remainder by shifts and additions/subtractions). Shift-based moduli of the form $m_i = 2^{a2^i} + 1$, $i = 0, 1, \dots, k$ [17], also satisfy requirement 3, as

$$\left(\prod_{j=0}^{i-1} m_j \right)^{-1} \bmod m_i = 2^{a2^i-1} - 2^{a-1} + 1, \quad i = 1, 2, \dots, k \quad (1.1)$$

for arbitrary natural a . However, such choices of moduli do not satisfy requirement 4. In fact, the bit length of m_i is larger than the bit length of product $m_0 m_1 \dots m_{i-1}$. Hiasat [12] generalized two well-studied moduli sets $\{2^p, 2^p - 1, 2^p + 1\}$ and $\{2^p, 2^p - 1, 2^{p-1} - 1\}$ to $\{2^k, 2^p - 1, 2^p + 1\}$ and $\{2^k, 2^p - 1, 2^{p-1} - 1\}$. The closed-form multiplicative inverse of the generalized moduli set is presented. The closed-form inverse makes it possible to not use multiplication at all during reconstruction. These sets of moduli satisfy all the requirements, but the number of moduli in the set is limited to three.

Chen, Li, and Zima [4] proposed a two-layer organization of modular arithmetic. The upper layer uses the special moduli for fast reduction and reconstruction, while the lower layer uses moduli within the word size to perform the computation using hardware arithmetic. The authors achieved significant speed-ups using this two-layer organization compared to the single-layer machine size moduli approach. The authors also discovered that moduli of the form $2^n - 2^k + 1$ outperform those of the form $2^n - 1$ in terms of overhead cost.

In this thesis, we investigate techniques for speeding up modular arithmetic through the use of sparse and scalable moduli.

In Chapter 2, we describe sparse balanced binary representation (SBB), a unique and efficient way to represent integers. We also introduce the concept of scaling in SBB form, allowing the reuse of special moduli sets to represent a larger range of integers.

In Chapter 3, we explore the techniques of converting integers to and from RNS using special moduli, including Mersenne, Fermat, and “trinomial” type numbers. We provide a detailed analysis of conversion algorithms and their time complexities, demonstrating speed-ups using special moduli compared to the regular division-with-remainder method. The results are supported by experimental evidence.

In Chapter 4, we study the conditions that could allow the generation of scalable moduli sets. We study the coprimality conditions of Fermat and “trinomial” type numbers and the two schemes, “shift” and “block” schemes, proposed by Zima and Steward [17] to generate

scalable moduli sets using Fermat type numbers. We propose a close relation between the inverses of the moduli in SBB form and the inverses under polynomial settings. We show that the moduli set with Fermat type numbers are scalable, and the inverses are easy to generate.

In Chapter 5, we implement the proposed acceleration techniques in C++ and provided benchmark results showing the superior performance of a two-layer modular arithmetic scheme using sparse and scalable moduli compared to the single-layer machine word size moduli approach. This demonstrates the effectiveness of using sparse and scalable moduli, particularly for large integer computations, in RNS reduction and reconstruction.

Chapter 2

Preliminaries

In this chapter, we describe the necessary terminology and concepts used throughout the thesis. In Chapter 2.1, we describe the sparse balanced binary representation, also known as non-adjacent form (NAF) or canonical signed digit representation. In Chapter 2.2, we define the concept of scaling, which allows the “reuse” of special moduli sets to represent arbitrary large integers.

2.1 Sparse Balanced Binary Representation

Consider $x \in \mathbb{Z}$ represented as

$$x = \sum_{i=0}^n b_i 2^i \text{ with } b_i \in \{-1, 0, 1\}. \quad (2.1)$$

We call (2.1) the *balanced binary representation*. This representation resembles the *balanced ternary representation* [14]. However, it uses base 2 instead of base 3. It shares many useful properties with balanced ternary representation.

For example,

1. the most significant digit in (2.1) determines the sign of the number x .
2. $x = 0$ if and only if $b_i = 0, i = 0, 1, \dots, n$. Equivalently, if there is a value of index i such that $b_i \neq 0$ then $x \neq 0$.

Definition 2.1.1. We say that i -th bit is **set** in (2.1) if $b_i \neq 0$. A **positively set bit** is called a p -bit, while a **negatively set bit** is called an n -bit and is denoted in the balanced binary representation by the digit $\bar{1}$.

For example, the standard binary representation of 30 is $11110_{(2)}$, which is a possible balanced binary representation. The same number has several different balanced binary representations, such as, $100\bar{1}10$ or $1000\bar{1}0$, with the last one being the sparsest, i.e., having the fewest set bits.

The motivation for using balanced binary representation is as follows: It is well-known that every integer has its unique binary representation. One of the common operations in computational algorithms is multiplication. Multiplication is often implemented using bitwise shift operations and addition. For example, multiplying an integer by $30 = 11110_{(2)}$ requires 4 shifts and 3 additions using a binary representation. However, 30 can be represented in a more efficient way: $100\bar{1}0$, requiring only 2 shifts and 1 subtraction.

The balanced binary representation (2.1) is not unique as demonstrated. However, the uniqueness can be obtained by imposing a simple constraint: no two adjacent digits can have nonzero values simultaneously.

Definition 2.1.2. An integer $x \in \mathbb{Z}$ is in **sparse balanced binary representation (SBB)** if

$$x = \sum_{i=0}^n b_i 2^i \text{ with } b_i \in \{-1, 0, 1\} \text{ and } b_i \cdot b_{i+1} = 0 \text{ for } i = 0, 1, \dots, n-1, \quad (2.2)$$

The additional constraint in balanced binary representation (2.1) —that no consecutive bits are set —makes the representation both sparsest and unique.

We describe an algorithm that converts the binary representation to the SBB representation. The idea of the algorithm is to scan from the least significant bit to the most significant bit, greedily capturing consecutive 1s of length 2 or more and converting them to $10\dots 0\bar{1}$.

The sparse balanced binary representation is known in other literature as the non-adjacent form [9], or canonical signed digit representation [10]. We introduce this new terminology for a known representation, as it emphasizes the sparsity features more naturally within our context.

2.2 Scaling

Finding a single set of special moduli for RNS is not difficult. In fact, heuristic methods can often identify such sets. However, a single set of special moduli can only represent a number within the product of all its elements, provided that the moduli are coprime. It would be ideal if we could develop a way to “reuse” the special moduli set, scaling it to represent an arbitrarily large integer.

Definition 2.2.1. Given x in its SBB form (as in 2.2), we define the **scaling** operation $\text{Scal}(u, x)$ as follows:

$$\text{Scal}(u, x) = \sum_{i=0}^n b_i 2^{ui}.$$

Definition 2.2.2. Let m_1, m_2 be two relatively prime moduli expressed in SBB representation

$$m_1 = \sum_{i=1}^n \alpha_i 2^{k_i} \quad \text{and} \quad m_2 = \sum_{i=1}^m \beta_i 2^{\ell_i}$$

with $\alpha_i, \beta_i \in \{1, -1\}$ and $k_i > k_{i+1}, k_n = 0, \ell_i > \ell_{i+1}, \ell_m = 0$.

Let f and g be inverses

$$f = m_1^{-1} \pmod{m_2}, \quad g = m_2^{-1} \pmod{m_1}$$

in SBB representation:

$$f = \sum_{i=1}^u \gamma_i 2^{a_i} \quad \text{and} \quad g = \sum_{i=1}^v \delta_i 2^{b_i}$$

with $\gamma_i, \delta_i \in \{1, -1\}$ and $a_i > a_{i+1}, b_i > b_{i+1}$.

Now consider scaled moduli M_1, M_2 :

$$M_1 = \sum_{i=1}^n \alpha_i 2^{c k_i} = \text{Scal}(c, m_1), \quad M_2 = \sum_{i=1}^m \beta_i 2^{c \ell_i} = \text{Scal}(c, m_2)$$

with an integer scaling factor $c > 1$. m_1, m_2 are **scalable** if

1. M_1 and M_2 are relatively prime, and

2. when $u \geq 2$ and $v \geq 2$, the inverses $F = M_1^{-1} \pmod{M_2}$ and $G = M_2^{-1} \pmod{M_1}$ have SBB representation

$$F = \sum_{i=1}^u \gamma_i 2^{A_i}, \quad G = \sum_{i=1}^v \delta_i 2^{B_i}$$

with

$$A_i - A_{i+1} = c(a_i - a_{i+1}), i = 1, 2, \dots, u - 2;$$

$$B_i - B_{i+1} = c(b_i - b_{i+1}), i = 1, 2, \dots, v - 2.$$

The scaling operation can be viewed as a method to extend the range of the moduli set. A set of special moduli can be scaled to represent a larger range of numbers. The scaling property is observed in a variety of moduli sets with different shapes, including Fermat type moduli, $(2^n + 1)$, and “trinomial” type moduli, $(2^n - 2^k + 1)$.

Example 2.2.3. Let $m_1 = 2^{100} - 2^{60} + 1, m_2 = 2^{100} - 2^{50} + 1, f = m_1^{-1} \pmod{m_2} = 2^{40} + 2^{30} + 2^{20} + 2^{10} + 1, g = m_2^{-1} \pmod{m_1} = 2^{100} - 2^{60} - 2^{40} - 2^{30} - 2^{20} - 2^{10} + 1.$

For any $c > 1$ $M_1 = 2^{100c} - 2^{60c} + 1, M_2 = 2^{100c} - 2^{50c} + 1$ and $F = M_1^{-1} \pmod{M_2} = 2^{40c} + 2^{30c} + 2^{20c} + 2^{10c} + 1, G = M_2^{-1} \pmod{M_1} = 2^{100c} - 2^{60c} - 2^{40c} - 2^{30c} - 2^{20c} - 2^{10c} + 1.$

Example 2.2.4. Let $m_1 = 2^{432} + 1, m_2 = 2^{324} + 1, f = m_1^{-1} \pmod{m_2} = 2^{324-1} + 2^{216-1} + 2^{108-1} + 1, g = m_2^{-1} \pmod{m_1} = 2^{432-1} - 2^{324-1} - 2^{216-1} - 2^{108-1} + 1.$

For any $c > 1$ $M_1 = 2^{432c} + 1, M_2 = 2^{324c} + 1, F = M_1^{-1} \pmod{M_2} = 2^{324c-1} + 2^{216c-1} + 2^{108c-1} + 1, G = M_2^{-1} \pmod{M_1} = 2^{432c-1} - 2^{324c-1} - 2^{216c-1} - 2^{108c-1} + 1.$

Example 2.2.5. Let $a = 2^{63} + 1, b = 2^{62} + 1. \gcd(a, b) = 1, c = a^{-1} \pmod{b} = 2^{62}, d = b^{-1} \pmod{a} = 2.$

Let $u = 2, \text{Scal}(2, a) = 2^{126} + 1, \text{Scal}(2, b) = 2^{124} + 1:$

1. $\gcd(\text{Scal}(2, a), \text{Scal}(2, b)) = 1.$
2. $\text{Scal}(2, a)^{-1} \pmod{\text{Scal}(2, b)}$ has more than 1 term.

This does not satisfy the definition of scaling.

Chapter 3

Conversion to and from RNS

In this chapter, we discuss the conversion methods to Residue Number System (RNS) (reduction) and conversion from RNS (reconstruction) given a special modulus. Section 3.1 discusses the common algorithms for converting to RNS using Mersenne type, Fermat type, and “trinomial” type moduli. Section 3.2 discusses the reconstruction problem and the Chinese Remainder Theorem.

