Arithmetic Algorithms for Error-Coded Operands

ALGIRDAS AVIŽIENIS

Abstract—A set of arithmetic algorithms is described for operands that are encoded in the "*AN*" error-detecting code with the low-cost check modulus $A = 2^a - 1$. The set includes addition, additive inverse (complementation), multiplication, division, roundoff, and two auxiliary algorithms: "multiply by $2^a - 1$," and "divide by $2^a - 1$." The design of a serial radix-16 processor is presented in which these algorithms are implemented for the low-cost *AN* code with $A = 15$. This processor has been constructed for the Jet Propulsion Laboratory STAR computer. The adaptation of "two's complement" arithmetic for an inverse-residue code is also described.

Index Terms—*AN* codes, arithmetic algorithms, arithmetic processor design, error-detecting codes, low-cost arithmetic error codes, residue codes.

I. Introduction

ARITHMETIC error codes are of interest to computer designers because they are preserved during arithmetic operations in the computer, and therefore can be applied to all computer words in order to attain concurrent error detection during the execution of programs. In addition to the theoretical properties of arithmetic error codes [1], [2] the designer must consider two additional aspects: the cost versus effectiveness tradeoffs of their application [3] and the implementation of the entire set of arithmetic algorithms of the computer using coded operands. This paper considers arithmetic algorithms for both major classes of arithmetic error codes: the *separate* (residue) and *nonseparate* (*AN*) codes [2]. Attention is concentrated on the low-cost radix-2 codes of both classes [3].

Algorithms for the residue codes have been developed as early as 1948 in the Raydac computer, which employed a variant of modulo 31 residue checking for its arithmetic unit [4]. Later, modulo 3 residue checking has been mentioned (without details) in descriptions of the IBM 7030 (Stretch) and the Univac III computers. Sets of algorithms were discussed by Garner [5], Davis [6], Sellers *et al.* [7], and Rao [8].

The first set of algorithms for the *AN* codes (with the low-cost check modulus $A = 2^a - 1$) has been devised for the STAR computer developed at the Jet Propulsion Laboratory [9], [10]. A complete 4-bit byte-organized arithmetic processor with $A = 15$ has been designed and constructed [11]. Experience with this processor led to the follow-on design of a second arithmetic processor that employed the separate modulo 15 inverse residue code [3], [12].

The nonseparate code considered in this paper is the *AN* code [1], 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 [1], and the *inverse-residue* code [3] which 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 given by

$$X'' = A - (A|X)$$

where $A|X$ means "the modulo A residue of X." $A|X$ is the value 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'', and therefore differs from the nonseparate systematic subcodes of *AN* codes [2].

The following sections present a set of algorithms for the low-cost (modulo $2^a - 1$) *AN*-coded operands and a discussion of their implementation in a byte-organized arithmetic processor. The last section discusses the algorithms for inverse-residue coded operands of the STAR computer.

II. Algorithms for *AN*-Coded Operands $(2^a - 1)X$

The entire set of binary arithmetical algorithms is conveniently adaptable to *AN* codes with the check modulus $2^a - 1$ and word length of ka bits [9], [11]. A fundamental constraint on the choice of A for binary n-bit *AN*-coded numbers $Z = AX$ is the requirement that the code should be complementable with respect to K(the values 2^n or $2^n - 1$) in order to implement the additive inverse algorithm. Assuming that the uncoded numbers X are complemented with respect to the integer value M, we have the requirement

$$K - AX = A(M - X)$$

which gives

$$K = AM,$$

that is, A must be a factor of K. Since A is an odd integer, this immediately excludes the choice $K = 2^n$. Consequently, a *complementable AN code* can be obtained if and only if

$$2^n - 1 = AM$$

is satisfied. All possible choices of A are available from a table of the prime factors of $2^n - 1$ [9]. The "$An + B$" code [1] allows other choices of A, but the algorithms are cumbersome to

Manuscript received January 31, 1973. 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 Department of Computer Science, University of California, Los Angeles, Calif. 90024, and the Jet Propulsion Laboratory, Pasadena, Calif. 91103.

implement. All codes with $n = ka$ and $A = 2^a - 1$ are complementable; for these codes the well-known "one's complement" algorithms apply directly, including complementation, sign detection, range extension, range contraction, left and right arithmetical shifting, and addition. In range extension the next allowed word length (after ka bits) is $(k+1)a$ bits and range is extended by prefixing a bits identical to the leftmost bit. Similarly, range can be contracted by dropping a leftmost bits when $a+1$ leftmost bits are identical. The existence of range extension and range contraction algorithms makes error-coded arithmetic with variable-length operands possible when a byte-organized processor is employed.

