ARITHMETIC ALGORITHMS FOR OPERANDS ENCODED IN TWO-DIMENSIONAL LOW-COST ARITHMETIC ERROR CODES*

Algirdas Avižienis

UCLA Computer Science Department
University of California
Los Angeles, CA 90024, USA

Abstract

A generalization of low-cost residue codes into two-dimensional encodings was presented and error detecting and error correcting properties of two dimensional inverse residue codes were discussed previously. This paper presents byte-serial checking, additive inverse (complementation), and addition algorithms for operands encoded in two-dimensional residue and inverse residue codes.

1. Introduction

A general approach to the cost and effectiveness study of low-cost arithmetic error codes has been presented in [AVIZ 71a]. This paper introduced the concepts of *inverse residue* codes and of *multiple* arithmetic error codes. The concept of *repeated use faults* was presented and the effectiveness of various arithmetic codes with respect to both *determinate* and *indeterminate* repeated-use faults was established. An important result was the proof that inverse residue codes can detect the "compensating" determinate repeated-use faults that are not detected by ordinary residue codes. The modulo 15 inverse residue code was applied in the JPL-STAR experimental computer [AVIZ 71b]. Further results on determinate faults were presented in [PARH 73] and [PARH 78]. An extension to signed-digit arithmetic is found in [AVIZ 81]. J. F. Wakerly has analyzed the detectability of unidirectional multiple errors [WAKE 75], and A.M. Usas has demonstrated the advantages of inverse residue codes for multiple unidirectional error detection, when compared to inverse checksum codes [USAS 78]. Bose and Rao have considered unidirectional one-line error correcting codes using a combination of byte parity and residue (not low-cost) encoding [BOSE 80].

A new generalization presented in [AVIZ 83] extended the application of low-cost inverse residue codes into two dimensions: row (byte) and column (line) residues. This extension improves the detection of errors, especially of those due to indeterminate faults, and provides certain error-correction capabilities. Of special interest to current VLSI implementations of arithmetic are the advantages offered by two-dimensional inverse residue codes in the detection and correction of errors that affect byte-wide communication paths and processing elements. Such paths are widely used in high-performance array processors, systolic arrays, and for inter-processor communication in large multi-processor systems. Byte-wide processing elements are very suitable for the implementation of large processing arrays [AVIZ 70], [TUNG 70] and variable-precision signed-digit arithmetic [AVIZ 62].

This paper presents the fundamental byte-serial arithmetic algorithms for operands encoded in two-dimensional low-cost inverse residue codes. The algorithms are:

(a) the line-residue checking algorithm;

(b) the additive inverse (complementation) algorithm;

(c) the addition algorithm.

A brief review of the error-detecting and error-correcting properties of 2-D inverse residue codes follows the description of arithmetic algorithms.

2. Model of the Byte-Serial Communication and Computation Path

We consider a *communication and computation path* consisting of b bit lines $(X^0, X^1, \ldots, X^{b-1})$. The binary operand X consists of kb bits, processed as k bytes $(X_0, \ldots, X_i, \ldots, X_{k-1})$ of b bits length each. Figure 1 shows the notation used in this paper.

Two types of low-cost residue encoding are applicable to the operand X:

(a) *Residue Code*: the k bytes carry an error-

* This research has been supported by ONR contract N00014-83-K-0493.

detecting code check byte X_k that represents the modulo 2^b-1 residue X' of the operand X: $X' = (2^b-1)|X$; and the operand is now $k+1$ bytes long. Usually the residue value $X'=0$ is represented by a string of b ones. If the all-zero operand can exist, its residue will be b zeros, unless explicitly disallowed.

(b) *Inverse Residue Code*: the inverse residue byte X_k represents the value X'' that is the (2^b-1)'s complement of X'. It is obtained as $X'' = (2^b-1)-X' = (2^b-1)-(2^b-1)|X$; and the operand is again $k+1$ bytes long. The residue value $X'=0$ is represented by b ones, and the inverse residue X'' in this case is represented by b zeros. The all-zero operand X has an inverse residue code X'' represented by b ones.

