APPROVAL SHEET

Title of Thesis: Fast Modular Exponentiation Using Residue Domain Representation: A Hardware Implementation and Analysis

Name of Candidate: Christopher Dinh Nguyen
Master of Science, 2013

Thesis and Abstract Approved: _____

Alan T. Sherman
Associate Professor
Department of Computer Science and
Electrical Engineering

_____

Dhananjay S. Phatak
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

CURRICULUM VITAE

Name: Christopher Dinh Nguyen.
Degree and date to be conferred: Master of Science, December 2013.
Secondary Education: Gwynn Park High, Brandywine, MD, 2002.
Collegiate institutions attended:

University of Maryland, Baltimore County,
Doctor of Philosophy, Computer Science, 2011–Present.
Master of Science, Computer Science, 2013.
Bachelor of Science, Computer Science, 2006.
Bachelor of Science, Mathematics, 2006.

Major: Computer Science.
Minor: None.

Professional positions held:

Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County. Research Assistant. (2013–Present).

Lockheed Martin. Senior Software Engineer. (2007–Present).

Department of Mathematics, Community College of Baltimore County. Adjunct Professor. (2012–2013).

Automation Technologies, Inc. (ATI). Systems Analyst. (2006–2007).

Department of Mathematics, University of Maryland, Baltimore County. Teaching Assistant. (2005–2006).

Learning Resource Center, University of Maryland, Baltimore County. Tutor. (2005–2006).

ABSTRACT

Title of Thesis: Fast Modular Exponentiation Using Residue Domain Representation: A Hardware Implementation and Analysis

Christopher Dinh Nguyen, Master of Science, 2013

Thesis directed by: Alan T. Sherman, Associate Professor
Department of Computer Science and
Electrical Engineering
Dhananjay S. Phatak, Associate Professor
Department of Computer Science and
Electrical Engineering

Using modular exponentiation as an application, we engineered on FPGA fabric and analyzed the first implementation of two arithmetic algorithms in Reduced-Precision Residue Number Systems (RP-RNS): the partial-reconstruction algorithm and quotient-first scaling algorithm.

Residue number systems (RNS) provide an alternative representation to the binary system for computation. They offer full parallel computation for addition, subtraction, and multiplication. However, base extension, division, and sign detection become harder operations. Phatak's RP-RNS uses a time-memory trade-off to achieve $\mathrm{O}(\lg N)$ running time for base extension and scaling, where N is the bit-length of the operands, compared with Kawamura's Cox-Rower architecture and its derivatives, which appear to take $\mathrm{O}(N)$ steps and therefore $\mathrm{O}(N)$ delay to the best of our knowledge.

We implemented the fully parallel RP-RNS architecture based on Phatak's description and architecture diagrams. Our design decisions included distributing the lookup tables among each channel, removing the adder trees, and removing the parallel table access thus trading size for speed. In retrospect, we should have hosted the tables in memory off the FPGA.

We measured the FPGA utilization, storage size, and cycle counts. The data we present, though less than optimal, confirms the theoretical trends calculated by Phatak. FPGA utilization grows proportional $K \log K$ where K is the number of hardware channels. Storage grows proportional to $\mathrm{O}\left(N^3 \lg \lg N\right)$. When using Phatak's recommendations, cycle count grows proportional to $\mathrm{O}\left(\lg N\right)$.

Our contributions include documentation of our design, architecture, and implementation; a detailed testing methodology; and performance data based on our implementation to enable others to replicate our implementation and findings.

Fast Modular Exponentiation Using Residue Domain Representation: A Hardware Implementation and Analysis

by

Christopher Dinh Nguyen

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Master of Science
2013

To my partner, Philip Vatcher, whose endless love and support continues to motivate me.

ACKNOWLEDGMENTS

First, I thank Dhananjay Phatak for the initial proposal to work on this project. His theoretical work on the reduced-precision residue number system (RP-RNS), which includes the RP-RNS algorithms and preliminary analysis, made this research possible.

Alan Sherman, as my advisor, provided helpful feedback on my drafts and provided general research advice. I look forward to my continuing work with him as I pursue the Ph.D. degree.

Chintan Patel and Ryan Robucci provided references and material regarding hardware design, hardware design languages, and debugging tools, which enabled me to realize the reduced-precision residue number system algorithms in FPGA hardware.

I am grateful to Stacey Hertz, Kristen Lantz, Daniel Eden, and other the leaders at Lockheed Martin Cyber Solutions who wished to remain anonymous for their continued support, advice, and recommendations. Working with our customers, they ensured I had the scheduling flexibility needed to complete my research.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Thesis 3

1.3 Aims of Our Work 3

1.4 Summary of Contributions 4

1.4.1 Hardware Verification and Performance Data 4

1.4.2 Testing Methodology and Enabling Repeatability 5

1.5 Outline 5

2 Background **7**

2.1 Mathematical Background 7

2.1.1 Divisibility 8

2.1.2 Ring of Integers Modulo n 10

2.1.3 Chinese Remainder Theorem 12

2.2 Computer Arithmetic Background 14

2.2.1 Number Systems 15

2.2.2 Algorithms and Implementations 20

2.3 Implementation Fabrics 30

2.4 Key Points 33

3 Related Work **34**

3.1 RNS-based Montgomery's Multiplication 34

3.2 Base Extensions for RNS-based Montgomery 35

3.3 RNS-based Montgomery Implementations 37

4 Reduced-Precision Residue Number System (RP-RNS) **39**

4.1 RP-RNS Defined 40

4.2 Forward Conversion Algorithm 42

4.2.1 Algorithm Description 42

4.2.2 Time and Space Analysis 43

4.3 Partial Reconstruction Algorithm 44

4.3.1 Pre-Computed Lookup Tables 45

4.3.2 Algorithm Description 46

4.3.3 Time and Space Analysis 49

4.4 Quotient-First Scaling Algorithm 49

4.4.1 Pre-Computed Lookup Tables 49

4.4.2 Algorithm Description 50

4.4.3 Time and Space Analysis 54

4.5 Modular Exponentiation Algorithm 55

4.5.1 Algorithm Description 55

4.5.2 Time and Space Analysis 56

4.6 Reverse Conversion Algorithm 58

4.7 RP-RNS Design Strategies 59

4.8 Conclusion 60

5 RP-RNS Implementations **61**

5.1 Software Implementation 61

5.2 Hardware Platform 62

5.3 Hardware Design 63

5.4 Hardware Architecture 66

5.5 Hardware Implementation 68

5.5.1 Controller 68

5.5.2 Hardware Channel 69

5.5.3 Redundant Residue Channel 70

5.5.4 Fraction Channel 71

5.6 Remarks 71

6 Testing Methodology and Results **72**

6.1 Purpose 73

6.2 Testing Methodology 73

6.2.1 Test Apparatus Descriptions 74

6.2.2 Test Apparatus Validation 74

6.2.3 Metric Description and Collection 77

6.3 Test Results 77

6.3.1 FPGA Utilization Metrics 78

6.3.2 Storage Metrics 81

6.3.3 Cycle Count Metrics 83

6.4 Analysis 85

7 Conclusion **88**

7.1 Design Improvements 88

7.2 Reflecting on Performance 90
7.3 Open Problems and Future Work 91
7.4 Contributions 92
7.5 Final Musings 92

A Notation **94**

B Tables of Performance Metrics **95**

Bibliography **107**

List of Tables

2.1 Summary Comparison of ASICs and FPGAs 32

4.1 Forward Conversion Algorithm ROMs (period in bold) 44

4.2 Partial Reconstruction Algorithm ROMs (Scaled by $2^6 = 64$) 46

4.3 QFS Algorithm Table-1 for our example. 51

4.4 QFS Algorithm Table-2 for our example. 53

4.5 RP-RNS modular multiplication results by iteration for Example 4. 57

4.6 Summary of running times for the RP-RNS algorithms. 57

4.7 Modular exponentiation space requirements in bits. 58

5.1 Xilinx Spartan-3E (XC3S500E) specifications 63

5.2 RP-RNS controller composition. 69

6.1 Utilization of Spartan-3E (XC3S500E) FPGA for K channels. 78

6.2 Inferred logic for our hardware implementation. 80

B.1 Cycle count for $p = 1$ sequential implementation. 95

B.2 Cycle count for $p = 3$ semi-parallel implementation. 99

B.3 Cycle count for $p = K$ full-parallel implementation. 103

List of Figures

2.1 Structure of an FPGA configurable logic block. 31
2.2 An example structure of an FPGA logic cell. 32

5.1 Spartan-3E development board by Digilent, Inc. 64
5.2 RP-RNS Architecture 67

6.1 Superlinear dependence of FPGA logic utilization on RNS base size. 80
6.2 Total size of precomputed lookup tables for our implementation. 82
6.3 Cutout of storage requirements for our synthesized hardware from Figure 6.2. 82
6.4 Cycle counts for divisors up to 1024-bits using four varieties of parallelization. 84
6.5 Cutout of theoretical cycle counts from Figure 6.4. 84

Chapter 1

Introduction

Computer arithmetic hardware – performing as basic operations addition, subtraction, multiplication, and division – underlies almost all intensive computation. Most arithmetic hardware use the binary number system to perform computations. We explore fast integer arithmetic hardware based in residue number systems (RNSs) using modular exponentiation as an application.

While binary works similarly to the decimal number system, in contrast RNSs represent a number as residues with respect to a set of moduli called the RNS base. RNSs are promising number systems because addition, subtraction, and multiplication are parallelizable and efficient hardware algorithms already exist for these operations. In contrast, other operations such as division become harder operations to compute. This limitation extends to modular reduction and modular exponentiation using modular reduction. Consequently RNS only finds use in special-computing applications such as digital signal processing or cryptography [24, 28].

Modular exponentiation as an application is interesting due to its significance in cryptography [13, 15, 34]. Moreso, RNS-based modular exponentiation has seen a theoretical

speed increase due to Phatak's recent algorithmic breakthroughs in partial reconstruction and scaling (i.e., division with constant divisor), which we call the reduced-precision residue number system (RP-RNS) [30–33]. The special feature of the RP-RNS algorithms is the use of low-precision approximations of fractions to narrow the result to two candidates before disambiguating the final result using a redundant residue channel requiring no more than 2 bits.

1.1 Motivation

An interesting property of RNSs is the maximum execution time to perform a parallelizable operation (e.g., addition) depends solely on the largest modulus in the RNS base. So using many small moduli is preferable to few large moduli for the parallelizable operations.

Software implementations of RNSs in general cannot take full advantage of parallelization when running on commodity hardware. As of the year 2013, most commodity hardware features at most 16 cores with 64-bit arithmetic logic units [11, 18, 29] and 64GB of memory. Favoring small moduli would not only waste the 64-bit hardware, but the RNS base size would exceed the number of cores available by a few orders of magnitude for system sizes we care about (i.e., greater than 1024 bits).

As an example, an RP-RNS using the first 15 prime numbers (the last core reserved for the redundant residue channel) achieves a 60-bit main modulus with at most 6-bit channels. This is an unacceptable waste of hardware resources. We can mitigate this waste by using moduli optimal for the hardware (e.g. 32-bit or 64-bit moduli), but even then software may not leverage all available processing cores making the software effectively sequential [39] thus forfeiting the benefits of RNSs. This motivates us to create specialized RP-RNS hardware.

1.2 Thesis

RP-RNS is a general approach to faster computation of harder RNS operations including the scaling operation. Large range RP-RNSs computing on numbers thousands of bits long requires a significant quantity of hardware to minimize storage and execution time. We believe we can engineer efficient RP-RNS hardware that can scale to word lengths several thousands of bits long.

1.3 Aims of Our Work

A preliminary analysis of Phatak's RP-RNS algorithms indicate a reduction in running time complexity to $\mathrm{O}\left(N \lg N\right)$ where N is the number of bits needed to represent a number in a given RNS. We can verify this claim by producing one efficient hardware implementation.

Current literature on residue domain modular exponentiation hardware use divisors of at least 1024 bits. So the first goal of our work is to engineer RP-RNS modular exponentiation hardware supporting at least a 1024-bit divisor to provide sufficient comparison to existing literature including Nozaki, et al. [27], and Gandino, et al. [16].

From there we want to determine whether there is a divisor limit for engineering our design using commodity field-programmable gate arrays (FPGAs) accessible to the average technology consumer. For concreteness, we assume this equates roughly to either 16 low-capacity FPGAs or 2 high-capacity FPGAs based on price.

Our third aim is to analyze our implementation, improve it, and compare its running time performance against the current state of the art, which is Gandino, et al. [16].

1.4 Summary of Contributions

This thesis contributes a hardware implementation of RP-RNS modular exponentiation based on Phatak's architecture [33]. Our implementation is usable without modification as a modular exponentiation component for traditional binary-based digital system. We summarize our contributions below, emphasizing the theoretical work and analysis of the RP-RNS were done by Phatak.

1.4.1 Hardware Verification and Performance Data

We verified our implementation through software simulations; then described the architecture in VHDL and simulated an FPGA realization of that HDL architecture. Finally we also performed a limited amount of work on actual physical FPGAs, synthesizing small toy prototype in actual hardware. Experiments on physical hardware were limited in scope due to two reasons: our design decision to store the precomputed lookup tables on the FPGA and an intermittent problem we identified with the vendor's FPGA synthesis software when generating the lookup tables and constants. While not a contribution, we acknowledge in retrospect we should have stored the tables in memory off the FPGA.

We extracted measurements directly from the hardware up to the limits of our implementation (15 channels). We extrapolated measurements via simulation for larger systems. The FPGA utilization grew at a $K \lg K$ rate where K is the size of the RNS base. The lookup table storage grew at a rate of $N^3 \lg \lg N$ rate, which agrees with the theoretical expectation, but ours had a large coefficient due to our design decision stated above. The execution time of a fully parallel implementation grew at the expected logarithmic rate with respect to the system modulus word length, N. The execution time of a sequential implementation grew

at a quadratic rate with respect to the divisor word length. Taking our decision design into account, this agrees with the theoretical expectation.

1.4.2 Testing Methodology and Enabling Repeatability

We provide a complete description of our design, architecture, and implementation to enable future researchers to recreate our hardware. We also provide a fully documented testing methodology including verification and validation. Together with our data, these contributions will enable future researchers to replicate our implementation and results.

1.5 Outline

Chapter 2 provides background information for this thesis. This thesis combines information from modern algebra, computer arithmetic, and very-large-scale integration (VLSI) hardware.

Chapter 3 discusses related work on RNS modular exponentiation hardware algorithms and implementations. This chapter is brief because RNS modular exponentiation is relatively new and the literature is sparse.

Chapter 4 details the theory behind our implementation. We introduce Phatak's reduced-precision residue number system [33], RP-RNS design strategies, and four of his algorithms: partial reconstruction (PR) [30], quotient-first scaling (QFS), modular reduction using QFS, and RNS modular exponentiation using QFS [31].

Chapter 5 contains our RP-RNS modular exponentiation implementations. We explain our design decisions, architecture, and implementation details. The discussion includes how our design deviates from Phatak's recommendations and the theoretical performance.

Chapter 6 presents results from experiments performed with our RP-RNS modular exponentiation implementation. We explain our testing methodology including validation and verification. We present our results covering the metrics of FPGA utilization, storage, and running time; we do not measure nor analyze power consumption. We also analyze our results.

Chapter 7 concludes this thesis with potential implementation improvements, reflections on our implementation, open problems, and ideas for future work. Appendix A is a reference of notation we use in this thesis. Appendix B contains tables of data related to Chapter 6.

Chapter 2

Background

The algorithms and VLSI architectures investigated in this thesis require background from computer arithmetic algorithms and hardware design.

The section on number theory provides the mathematical foundation for residue number systems (RNSs). The section on computer arithmetic discusses specific building blocks we use to construct our implementation in Chapter 5. This section also provides background for understanding related work in Chapter 3. The last section on hardware discusses two implementation fabrics: field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). We discuss advantages and disadvantages of each technology to assist the reader in understanding our decision to use FPGAs.

2.1 Mathematical Background

This section covers the necessary definitions and theorems for understanding the techniques presented in Section 2.2 and Chapters 3 and 4.

Our exposition follows the standard flow of an introductory algebra textbook. Theorems in this section are presented without proof unless the proof provides additional insight for discussions in later chapters. These definitions and theorems may be found in most textbooks on introductory modern algebra or abstract algebra such as Beachy [9], Dummit [14], or Nicholson [26]. Readers with an understanding of residue number theory or modular arithmetic may skip this chapter.

2.1.1 Divisibility

Within the details of every concept we present in this chapter, divisibility underlies each of them. The Division Algorithm, proved using divisibility, states when two integers are not divisible there is a non-zero remainder leftover. This leads to congruence relations and the construction of the integers modulo n. The greatest common divisor, defined in terms of divisibility, determines when a modular integer has an inverse. This builds up to the fundamental theorem of residue number systems: the Chinese Remainder Theorem.

For any integers a and b, we use the notation $a|b$ if there exists an integer c such that $ac = b$. We describe this relation as a *divides* b, a is a *divisor* of b, or a is a *factor* of b depending on which is most convenient. We may alternatively say b is a *multiple* of a.

For any integers a, b, and c, we say c is a *common divisor* of a and b if $c|a$ and $c|b$. If c is the largest positive integer satisfying this relation then c is the *greatest common divisor (GCD)*, denoted $\gcd(a, b)$.

When a and b share no common factor (i.e., $\gcd(a, b) = 1$), we say a and b are *relatively prime* or *coprime*. A set of numbers $\{a_1, a_2, \ldots, a_n\}$ are *pairwise coprime* if a_i and a_j are coprime for all $i \neq j$ such that $1 \leq i, j \leq n$.

The divisibility relation obeys the following properties:

1. $a|a$;

2. if $a|b$ then $a|bc$;

3. if $a|b$ and $b|c$ then $a|c$; and

4. if $a|b$ and $a|c$ then $a|(bx + cy)$ where x, y are integers.

As a special case of the last property, the GCD of two integers is the smallest positive linear combination of those integers. Another interpretation is the GCD is the smallest increment between two linear combinations of the two integers. This interpretation will be important when we introduce the modular multiplicative inverse.

The Division Algorithm of modern algebra formalizes our understanding of division and assures us that the division we are familiar with works correctly.

Theorem 1 (Division Algorithm)**.** *Let a and b be integers with $0 < b \leq a$. Then there exists unique integers q and r such that $a = qb + r$ and $0 \leq r < b$. We call q the quotient and r the remainder.*

Immediately following from this theorem is if $a \geq b$ then they share a remainder modulo q if and only if $q|\,(a - b)$.

A direct consequence of the uniqueness property allows us to define two functions: division and modular reduction (or modulo). The division function (denoted a/b or $a \operatorname{div} b$) outputs the quotient q. The modulo function (denoted $a \bmod b$) outputs the remainder r. Any theorem or algorithm that requires modular reduction uses the Division Algorithm.

An example of the Division Algorithm is the Euclidean Algorithm. This algorithm (see Algorithm 1) computes the GCD of two integers by using the fact that $\gcd(a, b) = \gcd(a, b - a)$, which leads to the identity $\gcd(a, b) = \gcd(b, a \bmod b)$. The efficiency of the Euclidean Algorithm follows from Lamé's Theorem [12], which states

Algorithm 1: Euclidean Algorithm

Input : Two integers a and b with $a \geq b$
Output : The GCD of a and b

```
1 begin
2     if b = 0 then
3         return a;
4     return Euclid(b, a mod b);
```

if $b \leq a$ and $b < F_{k+1}$ where F_{k+1} is the $(k+1)$-st Fibonacci number then the Euclidean Algorithm requires no more than k recursive calls. An extended version of the Euclidean Algorithm later enables us to compute modular inverses.

2.1.2 Ring of Integers Modulo n

The Division Algorithms helps us define congruence relations with respect to a positive integer n called a modulus.

Definition 1. Let a, b, and n be integers and $0 < n$. We say a and b are congruent modulo n if $n|\,(a - b)$. We denote this $a \equiv b \bmod n$.

As noted in the previous section, a and b share a common remainder when they are congruent modulo n. The congruence relation modulo n has the following properties:

1. $a \equiv a \bmod n$;
2. if $a \equiv b \bmod n$ then $b \equiv a \bmod n$; and
3. if $a \equiv b \bmod n$ and $b \equiv c \bmod n$ then $a \equiv c \bmod n$.

In the parlance of relations, this means the congruence relation modulo n is an equivalence relation. Speaking informally, only the remainders after division by n are relevant under the congruence relation modulo n. This leads us to the ring of integers modulo n.

Definition 2. The integers modulo n, denoted $\mathbf{Z}_n$, is defined to be the set of integers $\{0, 1, \ldots, n-1\}$.

Informally a ring is a set with the addition, subtraction, and multiplication operations defined. $\mathbf{Z}_n$, as a subset of the integers, is a ring with the same algebraic operations as the integers up to congruence:

- Addition: $(a + b) \bmod n$;
- Subtraction: $(a - b) \bmod n$; and
- Multiplication: $ab \bmod n$.

These are the basic modular arithmetic operations. We can define other operations in terms of these (e.g., modular exponentiation).

The additive inverse is a number b such that $a + b \bmod n \equiv 0$. This number always exists by letting $b = n - a$ and denote this as $-a$. Subtraction as we defined above is well-defined.

The multiplicative inverse (or modular inverse) is a number x such that $ax \bmod n \equiv 1$. Applying the Division Algorithm, this is equivalent to solving for x and y such that $ax + ny = 1$. Recall the GCD is the smallest increment between two successive linear combinations of two integers. So this equation is solvable if and only if a and n are coprime, which we formalize in the following lemma:

Lemma 1. *When a and n are coprime, the congruence $ax \equiv 1 \bmod n$ has a unique solution x such that $0 \leq x < n$. We denote x as a^{-1}.*

The Extended Euclidean Algorithm (Algorithm 2) provides a method to compute the modular inverse of an element of $\mathbf{Z}_n$ (see Algorithm 3). We use the modular inverse to compute the constructive form of the Chinese Remainder Theorem (Equation 2.1).

