The Residue Logarithmic Number System: Theory and Implementation

Mark G. Arnold
Lehigh University
Bethlehem, PA 18015 USA
marnold@eecs.lehigh.edu

Abstract

The Residue Logarithmic Number System (RLNS) represents real values as quantized logarithms which, in turn, are represented using the Residue Number System (RNS). Compared to the conventional Logarithmic Number System (LNS) in which quantized logarithms are represented as binary integers, RLNS offers faster multiplication and division times. RLNS and LNS use a table lookup involving all bits for addition. The width, dynamic range, precision and naïve table size of RLNS (with careful moduli selection) is as good as those for conventional LNS.

Conventional LNS can be more efficient than naïve addition lookup. First, commutativity allows interchanging arguments. Second, the addition function is often essentially zero, and does not have to be tabulated. In binary, comparisons are easy. In residue, comparisons are slow. Although RLNS inherently demands comparison, this paper shows a novel way comparisons can be performed in parallel to the lookup from a small table. This paper also describes a novel tool that generates synthesizable Verilog, making RLNS viable in practical applications that can benefit from shorter multiply and divide times.

Keywords: Residue Number System, Logarithmic Number System.

1. Introduction

This paper considers a number system that is a composite of two unusual number systems: the Residue Number System (RNS) and the Logarithmic Number System (LNS). Both trace their roots to ancient mathematical innovations: the Chinese Remainder Theorem for RNS [21] and the Gauss-Leonelli addition logarithm [3] for LNS. Interest in such unusual systems continues due to Application-Specific Processors (ASPs) that tolerate nonconformance to standards [3] in exchange for speed, cost and/or power advantages.

This paper deals with representation of real numbers. Since digital circuits are finite, an approximation is necessary, mapping a real value to some integer that represents it. To understand such mappings, we review briefly positional and non-positional number systems for integers.

1.1 Positional

Binary and decimal are examples of positional number systems, in which the position of each digit determines the associated weight. The left-most digits have larger weights, which allows easy comparison, sign detection and division by constants related to the weights. For simple radix systems (e.g. binary or decimal), the ratio between adjacent weights is a constant (e.g. 2 or 10). A Mixed Radix Number System (MRNS) is one in which this ratio is allowed to vary. MRNS relates the integer value x to the digits $(\bar{x_i})$ that form its representation using the following formula that shows how to compute the digits given the desired value:

$$\bar{x}_i = \left\lfloor x / \left(\prod_{j=0}^{i-1} M_j \right) \right\rfloor \mod M_i \qquad (1)$$

where $0 \leq i < n$, n is the number of digits and M_j is the ratio between the weights for the j and $j+1$ positions. The MRNS is capable of representing

$$M = \prod_{j=0}^{n-1} M_j \qquad (2)$$

unique values, $x_{\min} \leq x \leq x_{\max}$ where the $x_{\min}$ and the $x_{\max}$ are the integer minimum and maximum such that $M = x_{\max} - x_{\min} + 1$. Typically, $x_{\min} = 0$ and

$x_{\max} = M - 1$ forming unsigned integers, but signed integers are possible with $x_{\min} < 0$ at the cost of a smaller $x_{\max}$. For binary, $M_j = 2$ and thus $M = 2^n$, but in MRNS the M_j is different in each position and thus M is a composite number. Because (1) chooses the digits from a non-redundant set, i.e., $0 \leq \bar{x}_i < M_i$, the algorithms for addition, subtraction, comparison, etc. are analogous to those for binary, involving carries which make the speed of minimal-area circuits proportional to the value of n (e.g., ripple-carry adders).

An advantage of a positional number system is that it is easy to perform the inverse procedure of (1), and to convert the tuple of digits $\bar{x}_i, 0 \leq i < n$ into its integer value, x:

$$x = \sum_{i=0}^{n-1} \left(\bar{x}_i \cdot \prod_{j=0}^{i-1} M_j \right). \tag{3}$$

Related to the ability to perform (3) are several operations essential in Section 2. Division by the weight of the kth position $(M_0 \cdot M_1 \cdot ... \cdot M_{k-1})$ is known as *scaling*. Scaling occurs at insignificant cost in MRNS by considering the remaining positions (all but the rightmost k) as forming a different mixed radix. Unlike a constant radix system (e.g. binary) where this operation is visualized as shifting the remaining digits to the right, MRNS requires the casting of the remaining digits according to a new MRNS data type defined by the remaining M_js.

Another important operation in Section 2 is *base extension*, which takes a representation with a small number of digits, and produces the representation of the same value using a larger number of digits. Just as in binary, base extension is implemented in MRNS by simple concatenation of zeros on the left (interpreted according to new M_js that the designer chooses).

1.2 Non-positional

In contrast to positional number systems, the Residue Number System (RNS) [21] represents an integer $x_{\min} \leq x \leq x_{\max}$ with a set of moduli,

$$\hat{x}_i = x \bmod M_i \tag{4}$$

where the M_is must be chosen to be relatively prime to each other, i.e., for all $i \neq j$, $\gcd(M_i, M_j) = 1$. Each moduli is of equal importance—unlike MRNS digits, one cannot speak of lesser- or greater-significant moduli. Often M_is are chosen to be primes, which allows a number-theoretic logarithm function (also called index calculus) to be defined [21] that assists with multiplication and is the basis for the Logarithmic Residue Number System (LRNS) [24], a similar-sounding but distinct concept from the one (RLNS) proposed in this paper.

In conventional RNS, modulo-M addition, subtraction and multiplication of integers are carry-free. For example, to compute $z = (x - y) \bmod M$,

$$\hat{z}_j = (\hat{x}_j - \hat{y}_j) \bmod M_j, \tag{5}$$

where $0 \leq j < n$. These n channels of information can be processed in parallel. Even though the numeric interpretation (4) of the RNS $\hat{x}_j$s is entirely different than the numeric interpretation (1) of the MRNS $\bar{x}_j$s, the coding options and storage requirements are similar using the same M_js. Electronically, each MRNS digit $\bar{x_j}$ or RNS modulus $\hat{x}_j$ may be represented with one of several codes:

- Binary: $\eta_j = \lceil \log_2 M_j \rceil$ wires with 2-valued logic [16],
- Index: $\approx \lceil \log_2 M_j \rceil$ wires with 2-valued logic [24],
- One-hot: M_j wires with 2-valued logic [8],
- M_j-ary: 1 wire with multi-valued logic [23].

