IEEE 1972 Fault-Tolerant Computing Symposium

A PUBLICATION OF THE IEEE COMPUTER SOCIETY

Digest of Papers
1972 INTERNATIONAL SYMPOSIUM ON

FAULT-TOLERANT COMPUTING

Newton, Massachusetts, June 19-21, 1972

Sponsored by the Technical Committee on
Fault-Tolerant Computing
of the IEEE Computer Society
in cooperation with
The MIT C.S. Draper Laboratory
Cambridge, Massachusetts

72CH0623-9C

Copies available from:

IEEE Computer Society
8949 Reseda Boulevard
Northridge, California 91324

IEEE, Order Department
345 East 47th Street
New York, N. Y. 10017

PRICE: $12.50 Members
$15.00 Non-members

TABLE OF CONTENTS

AUTHORS AND CHAIRMEN OF SESSIONS

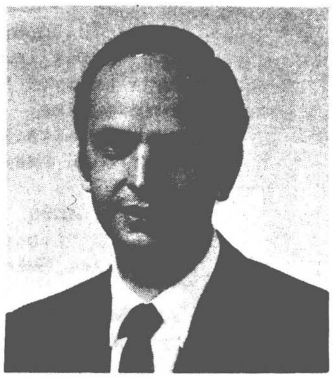
	Session	Page
dshead, H. G.	2	73
nderson, D. A.	1	30
rnold, T. F.	6	200
vizienis, A.	1	25
zema, P.	2	36
astin, D. A.	2	47
orgerson, B. R.	3	89
ouricius, W. G.	6	193
reuer, M. A.	2	53
urns, J. G.	2	42
arter, W. C.	1	14
	5	154
	6	193
hang, H.	2	Chairman
hang, L. C.	4	110
hia, D. K.	4	137
legg, F. W.	4	143
onnet, J. R.	3	94
ook, R. W.	5	160
orman, D.	2	42
ourvoisier, M.	2	36
iaz, M.	2	36
u, M-W.	4	120
uke, K. A.	1	14
lmendorf, W. R.	3	79
ohl, M. E.	2	62
illey, G. C.	3	105
irard, E. M.	2	47
ray, F. G.	6	205
reen, A. I.	6	Chairman
roenig, S. R.	5	170
echt, H.	6	189
irata, F.	4	131
siao, M. Y.	4	137
sieh, E. P.	6	193
ain, G. C.	2	73
essep, D. C.	1	14
	5	154
	6	193
ennedy, P. J.	5	165
ime, C. R.	4	148
ing, J.	3	Chairman
oga, Y.	4	131
indqvist, M. G.	3	100
aki, G. K.	5	170
cCluskey, E. J.	5	182
etze, G. A.	1	30
	5	176
iller, J. S.	3	84
onteiro, P.	1	8
ontelius, G.	3	100
ozawa, K.	2	68
Pasternak, E. J.	3	94
Patterson, W. W.	5	176
Pradhan, D. K.	1	20
uinn, T. M.	5	165
Ramamoorthy, C. V.	4	110
Ramos, S.	4	114
Rao, T. R. N.	1	8
Rault, J-C. L.	2	47
Reddy, S. M.	1	20
	4	126
Richard, J. P.	2	36
Ritani, K.	2	68
Sawin, D. H. III	5	170
Schertz, D.	4	Chairman
Shooman, M. L.	6	211
Siewiorek, D. P.	5	182
Sisson, W. H.	5	160
Smith, A. R. III	4	114
Srinivasan, C. V.	1	Chairman
Stakeberg, L-O.	3	100
Storey, T. F.	5	160
Susskind, A. K.	2	58
Toy, W. N.	5	160
Vancura, R. P.	4	148
Vandever, W. H.	3	84
Wadia, A. B.	5	154
Wagner, B. D.	3	94
Weiss, C. D.	4	120
Wood, P. E.	5	Chairman

1972 INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING

Symposium Chairman:	Albert L. Hopkins, Jr. C.S. Draper Laboratory and Department of Aeronautics and Astronautics Massachusetts Institute of Technology Cambridge, Massachusetts
Vice Chairman:	Sanford Cohen MIT C.S. Draper Laboratory Cambridge, Massachusetts
Treasurer:	Paul Barr Raytheon Company Sudbury, Massachusetts
Registration:	Daniel Dolan MIT C.S. Draper Laboratory Cambridge, Massachusetts
Arrangements:	Robert C. Millard MIT C.S. Draper Laboratory Cambridge, Massachusetts
Digest Editor:	Lutz P. Henckels General Radio Company Concord, Massachusetts
Editor, IEEE-TC Special Issue:	W. C. Carter IBM Research Center Yorktown Heights, New York
Program Chairman:	Gernot Metze University of Illinois Urbana, Illinois

PROGRAM COMMITTEE

Algirdas Avizienis Jet Propulsion Lab Pasadena, California	William C. Carter IBM Research Center Yorktown Heights, New York	Herbert Chang Bell Telephone Labs Naperville, Illinois
Harvey Garner University of Pennsylvania Philadelphia, Pennsylvania	Jack Goldberg Stanford Research Institute Menlo Park, California	Alan I. Green MIT Draper Lab Cambridge, Massachusetts
James King IBM Research Center Yorktown Heights, New York	Donald Schertz Bradley University Peoria, Illinois	C. V. Srinivasan Rutgers University Brunswick, New Jersey
Alfred Susskind Lehigh University Bethlehem, Pennsylvania	Stephen Szygenda Southern Methodist University Dallas, Texas	Paul E. Wood Honeywell Waltham, Massachusetts



The 1972 International Symposium on Fault-Tolerant Computing is the second annual symposium to be sponsored by the Technical Committee on Fault-Tolerant Computing of the IEEE Computer Society. This year's symposium is held in cooperation with the C.S. Draper Laboratory of the Massachusetts Institute of Technology, which has supported several members of the organizing committee.