Algorithm 2: Extended Euclidean Algorithm

Input : Integers a and b
Output : Integers x and y satisfying $ax + ny = \gcd(a, n)$

```
1 begin
2     (x, x_last) ← (0, 1);
3     (y, y_last) ← (1, 0);
4     while b ≠ 0 do
5         q ← a div b;
6         (a, b) ← (b, a mod b);
7         (x, x_last) ← (x_last − qx, x);
8         (y, y_last) ← (y_last − qy, y);
9     return (x_last, y_last);
```

Algorithm 3: $\mathbf{Z}_n$ Modular Inverse

Input : Integers a and n
Output : The modular inverse of a in $\mathbf{Z}_n$

```
1 begin
2     (x, y) ← ExtendedEuclid(a, n);
3     return x;
```

The Euclidean Algorithm computes the GCD of two integers, but discards the coefficients during computation. The Extended Euclidean Algorithm retains the coefficients x and y while computing the linear combination $ax + ny = \gcd(a, n)$.

2.1.3 Chinese Remainder Theorem

We are now ready to state the fundamental theorem behind RNSs: the Chinese Remainder Theorem. We present the theorem in a few forms. The first form (Theorem 2) states when a set of simultaneous linear modular congruences has a solution and provides conditions for uniqueness. We omit a formal proof of the theorem, but mention the proof in Beachy [9] invokes Lemma 1.

Theorem 2 (Chinese Remainder Theorem (CRT)). *Let* $\mathbf{M} = \{m_1, m_2, \ldots, m_K\}$ *be a set of pairwise coprime positive integers called moduli. Then for any integers* $z_1, z_2, \ldots, z_K$*, there exists an integer* Z *that satisfies the system of simultaneous linear congruences*

$$\begin{aligned} Z &\equiv z_1 \bmod m_1 \\ Z &\equiv z_2 \bmod m_2 \\ &\vdots \\ Z &\equiv z_K \bmod m_K \end{aligned}$$

where $z_1, z_2, \ldots, z_K$ *are the residues of* Z *over the moduli* $\mathbf{M}$*. Furthermore, all solutions are congruent to* Z *modulo* M *where* $M = m_1 m_2 \cdots m_K$ *and the solution in the range* $[0, M-1]$ *is unique.*

This form is the existence and uniqueness theorem most often found in introductory algebra texts, but it does not provide a method for constructing a solution. This leads to the second form (Equation 2.1). This formula, provided by Gauss, serves as a constructive proof of the solution Z referenced in the first form.

$$Z = \sum_{i=1}^{K} z_i M_i \left(M_i^{-1} \bmod m_i \right) \tag{2.1}$$

where $M_i = M/m_i$. The M_i^{-1} merits some justification. The definition of M_i ensures M_i and m_i are coprime and Lemma 1 ensures $M_i^{-1} \bmod m_i$ exists.

In general the solution to Equation 2.1 lies outside $\mathbf{Z}_M$. This leads to a third form (Equation 2.2) that computes the unique solution in the range $[0, M-1]$.

$$Z = \left(\sum_{i=1}^{K} M_i \left(z_i M_i^{-1} \bmod m_i \right) \right) \bmod M \tag{2.2}$$

We call this the fundamental theorem of RNSs because the Chinese Remainder Theorem further implies the ring $\mathbf{Z}_M$ is algebraically isomorphic to the product ring

$$\prod_{i=1}^{K} \mathbf{Z}_{m_i} = \mathbf{Z}_{m_1} \times \cdots \times \mathbf{Z}_{m_K}. \tag{2.3}$$

This means addition, subtraction, and multiplication performed in $\mathbf{Z}_M$ is equivalent to performing the same operations in each $\mathbf{Z}_{m_i}$ (called a residue channel) composing the product ring. These operations are parallelizable.

When M is the product of many small primes, the cost of performing ring operations in the product ring grow proportional to the largest factor of M instead of proportional to M itself. The distribution of prime numbers becomes an interest because that distribution determines the size of each residue channel. The Prime Number Theorem and its corollaries provide a bound on the distribution of prime numbers. Later analysis requires the theorem in the following form, which states that prime numbers are distributed logarithmically.

Theorem 3 (Prime Number Theorem)**.** *Let $\pi(x)$ denote the number of prime numbers less than or equal to x. For $x \geq 17$,*

$$\frac{x}{\ln(x)} < \pi(x) < 1.26\frac{x}{\ln(x)}. \tag{2.4}$$

2.2 Computer Arithmetic Background

Computer arithmetic studies number systems – the way digital systems represent numerical values – and the algorithms to manipulate them. Since these algorithms ultimately end up in hardware, there is an engineering aspect as well as a theoretical aspect to computer arithmetic.

We view the theoretical research process in terms of three phases. In the first phase, we define the number system and study the underlying mathematics. In the second phase, we define algorithms and architectures to perform arithmetic operations in the number system. In the final phase, we study the dependencies of resource utilization and execution time as functions of the input sizes.

We view the engineering aspect as determining whether a number system and algorithm is desirable for a particular application (i.e., special-purpose computing versus general-purpose computing). Desirability depends on many factors: manufacturing cost, performance, resource utilization, reliability, and power consumption. This list is not exhaustive.

We provide an overview of the binary and residue number systems before describing fundamental computer arithmetic algorithms and architectures.

2.2.1 Number Systems

A *number system* is a set of digit values together with a rule for interpreting a sequence of digits as a numerical value. Several number systems have been developed to include conventional number systems (e.g., binary), signed-digit number systems, and residue number systems. In this thesis, we address the binary number systems and residue number systems. Readers interested in a broad treatment of number systems and algorithms for manipulating them should refer to Koren [21].

It is common to classify number systems according to the following attributes: redundancy, weightedness, positional, and radix type.

Definition 3. A number system is *redundant* if there is a numerical value with multiple representations within that system. If every numerical value has a unique representation however, we say the system is *non-redundant*.

Definition 4. A number system is *weighted* if there is a sequence of weights $(w_{n-1}, w_{n-2}, \ldots, w_0)$ that relates a numerical value x to its sequence of digits $(x_{n-1}, x_{n-2}, \ldots, x_0)$ by the interpretation rule

$$x = \sum_{i=0}^{n-1} x_i w_i. \tag{2.5}$$

Otherwise the number system is unweighted.

Definition 5. A weighted number system is *positional* if the weight of a digit depends only on its position in the sequence.

Definition 6. A number system is *conventional* if it is non-redundant and positional.

Definition 7. A conventional number system is *fixed-radix* with radix r if $w_i = r^i$. Otherwise the system is *mixed-radix*. The interpretation rule for a fixed-radix system with radix r is

$$x = \sum_{i=0}^{n-1} x_i r^i. \tag{2.6}$$

Non-redundant positional number systems behave similarly to the commonly used decimal number system. We elaborate more on the first three properties within the context of each respective number system.

Binary Number Systems

Binary number systems are conventional fixed-radix number systems with radix 2. They are the standard number systems for general-purpose computing due to their many advantages. The arithmetic algorithms are mature, fast enough for most applications, and easy to implement. The numerical range of these systems is simple to extend. The two binary number systems we consider are the unsigned system and the two's complement signed system.

Definition 8. The *n-bit unsigned binary system* is a non-redundant fixed-radix system with radix 2 consisting of the set of n-tuples $(x_{n-1}, x_{n-2}, \ldots, x_0)$ with $x_i \in \{0, 1\}$ for all i such that $0 \leq i \leq n-1$. The interpretation rule follows Equation 2.6 with $r = 2$:

$$x = \sum_{i=0}^{n-1} x_i 2^i. \tag{2.7}$$

Definition 9. The *n-bit signed binary system*, or *n-bit two's complement binary system*, is identical to the unsigned binary system except $w_{n-1} = -2^{n-1}$.

The number of digits required by a binary number system depends on the range of values the system needs to cover. Often the number system can represent values beyond what a system requires. The range of the unsigned binary system is $[0, 2^n - 1]$ and the range of the signed binary system is $[-2^{n-1}, 2^{n-1} - 1]$. Between the unsigned and signed systems, only the most significant bit changes in weight. Therefore numbers in the range $[0, 2^{n-1} - 1]$ have the same representation when interpreted as unsigned or signed. For this same reason, we consider the two's complement signed system to be fixed-radix despite the negative weight of the most significant bit.

As a consequence of the positional property, the operations of comparison and sign detection are particularly efficient in binary.

Residue Number Systems

In contrast to binary number systems, residue number systems (RNSs) tend to find use in special-purpose computing such as digital signal processing (DSP). Many RNS algorithms use binary number system algorithms as building blocks. Despite this composition, operations such as addition, subtraction, and multiplication are more efficient in an RNS than in a binary system encompassing the same numerical range. However, RNSs have limitations

that prevent them from finding use in general-purpose computing. For example, fractions are not easily representable and efficient algorithms are not yet known for the other operations.

Definition 10. A *residue number system* (RNS) consists of a set of pairwise coprime moduli $\mathbf{M} = \{m_1, m_2, \ldots, m_K\}$ called the *RNS base* and the set of K-tuples $(z_1, z_2, \ldots, z_K)$ where $0 \leq z_i < m_i$ for each modulus. The interpretation rule

$$\begin{aligned} z_1 &= z \bmod m_1 \\ z_2 &= z \bmod m_2 \\ &\vdots \\ z_K &= z \bmod m_K \end{aligned} \tag{2.8}$$

relates z to its K-tuple: the residues of z in each channel of the RNS base.

Since the RNS base is pairwise coprime, the Chinese Remainder Theorem relates a K-tuple with its numerical value. The Chinese Remainder Theorem also implies the RNS is algebraically isomorphic to $\mathbf{Z}_M$ with the range $[0, M-1]$ where

$$M = \prod_{i=1}^{K} m_i. \tag{2.9}$$

As with the binary system, M depends on the range of values an RNS must cover and there are many RNS bases that are sufficient. Finding an optimal RNS base for a given range and hardware constraints is not trivial.

RNSs are unconventional number systems so their properties differ significantly from those of the conventional number systems described previously.

Weightedness RNSs are weighted, but not positional. However, a common convention in the literature is to order the residues in increasing moduli order with non-redundant residues preceding redundant residues. This is the convention we use in this thesis.

In RNSs, addition, subtraction, and multiplication are more efficient than in a binary system. The extent to which these operations are more efficient depends on the largest modulus in the base of the RNS, so many small moduli are desirable compared to few large moduli. The Prime Number Theorem establishes a limit on the performance gain since the distribution of primes grows logarithmically.

The remaining operations, which include sign detection, comparison, and scaling (division by a constant), are hard in RNSs compared to binary systems because RNSs are non-positional systems. Modular reduction by a constant uses the scaling operation because generating a quotient generates a remainder by the Division Algorithm.

Most digital systems use the binary system, so it is often necessary to convert forward from binary to an RNS to leverage the parallelism and then convert reverse from the RNS back to binary. Forward conversion uses residue channel arithmetic, which is efficient. Reverse conversion, however, requires high-radix arithmetic, which is slow.

Each RNS admits an associated mixed-radix system (AMRS), which is a positional weighted analog for the RNS. We assume, for simplicity, the moduli are in increasing order. Given the RNS base $\{m_1, m_2, \ldots, m_K\}$, define the AMRS weights as $\{a_1, a_2, \ldots, a_{K-1}\}$

where a_i is the product of the first i moduli:

$$\begin{aligned} a_1 &= m_1 \\ a_2 &= m_1 m_2 \\ &\vdots \\ a_i &= \prod_{j=1}^{i} m_j \\ &\vdots \\ a_{K-1} &= \prod_{j=1}^{K-1} m_j. \end{aligned} \tag{2.10}$$

Since AMRSs are positional, the advantage of parallel operations is lost in exchange for efficient sign detection and comparison operations.

2.2.2 Algorithms and Implementations

Up to this point we have discussed several number systems and operations without regard to specific algorithms or implementations. In this section, we discuss efficient algorithms and implementations for all operations we require, and quantify their performance.

We start with the generic building blocks of digital arithmetic and compose them to form modular arithmetic blocks. These initial designs use the binary number system and are our primitives. Number systems use different configurations of these primitives to achieve better running time than simply implementing larger-sized primitives. These form the higher level algorithms and architectures we discuss: Montgomery multiplication, RNS base extension, and RNS forward conversion and reverse conversion.

The primitives we present were designed prior to the invention of field-programmable gate arrays (FPGAs), which is the implementation fabric we use in this research. Following the convention found in the literature, we present all complexity analyses in terms of mathematical operations and operand bit-lengths. This convention ensures algorithm running time and memory usage are agnostic to a specific implementation fabric.

Fundamental Arithmetic Algorithms

The building blocks of digital arithmetic include binary adders, binary multipliers, and binary division circuits. Modern digital arithmetic algorithms and architectures comprise a subset of these fundamental blocks. There are several algorithms from which to choose for implementing these building blocks. We refer the reader to Koren [21] for details regarding these blocks including architectures and performance metrics.

The parallel prefix adders include some of the fastest binary adders today. Brent-Kung [10] and Kogge-Stone [20] are examples of parallel prefix adders and their gate delay is near optimal for binary adders. Wallace trees [40] and Dadda multipliers are examples of fast binary multipliers. Multipliers theoretically can achieve similar execution time to that of adders, but Koren notes multiplication is slower in practice.

The algorithms we present use division to facilitate modular reduction. Given arbitrary precision, division is equivalent to multiplying by the divisor's reciprocal. When the dividends are unbounded or the divisor is variable, we must compute the reciprocal dynamically. When the dividends are bounded and the divisor is constant, we can pre-compute the reciprocal and division becomes as easy as multiplication.

Algorithm 4: Modulo-m Adder

Input : Three integers x, y, and m such that $x, y < m$
Output : The sum of x and y modulo m

```
1 begin
2     s_0 ← x + y;
3     s_1 ← s_0 − m;
4     if s_0 < m then
5         s ← s_0;
6     else
7         s ← s_1;
8     return s;
```

Modulo-m Adder

The modulo-m adder computes the value $(x + y) \bmod m$ for integers x, y, and m. If we assume $x, y < m$, then the sum may exceed m, but not $2m$. We can detect whether the sum exceeds m and decide whether a correction by m is necessary. Algorithm 4 implements this approach and allows us to treat m as a parameter.

This architecture requires two binary adders and a binary comparator. The total gate delay of this structure is twice the gate delay of the adders and the total area is the sum of the area of the three components. The special case when m is a power of 2, a single binary adder is sufficient.

Modulo-m Reduction

The modular reduction block computes the value $x \bmod m$ given integers x and m. In general, modular reduction and division are equally hard. When m is a constant, we can use pre-computation to achieve a time-memory optimization.

Algorithm 5: Modular Reduction by m

Input : An integer x of length n.
Output : $x \bmod m$

```
1 begin
2     for i ← 0 to n do
3         if x_i = 1 then
4             p ← PowerTable(i);
5         else
6             p ← 0;
7         s ← ModularAdder(p, s, m);
8     return s;
```

Using the properties of modular arithmetic, we can modify Equation 2.7:

$$x \bmod m = \left(\sum_{i=0}^{n-1} x_i \left(2^i \bmod m \right) \right) \bmod m. \tag{2.11}$$

When x has a known maximum length n, then modular reduction requires at most $n - 1$ modular additions and n table lookups. The resulting algorithm (Algorithm 5) is faster and simpler than multiplication and division. We can extend this table to apply to arbitrarily long x by observing the periodicity property of the set $\{2^i \bmod m | i \in \mathbf{Z}\}$ noted in Mohan [4], which follows from the finiteness of $\mathbf{Z}_m$.

The execution time depends on the organization and execution time of the modulo-m adders. The memory requirements depend on the size of the set $\{2^i \bmod m | i \in \mathbf{Z}\}$. In the special case when m is a power of 2, reduction is equivalent to truncation.

Modular Multiplier

Three methods for implementing modular multiplication are

- multiplication followed by reduction;

Algorithm 6: Interleaved Modular Multiplication

Input : Two integers A and B less than M
Output : $AB \bmod M$

```
1  begin
2      /* Let A = (a_{n-1}, ..., a_0) and B = (b_{n-1}, ..., b_0). Define the
          partial product P_i = A b_i.                                        */
3      P ← 0;
4      for i ← n − 1 to 0 do
5          P ← 2P + P_i;
6          if P > M then
7              P ← P − M;
8          if P > M then
9              P ← P − M;
10     return P;
```

- interleaved multiplication and reduction steps; and
- Montgomery's method.

The first method is implementable using a binary multiplier and a modulo-m reduction unit. Since this architecture is sequential, the execution time, memory requirements, and number of gates is equal to the sum of that required for the multiplier and the reduction unit.

The second method interleaves the reduction steps into the multiplication. This is possible because efficient architectures compute multiplication as a summation of partial products. Algorithm 6 is an example algorithm demonstrating the technique. Correctness of the algorithm follows from the loop invariant $P < m$. Since $A, B < m$ and b_i is either 0 or 1, it follows that $P \leftarrow 2P + Ab_i < 3m$. At most two corrections (i.e., reductions) ensure $P < m$. By interleaving reduction into the multiplication, reduction is computable using a constant number of subtractions in lieu of a division operation. This results in better running time for small moduli.

Algorithm 7: Montgomery Multiplication

Input : Two integers x and y less than N
Output : $xy \bmod N$

```
1 begin
2     s ← xy;
3     t ← s(−N^{−1} mod R);
4     u ← tN;
5     v ← s + u;
6     w ← v/R;
7     return w;
```

These first two methods are slow because the cost of high-radix reduction is high. Other techniques, called high-radix methods, were developed to handle large moduli. One of those techniques is Montgomery's method [25].

Montgomery Multiplication

In computing the modular product, Montgomery's method replaces the complex reduction modulo-N operation with an easier reduction by another modulus (e.g., a power of 2) called Montgomery reduction.

Definition 11. Suppose N and R are two coprime integers such that $N < R$. Let T be an integer such that $0 \leq T < NR$. The *Montgomery reduction* of s modulo N is defined as $TR^{-1} \bmod N$.

The Montgomery product (MM) of two integers $x, y < 2N$ is an ordinary product followed by a Montgomery reduction: $w = xyR^{-1} \bmod N$. For convenience, we replicate as Algorithm 7 the version from Kawamura's Cox-Rower paper [19].

Since $x, y < 2N$ the result satisfies the inequality $w < 2N$.

Given an integer x, define $\tilde{x} = xR \bmod N$. When the multiplicand and multiplier are in this special form, the Montgomery product yields a result of the same form:

$$\begin{aligned} \mathrm{MM}\left(\tilde{x}, \tilde{y}\right) &= \tilde{x}\tilde{y}R^{-1} \bmod N \\ &= (xy)R \bmod N = wR \bmod N = \tilde{w}. \end{aligned} \tag{2.12}$$

Using this property it is possible to chain, without intermediary correction steps, Montgomery multiplication operations such as in modular exponentiation. The only additional steps required are conversion to the special form before the first Montgomery multiplication and conversion from the special form after the last Montgomery multiplication.

The actual Montgomery multiplication operation is faster than the modular multiplication techniques described previously. Overhead from the conversion steps, however nullify the performance gain. So for a single modular multiplication the Montgomery method is slower than classical techniques, but for repeated modular multiplications the Montgomery method is almost twice as efficient due to the easier reduction [23].

As an aside, using the binary system it is possible to interleave Montgomery's method into the multiplication similarly to how modular reduction can be interleaved into classical modular multiplication. The technique has been generalized to multiprecision arithmetic to make high-radix Montgomery multiplication. A comprehensive treatment is available in [35]. We are not aware of similar techniques for RNSs. This may be because RNS Montgomery multiplication (see Chapter 3) is fast enough not to require such methods.

Modular Exponentiation

Modular exponentiation computes the value $x^y \bmod n$, which is a series of modular products. Modular exponentiation units use the modular multiplier in addition to other logic.

Algorithm 8: Binary Modular Exponentiation

Input : Two integers x and $y = \left(y_{m-1}, \ldots, y_0\right)$ less than n such that $y_{m-1} = 1$
Output : $x^y \bmod n$

```
1 begin
2     z ← x mod n;
3     for i ← m − 2 to 0 do
4         z ← z^2 mod n;
5         if y_i = 1 then
6             z ← xz mod n;
7     return z;
```

Assuming an optimized modular multiplier, different techniques for exponentiation have been developed. The two fastest techniques are the binary exponentiation method (square-and-multiply method) and the sliding window method.

Binary exponentiation uses the binary representation of the exponent y, observing that $y = (1, y_{k-2}, \ldots, y_0)$ and

$$\begin{aligned} x^y &= x^{2^{k-1} + \sum_{i=0}^{k-2} 2^i y_i} \\ &= x^2 \cdot \prod_{i=0}^{k-2} x^{2^i y_i}. \end{aligned} \tag{2.13}$$

Using this form, exponentiation requires at most $2 \lg y - 2$ modular multiplication operations. The algorithm is detailed in Algorithm 8.

The sliding window technique is a generalization of the binary method, which assumes a window size of 1. Though the technique is well-understood, we note the literature on RNS modular exponentiation using the sliding window technique is sparse. For details, we refer the reader to [35].

Conversion Algorithms

Conversion algorithms enable systems using different number systems to interface with each other. The important conversion algorithms convert between binary and a chosen RNS. For now assume we have an RNS with base $\mathbf{M} = \{m_1, m_2, \ldots, m_K\}$ and $M = m_1 m_2 \cdots m_K$.

Forward conversion is the process of converting a number x from its binary representation to its residue representation. We can do this by simply reducing x by each modulus in the RNS base. Most forward conversion algorithms operate in this manner. We defer the algorithm details until we introduce the reduced-precision residue number system (RP-RNS) [33] (treated fully in Chapter 4) because our implementation of RP-RNS includes a forward conversion algorithm.

