A Fault-Tolerant Permutation Network Modulo Arithmetic Processor

Ming-Bo Lin and A. Yavuz Oruç, *Senior Member, IEEE*

***Abstract*—Conventional fault-tolerant modulo arithmetic processors rely on the properties of a residue number system with L redundant moduli to detect up to $L/2$ errors. In this paper, we propose a new scheme that combines r-out-of-s residue codes with Berger codes to concurrently detect any number of module errors without any redundant moduli. In addition, this scheme can tolerate L faults if L redundant moduli are used, and has the property of graceful degradation when the number of faulty moduli exceeds L. Finally, it is shown that the added cost for fault tolerance is much less than those were reported earlier in the literature.**

***Index Terms*— Concurrent error detection, fault tolerance, graceful degradation, module discard, modulo arithmetic processor, and permutation network.**

I. Introduction

As in other computations, reliable computing is critical in arithmetic processors as well. In particular, when such processors are used in critical applications such as flight navigation, medical analysis, and real time monitors.

In general, two approaches can be used to detect errors in a digital system. The first is off-line testing which requires interrupting the normal operation of the system to diagnose it for faults; the second is on-line error detection which can be carried out during the normal operation of the system. Off-line testing is effective for detection of hard (i.e, permanent) or long duration circuit faults only, while on-line testing (usually called concurrent error detection, or CED for short) can detect transient faults (or called soft) faults, which are predominant in modern digital systems.

Fault detection and correction techniques fall into three major categories. The first category deals with redundant residue number system (RRNS) which has a number of redundant moduli [19], [32]. The second category exploits the arithmetic codes [28]. The third category uses a discarding policy.

Most efforts to date in the first category rely on the fact that if the proper redundant moduli are included in the residue number system (RNS) code, then the special algebraic structure of the RNS allows the computational errors to be detected and corrected [2], [7], [11], [19], [32], [33]. In general, an RRNS with L redundant moduli can detect up to L residue digit errors or correct up to $\lceil \frac{L}{2} \rceil$ residue digit errors [19]. However, the hardware required for detecting and correcting the error is very complicated. In addition, these schemes are implemented by ROM lookup tables. This is ineffective when the underlying modulo arithmetic unit is not constructed by using ROM lookup tables such as those proposed by these authors [15], [16].

The second category for error detection and correction is based on arithmetic codes which have two varieties: AN codes [3], [19], where A is the generator of the code and N is the information represented, and residue codes [28]. Several totally self-checking error detection circuits for low-cost AN codes with generator $A = 2^r - 1$ have been suggested [8], [22], [23]. However, no effective error detection and correction procedure has been reported.

The residue code with check base A is the code which attaches to an arithmetic value X a check value X' where $X' = X \bmod A$, that is, it forms (X, X') pair. The essence of residue codes is the fact that they preserve their properties in arithmetic operations: addition, subtraction, and multiplication. That is, let N_1 and N_2 be two positive integers and R_1 and R_2 be their respective residues, then $(N_1 \pm N_2) \bmod A = (R_1 \pm R_2) \bmod A$ and $(N_1 N_2) \bmod A = (R_1 R_2) \bmod A$. A sufficient and necessary condition for using residue codes to detect and correct errors in a digital system has been given in [10]. Several residue generators have been reported in the literature [5], [24].

The third category of error detection and correction is based on discarding of the faulty modulus [27]. The rational behind this is that if the faulty modulus is known then after the residue digit represented by the faulty modulus is removed the remaining residue digits still represent a legitimate number. Therefore, the redundancy can be reduced considerably. In general, one can correct up to L digits in a system with L redundant moduli. Using this idea, Taheri *et al.* [30] designed an RNS processor with the capability of distributed fault detection. However, their design requires a complicated and expensive combination of residue decoders and their error detection mechanism is based on the parity code which is evidently insufficient for VLSI implementations [1], [26].

The error detection and correction schemes mentioned above except the arithmetic code method are founded on the redundant residue number system and they use ROM lookup tables throughout the entire system. The error detection and correction circuits for these schemes are very complicated and costly in both time and hardware [3], [7], [11], [19]. Once an error is located, the correction proceeds in an unchecking way although self-checking is used during the course of error

Manuscript received June 14, 1993; revised November 11, 1993 and February 24, 1994.

M.-B. Lin is with the Electronic Engineering Department, National Taiwan Institute of Technology, Taipei, Taiwan.

A. Y. Oruç is with the Electrical Engineering Department, Institute of Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA.

IEEE Log Number 9403698

detection [11], [12]. To reduce the hardware cost, an algorithm that combines the operations of scaling and single residue error correction into one circuit were proposed in [29]. Although this reduces the hardware cost of the scaling and error correction circuit by using the same mixed radix conversion circuit, the entire circuit, which is called an error correction circuit with scaling (EECS), is not self-checking. Therefore, the reliability of the circuit is in question. In most modulo arithmetic processors, the checkers and error correction circuits can no longer be assumed to be error free because checker circuits are constructed from the same components as the circuits that perform the arithmetic operations and hence are subject to the same types of failures [28]. Thus, the checkers and the error correction circuits must be able to detect and indicate this fact when errors occur inside these circuits.

