Improved Memoryless RNS Forward Converter Based on the Periodicity of Residues

A. B. Premkumar, *Senior Member, IEEE*, E. L. Ang, and Edmund M.-K. Lai, *Senior Member, IEEE*

***Abstract*—The residue number system (RNS) is suitable for DSP architectures because of its ability to perform fast carry-free arithmetic. However, this advantage is over-shadowed by the complexity involved in the conversion of numbers between binary and RNS representations. Although the reverse conversion (RNS to binary) is more complex, the forward transformation is not simple either. Most forward converters make use of look-up tables (memory). Recently, a memoryless forward converter architecture for arbitrary moduli sets was proposed by Premkumar in 2002. In this paper, we present an extension to that architecture which results in 44% less hardware for parallel conversion and achieves 43% improvement in speed for serial conversions. It makes use of the periodicity properties of residues obtained using modular exponentiation.**

***Index Terms*—Forward and reverse converters, periodicity property, processing elements, residue number system (RNS).**

I. Introduction

ARITHMETIC operations based on residue number systems (RNS) can be carried out without intermediate carry digits. However, intermodular operation and conversion between number systems are awkward and have prevented the wide-spread use of RNS. Converting a number from a binary representation to its RNS equivalent is known as forward conversion while the inverse operation is called reverse conversion. Even though reverse conversion is generally more complex, forward conversion for arbitrary moduli sets is not simpler. For special moduli sets of the 2^n type, forward converters require only modular adders and therefore can be easily implemented. However, forward conversion for arbitrary moduli sets is memory intensive. There are three main approaches for forward conversion. The first approach involves precomputing all possible values that the conversion requires and storing these values in memory [2], [3]. The second approach involves using efficient arithmetic units called processing elements (PE) along with some memory. In both cases, the memory size requirement increases as the dynamic range increases. The third approach is memoryless in that it involves only combinatorial logic in the design. A framework for memoryless forward conversion has recently been introduced [1].

In this paper, we present an improvement to the memoryless architecture in [1]. The improved design makes use of the cyclic properties of residues to reduce hardware requirement. This paper is organized as follows. Residue numbers and some early forward converters are introduced in Section II. In Section III, the converter proposed in [1] is briefly presented. Periodicity properties of residues are discussed in Section IV. An improved converter which combines the periodicity property with the converter proposed in [1] is then formulated. The advantages of the new converter are discussed in Section VI and comparisons between converters are made.

Manuscript received February 17, 2004; revised April 17, 2005. This paper was recommended by Associate Editor S. Rosing.

The authors are with the School of Computer Engineering, Nanyang Technological University, Singapore.

Digital Object Identifier 10.1109/TCSII.2005.857090

II. RNS and ROM-Based Forward Conversion

Any n-bit nonnegative integer X in the range $0 \leq X \leq 2^n - 1$ can be represented in the weighted binary system as $X = \sum_{j=0}^{n-1} b_j 2^j$ where $b \,\epsilon \{0,1\}$. In RNS, X is represented by a set of k residues as $X = \{r_0, r_{1,}, \ldots r_{k-1}\}$, where $r_i = X \bmod m_i$. In this system, the set $\{m_i, 0 \leq i < k\}$ constitute the moduli set [4]. The range of the RNS representation is given by $R = \prod_{i=0}^{k-1} m_i - 1$.

Converting a number from its binary representation to an RNS representation is performed by the forward converter. We shall briefly describe some earlier converters as well as a more recent one [1] on which the present work is based.

In Alia and Martinelli's method [2], the residue corresponding to the jth bit of an n-bit binary number with respect to m_i is determined and stored in a register. Two such registers storing the residues of adjacent bits are combined in a PE. For a given n-bit number, depending on the value of the bit b_j, either the register content or a 0 is output. The outputs of two adjacent PEs are combined using a modular m_i adder. This first level of computation requires $(n/2)$ modular adders. The outputs from several adders are combined in a second level of adders and this process is continued until the final residue is obtained. A modification to this method was proposed by them [3] in which the n-bit word is partitioned into (n/q) smaller words. The residues corresponding to each group of q bits are stored in a ROM which are then accessed and added by modular adders. Capocelli and Giancarlo [5] suggested storing the residue corresponding to the first bit in a group of bits, doubling this residue mod m_i and evaluating the residue of the next power of 2 in that group. This process can be pipelined and the residues corresponding to only one group of bits need to be stored. Hence, this results in more efficient use of ROM.

