Semi-Custom VLSI Design and Implementation of a New Efficient RNS Division Algorithm*

AHMAD A. HIASAT[1] AND HODA ABDEL-ATY-ZOHDY[2]

[1]*Elect. Eng. Dept., Princess Sumaya University, PO Box 1438, Amman 11941, Jordan*
[2]*Elect. & Sys. Eng. Dept., Oakland University, Rochester, MI 48309, USA*
Email: aahiasat@rss.gov.jo

In this paper we introduce a new algorithm for division in residue number system, which can be applied to *any* moduli set. Simulation results indicated that the algorithm is faster than the most competitive published work. To further improve this speed, we customize this algorithm to serve two specific moduli sets: $(2^k, 2^k-1, 2^{k-1}-1)$ and $(2^k+1, 2^k, 2^k-1)$. The customization results in eliminating memory devices (ROMs), thus increasing the speed of operation. A semi-custom VLSI design for this algorithm for the moduli $(2^k+1, 2^k, 2^k-1)$ has been implemented, fabricated and tested.

Received August 31, 1998; revised April 26, 1999

1. INTRODUCTION

The residue number system (RNS) has the advantage of carry-free arithmetic operations. Thus, using residue arithmetic would in principle increase the computer processing speed. In particular, addition, subtraction and multiplication can be performed on each residue digit concurrently and independently. However, there are drawbacks associated with RNS. These drawbacks include the difficulty of residue operations like division, sign and overflow detection.

Generally speaking, all reported algorithms on division in RNS [3, 4, 5, 6, 7, 8, 9, 10, 11] have the disadvantage of lengthy arithmetic operations, large execution time and complex hardware requirements. The complexity of these algorithms is due to mixed-radix conversion (MRC) and performing difficult residue operations.

The moduli sets $(2^k, 2^k-1, 2^{k-1}-1)$ and $(2^k+1, 2^k, 2^k-1)$ are particularly important in applications which require a high degree of precision [12, 13, 14]. Some of the recursive digital filters require a high degree of precision in their computations in order to accurately control the frequency characteristics and to eliminate the occurrence of instabilities. Moduli sets like $(2^k, 2^k-1, 2^{k-1}-1)$ and $(2^k+1, 2^k, 2^k-1)$ are among the very few systems that can deal with such critical situations [14]. The properties of these sets become more apparent in hardware considerations because most moduli are diminished or augumented powers of two. Residue addition for diminished powers of two is the carry-add type and multiplication by a power of two is equivalent to left rotation. Similarly, residue addition for augumented powers of two is the carry-subtract type. Therefore, they can play an increased role in implementing an RNS arithmetic unit for computers. The advances in VLSI technology have suggested novel approaches to the implementation of arithmetic units over finite rings. RNS supports the main VLSI design properties and features like simple connections, concurrency and modularity. Independence of residue digits eliminates complex interconnection patterns among different logic components. This independence leads to concurrency where an arithmetic operation can be carried on all residue digits concurrently. The similarity in processing architecture for each modulus offers functional and layout modularity.

In this paper, we present a new division algorithm. The main idea is based on selecting an approximate quotient that is guaranteed to produce a non-negative remainder, unless the division procedure is completed. The main features of this algorithm, as compared with others [3, 4, 5, 6, 7, 8, 9, 10, 11], are: no sign determination, overflow detection, scaling or MRC is needed. Moreover, no need for base extension, auxiliary or redundant moduli. The algorithm speed is not dependent on the number of moduli, but on dynamic range.

Neverthless, this new algorithm is still based upon converting the residue representation of the dividend and the divisor to a weighted code in order to derive some information regarding the position of the most-significant non-zero bit contained in a residue number. The algorithm is then customized to serve the above mentioned moduli sets. This customization reduces hardware and time requirements associated with converting the residue representation to a weighted code. The customized structure has been designed and implemented using VLSI design tools. The layout has,

*Part of this paper is based on [1]. Another part of this paper is based on [2].

then, been fabricated and tested to verify the integrity and functionality of the design.

In order to evaluate the performance of this new design, it has to be compared with other RNS division algorithms. Chren's algorithm [5] has a slightly better performance compared to another algorithm [4] in terms of the mean of residue operations needed for each division problem. Nevertheless, it requires MRC and residue scaling. If look-up tables are used, then MRC and scaling would require $(N-1)$ and $(2N-1)$ memory cycles respectively [12]. Chren's algorithm also requires a redundant modulus which represents another drawback. Gamberger [6] presented an algorithm which does not use MRC. The number of iterations for each division problem is proportional to the magnitude of the divisor. The mean of the number of iterations is, thus, very high. Moreover, the hardware implementation suggested by Gamberger is very complicated and expensive due to the utilization of auxiliary RNS. Hung and Parhami [11] introduced two RNS division algorithms based on the approximate-sign detection technique. The faster among the two [11] requires much more hardware than the other slower one. Hung and Parhami [11] indicated that intermediate to these algorithms are a number of choices that offer speed/cost tradeoffs. Although both the faster algorithm of Hung and Parhami [11] and the algorithm of Lu and Chiang [3] have the same time complexity, the latter one has a better hardware complexity.

The most competitive work, introduced by Lu and Chiang [3], does not use MRC, however, it utilizes the idea of the fractional representation of X/M to detect the parity of a residue number and hence to check if an overflow has taken place. Lu and Chiang's algorithm requires $2\log_2 Q$ steps, where Q is the quotient. Each step consists of several residue additions and subtractions, one residue multiplication, two memory access cycles and one multi-operand addition (in fact, in parts II and IV of Lu and Chiang's algorithm, more than one multi-operand addition might be needed). Realization II of the new algorithm requires $\log_2 Q$ steps where each step consists of one residue multiplication $(Q_i Y)$, one residue subtraction $(X - Q_i Y)$ that is performed in parallel with one residue addition $(Q = Q + Q_i)$, one multi-operand addition and two memory access cycles: one to get the fractional representations of residue digits, while the other is to obtain Q_i.

NOTE. Following the literature, the prevalent method of measuring execution time for residue arithmetic algorithms [4, 5, 12, 13], the mean of the basic residue arithmetic operations needed by each algorithm, is computed. The basic residue operations are: addition, subtraction and multiplication.

