ON NUMERICAL ANALYSIS IN RESIDUE NUMBER SYSTEMS

by

George Edward Lindamood

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Arts
1964

APPROVAL SHEET

Title of Thesis: On Numerical Analysis in Residue Number Systems

Name of Candidate: George Edward Lindamood
Master of Arts, 1964

Thesis and Abstract Approved: Werner C. Rheinboldt

Werner C. Rheinboldt
Research Professor
Computer Science Center

Date approved: May 4, 1964

ABSTRACT

Title of thesis: On Numerical Analysis in Residue Number Systems

George Edward Lindamood, Master of Arts, 1964

Thesis directed by: Professor Werner C. Rheinboldt

Recent attempts to utilize residue number systems in digital computers have raised numerous questions about adapting the techniques of numerical analysis to residue number systems. Among these questions are the fundamental problems of how to compare the magnitudes of two numbers, how to detect additive and multiplicative overflow, and how to divide in residue number systems. These three problems are treated in separate chapters of this thesis and methods are developed therein whereby magnitude comparison, overflow detection, and division can be performed in residue number systems. In an additional chapter, the division method is extended to provide an algorithm for the direct approximation of square roots in residue number systems. Numerous examples are provided illustrating the nature of the problems

considered and showing the use of the solutions presented in practical computations. In a final chapter are presented the results of extensive trial calculations for which a conventional digital computer was programmed to simulate the use of the division and square root algorithms in approximating quotients and square roots in residue number systems. These results indicate that, in practice, these division and square root algorithms usually converge to the quotient or square root somewhat faster than is suggested by the theory.

Positions held: Associate Engineer
Westinghouse Electric Corporation
Baltimore, Maryland

Research Programmer
Computer Science Center
University of Maryland
College Park, Maryland

ACKNOWLEDGEMENTS

The author wishes to express his sincere gratitude to his advisor, Dr. Werner C. Rheinboldt, Director of the Computer Science Center, University of Maryland, and to Mr. George Shapiro, Computer Technology Group, Westinghouse Electric Corporation, for their encouragement, patience, and guidance during the preparation of this thesis.

A portion of the results presented in this thesis were obtained under the sponsorship of the Electronic Technology Laboratory, Aeronautical Systems Division, United States Air Force, under Contract Number AF 33(657)7899. The computational results obtained at the Computer Science Center, University of Maryland, were supported in part by the National Aeronautics and Space Administration under NASA Research Grant NsG 398.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

A. Residue Number Systems. Residue number systems, in which an integer x is represented by its residues with respect to one or more mutually prime moduli $m_0, m_1, \ldots, m_n$, were known to the ancient Chinese. In fact, the so-called "Chinese Remainder Theorem" was stated in a restricted form by Sun-Tsu in the First Century A.D. (See Dickson [3], pp. 57-64.) In modern terminology, the Chinese Remainder Theorem can be stated in the following form:

If $m_0, m_1, \ldots, m_n$ are mutually prime (positive) integers, the congruences

$$x \equiv a_i \pmod{m_i}, \quad i = 0,1,\ldots,n, \qquad (0.1)$$

have a unique simultaneous solution modulo $M = m_0 m_1 \ldots m_n$. If $M_i = M/m_i$ and x_i is the unique integer modulo m_i such that

$$x_i M_i \equiv 1 \pmod{m_i}, \quad i = 0,1,\ldots,n,$$

then x satisfies the congruences (0.1) simultaneously if and only if x is of the form

$$x_0 M_0 a_0 + x_1 M_1 a_1 + \ldots + x_n M_n a_n + kM,$$

where k is an integer.

The proof of this theorem can be found in most books on elementary number theory. (For example, see Hardy and Wright [10], pp. 94-95, or Griffin [8], pp. 79-80.)

If x satisfies the congruences (0.1) simultaneously, and y satisfies the congruences

$$y \equiv b_i \pmod{m_i}, \quad i = 0,1,\ldots,n,$$

simultaneously, then it follows from the above theorem that

$$z \equiv x \pm y \pmod{M}$$

if and only if

$$z \equiv c_i \equiv a_i \pm b_i \pmod{m_i}, \quad i = 0,1,\ldots,n,$$

and that

$$w \equiv x \cdot y \pmod{M}$$

if and only if

$$w \equiv d_i \equiv a_i \cdot b_i \pmod{m_i}, \quad i = 0,1,\ldots,n.$$

B. Modular Arithmetic Computers. It was the above property of multiplication in residue number systems which first prompted Miroslav Valach, Professor in the Institute of Mathematical Machines, Prague, Czechoslovakia,

to suggest a digital computer based on a residue number system. (See Valach [28].) In all digital computers then existing (1955), multiplication was performed by a technique of repeated addition and "shifting" which took several times as long as one addition. Valach and his colleague, Antonin Svoboda, recognized that, if a residue number system were used in a "modular arithmetic" computer, multiplication could be performed as fast as addition, so that the speed of computation would be increased appreciably for most problems.

At a conference in Darmstadt in 1955, Howard Aiken and Warren Semon, then Director and Assistant Director, respectively, of the Harvard Computation Laboratory, were introduced to the concept of modular arithmetic computers by Svoboda. Upon their return to the United States, Aiken and Semon began their own investigation of the application of residue number systems to digital computers, and in 1956 they submitted a report on their work to the Wright Air Development Center, Wright-Patterson Air Force Base, Ohio. (See Reference [1], a revised version of that report.) As a result, the United States Air Force became sufficiently interested in modular arithmetic computers to support considerable research into their design and use. Among those funded by the Air Force

for such studies were: Aiken and Semon and their staff at the Harvard Computation Laboratory (see References [4] and [11]); Harvey Garner and his associates at the University of Michigan (see References [5] - [7], [13], and [18]; Lockheed Missiles and Space Company, Sunnyvale, California (see References [2], [15], [17], and [27]); Scope, Incorporated, Falls Church, Virginia (see Reference [19]); and Westinghouse Electric Corporation, Baltimore, Maryland (see References [16], [20], and [32]). It was at Westinghouse in 1962 that the author became interested in the problems involved in adapting residue number systems for use in modular arithmetic computers and it was there that he began the investigation which eventually let to this thesis.

C. Problems in Modular Arithmetic Computer Design.

The problems encountered by the investigators of modular arithmetic computers are of two types: first, those concerned with the "logical" organization of such computers and the development of the attendant circuitry; and second, those concerned with the theoretical difficulties in performing numerical analysis in residue number systems. The problems of the first type are the usual problems associated with

the design of any new computer, except that using residue number systems promises several interesting possibilities for more economical logical design than that in conventional computers. (An exploration of some of these possibilities will be the subject of a thesis by Robert L. Beadles which will be submitted to the University of Pittsburgh in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.) The theoretical problems, on the other hand, are more acute in that, if they are not solved, modular arithmetic computers will be unable to perform several very fundamental operations and therefore will be incapable of handling a large class of computational problems.

The purpose of this thesis is to present solutions to some of these theoretical problems in modular arithmetic computer design. In particular, this thesis treats the problems of how to compare the magnitudes of two numbers, how to detect overflow resulting from addition and multiplication, how to divide, and how to take square roots in residue number systems. In Chapters I - IV below, each of these problems is discussed in turn. Solutions are given, along with appropriate proofs, and examples are included

Also, specific mention should be made of those men whose ideas directly influenced the form of the results contained in this thesis. First, credit for suggesting various facets of the magnitude comparison methods described in Chapter I should be given to Garner [7], H.S. Shapiro [22], and Valach [30]. By using their ideas, it remained only for the author to combine their methods into a single, systematic approach and to provide the necessary proofs. Next, recognition should be given to the author's former supervisor at Westinghouse Electric Corporation, Mr. George Shapiro, who suggested using a table of powers of two (stored within the computer) in performing multiplicative overflow detection, division, and square root extraction and who also suggested that quotients and square roots in residue number systems be approximated by the quotient of an integer and an integral power of two. By following these suggestions, it was not too difficult for the author to work out the overflow detection, division, and square root procedures given in Chapters II, III, and IV, respectively. Last, credit is due to the author's thesis advisor, Dr. Werner Rheinboldt, for encouraging the author to investigate the practical behavior of the division and square root algorithms by programming the University of

Maryland's IBM 7090 computer to simulate modular arithmetic computers in performing divisions and square root extractions by these methods. By using these simulation programs, several thousand "sample" divisions and square root extractions were completed in a matter of minutes.

