Optimal Table-Lookup Schemes for Binary-to-Residue and Residue-to-Binary Conversions

Behrooz Parhami

Dept. of Electrical & Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

ABSTRACT

Alternate table-lookup schemes for conversion of binary to residue numbers, and vice versa, are presented. They are based on high-radix methods to minimize table size and to speed up the conversion process. Improved variants of VLSI-based pipelined binary-to-residue converters are derived along with balanced, highly regular, pipelined architectures for residue-to-binary conversion in VLSI.

Keywords: High-radix methods, Input/output conversions, Modular arithmetic, Residue number system, Table lookup.

I. INTRODUCTION

The residue number system (RNS) has found numerous applications in the design of high-performance systems in digital signal processing [7]. RNS-based systems are preferable to conventional ones in view of parallelism in arithmetic operations. Recently, very efficient RNS division algorithms have been developed [4] that can pave the way for the use of RNS representations in much wider areas. However, to achieve high performance, fast arithmetic is not sufficient. It is also imperative that conversions to and from RNS be fast and efficient.

To address this problem, let the number u be represented in a radix-R positional number system as follows:

$$u = (u_{n-1}, u_{n-2}, \ldots, u_1, u_0)_R \quad (1)$$

With a given set of moduli $\{m_r, \ldots, m_2, m_1\}$, the RNS representation of the number u is

$$u = (|u|_{m_r}, |u|_{m_{r-1}}, \ldots, |u|_{m_1}) \quad (2)$$

where $|u|_{m_i}$ represents the residue of u with respect to m_i. The binary-to-residue conversion problem is defined as:

Given: $(u_{n-1}, u_{n-2}, \ldots, u_0)$, R, $\{m_r, \ldots, m_2, m_1\}$

Obtain: $(|u|_{m_r}, |u|_{m_{r-1}}, \ldots, |u|_{m_1})$

Similarly, the residue-to-binary conversion is defined as:

Given: $(|u|_{m_r}, |u|_{m_{r-1}}, \ldots, |u|_{m_1})$, R, $\{m_r, \ldots, m_2, m_1\}$

Obtain: $(u_{n-1}, u_{n-2}, \ldots, u_0)$

In this paper, we present architectures and algorithms for fast and cost-effective conversions between binary and residue number representations. The architectures are tree-structured and achieve logarithmic latency, with latches used to make pipelining possible. ROM lookup tables are used in the design, with special attention paid to optimizing table size and speed. In VLSI parlance, our converters are both scalable and area-time efficient.

II. BINARY-TO-RESIDUE CONVERSION

A straightforward technique to convert a number u to RNS representation would be to find its residue with respect to each modulus through a division operation. This is clearly slow and inefficient. Hence the motivation to investigate simpler schemes for this conversion. The positional representation of u satisfies

$$u = \sum_{i=0}^{n-1} u_i R^i \quad (3)$$

where R is the radix, u_i is the ith digit, and the number of digits n is at least $\lceil \log_R u \rceil$. The following identity provides the basis for an alternate scheme to compute the residue of u with respect to a modulus m:

$$u \bmod m = [\sum_{i=0}^{n-1} (u_i R^i \bmod m))] \bmod m \quad (4)$$

Using table lookup, we can perform the conversion with a simple and highly regular circuit. We take the pair (u_i, i) as the index into a ROM table, obtaining $u_i R^i \bmod m$ as output. The value of R will affect the contents of the table and is not an explicit input.

By adding all the returned values modulo m, the result can be calculated very quickly. We need n operations of table lookup and $n - 1$ modular additions. These can be done sequentially using a single table and one modular adder or with various degrees of parallelism. If n tables are used, their contents can be specialized for particular values of i, with u_i provided as the only input.

A. The High-Radix Algorithm

It is possible to reduce the number of table-lookup and addition operations by using a larger radix. We pick a new radix $R' > R$ such that $n' = \lceil \log_{R'} u \rceil < n$ and thus reduce the number of lookup operations as well as the number of subsequent additions. There is also a side benefit that there will be fewer bits in the final sum, making the computation of its modulo-m residue less complex.