Piestrak, in [6] and subsequently in [7], [8] proposed residue generators based on the periodic properties of residues. His residue converters are based on half and full periods. They make use of two different periods, namely, the period $P(m_i)$ of odd moduli and the half period $HP(m_i)$ of odd moduli. $P(m_i)$ exists for all moduli while $HP(m_i)$ only exists for some odd moduli. Both these periods are calculated using recursion and a table of periods and half periods for various moduli are generated. Using properties associated with periodicity and half periodicity, Piestrak proposed four architectures that are suitable for n input generators modulo m_i. Ananda Mohan [9] proposed partitioning the given n-bit number based on the cyclic property of $2^j \ : \mathrm{mod} m_i$. Using the fact that $2^j, 2^{j+l_0}, 2^{j+2l_0}$ have the same residues due to the periodicity (l_0), fields of l_0 bits are added. The width of the result is confined to l_0 bits by adding the carry bit resulting from the addition to the result. The residue is then determined using techniques given in [2], [3]. This results in similar PEs being used in the transformation algorithm. Ananda Mohan later used these principles to propose residue converters using ROM. Although minor architectural differences exist, the converters discussed so far invariably use ROM. In the next section, a more recent work on forward converters that does not require the use of memory is considered.

III. Forward Conversion Based on Modular Exponentiation

In the context of the present work, modular exponentiation of n refers to $(2^n \bmod m)$ for some m. Modular exponentiation synthesizes the residue of *power of 2 moduli* using only combinatorial logic circuits. Hence, it does not need any memory or PE in residue computations. For the sake of completeness, we shall briefly introduce modular exponentiation by means of an example. A complete treatment of modular exponentiation can be found in [1].

Example 1: Determine the residue of $2^{s_3s_2s_1s_0} \bmod 13$, where $s_i \in \{0,1\}$

$$\begin{aligned} |2^{s_3s_2s_1s_0}|_{13} &= |2^{8s_3+4s_2+2s_1+s_0}|_{13} \\ &= |256^{s_3}16^{s_2}4^{s_1}2^{s_0}|_{13} \\ &= |(255s_3+1)(15s_2+1)\times 4^{s_1}2^{s_0}|_{13} \\ &= |(8s_3+1)(2s_2+1)\times 4^{s_1}2^{s_0}|_{13} \\ &= ||3s_3s_2+2s_2+8s_3+1|_{13}\times 4^{s_1}2^{s_0}|_{13}. \quad (1) \end{aligned}$$

In the above, the s_1 and s_0 bits are assigned to multiplex the functions synthesized using the s_3 and s_2 bits. In synthesizing the functions, four different cases have to be considered.

1) Case 1: If $s_1 = s_0 = 0$, (1) reduces to 1, 3, 9, and 1 for $s_3 = s_2 = 0; s_3 = 0, s_2 = 1; s_3 = 1 s_2 = 0$; and for $s_3 = s_2 = 1$; respectively. A function g_0 can be synthesized for different combinations of s_3 and s_2 as shown in

$$g_0 = 8s_3\bar{s_2} + 0 + 2\bar{s_3}s_2 + 1. \quad (2)$$

When the function g_0 is evaluated for different combinations of s_3 and s_2 with $s_1 = s_0 = 0$, we have $\{1,3,9,1\}$.

2) Case 2: If $s_1 = 0$ and $\dot{s}_0 = 1$, (1) yields $\{2, 6, |18|_{13}, 2\}$ for different combinations of s_3 and s_2. A second function g_1 can be obtained as follows:

$$g_1 = 0 + 4(s_2 \oplus s_3) + 2(\bar{s_3} \quad \text{or} \quad s_2) + s_3\bar{s_2}. \quad (3)$$

3) Case 3: Similarly, function g_2 can be obtained for $s_1 = 1$ and $s_0 = 0$

$$g_2 = 8(s_2 \oplus s_3) + 4(s_2 \quad \text{or} \quad \bar{s_3}) + 2s_3\bar{s_2} + 0. \quad (4)$$

4) Case 4: Function g_3 is obtained when $s_2 = s_0 = 1$

$$g_3 = 8(\bar{s_3} \text{ or } s_2) + 4s_3\bar{s_2} + 2(s_3 \oplus s_2) + s_3 \oplus s_2. \quad (5)$$