Finally, it should be added that, while the algorithms given in this thesis are rather long and complicated, they are, to the best of the author's knowledge, the most efficient solutions yet obtained for the problems considered. That is, judging from estimates of the number of operations required, these algorithms seem to use less computer time for their execution than the other existing solutions and - what is more important when one is designing a computer - they appear to require no special circuitry for their implementation, since they rely heavily on "standard" computer operations such as addition and "bit testing." Thus, it is the author's hope that these methods for comparing magnitude, detecting overflow, dividing, extracting square roots will comprise a contribution to the adaptation of residue number systems for use in digital computers and that this thesis will help remove a barrier in making modular arithmetic computers usable for general types of computation.

CHAPTER I

MAGNITUDE COMPARISON

A. Ordering in Residue Number Systems. Since most computer applications involve some use of the order properties of the real numbers, magnitude comparison is an essential operation in all digital computers. In conventional digital computers, magnitude comparison is performed simply by a sequence of "bit tests" which is the logical equivalent of the usual method of comparing two integers. (See Theorem 1.2 below.) In modular arithmetic computers, however, magnitude comparison must be performed in a residue number system where such operations are not so simple. To show that this is the case, let us consider several examples.

Given a residue number system with moduli $m_3 = 7$, $m_2 = 5$, $m_1 = 3$, and $m_0 = 2$, suppose we wish to find the smallest of the three "numbers"

$$\{3, 4, 2, 1\} \; ; \quad \{1, 3, 0, 0\} \; ; \quad \{5, 3, 1, 1\}$$

in this system. (Here the "number"

$$\{3, 4, 2, 1\}$$

represents the integer x such that

$$x \equiv 3 \pmod 7; \qquad x \equiv 4 \pmod 5;$$

$$x \equiv 2 \pmod 3; \qquad x \equiv 1 \pmod 2.$$

From the Chinese Remainder Theorem we know there is exactly one solution, namely

$$x = 59,$$

which satisfies these congruences and the condition

$$0 \leqslant x < 210 = m_3 m_2 m_1 m_0.$$

Hence, we write

$$\{3, 4, 2, 1\} \approx 59. \quad)$$

Since each of the residues in the number

$$\{1, 3, 0, 0\}$$

is less than or equal to the corresponding residues in the other two numbers, we might expect that

$$\{1, 3, 0, 0\}$$

is the smallest of the three. Our expectations are wrong, however, since

$$\{1, 3, 0, 0\} \approx 78$$

<u>is</u> smaller than

$$\{5, 3, 1, 1\} \approx 103,$$

but <u>not</u> smaller than

$$\{3, 4, 2, 1\} \approx 59.$$

Next, we might try ordering these three numbers "lexicographically;" that is, we might order the numbers by ordering their "first" residues (those with respect to m_3), then their "second" residues (those with respect to m_2), and so forth. But this ordering would give the result

$$\{1, 3, 0, 0\} < \{3, 4, 2, 1\} < \{5, 3, 1, 1\}$$

or equivalently,

$$78 < 59 < 103,$$

which is obviously wrong. Similarly, "reverse lexicographic" ordering, in which the "last" residues (those with respect to m_0) are ordered first, would give

$$\{1, 3, 0, 0\} < \{5, 3, 1, 1\} < \{3, 4, 2, 1\}$$

or

$$78 < 103 < 59.$$

More counter-examples can be found to show that other ordering schemes on the residues are equally unsuccessful.

To examine another aspect of this problem, let us consider the numbers

$$\{2, 3, 1, 0\} \; ; \; \{4, 0, 0, 0\} \; ; \; \{3, 4, 2, 1\}$$

in the same residue number system as before. Upon observing that each of the residues in the first number are exactly one less than the corresponding residues in the

last, we might conclude (correctly) that

$$\{2, 3, 1, 0\} \approx 58$$

represents the "next smaller" integer than

$$\{3, 4, 2, 1\} \approx 59.$$

(By definition, all numbers in a residue number system represent integers.) But it is not so obvious from their residues that

$$\{4, 0, 0, 0\} \approx 60$$

is the "next number greater" than

$$\{3, 4, 2, 1\} \approx 59$$

in this system. Furthermore, this problem becomes even more difficult when we consider the "second number greater" and so forth.

In our third and final example, suppose we provide for negative numbers in the above residue number system by decreeing that all integers x such that

$$M/2 = 105 < x < 210 = M = m_3 m_2 m_1 m_0$$

be regarded as representing the negative integers -104 through -1. The rule of correspondence is

$$x \longleftrightarrow x - M$$

(That is, we restrict our residue number system to the M consecutive integers -104 through 105 instead of the

integers 0 through 209 used in the two preceding examples.) Suppose we now wish to determine the signs of the numbers

$$\{4, 1, 1, 1\} \quad \text{and} \quad \{4, 3, 2, 1\}$$

in this system; that is, we wish to determine whether these numbers are greater or less than

$$\{0, 0, 0, 1\} \approx 105$$

in the "old" residue number system. It is not at all clear from the residues in these numbers that

$$\{4, 1, 1, 1\} \approx 151 \longleftrightarrow -59$$

is negative, while

$$\{4, 3, 2, 1\} \approx 53$$

is positive. Hence, it seems that the residues in a number cannot even be trusted to tell us whether or not that number is positive. In fact, about all they can be trusted to tell us is whether or not the number is zero, since a number is zero if and only if all its residues are zero.

As these examples clearly show, there is no obvious ordering scheme for the residues in these numbers which agrees with the "natural" ordering of the integers represented or which gives any significant information about

the signs of those integers. Therefore, our first problem is to devise some other method for using the residues in two numbers in a residue number system to determine which number represents the larger or smaller integer. The solution we shall give is based upon a generalization of the "positional notation" commonly used for the integers themselves.

B. Mixed-Radix Notation. As is well known in mathematics, we may represent any non-negative integer x in the form

$$x = a_n r^n + a_{n-1} r^{n-1} + \ldots + a_1 r + a_0, \qquad (1.1)$$

by using any integer $r > 1$ as a "radix." In referring to this representation for x, we usually indicate the radix by using an appropriate adjective, such as "decimal" or "octal", and mention only the coefficients a_i in a given order, say, $a_n a_{n-1} \ldots a_1 a_0$. Moreover, if we require those coefficients to be integers satisfying

$$0 \leqslant a_i < r, \quad i = 0, 1, \ldots, n,$$

then the above representation is unique. (That is, there is exactly one such representation for every non-negative integer x.) Clearly, this representation may be extended to negative integers by prefixing the entire representation with a minus sign.

If y is another non-negative integer satisfying

$$y = b_n r^n + b_{n-1} r^{n-1} + \ldots + b_1 r + b_0 , \quad (1.2)$$

the coefficients b_i again being integers such that

$$0 \leqslant b_i < r, \quad i = 0, 1, \ldots, n,$$

then we may compare the magnitudes of x and y by comparing their coefficients a_i and b_i, respectively, in "lexicographic" order. That is, we first compare a_n and b_n; if $a_n = b_n$, we compare a_{n-1} and b_{n-1}; and so forth until either we reach $a_0 = b_0$ or we find an index j such that $a_j \neq b_j$. In the former case, x and y are obviously equal. In the latter case, x and y are unequal, and if j is the <u>largest</u> index i such that $a_i \neq b_i$, then $x > y$ if and only if $a_j > b_j$.

Let us now formally summarize these properties of this notation - commonly called "positional notation" - by stating two theorems:

<u>Theorem 1.1</u> (Uniqueness of Representation) - <u>If the coefficients</u> a_i <u>in expression</u> (1.1) <u>are integers satisfying</u>

$$0 \leqslant a_i < r, \quad i = 0, 1, \ldots, n,$$

<u>then they are uniquely determined by</u> x.

* * *

It follows from this theorem that $a_i \neq b_i$ for some index i implies that $x \neq y$, since otherwise $a_n a_{n-1} \ldots a_1 a_0$ and $b_n b_{n-1} \ldots b_1 b_0$ would be distinct representations of the same integer. Hence, $x \neq y$ if and only if $a_i \neq b_i$ for at least one index i.

* * *

Theorem 1.2 (Magnitude Comparison) - If x and y are unequal non-negative integers satisfying (1.1) and (1.2) respectively, if all a_i's and b_i's are non-negative integers less than r, and if j is the largest index i such that $a_i \neq b_i$, then $x > y$ if and only if $a_j > b_j$.