In this paper, we improve the aforementioned results by introducing self-checking Berger code checkers into an arithmetic processor which was described in [15]. This new approach exploits the r-out-of-s code representation of the perands of an arithmetic processor to distribute the error detection into each modulus m_i. As such, it can be viewed as a "discard the faulty modulus" approach. The result is a simple modulo arithmetic processor that can detect any number of module errors without any redundant moduli. Furthermore, it can tolerate up to L faults if L redundant moduli are used. The remainder of this paper is organized as follows. Section II describes a cyclic permutation network and establishes its fault detection property. Sections III shows how to design a CED arithmetic processor using cyclic permutation networks. Section IV extends this CED arithmetic processor to obtain a fault-tolerant arithmetic processor and the paper is concluded in Section V.

II. Cyclic Permutation Network

In this section, we examine the effects of most common faults in NMOS circuits [1], [18] on cyclic permutation networks which will be used in Sections III and IV to construct a CED arithmetic processor and a fault-tolerant arithmetic processor.

We first recall the definition of an r-out-of-s residue code.

Definition 1: (*r-out-of-s residue code*) Let $m_1, m_2, \ldots, m_r$ be pairwise relatively prime numbers and $s = m_1 + m_2 + \cdots + m_r$. Then an r-out-of-s residue code is defined to be a code whose codewords are of length s and have r partitions each containing exactly one "1" . That is, an r-out-of-s residue codeword is a concatenation of r 1-out-of-m_i codewords for $1 \leq i \leq r$.

From this definition, it is easy to see that 1-out-of-m_i code is a special case of r-out-of-s residue code with $r = 1$ and $s = m_i$.

In general, a cyclic permutation network consists of three function blocks: a switching network, an input encoder, and some output buffers. The input encoder converts the binary input operand Y into its equivalent two-rail binary form to control the switches in the switching network. The switching network performs the required permutation operation on its r-out-of-s residue coded operand. . The output buffers provide

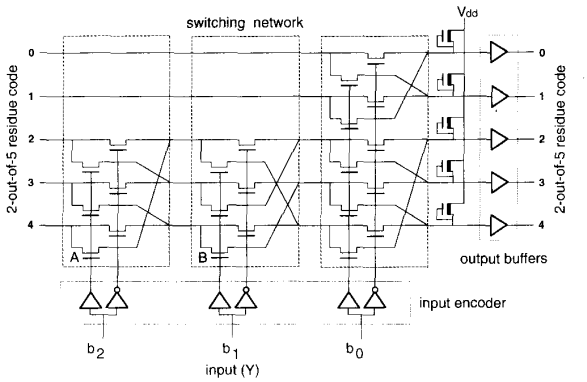


Fig. 1. A binary to 2-out-of-5 reside encoder on a cyclic permutation network.

the interface between two cyclic permutation networks. These buffers are not necessary for the operation of the cyclic permutation network but included to restore the signal strength and hence to reduce the delay [21]. The cyclic permutation networks in our arithmetic processors are implemented using NMOS switches or CMOS transmission gates. Fig. 1 shows how a binary to 2-out-of-5 residue encoder is implemented on a cyclic permutation network using NMOS switches.

We consider only transistor failures in our MOS circuits. Two common faults of MOS transistors are stuck-open and stuck-on faults [1].

Definition 2: (*uniform fault*) The faults in a circuit are called uniform if they all cause the transistors to be stuck-open or stuck-on but not both.

Definition 3: (*unidirectional error*) The errors in a circuit are called unidirectional if either all the errors in any codeword are from 0 to 1 or they are from 1 to 0, but not both.

To illustrate how a unidirectional error can occur on the network in Fig. 1, suppose that the network receives 01100 on its left inputs, and $b_2 = 1, b_1 = 1$, and $b_0 = 0$. The network will produce the codeword 01100 if all transistors are fault-free and no two lines are short-circuited. On the other hand, if both transistors A and B are stuck-open then the codeword changes to 01111. This is an example of a unidirectional error.

Definition 4: A path from an input to an output is said to be active if it is a signal propagation path under a given switching state.

Theorem 1: Assume that all output lines of the input encoder of the cyclic permutation network under consideration are error free. The errors caused by any uniform faults in the switching stages of a cyclic permutation network are unidirectional.

Proof: Assume that all pull-up transistors are fault free. Any transistor with stuck-open fault will disconnect the active path containing the faulty transistor from some input to its output. Therefore, the outputs will be always 1's due to using pull-up transistors. This causes a unidirectional error at the output of the cyclic permutation network. Similarly, any transistor with stuck-on fault might cause multiple paths. Some of these may combine both "0"-signal and "1"-signal. If such a case occurs, the resulting signal is justifiably assumed

to be "0" [21]. Therefore, the stuck-on transistor will absorb the "1"-output and causes a undirectional error at the output of the cyclic permutation network. ||

The following theorem is a corollary to Theorem 1.

Theorem 2: Any uniform faults in the switching stages of a cyclic permutation network can be detected by an inspection of the codewords at its outputs.

Proof: By the previous theorem, the errors caused by any uniform faults are unidirectional. This means the codeword at the output of the cyclic permutation network in question will have either none or multiple occurrences of 1's. But this invalidates the codeword as an r-out-of-s residue code and hence can be detected. ||

The following theorem extends this result to uniform faults at the output lines of the input encoder of a cyclic permutation network.

Theorem 3: Any uniform faults at the input encoder of a cyclic permutation network can be detected by an inspection of the codewords at its outputs.

Proof: Any stuck-on or stuck-open fault inside the input encoder of a cyclic permutation network will cause its output to be stuck-at-1, stuck-at-0, or floating. These will in turn cause the transistors in the switching stages to be stuck-on or stuck-open faults. If the result is stuck-on, multiple copies of 1's will propagate to its outputs for at least one code input. On the other hand, if the result is stuck-open fault, it will block the transmission of any 1's to its outputs for at least one code input. Both these faults, when they do not occur simultaneously, invalidate the r-out-of-s residue code at the output end of the cyclic permutation network, and therefore can be detected. ||

Theorem 4: The errors caused by any single stuck-at fault or line-open faults at the input lines of the input encoder of a cyclic permutation network cannot be detected by an inspection of codewords at its outputs.