3.1 Conversion to RNS

The process of converting to RNS, known as reduction, can be formally stated as follows:

Given an integer $a \in \mathbb{Z}$, and a set of moduli, $\{m_1, \dots, m_n\}$, find the set of residue modulo m_i in non-negative representation for $1 \leq i \leq n$, $\{a \bmod m_1, \dots, a \bmod m_n\}$.

Without specific assumptions about the moduli, the residue r_i can be obtained by division with remainder: $a = qm_i + r_i$ where $0 \leq r_i < m_i$. The time complexity of this division is $\Theta(\log^2 a)$ when using the naïve division algorithm. If we use the fastest known division algorithm, the Newton-Raphson division, the time complexity is $\Theta(\mathbf{M}(\log a))$ where $\mathbf{M}(n)$ is the time complexity of multiplying two n -bit numbers.

If we choose the moduli to be of a special form, such as Mersenne type ($2^n - 1$), Fermat type ($2^n + 1$), or “trinomial” type ($2^n - 2^p \pm 1$), the conversion can be done without division.

First, consider the conversion of a number to RNS in its binary representation by Mersenne/Fermat type numbers.

Given a number $a = \sum_{i=0}^m a_i 2^i$, it can be rewritten by grouping bits as

$$a = x_t 2^{nt} + x_{t-1} 2^{n(t-1)} + \cdots + x_1 2^n + x_0, \quad (3.1)$$

where

$$x_j = \sum_{i=0}^{n-1} a_{jn+i} 2^i, j = 0, 1, \dots, t \quad (3.2)$$

with $t = \lceil m/n \rceil$, and $a_k = 0$ for $k > m$. Since $2^n \equiv 1 \pmod{2^n - 1}$, it follows that

$$a \equiv x_t + x_{t-1} + \cdots + x_1 + x_0 \pmod{2^n - 1}. \quad (3.3)$$

Similarly, since $2^n \equiv -1 \pmod{2^n + 1}$, it follows from (3.1) that

$$a \equiv (-1)^t x_t + (-1)^{t-1} x_{t-1} + \cdots + (-1) x_1 + x_0 \pmod{2^n + 1}. \quad (3.4)$$

This shows that reducing an integer a modulo $2^n \pm 1$ has complexity $\Theta(\log a)$.

The key to the conversion lies in the congruence relation $2^n \equiv 1 \pmod{2^n - 1}$ and $2^n \equiv -1 \pmod{2^n + 1}$.

For “trinomial” type moduli, $2^n - (2^p \pm 1)$, a similar reduction applies. We have $2^n \equiv 2^p \pm 1 \pmod{2^n - (2^p \pm 1)}$. Using the same assumption as in Equation 3.1 and 3.2, we can express the number a as

$$a \equiv (2^p \pm 1)^t x_t + (2^p \pm 1)^{t-1} x_{t-1} + \cdots + (2^p \pm 1) x_1 + x_0 \pmod{2^n - (2^p \pm 1)}. \quad (3.5)$$

Unlike Equations 3.3 and 3.4, $(2^p \pm 1)^t$ does not translate nicely to simple bits arithmetic of x_t . However, we can still compute a using Horner’s method. We give the pseudocode for the conversion of a number to RNS by a trinomial type number in Algorithm 1.

Algorithm 1 Trinomial-Reduce(a, m) — Reduce a by $m = 2^n - (2^p \pm 1)$

Require: $0 \leq a$

Ensure: The number returned r satisfies $0 \leq r \leq 2^n - (2^p \pm 1)$

```

1:  $r \leftarrow \text{rem}(a, 2^n)$ 
2:  $q \leftarrow \text{quo}(a, 2^n)$ 
3: while  $q \neq 0$  do
4:    $a \leftarrow r + q \cdot (2^p \pm 1)$ 
5:    $r \leftarrow \text{rem}(a, 2^n)$ 
6:    $q \leftarrow \text{quo}(a, 2^n)$ 
7: end while
8: if  $r \geq 2^n - (2^p \pm 1)$  then  $r \leftarrow r - (2^n - (2^p \pm 1))$ 
9: end if
10: return  $r$ 

```

Consider $m = 2^n - 2^p \pm 1$ and a kn -bit number a , the number of iterations in Algorithm 1 is bounded by $\lceil \frac{(k-1)n}{n-p} \rceil \in \Theta(\log a)$. In each iteration, the number of bits in the number is reduced by $n - p$ bits. Assuming $p \leq cn$ for a fixed constant c , $0 < c < 1$, the number of iterations is bounded by $\lceil \frac{k-1}{1-c} \rceil \in \Theta(\log a)$.

In addition, operations within the loop (Line 4 to Line 6 of Algorithm 1) can be executed in linear time with respect to the number of bits. We break the cost line by line. At line 4, the multiplication of q by $2^p \pm 1$ can be done by shifting the number q left by p bits and adding/subtracting q , both of which can be done in $\Theta(\log a)$ time. The addition of r and $q \cdot (2^p \pm 1)$ can be done in $\Theta(\log a)$ time. The operations in lines 5 and 6 can be seen as extracting the bits of the number a . It is possible to extract the bits with a time complexity linear with respect to the number of bits extracted. The details will be discussed in 5.4. However, even considering the naïve method of extracting the bits, because of the special shape of the moduli, the remainder can be done by shifting the number a left $(1 + \lfloor \log_2 |a| \rfloor - n)$ bits first and right $(1 + \lfloor \log_2 |a| \rfloor - n)$ bits afterwards; the quotient can be done by shifting the number a right n bits. Both of these operations can be done in $\Theta(\log a)$ time. Hence, the time complexity of Algorithm 1 is determined to be $\Theta(\log^2 a)$.

At first glance, the “trinomial” reduction may appear inferior to the reduction of Mersenne/Fermat type moduli. However, if we make assumptions similar to Hiasat [11], we can see that the time complexity can be improved to $\Theta(\log a)$. Specifically, if we assume a has $2n$ bits, for example, a is the result of the multiplication of residues in RNS. Then the number of iterations in Algorithm 1 is $\lceil \frac{n}{n-p} \rceil$. If we use the same assumption, $p \leq cn$ for a fixed constant c , $0 < c < 1$, then the number of iterations is bounded by $\lceil \frac{1}{1-c} \rceil$, i.e., effectively bounded by constant and does not depend on n . If we use the same assumption as in Hiasat [11], $p \leq n/2$, the number of iterations is 2. The time complexity of the “trinomial” reduction is $\Theta(\log a)$ in this case. This indicates that the “trinomial” reduction is slower than the reduction of Mersenne/Fermat type moduli when converting arbitrarily large integers to RNS. However, the performance of reducing the results within RNS with “trinomial” reduction has the same time complexity as the reduction with Mersenne/Fermat type moduli.

The table below summarizes the time complexities of different conversion schemes to RNS.

Conversion Scheme	Time Complexity
Division with remainder	$\Theta(M(\log a))$
Mersenne / Fermat type	$\Theta(\log a)$
Trinomial type	$\Theta(\log^2 a)$

Table 3.1: Time complexities of different conversion schemes for converting a to RNS.

Experimental results, as shown in the benchmarks below, support the theoretical analysis.

Setup. The following benchmark reduces 16 square matrices of dimension 16 where each entry contains a number with n random bits by

- Mersenne type number, $2^{217} - 1$,
- Fermat type number, $2^{217} + 1$,
- Trinomial type number, $2^{217} - 2^{212} + 1$,
- $2^{217} - 2^{212} + 1$ using GMP division with remainder.

n	Mersenne (s)	Fermat (s)	Trinomial (s)	Division (s)
2^{21}	0.160154	0.176589	0.904386	27.0364
2^{22}	0.300846	0.320768	3.544444	53.9951
2^{23}	0.562254	0.599129	14.2362	107.976

Table 3.2: Benchmark results of reducing 16 square matrices of dimension 16 where each entry contains a number with n random bits by different moduli.

3.2 Conversion from RNS

After performing computations in RNS, the next step is to convert the results from RNS to integers. This process is referred to as the reconstruction problem, more commonly known as the Chinese remainder problem. The reconstruction problem can be formally stated as follows:

Given *moduli* $m_1, \dots, m_n \in \mathbb{Z}$ and corresponding residues $u_i \in \mathbb{Z}/m_i\mathbb{Z}$, $1 \leq i \leq n$, find an integer $u \in \mathbb{Z}$ such that

$$u \equiv u_i \pmod{m_i}$$

The algorithm that solves the reconstruction problem is an algorithm that “inverts” the conversion to RNS. We present a commonly-known theorem that guarantees the existence of a unique solution to the reconstruction problem.

Theorem 3.2.1. (*Chinese Remainder Theorem, CRT*) [8]

Let $m_1, \dots, m_n \in \mathbb{Z}$ be integers which are pairwise relatively prime - i.e.

$$\gcd(m_i, m_j) = 1 \text{ for } i \neq j$$

and let $u_i \in \mathbb{Z}_{m_i}$, $i = 1, \dots, n$ be n specified residues.

For any fixed integer $a \in \mathbb{Z}$, there exists a unique integer $u \in \mathbb{Z}$ which satisfies the following conditions:

$$a \leq u < a + m, \text{ where } m = \prod_{i=1}^n m_i$$

$$u \equiv u_i \pmod{m_i}, 1 \leq i \leq n.$$

However, CRT does not provide a method for reconstructing the solution. There are many techniques to reconstruct the solution. Common techniques to solve the reconstruction problem are interpolation and mixed-radix representation. The algorithm using the latter technique is also known as Garner’s algorithm. Detailed descriptions of these algorithms using the two common techniques can be found in standard Computer Algebra textbooks [7, 8].

Both techniques rely on three time-consuming operations:

1. Finding the inverse of a modulus modulo another modulus ($m_i^{-1} \pmod{m_j}$),
2. multiplying an integer by the inverse of a moduli ($a \cdot m_i^{-1}$), and
3. multiplying an integer by a moduli ($a \cdot m_i$).

As in the previous section, the sparsity properties of the moduli can be exploited to accelerate the reconstruction process.

An effective optimization is to accelerate the computation of multiplication. The standard naïve multiplication algorithm can be seen as a series of shifts and additions. The time complexity is $\Theta(\log^2 a)$ where a is the integer to be multiplied. This multiplication optimization leverages the sparsity of the moduli.

Since the moduli can be chosen in advance, we know exactly what bits are set. We can directly perform a number of predetermined shifts and additions to accomplish the multiplication. For a chosen moduli, the number of set bits are constant. As a result, the multiplication can be done in $\Theta(\log a)$ time.

For example, for the Fermat/Mersenne type moduli, $(2^n \pm 1)$, multiplication involves shifting the integer left by n bits and adding/subtracting the integer. For “trinomial” type moduli, $(2^n - 2^p \pm 1)$, multiplication of an integer a involves:

1. adding the results of shifting the integer a left by n bits,
2. subtracting the result of shifting the integer a left by p bits, and
3. adding/subtracting a itself.

The remaining challenges are determining inverses efficiently and ensuring the inverses have few set bits.

We discovered that, under certain restrictions, a known moduli set can be scaled to a larger moduli set. This approach enables the precomputation of inverses with known set bits. Carefully selecting an initial moduli set with few set bits in the inverses ensures that scaled inverses retain the same number of set bits. The details will be discussed in the next chapter.