To form a two-dimensional residue encoding, one more check line X^b is added to the communication and computation path (Figure 1). The lines $X^0, X^1, \ldots, X^{b-1}$ are summed modulo $2^{k+1}-1$ to get the *line-residue* of X. Two classes of 2-D low-cost residue codes can be employed:

(c) *Two-dimensional Residue Code*: the check bits X_i^b of the check line X^b represent the modulo $2^{k+1}-1$ *line-residue* X_L' :

$$X_L' = (2^{k+1}-1) \mid \sum_{j=0}^{b-1} X^j \text{ ; where } X^j = \sum_{i=0}^{k} X_i^j \, 2^i$$

(d) *Two-dimensional Inverse Residue Code*: the check bits X_i^b on the check line X^b represent the modulo $2^{k+1}-1$ *inverse line-residue* X_L'' :

$$X_L'' = (2^{k+1}-1) - X_L'$$

It is important to note that the bits $(X_k^{b-1}, \ldots, X_k^0)$ of the check byte X_k are treated as the most significant bits of the lines $X^{b-1}, \ldots, X^0$ when the line-residue of X is determined. The line-residue encoding is superimposed on the already encoded operand.

Check line X^b	line X^{b-1}		line X^j		line X^1	line X^0	byte symbol
X_0^b	X_0^{b-1}	$\cdots$	X_0^j	$\cdots$	X_0^1	X_0^0	X_0
$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$	$\vdots$
X_i^b	X_i^{b-1}	$\cdots$	X_i^j	$\cdots$	X_i^1	X_i^0	X_i
$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$	$\vdots$
X_{k-1}^b	X_{k-1}^{b-1}	$\cdots$	X_{k-1}^j	$\cdots$	X_{k-1}^1	X_{k-1}^0	X_{k-1}
X_k^b	X_k^{b-1}	$\cdots$	X_k^j	$\cdots$	X_k^1	X_k^0	check byte X_k

Figure 1. Model of the Path and the Operand X

3. A Byte-Serial Line-Residue Checking Algorithm

Given a Two-Dimensional Inverse Residue encoded operand X (as shown in Figure 1), the line-residue checking algorithm requires the computing of the *line check result* $R(L)$:

$$R(L) = (2^{k+1}-1) \mid \sum_{j=0}^{b} X^j; \text{ where } X^j = \sum_{i=0}^{k} X_i^j \, 2^i$$

An "all-ones" $R(L)$ indicates a valid encoding; all other values of $R(L)$ indicate an error.

In the byte-serial implementation, one byte of X becomes available at a time, with the least significant byte X_0 arriving first. The "line sum" $\sum_{j=0}^{b} X^j$ designated by $\sum(L)$, is computed as the weighted sum of the "count of ones" in each byte:

$$\sum(L) = \sum_{j=0}^{b} X^j = \sum_{i=0}^{k} \Big(\sum_{j=0}^{b} X_i^j\Big) \, 2^i$$

In order to get $R(L)$, the modulo $(2^{k+1}-1)$ residue of $\sum(L)$ must be computed. That requires an "end-around-carry" addition of the "overflow bits" $(S_{k+m}, \ldots, S_{k+1})$ of $\sum(L)$ that are in the positions $k+1, \ldots, k+m$ of $\sum(L)$. Since a full-length addition delay is inacceptable in high-speed byte-organized computing (e.g., systolic arrays), a *fast* line-residue checking algorithm is developed here that requires only a short, m-bit (with $2^m \geq b+1$) addition after $\sum(L)$ has been byte-serially computed.

The speed-up is accomplished by the simultaneous computing of *two tentative line sums* $\sum(L)$ and $\sum(L)' = \sum(L)+2^m$. The value of m is determined by the maximum value of the line sum $\sum(L)$. An upper bound for $\sum(L)$ is obtained by assuming all digits X_i^j $(0 \leq i \leq k; 0 \leq j \leq b)$ to have the value "1". (This situation cannot occur for a valid encoding, but could be caused by an error.) In this case,

$$\sum(L)_{\max} = (b+1)(2^{k+1}-1) = b2^{k+1} + (2^{k+1}-1) - b$$

and the "overflow bits" represent the value b with respect to the position $k+1$ of the line sum $\sum(L)$. The value of m is the smallest integer that satisfies the condition:

$$2^m - 1 \geq b \quad or \quad 2^m \geq b+1$$

For example, 8-bit bytes (b=8) will need m=4, regardless of operand length k (in bytes).