I wish to cite the efforts of Prof. Gernot Metze, Dr. Lutz P. Henckels, and Messrs. Daniel Dolan and Robert Millards who have all given their valuable enthusiastic, and long efforts to this symposium. I wish also to thank all those who have helped in many ways, including the authors themselves for their cooperation with the review and editorial activities, and the reviewers and program committee members whose efforts were indispensible.

The papers in this volume are worthy successors to those in the highly successful 1971 Symposium. They reflect the rapid progress and increasing interest in the field of Fault-Tolerant Computing which is currently under way, and which promises to continue to grow. The growth of the field is interdependent with the dissemination and exchange of information among all of those interested in and participating in it. Inevitably, the dissemination process in symposia like this one tends to be selective. To those whose efforts and interests are not so well represented in this volume as they might wish, I urge their continued effort to make their contributions to future symposia in this series. Fault-tolerant computing must advance on a broad front if it is to advance at all. One of the purposes of the sponsoring Technical Committee is to stimulate advancement, and for this we require the effective support of large numbers of workers in the diverse areas of the field.

Albert L. Hopkins, Jr., Symposium Chairman

The program of the 1972 International Symposium on Fault-Tolerant Computing, the second symposium in a series begun just last year, attests to the continued growth of interest in this new area. The Program Committee has selected 38 papers that present state-of-the-art results in a wide range of topics in fault-tolerance and has arranged them in six sessions. Various aspects of fault diagnosis, from practical comparisons of the efficiency of different test generation methods to theoretical results on the diagnostic capabilities of subsystems, again dominate the program and two sessions with eight papers each are scheduled. There are also two related sessions on fault-tolerant logic design and system design, including software systems, a session on applications of error protection codes and the design of check circuits, and a session on models for reliability prediction in hardware and software and other models for fault-tolerant computing. In addition, a panel of designers of fault-tolerant systems will present a retrospective analysis of the relative merits of different schemes for achieving fault-tolerance.

The program packs a wealth of information into three days. We hope you will share our enthusiasm and participate actively in the discussions.

Gernot Metze, Technical Program Chairman

As Editor of the 1972 Conference Digest, I wish to express my appreciation to Albert Hopkins, Jr. for his advice and support and to Linda Barron, my secretary, for her indispensable help in preparing the Digest. I want to thank, in particular, the authors whose splended cooperation made my task an easy and enjoyable one.

Lutz P. Henckels, Digest Editor

A RESIDUE CHECKER FOR ARITHMETIC AND LOGICAL OPERATIONS

P. Monteiro and T. R. N. Rao
Department of Electrical Engineering
University of Maryland
College Park, Maryland 20742

ABSTRACT

Until now the application of residue codes had limited itself to arithmetic type operations such as ADD, SHIFT, CYCLE, COMPLEMENT, etc. We establish here that residue codes can be very effectively used to check logical operations (AND, OR, EXOR) as well. The mathematical theory and logic presented here, enables us to design a residue checker organization to detect errors in all arithmetic and logical operations of a processor. Mathematical expressions characterizing the operations of a processor and checker are derived and are used to obtain a hardware implementation schematic.

Index Terms: Residue checker, arithmetic and logical operations, separate adder and checker, residue codes, and self-checking processors.

INTRODUCTION

As the complexity of modern computers is increasing, there is a growing need for efficient self detection and correction of hardware faults in computers. The conventional approach of duplication or triplication, and matching or majority vote-taking of the outputs, has given way to the relatively cheaper approach of error detection through arithmetic codes. The theory and implementation of these codes has received considerable attention in recent years, but there remains a gap between the theory and the development of schemes which are easily and cheaply implementable.

A class of codes known as AN codes has been studied in detail by Brown, Diamond [1] [2]and others, and Avizenis [3] has demonstrated a practical implementation of a 15N code for checking arithmetic operations in a computer. The theory of other residue codes has also been studied in great detail, and various theorems establishing the error correcting properties of such codes have been proved [4][5][6][7][8].

For checking logical operations, various codes have been studied [10][11]. An estimate of the relative cost of error checking through coding, as against that using triplication has been arrived at [12]. However, hitherto error checking by means of residue codes had restricted itself to checking arithmetic operations only. It is our aim in the following paragraphs, to arrive at a set of expressions for the results of all the arithmetic, as well as logical operations in the processor, and then describe a single unit which can be used to perform concurrent diagnosis of the above mentioned operations.

It must be mentioned here that the circuitry for addition and logical operations can also be checked by two-rail logic [15] [16] where each Boolean variable is represented by two lines, such that the true and complemented logic functions are available at each stage. All errors caused by a single failure are detected by this method.

Description of processor unit to be checked

A block diagram of the unit to be checked is given in Fig. 1. The accumulator A is an n-bit register and holds the result at the end of every operation. $(B_{n-1}, B_{n-2}, \ldots, B_o)$ represents the n-bit memory operand, that is fed in through the n parallel input data lines. In addition we have an n-bit parallel adder, the shift rotate control logic, the circuitry used for performing logical operations on words of length n, and the OP-code decoder.

Table 1 is a list of the operations that are performed by the processor (which are all the operations that will be checked).

GENERAL THEORY OF RESIDUE CHECKING

The residue code that will be used in the implementation of the checker, belongs to a class known as separate codes. In a separate residue code, a number N is coded as a pair (N, C(N)). C(N) is the check symbol for N. Addition of two code words $(N_1, C(N_1))$ and $(N_2, C(N_2))$ yields the code word $(N_1+N_2, C(N_1) * C(N_2))$, where a suitable operation '$*$' has to be defined. Note that the addition of code words involves the two operations of '$+$' and '$*$' on the information and check parts respectively. Also, if the equation $C(N_1)*C(N_2) = C(N_1+N_2)$ is satisfied, then the sum of two code words yields another code word. A model of a processor unit using a separate adder and checker is shown in Fig. 2a. Peterson [5] has proven a theorem of fundamental importance to schemes where adder and checker are independent units.

Theorem 1. (Peterson). If there are fewer check symbols than integers permissible in the range of N, (of the model given in Fig. 2(a)) and if the check symbols C(N) satisfy $C(N_1)*C(N_2) = C(N_1+N_2)$, then C(N) must be the residue of N modulo some base b, and $*$ is addition modulo b of these check symbols.

