# Reside Number Systems Methods and Apparatuses

**Inventer(s) : Dhananjay S. Phatak**

**phatak@umbc.edu**

Filed with the USPTO on 9th March 2011

## CROSS REFERENCE TO RELATED APPLICATIONS

This patent claims the benefit of priority from U.S. Provisional Patent Application Serial No. 61/311815, entitled "Ultrafast Residue Number System Using Intermediate Fractional Approximations That Are Rounded Directionally, Scaled and Computed," filed on March 9, 2010, incorporated herein by reference in its entirety.

## FIELD OF THE INVENTION

The invention relates to performing residue number system calculations, and in particular, to reduced complexity algorithms and hardware designs for performing residue number system calculations. This invention is in the field of the Residue Number Systems (RNS) and their applications.

## § 1 BACKGROUND OF THE INVENTION

The RNS uses a set "$\mathbb{M}$" of pair-wise co-prime positive integers called as the "moduli"

$\mathbb{M} \ \{m_1, m_2, \cdots, m_r, \cdots, m_K\}$ where $m_r > 1 \, \forall \, r \in 1, K$ and $\gcd m_i, m_j \ 1$ for $i \not\jmath$ (B-1)

Note that $\mid \mathbb{M} \mid$ the number of moduli $K$ (also referred to as the number of "channels" in the literature)
We use the term "component-modulus" to refer to any single individual modulus (ex: $m_r$) in the set $\mathbb{M}$.
For the sake of convenience, we also impose an additional ordering constraint: $m_i < m_j$ if $i < j$
Total-modulus $\mathcal{M} \triangleq m_1 \times m_2 \times \ldots \times m_K$. Typically $\mathcal{M} >> K$ (B-2)
Any integer $Z \in 0, \mathcal{M}-1$, can be uniquely represented by the ordered-touple (or vector) of residues:
$\forall \, Z \in 0, \mathcal{M}-1; \quad Z \equiv \overline{\mathbb{Z}} \ z_1, z_2, \cdots, z_K$ where, $z_r \ Z \bmod m_r, \quad r \ 1, \cdots, K$ (B-3)
Conversion from residues back to an integer is done using the **"Chinese Remainder Theorem"** as follows:

$Z \ Z_T \mod \mathcal{M}$ (B-4) where $Z_T \left( \sum_{r1}^{K} M_r \cdot \rho_r \right)$ (B-5)

and $\rho_r \ z_r \cdot w_r \mod m_r, \ r \ 1, \cdots, K$ where (B-6)

outer-weights $M_i \dfrac{\mathcal{M}}{m_i}$; & inner-weights $w_i \left( \dfrac{1}{M_i} \mod m_i \right)$ are constants for a given $\mathbb{M}$ (B-7)

The Residue Number Systems (abbreviated "RNS") have been around for while [1]. The underlying Residue Domain representation (or simply Residue Representation, abbreviated "RR") has some unique attributes (explained below) that make it attractive for signal processing. It is therefore not surprising that the early work in this area was contributed by the signal-processing community.

Thereafter, from the late 1970s through the mid 1980s, the field of cryptology was revolutionized by the invention of the 3 most fundamental and widely used cryptology algorithms, viz., Diffie-Hellman, RSA, and

Elliptic-curves. In the beginning, the aforementioned cryptology algorithms were not easy to implement in hardware. However, as semiconductor device sizes kept on shrinking, the hardware that could be integrated on a single chip kept on becoming larger as well as faster. As a result, today (in 2011) it possible to easily realize the cryptographic (abbreviated "crypto") algorithms in hardware. The word-lengths used in crypto methods are substantially larger as compared with wordlengths required by other applications; typical crypto word lengths today are at least 256 bits or higher. It turns out that the same attributes of the Residue Number Systems that are attractive for signal processing are also beneficial when implementing cryptographic algorithms at long word-lengths. Consequently the cryptology and computer-arithmetic communities also started researching RNS. This coincidental convergence of goals (to research and improve RNS, which is now shared by the signal processing, cryptology as well as computational/computer arithmetic communities) has in-turn led to a resurgence of interest as well as activity in the RNS [2].

## § 1.1 Advantages of the Residue Domain Representation

The main advantage of the Residue Number (RN) system is that in the Residue-Domain (RD ), the operations $\{\pm, \times, ?\}$ can be implemented on a per-component/channel basis, wherein the processing required in any single channel is completely independent of the processing required in any other channel. In other words, these operations can be implemented fully in parallel on a per-channel basis as follows:

$$\overline{\mathtt{Z}}\ \overline{\mathtt{X}}\ \boxed{\ddagger}\ \overline{\mathtt{Y}} \Rightarrow z_r\ \ x_r \boxed{\ddagger} y_r \mod m_r,\ \ r\ \ 1, \cdots, K \quad \text{where} \quad \boxed{\ddagger} \in \{\pm, \times, ?\} \tag{1}$$

Note that equality of two numbers can be checked by comparing their residues which can be done in parallel in all channels. In other words, in the RD , the most fundamental operations viz., addition/subtraction, equality check AND Multiplication can all be performed in parallel in each channel independently of any other channel(s). This independence of channels implies that each of the above operations can be implemented with $On$ computing effort (operations/steps), where

$$\lceil \lg \mathcal{M} \rceil\ \ n \tag{2}$$

i.e., $n$ is the number of bits required to represent the total-modulus $\mathcal{M}$ or the overall range of the RNS.

In contrast, in the regular integer domain, a multiplication is a convolution of the digit-strings representing the two numbers being multiplied. A convolution is substantially more expensive than add/subtract operations (Addition/Subtraction fundamentally require $On$ operations and can be implemented in $O \lg n$ delay using the "carry-look-ahead" method and its variants. Naive paper-and-pencil multiplication, requires $On^2$ operations. Asymptotically fastest multiply methods use transforms such as floating point FFT (Fast Fourier Transform) or number-theoretic transforms to convert the convolution in the original domain into a point-wise product in the transform domain, so that the number of operations required turns out to be $\approx On \lg n$; for further details, please refer to [3]).

Thus, performing the multiplications in the RD is substantially faster as well as cheaper (in terms of interconnect length and therefore h/w area as well as power consumption). Consequently, wherever multiplication is heavily used, adopting the RR can lead to smaller and faster realizations that also consume less power. For example:
(i) Filtering is heavily used in signal processing. Most of the effort in filtering is in the repeated multiply and add operations. It is therefore not surprising that the first practical use of the RNS was in synthesizing fast filters for signal processing.
(2) Multiplication (note that squaring is a special case of multiplication) also gets used heavily in long-wordlength cryptology algorithms. Therefore RD implementations of cryptological algorithms are also

smaller, faster and consume lower power.

## § 1.2 Disadvantages of the Residue Domain Representation

Together with the advantages, also come some of the disadvantages of the residue domain: when compared to the "easy operations" above, several fundamental operations are relatively a lot more difficult to realize in the RD [1, 4–6]:

1. Reconstruction or conversion back to a weighted, non redundant positional representation (ex, binary or decimal or the "mixed-radix" representation [4])
2. Base extension or change.
3. Sign and overflow detection or equivalently, a magnitude-comparison.
4. Scaling or division by a constant, wherein, the divisor is known ahead of time (such as the Modulus in the RSA or Diffie-Hellman algorithms)
5. Division by an arbitrary divisor whose value is dynamic, i.e., available only at run-time.

*Reconstruction in the regular format by directly using the CRT* turns out to be a slow operation. Note that a straightforward/brute-force application of the CRT entails directly implementing Equation (B-4). Accordingly, $Z_T$ is fully evaluated first, and then a division by the modulus $\mathcal{M}$ is carried out to retrieve the remainder ($Z$). For long word-lengths (ex, in cryptography applications) the final division by $\mathcal{M}$ is unacceptably slow and inefficient.

*Re-construction by evaluating the mixed-radix representation* takes advantage of the "mixed-radix" representation associated with every residue-domain-representation [4], wherein a number is represented by an ordered set of digits. The value of the number is a weighted sum where the weights are positional (just like the weights of a normal single radix decimal or binary representation). As a result, a digit-by-digit comparison starting with the most-significant-digit is feasible. However, to the best of our knowledge it takes $OK^2$ sequential operations (albeit on small sized operands of about the same size as the component-moduli $m_i$) in the residue-domain. The inherently sequential nature of this method makes it slow.

Moreover, at a first glance, it appears that for magnitude-comparison and division, the operands need to be fully reconstructed in the form of a unique digit string representing an integer either in the regular or the mixed-radix-format.

## § 1.3 Related Prior Art

### § 1·3·1 Base-extension or change

In a sign-magnitude representation or a radix-complement (such as the two's complement) representation, a 32 bit integer can be easily extended into a 64 bit value. The corresponding operation in the RNS is considerably more involved. Related Prior work in this area falls under two categories, each is briefly explained next.

#### § 1.3·1·A deploying a redundant modulus

Shenoy and Kumarersan [7] start by re-expressing the CRT in a slightly different form:
$Z \;\; Z_T - \alpha \cdot \mathcal{M}$ where $0 \leq \alpha \leq K-1$
In the above equation $\alpha \equiv \mathcal{R}_C$ is the only unknown. It is clear that knowledge of ($Z \mod m_e$), i.e., the

residue/remainder of $Z$, w.r.t. one extra/redundant modulus $m_e$ is sufficient to determine the value of $\mathcal{R}_C$ . They assume the availability of such an extra residue, which lets them evaluate $\alpha \equiv \mathcal{R}_C$.

This base extension method has been widely adopted in the literature. For example, Algorithms for modular multiplication developed by Bajard et. al. [8, 9] perform their computations in two independent RNS systems and change base from one to the other using the shenoy-kumaresan method. This is done so as to avoid a full reconstruction at intermediate steps. As a result, they end up requiring a base-conversion in each step and consequently, their algorithm requires $OK$ units of delay when $OK$ dedicated processing elements are available (where $K$ = the total number of moduli or channels in the RNS system).

#### § 1.3·1·B Iterative determination of $\mathcal{R}_C$

Another base-extension algorithm related to our work is described in [10–13]. They show a method to evaluate an approximate estimate $\widehat{\mathcal{R}_C}$ in a recursive, bit-by-bit (i.e., one bit-at-a-time) manner and then derive conditions under which the approximation is error-free. This method is at the heart of their base-extension algorithm.
The recursive structure of this method makes it relatively slower and cumbersome.

The idea of using the "fractional-representation" of CRT has been around for a while. For instance,
Vu [14,15] proposed using a Fractional interpretation of the CRT in the mid 1980s. However, he ends up using a very high (actually the FULL) precision: $\lceil \lg K \cdot \mathcal{M} \rceil$ bits (see equations (13) and (14) in reference [15]).

### § 1.3.2 Sign detection and magnitude comparison

In the RNS, the total range is divided into positive and negative intervals of the same length (to the extent possible). For example, if the set of RNS moduli is $\mathbb{M}\ \{2,3,5,7\}$ then $\mathcal{M}\ 210$ and the overall range of the RNS is $[-105, 104]$, where,
the numbers 1 through 104 represent ve numbers, and
the numbers 105 thru 209 represent $-$ve numbers from $-105$ to $-1$, respectively.
In general, all negative values in the range $[-\mathcal{M}-1, -1]$ satisfy the relation

$$-a \mod \mathcal{M} \equiv \mathcal{M} - a \tag{3}$$

Sign detection in the RNS is not straightforward, rather, it has been known to be relatively difficult to realize in the Residue Domain.
Likewise, comparison of magnitudes of two numbers represented as residue touples is also not straightforward (independent of whether or not negative numbers are included in the representation). For instance, with the same simple moduli set $\mathbb{M}\ \{2,3,5,7\}$ above, note that
$19 \equiv 1,1,4,5$ and $99 \equiv 1,0,4,1$ while $79 \equiv 1,1,4,1$ and the negative number $-101 \equiv 1,1,4,4$.
In other words, the touples of remainders corresponding to ve and $-$ ve numbers cannot be easily distinguished.

*Prior Work on Sign Detection and Magnitude Comparison*

Sign detection operation has been known to be relatively difficult to realize in the RNS for a while (early works date back to 1960's, for example [1, 16]). Recent works related to Sign detection in RNS have tended to focus

on using moduli having special forms [17, 18], which limits their applicability. The idea of "core-functions" was introduced in [19] in the context of coding-theory. RNS sign-detection algorithms based on idea of using "core-functions" have been published [20–22]. However, these methods are unnecessarily complicated and appear to be useful only with moduli with special properties [22], limiting their applicability.

Lu and Chiang [23, 24] introduced a method to use the **least significant bit (`lsb` )** to keep track of the sign. However, tracking the **`lsb`** of arbitrary (potentially all possible) numbers is not an easy task. In their quest to keep track of the **`lsb`** , Lu and Chiang first proposed an exhaustive method in their first publication [23]; which turns out to be infeasible for all but small toy examples because the size of their look-up table was the same as the total range $\mathcal{M}$. In the follow up publication, they abandoned the exhaustive look-up approach [24] and ended up unnecessarily using the full precision, just as Vu does in his work [15].

### § 1·3·3 Scaling or division by a constant

In general, "scaling" includes both multiplication as well as division by a fixed constant, (viz., the scaling factor $S_f$). Early versions of signal processors often deployed a fixed-point format which necessitated scaling to cover a wider dynamic range of input values. Consequently, scaling has been heavily used in signal processing. It is therefore not surprising that the early work in realizing the scaling operation in the residue-domain comes from the signal-processing community [25, 26].
Shenoy and Kumaresan [27, 28] introduced a scaling method that works only if the constant divisor has the special form

$$\begin{aligned} &D \;\; m_{d_1} \cdot m_{d_2} \cdots m_{d_s} \quad \text{wherein} \quad s < K \;\; \text{and} \\ &\{m_{d_1}, m_{d_s} \cdots m_{d_s}\} \subset \{m_1, m_2, \cdots, m_k\} \;\; \mathbb{M} \quad \text{the set of RNS moduli} \end{aligned} \tag{4}$$

i.e., the divisor is a factor of the overall modulus $M$. This restriction renders their method inapplicable in most cryptographic algorithms; because the modulus (aka, the constant divisor) $N$ is either a large prime number (as in elliptic curve methods) or a product of two large primes numbers (as in RSA). In either case, it does not share a factor with the total modulus $\mathcal{M}$.

All methods and apparata for scaling in the RNS that have been published thus far [21, 29–31]; including more recent ones [31–33] are either limited to special moduli or are more involved than necessary because they all attempt to estimate the remainder first, subtract it off and then arrive at the quotient, which is the quantity of interest in scaling. Consequently, none of the methods or apparata are even remotely similar to the new algorithm that I have invented for RNS division by a constant.

## § 2 SUMMARY OF THE INVENTION

The following presents a simplified summary in order to provide a basic understanding of some aspects of the claimed subject matter. This summary is not an extensive overview, and is not intended to identify key or critical elements, or to delineate any scope of the disclosure or claimed subject matter. The sole purpose of the subject summary is to present some concepts in a simplified form as a prelude to the more detailed description that is presented later. In one exemplary aspect, a method for performing reconstruction using a residue number system is disclosed. A set of moduli is selected. A reconstruction coefficient is estimated based on the selected set of moduli. A reconstruction operation is performed using the reconstruction coefficient. In another exemplary aspect, an apparatus for performing reconstruction using a residue number system includes means for selecting a set of moduli, means for estimating a reconstruction coefficient based on the selected set of moduli and means for performing a reconstruction operation using the reconstruction coefficient. In yet

another exemplary aspect, a computer program product comprising a non-volatile, computer-readable medium, storing computer-executable instructions for performing reconstruction using a residue number system, the instructions comprising code for selecting a set of moduli, estimating a reconstruction coefficient based on the selected set of moduli and performing a reconstruction operation using the reconstruction coefficient is disclosed. In yet another exemplary aspect, a method for performing division using a residue number system comprises selecting a set of moduli, determining a reconstruction coefficient and determining a quotient using an exhaustive pre-computation and a look-up strategy that covers all possible inputs. In yet another exemplary aspect, a method of computing a modular exponentiation in a residue number system includes iterating, without converting to a regular integer representation, by performing modular multiplications and modular squaring and computing the modular exponentiation as a result of the iterations.

## § 3 BRIEF DESCRIPTION OF DRAWINGS

Figure 1 : shows summation of fraction estimates (obtained via look-up-tables) to estimate the **Reconstruction Coefficient**.

Figure 2 : is a flow chart for the Reduced Precision Partial Reconstruction ( **"RPPR"** ) algorithm.

Figure 3 : is a schematic block diagram of a generic architecture to implement the **RPPR** algorithm.

Figure 4 : illustrates conventional method of incorporating negative integers in the RNS.

Figure 5 : illustrates sign and Overflow Detection by Interval Separation ( **SODIS** ).

Figure 6 : Flow chart for the Quotient First Scaling ( **QFS** ) algorithm.

Figure 7 : is a schematic timing diagram for the **QFS** algorithm.

Figure 8 : is a flow chart for the modular exponentiation algorithm.

Figure 9 : is a flow chart representation of a process of performing reconstruction using a residue number system.

Figure 10 : is a block diagram representation of a portion of an apparatus for performing reconstruction using a residue number system.

Figure 11 : is a flow chart representation of a process of performing division using a residue number system.

Figure 12 : is a block diagram representation of a portion of an apparatus for performing division using a residue number system.

Figure 13 : is a flow chart representation of a process of computing a modular exponentiation using a residue number system.

Figure 14 : is a block diagram representation of a portion of an apparatus for computing a modular exponentiation using a residue number system.

## § 4 DETAILED DESCRIPTIONS

In this section, first, I explain my moduli-selection method. After that, each new algorithm illustrated in detail. Since the **"RPPR"** algorithm is used in all others, it has been explained in more detail than other algorithms.

### Notations used in this document

**Notations-1 Math Functions, symbols**

The symbol $\equiv$ means "equivalent-to"; whereas the symbol $\triangleq$ means "is defined as"
LHS $\triangleq$ **L**eft **H**and **S**ide of a relation; RHS $\triangleq$ the right-hand-side
$a$ **mod** $b$ $\triangleq$ `the remainder when integer` $a$ `is divided by integer` $b$.
**ulp** $\equiv$ wight or value a **u**nit or a "1" in the **l**east-significant-**p**lace
**gcd** $\triangleq$ **g**reatest **c**ommon **d**ivisor (also known as highest common factor or **hcf**)
**lg** $\equiv$ log-to-base 2, **ln** $\equiv$ log-to-base-$e$, **log** $\equiv$ log-to-base-10
**floor** function: $\lfloor x \rfloor$ `the largest integer` $\leq x \equiv$ `Round to the nearest integer toward` $-\infty$
**ceiling** function: $\lceil x \rceil$ `the smallest integer` $\geq x \equiv$ `Round to the nearest integer toward` $+\infty$
**truncation** : `trunc`$x$ `only the integer-part of` $x \equiv$ `Round toward 0`
**O( )** $\equiv$ Order-of or the big-O function as defined in the algorithms literature (for example see [3]).
$|\cdot| \equiv$ cardinality if argument is a set; $\equiv$ absolute value of integer argument.
"RR " is abbreviation for "Residue Representation", "RD " is abbreviation for "Residue Domain",
"integer-domain" refers to the set of all integers $\mathbb{Z}$.
? denotes the "equality-check" operation.

**Notations-2 Algorithm Pseudo-code**

The pseudo-code syntax closely resembles **MAPLE** [34] syntax.
Lines beginning with # as well as everything between $/*$ and $*/$ are comments.