After $\sum(L)$ and $\sum(L)' = \sum(L)+2^m$ have been computed, the m "overflow bits" $(S_{k+m}, \ldots, S_{k+1})$ of $\sum(L)$ are added to the m least significant bits $(S_{m-1}, \ldots, S_0)$ of $\sum(L)$. The resulting carry-out C_m determines the choice of the bits $(S_k, \ldots, S_m)$:

(a) If $C_m=0$, $(S_k,\ldots,S_m)$ come from $\sum(L)$

(b) If $C_m=1$, $(S_k,\ldots,S_m)$ come from $\sum(L)'$

The "all ones" line check result $(S_i=1\ ; 0\leq i\leq k)$ indicates the absence of errors; any other line check result is an error indication. The determination whether $(S_k,\ldots,S_m)$ are "all ones" is done while these bits are computed. The final step is to test whether the bits $(S_{m-1},\ldots,S_0)$ also are "all ones" after the "end-around" addition of the "overflow bits".

4. The Byte-Serial Additive Inverse Algorithm

The additive inverse of an operand X is formed by obtaining the complement of X. Either "one's" or "two's" complements can be employed; the specifics are discussed in [AVIZ 71a] and [AVIZ 73].

The purpose of this section is to develop the corresponding complementation algorithm for the inverse line-residue X^b. When the "one's" complement $\bar{X}$ of X is formed, the "count of ones" in each byte X_i of X is :

$$\sum_{j=0}^{b-1} \bar{X}_i^j = b - \sum_{j=o}^{b-1} X_i^j$$

This leads to the relationship for $\sum(\bar{X})$:

$$\sum(\bar{X}) = \sum_{i=0}^{k} \left(b - \sum_{j=0}^{b-1} X_i^j\right) 2^i = b(2^{k+1}-1) - \sum(X)$$

Taking the line-residue modulo $(2^{k+1}-1)$, we get:

$$(2^{k+1}-1) \mid \sum(\bar{X}) =$$

$$=(2^{k+1}-1) \mid \{0-[(2^{k+1}-1) \mid \sum(X)]\}=$$

$$=(2^{k+1}-1) - (2^{k+1}-1) \mid \sum(X)$$

The relationship demonstrates that the line-residue of the "one's" complement $\bar{X}$ is obtained by taking the one's complement $(2^{k+1}-1)-[(2^{k+1}-1) \mid \sum(X)]$ of the line-residue $(2^{k+1}-1) \mid \sum(X)$ that was computed for the operand X. The same argument follows for the inverse line-residue.

If the "two's" complement X^* of X is to be formed, it is considered to be:

$$X^* = \bar{X}+2^0$$

In order to get the line-residue of X^*, the line-residue of 2^0 must be added modulo $2^{k+1}-1$ to the line-residue of X. For inverse line-residue encoding, this means the addition of $C_0^0=1$, as described in the next section.

5. The Byte-Serial Addition Algorithm

We consider the byte-serial addition of two operands X and Y, each kb bits long, to get the sum $Z=X+Y$. The addition is modulo $2^{kb}-1$ ("one's" complement), or modulo 2^{kb} ("two's" complement).

The check byte Z_k is obtained by adding the check bytes X_k and Y_k modulo 2^b-1. If "two's" complement is used, a "correction signal" input needs to be used, as defined in [AVIZ 73].

An algorithm to generate the inverse line-residue for Z from the inverse line-residues of X and Y is developed here. As first developed by Garner [GARN 58], the carries generated during the addition of X and Y need to be employed in the calculation of the inverse line-residue for Z.

The "count of ones" (designated by $\alpha(Z_i)$) in each byte Z_i $(0\leq i\leq k-1)$ of Z is:

$$\sum_{j=0}^{b-1} Z_i^j = \sum_{j=0}^{b-1} (X_i^j+Y_i^j) - \left(\sum_{j=1}^{b} C_i^j\right) - 2C_i^b + C_i^0 \ ;$$

$$or\!:\ \alpha(Z_i) = \alpha(X_i)+\alpha(Y_i)-\alpha(\mathit{internal}\ C_i)-2C_i^b + C_i^0 \ ;$$

where C_i^j is the carry *into* the j -th position of the sum byte Z_i, and $C_{i+1}^0 = C_i^b$ for $0\leq i\leq k-2$. For "one's" complement addition of X and Y, we also have $C_{k-1}^b = C_0^0$.