Multiplication and division of the AN-coded number Z by the check modulus $2^a - 1$ are needed in computing coded products and quotients. The "multiply by $2^a - 1$" algorithm requires one n-bit parallel modulo $2^n - 1$ addition. The $n - a$ bits-long coded operand Z is extended by a bits, shifted a positions left to form 2^aZ, and then added modulo $2^n - 1$ to its own (a bits extended) one's complement $\overline{Z}$ to form the n-bit result

$$(2^n - 1)|(2^aZ + \overline{Z}) = (2^a - 1)Z.$$

The algorithm is illustrated in Example 1.

Example 1–Multiply by $(2^a - 1) = 15$:

$$\begin{aligned} & & & & 1 = C_0 \\ 16X = & 0011 & 0111 & 0000 & \\ \overline{X} = & \underline{1111} & \underline{1100} & \underline{1000} & \\ 15X = & 0011 & 0011 & 1001. & \end{aligned}$$

The inverse operation, i.e., the "divide by $2^a - 1$" algorithm, generates the result $(2^a - 1)Z \div (2^a - 1)$ in bytes of a bits per step, employing an a-bit parallel adder to implement

$$Z = \overline{(2^n - 1)|\{(2^a - 1)Z + \overline{2^aZ}\}}.$$

The overbar designates one's complement and the addition is modulo $2^n - 1$, where n is the length of operand $(2^a - 1)Z$. The algorithm can be initiated because the "end-around" carry C_0 and the rightmost four bits (W_3, W_2, W_1, W_0) of $\overline{2^aZ}$ are deduced from the most significant bit Y_{n-1} of the operand $Y = (2^a - 1)Z$. The rule is as follows.

1) If $Y_{n-1} = 0$, then $C_0 = 0$ and $W_3W_2W_1W_0 = 1111$.
2) If $Y_{n-1} = 1$, then $C_0 = 1$ and $W_3W_2W_1W_0 = 0000$.

The algorithm is illustrated in Example 2.

Example 2–Divide by $(2^a - 1) = 15$:

$$\begin{aligned} Y = 15Z = & \ 0000 & 0010 & 0111 & 0100 & 0110 & 1011 & \\ W = \overline{16Z} = & \ 1111 & 1101 & 0110 & 0001 & 1010 & \boxed{1111} & = W_3W_2W_1W_0 \\ & \ \underline{\ 0\ } & \underline{\ 0\ } & \underline{\ 0\ } & \underline{\ 1\ } & \underline{\ 1\ } & \underline{\ 0\ } & = C_0 \\ \overline{Z} = & \ \underline{0}1111 & \underline{0}1111 & \underline{0}1101 & \underline{0}0110 & \underline{1}0001 & \underline{1}1010 & \\ Z = & \ 0000 & 0000 & 0010 & 1001 & 1110 & 0101. & \end{aligned}$$

A double-length coded product $(2^a - 1)XY$ of the n-bit operands $(2^a - 1)X$ and $(2^a - 1)Y$ is obtained by forming the one's complement binary product $(2^a - 1)^2XY$ and then dividing it by $2^a - 1$. The length of the result is $2n - a$ bits. A roundoff algorithm must be employed to obtain a single-length (n-bit) product.

Roundoff by truncation will not yield an AN-coded number; the algorithm must be modified. To round off the rightmost $n - a$ bits of the coded operand $(2^a - 1)XY$ we add the constant $(2^a - 1)G$, such that in $(2^a - 1)(XY + G)$ the rightmost $n - a$ bits are identical to the leftmost ("sign") bit and may be dropped. The "divide by $2^a - 1$" algorithm is applied to $(2^a - 1)XY$ to obtain the $n - a$ rightmost bits of XY. These bits are inspected and the value of G is selected to implement either up-rounding or down-rounding. The roundoff algorithm is illustrated in Example 3.