Thus every separate code closed under addition is of the form $(N, |N|_b)$, ($|N|_b$ being defined as the least non-negative integer congruent to N modulo b), for some suitable b.

Peterson's model can readily be generalized to include all elementary arithmetic and arithmetic type operations, such as SUB, SHIFT, CYCLE, etc. Fig. 2b is a model for the generalized separate processor and checker. Further, if addition in the processor is done modulo m, (where $m = 2^n$ or 2^n-1 depending on whether 2's or 1's complement arithmetic is used, and n is the number of bits of the operand register), then we have the following result, first established by Garner [4], and also used by Rao [8] in his error checking scheme for arithmetic operations.

Theorem 2. If N_1 and N_2 are elements of Z_m (where Z_m denotes the ring of integers modulo m) then the $(N, |N|_b)$ code is closed under addition if and only if b divides m.

Here b is called the check base. Thus if N represents the integer value of the accumulator contents at any instant, the residue $|N|_b$ is obtained in a check register, such that the contents of the accumulator, together with the contents of the check register form a code word at the end of an operation cycle.

Let N_1 denote the integer value of the accumulator contents at the beginning of an operation cycle. Let N_2 denote the integer value of the input on the parallel data lines from memory. An operation denoted by Φ_i can be defined as a function of either N_1, N_2 or both N_1 and N_2. Thus Φ_i can be a unary or a binary function. The integer value of the result of this operation is denoted by R, such that (for a binary operation) $R = \Phi_i(N_1, N_2)$. Thus R is the integer value of the output, which is stored in the accumulator at the end of an operation cycle.

Let N_{1b} denote the residue of N_1 modulo the check base b. We assume that N_{1b} is maintained in the check register. Let N_{2b} denote the residue of N_2 modulo b. Our aim is to find an operator Φ_c, such that $\Phi_c(N_{1b}, N_{2b}) = r$, and r is the residue of R modulo b, thus preserving the relation of congruence modulo b between the contents of the accumulator and those of the check register. In symbolic language, we wish to preserve the relation $|\Phi(N_1, N_2)|_b = \Phi_c(N_{1b}, N_{2b})$. If we can find such a Φ_c for every Φ, error checking is simply reduced to checking the relation of congruence between the accumulator contents and the check register contents periodically. Any violation of this congruence can be set up as an error signal.

In the following discussion we consider the accumulator to be of length n. In order to have simplified checking logic, 1's complement arithmetic will be used (which is equivalent to considering all operations modulo 2^n-1), and b will be of the form 2^k-1, where k divides n.

The fault coverage at the gate level depends on the scheme of addition and the logic being used. The discussion of error coverage is given in Section 5, with particular reference to the coverage obtainable with a mod 15 residue checker.

CHECKING ARITHMETIC OPERATIONS

The checking of arithmetic operations has been discussed in a previous paper[8]. For each operation that is executed in the processor, there will be a parallel operation in the checker, such that at the end of the operation cycle, the checker maintains a residue of the accumulator contents modulo b (assuming there has been no error in the computation).

The first part of Table 2 is a listing of the formulas for the results of various processor arithmetic and arithmetic type operations, and their corresponding checker operations. These can be easily derived along the lines of the derivation given below for a sample arithmetic type operation.

Consider the SHR operation, which shifts the contents of the accumulator to the right once. Let A(t) denote the n-bit binary array represented by the contents of the accumulator at instant t. Let $A_i(t)$ denote the binary digit stored in the i+1th bit (from right to left) of the accumulator at time t. Let $\delta_I(A(t))$ denote the integer value of the accumulator at time t, such that $\delta_I(A(t)) = N_1$. Assume each operation takes one unit of time. Let the operator Φ denote the SHR operation, and let R denote the integer value of the result of the operation, which is stored in the accumulator at the end of the operation.

We have,

$$R = \delta_I(A(t+1)) = \Phi(N_1, N_2)$$

$$A(t+1) = (A_{n-1}(t+1), A_{n-2}(t+1) \ldots\ldots A_0(t+1)) ,$$

where

$$A_{n-1}(t+1) = 0$$

$$A_j(t+1) = A_{j+1}(t) \quad \text{for} \quad j = 0, 1, \ldots\ldots n-2$$

$$R = \delta_I(A(t+1)) = \sum_{i=0}^{n-1} A_i(t+1) \cdot 2^i$$

$$= \sum_{i=0}^{n-2} A_{i+1}(t) \cdot 2^i = \sum_{i=0}^{n-2} A_{i+1}(t)\, 2^{i+1}/2$$

Let j = i + 1

$$R = 1/2 \sum_{j=1}^{n-1} A_j(t) \cdot 2^j + 1/2\, A_0(t) - 1/2\, A_0(t)$$

$$= 1/2 \sum_{j=0}^{n-1} A_j(t) \cdot 2^j - 1/2\, A_0(t) = (N_1 - A_0(t))/2$$

Therefore the result r of the checker is given by

$$r = |1/2(N_1 - A_0(t))|_b = |1/2(N_{1b} - A_0(t))|_b$$

Similarly formulas are derived for all other arithmetic and arithmetic type operations listed in Table 1.

We can see that except for corrections in the form of the least and most significant digits from the accumulator during the SHR and SHL operations, the checker operates independently of the processor in a parallel fashion. Thus the checker unit will have a k-bit check register, a k-bit parallel adder, a decoder, and appropriate gating as shown in Fig. 3.

CHECKING LOGICAL OPERATIONS

In trying to use a residue code for checking logical operations, we immediately run into a problem. The residue code is closed under operations like ADD, SUB, CYCLE etc., but is not closed under bitwise logical operations such as AND, OR and exclusive - OR. However by use of a simple relation between arithmetic and logical operations, stated in the theorem below, we can make use of the same unit for checking both-arithmetic as well as logical operations.

Notation: Let A(t) and B(t) each represent binary arrays of length n.