To take full advantage of the reduction of the number of table lookups and additions, we have to pick the new radix R' carefully so as to avoid complications due to the need for an initial and final radix conversion.

One solution is to pick R' to be an integer power of R. As an example, for $R' = R^2$, no conversion is needed to and from the new radix since two consecutive digits in the old representation will form one digit $u'_i = (u_{2i+1}, u_{2i})$ in the new representation. Half as many (u'_i, i) pairs are involved in the lookup process, with the number of additions halved as well. This *higher-radix conversion* is easily generalized to the case $R' = R^q$.

B. Effectiveness and Optimality

Even though conversion can be speeded up by selecting a larger radix, the extreme of choosing the radix equal to the maximal range of representation (which leads to only one table access) is clearly impractical. Selection of the radix not only affects the contents of the tables and the required number of additions, but also the size of the tables which is a critical issue in implementation. Thus, the design of a conversion architecture involves tradeoffs in layout regularity, table size, and performance.

In order to simplify the subsequent discussion, we focus on table size and number of additions needed. We consider the cost-effectiveness index E as being inversely proportional to the weighted sum of the relative memory size M and the relative number of additions A:

$$E = (1 + \alpha) / (M + \alpha A) \quad (5)$$

Table size and number of additions for radix 2 are taken as units of measurement and Equation (5) is defined in such a way that the effectiveness index becomes 1 for the radix-2 implementation. The weight parameter α reflects the unequal importance of total memory size and number of additions in the overall complexity.

To get a feeling for the magnitude of α, we note that determining a residue of a k-bit binary number, k two-entry lookup tables and k adders are required. Since each adder is significantly more complex than a 2-word table, α is fairly large, with its value being technology-dependent. To cover a safe interval, we have assumed $10 \le \alpha \le 1000$ in our numerical evaluations.

Figure 1 shows that the effectiveness index E improves initially as the radix R increases and that radices higher than 16 are not likely to be cost-effective unless α is very large. We have included radices that are not powers of 2 in order to show the variations more clearly. Such radices will likely be inefficient in view of radix conversion costs that have not been included in this analysis.

C. Implementation Issues

Figure 2 depicts an existing VLSI design for binary-to-residue conversion which is suitable for pipelining [2]. By arranging the additions in a tree structure, the ith residue $|u|_{m_i}$ can be obtained in $\log_2 n$ adder delays. We can thus transform the design into Figure 3, obtaining a significant speedup. Throughput is determined by the bottleneck in the pipeline which, depending on the detailed design and the technology used, is either the ROM delay or the delay associated with the final (widest) addition.

With r moduli and n digits in the high-radix representation of u, the number of tables is rn. Considering the $r(n-1)$ adders needed, the space complexity may be too great for some applications. As for the time complexity, let T_m and T_a denote ROM access and adder delays, respectively. With the above design, each pass through the pipeline requires $T_m + T_a \log_2 n$ time.

We can reduce the number of tables down to $O(r)$ and use r adders only, leading to time complexity $nT_m + T_a$ which is larger then that of the above design. This space reduction is based on the observation that Equation (4) can be evaluated iteratively by using only one modular multiplication table for u_iR^i and a modulo-m adder for the additions. Fundamentals of modulo-m multiplication can be found in [8]. Parhami and Lai have provided a couple of implementation schemes [5].

As shown in Figure 4, we use shifting to deal with successive digits of u using the same set of hardware elements. The tables in this scheme are general modular multiplication tables for u_iR^i where $i \in [0 .. n-1]$. Each table in the previous scheme also contained u_iR^i but for a specific value of i. The table size for m_i in this lower-complexity scheme is about the total of all tables for m_i in the previous scheme. The adder output must be fed back to produce the modulo-m summation.

Implementation of the modulo-m adder in the above scheme deserves a closer look. We can use either a logic circuit or a lookup table to do the modulo part after normal addition. The tabular version requires only small tables with $2m - 2$ entries for modulo-m addition and provides flexibility in choosing the modulus.