Example 3–Roundoff of Eight Bits: The eight rightmost bits of $XY = K = 1100\ 1011$. Roundup: add $15G$ where $G = 2^8 - K = 0011\ 0101$.

$$\begin{aligned} 15XY = & \ 0000 & 0010 & 1001 & 1110 & 0101 \\ 15G = & \ \underline{0000} & \underline{0000} & \underline{0011} & \underline{0001} & \underline{1011} \\ 15(XY + G) = & \ 0000 & 0010 & 1101 & 0000 & 0000. \end{aligned}$$

The division algorithm must produce two coded results: the quotient and the remainder. Given the $(2n - a)$-bit dividend $(2^a - 1)X$ and n-bit divisor $(2^a - 1)Y$, the n-bit quotient $(2^a - 1)Q$ is obtained by first forming $(2^a - 1)^2X$ and then executing binary division which yields the coded quotient $(2^a - 1)Q$ and the remainder $(2^a - 1)^2R$ satisfying

$$(2^a - 1)^2X = (2^a - 1)Q(2^a - 1)Y + (2^a - 1)^2R.$$

The coded remainder $(2^a - 1)R$ is obtained using the "divide by $2^a - 1$" algorithm. An uncoded quotient Q would result without the initial "multiply by $2^a - 1$" algorithm for the dividend. This initial algorithm may cause a special condition which is analyzed in the Appendix.

In summary, we note that the AN codes with the check modulus $2^a - 1$ display an exceptional adaptability to binary arithmetic and have a low-cost checking algorithm when the length of operands is chosen to be some multiple ka of the check length a. Byte-serial variable-length operand arithmetic can be implemented in bytes of a, $2a$, etc., bits. An experimental byte-serial processor using the above algorithms with $n = 32$, $a = 4$, and $2^a - 1 = 15$ has been constructed and tested as part of the STAR computer [11], [12]. It is described in the next section.

III. Design of a Processor for AN-Coded Operands

The operand precision of 28 bits was selected for the expected class of problems, and the check modulus $A = 15$ was chosen in order to provide 100 percent detection of single determinate repeated-use faults when binary numbers are transmitted and added in 4-bit bytes [3]. The length of coded numbers $15X$ is 32 bits. The choice of $A = 15$ provides 100 percent detection of such repeated-use faults for binary coded numbers up to 56 bits in length [3]. The STAR computer is a replacement system, and information is transmitted between its subsystems in 4-bit bytes in order to reduce the size of the data bus.

The block diagram of the AN-code arithmetical processor for the Star computer is shown in Fig. 1. The processor accepts five operation codes: clear add, add, subtract, multiply, and divide. One processor cycle consists of 10 byte times (clock

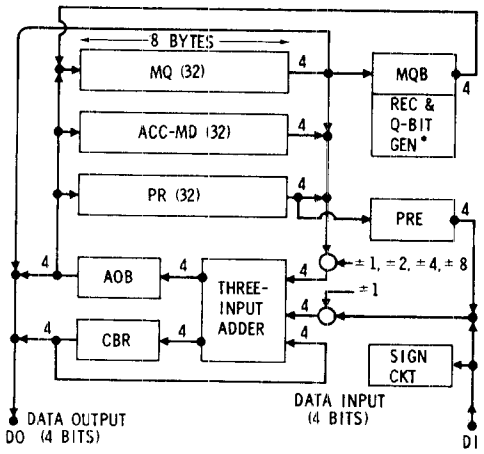

Fig. 1. *AN*-code arithmetic processor.