Let $\delta_I(A(t)) = N_1(t)$.

$\delta_I(B(t)) = N_2(t)$

Let '·', 'V' and '⊕' denote the bit-by-bit AND, OR and exclusive-OR operations on arrays A(t) and B(t).

Theorem 3:

$$\delta_I(A(t)) + \delta_I(B(t)) = \delta_I(A(t)\cdot B(t)) + \delta_I(A(t) \vee B(t))$$

Also

$$\delta_I(A(t) \oplus B(t)) = \delta_I(A(t)) + \delta_I(B(t)) - 2\delta_I(A(t)\cdot B(t))$$

The theorem can easily be proved by showing that the relation is true for a single bit i of the two arrays, and then summing up from i = 0 to i = n-1.

As an immediate consequence of the above we have

Corollary

$$|\delta_I(A(t) \vee B(t))|_b = |N_{1b}(t)+ N_{2b}(t)-|\delta_I(A(t)\cdot B(t))|_b|_b$$

$$|\delta_I(A(t)\oplus B(t))|_b = |N_{1b}(t)+ N_{2b}(t)- 2|\delta_I(A(t)\cdot B(t))|_b|_b$$

The formulas for the result of logical operations are given in the second half of Table 2.

HARDWARE IMPLEMENTATION

From the various formulas listed in Table 2 we can obtain sequences of microoperations that make up each arithmetic and logical operation.

From the results of the previous section we see that the logical operations considered can be expressed in terms of the ADD, AND, SUB and CYCLE LEFT (CYL) operations. The OR operation reduces to ADD and AND operations followed by SUB. The EXOR operation reduces to ADD and AND followed by CYL and SUB. Thus checking of logical operations by means of a residue code requires additional circuitry in the checker, consisting of just n two-input AND gates, as the circuitry for performing ADD, SUB and CYL operations is already available in the checker.

We can derive the Boolean equations corresponding to each sequence of microoperations for the checker, and these can then be translated into a hardware design. Design Studies have been carried out on a mod 15 residue checker for a 16 bit monolithic parallel processor [14]. The processor circuitry consists of about 800 gates. The checker circuitry consists of about 400 gates. The hardware increase is only of the order of 50%. Moreover the checker does not slow down processor operation. Error detection can be carried out during each instruction interpretation cycle, while the next operation is being decoded. A block diagram of the checker is given in Fig. 3.

Error Coverage

Although detailed studies on the exact error coverage obtainable by this scheme are not yet completed, some general comments are in order. The mod 15 residue check detects all failures for which the error is not a multiple of 15. All the gates that affect one, two or three successive bits of the accumulator are covered, as $|\pm 2^j|_{15}$ $|\pm 3.2^j|_{15}$, $|\pm 5.2^j|_{15}$ and $|\pm 7.2^j|_{15}$ are all non zero. All bursts of length 3 or less, and most bursts of length 4 or greater are detected.

The coverage at the gate level for addition circuitry depends on the scheme of addition being used. In the example being considered, the 16 bit operand in the ALU is divided into bytes of four bits each. Thus there will be four groups of digits. During addition of two words, carries are rippled through each group of four bits, but there is a separate fast carry structure look-ahead logic for propagation of the carry between one group and the next. Langdon and Tang [13] have shown, that when such is the case, a failure in a single carry line may cause a burst, such that the error value is not necessarily 2^i for some $i(0 \le i \le 15)$. This is because the carries do not propagate up to the position expected if they were allowed to ripple all the way through. The faulty carry may affect the sum bits in the particular group within which it is generated, but the carry into the succeeding group is independently and correctly generated, and hence the sum bits of the next group are correct. For a detailed discussion on the type of errors caused by individual gate failures in carry look-ahead adders, the reader is referred to the paper by Langdon and Tang [13].

CONCLUSION

The main effort in this paper, is directed towards showing how a single checker unit utilizing a residue code can be used for checking both, logical, as well as arithmetic operations in a digital computer. The cost of checking has been found to be about 50% of the cost of the processor, which is far below the cost involved in duplication and matching.

Using a mod 15 residue code as an example, we have shown that apart from single errors, all bursts of length 3 or less and most bursts of length 4 are detected with this code. Thus, the mod 15 checker covers a large percentage of errors that may occur due to faults in the ALU.

Future research in this area will concentrate on studies on syndrome generation and error correction using bi-residue codes. The theory of error correction for both, logical as well as arithmetic operations will be considered.

ACKNOWLEDGEMENTS

This work is supported in part by research grants NASA NGR-21-002-229, NSF GK-17308.

REFERENCES

[1] Brown, D. T., "Error Detecting and Error Correcting Binary Codes for Arithemtic Operations," IRE Trans. on EC Vol. EC-9, Sept. 1960, pp. 333-337.

[2] Diamond, J. M., "Checking Codes for Digital Computers," Proc. of the IRE, Vol. 43, April 1955, pp. 487-488.

[3] Avizenis, A., "Design of Fault Tolerant Computers," Proceedings of 1967 FJCC, Vol. 31, pp. 733-743.

[4] Garner, H. L., "Error Codes for Arithemtic Operations," IEEE Trans. MEC Vol. EC-15, No. 5, Oct. 1966, pp. 763-770.

[5] Peterson, W. W., "On Checking an Adder," IBM Journal of Research and Development, Vol. 2, April 1958, pp. 166-168.

[6] Chien, R. T., "Linear Codes for Single Error Correction in Binary Arithmetic and Transmission," 1963 IEEE Natl. Conv. Rec., pt. 4, pp. 133-138.

[7] Henderson, D. S., "Residue Class Error Checking Codes," Proc. 16th Natl. Meeting of the ACM, Los Angeles, Calif., Sept. 1961.

[8] Rao, T. R. N., "Error Checking Logic for Arithmetic Type Operations of a Processor," IEEE Trans. on EC Vol. C-17, pp. 845-849, Sept. 1968.

