A General Division Algorithm for Residue Number Systems*

Jen-Shiun Chiang
Mi Lu

Electrical Engineering Department
Texas A&M University
College Station, Texas 77843-3128
U.S.A.

Abstract

We present in this paper a novel general algorithm for signed number division in Residue Number Systems (RNS). A parity checking technique is used to accomplish the sign and overflow detection in this algorithm. Compared with conventional methods of sign and overflow detection, the parity checking method is more efficient and practical. Sign magnitude arithmetic division is implemented using binary search. There is no restriction to the dividend and the divisor (except zero divisor), and no quotient estimation is necessary before the division is started. In hardware implementations, the storage of one table is required for parity checking, and all the other arithmetic operations are completed by calculations. Only simple operations are needed to accomplish this RNS division. All these characteristics have made our algorithm simple, efficient, and practical to be implemented on a real RNS divider.

1 Introduction

Residue Number Systems (abbreviated as RNS) are attractive to many people. A RNS is composed of moduli that are independent of each other. A number in the RNS is represented by the residue of each modulus, and arithmetic operations are accomplished based on each modulus. Since the moduli are independent of each other, there is no carry propagation among them, and it is easy to implement RNS computations on a multi-ALU system. The operation based on each modulus can be performed by a separate ALU, and all the ALU's can work concurrently. These characteristics allow RNS computations to be completed more quickly — an attractive feature for people who need high speed arithmetic operations [1, 2, 3].

Overflow detection, sign detection, number comparison, and division in RNS are very difficult and time consuming [4, 5]. These shortcomings limited most of the previous RNS applications to additions, subtractions, and multiplications.

The general division algorithms can be classified into two groups [6]: multiplicative algorithms and subtractive algorithms. The multiplicative algorithms compute the reciprocal of the divisor; the quotient is obtained by the multiplication of this reciprocal and the dividend. Subtractive algorithms recursively subtract the multiple of the denominator from the numerator until the difference becomes less than the denominator. The multiple is then the quotient.

There are several RNS division algorithms that are classified as multiplicative algorithms [5, 7, 8]. These multiplicative algorithms use the mixed radix number conversion to find the reciprocal of the divisor and to compare numbers. Iteratively, the approximate quotient is made closer to the accurate one. Due to the involvement of the mixed radix number conversion, the arithmetic calculation is very complicated and needs a lot of stored tables. Among these multiplicative algorithms, Kinoshita's algorithm [8] uses mixed radix numbers to approximate the quotient, and requires either a decimal divider or the storage of a very large table. Banerji's algorithm [7] also uses the mixed radix number approach and requires a lot of storage. Chren [5] criticizes that the standard deviation of the mean of the execution time needed in this algorithm is high. Chren's algorithm [5] is modified from Banerji's. Chren made some effort to reduce the storage and to improve the standard deviation of the mean execution time, but the storage and the computation time needed by the mixed radix number conversion are still expensive.

On the other hand, there are several algorithms classified as subtractive algorithms [4, 9, 10]. These subtractive algorithms use the conventional division approach, and no mixed radix number conversion is necessary. Therefore, the arithmetic calculations are not complicated. However, its number comparison and sign detection consume a lot of time and hardware. Szabo's algorithm [4] is not a general division algorithm but a scaling algorithm. Keir et al. [9] present two algorithms, both of which involve the binary expansion of the quotient. The speed of Keir's first algorithm is not desirable. His second algorithm uses look-up tables so that the hardware requirements are huge. Lin's algorithm [10] is a modification of the

*This research is partially supported by the National Science Foundation under Grant No. MIP-8809328.

well known CORDIC division algorithm, but it needs a lot of comparators, which is not practical for general computing applications.

In this paper we use parity checking for sign and overflow detection. Compared to conventional methods, the parity checking method is more efficient and practical. Based on the extension of overflow and sign detection techniques, a new signed RNS division algorithm is presented. Basically, this is a subtractive division algorithm using an efficient method to detect overflows and compare numbers. We use sign magnitude arithmetic for RNS division. In this division algorithm, binary search is used. There are no restrictions to dividends and divisors (except zero divisor), and no quotient estimation is necessary before the division is executed. In a hardware implementation, only the storage of one table is required to perform the parity checking, and almost all other correlated arithmetic operations are completed by calculations. All these characteristics have made our algorithm simple, efficient, and practical.

2 Residue Codes

2.1 Residue Numbers and the Arithmetic of Residue Numbers

The RNS representation of an integer is defined as follows. Let $\{m_1, m_2, ..., m_n\}$ be a set of positive numbers all greater than 1. The m_i's are called moduli and the n-tuple set $\{m_1, m_2, ..., m_n\}$ is called the modulus set. Consider an integer number X. For each modulus in set $\{m_1, m_2, ..., m_n\}$ we have $x_i = X \bmod m_i$ (denoted as $|X|_{m_i}$). Thus a number X in RNS can be represented as

$$X = (x_1, x_2, ..., x_n),$$

given a specific modulus set $\{m_1, m_2, ..., m_n\}$. In order to avoid redundancy, the moduli of a residue number system must be pair-wise relatively prime.

Let $M = \prod_{i=1}^{n} m_i$. It has been proved, in [4], that if $0 \leq X < M$, the number X corresponds one to one with the RNS representation. If the result of one calculation exceeds M, we say that overflow occurs. All the numbers should be within the dynamic range M (i.e., $0 \leq X < M$). Then, the RNS arithmetic can be performed.