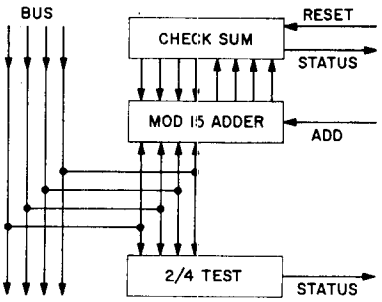

Fig. 2. Checker element of the STAR computer.

times). Operands are received on four data-input lines (DI), and results (both partial and final) are delivered on four data-output lines (DO). Partial results (partial sums, products, remainders) may be up to 10 bytes long, and the last byte is accompanied by a "perform check" signal on a control line. The final results (sum, difference, product, quotient) are always 8 bytes long and also have the perform check signal along with the eighth byte. They are followed by one nonnumerical condition-code byte, which employs a two-out-of-four encoding. There are three *singularity* codes: sum overflow, quotient overflow, and zero divisor, and three *result* codes: positive, zero, and negative. The latter three serve as branching information for "jump" class instructions. All partial and final results are sent to the *Checker*, which consists of a modulo 15 adder and accumulator, as shown in Fig. 2. Upon receipt of the perform check signal, the 4-bit check-sum accumulator is inspected for the "1111" ("all ones") check result. Any other check sum indicates that the result was not a properly *AN*-coded number, and a fault warning is issued. The Checker is outside the processor—there is one copy each in the three majority-voted test-and-repair processors [12].

Internally, the processor contains three 8-byte double-ranked shift registers (Fig. 1): the accumulator/multiplicand/divisor (ACC-MD) register, the product-remainder (PR) register, and the multiplier-quotient (MQ) register. There also are four 1-byte registers: adder output buffer (AOB), multiplier-quotient buffer (MQB), PR extension (PRE), and the carry-byte register (CBR). The *adder* is a three-input parallel adder that generates a *sum byte* (SB) and a *carry byte* (CB). The CB is stored in the 4-bit double-ranked CBR. The 8-bit output exists because the ACC-MD input to the adder supplies the multiples ±1, ±2, ±4, and ±8 of ACC-MD contents as operands for radix-16 multiplication and division. In this design all movement of numerical information takes place *four bits at once*, in the series parallel mode. At no point do two or more bits of the same byte pass through the same logic element; consequently, a fault will not damage more than one bit in one byte, and always the same bit position will be subject to damage when a sequence of bytes passes through the faulty element. The bit channels of the processor are physically isolated; for example, the ACC-MD, PR, and MQ shift registers consist of four 8-bit serial shift registers each. All data lines are 4 bits wide in Fig. 1. Multiplication and division are radix-16 operations; therefore, only 4-bit shifts are employed.

The *AN*-coded binary operands are 32 bits long. The one's complement form is used to represent negative numbers, and the binary point is assumed to be at the left end of the encoded 32-bit number, giving the range $-1/2 < Z < 1/2$ for any coded operand Z, and the range $-1/30 < X < 1/30$ for the uncoded operands X. This range was most convenient for the experimental model; other ranges can be readily employed by moving the binary point and appropriately adjusting the algorithms. However, the range $-1 < X < 1$ is not available for uncoded operands, since moving the binary point will scale the range of X by 2^j, and the jump in range will be from 16/30 to 32/30. The location of the binary point must be referred to the coded numbers in order to obtain simple sign detection and overflow detection algorithms. Zero is represented by a string of 32 ones; the all-zero number is not a properly coded number.

The *clear-add* algorithm needs one cycle for its execution. It replaces the contents of the ACC-MD by the input operand, which is added to zero and then stored in the ACC-MD. At the same time this sum is also returned on the output lines to the external Checker.

The *add* and *subtract* algorithms add the input operand or its digitwise complement, respectively, to the operand in the ACC-MD, store the sum in the ACC-MD, and also send it to the Checker. Additions that do not generate an end-around carry (eac) are completed in one cycle. If an eac is detected, the carry byte is sent to the Checker, a check is performed on this incomplete result, and a second cycle is employed to add the eac to the contents of ACC-MD. In case of overflow the "additive overflow" singularity code follows the result on the output lines.

