Application of symmetric redundant residues for fast and reliable arithmetic

Behrooz Parhami*
Dept. Electrical & Computer Eng., Univ. of California, Santa Barbara

(Invited Paper)

ABSTRACT

Despite difficulties in general division, magnitude comparison, and sign detection, residue number system arithmetic has been used for many special-purpose systems in light of its parallelism and modularity for the most common arithmetic operations of addition/subtraction and multiplication. Computation in RNS requires modular reduction, both for the initial conversion from binary to RNS and after each operation to bring the result back to within a valid residue range. Use of redundant residues simplifies this critical operation, leading to even faster arithmetic. One type of redundant mod-*m* residue, that keeps the representational redundancy to the minimum of 1 bit per residue, has the nearly symmetric range $[-m, m)$ and allows two values for each pseudoresidue: $\langle x \rangle_m$ or $\langle x \rangle_m - m$. We study the extent of simplification and speed-up in the modular reduction process afforded by such redundant residues and discuss its potential implications to the design of RNS arithmetic circuits. In particular, we show that besides cost and performance benefits, introduction of error checking and fault tolerance in arithmetic computations is facilitated when such redundant residues are used.

Keywords: computer arithmetic, error code, modular reduction, parallel processing, redundant representation, RNS

1. INTRODUCTION

Parallelism and modularity properties of residue arithmetic make it attractive for use in the design of high-performance and reliable systems. These features have been utilized in a wide variety of contexts, from error control in conventional binary processors (by means of residue encoding and checking) to custom realization of signal processing and other application-specific circuits with residue number system (RNS) arithmetic.[13, 21, 22]

One factor that affects the performance of all such applications is the frequent need for invoking modular reduction; i.e., computation of the modulo-*m* residue of a given number *x*. Modular reduction is needed for residue encoding, to perform residue checking, after each RNS arithmetic operation, and in virtually any other context where residues are used[9]. Redundant residues (pseudoresidues) have been found to offer benefits in simplifying modular reduction. Many forms of redundancy have been applied in an ad hoc manner to gain speed and cost benefits in various applications. A unified theory of redundant-residue systems is also starting to emerge[14].

In this paper, we consider a particular class of redundant residues called “double-range symmetric/signed” (DRS) pseudoresidues. DRS pseudoresidues, which are allowed to be in the range $[-m, m)$ instead of the standard nonredundant interval $[0, m)$, possess sufficient redundancy to allow significant simplification, and thus speed-up, in the modular reduction process; yet they imply only (a minimal) one bit of overhead for each redundant residue.

After presenting algorithms for modular reduction of arbitrary-width operands into DRS pseudoresidues, and arithmetic operations on such pseudoresidues, we study the effects of redundancy and show substantial benefits, in terms of circuit complexity and latency, in: (1) Residue encoding/checking, (2) Inner-product computation, (3) Modular multioperand addition, (4) RNS base extension and mixed-radix conversion, and (5) RNS to/from binary conversion. We conclude by analyzing the impact of the aforementioned cost/speed benefits on representative applications: filtering, arithmetic coding, and error detection/correction in redundant-modulus RNS.

* parhami@ece.ucsb.edu; phone 1-805-893-3211; fax 1-805-893-3262; http://www.ece.ucsb.edu/Faculty/Parhami/index.htm; Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA 93106-9560.

2. RNS WITH REDUNDANT RESIDUES

An RNS is characterized by a set of k pairwise relatively prime moduli $m_{k-1} > . . . > m_1 > m_0$ and a residue set R_i for the ith modulus, typically chosen to be $[0, m_i)$. Clearly, at most one of the moduli can be even and that one is usually taken to be a power of 2 to simplify the associated circuitry; all other moduli are odd. Roughly speaking, the set of moduli corresponds to the choice of a radix and the residue set to the digit set of positional fixed-radix number representations. Just as the digit set of nonredundant radix-r system can be chosen to be $[a, r + a)$, $a < 0$, the residue set R_i associated with m_i can be selected to be $[a_i, m_i + a_i)$. Such nonredundant RNS have been extensively studied and used, primarily, in signal processing applications. An RNS may have a handful of large moduli, typically chosen to be integers of the form 2^h or $2^h \pm 1$, or a larger number of small moduli, often chosen to be primes or powers of primes. The former category of RNS corresponds to very-high-radix number representation systems, in which digit manipulation circuits are quite complex but there are fewer such circuits, while the latter are akin to moderately high radices such as 16 or 64. Issues in choosing the moduli, and the associated cost/speed tradeoffs, are beyond the scope of this paper.

The primary advantage of RNS is that addition, subtraction, and multiplication can be performed independently and in parallel on the various residues. When the residues are small, this results in very high speed, particularly for multiplication which is slow/expensive with conventional number representation. Figure 1 depicts three implementations of a residue adder for a modulus m. The simpler ones in Figs. 1a and 1b, are combinational and sequential versions of the same scheme[2, 5]. They both add two h-bit residues, x and y, to form their sum $u + 2^h c$. If either c or the carry-out d of the addition $u + 2^h - m$ is 1 (the latter implying, for $c = 0$, that $x + y \geq m$), then the output is $v = x + y - m$ instead of $x + y$. The faster implementation in Fig. 1c forms v directly through a carry-save addition and an ordinary addition[17].