In all of the above functions, $\oplus$ denotes the bitwise exclusive-or logical operation. These functions use the positional weights of the binary number system in arriving at the values for different s_3 and s_2. Hence, they can be written as follows:

$$g_0 = \{s_3\bar{s_2}, 0, \bar{s_3}s_2, 1\} \quad (6)$$
$$g_1 = \{0, (s_2 \oplus s_3), (\bar{s_3} \quad \text{or} \quad s_2), s_3\bar{s_2})\} \quad (7)$$
$$g_2 = \{s_2 \oplus s_3), (s_2 \quad \text{or} \quad \bar{s_3}), s_3\bar{s_2}, 0\} \quad (8)$$
$$g_3 = \{\bar{s_3} \quad \text{or} \quad s_2), s_3\bar{s_2}, (s_3 \oplus s_2), (s_3 \oplus s_2)\}. \quad (9)$$

Forward conversion from binary to residues can be accomplished by determining the residue of each nonzero $2^{s_3s_2s_1s_0}$ term of the given binary number.

The conversion can either be performed on each nonzero bit sequentially or on all bits simultaneously in parallel. Trade-offs can be made between hardware cost and delay by using a combination of serial and parallel methods. Hardware complexity for the parallel implementation mainly arises from the mutliplexers since the gates used to generate the MIN terms in obtaining g_i can be shared among different exponents within the same modulus as well as among different moduli.

IV. Periodicity Properties of Residues

An interesting property of determining the residue of $2^n \bmod m$ is the cyclic nature of its residues. Some observations are made about the periodicity properties of these residues.

1) For certain odd moduli m_i, there exists a period given by $m_i - 1$ after which the residues repeat. Consider $m_1 = 5$. The residues of $2^n \bmod 5$ for different n are $1, 2, 4, 3, 1, 2, 4, 3, \ldots$ The periodicity of the residues is $5 - 1 = 4$. Whenever the period is $m_i - 1$, it is referred to as basic period.
2) There exists for certain other odd moduli m_j, a period that is shorter than the basic period after which the residues repeat.
3) In the case of even moduli other than those that belong 2^n family, a short period exists after an initial subset of residues [10], [11]. These three properties are illustrated in Example 2.

Example 2: Consider $m_2 = 9$. The residues of $2^n \bmod 5$ for different n are given by the set $\{1, 2, 4, 8, 7, 5, 1, 2, \ldots\}$. The residues repeat after $n = 5$. In this case, there exists a period that is shorter than the basic period, namely, 6. Now, consider $m_3 = 10$. The corresponding residue set in this case is $\{1, 2, 4, 8, 6, 2, 4, 8, 6, \ldots\}$. It is observed that there is a short period of 4 after the first residue, 1.

V. Forward Conversion Based on Periodicity Properties

In this section, we propose an improved forward converter using modular exponentiation and periodicity property of the residues discussed in the previous section. The proposed converter differs from other converters in that it does not use any PEs but is based on the converter proposed in [1].

In the case of odd moduli, the residues have either a basic period or a short period. The given binary number is partitioned based on either of these periods. Without loss of generality, either the basic or the short period can be assumed to be p. Hence, $2^n, 2^{n+p}, 2^{n+2p}, \ldots$ will all have the same residues. The partitioned fields are added using p-bit adders. The length of the result is confined to p bits by adding the carry bit back to the result. This is illustrated by the following example.

Example 3: Consider the following 32-bit number:

00 011 001 00 011 010 11 000 110 00 111 100.

The decimal equivalent of this number is 421 185 084 and its residue with respect to modulus 23 is 22. From [9, Table II], the periodicity of the residues is 11. Therefore, the number is partitioned into three 11-bit fields and added as shown below. A zero has been appended at the MSB position to make the number 33 bits to facilitate easy partitioning into 11-bit fields.

$$\begin{array}{cccccccccccc}
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
\hline
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0.
\end{array}$$

In the above, the top row corresponds to the most significant 11-bit field of the given binary number while the third row corresponds to the least significant 11-bit field. The last row is the result obtained by adding the three rows. In the addition, the carry of 1 is added to the LSB of the result so that the length of the final result is 11 bits. Forward conversion is now performed on the result using modular exponentiation on only 11 bits as opposed to all 32 bits as proposed in [1]. The residue computation of $2^n \bmod m$ can be applied sequentially or in parallel. This is discussed in the next section along with a comparison between [1], [9] and the proposed converter.