The *multiply* algorithm requires that the multiplicand $15X$ should be already in the ACC-MD register; it is placed there by clear-add command, or is left there by the preceding operation. One cycle is employed to load the multiplier $15Y$ into the MQ register and to derive sign information for the product in a duplexed sign-test circuit. The multiplier is then sensed one byte at a time; consequently, there are eight radix-16 steps, each one consisting of from one to three cycles. The multiplier byte is recoded into a form possessing not more than two nonzero (±1) bits, and appropriate multiples (±1, ±2, ±4, ±8) of the multiplicand are added to the partial product in PR. Every partial result (nine bytes, plus the eac byte) is also delivered to

the Checker to check the addition. None, one, or two cycles may be required. Each step is concluded with a contraction-and-shift cycle for the 9-byte partial result P^* of the preceding additions. This cycle rounds the partial result P^* to a 9-byte result P (which is a multiple of 16) by subtracting a constant $15N_i$ to get

$$P = P^* - 15N_i = 15(16T)$$

since

$$P^* = 15(16T + N_i).$$

Simultaneously, P is shifted one byte right (divided by 16) and the value of N_i ($0 \leqslant N_i \leqslant 15$) is stored in one byte of MQ, which has delivered a multiplier byte for decoding and thus can accommodate the "correction byte" N_i. At the end of the eighth step, the PR contains the result

$$P^{(8)} = (15X)(15Y) - 15N = 15(15XY - N).$$

The terminal step is now performed to get an 8-byte rounded product P as a result. One cycle is used to divide $P^{(8)}$ by 15, and the 8-byte result

$$P^{(8)} \div 15 = 15XY - N$$

(which is usually not a multiple of 15) is returned to PR and the Checker. However, the value of N has been stored in the MQ and is now available to form the 16-byte result by attaching MQ at the right end of PR:

$$P_{\text{long}} = (15XY - N) + N.$$

During the next cycle, N is sent to the Checker and added to the check sum of $15XY - N$, which was not tested. Internally, the roundoff constant G is computed during this cycle from the bytes of N. The algorithm is concluded by adding the coded roundoff constant $15G$ to get the final 8-byte rounded result

$$P_{\text{rnd}} = (15XY - N) + N \pm 15G = 15(XY \pm G)$$

where G is the up- or down-rounding constant that would have been used for an uncoded product XY. Although one's complement is being used for subtractions, only nine bytes of the partial products were needed for the steps of the algorithm. The contract-and-shift cycle permits this time-saving variation of one's complement multiplication. The multiplication time for nonzero operands is

$$t_m = 1 + \sum_{i=0}^{7} (k_i + 1) + 3 \text{ cycles}$$

where k_i is the count of nonzero digits in the recoded ith multiplier byte. Since a multiplier can have as many as 16 and as few as two nonzero digits (one +1 and one -1) in its recoded form, the variation of t_m is

$$14 \leqslant t_m \leqslant 28 \text{ cycles.}$$

Zero operands are detected upon arrival and a zero result is delivered immediately, taking only two cycles.

The *divide* algorithm requires that the divisor $15Y$ should be already in the ACC-MD register. One cycle is used to load the single-length (8-byte) dividend $15X$ into the PR register and to record the signs in the duplex sign circuits. Tests for zero dividend (giving an immediate zero result) and zero divisor (a singularity) are performed during this cycle, and in either case only one additional cycle is needed to complete the algorithm. The divisor is returned to the Checker in case of a "zero divisor" singularity. In the nonzero cases the next cycle is used to form $15(15X)$, that is, to multiply the dividend by 15 in order to get a properly coded quotient $15Q$, such that

$$15^2X = (15Y)(15Q) + 15^2R$$

is satisfied, where R is the remainder in uncoded division. The quotient $15Q$ is then generated in eight radix-16 steps, one byte of the quotient at a time. The first cycle of each step is employed to finish restoring the remainder (if required by the preceding step) and to shift the remainder one byte left (multiplying by 16). Four cycles of quotient generation follow. First, the magnitude of the remainder is diminished by eight times the divisor. The sign of this result selects the leftmost bit of the new quotient byte and decides whether +4 (restoration) or -4 (further decrease) will be the next multiple of the divisor. If restoration is called for in the last cycle, it will be performed during the shift cycle of the next step. All partial results are returned to the PR register and also are sent to the Checker. Two cycles are needed for the terminal step: one completes the storage of the quotient into MQ, the other is used to move the quotient into ACC-MD and to deliver it to the Checker. The total time for division is

$$t_d = 2 + 5 \times 8 + 2 = 44 \text{ cycles.}$$

The "quotient overflow" singularity occurs whenever the allowed operand range $-1/30 < Q < 1/30$ is exceeded. This implies $|15Q| \leqslant 1/2$ as a requirement, and the test is an attempt (during the first step, second cycle) to generate a bit of value 1 for the leftmost position of the coded-quotient magnitude; if it succeeds, the quotient will overflow. This condition leads to an immediate termination of the algorithm (in one cycle), and an output of a "zero" value pseudoresult and an appropriate singularity code. The remainder of division is not made accessible in this experimental model of the processor, although it can be obtained from the contents of PR.