If there are two numbers X and Y, the representations in RNS are as follows,

$$X = (x_1, x_2, ..., x_n)$$

and

$$Y = (y_1, y_2, ..., y_n).$$

We use $\otimes$ to represent the operator of additions, subtractions, and multiplications. The arithmetic in RNS can be represented as follows,

$$X \otimes Y = (z_1, z_2, ..., z_n)$$

where

$$z_i = |x_i \otimes y_i|_{m_i}.$$

From the definition of the mod operation, all moduli are positive. x_i may be less than y_i, which yields $x_i - y_i < 0$. In mod operation, if $x_i - y_i < 0$, then z_i is defined as

$$z_i = m_i + (x_i - y_i). \tag{1}$$

2.2 Number Comparison for Unsigned Numbers

As we know, number comparison and overflow detection in RNS are very difficult. It is necessary to find methods that are efficient, practical, and easily implemented.

Let parity indicate whether an integer number is even or odd. We say two numbers are of the same parity if they are both even or both odd. Otherwise the two numbers are said to be of different parities. We will use the properties of the parities of numbers to accomplish the number comparison.

In a survey of Soviet research on residue number systems [11], Miller et al. defined a function called the *core* function and explored its properties as follows:

Let $m_1, m_2, ..., m_n$ be the relatively prime moduli of a residue number system with product M. For fixed integers $w_1, w_2, ..., w_n$, the core R_N of an integer is defined as follows:

$$R_N = \sum_{i=1}^{m} w_i \left\lfloor \frac{N}{m_i} \right\rfloor,$$

where $\lfloor X \rfloor$ denotes the greatest integer function which is not greater than X. The coefficients w_i are fixed for the moduli set and do not depend on the integer N.

Theorem 1 *Let the moduli m_i and the core R_M be odd. Let $(a_1, a_2, ..., a_n)$ and $(b_1, b_2, ..., b_n)$ be the residue representations of integers A, $B \in [0, M)$. Then $A + B$ causes an overflow if*

(i) $(a_1 + b_1, ..., a_n + b_n)$ is odd, and A and B have the same parity; or

(ii) $(a_1 + b_1, ..., a_n + b_n)$ is even and A and B have different parities.

Let the interval $[0, M/2]$ represent positive numbers and the interval $(M/2, M)$ represent negative numbers.

Theorem 2 *If the moduli m_i and the core R_M are odd, and $(a_1, a_2, ..., a_n)$ is the residue representation of a non-zero integer $A \in [0, M)$, then A is positive if and only if*

$$(|2a_1|_{m_1}, ..., |2a_n|_{m_n})$$

is even.

According to the theory of core functions, if the core function of an RNS number is known, it is easy to detect overflows and the sign of the number. However, it is very difficult to find the core function in RNS by the method given in [11]. Discarding the core function and revising the theorems mentioned by

Miller et al. [11], the following theorems express the properties we need for comparison of unsigned numbers. Consider the whole dynamic range, $[0, M)$ of positive numbers from 0 to $(M-1)$. Let all m_i's in the modulus set $(m_1, m_2, ..., m_n)$ be odd numbers, and let $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_n)$ be two RNS numbers. Suppose $Z = X - Y = (z_1, z_2, ..., z_n)$, then we have the following theorem.

Theorem 3 *Let X and Y have the same parity and $Z = X - Y$. $X \geq Y$, iff Z is an even number. $X < Y$, iff Z is an odd number.*

Proof:
If $X \geq Y$, then $X - Y \geq 0$ and Z equals $X - Y$. We know from the mathematical axioms that the two numbers are with the same parity, and the result of the subtraction should be an even number. Therefore, $X \geq Y$ implies that Z is an even number.

On the other hand, suppose that Z is an even number and X and Y are with the same parity. If $X < Y$, then $X - Y < 0$. From equation (1) we have $Z = X - Y + M$. Since M is an odd number and $X - Y$ is even, Z must be an odd number. This contradicts the assumption that Z is even. Therefore, if Z is an even number and X and Y are with the same parity, then $X \geq Y$.

If $X < Y$, then $X - Y < 0$, from (1) we have $Z = X - Y + M$. Since m_i's are all odd numbers, M should be an odd number. In addition, $(X - Y)$ is an even number and this implies that Z is an odd number. Therefore, it is obvious that if $X < Y$, Z is an odd number.

On the other hand, suppose that Z is an odd number and X and Y are with the same parity. If $X \geq Y$, then $X - Y \geq 0$. Since X and Y are with the same parity, Z must be an even number. This contradicts the assumption that Z is an odd number. Therefore, if Z is an odd number and X and Y are with the same parity, then $X < Y$.

Theorem 3 shows us a method to compare two numbers if the parities of these two numbers are the same. Similarly, if the parities of two numbers are different, then the following theorem can tell us which one is bigger.

Theorem 4 *Let X and Y have different parities and $Z = X - Y$. $X \geq Y$, iff Z is an odd number. $X < Y$, iff Z is an even number.*

Proof:
If $X \geq Y$, then $X - Y \geq 0$ and Z equals $X - Y$. We know that the two numbers have different parities, and the result of the subtraction should be an odd number. Therefore, $X \geq Y$ implies that Z is an odd number.