Reverse conversion is the process of converting a number from its RNS representation to its binary representation. Before treating reverse conversion further, we discuss two types of reconstruction algorithms: partial reconstruction as shown in [30, 31] and full reconstruction. To define both types of reconstruction, we will consider a number Z in RNS with representation $(z_1, z_2, \ldots, z_K)$. In Section 2.1.3 we stated two forms of the Chinese Remainder Theorem. The first form (Equation 2.2) determines Z from its residues in $\mathbf{M}$, but Z may be larger than the RNS range M; we will refer to this value as Z_T, following the notation in [30, 31]. The second form (Equation 2.1) determines the smallest Z, which is both smaller than M and equals $Z_T \bmod M$. Using the Division Algorithm we can write Equation 2.14 relating Z_T and Z.

$$Z_T = Rc_Z M + Z \tag{2.14}$$

The value Rc_Z is the *reconstruction coefficient* for Z [30, 31]. *Full reconstruction* is the determination of Z_T in a conventional number system. *Partial reconstruction* is the determination of the reconstruction coefficient without a full reconstruction.

Reverse conversion requires full reconstruction to determine Z whereas other algorithms (e.g., base extension) may only require partial reconstruction. The Chinese Remainder Theorem is the basis for all modern reconstruction algorithms.

Gauss's algorithm is an example of a full reconstruction algorithm. Given M as the product of K moduli, reconstruction using Gauss's algorithm requires $K \lg M$-bit modular multiplication and modular addition operations. Kawamura's Cox-Rower architecture provides a full reconstruction algorithm using their base extension algorithm. This method requires $2K^2 + K$ modular multiplication and if correction is necessary, K^2 subtractions.

RP-RNS provides a partial reconstruction algorithm. Reverse conversion uses the reconstruction coefficient in combination with high-radix binary arithmetic. This approach requires $K \lg M$-bit multiplications and additions. Using this method, we avoid high-radix modular arithmetic, which may be slower. We provide a full treatment of RP-RNS in Chapter 4 and defer treating forward and reverse conversion until then.

Parallel Operations

Addition and multiplication in RNS are parallel operations because of the isomorphism between $\mathbf{Z}_M$ and the product ring formed by the moduli in the RNS base $\mathbf{M}$. Each channel, having a different modulus, requires its own modular adder and a modular multiplier for addition and multiplication. The speed of addition and multiplication equals the speed of the modular adder and multiplier of the largest channel. Area and memory requirements, respectively, equal the sum of the area and memory requirements needed for the operations across all channels.

2.3 Implementation Fabrics

The application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) are two of several fabrics on which we can implement hardware designs. Many of the algorithms and architectures we discussed were designed before the invention of FPGAs. These algorithms work equally well on both technologies. As different technologies however, they have different properties and use cases, so it makes sense to compare and the contrast the two to understand when to use one over the other. The treatment we provide is a condensed comparison. We refer the reader to Maxfield [22] for a comprehensive treatment.

ASICs are integrated circuits customized for a specific use. The engineer designs the ASIC and designs are fixed in hardware at fabrication time. This means any flaws discovered post-fabrication requires a new production run to fix. In combination with high pre-fabrication non-recurring expenses, ASICs can be expensive, which makes them a poor choice for prototyping and rapid development. However, they are excellent candidates for high-volume production of static designs as the cost per board is cheap and decreases as production increases. This allows manufacturers to amortize the non-recurring costs of production across each board. Despite the significant drawback of cost, there are many benefits to using ASICs. Since the device is manufactured to the design specification, ASICs benefit from high performance, efficient resource utilization, and smaller form factor technology. The benefit of smaller form factor technology is increased logic density and subsequently increasingly complex circuits.

FPGAs are a special type of ASIC. They are reconfigurable general-purpose integrated circuits consisting of a hierarchy of logic cells, logic slices, and configurable logic blocks (CLBs) (see Figure 2.1). At the bottom of the hierarchy are the logic cells, which minimally comprise reconfigurable lookup tables (LUTs) and storage elements. These two elements

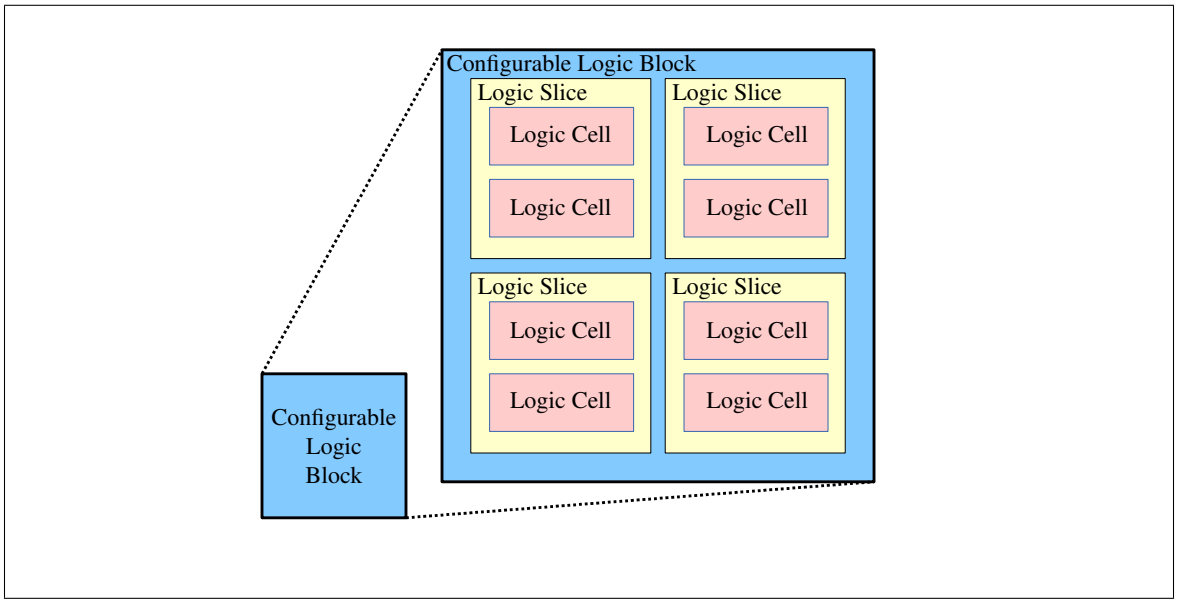

Figure 2.1: Structure of an FPGA configurable logic block.

make a logic cell a universal logic function. Logic cells may also contain multiplexers, arithmetic units, or shift registers. Logic cells are grouped into logic slices and further grouped into CLBs. Engineers configure FPGAs with logical circuits written in a hardware design language (HDL) such as VHDL or Verilog. The number of times an FPGA can be configured is virtually limitless. This means non-recurring expenses are negligible compared to ASICs and flaws are easily corrected by updating the HDL code. This makes FPGAs excellent candidates for prototyping, rapid development, production of dynamic designs, and low-volume production. The flexibility of FPGAs carries a performance cost. With logic cells as the basic unit over gates, implementing a simple function (e.g., an inverter) uses the same quantity of hardware as a more complex function (e.g., a full adder). This decreased logic density makes high-complexity circuits infeasible on FPGAs. Furthermore, since the FPGA is not optimized for the hardware design, FPGAs are less resource efficient and do not perform as well as ASICs.

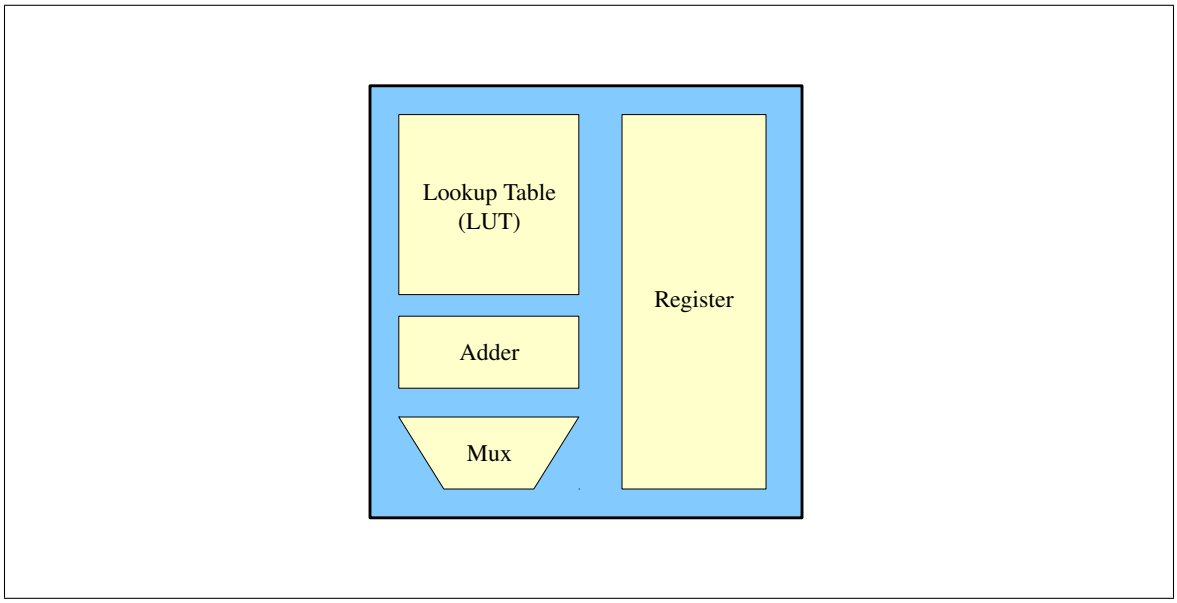

Figure 2.2: An example structure of an FPGA logic cell.

	ASICs	FPGAs
Logic count	$>$ 1.4 billion transistors	2 million cells [43]
Logic density	High	Low
Design complexity	High	Low
Design fixing	Manufacturing	Reconfigurable
Non-recurring engineering cost	High	Low
High volume production cost	Low	High
Common Use Case	Production	Prototyping

Table 2.1: Summary Comparison of ASICs and FPGAs

We summarize the contents of this section in Table 2.1, which we derived from Maxfield [22]. Neither technology is superior to the other and the best implementation medium depends on manufacturing and design requirements.

2.4 Key Points

Residue number systems allow for fast parallel implementation of arithmetic operations in lieu of slow high-radix binary algorithms. The Chinese Remainder Theorem is the core of residue number systems.

The RP-RNS algorithms we implement rely on the Chinese Remainder Theorem and Division Algorithm. The CRT provides the mapping between the range of the RNS and the residue channels. In Chapter 4 we describe how to implement the RP-RNS algorithms. The partial reconstruction algorithm uses a fractional form of the CRT. The quotient-first scaling algorithm uses a fractional form of the Division Algorithm, which leads to the RP-RNS modular exponentiation algorithm.

We follow Phatak's choice of RP-RNS base: the first K prime numbers. The Prime Number Theorem is a useful tool for analyzing the RP-RNS algorithms.

In the chapter that follows we discuss related work, all of which use Montgomery's method for performing modular multiplication. We will see, however, there is a fundamental limitation to this approach.

Chapter 3

Related Work

In this chapter, we discuss work related to RNS-based modular exponentiation techniques and implementations. Unless mentioned otherwise, all related work regarding implementations we discuss used custom application-specific integrated circuits. At the end of the chapter, we describe Gandino's implementations, which are the fastest implementations of which we are aware and what we use as our basis of comparison in Chapter 6.

Modern RNS-based modular exponentiation techniques use either Montgomery's method or the Division Algorithm directly. Most work in this area use a translation of Montgomery's method to RNS. The reduced-precision residue number system (RP-RNS) – the system on which our implementation is based – can use either method.

3.1 RNS-based Montgomery's Multiplication

In 1998, Bajard, et al. [5, 6] demonstrated a generic translation of Montgomery's algorithm to RNSs. This translation uses two RNS bases, the second of which is responsible for the reduction. At two points in their translation, the number becomes a multiple of one of the

Algorithm 9: RNS Montgomery Multiplication

1:	$[s]_A \leftarrow [x]_A [y]_A$	$[s]_B \leftarrow [x]_B [y]_B$
2a:	—	$[t]_B \leftarrow [s]_B [-N^{-1}]_B$
2b:	$[t]_{A \cup B} \Leftarrow [t]_B$	
3:	$[u]_A \leftarrow [t]_A [N]_A$	—
4:	$[v]_A \leftarrow [s]_A + [u]_A$	—
5a:	$[w]_A \leftarrow [v]_A [B^{-1}]_A$	—
5b:	$[w]_A \Rightarrow [w]_{A \cup B}$	

Note: Columns are for emphasis. The left column represents operations performed on residues in $\mathbf{A}$ and the right column represents operations performed on residues in $\mathbf{B}$. The arrows represent base extensions.

RNS bases and thus the residues vanish. This can be perceived as a form of lost information that requires recovery. The recovery method used is a base extension. The version of the translation we reference (Algorithm 9) comes from Kawamura [19].

Let $\mathbf{A}$ and $\mathbf{B}$ be RNS bases and let $N < A$ and B be coprime. The second step of Algorithm 7 requires a reduction modulo B. Since reduction is a hard operation in RNS, the reduction in the second step of Algorithm 7 takes place as an ordinary operation in base $\mathbf{B}$; the residues in base $\mathbf{A}$ are lost. Similar information loss occurs in the last step since B has no inverse modulo B. A method to recover the lost residues is necessary.

3.2 Base Extensions for RNS-based Montgomery

The base extension operation of a number z adds additional channels to an RNS base by generating z's residues for the added channels. There are two reasons this operation is desirable: a larger RNS base permits expression of a larger number range and it serves as a method to recover lost residues in Montgomery multiplication.

Definition 12 (Base Extension)**.** Let $\mathbf{M}$ be an RNS base with K moduli and $\mathbf{M}'$ be another RNS base with K' moduli such that $\mathbf{M} \subset \mathbf{M}'$ and $K < K'$. Let $z < M$ be an integer with residues $(z_1, z_2, \ldots, z_K)$ in $\mathbf{M}$. The *base extension* of z from $\mathbf{M}$ to $\mathbf{M}'$ is the set of residues of z in base $\mathbf{M}'$: $(z_1, z_2, \ldots, z_{K'})$.

Without loss of generality, we can assume the first K moduli in both bases correspond. The first K residues of z in $\mathbf{M}'$ will match the residues in $\mathbf{M}$. The *base extension operation* computes the remaining residues of z for the moduli $\{m_{K+1}, \ldots, m_{K'}\}$.

There is another equivalent formulation of base extension operation.

Theorem 4. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be RNS bases such that* $\mathbf{A} \cup \mathbf{B}$ *contains only pairwise coprime moduli. Then the mapping of* z *as residues of* $\mathbf{A}$ *to residues of* $\mathbf{A} \cup \mathbf{B}$ *is the base extension of* z *from* $\mathbf{A}$ *to* $\mathbf{A} \cup \mathbf{B}$*.*

Proof. Let $\mathbf{M} = \mathbf{A}$ and $\mathbf{M}' = \mathbf{A} \cup \mathbf{B}$. □

Each modern base extension algorithm requires a form of reconstruction. Since both partial and full reconstruction operations are sequential, base extension is the bottleneck operation for RNS Montgomery multiplication. Much effort has been expended to decrease this bottleneck.

The first base extension algorithms used Garner's algorithm [17] and Szabo and Tanaka's Mixed-Radix Conversion (MRC) algorithm [38]. Each perform a full reconstruction of z before computing the remaining residues in parallel. They are the slowest base extension algorithms because they use no additional information beyond the residues of z and the RNS base. The MRC algorithm requires $(K^2 + K)/2$ lookup tables with a total latency of $2K$ cycles where K is the number of moduli.

In 1988, Shenoy and Kumaresan [36,37] developed a fast base extension algorithm using integer domain representation and a redundant modulus. Unlike Garner's algorithm and the

MRC algorithm, Shenoy and Kumaresan's technique is a partial reconstruction algorithm. It requires $2(K+1)$ lookup tables with a total latency of $\lg(K+1)+2$ cycles. The total number of modular multiplications required is K^2+K or K^2+2K depending on whether an exact result is necessary. This is a large improvement over the MRC algorithm.

Bajard's [5, 6] translation of Montgomery's algorithm to RNSs is general in that the base extension steps do not depend on the implementation of the operation. They provided implementations using both the MRC algorithm and Shenoy and Kumaresan's algorithm. The advantage of using Shenoy and Kumaresan's base extension method is that the result need not be exact; only the second base extension needs to be exact. Since exactness is not necessary for the first base extension, the total number of modular multiplications is $2K^2+8K$.

In 2000, Kawamura, et al. [19] proposed a recursive base extension algorithm for their RNS Montgomery multiplication implementation: Cox-Rower architecture. Their base extension algorithm computes the reconstruction coefficient one bit per step. Since each base extension uses the same algorithm, the total number of modular multiplications required is $2K^2+9K$, which is slightly higher than Bajard's method.

In 2012, Gandino, et al. [16] took Bajard's and Kawamura's method and reorganized the steps in the RNS Montgomery multiplication algorithm to reduce the number of multiplications required by $3K$.

3.3 RNS-based Montgomery Implementations

In 2001, Nozaki, et al. [27] implemented an ASIC of the Cox-Rower architecture. Their implementation included eleven 32-bit Rower units. The main logic of their ASIC used 221,000 gates. For 1024-bit RSA they used 57 KB of read-only storage and 12 KB of

writable storage. They achieved a running time of 2.4 ms using an 80 MHz clock. They concluded a fully parallel implementation could achieve a processing time of less than 1 ms.

In addition to further optimizing Bajard's and Kawamura's algorithms and architectures, Gandino, et al. [16] compared six implementations of each: three-stage, four-stage, $\omega(c_i) \leq 3$ without base knowledge, $\omega(c_i) \leq 3$ with base knowledge, $\omega(c_i) \leq 2$ without base knowledge, and $\omega(c_i) \leq 2$ with base knowledge. They synthesized their architectures using Nangate 45 nm Open Cell Library and used the same parameters as Kawamura and Nozaki (i.e., eleven 32-bit Rower units). Their slowest implementation was plain Cox-Rower using the three-stage architecture. It executed an RNS Montgomery multiplication in 88 cycles with a cycle delay of 1.73, which approximates to 153 ns. Their fastest implementation was an optimized Cox-Rower using their $\omega(c_i) \leq 3$ architecture with base knowledge. It executed an RNS Montgomery multiplication in 78 cycles with a cycle delay of 1.12, which approximates to 88 ns.

The implementations we described are custom ASIC implementations. While we prefer to base our comparison on an FPGA implementation of the described algorithms and architectures, we are unaware of such implementations' existences. Since Gandino's implementation is the fastest ASIC implementation of which we are aware, we use this as the basis of our comparison.

Chapter 4

Reduced-Precision Residue Number System (RP-RNS)

In this chapter, we introduce Phatak's reduced-precision residue number system (RP-RNS) [33]. For brevity we choose initially to state without proof the features of RP-RNS to motivate its definition. Across the chapter, we weave the theorems that prove these features are desirable. We state the algorithms used to implement modular exponentiation in RP-RNS while maintaining compatibility with existing binary-based computing systems. We provide a correctness proof and a time and space analysis for each algorithm based on the assumptions of Phatak. We add the execution time can vary significantly depending on how much parallelism an implementation uses. We conclude the chapter with strategies for designing RP-RNS systems.

We explain the algorithms in order of dependence:

- binary-RNS forward conversion;
- partial reconstruction (PR);

- quotient-first scaling (QFS), which uses the output of the PR algorithm;
- modular exponentiation using QFS; and
- RNS-binary reverse conversion.

All algorithms in this chapter are previous work [30, 31]. The reverse conversion algorithm is a direct logical extension of Phatak's partial reconstruction algorithm.

4.1 RP-RNS Defined

Within the framework of traditional RNS operations, the RP-RNS algorithms are intermediate computations. The RP-RNS algorithms produce intermediate results, which need not be exact. Therefore they can make liberal use of approximation, trading exactness for speed.

Interpreting the Chinese Remainder Theorem (CRT) informally, in RNSs the residues of a number contain the minimum information required to perform arithmetic operations. Since approximation discards information, any approximations introduced into a computation can affect the precision of the final result. With this point in mind we can now define RP-RNS.

Definition 13. A *reduced-precision residue number system* consists of an RNS base $\mathbf{M}$ and a redundant modulus m_e not contained in $\mathbf{M}$ referred to as the extra residue channel.

The redundant modulus must satisfy the inequality $\gcd(M, m_e) < m_e$. When the RP-RNS base includes 2, letting $m_e = 4$ is sufficient. When the RP-RNS base consists of only odd primes, letting $m_e = 2$ is sufficient. This implies the redundant channel uses at most 2 bits [30, 31].

The RP-RNS algorithms use approximations based on the rational form of the CRT and Division Algorithm. The ability to estimate overflow of both the RNS range M and the

divisor leads to fast partial reconstruction and scaling algorithms. We note that despite its use of approximation, the partial reconstruction produces an exact result. The extra residue disambiguates the estimation since the estimated residues will be off exactly by M.

The values used in the rational form of the CRT and Division Algorithm require exact high-radix arithmetic to compute. To maintain the advantages of RNS, we must precompute and store the relevant values; this includes the fractions.

The precomputed values add a memory overhead. In Section 4.7, we shall see a good heuristic to minimize memory required by using many small moduli such as the first K prime numbers. We state the algorithmic analysis under the assumption the RP-RNS base consists of the first K prime numbers. From the Prime Number Theorem it follows [30, 31]

$$K \approx \mathrm{O}\left(\frac{N}{\ln N}\right) \approx \mathrm{O}\left(\frac{\lg M}{\ln \lg M}\right) \tag{4.1}$$

where N is the number of bits required to represent the range of the system.