It should be noted that even moduli frequently occur in the use of residue number applications. One such moduli set is $\{2m-1, 2m, 2m+1\}$. Hence, there is a need for forward conversion for even moduli. Residue computation in this case can be performed using the following theorem.

Theorem 1: Let m be an even integer that is not a power of 2. It can be expressed as $m = 2^p \times y$ where p is a positive integer and y is an odd number. The residue of an integer X modulo m can be expressed as

$$|X|_m = \left|\left\lfloor \frac{X}{2^p} \right\rfloor\right|_y \times 2^p + |X|_{2^p}. \quad (10)$$

Proof: $X = QP + R$ where Q is the quotient obtained by dividing X by P and R is the residue. Let P be even and composite not belonging to 2^n set. Let $P = y \times 2^p$. Then

$$R = X \bmod P. \quad (11)$$

X can also be written as

$$X = Q_1 \times 2^p + R_1 \quad (12)$$

where Q_1 is the quotient and R_1 is the residue obtained dividing X by 2^p

$$0 \le R_1 < 2^p$$

In this case

$$R_1 = |X|_{2^p}. \quad (13)$$

However, Q_1 can be written as

$$Q_1 = Q_2 \times y + R_2. \quad (14)$$

Q_2 is the quotient and R_2 is the remainder obtained by dividing Q_1 by y

$$0 \le R_2 < y.$$

But $Q_1 = \lfloor (X)/(2^p) \rfloor$ from (12). Since R_2 is the residue of Q_1

$$R_2 = |Q_1|_y = \left|\left\lfloor \frac{X}{2^p} \right\rfloor\right|_y.$$

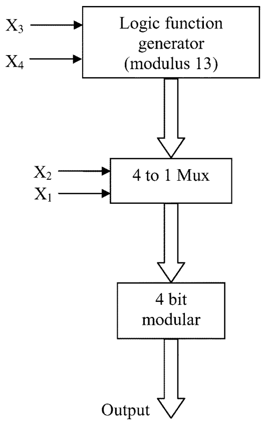

Fig. 1. Serial implementation of binary forms for $||8448||_{13}$.

We know that

$$X = QP + R = Q_1 \times 2^p + R_1.$$

From (13) and (14)

$$\begin{aligned} QP + R &= (Q_2 \times y + R_2) \times 2^p + |X|_{2^p} \\ &= Q_2 \times y \times 2^p + R_2 \times 2^p + |X|_{2^p} \\ &= Q_2 \times P + \left|\left\lfloor \frac{X}{2^p} \right\rfloor\right|_y \times 2^p + |X|_{2^p}. \end{aligned} \quad (15)$$

Comparing $X = QP + R$ with (15), we have

$$R = \left|\lfloor \frac{X}{2^p} \rfloor\right|_y \times 2^p + |X|_{2^p}. \quad (16)$$

■

Note that $|X|_{2^p}$ can be obtained by simply considering the least significant p bits of X. The first term in (10) is the $n-p$ most significant bits of the X, where X is an n-bit number. Evaluation of $|(X)/(2^p)|_y$ follows the procedure for odd modulus since y is odd. This is illustrated by the following example.

Example 4: To determine the residue of the same 32-bit number with respect to 24, the modulus is written as a composite number as $24 = 2^3 \times 3$. The residue $|X|_{2^3}$ is simply the three LSB and is therefore, 4. The remaining 29 bits are partitioned into 2-bit fields since the periodicity of 3 is 2 [9, Table II]. The two-bit fields are then added and the result is confined to 2 bits by adding the carry back to the result. The modular exponentiation is then performed and the residue is found to be 1. Using (10), the final residue works out to be 12.