The binary-coded representation is the most compact for storage purposes, but the one-hot-coded representation allows faster logic and lower power consumption [7]. In addition to electronics, optical [6] and quantum [18] implementations of RNS have been suggested.

The total number of bits needed to store a value with one-hot RNS (n_1) and binary-coded RNS (n_2) are

$$n_1 = \sum_{j=0}^{n-1} M_j, \qquad n_2 = \sum_{j=0}^{n-1} \lceil \log_2 M_j \rceil = \sum_{j=0}^{n-1} \eta_j. \tag{6}$$

By choosing one modulus as a power of two and the others slightly less than powers of two, the range of a similar-sized binary system, 2^{n_2}, is only slightly larger than M. In applications that do not need quite the full power-of-two range, careful selection of moduli allows binary-coded RNS to be as storage efficient as binary.

RNS addition speed depends on technology and coding, but in theory has the advantage of no carries. For the short words considered here, ripple-carry addition is probably good for both conventional binary and binary-encoded moduli. Let the delay of a simple gate be δ, the delay of a multiplexor be $\delta_{\text{mux}} = 2\delta$ and the delay of a full adder be, say, $\delta_{\text{FA}} \approx 3\delta$. An n_2-bit binary adder takes about $\delta_{\text{addBin}} \approx n_2\delta_{\text{FA}} \approx 3n_2\delta$. The delay of equivalent binary-encoded RNS depends on $\eta = \max(\eta_0, ..., \eta_{n-1})$. Modulo addition needs two η-bit ripple-carry adders in which the carry of the second chases one δ_{FA} behind the first. A final multiplexor chooses between the two possible results ($\hat{x}_j + \hat{y}_j$ or $\hat{x}_j + \hat{y}_j - M_j$), making $\delta_{\text{addRNSbin}} \approx (\eta + 1)\delta_{\text{FA}} + \delta_{\text{mux}} \approx 3\eta\delta + 5\delta$. Since $\eta < n_2$, binary-encoded RNS should be faster. One-hot encoding needs $\delta_{\text{addRNSOH}} \approx (1 + \eta)\delta$ with fan-in-2 gates. For example, with moduli $M_0 = 4$,

$M_1 = 3$, $M_2 = 5$, $M_3 = 7$ we get $\eta = 3$ and $n_2 = 10$; this means $\delta_{\text{addBin}} \approx 30\delta$, $\delta_{\text{addRNSbin}} \approx 14\delta$ and $\delta_{\text{addRNSOH}} \approx 4\delta$.

The problematic operation in RNS is the interpretation of the integer value represented by the moduli. There are two common approaches: the Chinese Remainder Theorem (CRT), which computes a sum of n products modulo the large number M; and the Mixed Radix Conversion (MRC), which computes serially each MRNS digit using a large number ($O(n^2)$) of simpler operations on moduli/digits [13]. Scaling, base extension, sign detection and comparison are all expensive operations in RNS since all moduli need to participate [21].

Base extension in RNS involves creation of a new modulus channel, $\hat{x}_n$, defined by an additional relatively prime M_n. This new channel reconstructs what x mod M_n should be from the existing moduli channels, $\hat{x}_0, ..., \hat{x}_{n-1}$. Unlike MRNS, all of the existing channels have to be processed to accomplish this. A sequential algorithm to accomplish this is to apply the CRT, but instead of doing the sum of products modulo M, base extension only requires the much smaller modulo M_n. For moderate M, another alternative is a ROM.

To divide x by the constant M_j is always easy in MRNS, but in RNS it depends on the value of x. Such scaling is less difficult when $\hat{x}_j = 0$. In this case, the RNS interpretation of the remaining moduli, x mod (M/M_j) can be multiplied modulo M/M_j on a channel-by-channel basis by the multiplicative inverse of M_j to form the RNS representation of $\lfloor x/M_j \rfloor$ mod (M/M_j). Unfortunately, this result is not in the original RNS, but in the RNS defined by the reduced moduli set $\{M_0, M_1, ...M_{n-1}\} - \{M_j\}$. Scaling is more difficult if $\hat{x}_j \neq 0$. In this case, the base extended value of $\hat{x}_j$ must be subtracted from x. Thus, $\lfloor (x - \hat{x}_j)/M_j \rfloor$ mod $(M/M_j) = \lfloor x/M_j \rfloor$ mod (M/M_j) is produced in the reduced moduli set.

2. Representing Reals

There are three basic systems for using integer arithmetic hardware of the kind described earlier to process real numbers: fixed point, floating point and the logarithmic number system (LNS). In each of these systems, it is possible to use binary or residue for representing the integer(s) that in turn map to real numbers, making six systems in all under consideration here. In this paper upper-case variables describe real values manipulated by an application. Corresponding lower-case variables are the integers that represent these values following the rules for the particular system. In turn, such lower-case integers are represented by a collection of digits or moduli ($\bar{x}_i$ or $\hat{x}_i$). The following subsections describe the mapping of integers to reals in these systems without regard to whether the underlying integer representation is binary or residue.

2.1 Fixed Point

The mapping from integer x to real value X in the fixed-point system is:

$$X = x/M_F, \tag{7}$$

where the designer chooses the constant M_F to be the product of a subset of M_is. (For binary, all $M_i = 2$, thus the interpretation of the integer simply involves moving the radix point.) x is treated as a signed integer; thus, the real X is also signed (although recovery of the sign will be difficult for residue fixed point). Fixed-point addition is easy since it is equivalent to integer addition. Since scaling is needed after each fixed-point multiplication so that the product is in the same format as the inputs, fixed point multiply is more difficult in RNS than integer multiplication.

2.2 Floating Point

Ignoring some of its more exotic features, the IEEE-754 standard [13] maps the integers x_m (mantissa), x_e (exponent) and x_s (sign of the value) to the hidden-bit normalized real value X as

$$X = (1 - 2x_s) \cdot 2^{x_e} \cdot (1 + x_m/M), \tag{8}$$

where $M = 2^{23}$ for single precision. In floating point, x_e is an integer and does not need scaling. (8) can be generalized where M is a product of relatively prime moduli allowing easier representation with residue [12].