The list below summarizes the design features that make RP-RNS efficient. None of these features alone are novel and have appeared in other RNS algorithms.

This list summarizes the design features that make RP-RNS efficient:

- the usage of approximation [19];
- the redundant modulus [36, 37];
- the usage of rational equations over integer equations [19];
- the precomputed lookup tables [5–8, 19, 36, 37]; and
- the choice of many small moduli.

What is novel is how RP-RNS combines these features and the efficiency gained. The RP-RNS approach is take rational equations of interest (e.g., CRT and Division Algorithm) and

approximate the constant portions of those equations. The precision of the approximation is chosen to bound the error such that the final result narrows down to two candidates. We shall see this as we encounter each algorithm in this chapter.

To aid in our understanding of RP-RNS, we shall carry along with us a small example system throughout this chapter. For our system, we shall use the RNS base $\mathbf{M} = \{2, 3, 5, 7, 11\}$ such that $M = 2310$ and an extra residue channel $m_e = 4$. We seek to compute the expression $x^y \bmod D$. For our example system we choose $D = 16$; equivalently we are performing modular exponentiation in $\mathbf{Z}_{16}$ using numbers represented as elements of $\mathbf{Z}_{2310}$. We make the reasonable assumption that $x, y < D$. For consistency we shall use $x = 5$ as our base and $y = 5$ as our exponent.

4.2 Forward Conversion Algorithm

Binary-based systems employing RNS-based components use conversion algorithms to maintain compatibility. RNS-based components employ a forward conversion algorithm to convert input from binary representation to RNS representation.

4.2.1 Algorithm Description

Using the properties of modular arithmetic, we can combine Equation 2.7 with Definition 10 to relate a number x's binary representation directly to its RNS representation:

$$\begin{aligned} x_j &= \sum_{i=0}^{n-1} x_i 2^i \bmod m_j \\ &= \sum_{i=0}^{n-1} x_i (2^i \bmod m_j) \bmod m_j \quad (1 \leq j \leq K) \end{aligned} \tag{4.2}$$

where x_j is the residue over the jth modulus.

Provided we have access to modular adders and the residues of the powers of 2 within each residue channel, Equation 4.2 gives us a method for computing x's RNS representation using x's binary representation (Algorithm 10).

Example 1. Let us convert $x = 5$ from its binary representation $(1, 0, 1)$ to its RP-RNS representation $(1, 2, 0, 5, 5; 1)$. We detail the $\bmod 11$ channel only. We provide the powers of 2 and their residues in Table 4.1.

$$x = (1, 0, 1) = \left(\left(2^2 \bmod 11\right) + \left(2^0 \bmod 11\right)\right) \bmod 11 = 5 \tag{4.3}$$

As our example system's theoretical range is $[0, 2310)$, we must be able to support 12-bit binary numbers.

Algorithm 10: Forward Conversion

Input : Integer z in binary representation with length n
Output : Integer z in RNS representation (including extra channel)

```
1 begin
2     foreach m_j ∈ M do
3         z̄_j ← 0;
4     for i ← n to 0 do
5         foreach m_j ∈ M do
6             z̄_j ← (z̄_j + PowerTable_j(z_i)) mod m_j;
7         z_e ← (z_e + PowerTable_e(z_i)) mod m_e;
8     return (z̄, z_e);
```

4.2.2 Time and Space Analysis

While slow, one can compute the residues of the powers of 2 using a division circuit. However, the most common forward conversion algorithm – detailed in Koren [21] – uses a

i	2^i	mod 2	mod 3	mod 5	mod 7	mod 11	mod 4
0	1	**1**	**1**	**1**	**1**	**1**	**1**
1	2	**0**	**2**	**2**	**2**	**2**	**2**
2	4	0	1	**4**	**4**	**4**	**0**
3	8	0	2	**3**	1	**8**	0
4	16	0	1	1	2	**5**	0
5	32	0	2	2	4	**10**	0
6	64	0	1	4	1	**9**	0
7	128	0	2	3	2	**7**	0
8	256	0	1	1	4	**3**	0
9	512	0	2	2	1	**6**	0
10	1024	0	1	4	2	1	0
11	2048	0	2	3	4	2	0

Table 4.1: Forward Conversion Algorithm ROMs (period in bold)

precomputed table to relate the powers of 2 to their residues. The largest channel, m_K, at $\approx \lg N$ bits is the slowest to convert where N is the number of bits required to represent a number in our RNS. Assuming we have access to an adder tree and parallel table lookup, then the running time is $\mathrm{O}\left(\lg N\right)$.

The lookup table requires up to K entries per residue channel depending on the size of the residue channel. Mohan [4] notes that we can reduce the size of the lookup tables by taking advantage of the periodicity properties of residue exponentiation. Therefore a lookup table requires no more entries than the sum of the moduli, which is $\mathrm{O}\left(K^2\right)$.

4.3 Partial Reconstruction Algorithm

In this section, we describe Phatak's partial reconstruction algorithm [30, 31]. In Section 2.2.2 we distinguished full reconstruction from partial reconstruction. Whereas full reconstruction is equivalent to reverse conversion, partial reconstruction only computes the

reconstruction coefficient Rc_Z in Equation 4.4.

$$Z = Z_T - Rc_Z M \tag{4.4}$$

where

$$Z_T = \left(\sum_{i=1}^{K} M_i \left(z_i M_i^{-1} \bmod m_i \right) \right). \tag{4.5}$$

Current algorithms for base extension incorporate partial reconstruction, however the two operations are not equivalent. It is therefore meaningless to compare the two. However, we can use a partial reconstruction algorithm to perform a base extension by reducing Equation 4.4 by some modulus in the extended base.

4.3.1 Pre-Computed Lookup Tables

The partial reconstruction algorithm lookup tables store truncated fractions for each residue channel. For convenience, following [30, 31], we introduce the following notation:

$$\rho_r = \left(z_r M_r^{-1} \bmod m_r \right) \tag{4.6}$$

and

$$f_r = \rho_r / m_r. \tag{4.7}$$

The set $\{f_r\}$ contains all possible fractions for a given channel. The f_r values are full-precision fractions. Since full-precision division is as expensive as computing the CRT, we approximate using reduced-precision fractions defined as

$$\hat{f}_r = \mathrm{trunc}\left(f_r, w_F \right) \tag{4.8}$$

ρ	mod 2	mod 3	mod 5	mod 7	mod 11
0	0	0	0	0	0
1	32	21	12	9	5
2	–	42	25	18	11
3	–	–	38	27	17
4	–	–	51	36	23
5	–	–	–	45	29
6	–	–	–	54	34
7	–	–	–	–	40
8	–	–	–	–	46
9	–	–	–	–	52
10	–	–	–	–	58

Table 4.2: Partial Reconstruction Algorithm ROMs (Scaled by $2^6 = 64$)

where $w_F \geq \lceil \lg K \rceil$ is the precision of $\hat{f}_r$. Since the fractions are fixed-precision, we may represent them as integers after scaling them by w_F or equivalently ignoring the decimal point.

Table 4.2 is the lookup table associated with our example. The values in Table 4.2 use the scaling technique we described.

4.3.2 Algorithm Description

Algorithm 11 [30, 31] is the partial reconstruction algorithm. It takes the RP-RNS representation of a number z and computes the reconstruction coefficient Rc_z. The algorithm optionally returns the set of ρ values and the number of non-zero ρ values, which other RP-RNS algorithms may use.

The algorithm requires a few precomputed values. Within each channel, the lookup tables and $M_r^{-1} \bmod m_r$ are precomputed. Within the redundant residue channel, $M_r \bmod m_e$ and $M \bmod m_e$ are precomputed.

Each channel uses its residue of z, z_r to compute ρ_r using integer-domain arithmetic. Each ρ_r serves as an index into the fraction lookup table for the algorithm. The algorithm computes the sum of the truncated fractions retrieved from the lookup tables. The integer portion of the sum becomes a reconstruction coefficient candidate. The alternate candidate is the integer succeeding the primary candidate. These are the only candidates possible because the truncated fractions are approximations of the full-precision fractions. To disambiguate the candidates, we compute a sum and comparison within the redundant residue channel. If the primary candidate is correct, the comparison will be true. If the comparison is false, then the comparison will be off by exactly $M \bmod m_e$ and the alternate candidate instead is the correct reconstruction coefficient.

Correctness of the algorithms follows from the analytical derivations in [30, 31].

Example 2. Continuing our example, $x = 5$ has RP-RNS representation $(1, 2, 0, 5, 5; 1)$. The M_r values are $(1155, 770, 462, 330, 210)$ and the precomputed values follow: the M_r^{-1} values are $(1, 2, 3, 1, 1)$; the $[M_r]_{m_e}$ values are $(3, 2, 2, 2, 2)$; and $[M]_{m_e} = 2$.

We compute the ρ_r values $(1, 1, 0, 5, 5)$ (Line 3). We use the ρ_r values as lookup table indices into Table 4.2 to derive the scaled truncated fractions $(32, 21, 0, 45, 29)$ (Line 4). Summing the scaled fractions results in the value 127 (Line 5). Scaling by w_F (division by 64 in this example) yields the primary and alternate candidates: 1 and 2 (Lines 7 and 8).

In parallel with Line 5 we compute Line 6: $Z_T \bmod m_e$ as 1. Computing the right-hand side of the equality on Line 9 results in 3. Since the two values are not equal, then the alternate candidate, 2, is the reconstruction coefficient. The algorithm returns 2. As an aside, we note that the two computed values differ by exactly $[M]_{m_e}$.

Algorithm 11: RP-RNS Partial Reconstruction

Input : Integer z in RNS representation (including m_e)
Output : Reconstruction coefficient of z: Rc_z

begin

/* Computation of S and z_{TE} are computable in parallel. Other steps are dependent on their previous steps. */

$\rho_j \leftarrow (z_j M^{-1}{}_j) \bmod m_j \quad 1 \leq j \leq K$;

$f_j \leftarrow \texttt{FractionTable}_j\left(\rho_j\right)$;

$S \leftarrow \sum_{j=1}^{K} f_j$;

$z_{TE} \leftarrow \left(\sum_{j=1}^{K} M_j \rho_j \bmod m_e\right) \bmod m_e$;

$Rc_A \leftarrow \texttt{RightShift}\left(S, w_F\right)$;

$Rc_B \leftarrow Rc_A + 1$;

if $z_{TE} = (M Rc_A + z) \bmod m_e$ **then**

  $Rc_z \leftarrow Rc_A$;

else

  $Rc_z \leftarrow Rc_B$;

return Rc_z and optionally $(\rho_1, \rho_2, \ldots, \rho_K)$ and $|\{\rho_1, \rho_2, \ldots, \rho_K | \rho_i \neq 0\}|$;

4.3.3 Time and Space Analysis

The running time of the partial reconstruction algorithm depends on the hardware components: memory, hardware channels, and the fraction adder for the precomputed fractions. The fraction adder is the primary bottleneck. Assuming an adder tree for the K fractions then the delay is $\mathrm{O}\left(\lg N\right)$ where N is the number of bits required to represent a number in our RNS.

The partial reconstruction algorithm requires storage for the lookup tables and other precomputed values. The i-th residue channel has $m_i - 1$ entries. Summing over all residue channels yields an upper bound of $Km_K \approx \mathrm{O}\left(K^2\right) \approx \mathrm{O}\left(\left(N/\ln N\right)^2\right)$ entries. Each entry uses w_F, or $\mathrm{O}\left(\lg N\right)$, bits of storage. Therefore the total number of bits is $\mathrm{O}\left(N^2/\ln N\right)$.

4.4 Quotient-First Scaling Algorithm

Scaling is a special case of general division in which the divisor is a constant. The quotient-first scaling (QFS) algorithm [30, 31] uses approximation in computing the Division Algorithm to perform true division and modular reduction. Whereas the partial reconstruction algorithm always outputs an exact reconstruction coefficient, the QFS algorithm can only narrow the quotient down to two candidates without employing a correction step such as sign-detection. For some applications, such as modular exponentiation, the estimate is sufficient.

4.4.1 Pre-Computed Lookup Tables

The QFS algorithm has two types of lookup tables called QFS Table 1 and QFS Table 2. Table 1 uses the ρ values from the partial reconstruction algorithm as its indices. Table 2

uses the reconstruction coefficient as its index. Unlike the partial reconstruction algorithm, these tables store both integer parts as residues and fractional parts as truncated fractions. The truncated fractions for the QFS algorithm require slightly higher precision than those for the partial reconstruction algorithm: $w_F \geq \lceil \lg K + 1 \rceil$.

Tables 4.3 and 4.4 are the lookup tables associated with our example. The fractional values use the same scaling technique previously described.

4.4.2 Algorithm Description

Algorithm 12 [30, 31] is the QFS algorithm. It takes the RP-RNS representation of a number z and computes an estimate of $z \operatorname{div} D$ where the divisor D is a pre-determined constant. The algorithm returns a flag relating to the exactness of the quotient. If the flag is true, then the quotient is known to be exact. If the flag is false, then the algorithm could not determine the exactness of the quotient and further checks are needed.

Unlike the partial reconstruction algorithm, the QFS algorithm requires only the lookup tables.

Algorithm 13 is a modular reduction algorithm based on the Division Algorithm. It takes the RP-RNS representation of a number z and computes $z \bmod D$ assuming the quotient $z \operatorname{div} D$ is known. Three cases arise depending on the quality of the quotient.

1. If the quotient is at most an underestimate (e.g., QFS algorithm) then the result may exceed D, but will be in the proper residue class modulo D.

2. If the quotient is exact then the result will fall in the range $[0, D - 1]$.

3. If the quotient is an overestimate then the result will underflow the range of the system.

m_r	ρ_r	Quotient (Q_r)	$Q_r \bmod 4$	$Q_r \bmod m_j$	Remainder (R_r)
2	0	0	0	$(0,0,0,0,0)$	0
	1	72	0	$(0,0,2,2,6)$	12
3	0	0	0	$(0,0,0,0,0)$	0
	1	48	0	$(0,0,3,6,4)$	8
	2	96	0	$(0,0,1,5,8)$	16
5	0	0	0	$(0,0,0,0,0)$	0
	1	28	0	$(0,1,3,0,6)$	56
	2	57	1	$(1,0,2,1,2)$	48
	3	86	2	$(0,2,1,2,9)$	40
	4	115	3	$(1,1,0,3,5)$	32
7	0	0	0	$(0,0,0,0,0)$	0
	1	20	0	$(0,2,0,6,9)$	40
	2	41	1	$(1,2,1,6,8)$	16
	3	61	1	$(1,1,1,5,6)$	56
	4	82	2	$(0,1,2,5,5)$	32
	5	103	3	$(1,1,3,5,4)$	8
	6	123	3	$(1,0,3,4,2)$	48
11	0	0	0	$(0,0,0,0,0)$	0
	1	13	1	$(1,1,3,6,2)$	8
	2	26	2	$(0,2,1,5,4)$	16
	3	39	3	$(1,0,4,4,6)$	24
	4	52	0	$(0,1,2,3,8)$	32
	5	65	1	$(1,2,0,2,10)$	40
	6	78	2	$(0,0,3,1,1)$	48
	7	91	3	$(1,1,1,0,3)$	56
	8	105	1	$(1,0,0,0,6)$	0
	9	118	2	$(0,1,3,6,8)$	8
	10	131	3	$(1,2,1,5,10)$	16
$0 \le \rho_r < m_r, Q_r = \left\lfloor \frac{M_r \rho_r}{D} \right\rfloor, R_r = \left\lfloor \frac{M_r \rho_r - Q_r D}{D} 2^{w_F} \right\rfloor, w_F = 6$					

Table 4.3: QFS Algorithm Table-1 for our example.

Algorithm 12: RP-RNS Quotient-First Scaling Estimation

Input : Integer z in RP-RNS representation

Output : $z \operatorname{div} D$ in RP-RNS representation and a confirmation flag for exactness

begin

/* Partial reconstruction algorithm dependence. */

$(\rho_1, \rho_2, \ldots, \rho_K, Rc_z) \leftarrow \texttt{PartialReconstruction}\left(\bar{z}, [z]_{m_e}\right)$;

$n \leftarrow |\{x | x \in \{\rho_1, \rho_2, \ldots, \rho_K, Rc_z\} \wedge x \neq 0\}|$;

/* Compute the integer part of the quotient. */

$\bar{Q}_i \leftarrow \texttt{QuotientTable1}\,(i, \rho_i, 5)$;

$\bar{Q}_{Rc_z} \leftarrow \texttt{QuotientTable2}\,(Rc_z, 4)$;

$\bar{Q} \leftarrow \sum_{i=1}^{K} \bar{Q}_i - \bar{Q}_{Rc_z}$;

$[Q]_{m_e i} \leftarrow \texttt{QuotientTable1}\,(i, \rho_i, 4)$;

$[Q_{Rc_z}]_{m_e} \leftarrow \texttt{QuotientTable2}\,(Rc_z, 3)$;

$[Q]_{m_e} \leftarrow \sum_{i=1}^{K} [Q]_{m_e i} - [Q_{Rc_z}]_{m_e}$;

/* Compute the fractional part of the quotient. */

$R_i \leftarrow \texttt{QuotientTable1}\,(i, \rho_i, 6)$;

$R_{Rc_z} \leftarrow \texttt{QuotientTable2}\,(Rc_z, 5)$;

$Q_f \leftarrow \sum_{i=1}^{K} R_i - R_{Rc_z}$;

/* Attempt to determine the exactness of the quotient. */

$Q_L \leftarrow \texttt{RightShift}\,(Q_f, w_F)$;

$Q_H \leftarrow \texttt{RightShift}\,(Q_f + n, w_F)$;

$\mathsf{QExact} \leftarrow (Q_L == Q_H)$;

/* Combine the integer and fractional part of the quotient. */

$\bar{Q} \leftarrow \bar{Q} + Q_L$;

$[Q]_{m_e} \leftarrow \bar{Q} + Q_L \bmod m_e$;

return $\left(\bar{Q}, [Q]_{m_e}, \mathsf{QExact}\right)$;

Rc_Z	Quotient	$Q \bmod 4$	$Q \bmod m_j$	Remainder
0	0	0	$(0,0,0,0,0)$	0
1	144	0	$(0,0,4,4,1)$	24
2	288	0	$(0,0,3,1,2)$	48
3	433	1	$(1,1,3,6,4)$	8
4	577	1	$(1,1,2,3,5)$	32
5	721	1	$(1,1,1,0,6)$	56
$0 \le Rc_Z \le K, Q_c = \left\lfloor \frac{MRc_Z}{D} \right\rfloor, R_c = \left\lceil \frac{MRc_Z - Q_cD}{D} 2^{w_F} \right\rceil, w_F = 6$				

Table 4.4: QFS Algorithm Table-2 for our example.

Depending on the necessity of exactness, the first two cases are acceptable. The third case is undefined since the value leaves the range of the system and therefore is never acceptable. This algorithm requires no additional precomputations.

Correctness of the algorithms follows from the analytical derivations in [30, 31].

Algorithm 13: RP-RNS Modular Reduction Estimation

Input : Integer z in RP-RNS representation
Output : $z \bmod D$ in RP-RNS representation and a flag confirming exactness

1 **begin**
2 $(Q, Q \bmod m_e, \mathsf{QExact}) \leftarrow$ `QuotientFirstScaling` (z);
3 $R \leftarrow z - QD$;
4 $R \bmod m_e \leftarrow z \bmod m_e - Q \bmod m_e D \bmod m_e$;
5 $\mathsf{RExact} \leftarrow \mathsf{QExact}$;
6 **return** $(R, R \bmod m_e, \mathsf{RExact})$;

Example 3. We continue our example by performing a modular squaring operation on our example. $x = 5$ has RP-RNS representation $(1, 2, 0, 5, 5; 1)$. Its square, $x^2 = 25$, has RP-RNS representation $(1, 1, 0, 4, 3; 1)$. The result we seek is $x^2 \bmod D = 9$, which has RP-RNS representation $(1, 0, 4, 2, 9; 1)$.

The M_r values are unchanged: $(1155, 770, 462, 330, 210)$. The precomputed values are also unchanged: the M_r^{-1} values are $(1, 2, 3, 1, 1)$; the $[M_r]_{m_e}$ values are $(3, 2, 2, 2, 2)$;

and $[M]_{m_e} = 2$. For the quotient tables, we refer to Tables 4.3 and 4.4. The divisor D is $(0, 1, 1, 2, 5; 0)$.

The partial reconstruction algorithm computes the ρ values and reconstruction coefficient: $(1, 2, 0, 4, 3)$ and 2. The total of non-zero values is 5.

Using the ρ values and reconstruction coefficient as indices, we compute the sums for the integer and fractional parts of quotient: $(1, 1, 1, 1, 1; 1)$ and 36. The fractional part of the quotient does not change the integer part of the quotient because $\lfloor 36/64 \rfloor = 0$. Also, adding the error from the non-zero ρ values and reconstruction coefficient, we have $\lfloor 41/64 \rfloor = 0$. Hence $Q_{LO} = Q_{HI}$ and the quotient is exact.

Since the quotient is exact and 1, computing $z - QD$ yields the result we expected. The algorithm returns $(1, 0, 4, 2, 9; 1)$.

4.4.3 Time and Space Analysis

The running time of the QFS algorithm also depends on the same hardware components as does the partial reconstruction algorithm, but less so. The fraction adder does not depend on the hardware channels, so the fraction adder can run mostly in parallel with the hardware channels. Furthermore, the modular adders sum many entries from the quotient tables.

The algorithm sums K entries from Quotient Table 1 and 1 entry from Quotient Table 2 per residue channel and in the fraction channel. It performs 1 addition to combine the integer and fractional parts of the quotient. We use Phatak's assumption that each hardware channel has an adder tree and parallel access to all quotient table lookups. Under this assumption, table lookup runs in $\lg N$ time and the summation using the adder tree runs in $\lg N$ time also where N is the number of bits required to represent a number in our RNS. Reducing the fractional part of the quotient and combining it with the integer part also runs in $\lg N$ making the total execution time $\mathrm{O}(\lg N)$.