IV. Conversion to an Inverse-Residue Code Processor

The AN-coded arithmetic processor of the STAR computer displays two disadvantages of the AN codes.

Disadvantage 1: One's complement arithmetic must be employed which is less convenient than two's complement, due to eac, especially in a byte-serial processor.

Disadvantage 2: The use of the nonseparate AN code makes the implementation of multiple-precision and floating-point operations relatively cumbersome.

In view of these limitations, the decision was made to design an inverse-residue code processor for two's complement arithmetic as a replacement for the original AN-code processor. The algorithms for residue code processors were known to exist [4]-[8]. The principal difficulty was posed by the adaptation of two's complement representation of negative numbers.

In the residue code processor (Fig. 3), the operands (X, Y) enter the Main Processor, while the check symbols $(X', Y'$, or $X'', Y'')$ enter the Check Processor. The operations of the two processors must be independent. Such independence is readily attainable for one's complement, following the argument used in the discussion of AN codes. In a two's complement processor a special condition occurs when the two's complement (modulo 2^n) Main Processor addition discards a left-end carry-out C_n. This reduction of the sum by 2^n requires the modulo A reduction of the residue sum in the Check Processor by the constant $A|2^n$. This constant has the value 1 for $A = 2^a - 1$ and $n = ka$. (The inverse-residue sum must be increased by 1 modulo $2^a - 1$.) The "correction signal" $C_n = 1$ that is conveyed from the Main Processor to the Check Processor makes them interdependent and raises the concern about the possibility of compensating errors. To prove that such compensating errors cannot occur we consider two cases for which the "correction signal" value is in error.

Case 1–"Correction Signal" $C_n = 1$ is Caused by an Error: An incorrect carry-out $C_n = 1$ will occur only due to the addition of an error value 2^j which will cause a carry $C_{j+1} = 1$ when the adder positions to the left $(j + 1, \cdots, n - 1)$ all have the carry-propagate condition (input bit pairs 0, 1 or 1, 0). The error value is $-2^n + 2^j$, the correction signal compensates only for the -2^n part, and the $+2^j$ error value remains detectable.

Case 2–Correction Signal $C_n = 1$ is Inhibited by an Error: This case is symmetric to the previous one. An error value of -2^j has prevented the carry-out $C_{j+1} = 1$, and this in turn has inhibited the signal $C_n = 1$ which should have been sent. The result $X + Y$ of the Main Processor differs by $2^n - 2^j$ from the correct result, and $2^n - 2^j$ is a multiple of $2^a - 1$ for the values $j = 0, a, 2a, \cdots, (k - 1)a$ when $n = ka$. This is an undetectable error for $A = 2^a - 1$ in general; however, it will be detected for all j because of the following. Since $C_n = 0$ is sent, the check processor computes the check sum $(2^n - 1)|(X'' + Y'')$, while the main processor sum is $X + Y - 2^j$ due to the error. The error value -2^j has not been compensated for in the check sum, and it remains detectable.

In both cases neither the Main Processor result nor the Check Processor result are equal to the correct values, but it has been shown that the check algorithm will always produce a disagreement, and the occurrence of the error will be detected. This proof overcomes the principal objection to two's complement arithmetic. The algorithms of the inverse-residue code processor in general follow those developed for one's complement arithmetic. The preceding two cases are illustrated in Examples 4 and 5.

