Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design

ALGIRDAS AVIŽIENIS, MEMBER, IEEE

Abstract—The application of error-detecting or error-correcting codes in digital computer design requires studies of cost and effectiveness tradeoffs to supplement the knowledge of their theoretical properties. General criteria for cost and effectiveness studies of error codes are developed, and results are presented for arithmetic error codes with the low-cost check modulus 2^a-1. Both separate (residue) and nonseparate (AN) codes are considered. The class of multiple arithmetic error codes is developed as an extension of low-cost single codes.

Index Terms—Arithmetic error codes, fault detection in computers, fault effectiveness of arithmetic codes, low-cost arithmetic codes, multiple arithmetic codes, self-repairing computers.

I. Methodology of Code Evaluation

A. Scope of the Problem

In this paper the name *arithmetic error codes* identifies the class of error-detecting and error-correcting codes which are preserved during arithmetic operations. Given the digital number representations, x, y, an arithmetic operation $*$, and an encoding f: $x \rightarrow x'$, we say that f is an arithmetic-error code with respect to $*$ if and only if there exists an algorithm $A*$ for coded operands to implement the operation $*$ such that

$$A * (x', y') \equiv (x * y)'.$$

The definition applies to single-operand operations and multioperand operations as well, i.e.,

$$A * (x') \equiv (* x)'$$

and

$$A * (x_1', x_2', \cdots, x_n') \equiv (x_1 * x_2 * \cdots * x_n)'$$

must be satisfied in those cases.

Arithmetic error codes are of special interest in the design of fault-tolerant computer systems, since they serve to detect (or correct) errors in the results produced by arithmetic processors as well as the errors which have been caused by faulty transmission or storage. The same encoding is applicable throughout the entire computing system to provide *concurrent diagnosis*, i.e., error detection which occurs concurrently with the operation of the computer. Real-time detection of transient and permanent faults is obtained without a duplication of arithmetic processors.

The economic feasibility of arithmetic error codes in a computer system depends on their cost and effectiveness with respect to the set of arithmetic algorithms and their speed requirements. The choice of a specific code from the available alternatives further depends on their relative cost and effectiveness values. This paper presents the results of an investigation of the cost and effectiveness of arithmetic error codes in digital system design. Other new results include several classes of multiple arithmetic error codes. The investigation was stimulated by the need for low-cost real-time fault detection in the fault-tolerant STAR computer [1], [2]. Favorable results led to the choice of arithmetic encoding of both data words and instruction addresses in this machine. Preliminary reports on parts of the results have been made on several occasions previously [3]–[9].

B. The Criteria of Cost

For the purposes of this paper a "perfect" computer is a reference computer in which logic faults do not occur. The specified set of arithmetic algorithms is carried out with prescribed speed and without errors. For a given algorithm, word length, and number representation system of the perfect computer the introduction of any error code will result in changes that represent the *cost* of the code. The components of the cost are discussed below in general terms applicable to all arithmetic error codes.

1) Word Length: The encoding introduces redundant bits in the number representation. A proportional hardware increase takes place in storage arrays, data paths, and processor units. The increase is expressed as a percentage of the perfect design. "Complete duplication" (100 percent increase) is the encoding which serves as the limiting case.

2) The Checking Algorithm: This tests the code validity of every incoming operand and every result of an instruction. A correcting operation follows when an error-correcting code is used. The cost of the checking algorithm has two interrelated components: the hardware complexity and the time required by checking. The complete duplication case requires only bit-by-bit comparison; other codes require more hardware and time. Provisions for fault detection in the checking hardware itself are needed and add to the cost.

3) The Arithmetic Algorithms: An encoding usually requires a more complex algorithm for the same arithmetic operation than the perfect computer. This cost is expressed by the incremental time and hardware required by the new

Manuscript received March 1, 1971; revised June 21, 1971. This work was supported in part by the National Aeronautics and Space Administration under Contract NAS 7-100, and was performed at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, Calif.

The author is with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, Calif. 91103, and the Department of Computer Science, University of California, Los Angeles, Calif. 90024.

algorithm. The reference case of complete duplication does not add any cost of this type (the algorithms are not changed, but they are performed in two separate processors). The set of arithmetic algorithms which is usually provided in a general-purpose processor is discussed in Section II.

C. The Criteria of Effectiveness

An *arithmetic error* occurs when a logic fault causes the change of one or more digits in the result of an algorithm. A *logic fault* is defined to be the deviation of one or more logic variables from the values specified in the perfect design. Logic faults differ in their duration, extent, and nature of the deviation from perfect values. The effectiveness of an arithmetic error code in a computer may be expressed in two forms: as a direct *value effectiveness*, and as a design-dependent *fault effectiveness.*

1) Value Effectiveness: The most direct measure of effectiveness is the listing of the error values that will be detected or corrected when the code is used. These values are determined by the properties of the code and are independent of the logic structure of the computer in which the code will be used. Value effectiveness for 100 percent detection (or correction) of some class of error values has been the main measure of arithmetic codes. For example, *single* error detection (or correction) is said to occur when *all* (100 percent) errors of value

$$\pm cr^i \qquad 0 < c < r \qquad 0 < i < n - 1$$

are detected (or corrected) in an n-digit, radix-r number [10], [11]. There is no direct reference for algorithms or their implementation. The present study considers value effectiveness with less than 100 percent detection. Such codes may be useful when their cost is low and when other means of fault tolerance supplement the codes in a computer.

2) Fault Effectiveness: The purpose of arithmetic error codes in digital systems is to detect the occurrence of logic faults. The detection enables the system to initiate corrective action (error correction, diagnosis, program restart, etc.). In order to assess the effectiveness of fault detection, the value effectiveness of a code must be translated into a measure of *fault effectiveness* for one or more specified types of logic faults. The translation is performed separately for every algorithm and requires an *error table* for every type of fault. The error table is generated from the description of the logic implementation of the algorithm α. The specified fault ϕ is applied to every logic circuit which is used by the algorithm. Every application yields an *error value* E (or a *set* of error values $\{E\}$) by which the fault will change the perfect value S of the result to the actual (incorrect) value $S* = S + E$. The error table $T(\alpha, \phi)$ lists all error values together with their relative frequencies of occurrence during the compilation of $T(\alpha, \phi)$. A comparison of $T(\alpha, \phi)$ with the detectable error values of the given code f shows which entries of the error table are not detectable. The fault effectiveness of f with respect to (α, ϕ) is the *percentage* of all occurrences of ϕ which will be detected (or corrected) when f is employed. Less than 100 percent fault-effective codes are of interest when their cost is low, because other methods of fault tolerance (especially program restarts) can be used to reinforce the codes [1], [2]. If the fault effectiveness for (α, ϕ) is not sufficient, it may be improved by redesigning the implementation of α to eliminate some or all of the undetectable entries of $T(\alpha, \phi)$.