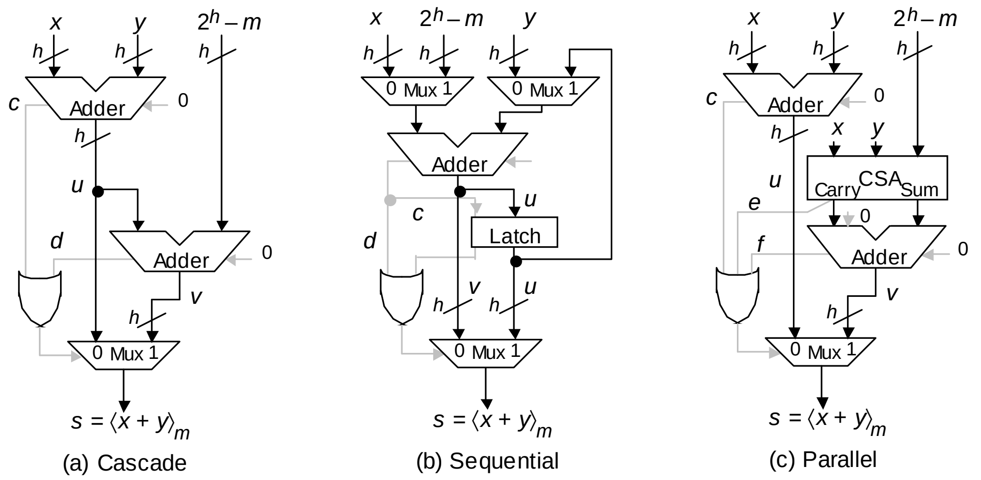

Fig. 1. Design of mod-m residue adder.

A residue multiplier is somewhat more complex, but still considerably simpler and faster than an ordinary binary multiplier. A bit-at-a-time implementation would use a modular adder to derive the h-bit mod-m product in h cycles. It is also possible to use an $h \times h$ combinational (tree or array) multiplier, followed by a mod-m reduction circuit for the $2h$-bit product. Alternatively, modular reduction can be fused with combining of partial products for speed/economy, particularly if one takes advantage of the fact that some bit patterns are unused. Here, we say nothing about implementing difficult RNS operations such as sign test, overflow detection, magnitude comparison, or general division. The methods to be discussed do not cure such difficulties but are meant as mechanisms for making the already efficient RNS operations even simpler and more efficient. To the extent that the difficult operations use the simpler ones as building blocks, the added efficiency benefits those as well.

Consider a modulus m satisfying $2^{h-1} < m \leq 2^h$. Conventional binary residues associated with this modulus are h-bit unsigned numbers in the range $[0, m)$; we refer to these as *single-range unsigned* (SRU) residues. Similarly, *single-range signed* (SRS) residues have a symmetric or almost symmetric range $[-\lfloor m/2 \rfloor, \lceil m/2 \rceil)$. In both of these cases, the residues are nonredundant in the sense that each mod-m equivalence class has a single representative in the residue set.

In a manner similar to the use of digit sets with more than r values for radix-r positional representations[1, 13], we can envisage the use of more than m values in a residue set. Using multiple representatives from some or all mod-m equivalence classes does not cause any problem in arithmetic because all operations are modulo m. We refer to residues from redundant sets as *pseudoresidues* and to the resulting RNS as *redundant-residue RNS*. This is to distinguish our systems from *redundant-modulus RNS* in which extra moduli, beyond the minimum set required for providing a desired dynamic range, are employed for the sake of error detection and/or correction.

Table I shows the nonredundant SRU and SRS residue sets along with several examples of redundant residue sets. The *redundancy index* associated with a modulus m having pseudoresidues in $[a, b]$, or $[a, b + 1)$, is defined as:

$$\rho = b + 1 - a - m \qquad (1)$$

The two single-width redundant residue sets use the entire range of values offered by an h-bit representation as valid pseudoresidues, in unsigned (SWU) or 2's-complement signed (SWS) format. Double-range pseudoresidues can assume twice as many values as a nonredundant residue, again in unsigned (DRU) or signed (DRS) form. A useful encoding of DRU is the carry-save format in which the pseudoresidue is the sum of two h-bit unsigned numbers that are separately represented. Triple-range unsigned (TRU) pseudoresidues with values in $[0, 3m - 3]$ have been called "relaxed residues" and used to simplify modular reduction[19]. Squared- or quadratic-range unsigned (QRU) residues are capable of holding the product of two SRU residues and have thus been used in inner-product computations of the type found in digital filters[11]. Double-width unsigned residues can be used in lieu of QRU for implementation simplicity. End-around residue sets allow an overflow of weight 2^n to be reinserted at the least-significant position[16], given the property $2^n = 1 \bmod m$. For example, mod-9 residues can be accumulated by using 6-bit pseudoresidues with end-around carry ($2^6 = 1 \bmod 9$).

Table I. Some choices for mod-m residue set.