2.3 LNS

The formal description of LNS is similar to normalized floating point, but LNS is simpler because it uses a single scaled exponent x/M_F instead of an integer exponent and separate linear mantissa. The value represented by an LNS word is given by

$$X = (1 - 2x_s) \cdot b^{x/M_F}. \tag{9}$$

The choices of b and M_F determine the relative precision [17]. Typically in binary LNS, $b = 2$ and $M = 2^{-F}$, where F is the precision. However, because any real $b \neq 1$ is permissible, choosing $b \approx 1$ and $M_F = 0$ allows an integer x to represent a real X with precision related to how close b is to one. This is a

major advantage over fixed point or floating point as it decouples the application-specific choice of the numeric precision from the implementation-specific choice of the radix or moduli. Assuming $b \approx 1$, the lossy conversion from an arbitrary real X to an integer x is:

$$x = \begin{cases} \lfloor 0.5 + \log_b X_{\max} \rfloor, & X_{\max} < |X| \\ \lfloor 0.5 + \log_b |X| \rfloor, & X_{\min} < |X| < X_{\max} \\ \lfloor 0.5 + \log_b X_{\min} \rfloor, & |X| < X_{\min}. \end{cases} \quad (10)$$

The designer-supplied constants $X_{\min}$ and $X_{\max}$ satisfy the constraints $b^{x_{\min}} \leq X_{\min}$ and $X_{\max} \leq b^{x_{\max}}$. Because of the lack of overflow detection in RLNS, it is desirable to choose $[X_{\min}, X_{\max}]$ well inside the RLNS dynamic range. Although some LNS implementations provide an exact zero [15, 17], here zero is represented as $X_{\min}$. By proper selection of $X_{\min}$, multiplication yields a small result that is still large enough not to underflow. A better alternative is to avoid converting zeroes altogether [3].

As in floating point, the value sign (indicating $X < 0$) is given as an explicit bit x_s. x is also signed, which is hard to retrieve with RLNS. When $x < 0$, $|X| < 1.0$. When $x > 0$, $|X| > 1.0$. Some RLNS operators require extracting the sign of x; however, many do not require this costly step.

2.4 Comparison of the Prior Systems

Table 1 shows on a scale of 1 (easiest) to 4 (hardest) a subjective estimate of the relative difficulty for common arithmetic operations in the six real-number systems, not considering the novel improvement to RLNS addition proposed later in this paper. Comparison and overflow detection are difficult in all residue-based systems. Sign detection is easy in all LNS and floating point variants because the sign is encoded explicitly. Residue floating point operations are hard because they all demand sign detection [12]. Multiplication for residue integers (not shown in Table 1) is easy but multiplication is more challenging for residue fixed point because of the need to scale after each multiplication. Unlike residue floating point, RLNS does not need sign detection for multiplication, which means RLNS multiplication is faster than in any of the other systems. In a typical signal-processing application with as many or more multiplications than additions, RLNS will be faster than the equivalent residue floating point. RLNS division is faster compared to any of the other systems.

Table 1 shows the difficulty of implementing RLNS addition using prior naïve approaches (described in Section 3.2), which require large memory, whereas the table also shows how much easier addition is in binary LNS using the standard LNS approach (described in Section 3.3), which requires less memory. Although the only prior RLNS literature [2, 5] have proposed sophisticated interpolation techniques, these attempts have not overcome the basic memory limitations imposed by the naïve approach. The contribution of this paper is to offer a novel algorithm (described in Section 4.2) that decreases the memory and delay for RLNS addition so that it will be closer to the difficulty of binary LNS.

Table 1. Difficulty of Ops in Various Systems.

	Fixed		Float		LNS	
	Bin	Res	Bin	Res	Bin	Res
+	2	1	2	4	3	4
*	3	3	3	4	2	**1**
/	4	4	4	4	2	**1**
Sign	1	4	1	1	1	1
<	1	4	1	4	1	4
Overflow	1	4	1	4	1	4

3. LNS Arithmetic Algorithms

The central idea for efficient LNS design is to keep numbers in logarithmic format for as long as possible. They are converted on input and output but all intermediate computations stay in LNS. Arithmetic algorithms are given $\hat{x_i}$s or $\bar{x_i}$s and must produce the result in identical format. This section gives LNS algorithms in a generic enough way that they could be implemented either with binary LNS or RLNS.

3.1 Easy Operators

For operators like multiply and divide, RLNS allows for easy implementation. For example, the following produces $Q = X/Y$ in LNS format (q, q_s) given the LNS representations (x, x_s) and (y, y_s):

$$\begin{aligned} q &= (x - y) \bmod M \\ q_s &= x_s \oplus y_s. \end{aligned} \quad (11)$$

3.2 Naïve Addition Algorithm

This subsection gives a naïve explanation of LNS addition. Later sections will look at improved versions. To add two real values, X and Y, and produce $T = X + Y$, the naïve LNS addition involves three steps:
- divide, $Z = X/Y$
- lookup, $W = 1 + Z$

• multiply, $T = Y \cdot W$.
This gives $T = Y\left(1 + \frac{X}{Y}\right) = X + Y$. LNS hardware manipulates representations x, y, w, t instead of reals X, Y, W, T. The *first step* of the naïve approach uses integer subtraction:

$$z = (x - y) \bmod M. \tag{12}$$

Prior implementations do this with binary (M as a power of 2), but it could be done using residue (M as a product of primes). There is a small danger of overflow or underflow in this internal computation of z. In binary LNS, it is easy to detect such overflow, which either indicates $x \gg y$ or $y \gg x$, and to choose correspondingly x or y as the result of the addition. Omitting such overflow detection hardware significantly reduces power consumption of binary LNS [3]. In such binary LNS as well as all RLNS circuits, the designer must prevent overflow by proper selection of the minimum and maximum real converted values relative to M, as in (10). Designers of integer RNS circuits [16] are familiar with such *a priori* overflow prevention. Overflow prevention with RLNS is easier since the dynamic range possible in RLNS for a given M is typically much larger than for the same M with RNS.

The *second step* in the naïve approach involves a table lookup using all the bits of z:

$$w = \begin{cases} s_b(z), & x_s = y_s \\ d_b(z), & x_s \neq y_s \end{cases} \quad w_s = \begin{cases} 0, & x_s = y_s \\ z > 0, & x_s \neq y_s \end{cases} \tag{13}$$

The function s_b is used when the value sign bits x_s and y_s are the *same*; d_b is used when they *differ*:

$$s_b(z) = \log_b(1 + b^z), \quad d_b(z) = \begin{cases} \log_b |1 - b^z|, & z \neq 0 \\ \log_b(X_{\min}), & z = 0 \end{cases}$$

Analogous to (10), $d_b(0)$ is a special case so that future computations will avoid underflow. Because (13) involves addresses with all bits of z, it conceals sign detection ($s_b(-z) \neq s_b(z)$), which is also evident in w_s.