VI. Comparative Analysis

Most digital signal processing (DSP) algorithms can be conveniently executed using a wordlength of 32 bits. A suitable moduli set would be $\{3, 5, 7, 11, 13, 17, 19, 23, 31\}$. Architectures using modular arithmetic need to implement circuits that generate functions such as those given by (6) to (9). Note that there are several terms common to g_i functions in these equations. Consequently, the circuit implementing these functions can be derived from commonly occurring terms. The common terms can even be shared among different moduli in the given set. In the implementation, the number of multiplexers required depends on whether serial or parallel technique is employed. The number of gates required for both serial and parallel techniques for our proposed converter based on this moduli set is estimated and compared

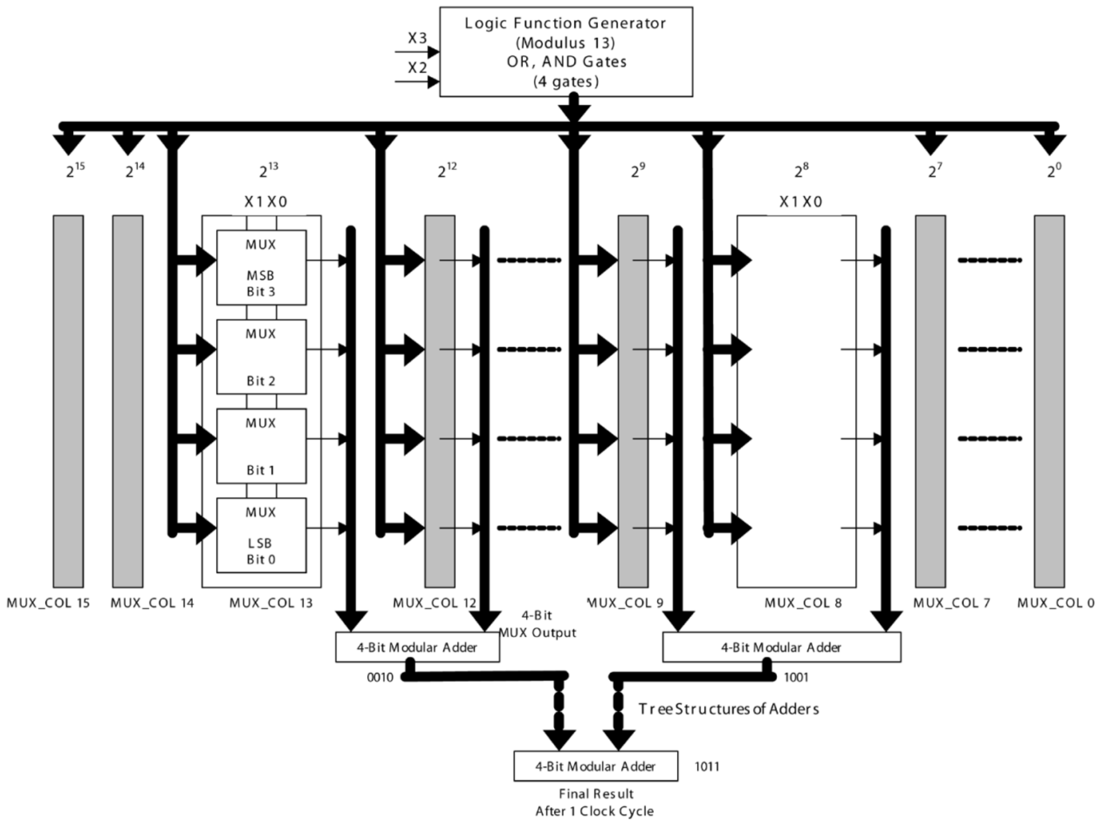

Fig. 2. Parallel implementation of binary forms for $||8448||_{13}$.

with that proposed in [1] and [9]. Figs. 1 and 2 show examples of the serial and parallel implementations of the proposed method. These figures are specific for Example 1. Reference [7, Table IV] discusses the number of gates that are required and delay issues with regard to specific moduli sets such as $2^n - 1$ and $2^{n-1} + 1$ that do not use memory. The other entries in [7, Table IV] discuss comparison for arbitrary moduli sets. However, these converters use ROM and hence are not considered for comparison here. Since the proposed forward converter is for arbitrary moduli sets that do not use memory, the proposed converter complexity has been compared with that in [1], [9]. Further, hardware and speed are two parameters that are used in the comparison and both these parameters are for modulus 19 only since this modulus has the longest period, namely, 18.

A. Serial Method

For comparison with [1], a worst case scenario when all bits in the given number are nonzero is considered. In Tables I and II, the second column explains how the conversion is accomplished. The exponent 5 in column 2 indicates that there are 32 bits in the binary number. Out of 5 bits, three MSB bits are used to generate g_i functions. The two LSB bits are used in multiplexing these functions.