During the compilation of the error table $T(\alpha, \phi)$ an application of the fault ϕ to a logic circuit changes the radix-r, n-digit perfect result $s \equiv (s_{n-1}, \cdots, s_1, s_0)$ to an "actual" (incorrect) result $s*$ which differs from s in at least one digit. The digit changes which have taken place are described by the error number $e \equiv (e_{n-1}, \cdots, e_1, e_0)$ defined digitwise as

$$e_i = s_i^* - s_i \qquad \text{for } 0 \leq i \leq n - 1.$$

The digits of e are in the range $-r+1 \leq e_i \leq r-1$, and e itself represents the error value E in the range

$$-(r^n - 1) \leq E \leq r^n - 1.$$

When e is recoded to have the minimum number of nonzero digits, this minimum number is defined to be the *arithmetic distance* between s and s*, as well as the *arithmetic weight* of e [11]. The weight of an error value has been employed to indicate its relative probability (single, double, etc.). The results of following sections show that the weight of an error number is data dependent in some algorithms and therefore not suitable as a general criterion of fault effectiveness.

D. Classes of Logic Faults

1) Single Faults: A single logic fault is the deviation of one logic variable from the design value. During an interval of time ΔT_i (to be called a *use*) it has two possible forms: a) the logic variable is "stuck-on-zero" (abbreviated S0) when it assumes the actual value 0 instead of the design value 1; and b) the logic variable is "stuck-on-1" (abbreviated S1) when it assumes the actual value 1 instead of the design value 0.

The circuits that are used to store, transmit, or generate digit values during an algorithm will be called *digit circuits.* A single fault is said to be *local* if its immediate effect changes the value of only one digit, i.e., the local fault in position i $(0 \leq i \leq n-1)$ of a radix-r operand adds the value

$$cr^i, \qquad -r + 1 \leq c \leq r - 1$$

to the affected number. The value of the error number is either cr^i, or an arithmetic function of cr^i, determined by the location of the fault and the microprogram of the algorithm. A single fault which immediately affects more than one digit is *distributed.* Its effect is expressed as the cumulative effect of two or more local faults.

2) One-Use and Repeated-Use Faults: With respect to the microprogram of the algorithm, there are one-use and repeated-use faults. The fault is a *one-use* fault when the faulty digit circuit is used only once before the checking algorithm is performed. Iterative algorithms (multiplication, division, byte-serial addition, etc.) employ the same digit circuits repeatedly in order to generate the result; if one of these

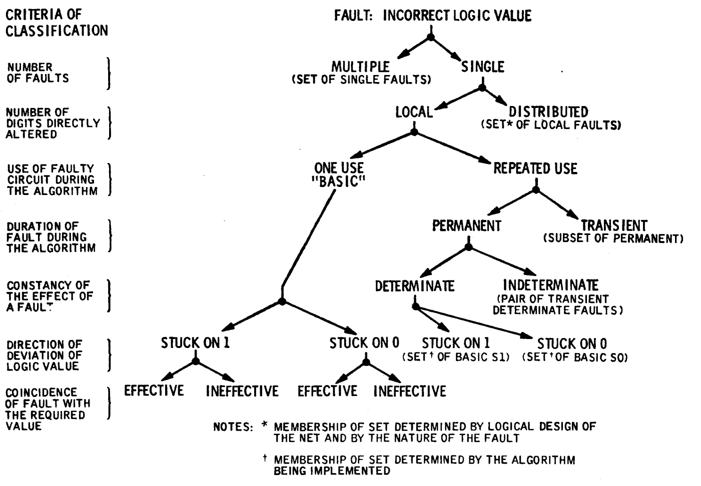
CRITERIA OF CLASSIFICATION
NUMBER OF FAULTS
NUMBER OF DIGITS DIRECTLY ALTERED
USE OF FAULTY CIRCUIT DURING THE ALGORITHM
DURATION OF FAULT DURING THE ALGORITHM
CONSTANCY OF THE EFFECT OF A FAULT
DIRECTION OF DEVIATION OF LOGIC VALUE
COINCIDENCE OF FAULT WITH THE REQUIRED VALUE
FAULT: INCORRECT LOGIC VALUE
MULTIPLE (SET OF SINGLE FAULTS)
SINGLE
LOCAL
DISTRIBUTED (SET* OF LOCAL FAULTS)
ONE USE "BASIC"
REPEATED USE
PERMANENT
TRANSIENT (SUBSET OF PERMANENT)
DETERMINATE
INDETERMINATE (PAIR OF TRANSIENT DETERMINATE FAULTS)
STUCK ON 1
STUCK ON 0
STUCK ON 1 (SET† OF BASIC S1)
STUCK ON 0 (SET† OF BASIC S0)
EFFECTIVE
INEFFECTIVE
EFFECTIVE
INEFFECTIVE
NOTES: * MEMBERSHIP OF SET DETERMINED BY LOGICAL DESIGN OF THE NET AND BY THE NATURE OF THE FAULT
† MEMBERSHIP OF SET DETERMINED BY THE ALGORITHM BEING IMPLEMENTED

Fig. 1. Classification of logic faults.

circuits is faulty, a *repeated-use* fault results. Repeated-use faults differ according to their one-use effectiveness, duration, and determinancy. The fault is ineffective during the use ΔT_i if the fault-induced value is identical to the design value. The fault is *transient* if it does not exist during one or more of the uses; otherwise it is *permanent*. A transient fault is equivalent to a permanent fault that is ineffective during some uses; consequently, transient faults are a subset of permanent faults. Some types of failures cause the logic value at a point to become uncertain, and it is interpreted randomly as either one or zero during the repeated uses of the faulty circuit. In these cases neither a constant S0 nor a constant S1 fault exists for all uses; the fault is *indeterminate* and is called "stuck-on-X," abbreviated SX. An indeterminate fault has the combined effect of two transient faults (one S0, the other S1) affecting the same variable.

3) Cumulative Fault Effects: A multiple (double, triple, etc.) fault occurs when two or more faulty logic variables exist during the same algorithm. Its effect is expressed as the cumulative effect of two or more single faults. A review of the fault model shows that the effect of any fault is equivalent to the cumulative effect of a set of local one-use faults. The *basic fault* is defined to be a local one-use fault (either S0 or S1 at ΔT_i). In the study of fault effectiveness, the effect of a basic fault is determined for every digit circuit and every algorithm of a processor or storage array. The effect of any other fault is then determined in two steps: a) identify the set of basic faults which corresponds to the given fault; and b) determine the effect (error value, or set of possible error values) of the given fault by applying sequentially the basic faults identified in step a.

The classification of faults is summarized in Fig. 1.

II. Fault Effects in Binary Arithmetic Processors

A. Basic Faults in Parallel Arithmetic

The set of arithmetic algorithms which is provided in a general-purpose processor includes at least the eight algorithms listed in Table I either separately or as parts of composite algorithms, multiplication, and division. In this section we determine the error magnitudes due to the existence of a basic (local, one-use) fault in a digit circuit of a radix-2 processor. A parallel design is assumed, in which the algorithms of Table I use the digit circuits of the processor only once, and the faults are single-use faults. The error magnitudes $|E|$ which can be generated by a basic fault and their arithmetic weights are presented in Table I. The radix-2 operands are n binary digits long $(0 \leq i \leq n-1)$. Two systems for the representation of negative numbers are considered: complements with respect to $N_1 = 2^n - 1$ (one's complements), and complements with respect to $N_2 = 2^n$ (two's complements). All operands and results are treated as unsigned integer values for checking purposes. The transfer (A1) is included in every other algorithm, thus the $|E| = 2^i$ of a transfer may occur in every case. If the same register is used to hold an operand and the result, a repeated-use fault may result.