All entities/variables with a bar on top are vectors/ordered-touples (ex, $\overline{\mathtt{Z}} \equiv z_1, \cdots, z_K$)

Operations that can be implemented in parallel in all channels are shown inside a square/rectangular box. for example $\overline{\mathtt{Z}} \ \overline{\mathtt{X}} \ \boxed{\ddagger} \ \overline{\mathtt{Y}} \Rightarrow z_r \ x_r \ddagger y_r \mod m_r, \ r \ 1, \cdots, K$

### Brief introduction to RNS and canonical Definitions

**Definition 1 : We define "Reconstruction-Remainders" to be the component-wise values $\rho_1, \rho_2, \cdots, \rho_K$ defined by relations ( B-6 ) above.**

Note that Equation ( B-4 ) can be re-written as $Z_T \ Z \ Q \cdot \mathcal{M}$ ( B-8.1 )

or equivalently as $Z \ Z_T - Q \cdot \mathcal{M}$ where, ( B-8.2 )

$Q \ \left\lfloor \frac{Z_T}{\mathcal{M}} \right\rfloor$ Quotient when $Z_T$ is divided by $\mathcal{M} \triangleq \mathcal{R}_C \quad \Rightarrow 0 \leq \mathcal{R}_C \leq K-1$ ( B-9 )

**Definition 2 : We define the coefficient of $\mathcal{M}$** (which is denoted by the variable $Q$) in Equations ( B-8.$*$ ) to be the **"Reconstruction-Coefficient"** and henceforth denote it by the dedicated symbol "$\mathcal{R}_C$"

**Definition 3 : Full reconstruction** of the integer corresponding to a residue-touple refers to the process of **retrieving the entire unique digit-string representing that integer in a non-redundant, weighted-positional format (such as two's complement or decimal or the mixed-radix format).**

**D-4 : Full-precision** $\equiv d_T$ base-$b$ digits; where $d_T \lceil \log_b \mathcal{M}K \rceil \approx \lceil \log_b \mathcal{M} \rceil \triangleq n_b$; since $\mathcal{M} \gg K$ ( B-10 )

If the base $b \ 2$ then $n_b \ n_2$ is the bit-length required to represent the total modulus $\mathcal{M}$ or the overall range of the RNS;

and is therefore also denoted simply by the variable $n$ without any subscript.

**Definition 5 : Any method/algorithm that simply determines the value of $\mathcal{R}_C$ without attempting to fully reconstruct $Z$ is referred-to as a "Partial-Reconstruction" (PR).**

*Evaluating $\mathcal{R}_C$ yields an exact equality (Eqn. ( B-8.2 )) for the target integer $Z$, without any "mod" i.e., remaindering operations in it (unlike the statement of CRT, Eqn. ( B-4 )). For most operations (especially division with a constant) such an exact equality for $Z$ suffices, i.e., there is no need to fully reconstruct $Z$. This is why Partial-Reconstruction (i.e., evaluating $\mathcal{R}_C$ ) is an important enabling step underlying most other operations*

## § 4.1 Moduli selection

Note that a modulus of value $m_r$ needs a table with $(m_r - 1)$ entries to cover all possible values of the reconstruction-remainder $\rho_r$ w.r.t. $m_r$, (excluding the value 0). Therefore, the total number of memory locations required by all moduli is

$$\text{\# memory locations required } \sum_{r1}^{K} m_r - 1 \approx \sum_{r1}^{K} m_r \triangleq \mathbb{T}_{\mathbb{M}} \tag{5}$$

Thus, in order to minimize the memory needed, each component modulus should be as small as it can be.

Therefore, in order to cover a range $[0, R]$ we select smallest consecutive $K$ prime numbers starting with either 2 or 3, such that their product exceeds $R$:

$$\mathbb{M} \ \{m1, m2, \cdots, m_K\} \ \{2, 3, \cdots, K\text{-th prime number}\}, \quad \text{where} \quad \prod_{t1}^{K} m_t \ \mathcal{M} > R \tag{6}$$

This selection leads to the following two analytically tractable approximations:

$\langle \underline{\mathbf{1}} \rangle$ The notation defines $m_K$ to be the $K$-th prime number. In other words, $K$ is the index of prime number whose value is $m_K$. Consequently, $K$ and $m_K$ can be related to each other via the well-known "*prime-counting*" function [35] defined as

$$\pi x \quad \text{The number of prime numbers} \leq x \quad \approx \quad \frac{x}{\ln x} \quad \text{and therefore} \tag{7}$$

$$K \quad \pi m_K \approx \left( \frac{m_K}{\ln m_K} \right) \tag{8}$$

$\langle \underline{\mathbf{2}} \rangle$ The overall modulus $\mathcal{M}$ becomes the well known "*primorial*" function [36] which for any positive integer $N$ is denoted as "$N\#$" and defined as

$$N\# \quad \begin{array}{l} 1 \quad \text{if} \quad N \ 1 \\ \text{product of all prime numbers} \ \leq N \ , \quad \text{otherwise} \end{array} \tag{9}$$

(Note that a the definition as well as the notation for the primorial is analogous to the well known "factorial" function ($N!$)). The primorial function satisfies well-known identities [36, 37]

$$2^N \ < \ N\# < 4^N \ 2^{2N} \quad \text{and} \tag{10}$$

$$N\# \ \approx \ Oe^N \quad \text{for large} \ N \tag{11}$$

As a result, to be able to represent $n$ bit numbers (i.e. the range [0,$2^n - 1$]) , in the residue domain using all

available prime numbers (starting with 2), the total modulus satisfies

$$\mathcal{M} \approx \exp m_K > 2^n \quad \text{and therefore} \tag{12}$$

$$m_K \approx \ln \mathcal{M} \ \ln 2^n \ n \cdot \ln 2 \tag{13}$$

Substituting this value of $m_K$ in Eqn. (8), $K$, the number of moduli required to cover all "$n$"–bit long numbers can be approximated as :

$$K \ \pi m_K \approx \left(\frac{m_K}{\ln m_K}\right) \approx \frac{n \ln 2}{\ln n \ln 2} \approx O\left(\frac{n}{\ln n}\right) \approx O\left(\frac{\lg \mathcal{M}}{\ln \ \lg \mathcal{M}}\right) \tag{14}$$

These analytic expressions are extremely important because they imply :

A.1 $K < m_K << \mathcal{M}$ (which follows from relations (13) and (14) above.) Moreover, both (15)

A.2 maximum-modulus $m_K$ as well as the number of moduli $K$ grow logarithmically w.r.t. $\mathcal{M}$ (16)

i.e., linearly w.r.t. the wordlength $n$ (since $n \ \lceil \lg \mathcal{M} \rceil$).

### § 4·1·1 moduli selection enables exhaustive look-up strategy that covers all possible inputs

The attributes A.1 and A.2 make it possible to exhaustively deploy pre-computation and lookup because they guarantee that the total amount of memory required grows as a low degree polynomial of the wordlength $n$.

In other words, the main novelty in my method of moduli selection and its real significance is the fact that I leverage the selection to enable an exhaustive pre-computation and look-up strategy that covers all possible input cases. This exhaustive pre-computation and look-up in turn makes my algorithms extremely simple, efficient and therefore ultrafast because I deploy the maximum amount of pre-computation possible, and perform as much of the task ahead of time as possible; so that there is not much left to be done dynamically at run-time (a perfect example of this is the new "Quotient First Scaling" algorithm for RNS division by a constant divisor that is explained in detail in Section § 4.5 below).

In other words, the "minimization" of the total number of look-up table entries is the best possible scenario, but it is not necessary to obtain the major benefits that are illustrated for the first time in this invention. ***There is a lot more flexibility in selecting the moduli as long as they do not make it infeasible to deploy the exhaustive precomputation strategy***.

Consider a concrete example: Suppose the claims section says "select moduli so as to minimize the total amount of look-up table memory required."

If the desired range is all 32-bit numbers, then the set of moduli $\mathbb{M} \ \{2,3,5,7,11,13,17,19,23,29\}$ minimizes the total number of look-up table entries required.

Now, one can replace any component modulus from the above set (for example, say the modulus 29) with another prime number (such as 31, 37 or even 101). The resulting moduli set does not *minimize* the total number of look-up table entries required, but it is sufficiently close and would not make much of a difference in the ability to deploy the exhaustive precomputation strategy. In the strict sense, however, the modified moduli set does not satisfy the "minimization" criteria and this fact might be used to wiggle around having to acknowledge the use of intellectual property claimed by this patent.

We would therefore like to clarify that the spirit of this part of the invention (i.e., the moduli selection method) can be better captured by the following description:

Select the set moduli so as to simultaneously bound both
(i) $m_k$ = the maximum value in the set of moduli $\mathbb{M}$ by a low degree polynomial in $n$,
as well as
(ii) $K$ = the total number of moduli in the set $\mathbb{M}$ $|\mathbb{M}|$ (also known as the number of RNS channels) by another low degree polynomial in the wordlength $n$.
(both polynomials could be identical which is a special case). Following usual practices, we consider any polynomial of degree $\leq 16$ to be a low-degree polynomial.

In closing we would like to point out some additional benefits of our moduli selection :
+1 : This selection is general, in the sense that for any value of $R$ multiple moduli sets always exist.
+2 : The moduli are relatively easy to find, since prime numbers are sufficiently densely abundant irrespective of the value of $R$.
+3 : It fully leverages the parallelism inherent in the RNS
+4 : limiting $m_K$ and $K$ to small values makes it more likely that the entire RNS fits in a single h/w module.

## § 4.2 The Reduced Precision Partial Reconstruction ( "RPPR" ) algorithm

This is a fundamental algorithm that underlies all other algorithms to follow. To speed-up the Partial-Reconstruction, we combine the information contained in both integer as well as fractional domains. We express the CRT in the form:

$$\frac{Z_T}{\mathcal{M}} \quad \mathcal{R}_C \frac{Z}{\mathcal{M}} \left( \sum_{r1}^{K} \frac{\rho_r}{m_r} \right) \triangle S \quad \text{where,} \quad f_r \ \frac{\rho_r}{m_r} \ \Rightarrow \tag{17}$$

$$\mathcal{R}_C \quad \lfloor S \rfloor \quad \textbf{the integer part of the sum of fractions,} \quad \text{and} \tag{18}$$

$$\frac{Z}{\mathcal{M}} \quad S - \lfloor S \rfloor \quad \textbf{the fractional part of the sum of fractions} \tag{19}$$

Relation (18) states that $\mathcal{R}_C$ can be approximately estimated as the **integer part** of a sum of at most $K$ proper fractions $f_r$, $r \ 1, \cdots, K$. (proper fractions because the numerator $\rho_r$ is a remainder w.r.t. $m_r$; and therefore it is strictly less than $m_r$).

To speed up such an estimation of the $\mathcal{R}_C$ , we leverage pre-computation and look-up: for each modulus $m_r$, we pre-calculate the value of each of the $(m_r - 1)$ fractions, i.e., all possible fractions that can occur, and store them in the look-up table (denoted by $\mathbb{T}_{m_r}$)

$$\mathbb{T}_{m_r} \ [\tfrac{1}{m_r}, \tfrac{2}{m_r}, \cdots, \tfrac{M_r-2}{m_r}, \tfrac{m_r-1}{m_r}] \Rightarrow \quad \text{the } i\text{-th entry in the table } \mathbb{T}_{m_r} i \ f_{i,r} \ \frac{i}{m_r} \tag{20}$$

(if $\rho_r \ 0$ then the table entry is 0 which need not be explicitly stored).

The important point is that The look-up table for each modulus $m_r$ can be accessed independent of (and therefore in parallel with) the look-up table for any other modulus $m_s$ where $r \not\$.

The fractional values (obtained from the tables) are then added up as illustrated in Figure 1

### § 4·2·1 Derivation of the algorithm and novel aspects therein

Ⓕ First, note that we need to estimate the **integer part** of a sum of fractions, i.e., we need to be able to accurately evaluate the **most-significant digits/portion of the sum** as illustrated in Figure 1. The important

point is that ***whenever a computation needs to generate the "most-significant" bits/digits of the target, approximation methods can be used*** . For instance, in a division, "Quotient" is a lot easier to approximate than the "Remainder".
In other words, using the rational-domain interpretation allows us to focus on values that represent the "most-significant" bits/digits of the target and therefore *approximation methods can be invoked.*

② *The implication is that the precision of the individual fractional values that get added need not be very high.* ***All that is required is that the fractions*** $\mathbf{f_{i,r}}\ \frac{\mathbf{i}}{\mathbf{m_r}}$ ***be calculated to enough precision so that when they are all added together, the error is small enough so as not to reach up-to and affect the least significant digit of the integer part*** (to the extent possible).
Let the radix/base of the number representation be $b$ and let $w_f$ be the number of fractional (radix-$b$) digits required. Then, for each fraction $f_i$ we generate an upper and lower bound as follows:

For $\rho_i\ \emptyset$, let $\frac{\rho_i}{m_i}\ \ f_i$ the exact value of the reconstruction-fraction in channel $i$ (21)

$f_i\ \ 0.d_1d_2\cdots d_{w_f}\,\big|\,d_{w_f1}d_{w_f2}\cdots$ (22)

Truncation of $f_i$ to $w_f$ digits yields an under-estimate: $\ \widehat{f}_{\texttt{i\_low}}\ \ 0.d_1d_2\cdots d_{w_f} \le f_i$ (23)

and $\qquad 0 \le f_i - \widehat{f}_{\texttt{i\_low}}\ \ 0.0\cdots 0d_{w_f1}d_{w_f2}\cdots\ \ < 1/b^{w_f}$ (24)

However, a ceiling or rounding-toward- to retain $w_f$ fractional digits adds a **ulp** to the least significant digit (lsd), yielding an over-estimate: (25)

$\widehat{f}_{\texttt{i\_high}}\ \ 0.d_1d_2\cdots d_{w_f}\ \ 1\ \ \widehat{f}_{\texttt{i\_low}}\ \mathbf{ulp} \ge f_i \quad$ where $\quad \mathbf{ulp}\ \ \frac{1}{b^{w_f}}$ (26)

and $\qquad 0 \le \widehat{f}_{\texttt{i\_high}} - f_i < 1/b^{w_f}$ (27)

combining (23) and (26) we get $\ \widehat{f}_{\texttt{i\_low}} \le f_i \le \widehat{f}_{\texttt{i\_high}}\ \ \widehat{f}_{\texttt{i\_low}}\ \mathbf{ulp}$ (28)

Summing relations (28) over all $i$ from $1, \cdots, K$ we obtain

$\widehat{f}_{\texttt{1\_low}} \cdots \widehat{f}_{\texttt{K\_low}} \le f_1 \cdots f_K\ \ \mathcal{R}_C\ \frac{Z}{\mathcal{M}} \le \widehat{f}_{\texttt{1\_low}} \cdots \widehat{f}_{\texttt{K\_low}}\ n_z \cdot \mathbf{ulp}$ (29)

where, $n_z$ `number_of_nonzero_residues_in_the_touple` (30)

The understand the upper limit in relation (29) above, note that each non-zero $\rho_i$ makes the corresponding over-estimate higher than the under-estimate by a **ulp**, as per Eqns (21), (23) and (26).

Let $\quad \widehat{S}_{\texttt{low}} \triangle \widehat{f}_{\texttt{1\_low}} \cdots \widehat{f}_{\texttt{K\_low}}$ $\quad$ and $\quad \widehat{I}_{\texttt{low}} \triangle \left\lfloor \widehat{S}_{\texttt{low}} \right\rfloor$ $\quad$ integer part of $\ \widehat{S}_{\texttt{low}}$ (31)

$\widehat{S}_{\texttt{high}} \triangle \widehat{S}_{\texttt{low}}\ n_z \cdot \mathbf{ulp}$ $\quad$ and $\quad \widehat{I}_{\texttt{high}} \triangle \left\lfloor \widehat{S}_{\texttt{high}} \right\rfloor$ $\quad$ integer part of $\ \widehat{S}_{\texttt{high}}$ (32)

Taking the "floor" of each expression in the inequalities in relations (29) above; substituting the floors from Eqns (31) and (32); and using the identity

$\left\lfloor \mathcal{R}_C\ \frac{Z}{\mathcal{M}} \right\rfloor\ \ \mathcal{R}_C$ $\quad$ we obtain (33)

$\widehat{I}_{\texttt{low}} \le \mathcal{R}_C \le \widehat{I}_{\texttt{high}}$ $\quad$ so that (34)

if $\quad \widehat{I}_{\texttt{low}}\ \ \widehat{I}_{\texttt{high}}$ $\quad$ then $\quad$ the estimate $\ \mathcal{R}_C\ \ I_{\texttt{high}}\ \ I_{\texttt{low}}$ $\quad$ must be exact (35)

since both upper and lower bounds converge to the same value. In practice (numerical simulations), this case

is encountered in an overwhelmingly large fraction of numerical examples. Moreover,

$$\text{Since} \quad n_z \leq K \Rightarrow n_z \cdot \mathbf{ulp} \leq K \cdot \mathbf{ulp} \tag{36}$$

$$\text{by selecting } w_f \text{ so as to ensure } \; K \cdot \mathbf{ulp} \; K/b^{w_f} < 1 \qquad \text{from (29) we get} \tag{37}$$

$$\widehat{S}_{\texttt{low}} \leq \mathcal{R}_C \, \frac{Z}{M} \leq \widehat{S}_{\texttt{low}} \; 1 \tag{38}$$

taking the floor of each expression in the relation above yields

$$\widehat{I}_{\texttt{low}} \leq \mathcal{R}_C \leq \widehat{I}_{\texttt{low}} \; 1 \tag{39}$$

In other words, even in the uncommon/worst cases, wherein, $\widehat{I}_{\texttt{low}} \; \widehat{I}_{\texttt{high}}$, relation (39) demonstrates that the estimate of $\mathcal{R}_C$ can be quickly narrowed down to a pair of consecutive integers.

③ It is intuitively clear that further **"disambiguation"** between these choices needs at least one bit of extra information. This information is obtained from the value ($Z \mod m_e$) where $m_e$ is the extra modulus. For efficiency, $m_e$ should be as small as possible. Accordingly, our method leads to only two scenarios:
[a] if $\mathcal{M}$ is odd then $m_e \; 2$ is sufficient for disambiguation
[b] otherwise if 2 is included in the set of moduli $\mathbb{M}$, then $m_e \; 4$ is sufficient for disambiguation

$$\Rightarrow m_e \in \{2, 4\} \qquad \text{(this is analytically proved in [38])} \tag{40}$$

Note that when $\mathbb{M}$ includes "2" as a modulus, it already contains the value ($z_1 \; Z \mod 2$), i.e., the least significant bit of the binary representation of $Z$. The value ($Z \mod 4$) therefore conveys only one extra bit of information beyond what the residue touple conveys.

④ It is reasonable to assume that for primary/external inputs the extra-info is available. The exhaustive pre-computations can also assume that the extra-info is available. Starting with these, we generate the extra-bit of information (either explicitly or implicitly) for every intermediate value we calculate/encounter. This is done in a separate dedicated channel. Let

$$W \qquad \boxed{*} \, X \quad \text{where } \boxed{*} \text{ is a unary operation.} \quad \text{Then,} \tag{41}$$

$$W \mod m_e \qquad \boxed{*} \, X \mod m_e \mod m_e \quad \text{for } \boxed{*} \in \{\texttt{left-shift, power}\} \tag{42}$$

If the operation is a right shift, then finding the remainder of the shifted value w.r.t. $m_e$, is slightly more involved but it can be evaluated using a method identical to **"Quotient_First_Scaling"**, i.e., **"Divide_by_Constant"**, which explained in detail in Section § 4.5 below.

Likewise, let

$$Z \qquad X \; \boxed{\otimes} \, Y \quad \text{where } \boxed{\otimes} \text{ is a binary operation.} \quad \text{Then,} \tag{43}$$

$$Z \mod m_e \qquad X \mod m_e \, \boxed{\otimes} \, Y \mod m_e \mod m_e \quad \text{for } \boxed{\otimes} \in \{\pm, \times\} \tag{44}$$

Finally, since division is fundamentally a sequence of shift and add/subtract operations, as long as we keep track of the remainder of every intermediate value w.r.t. $m_e$, we can also derive the values of ($\texttt{Quotient} \mod m_e$) and ($\texttt{Remainder} \mod m_e$). Thus all the basic arithmetic operations are covered.