The following notational convention has been adopted for this paper:

- $\{m_1, m_2, \ldots, m_N\}$, moduli set of N pairwise relatively prime positive integers.
- $M = \prod_{i=1}^{N} m_i$, dynamic range.
- For any integer $X \in [0, M)$, residue representation of X is: $X \overset{RNS}{\rightarrow} (x_1, x_2, \ldots, x_N)$.
- $x_i = |X|_{m_i}$, i.e. $x_i = X \pmod{m_i}$.
- $\hat{m}_i = M/m_i$.
- $|1/\hat{m}_i|_{m_i}$, multiplicative inverse of $\hat{m}_i$ (i.e. $||1/\hat{m}_i|_{m_i} \cdot \hat{m}_i|_{m_i} = 1$).
- $\lceil . \rceil$, the ceiling value of (.); that is the next integer greater than or equal to (.).
- $\lfloor . \rfloor$, the floor value of (.); that is the preceding integer less or equal to (.).
- Define a function $h(I)$ such that:

$$h(I) = \begin{cases} 1 + \lfloor \log_2 I \rfloor, & \text{if } I \text{ is an integer} > 0 \\ 0, & \text{if } I = 0 \\ \lfloor \log_2 I \rfloor, & \text{if } 0 < I < 1. \end{cases}$$

2. DIVISION ALGORITHM

Assume that X, Y and Q are non-negative integers such that $Q = \lfloor X/Y \rfloor$, $Y \neq 0$, then the following steps introduce the basic idea for division in RNS:

1. Set quotient Q to zero; $Q = 0$.
2. Find the position of the most-significant non-zero bit in the divisor Y, say k, that is $k = h(Y)$.
3. Find the position of the most-significant non-zero bit in the dividend X, say j, that is $j = h(X)$.
4. If $j > k$, then:
   $Q' = Q + 2^{j-k-1}$, $X' = X - 2^{j-k-1} * Y$, $Q = Q'$, $X = X'$. Go to Step 3.
5. If $j = k$, then:
   $X' = X - Y$, $j' = h(|X'|_M)$, so if $j' < j$ then $Q = Q + 1$.
   Otherwise, Q is unchanged. In either case, end procedure.
6. If $j < k$, then Q is unchanged. End procedure.

This basic algorithm can be used effectively with RNS division arithmetic. An important feature of this algorithm is the selection of the quotient to be 2^{j-k-1}, hence the quantity $X - (2^{j-k-1} * Y)$ is guaranteed to be non-negative as long as $X > Y$. It should be emphasized that X and Y in the above algorithm are binary representations of non-negative integers and that the algorithm is still correct when the fractional representation is adopted. The proofs of both cases, integer and fractional representations, are introduced in the next two subsections

2.1. Proof of correctness of the algorithm: integer representation

Before proving the algorithm, the following lemma has to be introduced.

LEMMA 1. *For any residue integers $X, Y \in [0, M)$, for which $j = h(X)$, $k = h(Y)$ and $j = k \neq 0$ then $j > j'$ if $X \geq Y$ and $j' \geq j$ if $X < Y$, where $j' = h(|X - Y|_M)$.*

Proof. Since $j = k$, then X and Y can be expressed as:
$X = 2^{j-1} + a$, and $Y = 2^{j-1} + b$, where: $0 \leq a, b \leq 2^{j-1} - 1$.

For the case $X \geq Y$ (i.e. $a \geq b$): $|X - Y|_M = X - Y \geq 0$, then $X - Y = a - b$, but since $0 \leq a - b \leq 2^{j-1} - 1$, then $j' = h(X - Y) = h(a - b) < j$.
For the case $X < Y$ (i.e. $a < b$): $|X - Y|_M = M + X - Y$, $j' = h(M + X - Y) = h(M + a - b)$. The minimum value of j' happens when $a = 0$, $b = (2^{j-1} - 1)$ and $M = Y + 1$. Upon substituting these values: $j' \geq h(2^{j-1} + 1) = j$, or equivalently: $j' \geq j$. □

Based on the above lemma, the proof of the algorithm is as follows:

- For the case $j > k$, and since $Y < 2^{k+1}$ and $X \geq 2^j$ then $X/Y > 2^{j-k-1} = Q_i$ (i.e. Q_i is the ith partial quotient). Hence the estimate of the quotient in each iteration is guaranteed to produce a positive remainder. Assume there are v iterations which satisfy the condition $j > k$, then the total partial quotients resulting from this case are Q, where: $Q = \sum_{i=1}^{v} Q_i$.
- For the case $j = k$ (i.e. $(v + 1)$th iteration), two possibilities are expected:
  1. $X < Y$. This case is detected according to Lemma 1 by evaluating $j' = h(X - Y)$. Hence, if $j' \geq j$ then $Q_{v+1} = 0$. The procedure is then stopped.
  2. $X \geq Y$. This case is detected according to Lemma 1 by evaluating $j' = h(X - Y)$. Hence, if $j' < j$ then $Q_{v+1} = 1$. The procedure is then stopped.
- For the case $j < k$, it is obvious that $X < Y$, hence the procedure has to be stopped.

Therefore, the quotient would be: $Q = \lfloor X/Y \rfloor = \sum_{i=1}^{v} Q_i + Q_{v+1}$, where:

$$Q_{v+1} = \begin{cases} 1, & \text{if } j_{v+1} > j' \\ 0, & \text{otherwise} \end{cases}$$

$j_{v+1} = h(X)$, in the $(v + 1)$th iteration (i.e. when $j = k$).

2.2. Proof of correctness of the algorithm: fractional representation

LEMMA 2. *In RNS, for any fractional representations $X/M, Y/M$ where $X, Y \in [0, M)$, $j = h(X/M)$, $k = h(Y/M)$ and $j = k$ then $j' \neq -1$ if $X \geq Y$, and $j' = -1$ if $X < Y$, where $j' = h((X - Y)/M)$.*

Proof. Since X/M and Y/M are of the same order (i.e. $j = k$) that is:
$2^j \leq X/M < 2^{j+1}$ and similarly $2^j \leq Y/M < 2^{j+1}$.

For the case $X \geq Y$: $|X - Y|_M = X - Y \geq 0$, then:
$0 \leq (X - Y)/M < 2^j$. Hence, $j' = h((X - Y)/M) < j$, or equivalently; $j' < j$.

Since the highest value of j is -1, then: $j' \leq -2$.

Note that for the special case, $(X - Y)/M = 0$, then by definition $h(0) = 0$.

Consequently, if $X \geq Y$ then $j' \neq -1$.