TABLE I
Error Magnitudes due to a Basic Fault in a Parallel Binary Processor

Number system / Algorithm		$N_1 = 2^n-1$ ("one's" complement)		$N_2 = 2^n$ ("two's" complement)	
		Error Magnitude $\lvert E \rvert$	Weight W	Error Magnitude $\lvert E \rvert$	Weight W
A1	Transfer (applies also to A2-A7 below)	2^i	1	2^i	1
A2	Left shift, k digits	2^k-1	2	2^k-1	2
A3	Right shift, k digits	$2^{n-1-k}(2^{k+1}-1)$	2	$2^{n-1-k}(2^{k+1}-1)$	2
A4	Range extension	$2^{n-1}(2^{k+1}-1)$	2	$2^{n-1}(2^{k+1}-1)$	2
	k digits	$2^n(2^k-1)$	2	$2^n(2^k-1)$	2
A5	Range contraction,	$c2^{n-k}$	$1 \leq w \leq \left\lfloor \frac{k}{2}+1 \right\rfloor$	$c2^{n-k}$	$1 \leq w \leq \left\lfloor \frac{k}{2}+1 \right\rfloor$
	k digits $(1 \leq c \leq 2^k-1)$	$2^{n-1-k}(2^{k+1}-1)$	2	$2^{n-1-k}(2^{k+1}-1)$	2
A6	Modulo N addition or	2^n-1-2^i	$2, (i=0, n-1)$	2^n-2^i	$1, (i=n-1)$
	modulo N subtraction		$3, (1 \leq i \leq n-2)$		$2, (0 \leq i \leq n-2)$
A7	Additive inverse (complementation)	2^i	1	2^i Also see (A6)	1
A8	Roundoff, k digits Also see (A6) for case (a)	2^k	1	2^k	1

Table I shows that error magnitudes of weights greater than one occur for a single basic fault. In (A2)–(A5) they assume the form $c2^i$, with $1 \leq c \leq 2^{k+1}-1$; that is, the nonzero digits in the error number are contained in at most $k+1$ adjacent positions. In modulo N addition and subtraction, every $|E| = 2^i$ with weight 1 has an associated $|E| = N - 2^i$, usually with weights 2 or 3. The origins of error values with weights greater than 1 are discussed next.

1) Arithmetic Shifts (A2, A3): These are subject to basic faults that affect the values of the end digits. In the k-digit right shift (A3) for both complement systems, the left-end digit x_{n-1} is replicated k times. A fault in x_{n-1} or the setting circuit affects $k+1$ left-end digits of the result, giving

$$|E| = \sum_{n-1-k}^{n-1} 2^i.$$

In the k-digit left shift (A2), k new digits are filled in at the right end. They are equal to x_{n-1} for $N_1 = 2^n - 1$, and they are zero for $N_2 = 2^n$. In both cases, a fault will generate

$$|E| = \sum_{0}^{k-1} 2^i.$$

2) *Range Extension and Contraction:* In the k-digit range extension (A4), k identical digits equal to x_{n-1} are attached at the left end. An incorrect value of x_{n-1} will give

$$|E| = \sum_{n-1}^{n-1+k} 2^i$$

a fault in the sensing circuit will give

$$|E| = \sum_{n}^{n-1+k} 2^i.$$

The value $|E| = 2^i$ $(n \leq i \leq n+k-1)$ occurs when one of the new digits is altered by a fault. The k-digit range contraction (A5) is the inverse operation, in which k identical digits $(x_{n-1}, \cdots, x_{n-k})$ are removed at the left end when they are equal to the leftmost remaining digit x_{n-k-1}. An incorrect removal gives $|E| = c2^{n-k}$, with $1 \leq c \leq 2^k - 1$. An incorrect value of x_{n-k-1} (e.g., 1 instead of 0) causes the removal of k identical digits (e.g., all 1s), giving

$$|E| = \sum_{n-k-1}^{n-1} 2^i.$$

3) *Modulo N Addition or Subtraction (A6):* This requires the "casting out" of N or of $-N$ from the sum or difference, respectively. A basic fault which locally generates $|E| = 2^i$ may cause an error in the "casting out," either by causing it unnecessarily, or by inhibiting it when it should take place. In both cases $|E| = N - 2^i$ occurs; its weight is 1, 2, or 3, depending on N and i.

4) *The Additive Inverse (A7):* This is the fixed subtraction $N - X$, called "complementation." For $N_1 = 2^n - 1$ it is the digitwise negation of x. For $N_2 = 2^n$, the negation of x is followed by the addition of 1 to the least significant digit, and the addition errors of (A6) may also occur.

5) *Roundoff (A8):* The roundoff of k digits $(i = 0, \cdots, k-1)$ is implemented by one of three methods: a) range test of $x_{k-1}, \cdots, x_0$ followed by addition of 0 or 1 to x_k; b) always setting x_k to 1; and c) truncation (without arithmetic). Cases a and b both may have $|E| = 2^k$; case a is also subject to the addition error of (A6).

B. Repeated-Use Faults in Binary Processors

Two classes of algorithms are subject to repeated-use faults: algorithms (A1)–(A8) of Table I in a byte-serial arithmetic processor, and multiplication and division in a parallel processor.

In a byte-serial processor, the kb digits long operands enter the processor in a sequence of k bytes, and the digit circuits are used k times. The length of each byte is $b > 1$ digits. The value of k is variable in some processors. A permanent local fault will affect the same relative position $h (0 \leq h \leq b-1)$ within each byte. The fault may be ineffective during some of the k uses.

Of the algorithms in Table I, byte-serial processing directly affects (A1)–(A3), (A6) and (A7). The error magnitudes 2^i and $N - 2^i$ are replaced by the sets of possible error magnitudes $\{|E_c|\}$ and $\{N - |E_c|\}$, with

$$E_c = \sum_{j=0}^{k-1} d_j 2^{bj+h}$$

where $d_j = 0$ if the fault is ineffective for the jth byte, $d_j = 1$ for an effective S1, and $d_j = -1$ for an effective S0. There are $2^k - 1$ nonzero magnitudes $|E_c|$ for a determinate (S0 or S1) local fault, and $(3^k - 1)/2$ nonzero magnitudes $|E_c|$ for an indeterminate (SX) local fault. Which one of the $2^k - 1$ or $3^k - 1$ nonzero sets of the coefficients d_j occurs is determined by the digit values of the operand or operands. An equal frequency of occurrence is assumed here. The arithmetic weights W are in the following ranges: 1) for $|E_c|: 1 \leq W \leq k$, 2) for $2^n - |E_c|: 2 \leq W \leq k+1$, and 3) for $(2^n - 1) - |E_c|: 2 \leq W \leq k+2$. The end-condition errors of the shifts (A2) and (A3), the range algorithms (A4) and (A5), and the roundoff (A8) do not differ from the parallel case $(k = 1, b = n - 1)$ and the results of Table I apply.