The QFS algorithm requires storage for the lookup tables. Both tables have the same space requirements per row, which must support the quotient and remainder. The number of rows in Quotient Table 1 equals the sum of the moduli ($\approx \mathrm{O}\left(K^2\right)$). Quotient Table 2 has one row per possible reconstruction coefficient for a total of $K+1$ rows. The total number of rows between both tables is therefore $\mathrm{O}\left(K^2\right)$. Each component of each row is no larger than $\mathrm{O}\left(\lg\lg K\right)$ bits. Therefore the total memory needed is $\mathrm{O}\left(K^3 \lg\lg K\right) \approx \mathrm{O}\left(N^3 \lg\lg N\right)$ bits.

The modular reduction estimation algorithm adds a negligible amount of time and space to the QFS algorithm. The algorithm adds time for one set of modular multiplications and modular subtraction for a minimum of 2 operations and a maximum of $2K$ operations depending on the number of hardware channels. The algorithm requires additional storage for the precomputed divisor D.

4.5 Modular Exponentiation Algorithm

The modular exponentiation algorithm in this section is a version of the binary modular exponentiation algorithm adapted to use the RP-RNS algorithms [30, 31].

4.5.1 Algorithm Description

Algorithm 14 [30, 31] is the RP-RNS modular exponentiation algorithm. Given a number x in RP-RNS representation and a number y in binary representation, it computes an estimate of $z \equiv x^y \bmod D$ where the divisor D is a pre-determined constant. It also returns an exactness flag that when true guarantees $z < D$ and when false only guarantees $z < 2D$. While the returned residue class will always be correct, we call this an estimate because it may exceed the divisor D.

The key modification to the binary modular exponentiation algorithm is the reduction operation. We replace the modular reduction by D with the modular reduction estimate algorithm (Algorithm 13). This algorithm places an additional requirement on the RP-RNS. The RP-RNS range must exceed $4D^2$. Phatak notes full double length values could be as large as $3D$ and therefore recommends a minimum range of $9D^2$. Correctness of the algorithms follows from the analytical derivations in [30, 31].

Algorithm 14: RP-RNS Modular Exponentiation

Input : $\bar{x}$ in RP-RNS representation and $y = \left(1, y_{n-2}, \ldots, y_0\right)$ in binary representation

Output : $\bar{z} \equiv \bar{x}^y \bmod D$ in RP-RNS representation where $\bar{z} < 2D$

```
1 begin
2     (z̄, z mod m_e, ZExact) ← ModRedEst (x̄, x mod m_e);
3     for i ← n − 2 to 0 do
4         (z̄, z mod m_e) ← (z̄, z mod m_e)^2;
5         (z̄, z mod m_e, ZExact) ← ModRedEst (z̄, z mod m_e);
6         if y_i = 1 then
7             (z̄, z mod m_e) ← (z̄, z mod m_e) × (x̄, x mod m_e);
8             (z̄, z mod m_e, ZExact) ← ModRedEst (z̄, z mod m_e);
9     return (z̄, z mod m_e, ZExact);
```

Example 4. In our last example, we compute $x^y \bmod D$ or $5^5 \bmod 16 = 5$. We do not provide the computation in complete detail. The curious reader can check for himself or herself. We instead provide the results after each modular multiplication in Table 4.5.

4.5.2 Time and Space Analysis

When performing modular exponentiation, the exponent y is within the range of the system due to the ultimate periodicity of repeated modular multiplication. So $\lg y \approx N$ where N is the number of bits needed to represent a number in our RNS. The binary modular

Step	Operation	Operation Result in RP-RNS
1.	$z \bmod D$	$(1, 2, 0, 5, 5; 1)$
2.	$z^2 \bmod D$	$(1, 0, 4, 2, 9; 1)$
3a.	$z^2 \bmod D$	$(1, 1, 1, 1, 1; 1)$
3b.	$xz \bmod D$	$(1, 2, 0, 5, 5; 1)$
Final Result	z	$(1, 2, 0, 5, 5; 1)$

Table 4.5: RP-RNS modular multiplication results by iteration for Example 4.

Algorithms	Running Time
Partial Reconstruction	$\mathrm{O}(\lg N)$
Quotient-first Scaling	$\mathrm{O}(\lg N)$
Modular Reduction Estimate	$\mathrm{O}(\lg N)$
RP-RNS Modular Exponentiation	$\mathrm{O}(N \lg N)$

Table 4.6: Summary of running times for the RP-RNS algorithms.

exponentiation algorithm performs roughly a minimum of N modular multiplications and a maximum of $2N$. In the modular multiplication, the limiter on the running time is the QFS algorithm. Using the analysis from the previous sections, we can conclude the running time is $\mathrm{O}(\lg N)$ where N is the number of bits needed to represent a number in our RNS.

The modular exponentiation algorithm structurally requires no storage. However, the modular reduction algorithm may have space requirements. RP-RNS modular exponentiation uses the partial reconstruction and QFS algorithms to perform the modular reduction.

We summarize the running times and storage requirements of all algorithms in Tables 4.6 and 4.7.

Precomputations	Space Requirements
Fraction Table	$\mathrm{O}\left(N^2/\ln N\right)$
Quotient Tables	$\mathrm{O}\left(N^3 \lg\lg N\right)$
Total	$\mathrm{O}\left(N^3 \lg\lg N\right)$

Table 4.7: Modular exponentiation space requirements in bits.

4.6 Reverse Conversion Algorithm

Binary-based systems employing RNS-based components use reverse conversion to convert the component output from RNS representation to binary representation. The CRT forms the basis for reverse conversion.

RP-RNS also employs a CRT-based reverse conversion algorithm. We can use the partial reconstruction algorithm to achieve a faster reverse conversion by taking advantage of Equation 4.4. We formalize this method as the RP-RNS reverse conversion algorithm (Algorithm 15). Correctness follows from the CRT and the correctness of the partial reconstruction algorithm.

Reverse conversion using Gauss's algorithm requires $4K$ high-radix arithmetic operations: K multiplications, K additions, and up to K subtractions and comparisons. Each subtraction requires a comparison with M because the algorithm does not know the reconstruction coefficient. This is a form of guess-and-check.

Algorithm 15 is of the same order requiring $2K + 1$ high-radix arithmetic operations: $K + 1$ multiplications, $K - 1$ additions, and one subtraction. However, Algorithm 15 is faster because the partial reconstruction algorithm computes the reconstruction coefficient, which eliminates all guesswork.

Algorithm 15: RP-RNS Reverse Conversion

Input : z in RP-RNS representation
Output : z in binary representation

```
1 begin
2     (ρ_1, ρ_2, ..., ρ_K, Rc_z) ← PartialReconstruction(z̄, z mod m_e);
3     for i ← 1 to K do
4         z ← z + ρ_i M_i;
5     z ← z − Rc_z M;
6     return z;
```

4.7 RP-RNS Design Strategies

We have seen how RP-RNS modular exponentiation achieves theoretically faster running time than do other forms of RNS modular exponentiation by using a time-memory tradeoff. For now, we shall focus on the theoretical aspects of designing RP-RNSs. We defer quantifying the price of the tradeoff until Chapter 6.

For the fast operations in RNS, more moduli results in higher performance due to the parallelism of RNS. For the slow operations in RNS, fewer moduli results in fewer high-radix operations. RP-RNS seeks to increase speed of the slow operations. It follows we should use few large moduli, regarding less the memory overhead from the RP-RNS algorithms. However, the RP-RNS algorithms completely avoid high-radix arithmetic because of the approximation. Therefore, Phatak suggests we choose the small prime moduli for our RP-RNS bases. It follows from the Prime Number Theorem that our base grows with $K \approx \mathrm{O}\left(\lg M / \ln \lg M\right)$.

Now suppose we bring storage into our design analysis. The lookup table for the partial reconstruction algorithm is negligible in size compared to the QFS tables. The size of each QFS table row is proportional to $\lg M \lg \lg M$. The number of rows roughly equal to the sum of the moduli in the RNS base. In most cases we minimize this sum by selecting small

prime moduli as Phatak suggests. However, optimal RP-RNS base selection is remains an open problem.

4.8 Conclusion

In this chapter, we defined the reduced-precision residue number systems. We explained the theory behind RP-RNSs and the algorithms to manipulate them. To summarize the system and algorithms, they work by taking advantage of constant divisors and precomputing low-radix approximations instead of computing high-radix arithmetic operations.

The RP-RNS modular exponentiation algorithm uses the partial reconstruction algorithm and quotient-first scaling algorithm to implement modular exponentiation using the Division Algorithm instead of RNS Montgomery multiplication.

We presented Phatak's time and space analyses using his assumptions. The QFS algorithm runs in $\mathrm{O}\left(\lg N\right)$ time and requires $\mathrm{O}\left(N^3 \lg N\right)$ bits. The QFS algorithm is the largest contributor to the modular reduction estimate algorithm's running time and storage requirements. The analyses provide insight into our results and analysis of our hardware implementation in Chapter 6.

Chapter 5

RP-RNS Implementations

We developed two working implementations of modular exponentiation using RP-RNS algorithms: software-based and hardware implemented on an FPGA. In this chapter we describe our design, architecture, and implementation.

We discuss two design decisions we made in which we exchanged performance for additional hardware channels. Our decisions increase the delay of performing QFS from $\mathrm{O}\left(\lg N\right)$ to $\mathrm{O}\left(N\right)$. The architecture we describe is standard and our implementation section provides enough detail for the reader to replicate a design similar to ours.

At the end of the chapter, we explain important points the reader should use to interpret the results in Chapter 6.

5.1 Software Implementation

Though our focus is hardware, we first developed a software implementation to fulfill two purposes: to validate our understanding of the RP-RNS system and algorithms and to

re-verify the correctness of the algorithms numerically. Beside analytical derivations, Phatak has done extensive simulations to numerically verify all the algorithms [30, 31].

We wrote our software implementation in C originally and later ported it to a Python-based mathematics software called Sage [3] for its support of vector-based operations, which represents the parallelism explicitly. We executed this software on commodity 64-bit architecture hardware, which carries two limitations on parallelization.

The first limitation is commodity hardware. As of the year 2013, most commodity hardware features at most 16 cores with 64-bit arithmetic logic units [11, 18, 29]. An RP-RNS using the first 15 prime numbers achieves a 60-bit main modulus with at most 6-bit channels; the last core is reserved for the redundant channel. We can mitigate this waste of hardware resources by using moduli optimal for the hardware (e.g. 32-bit or 64-bit moduli), but using larger moduli results in higher storage requirements as noted in Chapter 4.

The second limitation is due to software. Software may not leverage all available processing cores making the software effectively sequential [39]. Our software implementation suffers from this issue. These two limitations drive the need for our specialized RP-RNS hardware.

5.2 Hardware Platform

For our target chipset, we used a Xilinx Spartan-3E FPGA (XC3S500E) with a clock frequency of 50 MHz. We used the vendor's toolchain, Xilinx ISE v14.5, for design synthesis for compatibility and to take advantage of the performance offered by intellectual property blocks. Table 5.1 summarizes the interesting attributes of the chip. The logic cell structure includes a 4-input lookup table (LUT), a register, a 2-to-1 multiplexer, and carry/arithmetic logic. While not specific to this particular FPGA, the number of dedicated

Hardware Attributes	Quantity
Clock frequency	50 MHz
Configurable logic blocks	1,164
Logic slices	4,656
Lookup Tables/Flip-flops	9,312
Equivalent Logic cells	10,476
Equivalent System gates	500K
Distributed memory	74,496 bits
Block memory	360K bits
Multipliers	20
Clock domains	4
I/O pins	232

Table 5.1: Xilinx Spartan-3E (XC3S500E) specifications

multipliers is interesting because they are faster than logic cell-based equivalents. In a sense the number of dedicated multipliers can limit the number of residue channels on a single FPGA.

We note the literature often uses a higher performing FPGA and clock (e.g., Xilinx Virtex 5 at 100 MHz) and our hardware is non-standard. We chose this hardware simply because it was available. We hopefully provide enough information for a useful comparison in Chapter 6.

It is worth mentioning that while our development board (Figure 5.1) includes 256 MB of RAM, we did not leverage it out of concern for an execution time penalty.

5.3 Hardware Design

We developed our hardware implementation using VHDL, which is a platform-agnostic hardware design language (HDL).

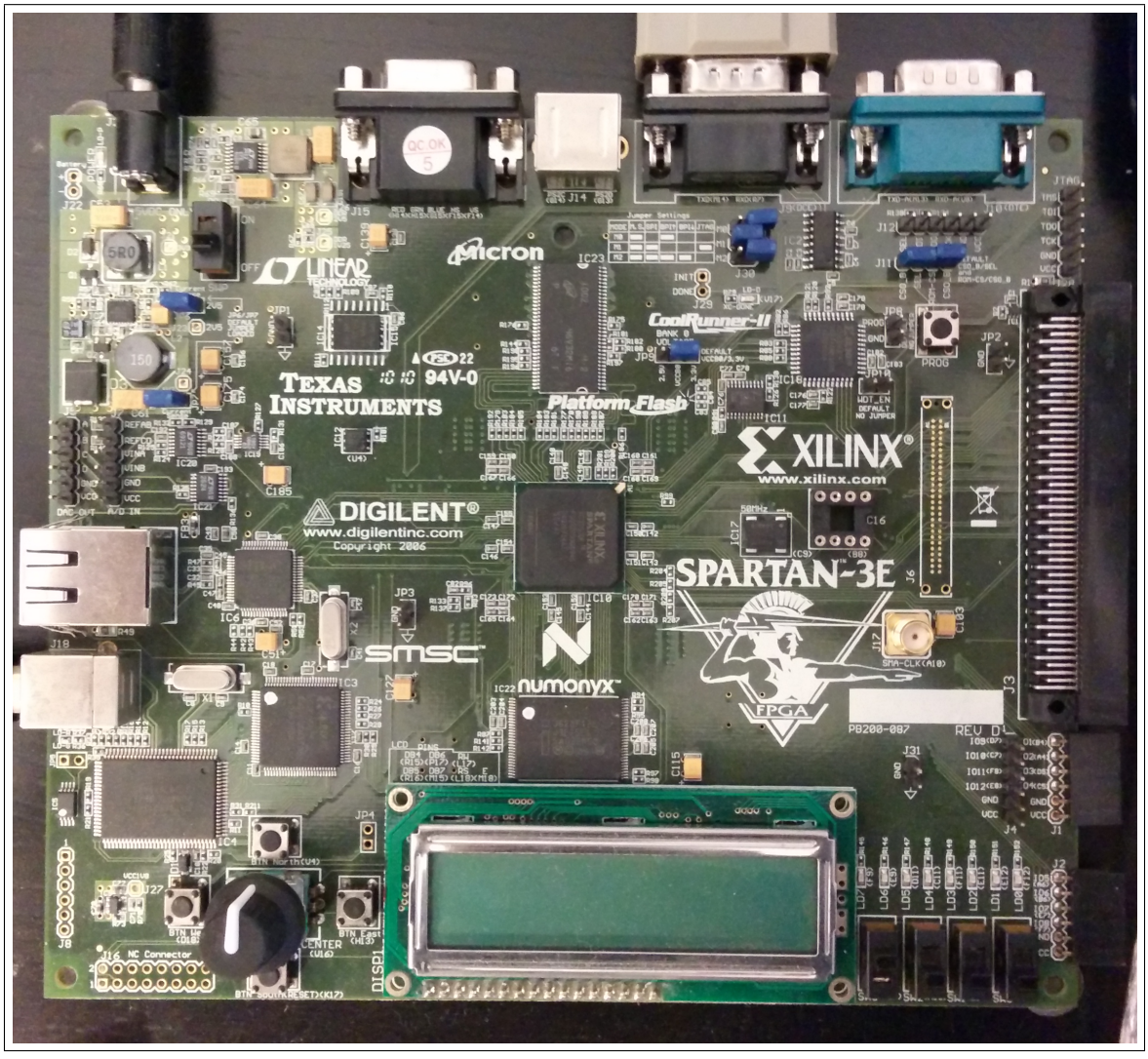

Figure 5.1: Spartan-3E development board by Digilent, Inc.

We originally planned a speed-optimized implementation and to fully implement Phatak's assumptions:

1. the RP-RNS base consisting of the first K prime numbers; and

2. use of parallel-access storage and adder trees.

We ultimately chose to design our hardware to allow for more residue channels by replacing hardware entities based on the number of logic cells used. Impacted hardware included the parallel-access storage and adder trees, which we replaced with equivalent sequential hardware. This carries a theoretical running time penalty from $\mathrm{O}\left(\lg N\right)$ to $\mathrm{O}\left(N\right)$ where N is the number of bits to represent a number in our RNS. In Chapter 6 we quantify this penalty more precisely. In attempt to maximize execution time, we decided to store the precomputed lookup tables on the FPGA close to the logic that uses them.

We chose to keep Phatak's choice of RP-RNS base for comparison with Phatak's theoretical results and due to the growth rate of the channel word lengths. In reality, however, our hardware design language (HDL) code supports arbitrary RNS bases and generates an appropriate FPGA implementation.

Though our emphasis is residue domain modular exponentiation, we designed our hardware to be compatible with existing binary-based digital systems. By implementing the forward and reverse conversion algorithms, our hardware is usable as a drop-in component into existing systems.

We designed our hardware with pre-configured parallelism in mind. We allow multiple residue channels to share a single hardware channel. Reducing the parallelism increases execution time by the same factor.

As a convenience, our HDL code also generates the lookup tables. In retrospect we determined this feature is both unnecessary and undesirable. It is unneeded because if the

RP-RNS modular exponentiation hardware can support a divisor D, then it can support all smaller divisors by simply rewriting the lookup tables. It is undesirable because the vendor's synthesis software could not handle the processing required to generate the lookup tables.

5.4 Hardware Architecture

Figure 5.2 is a graphical representation of our architecture. This architecture is standard and possesses no distinguishing features. Our hardware architecture comprises four major components: the controller, the hardware channels, the redundant residue channel, and the fraction channel.

The diagram depicts several moduli per hardware channel to illustrate the configurable parallelism mentioned earlier. We employ this when the number of residue channels exceeds the number of hardware channels we can instantiate on the FPGA. We denote the number of hardware channels as p. p represents the amount of parallelism inherent in the implementation.

The controller contains the logic for performing five operations: forward conversion, partial reconstruction, scaling, modular exponentiation, and reverse conversion. We implemented all operations using the descriptions contained in Chapter 4.

A hardware channel executes residue operations for each residue channel – possibly multiple channels. We divided the moduli equally across all hardware channels such that each hardware channel supports at most $m = \lceil K/p \rceil$ residue channels. To keep hardware small, we distributed the moduli in increasing order (i.e. channel 1 has the smallest m moduli, channel 2 has the next smallest m moduli, etc.). Furthermore, each hardware channel stores the lookup tables for its respective residue channels.

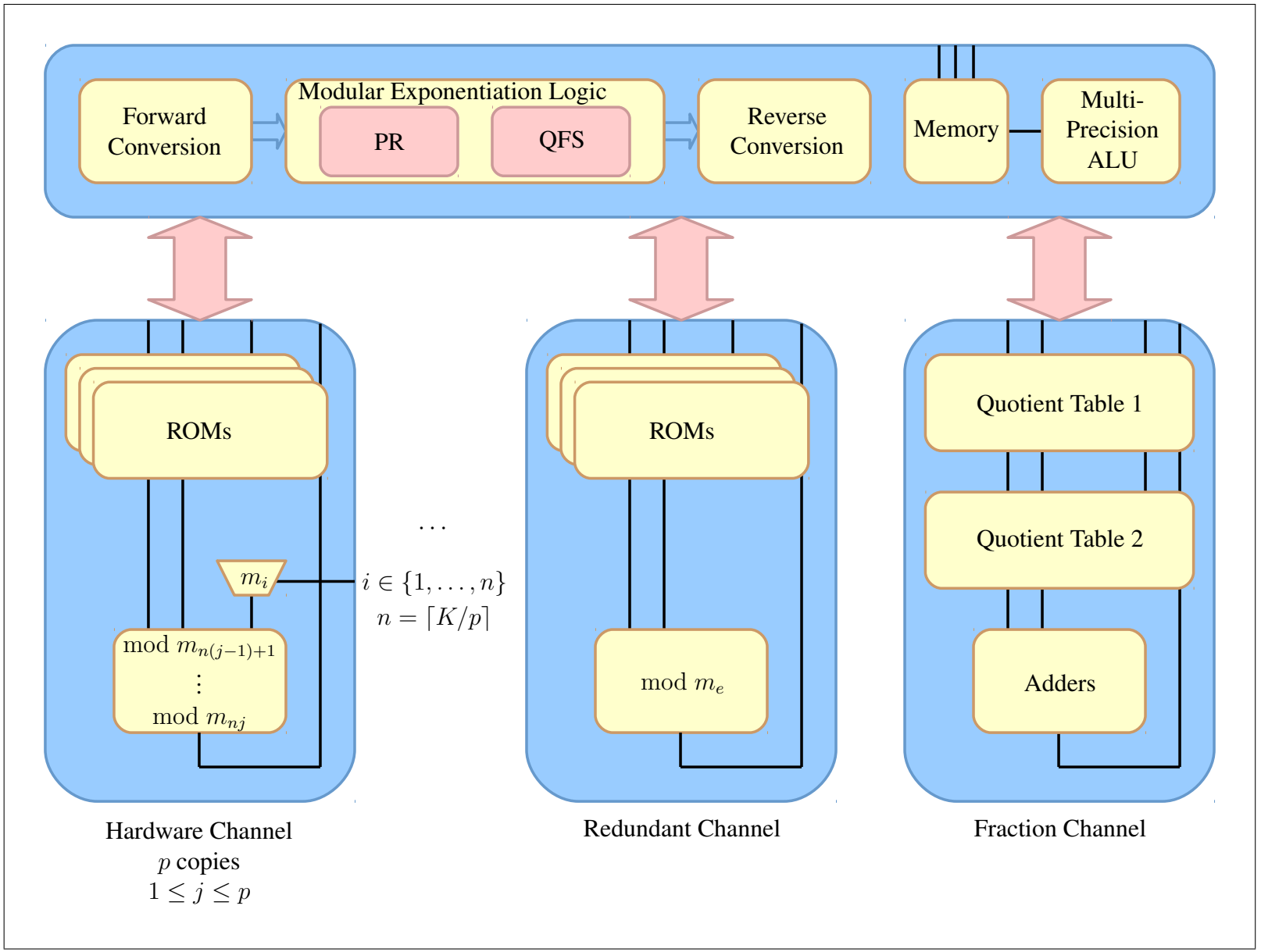

Figure 5.2: RP-RNS Architecture

The redundant residue channel is roughly identical to the hardware channels, but dedicated to the extra modulus m_e. The redundant residue channel also stores its own lookup tables. Since the redundant modulus is either 2 or 4, we were able to make some optimizations in implementing this channel.