### § 4.2.2 Analytical Results

**Result 1 Pre-conditions :** *Let the radix of the original (non-redundant, weighted and positional, i.e., usual) number representation be denoted by the symbol "b" (note that $b \; 10$ yields the decimal representation, $b \; 2$*

*gives the binary representation). Suppose integer $Z \;\; z_1, z_2, \cdots, z_K$ is being partially re-constructed and the extra-bit-of-information, i.e., the value of ($Z \mod m_e$) is also available. Let*

$$\frac{Z_T}{\mathcal{M}} \;\approx\; \widehat{S}\;\widehat{I}\;\widehat{F}\;\sum_{r1}^{K}\widehat{f}_r \quad \textit{where,} \tag{45}$$

$$\widehat{I} \qquad \lfloor \widehat{S} \rfloor \;\textit{the integer part of the approximate sum}\; \widehat{S}, \quad \textit{and} \tag{46}$$

$$\widehat{F} \qquad \widehat{S} - \widehat{I} \;\textit{the fractional part of the sum, and} \tag{47}$$

$$\widehat{f}_i \qquad \widehat{f}_{\texttt{i\_low}}\ \text{Trunc}w_F f_i \quad \textit{truncation of} \quad f_i \quad \textit{to} \quad w_F \quad \textit{digits} \quad \textit{where} \tag{48}$$

$$f_i \qquad \frac{\rho_i}{m_i} \Rightarrow 0 \le f_i < 1 \tag{49}$$

and $\rho_i$ values are the reconstruction-remainders defined in Equation (B-6)

$$\textit{Let} \quad \delta\left(\frac{Z_T}{\mathcal{M}} - \widehat{S}\right) \qquad \textit{be the total error in the approximate estimate}\;\; \widehat{S} \tag{50}$$

*and let the Reconstruction-Coefficient $\mathcal{R}_c$ be estimated as $\mathcal{R}_c \approx \widehat{I}$ then,* (51)

**Result 1 :** *In order to narrow the estimate of the* **Reconstruction Coefficient** $\mathcal{R}_c$ *down to two successive integers, viz., $\widehat{I}$ or ($\widehat{I}$ 1), it is* **sufficient** *to carry out the summation of the fractions (whose values can obtained from the look-up-tables) in a fixed-point format with no more than a total of $w_T$ radix-$b$ digits, wherein*

$$w_T \qquad w_I\; w_F \qquad \textit{where} \tag{52}$$

$$w_I \qquad \textit{Number of digits allocated to the Integer part, and} \tag{53}$$

$$w_F \qquad \textit{Number of digits allocated to hold the fractional part} \tag{54}$$

*where the precisions (i.e., the digit lengths) of the integer and fractional parts satisfy the conditions:*

R1.1 $$w_I \qquad \lceil \log_b K \rceil \tag{55}$$

R1.2 $$w_F \qquad \left\lceil \log_b K \cdot \boldsymbol{\Delta}_{\text{uuzf}} \right\rceil \quad \text{where, } K = \text{number of moduli} = |\mathbb{M}|, \quad \text{and} \tag{56}$$

$$\boldsymbol{\Delta}_{\text{uuzf}} \;\equiv\; \textbf{“Unicity Uncertainty Zone Factor”}, \text{ that satisfies } \boldsymbol{\Delta}_{\text{uuzf}} \ge 2 \tag{57}$$

R1.3 *The Rounding mode adopted in the look-up-tables (when limiting the pre-computed values of the fractions to the target-precision) as well as during the summation of fractions as per equation (51) must be* ***TRUNCATION****, i.e., discard excess bits.*

For the proof of the above result as well as all other analytical results stated below, please refer to [38].

**Result 2 :** *In order to disambiguate between the two possible values of the Reconstruction Coefficient i.e., select the correct value ($\widehat{I}$) or ($\widehat{I}$ 1), a small amount of extra information is sufficient.*

*R2.1 In particular, (prior) knowledge of the remainder of $Z$ (the integer being partially reconstructed), w.r.t. one extra component modulus $m_e$ that satisfies*

$$\gcd\mathcal{M}, m_e < m_e \qquad \textit{is sufficient for the disambiguation.} \tag{58}$$

*R2.2 For computational efficiency, the minimum value of $m_e$ that satisfies (58) should be selected. Such a selection gives rise to the following two canonical cases:*

① *$\mathcal{M}$ is odd : In this case, $m_e \ 2$ is sufficient for disambiguation.*
② *$\mathcal{M}$ contains the factor 2 : then, $m_e \ 4$ is sufficient for disambiguation.*

### § 4·2·3 RPPR Algorithm: illustrative examples

The pre-computations and Look-up-tables needed for the partial reconstruction are illustrated next.

#### § 4.2·3·A First an example with small values, to bootstrap the concepts

Let the set of moduli be $\mathbb{M}$ {3,5,7,11}, so that, K = 4, $\mathcal{M}$ = 1155, and let $m_e$ 2

| modulus | table entries for row $m_r$ : column $i \leftarrow \frac{i}{m_r}$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\downarrow m_r$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 3→ | 0.3 | 0.6 | | | | | | | | |
| 5→ | 0.2 | 0.4 | 0.6 | 0.8 | | | | | | |
| 7→ | 0.1 | 0.2 | 0.4 | 0.5 | 0.7 | 0.8 | | | | |
| 11→ | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |

Table 1: Look-up table for the **RPPR** algorithm for the RNS-ARDSP system with $\mathbb{M}$ = {3,5,7,11}. This table uses the value of $\rho_r$ as the address to look-up the fraction $\frac{\rho_r}{m_r} \Rightarrow$ explicit value of $\rho_r$ is needed. In this case $K$ 4 and $\boldsymbol{\Delta}_{\text{uuzf}}$ 2 and therefore $w_I$ $\lceil \log_{10} 4 - 1 \rceil$ 1 integer decimal-digit and $w_F$ $\lceil \log_{10} 4 \times 2 \rceil$ 1 fractional decimal-digit. Accordingly, all the entries in the table have a single fractional digit. In this toy example we have deliberately left the table entries in the fixed-point fractional form for the sake of clarity (rather than scaling them by the factor $10^1$ and listing them as integers).

Then, the look-up table for the **RPPR** algorithm is shown in Table 1.

The table consists of 4 subtables (one per-channel/modulus) that are independently accessible in parallel. For each value of $m_r$, the table consists of a row that simply stores the approximate pre-computed values of the fractions $\frac{1}{m_r}, \frac{2}{m_r}, \cdots, \frac{m_r - 1}{m_r}$.

#### § 4.2·3·B Further Optimization: Skip the computation of $\rho_r$ values

Note that instead of explicitly calculating $\rho_r$ and then using it as an index into a table, the residue $z_r$ could be directly used as an index into a table that stores the appropriate values of the precomputed fractions.

$$\frac{\rho_r}{m_r} \left( \frac{w_i \times z_r \mod m_r}{m_r} \right) \tag{59}$$

The resulting table is illustrated in Table 2.

In this toy example the number fractional digits required for intermediate computations is 1 which is not a sizable reduction from the full precision which is 3 digits.

The the following non-trivial long-wordlength example demonstrates the truly drastic reduction in precision that is afforded by our novel algorithm.

| modulus | table entries for row $m_r$ : column $i \leftarrow \frac{i \times w_i \mod m_r}{m_r}$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\downarrow m_r$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $3\rightarrow$ | 0.3 | 0.6 | | | | | | | | |
| $5\rightarrow$ | 0.2 | 0.4 | 0.6 | 0.8 | | | | | | |
| $7\rightarrow$ | 0.2 | 0.5 | 0.8 | 0.1 | 0.4 | 0.7 | | | | |
| $11\rightarrow$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 | 0.0 | 0.2 | 0.4 | 0.6 | 0.8 |

Table 2: **R**esidue **A**ddressed Look-up **T**able (**RAT**) for the **RPPR** algorithm for the RNS-ARDSP system with $\mathbb{M}$ = {3,5,7,11}. This table is a further optimized version of Table 1 above: here, the residue value $z_r$ is directly used as the address of a location that stores the corresponding value of $\frac{w_i \times z_r \mod m_r}{m_r}$ in the $r$th row (i.e., sub-table) for component-modulus $m_r$. Calculation of $\rho_r$ is not required when this table is used. Note that the only difference between this table and Table 1 is a permutation of the entries in sub-tables for those moduli $m_i$ for which the "inner-weights" are larger than unity (in this case $w_i > 1$ for the last two rows corresponding to moduli 7 and 11).

**§ 4.2·3·C** **A nontrivial long word-length example**

Now consider the partial reconstruction of numbers with a word-length = 256 bits. Here the range is $R \ 2^{256}$. In this case, first 44 prime-numbers are required to cover the entire range. Therefore $K \ 44$ and $\mathbb{M}$ = {2, 3, 5, 7, 11, $\cdots$ , 181, 191, 193}, $m_e \ 4$, and the product of all the component-moduli is $\mathcal{M}$ = 19896237639169098164041525154528515360273440272182105821220397609541391057227 0 and the ratio $\left(\frac{\mathcal{M}}{2^{256}}\right) \approx 1.718$ $m_K = m_{44}$ = 44th prime number = 193
The word-length required to represent $\mathcal{M}$ is

$$n \ \lceil \log_{10} \mathcal{M} \rceil \ \lceil 77.298 \rceil \ 78 \ \text{ decimal digits} \tag{60}$$

and the word length required for $Z_T$ is

$$d_T \ \lceil \log_{10} \mathcal{M} \times 44 \rceil \ \lceil 78.9 \rceil \ 79 \ \text{ digits} \tag{61}$$

Hence, any conventional full re-construction method to evaluate $\mathcal{R}_C$ requires at least a few operations on 79 digit-long integers.

In contrast, our new partial-reconstruction method requires a drastically smaller precision as well as a drastically small number of simple operations (only additions) to accurately evaluate the reconstruction coefficient $\mathcal{R}_C$ . As per Result R1.2 above, the number of fractional digits required in the look-up-tables as well as in intermediate computations is only

$$w_F \ \lceil \log_{10} 44 \times 2 \rceil \ 2 \ \text{ decimal fractional digits.} \tag{62}$$

When addition of such fractions is considered, the integer part that can accrue requires no more than 2 additional digits (since we are adding at most $K \ 44$ values, and each is a proper fraction their sum must be less than 44 and therefore requires no more than 2 decimal digits to store the integer-part).
**Therefore, the total number of digits required in all intermediate calculations is as small as 4, which is a drastic reduction from 79. (In general the reduction in precision is from $O\lg\mathcal{M}$ required by conventional methods versus the much smaller amount $O\lg\lg\mathcal{M}$ required by our method).**

**Another extremely important point: by appropriate scaling, all the fixed point fractional values in the table can be converted into integers. Correspondingly, all the fixed-point computations (additions and subtractions of these fractions) are also scaled and can therefore be realized as integer-only operations. The obvious scaling factor is** $10^{w_F}$. The resulting look-up-table that contains the scaled integers as its

entries is illustrated in Table 3.

| component modulus $\downarrow m_r$ | Table enties for row $m_r$ : **column i** $\leftarrow \left\lfloor \frac{i \times w_r \mod m_r \times C_s}{m_r} \right\rfloor$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | $\cdots$ | 189 | 190 | 191 | 192 |
| $2\rightarrow$ | 50 | | | | | | |
| $3\rightarrow$ | 66 | 33 | | | | | |
| | | | $\cdots$ | | | | |
| $\vdots$ | | | $\vdots$ | | | | |
| | | | | $\cdots$ | | | |
| $191\rightarrow$ | 71 | 42 | $\cdots$ | 57 | 28 | | |
| $193\rightarrow$ | 77 | 55 | $\cdots$ | | | 44 | 22 |

Table 3: The Redsidue Addressed Table **RAT** for the **RPPR** algorithm for the RNS-ARDSP system with first 44 prime numbers as moduli, i.e., $\mathbb{M}$ = {2,3,5,7,11,$\cdots$, 191, 193}. The fixed point truncated values of the fractions are scaled by a factor of $C_s \ 10^2$.

Note that un-scaling requires a division. But since the scaling factor is a power of the radix of the underlying number representation, un-scaling can be achieved simply by left-shift and truncation of integers. **Thus, with the scaling, floating point computations are entirely avoided.**

Next we formally specify the algorithm and simultaneously illustrate it for two examples:
1. ***Example 1***: find $\mathcal{R}_C$ for value X=641 in the small wordlength case.
inputs: $\overline{\overline{X}} \ 3,4,1,2$ (note that the fully reconstructed value for this touple viz., "641" is not known to the algorithm. It is only given the touple) and the extra-info value ($X \mod m_e \ 1$));
2. ***Example 2***: find $\mathcal{R}_C$ for the value "X=1" in the long=wordlength case.
(inputs: $\overline{\overline{X}} \ 1,1,\cdots,1,1$ and ($X \mod m_e \ 1$));
Right below every step of the algorithm, the computations actually performed for each of the two examples are also illustrated inside "comment-blocks"

### § 4·2·4 Specification of the algorithm via Maple-style pseudo-code

**Algorithm `Reduced_Precision_Partial_Reconstruction`** $(\overline{\mathtt{Z}}, z_e)$

# **Inputs :** residue-touple $\overline{\mathtt{Z}} \ z_1,z_2,\cdots,z_K$, extra-info $z_e$ = $(Z \mod m_e)$, $m_e \in \{2,4\}$

# **Output:** Exact value of the Reconstruction Coefficient $\mathcal{R}_c$

# **Pre-computation :** moduli, $\mathcal{M}$, $m_e$, all constants (ex, $\mathcal{M}_j, w_j \ 1/\mathcal{M}_j \mod m_j, \cdots$)
# create(Reconstruction__Table(s));

# *Step 1 :* using $z_r$ as the indexes, look up ultra low precision estimates $\widehat{f}_r, \ r \ 1..K$
# Note that this can be done in parallel in all channels

$n_z \leftarrow 0$ ;
for i from 1 to K do  # for each channel $i$
  if $z_r = 0$ then
    $\widehat{f}_r \leftarrow 0$;
    $n_{z_r} \leftarrow 1$;
  else
    $\widehat{f}_r \leftarrow z_r$th element in the look-up-table for $m_r$ ;
    $n_{z_r} \leftarrow 0$;
  fi;  # same as "end if"
od;  # same as "end for"

# Example 1 : $K = 4$ values read from Table 1 above (and scaled by the factor $C_s$ = 10) = [5,1,2,6]
# Example 2 : $K = 44$ pre-scaled values read from Table 2 above = [50, 61, $\cdots$, 71,77]

# *Step 2 :* Sum all the $\widehat{f}_r$ values with a total of only $w_T$ digits of precision to obtain the bounds

$\widehat{S}_{\text{low}} \leftarrow \sum_{r=1}^{K} \widehat{f}_r$ ;  $n_z \leftarrow \sum_{r=1}^{K} n_{z_r}$ ;  and  $\widehat{S}_{\text{high}} \leftarrow \widehat{S}_{\text{low}} + n_z$;

# Example 1 : $\widehat{S}_{\text{low}}$ = 5+1+2+6 = 14  and  $\widehat{S}_{\text{high}}$ = 14+4 = 18
# Example 2 : $\widehat{S}_{\text{low}}$ = (50 + 61 + $\cdots$ + 71 + 77) = 2581  and  $\widehat{S}_{\text{high}}$ = 2581+44 = 2625

# *Step 3:* unscale and take the floor of the bounds to obtain integer bounds on $\mathcal{R}_C$
# note that these can be realized as a right-shift followed by truncation
$\widehat{I}_{\text{low}} \leftarrow \left\lfloor \frac{\widehat{S}_{\text{low}}}{C_s} \right\rfloor$  and  $\widehat{I}_{\text{high}} \leftarrow \left\lfloor \frac{\widehat{S}_{\text{high}}}{C_s} \right\rfloor$
# Example 1 : $\widehat{I}_{\text{low}} \leftarrow \lfloor \frac{14}{10} \rfloor = 1$  and  $\widehat{I}_{\text{high}} \leftarrow \lfloor \frac{18}{10} \rfloor = 1$
# Example 2 : $\widehat{I}_{\text{low}} \leftarrow \lfloor \frac{2589}{100} \rfloor = 25$  and  $\widehat{I}_{\text{high}} \leftarrow \lfloor \frac{2625}{100} \rfloor = 26$

# *Step 4:* check if upper & lower integer bounds have same value. if yes, return it as the correct answer
if ($\widehat{I}_{\text{low}} = \widehat{I}_{\text{high}}$) then
  Return($\widehat{I}_{\text{low}}$);
fi;
# Example 1 : both upper and lower bounds converge to the same value 1 $\Rightarrow$ correct value of $\mathcal{R}_C$ = 1, and is returned
# Example 2 : Bounds do not converge to the same value $\Rightarrow$ need to disambiguate between {25,26} using extra info

# *Step 5:* **disambiguate** using extra-info
if ($Z_T \mod m_e = \left\{\widehat{I}_{\text{low}} \cdot \mathcal{M} \mod m_e + Z \mod m_e\right\} \mod m_e$) then
  Ans := $\widehat{I}_{\text{low}}$;
else
  Ans := $\widehat{I}_{\text{high}}$;
end if;
# Example 2 : it can be verified that: $Z_T \mod 4 = 1 \neq 25 \times 2 + 1 \mod 4 = 3 \mod 4$ but
# ($Z_T \mod 4 \checkmark 26 \times 2 + 1 \mod 4 = 1 \mod 4$

`Return(Ans);` `# Output = correct value of` $\mathcal{R}_c$ $\square$

**`End__Algorithm`**

The correctness of the algorithm can be proved by invoking **Results 1, 2** and other equations and identities presented in this document. In addition, the algorithm has been implemented in Maple (software) and exhaustively verified for small wordlengths (upto 16 bits). A large number of random cases ($> 10^5$) for long wordlengths (up to $2^{20} \approx$ million-bits) were also run and verified to yield the correct result.

### § 4·2·5 RPPR Architecture

Figure 3 illustrates the block diagram of an architecture to implement the **RPPR** algorithm. The main goal of the architecture is to fully leverage the **parallelism** inherent in the RNS. There are $K$ channels, each capable of performing all basic arithmetic operations, viz.,
$\{ \pm, \times,$ `division, shifts, powers, equality-check, comparison,` $\ldots\}$ modulo $m_r$
which is the component-modulus value for that particular channel.
In addition, each channel is also capable of accessing it's own look-up-table(s) (independent of other channels).
Finally there is a dedicated channel corresponding to the extra modulus $m_e = 2$ or $m_e = 4$ that evaluates $Z \bmod m_e$ for every non-primary integer $Z$ (non-primary refers to a value that is not an external input or is not one of the precomputed values).

We would like to emphasize that the schematic diagram is independent of whether the actual blocks in it are realized is in hardware or software. The parallelism inherent in the RNS is independent of whether it is realized in h/w or s/w. This should be contrasted with some other speed-up techniques (such as rendering additions/subtractions constant-time by deploying redundant representations) that are applicable only in hardware [39].

### § 4·2·6 Delay models and assumptions

In order to arrive at concrete estimates of delay, we assume a fully dedicated h/w implementation. Each channel has its own integer ALU that can perform all operations modulo any specified modulus. Among all the channels, the $K$-th one that performs all operations modulo-$m_K$ requires the maximum wordlength since $m_K$ is the largest component-modulus.

The maximum channel wordlength is : $\quad n_K = \lg m_K \approx O\lg n \approx \lg\ln \mathcal{M} \approx \lg\lg \mathcal{M}$ (63)

Note that this is drastically smaller than the wordlength $n_c$ required for conventional binary representation, which is roughly $On$, the number of bits required to represent $\mathcal{M}$.

In accordance with the literature, we make the following assumptions about delays of hardware modules
$\langle \mathbf{A-1} \rangle$ A carry-look-ahead adder can add/subtract two operands within a delay that is logarithmic w.r.t. the wordlength(s) of the operands.
$\langle \mathbf{A-2} \rangle$ Likewise, a fast hardware multiplier (which is essentially a fast multi-operand accumulation tree followed by a fast carry-lookahead-adder and therefore) also requires a delay that is logarithmic w.r.t. the wordlength of the operands.

More generally, a fast-multi-operand addition of $K$ numbers each of which is $n$-bits long requires a delay of

$$O\lg K \; O\lg n \lg K \quad \text{which becomes} \approx O\lg n \;\; \text{in our case.} \tag{64}$$

$\langle \underline{\mathbf{A-3}} \rangle$ Assuming that the address-decoder is implemented in the form of a "tree-decoder", a look-up table with $\mathbb{L}$ entries requires $\approx O\lg\mathbb{L}$ delay to access any of it's entries.
$\langle \underline{\mathbf{A-4}} \rangle$ We assume that dedicated shifter(s) is(are) available. A multi-stage-shifter (also known as a "barrel" shifter [4, 40]) implements shift(s) of arbitrary (i.e., variable) number of bit/digit positions, where the delay is $\approx O\lg$`maximum_shift_distance_in_digits`) units.

### § 4·2·7 Estimation of the total Delay

The preceding assumptions, together with Equation (63) imply that the delay $\Delta_{\text{CH}}$ all operations within individual channels can be approximated to be

$$\Delta_{\text{CH}} \approx \lg m_K \approx O\lg\lg n \approx \lg\lg\lg\mathcal{M} \tag{65}$$

which very small.