On the other hand, suppose that Z is an odd number and X and Y have different parities. If $X < Y$, then $X - Y < 0$. From equation (1) we have $Z = X - Y + M$. Since M is an odd number and $X - Y$ is odd, Z must be an even number. This contradicts the assumption that Z is odd. Therefore, if Z is an odd number and X and Y have different parities, then $X \geq Y$.

If $X < Y$, then $X - Y < 0$, from (1) we have $Z = X - Y + M$. Since m_i's are all odd numbers, M should be an odd number. In addition, $(X - Y)$ is an odd number and this implies that Z is an even number. Therefore, $X < Y$ implies that Z is an even number.

On the other hand, suppose that Z is an even number and X and Y have different parities. If $X \geq Y$, then $X - Y \geq 0$. Since X and Y have different parities, Z must be an odd number. This contradicts the assumption that Z is an even number. Therefore, if Z is an even number and X and Y are with different parities, then $X < Y$.

Table 1 is referred to when performing parity checking for number comparisons. The decimal numbers under the entry "#" are corresponding to the residue numbers for modulus set $(3, 5, 7)$, and the parities of them are given under the entry P.

The following is an example to illustrate the above theorems.

Example 1
Let the moduli be $m_1 = 3$, $m_2 = 5$, $m_3 = 7$, and hence $M = 3 \cdot 5 \cdot 7 = 105$. Consider $X_1 = (0, 3, 5)$ and $Y_1 = (1, 3, 0)$. From calculation we have $Z_1 = X_1 - Y_1 = (2, 0, 5)$. Look up Table 1, the parities of X_1, Y_1, and Z_1 are odd, even, and odd respectively. From Theorem 4 we know $X_1 > Y_1$.

In the decimal number system, $X_1 = 33$, $Y_1 = 28$, and $Z_1 = 5$, and the result is obvious.

Note that if the number M is big, the parity table may be huge.

2.3 Signed Numbers and the Properties

The method used to represent negative numbers in RNS is similar to that used in conventional radix number systems. Letting the dynamic range be M, we can define the positive and negative numbers as follows [4].

Definition 1 *If the dynamic range $M = \prod_{i=1}^{n} m_i$, then the range of a positive number X is defined as $0 \leq X \leq \lfloor \frac{M}{2} \rfloor$, and the range of a negative number Y is defined as $\lfloor \frac{M}{2} \rfloor < Y < M$. Like the radix number system, the negative numbers, -1, -2, ..., $-(\lfloor \frac{M}{2} \rfloor - 1)$, $-\lfloor \frac{M}{2} \rfloor$, are represented by the numbers, $(M-1)$, $(M-2)$, ..., $(\lfloor \frac{M}{2} \rfloor + 2)$, $(\lfloor \frac{M}{2} \rfloor + 1)$, respectively.*

Notice here, 0 is considered as a positive number.

From Definition 1 we can find that the complement of X is $M - X$. In a similar way, the representation of the complement of a number in RNS can be found in the following lemma.

Lemma 1 *Let the modulus set be $\{m_1, m_2, ..., m_n\}$, and the corresponding modulus set of a positive number X in RNS be $\{x_1, x_2, ..., x_n\}$. $-X$ in RNS can be represented by the complement of X which is equal to $\{(m_1 - x_1)_{m_1}, (m_2 - x_2)_{m_2}, ..., (m_n - x_n)_{m_n}\}$.*

Proof:
From Definition 1, $-X$ in RNS corresponds to $M - X$. Applying equation (1), the corresponding modulus set of $-X$ is $\{(m_1 - x_1)_{m_1}, (m_2 - x_2)_{m_2}, ..., (m_n - x_n)_{m_n}\}$.

The dynamic range of RNS can be divided into two halves, one for positive numbers and the other for negative numbers, as described in Definition 1. If the

#	(3	5	7)	P	#	(3	5	7)	P	#	(3	5	7)	P	#	(3	5	7)	P	#	(3	5	7)	P
0	0	0	0	0	21	0	1	0	1	42	0	2	0	0	63	0	3	0	1	84	0	4	0	0
1	1	1	1	1	22	1	2	1	0	43	1	3	1	1	64	1	4	1	0	85	1	0	1	1
2	2	2	2	0	23	2	3	2	1	44	2	4	2	0	65	2	0	2	1	86	2	1	2	0
3	0	3	3	1	24	0	4	3	0	45	0	0	3	1	66	0	1	3	0	87	0	2	3	1
4	1	4	4	0	25	1	0	4	1	46	1	1	4	0	67	1	2	4	1	88	1	3	4	0
5	2	0	5	1	26	2	1	5	0	47	2	2	5	1	68	2	3	5	0	89	2	4	5	1
6	0	1	6	0	27	0	2	6	1	48	0	3	6	0	69	0	4	6	1	90	0	0	6	0
7	1	2	0	1	28	1	3	0	0	49	1	4	0	1	70	1	0	0	0	91	1	1	0	1
8	2	3	1	0	29	2	4	1	1	50	2	0	1	0	71	2	1	1	1	92	2	2	1	0
9	0	4	2	1	30	0	0	2	0	51	0	1	2	1	72	0	2	2	0	93	0	3	2	1
10	1	0	3	0	31	1	1	3	1	52	1	2	3	0	73	1	3	3	1	94	1	4	3	0
11	2	1	4	1	32	2	2	4	0	53	2	3	4	1	74	2	4	4	0	95	2	0	4	1
12	0	2	5	0	33	0	3	5	1	54	0	4	5	0	75	0	0	5	1	96	0	1	5	0
13	1	3	6	1	34	1	4	6	0	55	1	0	6	1	76	1	1	6	0	97	1	2	6	1
14	2	4	0	0	35	2	0	0	1	56	2	1	0	0	77	2	2	0	1	98	2	3	0	0
15	0	0	1	1	36	0	1	1	0	57	0	2	1	1	78	0	3	1	0	99	0	4	1	1
16	1	1	2	0	37	1	2	2	1	58	1	3	2	0	79	1	4	2	1	100	1	0	2	0
17	2	2	3	1	38	2	3	3	0	59	2	4	3	1	80	2	0	3	0	101	2	1	3	1
18	0	3	4	0	39	0	4	4	1	60	0	0	4	0	81	0	1	4	1	102	0	2	4	0
19	1	4	5	1	40	1	0	5	0	61	1	1	5	1	82	1	2	5	0	103	1	3	5	1
20	2	0	6	0	41	2	1	6	1	62	2	2	6	0	83	2	3	6	1	104	2	4	6	0

