OPTIMAL TABLE LOOKUP SCHEMES FOR VLSI IMPLEMENTATION OF INPUT/OUTPUT CONVERSIONS AND OTHER RESIDUE NUMBER OPERATIONS

B. Parhami and **C. Y. Hung**
Electrical and Computer Engineering Dept.
University of California
Santa Barbara, California

Abstract - Residue number representation has become a viable alternative for fast, area-efficient VLSI realization of high-performance signal processing hardware. Wider applicability and improved cost/performance of residue-based VLSI implementations of signal processing algorithms are critically dependent on efficient realization of I/O data conversions and several other support functions that are treated in this paper. Alternate table-lookup schemes for conversion of binary to residue numbers, and vice versa, are presented. Input widths of lookup tables can be changed freely through a repartitioning scheme to provide tradeoffs between table size (area) and computation speed. 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. The input repartitioning method is shown to be applicable to other important residue number system operations, including sign detection, mixed-radix conversion, and base extension.

INTRODUCTION

The residue number system (RNS) has found numerous applications in the design of high-performance systems in digital signal processing [9]. RNS-based systems are preferable to conventional ones in view of parallelism in arithmetic operations. Recently, very efficient RNS division algorithms have been developed [5] that can pave the way for the use of RNS representations in much wider areas, both within and outside signal processing. However, to achieve high performance, fast arithmetic is not sufficient. It is also imperative that support functions such as I/O data conversions, sign detection, magnitude comparison, scaling, and error checking be fast and efficient.

The need for I/O data conversion to and from standard positional representation is self-evident. Sign detection and magnitude comparison are not only directly useful in threshold-based control and other decision situations

0-7803-2123-5/94 $4.00 © 1994 IEEE

but also form building blocks for synthesizing important functions such as overflow detection, scaling, and division. Another helpful tool is mixed-radix conversion which is used both for magnitude comparison and D/A conversion [6]. We also deal with base extension, a support function used extensively for scaling and error checking.

To address the input/output conversion problem, let the number X be represented in a radix-R positional number system (PNS) as follows:

$$X = \sum_{i=0}^{r-1} u_i R^i = (u_{r-1}, u_{r-2}, \ldots, u_1, u_0)_{\text{PNS}} \tag{1}$$

With the set of moduli $\{m_n, \ldots, m_2, m_1\}$, RNS representation of X is

$$X = (x_n, x_{n-1}, \ldots, x_1)_{\text{RNS}} \tag{2}$$

where $x_i = |X|_{m_i}$ represents the residue of X with respect to m_i. The binary-to-residue conversion problem is defined as obtaining the x_ks, given the u_js, m_is, and R. Similarly, residue-to-binary conversion is defined as obtaining the u_js from the m_is, x_ks, and R.

In this paper, we present tree-structured pipelined architectures for fast, cost-effective binary-to-residue and residue-to-binary conversion problems. ROM lookup tables are used in the design, with special attention paid to optimizing table size (layout area) and circuit speed. In VLSI parlance, our converters are both scalable and area-time efficient. We then note that sign detection, mixed-radix conversion, and base extension all require the computation of a weighted modular summation of the type needed for residue-to-binary conversion and that any such operation can benefit from our flexible table-lookup architecture.

BINARY-TO-RESIDUE CONVERSION

The following identity, derived from Equation (1), provides a basis for an alternate scheme to compute the residue of X with respect to a modulus m:

$$X \bmod m = \left[\sum_{i=0}^{r-1} (u_i R^i \bmod m)\right] \bmod m \tag{3}$$

To evaluate (3) based on table lookup, 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 r operations of table lookup and $r-1$ modular additions. These can be done sequentially using a single table and one modular adder or with various degrees of parallelism. If r tables are used, their contents can be specialized for particular values of i.

Alternate-Radix Algorithm

In some implementation platforms, it is desirable to reduce the number of table-lookup and addition operations by using a larger radix, and consequently larger lookup tables. In some others, we may want the opposite tradeoff. Both alternatives are accomplished in the same way: we pick a new radix R', usually such that R and R' are integer powers of a common factor. The conversion problem is now transformed into two problems; first from radix R to radix R' and then from radix R' to residue.