The above leads to an expression for $\sum(Z)$ when

$$\sum(Z) = \sum_{i=0}^{k-1} \alpha(Z_i)2^i + \alpha(Z_k)2^k \quad ;$$

or

$$\sum(Z)= \sum{}^*(X)+\sum{}^*(Y)-\sum{}^*(\mathit{internal}\ C)-2^kC_{k-1}^b+$$

$$+C_0^0+2^k[\alpha(X_k)+\alpha(Y_k)-\alpha(C_k)]$$

The count $\alpha(C_k)$ is the *total* count of carries C_k^j for $1\leq j\leq b$, since $C_k^0 = C_k^b$ in the modulo 2^b-1 addition of the check bytes X_k and Y_k ; i.e.,:

$$\alpha(C_k) = \sum_{j=1}^{b} C_k^j = \alpha(\mathit{int}\ C_k)+C_k^b$$

Two cases need to be discussed separately:

(a) "One's" complement (modulo $2^{kb}-1$) addition of X and Y ;

(b) "Two's" complement (modulo 2^{kb}) addition of X and Y.

For "one's" complement, $C_{k-1}^b=C_0^0$ is the "end-around-carry", and the expression for $\sum(Z)$ is :

$$\sum(Z)=\sum{}'(X)+\sum{}'(Y)-\sum{}'(int.\ C)-2^k C^b_{k-1}+C^b_{k-1}+$$
$$+2^k\alpha(X_k)+2^k\alpha(Y_k)-2^k\alpha(int\ C_k)-2^k C^b_k$$

The expression reduces to:

$$\sum(Z)=\sum(X)+\sum(Y)-[\sum(int\ C)+2^k(C^b_k+C^b_{k-1})-C^b_{k-1}]$$

Taking the line-residue modulo $A=2^{k+1}-1$ of $\sum(Z)$, we get:

$$A|\sum(Z)=A|\{A|\sum(X)+$$
$$+A|\sum(Y)-A|[\sum(int.\ C)+2^k(C^b_k+C^b_{k-1})-C^b_{k-1}]\}$$

This relationship shows that the line-residue of Z can be predicted from the line-residues of X and Y, as well as a line-residue computed from the internal carries formed during the summation of X and Y. The two "end-around" carries C^b_{k-1} and C^b_k also need to be included in the calculation.

Common-mode errors can occur if the carries are incorrectly determined. To avoid such errors, *separate* and *independent* carry-forming circuits need to be employed to form the carries for line-residue determination.

For two's complement, the "correction signal" C^b_{k-1} must be added (modulo 2^b-1 to the modulo 2^b-1 sum of inverse residue check bytes X_k and Y_k [AVIZ 73].

The expression for $\sum(Z)$ is now:

$$\sum(Z)=\sum{}^*(X)+\sum{}^*(Y)-\sum{}^*(int.\ C)-2^k C^b_{k-1}+C^0_0+$$
$$+2^k\alpha(X_k)+2^k\alpha(Y_k)-2^k\alpha(int\ C_k)-2^k C^b_k+2^k C^b_{k-1}$$

The expression is reduced to:

$$\sum(Z)=\sum(X)+\sum(Y)-[\sum(int\ C)+2^k C^b_k-C^0_0]$$

where $C^0_0=1$ only exists if one of the two operands is being complemented (in "two's" complement) simultaneously with the addition. Once again, we take the line-residue modulo $A=2^{k+1}-1$ of $\sum(Z)$ as follows:

$$A\ |\sum(Z)=$$
$$=A\ |\{A\ |\sum(X)+A|\sum(Y)-A\ |[\sum(int.\ C)+2^k C^b_k-C^0_0]\}$$

The difference between "two's" and "one's" complement cases is quite small with respect to computing the line-residue $(2^{k+1}-1)\ |\sum(Z)$.

In practical implementation of byte-serial arithmetic the "two's" complement addition (and subtraction) is strongly preferable because there is no "end-around-carry" that requires either a second addition or the generation of two "tentative" sums - with and without the end-around-carry.

6. Detection of Unidirectional and Bidirectional Errors

In this and the following sections 7 and 8, the error-detecting and error-correcting properties of the two-dimensional codes that were first presented in [AVIZ 83] are reviewed, illustrated, and extended to two and three adjacent lines.

Given a modulo 2^b-1 inverse residue code, the undetectable unidirectional errors are those that have *error values E* congruent to zero modulo 2^b-1, where

$$E=\sum_{j=0}^{b-1}\left(\sum_{i=0}^{k}E^j_i\right)2^j$$

All other unidirectional errors will be detected; however, there are no error correction properties.

One bit-line determinate ("stuck line") faults that cause unidirectional errors will always be detected as long as the condition:

$$(k+1)<(2^b-1)$$

is satisfied. For *two* adjacent "stuck lines," the condition is:

$$3(k+1)<(2^b-1)$$

For m adjacent "stuck lines," the condition is:

$$(2^m-1)(k+1)<(2^b-1)$$

For the purpose of this discussion, lines 0 and $b-1$ are considered adjacent.