This idea can be incorporated into the original conversion scheme. We replace each adder with a tabular modulo-m adder. This will make the implementation simpler but requires more space for the tables. An alternative is using carry-save adders (CSAs) for the internal additions and a modulo-m adder at the last step. If a $(\log_2 2m)$-bit CSA is faster than access to a $2m$-entry ROM table, then we can accelerate the computation by minimizing the number of table references. However, the last modulo-m addition needs $n(m-1)$ table entries because the CSAs don't adjust the result to within $[0..m-1]$. For the scheme of Figure 4, a tabular adder is preferred because of its flexibility in choosing the modulus and its regularity in layout.

D. Comparison to Previous Work

Two schemes to do binary-to-residue conversion have been proposed here. Although our schemes are based on [2], we use fewer adder levels and obtain higher performance. We also provide a lower-complexity scheme for situations where performance can be traded off to minimize the area. Our performance comparisons were done from a theoretical point of view. Actual performance indices must be obtained through simulation or physical implementation in order to provide realistic evaluation and comparison to other schemes proposed earlier [1], [3].

III. RESIDUE-TO-BINARY CONVERSION

Our method for residue-to-binary conversion is based on a "high-radix" property similar to that used in the binary-to-residue conversion. The major purpose of raising the radix is to balance the size of the lookup tables. Balancing the size of the conversion tables among the moduli leads to the regularity suitable for VLSI realization and a higher performance. Hardware structure of the basic conversion scheme is depicted in Figure 5.

A. The High-Radix Algorithm

The weight w_i associated with the modulus m_i is a value having the RNS representation $(0, \ldots, 0, 1, 0, \ldots, 0)$, where the ith digit is one and all others are zero. With $M = \prod_{i=1}^{r} m_i$, the conversion process can be formulated as:

$$u = [\sum_{i=0}^{n-1} (w_i |u|_{m_i}) \bmod M] \bmod M \quad (6)$$

To realize the conversion based on Equation (6), we take $|u|_{m_i}$ as the index into a table and get $(w_i |u|_{m_i}) \bmod M$ directly. The required table has $\sum_{i=0}^{n-1} m_i$ entries.

In a fully parallel implementation, we use CSAs to compute the sum in Equation (6). This requires $r-1$ adders arranged in a tree with $\lceil \log_2 r \rceil$ levels. For a residue representation system with many moduli, we can use high-radix residue-to-binary conversion to increase the speed of this summation step. Instead of taking a single residue for table access, we may take multiple residues at a time to index each table. Let $S = \{m_1, m_2, \ldots, m_r\}$ be the set of moduli and $P = \{s_1, s_2, \ldots, s_l\}$ be a partition over S with $s_i \subseteq S$, $s_i \cap s_j = \emptyset$ for $i \neq j$, and $S = \cup s_i$. We look up all the residues in the subset s_p at once to produce the partial result u_p

$$u_p = [\sum_{m_i \in s_p} (w_i |u|_{m_i})] \bmod M \quad (7)$$

with the result of the conversion obtained by

$$u_s = \sum_{i=1}^{l} u_p \quad (8)$$

$$u = u_s \bmod M \quad (9)$$

The summation step in Equation (8) needs a $\lceil \log_2 l \rceil$-level CSA tree and $l-1$ adders. Since $l \leq r$, the summation step can be accelerated.

The final result u is obtained after the modulo-M computation of Equation (9). This step requires particular caution and will be discussed in the Subsection C.

B. Partitioning Strategies

The only remaining problem is how to pick an optimal or good partition over S. As an example, for $S = \{2, 3, 5, 7, 11, 13, 17, 19\}$, a balanced partition would be:

$$P = \{\{2, 3, 5, 7\}, \{11, 19\}, \{13, 17\}\}$$

This partition effectively transforms the original RNS to the "high-radix" system with moduli 210, 209, and 221. Hence uniform 256-entry ROMs can be used.