[9] Rao, T. R. N., "Biresidue Error-Correcing Codes for Computers Arithmetic," IEEE Trans. on Computers, Vol. C-19, No. 19, May 1970, pp. 398-402.

[10] Hamming, R. W., "Error Detecting and Error Correcting Codes," Bell System Tech. Journal, Vol. 29, No. 2, April 1960, pp. 147-160.

[11] Peterson, W. W. and Rabin, H. O., "On Codes for Checking Logical Operations," IBM Journal, 3, 163, April 1959.

[12] Garcia, O. N. and Rao, T. R. N., "On the Methods of Checking Logical Operations," Proc. Second Annual, Princeton Conf. on Information Sciences and Systems, March 1968.

[13] Langdon, G. G., and Tang, C. K., "Concurrent Error Detection for Group Look Ahead Binary Adders."

[14] Periodic Progress Report, "Monolithic Parallel Processors, by RCA Electronic Components, Simerville, N. J., Prepared for Goddard Space Flight Center, Greenbelt, Md., Dec. 1968.

[15] F. F. Sellers, M. Y. Hsaio, L. W. Bearnson, "Error Detecting Logic for Digital Computers," McGraw-Hill, 1968.

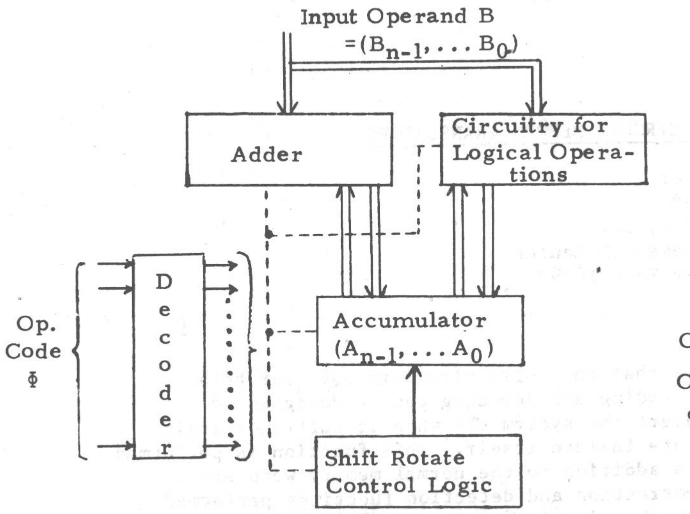
[16] P. Fagg, "IBM S/360 Engineering," Fall Joint Computer Conference, pp. 205-237, 1964.

TABLE 1: Description of Processor Operations.

Symbol	Op. Code	Description
Φ_1	ADD	Adds contents of accumulator and parallel data lines and stores result in accumulator.
Φ_2	SUB	Subtracts contents of parallel data lines from accumulator and stores result in accumulator.
Φ_3	SM	Subtracts contents of accumulator from parallel data lines, and stores result in accumulator.
Φ_4	COMP	Complements the contents of the accumulator bitwise.
Φ_5	SHL	Shifts the contents of the accumulator to the left once.
Φ_6	SHR	Shifts the contents of the accumulator to the right once.
Φ_7	CYR	Rotates the contents of the accumulator to the right once.
Φ_8	CYL	Rotates the contents of the accumulator to the left once.
Φ_9	CLA	Clears accumulator and loads value on parallel data lines into accumulator.
Φ_{10}	CLZ	Clears accumulator to zero.
Φ_{11}	AND	Performs bitwise logical AND operation on contents of accumulator and parallel data lines and stores result in accumulator.
Φ_{12}	OR	Performs a bitwise logical OR operation on contents of accumulator and input on parallel data lines and stores result in accumulator.
Φ_{13}	EXOR	Performs bitwise logical exclusive - OR operation on contents of accumulator and input on parallel data lines, and stores result in accumulator.

TABLE 2: Characterization of Arithmetic and Logical Operations of a Processor

Operation ϕ_i	Result $R = \phi_i(N_1, N_2)$ $m = 2^n - 1$	Operation ϕ_{ci}	$Q = \phi_{ci}(N_{1b}, N_{2b})$ $b = 2^k - 1$
	A. ARITHMETIC OPERATIONS		
ϕ_1(ADD)	$\lvert N_1 + N_2 \rvert_m$	ϕ_{c1}	$\lvert N_{1b} + N_{2b} \rvert_b$
ϕ_2(SUB)	$\lvert N_1 - N_2 \rvert_m$	ϕ_{c2}	$\lvert N_{1b} - N_{2b} \rvert_b$
ϕ_3(SM)	$\lvert N_2 - N_1 \rvert_m$	ϕ_{c3}	$\lvert N_{2b} - N_{1b} \rvert_b$
ϕ_4(COMP)	$\lvert - N_1 \rvert_m$	ϕ_{c4}	$\lvert - N_{1b} \rvert_b$
ϕ_5(SHL)	$\lvert 2N_1 - A_{n-1}(t) \rvert_m$	ϕ_{c5}	$\lvert 2N_{1b} - A_{n-1}(t) \rvert_b$
ϕ_6(SHR)	$\lvert 1/2(N_1 - A_0(t)) \rvert_m$	ϕ_{c6}	$\lvert 1/2(N_{1b} - A_0(t)) \rvert_b$
ϕ_7(CYR)	$\lvert 1/2 N_1 \rvert_m$	ϕ_{c7}	$\lvert 1/2\, N_{1b} \rvert_b$
ϕ_8(CYL)	$\lvert 2N_1 \rvert_m$	ϕ_{c8}	$\lvert 2N_{1b} \rvert_b$
ϕ_9(CLA)	N_2	ϕ_{c9}	N_{2b}
ϕ_{10}(CLZ)	0	ϕ_{c10}	0
ϕ_{11}(SET)	0	ϕ_{c11}	0
	B. LOGICAL OPERATIONS		
ϕ_{12}(AND)	$\lvert \delta_I(A(t) \cdot B(t)) \rvert_m$	ϕ_{c12}	$\lvert \delta_I(A(t) \cdot B(t)) \rvert_b$
ϕ_{13}(OR)	$\lvert \delta_I(A(t) \vee B(t)) \rvert_m$	ϕ_{c13}	$\lvert N_{1b} + N_{2b} - \lvert \delta_I(A(t) \cdot B(t)) \rvert_b \rvert_b$
ϕ_{14}(EXOR)	$\lvert \delta_I(A(t) \oplus B(t)) \rvert_m$	ϕ_{c14}	$\lvert N_{1b} + N_{2b} - 2 \lvert \delta_I(A(t) \cdot B(t)) \rvert_b \rvert_b$



Input Operand B
=(B n-1, ... B 0)
Adder
Circuitry for Logical Operations
Op. Code Φ
Decoder
Accumulator (A n-1, ... A 0)
Shift Rotate Control Logic

Fig. 1: Arithmetic and Logical Unit to be Checked.