These are clearly shown in the case of a 4-bit exponent in Example 1. Since each term in g_i is accessed serially and residue computation for each bit is performed taking one bit at a time, a single multiplexer is sufficient. Hence, for a 32-bit number, the residue computation is repeated a maximum of 32 times. Each computation requires 5 clock cycles since there are 5 bits in each residue and each bit in the residue is output sequentially. A total of 132 nand gates is required for generating the g_i functions. All 32 5-bit residues are added using a single modular adder to determine final residue.

The proposed converter requires the given decimal number to be partitioned into fields based on the periodicity and then added. In the chosen set of moduli, a maximum period of 18 occurs for modulus 19 [9]. Residue computation is performed 18 times and the 5-bit residues are added using a single modulo 19 adder. Each of the modulo 19 adder uses 5 Full Adders (FA), 5 D Flip Flops (FF) and two numbers of five 2-to-1 multiplexers and 5 inverters. The serial method uses one 4-to-1 multiplexer which is built using three 2-to-1 multiplexers. Hence, the total number of 2-to-1 multiplexers required is 13. Each 2-to-1 multiplexer requires 9 nand gates and hence a total of $13 \times 9 = 117$ nand gates is required for multiplexers. [1] also requires a modulo 19 adder and a 4-to-1 multiplxer. Hence, the number of nand gates required in this method is the same as that used in the proposed method. [9] uses memory access for computing residues and for the chosen modulus 19, one needs 19 look ups or ROM locations only. This is based on the assumption that a residue is read from the ROM each time an access is made. The addition time using a carry propagation adders is the 10 full adder delay and this is common to all three methods. Table I compares the different architectures.

The proposed method is faster by about 43%. However, if we consider a more moderate distribution of ones and zeros in the given number, the increase in speed may not be as high as 43%, but still better than what is achieved in [1]. Since the number of gates in both methods are more or less the same, no comparison between the two methods in terms of hardware or area is made.

B. Parallel Method

In this method residue computation is performed on all bits simultaneously and all terms required in g_i are accessed in par-

TABLE I
SERIAL METHOD: COMPARISON BETWEEN [1], [9] AND PROPOSED CONVERTER

Method	No. bits in exponent	No. of 2 to 1 MUX	No. of Gates used in MUX	Total No. of Gates (MUX+Logic+Inverter + D Flip Flops)	Total No. of clock cycles (Max)	No. of FAs
Ref [1]	5 bits (4 to 1 MUX)	13	117 (includes MUX in adders)	117+132+5+5FF	32×5	5
Proposed Method	5 bits (4 to 1 MUX	13	117 (includes MUX in adders)	117+132+5+5FF	18×5	5
Ref [9]	-	10	90 (includes MUX in adders)	ROM+5INV+5FF	18	5

TABLE II
PARALLEL METHOD: COMPARISON BETWEEN [1], [9] AND PROPOSED CONVERTER

Method	No. bits in exponent	No. of 2 to 1 MUX	No. of Gates used in MUX	Total No. of Gates (MUX+Logic+Invertes +D Flip Flops)	No. of FAs
Ref [1]	5 bits (4 to 1 MUX	790	7110 (includes MUX in adders)	7110+132+155+155FF	155
Proposed Method	5 bits (4 to 1 MUX	450	4050 (includes MUX in adders)	4050+132+85+85FF	85
Ref [9]		170	1530 (includes MUX in adders)	ROM+1530+85INV+85FF	85

allel. The residues that are output are added using a tree structure of adders. This increases the speed of conversion but at the expense of increased number of multiplexers and adders. In the proposed converter, residue computation is performed only on 18 bits due to the periodicity of 18 for modulus 19. This requires 18 sets each set containing 5 numbers of 4-to-1 multiplexers. In all, for obtaining g_i alone, $18 \times 5 \times 3 = 270$ 2-to-1 multiplexers are needed. For each 5-bit adder, 2 numbers of 5-bit 2-to-1 multiplexers are needed and so for 18 adders, one needs $18 \times 10 = 180$ 2-to-1 multiplexers. Hence, a total of $270 + 180 = 450$ 2-to-1 multiplexers are needed. Therefore, the total hardware requirement is 85 FAs, 450 2-to-1 multiplexers, 85 D latches and 85 inverters in addition to 132 gates that are required to generate the logic functions.