The *PM* (pattern miss) [AVIZ 81] percentages for "stuck line" faults remain very low after the left side of the inequalities above exceeds the limit that guarantees *PM* percentage of 0%. For example, for one "stuck line," when $k+1=2^b-1$ is reached, we have:

$$PM(\text{Inv. Residue})=100/(2^{k+1})$$

since only one of the 2^{k+1} possible error patterns on the "stuck line" (all zeros $\rightarrow$ all ones, or vice versa) goes undetected. The situation is not as favorable with "stuck byte" faults, as discussed next.

There is one undetectable one-byte unidirectional error; it results when an all-zero byte X_i is changed to an all-ones byte, or vice versa. The *PM* percentage for this "stuck byte" fault is $(100/2^b)$%. Introduction of byte parity bits will detect only one of the two (stuck-on-one and stuck-on-zero) "stuck bytes"; the other one remains undetectable.

The "stuck byte" detection problem is fully solved by the use of two-dimensional inverse residue encoding. There is one additional check bit X^b_i for each byte X_i $(i=0,\ldots,k)$. The check bits $(X^b_k,\ldots,X^b_0)$ represent the modulo $2^{k+1}-1$ inverse line-residue Y^* of the operand X that is now interpreted as b lines X^j ($j=0, \ldots, b-1$) of $k+1$ bits length each. It is evident that every "stuck byte" now will be detected by the use of Y^* as long as the condition:

$$(b+1) < (2^{k+1}-1)$$

is satisfied. For two adjacent "stuck bytes," the condition is:

$$3(b+1) < (2^{k+1}-1);$$

for p adjacent "stuck bytes" it is:

$$(2^p-1)(b+1) < (2^{k+1}-1)$$

The bytes X_0 and X_k are considered adjacent in this analysis.

The two-dimensional inverse residue is clearly superior to the byte-parity encoding, since the "stuck byte" condition subsumes all other possible error patterns (double, quadruple, etc.) in the byte, while all "even error" patterns go undetected when byte parity is the only form of encoding.

In general, the remaining undetectable errors in the operand X are those that are missed by *both* checks: modulo 2^b-1 over the bytes (not including the check line bits X_i^b), and modulo $2^{k+1}-1$ over the lines, with the check byte bits X_k^j included in each line j. Most unidirectional errors are detectable; furthermore, the detection of bidirectional errors is significantly improved, as discussed below.

It has been noted that low-cost inverse residue codes are considerably less effective in detecting bidirectional errors due to indeterminate repeated-use faults [AVIZ 71a]. The addition of the line-residue (i.e., the second dimension of encoding) allows the detection of *all* bidirectional errors that affect a single line, as well as *all* bidirectional double errors affecting any two bits of the operand X. The double, quadruple, and other even "half-and-half" bidirectional errors on one line that were undetected by the byte check are now detected by the line check, while those in one byte are detected by the byte check.

The remaining undetectable bidirectional errors are those that are simultaneously undetectable by the byte check and the line check. An illustration is the quadruple error that changes Z to Z^* as shown below:

$$Z = \begin{matrix}0\,1\\1\,0\end{matrix} => Z^* = \begin{matrix}1\,0\\0\,1\end{matrix}$$

Here an even number of opposite-direction changes occurs simultaneously in the bytes and lines of the operand X. In general, all quadruple errors of this type (at four corners of a rectangle of bits within the operand X) are undetectable.

7. Correction of Single-Bit and Unidirectional Single-Line Errors

The introduction of the inverse line-residue Y^* also makes single-bit error correction possible. As shown in [AVIZ 71a], the low-cost inverse residue codes have the "partial error location" property. Therefore a single-bit error value $E_i^j=\pm 1$ ($0 \le j \le b-1$; $0 \le i \le k$) will produce a unique indication for line j in the modulo 2^b-1 check and for the byte i in the modulo $2^{k+1}-1$ check, making a correction of E_i^j possible in the operand X. The single-bit error $E_i^b=\pm 1$ that occurs in the check line b will produce the indication for byte i ($0 \le i \le k$) in the modulo $2^{k+1}-1$ check, but *no* error indication at all in the modulo 2^b-1 check, since it does not include the bits of the check line. Correction of E_i^b is therefore possible.