Chapter 4

Searching for Scalable Moduli Sets

Finding a special moduli set that is scalable is crucial in accelerating the reconstruction process. In Chapter 4.1, we discuss common schemes for generating a set of scalable moduli. In Chapter 4.2, we discuss the work of searching for the patterns in the inverses of two moduli in the set. We propose that the results in the polynomials can be used to identify scalable moduli with special focus on Fermat type numbers.

4.1 Generating Scalable Moduli Sets

First, we define a scalable moduli set.

Definition 4.1.1. (*Scalable Moduli Set*)

*A moduli set is **scalable** if any two moduli in the set are scalable.*

A pair of Mersenne type numbers is not scalable. Consider two coprime moduli, $2^n - 1$, $2^m - 1$, since $\gcd(2^n - 1, 2^m - 1) = 2^{\gcd(n,m)} - 1$, we get that $\gcd(n, m) = 1$. If we scale the moduli by $u > 1$, $\text{Scal}(u, 2^n - 1) = 2^{un} - 1$ and $\text{Scal}(u, 2^m - 1) = 2^{um} - 1$. Now, the gcd of the scaled moduli is $2^u - 1$, which means that the scaled moduli are no longer coprime. Although Mersenne type numbers are not scalable, they are still good candidates for the moduli set due to their efficient conversion to RNS and efficient multiplication properties.

Our focus will be on Fermat type numbers and “trinomial” type numbers. Zima and Steward [17] proposed two schemes, “shift” scheme and “block” scheme, to generate scalable moduli sets of Fermat type numbers.

For Fermat type numbers, scaling preserves relative primality. It is well-known that $2^m + 1 \perp 2^n + 1$ if and only if $\nu_2(m) \neq \nu_2(n)$ [3, 16] where $\nu_2(a)$ is the binary valuation of integer a (the maximum degree of 2 contained in a). Consider two coprime Fermat type moduli, $2^m + 1, 2^n + 1$, by the previous proposition, $\nu_2(m) \neq \nu_2(n)$. If we scale the moduli by $u > 1$, we have $\text{Scal}(u, 2^m + 1) = 2^{um} + 1$ and $\text{Scal}(u, 2^n + 1) = 2^{un} + 1$. The relative primality is preserved under scaling as $\nu_2(um) \neq \nu_2(un)$. This forms the foundation of the two schemes discussed below.

4.1.1 “Shift” Scheme

The “shift” scheme begins by selecting a Fermat type modulus $2^a + 1$ arbitrarily. Subsequent moduli are generated by doubling the exponent a , which can be efficiently achieved using a left bit shift. It is easy to see that every two moduli in the set are relatively prime.

The scheme to generate a moduli set is as follows: Let the moduli be $m_i = 2^{a2^i} + 1$, where a is an arbitrary integer and $i = 0, 1, \dots, k$.

For example, if $a = 1$, the moduli set generated by the “shift” scheme is $\{2^1 + 1, 2^2 + 1, 2^4 + 1, \dots, 2^{2^i} + 1\}$.

A drawback of the “shift” scheme is the significant imbalance among the moduli. The last modulus has the same magnitude as the product of all the previous moduli, making the calculations of the last modulus the bottleneck.

4.1.2 “Block” Schemes

The “block” schemes aim to fix the imbalance observed in the “shift” scheme. The “block” schemes generate moduli with exponents of the same bit length by varying the binary valuation of the exponents.

We present two sample schemes to generate a moduli set of size b :

1. generate the initial exponent, $e_1 = 2^b - 1$. The rest of the exponents follow the recurrence $e_{k+1} = e_k - 2^{k-1}$. The recurrence can be rewritten as the following relation $e_k = 2^b - 2^{k-1}$ for $k = 2, 3, \dots, b$.
2. generate the final exponent, $e_b = 2^{b-1}$. The rest of the exponents follow the relation $e_k = 2^{b-1} + 2^{b-k-1}$ for $k = 1, 2, \dots, b - 1$.

The key to generating a moduli set is ensuring that each exponent has a unique binary valuation. For a moduli set of size b , as long as we keep the binary valuation of the exponents different (this can be achieved by varying the number of trailing zeroes of the binary form of the exponents), we have the freedom to choose the bits preceding the first 1 (counting from the right) in the binary form of the exponents.

For a given moduli size b , many “block” style moduli sets are possible. For example, let $b = 4$. The moduli exponents generated using the first scheme are 15, 14, 12, 8. The moduli set is $\{2^{15} + 1, 2^{14} + 1, 2^{12} + 1, 2^8 + 1\}$. The moduli exponents generated using the second scheme are 12, 10, 9, 8. The moduli set is $\{2^{12} + 1, 2^{10} + 1, 2^9 + 1, 2^8 + 1\}$. The exponent set 12, 10, 11, 8 is another valid set. This set is a modification of the second scheme, where the bits prior to the first 1 in the binary form of the third modulus (9) are altered.

Moduli sets generated by the “block” scheme usually include an additional modulus, 2^{e_b} . The additional modulus is guaranteed to be coprime with the rest of the moduli in the set.

Beyond the two Fermat type number schemes, it is appealing to find a moduli set in the form of “trinomial” type numbers. While the “block” schemes mitigate the imbalanced drawback of the “shift” scheme, there remains a need for moduli sets with balanced bit sizes. The “trinomial” scheme offers an effective alternative for generating moduli with balanced bit sizes. The “trinomial” scheme is discussed in the next section.

4.1.3 “Trinomial” Scheme

The search for scalable moduli sets in the form of “trinomial” is a natural extension of Fermat type numbers. We want to address the imbalance of the moduli generated by the “block” scheme while adding as few terms as possible to the Fermat type numbers so that the multiplication by the moduli stays simple. A set of “trinomial” type moduli is in the form of $2^n - 2^k \pm 1$ where n is given and k can be varied to generate different moduli. Compared to the “block” scheme, introducing an additional term 2^k allows for the generation of moduli sets perfectly balanced in bit size.

As demonstrated in the previous chapter, “trinomial” type numbers enable efficient conversions to RNS. Its sparse set bits allow for efficient multiplication. For this scheme, we focus on the “trinomial” type numbers that are in the form of $2^n - 2^k + 1$. Similar statements can be made for the other form of “trinomial” type numbers, $2^n - 2^k - 1$.

We present the following propositions:

Proposition 4.1.2. (*Simplifying GCD*)

Assuming $a < b$, we claim the

$$\begin{aligned}\gcd(2^n - 2^a + 1, 2^n - 2^b + 1) &= \gcd(2^{n \bmod (b-a)} - 2^{a \bmod (b-a)} + 1, 2^{b-a} - 1) \\ &= \gcd(2^{n \bmod (b-a)} - 2^{b \bmod (b-a)} + 1, 2^{b-a} - 1)\end{aligned}$$

Proof. We consider the Euclidean algorithm sequence of the two moduli:

$$\begin{aligned}\gcd(2^n - 2^a + 1, 2^n - 2^b + 1) &= \gcd(2^n - 2^b + 1, 2^b - 2^a) \\ &= \gcd(2^n - 2^b + 1, 2^a(2^{b-a} - 1)) \\ &= \gcd(2^{n-(b-a)} - 2^a + 1, 2^{b-a} - 1) \\ &= \gcd(2^{n-(b-a)} - 2^a + 2^{b-a}, 2^{b-a} - 1) \\ &= \dots \\ &= \gcd(2^{n \bmod (b-a)} - 2^{a \bmod (b-a)} + 1, 2^{b-a} - 1)\end{aligned}$$

Note that $a \equiv b \pmod{b-a}$. □

Proposition 4.1.3. (Sufficient conditions for the coprimality of “trinomial” type numbers (1)) Consider two moduli:

$$m_1 = 2^n - 2^k + 1, m_2 = 2^n - 2^\ell + 1, k > \ell$$

If $n \equiv k \pmod{k-\ell}$ or $k \equiv 0 \pmod{k-\ell}$ then $\gcd(m_1, m_2) = 1$.

Proof. We have $\gcd(m_1, m_2) = \gcd(2^{n \bmod (k-\ell)} - 2^{k \bmod (k-\ell)} + 1, 2^{k-\ell} - 1)$ from Proposition 4.1.2. When $n \equiv k \pmod{k-\ell}$ or $k \equiv 0 \pmod{k-\ell}$, the gcd result is 1. □

If we assume $n \equiv k \pmod{k-\ell}$, the pair of two moduli is scalable.

Proposition 4.1.4. Let $m_1 = 2^n - 2^k + 1, m_2 = 2^n - 2^\ell + 1$ with $n \equiv k \pmod{k-\ell}$. We write $n = k + p(k-\ell)$. m_1 and m_2 are scalable.

Furthermore, the inverses follow the following pattern:

$$\begin{aligned}m_1^{-1} &\equiv \sum_{i=0}^p 2^{i(k-\ell)} \pmod{m_2} \\ m_2^{-1} &\equiv -\sum_{i=0}^p 2^{i(k-\ell)} + 1 \pmod{m_1}\end{aligned}$$

Proof. We rewrite $n = k + p(k - \ell)$ for some $p \in \mathbb{Z}$. Let $\text{Scal}(c, m_1) = 2^{cn} - 2^{ck} + 1$, $\text{Scal}(c, m_2) = 2^{cn} - 2^{c\ell} + 1$. We have $cn = ck + p(ck - c\ell)$. That is, $cn \equiv ck \pmod{(ck - c\ell)}$. The scaled moduli are coprime.

We rewrite m_1 as $m_1 \equiv 2^\ell - 2^k \pmod{m_2}$. Consider two adjacent terms in the claimed inverse $m_1^{-1} \pmod{m_2}$,

$$(2^\ell - 2^k)(2^{i(k-\ell)}) = 2^{i(k-\ell)+\ell} - 2^{i(k-\ell)+k}.$$

$$(2^\ell - 2^k)(2^{(i+1)(k-\ell)}) = 2^{(i+1)(k-\ell)+\ell} - 2^{(i+1)(k-\ell)+k}.$$

We can see that the cancellation happens.

$$(2^\ell - 2^k)(2^{i(k-\ell)} + 2^{(i+1)(k-\ell)}) = 2^{i(k-\ell)+\ell} - 2^{(i+1)(k-\ell)+k}.$$

Hence,

$$(2^\ell - 2^k) \left(\sum_{i=0}^p 2^{i(k-\ell)} \right) = 2^\ell - 2^{p(k-\ell)+k} = 2^\ell - 2^n \equiv 1 \pmod{m_2}.$$

Similarly, $m_2 \equiv 2^k - 2^\ell \pmod{m_1}$. Similar result follows.

We can choose $k - \ell \neq 1$ such that the inverses are in the SBB form. It follows that such pair of moduli is scalable. \square

We can get another sufficient condition for coprimality of “trinomial” type numbers if we make more assumptions.

Proposition 4.1.5. *(Sufficient conditions for the coprimality of “trinomial” type numbers (2)) Consider two moduli:*

$$m_1 = 2^n - 2^k + 1, m_2 = 2^n - 2^\ell + 1, k > \ell$$

If $k \equiv 1 \pmod{(k - \ell)}$, m_1 and m_2 are coprime if and only if $n \pmod{(k - \ell)}$ and $k - \ell$ are coprime.