Efficiency and Optimality

Selection of the radix affects not only the contents of the tables and the required number of additions, but also the conversion speed and the cost of tables and adders. Thus, the design of a conversion architecture involves tradeoffs in layout regularity, table size, and performance.

We define the area efficiency index E as being inversely proportional to the weighted sum of the relative memory size M and the relative number of additions A:

$$E = \frac{1+\alpha}{M+\alpha A} \tag{4}$$

Table size and number of additions for radix 2 are taken as units of measurement and Equation (4) is defined in such a way that E becomes 1 for the radix-2 implementation. Since additions are performed with residue-size operands, adder width is independent of the radix chosen. The relative number of additions A is thus proportional to the cost of adders. The weight parameter α models the unequal costs of adder and memory. Specifically,

$$\alpha = \frac{\text{Adder cost (radix-2)}}{\text{Memory cost (radix-2)}} = \frac{\text{Cost of } k^2 \text{ bits of adders}}{\text{Cost of } 2k^2 \text{ bits of memory}},$$

where k is the number of bits of the input binary number. The ratio α depends on layout density and the technology used for the two types of circuits.

Figure 1 shows the area efficiency index E versus the radix R for several values of α. We see that E improves initially as R increases and that radices higher than 16 are not likely to be cost-effective unless α is very large. The optimal radix is between 4 and 16 for the range of α plotted.

The area efficiency index E is a good measure when the design is fully pipelined at maximal throughput so that the cost of hardware is the main concern. In other cases (i.e. when pipelining is not used or when minimizing latency is also important), a better basis for comparison of competing designs is area-time product. As an example, when computation time is linear in the number of additions, the area-time efficiency index E' becomes:

$$E' = \frac{E}{T} = \frac{1+\alpha}{(M+\alpha A)A}$$

Figure 2, depicting E' versus the radix R, exhibits the same general trend as Figure 1, but higher radices gain a little because they imply faster conversion.

The optimal radix is between 8 and 32 for the range of α plotted. Hence the conclusion that the optimal radix is fairly small for both optimality measures and for a wide range of technology-dependent parameters.

Implementation Issues

A VLSI design for binary-to-residue conversion which is suitable for pipelining is given in [2]. By arranging the additions in a tree structure, the ith residue $|X|_{m_i}$ can be obtained in $\log_2 r$ adder delays (Figure 3), leading to a significant speedup. With n moduli and r digits in the high-radix representation of X, the number of tables is nr. Considering the $n(r-1)$ adders needed, the space complexity may become unacceptable. As for the time complexity, let T_m and T_a denote ROM access and adder delays, respectively. Then, each pass through the pipeline requires $T_m + T_a \log_2 r$ time.

We can reduce the number of tables to $O(n)$ and use n adders only, leading to time complexity $r \max(T_m, T_a)$ which is larger then that of the above design. As shown in Figure 4, we use shifting to deal with successive digits of X using the same set of hardware elements. The tables in this scheme are modular tables for $u_i R^i$ where $i \in [0, r-1]$. Compared to the two previous schemes, the number of adders is reduced by a factor of r, while the total table size remains the same.

Figure 5 shows another design that reduces the total memory size as well. Two tables may be required for each modulus. The table on the right provides $u \bmod m$ given u. This table can be omitted when $R \leq m$. The table on the left produces $xR \bmod m$ given x. When $R \leq m$ and $R \approx m$, the total table size is reduced by a factor of about r.

Table-based modular addition can be incorporated into the original conversion scheme. 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 $r(m-1)$ table entries.

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. Actual performance indices must be obtained through simulation or physical implementation in order to augment our theoretical analyses and to provide realistic comparison to other schemes proposed earlier [1, 3].

RESIDUE-TO-BINARY CONVERSION

Our method for residue-to-binary conversion is based on repartitioning of input residue digits in a way similar to that used in the binary-to-residue conversion. The conversion process can be formulated as:

$$X = \left[\sum_{i=1}^{n}(w_i x_i \bmod M)\right] \bmod M, \tag{5}$$

where $M = \prod_{i=1}^{n} m_i$, and the weight w_i associated with the modulus m_i is a value having the RNS representation $(0, \ldots, 0, 1, 0, \ldots, 0)$, with only the ith digit being 1. To realize the conversion based on Equation (5), we take x_i as the index into a table and get $(w_i x_i \bmod M)$ directly.