The delay estimation is summarized in Table 4.

| Algorithm *Step no:* and operation(s) performed | can individual channels work in parallel? | Approximate Delay as a function of wordlength $n$ | Justification |
|---|---|---|---|
| *1:* Compute or look-up $\rho_r$ values | yes | $O\lg\lg n$ | Equation (65) |
| *2:* Using $\rho_r$ as the index look up estimates $\widehat{f}_r$ | yes | $O\lg\lg n$ | Equation (65) |
| *3:* Add all the estimates | No | $O\lg K \approx O\lg n$ | Assumption $\langle \underline{\mathbf{A-2}} \rangle$ and Equation (64) |
| *4:* Un-scale the sum back and truncate | No | $O\lg\lg n$ | realized via a shift and truncation |
| *5:* Check if upper and lower bounds converge to the same value | No | $O1$ | obvious, equality check on small values |
| *6:* Disambiguation | No | $O\lg\lg n$ | $m_e \in \{2,4\} \Rightarrow$ tiny operands |
| Overall delay ≡ Latency | | $O\lg n$ | dominant "functional" component |

TABLE 4: ESTIMATION OF THE DELAY REQUIRED BY THE **RPPR** ALGORITHM

As seen in the table, the dominant delay is in Step 3, the accumulation of values read from the per-channel

look-up tables.
Therefore, the overall delay is $\approx O\lg n \approx O\lg\lg\mathcal{M}$

### § 4·2·8 Memory required by the PR algorithm

The $r$-th channel associated with modulus $m_r$ has its own Look-up table with $(m_r - 1)$ entries (since the case where the remainder is "0" need not be stored). Hence, the number of storage locations needed is

$$\sum_{r1}^{K} m_r - 1 < K \cdot m_K \approx OK^2 \approx On^2 \tag{66}$$

Each location stores a fractional value that is no longer than $w_F$ digits $\approx O\lg n$ bits. Therefore,

$$\text{total storage (in bits)} = On^2 \text{ (locations) } \times O\lg n \text{ (bits per locations) } \approx On^2 \lg n \text{ bits} \tag{67}$$

There are several important points to note:

**1** ▷ Although the above estimate makes it look as-though it is a single chunk of memory, in reality it is not. Realize that each channel has its own memory that is independently accessible. The implications are:

**2** ▷ The address-selector (aka the decoder) circuitry is substantially smaller and therefore faster (than if the memory were to be one single block).

**3** ▷ It is a READ-ONLY memory, the precomputed values are to be loaded only once, they never need to be written again. In a dedicated VLSI implementation, it would therefore be possible to utilize highly optimized, smaller and faster cells.

## § 4.3 Base change/extension

Those familiar with the art will realize that once the **"RPPR"** algorithm yields an exact equality for the operand (being partially re-constructed), a base-extension or change is straightforward.

Without loss of generality, the algorithm is illustrated via an example which extends a randomly generated 32-bit long unsigned integer to a 64-bit integer (without changing the value), which requires an extension of the residue touple as shown below.

In order to cover the (single-precision) range $[0,\ 2^{32}]$ , the moduli set required is

$\mathbb{M}\ \mathbb{M}_{32}\ \{2,3,5,7,11,13,17,19,23,29\}$ so that $K\ |\mathbb{M}|\ 10,$ and total product $\mathcal{M}\ 6469693230,$

the reconstruction weights are $\mathcal{M}_i\ \frac{\mathcal{M}}{m_i}, i\ 1,\cdots,10$ 3234846615, 2156564410, 1293938646,

924241890, 588153930, 497668710, 380570190, 340510170, 281291010, 223092870

and the inner weights are $w_i\ \left(\frac{1}{\mathcal{M}_i}\right)\ \bmod m_i, i\ 1,\cdots,10\ 1,1,1,3,1,11,4,9,11,12$

In order to cover the double-precision range $[0,\ 2^{64}]$ , the extended moduli set required is

$\mathbb{M}_{\text{ext}}\ \mathbb{M}_{64}\ \{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53\}$ so that $K_{\text{ext}}\ |\mathbb{M}_{\text{ext}}|\ 16,$ and total product $\mathcal{M}_{\text{ext}}\ 32589158477190044730$

Let the single precision operand be $Z\ 1355576195 \equiv \overline{\mathbb{Z}}\ 1,2,0,1,6,12,3,10,10,21$ and $z_e\ Z \bmod 4\ 3.$ The CRT expresses the value $Z$ in the form

$$Z\ 3234846615 \times \rho_1\ 2156564410 \times \rho_2 \cdots 223092870 \times \rho_{10} - 6469693230\ \mathcal{R}_{C_z} \tag{68}$$

where, $\rho_i\ z_i \times w_i \bmod m_i$ is the reconstruction remainder for channel $i,$ for $i\ 1,\cdots,10$ and $\mathcal{R}_{C_z}$ the reconstruction coefficient for $Z$
The only unknown in Eqn (68) is $\mathcal{R}_{C_z}$ which can be determined using the **"RPPR"** algorithm yielding an exact integer equality, enabling a straightforward evaluation of the extra-residues needed to extend the residue touple. For example the first extra-modulus in the example at hand is $m_{11}\ 31$. Accordingly, $\overline{\mathbb{Z}}_{\texttt{ext}}11\ z_{11}\ Z \bmod 31$
Note that $3234846615 \bmod 31, \cdots, 6469693230 \bmod 31$ are all constants for a given RNS and can be pre-calculated and stored. In general we always assume that whichever values can be pre-computed are actually pre-computed. Thus

$$\theta_{i,j}\ \mathcal{M}_i \bmod m_{e_j} \quad \text{for} \quad i\ 1,\cdots,K \quad \text{and} \quad j\ K\ 1,\cdots,K_{\text{ext}} \tag{69}$$

are all pre-computed and stored, so that the operation of evaluating the remainder w.r.t. an extra modulus $m_{e_r}$ in the extended RNS system (such as the modulus 31 in the running example at hand) simplifies to

$$Z \bmod m_{e_r}\ \left(\theta_{1,e_r}\rho_1 \quad \theta_{2,e_r}\rho_2 \quad \cdots \quad \theta_{K,e_r}\rho_K \ -\ \Delta_{e_r}\ \mathcal{R}_{C_z}\right) \bmod m_{e_r} \tag{70}$$
where, $\Delta_{e_r}\ \mathcal{M} \bmod m_{e_r}$

Next, we specify the algorithm exactly in Maple-style pseudo-code.

### § 4.3.1 Specification of the algorithm via maple-style pseudo-code

**Algorithm `Base_extension_using_RPPR_method`** $(\overline{\mathbb{Z}}, z_e)$

```
/*  Inputs :  residue-touple Z̄ z_1,z_2,···,z_K, extra-info z_e = (Z mod m_e), m_e 4
    corresponding to a 32 bit unsigned integer Z    */

# Output:  residue touple for the 64-bit extension of Z

/*  Pre-computation :  original moduli-set M_32 {2,3,···,29}, |M_32| K_32 10
    extended moduli-set M_64 {2,3,···,53},  |M_64| K_64 16
    all constants (ex, M_j,w_j  1/M_j  mod m_j,···)
    Reconstruction__Table(s) for both M_32 and M_64, etc.     */

# Step 1 :      evaluate R_{C_z} using the "RPPR" algorithm
 R_{C_z}  = Reduced_Precision_Partial_Reconstruction ( Z̄, z_e) ;

# Step 2 :      evaluate the extra residues as per Eqns (68) and (70)
# This can be executed in parallel in all channels corresponding to the extension moduli
```

```
for i from 1 to K_e do      # in each extension channel i
   sum := 0;
   for j from 1 to K_e do
      sum sum  ρ_j × θ_{i,j} ;
   od;
   z_i  sum − Δ_i  R_{C_z}  mod m_i ;
   Z̄  concat Z̄, z_i ;          # append the new residue to the residue touple
od ;

Return(Z̄)          □
```

**End__Algorithm**

## § 4.4 Sign and Overflow Detection by Interval Separation ( SODIS )

The next canonical operation I have sped-up is sign-detection. The new algorithms for sign-detection in the RNS are illustrated in this section.

As illustrated in Figure 4, fundamentally, the RNS representation does allow a separation of positive and negative integers into distinct non-overlapping regions (what is meant here is that the RNS mapping is not so strange as to "mix" positive and negative numbers throughout the entire range. It takes an "interval" (namely the interval including all negative integers) and faithfully (i.e., without changing the length of the interval) simply translates (or displaces) it into another another "interval" which is not surprising since the "mapping" corresponding to the translation is the simple first degree equation describing the "modulo" operation, i.e., Eqn. (3))

Q : Where then is the problem in sign detection?

A : (i) Note that a "re-construction" of the overall magnitude is necessary

(ii) For the efficiency of representation (i.e., in order not to waste capacity) the following additional constraint is also imposed in most RNS implementations.

$$F_{\text{max}}^{-} \;\; F_{\text{max}} \; 1 \tag{71}$$

In other words, ALL the unsigned integers in the range [0,$\mathcal{M}-1$] are utilized, not a single digit value is wasted.
An unfortunate by-product of this quest for representational efficiency is that consecutive integers $F_{\text{max}}$ and $F_{\text{max}}^{-}$ end up having opposite signs. Consequently, re-construction must be able to distinguish between consecutive integers, i.e., the resolution of the re-construction must be full, i.e., in fractional computations

$$\textbf{ulp} < \frac{1}{\mathcal{M}} \tag{72}$$

This, in-turn requires that all fractional computations must be carried out to the full precision, thereby rendering them slow.

The main question therefore is whether it is possible to make do with the drastically reduced precision we wish to deploy? and if so, then how?

The answer is to insert a sufficiently large "separation-zone" between the positive and negative regions, as

illustrated in Figure 5.

In Figure 5, note that

both "0" as well as $\mathcal{M}_e$ correspond to the actual magnitude 0 (73)
unsigned integers $\{1, \cdots, F_{\text{max}}\}$ represent ve values and (74)
unsigned integers $\{F^{-}_{\text{max}}, \cdots, \mathcal{M}_e - 1\}$ represent $-$ve values
$\{-\mathcal{M}_e - F^{-}_{\text{max}}, \cdots, -1\}$, respectively, wherein (75)
Unsigned Integer $F_{\text{max}}$ represents the maximum positive magnitude allowed, and (76)
Unsigned Integer $F^{-}_{\text{max}}$ represents the max. $-$ve magnitude allowed = $(\mathcal{M}_e - F^{-}_{\text{max}})$ (77)
The interval between $F_{\text{max}}$ and $F^{-}_{\text{max}}$ is the separation zone (78)

Most practical/useful number representations try to equalize the number of ve values and the number of $-$ve values included in order to attain maximal amount of symmetry. This yields the constraint

$$F^{-}_{\text{max}} \approx \mathcal{M}_e - F_{\text{max}} \tag{79}$$

Intuitively, it is clear that equal lengths should be allocated to
(1) the ve interval
(2) the $-$ve interval and
(3) the separation-zone between the opposite polarity intervals.

For this to be possible, the extended modulus $\mathcal{M}_e$ must satisfy $\mathcal{M}_e > 3 \cdot F_{\text{max}}$ (80)

Finally, the attainment of maximal possible symmetry dictates that to the extent possible, the separation-zone must be symmetrically split across the mid-point of the range $[0, (\mathcal{M}_e - 1)]$. Note that Figure 5 incorporates all these symmetries.

With the separation interval in place, note that all ve numbers $Z$ within the range [1,$F_{\text{max}}$] when represented as fractions of the total magnitude $\mathcal{M}_e$ now satisfy

$$0 < \frac{Z}{\mathcal{M}_e} < \frac{1}{3} \tag{81}$$

Likewise, all $-$ve numbers $Z^{-}$when represented as fractions of the total magnitude $\mathcal{M}_e$ satisfy

$$\frac{2}{3} < \frac{Z^{-}}{\mathcal{M}_e} < 1 \tag{82}$$

But Eqn (19) (repeated here for the sake of convenience) states that

$$\frac{Z}{\mathcal{M}} \; S - \lfloor S \rfloor \quad \text{the fractional part of the sum of fractions} \approx \sum_{r1}^{K} \widehat{f}_r \tag{83}$$

In other words, the **separation interval enables the evaluation of the sign of the operand** under consideration **by examining one (or at most two) most significant digits** of the accumulated sum of fractions.

Recall that in the partial reconstruction, the integer part of the sum-of-fractions was of crucial importance. It is quite striking that when the interval separation is properly leveraged as illustrated herein, the most significant digit(s) of the fractional part also convey equally valuable information, viz., the sign of the operand.

As per Equations (81) and (82) the natural choice of the detection boundaries $T$ and $T^{-}$ is specified by the relations

$$\frac{T}{\mathcal{M}_e} \quad \frac{1}{3}\ 0.333\cdots \qquad \text{and} \tag{84}$$

$$\frac{T^{-}}{\mathcal{M}_e} \quad \frac{2}{3}\ 0.666\cdots \tag{85}$$

However, note that even if the "detection boundaries" $T$(for ve numbers) and $T^{-}$(for $-$ve numbers) are moved slightly into the "separation zone", as illustrated in Figure 5,
*the sign-detection outcome does not change*.

For the ease of computation, we therefore set

$$\frac{T}{\mathcal{M}_e} \quad \frac{4}{10}\ 0.4 \qquad \text{and} \tag{86}$$

$$\frac{T^{-}}{\mathcal{M}_e} \quad \frac{6}{10}\ 0.6 \tag{87}$$

### § 4·4·1 Specification of sign-detection algorithm via Maple-style pseudo-code

**Algorithm** **`Eval__Sign`** ( $\overline{\mathtt{Z}}$, $z_e$ )

`#` **Input(s) :** `Given an integer` $Z$ `represented by the residue touple/vector` $\overline{\mathtt{Z}}$ $z_1,\cdots,z_K$
`#and one extra value, viz.,` $z_e$ $Z \mod m_e$) `where` $m_e$ 4

/$*$ **Output(s) :** the **Sign** of $Z$, defined as

$$\textbf{Sign}Z \begin{cases} 0 & \text{if } Z\ 0 \\ 1 & \text{if } Z > 0 \\ -1 & \text{otherwise} \end{cases} \tag{88}$$

The algorithm also returns two more values in addition to the sign
(i) the value of the reconstruction coefficient $_{\mathcal{R}_C}$ for the input and
(ii) Approx_overflow_estimate which is a flag defined as follows:

$$\texttt{Approx\_overflow\_estimate} \begin{cases} 1 & \text{if overflow is detected for sure} \\ 0 & \text{otherwise} \end{cases} \tag{89}$$

further computation is needed to determine whether there is an overflow in the 2nd case above

**Pre-computation :** `Everything needed for the` **`"RPPR"`** `algorithm, and in addition` $F_{\mathtt{max}}$, $F^{-}_{\mathtt{max}}$, `decision boundaries` $T$`and` $T^{-}$, `etc.` $*$/

# *Step 1 :* Look up pre-stored estimates $f_r, r\ 1..K$

```
for i from 1 to K do          # for each channel  i
   if z_r 0 then
      f̂_r 0;
      n_{z_r} 1;
   else
      f̂_r z_r-th element in the look-up-table for m_r ;
      n_{z_r} 0;
   end if;
od;
```

# *Step 2 :* Sum all the $f_r$ values with only $w_T$ total digits

$\widehat{S}_{\text{low}} \sum_{r1}^{K} \widehat{f}_r$ ; $n_z \sum_{r1}^{K} n_{z_r}$ ; and $\widehat{S}_{\text{high}}\ \widehat{S}_{\text{low}}\ n_z$;

```
if  (n_z  K)  then      #  all components = 0 ⇒ Z  0
   Return(0, 0, 0) ;
fi ;
```

# *Step 3:* unscale and separate integer and fractional parts

$\widehat{I}_{\text{low}} \left\lfloor \frac{\widehat{S}_{\text{low}}}{C_s} \right\rfloor$ ; $\widehat{F}_{\text{low}}\ \widehat{S}_{\text{low}} - \widehat{I}_{\text{low}}$ ; and $\widehat{I}_{\text{high}} \left\lfloor \frac{\widehat{S}_{\text{high}}}{C_s} \right\rfloor$ ; $\widehat{F}_{\text{high}}\ \widehat{S}_{\text{high}} - \widehat{I}_{\text{high}}$ ;

# important substitutions

$\widehat{F}\ \widehat{F}_{\text{low}}$ ; and $\widehat{I}\ \widehat{I}_{\text{low}}$ ;

# *Step 4:* determine the temporary sign

```
Approx_overflow_estimate := 0 ;
if (F̂ < T) then
   Temp_Sign := 1;
else if (F̂ > T⁻) then
   Temp_Sign := −1;
else
   Approx_overflow_estimate := 1;
   if  (F̂ < 1/2)  then
      Temp_Sign := 1;
   else
      Temp_Sign := −1;
   end if;
end if;
```

# *Step 5:* determine $\mathcal{R}_C$

```
if  (Î_high  Î_low)  then
   R_c  Î
else
   if  (Z_T  mod 4 {Î · M  mod 4 Z  mod 4} mod 4)  then
```

```
        R_c Î;
    else
        R_c Î 1;
    fi;
fi;

if  (R_c Î)  then
    Sign := Temp_Sign ;
else
    Sign := −1 × Temp_Sign ;
fi ;

Return( Sign, R_c , Approx_overflow_estimate)        □
End__Algorithm
```

### § 4·4·2 Overflow Detection

Since we are dealing with sign-detection of integers, an underflow of "magnitude" simply results in the value "0"; no other action needs to be taken in case of magnitude underflow.

However, a magnitude overflow, must be detected and flagged. let $A$ and $B$ be the operands and let $\otimes$ denote some operation, then "overflow of magnitude" includes both cases

case 1 $A \otimes B >$ **Posedge** = $F_{\max}$ and (90)

case 2 $A \otimes B <$ **Negedge** $-\mathcal{M} - F^{-}_{\max}$ (91)

or dividing both sides by $\mathcal{M}$

$$\frac{A \otimes B}{\mathcal{M}} > \frac{F_{\max}}{\mathcal{M}} \quad \text{and} \tag{92}$$

$$\frac{A \otimes B}{\mathcal{M}} < \frac{F^{-}_{\max}}{\mathcal{M}} \tag{93}$$

However, recall that the decision boundaries $T$ and $T^{-}$ are shifted by a small amount into the "separation region". As a result, whenever input values in the range [$F_{\max}$,$T$] or in the range [$T^{-}$,$F^{-}_{\max}$] are encountered, they will be wrongly classified as being within the correct range even though they are actually outside the designated range. The only solution to this problem is to separately evaluate the sign of either $Z - F_{\max}$ or $Z - F^{-}_{\max}$ to explicitly check for overflow.

#### § 4·4·2·A Specification of overflow detection algorithm via Maple-style pseudo-code

**Algorithm** **Eval__overflow** ($\overline{Z}$, $z_e$, Sign, approx_overflow)

```
# Note that every invocation of this algorithm must be immediately preceded by
# an invocation of the Eval__sign algorithm
```

/* **Precomputations :** same as those for algorithm **Eval__sign**
**Inputs :** $\overline{\mathtt{Z}}$, $z_e$, Sign of $Z$, approx_overflow for $Z$
The last two values are obtained as a result of the execution of the
**Eval__sign** algorithm immediately preceding the invocation of this algorithm.

**Output(s) :** overflow flag defined as

$$\mathtt{overflow} \begin{cases} 1 & \text{if overflow is detected for sure} \\ 0 & \text{no overflow} \end{cases} \tag{94}$$

*/

# *Step 1 :* handle the trivial cases first

```
if  (Sign == 0)  then
   Return(0);
fi;


if (approx_overflow == 1)  then
   Return(1);
fi;
```

# *Step 2 :* Determine the argument "$TZ$" for auxiliary sign-detection.
# note that the residue touple $\overline{\mathtt{TZ}}$ corresponding to the integer $TZ$ is directly determined
# via component-wise subtractions in the Residue Domain

```
if  (Sign == 1)  then
   TZ   Z ⊖ F_max ;          # ⊖ denotes component-wise subtraction in the residue domain
   tz_e  (z_e − F_max  mod 4)  mod 4 ;          # disambiguation-bootstrapping
else if (Sign == −1) then
   TZ  Z⊖F⁻_max ;
   tz_e  (z_e − F_max  mod 4)  mod 4 ;          # keep track of all values modulo 4
end if;
```

# *Step 3 :* determine the sign of $TZ$, (which is denoted by the variable $\mathsf{S}_{tz}$ herein

```
S_tz ,  tmp_rc,  approx_overflow_tz := Eval__sign(TZ, tz_e) ;
if (Sign == 1) then
   if ( S_tz == 1) then
      overflow := 1 ;
   else
      Overflow := 0 ;
   end if;
else if (Sign == −1) then
   if ( S_tz == −1) then
      Overflow := 1;
   else
      Overflow := 0;
```

```
   end if;
end if;

Return(overflow);      □
End__Algorithm
```

Once these building blocks are specified, the overall **SODIS** algorithm is specified next.