Proof. Based on Proposition 4.1.2, if $k \equiv 1 \pmod{(k - \ell)}$, the gcd simplifies to $\gcd(m_1, m_2) = \gcd(2^{n \pmod{(k - \ell)}} - 1, 2^{k - \ell} - 1)$, reducing the problem to the coprimality of Mersenne type numbers. \square

As in the “block” schemes, it is also common to include two additional moduli of the form 2^n and $2^n + 1$ in the “trinomial” scheme. The coprimality between the “trinomial” type number and 2^n follows from the fact that 2 is the sole prime factor of 2^n , whereas the “trinomial” type number is odd. The coprimality between the “trinomial” type number and $2^n + 1$ follows from the fact that $\gcd(2^n - 2^k + 1, 2^n + 1) = \gcd(2^n + 1, 2^k) = 1$.

These additional moduli are coprime and scalable with respect to all other moduli in the set.

Proposition 4.1.6. *Let $a = 2^n - 2^m + 1$, $b = 2^n$. Let $n = qm + r$, $0 \leq r < m$. Then,*

$$a^{-1} \bmod b = \sum_{i=0}^q 2^{im},$$

$$b^{-1} \bmod a = -\sum_{i=0}^q 2^{im} + 2^{m-r}.$$

Proof. Consider the equivalence of $a \equiv 1 - 2^m \bmod b$, we have

$$(1 - 2^m)(2^{im}) = 2^{im} - 2^{(i+1)m},$$

$$(1 - 2^m)(2^{(i+1)m}) = 2^{(i+1)m} - 2^{(i+2)m}.$$

Hence,

$$(1 - 2^m) \left(\sum_{i=0}^q 2^{im} \right) = 1 - 2^{(q+1)m}.$$

Since $(q+1)m > n$, then $a^{-1} \bmod b$ is $\sum_{i=0}^q 2^{im} \bmod b$.

Similarly, $b \equiv 2^m - 1 \bmod a$. $(2^m - 1) \sum_{i=0}^q -2^{im} = 1 - 2^{(q+1)m}$. Since $b \cdot 2^{m-r} = 2^{(q+1)m}$, combining the two results, we get $b^{-1} \bmod a = \sum_{i=0}^q -2^{im} + 2^{m-r}$. \square

Proposition 4.1.7. *Let $a = 2^n - 2^k + 1$, $b = 2^n + 1$. Then,*

$$a^{-1} \bmod b = 2^{n-k},$$

$$b^{-1} \bmod a = -2^{n-k} + 1.$$

Proof. We can verify that the proposed closed-form inverses are correct by calculating the product of the proposed inverse and the modulus itself.

Consider $a \equiv -2^k \pmod{b}$, $b \equiv 2^k \pmod{a}$,

$$(2^{n-k})(-2^k) = -2^n \equiv 1 \pmod{b}.$$

$$(-2^{n-k} + 1)(2^k) \equiv -2^n + 2^k \equiv 1 \pmod{a}.$$

□

The closed-form expressions for the inverses guarantee that the two additional moduli are scalable with respect to the “trinomial” type moduli.

It is worth noting that “trinomial” type moduli are not generally scalable. For example, given $a = 2^{60} - 2^{53} + 1$, $b = 2^{60} - 2^{56} + 1$, $\gcd(a, b) = 1$. $\text{Scal}(2, a) = 2^{120} - 2^{106} + 1$, $\text{Scal}(2, b) = 2^{120} - 2^{112} + 1$. $\gcd(\text{Scal}(2, a), \text{Scal}(2, b)) = 7$. The scaled moduli are no longer coprime.

4.2 Searching for Inverses

All types of moduli allow for efficient multiplication. The previous chapter discussed various strategies for generating moduli sets. There is still one important question pending - how to find the inverses of the moduli in the set. Having a scalable sparse inverse is crucial in accelerating the reconstruction process. There are two major time-consuming operations in the reconstruction process related to the inverse. The first is the precomputation of the inverse (line 2 to line 5 of Algorithm 2) and the second is the multiplication by the inverse (line 10 of Algorithm 2).

Sparse inverses enable faster multiplication as mentioned in the previous chapter. Additionally, they allow for the precomputation of inverses. The “block” schemes are reliable ways to generate moduli sets. Through experiments, we discovered that every two moduli in the set generated by the “block” schemes are often scalable. To better study why the “block” schemes generate scalable moduli sets, we investigate the patterns in the inverses through extended Euclidean algorithms. We show that any pair of relatively prime Fermat numbers is scalable, and their inverses are easy to generate.

4.2.1 Fermat Type Inverses

Sometimes, it is convenient to represent a moduli in SBB as the result of evaluation of a polynomial. For example, $m = 2^n + 1$ can be viewed as $f(2)$ for $f(x) = x^n + 1$. By viewing

the modulus as a polynomial and evaluating at powers of 2, it naturally encodes a family of moduli under scaling. For example, $\text{Scal}(u, m) = 2^{um} + 1 = f(2^u)$.

Here are several easy-to-prove statements which help to justify a choice of scalable moduli.

Proposition 4.2.1. *Consider polynomial $f(x)$ with coefficients from $\{-1, 0, 1\}$. Then for any $\ell \geq 2$, the result of evaluation $f(2^\ell)$ is an integer in SBB form with the number of set bits equal to the support of $f(x)$.*

Proof. Consider $f(x) = \sum_{i=0}^n b_i x^i$, where $b_i \in \{-1, 0, 1\}$. Then, $f(2^\ell) = \sum_{i=0}^n b_i 2^{i\ell}$. The difference in exponents, $(i+1)\ell - i\ell = \ell \geq 2$. Hence, no two consecutive bits are set in the result. Hence, the result is in SBB form. Since the number is already in SBB form, it is the sparsest representation. Hence, no further simplification is possible. Consequently, the number of set bits is equal to the support of $f(x)$. \square

Proposition 4.2.2. *Consider polynomial $f(x) \in \mathbb{Z}[x]$: $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_x x + a_0$ with coefficients a_i in SBB form. Let $l(a_i)$ be the bit-length of a_i and $s(a_i)$ be the number of set bits in a_i . Then for any $\ell \geq \max_{i=0\dots n} l(a_i) + 1$, the result of evaluation $f(2^\ell)$ is an integer in SBB form with the number of set bits equal to $\sum_{i=0}^n s(a_i)$.*

Proof. Following the same logic, after evaluation, the gap between a_i and a_{i+1} is $\ell \geq \max_{i=0\dots n} l(a_i) + 1$. If we substitute a_i with a SBB number, ℓ guarantees that no two consecutive bits are set. Hence, the result is in SBB form. The number of set bits is equal to the sum of the set bits of a_i . \square

Proposition 4.2.3. *Consider two Fermat type polynomials $f(x) = x^n + 1, g(x) = x^m + 1$. If $\gcd(f, g) = 1$ then for any value of $\ell \geq 1$ numbers $f(2^\ell)$ and $g(2^\ell)$ are relatively prime.*

Proof. Proof follows immediately from:

- $\gcd(x^n + 1, x^m + 1) = 1$ iff $\nu_2(n) \neq \nu_2(m)$
- $\gcd(2^n + 1, 2^m + 1) = 1$ iff $\nu_2(n) \neq \nu_2(m)$
- for any $\ell > 0$ $\nu_2(\ell n) \neq \nu_2(\ell m)$ iff $\nu_2(n) \neq \nu_2(m)$

\square

With all the necessary propositions established, we can now proceed to discuss patterns in the inverses.

For two Fermat type numbers, we can analyze their structure through polynomials in the form of $x^n + 1$ and $x^m + 1$. For example, let $a = 2^{432} + 1$, $b = 2^{324} + 1$. We can consider the polynomials $f(x) = x^{432} + 1$, $g(x) = x^{324} + 1$ respectively. $f(x)$ and $g(x)$ can be seen as the evaluation of the polynomial $F(y) = y^4 + 1$, $G(y) = y^3 + 1$ at $y = x^{108}$. This approach allows us to study the inverses in a simplified form where the exponents are coprime.

For the discussion below, we make the following assumptions: let $f = x^n + 1$, $g = x^m + 1$.

We present the following facts:

Let u and v be natural with $u > v$. Write $u = qv + r$, ($0 \leq r < v$). Then

1. $x^u + 1 = Q(x^v - 1) + R$, with

$$R = x^r + 1, \quad Q = x^{u-v} + x^{u-2v} + \dots + x^{u-qv}.$$

2. $x^u - 1 = Q(x^v - 1) + R$, with

$$R = x^r - 1, \quad Q = x^{u-v} + x^{u-2v} + \dots + x^{u-qv}.$$

3. $x^u + 1 = Q(x^v + 1) + R$, with

$$R = x^r + 1, \quad Q = x^{u-v} - x^{u-2v} + \dots - x^{u-qv}$$

for even q , and

$$R = -x^r + 1, \quad Q = x^{u-v} - x^{u-2v} + \dots + x^{u-qv}$$

for odd q .

4. $x^u - 1 = Q(x^v + 1) + R$, with

$$R = x^r - 1, \quad Q = x^{u-v} - x^{u-2v} + \dots - x^{u-qv}$$

for even q , and

$$R = -x^r - 1, \quad Q = x^{u-v} - x^{u-2v} + \dots + x^{u-qv}$$

for odd q .

A [brief justification](#) is included in the appendix.

It is important to note that $\mathbb{Z}[x]$ is not an Euclidean domain. We are doing the computation in a less restrictive Euclidean domain $\mathbb{Q}[x]$. Considering the polynomials in $\mathbb{Q}[x]$, we claim that in the Extended Euclidean algorithm steps, the cofactors s_i, t_i always satisfy the conditions $\deg(s_i) < \deg(g)$ and $\deg(t_i) < \deg(f)$.

From the facts presented above, the remainders in the Extended Euclidean algorithm will be in the form of $\pm x^r \pm 1$. However, this does not make the facts above less useful.

Proposition 4.2.4. *Let $f = \pm x^u \pm 1, g = \pm x^v \pm 1$, the remainder R and Q of $f = Qg + R$ can be found by properly negating the polynomials into the cases presented above.*

Proof. If both the leading coefficient of f and g are negative, we can consider $-f = x^u \pm 1, -g = x^v \pm 1$. We can use the above facts to find the remainder R' and Q' of $-f = Q'(-g) + R'$. Then, the actual quotient and remainder $Q = Q'$ and $R = -R'$.

If the leading coefficient of f is negative, suppose $f = -x^u \pm 1, g = x^v \pm 1$, we wish to compute $f = Qg + R$. We can compute the R' of $-f = x^u \pm 1, g = x^v \pm 1$ with $-f = Q'g + R'$. Then, the actual quotient and remainder $Q = -Q'$ and $R = -R'$.

If the leading coefficient of g is negative, we similarly compute $f = Q'(-g) + R'$ and the actual quotient and remainder $Q = -Q'$ and $R = R'$. \square

Proposition 4.2.5. *Let $f = x^n + 1, g = x^m + 1$ where $n > m$.*

1. *The cofactors s, t in the steps of the Extended Euclidean algorithm (excluding the last step if f, g are coprime) are in $\mathbb{Z}[x]$ where $\deg(s) < \deg(g)$ and $\deg(t) < \deg(f)$ with the nonzero coefficients ± 1 .*
2. *If the cofactors have more than 1 term, the difference in adjacent degrees of the terms in the cofactors ordered by the degree is $h = \gcd(n, m)$. Specifically, $s = \pm x^{m-h} \pm x^{m-2h} \pm \dots \pm 1, t = \pm x^{n-h} \pm x^{n-2h} \pm \dots \pm 1$.*

Proof. We let the remainder sequence of $r_0 = n, r_1 = m$ to be $r_0, r_1, \dots, r_n, r_{n+1} = 0$. To establish the induction, we develop a recursive extended Euclidean algorithm. Suppose we have a function `reggcd(a, b)` that returns the cofactors s, t of the Extended Euclidean algorithm. We define the base case of the recursive Extended Euclidean algorithm: `reggcd(a, 0)` returns $[1, 0]$ just like the regular extended Euclidean algorithm.