The fraction channel is similar to the redundant residue channel, except dedicated to the approximated fractional values. The channel's arithmetic logic must support word lengths of w_T bits where $w_T = w_I + w_F$ and $w_I = \lceil \lg K \rceil$. w_F is the fractional precision specified in the RP-RNS algorithms. w_I supports the overflow of adding two truncated fractions.

5.5 Hardware Implementation

In this section we expand on our architecture and explain our implementation in detail. We quantify the number of registers, arithmetic blocks, and lookup tables. We do not provide a gate-level description since we used the hardware synthesizer's logic inference where possible to take advantage of the performance offered by the vendor's intellectual property blocks.

5.5.1 Controller

The controller comprises several registers, multiplexers, counters, and two state machines. We summarize the quantities and sizes of these components in Table 5.2.

The I/O registers store the base, exponent, and solution in binary form and provide an interface to binary-based systems. One register stores the reconstruction coefficient used in the partial reconstruction algorithm, QFS algorithm, and reverse conversion algorithm. The residue registers store the base, solution, ρ values, quotient, and intermediate values (e.g., reducing the scaled fractional part of the quotient).

Purpose	Size	Quantity
I/O registers	$\lg M$ bits	3
Reconstruction coefficient register	$\lceil \lg K \rceil$ bits	1
Residue registers	$\approx \lg M$ bits	5
Multiplexers	$\lceil K/p \rceil$-to-1	$3p$
Counters	$\lceil \lg K \rceil$	2

Table 5.2: RP-RNS controller composition.

The multiplexers map the residue registers to the hardware channels. Two sets are for channel input and the other for channel output. The multiplexers use a counter to determine which portions of the residues with which to compute. Our implementation uses a second counter for keeping track of accumulations for the partial reconstruction algorithm.

The two state machines implement the algorithms. The higher-level state machine implements the modular exponentiation and conversion logic. The lower-level state machine implements the partial reconstruction and QFS algorithms.

The state machines are less than optimal in two respects. First, to enable easy software debugging, the state machines use more states than necessary to implement the algorithms. Second, the state machines do not take full advantage of the parallelism inherent in the partial reconstruction and QFS algorithms. The reconstruction coefficient computation and ρ-based quotient accumulations are parallelizable, but we did not implement this to save on FPGA resource utilization and possibly gain an additional hardware channel.

5.5.2 Hardware Channel

Each hardware channel has a modular ALU, multiplexers, storage for constants, and storage for the lookup tables.

A hardware channel's modular ALU is large enough to support the largest modulus distributed to that channel. Our choice of moduli distribution results in a design using the

smallest modular ALUs possible. A time-optimized hardware channel would include a modular adder tree and an additional modular ALU to take advantage of the full parallelism inherent in the QFS algorithm.

We designed the hardware to keep the lookup tables local to their respective residue channels. There are a total of $4K$ lookup tables: 1 for forward conversion, 1 for partial reconstruction, and 2 for QFS. This decision enables simultaneous querying of all $4K$ tables, which can increase performance. On FPGAs, the number of entries in each memory element must be a power of 2. More often than not this simplifying requirement leads to waste, which we quantify in Chapter 6.

We did not specify explicit memory constructs for storing the non-table constants and left that choice to the hardware synthesis software.

The multiplexers enable hardware resource sharing within each channel. The hardware channel, based on controller input, uses the multiplexers to select the modulus for the modular ALU and to select which lookup tables to reference. Each hardware channel has a total of $4p$ multiplexers.

5.5.3 Redundant Residue Channel

The redundant residue channel consists of a modular ALU and storage for constants and lookup tables. Since the redundant modulus is either 2 or 4, we were able to make some optimizations in the design.

The first optimization is the modular ALU. It requires at most 3 inputs for a multiply-accumulate operation, so we were able to implement it as a lookup table.

The second optimization is the lookup tables. We used the same storage approach for the redundant channel as the hardware channels resulting in 2 lookup tables for the QFS

algorithm. Since the modulus is a power of 2, we implemented forward conversion using truncation instead of lookup tables.

5.5.4 Fraction Channel

The fraction channel consists of a w_T-bit ALU and storage for 2 look-up tables. Phatak notes that the ALU should include a K-operand adder tree. Our original hardware implementation supported the adder tree, but was removed due to area constraints. The implementation discussed in this thesis does not include the adder tree and instead uses a simple 2-operand adder in a sequential manner.

The two look-up tables are for the truncated fraction remainders specified in the QFS algorithm. We stored the partial reconstruction look-up tables with the hardware channels because the values and table size depends solely on the residue channel.

5.6 Remarks

Figure 5.2 reflects our hardware architecture, which is typical.

As a design decision, we modified two of Phatak's assumptions: exchanging the per-hardware channel adder tree for a single adder in each channel; and supporting single-access storage due to the hardware synthesizer's logic inference. This increases the execution time complexity of the modular multiplication from $\mathrm{O}\left(\lg N\right)$ to $\mathrm{O}\left(N\right)$. In turn, this increases the execution time complexity of the modular exponentiation from $\mathrm{O}\left(N \lg N\right)$ to $\mathrm{O}\left(N^2\right)$. We kept Phatak's assumption regarding the choice of RP-RNS base.

We made these decisions to optimize FPGA resource utilization to accommodate more hardware channels. Moving forward, the reader should keep these decision in mind when interpreting the results in Chapter 6, particularly regarding the execution time.

Chapter 6

Testing Methodology and Results

In this chapter we describe our testing methodology and results. Our testing methodology includes a description of our test apparatuses, validation of the apparatuses, and the metrics we collect. We present standard metrics including FPGA utilization, storage, and cycle counts. All measurements were derived directly from the physical hardware or a simulation validated against the hardware. We conclude this chapter with our findings, which include the following.

1. The size of the hardware logic grows at a rate $K \lg K$ where K is the RNS base size.

2. Storing the precomputed tables as distributed RAM on the FPGA requires significantly more storage than the projected theoretical model. This is a result of our design decisions and not reflective of the theory.

3. We confirmed the overall storage growth rate, however, agrees with the theoretical model up to a difference in coefficients.

4. Storing the precomputed tables on the FPGA leads to a hardware logic growth rate proportional to the theoretical memory growth rate.

5. Our design compromises led to a suboptimal growth in execution time.

6. We confirmed the theoretical logarithmic performance trend of RP-RNS modular multiplication.

6.1 Purpose

Commonly cited metrics in the literature include hardware area, storage requirements, and execution time (or cycle count). We designed experiments to collect these metrics. Another common metric often studied separately is power consumption. We chose not to measure power consumption since the additional chips on our development board would have complicated the collection process.

As an additional note, the literature we cited in Chapter 3 implemented their hardware as ASICs and measure hardware area as the number of logic gates. Since we implemented our hardware on FPGA, the closest applicable metric we can collect is the FPGA resource utilization.

6.2 Testing Methodology

We designed experiments to collect three different metrics: FPGA utilization, storage, and cycle counts. Our experiments incorporated testing our implementation on three different platforms: synthesized hardware, hardware simulation, and software simulation. We used all three platforms to collect data, but most of the data came from our software simulation for the ease of automation.

6.2.1 Test Apparatus Descriptions

Our synthesized hardware test apparatus consisted of the FPGA development board and a personal computer. From the computer, we commanded the hardware, via serial Universal Asynchronous Receiver/Transmitter (UART) as the communications protocol, using a test harness written in the Python programming language.

For hardware simulation, we used our FPGA vendor's simulation software: Xilinx iSim v14.5. The hardware simulator has two modes: pre-synthesis and post-synthesis simulation. We used both modes in our testing, however this made no difference as we describe in our discussion of validation.

We wrote our software simulation in the Python programming language. Our software simulation differs from our software implementation described in Chapter 5; in addition to verifying correctness, our software simulation performs all operations that could cause variations in our cycle count (e.g., resource sharing and residue channel modular multiplication).

6.2.2 Test Apparatus Validation

It is important to confirm the validity of our simulations as the data are valid only if the simulation is valid. We consider our apparatuses valid when all tested inputs and outputs between all apparatuses are in agreement. In our setup we trusted the vendor's synthesis software for correct FPGA utilization metrics. We used the VHDL code directly to derive the storage metric. The two remaining items requiring validation are functional correctness and performance models.

Our validation method could be considered unconventional, but is justifiable. A chain of validations underlies the flow of our argument. We start with the software implementation and through a validation chain prove correctness of the hardware. Once we know the

hardware is functionally correct, we use a second validation chain to prove our software simulation correctly models performance. We now describe this process in detail.

Validation Process Summary

We started off with our software implementation to numerically verify the functional correctness of the RP-RNS modular exponentiation algorithm and its dependent algorithms. From there we used the software implementation to validate correctness of the hardware simulation. For FPGAs this is sufficient to confirm the physical hardware has the correct wiring, however the hardware may not be correct due to an incorrect timing model. At this stage, an appropriate timing model is equivalent to correct hardware.

We developed a conservative timing model by doubling values from the vendor's data sheets and used counters to integrate the delay model into the VHDL. This method has the benefit of producing an accurate hardware simulation timing model since both derive from the same VHDL. We verified the correctness of the hardware. In the final step we used the hardware simulation to develop the cycle count model for the software simulation. As an extra sanity check, we compared the inputs, outputs, and cycle counts of the software simulation to the physical hardware.

Validation Process Details

Now let us discuss quantity of validation. Phatak [30, 31] performed software verification for correctness using the Maple mathematics software [1]. He verified the algorithms with random bases and exponents for main moduli up to 2^{20} bits (i.e., 2^{19}-bit divisor). We independently verified the same algorithms using Sage [3], which is similar to Maple and built on top of Python [2]. Our independent verification included main moduli up to 2^{16} bits (i.e., 2^{15}-bit divisor).

For each link in our validation process, we used 100 sets of random inputs – base, exponent, and divisor – for all divisor bit-lengths between 4 and 20 bits producing 1,600 data points at each step in the validation process. For the hardware and software simulators we were able to verify up to 256 bit divisors.

Validation Process Limitations

We acknowledge the quantity of physical tests is less than ideal. Three factors limited the number of physical tests we could perform.

The first was a flaw in the vendor toolchain: the vendor's synthesis tool and hardware simulator often generated different lookup tables. The simulator always generated the correct tables whereas the synthesizer often would not. Manual modification of the simulator's lookup tables to match the synthesized hardware resulted in agreement.

The second was the fragility of the protocol we used to interface the FPGA with the test driver software. Any deviation would put the FPGA into a bad state requiring a reset of the FPGA and the test driver software. Despite these issues, we are confident the 1,600 tests were sufficient to validate our three platforms.

The third was the autonomy feature of our hardware implementation. The precomputed tables are independent of the system in a sense. Aside from different values in the QFS tables, two synthesized instances may be otherwise identical. Embedding the lookup table generation into the VHDL required hardware re-synthesis when the divisor changed.

Our validation process resulted in three consistent test apparatuses including a functionally correct hardware implementation. While we feel our process was sufficient, we would have preferred a more direct and faster validation process.

6.2.3 Metric Description and Collection

Our choice of metrics are standard: FPGA resource utilization, storage, and cycle counts. The collection process for these metrics is straightforward.

Information regarding FPGA resource utilization comes from the vendor's synthesis report as an occupied logic slice count. It is hard to compare utilization between two FPGAs because their logic slice structure may differ. Using the vendor's synthesis report we track three additional resource types: flip-flops, logic lookup tables (LUTs), and routing LUTs. These resources are contained in the occupied slice count.

We determine storage by adding up the size of the precomputed RP-RNS algorithm tables using bits as our unit. We wrote software to compute this value based on the VHDL code.

In addition to cycle counts, we provide execution time based on our FPGA's clock rate of 50MHz. We measured this data from all three test apparatuses though most of our data came from the software simulator.

6.3 Test Results

In this section we present our data for each metric described in the previous section: FPGA resource utilization, storage, and cycle count. As our metrics are sensitive to the choice of RP-RNS base, we remind the reader that we selected the first K prime numbers for our RNS bases. In the sections that follow, we state briefly with each metric whether the results depend on the RP-RNS base choice and defer detailed discussion to the analysis.

K	Slice Flip-flops	Logic LUTs	Routing LUTs	4-input LUTs	Occupied Slices
4	952 (10.22%)	2,119	69	2,188 (23.50%)	1,344 (28.87%)
5	1,112 (12.03%)	2,496	80	2,576 (27.66%)	1,592 (34.19%)
6	1,217 (13.07%)	2,967	90	3,057 (32.83%)	1,857 (39.88%)
7	1,364 (14.65%)	3,513	85	3,598 (38.64%)	2,178 (46.78%)
8	1,506 (16.17%)	3,871	91	3,962 (42.55%)	2,408 (51.72%)
9	1,643 (17.64%)	4,654	106	4,760 (51.12%)	2,854 (61.30%)
10	1,872 (20.10%)	5,416	116	5,532 (59.41%)	3,330 (71.52%)
11	2,011 (21.60%)	6,448	114	6,602 (70.90%)	3,920 (84.19%)
12	2,178 (23.39%)	6,608	130	6,738 (72.36%)	4,040 (86.77%)
13	2,359 (25.32%)	7,833	130	7,963 (85.51%)	4,654 (99.96%)
14	2,491 (26.75%)	8,541	134	8,675 (93.16%)	4,654 (99.96%)
15	2,692 (28.91%)	9,836	132	9,968 (107.04%)	5,322 (114.30%)
Available	9,312			9,312	4,656

Table 6.1: Utilization of Spartan-3E (XC3S500E) FPGA for K channels.

6.3.1 FPGA Utilization Metrics

Table 6.1 captures the impact of RNS base size on the utilization of our Spartan-3E (XC3S500E) FPGA. The FPGA utilization metrics depend solely on the number of moduli because the channel growth is negligible (i.e., $\approx \lg \lg M$ where M is the range of the system). The most interesting columns are the numbers counting slice flip-flops, LUTs, and occupied slices. Since we derived these data from the physical hardware, the data set is small and what we can say is limited.

The flip-flops are related to the registers in our implementation. The number of flip-flops increases at a rate proportional to K. This rate actually depends on M because K is logarithmically related to M. This is exactly what we expect, however. While it is hard to see the impact from this data set, we would see a greater impact on a larger data set. This FPGA would run out of flip-flops near 40 moduli.

Now we turn to the relation between flip-flops and LUTs. First note the number of flip-flops and LUTs available is twice the number of total slices available. Furthermore, the number of flip-flops and LUTs are the same. This is because slices contain two identical logic cells consisting of both flip-flops and LUTs (see Chapter 2). Therefore correlating these two columns is pointless.

The distinction between logic LUTs and routing LUTs helps quantify how much FPGA space is lost due to inefficient design. If we were to plot the number of routing LUTs against K, we would see a trend that is either logarithmic or linear. Furthermore, the proportion of routing LUTs to logic LUTs decreases. For now we will just say that this trend is deceptive and defer explanation to the analysis section.

The most interesting trend with respect to Table 6.1 is the superlinear growth rate of the total number of LUTs as a function of the RNS base size. We provide a graph of this trend in Figure 6.1. We used the Levenberg-Marquardt algorithm found in most statistical software packages to fit the following superlinear curve to our data:

$$\#\,\mathrm{LUT}\,(K) \approx 151.094K\log\,(K) + 663.406. \tag{6.1}$$

The limited data prevents us from drawing significant conclusions, but the clear superlinear trend indicates there is a stronger contributor to LUT utilization than the hardware channel logic. The logarithmic increase leads us to suspect the precomputed lookup tables as a factor., particularly the QFS tables.

The synthesis report also provided details regarding inferred logic, which we summarize in Table 6.2. We can infer from these data that the synthesizer uses roughly the same structure for all of the hardware channels.

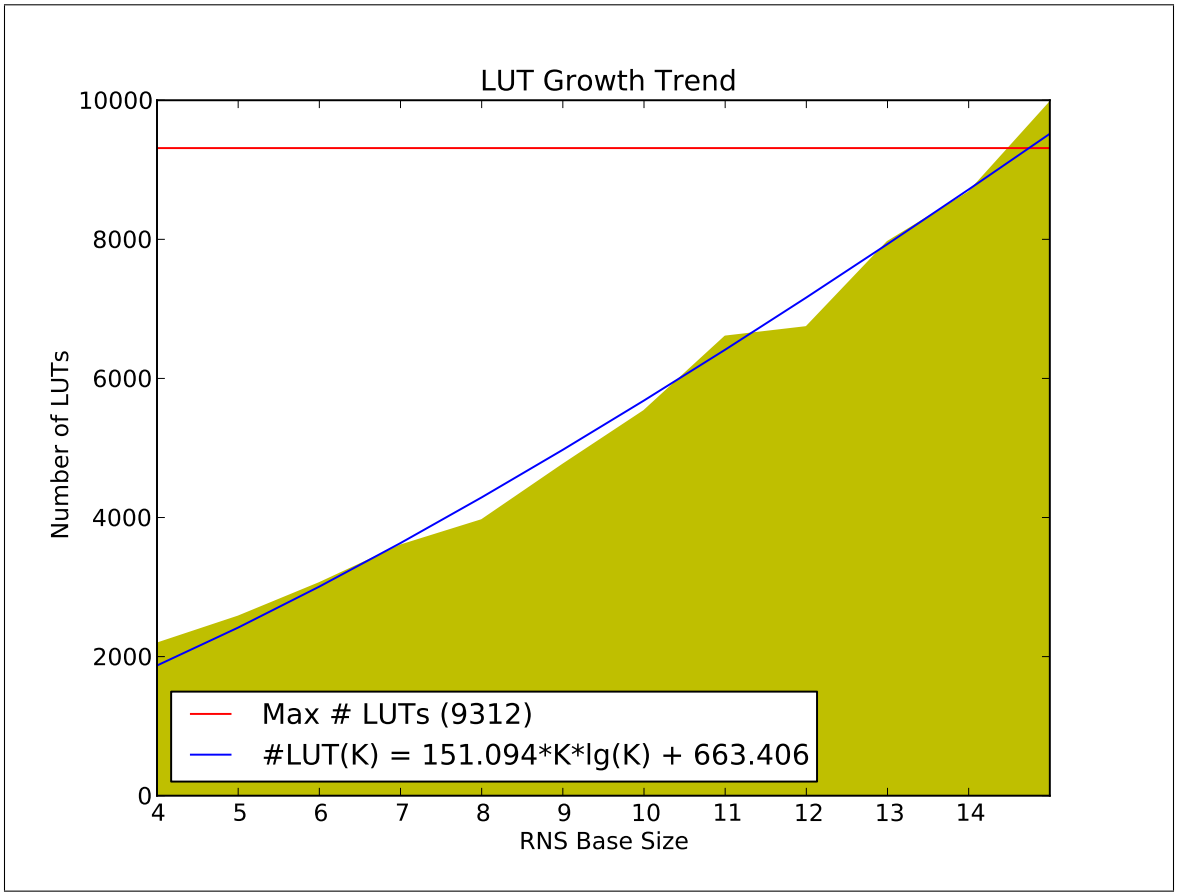

Figure 6.1: Superlinear dependence of FPGA logic utilization on RNS base size.

	Hardware Channel	Redundant Channel	Fraction Channel	Modular ALU
# ROMs	$3\lceil K/p \rceil <$	3	2	–
# Adders	3	5	4	4
# Multipliers	–	1	–	–
# Comparators	7	4	3	2
# Counters	–	–	–	1
# Multiplexers	–	–	–	1

Table 6.2: Inferred logic for our hardware implementation.

6.3.2 Storage Metrics

Figures 6.2 and 6.3 illustrate the precomputed lookup table storage for our implementation. We emphasize these data are specific to our implementation due to decisions made during hardware design. These metrics are absolute for our implementation since they were computed using the VHDL specifying our design. We also emphasize our implementation lies between the theoretical minimum and theoretical maximum from an unoptimized implementation.

The storage requirements depend on two factors. The first factor is the choice of RP-RNS base, which we used the first K prime numbers with 4 as the redundant modulus. As noted in Chapter 4, the number of QFS table entries equals the sum of the moduli. We aimed to minimize the sum of the moduli; whether we succeeded or not requires solving a separate optimization problem. The second factor is how the lookup tables are embedded into an RP-RNS implementation. We divided each LUT to be local to its respective channels. Despite Phatak's recommendation, we did not enable parallel access to Quotient Table 1. We explore the impact of these decisions as they relate to the results in the analysis section.

We performed a regression against Phatak's $\mathrm{O}\left(K^3 \lg \lg K\right)$ result from Chapter 4:

$$\mathrm{Memory}\left(K\right) \approx 18.783 K^3 \lg \lg K + 5.234 * 10^7 \mathrm{GB}. \tag{6.2}$$

The overall trend of the graph resembles the theoretical results excepting two properties: the large coefficients and the alternation between linear trends and sudden jumps. Both of these are a result from our design decisions and how the hardware translates to synthesized logic.