If, as is commonly assumed, $w_i x_i \bmod M$ is read from a separate table for each modulus m_i, imbalance in the size of moduli leads to irregular hardware structure. Even with uniform moduli, the modulus size that is optimal for residue arithmetic may not be optimal, or even feasible, for table lookup.

High-Radix Algorithm

A "high-radix" residue representation can be created by simple combination of residues. Instead of taking a single residue for table access, we may take multiple residues at a time to index each table. Let $S = \{m_1, m_2, \ldots, m_n\}$ 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,

$$u_p = \left[\sum_{m_i \in s_p}(w_i x_i)\right] \bmod M, \tag{6}$$

with the result of the conversion obtained by

$$X = \left(\sum_{p=1}^{l} u_p\right) \bmod M. \tag{7}$$

The summation step in Equation (7) needs a $\lceil \log_2 l \rceil$-level CSA tree and $l - 1$ adders. Since $l \leq n$, the summation step can be accelerated.

Repartitioning of Input Bits

Finding an optimal partition for the high-radix method in the previous subsection is difficult. Besides, even the best partition may be highly unbalanced for particular sets of moduli. Furthermore, it only steps up the radix; it does not offer any help when the modulus size is already too large for cost-effective table lookup implementation. 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 basis of the input repartitioning technique is the observation that the original weighted modular summation in Equation (5) can be rewritten as another modular summation with all input bits x_{ik} individually weighted:

$$X = \left[\sum_{i=1}^{n}\sum_{k=0}^{s_i-1}(w_{ik}x_{ik} \bmod M)\right] \bmod M,$$

where s_i is the number of bits required to represent the residue x_i, x_{ik} is the kth bit of x_i, and $w_{ik} = 2^k w_i \bmod M$. As the weight w_{ik} is known for each bit of each residue digit, we can arbitrarily partition all residue digits at the bit level for input to the lookup tables, and prepare the table contents according to the partition, as long as each bit x_{ik} goes to exactly one lookup table. Figure 6 shows an example of such a repartitioning. With the above observation, the analysis of complexity becomes very similar to that of Section II and is thus omitted here for brevity.

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 $\sum m_i$. Since u_p is in the interval $[0, M-1]$ and $u_s = \sum u_p \in [0, l(M-1)]$, trial subtraction and testing can become quite inefficient.

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 X iteratively. In fact, only one table lookup and one subtraction would be needed. We have shown 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 X. The number of table entries is thus bounded by

$$2^{(\lceil \log_2(l(M-1)) \rceil - \lceil \log_2 M \rceil + 1)}.$$

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

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 [8].

OTHER APPLICATIONS

The repartitioning of residue digits, based on the associativity and commutativity of modular summation, applies to a number of other residue number operations. Many operations in RNS have the common form

$$c = (\textstyle\sum_i a_i x_i) \bmod b. \tag{8}$$

In most practical cases, each digit x_i is represented as a binary string, and consequently the above can be rewritten as

$$c = (\textstyle\sum_{i,j} a_{ij} x_{ij}) \bmod b. \tag{9}$$

In a lookup table implementation, the conventional method requires n lookup tables each with m_i entries. The implication of Equation (9) is that individual bits of all residue digits can be arbitrarily combined to access the lookup tables, as shown in the example in Figure 6. The lookup table can thus be freely structured to match the application requirement and the implementing technology in an optimal way.

We show that the repartitioning technique is applicable to algorithms for approximate sign detection, mixed-radix conversion, and base extension. These algorithms can in turn be used in scaling, division, overflow detection, and error detection and correction. The following notation is used. The product of all moduli is M, the products of all moduli but the ith one is M_i, and the multiplicative inverse of M_i with respect to m_i is $\hat{M}_i$. Note that the previously defined weight w_i for residue-to-binary conversion is equivalent to $\hat{M}_i M_i$, since $w_i \equiv 1 \bmod m_i$ and $w_i \equiv 0 \bmod m_j$ for $j \neq i$.