Consider the Bezout identity $sa + tb = \gcd(a, b)$. We want to obtain the values of s and t . Let $a = qb + r$, and by the Euclidean algorithm, we have $\gcd(a, b) = \gcd(b, r)$. Then,

suppose we can get $\text{regcd}(\mathbf{b}, \mathbf{r}) = [s', t']$ such that $s'b + t'r = \gcd(b, r)$. Then, we can compute s, t such that $s = t', t = s' - qt'$. We can recursively find the value of s and t for any input a, b . This process eventually terminates as the Euclidean algorithm terminates.

The recursive calls depend on the remainder sequence of n, m . For any $f = x^{r_0} + 1, g = x^{r_1} + 1$, the recursive Extended Euclidean algorithm recursively calls on $(x^{r_1} + 1, \pm x^{r_2} \pm 1), (\pm x^{r_2} \pm 1, \pm x^{r_3} \pm 1), \dots, (\pm x^{r_{n-1}} \pm 1, \pm x^{r_n} \pm 1), (\pm x^{r_n} \pm 1, \pm x^0 \pm 1)$.

Here, we want to be careful about its terminating condition. If the last remainder $\pm x^0 \pm 1 = 0$, then the f, g are not coprime, where the gcd is the last non-zero remainder. Otherwise, if the last remainder $\pm x^0 \pm 1 = \pm 2$, then f, g are coprime. We will show that the cofactors s, t are in $\mathbb{Z}[x]$ with the coefficients ± 1 and with the correct degree constraint up to the terminating call.

Without loss of generality, we assume that f, g are coprime in the following proof.

The inductive hypothesis is that we assume that the claim is true for the input of the recursive extended Euclidean algorithm $(x^{r_i} + 1, x^{r_{i+1}} + 1)$. We want to show that the claim is true for the previous recursive call $(x^{r_{i-1}} + 1, x^{r_i} + 1)$.

We number the input pair $(\pm x^{r_i} \pm 1, \pm x^{r_{i+1}} \pm 1)$ as i th call.

At the terminating call, $((n+1)$ th call), $a = \pm x^0 \pm 1, b = 0$, the returned cofactors pair is $(1, 0)$.

In the previous recursive call, n th call, $a = \pm x^{r_n} \pm 1, b = \pm x^0 \pm 1$, the returned cofactors pair is $(s, t) = (0, 1)$. The cofactors only have 1 term, so we need to look back at another call.

At the $(n-1)$ th call, $a = \pm x^{r_{n-1}} \pm 1, b = \pm x^{r_n} \pm 1$, the returned cofactors pair is $(s, t) = (1, -Q_n)$ where $Q_n = (\pm x^{r_{n-1}-r_n} \pm x^{r_{n-1}-2r_n} \pm \dots \pm 1)$. Note that $r_n = \gcd(n, m)$. The cofactor t will have at least two terms with the distance between the terms equal to $r_n = \gcd(n, m)$.

At the $(n-2)$ th call, $a = \pm x^{r_{n-2}} \pm 1, b = \pm x^{r_{n-1}} \pm 1$, the returned cofactors pair is $(s, t) = (-Q_n, 1 + Q_{n-1}Q_n)$. where $Q_{n-1} = (\pm x^{r_{n-2}-r_{n-1}} \pm x^{r_{n-2}-2r_{n-1}} \pm \dots \pm x^{r_n})$. The cofactors s, t will have at least two terms with the distance between the exponents terms equal to $r_n = \gcd(n, m)$.

The cofactor s will have at least two terms with the distance between the degrees of the terms equal to $r_n = \gcd(n, m)$.

Hence, we have shown that the **base case** is true, the cofactors are in $\mathbb{Z}[x]$ with the coefficients ± 1 and with the correct degree constraint; and the difference between the

degrees of the terms in the cofactors is $\gcd(n, m)$. We have shown that the base case is true. Suppose we are at i th recursive call where $(a, b) = (\pm x^{r_i} \pm 1, \pm x^{r_{i+1}} \pm 1)$, where the returned cofactor pair is (s, t) . We want to show that the claim is true for the previous $(i - 1)$ th call where $(a, b) = (\pm x^{r_{i-1}} \pm 1, \pm x^{r_i} \pm 1)$. According to the recursive extended Euclidean algorithm, the returned cofactor pair of the $(i - 1)$ th iteration is $(s', t') = (t, s - Q_i t)$ where $Q_i = \pm x^{r_{i-1}-r_i} \pm \dots \pm x^{r_{i+1}}$. We know that $\deg(s) < r_{i+1}$ and $\deg(t) < r_i$. Consider the product of $Q_i t$, since $\deg(t) < r_i$, we can write $t = \pm x^{r_i-r_n} \pm x^{r_i-2r_n} \pm \dots \pm x^0$, $s = \pm x^{r_{i+1}-r_n} \pm x^{r_{i+1}-2r_n} \pm \dots \pm x^0$.

We expand the product $Q_i t$:

$$\begin{aligned} Q_i t &= (\pm x^{r_{i-1}-r_i} t \pm \dots \pm x^{r_{i+1}} t) \\ &= (\pm x^{r_{i-1}-r_i} t \pm \dots \pm x^{r_{i-1}-q_i r_i} t) \\ &= (\pm x^{r_{i-1}-r_n} \pm x^{r_{i-1}-2r_n} \pm \dots \pm x^{r_{i-1}-r_i}) \pm \dots \\ &\quad \pm (\pm x^{r_{i-1}-(q_i-1)r_i-r_n} \pm x^{r_{i-1}-(q_i-1)r_i-2r_n} \pm \dots \pm x^{r_{i-1}-q_i r_i}) \end{aligned}$$

The degree of the terms of $Q_i t$ is greater than or equal to r_{i+1} and smaller than r_{i-1} . All the coefficients of $Q_i t$ are ± 1 . Hence, s and $Q_i t$ have no overlapping terms with the same degree. Thus, it is impossible for t' to have terms with coefficients other than ± 1 . Furthermore, the largest degree term of s is $x^{r_{i+1}-r_n}$, and the smallest degree term of $Q_i t$ is $x^{r_{i-1}-q_i r_i} = x^{r_{i+1}}$. Hence, the difference between the degrees of the terms of s' is $r_n = \gcd(n, m)$. Since $t' = t$, the difference between the degrees of the terms follows from the inductive hypothesis. Hence, $\deg(s') < r_i$ and $\deg(t') < r_{i-1}$. The coefficients of s, t are ± 1 , and the coefficients of Q_i are ± 1 . Hence, the claim is true for the $(i - 1)$ th call.

By induction, the claim is true for all i . □

Proposition 4.2.5 answers the question why the extended Euclidean algorithm of the block-Fermat type moduli set always reaches the step where $sf + tg = 2$. In addition, it ensures that the coefficients of the cofactors are ± 1 . The inverses can be found by dividing the cofactors by 2. $f^{-1} \bmod g = \frac{1}{2}s$ and $g^{-1} \bmod f = \frac{1}{2}t$.

While Proposition 4.2.5 fixes the shape of the coefficient, it does not provide a method to find the cofactors themselves. The cofactors of concrete cases can be found through computational means, but it would be more desirable to have a closed-form expression of the cofactors.

Closed-form expressions for the cofactors are known in some specific cases. We present the following propositions:

Proposition 4.2.6. Consider the choice of $f = x^{m+1} + 1, g = x^m + 1$ where $m \geq 1$, $f^{-1} \bmod g = \frac{1}{2} \sum_{i=0}^{m-1} x^i$, $g^{-1} \bmod f = \frac{1}{2} (-\sum_{i=1}^m x^i + 1)$.

Proof. Consider $s(x) = x^{m-1} + x^{m-2} + \dots + x + 1$ and $t(x) = -(x^m + x^{m-1} + \dots + x^2 + x) + 1$.
 $sf = x^{2m} + \dots + x^{m+1} + x^{m-1} + \dots + 1 = \sum_{i=0}^{2m} x^i - x^m$, $tg = -x^{2m} - \dots - x^{m+1} + x^m - x^m - \dots - x + 1 = -\sum_{i=0}^{2m} x^i + x^m + 2$.

Hence, $sf + tg = 2$. Therefore, the inverse of $f \bmod g$ is $\frac{1}{2}s$. Similarly, the inverse of $g \bmod f$ is $\frac{1}{2}t$. \square

Example 4.2.7.

```
> f := x^63 + 1;
> g := x^62 + 1;
> s := sum(x^i, i = 0 .. 61);
> t := -sum(x^i, i = 1 .. 62) + 1;
> expand(f*s + g*t);
```

2

Given the inverses in polynomial form, we aim to evaluate the polynomials to produce results in SBB form. Proposition 4.2.3 guarantees coprimality under scaling. If the inverses are also expressed in SBB form, they can be scaled such that the distances between set bits are uniformly scaled. This allows us to obtain a scalable pair of moduli.

Given $f = x^n + 1, g = x^m + 1, f^{-1} \bmod g = \frac{1}{2}s, g^{-1} \bmod f = \frac{1}{2}t$, we would like to obtain the inverses $f(2^\ell)^{-1} \bmod g(2^\ell)$ and $g(2^\ell)^{-1} \bmod f(2^\ell)$ for some ℓ in SBB form. The inverses in polynomials almost lead to the inverses in SBB form, but both $f^{-1} \bmod g$ and $g^{-1} \bmod f$ contain a constant term of $\pm \frac{1}{2}$. This can be solved by considering $f^{-1} \bmod g \equiv \frac{1}{2}s \pm \frac{1}{2}g \bmod g$ and $g^{-1} \bmod f \equiv \frac{1}{2}t \pm \frac{1}{2}f \bmod f$. This allows us to obtain the inverses in SBB form for $\ell \geq 2$.

4.2.2 Number of Set Bits of the Inverses

A natural question that arises is which moduli set generating scheme performs better. For the purpose of accelerating modular arithmetic, it is desirable for the inverses to have as few set bits as possible.

The results from the previous section provide insight into the shape and the number of set bits for inverses of Fermat type numbers. Hence, we shall focus on this relatively well-studied moduli shape.

Consider the Fermat type numbers in polynomial form. Suppose we have $f = x^n + 1$, $g = x^m + 1$ with $\gcd(f, g) = 1$, the previous section shows that we can find $f^{-1} \bmod g = \frac{1}{2}s \pm \frac{1}{2}g$ and $g^{-1} \bmod f = \frac{1}{2}t \pm \frac{1}{2}f$. The two inverses have different numbers of terms and the number is controlled by the degree of f and g .

- The number of terms of s is $m/\gcd(n, m)$, and
- the number of terms of t is $n/\gcd(n, m)$.

Suppose $n > m$, then $f^{-1} \bmod g$ has fewer terms than $g^{-1} \bmod f$. Hence, it is always desirable to compute the inverse of the larger modulus with respect to the smaller modulus. In other words, it is always desirable to arrange the moduli set in the order of decreasing size to minimize the overall number of set bits in the inverses.

From Proposition 4.2.5, for given m, n , we know the number of set bits of the inverses also depends on the gcd results of m and n . To investigate the greatest common divisor of m and n , we take a closer look at the exponents generated by the two “block” moduli generating schemes.