The *third step* produces the result using an adder and an XOR gate:

$$\begin{aligned} t &= (y + w) \bmod M \\ t_s &= y_s \oplus w_s. \end{aligned} \tag{14}$$

Again, residue works for t as long as M is large enough and $x_{\max}$ and $x_{\min}$ are close enough together that overflow cannot occur in the application. The significant cost in area and delay is the lookup table, which grows exponentially with the bits in z. The delay is $\delta_{\text{Naive}} = 2\delta_{\text{add}} + \delta_M$ but this comes at the cost of $2M$ words of memory, which probably makes the access time δ_M slow. Using the $M_0 = 4$, $M_1 = 3$, $M_2 = 5$, $M_3 = 7$ example from Section 1.2 and considering the encoding options of binary LNS, binary-encoded RLNS and one-hot RLNS, the delays are $60\delta + \delta_M$, $28\delta + \delta_M$ and $8\delta + \delta_M + \delta_C$, respectively, where $\delta_C \approx 2\delta$ (for $M_j \leq 7$) is the delay to compress the one-hot-coded moduli to binary-coded moduli so that the memory address bus will be the same width. Naïve Multiply Accumulate (MAC) delays are $90\delta + \delta_M$, $42\delta + \delta_M$ and $14\delta + \delta_M$, respectively, which illustrates why, if we could overcome the delay of the large Naïve memory, RLNS would become very competitive for MAC-intensive applications, like FFTs, DCTs, FIR filters, etc.

3.3 Standard Addition Algorithm

Prior LNS designers [22] have realized the inefficiency of the naïve approach. Many table-reduction techniques are possible for binary LNS [15]. All of these compression techniques have in common the assumption that sign detection and comparison are inexpensive (in binary the byproduct of (12) produces a carry which detects the sign of z). The following describes the standard algorithm that capitalizes on two algebraic properties involving the real value Z:

$$1 + Z = Z + 1 = \left(1 + \frac{1}{Z}\right) \cdot Z \tag{15}$$

$$\lim_{Z \to 0}(1 + Z) = 1. \tag{16}$$

When rephrased in terms of an LNS representation, z, these describe properties that s_b exhibits:
• Commutativity, $s_b(z) = s_b(-z) + z$;
• Essential zero, $\lim_{z \to -\infty} s_b(z) = 0$.
d_b has the same properties. Let M_{ez} be the point where the designer decides it is acceptable to approximate $s_b(z)$ and $d_b(z)$ as z for $z \geq M_{\text{ez}}$. Also it is acceptable to approximate $s_b(z)$ and $d_b(z)$ as 0 for $z \leq -M_{\text{ez}}$ Some designers [17] have chosen M_{ez} as the point at which the error becomes one-half of an ulp. It is possible to choose a smaller M_{ez} if the application will accept a few errors greater than this [3]. It is also possible to choose a larger M_{ez} than required so that it matches the table size, typically a power of two. For RLNS, M_{ez} will be a product of a subset of moduli and will not be constrained to analytical formulae as in [17], but rather be part of the RLNS design exploration enabled by the novel tools introduced later in this paper.

The following improved algorithm is, with few variations, the standard method used for addition in prior LNS applications [3, 22]. The *first step* computes $z = |x - y|$ by selecting either z_α or z_β:

$$\begin{aligned} z_\alpha &= (x - y) \bmod M \\ z_\beta &= (y - x) \bmod M \end{aligned} \tag{17}$$

$$z = \begin{cases} z_\alpha \bmod M_{ez}, & z_\alpha \geq 0 \\ z_\beta \bmod M_{ez}, & z_\alpha < 0. \end{cases}$$

Although this requires an extra subtraction, it has the advantage that the two subtractions occur in parallel. In binary LNS the only extra delay is for a multiplexor. In RLNS, the standard algorithm will result in excessive delay for sign detection. The above modulo M_{ez} operation is easy. In binary LNS, it means ignoring high order bits. In RLNS it means ignoring some of the moduli. The purpose of this step is to discard the high-order information from z prior to lookup. The *second step* in the standard algorithm uses the reduced width z:

$$w = \begin{cases} s_b(z), & x_s = y_s \\ d_b(z), & x_s \neq y_s. \end{cases} \tag{18}$$

The *third step* needs to consider four cases:

$$t = \begin{cases} y, & z_\alpha \leq -M_{ez} \\ (x+w) \bmod M, & -M_{ez} \leq z_\alpha < 0 \\ (y+w) \bmod M, & 0 \leq z_\alpha < M_{ez} \\ x, & M_{ez} \leq z_\alpha. \end{cases} \tag{19}$$

Some implementations [22] have checked these with two two-input multiplexors (one based on $z_\alpha > 0$ and the other based on $|z_\alpha| > M_{ez}$); however, for this paper it is convenient to consider this as a four-input multiplexor. The value sign comes from the sign of the larger input:

$$t_s = \begin{cases} y_s, & z_\alpha < 0 \\ x_s, & z_\alpha \geq 0. \end{cases} \tag{20}$$

The advantage is that the standard approach needs only $2 \cdot M_{ez}$ words of memory versus $2 \cdot M$ in the naïve approach. (s_b and d_b need the same size tables.) In RLNS M is the product (2) of all moduli, whereas M_{ez} is the product of a subset of moduli. $M/M_{ez} \geq 3$ will be the product of the one or more moduli discarded before addressing the table. The number of address bits saved, $\lceil \log_2(M) \rceil - \lceil \log_2(M_{ez}) \rceil$, is significant, typically two to five bits. The problem for RLNS is that the standard algorithm needs early sign detection and comparison to produce the address prior to the lookup, which is also a slow step. The delay for the standard algorithm is $\delta_{\text{std}} = 2\delta_{\text{add}} + \delta_{\text{comp}} + 2\delta_{\text{mux}} + \delta_{M_{ez}}$, where $\delta_{M_{ez}}$ is the delay of the smaller (probably faster) memory. In binary LNS, the standard algorithm increases speed much more than the slight delay introduced by the muxes. In RLNS, comparison is slower than the other components.