The correction property can be extended to most unidirectional single-line errors as follows. If we assume a determinate single-line fault on line j, the error values $E(j)$ will fall into the range:

$$-2^j\sum_{i=0}^{k} E_i^j \le E(j) \le 2^j\sum_{i=0}^{k} E_i^j$$

The positive values will be due to a stuck-on-one (s-o-1) and negative values — due to a stuck-on-zero (s-o-0). The actual byte check results will assume the values $C(j)=(2^b-1) \mid E(j)$, and as long as $(k+1)<(2^b-1)$ holds, all error values due to a s-o-1 fault will be detectable and have a unique byte check result $C(j)$ in the range

$$0 \le C_1(j) \le (2^b-1) \mid (k+1)2^j$$

Similarly, the error values due to a s-o-0 fault will have the byte check result in the range:

$$0 \le C_0(j) \le (2^b-1) \mid (-2^j)(k+1)$$

However, many other error patterns (on two or more lines) can produce the same values of check results, and error correction is not possible with the byte residue encoding alone.

To obtain single-line unidirectional error correction, we use the additional information provided by the line check result obtained from the inverse line-residue encoding. Given a byte check result $C_1(j)$ discussed above, we find its value to be N, represented by b bits $(N_{b-1}, \ldots, N_0)$.

First we form the hypothesis that N is due to a single-line stuck-on-one determinate fault on line j ($0 \le j \le b-1$). If the fault is on line $j=0$, then $N(0)=N$ error bits $E_i^0=1$ in line 0 will produce the byte check result N. We determine the numbers $N(j)$ of error bits $E_i^j=1$ on lines $j=1, \ldots, b-1$ respectively that would be needed to produce the byte check result N by end-around shifting N to the right $b-1$ times. The shifts will produce the numbers $N(1), \ldots, N(b-1)$ in succession.

The number of error bits $E_i^j = \bar{1}$ (due to a stuck-on-zero line) that would be needed to produce $N(j)$ for any $0 \le j \le b-1$ is given by $(2^b-1)-N(j)$, that is, the "one's complement" of $N(j)$. All values of $N(j)$ and $(2^b-1)-N(j)$ that are greater than $k+1$ are discarded as impossible solutions.

To test the hypothesis that a given byte check result N is due to a single-line determinate fault, we use the line check result

$$R = (2^{k+1}-1) \mid \sum_{j=0}^{b} \left(\sum_{i=0}^{k} X_i^j 2^i \right)$$

This result will contain $N(j)$ digits $R_i = 1$ $(0 \le i \le k+1)$ if there is a single-line determinate (stuck-on-one) fault in the line j. The presence of each $R_i = 1$ indicates that the digits X_i^j should be corrected by the $1 \to 0$ change.

The line check result R will contain $(2^b-1)-N(j)$ digits $R_i = 0$ $(0 \le i \le k+1)$ if there is a single-line determinate (stuck-on-zero) fault in the line j. The presence of each $R_i = 0$ indicates that the digit X_i^j should be corrected by the $0 \to 1$ change.

Example 1: Line Correction

Consider an operand X with seven bytes $(k=7)$ of 4 bits each $(b=4)$. Inverse-residue coding is used for the bytes (modulo $2^b-1=15$) and for the lines (modulo $2^{k+1}-1=255$). The encoded operand (following Figure 1) is shown below:

check line	line 3	line 2	line 1	line 0	
1	0	1	0	0	byte 0
0	0	0	1	1	byte 1
0	1	0	0	1	byte 2
0	0	0	0	0	byte 3
0	1	0	1	1	byte 4
1	0	0	1	0	byte 5
0	1	0	1	0	byte 6
0	0	1	1	0	check byte 7

The byte check result (modulo 15) is $N=1111$, and the line check result (modulo 255) is $R=11111111$. No errors are indicated.

Now assume a stuck-on-one line 2 and set all digits in line 2 to one. The new byte check result is $N=1001$. The single-line determinate fault possibilities are:

Stuck-on-One	*Stuck-on-Zero*
$N(0) = 1001 = 9$	$15-N(0) = 6$
$N(1) = 1100 = 12$	$15-N(1) = 3$
$N(2) = 0110 = 6$	$15-N(2) = 9$
$N(3) = 0011 = 3$	$15-N(3) = 12$

The values greater than $k+1=8$ are discarded, and the remaining possibilities are: line 2 (6 errors) or line 3 (3 errors) stuck-on-one, and line 0 (6 errors) or line 1 (3 errors) stuck-on-zero.