Finding an optimal partition for this method as described in Subsection A is quite difficult and requires extensive searching. Besides, even the best partition may be highly unbalanced for particular sets of moduli. Fortunately, however, combining does not have to occur for entire residues. We can combine residues into equal-size blocks of bits, with each block containing chunks of one or more residues. The counterpart of Equation (7) in this case becomes more complicated, but since its evaluation is done by table lookup, the required hardware is equally simple and, in addition, highly regular.

With the above observation, the analysis of complexity becomes very similar to that of Section II and is thus omitted here for brevity.

C. Implementation Considerations

Implementation of the above scheme is straightforward, but efficiency in the summation step doesn't guarantee overall efficiency. We have to pay attention to the final modulo step also. The major problem of this final step is the modulo-M addition. The method of Section II is not appropriate here because M is much larger than the individual moduli. The table size is dictated by $\prod m_i$ rather than $\sum m_i$.

From Equations (7) and (8), we know that u_p is in the interval $[0 \,..\, M-1]$ and $u_s = \sum u_p \in [0 \,..\, l(M-1)]$. Therefore, trial subtraction and testing can become quite inefficient since it may require up to l steps.

If we use a few most significant bits of u_s to determine the amount to subtract from u_s, then we do not have to approach u iteratively. In fact, only one table lookup and one subtraction would be needed. We have shown (proof omitted here) that for q an integer, qM and $(q+1)M$ differ at some bit of the binary representation to the left of the $\lfloor \log_2 M \rfloor$th bit from the right and that once a stored multiple of M based on the high-order bits of u_s is subtracted from u_s, at most one subtraction is needed to find u. The table size is thus bounded by

$$2(\lceil \log_2(l(M-1)) \rceil - \lfloor \log_2 M \rfloor + 1).$$

For typical values of M, $\lceil \log_2(l(M-1)) \rceil - \lfloor \log_2 M \rfloor + 1$ can be approximated by $\lceil \log_2 l \rceil + 1$. Since l is small in practice, the storage requirements are quite modest. For instance, with $l = 16$, we need to use five bits from u_s to look up a 32-entry table.

D. Comparison to Previous Work

Alia and Martinelli [2] give asymptotic results assuming that all moduli are of the same order. Obviously, such asymptotic results are not practical since the moduli tend to be fairly small and thus significantly different in magnitude. We have provided a "high-radix" residue-to-binary conversion scheme using a method to balance the table size and to speed up the conversion. The regular layout of the scheme makes it particularly suitable for VLSI implementation. Fast computation of the residue of a large number is new to this research and makes our approach highly competitive with previous ones [6].

IV. CONCLUSION

To meet the speed requirement of special-purpose digital systems based on residue number representation, we have proposed fast schemes to perform conversions between positional number systems and RNS. Lookup tables are used for conversions in either direction in order to accelerate the process. The proposed architectures are suitable for VLSI realization with pipelining and can thus support very high conversion rates.

ACKNOWLEDGEMENT

The assistance of Mr. Frank H.-F. Lai in the preparation of an early draft of this paper is gratefully acknowledged.

REFERENCES

[1] Alia, G. and E. Martinelli, "A VLSI Algorithm for Direct and Reverse Conversion from Weighted Binary Number System to Residue Number System", *IEEE Trans. Circuits and Systems*, Vol. 31, No. 12, pp. 1033-1039, 1984.

[2] Alia, G. and E. Martinelli, "VLSI Binary-Residue Converters for Pipelined Processing", *The Computer Journal*, Vol. 33, No. 5, pp. 473-474, 1990.

[3] Capocelli, R.M. and R. Giancarlo, "Efficient VLSI Networks for Converting an Integer from Binary System to Residue Number System and Vice Versa", *IEEE Trans. Circuits and Systems*, Vol. 35, No. 11, pp. 1425-1430, Nov. 1988.

[4] Hung, C.Y. and B. Parhami, "An Approximate Sign Detection Method for Residue Numbers and Its Application to RNS Division", To appear in *Computers & Mathematics with Applications*.