For the case $X < Y$: $|X - Y|_M = M - (Y - X)$, then $j' = h((M + X - Y)/M)$, or $j' = h(1 - (Y - X)/M)$ but since $X < Y$, then: $0 < (Y - X)/M < 2^j$, or $j' \geq h(1 - 2^j)$.

Since $X, Y \leq M$, then the maximum value of j is -1. Therefore, $0 > j' \geq h(\frac{1}{2})$. Or: $j' = -1$.

The proof of the algorithm for the case when the dividend and divisor are fractional quantities uses Lemma 2 and follows the same approach given in the proof of Realization I. The proof leads to the result that the quotient Q can be expressed as:

$$Q = \left\lfloor \frac{X}{Y} \right\rfloor = \sum_{i=1}^{v} Q_i + Q_{v+1}$$

$$Q_{v+1} = \begin{cases} 1, & \text{if } j' \neq -1 \\ 0, & \text{otherwise.} \end{cases}$$ □

3. REALIZATION OF THE ALGORITHM IN RNS

3.1. Realization I

This realization is based on the integer representation outlined in the previous section. It is quite useful for small and medium dynamic ranges where all the bits of residue digits can be applied to a single RAM in order to evaluate $h(I)$. The proposed structure for Realization I, shown in Figure 1, can be described as follows: by applying Y to a RAM 1, $k = h(Y)$ is evaluated, where k is expressed in r bits, $r = \lceil \log_2(\log_2 M) \rceil$. Similarly, by applying X to RAM 1 $j = h(X)$ is also evaluated, where j is also expressed in r bits. The partial quotient Q_i is computed by applying j and k to RAM 2. A residue multiplier then multiplies the partial quotient with Y. The output of the multiplier, i.e. $Q_i Y$, is subtracted from X to produce a new remainder. The procedure is repeated and the residue adder accumulates partial quotients, until $j < k$.

RAM 1 accepts $\sum_1^N n_i$ bits, where $n_i = \lceil \log_2 m_i \rceil$ address lines (i.e. bits of residue representation of X or Y), thus it has a size of $(2^{\sum_1^N n_i} \times r)$ bits, $r = \lceil \log_2(\log_2 M) \rceil$. However, RAM 2 accepts the bits of j and k (a total of $2r$ bits), thus its size would be $(2^{2r} \times \sum_1^N n_i)$. In the case that $j - k$ is first evaluated before being applied to RAM 2, then RAM 2 would have the size $(2^{r+1} \times \sum_1^N n_i)$.

This realization requires $\log_2 Q$ iterations. Each iteration consists of two consecutive memory cycles followed by two consecutive residue operations. The realization is very attractive for many digital-signal processing applications which utilize small and medium dynamic ranges.

3.2. Realization II

This realization is based on the fractional representation outlined in the previous section. It is quite useful for large dynamic ranges where bits of residue digits cannot be applied to a single RAM in order to evaluate $h(I)$.

Van Vu [15] developed a conversion technique based on the CRT. This technique uses fractional representation of

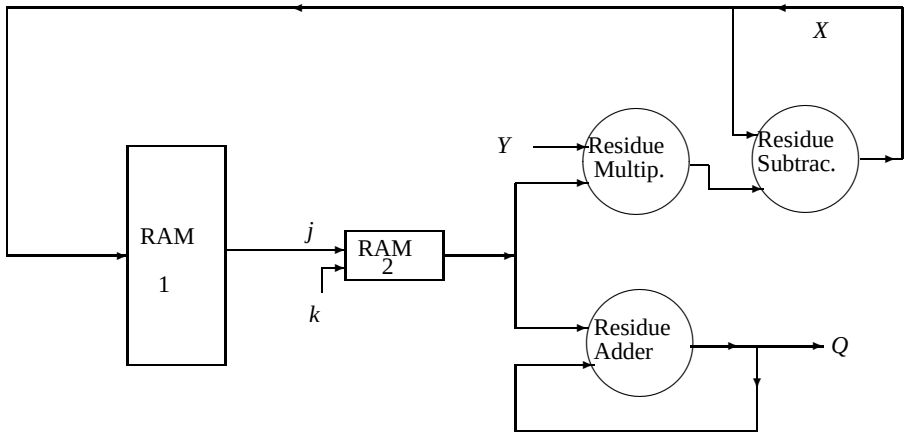

FIGURE 1. Proposed implementation of the algorithm (Realization I).

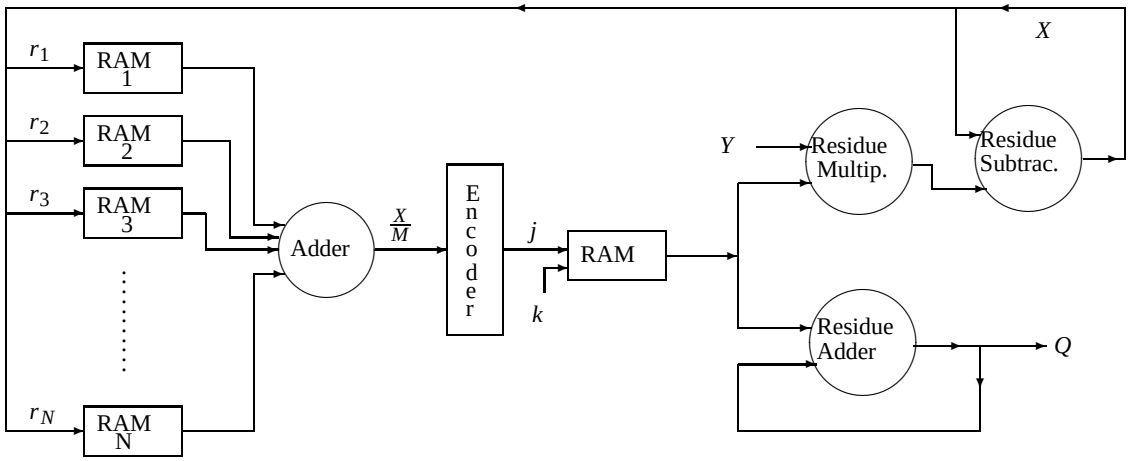

FIGURE 2. Proposed implementation of the algorithm (Realization II).

weighted numbers. This technique is given by

$$\frac{X}{M} = \sum_{i=1}^{N} \frac{1}{m_i} \left| \frac{x_i}{\hat{m}_i} \right|_{m_i} - p \qquad (1)$$

where p is a non-negative integer. Hence, the value of X/M can be obtained by evaluating the right-hand side of (1). This is basically done by letting each x_i address a table which stores

$$\frac{1}{m_i} \left| \frac{x_i}{\hat{m}_i} \right|_{m_i}.$$

The output of these N tables can be added using a multi-operand adder. Any integer overflow resulting from this adder is disregarded since it represents the integer part of the summation result. The fractional value stored in tables should be expressed using t bits where $t \geq \lceil \log_2 MN \rceil$ if M is odd and $t \geq \lceil \log_2 MN \rceil - 1$ otherwise [15].