Proof: Since no stuck-at fault or line-open fault at the input lines of the input encoder of a cyclic permutation network can set both output lines of the input encoder to the same values (1 or 0), it can never invalidate an r-out-of-s residue coded codeword, and hence cannot be detected by an inspection of codewords at the outputs of the network. ||

One way to detect the faults at the input lines of the input encoder of a cyclic permutation network is to use two-rail code. However, it is not easy to design a self-checking r-out-of-s residue code to two-rail code conversion circuit. A better alternative is to encode the inputs of the input encoder by a Berger code [4] and use a self-checking Berger code checker [17], [20], [25].

Definition 5 (Berger code): A Berger code is a binary block code consisting of $(k + \lceil \lg(k+1) \rceil)$-tuples, $(b_0, b_1, \ldots, b_{k-1}, g_0, g_1, \ldots, g_{\lceil \lg(k+1) \rceil - 1})$, where $(b_0, b_1, \ldots, b_{k-1})$ is the information part and $(g_0, g_1, \ldots, g_{\lceil \lg(k+1) \rceil - 1})$ is the check part (Berger check) which represents the number of 0's in the information part.

An example of Berger code encoded 1-out-of-5 binary residue decoder and its function are shown in Fig. 2 and Table I, respectively. Any of the inputs on the left, upon receiving a 1, causes the row of transistors it feeds to conduct and this forces the outputs to which the transistors are connected to be grounded. For example, if the third input is 1 then b_2, b_0, and g_0 are grounded (0) and b_1 and g_1 remain connected to $V_{dd}(1)$.

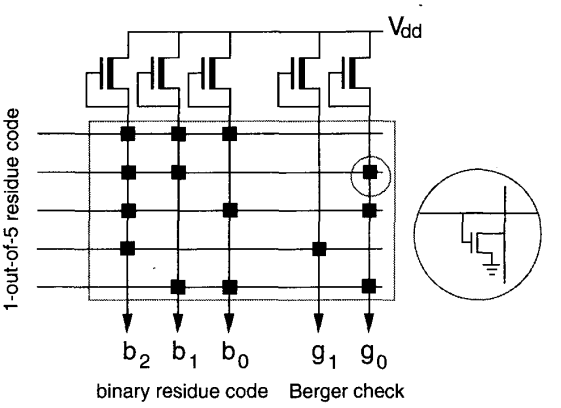


Fig. 2. Circuit diagram. Berger code encoded 1-out-of-5 binary residue decoder.

TABLE I
TRUTH TABLE

input	output b_2	b_1	b_0	g_1	g_0
0	0	0	0	1	1
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0

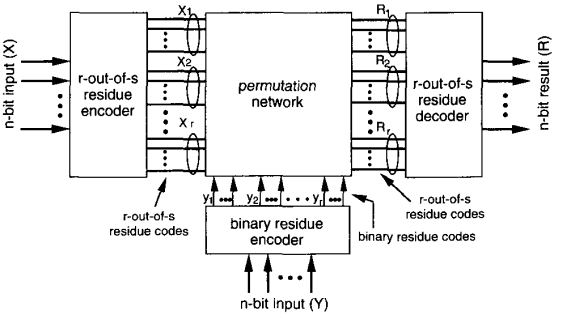


Fig. 3. A cyclic permutaion network airthmetic processor.

The Berger code encoded binary residue decoder not only can detect the errors caused by the faults inside the decoder but also can detect the errors at the input lines of the decoder [18].

It follows that with the addition of a Berger code encoded 1-out-of-m_i binary residue decoder, our cyclic permutation networks can be made to detect any uniform faults including those at the inputs of the input encoder.

III. A CED Modulo Arithmetic Processor

The general structure of cyclic permutation network modulo arithmetic processor is shown in Fig. 3. It consists of four major parts: 1) two inputs r-out-of-s residue encoders for operands X and Y; 2) a cyclic permutation network; and 3)