[5] Parhami, B. and H.-F. Lai, "Alternate Memory Compression Schemes for Modular Multiplication", *IEEE Trans. Signal Processing*, Vol. 41, No. 3, pp. 1378-1385, Mar. 1993.

[6] Shenoy, A.P. and R. Kumaresan, "Residue to Binary Conversion for RNS Arithmetic Using Only Modular Look-Up Tables", *IEEE Trans. Circuits and systems*, Vol. 35, No. 9, pp. 1158-1162, Sep. 1989.

[7] Soderstrand, M.A., W.K. Jenkins, G.A. Jullien, and F.J. Taylor (Editors), *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, IEEE Press Reprint Collection, 1986.

[8] Waser, S. and M.J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart and Winston, 1983. *Note*: Other standard texts in computer arithmetic also provide the needed introduction.

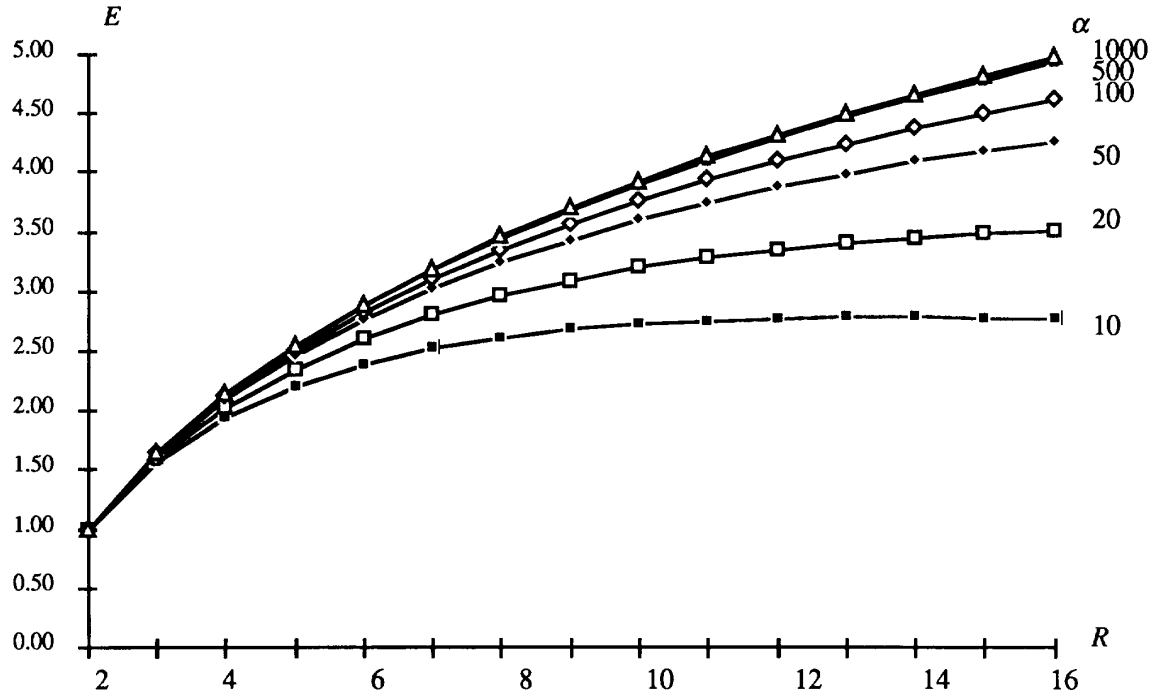

Figure 1. Effectiveness index for binary-to-residue conversion scheme as a function of the radix R.