Code	Type of residue	Residue set	ρ
SRU	Single-range unsigned	$[0, m)$	0
SRS	Single-range signed	$[-\lfloor m/2 \rfloor, \lceil m/2 \rceil)$	0
SWU	Single-width unsigned	$[0, 2^h)$	$2^h - m$
SWS	Single-width signed	$[-2^{h-1}, 2^{h-1})$	$2^h - m$
DRU	Double-range unsigned	$[0, 2m)$	m
DRU_{CS}	DRU, carry-save variant	$[0, 2m - 2]$	$m - 1$
DRS	Double-range signed	$(-m, m)$ or $[-m, m)$ or $[-m, m]$	$m - 1$ or m or $m + 1$
TRU	Triple-range unsigned	$[0, 3m)$	$2m$
QRU	Squared-range unsigned	$[0, (m - 1)^2]$	$m^2 - 3m + 2$
DWU	Doublewidth unsigned	$[0, 2^{2h})$	$2^{2h} - m$
EAU	End-around unsigned	$[0, 2^n)$	$2^n - m$

As evident from Table I and discussion of the examples it contains, redundant residue sets have been applied in an ad hoc fashion as tools for speeding up or simplifying circuit realizations[3, 8, 11, 12, 16, 19]. Only very recently have these been explicitly recognized as redundant residues and, thus, received a unified treatment[14]. A general theory of redundant-residue RNS representations allows us to obtain such implementation strategies as special cases, thus facilitating design tradeoffs and fostering the portability of techniques and results among different application contexts.

Note that among the redundant-residue examples listed in Table I, SWU and SWS imply no representational redundancy in the sense that the width h of their pseudoresidues can be the same as that of conventional residues. These do lead to some simplification in residue arithmetic, but significant speed benefit requires at least a (near) doubling of the range. Intuitively, this is because SWU and SWS allow two distinct representations only for a subset of the m residues; other residues must always be derived precisely, as there is no room for error in the single-representation cases. The closer m is to 2^h, the lower the redundancy and, hence, the more rigid the implementation requirements. For this reason, double-range redundant pseudoresidues can be viewed as having the minimal redundancy for our purposes because they entail pseudoresidues that are only 1 bit wider than ordinary residues. Among these, we prefer DRS because its narrower range of product (approximately $2m^2$ versus $4m^2$ possible values) can lead to simpler circuit realizations.

3. MODULAR REDUCTION TO PSEUDORESIDUES

In this section, we consider the problem of reducing a k-bit unsigned or 2's-complement number into an $(h + 1)$-bit DRS pseudoresidue. Unless otherwise noted, we consider the nearly symmetric variety of DRS with pseudoresidues in the range $[–m, m)$. Much of what we derive is directly applicable, or easily adapted, to the other two varieties of DRS shown in Table I. The k-bit input argument can itself be viewed as a pseudoresidue with the range $[0, 2^k)$ or $[–2^{k–1}, 2^{k–1})$. Hence, we formulate our problem as conversion between mod-m pseudoresidues of varying ranges. In what follows, we discuss some of the more useful conversions. Similar methods can be used for virtually any such conversion.

First, note that a DRS pseudoresidue X can be converted to a conventional residue x. Symbolically, this type of residue set conversion is denoted by:

$$(X_h \ldots X_1X_0)_{\text{2's-compl}} \in [–m, m) \longrightarrow_{\text{mod-}m} (x_{h–1} \ldots x_1x_0)_{\text{two}} \in [0, m) \tag{2}$$

This conversion is done by simply adding m to the pseudoresidue X if it is negative. Thus, the circuit implementing this conversion consists of an h-bit adder, with one input connected to $X_{h–1:0}$ and the other input to $X_h \times m$, where the latter number is formed by fanning out X_h to all bit positions where m has a 1 in its binary representation. Thus, the cost and latency of this circuit are essentially those of an h-bit adder when it is built for a specific m. If the circuit must work for different values of m, where m is an input, then $X_h \times m$ is the output of a mux which has 0 and m as inputs.

Next, we consider modular reduction for an arbitrary $(h + 1)$-bit 2's-complement number to a DRS pseudoresidue of the same width. Symbolically:

$$(X_h \ldots X_1X_0)_{\text{2's-compl}} \in [–2^h, 2^h) \longrightarrow_{\text{mod-}m} (Y_h \ldots Y_1Y_0)_{\text{2's-compl}} \in [–m, m) \tag{3}$$

The conversion consists of adding m to X if it is negative and subtracting m from it (adding $2^{h+1} – m$) if it is nonnegative. Circuit realization consists of an $(h + 1)$-bit adder with inputs being X and $X_h' \oplus m$ (selective complement of m) and its c_{in} connected to X_h'. An h-bit unsigned input can be reduced to a DRS pseudoresidue by unconditionally subtracting m from it. This corresponds to the symbolic representation:

$$(X_{h–1} \ldots X_1X_0)_{\text{two}} \in [0, 2^h) \longrightarrow_{\text{mod-}m} (Y_h \ldots Y_1Y_0)_{\text{2's-compl}} \in [–m, m) \tag{4}$$

Now consider converting from a pseudoresidue X of width $2h + 1$ to DRS. For most RNS applications where h is relatively small, the following table lookup scheme is attractive. Divide X into two parts: the upper $h + 2$ bits $X_{2h:h–1}$ and the lower $h – 1$ bits $X_{h–2:0}$. Use $X_{2h:h–1}$ as the address into a $2^{h+2} \times h$ lookup table that holds $x_{\text{hi}} = m – (2^{h–1}X_{2h:h–1} \bmod m)$. Compute the desired DRS pseudoresidue as $X_{h–2:0} – x_{\text{hi}}$.