The proposed structure for Realization I, shown in Figure 1, can be described as follows: apply the residue digits of the divisor Y to the fractional representation circuit to obtain Y/M. This requires a memory cycle followed by a multi-operand addition. Y/M is applied to a priority encoder to evaluate $k = h(Y/M)$. Next, and using the same approach, $j = h(X/M)$ is evaluated. The bits of j and k (a total of $2r$ bits), or their difference, are applied to a RAM to produce the partial remainder Q_i. This Q_i is applied to a residue multiplier to compute $Q_i Y$. The output of the multiplier ($Q_i Y$) is subtracted from X using a residue subtractor. The output of the residue subtractor is applied again to the fractional representation circuit as long as $j \geq k$.

4. EVALUATION

The complexity of the proposed residue-based divider is highly dependent on the complexity of individual components being used in the design (e.g. residue multiplier, adder, etc.). Many residue-based multipliers can be found in the literature [16, 17, 18, 19]. These are different in their structures, area and time complexities (i.e. gate number, silicon area, time delay, etc.). The same statement can be made about other components in this proposed divider [12].

TABLE 1. Simulation results.

Moduli Set	MORO		MOMA	
	Ours	Lu and Chiang's	Ours	Lu and Chiang's
M1	2.68	10.34	0	0
M2	2.67	10.36	3.041	5.496
M3	2.72	10	3.089	5.504

For example, the area and time complexities of the priority encoder presented in [20] are given by $O(n)$ and $O(\log n)$, respectively.

The proposed design in Figure 2 consists of different devices; namely RAMs, a multi-operand binary adder, a priority encoder followed by a RAM, a residue-based adder, subtractor and multiplier. For the first N RAMs, the size of the ith RAM is $(2^{n_i} \times n)$, where $n_i = \lceil \log_2 m_i \rceil$. The multi-operand adder accepts N operands of n bits each. The least significant (LS) n output bits of this adder constitute X/M. The encoder accepts these LS n bits of the adder and produces $h(I)$, expressed in r bits. The RAM following the encoder outputs the residue representation of the estimated quotient. This estimated quotient is, then, applied to different residue-based arithmetic components. Each of these components accepts N residue digits from each operand, where each residue digit is expressed in n_i bits.

To compare the performances of this algorithm and Lu and Chiang's algorithm, computer programs simulating both algorithms have been developed to calculate the mean of residue operations (MORO). The mean of multi-operand additions (MOMA) has been calculated and compared, where it applies. For simulation purposes, three moduli sets were selected to serve different dynamic ranges. These sets are: M1 $=$ $(7, 11, 13, 15)$, M2 $=$ $(11, 13, 15, 19, 23, 29, 31)$ and M3 $=$ $(29, 31, 43, 47, 53, 55, 59, 61, 63)$. Simulation results are listed in Table 1. For moduli set M1, the results are exact. All possible combinations of dividends and divisors were simulated. However, for M2 and M3, a sample of 200 million randomly generated numbers within the dynamic range defined by each moduli set was simulated. The simulation of the new algorithm is based on Figure 1 for M1, and on Figure 2 for M2 and M3. Simulation of Lu and Chiang's algorithm is based on the flowchart given in [3].

For the moduli set M1, Table 1 indicates that the new algorithm is four times faster than Lu and Chiang's algorithm. This conclusion applies to every moduli set where all the bits of residue digits can be applied simultaneously to a single RAM.

For other moduli sets like M2 and M3 which have very large dynamic ranges, the new algorithm is still four times faster regarding the number of basic residue operations. Moreover, the average number of multi-operand additions needed is almost half that needed by the other algorithm.

5. CUSTOMIZED DIVISION ALGORITHM

In Realization II, determining the position of the highest power of two contained in any residue number I, which we referred to as $h(I)$, is an important time-delay element in the operation of the proposed residue divider. In this section, we customize the same algorithm to serve two specific moduli sets: $(2^k, 2^k-1, 2^{k-1}-1)$ and $(2^k+1, 2^k, 2^k-1)$. This customization results in eliminating the need of ROMs and thus reducing the delay contributed by evaluating $h(I)$.

5.1. Evaluating $h(I)$ for the moduli set $(2^k, 2^k-1, 2^{k-1}-1)$

Define $m_1 = 2^k$, $m_2 = 2^k-1$, $m_3 = 2^{k-1}-1$. Then, $\hat{m}_1 = (2^k-1)(2^{k-1}-1)$, $\hat{m}_2 = 2^k(2^{k-1}-1)$, $\hat{m}_3 = 2^k(2^k-1)$. The residue representation of X is (r_1, r_2, r_3). The multiplicative inverses for $\hat{m}_1$, $\hat{m}_2$ and $\hat{m}_3$ are [21]: $2^{k-1}+1$, 2^k-3 and 2^{k-2}, respectively. Substituting the corresponding values of $\hat{m}_i$ and their multiplicative inverses in (1):

$$\frac{X}{M} = \frac{1}{2^k}|(2^{k-1}+1)r_1|_{2^k} + \frac{1}{2^k-1}|(2^k-3)r_2|_{2^k-1} + \frac{1}{2^{k-1}-1}|2^{k-2}r_3|_{2^{k-1}-1} - p. \quad (2)$$

Since $X < M$, then $X/M < 1$. Hence

$$\frac{X}{M} = \mathrm{FRAC}\left(\frac{1}{2^k}|(2^{k-1}+1)r_1|_{2^k} + \frac{1}{2^k-1}|(2^k-3)r_2|_{2^k-1} + \frac{1}{2^{k-1}-1}|2^{k-2}r_3|_{2^{k-1}-1}\right) \quad (3)$$

where FRAC(...) denotes the fractional part of the operand.

The circular shift property [13] states that modulo (2^p-1) multiplication of an integer by 2^n, where p and n are positive integers, is equivalent to n-bits circular left-shift (e.g. $|2^3(27)|_{31} \overset{\text{binary}}{\rightarrow} \overset{\text{3-bits}}{\overleftarrow{11011}} = 11110 \overset{\text{decimal}}{\rightarrow} 30$).