Approximate Sign Detection

In [5], Hung and Parhami propose an approximate sign detection procedure and demonstrate its use in integer division of RNS numbers. Integers X in the range $-M(1/2 - \epsilon) \leq X \leq M(1/2 - \epsilon)$ are represented, where ϵ is a precision parameter of the algorithm. Given the x_i's the approximate sign detection algorithm determines whether the sign of X is positive, negative, or indeterminate. When the returned sign is indeterminate, $|X|$ is guaranteed to be no larger than ϵM.

The procedure computes the modular sum $EF(X)$ as an approximation to $|X|_M / M$ using scaled approximate CRT decoding.

$$EF(X) = \left[\sum_{i=1}^{n} \text{Round}\left(\frac{x_i \hat{M}_i}{m_i}, t\right)\right] \bmod 1, \tag{10}$$

where t in the rounding function controls the precision of rounding, and is related to ϵ. The sign is determined by comparing $EF(X)$ against some constants. Fractional values $\text{Round}\left(\frac{x_i \hat{M}_i}{m_i}, t\right)$, for each i and each $x_i \in [0, m_i - 1]$, are precomputed and stored in lookup tables. As the evaluation of $EF(X)$ falls under the modular summation form of Equation (8), the input repartitioning strategy can be used to bring flexibility to the implementation.

Parallel Mixed-Radix Conversion

In [4], Huang presents a parallel algorithm for mixed-radix conversion. The mixed-radix conversion problem is that of converting from residue representation $X = (x_n, \ldots, x_1)$ to mixed-radix form $< y_n, \ldots, y_1 >$ so that

$$X = y_1 + y_2 m_1 + y_3 m_1 m_2 + \cdots + y_n m_1 m_2 \cdots m_{n-1}.$$

The number X is unsigned, and $0 \leq y_i < m_i$.

Huang's algorithm utilizes the weighted modular summation form in the Chinese Remainder Theorem (CRT):

$$X = \left[\sum_{i=1}^{n} x_i \hat{M}_i M_i\right] \bmod M.$$

With x_i's as inputs, lookup tables provide the quantity $(x_i \hat{M}_i M_i) \bmod M$ in mixed-radix representation. The summation is then carried out in mixed-radix representation as well. The formulation for mixed-radix conversion again matches the modular summation form of Equation (8), and thus can benefit from the input repartitioning scheme.

Base Extension with Redundant Modulus

In [7], Shenoy and Kumaresan show a procedure for base extension of residue numbers when a redundant residue is available. Base extension refers to the problem of finding some unknown residues given known residues and the condition that the number is representable by the known residues. Conventionally, base extension is carried out with mixed-radix conversion, and takes $O((n+k)n)$ operations, where n and k are the number of known and unknown residues, respectively. The method by Shenoy and Kumaresan takes $O(nk)$ operations, and therefore offers significant improvement over the conventional method when k is small, say, $O(1)$. For brevity, we shall restrict $k = 1$ in the following discussion, but the algorithms can be easily extended to multiple unknown residues. Let $x_1, \ldots, x_n$ be the known residues, x_R be the redundant residue with respect to modulus $m_R \geq n$. The unsigned number X represented is in the range $0 \leq X < M$, where $M = \prod_{i=1}^{n} m_i$ (excluding m_R and m_{n+1}). We wish to find x_{n+1}, the residue with respect to m_{n+1}.

The algorithm is based on an alternate form of CRT decoding,

$$X = \left[\sum_{i=1}^{n} M_i |x_i \hat{M}_i|_{m_i}\right] - A(X)M, \tag{11}$$

where $A(X)$, also called overflow count, indicates how many times the n-term summation in Equation (11) overflows M, and is in $[0, n-1]$. The algorithm finds the unknown residue in two steps. First, it applies Equation (11) to compute $A(X)$ using the redundant residue.

$$A(X) = |A(x)|_{m_R} = \left[\left(\sum_{i=1}^{n} m_i^{-1} |x_i \hat{M}_i|_{m_i}\right) - x_R M^{-1}\right] \bmod m_R. \tag{12}$$

Equation (12) yields $A(X)$ since $0 \leq A(X) < n$ and $n \leq m_R$. In the second step, Equation (11) is applied again to find the unknown residue.