An arithmetic-based method involving short multiplications by constants can be used in lieu of table lookup. We are investigating the possibility of a more direct approach, but for now we advocate the use of a method due to Posch and Posch[19, 20] which performs reduction into an $(h + 2)$-bit TRU pseudoresidue (Table I) using a pair of short multiplications by constants. Let this TRU pseudoresidue be $(Y_{h+1} \ldots Y_1Y_0)_{\text{two}}$. We then add 0, $–m$, or $–2m$ to this value depending on the 3 MSBs $Y_{h+1}Y_hY_{h–1}$ (000, 001, and otherwise, respectively); this needs a simple control logic circuit, a multiplexer, and an adder. Note that the method of Posch and Posch works for $2h + 4$ bits to handle the product of two $(h + 2)$-bit TRU pseudoresidues. This property, symbolized as follows, may prove useful in some contexts:

$$(X_{2h+3} \ldots X_1X_0)_{\text{two}} \in [0, 2^{2h+4}) \longrightarrow_{\text{mod-}m} (Y_{h+1} \ldots Y_1Y_0)_{\text{two}} \in [0, 3m) \longrightarrow_{\text{mod-}m} (Z_{h–1} \ldots Z_1Z_0)_{\text{2's-compl}} \in [–m, m) \tag{5}$$

It is also helpful to contemplate reducing from the set of 2^h values $2^g \times [0, 2^h)$ to a DRS pseudoresidue. Let b represent the h-bit constant $2^g \bmod m$. Then, $2^gx \bmod m = bx \bmod m$. So, we need a multiplication by the precomputed constant b and modular reduction of the $2h$-bit unsigned product.

Finally, an arbitrarily wide k-bit number can be reduced to a DRS pseudoresidue by dividing it into various segments, reducing each segment to a mod-m residue or inverse residue (if the rightmost segment is $h – 1$ bits, it needs no conversion), pairing residues with inverse residues, performing subtraction to get half as many DRS pseudoresidues, and finally adding all the pseudoresidues to get the final result. Addition of pseudoresidues will be discussed in the next section alongside other arithmetic operations.

4. ARITHMETIC OPERATIONS WITH PSEUDORESIDUES

In this section, we discuss how basic arithmetic operations and a few useful composite operations are performed with pseudoresidues. Note that two types of arithmetic with pseudoresidues can be envisaged: hybrid (mixed with ordinary residues) or pure. For example, 2-operand addition involving pseudoresidues might take the following forms:

$$\text{Pseudoresidue} + \text{Residue} \to \text{Pseudoresidue} \qquad (6)$$

$$\text{Pseudoresidue} + \text{Pseudoresidue} \to \text{Pseudoresidue} \qquad (7)$$

The first kind is akin to carry-save addition in that it might be used to accumulate the sum of a set of numbers (represented with ordinary residues) while taking advantage of speed/ease of arithmetic with pseudoresidues; the second kind is of the same flavor as adding redundant operands in (generalized) signed-digit arithmetic. In this paper, we are interested in operating on numbers that are represented with pseudoresidues; thus, it is the second type of arithmetic operation above that will be discussed.

Addition of DRS pseudoresidues can of course be performed via standard binary addition followed by modular reduction using the methods discussed in Section 3. However, we can avoid the two-stage process if we note that $X + Y$ can fall outside the range of pseudoresidues only if the operands have like signs. If both operands are nonnegative, subtracting m is guaranteed to bring the sum to within $[-m, m)$, while if both are negative, adding m will do the trick. Hence, we can perform addition directly:

$$(X_h \dots X_1X_0)_{\text{2's-compl}} + (Y_h \dots Y_1Y_0)_{\text{2's-compl}} \in [-2m, 2m-2] \longrightarrow_{\text{mod-}m} (S_h \dots S_1S_0)_{\text{2's-compl}} \in [-m, m) \qquad (8)$$

Figure 2 depicts an efficient hardware implementation for DRS adder. Based on the signs of the two input operands, one of the three values 0, m, or $-m$ is added to $X + Y$ by means of a carry-save adder followed by an $(h + 1)$-bit binary adder. The carry out of position $h + 1$ is ignored for both the CSA and the binary adder. Comparing the adder in Fig. 2 to that in Fig. 1c reveals both lower cost and greater speed. The speed advantage is even more pronounced if the adder is built for a constant m, because in this case, the gate network producing the rightmost input to the CSA is simplified.

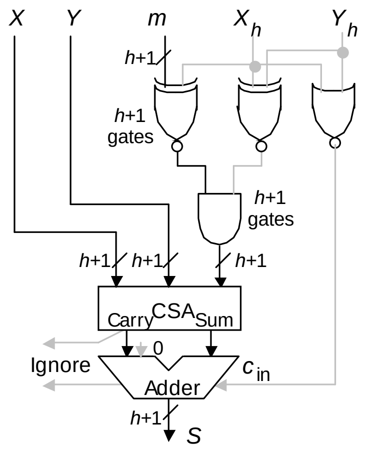

Fig. 2. DRS modular pseudoresidue adder.

Note that in Fig. 2, the XNOR and AND gates can be replaced by 2-to-1 multiplexers realizing the $h + 1$ functions $m_i(X_hY_h) \vee m_i'(X_h \vee Y_h)'$. Thus, for a fixed m, the two bits X_hY_h and $(X_h \vee Y_h)'$ are simply fanned out to appropriate positions in the rightmost CSA input in Fig. 2. As a rule, we present our designs assuming that m is an input parameter. This will allow our units to be time-shared for performing operations associated with several different moduli if desired. Simplifications resulting from a specific choice of m are straightforward. The only requirement for the aforementioned shared use is that all residues sharing the unit have $(h + 1)$-bit redundant representations; that is, all m values must be in the range $(2^{h-1}, 2^h]$. However, practical RNS applications already use moduli that are comparable in magnitude, leading to residues of exactly or approximately the same width.