* * *

The proofs of these theorems can be obtained quite easily with the aid of the following lemma:

Lemma 1.1 - If the coefficients a_i in expression (1.1) are integers satisfying

$$0 \leqslant a_i < r, \quad i = 0, 1, \ldots, n.$$

then

$$r^j > a_{j-1} r^{j-1} + a_{j-2} r^{j-2} + \ldots + a_1 r + a_0$$

holds for (1.1) whenever j is any integer such that

$$0 < j \leqslant n+1.$$

* * *

Since both these theorems and this lemma are widely known to be true and since their proofs can be found in numerous books on real analysis, we state them here without proof. However, we shall soon give these proofs for a more general notation when we state and prove Lemma 1.2 and Theorems 1.3 and 1.4.

Let us now broaden somewhat the scope of these theorems by extending them to apply to a more general notation. In particular, let us replace the radix r in (1.1) with several radices $r_n, r_{n-1}, \ldots, r_0$, all of which are integers greater than one, and let us rewrite (1.1) in the form

$$x = a_n r_{n-1} r_{n-2} \cdots r_0 + a_{n-1} r_{n-2} r_{n-3} \cdots r_0 + \cdots + a_1 r_0 + a_0, \tag{1.3}$$

where the coefficients a_i are integers such that

$$0 \leqslant a_i < r_i, \quad i = 0, 1, \ldots, n.$$

The representation of the integer x by the coefficients a_i obtained in this manner is called "mixed-radix notation" in contrast with the "fixed-radix notation" associated with (1.1) above. This notation has essentially the same properties as those given in the above lemma and theorems for fixed-radix notation. Indeed, Lemma 1.1 and Theorems 1.1 and 1.2 are but restricted versions - for the special case

in which all the r_i's are equal to r - of the following lemma and theorems.

* * *

Lemma 1.2 - If the coefficients a_i in expression (1.3) are integers satisfying

$$0 \leqslant a_i < r_i, \qquad i = 0, 1, \ldots, n,$$

then

$$r_{j-1}r_{j-2}\cdots r_0 > a_{j-1}r_{j-2}r_{j-3}\cdots r_0 + \cdots + a_1 r_0 + a_0$$

holds for (1.3) whenever j is any integer such that

$$0 < j \leqslant n+1.$$

Proof: If $0 < j \leqslant n+1$, it follows from the conditions on the coefficients a_i that

$$\begin{aligned} r_{j-1}r_{j-2}\cdots r_0 &> r_{j-1}r_{j-2}\cdots r_0 - 1 \\ &= (r_{j-1}-1)r_{j-2}r_{j-3}\cdots r_0 + \cdots \\ &\quad + (r_1 - 1)r_0 + (r_0 - 1) \\ &\geqslant a_{j-1}r_{j-2}r_{j-3}\cdots r_0 + \cdots \\ &\quad + a_1 r_0 + a_0. \end{aligned}$$

This is the desired result.

* * *

Theorem 1.3 (Uniqueness of Mixed-Radix Representation) - Under the conditions given in Lemma 1.2, the coefficients a_i in expression (1.3) are uniquely determined by x.

Proof: Assume that both (1.3) and

$$x = c_n r_{n-1} r_{n-2} \cdots r_0 + c_{n-1} r_{n-2} r_{n-3} \cdots r_0 + \cdots$$
$$\cdots + c_1 r_0 + c_0$$

are expressions for x such that the same radices r_i are used in both expressions and such that the a_i's and c_i's are non-negative integers less than r_i for $i = 0, 1, \ldots, n$. Assume also that $a_i \neq c_i$ for some index i and let j be the largest such index. Then,

$$(c_j - a_j) r_{j-1} r_{j-2} \cdots r_0 + (c_{j-1} - a_{j-1}) r_{j-2} r_{j-3} \cdots r_0$$
$$+ \cdots + (c_1 - a_1) r_0 + (c_0 - a_0)$$
$$= x - x$$
$$= 0.$$

Let us now assume without loss of generality that $a_j < c_j$. Then, it follows from the above equation that

$$r_{j-1} r_{j-2} \cdots r_0 \leq (c_j - a_j) r_{j-1} r_{j-2} \cdots r_0$$
$$= (a_{j-1} - c_{j-1}) r_{j-2} r_{j-3} \cdots r_0$$
$$+ \cdots + (a_1 - c_1) r_0 + (a_0 - c_0)$$
$$\leq a_{j-1} r_{j-2} r_{j-3} \cdots r_0 + \cdots$$
$$\cdots + a_1 r_0 + a_0,$$

since the a_i's and c_i's are non-negative integers. But this contradicts Lemma 1.2. Therefore, $a_i = c_i$ must hold for $i = 0, 1, \ldots, n$, which is the desired result.

* * *

Theorem 1.4 (Magnitude Comparison in Mixed-Radix Notation) - Let x and y be distinct non-negative integers such that x satisfies (1.3) and y satisfies

$$y = b_n r_{n-1} r_{n-2} \cdots r_0 + b_{n-1} r_{n-2} r_{n-3} \cdots r_0 + \cdots + b_1 r_0 + b_0, \tag{1.4}$$

where the r_i's are the same in (1.3) and (1.4) and the a_i's and b_i's are non-negative integers less than r_i for $i = 0, 1, \ldots, n$. If j is the largest index i such that $a_i \neq b_i$, then $x > y$ if and only if $a_j > b_j$.

Proof: In the light of Theorem 1.3, it is obvious from the assumption that $x \neq y$ that j exists and that it will be sufficient to show that $a_j < b_j$ implies $x < y$ and $a_j > b_j$ implies $x > y$.

If we assume that $a_j < b_j$, then it follows from Lemma 1.2 that

$$\begin{aligned}(b_j - a_j) r_{j-1} r_{j-2} \cdots r_0 &\geqslant r_{j-1} r_{j-2} \cdots r_0 \\ &> a_{j-1} r_{j-2} r_{j-3} \cdots r_0 + \cdots \\ &\quad \cdots + a_1 r_0 + a_0 \\ &\geqslant (a_{j-1} - b_{j-1}) r_{j-2} r_{j-3} \cdots r_0 \\ &\quad + \cdots + (a_1 - b_1) r_0 \\ &\quad + (a_0 - b_0),\end{aligned}$$

since the a_i's and b_i's are non-negative integers.

Therefore,

$$y - x = (b_j - a_j)r_{j-1}r_{j-2}\cdots r_0 + (b_{j-1} - a_{j-1})r_{j-2}r_{j-3}\cdots r_0 + \ldots + (b_1 - a_1)r_0 + (b_0 - a_0) > 0,$$

which is equivalent to $x < y$.

Similarly, if $a_j > b_j$, we need only interchange the a_i's and b_i's in the above expression to obtain $x - y > 0$. Hence, the proof is complete.

* * *

As in the fixed-radix notation associated with (1.1), negative integers may be represented in mixed-radix notation by placing a minus sign before the entire representation. However, another representation, in which the minus sign is replaced by the use of both positive and negative coefficients, suggests itself. In particular, let us require as before that the radices r_i be integers greater than one and that the coefficients a_i in (1.3) be integers. But now let the a_i's satisfy

$$|a_i| < r_i/2, \tag{1.5a}$$

if r_i is an odd integer, and

$$-r_i/2 < a_i \leqslant r_i/2, \tag{1.5b}$$

if r_i is an even integer. (There is no reason why we couldn't have a_i satisfy

$$-r_i/2 \leqslant a_i < r_i/2$$

instead of (1.5b) above, when r_i is even. If we did this, a few "$<$" and "$\leqslant$" signs would have to be interchanged in Lemma 1.3 and Theorems 1.5 and 1.6 below, but the lemma and theorems themselves would remain essentially intact. There is, however, a slight advantage in our using the restriction (1.5b), but we shall postpone our explanation of it until we have proved Lemma 1.3 below.) With these new conditions on the coefficients a_i, the notation resulting from (1.3) is called "two-sided mixed-radix notation."

In order that this new notation retain the desirable uniqueness and ordering properties of "ordinary" mixed-radix notation set forth above in Theorems 1.3 and 1.4, it is necessary to require that <u>at most one</u> of the radices r_i be even. (Since it will soon become necessary for those radices to be <u>mutually prime</u>, the restriction of at most one even radix is a natural one and certainly retains sufficient generality for our purposes.) Furthermore, to insure that the numbers representable in

two-sided mixed-radix notation are distributed "symmetrically" about zero, it is also necessary to stipulate that, if any of the radices *is* even, it be designated r_0.