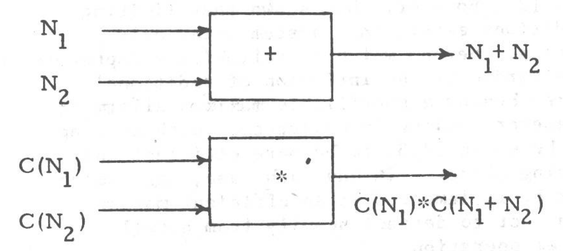


Fig. 2(a): Peterson's Model for Separate Adder and Checker.

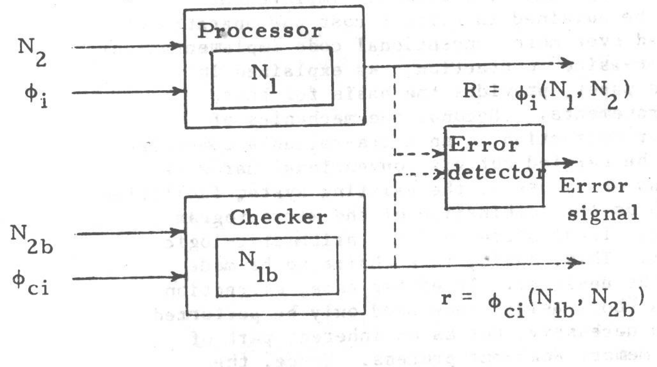


Fig. 2(b): Generalized Model for Separate Processor and Checker.

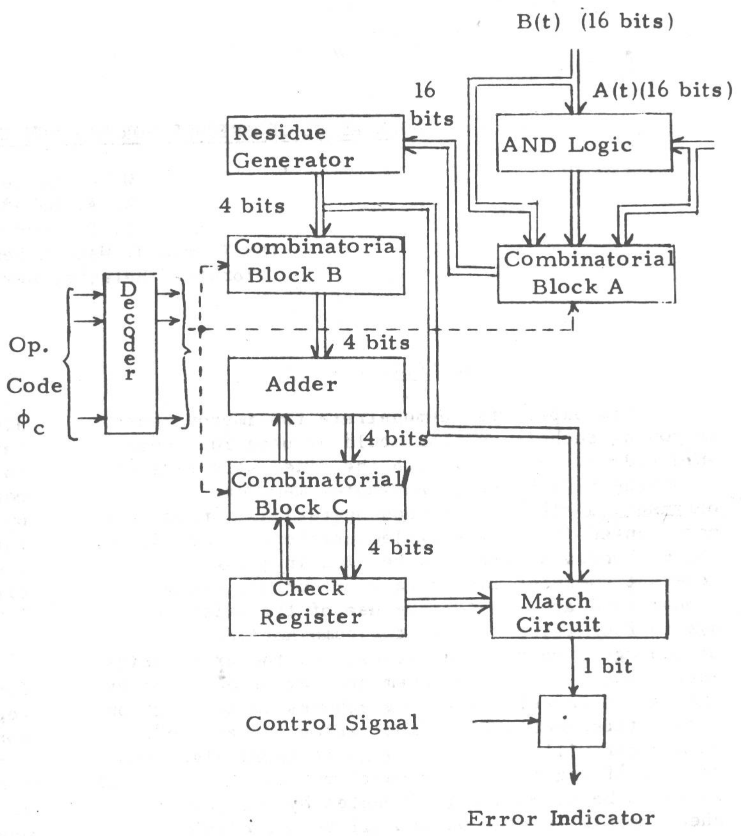


Fig. 3: Block Diagram for Checker

LOOK-ASIDE TECHNIQUES FOR MINIMUM CIRCUIT MEMORY TRANSLATORS

W. C. Carter
K. A. Duke*
D. C. Jessep, Jr.
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT

This paper will demonstrate two improvements in coding techniques that could be used for memory word coding. First, within the fixed structure of a Hamming SEC/DED code, an improvement can be obtained in circuit cost and operational speed over more conventional code implementations. Second, the mechanics of error correction in a fault tolerant computer may be carried out via conventional hardware means or by use of the existing system facilities such as the combination of the microprogram unit, local store, and the arithmetic-logic unit. These improvements may be obtained by the use of Rotational Coding schemes in conjunction with a technique called "Look-aside Correction". This paper will first show a generalized algorithm for specifying the parity check matrix of Rotational Codes. The structure implemented by the parity check matrix in this paper will be not merely encoding and decoding circuitry, but will translate between rotational code forms and byte-parity encoded forms. The unique feature of these translators is that use of the Rotational Code permits the error correction to be performed on only a subset of the data word bits, and only if a single error condition has been detected. The correction mechanism may be either a hardware logic circuit of firmware. The paper concludes with a comparison of the circuit requirements and correctional speed of the Hamming (72,64) Single-Error-Correcting, Double-Error-Detecting Code, as it would normally be implemented, and a Rotational Code Translator also operating on 64 data bits and eight check bits. The latter translator is seen to provide cheaper implementation with faster average speed of error correction.