The new line check result is

$$R = (R_7, \ldots, R_0) = 01111110$$

The six ones in R indicate that the "line 2 stuck-on-one" hypothesis is valid, and the corresponding six positions in line 2 are corrected by setting them to zero.

The single-line, unidirectional error correction algorithm can not be completed only in the cases in which two conditions occur simultaneously:

(a) More than one line is *indicated* by the occurrence of identical values of $N(j)$ or of $15-N(j)$ for two or more lines j of X.

(b) The correction pattern indicated by the line check result R is actually *applicable* to more than one line j of the operand X, i.e., the lines have all zeros (or all ones) in the positions to be corrected.

Example 2 below illustrated condition (2); the subsequent discussion deals with condition (b).

Example 2: Correction Ambiguity

Now assume that line 1 is stuck-on-zero. The byte check result is $N=0101$, and the possibilities are:

Stuck-on-One	*Stuck-on-Zero*
$N(0) = 0101 = 5$	$15-N(0) = 10$
$N(1) = 1010 = 10$	$15-N(1) = 5$
$N(2) = 0101 = 5$	$15-N(2) = 10$
$N(3) = 1010 = 10$	$15-N(3) = 5$

The remaining possibilities all point to five errors. The modulo 255 line check result is

$$R = 00001101$$

The five zeros in R (positions 7,6,5,4,1) indicate a stuck-on-zero on line 1 or line 3. To resolve the ambiguity, we find that line 3 already has "1" digits in positions 6 and 4, and cannot be corrected there; therefore the stuck line must be line 1.

It is possible that both potential corrections could be carried out in Example 2 above; that is, both line 1 and line 3 could have zeros in positions 7,6,5,4,1. In such a case, the error has been detected, but a correction is not possible, since both conditions (a) and (b) occur simultaneously.

8. Two-Line and Three-Line Errors

A more critical case than the ambiguity discussed above would be that of a mis-correction, in which the restored pattern would differ from the original one, such as in the case of triple errors encountered by the Hamming SEC/DED code.

A mis-correction for two-dimensional inverse residue codes will occur if the bit pattern of the operand X changes in more than one line, but *both* the byte check result value N and the line check result value R remain the same as for a single-line error. This will happen when:

(a) the byte check result is altered by $\pm c(2^b-1)$

(b) the line check result is altered by $\pm c(2^{k+1}-1)$

(c) both (a) and (b) occur simultaneously.

In cases (a) and (b) the other check result remains unchanged.

It is readily shown that a mis-correction cannot occur if only *two* adjacent lines (*or* bytes) are affected by the fault; the detection is guaranteed in all cases. When *three* adjacent lines (or bytes) are affected, a mis-correction can occur. The byte check result will be altered by $\pm(2^b-1)$ when the following changes are imposed on a correctable unidirectional single-line error pattern:

(a) two error bits from line j are moved one line to the right, causing a net change in N of
$$2(2^{b-1})-2 = 2^b-2;$$

(b) one error bit from line j is moved one line to the left, causing a net change in N of $2-1=1$.

The total change in N is then $(2^b-2)+1=2^b-1$, and it will lead to a mis-correction if the following two conditions are satisfied:

(1) there are no further error changes, and

(2) the positions in line j that would be mis-corrected actually *do* contain correctable bit values.

An example of the conditions under which a mis-correction will occur is shown in Example 3 below.

Example 3: Conditions for Mis-Correction

Consider the encoded operand below (same format as in Example 1). Without changes, both N=1111 and R=11111111 are obtained.

check line	line 3	line 2	line 1	line 0	
1	0	1	0	0	byte 0
1	1 Ø	1	0	0	byte 1
1	1	1	1 Ø	1	byte 2
0	0	1	1 Ø	0	byte 3
1	1	1 Ø	1	1	byte 4
0	0	1 Ø	1	0	byte 5
0	1	1 Ø	1	0	byte 6
0	1	1	0	0	check byte 7

The unidirectional ($0 \rightarrow 1$) errors affect three adjacent lines (3,2,1) as shown, and impose exactly *six* changes. Now we get N=1001 and R=0111110. This is *exactly* the same condition as in Example 1, and "line 2 stuck on one" hypothesis is validated, since bytes 1 through 6 contain ones in line 2. Setting those six bits to zero will cause a mis-correction.

9. Conclusions

It is concluded that the two-dimensional codes are very nearly 100% (except in the cases of ambiguity as illustrated in Example 2) single-line correcting, and full 100% double-adjacent-line detecting codes with respect to unidirectional errors. The probability of mis-correction in the case of three-adjacent-line unidirectional errors remains very low, since a very specific error pattern and original pattern of X must coincide to cause a mis-correction.