Negation (change of sign) for symmetric DRS pseudoresidues in $(–m, m)$ or $[–m, m]$ is done by 2's complementation. For pseudoresidues in $[–m, m)$, negation may lead to an invalid pseudoresidue m which must be converted to 0. This can be done by detecting $–m$ at the input and forcing the output to zero. Subtraction can be performed by negating Y and adding. To negate Y, its bitwise complement is formed and a 1 is inserted in lieu of the 0 LSB in the left input to the adder in Fig. 2. Of course, the gate network must also be modified to allow either addition or subtraction to be performed. The only remaining issue is to show that in case of $Y = –m$, no problem arises. Note that in this case, the sum is in the slightly wider range $[–2m, 2m)$. However, because we reduce the sum by m whenever both summands are nonnegative, the output will be in $[–m, m)$.

Multiplication of two DRS pseudoresidues produces a $(2h + 1)$-bit 2's-complement number. This number can be reduced to a pseudoresidue as discussed in Section 3. The hardware realization of the preceding scheme consists of an ordinary binary multiplier and a reduction circuit. As usual, it is possible to interlace modular reduction with the product accumulation. Consider the following symbolic representation of our reduction:

$$(X_h \ldots X_1X_0)_{\text{2's-compl}} \times (Y_h \ldots Y_1Y_0)_{\text{2's-compl}} \in [–m(m – 1), m^2] \longrightarrow_{\text{mod-}m} (S_h \ldots S_1S_0)_{\text{2's-compl}} \in [–m, m) \qquad (9)$$

Multiplication via left-shift-add iterations is the best one to use for modular multiplication (P is initialized to 0):

$$2P + Y_{h–j}X \in [–3m, 3(m – 1)] \longrightarrow_{\text{mod-}m} \text{New } P \in [–m, m) \qquad (10)$$

The left-hand side of (10) can be reduced to a pseudoresidue via one doubling (simpler form of addition that does not require the CSA in Fig. 2) and a modular pseudoresidue addition. Note that doubling is done by computing $2P \pm m$, where addition or subtraction is chosen based on the sign of P. The resulting circuit will have a critical path that goes through two adders, one CSA, and 5 gate levels (XOR is counted as 2 levels). The first adder can be replaced by a CSA forming $2P \pm m + Y_{h–j}X$ as a carry-save number. The result will have a value in $[–2m, 2(m – 1)]$. Using a lookahead circuit to determine the sign of this value allows us to add $\pm m$ to it to bring the result to within $[–m, m)$ by reducing a positive value and increasing a negative value. In this way, the first adder of the previous design is replaced by a CSA and a single lookahead circuit, leading to a simpler and faster circuit.

Multiply-add operation can be handled by using the bits of the additive operand, from MSB to LSB, to replace the 0 LSB of $2P$ in the multiplication iterations. Squaring can be done via multiplication or directly to take advantage of circuit simplifications. In particular, when h is moderate but not very small, direct lookup table based realization of squaring might be feasible where multiplication is not. In this latter case, pseudoresidues offer no advantage for the squaring itself because the tables can hold conventional residues.

Multioperand addition with pseudoresidues leads to no new problem if performed sequentially using the 2-operand adder of Fig. 2 or by a tree of such adders. Similarly, if accumulation is done before modular reduction, conventional CSA trees can be used for the accumulation phase with appropriate provisions for correct handling of signed values[13]. Piestrak's method of multioperand modular addition with end-around carries[16] is also easily adapted to pseudoresidues. It is noteworthy that RNS arithmetic (with or without pseudoresidues) offers little, if any, advantage over binary arithmetic for multioperand addition involving a large number n of operands. Binary arithmetic requires $O(\log n)$ latency for carry-save addition, besides the final carry-propagate addition. Multioperand addition in RNS implies $O(\log n)$ levels of modular addition, using an adder similar to that in Fig. 2, or $O(\log n)$ levels of ordinary carry-save addition followed by modular reduction. In the first case, the slower modular adders lead to binary arithmetic winning in most cases. In the second case, the modular reduction unit is likely to be slower than the final fast adder of binary arithmetic.

For inner-product computation, a scheme based on very wide pseudoresidues can be used[11]. The required hardware for sequential realization, or one cell in a pipelined linear-array, is shown in Fig. 3. A running total of width $2h + 1$ is maintained to which the $(2h + 1)$-bit product pseudoresidue is added to get a $(2h + 2)$-bit result which is latched. In case the two MSBs of the $(2h + 2)$-bit result are different, the result has overflowed the target $(2h + 1)$-bit range and must be reduced/increased by a multiple of m to bring it to within the target range. The multiple of m used is 2^hm because it allows us to do the adjustment by an $(h + 1)$-bit adder. Of course if increasing the number of data lines is acceptable from area and pin-out viewpoints, the running total can be kept in carry-save form and the design in Fig. 3 accordingly modified to use carry-save adders. The final conversion of the $(2h + 1)$-bit result to an $(h + 1)$-bit pseudoresidue is done following the process outlined in Section 3.