4. Novel Addition Algorithm

Section 4.2 proposes a new RLNS addition algorithm, which is as memory efficient as the standard one. The novel algorithm postpones sign detection so that speculative table lookup can proceed in parallel with sign detection (as happens in the naïve approach where sign detection is inherent in the lookup using all bits of z). Since lookup in the novel algorithm is not dependent on sign detection and comparison, the novel algorithm is faster than the standard algorithm for RLNS.

Before presenting this novel RLNS algorithm, it will be helpful to illustrate some concepts using a simpler speculative algorithm that could be implemented with either conventional binary LNS or with RLNS. Section 4.1 presents this easier-to-explain but less-memory-efficient algorithm in terms of integer operations, like $(x - y) \bmod M$. The main property that distinguishes this easier-to-explain algorithm from the standard one (Section 3.3) is that the new algorithm delays comparison until the final step. Section 4.2 uses somewhat obscure, special RLNS properties to optimize memory requirements of the algorithm.

4.1 Simple Speculative Algorithm

The *first step* of the speculative algorithm starts by computing both positive and negative versions of z:

$$\begin{aligned} z_\alpha &= (x - y) \bmod M \\ z_\beta &= (y - x) \bmod M \\ z &= z_\alpha \bmod M_{ez}. \end{aligned} \tag{21}$$

At this point we do not know which of these is positive; therefore, unlike the standard algorithm, it is not possible to compute the absolute value. Instead this algorithm will speculate and arbitrarily choose z_α as z and then, as the *second step*, look up the function value at both positive and corresponding negative arguments:

$$\begin{aligned} w_\alpha &= \begin{cases} s_b(z), & x_s = y_s \\ d_b(z), & x_s \neq y_s \end{cases} \\ w_\beta &= \begin{cases} s_b(-z), & x_s = y_s \\ d_b(-z), & x_s \neq y_s. \end{cases} \end{aligned} \tag{22}$$

The memory requirements for (22) will be $4 \cdot M_{ez}$, which is twice the standard algorithm but better than the $2 \cdot M$ needed by the naïve algorithm. Only at the time of the *third step* is the result of comparison required:

$$t = \begin{cases} y, & z_\alpha \leq -M_{ez} \\ (x + w_\alpha) \bmod M, & -M_{ez} \leq z_\alpha < 0 \\ (y + w_\beta) \bmod M, & 0 \leq z_\alpha < M_{ez} \\ x, & M_{ez} \leq z_\alpha. \end{cases} \tag{23}$$

Table 2. Residue representation ($z_0 = z \bmod M_0$, $z_1 = z \bmod M_1$) of $-10 \le z \le 9$ for moduli $M_0 = 2$ and $M_1 = 5$. The fact that there is no more than one odd $\hat{z}_1$ (in bold) per line illustrates (25), and means only even $\hat{z}_1$ need to be stored. These can be addressed by $\lfloor \hat{z}_1/2 \rfloor$.

$z \ge 0$				$z < 0$			
z	$\lfloor \hat{z}_1/2 \rfloor$	$\hat{z}_0$	$\hat{z}_1$	z	$\lfloor \hat{z}_1/2 \rfloor$	$\hat{z}_0$	$\hat{z}_1$
0	0	0	0				
1	0	1	**1**	-1	2	1	4
2	1	0	2	-2	1	0	**3**
3	1	1	**3**	-3	1	1	2
4	2	0	4	-4	0	0	**1**
5	0	1	0	-5	0	1	0
6	0	0	**1**	-6	2	0	4
7	1	1	2	-7	1	1	**3**
8	1	0	**3**	-8	1	0	2
9	2	1	4	-9	0	1	**1**
				-10	0	0	0

t_s is calculated as in (20). In RLNS to select the appropriate case, (23) requires comparison (say, using MRC to convert to the high-order MRNS digit $\bar{z}_{n-1}$). Using an asymmetrical range (workable for many applications, like MPEG [5]), simple MRC/base extension can simultanously provide sign detection:

$$t = \begin{cases} y, & \lceil M_{n-1}/2 \rceil \le \bar{z}_{n-1} \le M_{n-1} - 2 \\ (x + w_\alpha) \bmod M, & \bar{z}_{n-1} = M_{n-1} - 1 \\ (y + w_\beta) \bmod M, & \bar{z}_{n-1} = 0 \\ x, & 1 \le \bar{z}_{n-1} \le \lfloor M_{n-1}/2 \rfloor. \end{cases} \tag{24}$$

4.2 Novel Algorithm

If the speed and memory for RLNS addition can approach that of binary LNS, designers will have reason to exploit the faster multiply and divide offered by RLNS. The reason the standard algorithm (Section 3.3) uses half the memory compared to the one just presented in Section 4.1 is that the standard algorithm can remove the sign bit of the binary z before the memory is addressed without losing the information needed to construct the correct result. This sign allows reconstruction using $s_b(z) = s_b(-z) + z$. If we can identify some other bit in the RLNS representation that could be removed yet preserve this same property, equivalent memory savings will result. The removed bit cannot be the sign bit because in residue no single bit can reveal the sign.

Table 3. Half-sized table of $s_b(z)$ and $s_b(-z)$ addressed by $\lfloor \hat{z}_1/2 \rfloor$ and $\hat{z}_0$ for $M_0 = 2$, $M_1 = 5$ and $M_{ez} = 10$.

$\lfloor \hat{z}_1/2 \rfloor$	$\hat{z}_0$	$s_b(z)$	$s_b(-z)$
0	0	$s_b(0)$	$s_b(-10)$
0	1	$s_b(5)$	$s_b(-5)$
1	0	$s_b(2)$	$s_b(-8)$
1	1	$s_b(7)$	$s_b(-3)$
2	0	$s_b(4)$	$s_b(-6)$
2	1	$s_b(9)$	$s_b(-1)$

Table 2 shows the residue representation, $(z_0 z_1)$, of z with moduli subset $M_0 = 2$ and $M_1 = 5$ for $-10 \le z \le 9$. In this example, $M_{ez} = M_0 \cdot M_1 = 10$, which means there would be ten positive and ten negative arguments of z to tabulate for the speculative approach given earlier. These moduli are for illustration as these are probably too small for practical use. Looking only at the subset $(z_0 z_1)$ it is ambiguous whether the original z was positive or negative. For example, with $z_0 = 0, z_1 = 2$, the corresponding z could be either 2 or -8. Until MRC involving additional moduli, z_2..., has occurred, we are not sure. This is why the speculative approach computes both $w_\alpha = s_b(z)$ and $w_\beta = s_b(-z)$. It is also why the speculative approach takes twice the memory of the standard approach.