Given a moduli size b , the first scheme generates moduli with exponents of the form $e_k = 2^b - 2^{b-k}$. Suppose the two moduli chosen to compute the gcd is $e_k = 2^b - 2^{b-k}$, $e_{k+l} = 2^b - 2^{b-k-l}$ where $l > 0$. To compute the gcd, $\gcd(e_k, e_{k+l})$, we can first factor out all the powers of 2, and compute the gcd of the powers of 2 first.

$$\begin{aligned} \gcd(e_k, e_{k+l}) &= \gcd(2^{b-k}, 2^{b-k-l}) \gcd(e_k/2^{b-k}, e_{k+l}/2^{b-k-l}) \\ &= 2^{b-k-l} \gcd(e_k/2^{b-k}, e_{k+l}/2^{b-k-l}) \end{aligned}$$

where $e_k/2^{b-k} = 2^k - 1$ and $e_{k+l}/2^{b-k-l} = 2^{k+l} - 1$. $\gcd(e_k/2^{b-k}, e_{k+l}/2^{b-k-l})$ follows from a well-known result, $\gcd(2^k - 1, 2^{k+l} - 1) = 2^{\gcd(k, k+l)} - 1$. The two expressions are coprime if and only if k and $k + l$ are coprime.

Compare this with the second scheme, the second scheme generates exponents of moduli of the form $e_k = 2^{b-1} + 2^{b-k-1}$, $e_{k+l} = 2^{b-1} + 2^{b-k-l-1}$. We apply the same reasoning to get the following:

$$\begin{aligned} \gcd(e_k, e_{k+l}) &= \gcd(2^{b-k-1}, 2^{b-k-l-1}) \gcd(e_k/2^{b-k-1}, e_{k+l}/2^{b-k-l-1}) \\ &= 2^{b-k-l-1} \gcd(2^k + 1, 2^{k+l} + 1) \end{aligned}$$

We also consider the special case, if $e_{k+l} = e_b$, then the gcd is 2^{b-k-1} .

The gcd of the exponents depends on the gcd result of $2^k + 1$ and $2^{k+l} + 1$. The expressions are coprime when $\nu_2(k) \neq \nu_2(k+l)$.

Given the same k, l , $\gcd(2^k - 1, 2^{k+l}) = 1$ implies $\gcd(2^k + 1, 2^{k+l} + 1) = 1$ but not the reverse. Hence, the first scheme generates moduli with exponents that are more likely to have a larger gcd than the second scheme. This further implies that the first scheme generates inverses with fewer set bits than the second scheme given the same k, l .

It is worth noting that oftentimes the two “greedy” schemes are not optimal in terms of the number of set bits in the inverses. We compare the total number of terms in the inverses given an exponent set of size b in polynomial form. For example, if we want to generate a moduli set of size $b = 6$, the first scheme generates the exponents $\{63, 62, 60, 56, 48, 32\}$. The total amount of terms in the inverses is 289. The second scheme generates the exponents $\{48, 40, 36, 34, 33, 32\}$. The total amount of terms in the inverses is 233. However, if we consider an exponent set that follows neither of the two schemes, $\{63, 56, 48, 42, 36, 32\}$, the total amount of terms in the inverses is 141.

Chapter 5

Implementation

To demonstrate the effectiveness of the benefits of the proposed method, it is essential to benchmark the performance of the proposed method with a real-world situation. One typical use case of modular arithmetic is in matrix multiplication with enormous entries in bit size. We have implemented the reduction and reconstruction algorithms in C++ using the GNU Multiple Precision Arithmetic Library (GMP).

The implementation is grouped under two main abstract classes/interfaces: `Modulus` and `Reconstruction`.

Before examining the main classes, we will first look at the helper classes that provide necessary supporting data structures.

5.1 Helper Classes

5.1.1 SparseBits

The `SparseBits` class stores the sparse representation of a number. For example, the number $2^{100} + 2^{50} + 2^0$ can be compactly stored as `SparseBits(100, 50, 0)`. The `SparseBit` class contains a vector of `mp_bitcnt_t` to store the exponents.

The class provides the following interfaces for a `SparseBits` object, `s`:

- `s.expression()`: returns a string that represents `SparseBits` in the form of $2^x + 2^y + 2^z + \dots$

- `s.exponents()`: returns the underlying array in the form `x, y, z, ...`.
- `s.convert(mpz_num)`: converts the `SparseBits` class to an `mpz_t` type and store the result in `mpz_num`.
Require: `mpz_num` has already been initialized
- `SparseBits_multiply(result, op1, op2)`: Multiplying a `mpz_t` number by a `SparseBits` number. The multiplication by `SparseBits` is performed by shifting and adding.
Require: `result` and `op1` have already been initialized.

5.1.2 MultiplicationByShifting

The `MultiplicationByShifting` class is a wrapper around the `SparseBits` class such that it allows the multiplication in the form of $2^x + 2^y + 2^z \dots - (2^a + 2^b - 2^c \dots)$ to be performed using bit shifting (the same method used in `SparseBits`)

The class provides the following interfaces for a `MultiplicationByShifting` object, `m`:

- `m.expression()`: returns a string that represents the multiplication by shifting in the form $2^x + 2^y + 2^z \dots - (2^a + 2^b - 2^c \dots)$.
- `m.exponents()`: returns a string of the underlying `SparseBits` in the form `x, y, z, ..., a, b, c, ...`
- `m.convert(mpz_num)`: converts the `MultiplicationByShifting` class to an `mpz_t` type and store the result in `mpz_num`.
Require: `mpz_num` has already been initialized.
- `MultiplicationByShifting_multiply(result, op1, m)`: multiply a `mpz_t` number `op1` by `m` through multiplication by shifting and store the result in `result`.
Require: `result` and `op1` have already been initialized.

5.2 Modulus

The `Modulus` abstract class defines the following interfaces for a modulus m :

- `m.expression()`: returns a string that represents the modulus
- `m.convert(mpz_num)`: converts the modulus class into `mpz_t` and store the result in `mpz_num`
Require: `mpz_num` has already been initialized
- `m.reduce(mpz_num)`: reduces the `mpz_num` using the modulus in place.
Require: `mpz_num` has already been initialized
- `m.reduce(mpz_store, mpz_num)`: reduces the `mpz_num` using the modulus and store the result in `mpz_store`.
Require: `mpz_store` and `mpz_num` have already been initialized.
- `m.convertToMultiplicationByShifting()`: converts the modulus into a `MultiplicationByShifting` object and return it.

In the implementation, we name Mersenne type numbers as `Marge` (Minus-Large) and Fermat type numbers as `Parge` (Plus-Large). The concrete classes (e.g. `Marge`, `Parge`, etc.) override the above interfaces by implementing suitable algorithms for the specific modulus.

5.3 Reconstruction

`Reconstruction` abstract class defines the following interfaces for a reconstruction r :

- `r.reconstruct(answer, residues, moduli)`: reconstructs the answer from the given residues and moduli.

The concrete class (e.g. `Garner`, `MargeReconstruction`, etc.) overrides the above interfaces by implementing suitable algorithms for the specific reconstruction.

The reconstruction algorithm is based on Garner's algorithm (Algorithm 5.1 of [8]).

Algorithm 2 SIMPLE(r, m)

Require: $\forall i \in [0, N - 1] : 0 \leq r_i < m_i$ **Ensure:** $a \equiv r_i \pmod{m_i}$

```
1:  $M \leftarrow 1$ 
2: for  $i = 1$  to  $N - 1$  do
3:    $M \leftarrow Mm_{i-1}$ 
4:    $M_i \leftarrow M^{-1} \pmod{m_i}$ 
5: end for
6:  $a_0 \leftarrow r_0$ 
7:  $M \leftarrow m_0$ 
8: for  $i = 1$  to  $N - 1$  do
9:    $t \leftarrow r_i - a$ 
10:   $t \leftarrow tM_i \pmod{m_i}$ 
11:   $a \leftarrow a + tM$ 
12:   $M \leftarrow Mm_i$ 
13: end for
14: return  $a$ 
```

The description of standard Garner’s algorithm can be found in many Computer Algebra texts. The algorithm computes the mixed-radix digits first and then combines them to get the final result. However, for our purposes, only the final result is desired. Hence, we use a simpler version of Garner’s algorithm (Algorithm 2) that only computes the final result.

There is still some technicality that we need to address. In Line 9 of Algorithm 2, t is assigned the value of $r_i - a$. There is no guarantee that t will be positive. In Line 10, this possibly negative t is multiplied by a product of moduli M_i and then reduced by m_i . This made the fast multiplication through shifting and adding difficult as it requires both operands to be positive.

This problem can be circumvented by exploiting the internal representations of the GMP integer. The sign of a GMP integer is represented by the sign of the size of the limb (`_mp_size`) [6]. This design choice makes negating a GMP integer a constant time operation. Hence, we can negate t if necessary and perform the fast multiplication and reduction as usual. After the multiplication and reduction, we can negate again if necessary.

At the time of writing this thesis, the only supported data structure for benchmarking is a custom-written `SquareMatrix` class with basic arithmetic operations (addition, scalar-matrix multiplication, matrix-matrix multiplication).

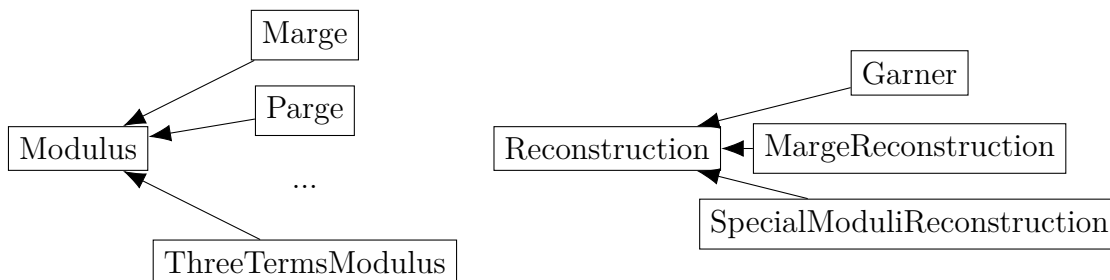


Figure 5.1: The class hierarchy diagram

5.4 Bits Extraction

Extracting bits from integers is a common operation in computer science. In practice, the numbers are stored in multi-precision format. One of the renowned examples is the GNU Multiple Precision Arithmetic Library (GMP).

It is mentioned [16] that the extraction time complexity of $\text{BITS}(n, a, b)$ is not linear in $b - a$ due to the restriction of the GMP library at the time of writing. Extracting a range of bits required two function calls, `mpz_tdiv_r_2exp` and `mpz_tdiv_q_2exp`, making the time complexity linear with respect to the number of bits in n .

It turns out that an extraction linear to $b - a$ is possible. However, it needs to be done by using the internals of a GMP integer.

The idea is to access the underlying internal bits of a GMP integer. The internal bits are stored in an array of limbs. Each limb is a machine word. We can copy only the limbs that are needed by the extraction operation to a new GMP integer. We can then perform `mpz_tdiv_r_2exp` and `mpz_tdiv_q_2exp` linearly with respect to $b - a$.

5.5 Benchmark

In this section, we will evaluate the performance of specially chosen moduli on different large matrix multiplication workloads and show their effectiveness in accelerating the computation. For convenience, we made the assumption that all the matrix entries are non-negative and that the representation used in RNS is the non-negative representation instead of the symmetric representation.