$$x_{n+1} = \left[\left(\sum_{i=1}^{n} M_i |x_i \hat{M}_i|_{m_i} \right) - A(X)M \right] \bmod m_{n+1}. \tag{13}$$

In a direct lookup-table implementation of the algorithm, we need $2n$ tables to provide $(m_i^{-1}|x_i\hat{M}_i|_{m_i}) \bmod m_R$ and $(M_i|x_i\hat{M}_i|_{m_i}) \bmod m_{n+1}$ values given x_i as input. With the flexible table lookup scheme, both steps can be implemented with table size independent of the modulus size.

Note that both (12) and (13) have modulo m_i in each term of the modular summation, and therefore do not comply with the common form of Equation (8). The $A(X)$ computed with repartitioned table inputs indeed is different from $A(X)$ computed with original lookup table structure, but as long as the inputs x_is are repartitioned the same way in both steps, the final result is correct. The repartitioned CRT decoding essentially defines a new overflow count, which is in turn used to find the unknown residue by applying the same repartitioned CRT decoding again.

Base Extension with Approximate Decoding

The base extension method by Shenoy and Kumaresan requires a redundant modulus, $m_R \geq n$, and utilizes only $1/m_R$ of the potential dynamic range. Hence, a relatively large fraction $(m_R - 1)/m_R \leq 1 - 1/n$ of the dynamic range is excluded. The following procedure achieves base extension with the same computational complexity, and allows $(1 - \epsilon)$ times the potential dynamic range to be used, where ϵ can be made arbitrarily small.

Our algorithm also first computes $A(X)$ then uses $A(X)$ to compute the unknown residue. We compute $A(X)$ differently using

$$A(X) = \mathrm{Int} \left[\frac{\epsilon}{2} + \sum_{i=1}^{n} \mathrm{Round} \left(\frac{|x_i \hat{M}_i|_{m_i}}{m_i}, t \right) \right] \tag{14}$$

for input X in $[0, (1-\epsilon)M]$. The rounding function is defined as having error in $[-t/2, t/2)$. The parameter t is related to the exclusion fraction ϵ by $nt < \epsilon$. Given $A(X)$, the unknown residue x_{n+1} is evaluated from Equation (13).

We provide an informal justification of Equation (14) as follows. $A(X)$ can be written as

$$A(X) = \mathrm{Int} \left(\sum_{i=1}^{n} \frac{|x_i \hat{M}_i|_{m_i}}{m_i} \right). \tag{15}$$

Equation (14) uses rounded terms instead of exact terms in Equation (15). The additional term $\epsilon/2$ can be viewed as being distributed to each term to change symmetric rounding to rounding up. As Int always rounds down, $A(X)$ can be computed by applying Int to the sum of rounded-up terms provided that the total rounding error does not push the sum to or over integer units, e.g., from 3.8 to 4.0 or 4.1. The upper portion of dynamic range is excluded to give room for the the rounding errors.

As in the previous section, application of the table repartition technique yields a value of $A(X)$ different from the original formulation. However, as long as the same repartitioning is applied to the second step, computing the unknown residue, correct result is obtained.

CONCLUSION

To meet the speed requirement of special-purpose digital systems based on residue number representation, we have proposed flexible, fast, and cost-effective schemes to perform conversions between positional number systems and RNS. The proposed approach can use uniform lookup tables of varying sizes in order to speed up the conversion process while providing opportunities for speed/cost tradeoffs. The uniformity in the size of the lookup tables, independent of the multiplicity and magnitude of the RNS moduli, is a novel feature of our work which has significant implications for regular, area-efficient VLSI implementations and high-throughput pipelined designs. The input repartitioning method used in I/O data conversions was shown to benefit several other residue number operations such as sign detection, mixed-radix conversion, and base extension that are useful independently and also as building blocks for synthesizing other important functions.

References

[1] G. Alia 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*, 31(12):1033–1039, 1984.

[2] G. Alia and E. Martinelli. VLSI binary-residue converters for pipelined processing. *The Computer Journal*, 33(5):473–474, 1990.

[3] 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 and Systems*, 35(11):1425–1430, November 1988.