Table 2 shows positive z and the corresponding negative z on the same line to illustrate that

$$((-z) \bmod M_i) \bmod 2 = 0 \vee (z \bmod M_i) \bmod 2 = 0 \tag{25}$$

holds for odd M_i. With this, a similar table reduction as the standard approach is possible. In Table 2, odd values of $\hat{z}_1$ are shown in boldface. Notice there is never a case where $\hat{z}_1$ is odd in one column when the corresponding $\hat{z}_1$ in the other column is also odd. The table also shows $\lfloor \hat{z}_1/2 \rfloor$, the importance of which will be described shortly.

The least significant bit, $(\hat{z}_i) \bmod 2$, can be used to partition the range of z into two sets in the same way the sign bit partitioned the range of z for the standard algorithm. We only need to store the function for positive z in the standard algorithm; analogously, we only need to store the function for even $\hat{z}_i$ in the novel algorithm. This allows us to eliminate the least significant bit of $\hat{z}_1$ before using it to address the table. Such a near half-sized $s_b(z)$ table of $M_{ez} + 2$ words is shown in Table 3. $\lfloor \hat{z}_1/2 \rfloor$ addresses this table, from which either

$s_b(z)$ or $s_b(-z)$ may be reconstructed. Once MRC and sign detection have completed, we can choose which of the two outputs will yield the correct RLNS result. Table 3 has a mixture of positive and negative z with no easily discernable pattern: 0, -1, 2, -3, 4, 5, -5, -6, 7, -8, 9, -10; yet this partitioning is just as effective as the all-positive one used in the standard algorithm.

The *first step* of the novel algorithm is similar to the standard algorithm (17), except that rather than selecting the absolute value, the novel algorithm selects between z_α and z_β based on whether $\widehat{z_{\alpha_i}}$ is even:

$$\begin{aligned} z_\alpha &= (x-y) \bmod M \\ z_\beta &= (y-x) \bmod M \\ z &= \begin{cases} z_\alpha \bmod M_{\text{ez}}, & \widehat{z_{\alpha_i}} \bmod 2 = 0 \\ z_\beta \bmod M_{\text{ez}}, & \widehat{z_{\alpha_i}} \bmod 2 = 1. \end{cases} \end{aligned} \tag{26}$$

In most cases, this selection will be unique, but in a few cases either possibility works, as in the example for $z = -5$.

The *second step* uses a table like Table 3 to obtain w_α and w_β, for the same purpose as in the speculative algorithm. The addresses are restricted to the ones that are tabulated because of the selection of z in (26), but the tabulated z may not be what the RLNS computation needs. For this reason, a multiplexor on the output of the table interchanges $s_b(z)$ and $s_b(-z)$ when $\widehat{z_{\alpha_i}} \bmod 2 = 1$. The following describes the effect of looking up w_α and w_β from a half-sized table like Table 3 followed by the multiplexor:

$$w_\alpha = \begin{cases} s_b(z), & x_s = y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 0 \\ s_b(-z), & x_s = y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 1 \\ d_b(z), & x_s \neq y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 0 \\ d_b(-z), & x_s \neq y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 1. \end{cases} \tag{27}$$

$$w_\beta = \begin{cases} s_b(-z), & x_s = y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 0 \\ s_b(z), & x_s = y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 1 \\ d_b(-z), & x_s \neq y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 0 \\ d_b(z), & x_s \neq y_s \wedge \widehat{z_{\alpha_i}} \bmod 2 = 1. \end{cases}$$

In the *third step*, t and t_s are calculated as in (23) and (20), respectively. Figure 1 shows the proposed hardware for RLNS addition. The delay is $\delta_{\text{novel}} = \delta_{\text{add}} + \delta_{\text{mux}} + \max(\delta_{M_{\text{ez}}} + \delta_{\text{add}} + 2\delta_{\text{mux}}, \delta_{\text{comp}})$. Assuming δ_{comp} is larger, $\delta_{\text{novel}} \geq \delta_{\text{add}} + \delta_{\text{mux}} + \delta_{\text{comp}}$, which is an improvement of $\delta_{\text{add}} + \delta_{\text{mux}} + \delta_{M_{\text{ez}}}$ over the standard algorithm.

Using the Section 1.2 example (moduli 4, 3, 5 and 7), δ_{novel} becomes $16\delta + \delta_{\text{comp}}$ for binary-encoded RLNS, $6\delta + \delta_{\text{comp}}$ for one-hot RLNS. $\delta_{\text{comp}} = (n-1)(\delta_{\text{add}} + \delta_I)$, where δ_I is the time for the multiplicative inverse needed in the MRC. One-hot multiplicative inverses are free, which reduces the delay to $\delta_{\text{comp}} \approx (n-1)\delta_{\text{add}} \approx 12\delta$, thus the one-hot RLNS add may be as fast as 18δ, provided that $\delta_{M_{\text{ez}}} + \delta_C \leq 4\delta$. With the more realistic assumption $\delta_{M_{\text{ez}}} > 2$, the one-hot RLNS delay is $16\delta + \delta_{M_{\text{ez}}}$. The one-hot RLNS MAC delay is then $20\delta + \delta_{M_{\text{ez}}}$. A 32-bit binary-encoded RNS [11] (in a DCT application with integer precision and dynamic range equivalent to the above 20-bit one-hot RLNS) has $\eta = 8$, and $\delta_{\text{add}} = 29\delta$. The integer RNS MAC delay, assuming index calculus, will be at least $58 + 2\delta_{M_{\text{ez}}}$. One-hot RLNS is two to three times faster than index-calculus RNS in this application whilst having a word size that is one third smaller. The s_b/d_b table in the novel RLNS unit will take 128 words, whereas an index-calculus approach to 32-bit integer RNS multiplication takes three tables of 256 words each, a factor of six savings for RLNS.

5. RLNS Design Tool

The advantages of unusual number systems like LNS [20] and RNS [21] have been documented for decades, yet few designers actually use them. Hardware Description Languages (HDLs), like Verilog and VHDL, are now commonplace. HDLs support conventional binary arithmetic as a standard part of the language: `x+y` causes synthesis tools to generate binary adder hardware without designer effort. Designers are accustomed to the convenience of this level of abstraction—implementing an unconventional number system requires much more effort. Recently, software tools have begun to appear that generate easy-to-use HDL code for non-standard number systems such as RNS [19, 9] and LNS [10, 14]. It is even more difficult to design RLNS ALUs from scratch using HDLs because of the multiple abstraction levels (logarithm, residue and moduli) involved. To overcome this problem, a novel tool, `rlnstool.c`, [1] is introduced here that generates synthesizable RLNS Verilog based on designer-specified moduli, M_i, and the logarithm base, b.