Setup. All the benchmarks were performed on a server with AMD EPYC 7502P 32C @ 2.5 GHz with 503GB of RAM. The matrix calculation happens on the custom-written `SquareMatrix` class which wraps around different computation schemes (FFLAS, GMP, special moduli RNS, etc.).

5.5.1 RNS Conversion Comparison

To demonstrate the effectiveness of the special form moduli on conversion from/to RNS. We compare the time spent on the conversion of a 32×32 matrix to RNS and immediate reconstruction from RNS for different moduli sets.

Metrics. We measure the time spent on reduction and reconstruction using different moduli sets. The reconstruction time for Fermat type and “trinomial” type moduli does not include the time to compute inverses, as the inverses are obtained from scaling.

Table Legend.

- “Size”: the bitsize of the matrix entries.
- “R.Red”: seconds spent for the regular (division-based) reduction.
- “M.Red”: seconds spent for the reduction by moduli of the form $2^n - 1$.
- “T.Red”: seconds spent for the reduction by moduli of the form $2^n - 2^k + 1$.
- “P.Red”: seconds spent for the reduction by moduli of the form $2^n + 1$.
- “R.Rec”: seconds spent for the regular reconstruction [8] (time to compute inverses + reconstruction time).
- “M.Rec”: seconds spent for the reconstruction for moduli of the form $2^n - 1$ (time to compute inverses + reconstruction time).
- “T.Rec”: seconds spent for the reconstruction for moduli of the form $2^n - 2^k + 1$.
- “P.Rec”: seconds spent for the reconstruction for moduli of the form $2^n + 1$.

Size	R.Red	M.Red	T.Red	P.Red	R.Rec	M.Rec	T.Rec	P.Rec
2^{18}	3.23	0.09	0.18	0.10	1.98 + 4.49	1.43 + 1.73	0.27	0.87
2^{19}	8.59	0.18	0.36	0.19	4.92 + 11.34	3.46 + 4.15	0.54	1.74
2^{20}	21.08	0.34	0.73	0.36	12.90 + 26.55	9.41 + 10.05	1.08	3.46
2^{21}	54.60	0.67	1.41	0.70	29.48 + 60.29	21.95 + 24.19	2.15	6.92
2^{22}	116.48	1.31	2.81	1.38	69.59 + 134.46	53.50 + 54.94	4.32	14.28

Table 5.1: Timing (in seconds) for the input 32 by 32 matrix conversion to RNS and immediate reconstruction from RNS.

Analysis. The results show that the conversion to/from RNS is significantly faster than regular techniques when using special form moduli.

The reduction time for moduli of the shape $2^n \pm 1$ is faster than the “trinomial” type moduli ($2^n - 2^k + 1$) due to its straightforward reduction process. However, the “trinomial” and Fermat type moduli allow for a faster reconstruction process as the inverses can be precomputed from scaling. The “trinomial” type moduli set has a faster reconstruction time compared to the Fermat type moduli set due to fewer set bits in the SBB form of the inverse.

Overall, the “trinomial” type moduli set demonstrates a better performance in converting to and from RNS.

5.5.2 Two-level Modular Scheme

At this moment, the well-studied special scalable moduli sets are very few, and the sets found are quite small. The power of modular arithmetic only shows when the size of the moduli set is large. Also, to show the effectiveness of such a moduli set, we need to perform computations of large numbers. Hence, it is natural to combine the power of special-form moduli and machine-size moduli to form a two-level modular scheme.

The idea is to select large moduli discussed in the previous sections on the first layer, and reduce the problem to several problems with entries bit-size amenable for FFLAS-FFPACK. On the second layer, simultaneous conversion [5] is used. Results from multiple calls to FFLAS-FFPACK are used to reconstruct the final answer using accelerated reconstruction with specially selected moduli.

Metrics. We measure the time spent on the computation of the matrix product and the time spent on the conversion to and from RNS. Any additional time such as randomizing the matrix entries is excluded from the measurements to our best ability.

Table Legend.

- “Dim”: matrix dimension.
- “Size”: bitsize of the matrix entries.
- “FFLAS”: direct use of FFLAS-FFPACK machine size prime RNS implementation.
- “Marge”: first layer with 9 moduli of the form $2^n - 1$.
- “Parge”: first layer with 9 moduli of the form $2^n + 1$ with 2^n .
- “Trinom”: first layer with 9 moduli of the form $2^n - 2^k + 1$ with 2^n and $2^n + 1$.
- “overh”: time spent for conversion to and from RNS in the first layer (note, that in column Marge overhead includes time to compute inverses, while in column Trinom inverses are not computed and obtained from scaling).
- “GMP”: time spent on the computation of the matrix product using the GMP library directly.
- “DNF”: did not finish in 10 hours.

Dim	Bitsize	FFLAS	Marge	Parge (overh)	Trinom (overh)	GMP
8	$\approx 2^{15}$	1.09	0.17 (0.06)	0.23 (0.06)	0.14 (0.02)	0.02
-	$2^{18} - 2^{19}$	102.69	8.14 (1.10)	10.36 (0.44)	7.11 (0.24)	0.48
-	$\approx 2^{19}$	406.58	29.09 (2.60)	39.07 (0.86)	27.01 (0.43)	1.01
-	$\approx 2^{20}$	1627.49	111.21 (6.17)	152.80 (1.68)	105.43 (0.84)	2.14
-	$\approx 2^{21}$	DNF	241.43 (10.38)	341.30 (2.42)	234.49 (1.24)	3.74
16	$\approx 2^{15}$	1.18	0.44 (0.20)	0.50 (0.20)	0.35 (0.12)	0.18
-	$2^{18} - 2^{19}$	105.31	13.23 (4.27)	14.80 (1.60)	9.68 (0.81)	3.83
-	$\approx 2^{19}$	416.94	45.02 (10.26)	53.55 (3.26)	35.66 (1.55)	8.09
-	$\approx 2^{20}$	1656.10	156.94 (24.49)	200.43 (6.33)	136.28 (3.16)	17.05
-	$\approx 2^{21}$	DNF	335.35 (41.18)	450.83 (9.46)	298.61 (4.78)	29.88
32	$\approx 2^{15}$	1.61	1.45 (0.71)	1.63 (0.68)	1.06 (0.38)	1.38
-	$2^{18} - 2^{19}$	124.33	38.40 (16.86)	34.80 (6.33)	23.72 (3.11)	30.58
-	$\approx 2^{19}$	479.36	110.34 (40.96)	113.14 (12.64)	75.88 (6.13)	64.71
-	$\approx 2^{20}$	2003.69	361.60 (97.89)	407.95 (25.29)	276.36 (12.46)	136.14
-	$\approx 2^{21}$	DNF	750.86 (164.50)	901.29 (37.81)	600.62 (18.67)	238.54
64	$\approx 2^{15}$	3.83	4.67 (2.73)	6.59 (2.60)	3.40 (1.39)	11.41
-	$2^{18} - 2^{19}$	175.90	178.32 (67.27)	168.41 (25.09)	122.48 (12.37)	244.62
-	$\approx 2^{19}$	657.89	467.61 (163.80)	448.42 (50.55)	328.90 (24.23)	518.14
-	$\approx 2^{20}$	2638.34	1,257.15 (391.69)	1,296.44 (100.38)	917.87 (49.68)	1089.47
-	$\approx 2^{21}$	DNF	2,417.02 (656.20)	2,663.37 (150.48)	1,839.80 (74.55)	1912.54

Table 5.2: Timing (in seconds) of square integer matrix multiplication benchmark

Analysis. The results showed that the two-level modular scheme is effective in accelerating the computation of large matrix multiplication. One significant advantage of the two-level modular scheme is that it further extends the range of single-layer machine size moduli approach. The first layer of the scheme reduces the problem to smaller problems that can be solved by FFLAS-FFPACK with a small overhead. When the input bit size is larger than 2^{20} , the single-layer machine size moduli approach cannot finish the computation within a reasonable amount of time, while the two-level modular scheme can still compute the result quickly.

Comparing the results within the three different kinds of first-layer moduli sets, we can see moduli of the form $2^n - 1$ have the largest overhead. This is due to a slower reconstruction process of Mersenne type moduli as there are no known precomputation techniques for the inverses. The Fermat type moduli set has a slightly larger overhead compared to the “trinomial” type moduli set due a larger amount of set bits in the inverse.

Apart from the overhead, we observed that the “trinomial” type moduli set also spends less time on the computation within RNS. This is due to the fact that each modulus in the “trinomial” type moduli set has a balanced bit size. The scheme that generates Fermat

and Mersenne type moduli all have the problem of generating unbalanced moduli. When the input bit size is large, the unbalance in the moduli set will get amplified due to scaling, which results in a slower computation time within RNS.

Overall, the “trinomial” moduli set demonstrated a better overall performance due to its low overhead and balanced moduli in bit size.

By leveraging the two-level modular scheme, we were able to outperform the GMP library on 32 by 32 matrices with randomized entries up to 2^{19} bits, and on 64 by 64 matrices with randomized entries up to 2^{21} bits. The corresponding cells have been highlighted in the table.

Chapter 6

Conclusions

In this thesis, we investigated the acceleration of modular arithmetic through the use of sparse and scalable moduli. We described the sparse balanced binary representation (SBB) and defined the concept of scaling in SBB form. We explored the techniques of converting integers to and from RNS using special moduli, including Mersenne, Fermat, and “trinomial” type numbers, and the corresponding schemes to generate scalable moduli sets. We proposed a close relation between the inverses of the moduli in SBB form and the inverses under a polynomial setting, which allows for the generation of scalable moduli sets with Fermat type numbers.

The benefit of using special form moduli was demonstrated through benchmark results, showing significant speedups over conventional division methods. The two-level modular arithmetic scheme was shown to be effective in accelerating the computation of large matrix multiplication, offering scalable solutions for large integer operations.

While this thesis has demonstrated significant improvements in modular arithmetic using sparse and scalable moduli, several open questions remain. Future research could explore the following:

- A deterministic scheme to generate scalable moduli sets with “trinomial” type numbers.
- Closed-form expressions for the inverses of the sparse and scalable moduli in SBB form.
- Parallelizing the computation in RNS to further accelerate the computation.

References

- [1] Aviezri S. Fraenkel. “The Use of Index Calculus and Mersenne Primes for the Design of a High-Speed Digital Multiplier”. In: *Journal of the ACM* 8.1 (Jan. 1961), pp. 87–96. ISSN: 0004-5411, 1557-735X. DOI: [10.1145/321052.321057](https://doi.org/10.1145/321052.321057). URL: <https://dl.acm.org/doi/10.1145/321052.321057> (visited on 09/04/2024).
- [2] Daniel Bernstein. *Scaled Remainder Trees*. Aug. 20, 2004. URL: <https://cr.yp.to/arith/scaledmod-20040820.pdf>.
- [3] John J. Cade et al. “E3288”. In: *The American Mathematical Monthly* 97.4 (1990). Publisher: [Taylor & Francis, Ltd., Mathematical Association of America], pp. 344–345. ISSN: 00029890, 19300972. DOI: [10.2307/2324525](https://doi.org/10.2307/2324525). URL: <http://www.jstor.org.proxy.lib.uwaterloo.ca/stable/2324525> (visited on 09/06/2024).
- [4] Benjamin Chen, Yu Li, and Eugene Zima. “On a Two-Layer Modular Arithmetic”. In: *ACM Communications in Computer Algebra* 57.3 (Sept. 2023), pp. 133–136. ISSN: 1932-2240. DOI: [10.1145/3637529.3637534](https://doi.org/10.1145/3637529.3637534). URL: <https://dl.acm.org/doi/10.1145/3637529.3637534> (visited on 09/04/2024).
- [5] Javad Doliskani et al. “Simultaneous Conversions with the Residue Number System Using Linear Algebra”. In: *ACM Transactions on Mathematical Software* 44.3 (Sept. 30, 2018), pp. 1–21. ISSN: 0098-3500, 1557-7295. DOI: [10.1145/3145573](https://doi.org/10.1145/3145573). URL: <https://dl.acm.org/doi/10.1145/3145573> (visited on 07/21/2024).
- [6] Free Software Foundation, Inc. *Integer Internals (GNU MP 6.3.0)*. URL: <https://gmplib.org/manual/Integer-Internals>.
- [7] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Third edition. OCLC: 843762781. Cambridge: Cambridge University Press, 2013. ISBN: 978-1-139-85606-5.