#### § 4.4·2·B Specification of sign and overflow detection algorithm via Maple-style pseudo-code

**Algorithm** `Sign__and__Overflow__Detection__by__Interval__Separation` $(\overline{\mathrm{Z}},\ z_e)$

```
# this algorithm is abbreviated as " SODIS "
# Inputs :  Z̄, z_e
# Outputs :  Sign(Z), overflow,  R_{C_z}

Sign,  R_{C_z} , approx_overflow := Eval__sign (Z̄, z_e) ;
overflow := Eval__overflow (Z̄, z_e,  Sign, approx_overflow) ;
Return(Sign, overflow,  R_{C_z} ) ;      □
End__Algorithm
```

Those familiar with the art shall realize that using the algorithms presented in this section, a comparison of of two numbers say $A$ and $B$ can be realized extremely fast, without ever leaving the residue domain in a straightforward manner by detecting the sign of $(A - B)$.

## § 4.5 The Quotient First Scaling ( QFS ) algorithm for dividing by a constant

Assume that a double-length (i.e., $2n$-bit) dividend $X$ is to be divided by an $n$-bit divisor $D$, which is a constant, i.e., it is known ahead of time. The double length value $X$ is variable/dynamic. It is either an external input or more typically it the result of a squaring or a multiplication of two $n$-bit integers. It is assumed that the extra-bit of information, i.e., the value of $(X \mod m_e)$ is available. Given positive integers $X$ and $D$, a division entails computing the quotient $Q$ and a remainder $R$ such that

$$X \;\; Q \times D \; R \qquad \text{where} \qquad 0 \leq R < D \qquad \text{so that} \qquad Q \;\; \left\lfloor \frac{X}{D} \right\rfloor \tag{95}$$

To derive the Division algorithm, start with the alternative form of the Chinese Remainder Theorem (CRT) which expresses the target integer via an exact integer equality of the form illustrated in Equations (B-8.*). Express the double-length Dividend $X$ as

$$X \;\; X_T - \mathcal{M} \cdot \; \mathcal{R}_{C_x} \;\; \left( \sum_{r1}^{K} M_r \cdot \rho_r \right) - \mathcal{M} \cdot \; \mathcal{R}_{C_x} \tag{96}$$

where the exact value of **Reconstruction Coefficient** $\mathcal{R}_{C_x}$ is determined using the **"RPPR"** algorithm explained in Section 4.2 above. In other words, there is no unknown in the above exact integer equality expressing the value of the dividend $X$.

To implement Division, evaluate the quotient $Q$ as follows:

$$Q \quad \left\lfloor \frac{X}{D} \right\rfloor \left\lfloor \sum_{r1}^{K} \frac{M_r \cdot \rho_r}{D} - \frac{\mathcal{R}_{C_x} \cdot \mathcal{M}}{D} \right\rfloor \left\lfloor \left\{ \sum_{r1}^{K} Q_r \ \frac{R_r}{D} \right\} - Q_{\text{RC}} \ \frac{R_{\text{RC}}}{D} \right\rfloor \tag{97}$$

$$\left\lfloor \left\{ \sum_{r1}^{K} Q_r \ f_r \right\} - Q_{\text{RC}} \ f_{\text{RC}} \right\rfloor \qquad \text{where} \tag{98}$$

$$Q_r \ , \ Q_{\text{RC}} \quad \left\lfloor \frac{M_r \cdot \rho_r}{D} \right\rfloor \ , \ \left\lfloor \frac{\mathcal{R}_{C_x} \cdot \mathcal{M}}{D} \right\rfloor \quad \text{precomputed quotient values, and} \tag{99}$$

$$R_r \ , \ R_{\text{RC}} \quad M_r \cdot \rho_r - Q_r \cdot D \ , \ \mathcal{R}_{C_x} \cdot \mathcal{M} - Q_{\text{RC}} \cdot D \quad \text{precomputed remainders, and} \tag{100}$$

$$f_r \ , \ f_{\text{RC}} \quad \frac{R_r}{D} \ , \ \frac{R_{\text{RC}}}{D} \ \text{remainders expressed as fractions of the divisor } D \tag{101}$$

The exact integer-quotient can be written as

$$Q \ \left( \sum_{r1}^{K} Q_r \right) - Q_{\text{RC}} \left\lfloor \left( \sum_{r1}^{K} f_r \right) - f_{\text{RC}} \right\rfloor \ Q_I \ Q_f \qquad \text{where} \tag{102}$$

$$Q_I \ \left( \sum_{r1}^{K} Q_r \right) - Q_{\text{RC}} \quad \text{the contribution of Integer-part, (hence the subscript "I"), and} \tag{103}$$

$$Q_f \ \left\lfloor \left( \sum_{r1}^{K} f_r \right) - f_{\text{RC}} \right\rfloor \quad \text{the contribution of fractional-part (hence the subscript "f")} \tag{104}$$

Since exact values of $Q_r$ and $Q_{\text{RC}}$ are pre-computed and looked-up, the value of $Q_I$ in
Eqn (103) above is exact. However, since we use approximate precomputed values of the fractions truncated to drastically small precision, the value of $Q_f$ calculated via Eqn (104) above is approximate. As a result, the value of $Q$ that is calculated is also approximate. We indicate approximate estimates by a hat on top, which yields the relations :

$$\widehat{Q} \quad Q_I \ \widehat{Q_f} \qquad \text{where} \tag{105}$$

$$\widehat{Q_f} \quad \left\lfloor \left( \sum_{r1}^{K} \widehat{f_r} \right) - \widehat{f_{\text{RC}}} \right\rfloor \tag{106}$$

Our selection of moduli (explained in detail in Section 4.1 above) leads to the fact that the number of of memory-locations required for an exhaustive look-up turns out to be a small degree (quadratic) polynomial of $n \ \lg \mathcal{M}$. This amount of memory can be easily integrated in h/w modules in today's technology for word-lengths up to about $2^{17} \approx 0.1$-Million bits (which should cover all word-lengths of interest today as well as in the foreseeable future).

Note that the **Reconstruction Coefficient** $\mathcal{R}_{C_x}$ can also assume only a small number of values (no more than $(K-1)$ where $K$ is the number of moduli as per Eqns (B–4, B–5) and (B–9) Hence, quotient values $Q_{\text{RC}}$ and the fractions $f_{\text{RC}} \ \frac{R_{\text{RC}}}{D}$ can also be pre-computed and stored for all possible values the **Reconstruction**

**Coefficient** $\mathcal{R}_{C_x}$ can assume.

### § 4·5·1 Further novel optimizations

① *Store the pre-computed Quotient values directly as residue touples*

Note that the quotient values themselves could be very large (about the same word-length as the divisor $D$). However, we need not store these long-strings of quotient values, since in many applications (such as modular exponentiation) the quotient is only an intermediate variable required to calculate the remainder.
Obviously the extra bit of information conveyed by $(Q_{rs} \mod m_e)$ is also pre-computed and stored together with the touple representing the exact integer quotient

$$Q_{ir} \left\lfloor \frac{r \cdot M_i}{D} \right\rfloor \quad \text{for } i \ 1, \cdots, K \quad \text{and } r \ 1, \cdots, m_r - 1 \tag{107}$$

The total memory required to store either the full-length long-integer value or storing the residues w.r.t. the component moduli as a touple is about the same. By opting to store only the residue-touples, we eliminate the delay required to convert integer quotient values into residues, without impacting the memory requirements significantly.

② *Only fractional remainders truncated to drastically reduced precision $O\lg K \approx O\lg\lg \mathcal{M}$ need to be pre-computed and stored (exactly similar to the* **"RPPR"** *algorithm*).

③ *simple scaling converts all fractional storage/computations into integer values.*

Thus, the **QFS** algorithm needs 2 distinct Quotient_Tables.

### § 4·5·2 Quotient-Tables explained via a small numerical example

I believe that the tables can be best illustrated by a concrete-small example. Assume that the divisor $D \ 209 \ 11 \times 19$ (i.e. $D$ is representable as an 8-bit number). The dividends of interest are therefore up-to 16-bit long numbers. In this case the moduli turn out to be [2, 3, 5, 7, 11, 13, 17]. Even if the first two moduli (viz., 2 and 3) are dropped, the product still exceeds the desired range $0, 2^{16}$. Therefore we select

$$\mathbb{M} \ \{5, 7, 11, 13, 17\} \Rightarrow K \ 5, \ \ \mathcal{M} \ 85085 \quad \text{and the extra-modulus } m_e \ 2 \tag{108}$$

To realize division by this divisor $D \ 209$, the first table required is shown in Table 5.
This table is referred to as "Quotient_Table_1" (or also as the "Quotient_Touples_Table" ). It stores all possible values of Quotients required to evaluate the first term (the sum) in Eqn (103). The entries (rows) corresponding to each component-modulus $m_r$ constitute a sub-table of all possible values $\rho_r$ can assume for that value of $m_r$. For the sake of clarity, we have used a "double-line" to separate one sub-table from the next.

To illustrate the pre-computations, we explain the last sub-table, in Quotient_Table_1 corresponding to the component-modulus "$m_5 \ 17$", wherein, $M_{17} \ \frac{\mathcal{M}}{17} \ 5005$.
This sub-table has 16 rows. The first row corresponds to $\rho_5 \ 1$ the second row corresponds to $\rho_5 \ 2$ and so on. Now, we explain each entry in the penultimate row in Table 1 above (this row corresponds to $\rho_5 \ 15$). The value in the 3rd column titled "Quotient ...." lists the quotient, i.e., $\left\lfloor \frac{M_{17} \times 15}{D} \ \frac{5005 \cdot 15}{209} \right\rfloor \ 359$. The next entry (within the angled-brackets $\langle\rangle$) simply lists the value of $Q_{5\_15} \mod m_e \ 359 \mod 2 \ 1$. The 4th column stores the residue-touple $4, 2, 7, 8, 2$ representing the quotient 359.

| modulus $m_r$ $\downarrow$ | $\rho_r$ $1,2,\cdots$ $m_r-1$ $\downarrow$ | Quotient $Q_r$ $\lfloor \frac{M_r \cdot \rho_r}{D} \rfloor$ and $\langle Q_r \mod 2 \rangle$ | moduli $m_j, j\ 1 \cdots K$ [5, 7, 11, 13, 17] $Q_r \mod m_j,\ j\ 1 \cdots K$ | Remainder $R_r$ $M_r \cdot \rho_r - Q_r \cdot D$ Scaled Fractional Rem = trunc $\frac{R_r}{D} \times 10^{w_f}$ $\downarrow$ |
|---|---|---|---|---|
| 5 | 1 | 81 $\langle 1 \rangle$ | [1, 4, 4, 3, 13] | 42 |
| | 2 | 162 $\langle 0 \rangle$ | [2, 1, 8, 6, 9] | 84 |
| | 3 | 244 $\langle 0 \rangle$ | [4, 6, 2, 10, 6] | 26 |
| | 4 | 325 $\langle 1 \rangle$ | [0, 3, 6, 0, 2] | 68 |
| $\vdots$ | | | $\cdots$ | $\vdots$ |
| 17 | 1 | 23 $\langle 1 \rangle$ | [3, 2, 1, 10, 6] | 94 |
| | 2 | 47 $\langle 1 \rangle$ | [2, 5, 3, 8, 13] | 89 |
| | $\vdots$ | | | |
| | 15 | 359 $\langle 1 \rangle$ | [4, 2, 7, 8, 2] | 21 |
| | 16 | 383 $\langle 1 \rangle$ | [3, 5, 9, 6, 9] | 15 |

Table 5: Quotient_Table_1 for **RNS- ARDSP** with moduli $\mathbb{M}\ 5, 7, 11, 13, 17$ and divisor $D\ 209$. In this case, two digits suffice to store the scaled fractional-remainders in the last column.

The last column in Table 1 stores the fixed point fractional remainder values scaled by the multiplying factor $b^{w_f}\ 10^2$ to convert them into integers. For instance, in the penultimate row: the actual remainder is $5005 \times 15 - 359 \times 209\ 44$, corresponding fractional remainder is
$\frac{44}{209} \approx 0.21052....$ which when truncated to two decimal places yields 0.21
Accordingly, `trunc(`$\frac{44}{209} \times 10^2$`) = 21` and this is the value stored in the last column.

We would like to point out that the actual full-wordlength-long integer values of quotients $Q_r$ (that are listed in column 3 in the table) need not be (and hence are not) stored in a real (h/w or s/w)
implementation of the algorithm (the full decimal $Q_r$ values were included in column 3 in Table 1 above, merely for the sake of illustration). In an actual implementation, only the extra-information, i.e., $\langle Q_r \mod 2 \rangle$ values (shown inside the angled-braces $\langle\ \rangle$ in column3) and the residue-domain touples representing $Q_r$ (as shown in column 4 in the table) are stored. For example, in the penultimate row, actual quotient value "359" need not be stored, only $\langle 359 \mod 2 \rangle\ \langle 1 \rangle$ would be stored, together with the touple of residues of 359 w.r.t. the component moduli
= $359 \mod 5, \cdots, 359 \mod 17\ 4, 2, 7, 8, 2$ as shown in column 4 therein.

Next we explain Table 6, which shows Quotient_Table_2 (also referred to as the "Quotient_Rc_Table" )

This table covers all possible values of the **Reconstruction Coefficient** $\mathcal{R}_{C_x}$ in Eqns (96)–(98). Like Table 5, the values in column 2 (i.e., the full-wordlength-long integer values of quotient $Q_c$) are not stored in actual implementation, (they are included in the table only for the sake of illustration). In actual implementations, only the residues of $Q_c$ with respect to (w.r.t.) 2 (shown inside angled braces $\langle\ \rangle$ in column 2) and the touple of residues of $Q_c$ w.r.t. the component-moduli are stored as illustrated in the third column of the Table. The last column stores the fixed point fractional remainder values scaled by the factor $10^{w_f}$ to convert them into integers.

Another nontrivial distinction of Quotient_Table_2 from all previous tables is the fact that the fractional values in the last column are always rounded-up (the mathematical expression uses the "ceiling" function).

| $\mathcal{R}_C$ $1,2,\cdots K$ $\downarrow$ | Quotient $Q_c \ \lfloor \frac{M \cdot \mathcal{R}_C}{D} \rfloor$ $\langle Q_c \mod 2 \rangle$ | moduli $m_j, j \ 1 \cdots K$ [5, 7, 11, 13, 17] $Q_c \mod m_j, \ j \ 1 \cdots K$ | Remainder $R_c \ \mathcal{M} \cdot \mathcal{R}_C - Q_c \cdot D$ Scaled Fractional Remainder ceil $\frac{R_c}{D} \times 10^{w_f} \ \downarrow$ |
|---|---|---|---|
| 1 | 407 $\langle 1 \rangle$ | [2, 1, 0, 4, 16] | 11 |
| 2 | 814 $\langle 0 \rangle$ | [4, 2, 0, 8, 15] | 22 |
| 3 | 1221 $\langle 1 \rangle$ | [1, 3, 0, 12, 14] | 32 |
| 4 | 1628 $\langle 0 \rangle$ | [3, 4, 0, 3, 13] | 43 |
| 5 | 2035 $\langle 1 \rangle$ | [0, 5, 0, 7, 12] | 53 |

Table 6: Quotient_Table_2 for **RNS- ARDSP** with moduli [5,7,11, 13, 17] and divisor $D$ 209.

Note that the last term in Equations (96) and (98), has a negative sign. As a result, when rounding the fractional remainders, we must "over-estimate" them, so that when this value is *subtracted* to obtain the final quotient estimate, we never over-estimate. In other words, the use of "ceiling" function is necessary to ensure that we are always "under-estimating" the total quotient.

### § 4·5·3 Specification (pseudo-code) of the QFS Algorithm

Like the **RPPR**-algorithm, we illustrate the division algorithm with 2 examples:
(i) first with small sized operands (dividend $X$ 3249, divisor $D$ 209) so that the reader can replicate the calculations by hand/calculator if needed.
(ii) The 2nd numerical example is a realistic long-wordlength case.
Instead of separating the pseudo-code and numerical illustration, we have waved in the numerical illustration of each step of the algorithm for the running (small) example at hand by including the numerical calculations into the pseudo code as comment blocks.

**Algorithm Quotient___First___Scaling__Estimate** ($\overline{\mathrm{X}}$, $\langle X \mod m_e \rangle$)

**# Inputs :** Dividend $X$ as a residue-touple $\overline{\mathrm{X}}$ $x_1, \cdots, x_K$ and $\langle X \mod m_e \rangle$, where $m_e \in \{2, 4\}$

**# Pre-computations :** Moduli, extra_modulus $m_e$, all constants $\mathcal{M}, M_r, w_r, r \ 1, 2, \cdots, K$ etc.

create (Reconstruction_Tables) ; create (Quotient_Tables) ;

*# Step 1 : use the* **RPPR***-algorithm to find the Reconstruction-(Remainders & Coefficient) for $X$*

```
(1.1)  ρ_1, ··· , ρ_K, R_{C_x}  RPPR(X̄, m_e, ⟨X mod m_e⟩)
(1.2)  nonzero_rrems := 0;
(1.3)  for i from 1 to Nmoduli do
           if ρ_i ∅ then nonzero_rrems := nonzero_rrems + 1; fi;
       od;
(1.4)  if ( R_{C_x}  ∅) then      nonzero_rcx := 1;
       else
           nonzero_rcx := 0;
       fi;
```

*# In the numerical example:* $\rho \ 10, 2, 2, 5, 2$; $\mathcal{R}_{C_x} := 2$; `nonzero_rrems := 5; nonzero_rcx := 1;`

*# Step 2 : Using the $\rho_i$ and the the $\mathcal{R}_{C_x}$ values as "indexes", look-up in parallel the touples $\overline{\mathrm{T}}_i \rho_i$,*

*# scaled remainders* $\texttt{Rr}_i$, QRcx, RRcx, *and the corresponding* extra_info *values (all added in* $|\ |$*)*

(2.1) $\overline{\texttt{T}}_i$ Quo_Tab_1[i, $\rho_i$, 3]; $\texttt{Rr}_i$ Quo_Tab_1[i, $\rho_i$, 4]; $i\ 1,\cdots,K$

(2.2) $\overline{\texttt{QRcx}}$ Quo_Tab_2[ $\mathcal{R}_{C_x}$ , 3]; RRcx Quo_Tab_2[rcx, 4];

(2.3) extra_info_$\texttt{T}_i$ Quo_Tab_1[i, $\rho_i$, 2]; extra_info_QRcx Quo_Tab_2[ $\mathcal{R}_{C_x}$ , 2];

/* *In the example:* $\overline{\texttt{T}}_1\ 4,1,8,5,1,\ \overline{\texttt{T}}_2\ 2,6,7,10,11,\ \overline{\texttt{T}}_3\ 4,4,8,9,6,\ \overline{\texttt{T}}_4\ 0,3,4,4,1,$
$\overline{\texttt{T}}_5\ 2,1,8,6,9$ *and* $\overline{\texttt{QRcx}}\ 4,2,0,8,15$. *(*$\texttt{Rr}_1$,$\texttt{Rr}_2$, $\texttt{Rr}_3$,$\texttt{Rr}_4$, $\texttt{Rr}_5$,Rrcx*) := (47 63, 1, 78, 84, 22)*
*Note that there is no "extra-information" associated with the fractional-remainder values* */

*# Step 3 : Execute accumulations in parallel in all the RNS and extra channel(s)*

(3.1) $\overline{\texttt{Q}}_I \quad \left(\boxed{+}_{i\ 1}^{K}\ \overline{\texttt{T}}_i\right)\ \boxed{-}\ \overline{\texttt{QRcx}}\ ;$ (3.2) $\widehat{\texttt{Q}}_f\ \left(\sum_{j1}^{K}\texttt{Rr}_j\right) - \texttt{RRcx}\ ;$

/* $\boxed{\pm}$ *denotes a component-wise addition/subtraction of touples in parallel in all the RNS channels.*
*"*$\sum$*" denotes addition of scalars in the extra channel(s) In the example:*
$\overline{\texttt{Q}}_I\ 4,1,8,5,1\ \boxed{+}\ 2,6,7,10,11\ \boxed{+}\ 4,4,8,9,6\ \boxed{+}\ 0,3,4,4,1\ \boxed{+}\ 2,1,8,6,9\ \boxed{-}\ 4,2,0,8,15$
$8 \mod 5,\cdots,13 \mod 17\ 3,6,2,0,13$ *and* $\widehat{\texttt{Q}}_f\ 47\ 63\ 1\ 78\ 84 - 22\ 251$ */

*# Step 4 : Set* $\widehat{\texttt{Q}}$f_unscaled *:= Unscaled* $\widehat{\texttt{Q}}_f$*. Also evaluate bounds on* $\widehat{\texttt{Q}}_f$*, and check if* $\widehat{\texttt{Q}}_f$ *is exact.*