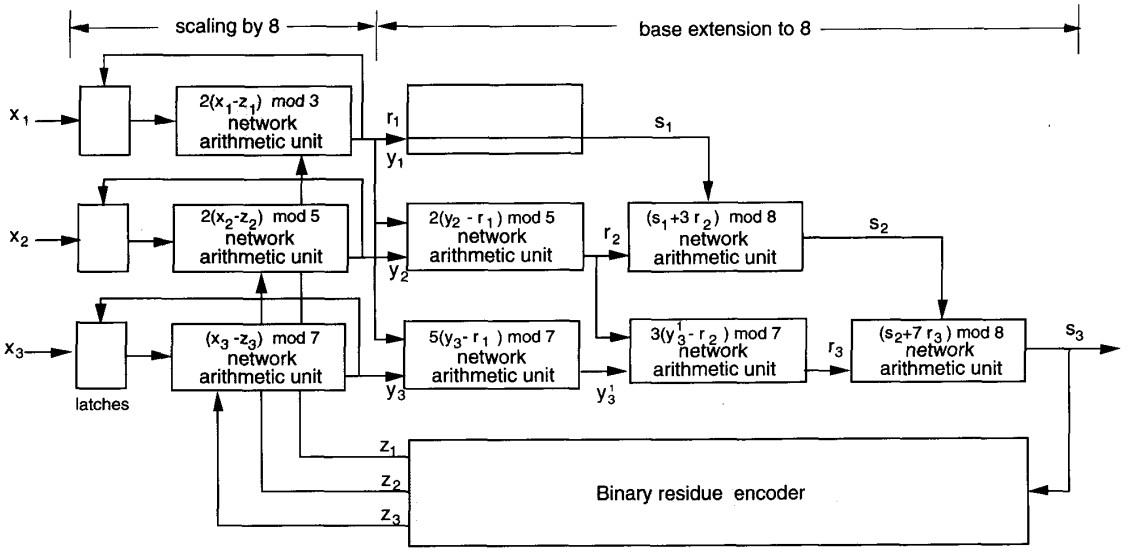


Fig. 4. 3-out-of-15 residue decoder on a cyclic permutation network; $r = 3$, $t = 3$, $m_1 = 3$, $m_2 = 5$, $m_3 = 7$.

an output r-out-of-s residue decoder. The operation of this processor was described in detail in [15].

All four parts are constructed from cyclic permutation networks. The outputs of the binary residue encoder for operand Y are encoded by Berger code and then checked by self-checking Berger code checkers [17], [20]. Each self-checking Berger code checker receives two inputs: binary residue code and Berger check and then generates one pair of complement outputs, i.e., 1-out-of-2 code. When the circuit is fault-free, it outputs $(0,1)$ or $(1,0)$; otherwise it outputs $(1,1)$ or $(0,0)$.

The operation of the r-out-of-s residue decoder is described as follows. First, we carry out the residue decoder by using an extension of Garner's algorithm. Let $(r_1, r_2, \ldots, r_p)$ and $(x_1, x_2, \ldots, x_p)$ be the mixed radix and RNS representations of X with moduli $m_1, m_2, \ldots, m_r$, respectively. Garner's algorithm receives $x_1, x_2, \ldots, x_r$ as input and produces a single output. The algorithm given below is a modified version of Garner's algorithm and computes t bits of the binary output at a time, where t is some positive number between 1 and $\lg M$ and $M = m_1 m_2 \ldots m_r$.

Algorithm (r-out-of-s residue decoder)

Begin

Step 1: Determine constants $c_{i,j}$ satisfying $m_i c_{i,j} = 1 \bmod m_j$, for $1 \le i < j \le r$, and constants $c_{2^t,i}$ satisfying $2^t c_{2^t,i} = 1 \bmod m_i$, for $1 \le i \le r$, where t is a positive integer $\ge \max_{1 \le i \le r} \{\lceil \lg m_i \rceil\}$. This step is carried out off line and is not part of the residue decoder.

Step 2: Let $y_i = x_i$, for all i, $1 \le i \le r$.

Step 3: Base extension to mod 2^t.

3.1 Compute

$$r_1 = y_1 \bmod m_1$$
$$r_2 = (y_2 - r_1)c_{12} \bmod m_2$$
$$r_3 = ((y_3 - r_1)c_{13} - r_2)c_{23} \bmod m_3$$
$$\vdots$$
$$r_r = (\ldots((y_r - r_1)c_{1r} - r_2)c_{2r} - \cdots - r_{r-1})c_{r-1,r} \bmod m_r \quad (1)$$

3.2 Compute

$$S_1 = r_1 \bmod 2^t$$
$$S_2 = S_1 + r_2 m_1 \bmod 2^t$$
$$S_3 = S_2 + r_3 m_1 m_2 \bmod 2^t$$
$$\vdots$$
$$S_p = S_{r-1} + r_r m_1 m_2 \ldots m_{r-1} \bmod 2^t \quad (2)$$

Step 4: Scaling by 2^t.

Compute $y_i = (x_i - z_i)c_{2^t,i}$ and set $x_i = y_i$, for all i, $1 \le i \le r$, where $z_i = S_r \bmod m_i$.

Step 5: Repeat Steps 3 and 4 for $\lceil \frac{\lg M}{t} \rceil - 1$ times.

End

Fig. 4 depicts a network implementation of the residue decoder for $r = 3, m_1 = 3, m_2 = 5, m_3 = 7$, and $t = 3$. The number x (take $(0,2,3) = 87$ as an example) enters the circuit on the left in RNS, and exits it in binary on the right in three iterations; the first iteration computes the least significant three bits (=111), the second iteration computes the next three significant bits (=010), and the last iteration computes the most significant bit (=001).

Because we use r-out-of-s residue codes to encode each cyclic permutation network inside the modulo arithmetic processor, and since those cyclic permutation networks with the same modulus are cascaded straight without any interface using non-r-out-of-s residue codes between them, it suffices to use self-checking Berger code checkers in the last stage of each modulus or where the outputs are converted to binary. Fig. 5 shows an example of CED cyclic permutation network