- [8] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Boston, MA: Springer US, 1992. ISBN: 978-0-7923-9259-0 978-0-585-33247-5. DOI: [10.1007/b102438](https://doi.org/10.1007/b102438). URL: <http://link.springer.com/10.1007/b102438> (visited on 09/05/2024).
- [9] Darrel Hankerson, Scott Vanstone, and Alfred Menezes. *Guide to Elliptic Curve Cryptography*. OCLC: 1058873580. New York: Springer-Verlag, 2004. ISBN: 978-1-280-18846-6.
- [10] R.M. Hewlitt and E.S. Swartzlantler. “Canonical signed digit representation for FIR digital filters”. In: *2000 IEEE Workshop on SiGNAL PROCESSING SYSTEMS. SiPS 2000. Design and Implementation (Cat. No.00TH8528)*. 2000 IEEE Workshop on SiGNAL PROCESSING SYSTEMS. SiPS 2000. Design and Implementation. Lafayette, LA, USA: IEEE, 2000, pp. 416–426. ISBN: 978-0-7803-6488-2. DOI: [10.1109/SIPS.2000.886740](https://doi.org/10.1109/SIPS.2000.886740). URL: <http://ieeexplore.ieee.org/document/886740/> (visited on 08/25/2024).
- [11] A. A. Hiasat. “A Suggestion for a Fast Residue Multiplier for a Family of Moduli of the Form $(2n - (2p - 1))$ ”. In: *The Computer Journal* 47.1 (Jan. 1, 2004), pp. 93–102. ISSN: 0010-4620, 1460-2067. DOI: [10.1093/comjnl/47.1.93](https://doi.org/10.1093/comjnl/47.1.93). URL: <https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/47.1.93> (visited on 09/04/2024).
- [12] Ahmad Hiasat. “General Frameworks for Designing Arithmetic Components for Residue Number Systems”. In: *Intelligent Methods in Computing, Communications and Control*. Ed. by Ioan Dzitac et al. Cham: Springer International Publishing, 2021, pp. 82–92. ISBN: 978-3-030-53651-0.
- [13] Joris van der Hoeven. “Fast Chinese Remaindering in Practice”. In: *Mathematical Aspects of Computer and Information Sciences*. Ed. by Johannes Blömer et al. Cham: Springer International Publishing, 2017, pp. 95–106. ISBN: 978-3-319-72453-9.
- [14] Donald Ervin Knuth. *The art of computer programming. 2: Seminumerical algorithms*. Reading, Mass: Addison-Wesley, 1969. 624 pp. ISBN: 978-0-201-03802-6.
- [15] A. Schönhage. “Multiplikation großer Zahlen”. In: *Computing* 1.3 (Sept. 1966), pp. 182–196. ISSN: 0010-485X, 1436-5057. DOI: [10.1007/BF02234362](https://doi.org/10.1007/BF02234362). URL: <http://link.springer.com/10.1007/BF02234362> (visited on 05/29/2023).
- [16] Alex Stewart. “Base-2 Cunningham Numbers in Modular Arithmetic”. Master Thesis. University of Waterloo, Jan. 2006. 29 pp.

- [17] E. V. Zima and A. M. Stewart. “Cunningham numbers in modular arithmetic”. In: *Programming and Computer Software* 33.2 (Mar. 2007), pp. 80–86. ISSN: 0361-7688, 1608-3261. DOI: [10.1134/S0361768807020053](https://doi.org/10.1134/S0361768807020053). URL: <http://link.springer.com/10.1134/S0361768807020053> (visited on 07/21/2024).

APPENDICES

Appendix A

Appendix

A.1 Co-primality Results

It is well known (see [14]) that for integer $n > 0, m > 0$

$$\gcd(x^n - 1, x^m - 1) = x^{\gcd(n,m)} - 1. \quad (\text{A.1})$$

A.1.1 Co-primality of $x^n + 1$ and $x^m - 1$

Consider $x^n + 1$ and $x^m - 1$ for integer $n > 0, m > 0$. Let $\nu_2(a)$ be the binary valuation of integer a (the maximum degree of 2 that is contained in a). Then

$$\gcd(x^n + 1, x^m - 1) = \begin{cases} 1, & \nu_2(n) \geq \nu_2(m) \\ x^{\gcd(n,m)} + 1, & \text{otherwise} \end{cases}$$

Elementary proof based on (A.1) uses the following simple facts:

1.

$$\gcd(2n, m) = \begin{cases} \gcd(n, m), & \nu_2(n) \geq \nu_2(m) \\ 2 \gcd(n, m), & \text{otherwise} \end{cases}$$

Write $n = 2^{\nu_2(n)} n_o$, $m = 2^{\nu_2(m)} m_o$ with odd n_o, m_o , and use $\gcd(n, m) = 2^{\min(\nu_2(n), \nu_2(m))} \gcd(n_o, m_o)$, $\gcd(2n, m) = 2^{\min(\nu_2(n)+1, \nu_2(m))} \gcd(n_o, m_o)$.

2. Polynomials $x^n + 1$ and $x^n - 1$ are relatively prime, as

$$(x^n + 1) - (x^n - 1) = 2,$$

and any common factor must divide 2. Therefore, for any polynomial $f(x)$

$$\gcd(x^{2n} - 1, f(x)) = \gcd(x^n - 1, f(x)) \gcd(x^n + 1, f(x)),$$

or

$$\gcd(x^n + 1, f(x)) = \frac{\gcd(x^{2n} - 1, f(x))}{\gcd(x^n - 1, f(x))}.$$

Now

$$\gcd(x^n + 1, x^m - 1) = \frac{\gcd(x^{2n} - 1, x^m - 1)}{\gcd(x^n - 1, x^m - 1)} = \frac{x^{\gcd(2n, m)} - 1}{x^{\gcd(n, m)} - 1}.$$

If $\nu_2(n) \geq \nu_2(m)$ then $\gcd(x^n + 1, x^m - 1) = 1$.

If $\nu_2(n) < \nu_2(m)$ then $\gcd(x^n + 1, x^m - 1) = \frac{x^{2^{\nu_2(n)} \gcd(n, m)} - 1}{x^{\gcd(n, m)} - 1} = x^{\gcd(n, m)} + 1$.

A.1.2 Co-primality of $x^n + 1$ and $x^m + 1$

Based on the above we have very short proof of the following

$$\gcd(x^n + 1, x^m + 1) = \begin{cases} 1, & \nu_2(n) \neq \nu_2(m) \\ x^{\gcd(n, m)} + 1, & \text{otherwise} \end{cases}$$

For $\nu_2(n) = \nu_2(m)$ we have

$$\gcd(x^{2n} - 1, x^m + 1) = \gcd(x^n - 1, x^m + 1) \gcd(x^n + 1, x^m + 1).$$

Since $\gcd(x^{2n} - 1, x^m + 1) = x^{\gcd(2n, m)} + 1$, as $\nu_2(2n) > \nu_2(m)$ and $\gcd(2n, m) = \gcd(n, m)$ result follows.

For $\nu_2(n) \neq \nu_2(m)$ assume $\nu_2(n) < \nu_2(m)$ (opposite case is symmetric,). We have (as $\nu_2(2n) \leq \nu_2(m)$)

$$1 = \gcd(x^{2n} - 1, x^m + 1) = \gcd(x^n - 1, x^m + 1) \gcd(x^n + 1, x^m + 1),$$

and result follows.

A.2 Justifications of Basic Facts

Consider 2^u divided by $2^v - 1$ or $2^v + 1$. Write $u = qv + r$, $0 \leq r < v$.

- 2^u divided by $2^v - 1$

We claim that $R = 2^r$ and $Q = 2^{u-v} + 2^{u-2v} + \dots + 2^{u-qv}$. Symbolically,

$$2^u = 2^{qv+r} = Q \cdot (2^v - 1) + R$$

where $0 \leq R < 2^v - 1$.

Base case, when $q = 0$:

$$2^u = 0 \cdot (2^v - 1) + 2^r$$

Suppose that the statement is true for $q = k$, we show that it is also true for $q = k+1$:

$$\begin{aligned} 2^{(k+1)v+r} &= 2^{kv+r} \cdot 2^v \\ &= \left((2^{(k-1)v+r} + 2^{(k-2)v+r} + \dots + 2^r) (2^v - 1) + 2^r \right) \cdot 2^v \\ &= (2^{kv+r} + 2^{(k-1)v+r} + \dots + 2^{v+r}) (2^v - 1) + 2^{r+v} \\ &= (2^{kv+r} + 2^{(k-1)v+r} + \dots + 2^{v+r} + 2^r) (2^v - 1) + 2^r \end{aligned}$$

Hence, the statement is true for all $q \geq 0$.

Since $r \geq 0$, the case 1 and 2 follows. (For the +1 case, when v is small, the divisor may further divide R .)

- 2^u divided by $2^v + 1$

We claim that $R = (-1)^q \cdot 2^r$ and $Q = (-1)^0 \cdot 2^{u-v} + (-1)^1 \cdot 2^{u-2v} + \dots + (-1)^{q-1} \cdot 2^{u-qv}$. Symbolically,

$$2^u = 2^{qv+r} = Q \cdot (2^v + 1) + R$$

where $0 \leq R < 2^v + 1$.

Base case, when $q = 0$:

$$2^u = 0 \cdot (2^v + 1) + (-1)^0 \cdot 2^r$$

Suppose that the statement is true for $q = k$, we show that it is also true for $q = k+1$:

$$\begin{aligned}
& 2^{(k+1)v+r} \\
&= 2^{kv+r} \cdot 2^v \\
&= \left((-1)^0 \cdot 2^{(k-1)v+r} + (-1)^1 \cdot 2^{(k-2)v+r} + \dots + (-1)^{k-1} \cdot 2^r \right) (2^v + 1) + (-1)^k \cdot 2^r \cdot 2^v \\
&= \left((-1)^0 \cdot 2^{kv+r} + (-1)^1 \cdot 2^{(k-1)v+r} + \dots + (-1)^{k-1} \cdot 2^{v+r} \right) (2^v + 1) + (-1)^k \cdot 2^{r+v} \\
&= \left((-1)^0 \cdot 2^{kv+r} + (-1)^1 \cdot 2^{(k-1)v+r} + \dots + (-1)^{k-1} \cdot 2^{v+r} + (-1)^k \cdot 2^r \right) (2^v + 1) \\
&\quad + (-1)^{k+1} \cdot 2^r
\end{aligned}$$

Hence, the statement is true for all $q \geq 0$.

Since $r \geq 0$, the case 3 and 4 follow. When the remainder is negative, namely when q is odd, we can add 2^v to the remainder to make it positive such that it is a proper division.