Table 1: Parity Table for Modulus Set (3,5,7)

moduli are all pair-wise prime and are all odd numbers, then the maximum positive number is $\frac{M-1}{2}$. A negative number's magnitude can be found by applying Definition 1 and Lemma 1; it must fall in the positive range. In this case, the unsigned number comparisons described in Theorem 3 and 4 are applicable. The following definition is to define overflows in the positive range of the RNS.

Definition 2 *Suppose that there are two positive numbers in RNS,* $X = (x_1, x_2, ..., x_n)$ *and* $Y = (y_1, y_2, ..., y_n)$. *Overflow exists if* $X + Y > \frac{M-1}{2}$.

Notice that Definition 2 considers the case that both X and Y are positive numbers. This definition is referred to when we discuss overflow detection in the addition of two numbers.

Corollary 1 *The overflow detection theory in Definition 2 applies to the addition of only two numbers.*
Proof:
The maximal number in Definition 2 is $(\frac{M-1}{2})$, and the maximal sum of two numbers is $(M-1)$ which is within the dynamic range. If there are 3 numbers or more, the maximal sum of those numbers is greater than $(M-1)$ which is out of the dynamic range, and by the definition of RNS the sum is not correct. Therefore, the overflow detection theory described in Definition 2 is correct only for two-number additions.

2.4 Multiplicative Inverse

Consider the number $|b|_m$, Szabo and Tanaka [4] define the multiplicative inverse as follows.

Definition 3 *If* $0 \leq a < m$ *and* $|ab|_m = 1$, *a is called the multiplicative inverse of b* mod m, *and is denoted as* $|b^{-1}|_m$.

Notice that the multiplicative inverse of a number does not always exist. The following theorem from [4] describes the condition of its existence and the proof is omitted.

Theorem 5 *The quantity* $|b^{-1}|_m$ *exists if and only if the greatest common divisor of b and m,* $gcd(b, m)$, *is equal to 1, and* $|b|_m \neq 0$. *In this case* $|b^{-1}|_m$ *is unique.*

We have already developed several efficient methods (given in Theorem 3, Theorem 4, and Definition 2) for number comparison and overflow detection in order to perform the addition of two positive numbers. We use these theorems to derive the division algorithm for signed RNS numbers in the following section.

3 Division Algorithm

We now present a division algorithm in RNS using sign magnitude arithmetic and binary search.

3.1 Descriptions of the Algorithm

Given two numbers, dividend X and divisor Y, the division in RNS is to find the quotient $Z = \lfloor \frac{X}{Y} \rfloor$, where $\lfloor \frac{X}{Y} \rfloor$ denotes the greatest integer which is not greater than $\frac{X}{Y}$. As mentioned before, this algorithm is classified as a subtractive algorithm. Therefore, it is necessary to detect the sign in the subtraction and the overflow in the addition. Theorem 3 and 4 provide an efficient way to perform the number comparison. The absolute value of the dividend and the divisor are used when performing the division calculation, and the the overflow in the addition of two numbers is detected by applying Definition 2. In addition, the signs of the dividend and the divisor need to be detected, and the negative numbers need to be complemented. After finishing the division on two absolute values, it is necessary to transfer the quotient to the proper representation(positive or negative) in RNS. Given modulus set $(m_1, m_2, ..., m_n)$ with dividend $X = (x_1, x_2, ..., x_n)$ and divisor $Y = (y_1, y_2, ..., y_n)$, we are to find the quotient Z, where $Z = \lfloor \frac{X}{Y} \rfloor$. The dynamic range, M, of the RNS is $M = \prod_{i=1}^{n} m_i$. Corollary 1 tells us that the overflow detection can be applied only to the addition of two numbers, a special case of which is the

addition of two equal numbers. In other words, multiplying a number by 2 is allowed, and our algorithm is developed on this basis (see Part II below).