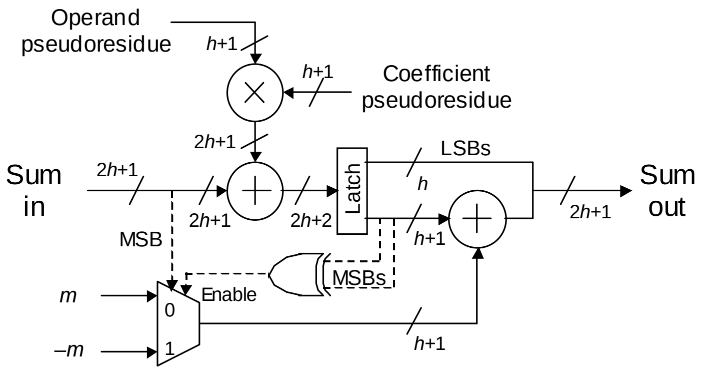

Fig. 3. Modulo-*m* multiply-accumulate cell.

Example 1 (FIR filter implementation): A finite impulse response filter can be implemented in RNS via an initial conversion of the input to RNS representation, performing filter calculations independently for each residue channel, and a final RNS to binary conversion. The main computational part of the filter for each residue is a linear array of *n* cells of the type shown in Fig. 3, one for each tap. The preceding adaptation of an ordinary RNS-based FIR filter to one based on pseudoresidues will work and is somewhat more efficient that the nonredundant version. However, pseudoresidues allow us to take a different approach which is considerably more efficient. Let us use our standard 2's-complement encoding for input psudoresidues but encode all intermediate pseudoresidues as binary signed-digit (BSD) numbers using the radix-2 digit set {–1, 0, 1}. Conceptually, this encoding is very similar to carry-save representation. The pseudoresidue X is encoded as two residues X^+ in [0, m) and X^- in [0, –m], with $X = X^+ – X^-$. The (2h + 1)-bit output of the multiplier is broken into two parts: the lower h – 1 bits constitute Y^+ and the upper h + 2 bits are converted via a lookup table or special circuit to a negative residue in [0, –m]. The output of this block, together with the lower h – 1 bits form a BSD pseudoresidue. This is added to the pseudoresidue representing the running total to form an (h + 1) digit BSD result having a wider range than our pseudoresidues. This wider result is brought to within [–m, m) in the early part of the next stage by adding to it a constant whose value is determined only by the MSBs of the positive and negative components. The speed and cost advantage of the design in Fig. 4 over that in Fig. 3 is quite obvious. The two BSD adders are only slightly slower than carry-save adders and the latency of one of them is completely overlapped with the multiplication. In addition to each cell being faster, the final conversion is also sped up because it consists of a correction (identical to that in the left half of Fig. 4) followed by recoding the BSD pseudoresidue to 2's-complement format. ■

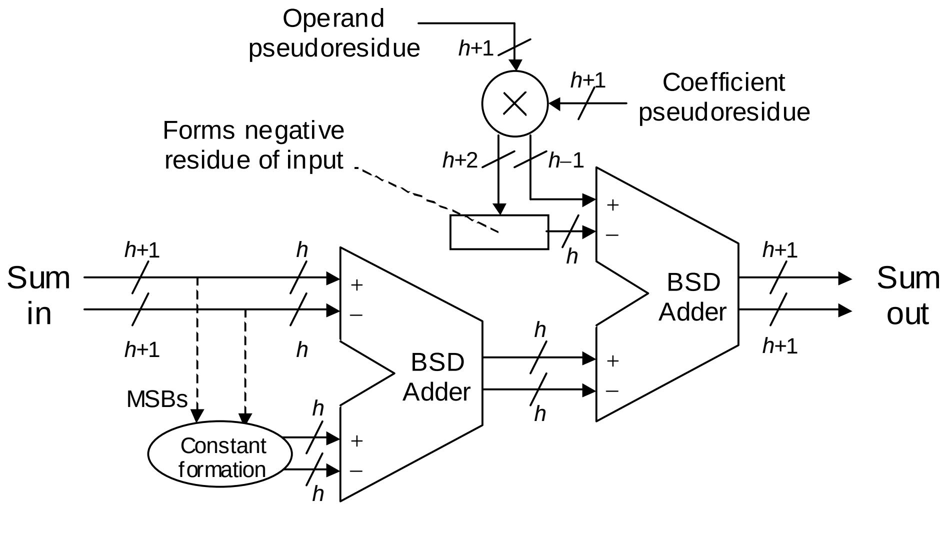

Fig. 4. RNS FIR filter implementation with BSD-encoded intermediate pseudoresidues.

5. AUXILIARY RNS OPERATIONS

Having discussed conventional arithmetic operations with pseudoresidues, we now want to show that certain auxiliary operations needed for successful implementation of RNS arithmetic are also facilitated by redundant residues. We have already covered binary to residue conversion in Section 3. Here, we discuss operations such as RNS to mixed-radix conversion, base extension, RNS to binary conversion, approximate sign detection, which is needed for division among other operations[6], and approximate Chinese remainder theorem (CRT) decoding[7].

Base extension refers to deriving a subset of RNS residues for a number from the remaining residues, provided that the number being represented is within the range of the latter subset. Both base extension and RNS to binary conversion can be performed by doing RNS to mixed-radix conversion first. For this reason, we present the latter in some detail.