We shall now show that integers represented in this two-sided mixed-radix notation may be compared exactly as in their "ordinary" positional notation; *i.e.*, by comparing their "coefficients" in lexicographic order.

* * *

Lemma 1.3 - *For any integer* j *such that*

$$0 < j \leqslant n+1,$$

the following inequalities hold for (1.3), *provided that the* a_i*'s are integers satisfying the conditions* (1.5a) *and* (1.5b): *if* r_0 *is an even integer* (*and* $r_1, r_2, \ldots, r_n$ *are odd*),

$$-(r_{j-1}r_{j-2}\cdots r_0)/2 < a_{j-1}r_{j-2}r_{j-3}\cdots r_0 + \cdots + a_1 r_0 + a_0 \leqslant (r_{j-1}r_{j-2}\cdots r_0)/2;$$

if r_0 *is an odd integer* (*as are* $r_1, r_2, \ldots, r_n$),

$$|a_{j-1}r_{j-2}r_{j-3}\cdots r_0| + \cdots + |a_1 r_0| + |a_0| < (r_{j-1}r_{j-2}\cdots r_0)/2.$$

Proof: To avoid tedious repetition, we shall prove this lemma and the two theorems following it only for the

case where r_0 is even. For the other case, where r_0 is odd, the proofs are quite similar.

From the conditions (1.5) on the coefficients a_i, it follows that

$$
\begin{aligned}
-(r_{j-1}r_{j-2}\cdots r_0)/2 &= -\left[(r_{j-1}-1)r_{j-2}r_{j-3}\cdots r_0 + \cdots \right. \\
&\qquad \left. \cdots + (r_1 - 1)r_0 + r_0\right]/2 \\
&< a_{j-1}r_{j-2}r_{j-3}\cdots r_0 + \cdots + a_1 r_0 + a_0 \\
&\leqslant \left[(r_{j-1}-1)r_{j-2}r_{j-3}\cdots r_0 + \cdots \right. \\
&\qquad \left. \cdots + (r_1 - 1)r_0 + r_0\right]/2 \\
&= (r_{j-1}r_{j-2}\cdots r_0)/2,
\end{aligned}
$$

for any integer j such that

$$0 < j \leqslant n+1.$$

This is the desired result.

* * *

If we required that

$$-r_0/2 \leqslant a_0 < r_0/2,$$

when r_0 is an even integer, then the conclusion of the preceding lemma would be that

$$
\begin{aligned}
-(r_{j-1}r_{j-2}\cdots r_0)/2 &\leqslant a_{j-1}r_{j-2}r_{j-3}\cdots r_0 + \cdots \\
&\qquad \cdots + a_1 r_0 + a_0 \\
&< (r_{j-1}r_{j-2}\cdots r_0)/2
\end{aligned}
$$

when r_0 is even. If this were the case, if $j = n+1$, and if x were the integer

$$-(r_{j-1}r_{j-2}\cdots r_0)/2,$$

then the radices $r_n, r_{n-1}, \ldots, r_0$ would be (barely) sufficient to determine the two-sided mixed-radix representation of x via (1.3), but they would not be sufficient to determine the same representation for $|x|$. (That is, an additional radix r_{n+1} would be needed to determine the two-sided mixed-radix representation for $|x|$.) However, if we assume that the conditions (1.5) hold, then Lemma 1.3 assures us that whenever the radices $r_n, r_{n-1}, \ldots, r_0$ are sufficient to determine the two-sided mixed-radix representation of an integer x, they are also sufficient to determine the same representation of $|x|$ (but not conversely). This is why we prefer that a_0 satisfy

$$-r_0/2 < a_0 \leqslant r_0/2$$

when r_0 is even.

* * *

Theorem 1.5 (Uniqueness of Two-Sided Mixed-Radix Notation) - Under the conditions (1.5), the two-sided mixed-radix (integer) coefficients a_i in (1.3) are uniquely determined by x.

Proof: Assume that x and its two-sided mixed-radix coefficients a_i satisfy (1.3) and (1.5), respectively. Assume also that x also satisfies

$$x = c_n r_{n-1} r_{n-2} \cdots r_0 + c_{n-1} r_{n-2} r_{n-3} \cdots r_0 + \cdots$$
$$\cdots + c_1 r_0 + c_0,$$

where the same radices r_i are used in both the above expression and (1.3), r_0 being even and all other r_i's odd, and where the c_i's are integers such that

$$|c_i| < r_i/2, \quad i = 1, 2, \ldots, n,$$

and

$$-r_0/2 < c_0 \leqslant r_0/2.$$

It then follows immediately from these conditions on the a_i's and c_i's that

$$|a_i - c_i| \leqslant r_i - 1, \quad i = 0, 1, \ldots, n.$$

Now assume further that $a_i \neq c_i$ for some index i and let j be the largest such index. Then, as in the proof of Theorem 1.3, we have

$$(c_j - a_j) r_{j-1} r_{j-2} \cdots r_0 + (c_{j-1} - a_{j-1}) r_{j-2} r_{j-3} \cdots r_0$$
$$+ \cdots + (c_1 - a_1) r_0 + (c_0 - a_0)$$
$$= x - x$$
$$= 0.$$

Assuming (without loss of generality) that $a_j < c_j$ and combining this with the above results gives

$$\begin{aligned} & r_{j-1}r_{j-2}\cdots r_0 \\ & \quad \leqslant (c_j - a_j)r_{j-1}r_{j-2}\cdots r_0 \\ & \quad = (a_{j-1} - c_{j-1})r_{j-2}r_{j-3}\cdots r_0 + \cdots \\ & \qquad \cdots + (a_1 - c_1)r_0 + (a_0 - c_0) \\ & \quad \leqslant (r_{j-1} - 1)r_{j-2}r_{j-3}\cdots r_0 + \cdots \\ & \qquad \cdots + (r_1 - 1)r_0 + (r_0 - 1) \\ & \quad = r_{j-1}r_{j-2}\cdots r_0 - 1 \\ & \quad < r_{j-1}r_{j-2}\cdots r_0, \end{aligned}$$

which is clearly a contradiction. Hence, $a_i = c_i$ must hold for $i = 0, 1, \ldots, n$, which completes the proof.

* * *

Theorem 1.6 (Magnitude Comparison in Two-Sided Mixed-Radix Notation) - Let x and y be distinct integers satisfying (1.3) and (1.4) respectively, where the same radices r_i are used in both expressions. Let the coefficients a_i in (1.3) satisfy the conditions (1.5) and the coefficients b_i in (1.4) satisfy the similar conditions:

$$|b_i| < r_i/2, \quad i = 1, 2, \ldots, n$$

and

$$-r_0/2 < b_0 \leqslant r_0/2,$$

if r_0 *is an even integer;*

$$|b_i| < r_i/2, \quad i = 0, 1, \ldots, n,$$

if r_0 *is an odd integer. Let* j *be the largest index* i *such that* $a_i \neq b_i$. *Then,* $x > y$ *if and only if* $a_j > b_j$.

Proof: As for Theorem 1.4, the existence of j is guaranteed by the assumption that $x \neq y$; and as in the proof of Theorem 1.4, it is sufficient here to show that $a_j < b_j$ implies $x < y$ and that $a_j > b_j$ implies $x > y$.

As in the proof of Theorem 1.5, it follows from the conditions on the a_i's and b_i's that

$$|a_i - b_i| \leq r_i - 1, \quad i = 0, 1, \ldots, n.$$

Assuming now that $a_j < b_j$, we have

$$\begin{aligned}
(b_j - a_j)r_{j-1}r_{j-2}\cdots r_0 &\geq r_{j-1}r_{j-2}\cdots r_0 \\
&> r_{j-1}r_{j-2}\cdots r_0 - 1 \\
&= (r_{j-1} - 1)r_{j-2}r_{j-3}\cdots r_0 + \cdots \\
&\quad \cdots + (r_1 - 1)r_0 + (r_0 - 1) \\
&\geq (a_{j-1} - b_{j-1})r_{j-2}r_{j-3}\cdots r_0 + \cdots \\
&\quad \cdots + (a_1 - b_1)r_0 + (a_0 - b_0).
\end{aligned}$$

Hence,

$$y - x = (b_j - a_j)r_{j-1}r_{j-2}\cdots r_0 + (b_{j-1} - a_{j-1})r_{j-2}r_{j-3}\cdots r_0 + \ldots + (b_1 - a_1)r_0 + (b_0 - a_0)$$

$$> 0,$$

which is equivalent to $x < y$.