This algorithm can be divided into four parts. Part I detects the signs of the dividend and the divisor and transfers them to positive numbers. Part II finds 2^j, such that $Y \cdot 2^j \leq X < Y \cdot 2^{j+1}$, and finds the difference between 2^j and the quotient. Part III deals with the case $Y \cdot 2^j \leq X \leq \frac{M-1}{2} < Y \cdot 2^{j+1}$ which is from Part II, and finds out the difference between 2^j and the quotient. Part IV transfers the quotient to the proper representation in RNS (positive or negative).

Part I.

The largest number in the positive range of the RNS is $\frac{M-1}{2}$, and for convenience we set a variable $M_p = \frac{M-1}{2}$. Using Theorem 3 and 4, we compare the dividend and the divisor with M_p. If the dividend or the divisor is less than or equal to M_p, then the dividend or the divisor is positive, and nothing needs to be changed. On the other hand, if the dividend or the divisor is greater than M_p, then the dividend or the divisor is negative, and it should be complemented. If either the dividend or the divisor is negative, we have to set the sign variable, SIGN, to 1. SIGN will be used to convert the quotient to a proper form in Part IV.

Part II.

We find the proper 2^j such that $Y \cdot 2^j \leq X < 2^{j+1}$ in the following way. Two variables, LowerBound and UpperBound, are set to record the range in which the value of the quotient is to be found. The LowerBound and the UpperBound will dynamically change, as the algorithm is executed. In iteration j, LowerBound= 2^j and UpperBound= 2^{j+1}, we repeatedly compare $(2^j \cdot Y)$ with X and detect whether $(2^{j+1} \cdot Y)$ is greater than $\frac{M-1}{2}$, until we find some j, denoted as $\hat{j}$, such that $Y \cdot 2^{\hat{j}} \leq X < Y \cdot 2^{\hat{j}+1}$. QuotientBase records the LowerBound when the procedure halts, and it is unchanged until the end of the division operation. QuotientExt records the difference between the exact quotient value and the QuotientBase, and the initial value of QuotientExt is set to 0. The final value of the quotient is equal to QuotientBase+QuotientExt.

In each iteration, the UpperBound is updated by doubling its value. Two cases may occur when the above procedure halts. In one case, (UpperBound$\cdot Y$) is smaller than $\frac{M-1}{2}$. Then a binary search starts to find the difference between QuotientBase and the quotient, and we need $\hat{j} - 1$ steps to finish this part, since 2^j integers exist in the range $[2^j, 2^{j+1})$. In each step of the binary search, we have to compare X with $\left(Y \cdot \frac{\text{UpperBound+LowerBound}}{2}\right)$. Here, $\left(Y \cdot \frac{\text{UpperBound+LowerBound}}{2}\right)$ for each modulus, $[|2^{-1}|_{m_i} \cdot |Y|_{m_i} \cdot (|\text{UpperBound}|_{m_i} + |\text{LowerBound}|_{m_i})]_{m_i}$, is to be found. Hence, the multiplicative inverse of 2, $|2^{-1}|_{m_i}$, needs to be prepared. If $\left(X - Y \cdot \frac{\text{UpperBound+LowerBound}}{2}\right) < 0$, then set UpperBound= $\frac{\text{UpperBound+LowerBound}}{2}$ and QuotientExt= 2·QuotientExt. Otherwise set LowerBound= $\frac{\text{UpperBound+LowerBound}}{2}$ and QuotientExt= (2·QuotientExt+1). When this procedure is finished, go to Part IV.

In the other case, UpperBound$\cdot Y$ may be greater than $\frac{M-1}{2}$ and overflow thus occurs. Then we go to Part III.

Part III.

If there is an overflow, it means that $(Z \cdot Y)$ lies between $Y \cdot 2^j$ and $\frac{M-1}{2}$. We therefore update UpperBound as $\frac{\text{UpperBound+LowerBound}}{2} = 2^j + 2^{j+1}$, and LowerBound as 2^j. QuotientExt is updated as QuotientExt=(QuotientExt·2). Then we examine whether $\left(\frac{\text{UpperBound+LowerBound}}{2} \cdot Y\right)$ overflows again. Continue this procedure until $\left(\frac{\text{UpperBound+LowerBound}}{2} \cdot Y\right)$ does not overflow. If $\left(\frac{\text{UpperBound+LowerBound}}{2} \cdot Y\right)$ does not overflow and $\left(X - \frac{\text{UpperBound+LowerBound}}{2} \cdot Y\right) \geq 0$, set LowerBound= $\frac{\text{UpperBound+LowerBound}}{2}$ and QuotientExt=(QuotientExt·2 + 1), and detect overflow again. If $\left(\frac{\text{UpperBound+LowerBound}}{2} \cdot Y\right)$ does not overflow and $\left(X - \frac{\text{UpperBound+LowerBound}}{2} \cdot Y\right) < 0$, set UpperBound= $\frac{\text{UpperBound+LowerBound}}{2}$ and QuotientExt=(QuotientExt·2), and perform the similar operations as defined in the binary search in Part II. As in Part II, after $\hat{j} - 1$ steps go to Part IV.

In the above procedure, if (UpperBound − LowerBound) = 1, then the job is finished. Let the quotient equal LowerBound and go to Part IV to get the proper quotient expression (as a positive number or a negative number).

Part IV.

The absolute value of Quotient equals the sum of QuotientExt and QuotientBase. From Part I, the exact quotient may be negative. Therefore, if SIGN=1, the absolute value of the found quotient should be complemented.