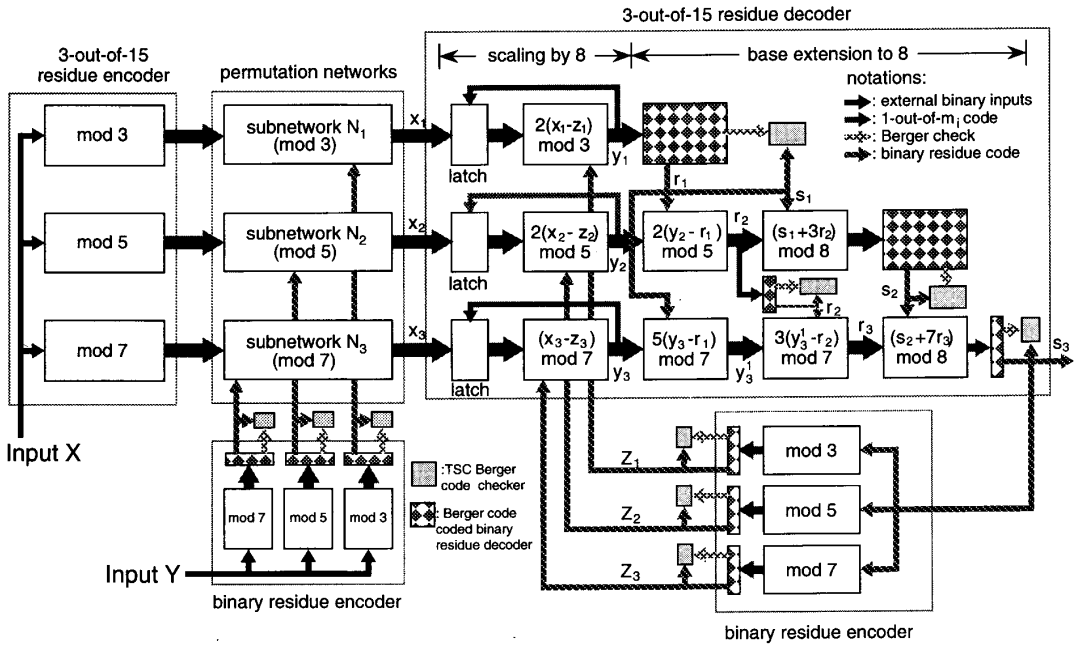


Fig. 5. A cyclic permutation network modulo arithmetic processor with concurrent error detection.

modulo arithmetic processor with three moduli, $m_1 = 3, m_2 = 5, m_3 = 7$.

To collect the faults detected by the self-checking Berger code checkers and produce a final error indication signal, the two-rail outputs of the self-checking Berger code checkers are fed into a two-rail code checker tree [28]. This tree maps m input pairs, $\{(a_0, b_0)(a_1, b_1) \ldots (a_{m-1}, b_{m-1})\}$ to an output pair, (z_0, z_1). The output pair is complementary if and only if each and every input pair is complementary [31]. In general, to reduce the hardware cost, the tree is designed by modular approach, that is, it is divided into h levels. A complete tree with h levels of cells, where each cell is an m-bit two-rail comparator, can be used to compare m^h bits and contains $(m^h-1)/(m-1)$ cells. The value of m determines the number of code inputs required for testing the tree, that is, 2^m, where m is the size of the bit vectors compared by each cell [13], [14]. An example can be found in [28].

IV. A Fault-Tolerant Processor

Fault-tolerance is the ability of a system to continue to perform its functions after the occurrence of faults. In a modulo arithmetic processor, since each module works independently except in the final residue decoder stage where they are combined together, any correction operation must be done in the residue decoder stage.

Most of the earlier efforts on fault-tolerant processor design have focused on the stages before the final residue decoder. These efforts assume that the residue decoder is more reliable than the other circuits and the correction circuits are error-free [6], [7], [12], [29]. However, this is not the case when the required output is a binary number which requires a residue decoder to combine the values from all moduli into one result and the correction circuit is built by the same technology as all the other circuits in the processor. Another shortcoming of these efforts is that they assume overflows and errors could not occur simultaneously [11], [12], [29].

Two approaches are generally used to achieve fault-tolerance in digital systems. One is a fault masking technique that uses redundant circuits which work in parallel and vote for the outputs of the system. In this technique, we do not need to know the exact faulty module. However, we must maintain multiple copies of the system which requires excessive hardware cost.

The second approach is based on reconfiguring the system so as to circumvent the faults. In this approach, three steps are needed: fault detection, fault location, and fault recovery. Most of the published literature on fault-tolerant modulo arithmetic processors belong to this class [12], [29]. The bulk of this technique is spent in fault detection and fault location. Therefore, if the faulty modulus can be located easily then the reconfiguration process is reduced to a simple one.

Taheri *et al.* [30] proposed a bit-sliced ROM based modulo arithmetic processor which can detect the parity error on the basis of each modulus. Once detected, faulty moduli are discarded. The essence behind this is the following theorem.

Definition 6: A redundant residue number system (RRNS) is defined to be a residue number system with L additional moduli. All $L+r$ moduli must be pairwise relatively prime to ensure a unique representation for each number in the system and each redundant modulus must be greater than any modulus m_i; $1 \leq i \leq r$.

Theorem 5 ([30]): An RRNS with one redundant modulus allows correction of one error if the erroneous modulus is discarded.