Similarly, if $a_j > b_j$, interchanging the a_i's and b_i's in the above expressions gives $x - y > 0$. This completes the proof.

* * *

At this point, let us pause to reflect upon what we have established in these theorems, We have shown (in Theorems 1.4 and 1.6) that, if we can determine the ("ordinary" or two-sided) mixed-radix coefficients of integers from their residues in a residue number system, then we can compare the magnitudes of those integers by comparing their coefficents in lexicographic order - that is, by "bit testing" in a computer. Furthermore, we have shown in Lemma 1.3 and Theorem 1.5 that, if we use the two-sided mixed-radix coefficients of an integer x, we can determine the sign of x from the sign of its "leading" (or highest order) non-zero coefficient, since it follows

immediately from that lemma that the signs of x and its leading non-zero coefficient are identical. Therefore, by introducing the above mixed-radix notations and by showing that integers may be uniquely represented and readily compared in these notations, we have reduced - or at least, transformed - the problem of magnitude comparison and sign detection in residue number systems to one of converting integers from their residue representation to their mixed-radix representation. We now turn our attention to the "new" problem of performing that conversion.

C. Conversion to Mixed-Radix Notation. In order to obtain some information about the relationship between the mixed-radix coefficients for an integer x and the residues of x, let us examine (1.3) more closely. Since all of the terms except the last on the right side of that equation contain r_0 as a factor, it is immediately obvious that

$$x \equiv a_0 \pmod{r_0}.$$

Therefore, $x - a_0$ is exactly divisible by r_0 and

$$x_1 = (x - a_0)/r_0$$

is an integer. Combining this definition of x_1 with (1.3) gives

$$x_1 = a_n r_{n-1} r_{n-2} \cdots r_1 + a_{n-1} r_{n-2} r_{n-3} \cdots r_1 + \cdots$$
$$\cdots + a_2 r_1 + a_1,$$

from which it is again obvious that

$$x_1 \equiv a_1 \pmod{r_1}.$$

By continuing in this manner, we may define the integers $x_2, x_3, \ldots, x_n$ by

$$x_i = (x_{i-1} - a_{i-1})/r_{i-1}, \quad i = 1, 2, \ldots, n, \tag{1.6}$$

where $x_0 = x$. From this definition and from (1.3) it follows that x_i also satisfies

$$x_i = a_n r_{n-1} r_{n-2} \cdots r_i + a_{n-1} r_{n-2} r_{n-3} \cdots r_i + \cdots$$
$$\cdots + a_{i+1} r_i + a_i,$$

for $i = 0, 1, \ldots, n$, so that by definition

$$x_i \equiv a_i \pmod{r_i}, \quad i = 0, 1, \ldots, n. \tag{1.7}$$

If we now assume that the radices $r_0, r_1, \ldots, r_n$, are the mutually prime moduli for a residue number system, then we may use the congruences (1.7) to deduce the mixed-radix coefficients a_i of an integer x from the residue representations of $x\ (= x_0), x_1, \ldots,$ and x_n in that number system. That is, from the residues $d_{00}, d_{01}, \ldots, d_{0n}$ of x such that

$$x_0 = x \equiv d_{0i} \pmod{r_i}, \quad i = 0, 1, \ldots, n,$$

we shall find the coefficients a_i by calculating the

residues of $x_1, x_2, \ldots,$ and x_n from the d_{0i}'s. To do this, we note first that

$$a_0 \equiv d_{00} \pmod{r_0},$$

since x is congruent to both a_0 and d_{00} modulo r_0 and since "congruence" is an equivalence relation.

If we now assume slightly more than this, i.e., that $a_0 = d_{00}$, then it follows immediately from (1.6) and the elementary properties of congruences that

$$r_0 x_1 = x - a_0 = x - d_{00} \equiv d_{0i} - d_{00} \pmod{r_i} \qquad (1.8)$$

where $i = 0, 1, \ldots, n$. But since the radices r_i are assumed to be relatively prime, we can eliminate r_0 from (1.8) by defining d_{1i} to be the uniquely determined integer modulo r_i such that

$$r_0 d_{1i} \equiv d_{0i} - d_{00} \pmod{r_i}, \quad i = 1, 2, \ldots, n.$$

It then follows from this definition, the congruences (1.8), and the elementary properties of congruences that

$$x_1 \equiv d_{1i} \pmod{r_i}, \quad i = 1, 2, \ldots, n.$$

In particular, we have

$$a_1 \equiv d_{11} \pmod{r_1},$$

since x_1 is congruent to both a_1 and d_{11} modulo r_1.

Again, if we assume the slightly stronger condition that

$$a_1 = d_{11},$$

we may repeat the above line of reasoning to obtain

$$r_1 x_2 \equiv d_{1i} - d_{11} \pmod{r_i},\ i = 1, 2, \ldots, n,$$

as in (1.8). Furthermore, if we define d_{2i} to be the uniquely defined integer modulo r_i such that

$$r_1 d_{2i} \equiv d_{1i} - d_{11} \pmod{r_i},\ i = 1, 2, \ldots, n,$$

it follows as before that

$$x_2 \equiv d_{2i} \pmod{r_i},\ i = 2, 3, \ldots, n,$$

and that

$$a_2 \equiv d_{22} \pmod{r_2}.$$

By again assuming that $a_2 = d_{22}$, $a_3 = d_{33}$, etc. we can repeat this same procedure again and again to obtain d_{3i}'s, d_{4i}'s, etc. by defining d_{ji} to be the unique integer modulo r_i such that

$$x = x_0 \equiv d_{0i} \pmod{r_i},\ i = 0, 1, \ldots, n,$$

and

$$r_{j-1} d_{ji} \equiv d_{j-1,i} - d_{j-1,j-1} \pmod{r_i}, \qquad (1.9)$$

for $i = j, j+1, \ldots, n$, and $j = 1, 2, \ldots, n$. If we assume for some j such that $0 < j \leq n$ that

$$x_{j-1} \equiv d_{j-1,i} \pmod{r_i},\ i = j-1, j, \ldots, n,$$

and that

$$a_{j-1} = d_{j-1,j-1},$$

it follows immediately from (1.6), (1.9), and the relative primeness of the moduli r_i that

$$r_{j-1}x_j = x_{j-1} - a_{j-1} \equiv d_{j-1,i} - d_{j-1,j-1} \equiv r_{j-1}d_{ji} \pmod{r_i}$$

and

$$x_j \equiv d_{ji} \pmod{r_i}, \quad i = j, j+1, \ldots, n.$$

Thus, by using induction on j and applying (1.7), we obtain the proof of

Theorem 1.7 (Residue to Mixed-Radix Conversion) - *If the residues d_{ji} modulo r_i satisfying (1.9) are chosen in such a way that*

$$d_{ii} \equiv a_i \pmod{r_i}$$

implies that

$$d_{ii} = a_i, \quad i = 0, 1, \ldots, n,$$

then the integers $d_{00}, d_{11}, \ldots, d_{nn}$ are precisely the mixed-radix coefficients $a_0, a_1, \ldots, a_n$, respectively, appearing in (1.3).

* * *

Obviously, the key point in this theorem is that the congruence of d_{ii} and $a_i \pmod{r_i}$ must imply the stronger condition that $d_{ii} = a_i$, for $i = 0, 1, \ldots, n$.

To guarantee that this is the case, we need only require that the residues d_{ji} be subject to the same conditions as the coefficients a_i. This gives

$$|d_{ii} - a_i| < r_i,$$

which when combined with the fact that r_i divides $(d_{ii} - a_i)$ - which is equivalent to

$$d_{ii} \equiv a_i \pmod{r_i} -$$

does indeed yield the result that $d_{ii} = a_i$. Moreover, it is clear from this that whether the mixed-radix coefficients referred to in Theorem 1.7 are the ordinary or two-sided variety depends entirely upon the restrictions placed on the residues d_{ji}. In particular, if we require that the integers d_{ji} (which are, by definition, residues modulo r_i) satisfy

$$0 \leqslant d_{ji} < r_i, \quad i = 0, 1, \ldots, n,$$

then the integers $d_{00}, d_{11}, \ldots, d_{nn}$ in Theorem 1.7 are the <u>ordinary</u> mixed-radix coefficients for x. If, on the other hand, we require that the d_{ji}'s satisfy

$$-r_0/2 < d_{00} \leqslant r_0/2, \tag{1.10a}$$

$$|d_{ji}| < r_i/2, \quad i = 0,1,\ldots, n \tag{1.10b}$$

if r_0 is even, and

$$|d_{ji}| < r_i/2, \quad i = 0, 1, \ldots, n \tag{1.10c}$$

when r_0 is odd, then the integers $d_{00}, d_{11}, \ldots, d_{nn}$ are the two-sided mixed-radix coefficients for x. Thus, by equating the mutually prime moduli for a residue number system with the radices r_i of mixed notation and by subjecting the residues in that system to the same conditions as those on the mixed-radix coefficients, we can obtain - via equations (1.6), (1.7), and (1.9) above - either the ordinary or the two-sided mixed-radix coefficients of an integer from its residues.