[4] C. H. Huang. A fully parallel mixed-radix conversion algorithm for residue number applications. *IEEE Trans. Computers*, 32(4):398–402, 1983.

[5] C. Y. Hung and B. Parhami. An approximate sign detection method for residue numbers and its application to RNS division. *Computers and Mathematics with Applications*, 27(4):23–35, February 1994.

[6] W. K. Jenkins. Techniques for residue-to-analog conversions for residue-encoded digital filters. *IEEE Trans. Circuits and Systems*, 25:555–562, 1978. Also in [9] pp. 31–38.

[7] A. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *IEEE Trans. Computers*, 38(2):292–297, 1989.

[8] A. P. Shenoy and R. Kumaresan. Residue to binary conversion for RNS arithmetic using only modular look-up tables. *IEEE Trans. Circuits and Systems*, 35(9):1158–1162, September 1989.

[9] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, editors. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing.* IEEE Press, 1986.

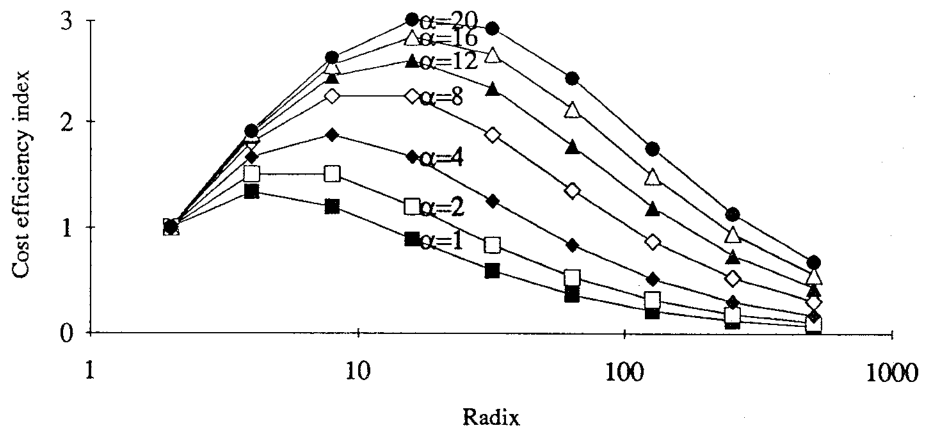

Figure 1 Cost efficiency index for binary-to-residue conversion versus the radix.

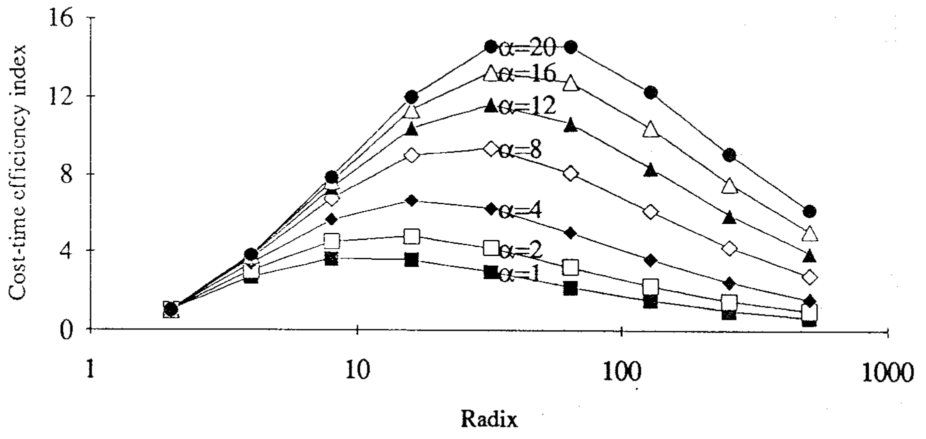

Figure 2 Cost-time efficiency index for binary-to-residue conversion.