It has been shown that byte-serial arithmetic can be carried out with operands which are encoded in two-dimensional residue and inverse-residue codes. Two-dimensional encodings provide a very powerful error-detecting and a substantial error-correcting capability for byte-serial arithmetic. Promising application areas are systolic arrays, multiple-precision arithmetic, and high-speed array computing.

REFERENCES

[AVIZ 62] Avižienis, A. "On a Flexible Implementation of Digital Computer Arithmetic," *Information Processing 1962*, C.M. Popplewell, ed., North Holland Publishing Co., Amsterdam, 1963, pp. 664-670.

[AVIZ 70] Avižienis, A., Tung, C., "A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors," *IEEE Trans. on Computers,* C-19: 733-745, August 1970.

[AVIZ71a] Avižienis, A. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Trans. on Computers,* C-20: 1322-1331, November 1971.

[AVIZ71b] Avižienis, A., et al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. on Computers,* C-20: 1312-1321, November 1971. Reprinted in *Best Computer Papers of 1971,* L. Petrocelli, ed., Auerbach Publishers, 1972, pp. 165-185.

[AVIZ 73] Avižienis, A. "Arithmetic Algorithms for Error-Coded Operands", *IEEE Trans. on Computers,* C-22: 567-572, June 1973.

[AVIZ 81] Avižienis, A., "Low-Cost Residue and Inverse Residue Error-detecting Codes for Signed-Digit Arithmetic," *Proc. 5th IEEE Symposium on Computer Arithmetic,* 1981, pp. 165-168.

[AVIZ 83] Avižienis, A. and C. S. Raghavendra, "Applications for Arithmetic Error Codes in Large, High-Performance Computers," *Proceedings, 6th IEEE Symposium on Computer Arithmetic,* 1983, pp. 169-173.

[BOSE 80] Bose, B., Rao, T.R.N., "Unidirectional Error Codes for Shift Register Memories", *Digest FTCS-10,* Kyoto, Japan, October 1980, pp. 26-28.

[GARN 58] Garner, H. "Generalized Parity Checking", *IRE Trans. El. Computers,* EC-7: 207-213, September 1958.

[PARH 73] Parhami, B., Avižienis, A., "Application of Arithmetic Error Codes for Checking of Mass Memories," *Digest of the 1973 Int. Symposium on Fault-Tolerant Computing,* pp. 47-51, June 1973.

[PARH 78] Parhami, B., Avižienis, A., "Detection of Storage Errors in Mass Memories Using Low-Cost Arithmetic Codes," *IEEE Trans. on Computers,* C-27-4: 302-308, April 1978.

[TUNG 70] Tung, C., Avižienis, A., "Combinational Arithmetic Systems for the Approximation of Functions," *AFIPS Conf. Proc.* (1970 Spring Joint Computer Conference), 36: 95-107, 1970.

[USAS 78] Usas, A.M., "Checksum Versus Residue Codes for Multiple Error Detection," *Digest of the 8th Annual International Conf. on Fault-Tolerant Computing,* p. 224, 1978.

[WAKE 75] Wakerly, J. F., "Detection of Unidirectional Multiple Errors Using Low-Cost Arithmetic Codes", *IEEE Transactions on Computers,* C-24: 210-212, February 1975.

APPENDIX

Example 4: Line-Residue Checking

The byte-serial line-residue checking algorithm of Section 3 is illustrated below. The operand X is from Example 1, with the "line 2 stuck-on-one" error. Here $m=3$ and $k=8$.

check line	0	0	1	0	0	0	0	1
line 3	0	1	0	1	0	1	0	0
line 2	1	1	1	1	1	1	1	1
line 1	1	1	1	1	0	0	1	0
line 0	0	0	0	1	0	1	1	0

```
            ┌──────────────────────────────→ 0   1   0  ┐
          ┌─┴─┐                                          │ +
Σ(L)=0 1 0      0   1   1   1   1   1   0   0  ┘
     s11 s10 s9             c3  →   0   1   1   0
                                      └─────┬─────┘
Σ(L)'=0 1 0     1   0   0   0   0   s2  s1  s0
```

$\sum(L)' = \sum(L)+2^3$, since $m=3$

Since $C_3=0$, $\sum(L)$ is selected as the check result, with $s_2,s_1,s_0=1\ 1\ 0$