1. INTRODUCTION

1.1. The Use of Codes in a Fault-Tolerant Computing System

Computing system designers have utilized a variety of coding techniques to enhance the failure toleration capacity of the system, but normally at a significant increment to system cost. Moreoever, the coding circuitry used for reliability improvement can constitute a significant number of logic gates and even become a liability to its own objective if it is designed in an indiscriminate fashion. A recent paper [1] presented design techniques to be used for implementing memory word coding so that the logic circuitry used for both encoding and decoding can be designed to alert the system CPU when it suffers a logic gate failure itself. This function is performed in addition to the normal memory word error correction and detection functions performed by the circuitry. The extra cost in logic gates of this self-testability property is less than 7.5% of the total cost of coding circuitry for a 32 bit word and approaches 1.0% for larger memory words.

An alternate vehicle to coding alone for achieving fault tolerance is that of replacing faulty units by spares. Recent work [2], however, has shown that limiting conditions exist, in a system using only sparing, beyond which no reliability improvements is afforded by the inclusion of additional spares beyond a specifiable maximum allotment. Moreoever, coding in conjunction with sparing can be shown [2,3] to be more efficient than sparing alone. In any such case, the coding must be implemented in an efficient manner so as not to detract heavily from normal system operation.

1.2. A Theory of Design of Coding Circuitry

This paper will demonstrate two distinct improvements in the application of a particular coding technique as it would be used for memory word coding. First, even within a rather fixed structure of the chosen code, as it is specified by the matrix descriptors of coding theory, a dramatic improvement can be obtained in circuit cost and operational speed over more conventional code implementations. "Look-aside" correction, as explained in this paper, provides the basis for these improvements. Second, the mechanics of error correction in an ultra-reliable computer may be carried out via conventional hardware means or by use of the existing system facilities such as the combination of the microprogram unit, local store, and the arithmetic-logic unit. This, again, is a choice to be made by the designer. In either case, correction in a "Look-aside" mode need only be performed when necessary, not as an inherent part of the memory Read-out process. Hence, the primary purpose of this paper will be to demonstrate how minimum implementation cost and minimum delay coding circuitry can be prescribed for a particular class of codes.

* IBM - Systems Development Division
South Road
Poughkeepsie, New York 12601

2. THE PARTICULAR CODE UNDER CONSIDERATION

To properly preface the discussion of the code, it is necessary to delineate carefully some assumptions and terminology to be used in this discussion. The coding circuitry used will actually not merely encode or decode the memory word; the word read out of store will be transformed from its memory coded form into a byte-parity encoded form. The word to be stored will be removed from the bus in a byte-parity encoded form and transformed into an encoded form for storage. Hence, the coding circuitry performing the encoding and decoding is actually translating from one coded form to another, depending on whether data is being placed on, or removed from the bus. Thus, the coding circuitry will be referred to as a translator.

It will be assumed in this paper that the reader is familiar with the concept of the Parity Check Matrix (PCM) and the use of the parity relations between data bits and check bits prescribed by the rows of the PCM to form syndromes; this approach has already received ample treatment in the literature [4]. If the word to be encoded according to the PCM is composed of m bytes of b bits per byte, there are k = mb total data bits. Given k, the normal Hamming relationship [4] determines the number of check bits, r. The PCM will then have n = k + r columns and r rows (and syndromes).

2.1. The Rotational Code

In terms of the PCM just discussed it is now possible to define a class of codes, the Rotational Codes, which will serve as a focal point of this paper. The Rotational Codes are so named because of the appearance of the PCM. The PCM for the Rotational Codes is formed according to the following rules which apply to the case $m = r$. Modifications for $m \neq r$ will be discussed later.

(1) For the r check bit columns of the PCM, choose the r combinations of one 1 and $(r - 1)$ 0's so that the r columns have a 1 in the 1st, rth, $(r - 1)$st,...,2nd rows only, with 0's elsewhere in these columns.

(2) Starting with the columns of the PCM corresponding to only the first byte (the first b columns), assign the entire first row of these columns to be 1's.

(3) For the same b columns of the PCM in step (2), assign each column using the list in Table 1. First, pick b columns or as many columns as possible with three 1's and with min $r \leq b$. If there are less than b such columns, repeat the process for five 1's, then for seven 1's, nine 1's, eleven 1's, until b columns have been selected. Table 2 shows the maximum number of columns with three, five, seven or nine 1's which may be chosen for r bytes for $r = 4,\ldots,11$.

(4) The PCM columns for the succeeding bytes are obtained from the columns formulated for the first byte by vertical rotational of the PCM rows corresponding to byte 1. The row of b 1's, formulated in step (2) as the first row of the PCM columns corresponding to byte 1, now becomes the rth row for the columns corresponding to byte 2, the $(r-1)$st row for the columns corresponding to byte 3. Thus, the row of b 1's will be the $[1 + ((r + 1 - j) \bmod r)]$st row of the jth byte, $j = 1,2,\ldots,r$. Each row is then seen to rotate vertically one row as first the columns for the jth byte are observed and then those of the $j + 1$st byte.

(5) At the completion of the construction of the PCM, there should be mb+r columns, each with an odd number of 1's in it. Each b column-wide section of the first mb columns will be a vertical rotation away from the sections immediately adjacent to it. Each of the final r columns will contain only a single 1 and each of these columns will be a vertical rotation away from the columns immediately adjacent to it in this checkbit portion of the PCM.

There are two considerations paramount to this technique of specifying the PCM: first, variations in the algorithm for relative sizes of m, b and r, and second, the construction of a PCM in which all columns are distinct (no two are identical. To generalize ~~the~~ the above algorithm for PCM construction, the following steps must be taken.

Divide the m bytes evenly, if possible, into r sets, T_i. If $m = dr + e$, $o < e < r$, put $d + 1$ bytes into the first e sets, T_1, $T_2,\ldots,T_e$, ($(d+1)b \leq$ max b in Table 2) and d bytes into the last $(r-e)$ sets, T_{e+1}, $T_{e+2},\ldots,$ T_r. Let set T_i correspond to the ith check bit and the ith row. Begin by putting $b(d+1)$ 1's into the first row under T_1, $b(d+1)$ 1's into the second row under T_2, and continue for the first e sets. Now put bd 1's into the $(e + 1)$st row under the set T_{e+1}, and continue until under each set there are $b(d + 1)$ or bd 1's, each in a separate row. As before, the columns are constructed by the use of Table 1 beginning with three 1's in a column, then sucessively, the columns can be filled with five 1's, seven 1's, etc., until all data bit columns of the PCM are assigned a 1-0 pattern.