3.2 The Correctness of the Algorithm

The absolute value of a quotient is equal to the quotient of the absolute value of the dividend and the absolute value of the divisor. The sign of the quotient depends on the signs of the dividend and the divisor. If the signs of the dividend and the divisor are different, then the quotient is negative. Otherwise, the quotient is positive.

The dynamic range of the positive numbers in RNS, $[0, \frac{M-1}{2} = M_p]$, can be divided into several subintervals. The boundaries of the subintervals are as follows:

0, $(2^0 \cdot Y)$, $(2^1 \cdot Y)$, $(2^2 \cdot Y)$, ..., $(2^j \cdot Y)$, ..., $(2^n \cdot Y)$, M_p.

We are to find the proper j such that $2^j \cdot Y < X <$

$2^{j+1} \cdot Y$, i.e., to find the subinterval $[2^j, 2^{j+1})$ in which the quotient lies. Usually M_p is not a power of 2, therefore, if $2^n \cdot Y < X < M_p$, i.e., when $j = n$, the search for j has to be stopped, and a binary search is to start. In this case two variables, UpperBound and LowerBound, are set as UpperBound= $2^{(n+1)}$, and LowerBound= 2^n. Then we continuously calculate $\frac{\text{UpperBound}+\text{LowerBound}}{2}$ for the quotient estimation, and compare $\left(\frac{\text{UpperBound}+\text{LowerBound}}{2} \cdot Y\right)$ with X. Recursively substituting the UpperBound or LowerBound with this estimated value, we can reduce the range in which the quotient lies, and finally find the quotient value. Notice that initially, UpperBound= 2·LowerBound, hence

$$\frac{\text{UpperBound} + \text{LowerBound}}{2} = \text{LowerBound} + \frac{\text{LowerBound}}{2}. \quad (2)$$

If UpperBound is updated with this value, then the new estimation is equal to

$$\frac{\text{LowerBound} + \text{LowerBound} + \frac{\text{LowerBound}}{2}}{2} = \text{LowerBound} + \frac{\text{LowerBound}}{4}.$$

If UpperBound remains the same, as big as 2·LowerBound, and LowerBound is updated with the value in (2), then the new estimation will be

$$\text{LowerBound} + \frac{\text{LowerBound} + \frac{\text{LowerBound}}{2}}{2} = \text{LowerBound} + \frac{\text{LowerBound}}{2} + \frac{\text{LowerBound}}{4},$$

and so on. It is easy to find that the quotient can be represented by

$$\text{LowerBound} + \sum_i \lambda_i \frac{\text{LowerBound}}{2^i}, \quad \text{with } \lambda_i = 0 \text{ or } 1.$$

The first term is then recorded as QuotientBase in the algorithm and the second term QuotientExt. The quotient in the division is found as QuotientBase+QuotientExt which is hence correct.

3.3 Division Algorithm

The flow chart of the algorithm is shown in Figure 1, and the detailed division algorithm is as following.

```
/* Suppose that m1, m2, ..., and mN are
/* N moduli which are pair-wise prime and
/* all odd numbers. Let M=m1*m2*m3*...*mN,
/* and Mp=(M-1)/2 be the largest positive
/* number. We use the equation
/* Dividend/Divisor=Quotient+
/* Remainder/Divisor, and the Quotient is
```

The symbols used in the flowchart are listed below each followed by the corresponding variable used in the algorithm.
X: Dividend,
Y: Divisor,
Z: Quotient,
UB: UpperBound,
LB: LowerBound,
QE: QuotientExt,
QB: QuotientBase,
B: Bounding,
$\overline{Z}$: COMPLEMENT(Z).