It is interesting to note that, if all the residues in the residue number system are made to satisfy the conditions (1.10) which give the two-sided mixed-radix coefficients via Theorem 1.7, the modular arithmetic computer using these residues requires only about half as much circuitry as the one using the non-negative residues which give the ordinary mixed-radix coefficients. The reason for this is that the former computer need only compute with the integers 0 thru M/2 plus a "sign bit" whereas the latter computer must use all of the integers 0 thru M-1. (M is the product of the moduli.)

To illustrate the conversion algorithm of Theorem 1.7 for ordinary mixed-radix notation, let us reconsider the first example given at the beginning of this chapter. (See pp.9-11.) Since the even modulus 2 is used, we must set $r_0 = 2$; the other moduli may be indexed arbitrarily, say,

$$r_1 = 3, \quad r_2 = 5, \quad \text{and} \quad r_3 = 7.$$

Using the residues given previously for

$$x = 59, \quad y = 78, \quad \text{and} \quad z = 103,$$

we obtain the ordinary mixed-radix coefficients for x, y, and z from their respective sets of residues d_{ji} which are given in Table I.

The first row in each set of residues in Table I contains the residues d_{00}, d_{01}, d_{02}, d_{03} of the corresponding integer x, y, or z modulo r_0, r_1, r_2, r_3 respectively. The second row of each set is calculated from the first in accordance with equation (1.9) and contains the integers d_{11}, d_{12}, d_{13} modulo r_1, r_2, r_3 respectively, such that

$$r_0 d_{1i} \equiv d_{0i} - d_{00} \pmod{r_i}, \quad i = 1, 2, 3.$$

(For instance, for $x = 59$, the second row in Table I contains

Table I - Ordinary Mixed-Radix Notation

$r_3 = 7 \quad r_2 = 5 \quad r_1 = 3 \quad r_0 = 2$

j \ i	0	1	2	3
0	1	2	4	3
1	-	2	4	1
2	-	-	4	2
3	-	-	-	1

x = 59

j \ i	0	1	2	3
0	0	0	3	1
1	-	0	4	4
2	-	-	3	6
3	-	-	-	2

y = 78

j \ i	0	1	2	3
0	1	1	3	5
1	-	0	1	2
2	-	-	2	3
3	-	-	-	3

z = 103

$d_{11} = 2, \quad d_{12} = 4, \quad$ and $\quad d_{13} = 1,$

which satisfy the congruences

$$2 \cdot d_{11} \equiv 2 - 1 \equiv 1 \pmod 3$$

$$2 \cdot d_{12} \equiv 4 - 1 \equiv 3 \pmod 5$$

$$2 \cdot d_{13} \equiv 3 - 1 \equiv 2 \pmod 7,$$

respectively.) Similarly, the third row of each set is calculated from the second and the fourth row is calculated from the third, again by using equation (1.9).

From the "diagonal" entries for x in Table I, we obtain the ordinary mixed-radix coefficients for x:

$$\begin{aligned} x &= a_3 r_2 r_1 r_0 + a_2 r_1 r_0 + a_1 r_0 + a_0 \\ &= d_{33} r_2 r_1 r_0 + d_{22} r_1 r_0 + d_{11} r_0 + d_{00} \\ &= 1(5 \cdot 3 \cdot 2) + 4(3 \cdot 2) + 2(2) + 1 \\ &= 59 \end{aligned}$$

or, by using the more concise notation

$$[a_3, a_2, a_1, a_0] = [d_{33}, d_{22}, d_{11}, d_{00}],$$

we have

$$x \sim [1, 4, 2, 1].$$

Similarly, for y and z we have

$$\begin{aligned} y &= 2(5 \cdot 3 \cdot 2) + 3(3 \cdot 2) + 0(2) + 0 \\ &= 78 \sim [2, 3, 0, 0] \end{aligned}$$

and

$$z = 3(5\cdot3\cdot2) + 2(3\cdot2) + 0(2) + 1$$
$$= 103 \sim [3, 2, 0, 1].$$

Now we can compare x, y, and z by applying Theorem 1.4: the "leading" coefficients for x, y, and z are all unequal, so comparing x, y, and z reduces to comparing their leading coefficients. Since

$$1 < 2 < 3,$$

we conclude that

$$[1, 4, 2, 1] < [2, 3, 0, 0] < [3, 2, 0, 1]$$

or

$$x < y < z.$$

Hence, $x = 59 \approx \{3, 4, 2, 1\}$ is the smallest of the three numbers x, y, and z.

To illustrate the use of two-sided mixed-radix notation, let us now consider the integers given in the second example at the beginning of this chapter. (See pp.11-12.) For

$$u = 58 \approx \{2, 3, 1, 0\}, \quad v = 60 \approx \{4, 0, 0, 0\},$$
$$\text{and} \quad w = 59 \approx \{3, 4, 2, 1\},$$

we must adjust some of the residues so that they all satisfy the conditions (1.10). This gives

$$u \approx \{2, -2, 1, 0\}, \qquad v \approx \{-3, 0, 0, 0\},$$

and $$w \approx \{3, -1, -1, 1\}.$$

The residues d_{ji} for u, v, and w are given in Table II. The successive rows of residues for each of u, v, and w are obtained exactly as for x, y, and z in Table I except that the d_{ji}'s in Table II satisfy the conditions (1.10). Thus, we have

$$\begin{aligned} u &= d_{33}r_2r_1r_0 + d_{22}r_1r_0 + d_{11}r_0 + d_{00} \\ &= 2(5\cdot3\cdot2) + 0(3\cdot2) - 1(2) + 0 \\ &= 58 \end{aligned}$$

or, in the more concise notation,

$$u \sim [2, 0, -1, 0];$$

also,

$$\begin{aligned} v &= 2(5\cdot3\cdot2) + 0(3\cdot2) + 0(2) + 0 \\ &= 60 \sim [2, 0, 0, 0] \end{aligned}$$

and

$$\begin{aligned} w &= 2(5\cdot3\cdot2) + 0(3\cdot2) - 1(2) + 1 \\ &= 59 \sim [2, 0, -1, 1]. \end{aligned}$$

Applying Theorem 1.6 to compare u, v, and w, we see that the first three coefficients for u and w are the same: 2, 0, -1. Hence, we compare u and w by

Table II - Two-Sided Mixed-Radix Notation

$r_3 = 7 \quad r_2 = 5 \quad r_1 = 3 \quad r_0 = 2$

j \ i	0	1	2	3
0	0	1	-2	2
1	-	-1	-1	1
2	-	-	0	3
3	-	-	-	2

u = 58

j \ i	0	1	2	3
0	0	0	0	-3
1	-	0	0	2
2	-	-	0	3
3	-	-	-	2

v = 60

j \ i	0	1	2	3
0	1	-1	-1	3
1	-	-1	-1	1
2	-	-	0	3
3	-	-	-	2

w = 59

comparing their last coefficients, 0 and 1, respectively, from which we conclude that

$$[2, 0, -1, 0] < [2, 0, -1, 1]$$

or

$$u < w.$$

Similarly, the first two coefficients of v and w are the same, so we compare v and w by comparing their third coefficients, 0 and -1, respectively. We get

$$[2, 0, 0, 1] > [2, 0, -1, 1]$$

or

$$v > w.$$

It might be noted from these examples that a fair amount of computation is needed to obtain all the residues d_{ji} necessary to convert an integer from residue notation to mixed-radix notation. This computation can be performed most quickly and efficiently in a modular arithmetic computer if a permanently stored table of residues is used to "eliminate" the moduli $r_0, r_1, \ldots, r_{n-1}$ in the congruences (1.9) from which the successive rows of residues d_{ji} are calculated. Such a table consists of the integers s_{ji} such that

$$s_{ji} = 0, \quad i = 0, 1, \ldots, j,$$

and

$$r_j s_{ji} \equiv 1 \pmod{r_i}, \quad i = j+1, j+2, \ldots, n,$$

where

$$j = 0, 1, \ldots, n-1.$$

The entries s_{ji} in this table are used to calculate the residues d_{ji} from the congruence

$$d_{ji} \equiv (d_{j-1,i} - d_{j-1,j-1}) s_{j-1,i} \pmod{r_i} \qquad (1.11)$$

which is equivalent to (1.9). (Note, however, that this table can be used for only one particular ordering of the moduli $r_0, r_1, \ldots, r_n$. If the moduli are re-indexed, a different table is required. In practice, though, it is very doubtful whether more than one "indexing" would ever be necessary.) Since the calculations required in (1.11) can be performed simultaneously for a fixed j and for $i = 0, 1, \ldots, n$ in a modular arithmetic computer, only one subtraction and one multiplication are needed to calculate each row of residues d_{ji} from the preceding one. Hence, when the $n+1$ moduli $r_0, r_1, \ldots, r_n$ are used in the computer, the entire conversion process can be accomplished with n subtractions and n multiplications.