To illustrate, consider the use of $b = \sqrt[2^4]{2}$, $n = 4$, $M_0 = 4$, $M_1 = 3$, $M_2 = 5$, $M_3 = 7$, which has similar precision and dynamic range compared to a 10-bit binary LNS that was overflow free and visually satisfactory in an MPEG application [3]. Before compiling `rlnstool.c`, the designer supplies `rlnstool.h` with this information:

```
#define BASE      1.044273782
#define MAXMODULI 4
#define MODULUS0  4
#define MODULUS1  3
#define MODULUS2  5
#define MODULUS3  7
```

`rlnstool.c` assumes that: $M_{\text{ez}} = \prod_{j=0}^{n-2} M_j$; the discarded modulus, M_{n-1}, is odd; and $x_{\min} = \lfloor M_{n-1}/2 \rfloor M_{\text{ez}}$; and $x_{\max} = \lceil M_{n-1}/2 \rceil M_{\text{ez}} - 1$. The best memory savings will result if M_{n-1} is the largest

moduli. The tool generates separate files for one-hot-encoded (`reg [19:0] a1, b1 ...`, i.e., $1 + n_1 = 1 + 4 + 3 + 5 + 7 = 20$ bits) and binary-encoded (`reg [10:0] a2, b2, ...`, i.e., $1 + n_2 = 1 + 2 + 2 + 3 + 3 = 11$ bits). Each file expects a macro that identifies the expected moduli set and encoding. Here, binary-encoded files expect `'MODULI_4_3_5_7`; one-hot-encoded files expect `'MODULI_ONEHOT_4_3_5_7`. To assist with simulation debugging, the tool generates nonsynthesizable Verilog functions that convert these formats into Verilog floating-point `real`s:

```
real A;
...
A = residueLNS1hot2real(a1); //cvt1rlns.v
A = residueLNS2real(a2);     //cvt2rlns.v
```

The tool also generates nonsynthesizable Verilog tasks that print the residue representation as well as the associated real during simulation:

```
print_1hot_residueLNS(a1); //pri1rlns.v
print_residueLNS(a2);      //pri2rlns.v
```

The actual hardware for one-hot RLNS arithmetic is described by synthesizable modules that the tool generates:

```
mul_1hot_residueLNS mu1(p1, a1, b1); //mul1rlns.v
div_1hot_residueLNS dv1(q1, a1, b1); //div1rlns.v
add_1hot_residueLNS ad1(s1, a1, b1); //add1rlns.v
                                     //also nad1rlns.v
```

Two versions of RLNS addition are generated by the tool: the new one (`add1rlns.v`) proposed in Section 4.2, and the naïve one (`nad1rlns.v`) explained in Section 3.2. Similar modules are generated for binary-encoded RLNS:

```
mul_residueLNS mu2(p2, a2, b2); //mul2rlns.v
div_residueLNS dv2(q2, a2, b2); //div2rlns.v
add_residueLNS ad2(s2, a2, b2); //add2rlns.v
                                //also nad2rlns.v
```

The best RLNS design methodology would be to perform *a priori* analysis and avoid all errors due to underflow and overflow, possibly using the Interval Logarithmic Number System (ILNS) [4] simulation prior to hardware design; however, a complete proof that a particular choice of b, M, M_{ez}, $x_{\min}$, $x_{\max}$, $X_{\min}$ and $X_{\max}$ achieves this could be quite involved. Instead, each of the above arithmetic modules has a novel feature to assist with debugging and design exploration of various moduli and other design parameters. The designer may optionally define a Verilog macro, `'CHECK_RELERR`, with a relative error criterion, for example 0.08. When simulating arithmetic with this definition, any computation whose RLNS result exceeds this bounds in relation to the same computation performed with `real`s will cause an error report. Such errors could be the result of a) inherent RLNS relative error if the debugging criterion is set too low or the base is too big; b) inherent underflow or overflow of the result due to a small M; c) approximation error of s_b or d_b due to small M_{ez} and/or asymmetry of $x_{\min}$ and $x_{\max}$; or d) overflow of z due to $x \ll y$ or $x \gg y$. For a given set of parameters, error a) and b) cannot be avoided–a larger wordsize will be required; c) and d) could be avoided by providing more expensive addition hardware (e.g. that compares x and y rather than z) with the same wordsize.

6. Conclusions

RLNS, a combination of LNS and RNS, offers the potential of faster real multiply and divide than either of its constituent parts. RLNS representations can be as compact as LNS and more compact than RNS. For example, Fernandez et al. [11] report implementation of the DCT using RNS fixed-point with moduli 256, 255, 253 and 251. Their 32-bit binary-encoded-moduli RNS representation is approximately three times the size of the equivalent RLNS representation with the moduli 4, 3, 5, and 7, which should be sufficient for such multimedia applications. The difficulty with naïve RLNS is how to compress the addition table as much as is possible with standard binary LNS without incurring extra delay due to residue comparison and sign detection. To improve speed and memory efficency, a new RLNS addition algorithm was proposed in which the comparison and sign detection occur in parallel with the table lookup. Special properties of RLNS allow this novel algorithm to produce the correct result after eliminating one bit from an odd modulus used as part of the table address. An RLNS design tool was explained that enables designers to explore parameters and generate synthesizable Verilog HDL for one-hot and binary encoded RLNS that uses this algorithm. The combination of fast multiply and divide, an improved addition algorithm and the convenient design tool (that hides much of the obscure mathematics explained here) make RLNS a practical option for high-throughput, low-precision applications, such as signal and multimedia processing.

References

[1] M. G. Arnold, "RLNStool version 1.0", `http://www.cse.lehigh.edu/~caar/rlnstool.html`

[2] M. G. Arnold, *Extending the Precision of the Sign Logarithm Number System*, M.S. Thesis, University of Wyoming, Laramie, 1982.

[3] Mark G. Arnold, "Reduced Power Consumption for MPEG Decoding with LNS," *Application-specific Syst., Arch., Proc. (ASAP)*, IEEE, San Jose, California, pp. 65-75, Jul. 2002.