(4.1) $\widehat{\texttt{Q}}\texttt{f\_unscaled} := \left\lfloor \frac{\left(\widehat{\texttt{Q}}_f\right)}{\texttt{b}^{\texttt{w}}} \right\rfloor ;$ (4.2) $\widehat{\texttt{Q}}_{\texttt{f\_high}} \left\lfloor \frac{\left(\widehat{\texttt{Q}}_f \text{ nonzero\_rrems + nonzero\_rcx}\right)}{\texttt{b}^{\texttt{w}}} \right\rfloor ;$

*# here,* b *is the base and* w *is the precision ⇒ only left-shift followed by truncation suffices*

(4.3) $\widehat{\texttt{Q}}_{\texttt{f\_low}}\ \widehat{\texttt{Q}}\texttt{f\_unscaled}$

(4.4)
```
if  Q̂_f_low Q̂_f_high then
      Q_is_exact := 1 ;
else
      Q_is_exact := 0 ;
fi;
```
*# In the example :*
*#* $\widehat{\texttt{Q}}_{\texttt{f\_low}}\ \lfloor \frac{251}{10^2} \rfloor\ 2$; *and*
*#* $\widehat{\texttt{Q}}_{\texttt{f\_high}}\ \lfloor \frac{25151}{10^2} \rfloor\ 2; \Rightarrow$
*# in the example,* Q_is_exact := 1;

*# Step 5 : evaluate* $\widehat{\widehat{\texttt{Q}}}$*: convert* $\widehat{\texttt{Q}}$f_unscaled *into a residue-touple and add it to* $\overline{Q_I}$

(5.1) $\widehat{\widehat{\texttt{Q}}}_{\texttt{f\_touple}}\ \texttt{vector}\left(\texttt{K},\ \texttt{i} \rightarrow \widehat{\texttt{Q}}\texttt{f\_unscaled} \mod \texttt{m}_\texttt{i}\right)\ ;$

(5.2) $\widehat{\widehat{\texttt{Q}}}\ \overline{Q_I}\ \boxed{+}\ \widehat{\widehat{\texttt{Q}}}_{\texttt{f\_touple}}$ *# In the example :* $\widehat{\widehat{\texttt{Q}}}\ 3,6,2,0,13\ \boxed{+}\ 2,2,2,2,2\ 0,1,4,2,15$

*# Step 6 : Also generate* $\widehat{Q} \mod m_e$*; the "disambiguation-bootstrapping" step*

(6.1) extra_info_$\widehat{\texttt{Q}}$ $\left[\left(\sum_{i1}^{K}\texttt{extra\_info\_T}_i\right) - \texttt{extra\_info\_QRcx}\right] \mod m_e;$

(6.2) extra_info_$\widehat{\texttt{Q}}$ extra_info_$\widehat{\texttt{Q}}$ $\widehat{\texttt{Q}}\texttt{f\_unscaled} \mod m_e$

# *in the example :* `extra_info_`$\widehat{\texttt{Q}}$ $1\,0\,0\,0\,0-0 \mod 2 \quad 2 \mod 2\ 1$

(7) **Output :** `Return(`$\widehat{\widehat{\texttt{Q}}}$`,` $\widehat{Q} \mod m_e$ `,Q_is_exact) ;` □

***End__Algorithm***

/* *In the example : Using the CRT, it can be verified that* $\widehat{\widehat{\texttt{Q}}}\ 0,1,4,2,15 \equiv 15$ *and* $\widehat{Q} \mod m_e \checkmark 1$.
*It is also easy to independently check that* $\lfloor \frac{3249}{209} \rfloor\ 15$, *verifying the returned value of flag* `Q_is_exact` */ We would like to clarify some important issues regarding the **QFS** algorithm.

① From the residue touple $\widehat{\widehat{\texttt{Q}}}$, returned by the algorithm, the remainder can be directly estimated as a residue-touple; and the extra info value ($\widehat{R} \mod m_e$) can also be evaluated using the fundamental division relation (Eqn (95) above):

$$\widehat{\widehat{\texttt{R}}}\ \overline{\texttt{X}} \ \boxed{-}\ \widehat{\widehat{\texttt{Q}}}\ \boxed{\times}\ \overline{\texttt{D}} \tag{109}$$

$$\widehat{R} \mod m_e\ X \mod m_e - \widehat{Q} \mod m_e \times D \mod m_e \mod m_e \tag{110}$$

② Note that the input $X$ is made available to the algorithm only as a residue touple, not as a fully reconstructed decimal or binary integer. In addition, one extra bit conveyed by ($X \mod m_e$) is also required by the algorithm. Given these inputs, the algorithm generates $\widehat{\widehat{Q}}$ as well as $\widehat{Q} \mod m_e$ (and therefore $\widehat{\widehat{R}}$ and $\widehat{R} \mod m_e$ as per Eqns (109) and (110)), thereby demonstrating that
*the outputs are delivered consistently in the same format as the inputs.*

③ The integer estimate $\widehat{Q}$ corresponding to the residue-touple $\widehat{\widehat{\texttt{Q}}}$ can take only one of the two values
[a] If the variable/flag "`Q_is_exact`" is set to the value "1", then $\widehat{Q}\ Q$, i.e., the estimate equals the exact integer quotient. In practice (numerical experiments) this happens in an overwhelmingly large number of cases.
[b] otherwise, the flag `Q_is_exact` 0, indicating that *the algorithm could not determine whether or not* $\widehat{Q}$ *is exact. (because of the drastically reduced precision used to store the pre-computed fractions)* In this case $\widehat{Q}$ could be exact, i.e., $\widehat{Q}\ Q$
or $\widehat{Q}\ Q-1$, i.e., $\widehat{Q}$ can under-estimate $Q$ by a **ulp**.
Further disambiguation between these two values is possible by calculating the estimated-remainder $\widehat{R}$ and checking whether $(\widehat{R}-D)$ is ve or −ve
Let the exact integer remainder be denoted by $R$. It is clear that the estimated integer-remainder $\widehat{R}$ can have only two possible values:

[a] if $\widehat{Q}$ is exact, then $\widehat{R}\ R$ , i.e., $\widehat{R}$ is also exact; or (111)

[b] $\widehat{Q}\ Q-1 \ \Rightarrow\ \widehat{R}\ X-Q-1D\ X-Q\cdot D\ D\ R\ D$ (112)

In other words, in the relatively infrequent case [b], performing a sign-detection on $(\hat{R}-D)$ is guaranteed to identify the correct $Q$ and $R$ in all cases. (if $(\hat{R}-D)$ is $ve$, then it is clear that $\widehat{Q}$ underestimated $Q$ by a **ulp**; otherwise $\widehat{Q}\ Q$)

### § 4·5·4 Estimation of the Delay of and the Memory required for the QFS Algorithm

#### § 4.5·4·A Delay model and Latency estimation

We assume dedicated h/w implementation of all channels (including the extra channels). Within each channel the look-up tables are also implemented in h/w (note that the tables need not be "writable"). All tables are independently readable in parallel with a latency of $O\lg n$. Likewise, since each component modulus is small as well as the number of channels ($K$) is also small, we assume that a dedicated adder-tree is available in each channel for the accumulations modulo the component-modulus for that channel. The latency of the accumulations can be also shown to be logarithmic in the word-length, i.e., $O\lg n$. Likewise, we assume that a fast, multistage or barrel shifter is available per channel so that delay of "variable: shifts is also $O\lg n$.

Figure 7 illustrates a timing diagram showing the sequence of successive time-blocks in which the various steps of the **QFS** algorithm get executed. At the top of each block, we have also shown its latency as a function of (the overall RNS word-length) $n$, under the assumptions stated above.

Since the maximum latency of any of the blocks is $O\lg n$, the overall/total latency of the h/w implementation is estimated to be $O\lg n$.

#### § 4.5·4·B Memory requirements

In addition to the reconstruction table, we also need the Quotient Tables. The total number of number of entries in both parts of the Quotient table is $OK^2/2\ OK-1\ OK^2$. In this case, each table entry has $K\ 1$ components, wherein, each component is no bigger than $O\lg\lg K$ bits.
Consequently the total storage (in bits) that is required is $\approx OK^3 \lg\lg K bits \approx On^3 \lg\lg n$ bits.

## § 4.6 Modular exponentiation entirely within the Residue Domain

Modular exponentiation refers to evaluating ($X^Y \mod D$). In many instances, in addition to $D$, the exponent $Y$ is also known ahead of time (ex: in the RSA method,$Y$ is the public or private-key). Our method does not need $Y$ to be a constant, but we assume that it is a primary/external input to the algorithm and hence available in any desired format (in particular, we require the exponent $Y$ as a binary integer, i.e., a string of $w$-bits).

$$\text{Let } Y\ y_{w-1}2^{w-1}\ y_{w-2}2^{w-2} \cdots y_2 2^2\ y_1 2^1\ y_0 \tag{113}$$

$$\cdots y_{w-1} * 2\ y_{w-2} * 2\ y_{w-3} * 2 \cdots y_0 \tag{114}$$

To the best of our knowledge, one of the fastest methods to perform modular exponentiation expresses the exponent $Y$ as polynomial of radix 2, parenthesized as shown Eqn (114) above (known as the "Horner's method" of evaluating a polynomial). Since the coefficient of the leading-term in (113) must be non-zero, (i.e. $y_{w-1}$ = 1), the modular exponentiation starts with the initial value `Ans :=` $X^2 \mod D$. If $y_{w-2}$ $\emptyset$ then

the result is multiplied (modulo-D) by X and is then squared. This operation is repeatedly performed in a loop as shown below:

**# Initialization :** `Ans =` $X^2$ `mod D;` ..... mod_red_1

**# Loop :** `for i from` $w-2$ `by` $-1$ `to 1 do`

`curbit :=` $\overline{Y}i$`;`

`if curbit = 1 then`

`Ans :=` $\text{Ans} \times X \mod D$ `;` ..... mod_red_2

`fi;`

`Ans :=` $(\text{Ans})^2 \mod D$`;` ..... mod_red_3

`od;`

`if` $(y_0\ 1)$ `then Ans =` $\text{Ans} \times X \mod D$`; fi;` ..... mod_red_4

The obvious speedup mechanism is to deploy the **QFS** algorithm to realize each modular-reduction, aka, remaindering operation. (the remaindering operations needed in modular-exponentiation are tagged with the label "mod_red_$n$" inside a box at the end of the corresponding line in the maple-style pseudo-code above).

### § 4·6·1 Further optimization: Avoiding Sign-Detection at the end of QFS

**Result 3 :** *Directly using the estimate* $\widehat{\overline{Q}}$ *to evaluate* $\widehat{\overline{R}}$ *as a residue-touple (as per Eqn (109) above), corresponds to an estimated integer-remainder* $\hat{R}$ *that is in the same residue class (w.r.t. the Divisor* $D$*) as the correct remainder* $R$

**Proof :** Immediately follows from the definition of the residue class:

*Definition 1 : Integers* $p$ *and* $q$ *are in the same residue class w.r.t.* $D$ *iff* $(p \mod D\ \ q \mod D)$

Eqns (111) and (112) show that $\hat{R} \in \{R, R\ D\} \Rightarrow$, it is in the same residue-class as the exact integer remainder $R$. □

Next, we show that as long as the range of the RNS system is sufficiently large, it is possible to use incorrect values for the remainder at intermediate steps of modular exponentiation, (as long as they are in the proper residue class); and still generate the correct final result.

**Result 4 :** *If the inputs* $X_1$ *and* $X_2$ *to the* **QFS** *algorithm are in the same residue class w.r.t. the (constant/known) divisor* $D$ *then the remainder estimates* $\hat{R}_1$ *and* $\hat{R}_2$ *evaluated using the quotient estimates* $\hat{Q}_1$ *and* $\hat{Q}_2$ *returned by the* **QFS** *algorithm both satisfy the constraints*

$$\hat{R}_1 \text{ can assume only one of the two values } : \hat{R}_1\ \ R \text{ or } \hat{R}_1\ \ R\ D \tag{115}$$

$$\hat{R}_2 \text{ can assume only one of the two values } : \hat{R}_2\ \ R \text{ or } \hat{R}_2\ \ R\ D \tag{116}$$

*where* $R$ *is the correct/exact integer remainder. (this holds even if the "Q_is_exact" flag is set to 0, indicating that the algorithm could not determine whether or not the quotient estimate equals the exact quotient).*

**Result 5 :** *If the range of the RNS is sufficiently large, then there is no need for a sign-detection at the end of the* **QFS** *algorithm in order to identify the correct remainder in intermediate steps during the modular-exponentiation operation.*

**Proof :** Assume that at the end of some intermediate step $i$, $\hat{Q}\ \ Q-1$ thereby causing

$$\text{Ans}_i\ \ \hat{R}_i\ \ R_i\ D \quad \text{instead of the correct value} \quad \text{Ans}_i\ \ \hat{R}_i\ \ R_i; \tag{117}$$

Then, as seen in the pseudo-code for modular exponentiation (which is illustrated in Section § 4.6.2) above, the next operation is either a modular-square or a modular-multiplication :

$$\texttt{Ans}_{i1}\ \texttt{Ans}_i{}^2 \mod D \qquad \text{or} \qquad \texttt{Ans}_{i1}\ \texttt{Ans}_i \times X \mod D \Rightarrow \tag{118}$$

$$\texttt{Ans}_{i1}\ R_i\ D^2 \mod D \qquad \text{or} \qquad \texttt{Ans}_{i1}\ R_i\ D \times X \mod D \tag{119}$$

instead of the correct values

$$\texttt{Ans}_{i1}\ R_i^2 \mod D \qquad \text{or} \qquad \texttt{Ans}_{i1}\ R_i \times X \mod D$$

However, note that

$$R_i^2 \text{ is in the same residue class w.r.t. } D \text{ as } R_i\ D^2 \qquad \text{and} \tag{120}$$

$$R_i\ D \times X \text{ is in the same residue class w.r.t. } D \text{ as } R_i \times X \tag{121}$$

Therefore from claim 2 above, it follows that in either paths (modular-square or modular-product-by-$X$) the answers obtained at the end of the next step satisfy the exact same constraints, specified by Equations (115) and (116), independent of whether the answers (remainders) at the end of the previous step were exact or had an extra $D$ in them; which shows that performing a sign-detection on the $\hat{Q}$ returned by the **QFS** algorithm is not necessary. □

**Result 6 :** *A single precision RNS range $\geq 3D$, and correspondingly a double-precision range $\geq 9D^2$ is sufficient to obviate the need for a sign-detection*

**Proof :** Since the correct remainder satisfies the constraints $0 \leq R < D$, it is clear that the erroneous remainder value $R\ D$ satisfies

$$0 < D \leq R\ D < 2D \tag{122}$$

As a result, the estimated remainder could be as high as about/almost $2D$. We therefore set the single-precision range-limit to be $3D$ so that the full double length values could be as large as $3D^2$ $9D^2$. Accordingly, we select $K$-smallest-consecutive prime numbers such that their product exceeds $9D^2$. With this big a range, either modular-square or modular-multiplication using an inexact remainder does not cause overflow, as per constraint (122) above □

### § 4.6.2 The ME-FWRD algorithm: maple-style pseudo-code

# First we specify a procedure ("proc" in maple) which is a small wrapper around the **QFS** algorithm

**QFS _rem_estimate := proc(**$\overline{\texttt{X}}, \texttt{X} \mod m_e$**)**

$\hat{\overline{\texttt{Q}}}$, $\hat{\texttt{Q}}$_mod_me, Q_is_exact := Quotient___First___Scaling__Estimate ($\overline{\texttt{X}}$, $\langle X \mod m_e \rangle$)

R_is_exact := Q_is_exact; # *if $\hat{Q}$ is exact then so is $\hat{R}$*

$\hat{\texttt{R}}$_mod_me := [X $\mod m_e -$ $\hat{\texttt{Q}}$_mod_me $\times D \mod m_e$] $\mod m_e$; # *bootstrapping...*

$\hat{\overline{\texttt{R}}}$ $\overline{\texttt{X}}$ $\boxed{-}$ $\hat{\overline{\texttt{Q}}}$ $\boxed{\times}$ $\overline{\texttt{D}}$;

Return($\hat{\overline{\texttt{R}}}$, $\hat{\texttt{R}}$_mod_me, R_is_exact);

**end proc ;**

**Algorithm ModExp__Fully__Within__Residue__Domain** ($\overline{\texttt{X}}, \texttt{X} \mod m_e, \overline{\texttt{Y}}$)

**# Inputs :** $X$ as a residue-touple, the extra-info, and $Y$ as a $w$-bit binary-number

*# We assume that the constraint X < M has been enforced before converting the primary input X into*
*# a residue-touple*
**# Pre-computations :** moduli where $\mathcal{M} \geq 9D^2$, D_mod_me, $\overline{D} \equiv$ residue-touple for $D, \cdots$
# and everything required by the **QFS** algorithm

**# Initializations**
$\overline{Ans}$, Ans_mod_me, Ans_is_exact:= qfs_rem_estimate($\overline{X}$, $\langle X \mod m_e \rangle$);

$\overline{Ans}$:= $\overline{Ans}$ $\boxed{\times}$ $\overline{Ans}$;
Ans_mod_me:= Ans_mod_me$^2 \mod m_e$ *# bootstrapping...*
$\overline{Ans}$, Ans_mod_me, Ans_is_exact:= qfs_rem_estimate($\overline{Ans}$, Ans_mod_me) ;

for i from $w-2$ by $-1$ to 1 do *# Loop*
    curbit := $\overline{Y}i$;
    if curbit = 1 then
        $\overline{Ans}$:= $\overline{Ans}$ $\boxed{\times}$ $\overline{X}$; Ans_mod_me:= (Ans_mod_me $\times$ X_mod_me) $\mod m_e$;
        $\overline{Ans}$, Ans_mod_me, Ans_is_exact:= qfs_rem_estimate($\overline{Ans}$, Ans_mod_me) ;
    fi;
    $\overline{Ans}$:= $\overline{Ans}$ $\boxed{\times}$ $\overline{Ans}$; Ans_mod_me:= Ans_mod_me$^2 \mod m_e$ ;
    $\overline{Ans}$, Ans_mod_me, Ans_is_exact:= qfs_rem_estimate($\overline{Ans}$, Ans_mod_me) ;
od;

if ($y_0$ 1) then *# Ans =* Ans $\times$ $X \mod D$
    $\overline{Ans}$:= $\overline{Ans}$ $\boxed{\times}$ $\overline{X}$; Ans_mod_me:= (Ans_mod_me $\times$ X_mod_me) $\mod m_e$ ;
    $\overline{Ans}$, Ans_mod_me, Ans_is_exact:= qfs_rem_estimate($\overline{Ans}$, Ans_mod_me) ;
fi;

**# Outputs :** *remainder-touple, extra-info, exactness-flag*

Return($\overline{Ans}$, Ans_mod_me, Ans_is_exact); □

**End__Algorithm**

Correctness of the algorithm follows from the analytical results presented so far. Moreover the algorithm was implemented in Maple and extensively tested on a large number of cases.

### § 4.6.3 Delay Estimation of the Proposed Modular-Exponentiation Algorithm

Pre-computation costs are not considered (they represent one-time fixed costs).
(i) The main/dominant delay is determined by the delay of the loop.
Assuming that the exponent $Y$ is about as big as $D$,
the number of times the exponentiation loop is executed = $\lg Y \approx On$ times.

(ii) Determination of the Quotient estimate is the most time-consuming operation in each iteration of the loop and it requires $O\lg n$ delay (as explained in Section § 4.5.4-A).
As a result, each iteration of the loop requires ($O\lg n$ delay.

(iii) Therefore, the total delay is $On \lg n$.

The **memory requirements** are exactly the same as those of the **QFS** algorithm : $\approx On^3 \lg\lg n$ bits as shown

above (in Section § 4.5.4-B).

### § 4·6·4 Some Remarks about the `ME-FWRD` algorithm

① In a remaindering operation, it is possible to *under-estimate* the quotient, but it is not acceptable to over-estimate the quotient even by a **ulp** for the following reason:

if $\hat{Q}$ is an over-estimate, then $\hat{R}\ X - \hat{Q} \times D \leq 0$ and therefore gets evaluated as (123)

$\hat{R} \equiv \mathcal{M} - |\hat{R}|$ **which is not in the same residue class w.r.t. D as the correct remainder** $R$ (124)

② The algorithm always works in full (double) precision mode. In the RNS, increased word length simply requires some more channels. In a dedicated h/w implementation, all the channels can execute concurrently, fully leveraging the parallelism inherent in the system. Hence, the incremental delay (as a result of doubling the word-length) is minimal: Since doubling the word-length adds one-level to each adder/accumulation-tree (within each RNS-channel),
the incremental delay is $\approx O1$.