Parallel multiplication and division may be intolerably slowed down by the checking of individual additions and shifts, therefore, the repeated-use error magnitudes are of interest. It is assumed that the partial products or partial remainders are not checked, but returned to the accumulator as operands for the next step. The effect of a local fault in the digit circuits is cumulative, and different positions of the result are affected by successive steps because of the shifting. The set of expected error magnitudes is determined by the details of the algorithm.

Most readily susceptible to analysis are algorithms that employ fixed shifts of b bits. In this case the error numbers caused by a local fault in the digit circuits are the same as those developed during an addition or shift in a byte-serial processor with byte length b. End-condition setting in shifts, multiplier digit recoding, and quotient digit selection may contribute additional error values. More error values are also contributed by the multiple-forming circuits which shift the multiplicand (or divisor) left to obtain the multiples 2, 4, 8, etc. For example, a fault in the multiplicand register with provisions to add $c_j = 0, \pm 1, \pm 2$ times the operand to the partial product during the jth step affects one of two adjacent positions $(i, i+1)$ of the sum. The sets of possible error magnitudes are $\{|E|\}$ and $\{N - |E|\}$, with

$$E = \sum_{j=0}^{k-1} c_j d_j 2^{bj}$$

where b is the length of one right shift. The set of error magnitudes for any given variation of an algorithm and logic structure of the processor is obtained as the cumulative effect (sum) of appropriately shifted contributions of the error magnitudes in Table I.

III. Low-Cost Radix-2 Arithmetic Codes

A. Implementation of Arithmetic Error Codes

Arithmetic error codes are classified into separate and nonseparate codes [13]. Both classes possess many common properties, but differ significantly in their implementation.

The nonseparate code considered is the AN code [10], [11], which is formed when an uncoded operand x is multiplied by the *check modulus A* to give the coded operand Ax. The separate codes are the *residue* code [14], and the *inverse residue* code [7], [9] which is a previously unexplored variant of the residue code. The inverse residue code has significant advantages in fault detection of repeated-use faults. The modulo A inverse residue encoding for a number x attaches a check symbol x″ to form the pair (x, x″). The value of x″ is X'':

$$X'' = A - (A \mid X) = A - X'$$

where $A| X$ designates the modulo A residue of X. $A| X$ is the value X' of the check symbol x′ employed in modulo A residue encoding (x, x′). The inverse residue code is a separate code, since it has no arithmetic interaction between x and x″ [13], and should not be confused with the nonseparate systematic subcodes of AN codes [12], [13].

The set of undetectable error magnitudes $|E_m|$ (called *misses* in the subsequent discussion) for both AN and residue codes consists of all multiples of the check modulus A:

$$|E_m| = KA, \qquad K = 1, 2, \cdots, \lfloor (r^n - 1)/A \rfloor$$

for n-digit radix-r operands. The effectiveness and the cost of arithmetic checking depends very strongly on the choice of the check modulus A. The *checking algorithm* which establishes whether a detectable (or correctable) error exists in the result z for both classes of codes computes the modulo A residue $A|Z$, where Z is the unsigned integer value of z. The increase in word length is the same for both classes of codes. For radix-2 it is $\lceil \log_2 A \rceil$ bits.

The most significant differences of implementation are caused by the property of separateness. For residue codes, the operands x, y and their check symbols x', y' enter separate (*main* and *check*) processors which produce the main result z (value Z) and the check result z' (value Z'). The checking algorithm computes $A|Z$ and compares it to Z'. If the values are equal, either the correct result has been obtained, or a miss has occurred. Disagreement indicates a fault in either the main or the check processor; the uncertainty precludes fault location and error correction without supplementary procedures. An exception in the check procedure occurs for division $X \div Y$ which produces the quotient Q and the remainder P. The checking algorithm computes both $A| Q$ and $A| P$. The check processor computes the value $(A| Q)\cdot Y' + (A| P)$ which is compared to X' for equality [15]. The *inverse residue code* differs from the residue code in only one respect: the check result has the value $Z'' = A - (A|Z)$ when an error has not occurred. The checking algorithm computes $A|Z$ and forms the check sum $F = A|[(A|Z) + Z'']$, where $F = 0$ indicates that either the result is correct, or a miss has occurred.

For the nonseparate AN code the checking algorithm computes $A|Z$, where Z is the value of a result. $A|Z = 0$ indicates either a correct result or a miss. A nonzero $A|Z$ indicates a fault; for certain choices of A the value of $A|Z$ indicates the error value E for error correction [10], [11], [13]. The algorithms of the processor are designed to compute with product-coded numbers [6]. All intermediate steps of the algorithms must preserve product coding in order to retain the error-checking properties in the result. The hardware cost of AN codes is in the greater complexity of the main processor, while for residue codes it is in the separate check processor.

B. The Low-Cost Checking Algorithm

A practical checking method must satisfy both cost and effectiveness constraints. For radix-2 numbers, every odd integer $A > 1$ will detect weight 1 error magnitudes. The search for values of A which have a low-cost checking algorithm identified the class of *low-cost* arithmetic codes [3] which employ check moduli of the form

$$A = 2^a - 1, \qquad \text{with integer } a > 1.$$

The parameter a is called the *group length* of the code. Since division is a complex algorithm, the checking algorithms for most odd $A > 1$ are relatively costly and slow. The check modulus $2^a - 1$ is an exception because the congruence

$$K_i r^i \equiv K_i \text{ modulo } (r - 1), \qquad r \doteq 2^a$$

allows the use of modulo $2^a - 1$ summation of the k groups (a-bit segments of value K_i, with $0 \le K_i \le 2^a - 1$) that compose the ka-bit number Z to compute the *check sum* $(2^a - 1)|Z$. Division by A is replaced by an "end-around carry" addition algorithm, which "casts out $2^a - 1$'s" in a byte-serial or parallel implementation.

It is also important to note that the low-cost check moduli $2^a - 1$ are exceptionally compatible with binary arithmetic. A complete set of algorithms has been devised for AN-coded operands [3], [7], and an experimental byte-serial processor with four-bit bytes, $ka = 32$, $a = 4$, and $A = 15$ has been constructed for the STAR computer [1], [6]. While AN codes are limited to one's complement ($N = 2^n - 1$) algorithms, the two's complement ($N = 2^n$) algorithms can be carried out as well with the separate residue and inverse residue codes, which also display implementation advantages for multiple-precision algorithms. A set of algorithms for a two's complement inverse residue code processor (including multiple precision) has been developed to replace the AN code processor of the STAR computer [2].

C. Fault Effectiveness: One-Use Faults

It was already noted that the check moduli $2^a - 1$, with $a > 1$, will detect all weight 1 error magnitudes 2^j, with $0 \le j \le ka - 1$. Furthermore, all error values which can be confined within $a - 1$ adjacent bits of the error number (bursts of length $a - 1$ or less) will be detected, since their error magnitudes are $g2^j$, with g in the range $1 \le g \le 2^{a-1} - 1$. Only one error magnitude (out of $2^a - 1$ possibilities) confined within a adjacent bits is undetectable (that described by a adjacent 1's). This is important with respect to algorithms (A1)–(A5) of Table I, which contains error magnitudes of the forms $(2^k - 1)2^j$ and $(2^{k+1} - 1)2^j$. The choice of $a \ge k + 2$ will guarantee complete fault detection for these algorithms.