Therefore, to simplify the terms on the right-hand side (RHS) of (3), we proceed as follows:

- Evaluate $(1/2^k)|(2^{k-1}+1)r_1|_{2^k}$.
  Assuming that the binary form of r_1 is given by: $b_{1(k-1)}b_{1(k-2)}\ldots b_{11}b_{10}$, then,
  $|(2^{k-1}+1)r_1|_{2^k} = |2^{k-1}r_1 + r_1|_{2^k} \overset{\text{binary}}{\rightarrow}$
  $|b_{1(k-1)}b_{1(k-1)}\ldots b_{11}b_{10}\overbrace{00\ldots000}^{(k-1)\text{zeros}} + b_{1(k-1)}b_{1(k-2)}\ldots b_2b_1b_0|_{2^k}$.
  Recalling that for mod2^k, only the LS k bits are significant, then $|(2^{k-1}+1)r_1|_{2^k} = b_x b_{1(k-2)}\ldots b_{11}b_{10}$, where $b_x = (b_{10}$ XOR $b_{1(k-1)})$.
  Now, let R_1 represent the binary form of $(1/2^k)|(2^{k-1}+1)r_1|_{2^k}$, then R_1 is obtained by multiplying the binary form of $1/2^k$ by that of $|(2^{k-1}+1)r_1|_{2^k}$. That is,
  $$R_1 = 0.b_x b_{1(k-2)}\ldots b_{11}b_{10}. \quad (4)$$

- Evaluate $[1/(2^k-1)]|(2^k-3)r_2|_{2^k-1}$.
  Assuming that the binary form of r_2 is given by: $b_{2(k-1)}b_{2(k-2)}\ldots b_{21}b_{20}$, then $|(2^k-3)r_2|_{2^k-1} = |2^k r_2 - 3r_2|_{2^k-1}$. But since $|2^k r_2|_{2^k-1} = |r_2|_{2^k-1}$, then $|(2^k-3)r_2|_{2^k-1} = |-(2r_2)|_{2^k-1}$.
  Based on the circular left-shift property:
  $|-(2r_2)|_{2^k-1} \overset{\text{binary}}{\rightarrow}$
  $|-(b_{2(k-2)}b_{2(k-3)}\ldots b_{21}b_{20}b_{2(k-1)})|_{2^k-1}$.

  Or equivalently,
  $|-(b_{2(k-2)}b_{2(k-3)}\ldots b_{21}b_{20}b_{2(k-1)})|_{2^k-1} =$
  $|\overbrace{(11\ldots111)}^{k\text{-ones}(=2^k-1)} -$
  $(b_{2(k-2)}b_{2(k-3)}\ldots b_{21}b_{20}b_{2(k-1)})|_{2^k-1}$.
  Assuming $r_2 \neq 0$, then

  $$|(2^k-3)r_2|_{2^k-1} \overset{\text{binary}}{\rightarrow} \overline{b}_{2(k-2)}\overline{b}_{2(k-3)}\ldots\overline{b}_{21}\overline{b}_{20}\overline{b}_{2(k-1)}$$

  where $\overline{b}$ denotes the complement of the bit b.
  However, if $r_2 = 0$, then $|(2^k-3)r_2|_{2^k-1} = 0$. On the other hand, the term $1/(2^k-1)$ can be written as $2^{-k}/(1-2^{-k})$. Recall that any fraction in the form $q/(1-q)$, where $|q|<1$, can be expanded in a power series form as: $q/(1-q) = \sum_{i=1}^{i=\infty} q^i$. Therefore: $2^{-k}/(1-2^{-k}) = 2^{-k}+2^{-2k}+2^{-3k}+2^{-4k}+\ldots$. Based on error analysis introduced in [15], then the MS $(3k+1)$ bits are the only significant bits in our computations. Let R_2 represent the binary form of $[1/(2^k-1)]|(2^k-3)r_2|_{2^k-1}$, then R_2 is obtained by considering the MS $(3k+1)$ bits of multiplying $1/(2^k-1)$ by $|(2^k-3)r_2|_{2^k-1}$. That is

  $$R_2 = 0.\overbrace{\overline{b}_{2(k-2)}\overline{b}_{2(k-3)}\ldots\overline{b}_{21}\overline{b}_{20}\overline{b}_{2(k-1)}}^{k\text{bits}} * 0\overbrace{\overline{b}_{2(k-2)}\overline{b}_{2(k-3)}\ldots\overline{b}_{21}\overline{b}_{20}\overline{b}_{2(k-1)}}^{k\text{bits}} * \overbrace{\overline{b}_{2(k-2)}\overline{b}_{2(k-3)}\ldots\overline{b}_{21}\overline{b}_{20}\overline{b}_{2(k-1)}\overline{b}_{2(k-2)}}^{k\text{bits}} \quad (5)$$

  where $*$ implies that terms are concatenated.
- Evaluate $[1/(2^{k-1}-1)]|2^{k-2}r_3|_{2^{k-1}-1}$.
  Assuming that the binary form of r_3 is given in $(k-1)$ bits by: $b_{3(k-2)}b_{3(k-3)}\ldots b_{31}b_{30}$, then based on the circular left-shift property: $|2^{k-2}r_3|_{2^{k-1}-1} \overset{\text{binary}}{\rightarrow} b_{30}b_{3(k-2)}b_{3(k-3)}\ldots b_{31}$.
  On the other hand, the term $1/(2^{k-1}-1)$ can be written as $2^{-(k-1)}/(1-2^{-(k-1)})$. Thus, it can be expanded in a power series form as: $2^{-(k-1)}/(1-2^{-(k-1)}) = 2^{-(k-1)}+2^{-2(k-1)}+2^{-3(k-1)}+2^{-4(k-1)}+\ldots$. Now, let R_3 be the binary form of $[1/(2^{k-1}-1)]|2^{k-2}r_3|_{2^{k-1}-1}$, then R_3 is obtained by considering the MS $(3k+1)$ bits of multiplying $1/2^{k-1}-1$ by $|2^{k-2}r_3|_{2^{k-1}-1}$. That is

  $$R_3 = 0.\overbrace{\overbrace{b_{30}b_{3(k-2)}\ldots b_{32}b_{31}}^{(k-1)\text{bits}}\overbrace{b_{30}b_{3(k-2)}\ldots b_{32}b_{31}}^{(k-1)\text{bits}}}^{} * \overbrace{\overbrace{b_{30}b_{3(k-2)}\ldots b_{32}b_{31}}^{(k-1)\text{bits}}\overbrace{b_{30}b_{3(k-2)}b_{3(k-3)}b_{3(k-4)}}^{4\text{bits}}}^{}. \quad (6)$$

Therefore, (3) can be rewritten as

$$\frac{X}{M} = \text{FRAC}(R_1+R_2+R_3). \quad (7)$$

Thus $h(X/M)$ is nothing but the position of the MS non-zero bit of X/M.