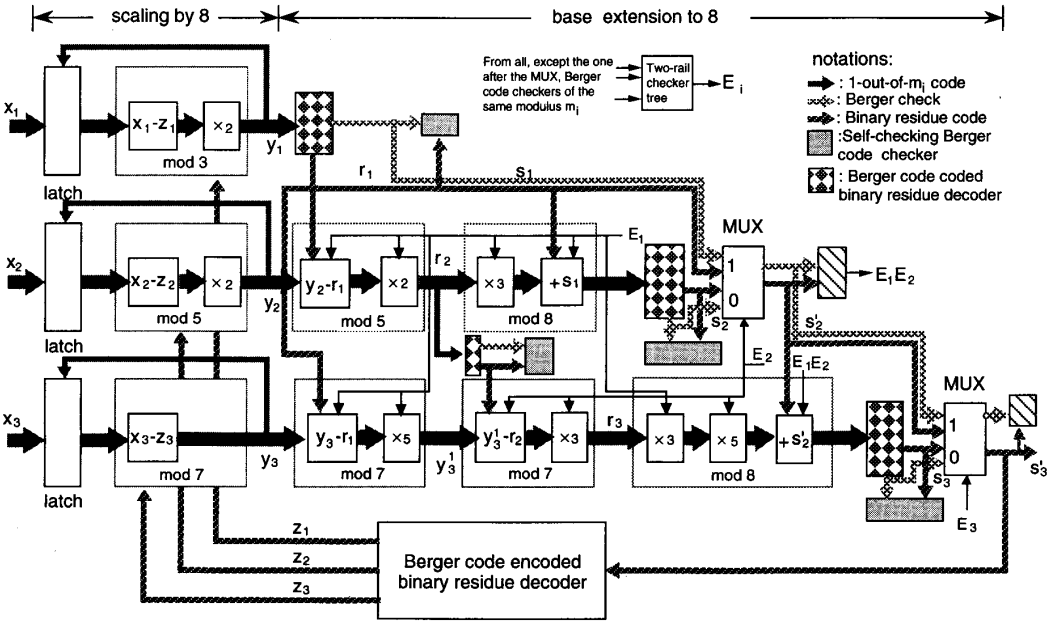


Fig. 6. Residue decoder of a fault-tolerant cyclic permutation network modulo arithmetic processor with $r = 2$, $L = 1$, $m_1 = 3$, $m_2 = 5$ and $m_3 = 7$.

The next theorem then follows immediately.

Theorem 6: An RRNS with L redundant moduli corrects up to L errors if the erroneous moduli are discarded.

Taheri et al. used this result to obtain their fault-tolerant bit-sliced ROM based modulo arithmetic processor. However, their design has two shortcomings. First, their fault model assumes only single errors which does not hold for VLSI circuits [1]. Second, they require $\binom{r+L}{r}$ combinations of residue digits and therefore $\binom{r+L}{r}$ copies of residue decoders to obtain the final correct output according to the error flag. It is clear that this requires too much extra hardware.

In the following, we combine the features of cyclic permutation networks, unidirectional detecting codes (1-out-of-m_i code and Berger code), and self-checking Berger code checkers to propose an approach which can tolerate L faults if L redundant moduli are used. This approach is also based on the principle of discarding faulty modulus.

The general structure of our fault-tolerant modulo arithmetic processor has the same architecture as the CED modulo arithmetic processor described in the previous section. However, some modifications in the residue decoder must be made so that it can be easy to remove a faulty modulus from the system. As shown in Fig. 6, two 2×1 multiplexers are added to the residue decoder. In general, the residue decoder requires $(r-1)$ 2×1 multiplexers. In addition, each modulo arithmetic network is confined into a single operation stage with two states: state 0 is the identity permutation and state 1 is the normal operation. These two states are controlled by error indication signals E_i's.

In order to generate an appropriate set of error indication signals to control the operations of cyclic permutation networks or the 2×1 multiplexers, the cyclic permutation networks that operate on the same modulus have an individual self-checking two-rail checker tree. These cyclic permutation networks constitute a residue digit module with modulus m_i.

Once a faulty module within a residue digit module with modulus m_i is detected, all output is removed from this residue digit module (i.e., is effectively removed this faulty modulus) to prevent data from reaching succeeding stages. This can be done by setting all corresponding arithmetic network stages to their identity permutations. To see this, let us recall the algorithm described in the previous section. In order to remove the effect of residue digit module m_i, it is necessary to set all r_i to 0 and c_{ij} to 1 for $1 \leq j \leq r$ in Step 3.1 and Step 3.2 and set S_i to 0 and m_i to 1 in Step 3.2 of the algorithm. For example, in a modulo arithmetic processor with $r = 3, m_1 = 3, m_2 = 5$, and $m_3 = 7$, if any module within residue digit module with modulus 3 is faulty then we discard this residue digit module by setting $r_1 = 0, c_{12} = 1$, and $c_{13} = 1$ in Step 3.1 and Step 3.2; $S_1 = 0$ and $m_1 = 1$ in Step 3.2. Therefore, if residue code is $(x_1, x_2, x_3) = (2, 3, 1)$, then the result in binary representation is 8. Note that if modulus 7 is a redundant then 8 is within the range spanned by modulus 3 and 5.

The settings $r_i = 0, c_{ij} = 1$, and $m_i = 1$ are easily affected by the cyclic permutation networks. Since the value of r_i is combined into the subtraction; c_{ij} and m_i are combined into multiplication, these settings correspond to the identity permutations. As for S_i, due to the fact that S_i is sent to residue digit module $i + 2$, if residue digit module $i + 1$ is faulty, a 2×1 multiplexer is used to select and send S_i or S_{i+1} into residue digit module $i+2$ circuit to obtain a correct result.

An example of fault-tolerant cyclic permutation network modulo arithmetic processor with $r = 2, m_1 = 3, m_2 = 5, L = 1, m_3 = 7$ is shown in Fig. 6. Each of the control signals, E_i's, comes from the output via an XNOR gate of

the two-rail checker tree of each residue digit module, m_i, for $1 \leq i \leq 3$, are used to control the states of cyclic permutation networks and select the sources of the multiplexers. When $E_i = 1$ (indicating residue digit module m_i is faulty), E_i signal forces the switch states of all cyclic permutation networks which it is connected to identity permutations. This effectively removes the effects of the residue digit module m_i from the system. As indicated in the figure, the number of faulty residue digit modules that system can tolerate is not constrained to 1. In this figure, if another residue digit module, $m_2 = 5$, is also faulty, then after it is removed the system is still workable but with the range spanned by modulus $m_1 = 5$ only. That is to say, the operating range of the system is degraded gracefully from three moduli to one modulus. In general, if L redundant residue digit modules are used then up to L faults can be tolerated in a gracefully degrading fashion until all residue digit modules are faulty.