For operands of length $n = ka$ bits, the check modulus

2^a-1 will detect the one's complements $(2^{ka}-1)-|E|$ of all detectable error magnitudes $|E|$. Some weight 2 error magnitudes will not be detected; the undetectable error numbers are caused by one S1 and one S0 basic fault with a certain separation. The fraction f_2 of undetected weight 2 error magnitudes for $a>2$ is

$$f_2 = (k-1)a/[2a(ka-3)+6/k].$$

For $a>2, f_2<1/2a$ holds [3]. For example, given $ka=24$, $a=3$ yields $f_2=0.166$, $a=4$ yields $f_2=0.118$, and $a=6$ yields $f_2=0.071$. The case of $a=2$ is an unfavorable exception, yielding $f_2 \cong 0.5$ for any value of k. The analysis may be continued for higher weights, due to several independent basic faults; however, errors due to repeated use of a single faulty circuit are of more immediate interest.

D. Fault Effectiveness: Determinate Repeated-Use Faults

For the case of a determinate local repeated-use fault of Section II-B, which considers kb bits long operands processed in k bytes of b bits each, an analytic solution indicates very effective fault detection for the choice $b=a$ [4]. All possible 2^k-1 error magnitudes (and their one's complements) are detected by the check modulus 2^a-1 for $k<2^a-1$. Only one miss (undetectable error) occurs when $k=2^a-1$; the count of misses ϵ for $k\geq 2^a-1$ is given by the expression

$$\epsilon = \sum_{j=1}^{\lfloor k/(2^a-1)\rfloor} k!/[j(2^a-1)]![k-j(2^a-1)]!.$$

For example, the check modulus $\alpha=15$ with byte length $b=4$ allows no misses for words up to $n=56$ bits, and $\alpha=31$ with $b=5$ up to $n=150$ bits. The expressions for the miss count ϵ are derived by considering all possible ways in which result value 2^a-1 consisting of all ones can be generated by modulo 2^a-1 summation of k contributions of either 0 or 2^h, with $0\leq h\leq a-1$.

For any choice of the pair (a, b) and the word length $n=kb=ca$, it has been shown that the first miss occurs when the word length reaches the value

$$n' = c'a(2^{a/k'}-1)$$

where $c'a=k'b$ is the least common multiple of a and b [4]. Consequently, the maximum value of n' results when $k'=1$, giving $b=c'a$, and

$$n'_{\max} = c'a(2^a-1) = b(2^a-1).$$

The choice of $b=2a$ will double the "safe length"; for example, $\alpha=15$ and $b=8$ allows no misses for words up to 112 bits, and $\alpha=7$ and $b=6$ up to 36 bits. The minimum value of n' is obtained when a and b are relatively prime; in this case we have $n'_{\min}=ab$.

The effectiveness of any choice of the pair (a, b) can be expressed in terms of the percentage of misses among all possible 2^k-1 error magnitudes which can be caused by a local determinate fault. Given a miss count ϵ, the *miss percentage* is obtained as $100\,\epsilon/(2^k-1)$, where $n=kb$ is the word length of the operands. The miss percentages for various word lengths were obtained using a computer program which tabulated all misses for word lengths up to $k=18$ bytes, check lengths $2\leq a\leq 12$, and byte lengths $2\leq b\leq 10$ and $b=12$ [4]. The maximum word length of 18 bytes results in a total of $2^{18}-1=262\,143$ possible nonzero error magnitudes. In selected cases the maximum word length was extended to 20 bytes, i.e., $2^{20}-1=1\,048\,575$ possible nonzero error magnitudes. The miss percentages (for the same values of b) were also tabulated for eleven moduli A which detect all weight 2 and five check moduli which detect all weight 2 and 3 error magnitudes [11]. The word lengths used were n, with the requirement that 2^n-1 should be divisible by A.

The results of the tabulation (available in [4]) show that for a and b relatively prime, the percentage of misses rapidly becomes $100/(2^a-1)$ after the first miss which occurs at word length $n'=ab$ (the minimal case). For other pairs (a, b), the miss percentages beyond the word length n' tend to overshoot $100/(2^a-1)$ and then go below $100/(2^a-1)$ with increasing word length. The weight 2 and weight 2, 3 detecting check moduli A display miss percentages which are comparable to those of relatively prime (a, b).

E. Fault Effectiveness: Indeterminate Repeated-Use Faults

A local indeterminate fault (used m times) will contribute to the error magnitude in one of 3^m possible ways. During each use the contribution will be 0, 2^i, or -2^i with various values of i. For the same repeated-use model as used in the preceding section, the choice $b=a$, and the word length ka, the number of misses ϵ' due to the indeterminate fault (excluding the determinate subset) is given by the expression

$$\epsilon' = \sum_{j=1}^{\lfloor k/2\rfloor} k!/2[(k-2j)!](j!)^2.$$

The total count of possible nonzero error magnitudes is $(3^k-1)/2$. The miss percentage $100\,\epsilon'/2(3^k-1)$ is highest for $k=2$ and gradually decreases with increasing k. For values $k\geq 2^a-1$ the determinate subset contributes the miss count ϵ, and the total number of misses is $\epsilon+\epsilon'$. We also note that the value of ϵ' is independent of a. Table II lists the miss percentages (excluding the determinate subset) for the byte counts $2\leq k\leq 12$.

Given any pair (a, b), the first miss due to an indeterminate fault (excluding the determinate subset) occurs when the word length exceeds the least common multiple of a and b, that is, the first miss occurs for the word length n'', where

$$n'' > c'a$$

where $c'a=k'b$ is the least common multiple. Consequently, the maximum safe length n is attained for a and b relatively prime, with $n''_{\max}>ab$. In this case the first miss is due to the determinate subset and occurs for $n''=ab$. For other choices of the pair (a, b) we observe $n''_{\max}<n'_{\max}$.

The total miss percentages $100\,(\epsilon'+\epsilon)/2(3^k-1)$ are of interest in the cases $b\neq a$ as well. An exhaustive tabulation by means of a computer program was performed for word lengths up to $k=12$ bytes; that is, $(3^{12}-1)/2=265\,720$ nonzero error magnitudes were considered. The check lengths

TABLE II

k	$(3^k-1)/2$	ϵ'	Miss (percent)
2	4	1	25.00
3	13	3	23.08
4	40	9	22.50
5	121	25	20.66
6	364	70	19.23
7	1093	196	17.93
8	3280	553	16.86
9	9841	1569	15.94
10	29 524	4476	15.16
11	88 573	12 826	14.48
12	265 720	36 894	13.88

were again $2 \leq a \leq 12$, and the byte lengths were $2 \leq b \leq 10$ and $b=12$. It was observed that for relatively prime pairs (a, b) the miss percentages were close to $100/(2^a-1)$, becoming greater for pairs with common divisors, and reaching the maximal values of Table II for $b=a$ and $b=c'a$. Complete results of the tabulation are presented in [4].