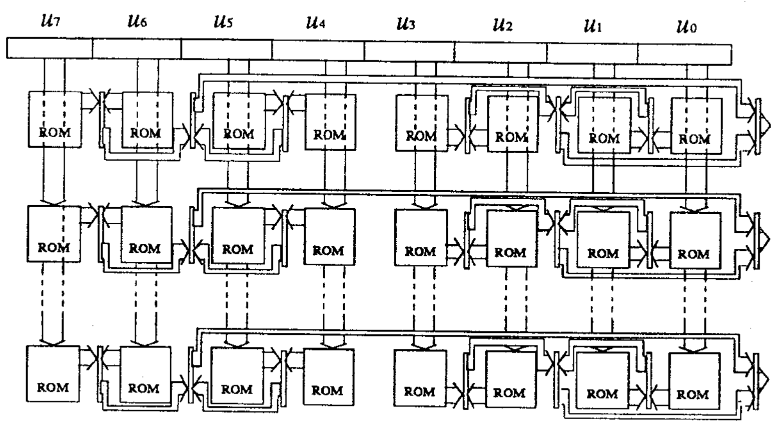

Figure 3 Pipelined binary-to-residue converter design with trees of adders.

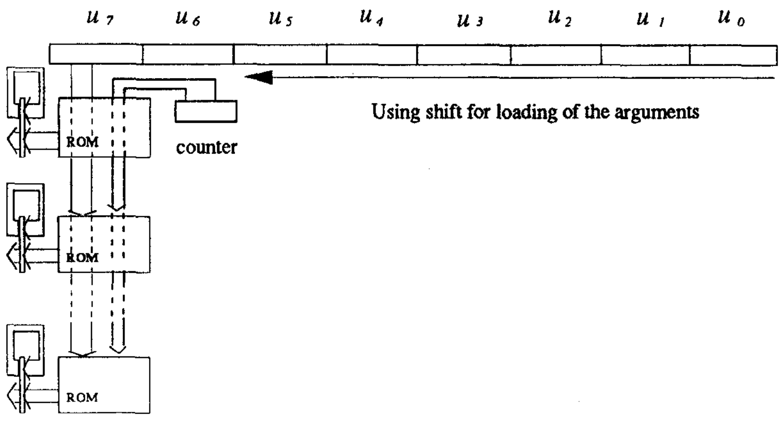

Figure 4 A lower-complexity scheme for binary-to-residue conversion.

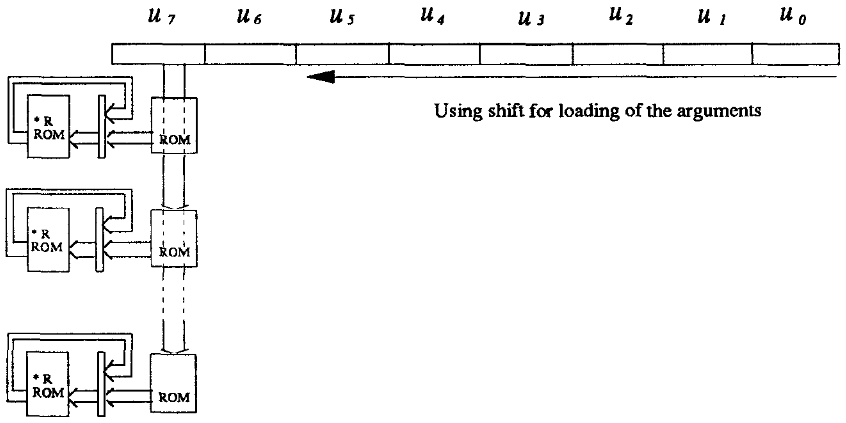

Figure 5 A more memory-efficient design for binary-to-residue conversion

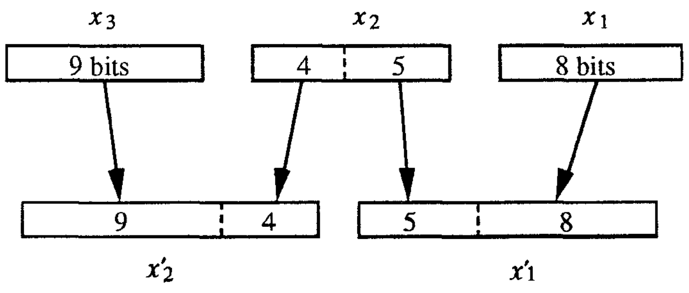

Figure 6 Example of repartitioning for table lookup.