V. Conclusion

In this paper, we introduced a cyclic permutation network modulo arithmetic processor which combines the features of cyclic permutation networks and unidirectional error detecting codes: 1-out-of-m_i codes and Berger codes. The resulting processor can concurrently detect any number of faulty moduli without any redundant moduli and also has fault-tolerant capability. It can tolerate L faulty moduli without any performance degradation if L redundant moduli are used in the system. In addition, it has the characteristic of graceful degradation when the number of faulty moduli exceeds the number of redundant moduli.

Comparing our design with the work of Taheri *et al.* [30], our design has the following advantages. First of all, in contrast to their single error assumption, our design can detect all unidirectional errors. Second, to tolerate L faults, the residue decoder of Taheri et. al. needs $\binom{r+L}{r}$ copies of residue decoders. In our design only one residue decoder with $r + L$ residue digit inputs is required. Thus the hardware cost is minimal. Third, to degrade the system gracefully, they need much more hardware support whereas in our design no extra hardware is required. That is to say we use the same hardware for both fault-tolerance and graceful degradation.

Comparing our design with other similar efforts [12], [29], our design also has several advantages. First of all, we do not need to assume that overflow and error could not occur simultaneously because we do not use the properties of RRNS. Instead, we mix the features of cyclic permutation networks and unidirectional detecting codes: 1-out-of-m codes and Berger codes, into our circuits. Second, [12], [29] need at least L redundant moduli for detecting L faults. However, our design does not need any redundant moduli if the goal of the system is only to detect errors. Third, [12], [29] only consider the errors which occur before the residue decoder. However, this is not the case in most practical applications. Since the residue decoder also is a complex circuit, it, in general, is as likely to fail as the other circuits in the system. In our design, the error detection is for the entire system. Finally, they all rely on the MRX algorithm [9] which requires excessive hardware to do the error detection. However, in our design we use the self-checking Berger code checkers [17] which have less hardware cost than the MRX algorithm.

As for future research, a major task that remains is an effective MOS VLSI implementation of the self-checking Berger code checkers used in our CED and fault-tolerant arithmetic processors.

References

[1] J. A. Abraham and W. K. Fuchs, "Fault and error models for VLSI," *Proc. IEEE*, vol. 74, pp. 639–654, May 1986.

[2] F. Barsi and P. Maestrini, "Error correcting properties of redundant residue number systems," *IEEE Trans. Comput.*, vol. C-22, pp. 307–315, March 1973.

[3] ——, "Error detection and correction by product codes in residue number systems," *IEEE Trans. Comput.*, vol. C-23, pp. 915–923, September 1974.

[4] J. M. Berger, "A note on error correction codes for asymmetric channels," *Inform. and Contr.*, vol. 4, pp. 68–73, March 1961.

[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. Circ. and Syst.*, vol. 35, pp. 1425–1430, Nov. 1988.

[6] R. J. Cosentino, "Fault tolerance in a systolic residue arithmetic processor array," *IEEE Trans. Comput.*, vol. 37, pp. 886–890, July 1988.

[7] M. H. Etzel and W. K. Jenkins, "Redundant residue number systems for error detection and correction in digital filters," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-28, pp. 538–544, Oct. 1980.

[8] N. Gaitanis, "Totally self-checking checkers for low-cost arithmetic codes," *IEEE Trans. Comput.*, vol. C-34, pp. 596–601, July 1985.

[9] H. L. Garner, "The residue number system," *IRE Trans. Elec. Comput.*, vol. EC-8, pp. 140–147, June 1959.

[10] ——, "Error codes for arithmetic operations," *IEEE Trans. Elec. Comput.*, vol. EC-15, pp. 763–770, October 1966.

[11] W. K. Jenkins, "The design of error checkers for self-checking residue number arithmetic," *IEEE Trans. Comput.*, vol. C-32, pp. 388–396, April 1983.

[12] W. K. Jenkins and E. J. Altman, "Self-checking properties of residue number error checkers based on mixed radix conversion," *IEEE Trans. Circ. and Syst.*, vol. 35, pp. 159–167, Feb. 1988.

[13] J. Khakbaz, "Totally self-checking checkers for 1-out-of-n code using two-rail codes," *IEEE Trans. Comput.*, vol. C-31, pp. 677–681, July 1982.

[14] J. Khakbaz and E. J. McCluskey, "Current error detection and testing for large PLA's," *IEEE Trans. Elec. Devices*, vol. ED-29, pp. 756–764, April 1982.

[15] M.-B. Lin and A. Y. Oruç, "The design of a network-based arithmetic processor," *UMIACS-TR-91-141*, Oct. 1991.

[16] ——, "Constant time inner product and matrix computations on permutation-network-based processors," in *Proc. 1992 Int. Conf. Parallel and Distrib. Syst.*, Dec. 16–18, 1992, Hshin-Chu, Taiwan, pp. 441–448.

[17] J. Ch. Lo and S. Thanawastien, "The design of fast totally self-checking Berger code checkers," *Dig. Papers 18th Int. FTC Symp.*, pp. 226-231, July 1988.