EXAMPLE. Consider the moduli set {16, 15, 7}. To find $h(X/M)$ where $X = (8, 12, 4)$ (i.e. $X = 312$ and $M = 1680$), then:
$r_1 = 8 \overset{\text{binary}}{\rightarrow} 1000$, so $R_1 = 0.1000$
$r_2 = 12 \overset{\text{binary}}{\rightarrow} 1100$, so $R_2 = 0.0110\ 0110\ 0110\ 0$
$r_3 = 4 \overset{\text{binary}}{\rightarrow} 100$, so $R_3 = 0.010\ 010\ 010\ 010\ 0$.
Therefore, $X/M = \text{FRAC}(R_1+R_2+R_3) = 0.00\underline{1}011111000$
Since the underlined MS non-zero bit is in the third location, then $h(X/M) = -3$.

In order to implement (7), one three-operand binary adder is needed. A carry-save adder (CSA) followed by a carry-propagate adder (CPA) can realize the addition of the three operands.

5.2. Evaluating $h(I)$ for the moduli set $(2^k+1, 2^k, 2^k-1)$

The residue decoder introduced by Sweidan and Hiasat [22] has the advantages of reduced hardware requirements and extremely wide fixed-point dynamic ranges since its upper bound is not limited by a memory size. Moreover, it requires 'only' a total of four $2k$-bit binary adders, which makes it very attractive compared to other published decoders [23, 24]. In this paper, we propose a hardware layout that can decode residue digits of the moduli set $(2^k+1, 2^k, 2^k-1)$ into binary equivalent. The new layout is an improvement of that presented in [21]. In this new contribution, we are reducing the number of adders needed for the decoding operation from 'four' $2k$-bit binary adders into 'one' $2k$-bit three-operand binary adder.

It has been proved in [21] that

$$\lfloor X/2^k \rfloor = |A+B+C-r_1|_{2^{2k}-1} \quad (8)$$

where:

$$A = |(2^{2k-1}+2^{k-1})r_3|_{2^{2k}-1}$$
$$B = |(2^{2k}-2^k-1)r_2|_{2^{2k}-1}$$
$$C = |(2^{2k-1}+2^{k-1})r_1|_{2^{2k}-1}.$$

Assuming that r_1, r_2 and r_3 have the following binary format:

$$r_1 = b_{1k}b_{1(k-1)}\ldots b_{11}b_{10}$$
$$r_2 = b_{2(k-1)}b_{2(k-2)}\ldots b_{21}b_{20}$$
$$r_3 = b_{3(k-1)}b_{3(k-2)}\ldots b_{31}b_{30},$$

then using circular left-shift property, A, B and C can be

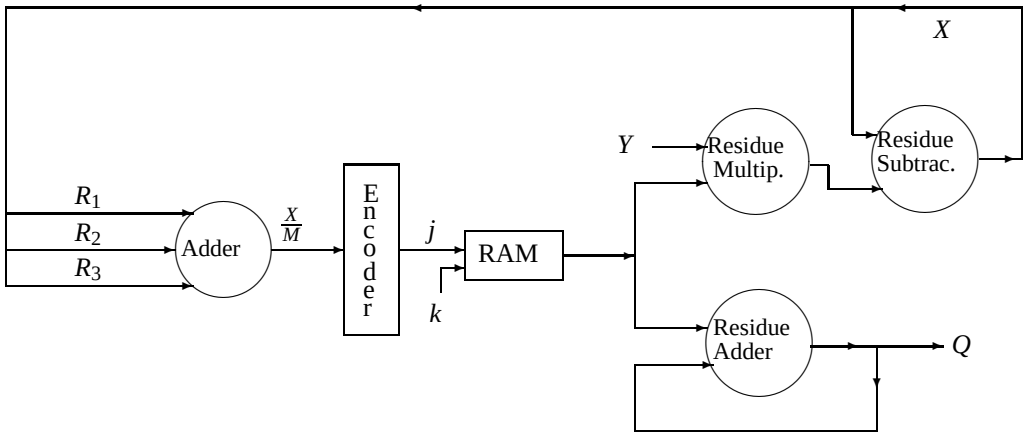

FIGURE 3. Proposed implementation of the division algorithm customized for moduli sets: $(2^k, 2^k-1, 2^{k-1}-1)$ and $(2^k+1, 2^k, 2^k-1)$.

expressed as [21]:

$$A = b_{30}b_{3(k-1)}\ldots b_{32}b_{31}b_{30}b_{3(k-1)}\ldots b_{32}b_{31}$$
$$B = \overline{b}_{2(k-1)}\overline{b}_{2(k-2)}\ldots\overline{b}_{21}\overline{b}_{20}\underbrace{11\ldots1}_{k\text{ ones}}$$
$$C = b_{1x}b_{1(k-1)}\ldots b_{12}b_{11}b_{1x}b_{1(k-1)}\ldots b_{12}b_{11}$$

where: $b_{1x} = b_{10}$ OR b_{1k}. By redefining $R'_1 = A$, $R'_2 = B - r_1$, and $R'_3 = C$, then (8) can be rewritten as

$$\lfloor X/2^k \rfloor = |R'_1 + R'_2 + R'_3|_{2^{2k}-1}. \tag{9}$$

- Case I: Since $R'_1 = A$, then the binary representation is the same:

$$R'_1 = b_{30}b_{3(k-1)}b_{3(k-2)}\ldots b_{32}b_{31} * b_{30}b_{3(k-1)}b_{3(k-2)}\ldots b_{32}b_{31}. \tag{10}$$