[4] M. G. Arnold, J. Garcia, M. Schulte, "The Interval Logarithmic Number System," *16th IEEE Symp. Computer Arithmetic*, Santiago de Compostella, Spain, pp. 253-261, Jun. 2003.

[5] M. G. Arnold and J. Ruan, "Bipartite Implementation of the Residue Logarithmic Number System," accepted for *Adv. Signal Proc. Alg., Arch. and Implement. XV*, SPIE, vol. 5910, San Diego, CA, Aug. 2005.

[6] C. D. Capps, et al., "Optical Arithmetic/Logic Unit Based on Residue Arithmetic and Symbolic Substitution," *App. Optics*, vol. 27, pp. 1682-86, 1988.

[7] W. A. Chren, "One-Hot Residue Coding for Low Delay-Power Product CMOS Design," *IEEE Trans. Circuits and Syst.,* vol. 45, no. 3, pp. 303-313, Mar. 1998.

[8] R. Conway, et al., "New One-hot RNS Structures for High-speed Signal Processing," *Adv. Signal Proc. Alg., Arch. and Implement. XII*, SPIE, vol. 4791, Seattle, pp. 381-392, Jul. 2002.

[9] A. Del Re, A. Nannaelli, M. Re, "A Tool for Automatic Generation of RTL-level VHDL Description of RNS FIR Filters," *Proc. Design Auto. Test Europe (DATE)*, Vol. 1, Feb. 2004.

[10] J. Detrey and F. de Dinechim, "A VHDL Library of LNS Operations," *37th Asilomar Conf. Signals, Syst., Comput.*, Pacific Grove, CA, Sep. 2003.
`http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/`

[11] P. G. Fernandez, et al., "Fast RNS-based DCT Computation with Fewer Multiplication Stages," *Proc. XV Design Circ. Integr. Syst. Conf. (DCIS)*, Montpellier, pp. 276-281, Nov. 2000.

[12] E. Kinoshita and Ki-Ja Lee, "A Residue Arithmetic Extension for Reliable Scientific Computation," *IEEE Trans. Comput.*, vol 46, pp. 129-138, Feb. 1997.

[13] Israel Koren, *Computer Arithmetic Algorithms*, Brookside Court Publishers, Amherst, Massachusetts, 1998

[14] B. Lee and N. Burgess, "A Dual-Path Logarithmic Number System Addition/Subtraction Scheme for FPGA," *13th Intl. Conf. on Field Prog. Logic App.*, Lisbon, vol. 2778, pp. 808-817, Sep. 2003.

[15] D. M. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974-982, Aug. 1994.

[16] V. Paliouras and T. Stouratis, "Multifunction Architectures for RNS Processors," *IEEE Trans. Circ. Syst.*, vol. 46, pp. 1041-54, Aug. 1999.

[17] V. Paliouras and T. Stouraitis, "Low Power Properties of the Logarithmic Number System," *15th IEEE Symp. Computer Arithmetic*, Vail, pp. 229-236, Jun. 2001.

[18] K. Qing and M. J. Feldman, "Single Flux Quantum Circuits Using the Residue Number System," *IEEE Trans. Appl. Supercond.*, vol. 5, pp. 2988-91, Jun. 1995.

[19] D. Soudris, et al., "A Methodology for Implementing FIR Filters and CAD Tool Development for Designing RNS-based Systems," *International Symp. Circ. Syst.*, IEEE, pp. 129-132, May 2003.

[20] E. E. Swartzlander et al., "The Sign/Logarithm Number System," *IEEE Trans. Comput.*, vol. C-24, pp. 1238-1242, Dec. 1975.

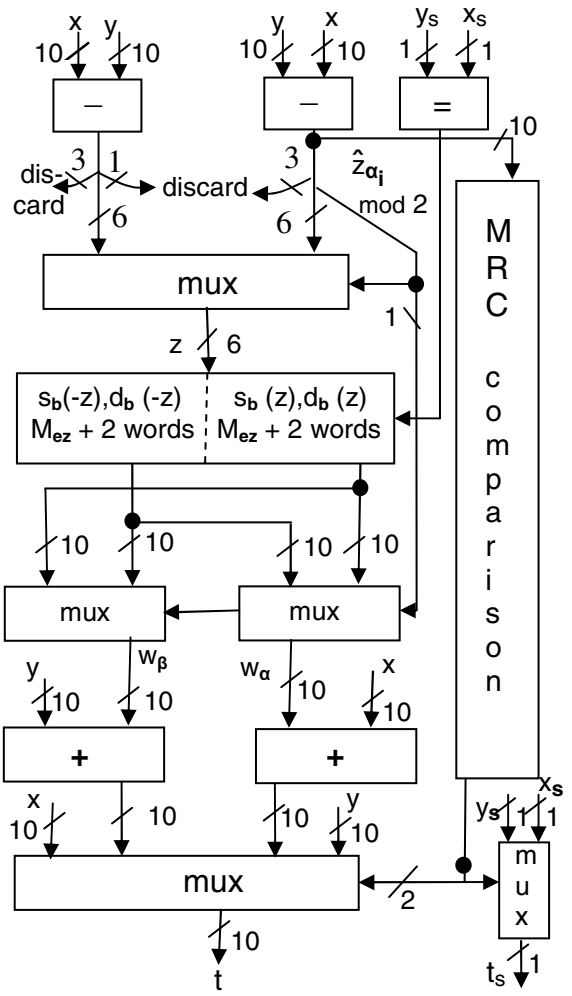

Figure 1. Novel RLNS Addition Unit, with bus widths assuming moduli 4, 3, 5 and 7.

[21] N. S. Szabo and R. I Tanaka, *Residue Arithmetic and its Application to Computer Systems,* McGraw Hill, 1967.

[22] F. J. Taylor, et al., "A 20 Bit Logarithmic Number System Processor," *IEEE Trans. Comput.*, vol. C-37, pp. 190-199, 1988.

[23] S. Wei and K. Shinizu, "Residue Arithmetic Circuits Based on Signed-Digit Multi-Valued Arithmetic Circuits," *27th Intl. Symp. Multi-Valued Logic*, Fukuoka, Japan, pp. 276-281, May 1998.

[24] G. Zelniker and F. J. Taylor, "A Reduced Complexity Finite Field ALU," *IEEE Trans. Circ. Syst.*, vol. 38, pp. 1571-73, Dec. 1991.