Example 2 (RNS to mixed-radix conversion): Consider an RNS representation with the k moduli $m_{k-1} > \dots > m_1 > m_0$. The mixed-radix number system corresponding to this RNS has k digits, with the ith digit having the same range as the ith residue and a weight equal to $m_{i-1} \dots m_0$ (the weight is 1 for the digit in 0th position). In our case, the residues are redundant, so the mixed-radix number is also obtained with a redundant digit set. However, as in the case of fixed-radix positional systems, conversion from this redundant mixed-radix representation to nonredundant form is straightforward. The conversion process is iterative. In the ith iteration, the ith residue is taken to be the ith mixed-radix digit. This digit is subtracted from all remaining residues to get a number that is divisible by m_i. Division by m_i is then performed through multiplication of residues by the multiplicative inverse of m_i. To the extent that modular addition and multiplication are faster with pseudoresidues, the iterations are speeded up. At the end, either the redundant mixed-radix number is converted to nonredundant form or it is used directly to deduce the information of interest such as sign or relative magnitude through special lookahead circuits. Table II contains a numerical example in which $u = (6\ {}^-2\ {}^-2\ 1)_{\text{RNS}}$ is converted from a 4-modulus RNS (7, 5, 3, 2) to $(1\ {}^-2\ {}^-3\ \ 1)_{\text{MR}}$ in the equivalent mixed-radix format The computation is easily checked by noting that the mixed-radix number has the value 30 – 12 – 6 + 1 = 13. ■

Table II. Example of RNS to mixed-radix conversion with pseudoresidues.

RNS moduli →	$m_3 = 7$	$m_2 = 5$	$m_1 = 3$	$m_0 = 2$
RNS representation of u	6	${}^-2$	${}^-2$	1
Subtract $u_0 = 1$	5	${}^-3$	${}^-3$	
Multiplicative inverse of 2	4	3	2	
Multiply by 2^{-1}	6	${}^-4$	${}^-3$	
Subtract $u_1 = -3$	2	${}^-1$		
Multiplicative inverse of 3	5	2		
Multiply by 3^{-1}	3	${}^-2$		
Subtract $u_2 = -2$	5			
Multiplicative inverse of 5	3			
Multiply by 5^{-1}	1			
Mixed-radix digits v_i	1	${}^-2$	${}^-3$	1
Position weights w_i	30	6	2	1
Worth of each digit $v_i \times w_i$	30	–12	–6	1

Approximate CRT decoding, and consequently approximate sign detection, can be performed with pseudoresidues in virtually the same way as with residues. In approximate CRT decoding, the residues are used as indices into a table from which the normalized contribution of that residue to the magnitude of the number, with the entire range of numbers normalized to [0, 1), is read out with a few bits of precision. These values are then added, modulo-1, to form an approximate normalized magnitude in [0, 1). Modulo-1 addition means that any integral bit that is obtained is simply dropped. So, this is much simpler than mod-M addition required by standard CRT decoding, where M is a large integer constituting the dynamic range of our RNS. The wider the lookup tables used, the more precise this decoding becomes. If residues are supplied directly as inputs to separate tables, then the effect of using pseudoresidues is to double the size of the tables. To reduce the size penalty, one can use a technique similar to optimal bit-groupings for RNS to binary conversion[10]. In this way, the total table size is increased by a factor of $1 + 1/h$ instead of doubling.

6. FAULT TOLERANCE IMPLICATIONS

There are two ways in which one might discuss fault tolerance in connection with redundant residues: (1) Impact of redundant residues on conventional residue or inverse-residue checking of arithmetic operations, and (2) Application of error checking and/or fault tolerance methods to RNS arithmetic operations performed with pseudoresidues.

In residue checking of arithmetic operations, an operand u is encoded as (u, u mod m), where m is a suitably chosen check modulus. If u is k bits wide and mod-m residues are h bits wide, the relative representational redundancy is h/k. Using a DRS pseudoresidue in lieu of an ordinary residue increases the representational redundancy to $(h + 1)/k$. This small increase, from 4 bits to 5 bits, say, for a residue check attached to 32-bit data words, simplifies the initial residue encoding, residue arithmetic within the ALU needed to check the main computation, and the final residue generation for comparison with the predicted residue. On the negative side, the residue comparison itself becomes more complicated. This may be a worthwhile tradeoff, especially if within the context of a pipelined implementation, the residue comparison for one stage can be overlapped with computation in the next stage.

Error-checked or fault-tolerant arithmetic with pseudoresidues can take several forms. Here, we briefly discuss the use of redundant-modulus RNS whereby the RNS is extended with one or more moduli beyond the number needed to provide adequate dynamic range. Let us focus first on the use of an extra modulus m_k ($m_k > m_{k-1} > . . . > m_0$) in a k-modulus RNS. Computation is performed as usual within the $k + 1$ residue channels. Checking of the computation consists of predicting the kth residue on the basis of the remaining residues (base extension) and comparing the result to the computed value. Thus, the checking problem is of the same order of complexity as mixed-radix conversion discussed in Section 5. Again, as suggested in the preceding paragraph, the check computation can be overlapped with the main computation in the next pipeline stage to minimize the time overhead.

Error correction requires the use of more than one redundant modulus. For example, single error correction can be done with two redundant moduli m_{k+1} and m_k. Again the key component of the fault diagnosis circuits is base extension. Several algorithms for the correction process and its corresponding hardware realization are available. The pipelined systolic design suggested by Di Claudio, Orlandi, and Piazza[4] is particularly attractive for use with our pseudoresidues. In this scheme, the moduli, of which there are an even number, are paired and each pair of residues is derived from base extension. The results are compared to computed residues and one or the other value is selected by means of a relatively simple control circuit. Details remain to be worked out.