## § 4.7 Convergence division via reciprocation to handle arbitrary, dynamic divisors

let $X$ be a $2n$ bit dividend

$$X\ \ X_u \cdot 2^n\ X_l \tag{125}$$

where $X_u$ is the upper-half (more-significant $n$ bits) and $X_l$ is the lower-half. Let $D$ be an $n$ bit-long divisor. Then, the quotient $Q$ is

$$Q\ \left\lfloor \frac{X}{D} \right\rfloor\ \left\lfloor \frac{X_u \cdot 2^n\ X_l}{D} \right\rfloor\ \left\lfloor \frac{X_u \cdot 2^n}{D} \right\rfloor\ \left\lfloor \frac{X_l}{D} \right\rfloor \delta \quad \text{wherein} \quad \delta\ \{0,1\} \tag{126}$$

Since $X_l$ and $D$ are both $n$ bit long numbers $X_l < 2D \quad \Rightarrow$ (127)

$$\left\lfloor \frac{X_l}{D} \right\rfloor \begin{cases} 0 & \text{if } X_l < D \\ 1 & \text{otherwise} \end{cases} \tag{128}$$

The remaining term is

$$\left\lfloor \frac{X_u \cdot 2^n}{D} \right\rfloor \quad \text{wherein} \quad \frac{X_u \cdot 2^n}{D}\ \frac{X_u}{D_f} \quad \text{where} \quad D_f\ \frac{D}{2^n} \quad \Rightarrow \quad \frac{1}{2} \leq D_f < 1 \tag{129}$$

In the inequality above, the lower bound $\frac{1}{2}$ follows from the fact that the leading bit of an $n$-bit long number $D$ is 1 (if not, the word-length of $D$ would be smaller than $n$). Also note that the maximum value of the $n$-bit integer $D$ can be $2^n - 1$, which yields the upper bound 1 on $D_f$.

Let $D_{\text{f\_inv}}\ \frac{1}{D_f} \quad \Rightarrow 1 < D_{\text{f\_inv}} \leq 2$ and let $D_{\text{f\_inv}}\ 1\ F$ where $0 < F \leq 1$ (130)

Then, $\frac{X_u}{D_f}\ x_u \times D_{\text{f\_inv}}\ X_u 1\ F\ X_u\ X_u F \quad \Rightarrow \quad \left\lfloor \frac{X_u}{D_f} \right\rfloor\ X_u\ \lfloor X_u F \rfloor$ (131)

From the last equality it is clear that in order to correctly evaluate $\lfloor X_u D_{\text{f\_inv}} \rfloor$, the value of $F$ (which is the

fractional part of $D_{\text{f\_inv}}$) must be evaluated upto at least $n$ bits of precision.

$$\text{To evaluate } D_{\text{f\_inv}}, \text{ let } Y\ 1-D_f \Rightarrow 0<Y\leq\frac{1}{2} \tag{132}$$

$$\text{Note that the integer } Y_{\text{int}}\ 2^nY\ 2^n-D \tag{133}$$

$$\tag{134}$$

Substituting $D_f$ in terms of $Y$, yields

$$D_{\text{f\_inv}}\ \frac{1}{D_f}\ \frac{1}{1-Y}\ \frac{1\,Y}{1-Y^2}\ \frac{1\,Y1\,Y^2}{1-Y^4}\ \cdots\ \frac{1\,Y1\,Y^2\cdots1\,Y^{2^t}}{1-Y^{2^{2t}}} \tag{135}$$

In the last set of equalities, since the numerator and the denominator of each successive expression are both multiplied by the same

$$\text{factor of the form}\quad \left(1\ Y^{2^i}\right)\qquad \text{at step}\quad i,\ i\ 0,1,\cdots \tag{136}$$

the original value of the reciprocal does not change. Also note that each successive multiplication by a factor of the above form doubles the number of leading ones in the denominator. As a result the denominator in the successive expressions in Eqn (135) approaches the value 1 from below (it becomes $0.11111\cdots$).
It is well known [4] that
when the number of leading-ones in the denominator exceeds the word-length (i.e., $n$ bits),
the error $\epsilon$ in the numerator also satisfies the bound $|\epsilon|<2^{-n}$
and the iterations can be stopped. In other words, when $t$ leads to the satisfaction of the constraint

$$1-Y^{2^{2t}}\leq 1-2^n\quad\Rightarrow\quad \lg Y2^{2t}\geq n\quad\Rightarrow\quad t\geq\frac{1}{2}(\lg n-\lg\lg Y) \tag{137}$$

the iterations can be stopped and the approximation

$$D_{\text{f\_inv}}\ \frac{1\,Y1\,Y^2\cdots1\,Y^{2^t}}{1-Y^{2^{2t}}}\approx\frac{1\,Y1\,Y^2\cdots1\,Y^{2^t}}{1} \tag{138}$$

can be used. Thus, number of iterations in a convergence division is $O\lg n$.
In contrast any digit-serial division fundamentally requires $On$ steps.

It turns out that the above convergence method is equivalent to newton-style convergence iterations (for details, please see any textbook on Computer Arithmetic [4, 5]). Newton's method is quadratic which means that the error

$$\epsilon_{n1}\ \text{ after the } n\ 1\text{th iteration }\ \approx O{\epsilon_n}^2 \tag{139}$$

which in-turn implies that the number of accurate bits doubles after each iteration (which is why convergence-division is the method of choice in high speed implementations).

From (138) it is clear that

$$D_{\text{f\_inv}}\approx 1\ Y1\ Y^2\cdots1\ Y^{2^t}$$

Accordingly the products need to be accumulated, so as to yield a precision of $2^n$-bits at the end. Since a product of two $n$ bit numbers (which includes a square) can be upto $2n$ bits long, the lower half of the double length product must be discarded retaining only the $n$ most significant bits at every step. Each such retention

of $n$ most significant bits is tantamount to division by a constant, viz., $2^n$. Thus the **QFS** algorithm needs to be invoked at every step in the convergence division method. In general the **SODIS** algorithm is also needed at each step. Those familiar with the art will appreciate that using all the preceding algorithms unveiled herein, an ultrafast convergence division via reciprocation can be realized without ever having to leave the residue domain at any intermediate step.

## § 5 Application Scenarios

The difficulty of implementing some basic canonical operations such as base-extension, sign-detection, scaling, and division has prevented the widespread adoption of the RNS. The algorithms and apparata unveiled herein streamline and expedite all these fundamental operations, thereby removing all roadblocks and unleashing the full potential of the RNS. Since a number representation is a fundamental attribute that underlies all computing systems, expediting all arithmetic operations using the RNS can potentially affect all scenarios that include computing systems. The scenarios that are most directly impacted are listed below.

① At long wordlengths the proposed system yields substantially faster implementations. Therefore, cryptographic processors are likely to adopt the RNS together with the algorithms unveiled in this document. All other long wordlength applications (such as running Sieves for factoring large numbers or listing the prime numbers within a given interval, etc) will substantially benefit from hardware as well as software implementations of the proposed number system and the accompanying algorithms.

② Digital Signal Processing is dominated by multiply and add operations. The proposed representation is therefore likely to be adopted in DSP processors.
Scaling is particularly easy if the scaling factor is divisible by one or more of the component moduli. My method of selecting moduli uses all prime numbers up to a certain threshold value. So there is ample scope to select a scale factor that is divisible by one or more of the moduli. This is an added advantage of the method of moduli selection that I have adopted.

③ Ultra-fast counters, constant-time or wordlength-independent up/down counters are significantly faster as well as simpler to realize when the RNS system is used.
Consequently, the theory as well as designs of such counters should switch over to using the RNS and the accompanying algorithms.

④ Memory and cache organization/access is another potentially significant application area. Conceptually, memory needs be abstracted as though it were a "linear" storage because the indexing calculations in conventional (binary/decimal) number representations are easier when the memory is logically organized linearly. Adopting the RNS allows a different conceptual organization of storage resources (ex: under RNS, storage can be conceptualized as a collection of buckets)

⑤ Realization of hash functions is faster and easier when there is native/hardware support for modulo operations that are required in the RNS. That in turn opens up other possibilities to further streamline and expedite other algorithms and/or apparata (such as Bloom filters, for instance).

⑥ Coding Theory and practice have pretty much revolved around conventional number representations. RN systems offer a rich mix of choices to further improve coding theory and practice.

⑦ Hardware implementations based on the RNS are inherently more parallel, since all channels can do their processing independently, thereby increasing the "locality" of processing and drastically reducing long interconnects. This in turn makes the circuits more compact and faster while simultaneously requiring substantially lower amount of power (than equivalent circuits based on conventional number

representations). Moreover, the independence of channels makes the hardware lot easier to test (VLSI testing is a critically important issue). Finally, hardware realizations based on the RNS are more reliable.

⑧ Even when the RNS is implemented in software, it can still improve the utilization of the multi-core processors today. Channel(s) in the RNS could be dynamically mapped onto threads which could in turn be dynamically allocated and executed on any of the multiple cores.

⑨ Random number generation based on RNS system is another area that appears to have a great potential.

⑩ Theoretical Issues: for instance the Remainder Theorem constitutes an "orthogonal" decomposition (in a sense analogous to the Discrete Fourier Transform, i.e., the DFT). This is why multiplication (which is a convolution) becomes so simple, all the cross-terms go away....
What kind of redundancy is there in RNS representation ?
The novelty of the methods unveiled herein lies in their use of both the integer as well as fractional parts rather than sticking to only one. Can such methods be further extended and applied to other well know hard problems ? are these methods related to "Interior Point Methods"?

## 5.1 Distinctions and novel aspects of this invention

① All of the algorithms make maximal use of those intermediate variables whose values can be expressed as *the most significant digits* of a computation (the reader can verify that this is the case in the **"RPPR"** , **SODIS** , as well as the **QFS** algorithm).
This enables the use of *approximation* methods.

② *Accuracy of approximation is in turn related to the precision required.* The algorithms therefore use the minimal amount of precision necessary for the computation.
I leverage the rational domain interpretation (i.e., joint integer as well as fractional domain interpretations) of the Chinese Remainder Theorem in order to drastically reduce the precision of the fractional values that need to be pre-computed and stored in look-up tables. It turns out that a drastic reduction of precision from $n$-bits to $\lceil \lg n \rceil$ bits still allows a highly accurate estimation of some canonical intermediate variables, wherein the estimate can be off only by a **ulp**. In other words, the exact value of the computation can be narrowed down to a pair of successive integers. This strategy is adopted in all the methods (viz, **RPPR**, **SODIS** , as well as the **QFS** algorithms)
The novel "disambiguation" step then selects the right answer(by disambiguating between the two choices) in all cases.
In other words, in a fundamental sense, I have identified the optimal mix of which and how much information from the fractional domain needs be combined with which specific portion of the information available from the integer-domain interpretation of the CRT in order to achieve ultrafast execution; and developed new methods that fully exploit that optimal mix.

③ My Moduli selection method simultaneously achieves three optimizations:
O1 : It maximizes the amount of pre-computation to the fullest extent; making it possible to deploy *exhaustive look-up tables* that cover all possible input cases.

O2 : Simultaneously, it also minimizes the amount of memory required (otherwise an exhaustive look-up would not be feasible at long bit-lengths).

O3 : It minimizes the size that each individual component-modulus $m_i$ can assume. The net effect is that the RNS is realized via a moderately large number of channels, each of which has a very small modulus.
In other words, the moduli-selection brings out the parallelism inherent in the RNS to the fullest

extent.

④ The said moduli selection therefore leads to two critical benefits

B+1 : Exhaustive pre-computation implies that there is very little left to be done at run-time, which leads to ultrafast execution (a good example is the Quotient First Scaling ( **QFS** ) algorithm).

B+2 : Exploiting the parallelism to the fullest extent while also using the smallest amount of memory further speeds up execution and cuts down on area and power consumption of hardware realizations.

⑤ All of the prior works published hitherto had a narrower focus. For example, [7], [10] (and their derivatives that have appeared since) are mainly concerned with "base-extension". On the other hand, Vi's first paper [14] was aimed at "fault detection in flight control systems"; while the follow-on journal paper [15] was focused on "sign-detection". Likewise, Lu's work [23, 24] was mainly focused on a more efficient sign-detection and its application to division in the RNS.
In contrast, we have developed a unified framework that expedites *all* the difficult RNS operations simultaneously.

⑥ The algorithms can be implemented in software (wherein the computation within each channel is done within a separate thread and the multiple threads get dynamically mapped onto different cores in a multi-core processor) or in hardware. In either case they offer a wide spectrum of choices that trade-off *polynomially* increasing amounts of pre-computations and look-up-table memory to achieve higher speed. In other words the algorithms are flexible and allow the designer a wide array of choices for deployment.

FIG. 9 is a flow chart representation of a process 900 of performing reconstruction using a residue number system. At box 902, a set of moduli is selected. At box 904, a reconstruction coefficient is estimated based on the selected set of moduli. At box 906, a reconstruction operation using the reconstruction coefficient is performed. As previously discussed, in some designs, additional operations may also be performed using the reconstruction operation.

In some designs, the operation of selecting the set of moduli is done so as to enable an exhaustive pre-computation and look-up strategy that covers all possible inputs. In some designs, the determination of reconstruction coefficient may be performed in hardware such that the determination is upper limited by delay of O(log n) where n is an integer number representing wordlength.

FIG. 10 is a block diagram representation of a portion of an apparatus 1000 for performing reconstruction using a residue number system. The module 1002 is provided for selecting a set of moduli. The module 1004 is provided for estimating a reconstruction coefficient based on the selected set of moduli. The module 1004 is provided for performing a reconstruction operation using the reconstruction coefficient.

FIG. 11 is a flow chart representation of a process 1100 of performing division using a residue number system. At box 1102, a set of moduli is selected. At box 1104, a reconstruction coefficient is determined. At box 1106, a quotient is determined using an exhaustive pre-computation and a look-up strategy that covers all possible inputs.

FIG. 12 is a block diagram representation of a portion of an apparatus 1200 for performing division using a residue number system. The module 1202 is provide for selecting a set of moduli. The module 1204 is provided for determining a reconstruction coefficient. The module 1206 is provided for determining a quotient using an exhaustive pre-computation and a look-up strategy that covers all possible inputs.

FIG. 13 is a flow chart representation of a process 1300 of computing a modular exponentiation using

a residue number system. At box 1302, iterations are performed without converting to a regular integer representation, by performing modulator multiplications and modular squaring. At box 1304, the modular exponentiation is computed as a result of the iterations.

FIG. 14 is a block diagram representation of a portion of an apparatus 1400 for computing a modular exponentiation using a residue number system. The module 1402 is provided for iterating, without converting to a regular integer representation, by performing modular multiplications and modular squaring. The module 1404 is provided for computing the modular exponentiation as a result of the iterations.

It is noted that in one or more exemplary embodiments described herein, the functions and modules described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored on or transmitted over as one or more instructions or code on a computer-readable medium. Computer-readable media includes both computer storage media and communication media including any medium that facilitates transfer of a computer program from one place to another. A storage media may be any available media that can be accessed by a computer. By way of example, and not limitation, such computer-readable media can comprise RAM, ROM, EEPROM, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that can be used to carry or store desired program code in the form of instructions or data structures and that can be accessed by a computer. Also, any connection is properly termed a computer-readable medium. For example, if the software is transmitted from a website, server, or other remote source using a coaxial cable, fiber optic cable, twisted pair, digital subscriber line (DSL), or wireless technologies such as infrared, radio, and microwave, then the coaxial cable, fiber optic cable, twisted pair, DSL, or wireless technologies such as infrared, radio, and microwave are included in the definition of medium. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk and blue-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above should also be included within the scope of computer-readable media.

As utilized in the subject disclosure, the terms ąřsystem,ąś ąřmodule,ąś ąřcomponent,ąś ąřinterface,ąś and the like are likewise intended to refer to a computer-related entity, either hardware, a combination of hardware and software, software, or software in execution. Components can include circuitry, e.g., processing unit(s) or processor(s), that enables at least part of the functionality of the components or other component(s) functionally connected (e.g., communicatively coupled) thereto. As an example, a component may be, but is not limited to being, a process running on a processor, a processor, a machine-readable storage medium, an object, an executable, a thread of execution, a program, and/or a computer. By way of illustration, both an application running on a computer and the computer can be a component. One or more components may reside within a process and/or thread of execution and a component may be localized on one computer and/or distributed between two or more computers.

The aforementioned systems have been described with respect to interaction between several components and modules. It can be appreciated that such systems, modules and components can include those components or specified sub-components, some of the specified components or sub-components, and/or additional components, and according to various permutations and combinations of the foregoing. Sub-components also can be implemented as components communicatively coupled to other components rather than included within parent component(s). Additionally, it should be noted that one or more components may be combined into a single component providing aggregate functionality or divided into several separate sub-components and may be provided to communicatively couple to such sub-components in order to provide integrated functionality. Any components described herein may also interact with one or more other components not specifically described herein but generally known by those of skill in the art.

Moreover, aspects of the claimed subject matter may be implemented as a method, apparatus, or article

of manufacture using standard programming and/or engineering techniques to produce software, firmware, hardware, or any combination thereof to control a computer or computing components to implement various aspects of the claimed subject matter. The term "article of manufacture" as used herein is intended to encompass a computer program accessible from any computer-readable device, carrier, or media. For example, computer readable media can include but are not limited to magnetic storage devices (e.g., hard disk, floppy disk, magnetic strips, optical disks (e.g., compact disk (CD), digital versatile disk (DVD), smart cards, and flash memory devices (e.g., card, stick, key drive. Additionally it should be appreciated that a carrier wave can be employed to carry computer-readable electronic data such as those used in transmitting and receiving voice mail or in accessing a network such as a cellular network. Of course, those skilled in the art will recognize many modifications may be made to this configuration without departing from the scope or spirit of what is described herein.

What has been described above includes examples of one or more embodiments. It is, of course, not possible to describe every conceivable combination of components or methodologies for purposes of describing the aforementioned embodiments, but one of ordinary skill in the art may recognize that many further combinations and permutations of various embodiments are possible. Accordingly, the described embodiments are intended to embrace all such alterations, modifications and variations that fall within the spirit and scope of the appended claims. Furthermore, to the extent that the term ąřincludesąś is used in either the detailed description or the claims, such term is intended to be inclusive in a manner similar to the term ąřcomprisingąś as ąřcomprisingąś is interpreted when employed as a transitional word in a claim.

## § 6 Claims

what is claimed is:

1. A method of performing reconstruction using a residue number system, comprising;

   selecting a set of moduli;

   estimating a reconstruction coefficient based on the selected set of moduli; and

   performing a reconstruction operation using the reconstruction coefficient.

2. The method of claim 1, wherein the selecting the set of moduli is done so as to enable an exhaustive pre-computation and look-up strategy that covers all possible inputs.

3. The method of claim 1, wherein the reconstruction coefficient is determined in a delay limit of $O\log n$.

4. The method of claim 1 wherein the estimating comprises:

   computing a plurality of reconstruction reminders; and

   quantizing the plurality of reconstruction reminders.

5. The method of claim 4 wherein the quantization comprises:

   expressing the reconstruction remainders as proper fractions;

   pre-computing the proper fractions in a pre-determined radix b;

   truncating the proper fractions to a precision of no more than $(\lceil \log_b \log_b \mathcal{M} \rceil)$ radix-$b$ fractional digits;

   scaling the truncated proper fractions by a scale factor so that multiplication by the scale factor simply amounts to a left-shifting of base-b digits and yields an integer value; and

   storing the resulting integer values in look-up tables, wherein each RNS channel $i$ with component-modulus $m_i$ requires one look-up table with $(m_i - 1)$ entries.

6. The method of claim 5, wherein, channel look-up tables are read-only and are accessed completely independently of one another

7. The method of claim 1 wherein all the operands are integers and all the arithmetic operations are carried out with an ultra-low precision of no more than $(\lceil \log_b K \rceil \lceil \log_b log_b M \rceil)$ radix-$b$ digits.

8. The method of claim 1 wherein the estimate consists of a pair of consecutive integers, one of which is the correct value of the reconstruction coefficient.

9. The method of claim 8, wherein, a disambiguation step is required to select the correct answer from among the two choices, by using an independent extra bit of information which is maintained in the form of one extra residue, i.e., remainder, with respect to an extra "disambiguator–modulus" $m_e$ that satisfies the condition: $\gcd\mathcal{M}, m_e < m_e$

10. The method of claim 9, wherein, a systematic "disambiguation–bootstrapping" process is required (and is therefore adopted) to ensure that this extra remainder is always available for any value that the method encounters.