- Case II: Since $R'_2 = B - r_1$, then for the case $r_1 < 2^k$, and using the 2's complement notation, $R'_2 = B + (1\text{'s } \textit{complement of } r_1) + 1$. Noting that the LS k bits of B are all ones, then the LS k bits of the result of the subtraction are simply the 1's complement of r_1 and an overflow of 1 at the $(k+1)$th bit. Based on 2's complement, this overflow indicates that the result of subtraction is positive, hence it can be disregarded. However, when $|r_1|_{2^k} = r_2 = 0$, then $R'_2 = |2^{2k} - 1|_{2^{2k}-1} = 0$. Therefore, R'_2 can be expressed in binary format as

$$R'_2 = \begin{cases} 0, & \text{if } |r_1|_{2^k} = r_2 = 0 \\ \overline{b}_{2(k-1)}\overline{b}_{2(k-2)}\ldots\overline{b}_{21}\overline{b}_{20} * \overline{b}_{1(k-1)}\overline{b}_{1(k-2)}\ldots\overline{b}_{11}\overline{b}_{10}, & \text{otherwise.} \end{cases} \tag{11}$$

- Case III: $r_1 = 2^k$. In this case, the $(k+1)$th bit of r_1 is 1, thus the values R'_2 and B are the same because in the computation of R'_2 we used the LS $(k-1)$ bits of r_1, which are all zeros in this case. Therefore, the format of R'_2 is not changed. However, to take care of this non-zero $(k+1)$th bit of r_1, it has to be subtracted from R'_3. Therefore

$$R'_3 = \begin{cases} b_{1x}b_{1(k-1)}\ldots b_{12}b_{11}b_{1x}b_{1(k-1)}\ldots b_{12}b_{11}, & \text{if } r_1 \neq 2^k \\ \overline{b}_{1x}\overline{b}_{1(k-1)}\ldots\overline{b}_{12}\overline{b}_{11}b_{1x}b_{1(k-1)}\ldots b_{12}b_{11}, & \text{if } r_1 = 2^k. \end{cases} \tag{12}$$

Equation (9) is simply accomplished by adding R'_1, R'_2 and R'_3. The output should then be incremented by any output-carry. Nevertheless, a carry resulting from this adder can be neglected as long as the output does not have the value $(2^n - 1)$, where $0 \leq n \leq 2k$. This can be justified by the fact that $h(I) = h(I+1)$ if $I \neq (2^n - 1)$. However, if $I = 2^n - 1$, then the output carry would be significant and the priority encoder proposed in implementing the residue divider can take care of this special case.

A single carry-save adder can add these three operands. Few logic gates are also needed to detect the $(k+1)$th bit of r_1 and to select the proper format of R'_3.

Using the formula $X = \lfloor X/2^k \rfloor 2^k + r_2$, then the value of X can be obtained by concatenation of the k bits of r_2 to the $2k$ output bits of the three-operand adder. These $3k$ bits and the carry are applied to the priority encoder.

EXAMPLE. Consider the moduli set {17, 16, 15}. To find $h(X)$ where $X = (11, 11, 2)$ (i.e. $X = 827$), then:

$$r_1 = 11 \overset{\text{binary}}{\rightarrow} 01011, \text{ so } R'_3 = 1101\ 1101$$
$$r_2 = 11 \overset{\text{binary}}{\rightarrow} 1011, \text{ so } R'_2 = 0100\ 0100$$
$$r_3 = 2 \overset{\text{binary}}{\rightarrow} 0010, \text{ so } R'_1 = 0001\ 0001.$$

Therefore, adder output = $(R'_1 + R'_2 + R'_3) = 0011\ 0010$, where the overflow has been neglected. Since $X = \lfloor X/2^k \rfloor 2^k + r_2$, then concatenation of bits of r_2 to that of the adder yields: $h(X) = h(0011\ 0010\ 1011) = 10$ (i.e. 10th position).

Figure 3 shows the new proposed hardware realization of an RNS divider for the moduli sets $(2^k, 2^k - 1, 2^{k-1} - 1)$

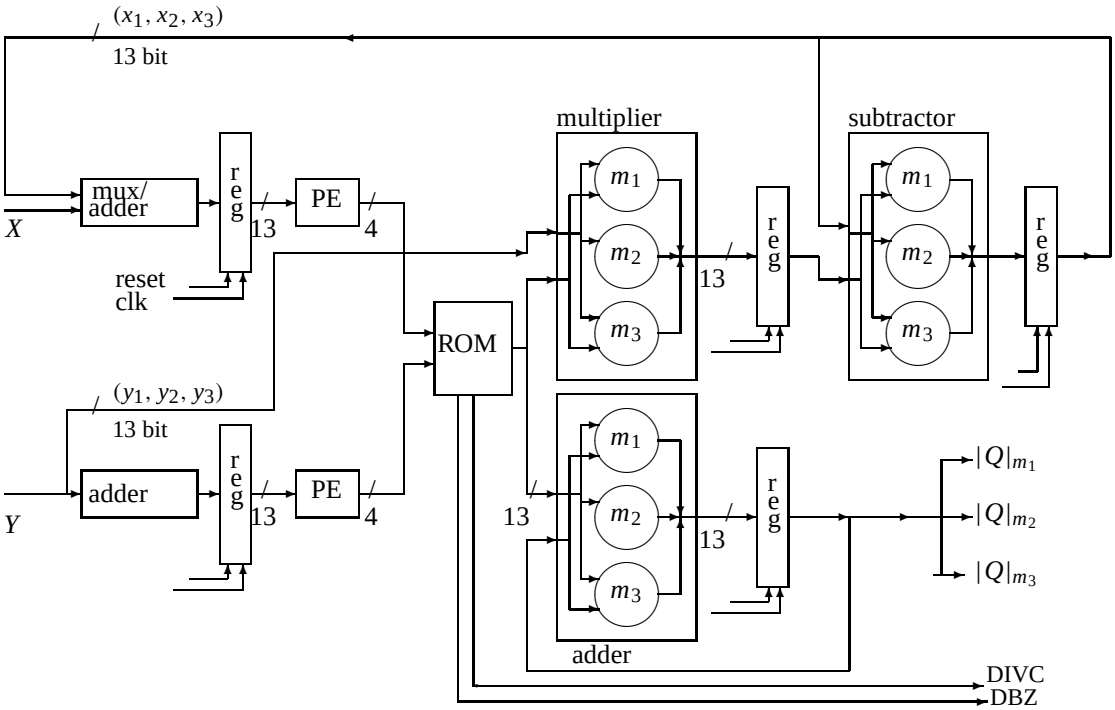

FIGURE 4. Block diagram of the implemented divider.

and $(2^k+1, 2^k, 2^k-1)$. The operation of this divider is self-explanatory. The propagation delay in Figure 3, as compared with that in Figure 2, has been reduced by a memory access cycle per iteration. Recalling that the memory access cycle is very significant compared with the delay of other components and that division is an iterative procedure, then this reduction will, eventually, be increasingly significant as the number of iterations per division problem is increased. This implies that the new proposed realization is much faster for these particular moduli sets. Moreover, the reduction in hardware requirements is another substantial improvement.