7. CONCLUSIONS

We have shown that using certain redundant residues leads to circuit simplifications and speed-up in RNS arithmetic. Our focus was on DRS pseudoresidues that offer the benefits of representational redundancy while keeping the overhead to the minimum of 1 bit per residue. However, other representations with the same amount of redundancy are possible. These include any residue set that forms a subinterval of $[-2^h, 2^h)$ or $[0, 2^{h+1})$. Whether or not other residue sets from this category offer advantages over DRS pseudoresidues remains to be established. Even though it appears that using greater redundancy to some extent defeats the short-word modular nature of RNS arithmetic, going beyond DRS (perhaps to 2 bits of redundancy per residue) might be justifiable with additional studies.

With regard to arithmetic operations, we have considered addition/subtraction, multioperand addition, multiplication, and inner-product computation. Besides the more difficult RNS arithmetic operations such as division and square-root extraction, future work may focus on the impact of certain specializations of general arithmetic operations. For example, it would be interesting to see if more efficient methods for multiplication and division by constants can be devised or if squaring can be significantly simplified[18].

Finally, fault tolerance implications need further study to quantify the benefits in terms of fault coverage and system reliability improvement. Extensions in this area include the possibility of encoding DRS pseudoresidues in BSD format. In the context of residue checking, this increases the overhead ($2h$ instead of $h + 1$ check bits for a k-bit data word). Some benefits of the BSD representation include faster arithmetic and the possibility of using parity-preserving transformations[15] to check the residue channel itself. This contributes to improved fault tolerance, particularly with regards to certain fault patterns that tend to have compensating effects on the main and residue computations.

REFERENCES

1. D.E. Atkins, "An introduction to the role of redundancy in computer arithmetic," *Computer*, 8, pp. 74-76, June 1975.
2. M. Bayoumi and G. Jullien, "A VLSI implementation of residue adders," *IEEE Trans. Circuits and Systems*, 34, pp. 284-288, 1987.
3. N. Burgess, "Efficient RNS to binary conversion using high-radix SRT division," *Proc. 32nd Asilomar Conf. Signals, Systems, and Computers*, pp. 1240-1243, IEEE Press, 1998.
4. E.D. Di Claudio, G. Orlandi, and F. Piazza, "A systolic redundant residue arithmetic error correction circuit," *IEEE Trans. Computers*, 42, pp. 427-432, 1993.
5. M. Dugdale, "VLSI implementation of residue adders based on binary adders," *IEEE Trans. Circuits and Systems II*, 39, pp. 325-329, 1992.
6. C.Y. Hung and B. Parhami, "An approximate sign detection method for residue numbers and its application to RNS division," *Computers & Mathematics with Applications*, 27, pp. 23-35, 1994.
7. C.Y. Hung and B. Parhami, "Error analysis of approximate CRT decoding," *IEEE Trans. Computers*, 44, No. 11, pp. 1344-1348, 1995
8. C.K. Koc and C.Y. Hung, "Fast algorithms for modular reduction," *IEE Proc. – Computers and Digital Techniques*, 145, pp. 265-271, July 1998.
9. H. Krishna et al, *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error-Control Techniques*, CRC Press, 1994.
10. B. Parhami and C.Y. Hung, "Optimal table lookup schemes for VLSI implementation of input/output conversions and other residue number operations," *VLSI Signal Processing VII* (Proc. IEEE Workshop), pp. 470-481, 1994.
11. B. Parhami, "A note on digital filter implementation using hybrid RNS-binary arithmetic," *Signal Processing*, 51, pp. 65-67, 1996.
12. B. Parhami, "Modular reduction by multi-level table lookup," *Proc. 40th Midwest Symp. Circuits and Systems*, pp. 381-384, IEEE Press, 1997.
13. B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford Univ. Press, 2000.
14. B. Parhami, "RNS representations with redundant residues," *Proc. 35th Asilomar Conf. Signals, Systems, and Computers*, pp. 1651-1655, IEEE Press, 2001.
15. B. Parhami, "Parity-preserving transformations in computer arithmetic," *Advanced Signal Processing Algorithms, Architectures, and Implementations XII* (Proc. SPIE Conf.), this volume, 2002.
16. S.J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Computers*, 43, No. 1, pp. 68-77, 1994.
17. S.J. Piestrak, "Design of high-speed residue-to-binary number system converter based on Chinese remainder theorem," *Proc. Int'l Conf. Computer Design*, pp. 508-511, 1994.
18. S.J. Piestrak, "Design of squarers modulo A with low-level pipelining," *IEEE Trans. Circuits and Systems II*, 49, pp. 31-41, 2002.
19. K.C. Posch and R. Posch, "Approaching encryption at ISDN speed using partial parallel modulus multiplication," *Microprocessing and Microprogramming*, 29, pp. 177-184, 1990.
20. K.C. Posch and R. Posch, "Modulo reduction in residue number systems," *IEEE Trans. Parallel and Distributed Systems*, 6, pp. 449-454, 1995.
21. M.A. Soderstrand, W.K. Jenkins, G.A. Jullien, and F.J. Taylor (Eds.), *Residue Number System Arithmetic*, IEEE Press, 1986.
22. N.S. Szabo and R.I. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology*, McGraw-Hill, 1967.