Example 4–Case 1:

$X'' = 1011$ $\quad X = 1111\ \ 0100$

$Y'' = \underline{0101}$ $\quad Y = \underline{0000\ \ 1010}$

$15|(X'' + Y'') = 0001$ ——— $0\ 1111\ \ 1110$ correct sum S

correct $C_n = \underline{\ \ 0\ \ }\leftarrow$ $\quad \underline{\ \ 1\ \ }$ error adds 2^4

correct $S'' = 0001$ ——— $1\ 0000\ \ 1110$ incorrect sum S^*

incorrect $C_n = \underline{\ \ 1\ \ }\leftarrow$

incorrect $S'' = 0010$ $\quad$ the error is detected.

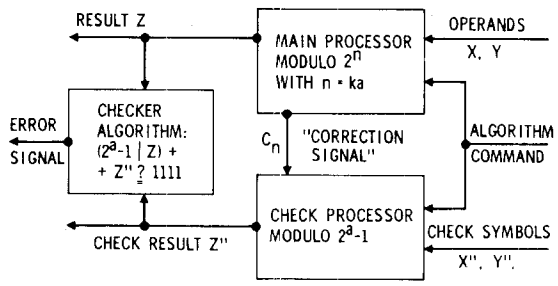

Fig. 3. Inverse residue code arithmetic processor.

Example 5–Case 2:

$X'' = 1011$ $\quad X = 1111\ \ 0100$

$Y'' = \underline{0100}$ $\quad Y = \underline{0001\ \ 1010}$

$15|(X'' + Y'') = 1111$ ——— $1\ 0000\ \ 1110$ correct sum S

correct $C_n = \underline{+1}\leftarrow$ $\quad \underline{-1}$ error adds -2^4

correct $S'' = 0001$ ——— $0\ 1111\ \ 1110$ incorrect sum

incorrect $C_n = \ \ 0\leftarrow$

incorrect $S'' = 1111$ $\quad$ the error is detected.

Appendix

The Division Anomaly

Given the dividend AX and the divisor AY, the restoring division algorithm will yield the results Q and AR such that

$$AX = Q \times AY + AR, \quad 0 \leqslant |AR| < |AY|, \text{ or } 0 \leqslant |R| < |Y|$$

is satisfied, and the sign of AR is the same as the sign of AX. The dividend A^2X and divisor AY should yield AQ and A^2R such that

$$A^2X = AQ \times AY + A^2R, \quad 0 \leqslant |A^2R| < |A^2Y|,$$

$$\text{or } 0 \leqslant |R| < |Y|$$

is the required condition. However, the direct application of the restoring algorithm will give

$$A^2X = Q^* \times AY + R^*, \quad 0 \leqslant |R^*| < |AY|.$$

The largest quotient Q^* that is computed may not be the required multiple AQ of A, but exceed it by the value K:

$$Q^* = AQ + K.$$

This, in turn, will make R^* less than the required multiple $A^2R = A^2X - AQ \times AY$ of A^2:

$$R^* = A^2X - AQ \times AY - K \times AY$$

such that

$$A^2R = R^* + K \times AY, \quad 0 \leqslant K < A.$$

In radix 2^a division with $A = 2^a - 1$ we have

$$0 \leqslant K \leqslant 2^a - 2$$

as the value of the "quotient overshoot" K.

The correction of the quotient consists of choosing the last quotient digit q_0 such that the quotient has the value AQ and the remainder A^2R has the same sign as the dividend A^2X. This means that q_0 can be in the range

$$-(2^a - 2) \leq q_0 \leq 1.$$

The only positive value of q_0 can be $q_0 = 1$, since the division criterion gives the last partial remainder $R(1)$ such that

$$0 \leq |R(1)| < |AY| \text{ or } 0 \leq 2^a \times |R(1)| < 2^a \times |AY|.$$

When $A = 2^a - 1$, we need to satisfy $|A^2R| < |A^2Y|$ and this can be attained with $q_0 \leq 1$ because of what preceded.

In the STAR computer implementation of division, the value of q_0 is obtained by computing the modulo $A = 15$ residue N of all preceding radix 16 quotient digits:

$$N = 15|(q_7 + \cdots + q_1).$$

The value $N = 0$ indicates $q_0 = 0$. When $1 \leq N \leq 13$, it is the value of the "quotient overshoot" and $q_0 = -N$ is selected. When $N = 14$, the test is made for $q_0 = 1$; if this changes the sign of the remainder, then $q_0 = -14$ is selected. A negative value of q_0 requires a subtraction to generate the final quotient AQ.