6. VLSI IMPLEMENTATION OF A RESIDUE-BASED ARITHMETIC DIVIDER

A pipelined design for a residue-based arithmetic divider for the moduli set $(2^k+1, 2^k, 2^k-1)$ has been implemented, fabricated and tested.

The detailed design of the implemented circuit is shown in Figure 4. Data path sizes are also shown. The implementation was accomplished using Octtools-5.2 with a standard cell MSU2.3 library. For prototype purposes, k was selected to be four. Thus, the total number of input pins is 13 for each operand. The clock, an X/Y selector and reset are another three inputs. Similarly, the output quotient is expressed in 13 bits. Division-completed (DIVC) is a one-bit output that goes high to validate the output quotient and sets the flag that the division process is completed. Division-by-zero (DBZ) is another output bit which sets a flag if the divisor is zero. The design has an integrated circuit area of (1.792×1.675) mm^2. The tiny padframe $(40PC22 \times 22)$ was used to accommodate this design. Test results showed that the design can run at a clock speed of 15 MHz. The number of clock cycles required for each division problem depends on both the dividend and the divisor. However, the average number over different division problems is eight clock cycles.

7. CONCLUSIONS

This paper has presented a new general division algorithm for RNS, which is faster than other previously proposed algorithms. The algorithms were then customized to serve two specific moduli sets: $(2^k, 2^k-1, 2^{k-1}-1)$ and $(2^k+1, 2^k, 2^k-1)$. An RNS divider would then require a binary adder, a priority encoder, a ROM, a residue adder, a residue subtractor and a residue multiplier only. These reduced hardware requirements and processing time qualify the new realization to be very practical for many computing applications and therefore enable RNS to play an increased role in designing arithmetic logic units for general purpose computers. The proposed customized hardware has been implemented on silicon and test results have been presented.

REFERENCES

[1] Hiasat, A. and Abdel-Aty-Zohdy, H. (1997) Design and implementation of an RNS division algorithm. *Proc. 13th Symp. Computer Arithmetic (Asilomar, CA)*, pp. 240–249.

[2] Hiasat, A. and Abdel-Aty-Zohdy, H. (1995) High-speed division algorithm for residue number system. *Proc. 1995 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, pp. 1996–1999.

[3] Lu, M. and Chiang, J. (1992) A novel division algorithm for residue number system. *IEEE Trans. Comput.*, **41**, 1026–1032.

[4] Banerji, D., Cheung, T. and Ganesan, V. (1981) A high-speed division method in residue arithmetic. *Proc. 5th IEEE Symp. Comput. Arithmetic*, pp. 158–164.

[5] Chren Jr., W. (1990) A new residue number division algorithm. *Computer Math. Appl.*, **19**, 13–29.

[6] Gamberger, D. (1991) New approach to integer division in residue number system. *Proc. 10th Symp. Comput. Arith.*, pp. 84–91.

[7] Kier, Y., Cheney, P. and Tannenbaum, M. (1962) Division and overflow detection in residue number systems. *IRE Trans. Electron. Comput.*, **11**, 501–507.

[8] Lin, L., Leiss, E. and Mcinnis, B. (1984) Division and sign detection algorithm for residue number systems. *Comput. Math. Appl.*, **10**, 331–342.

[9] Kinoshita, E., Kosako, H. and Kojima, Y. (1973) General division in the symmetric residue number system. *IEEE Trans. Computers*, **22**, 134–142.

[10] Hitz, M. and Kaltofen, E. (1995) Integer division in residue number system. *IEEE Trans. Computers*, **44**, 240–248.

[11] Hung, C. and Parhami, B. (1994) An approximate sign detection method for residue numbers and its applications to RNS division. *Computer Math. Appl.*, **27**, 23–35.

[12] Soderstrand, M., Jenkins, W., Jullien, G., Taylor, F. (eds) (1986) *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing.* IEEE Press, New York.

[13] Szabo, N. and Tanaka, R. (1967) *Residue Arithmetic and Its Applications to Computer Technology*. McGraw Hill, New York.

[14] Jenkins, W. (1979) Recent advances in residue number techniques for recursive digital filtering. *IEEE Trans. Acoust. Speech and Signal Processing*, **27**, 19–31.

[15] Van Vu, T. (1985) Efficient implementations of chinese remainder theorem for sign detection and residue decoding. *IEEE Trans Comput.*, **34**, 646–651.

[16] Hiasat, A. (1996) Semi-custom VLSI design for RNS multipliers using combinational logic approach. *ICECS'96*, **2**, 935–938.

[17] Radhakrishnan, D. and Yuan, Y. (192) Novel approaches to the design of VLSI RNS multipliers. *IEEE Trans. Circ. Sys-II: Analog and Digital Signal Processing*, **39**, 52–57.

[18] Alia, G. and Martinelli, E. (1991) A VLSI modulo m multiplier. *IEEE Trans. Comp.*, **40**, 873–878.

[19] Elleithy, K. and Bayoumi, M. (1995) A systolic architecture for modulo multiplication. *IEEE Trans. Circ. Sys-II: Analog and Digital Signal Processing*, **42**, 725–729.

[20] Wada, K., Hagihara, K. and Tokura, N. (1984) Area-time optimal fast implementations of several functions in a VLSI model. *IEEE Trans. Computers*, **33**, 435–440.

[21] Hiasat, A. and Zohdy, H. (1998) Residue to binary converter for the moduli $(2^k, 2^k - 1, 2^{k-1} - 1)$. *IEEE Trans. Circuits and Systems—Part II*, **45**, 204–209.

[22] Sweidan, A. and Hiasat, A. (1988) New efficient memoryless, residue to binary converter. *IEEE Trans. Circuits and Systems*, **35**, 1441–1444.

[23] Bernardson, B. (1985) Fast memoryless, over 64 bits, residue to decimal converter. *IEEE Trans. Circuits and Systems*, **32**, 298–300.

[24] Ibrahim, K. and Saloum, S. (1988) An efficient residue to binary converter design. *IEEE Trans. Circuits and Systems*, **35**, 1156–1158.