[18] G. P. Mak, J. A. Abraham, and E. S. Davidson, "The design of PLAs with concurrent error detection," *Dig. 12th Annual Int. Symp. Fault-Tolerant Comput.*, pp. 303–310, 1982.

[19] D. Mandelbaum, "Error correction in residue arithmetic," *IEEE Trans. Comput.*, vol. C-21, pp. 538–545, June 1972.

[20] M. A. Marouf and A. D. Friedman, "Design of self-checking checkers for Berger codes," *Dig. 8th Annual Int. Symposium Fault-Tolerant Comput.*, pp. 179–184, 1978.

[21] A. Mukherjee, *Introduction to nMOS & CMOS VLSI systems design*. Englewood Cliffs, NJ: Prentice-Hall, 1986, ch. 6.

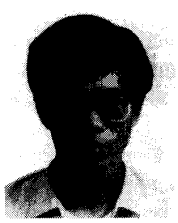
[22] D. Nikolos, A. M. Paschalis, and G. Philokyprou, "Efficient design of totally self-testing checkers for all low-cost arithmetic codes," *IEEE Trans. Comput.*, vol. C-37, pp. 807–814, July 1988.

[23] S. J. Piestrak, "Design of high-speed and cost-effective self-testing checkers for low-cost arithmetic codes," *IEEE Trans. Comput.*, vol. C-39, pp. 360–374, March 1990.

[24] ——, "Design of residue generators and multi-operand modular adders using carry-save adders," *IEEE 10th Symp. Computer Arithmetic*, pp. 100–107, 1991.

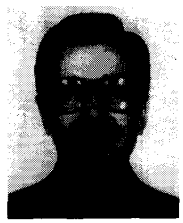
[25] ——, "The minimal test set for multioutput threshold circuits implemented as sorting networks," *IEEE Trans. Comput.*, vol. 42, pp. 700–712, June, 1993.

[26] D. K. Pradhan and J. J. Stiffler, "Error-correcting codes and self-checking circuits," *IEEE Comput.*, pp. 27–37, March 1980.
[27] V. Ramachandran, "Single residue error correction in residue number systems," *IEEE Trans. Comput.*, vol. C-32, pp. 504–507, May 1983.
[28] T. R. N. Rao and E. Fujiwara, *Error-control coding for computer systems*. Englewood Cliffs, NJ: Prentice-Hall, 1989, ch. 7–9.
[29] C. Su and H. Lo, "An algorithm for scaling and single residue error correction in residue number systems," *IEEE Trans. Comput.*, vol. 39, pp. 1053–1064, August 1990.
[30] M. Taheri, G. A. Jullien, and W.C. Miller, "Fault detection in distributed RNS processing," in *IEEE Proc. 1987 Int. Conf. Computer Design*, ICCD-87, 1987, pp. 302–305.
[31] Yuval Tamir and Carlo H. Sequin, "Design and application of self-testing comparators implemented with MOS PLA's," *IEEE Trans. Comput.*, vol. C-33, pp. 493–506, June 1984.
[32] R. W. Watson and C. W. Hastings, "Self-checked computation using residue arithmetic," *IEEE Proc.*, pp. 1920–1931, Dec. 1966.
[33] S. Sik-sang Yau and Y. Liu, "Error correction in redundant residue number systems," *IEEE Trans. Comput.*, vol. C-22, pp. 5–11, January 1973.
[34] E. Fujiwara and K. Matsuoka, "A self-checking generalized prediction checker and its use for built-in testing," *IEEE Trans. Comput.*, vol. C-36, pp. 86–93, January 1987.
[35] T. Nanya and T. Kawamura, "Error secure/propagating concept and its application to the design of strongly fault-secure processors," *IEEE Trans. Comput.*, vol. C-37, pp. 14–24, Jan. 1988.
[36] D. K. Pradhan and J. J. Stiffler, "Error-correcting codes and self-checking circuits," *IEEE Comput.*, pp. 27–37, Mar. 1980.



Ming-Bo Lin was born in Taipei, Taiwan on March 26, 1959. He received the B.Sc. and M.Sc. degrees (B.Sc. with honors) in electronic engineering from the National Taiwan Institute of Technology, Taipei, in 1981, and the National Taiwan University, Taipei, in 1986, respectively. He received the Ph.D. degree, also in electrical engineering from University of Maryland, College Park, in 1992.

He is currently an associate professor with the Electronic Engineering Department at National Taiwan Institute of Technology, Taipei, where he has served since 1986. From 1990 to 1992, he was awarded a fellowship from the Ministry of Education of Taiwan to pursue his Ph.D. degree at the University of Maryland, College Park. His research interests include parallel algorithms, VLSI system design, computer system architecture and parallel processing, computer arithmetic, and fault-tolerant computing



A. Yavuz Oruç (SM'92) received the B.Sc. degree in electrical engineering from the Middle East Technical University, Ankara, Turkey in 1976, the M.Sc, degree in electronics from the University of Wales, Cardiff, United Kingdom in 1978, and the Ph.D degree in electrical engineering from Syracuse University, Syracuse, NY in 1983.

Since January 1988, he has been an associate professor with the Department of Electrical Engineering at the University of Maryland, College Park. Prior to joining the University of Maryland, he was on the faculty of the Department of Electrical, Computer and Systems Engineering at Rensselaer Polytechnic Institute, Troy, NY. His research interests include computer architecture, parallel computing, and interconnection networks.

Dr. Oruç is a member of the IEEE Communications, Computer and Information Theory Societies.