In [1], the conversion is accomplished using 160 numbers of 4-to-1 or 480 2-to-1 multiplexers and 31 modulo 19 adders connected using a tree structure. Therefore, the area requirement is that used by 155 FAs, 155 D latches, 155 inverters and 790 numbers of 2-to-1 multiplexers. Method proposed in [9] uses 19 memory locations for storing the residues and accesses these residues in a sequential manner. All three schemes require 5 levels in the adder tree and as such the adder timing is the same and therefore not included in the comparison.

As far as the hardware is concerned, the proposed architecture uses about 44% less than that used in [1] and 66% less than that used in [9]. In the latter case, ROM is assumed to be made up of D flip—flops. Table II compares the performance of different methods in terms of hardware.

In our comparison in Table I, we have assumed a worst case when all bits in the given number are nonzero. However, if we consider a case when the number nonzero bits is 50%, then the total number of clock cycles for the method proposed in [1] will be 16×5. Since the distribution of 1s and 0s is assumed to be 50%, it can be assumed that there are nine 1s in 18-bit field. Hence, in the case of the proposed method the number of cycles required is 9×5, a similar increase in speed as in the case of all nonzero bits.

The gains in speed and hardware do not necessarily imply that these could be achieved irrespective of the moduli set chosen. The proposed method depends on periodicity of the residues and hence very much dependent on the choice of the moduli set. Hence, care needs to be exercised in choosing a moduli set that has low periodicity in the residues.

VII. CONCLUSION

Forward conversions in RNS have been traditionally implemented using lookup tables. Modifications to memory based systems mainly involved using PEs. Although the use of PEs reduces memory requirements, they have not completely eliminated the use of memory. Furthermore, memory based converters require reprogramming for different moduli sets. In this paper we have presented an extension to memoryless binary to RNS converters that makes them less hardware intensive. The method makes use of the periodicity properties of residues obtained using modular exponentiation. As a result, the new converter requires 40% less hardware for parallel conversion and achieves 43% improvement in speed for serial conversions.

ACKNOWLEDGMENT

The authors are indebted to Dr. P. V. Ananda Mohan for his valuable comments and clarifications during the course of this work.

REFERENCES

[1] A. B. Premkumar, “A formal framework for conversion from binary to residue numbers,” *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 46, no. 2, pp. 135–144, Feb. 2002.
[2] G. Alia and E. Martinelli, “A VLSI algorithm for direct and reverse conversion from weighted binary number to residue number system,” *IEEE Trans. Circuits Syst.*, vol. CAS-31, no. 12, pp. 1425–1431, Dec. 1984.
[3] ——, “VLSI binary—Residue converters for pipelined processing,” *Comput. J.*, vol. 33, no. 5, pp. 473–475, 1990.
[4] N. S. Szabo and R. I. Tanaka, *Residue Arithmetic and Its Applications to Computer Technology*. New York: McGraw-Hill, 1967.
[5] R. M. Capocelli and R. Giancarlo, “Efficient VLSI networks for converting an integer from binary system to residue number system and vice versa,” *IEEE Trans. Circuits Syst.*, vol. 35, no. 11, pp. 1425–1431, Nov. 1988.
[6] S. Piestrak, “Design of residue generators and multioperand modular adders using carry save adders,” in *Proc. 10th IEEE Symp. Comput. Arith.*, Jun. 1991, pp. 100–107.
[7] ——, “Design of residue generators and multioperand modular adders using carry save adders,” *IEEE Trans. Comput.*, vol. 43, no. 1, pp. 68–77, Jan. 1994.
[8] ——, “Design of squarers modulo a with low-level pipelining,” *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 49, no. 1, pp. 31–41, Jan. 2002.
[9] P. V. A. Mohan, “Novel design for binary to RNS converters,” in *Proc. IEEE Int. Symp. Circuits Syst.*, 1994, pp. 357–360.
[10] ——, *Residue Number Systems: Algorithms and Architectures*. Norwell, MA: Kluwer, 2002.
[11] ——, “Efficient design of binary to RNS converters,” *J. Circuits Syst. Comput.*, vol. 9, no. 3/4, pp. 145–154, 1999.