It is noted that the most favorable choices of pairs (a, b) in the determinate faults are the least desirable choices for indeterminate faults, and vice versa. The choice of the most suitable values therefore depends on the relative frequencies of these two types of faults.

F. Repeated-Use Faults in Residue Codes

The results of the preceding sections on repeated-use faults apply directly to the fault effectiveness of the low-cost AN codes $(2^a-1)X$. The low-cost residue codes in the byte-serial processor suffer a serious disadvantage because of a new variety of an undetectable repeated-use determinate fault. The miss occurs when the check symbol x' of value $(2^a-1)|X$ uses the same digit circuits as the operand x. In this case, the fault affects the relative position h $(0 \leq h \leq b-1)$ in x' as well as in every byte of x, and a compensating error may occur. In the preferred choice $b=a$, the miss will occur whenever the position h in x' and *exactly one* position in x are altered by an S0 or S1 fault. For example, consider the modulo 15 residue encoding

$$x = 0010, 0011, 0101, \qquad x' = 1010.$$

An S1 fault sets the rightmost $(h=0)$ bit to 1 in every byte of x and in x' (boldface indicates changed bits) to give $x*$, $x'*$:

$$x* = 0011, 0011, 0101, \qquad x'* = 1011.$$

The checking algorithm yields $15|X* = 1011$ which is equal to $X'*$, and a "compensating miss" occurs which is independent of the length of x as long as only one byte in x is affected.

The compensating miss is eliminated by the use of the *inverse* residue code in which $X''=(2^a-1)-X'$ is substituted for X'. Consider the preceding example with the inverse residue $X''=1111-1010=0101$ replacing X'. The same S1 fault causes

$$x* = 0011, 0011, 0101, \qquad x''* = 0101.$$

The check yields $15|X* = 1011$; adding $X''*$ modulo 15 gives the result 0001 which indicates an error, since it is not equal to 1111.

The fault remains detectable even when one change each occurs in x and x'. Consider the previous example with a new operand y and its inverse residue y'':

$$y = 1000, 1101, 0101, \qquad y'' = 0100.$$

The check gives $15|Y=1011$, and $15|Y+Y''=1111$, i.e., no error. The previous S1 fault causes

$$y* = 1001, 1101, 0101, \qquad y''* = 0101.$$

The check gives $15|Y*=1100$ and $15|Y*+Y''*=0010$, indicating an error.

The compensating miss does not occur because the change $0 \rightarrow 1$ in y'' corresponds to the change $1 \rightarrow 0$ in y'. The first miss will occur when $y*$ consists of 14 bytes, each containing a zero in the rightmost position $n=0$, and y'' also has a zero in $h=0$. All results of the determinate fault effectiveness study are directly applicable to the low-cost inverse residue codes. This result led to the choice of modulo 15 inverse residue codes for both data words and address parts of instructions in the fault-tolerant STAR computer [2].

IV. Multiple Arithmetic Error Codes

A. Multiple Low-Cost Codes

The preceding section treated *single* codes which use only one check modulus. A study of fault-locating properties of the low-cost codes led to the observation that the use of *multiple* codes with two or more check moduli could provide complete fault location, corresponding to error correction [4], [7]. Continued study of multiple encodings has led to the development of several new varieties of arithmetic error codes, first discussed in [9].[1]

First it is shown that a single low-cost check modulus 2^a-1 has partial error-location properties in both AN and residues codes. Consider the error value pairs $(0 \leq i \leq ka-1)$:

$$\{2^i; -(2^{ka}-1)+2^i\} \quad \text{and} \quad \{-2^i; (2^{ka}-1)-2^i\}$$

that may be caused by a basic fault during a transfer or one's complement additive inverse, shift, or addition (the operand is ka bits long). Writing the value 2^i as a radix-2^a number, we have

$$2^i = 2^h 2^{ja}, \qquad h = i - ja.$$

The index $h=i-ja$ is called the *intra-group index* and j is called the *group index*. Their ranges are

$$0 \leq h \leq a-1 \quad \text{and} \quad 0 \leq j \leq k-1.$$

It is evident that

[1] Multiple arithmetic encodings have been recently described in [16] and [17]. It must be noted that the use of multiple check moduli for single-error correction was first described in [4, pp. 12–13], and in [7, pp. 36–37], and details were presented in [9], considerably prior to [16] and [17]. References [4], [7], and additional communication on the topic were supplied to Garcia at a UCLA short course in April 1968.

$$2^a \mid 2^h 2^{ja} = 2^a \mid [-(2^{ka} - 1) + 2^h 2^{ja}] = 2^h$$

$$2^a \mid (-2^h 2^{ja}) = 2^a \mid [(2^{ka} - 1) - 2^h 2^{ja}] = (2^a - 1) - 2^h.$$

The sign and the intra-group index h are uniquely identified for the error values $\pm 2^i$, even if the value of the end-around carry is incorrect due to the addition of $\pm 2^i$. The a-bit residue 2^h has a single 1 digit, and $(2^a-1)-2^h$ has a single 0 digit. For example, (with $h=3$, $a=4$) the residue is 1000 for the error 2^{3+4j}, and 0111 for -2^{3+4j}.

In the case of AN low-cost codes, the modulo 2^a-1 checking algorithm directly yields the check sum residues described above. In the case of residue low-cost codes, the main result X and the check result X' are computed. The checking algorithm must compute the a-bit check sum F:

$$F = (2^a - 1) \mid [(2^a - 1) \mid X + (2^a - 1) - X'].$$

A correct result (X, X') will yield the all ones form of $F=0$. It is readily shown that an erroneous main result $X \pm 2^i$ yields $F=(2^a-1)|(\pm 2^h)$, identical to the check sums of the AN code. An erroneous check result $(2^a-1)|(X' \pm 2^h)$ yields $F=(2^a-1)|(\mp 2^h)$, and the sign information becomes ambiguous: 1000 indicates the error $+2^{3+4j}$ in the main result, or the error -2^3 in the check result. The ambiguity is eliminated by the inverse residue codes which use $X''=(2^a-1)-X'$ as the check result. The check sum for the inverse residue code is

$$G = (2^a - 1) \mid [(2^a - 1) \mid X + X''].$$

When X'' is correct, $G=0$ is represented by the all ones form. An error in the main result X gives the same check sum as for the residue code. An erroneous check result has the value $(2^a-1)|(X'' \pm 2^h)$, which replaces X'' and yields the check sum $G=(2^a-1)|(\pm 2^h)$. Both the sign and the intra-group index h are known. The group index j remains unknown; it is also not known whether the check result or the main result is in error.

The preceding result has two applications. First, it has been used to derive the miss percentage equations for repeated-use faults in Section III. Second, it has led to the observation that the use of more than one low-cost check modulus will permit the unique identification of the bit index i of the error values $\pm 2^i$, and subsequent error correction, while using only the low-cost check moduli $2^{a_1}-1$, $2^{a_2}-1$, etc. [4], [7].