Acknowledgment

The logic design of the processors was performed and construction of the *AN*-code processor was directed by D. A. Rennels and A. D. Weeks from the Spacecraft Computers Section of the Jet Propulsion Laboratory. The author wishes to thank them for numerous stimulating discussions, as well as D. K. Rubin and J. J. Wedel of the JPL who also have contributed valuable advice and constructive criticism.

References

[1] W. W. Peterson, *Error Correcting Codes*. New York: Wiley, 1961, pp. 236–244.
[2] H. L. Garner, "Error codes for arithmetic operations," *IEEE Trans. Electron. Comput.*, vol. EC-15, pp. 763–770, Oct. 1966.
[3] A. Avižienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Trans. Comput.*, vol. C-20, pp. 1322–1331, Nov. 1971.
[4] R. M. Bloch, R. V. D. Campbell, and M. Ellis, "The logical design of the Raytheon computer," *Math. Tables and Other Aids to Computation*, vol. 3, pp. 286–295, 317–323, Oct. 1948.
[5] H. L. Garner, "Generalized parity checking," *IRE Trans. Electron. Comput.*, vol. EC-7, pp. 207–213, Sept. 1958.
[6] R. A. Davis, "A checking arithmetic unit," in *1965 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 27. Montvale, N.J.: AFIPS Press, 1965, pp. 705–713.
[7] F. F. Sellers, Jr., M.-Y. Hsiao, and L. L. Bearnson, *Error Detecting Logic for Digital Computers*. New York: McGraw-Hill, 1968, ch. 8, 10.
[8] T. R. N. Rao, "Error-checking logic for arithmetic-type operations of a processor," *IEEE Trans. Comput.*, vol. C-17, pp. 845–849, Sept. 1968.
[9] A. Avižienis, "A set of algorithms for a diagnosable arithmetic unit," Jet Propulsion Lab., Pasadena, Calif., Tech. Rep. 32-546, Mar. 1964.
[10] —, "An experimental self-repairing computer," in *Information Processing '68, Proc. IFIP Congr. 1968*, vol. 2, A. J. H. Morrell, Ed. Amsterdam: North-Holland, 1969, pp. 872–877.
[11] —, "The diagnosable arithmetic processor," *Space Programs Summary No. 37-37* (Jet Propulsion Lab., Pasadena, Calif.), vol. 4, pp. 76–80, Feb. 1966.
[12] 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," *IEEE Trans. Comput.*, vol. C-20, pp. 1312–1321, Nov. 1971.

Algirdas Avižienis (S'55–M'56–F'73) was born in Kaunas, Lithuania, on July 8, 1932. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana, in 1954, 1955, and 1960, respectively.

From 1955 to 1956 he was a Research Engineer at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena. During graduate studies at the Digital Computer Laboratory, University of Illinois, he was a Fellow in 1954–1955 and 1956–1958, and a Research Assistant in 1959–1960, participating in the design of the Illiac II computer. In 1960 he rejoined the Jet Propulsion Laboratory and initiated the JPL self-testing and repairing (Star) computer research project. In 1962 he joined the faculty of the School of Engineering and Applied Science at the University of California, Los Angeles. He is currently Professor and Vice Chairman in the Department of Computer Science and conducts research in digital system design and fault tolerance. He has also remained associated with the Jet Propulsion Laboratory as principal investigator of the STAR computer research project.

Dr. Avižienis is a member of Sigma Xi, Tau Beta Pi, Eta Kappa Nu, and the Association for Computing Machinery, and is the Chairman of the Technical Committee on Fault-Tolerant Computing and a member of the Governing Board of the IEEE Computer Society. He has served as Chairman of the First International Symposium on Fault-Tolerant Computing (1971) and as Program Chairman of COMPCON 1972. In 1968 he received the Computer Society Honor Roll award for organizing and chairing the first Workshop on the Organization of Reliable Automata. He received the NASA Apollo Achievement Award in 1969.