To illustrate the use of such a stored table, let us use the table of residues s_{ji} given in Table III to convert to two-sided mixed-radix notation the numbers

$p \approx \{-3, 1, 1, 1\}$ and $q \approx \{-3, -2, -1, 1\}$

given in the third example at the beginning of this chapter. (We are now using residues satisfying (1.10).) The resulting sets of residues d_{ji} for p and q are given in Table III. The first rows in those sets are simply the residues of p and q modulo r_0, r_1, r_2, r_3, respectively, and the second, third, and fourth rows are calculated from the first rows by using (1.11).

For $q = 53$ the calculation of the second row is performed as follows: the residue d_{00} is subtracted from each of the residues d_{00}, d_{01}, d_{02}, d_{03} in the first row, the four subtractions being performed simultaneously and independently with respect to the four moduli r_0, r_1, r_2, r_3, respectively. This gives

$$0 \equiv 1 - 1 \pmod 2$$

$$1 \equiv -1 - 1 \pmod 3$$

$$2 \equiv -2 - 1 \pmod 5$$

$$3 \equiv -3 - 1 \pmod 7.$$

Next, the residues obtained from these subtractions are multiplied by the entries s_{00}, s_{01}, s_{02}, s_{03}, in the

first row of the stored table, respectively, the multiplications being performed simultaneously modulo r_0, r_1, r_2, r_3, respectively. This gives the residues d_{10}, d_{11}, d_{12}, d_{13} in the second row:

$$d_{10} = 0 \equiv 0(0) \pmod 2$$

$$d_{11} = -1 \equiv 1(-1) \pmod 3$$

$$d_{12} = 1 \equiv 2(-2) \pmod 5$$

$$d_{13} = -2 \equiv 3(-3) \pmod 7.$$

Similarly, the third row of residues for q is calculated from the second row by first subtracting d_{11} from each of the elements d_{10}, d_{11}, d_{12}, d_{13} in the second row and then multiplying the results by the respective elements s_{10}, s_{11}, s_{12}, s_{13} in the second row of the stored table. Finally, the last row of residues for q is calculated from the third row by subtracting d_{22} from each of d_{20}, d_{21}, d_{22}, d_{23} and multiplying the results by the elements s_{20}, s_{21}, s_{22}, s_{23}, respectively, in the last row of the stored table of s_{ji}'s. The calculation of the residues d_{ji} for $p = -59$ is performed in the same manner.

Note that, whereas previously in (1.9) the residues d_{ji} were undefined for $i < j$, we now have $d_{ji} = 0$ when $i < j$, which results from setting $s_{ji} = 0$ for

Table III - Mixed-Radix Conversion by Stored Table

$r_3 = 7 \quad r_2 = 5 \quad r_1 = 3 \quad r_0 = 2$

j \ i	0	1	2	3
0	0	-1	-2	-3
1	0	0	2	-2
2	0	0	0	3

s_{ji}

j \ i	0	1	2	3
0	1	1	1	-3
1	0	0	0	-2
2	0	0	0	-3
3	0	0	0	-2

$p = -59$

j \ i	0	1	2	3
0	1	-1	-2	-3
1	0	-1	1	-2
2	0	0	-1	2
3	0	0	0	2

$q = 53$

$i \leqslant j$. We do this merely for the sake of convenience in performing the above calculations in a computer.

The "diagonal" elements d_{00}, d_{01}, d_{02}, d_{03} obtained in this way are the two-sided mixed-radix coefficients for p and q, respectively. Hence, from Table III we have

$$p = d_{33}r_2r_1r_0 + d_{22}r_1r_0 + d_{11}r_0 + d_{00}$$
$$= -2(5\cdot3\cdot2) + 0(3\cdot2) + 0(2) + 1$$

and

$$q = 2(5\cdot3\cdot2) - 1(3\cdot2) - 1(2) + 1,$$

or

$$p \sim [-2, 0, 0, 1] \quad \text{and} \quad q \sim [2, -1, -1, 1].$$

Since the leading non-zero coefficients for p and q are -2 and 2, respectively, we conclude immediately that p is negative and q is positive.

We have now shown how magnitude comparison can be performed for integers in residue number systems by calculating the mixed-radix coefficients from the residues of the given integers and then comparing those coefficients in lexicographic order. We have also shown how the sign of an integer can be determined from its leading non-zero

two-sided mixed-radix coefficient. Finally, we have shown how the calculation of the mixed-radix coefficients from the residues of an integer can be performed efficiently in a modular arithmetic computer by using a stored table of residues. Thus, we have added magnitude comparison and sign detection to the set of operations which can be performed readily in a modular arithmetic computer. We shall now make use of these operations in devising methods to perform other fundamental operations in these computers.

CHAPTER II

OVERFLOW DETECTION

A. Overflow in Residue Number Systems. "Overflow" is the term designating the situation which occurs when a digital computer generates a number "too large" for itself; that is, when some operation performed by the computer results in a number outside the range of numbers the computer is designed to handle normally. If overflow occurs, the computer in some manner "truncates" the number beyond its range to produce a number which is within its range and which is used in place of the original one in subsequent calculations. But since certain important arithmetic properties of the original number may not be preserved in this truncation, erroneous answers may result unless the overflow is detected and the subsequent calculations are modified accordingly. Therefore, some means of detecting overflow under program control must be provided in every digital computer.

In "conventional" digital computers using, for example, N-digit binary numbers, one or more "extra" high-order digits are built into the register (called the accumulator) where arithmetic operations take place. When some operation produces a number requiring more than N digits for its binary representation, overflow occurs and is detected immediately by a "carry" into one or more of the extra digits in the accumulator. Special "transfer-on-overflow" instructions are used by the computer programmer to test these high-order digits to determine if it is necessary to "shift" the number in the accumulator to compensate for the overflow.

In modular arithmetic computers, this situation is slightly different. Overflow still occurs whenever some operation produces a number beyond the computer range, but since all arithmetic operations are performed modulo M, the product of the moduli, in these computers, no "carries" are ever generated. For instance, in a modular arithmetic computer in which the moduli are 2, 3, 5, and 7 and the computer range is the set of all integers from -104 to 105 inclusive, overflow occurs when the numbers

$\{1, -2, 0, 0\} \approx 78$ and $\{3, -1, -1, 1\} \approx 59$ are added. The "true" sum, 137, of these numbers is outside the computer range, so it is "truncated" to give the unique integer x modulo M $(= 210)$ within the computer range and such that

$$x \equiv 137 \pmod{210}.$$

Thus,

$$x \approx \{-3, 2, -1, 1\} \approx -73$$

is the "computed" sum of 78 and 59 in this computer, which is a most astounding result since we usually expect the sum of two positive integers to be positive.

In general, whenever the sum, difference, or product of two integers in a modular arithmetic computer lies outside the computer's range, the "computed" result will be the unique integer which is within the computer range and which is congruent modulo M to the "true" sum, difference, or product. While this form of truncation may permit the programmer to ignore all overflows and yet obtain the correct results in many cases, it is still often necessary to know whether or not the computed sum, difference, or product is exactly equal to the true result. Therefore, we shall now consider the problem of detecting

overflow in residue number systems. We shall make no restrictions on the moduli used (other than those needed for magnitude and, where applicable, sign detection), but we shall allow the computer range to be only the integers 0 through M-1 inclusive, the integers $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ inclusive (where M is odd), or the integers $-\frac{M}{2}+1$ through $\frac{M}{2}$ inclusive (where M is even). The reason for restricting ourselves to only three possible ranges for the computer is that, for a given set of moduli, the behavior of overflow varies considerably with the range used. (Also, it is extremely unlikely whether any range other than these would be useful in a practical modular arithmetic computer.) Finally, since addition, subtraction, and multiplication are the only arithmetic operations which can cause overflow in a modular arithmetic computer, we shall treat only the detection of additive overflow (which includes overflow resulting from subtraction) and multiplicative overflow.