The linear increases derive directly from the theory. As the divisor length increases, the RNS base increases, which requires adding additional columns to the partial reconstruction table and QFS tables.

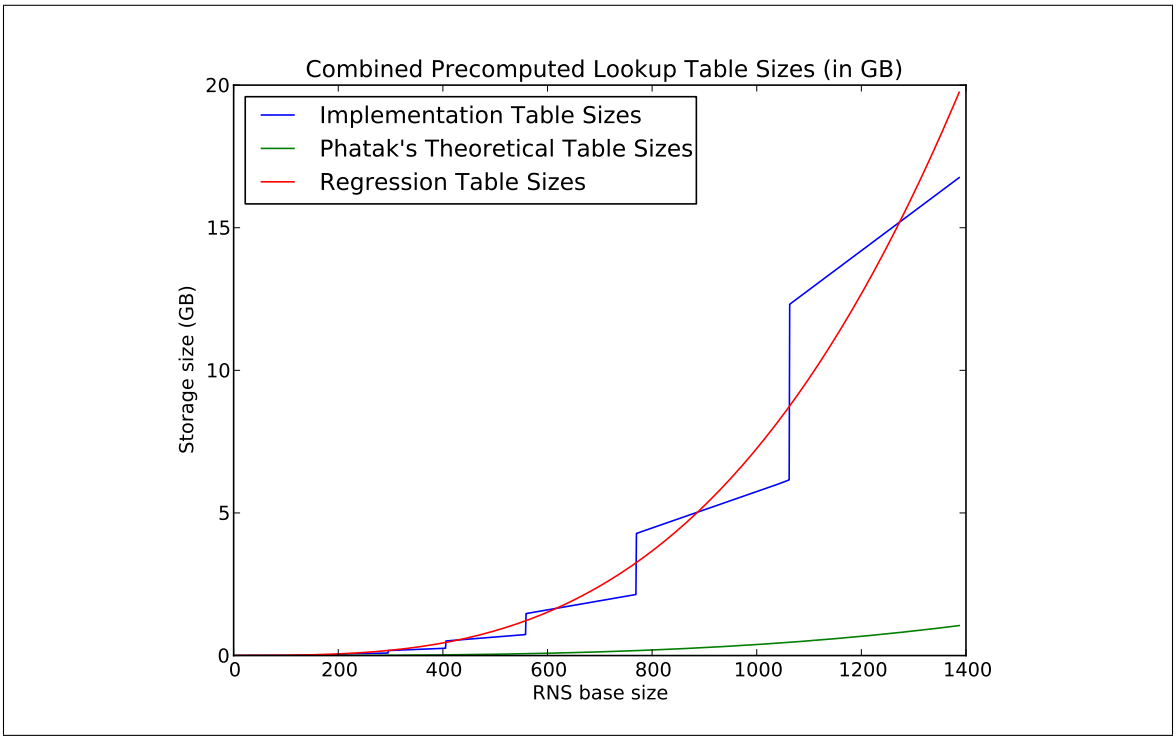

Figure 6.2: Total size of precomputed lookup tables for our implementation.

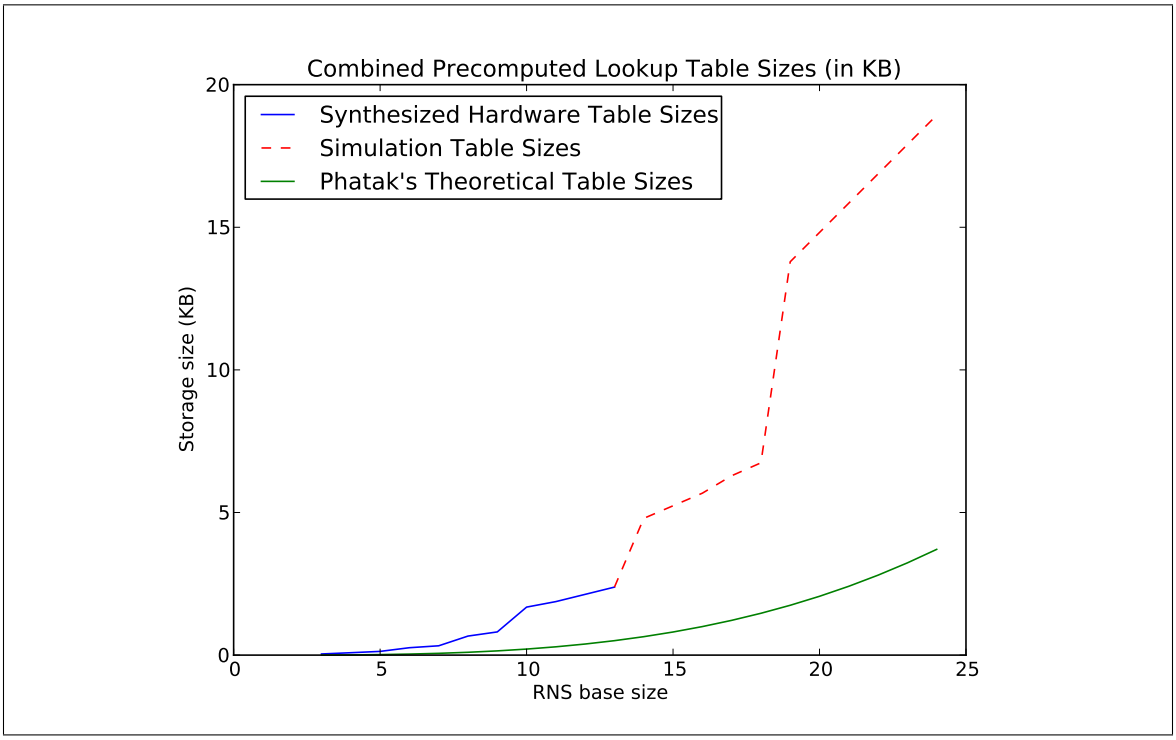

Figure 6.3: Cutout of storage requirements for our synthesized hardware from Figure 6.2.

The jumps are a result of how the hardware realizes the implementation. Our implementation used the synthesizer's logic inference to generate the lookup tables. This requires the number of table entries to be a power of two for all tables. The jumps occur when the number of QFS table entries crosses a power of two because the number of QFS table entries depends on the sum of the moduli. Note the partial reconstruction tables do not contribute significantly to these jumps because adding additional channels does not require modifying existing channels.

6.3.3 Cycle Count Metrics

In this section we only discuss data we consider to have high utility. We relegate the raw data tables to Appendix B.

Before we proceed, we caution the reader on two points. First, other RNS modular exponentiation studies use ASICs optimized for a set of fixed divisor lengths; their results do not include a wide range of divisors. Second, unlike other studies in the literature, we did not optimize our implementation for speed. Therefore, the cycle count data we present are valid only for our implementation. While a direct comparison with the literature is invalid, the trends deriving from our data are valuable and useful.

Figure 6.4 summarizes our cycle count data for computing a single multiplication modulo-D. There are four types of data represented on this graph. Each type comes as a pair depicting average case and worst case cycle counts. The first three types of data represent different levels of parallelization (top-to-bottom): single hardware channel (sequential), three hardware channels (semi-parallel), and K hardware channels (full parallel). The fourth type of data is a theoretical modification of our implementation; we address this data further in the analysis section. Though not well-represented in the figure, the confidence interval

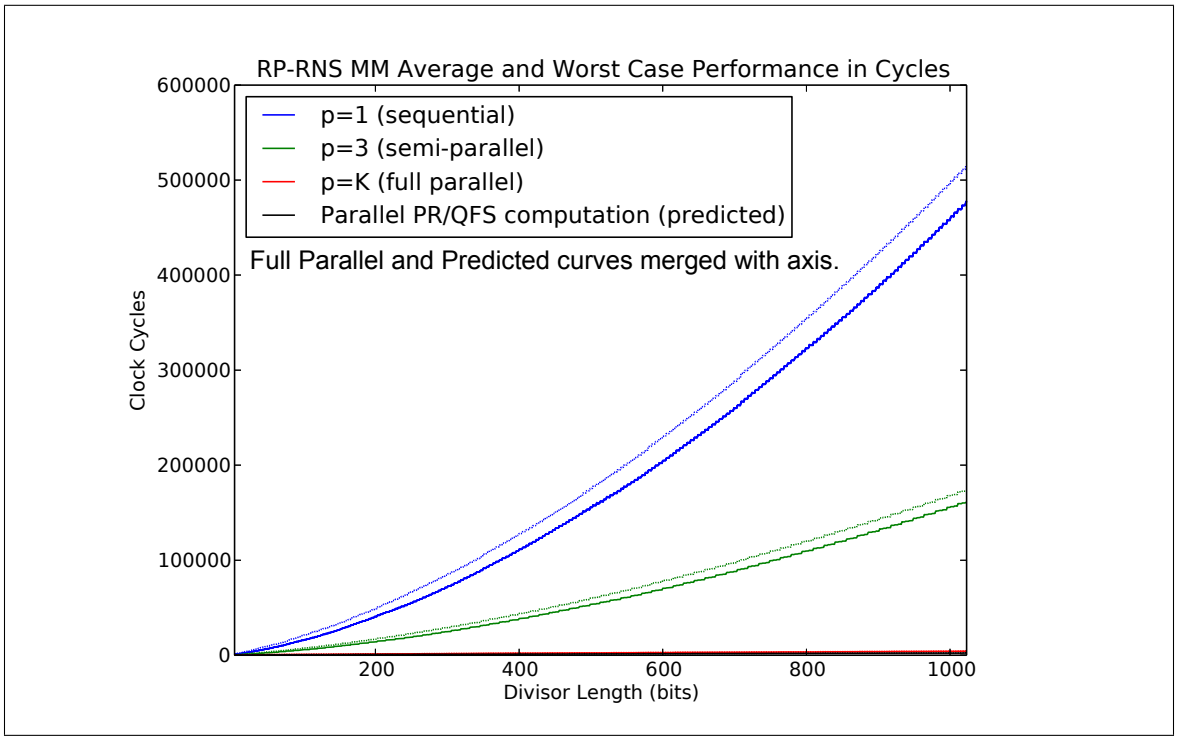

Figure 6.4: Cycle counts for divisors up to 1024-bits using four varieties of parallelization.

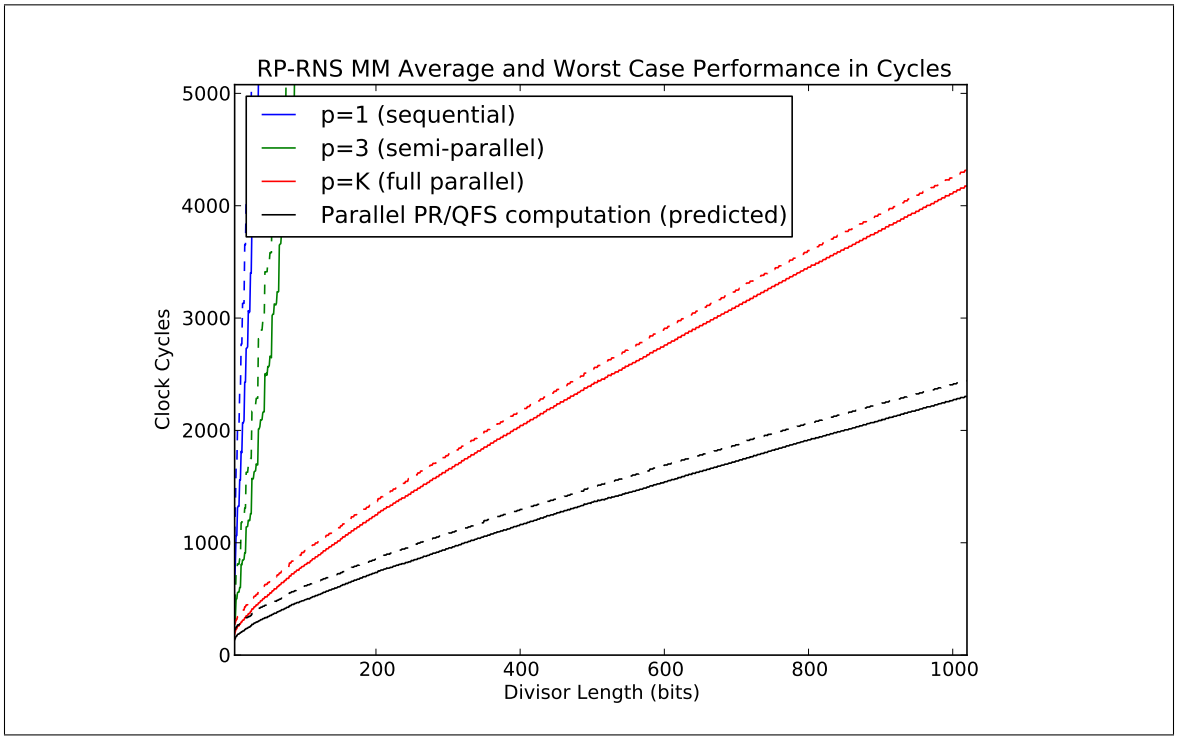

Figure 6.5: Cutout of theoretical cycle counts from Figure 6.4.

around the mean is quite small. The 95% confidence interval around the mean was within ± 3 cycles for small divisors and no more than ± 10 cycles for 1024-bit divisors.

As we explain our result, the reader should keep in mind that the total number of residue channel modular multiplications to implement modulo-D multiplication is $3K$: the initial multiplication, the ρ computation, and the remainder computation.

The first trend worth noting is the quadratic growth of the cycle count for the sequential and semi-parallel runs. This is due to our decision to forego Phatak's assumptions and our suboptimal implementation of the state machine. Since the number of channel modular multiplications is constant, the main contributing factor is the K modular additions. Our fully parallel implementation, despite our design decisions, comes close to the theoretical speed estimates of our implementation (see Figure 6.5).

The second trend worth noting is the variance between the average and worst case decreases rapidly as we increase the parallelism. The two variable computations in our implementation are the residue channel modular multiplications and the modular reduction of the fractional part of the quotient – the former having the most variance. The difference is a result of sequential modular multiplications. In a full parallel implementation, we see this difference becomes negligible and obeys the $\lg N$ trend overall.

6.4 Analysis

In the previous section we presented our data and highlighted the interesting trends. We also identified how some trends were misleading, saving the details for this section. In this section we combine our metrics to reason about our implementation and its limitations.

We shall first address the fourth type of cycle count data (see Figure 6.5). These data do not derive directly from our implementation; they are a theoretical modification of

our implementation using the same timing model. We generated the data using software simulation.

There is a lot of potential parallelism inherent in the QFS algorithm. The reconstruction coefficient and quotient accumulations are parallelizable by adding a second adder to the fraction channel. This is the first modification. Our second modification is to replace the sequential adders with the adder tree Phatak suggests. The fourth type of data represents these theoretical modifications and demonstrates a dramatic performance increase. Even with this modification our unoptimized implementation does not outperform the hardware in Gandino, et al. [16].

Earlier we noted the invalidity of comparing our unoptimized FPGA implementation against the optimized ASICs in the literature. Had our implementation outperformed the implementations in the literature, we would be able to conclude the superiority of RP-RNS modular exponentiation with respect to speed. However, we did not achieve this for the system sizes documented in the literature. There is not enough data to support that other systems will continue to outperform our implementation as the system size increases. Furthermore, it is possible an optimized version of our implementation could outperform the current literature. For these reasons our running time results are inconclusive.

Now we address the other two metrics (i.e. FPGA resource utilization and storage). We alluded to the relation between the storage and the number of utilized logic LUTs. Since we used distributed storage on the FPGA, the synthesizer stored the precomputed tables in the logic LUTs. We reached our FPGA's capacity at a meager 13 channels, which only achieves a 22-bit divisor. We asked whether a high-capacity FPGA would meet our storage and resource utilization requirements.

Using our choice of RNS base, a 4096-bit divisor requires over 1500 moduli. We note this requires nearly 18GB of storage. The Xilinx Virtex-7 (XC7VX1140T) is a high-capacity

FPGA costing nearly $40,000 as of October 2013. It can store at most 10.42 MB of data. Our choice to store the tables on the FPGA renders the system cost prohibitive for us.

Let us assume for now storing the precomputed tables is not a problem. The number of flip-flops scaled linearly with the number of residue channels. The amount of hardware scaled linearly with the number of hardware channels. These growth rates together, however, do not necessarily imply the logic scales at a rate affordable to implement on cheap commodity hardware. So we ran an experiment in which we removed the precomputed tables from the VHDL. We verified our Spartan-3E could support a full parallel implementation using the first 30 prime numbers as our RNS base.

A high-capacity Virtex-7 has 5.5 times the capacity as a Spartan-3E. The Virtex-7 also uses 6-input LUTs instead of the 4-input LUTs used by the Spartan-3E. The modulus size does not grow very fast; the largest modulus for a 8192-bit divisor does not exceed 16-bits. We estimate a high-capacity FPGA could support at least 4 times as many channels (i.e. 120 per chip). This implies the logic of our implementation is scalable even though we could not synthesize systems for divisors in excess of 22-bits.

Chapter 7

Conclusion

We address the following three questions. The first is whether we can engineer RP-RNS modular exponentiation hardware supporting at least a 1024-bit divisor. The second is whether there is a divisor size limit for engineering this system using commodity FPGAs. The third is how the well our hardware performs against the Gandino's work with respect to running time.

7.1 Design Improvements

Despite having refined our design to optimize for FPGA resource utilization, we were unable to synthesize hardware for divisors exceeding 20 bits. Our decision to store the precomputed lookup tables on the FPGA limited our system size and was due to our lack of familiarity with FPGA hardware design. In retrospect, we could have foreseen this obstacle by better understanding the technology and its limitations.

The advantage to storing the tables on the FPGA is the unlimited ability to partition the tables to maximize the parallelism. It is infeasible with today's FPGAs to implement

RP-RNS modular exponentiation as we attempted. As FPGA capacity has only doubled over the last 10 years [43, 44], we do not foresee this problem being overcome.

While our ideas are not novel, we propose two improvements to our implementation:

1. external lookup table storage; and

2. pairing each FPGA with its own dedicated storage element.

By removing the lookup tables from the hardware, we were able to achieve 30 hardware channels. For high-capacity FPGAs made in 2013 (e.g., Virtex-7), up to 120 hardware channels are possible.

The capacities of FPGAs have almost doubled every three years over the last decade [41–43]. We predict a full-parallel 1,000 channel RP-RNS FPGA implementation will become reality within ten years. For now though, the number of channels needed to engineer systems of sufficient size (i.e., 1024-bit and higher) requires multiple FPGAs.

The lookup tables require persistence, high-capacity, and fast random access read. Sufficient technology exists to support these requirements such as a combination of SDRAM for picosecond random-access reads and NAND flash memory for persistence; both offer high capacity storage. For an ASIC composed of multiple FPGAs, it is tempting to use a single storage device to save circuit area and cost. Even with picosecond read access, however, the storage becomes the bottleneck when the system requires hundreds (e.g., for 1024 bits) or thousands (e.g., for 4096 bits) of channels. So we propose each FPGA have its own dedicated storage element or preferably multiple dedicated storage elements.

These two improvements together would enable us to fully implement Phatak's recommendations: parallel table access and per-channel adder trees.

7.2 Reflecting on Performance

To address the first questions related to divisor size, our implementation as-is is incapable of supporting even modest divisors.

Assuming our two improvements were implemented, a single high-capacity FPGA could support a fully parallel 512-bit RP-RNS modular exponentiation hardware implementation. For 1024-bit, a fully parallel hardware implementation would require 3 medium-capacity FPGAs (i.e., approximately 75 channels).

Let us suppose the system needed to fit on only one FPGA. We have two options to address this: resource sharing and RNS base modification.

We could share each hardware channel among multiple residue channels by taking a penalty on execution time. The penalty would scale linearly with the number of residue channels per hardware channel. For example, a 4096-bit system requires over 1,500 and would require around 15 high-capacity FPGAs. We could fit the logic on a single FPGA, but the running time would suffer by around a factor of 15.

Nothing in RP-RNS requires the base consist of the first K prime numbers. We could reduce the number of channels by using a different RNS base. Based on Phatak's analysis, this would entail larger lookup tables and less than optimal execution time. However, the question of whether resource sharing is preferable to modifying the RNS base is an open problem.

Addressing the question of running time now, we verified the performance trend stated in Phatak's theoretical papers in Figure 6.5. Since the literature did not provide a regression for various divisor lengths, we were unable to determine the divisor size beyond which our implementation would outperform systems based on Bajard's RNS-based Montgomery

multiplication including Kawamura. We, however, remain optimistic that a divisor size exists even for our unoptimized implementation due to Phatak's theoretical results.

7.3 Open Problems and Future Work

Our first open problem is regarding RP-RNS as a general-purpose system of computation. Subsequent open problems relate to our implementation of RP-RNS modular exponentiation. Our last open problem is of a theoretical nature.

As of the year 2013, RP-RNS is not yet ready for general-purpose use. Phatak's theoretical work on sign-detection [33] and his current research on general division appear promising. Hardware prototypes such as our implementation are necessary to analyze how these new algorithms perform.

While we studied the modular exponentiation algorithm, we did not focus on the partial reconstruction algorithm specifically. A more focused study on partial reconstruction to include a dedicated hardware implementation is desirable. A follow-on to this study should focus on an implementation of Bajard's RNS-based Montgomery translation using base extension based on the partial reconstruction algorithm.

A direct follow-on to our research includes implementing the improvements we described earlier, fully implementing Phatak's assumptions, and repeating the experiments described in this report.