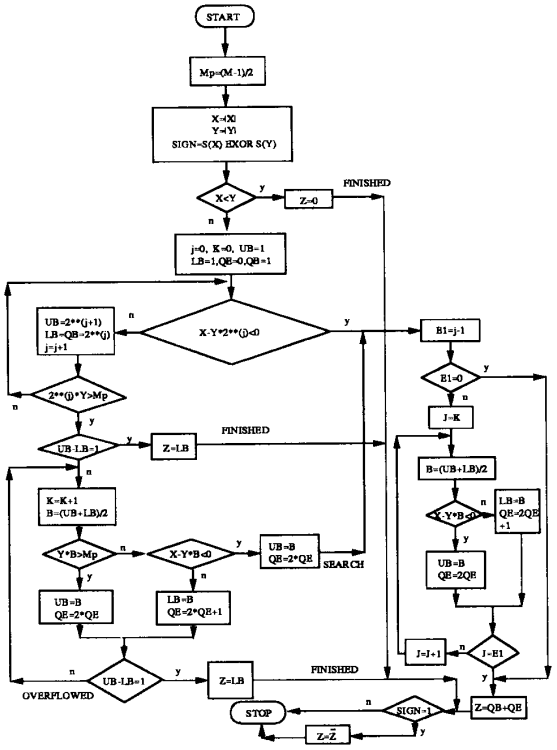

Figure 1: Flowchart of the Division Algorithm

```
/* is to be found.

START PROCEDURE;
/* Check the signs of Dividend and Divisor
/* and set a register SIGN as
/* SIGN=[S(Dividend) EXOR S(Divisor)] to
/* save the sign of the Quotient, where
/* S(..) means the sign of .. . Define S(..)
/* =1 for a negative number, and S(..)=0 for
/* a positive number.
  Mp=(M-1)/2
  S(Dividend)=0, S(Divisor)=0
  IF Dividend < 0
```

```
    THEN Dividend=COMPLEMENT(Dividend)
         /* COMPLEMENT(Dividend)=M-Dividend
      S(Dividend)=1
  END IF;
  IF Divisor < 0
    THEN Divisor=COMPLEMENT(Divisor)
      S(Divisor)=1
  END IF;
  SIGN=S(Dividend) EXOR S(Divisor)
  IF Dividend<Divisor
     THEN Quotient=0, GO TO FINISHED;
/* Find out which interval the quotient
/* falls in, i.e., find j, such that
/* 2**j<Quotient<2**(j+1)
  j=0, K=0, UpperBound=1, LowerBound=1,
  QuotientExt=0, QuotientBase=1
   WHILE (Dividend-Divisor*2**(j))>=0
    THEN DO
     UpperBound=2**(j+1)
     LowerBound=QuotientBase=2**(j)
     j=j+1
     IF 2**(j)*Divisor>Mp
       THEN
         IF (UpperBound-LowerBound)=1
         THEN Quotient=LowerBound,
              GO TO FINISHED;
         END IF;
OVERFLOW:K=K+1
         Bounding=(UpperBound+LowerBound)/2
         IF Divisor*Bounding>Mp
          THEN
           UpperBound=Bounding
           QuotientExt=2*QuotientExt
          ELSE
           IF (Dividend-Divisor*Bounding)<0
            THEN
              UpperBound=Bounding
              QuotientExt=2*QuotientExt
              GO TO SEARCH
            ELSE
              LowerBound=Bounding
              QuotientExt=2*QuotientExt+1
            END IF;
          END IF;
          IF (UpperBound-LowerBound)=1
           THEN
             Quotient=LowerBound
             GO TO FINISHED;
           ELSE
             GO TO OVERFLOW;
          END IF;
     END IF;
   END WHILE;
/* Binary search for QuotientExt
SEARCH:E1=j-1
      IF E1<>0
       THEN
        FOR J=K TO E1
        DO
         Bounding=(UpperBound+LowerBound)/2
         IF (Dividend-Divisor*Bounding)<0
          THEN
           UpperBound=Bounding
           QuotientExt=2*QuotientExt
          ELSE
           LowerBound=Bounding
           QuotientExt=2*QuotientExt+1
          END IF;
        END FOR;
      END IF;
/* The found quotient equals the sum of
/* QuotientBase and QuotientExt. If SIGN=1
/* it means that Quotient is a negative
/* number, then the final Quotient is the
/* complement of the found Quotient.
Quotient=QuotientBase+QuotientExt
FINISHED:IF SIGN=1
         THEN Quotient=COMPLEMENT(Quotient)
         END IF;
END PROCEDURE;
```

3.4 Example of the Division Algorithm

Suppose that moduli are $m_1 = 3$, $m_2 = 5$, and $m_3 = 7$. Given $X = (2,1,1) = -34$ and $Y = (2,0,5) = 5$, find quotient $Z = \frac{X}{Y}$.

Since the moduli are $m_1 = 3$, $m_2 = 5$, and $m_3 = 7$, we can find $M = m_1 \cdot m_2 \cdot m_3 = 105$, $M_p = \frac{M-1}{2} = 52 = (1,2,3)$. The multiplicative inverses of 2, which are used in the calculation of $\frac{\text{UpperBound}+\text{LowerBound}}{2}$, corresponding to m_1, m_2, and m_3 are $|2^{-1}|_{m_1} = 2$, $|2^{-1}|_{m_2} = 3$, and $|2^{-1}|_{m_3} = 4$ respectively. Parity checking uses Table 1. The quotient can be calculated in the following steps with the required variables. The short notations of these variables are listed in Figure 1.

1. $S(-34) = 1$,	$S[(2,1,1)] = 1$,
$S(5) = 0$,	$S[(2,0,5)] = 0$,
SIGN=1,	SIGN=1,
COMP$(-34) = 34$.	COMP$[(2,1,1)] = (1,4,6)$.
2. $34 > 5 \cdot 2^0$,	$(1,4,6) > (2,0,5)$,
$j = 0$.	$j = 0$.
3. $34 > 5 \cdot 2^1$,	$(1,4,6) > (2,0,5) \cdot (2,2,2)$ $= (1,0,3)$,
$j = 1$.	$j = 1$.
4. $34 > 5 \cdot 2^2$,	$(1,4,6) > (1,0,3) \cdot (2,2,2)$ $= (2,0,6)$,
$j = 2$.	$j = 2$.
5. $5 \cdot 2^3 > 34 > 5 \cdot 2^2$,	$(2,0,5) \cdot (2,2,2) = (1,0,5)$ $> (1,4,6) > (2,0,6)$,
QB=2^2,	QB=$(1,4,4)$,
$J = 0$,	$J = 0$,
UB= 2^3,	UB= $(2,2,2) \cdot (2,2,2) \cdot$ $(2,2,2) = (2,3,1)$,
LB= 2^2,	LB= $(2,2,2) \cdot (2,2,2)$ $= (1,4,4)$,
B= $\frac{2^3+2^2}{2} = 6$,	B= $\frac{(2,3,1)+(1,4,4)}{2} = (0,1,6)$,
QE= $2 \cdot 0 + 1 = 1$,	QE= $(0,0,0) + (1,1,1)$ $= (1,1,1)$,
Set LB=B.	Set LB=B= $(0,1,6)$.
6. $5 \cdot 2^3 > 34 > 5 \cdot 6$,	$(1,0,5) > (1,4,6) >$ $(2,0,5) \cdot (0,1,6) = (0,0,2)$,
$J = 1$,	$J = 1$,