11. A method of performing division using a residue number system, comprising:

    selecting a set of moduli;

    determining a reconstruction coefficient; and

    determining a quotient using an exhaustive pre-computation and a look-up strategy that covers all possible inputs.

12. The method of claim 11, wherein, the disambiguation bootstrapping information regarding the determined quotient Q is also computed.

13. A method of computing a modular exponentiation in a residue number system, comprising:

    iterating, without converting to a regular integer representation, by performing modular multiplications and modular squaring;

    computing the modular exponentiation as a result of the iterations.

14. The method of claim 13, wherein there is no conversion between distinct moduli sets within a residue domain at any intermediate step throughout the computing process.

15. An apparatus for performing reconstruction using a residue number system, comprising:

    means for selecting a set of moduli;

    means for estimating a reconstruction coefficient based on the selected set of moduli; and

    means for performing a reconstruction operation using the reconstruction coefficient.

16. A computer program product comprising a non-volatile, computer-readable medium, storing computer-executable instructions for performing reconstruction using a residue number system, the instructions comprising code for:

selecting a set of moduli;

estimating a reconstruction coefficient based on the selected set of moduli; and

performing a reconstruction operation using the reconstruction coefficient.

## § 7 BRIEF ABSTRACT

A method for performing reconstruction using a residue number system includes selecting a set of moduli. A reconstruction coefficient is estimated based on the selected set of moduli. A reconstruction operation is performed using the reconstruction coefficient.

# References


[1] N. Szabo and R. Tanaka, *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[2] A. Omondi and B. Premkumar, *Residue number systems: theory and implementation*. World Scientific Pub Co Inc, 2007.

[3] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms, Second Edition*. The MIT Press and McGrawHill, 2001.

[4] I. Koren, *Computer Arithmetic Algorithms*. 2nd Edition, A K Peters Publishers, Natic, Massachusetts, 2002.

[5] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*. Oxford University Press, 2000.

[6] Milos D. Ercegovac and Tomas Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.

[7] A. Shenoy and R. Kumaresan, "Fast base extension using a redundant modulus in RNS," *IEEE Transactions on Computers*, pp. 292–297, 1989.

[8] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "Modular multiplication and base extensions in residue number systems," in *In Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pp. 59–65, IEEE, 2001.

[9] M. Stojancic, M. Maddury, and K. Tomei, "Residue number system based pre-computation and dual-pass arithmetic modular operation approach to implement encryption protocols efficiently in electronic integrated circuits," Apr. 11 2006. US Patent 7,027,598.

[10] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, "Cox-rower architecture for fast parallel montgomery multiplication," *Lecture Notes in Computer Science*, pp. 523–538, 2000.

[11] S. Kawamura, "Modular arithmetic apparatus and method having high-speed base conversion function," Dec 2003. US Patent 6,662,201.

[12] S. Kawamura, "Modular arithmetic apparatus and method having high-speed base conversion function," Sep 2004. US Patent 6,807,555.

[13] M. Shiba and S. Kawamura, "Arithmetic method and apparatus and crypto processing apparatus for performing multiple types of cryptography," Oct. 2 2007. US Patent 7,277,540.

[14] T. VU, "The use of residue arithmetic for fault detection in a digital flight control system," *NAECON 1984*, pp. 634–638, 1984.

[15] T. Van Vu, "Efficient implementations of the Chinese remainder theorem for sign detection and residue decoding," *IEEE Transactions on Computers*, vol. 100, no. 34, pp. 646–651, 1985.

[16] D. Banerji and J. Brzozowski, "Sign detection in residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 18, pp. 313–320, 1969.

[17] M. Akkal and P. Siy, "Optimum RNS sign detection algorithm using MRC-II with special moduli set," *Journal of Systems Architecture*, vol. 54, no. 10, pp. 911–918, 2008.

[18] T. Tomczak, "Fast Sign Detection for RNS $(2^n - 1, 2n, 2^n\ 1)$," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, pp. 1502–1511, July 2008.

[19] I. Akushskii, V. Burcev, and I. Pak, "A new positional characteristic of nonpositional codes and its applications," *Coding Theory and the Optimization of Complex Systems, Nauka, Kazakhstan*, 1977.

[20] N. Burgess, "Scaled and unscaled residue number system to binary conversion techniques using the core function," in *Proc. of 13th IEEE Symp. on Computer Arithmetic (ARITH-13)*, pp. 250–257, 1997.

[21] N. Burgess, "Scaling an RNS number using the core function," in *Proc. of the 16th IEEE Symposium on Computer Arithmetic*, pp. 262–269, 2003.

[22] M. Abtahi and P. Siy, "The factor-2 sign detection algorithm using a core function for rns numbers," *Computers & Mathematics with Applications*, vol. 53, no. 9, pp. 1455–1463, 2007.

[23] J. Chiang and M. Lu, "A general division algorithm for residue number systems," in *Proc. of the 10th IEEE Symposium on Computer Arithmetic*, pp. 76–83, 1991.

[24] M. Lu and J. Chiang, "A novel division algorithm for the residue number system," *IEEE Transactions on Computers*, pp. 1026–1032, 1992.

[25] G. Jullien, "Residue number scaling and other operations using ROM arrays," *IEEE Transactions on Computers*, vol. 100, no. 27, pp. 325–336, 1978.

[26] F. Taylor and C. Huang, "An autoscale residue multiplier," *IEEE Trans. on Computers*, pp. 321–325, 1982.

[27] A. Shenoy and R. Kumaresan, "An accurate scaling technique in improved residue number system arithmetic," in *Proc. of the IEEE ICASSP'87*, vol. 12, pp. 33.8.1–33.8.4, 1987.

[28] M. Shenoy and R. Kumaresan, "A fast and accurate RNS scaling technique for high speed signalprocessing," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37, no. 6, pp. 929–937, 1989.

[29] F. Barsi and M. Pinotti, "Fast base extension and precise scaling in RNS for look-up table implementations," *IEEE transactions on signal processing*, vol. 43, no. 10, pp. 2427–2430, 1995.

[30] U. Meyer-Base and T. Stouraitis, "New power-of-2 RNS scaling scheme for cell-based IC design," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, vol. 11, no. 2, pp. 280–283, 2003.

[31] S. Robinson and W. Chren Jr, "Arithmetic circuits for use with the residue number system," Jan 2007. US Patent 7,165,085.

[32] G. Cardarilli, A. Del-Re, A. Nannarelli, and M. Re, "Programmable power-of-two RNS scaler and its application to a QRNS polyphase filter," in *Proc. of the IEEE International Symposium on Circuits and Systems, ISCAS'05*, pp. 1102–1105, 2005.

[33] Y. Kong and B. Phillips, "Fast Scaling in Residue Number System," *IEEE Transactions on VLSI*, vol. 17, pp. 443–447, Mar. 2009.

[34] "Maple", well known software package for Math, Sciences and Engineering.
Vendor-web-site: http://www.maplesoft.com.

[35] The "Prime-counting function", nicely illustrated on wikipedia
http://en.wikipedia.org/wiki/Prime_counting_function.

[36] Primorial, defined and described on wikipedia: http://en.wikipedia.org/wiki/Primorial.

[37] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*. Springer, 2005.

[38] D. S. Phatak, "RNS- **ARDSP** : A novel, fast Residue Number System using **A**pproximate **R**ational **D**omain **S**caled **P**recomputations
Part I : Reduced Precision Partial Reconstruction ( **"RPPR"** ) Algorithm," *20th IEEE intl. symposium on Computer Arithmetic, (ARITH'20)*, 2011. under review.

[39] D. S. Phatak, T. Goff, and I. Koren, "Constant-time Addition and Simultaneous Format Conversion Based on Redundant Binary Representations," *IEEE Trans. on Computers*, vol. TC–50, pp. 1267–1278, Nov. 2001.

[40] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective, Second Edition*. Addison Wesley, 1993.

FIGURES AND DRAWINGS

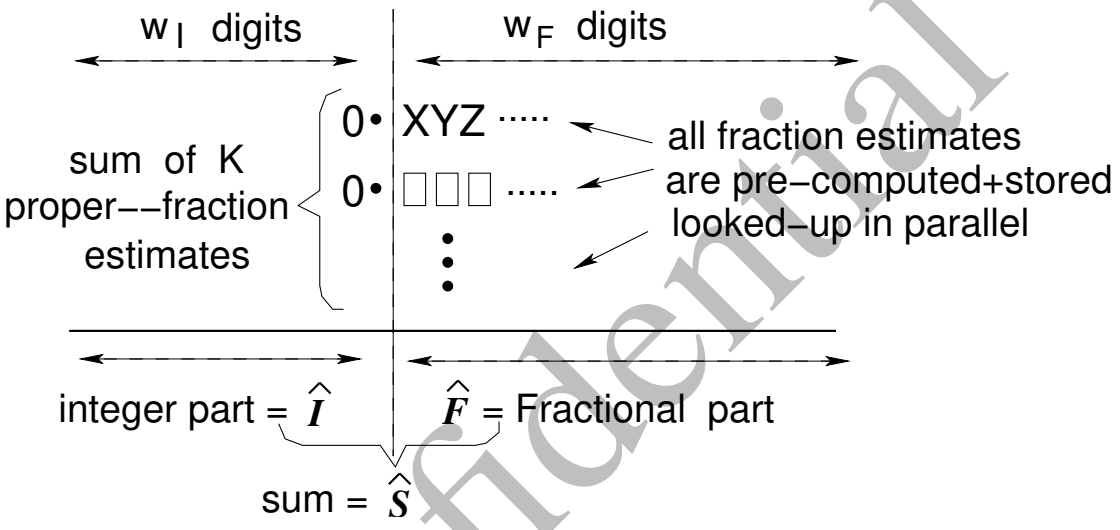


Figure 1: Summation of fraction estimates (obtained via look-up-tables) to estimate the **Reconstruction Coefficient** $\mathcal{R}_C$

Figure 2: Flow chart for the **"RPPR"** algorithm hand drawn, appears on the next page

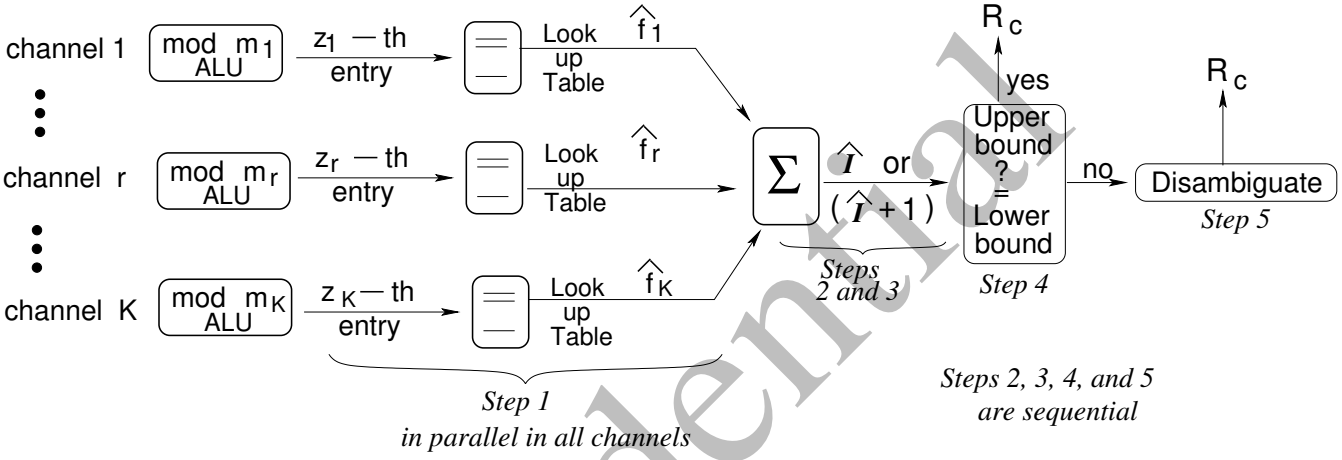


Figure 3: Schematic block diagram of a generic architecture to implement the **RPPR** algorithm. Below each block, we have indicated the step(s) of the algorithm that that block executes. "$\Sigma$" is a (fast, tree-based) multi-operand-adder. Extra h/w also includes a barrel shifter [40] to perform variable amounts of shifts.

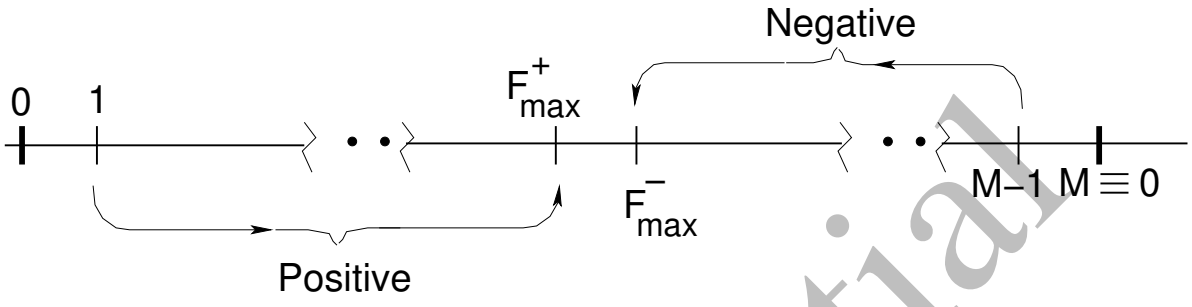


Figure 4: Conventional method of incorporating negative integers in the RNS. Note that in order to achieve the highest possible representational efficiency; $F_{\max}$and $F^{-}_{\max}$are mapped to consecutive integer values, yielding $F_{\max}\ 1\ F^{-}_{\max}$, so that all the unsigned integer values in the range $[0,\mathcal{M}-1]$ get utilized, not a single value is wasted. An unfortunate by product of this quest for high representational efficiency is that two consecutive integers corresponding to the last ve value and the first $-$ve value have opposite signs. As a result, distinguishing between those two integers requires full precision calculations. Recognizing the distinction between those two values is not feasible if the precomputed intermediate fractional values have the drastically limited precision of only $O\lg\lg\mathcal{M}$ digits.

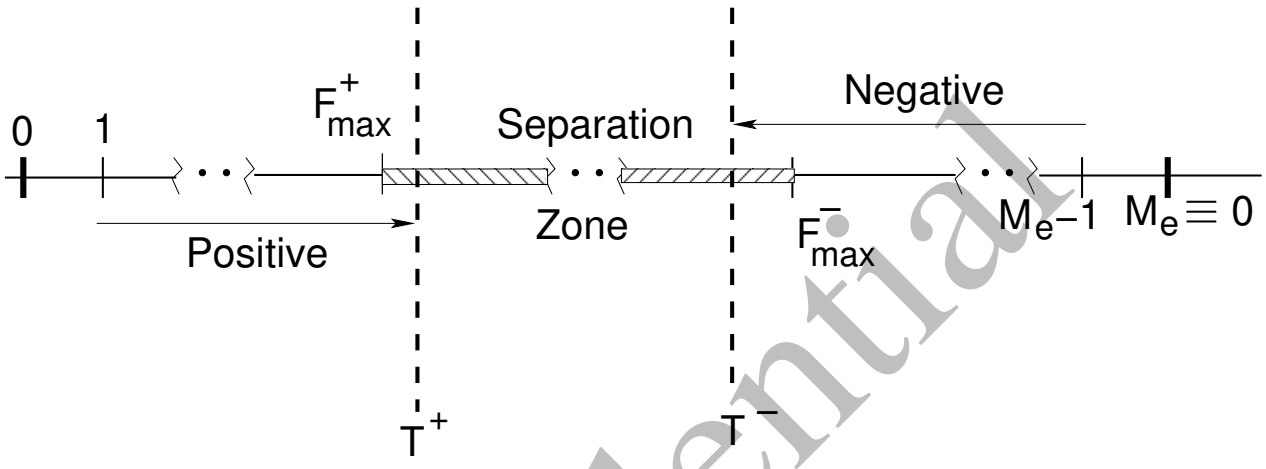


Figure 5: My method deliberately introduces a sufficiently large "separation zone" between regions corresponding to positive and negative values. In other words, I tradeoff representational efficiency in favor of substantially lower computational latency. In general (especially at smaller word-lengths) this can require increasing the total modulus $\mathcal{M}$ to a higher value $\mathcal{M}_e$ (by adding one extra component modulus to the original set of moduli if necessary). By properly adjusting the relative lengths of the 3 intervals (viz., the intervals containing
(i) zero and the ve integers;
(ii) the separation-zone; and
(iii) the $-ve$ integers),
it is possible to detect the sign correctly even when a drastically limited precision of only $O\lg\lg\mathcal{M}_e$ digits is used in the intermediate as well as the pre-computations.

Figure 6: Flow chart for the Quotient First Scaling ( **QFS** ) algorithm (next two pages)

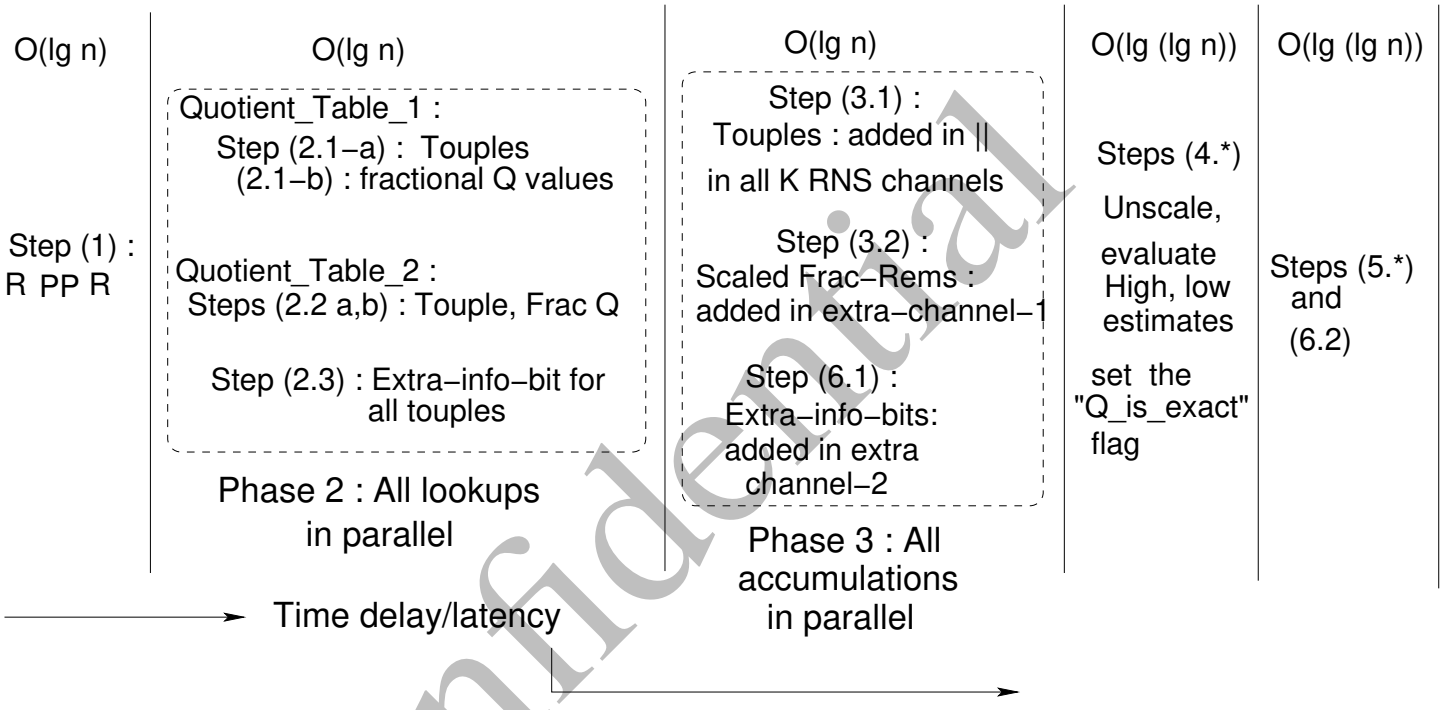


Figure 7: Schematic timing diagram for the **QFS** algorithm. Below each time-block, we have indicated the steps of the algorithm that are executed in that block/phase. At the top of each block we have indicated the latency, derived from the assumptions stated in Sections § 4.2.6 and § 4.5.4-A.

Figure 8: Flow chart for the modular exponentiation algorithm (next 3 sheets, hand drawn)