The last open hardware problem we discuss concerns ASICs. Beyond the modular ALU, our implementation used synthesizer-inferred logic blocks. FPGA vendors protect these intellectual property (IP) blocks and these blocks vary from vendor to vendor. An ASIC requires a complete gate-level specification. Since each hardware channel will be similar in structure, a single specification should be sufficient; increasing the word length of the

hardware channel components should be trivial. Due to the design fixing of ASICs, we recommend this gate-level specification first be implemented using FPGAs with external memory.

An open theoretical problem concerns optimal RP-RNS base selection. It remains unproven whether the first K smallest primes are optimal with respect to execution time and table size. We suggest optimal base selection research address optimality with respect to three constraints at a minimum: execution time, table size, and number of hardware channels.

7.4 Contributions

The sum of our contributions is to guide future investigations that will expand on our work.

We provided a description of our design, architecture, and implementation in Chapter 5, a testing methodology in Chapter 6, and performance data including FPGA utilization, storage requirements, and execution time in Chapter 6. In Appendix B we provide tables of our data. Together these should be sufficient for future researchers to replicate our implementation and findings.

Our reflections in this chapter provide a direct path for another to pick up our work and what our next steps would be. Furthermore, the open problems we described provide other avenues for exploring RP-RNS hardware.

7.5 Final Musings

RNS is an interesting number system due to its ability to perform arithmetic in parallel. RNS-based Montgomery approaches are tailored to address the modular exponentiation

problem. The RP-RNS approach has the potential to address all arithmetic operations and bring RNS to general-purpose computing. The way it accomplishes this is equally interesting. Our hardware implementation confirms this system is on the right track.

FPGAs were sufficient for our research, however, high-capacity FPGAs are still expensive and slow compared to mass-produced ASICs. The development of an efficient RP-RNS ASIC implementing all RP-RNS algorithms we believe will help bring RNS to commodity hardware. It is our conviction that RNS is the future of general-purpose computing and RP-RNS will be the first generation.

Appendix A

Notation

Set	$\mathbf{S}$, $\{\cdot\}$
Approximation of f	$\hat{f}$
a divides b	$a \vert b$
Greatest common divisor	$\gcd(a, b)$
Congruence modulo n	$x \equiv a \bmod n$
Modular reduction by n	$x = a \bmod n$
Integers modulo n	$\mathbf{Z}_n$
Product ring	$\prod_{i=1}^{K} \mathbf{Z}_{m_i} = \mathbf{Z}_{m_1} \times \cdots \times \mathbf{Z}_{m_K}$
RNS base	$\mathbf{M} = m_1, m_2, \ldots, m_K$
RNS representation of z	$(z_1, z_2, \ldots, z_n)$
RNS representation of z in base $\mathbf{A}$	$[z]_A$
Residue of z with respect to modulus m	$[z]_m$

Appendix B

Tables of Performance Metrics

Table B.1: Cycle count for $p = 1$ sequential implementation.

Includes all synthesized hardware sizes and divisors that are powers of 2.		
Divisor Length	Average Number of Cycles	95% Confidence Interval
4	639.190	$[631.182, 647.198]$
5	819.010	$[810.754, 827.266]$
6	1064.820	$[1055.931, 1073.709]$
7	1063.820	$[1054.598, 1073.042]$
8	1310.480	$[1298.715, 1322.245]$
9	1326.830	$[1315.533, 1338.127]$
10	1324.280	$[1313.401, 1335.159]$
11	1559.190	$[1547.018, 1571.362]$
12	1559.000	$[1545.675, 1572.325]$
13	1813.800	$[1799.194, 1828.406]$
14	1801.970	$[1787.470, 1816.470]$

Continued on next page

TableB.1 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
15	2068.660	[2052.401, 2084.919]
16	2067.340	[2050.513, 2084.167]
17	2087.780	[2072.062, 2103.498]
18	2428.590	[2412.616, 2444.564]
19	2429.690	[2411.685, 2447.695]
20	2730.390	[2714.142, 2746.638]
32	4233.980	[4209.875, 4258.085]
40	5550.050	[5515.338, 5584.762]
60	8455.450	[8424.423, 8486.477]
64	9037.750	[9001.851, 9073.649]
80	12422.870	[12370.669, 12475.071]
100	15718.040	[15639.040, 15797.040]
120	20217.220	[20152.640, 20281.800]
128	21981.100	[21929.011, 22033.189]
140	24649.760	[24589.562, 24709.958]
160	29358.760	[29285.344, 29432.176]
180	35052.080	[34972.065, 35132.095]
200	40770.790	[40652.050, 40889.530]
220	46603.170	[46410.017, 46796.323]
240	51315.930	[51155.887, 51475.973]
256	56571.770	[56450.916, 56692.624]
260	58017.860	[57894.065, 58141.655]

Continued on next page

TableB.1 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
280	65149.890	$[65026.520, 65273.260]$
300	71245.470	$[71131.738, 71359.202]$
320	79227.060	$[79098.036, 79356.084]$
340	85934.070	$[85791.229, 86076.911]$
360	94896.830	$[94732.110, 95061.550]$
380	102084.850	$[101894.682, 102275.018]$
400	111510.160	$[111378.010, 111642.310]$
420	119399.600	$[119234.492, 119564.708]$
440	127534.790	$[127369.762, 127699.818]$
460	135895.820	$[135711.434, 136080.206]$
480	146582.680	$[146315.928, 146849.432]$
500	155740.360	$[155378.202, 156102.518]$
512	159790.670	$[159372.554, 160208.786]$
520	163931.400	$[163537.577, 164325.223]$
540	172598.410	$[172239.324, 172957.496]$
560	182214.810	$[181898.443, 182531.177]$
580	194462.650	$[194207.467, 194717.833]$
600	204748.890	$[204529.291, 204968.489]$
620	215339.490	$[215098.020, 215580.960]$
640	226539.260	$[226287.954, 226790.566]$
660	237213.770	$[236973.261, 237454.279]$
680	248500.110	$[248273.906, 248726.314]$

Continued on next page

TableB.1 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
700	260140.350	$[259894.492, 260386.208]$
720	272681.610	$[272434.274, 272928.946]$
740	285208.240	$[284959.446, 285457.034]$
760	297631.400	$[297341.765, 297921.035]$
780	310427.670	$[310146.342, 310708.998]$
800	323218.760	$[322855.589, 323581.931]$
820	335596.870	$[335247.574, 335946.166]$
840	349083.290	$[348716.546, 349450.034]$
860	359429.250	$[359014.147, 359844.353]$
880	372424.310	$[372113.527, 372735.093]$
900	387142.520	$[386790.804, 387494.236]$
920	401497.340	$[401154.530, 401840.150]$
940	415987.720	$[415653.802, 416321.638]$
960	431409.630	$[431067.762, 431751.498]$
980	446333.160	$[445972.914, 446693.406]$
1000	457751.840	$[457391.392, 458112.288]$
1020	473487.370	$[473119.886, 473854.854]$
1023	477337.840	$[476972.226, 477703.454]$

End of table

Table B.2: Cycle count for $p = 3$ semi-parallel implementation.

Includes all synthesized hardware sizes and divisors that are powers of 2.		
Divisor Length	Average Number of Cycles	95% Confidence Interval
4	293.250	$[289.987, 296.513]$
5	329.890	$[326.975, 332.805]$
6	500.740	$[496.725, 504.755]$
7	500.320	$[496.268, 504.372]$
8	555.600	$[551.008, 560.192]$
9	561.590	$[557.229, 565.951]$
10	560.010	$[555.827, 564.193]$
11	601.240	$[597.139, 605.341]$
12	601.810	$[597.298, 606.322]$
13	808.050	$[802.169, 813.931]$
14	802.820	$[797.032, 808.608]$
15	846.650	$[840.659, 852.641]$
16	846.540	$[840.428, 852.652]$
17	853.390	$[847.647, 859.133]$
18	910.670	$[905.287, 916.053]$
19	910.340	$[904.241, 916.439]$
20	1140.640	$[1134.388, 1146.892]$
32	1636.280	$[1627.797, 1644.763]$
40	2094.750	$[2082.419, 2107.081]$
60	3121.770	$[3111.247, 3132.293]$

Continued on next page

TableB.2 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
64	3230.240	$[3218.257, 3242.223]$
80	4514.810	$[4497.036, 4532.584]$
100	5795.570	$[5767.905, 5823.235]$
120	7367.500	$[7344.915, 7390.085]$
128	7669.910	$[7652.441, 7687.379]$
140	8609.430	$[8589.379, 8629.481]$
160	10405.900	$[10380.991, 10430.809]$
180	12515.770	$[12488.033, 12543.507]$
200	14069.970	$[14030.521, 14109.419]$
220	16278.740	$[16213.080, 16344.400]$
240	17653.180	$[17599.371, 17706.989]$
256	19920.740	$[19879.154, 19962.326]$
260	20175.200	$[20133.470, 20216.930]$
280	22911.150	$[22868.969, 22953.331]$
300	24732.060	$[24693.835, 24770.285]$
320	27732.780	$[27688.683, 27776.877]$
340	29726.250	$[29678.167, 29774.333]$
360	33003.280	$[32947.244, 33059.316]$
380	35124.790	$[35060.639, 35188.941]$
400	38671.590	$[38627.086, 38716.094]$
420	41008.740	$[40953.257, 41064.223]$
440	43417.460	$[43362.437, 43472.483]$

Continued on next page

TableB.2 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
460	46952.170	[46889.345, 47014.995]
480	49819.820	[49730.170, 49909.470]
500	53699.930	[53577.261, 53822.599]
512	54259.620	[54120.272, 54398.968]
520	56060.790	[55928.974, 56192.606]
540	58587.670	[58467.958, 58707.382]
560	62651.430	[62544.266, 62758.594]
580	66040.490	[65955.324, 66125.656]
600	70390.190	[70315.865, 70464.515]
620	73505.030	[73423.819, 73586.241]
640	76832.160	[76748.168, 76916.152]
660	81375.670	[81294.853, 81456.487]
680	84701.000	[84624.614, 84777.386]
700	88129.800	[88047.354, 88212.246]
720	93249.770	[93166.227, 93333.313]
740	96941.370	[96857.786, 97024.954]
760	100615.130	[100518.854, 100711.406]
780	105992.990	[105897.749, 106088.231]
800	109764.620	[109642.479, 109886.761]
820	113355.640	[113238.656, 113472.624]
840	119037.310	[118913.514, 119161.106]
860	122514.410	[122374.053, 122654.767]

Continued on next page

TableB.2 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
880	126305.560	$[126201.075, 126410.045]$
900	130659.200	$[130542.229, 130776.171]$
920	136706.840	$[136591.899, 136821.781]$
940	140960.570	$[140848.674, 141072.466]$
960	145493.120	$[145379.492, 145606.748]$
980	151799.460	$[151678.154, 151920.766]$
1000	155637.900	$[155516.090, 155759.710]$
1020	160283.530	$[160160.732, 160406.328]$
1023	160884.360	$[160762.139, 161006.581]$

End of table

Table B.3: Cycle count for $p = K$ full-parallel implementation.

Includes all synthesized hardware sizes and divisors that are powers of 2.		
Divisor Length	Average Number of Cycles	95% Confidence Interval
4	185.850	$[184.152, 187.548]$
5	206.910	$[205.353, 208.467]$
6	233.430	$[231.945, 234.915]$
7	233.520	$[232.049, 234.991]$
8	257.600	$[255.979, 259.221]$
9	259.790	$[258.264, 261.316]$
10	259.170	$[257.685, 260.655]$
11	277.390	$[276.050, 278.730]$
12	277.400	$[275.853, 278.947]$
13	294.820	$[293.339, 296.301]$
14	293.660	$[292.214, 295.106]$
15	311.390	$[309.907, 312.873]$
16	311.270	$[309.722, 312.818]$
17	312.830	$[311.356, 314.304]$
18	333.440	$[332.039, 334.841]$
19	333.210	$[331.597, 334.823]$
20	350.380	$[349.097, 351.663]$
32	431.380	$[429.930, 432.830]$
40	484.450	$[482.633, 486.267]$
60	589.590	$[588.386, 590.794]$

Continued on next page

TableB.3 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
64	610.200	$[608.800, 611.600]$
80	705.210	$[703.590, 706.830]$
100	789.580	$[787.424, 791.736]$
120	892.120	$[890.553, 893.687]$
128	928.030	$[926.854, 929.206]$
140	981.430	$[980.138, 982.722]$
160	1068.280	$[1066.867, 1069.693]$
180	1159.690	$[1158.267, 1161.113]$
200	1248.920	$[1247.061, 1250.779]$
220	1335.200	$[1332.316, 1338.084]$
240	1398.620	$[1396.366, 1400.874]$
256	1467.840	$[1466.163, 1469.517]$
260	1485.670	$[1484.007, 1487.333]$
280	1572.340	$[1570.786, 1573.894]$
300	1643.170	$[1641.835, 1644.505]$
320	1730.410	$[1728.948, 1731.872]$
340	1801.140	$[1799.542, 1802.738]$
360	1886.310	$[1884.526, 1888.094]$
380	1954.180	$[1952.288, 1956.072]$
400	2043.650	$[2042.435, 2044.865]$
420	2113.090	$[2111.614, 2114.566]$
440	2184.170	$[2182.731, 2185.609]$

Continued on next page

TableB.3 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
460	2252.490	$[2250.887, 2254.093]$
480	2338.880	$[2336.764, 2340.996]$
500	2411.260	$[2408.506, 2414.014]$
512	2442.260	$[2439.024, 2445.496]$
520	2472.010	$[2469.084, 2474.936]$
540	2535.700	$[2533.069, 2538.331]$
560	2604.040	$[2601.799, 2606.281]$
580	2690.780	$[2689.032, 2692.528]$
600	2759.540	$[2758.086, 2760.994]$
620	2827.660	$[2826.096, 2829.224]$
640	2900.300	$[2898.713, 2901.887]$
660	2967.060	$[2965.541, 2968.579]$
680	3035.120	$[3033.706, 3036.534]$
700	3104.780	$[3103.308, 3106.252]$
720	3172.450	$[3171.002, 3173.898]$
740	3244.600	$[3243.235, 3245.965]$
760	3315.420	$[3313.782, 3317.058]$
780	3385.880	$[3384.299, 3387.461]$
800	3453.250	$[3451.334, 3455.166]$
820	3519.420	$[3517.585, 3521.255]$
840	3588.980	$[3587.114, 3590.846]$
860	3640.870	$[3638.785, 3642.955]$

Continued on next page

TableB.3 – continued from previous page

Divisor Length	Average Number of Cycles	95% Confidence Interval
880	3706.460	$[3704.885, 3708.035]$
900	3778.060	$[3776.368, 3779.752]$
920	3847.950	$[3846.272, 3849.628]$
940	3915.920	$[3914.394, 3917.446]$
960	3986.880	$[3985.270, 3988.490]$
980	4054.030	$[4052.375, 4055.685]$
1000	4106.250	$[4104.665, 4107.835]$
1020	4176.220	$[4174.571, 4177.869]$
1023	4191.880	$[4190.251, 4193.509]$

End of table

Bibliography

[1] Maple. http://www.maplesoft.com/.

[2] Python programming language – official website. http://www.python.org/.

[3] Sage: Open source mathematics software. http://www.sagemath.org/.

[4] ANANDA MOHAN, P. Novel design for binary to RNS converts. In *Circuits and Systems, 1994, ISCAS '94., 1994 IEEE International Symposium on* (1994), vol. 2, pp. 357–360 vol. 2.

[5] BAJARD, J.-C., DIDIER, L.-S., AND KORNERUP, P. An IWS Montgomery modular multiplication algorithm. In *Proceedings of the 13th Symposium on Computer Arithmetic (ARITH '97)* (Washington, DC, USA, 1997), ARITH '97, IEEE Computer Society, pp. 234–.

[6] BAJARD, J.-C., DIDIER, L.-S., AND KORNERUP, P. An RNS Montgomery modular multiplication algorithm. *IEEE Trans. Comput. 47*, 7 (July 1998), 766–776.

[7] BAJARD, J.-C., DIDIER, L.-S., AND KORNERUP, P. Modular multiplication and base extensions in residue number systems. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (Washington, DC, USA, 2001), ARITH '01, IEEE Computer Society, pp. 59–.

[8] BAJARD, J.-C., AND IMBERT, L. A full RNS implementation of RSA. *IEEE Trans. Comput. 53*, 6 (June 2004), 769–774.

[9] BEACHY, J., AND BLAIR, W. *Abstract Algebra: Third Edition*. Waveland Press, 2006.

[10] BRENT, R. P., AND KUNG, H. T. A regular layout for parallel adders. *Computers, IEEE Transactions on C-31*, 3 (1982), 260–264.

[11] BUCK, I. GPU computing: Programming a massively parallel processor. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on* (2007), pp. 17–17.

[12] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[13] DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *IEEE Trans. Inf. Theor. 22*, 6 (Sept. 2006), 644–654.

[14] DUMMIT, D., AND FOOTE, R. *Abstract Algebra*. John Wiley & Sons, 2004.

[15] EL GAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology* (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 10–18.

[16] GANDINO, F., LAMBERTI, F., PARAVATI, G., BAJARD, J., AND MONTUSCHI, P. An algorithmic and architectural study on Montgomery exponentiation in RNS. *Computers, IEEE Transactions on 61*, 8 (2012), 1071–1083.

[17] GARNER, H. L. The residue number system. In *Papers presented at the March 3-5, 1959, Western Joint Computer Conference* (New York, NY, USA, 1959), IRE-AIEE-ACM '59 (Western), ACM, pp. 146–153.

[18] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. Introduction to the Cell multiprocessor. *IBM J. Res. Dev. 49*, 4/5 (July 2005), 589–604.

[19] KAWAMURA, S., KOIKE, M., SANO, F., AND SHIMBO, A. Cox-rower architecture for fast parallel Montgomery multiplication. In *Proceedings of the 19th international conference on Theory and application of cryptographic techniques* (Berlin, Heidelberg, 2000), EUROCRYPT'00, Springer-Verlag, pp. 523–538.

[20] KOGGE, P. M., AND STONE, H. S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on C-22*, 8 (1973), 786–793.

[21] KOREN, I. *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[22] MAXFIELD, C. *The Design Warrior's Guide to FPGAs*. Academic Press, Inc., Orlando, FL, USA, 2004.

[23] MENEZES, A. J., VANSTONE, S. A., AND OORSCHOT, P. C. V. *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[24] MOHAN, P. *Residue Number Systems: Algorithms and Architectures*. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 2002.

[25] MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of Computation 44* (1985), 519–521.

[26] NICHOLSON, W. K. *Introduction to Abstract Algebra*, 4th ed. John Wiley & Sons, 2012.

[27] NOZAKI, H., MOTOYAMA, M., SHIMBO, A., AND KAWAMURA, S.-I. Implementation of RSA algorithm based on RNS Montgomery multiplication. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems* (London, UK, UK, 2001), CHES '01, Springer-Verlag, pp. 364–376.

[28] OMONDI, A., AND PREMKUMAR, B. *Residue Number Systems: Theory and Implementation*. Advances in computer science and engineering: Texts. Imperial College Press, 2007.

[29] OWENS, J. GPU architecture overview. In *ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), SIGGRAPH '07, ACM.

[30] PHATAK, D. S. RNS-ARDSP: A novel, fast residue number system using approximate rational domain scaled precomputations, part i RP-PR: Reduced precision partial reconstruction and its application to fast RNS base extensions and/or base-changes. Tech. Rep. TR-CS-10-01, University of Maryland, Baltimore County, Jan. 2010. UMBC Technical Report TR-CS-10-01. Revised: November, 2013.

[31] PHATAK, D. S. RNS-ARDSP: A novel, fast residue number system using approximate rational domain scaled precomputations, part ii: SODIS (sign and overflow detection by interval separation) algorithm, the quotient first scaling/division-by-a-constant algorithm and their applications to expedite modular reduction and modular exponentiation. Tech. Rep. TR-CS-10-02, University of Maryland, Baltimore County, Jan. 2010. UMBC Technical Report TR-CS-10-02. Revised: November, 2013.

[32] PHATAK, D. S. RNS-ARDSP: A novel, fast residue number system using approximate rational domain scaled precomputations, part iii: Fast convergence division algorithms and a broad overview of other potential enhancements to RNS and their applications. Tech. Rep. TR-CS-10-03, University of Maryland, Baltimore County, Jan. 2010. UMBC Technical Report TR-CS-10-03. Revised: November, 2013.

[33] PHATAK, D. S. Residue number systems methods and apparatuses. Patent Application, Sept. 2011. US 2011/0231465 A1.

[34] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.

[35] RODRÍGUEZ-HENRÍQUEZ, F., SAQIB, N. A., DÍAZ-PÈREZ, A., AND KOC, C. K. *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[36] SHENOY, A., AND KUMARESAN, R. Residue to binary conversion for RNS arithmetic using only modular look-up tables. *Circuits and Systems, IEEE Transactions on 35*, 9 (1988), 1158–1162.

[37] SHENOY, A., AND KUMARESAN, R. Fast base extension using a redundant modulus in RNS. *Computers, IEEE Transactions on 38*, 2 (1989), 292–297.

[38] SZABÓ, N., AND TANAKA, R. *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill series in information processing and computers. McGraw-Hill, 1967.

[39] VAN ROSSUM, G. The python language reference: Release 2.6.4, 2009.

[40] WALLACE, C. S. A suggestion for a fast multiplier. *Electronic Computers, IEEE Transactions on EC-13*, 1 (1964), 14–17.

[41] XILINX INC. *Virtex-5 Family Overview (DS100)*, v5.0 ed., Feb. 2009.

[42] XILINX INC. *Virtex-6 Family Overview (DS150)*, v2.4 ed., Jan. 2012.

[43] XILINX INC. *7 Series FPGAs Overview (DS180)*, v1.14 ed., July 2013.

[44] XILINX INC. *Spartan-3E FPGA Family Data Sheet (DS312)*, v4.1 ed., July 2013.