The check modulus $A^i=2^{a_i}-1$ has the group length of a_i bits. Given the pair (a_1, a_2) with GCD $(a_1, a_2)=1$, there will be $a_1 a_2$ distinct pairs of intra-group indices

$$\{h_1, h_2\}, \qquad 0 \le h_1 \le a_1 - 1, \qquad 0 \le h_2 \le a_2 - 1.$$

For example, $a_1=3$ and $a_2=4$ yield twelve pairs of indices:

$$h_1 = |\,2, 1, 0\,|\,2, 1, 0\,|\,2, 1, 0\,|\,2, 1, 0\,|$$
$$h_2 = |\,3, 2, 1, 0\,|\,3, 2, 1, 0\,|\,3, 2, 1, 0\,|.$$

The same observation applies to sets of three or more group lengths $\{a_1, a_2, \cdots, a_m\}$ which are pairwise prime. The length of the binary number for which distinct sets of intra-group indices $\{h_1, h_2, \cdots, h_m\}$ exist is p bits, while the encoding requires s bits, with

$$p = \prod_{i=1}^{m} a_i \quad \text{and} \quad s = \sum_{i=1}^{m} a_i.$$

For example, the choice of $a_1=3$, $a_2=4$, $a_3=5$ will give $p=3 \cdot 4 \cdot 5=60$ distinct sets of three intra-group indices with $s=3+4+5=12$ bits used for encoding [4], [7].

The effect of the m-tuple low-cost code with m pairwise prime group lengths $\{a_1, a_2, \cdots, a_m\}$ is the same as the effect of a code with a single check modulus 2^p-1 with respect to single-error correction and double-error detection for error values $\pm 2^i$ and $\mp(2^p-1-2^i)$ over $0 \le i \le p-1$. Burst-error detection is 100 percent effective for all bursts up to and including $s-1$ adjacent positions. Most important, the m separate low-cost checking algorithms are retained by an m-tuple low-cost code. One low-cost check is sufficient to detect the error values for which correction is possible; the other checks need to be performed only when an error is indicated and may share the same hardware.

Both AN and residue codes are suitable for multiple low-cost encoding. In the case of ordinary and inverse residue codes, the use of more than one check modulus resolves whether the error is in the main or in the check result: if only one check result indicates an error, it is incorrect; if all check results indicate an error, then it is traced to the bit i in the main result by the set of intra-group indices. The sign ambiguity of single residue codes is eliminated, and correction takes place either in the main result, or in the incorrect residue. An important difference between multiple low-cost residue and AN codes is the length of the uncoded information word. The nonseparate AN codes allow $p-s$ information bits, while the separate residue codes allow p information bits, with the s check bits added on as separate check symbols. Residue codes with the same number of check bits provide the same performance for a longer information word. The separateness of residue codes leads to a simpler design of the main processor which deals with uncoded operands, rather than with multiples of the check moduli which are used in the AN code processor.

The use of two or more low-cost check moduli permits multiple "mixed" low-cost encodings. A *mixed low-cost code* is a single or multiple low-cost AN code (p bits long) with a low-cost residue encoding (single or multiple) of the AN-coded words. Given the moduli $\{A_1, \cdots, A_m\}$, the mixed codes possess the same error-location properties as the corresponding *uniform* (AN or residue) multiple codes. For an example, consider the moduli $\{7, 15, 31\}$, with $a_1=3$, $a_2=4$, $a_3=5$. The uniform residue code has $p=3 \cdot 4 \cdot 5=60$ information bits and $s=3+4+5=12$ check bits. The uniform AN code has $p-s=48$ information bits encoded with $A=7 \cdot 15 \cdot 31=3255$; however, the checking algorithms remain separate modulo 7, 15, 31 low-cost checks. Six versions of the mixed code are available: three with double-residue encoding: (7, 15), (7, 31), (15, 31); and three with single-residue encoding: (7), (15), (31). In all six cases the AN-coded word must remain $p=60$ bits long; e.g., the AN code with $A_3=31$ has 55 information bits and 5 check bits, plus the 7 check bits of the double residue code with $A_1=7$, $A_2=15$. The error location algorithm uses the intra-group indices as in the uniform codes; an error in the main result is identified by the AN code check.

B. "Hybrid-Cost" Forms of Multiple Codes

In this section it is shown that the partial error-location property of the low-cost codes provides a low-cost extension of the range of other (non-low-cost) error-correcting codes. *Hybrid-cost* arithmetic error codes are multiple codes with a set of check moduli $\{A_1, A_2, \cdots, A_m\}$ which includes one or more low-cost check moduli A_i as well as one or more non-low-cost check moduli A_j' with the properties of error correction [10]–[13].

A hybrid-cost code (for example, the double code with moduli A, A'), offers two advantages over one error-correcting check modulus A'. First, the low-cost code (modulo 2^a-1) checking algorithm alone is sufficient to detect errors which are corrected by A'. Second, suitable choices of the pairs (A, A') permit the use of the intra-group index h of the low-cost code ($h=0, 1, \cdots, a-1$) to extend the range covered by A'. Given a single-error-correcting check modulus A' with the period of g bits, and the low-cost check modulus $A=2^a-1$ such that GCD $(g, a)=1$, it is evident that the intra-group index h extends the range of the hybrid-cost code to $p'=g\cdot a$ bits. For example, $A'=23$ gives distinct values of the residue $23|(\pm 2^i)$ for $0\leq i\leq 10$, $11<i<21$, etc., identifying uniquely the index i and the sign of $\pm 2^i$ for an 11-bit operand [10]. Together with $A=2^a-1$, the length for unique identification of the index and sign is $11a$ bits as long as GCD $(11, a)=1$ [9]. The use of $f\leq 2$ low-cost check moduli $(A_1, \cdots, A_f)$ with some A' will give the combined effect of the f-tuple low-cost code with the error-correcting properties of A', as long as the check moduli have pairwise GCD $=1$.

Three distinct classes of f-tuple hybrid-cost codes (with $f\geq 2$) can be identified: 1) uniform AN codes; 2) uniform residue codes; and 3) mixed (AN+residue) codes. The codes are similar to low-cost multiple codes described previously with the exception that one or more check moduli A_j' are non-low-cost. Differences between the three classes of codes appear in their implementation. The hybrid-cost AN codes $AA'X$ have the disadvantage of a costlier and slower implementation of arithmetic algorithms, since A' is not a low-cost check modulus. The hybrid-cost residue codes avoid these difficulties because they are separate. The use of more than one check modulus resolves the question whether the error is in the main or in the check result. In a double hybrid-cost residue code with the check moduli (A, A') the low-cost modulo A check is carried out each time for error detection. An error indication initiates the modulo A' check. If the latter does not indicate an error, then the modulo A check result is incorrect, and correction of the check result follows. If the modulo A' check result also indicates an error, then the main result is corrected, using both check results.