UB= 2^3,	UB= $(2,3,1)$,
LB= 6,	LB= $(0,1,6)$,
B= $\frac{2^3+6}{2}=7$,	B= $\frac{(2,3,1)+(0,1,6)}{2}=(1,2,0)$,
QE= $2\cdot 1+0=2$,	QE= $(2,2,2)\cdot(1,1,1)$
	$+(0,0,0)=(2,2,2)$,
Set UB=B.	Set UB=B= $(1,2,0)$.
7. $\|Z\|$ =QB + QE	$\|Z\|$ =QB+QE= $(1,4,4)$
$=2^2+2=6$.	$+(2,2,2)=(0,1,6)$.
8. SIGN= 1,	SIGN= 1,
Z=COMP(6),	Z=COMP$(0,1,6)$,
$=-6$.	$=(0,4,1)$.

3.5 Discussions

This algorithm requires four parts of calculation. Constant time is needed in Part I to find the absolute values of the dividend and the divisor, and in Part IV to transfer the absolute value of the quotient, $|Z|$, to the proper form. In Part II and III, our algorithm needs $(2\cdot\log_2 Z)$ steps to finish the division operation. The first $\log_2 Z$ steps find the range which the quotient falls in, and the second $\log_2 Z$ steps find the difference between QuotientBase and the quotient. Each step needs several RNS addition and subtraction operations, one RNS multiplication, and a table look-up for the parities. The RNS arithmetic operations do not need quotient estimation, base extension, or mixed radix number conversion, which makes this algorithm very fast and easy to implement compared to previous proposals.

4 Conclusions

We have presented a division algorithm which needs only simple RNS arithmetic operations, and can be easily implemented. This is a general division algorithm, with no restrictions to either dividend or divisor. No estimation of the quotient is required before the division is executed. These characteristics make the calculation less complicated, more efficient, and speedier.

We also presented a very good and easy technique for overflow detection and number comparison. In the traditional way of detecting overflow and comparing numbers in RNS, mixed radix numbers have to be used. This is time consuming and requires complex hardware. Our method is more efficient and less complicated than the existing algorithms.

A parity-checking technique is presented in this paper for number comparisons and overflow detection. With today's advanced VLSI technology, we will have no difficulty building a parity table that lists all the moduli with parities. Some small tables may also be needed to store data such as the values of the multiplicative inverse of 2, $|2^{-1}|_{m_i}$. Except the tables mentioned above, no other table are required, and all we need is simple arithmetic calculations. This algorithm can be easily implemented on hardware and can achieve good time performance which is logarithmic to the size of the quotient.

Acknowledgements

The authors would like to thank the anonymous reviewer for his helpful comments.

References

[1] W. K. Jenkins and B. J. Leon, "The use of residue number systems in the design of finite impulse response digital filters," *IEEE Transactions on Circuits Systems*, vol. CAS-24, no. 4, pp. 199–201, 1973.

[2] F. J. Taylor, "A VLSI residue arithmetic multiplier," *IEEE Transactions on Computers*, vol. C-31, pp. 540–546, June 1982.

[3] D. D. Miller and J. N. Polky, "An implementation of the LMS algorithm in the residue number system," *IEEE Transactions on Circuits System*, vol. CAS-31, pp. 452–461, May 1984.

[4] N. S. Szabo and R. I. Tanaka, *Residue Arithmetic and Its Application to Computer Technology*. New York: McGraw-Hill, 1967.

[5] W. A. Chren Jr., "A new residue number system division algorithm," *Computers Math. Applic.*, vol. 19, no. 7, pp. 13–29, 1990.

[6] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital System Designers*, ch. 5, p. 172. New York: Holt, Rinehart & Winston, 1982.

[7] D. K. Banerji, T. Y. Cheung, and V. Ganesan, "A high-speed division method in residue arithmetic," in *5th IEEE Symp. on Computer Arithmetic*, pp. 158–164, 1981.

[8] E. Kinoshita, H. Kosako, and Y. Kojima, "General division in the symmetric residue number system," *IEEE Transactions on Computers*, vol. C-22, pp. 134–142, February 1973.

[9] Y. A. Keir, P. W. Cheney, and M. Tannenbaum, "Division and overflow detection in residue number systems," *IRE Transactions on Electronic Computers*, vol. EC-11, pp. 501–507, August 1962.

[10] M.-L. Lin, E. Leiss, and B. McInnis, "Division and sign detection algorithm for residue number systems," *Computers Math. Applic.*, vol. 10, no. 4/5, pp. 331–342, 1984.

[11] D. D. Miller, J. N. Polky, and J. R. King, "A survey of Soviet developments in residue number theory applied to digital filtering," in *26th Midwest Symp. Circuits Systems*, August 1983.