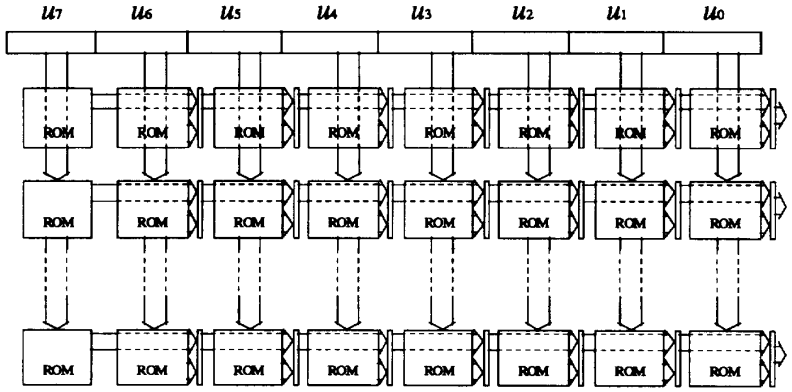

Figure 2. VLSI binary-to-residue converter architecture suitable for pipelining.

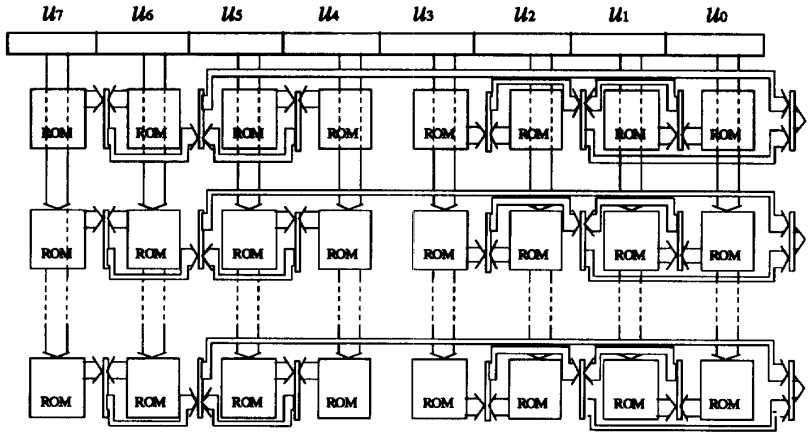

Figure 3. Pipelined binary-to-residue converter design with trees of adders.

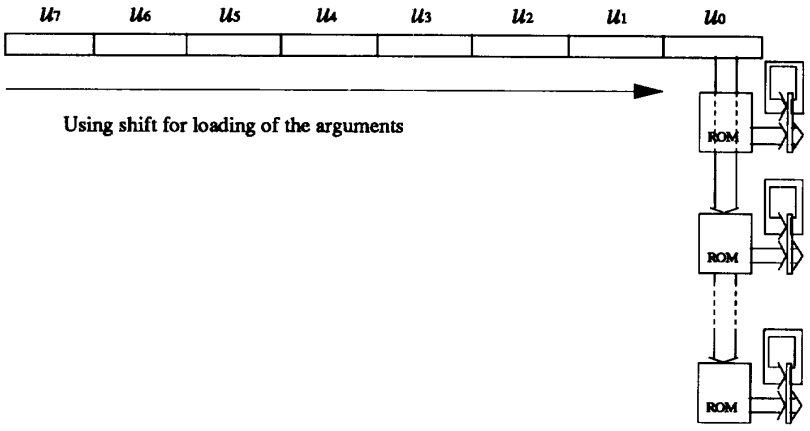

Figure 4. A lower-complexity scheme for binary-to-residue conversion.

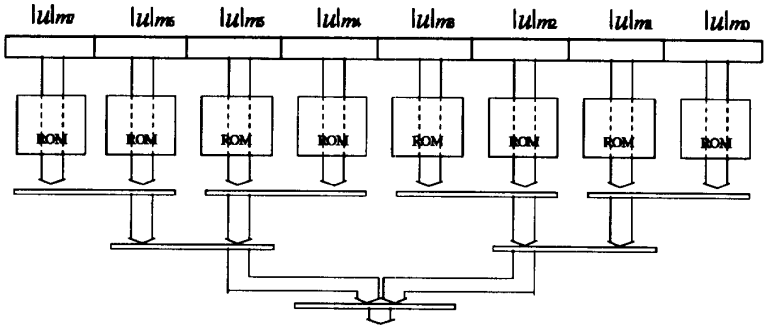

Figure 5. Basic hardware architecture for residue-to-binary conversion.