The mixed hybrid-cost codes have two major variants: 1) low-cost AN code with modulo A' residue encoding; 2) error-correcting $A'X$ code with modulo-A low-cost residue encoding. The first variant gives simple algorithms in the main processor, but must resolve the problem (existing also for hybrid-cost residue codes) of checking the error-correcting modulo A' residue if the modulo A' check is used only after detection using low-cost A. The second variant (preferably with inverse residue code) gives simple residue checking for error detection, but requires ccomplex algorithms in the main processor which operates on multiples of the non-low-cost check modulus A'. Other minor variants of mixed hybrid-cost codes are created when two or more check moduli are used for the AN part and/or the residue part. Each part, in turn, can be low cost or hybrid cost.

In conclusion it is noted that the use of multiple low-cost and hybrid-cost arithmetic encodings offers a variety of implementations. Fault location and error correction by means of multiple encodings employs the low-cost codes alone as well as to extend the range covered by error-correcting codes. It is also important to observe that multiple encodings permit the use of residue codes for error correction, since they distinguish whether the error is in the main result or in one of the check results. This information is not available with one residue and the generally less convenient nonseparate AN codes have to be used in single encodings. Detailed consideration of multiple encodings is presented in [9]. Finally, it should be noted that the concepts of multiple encoding (AN, residue, and mixed) are applicable to multiple nonlow-cost check moduli as well.

Acknowledgment

The author wishes to acknowledge stimulating discussions with D. A. Rennels, D. K. Rubin, J. J. Wedel, and A. D. Weeks of the Jet Propulsion Laboratory, Pasadena, Calif.

References

[1] A. Avižienis, "A experimental self-repairing computer," in *Information Processing 68, Proc. 1968 IFIP Congress*, vol. 2, A. J. H. Morrell, Ed. Amsterdam: North-Holland, 1969, pp. 872–877.
[2] A. Avižienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," this issue, pp. 1312–1321.
[3] A. Avižienis, "A set of algorithms for a diagnosable arithmetic unit," Jet Propulsion Lab., Pasadena, Calif., Tech. Rep. 32-546, Mar. 1, 1964.
[4] ——, "A study of the effectiveness of fault-detecting codes for binary arithmetic," Jet Propulsion Lab., Pasadena, Calif., Tech. Rep. 32-711, Sept. 1, 1965.
[5] ——, "Codes for fault detection in digital arithmetic processors," in *Information Processing 1965, Proc. IFIP Cong.*, vol. 2, W. A. Kalenich, Ed. Washington: Spartan, 1966, p. 634.
[6] ——, "The diagnosable arithmetic processor," Jet Propulsion Lab., Pasadena, Calif., Space Programs Summary 37-37, vol. IV, pp. 76–80, Feb. 1966.
[7] ——, "Concurrent diagnosis of arithmetic processors," in *Dig. 1st Annu. IEEE Comput. Conf.*, pp. 34–37, Sept. 1967.
[8] ——, "Application of codes in digital computer systems," presented at the 1967 Int. Symp. Inform. Theory, San Remo, Italy.
[9] ——, "Digital fault diagnosis by low-cost arithmetical coding

techniques," in *Proc. Purdue Centennial Year Symp. Inform. Processing*, vol. 1. Lafayette, Ind.: Purdue Univ. Eng. Exp. Sta., Apr. 28–30, 1969, pp. 81–91.
[10] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 333–337, Sept. 1960.
[11] W. W. Peterson, *Error Correcting Codes.* New York: Wiley, 1961, pp. 236–244.
[12] D. S. Henderson, "Residue class error checking codes," in *Preprints Papers, 16th Natl. Meet. Ass. Comput. Mach.*, 1961.
[13] H. L. Garner, "Error codes for arithmetic operations," *IEEE Trans. Electron. Comput.*, vol. EC-15, pp. 763–770, Oct. 1966.
[14] W. W. Peterson, "On checking an adder," *IBM J. Res. Develop.*, vol. 2, pp. 166–168, Apr. 1958.
[15] H. L. Garner, "Generalized parity checking," *IRE Trans. Electron. Comput.*, vol. EC-7, pp. 207–213, Sept. 1958.
[16] T. R. N. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Trans. Comput.*, vol. C-19, pp. 398–402, May 1970.
[17] T. R. N. Rao and O. N. Garcia, "Cyclic and multiresidue codes for arithmetic operations," *IEEE Trans. Inform. Theory*, vol. IT-17, pp. 85–91, Jan. 1971.

Proving Programs to be Correct

JAMES C. KING

Abstract—This paper formally describes a technique for proving that computer programs will always execute correctly. In order to do this, an abstract model for a program and its execution is given. Then, correctness of programs and proofs of correctness of programs are defined with respect to that model.

Index Terms—Abstract programs, debugging, program correctness, program execution model, state vector, theory of programming.

Introduction

THIS paper is concerned with proving that computer programs will always execute correctly. The motivation for this has existed since shortly after people began to write programs for computers. At that time, von Neumann and Goldstine presented ideas similar to those explained here [6]. Computers and the programs that make them operate are playing a more and more important role in our everyday lives. Some means for assuring that these programs will always run properly is critically needed.

A Model of Computation

A firm basis for discussing the correctness of programs is provided by establishing an abstract model for computations. The model is intended to characterize grossly the way most current digital computers perform their computations. A "program" is defined to be a finite ordered set of "statements" $s_1, s_2, \cdots, s_m$. A program operates on a fixed set of variables $x_1, x_2, \cdots, x_n$ that assume values from the domains $D_1, D_2, \cdots, D_n$, respectively. There are three types of program statements of the forms:

1) assign: $\{x_k, f(x_1, x_2, \cdots, x_n), j\}$;
2) test: $\{p(x_1, x_2, \cdots, x_n), j_1, j_2\}$;
3) halt.

The program variables can be interpreted to be storage cells of a computer memory and the three basic statement types are a means of summarizing the gross effect on that memory caused by a computer executing sequences of instructions.

The value of any one program variable (x_k) can be changed by the execution of an assign statement. The function f, possibly different for each assign statement, is a total function over the program variables $x_1, x_2, \cdots, x_n$ and results in a value in domain D_k which is used to replace the value of x_k. Each of j, j_1, and j_2 are integer constants between 1 and m (the total number of program statements) and are program statement subscripts. They occur in the program statements to indicate the order of execution of statements as explained later. Each test statement contains a total predicate $p(x_1, x_2, \cdots, x_n)$ (possibly different for each test statement) which results in *true* or *false* when evaluated for fixed values of the program variables.

A "state vector" of a program is an $(n+1)$-tuple of values $\langle N, a_1, a_2, \cdots, a_n \rangle$ where N is the statement subscript of the "next" statement to be executed ($1 \leq N \leq m$) and for $1 \leq i \leq n$ a_i is the value associated with the program variable x_i and as such must come from the domain D_i. An execution of a program $P = \{s_1, s_2, \cdots, s_m\}$ begins with an "initial state vector" $v_1 = \langle 1, a_1, a_2, \cdots, a_n \rangle$ and develops an "execution sequence" of state vectors $v_1, v_2, v_3, \cdots$. Sup-

Manuscript received March 1, 1971; revised June 11, 1971. This work was performed while the author was at the Carnegie-Mellon University, Pittsburgh, Pa., and is part of his Ph.D. dissertation [4]. This work was supported by the Advanced Research Project Agency of the Office of the Secretary of Defense (F44620-67-C0058).

The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y.