For the case of $m < r$, it may be necessary to rotate the rows more than one position at a time. It will not generally be possible, in such a case, to perform r total steps of rotation. If $m > r$, it may be necessary to subdivide one or more bytes of the PCM to obtain a step-wise rotation for each row.

No matter what the relationship of m and b, a minimum number of 1's is used. Use of a minimal number of 1's (starting with three, then five, seven and so on) is advantageous because the fewer 1's there are in the PCM, the fewer the interconnections and the fewer Exclusive-OR (XOR) circuits used. It is best to choose the 1's such that the number in each row is balanced so the circuit delays are approximately equal.

The final result of these steps is a PCM in which all columns are distinct and in which as few a number of 1's as possible appears, for minimal circuit or delay implementation. The next step is to consider how this PCM can be used to provide the encoding and decoding operations necessary and to demonstrate the possibilities for error correction by use of either normal hardware correctors or the microprogram unit and the arithmetic unit.

2.2. Decoding and Encoding with a Rotational Code

The general structure of the PCM formulated for the rotational class of codes in the previous section will have characteristics of importance to coding. The PCM contains a minimum number of 1's for a Hamming Single Error Correcting - Double Error Detecting (SEC/DED) code [5]. Every set of columns corresponding to one byte of data will have one solid row of 1's appearing in it. Additionally, this same row will have other 1's corresponding only to data bits 1's appearing in it. This latter set of 1's will be referred to as the Parity Subset. In conventional coding schemes, the parity of all data bits which correspond to a 1 in the ith row of the PCM is determined directly to form the syndrome S_i. In rotational coding schemes, the syndrome is formed in such a fashion as to permit formation of byte parity as an intermediate step in the process, thus facilitating bus transmission. Define the following variables:

y_i - the parity over the ith row Parity Subset
x_i - the parity over the ith byte
p_i - the parity bit to maintain odd parity across the ith byte.

The distinction is made here on the parity of a byte (x_i) and the parity bit for the same byte (p_i). If the parity for a byte is even, the parity bit will be a 1 if odd parity is required for error detection.

Consider odd parity to be used with a rotational code PCM. Then, for the ith row of the PCM, we have the parity equations:

$$x_i \oplus y_i \oplus c_i = 1$$

and

$$x_i \oplus p_i = 1$$

where " $\oplus$ " denotes the XOR operation and c_i is the check bit which corresponds to a 1 entry in the ith row. Combining the two by an XOR,

$$(1) \quad p_i = y_i \oplus c_i.$$

The implication of this equation is that the parity bit for the ith data byte can now be generated as the parity of the ith row Parity Subset and the check bit, c_i (and does not have to include the actual bits of the ith data byte). The ith date byte, with parity x_i and parity bit p_i, are put into the MDR. Each byte of the MDR is parity checked to ensure data integrity before use after a READ or before encoding after a WRITE begins. Now, for the rotational code, then, the ith syndrome, S_i, is formed from the expression

$S_i = x_i \oplus p_i$ which checks the MDR.

Slight deviations in this are required if $m \neq r$ because of the way the PCM was formed. If $m \neq r$, the parity and syndrome bits are formed as for $m = r$ for syndromes corresponding to PCM rows in which a full byte-wide row of 1's is found. Otherwise, parity and syndrome bits are formed from the XOR of only the bits having 1's in the corresponding row of the PCM.

$S_i = 1$, $i = 1,2,\ldots,r$, for no error in the ith byte. The no error signal, NE, then is formed from NE = $\bigwedge_{i=1}^{r} S_i$ and is 1 for an error free word. If NE = 0, it becomes necessary to determine if it is a single or double error. Anytime a single data error exists, an odd number of syndromes change. If one, and only one, syndrome changes, the error is in a check bit [1,5]. This classification ability derives from the code property that each column has an odd number of 1's in it, with columns having single 1's in them reserved for check bit columns. Thus, a single error has been detected if NE = 0 and if the parity across the syndromes changes. Hence, the single error signal is SE = $\overline{NE} \wedge (S_1 \oplus S_2 \oplus \ldots + S_r)$ for r even and
SE = $\overline{NE} \quad (S_1 \oplus S_2 \oplus \ldots \oplus S_r)'$
for r odd. If there is an error signal (NE = 0) and it is not a single error $(\underline{SE = 0})$, then it is a double error (DE) and DE = $\overline{NE} \wedge \overline{SE} = 1$.

The basic steps of the Memory Read process, then, are given as follows:

1. Formulate a parity bit for each byte by use of the PCM and Equation (1).

2. Form the syndromes from the parity check on the byte parity and its parity bit (formed in 1) for each data byte.

3. Determine if an error condition exists in the data read out. If the data contains no errors, gate the word out to the CPU; otherwise, correct any single error or notify the CPU of any double errors.

The Memory Write process consists of accepting a set of parity encoded data bytes from the bus, checking the parity for each byte, re-encoding the data bits according to the PCM, and gating away the newly re-encoded word into the memory. Once the byte parity of the incoming word has been checked, the check bits for the newly re-encoded form can be generated using the circuitry provided to implement the PCM function for the Read process. Rearrangement of the basic equation, $c_i \oplus y_i = p_i$, for the Read process gives $p_i \oplus y_i = c_i$ for the Write process. Hence the parity bit for each byte and the Parity Subset denoted by y_i in the equation above, are XOR'ed together to provide the check bits required by the data bits in each byte prior to storage. Once the check bits are generated, the word (data and check bits) can be stored. It is thus important to note that this translator literally makes use of the same logic circuitry to both decode and encode and the MDR parity check circuitry is used to check and form a syndrome. The hardware implementation of one such system