B. Additive Overflow. To detect overflow occurring in addition and subtraction in modular arithmetic computers, we compare the magnitude of the computed sum or difference with that of one of the (two) addends or that

of the minuend. We determine whether or not overflow has occurred by checking to see if the computed sum of difference satisfies the order relations which normally hold between the true sum or difference and the addends or minuend. If these relations are not satisfied by the computed result, we conclude that overflow has occurred.

For the case in which the computer range consists of the integers 0 through M-1, we define z and w to be the computed sum and difference, respectively, of the integers x and y. We assume that x and y are within the computer range, as are z and w. Since, by definition,

$$z \equiv x + y \pmod{M}$$

and since, by the above assumption,

$$0 \leqslant x + y < 2M,$$

it follows immediately that whenever overflow occurs in addition - that is, $x + y \geqslant M$ - then z is given by

$$z = x + y - M.$$

Hence, when overflow occurs, we have

$$z < x,$$

since y is less than M by assumption. On the other hand, if no overflow occurs, then

$$z = x + y \geqslant x.$$

Clearly, since the above expressions for z are "symmetrical" in x and y, the same relations hold between z and y.

In the same manner, it follows from the assumptions on x and y that

$$-M < x - y < M.$$

Since

$$w \equiv x - y \pmod{M},$$

it is then clear that w is given by

$$w = x - y + M$$

whenever overflow occurs in subtraction - that is, whenever $x - y < 0$. Therefore, since y is less than M, it follows as before that

$$w > x$$

whenever overflow occurs. On the other hand, if no overflow occurs, then we have

$$w = x - y \leqslant x.$$

This proves

Theorem 2.1 (Additive Overflow Detection) - In the residue number system whose range is the set of integers from 0 through M-1 inclusive, overflow occurs in

addition if and only if the computed sum is less than either of the addends and overflow occurs in subtraction if and only if the computed difference is greater than the minuend.

* * *

For example, in the residue number system based on the moduli 2, 3, 5, and 7 and whose range is 0 through 209, we detect overflow in the addition and subtraction of

$$x = 91 \approx \{0, 1, 1, 1\}$$

and

$$y = 127 \approx \{1, 2, 1, 1\}$$

by noting that their sum

$$z = 8 \approx \{1, 3, 2, 0\}$$

is less than x (and less than y) and that their difference

$$w = 174 \approx \{6, 4, 0, 0\}$$

is greater than x. As before, we perform these comparisons by converting z, w, and x to mixed-radix notation and comparing their mixed-radix coefficients as prescribed in Theorem 1.4:

$$z \sim [0, 1, 1, 0] < [3, 0, 0, 1] \sim x;$$

$$x \sim [5, 4, 0, 0] > [3, 0, 0, 1] \sim x.$$

For the case where M is an odd integer and the computer range is $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ inclusive, we may consider overflow in addition and subtraction simultaneously by regarding the subtraction of y from x as the addition of $(-y)$ to x, x and y both being integers within the computer range. Therefore, we define z to be the computed sum of x and y, or equivalently, the computed difference of x and y', where $y' = -y$. Now, if x and y have opposite signs or if either is zero, overflow is impossible since $x + y$ must lie between x and y and therefore must be within the computer range. But, if x and y are both positive, then it follows from the assumptions on the computer range that

$$0 < x + y < M,$$

so that overflow occurs whenever

$$M/2 < x + y < M.$$

Since z is also within the computer range - that is,

$$-M/2 < z < M/2 \quad - \tag{2.1}$$

and since, by definition,

$$z \equiv x + y \pmod{M}, \tag{2.2}$$

it follows that

$$-M/2 < z = x + y - M < 0.$$

Hence, when x and y are both positive and overflow

occurs, z is negative. Similarly, if x and y are both negative and overflow occurs, z must be positive since (2.1), (2.2), and

$$-M < x + y < -M/2$$

imply that

$$0 < z = x + y + M < M/2.$$

For the case where the computer range is $-\frac{M}{2} + 1$ through $\frac{M}{2}$, we may reason almost exactly as we have done immediately above, except that we must provide for the case where $y' = M/2$. We do this simply by regarding the subtraction of $M/2$ as the addition of $M/2$ since both operations give the same computed result. Then, it follows as before that overflow is impossible whenever x and y have opposite signs or whenever either is zero. If x and y are positive, then from

$$0 < x \leqslant M/2, \quad 0 < y \leqslant M/2, \quad -M/2 < z \leqslant M/2,$$

and (2.2) it follows that

$$M/2 < x + y \leqslant M$$

and

$$z = x + y - M$$

if overflow occurs. Hence, if both x and y are positive and overflow occurs, z will be negative or zero. Similarly,

if x and y are both negative, then

$$-M/2 < x < 0 \qquad \text{and} \qquad -M/2 < y < 0.$$

If overflow occurs, then

$$-M < x + y \leqslant -M/2,$$

so that

$$z = x + y + M.$$

Therefore, if overflow occurs when both x and y are negative, z is positive.

Now let us consider what happens if x and y have the same sign and overflow does not occur. If no overflow occurs, then z is simply the true sum of x and y and hence z has the same sign as both x and y, regardless of the computer range. This completes the proof of

Theorem 2.2 (Additive Overflow Detection) - If the computer range is the set of integers $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ or $-\frac{M}{2}+1$ through $\frac{M}{2}$ inclusive, then overflow occurs in addition if and only if both summands are non-zero and have the same sign while their computed sum is zero or has the opposite sign.

* * *

For example, using the moduli 2, 3, 5, and 7 and the computer range $-\frac{M}{2}+1 = -104$ through $\frac{M}{2} = 105$

in our residue number system, we detect overflow in the addition of the integers

$$x = 83 \approx \{-1, -2, -1, 1\}$$

and

$$y = 71 \approx \{1, 1, -1, 1\}$$

by noting that both are positive while their computed sum

$$z = -56 \approx \{0, -1, 1, 0\}$$

is negative. We determine the signs of x, y, and z from the signs of their leading two-sided mixed-radix coefficients:

$$x \sim [3, -1, -1, 1]; \quad y \sim [2, 2, -1, 1];$$

$$z \sim [-2, 1, -1, 0].$$

This completes our treatment of additive overflow detection. We turn now to the problem of detecting overflow in multiplication in residue number systems.

C. Multiplicative Overflow. Detecting multiplicative overflow in modular arithmetic computers is somewhat more difficult than detecting additive overflow, primarily because the numbers generated in multiplication may be "farther" outside the computer range than those generated in addition. That is, if K is the largest of the absolute values of the integers within the computer range, then

the absolute value of the true sum of two numbers in the computer cannot exceed $2K$, while the absolute value of their true product may be as large as K^2. Therefore, the relationship between their true and computed products is, in general, more complex than that between their true and computed sums. However, it turns out that the technique of comparing the sign and magnitude of the computed result with those of the operands, as was done above to detect additive overflow, can still be used in many instances to detect multiplicative overflow.

If z is the computed product of the integers x and y in some residue number system, then z should be zero if and only if at least one of x and y are zero. Hence, if $z = 0$, multiplicative overflow has occurred if x and y are both non-zero. And if

$$|x| \geq 1 \quad \text{and} \quad |y| \geq 1,$$

then both

$$|z| \geq |x| \quad \text{and} \quad |z| \geq |y|$$

should be true, at least so long as no overflow has occurred. Therefore, if either of

$$|z| < |x| \quad \text{or} \quad |z| < |y|$$

holds for x and y non-zero, then multiplicative overflow

must have occurred. Furthermore, if x and y have the same sign, then z should be positive; and if x and y have opposite signs, z should be negative. Hence, overflow is also indicated by the presence of the "wrong" sign on z.

But while these tests are sufficient to detect multiplicative overflow, they are not necessary. (For a counter-example, consider the multiplication of 16 by itself in the residue number system with moduli 2, 3, 5, and 7 and with either the range 0 through 209 or the range -104 through 105.) Therefore, we must find a method for detecting multiplicative overflow when the above sign and magnitude tests do not indicate that overflow has occurred. That is, we must ascertain whether or not

$$|x \cdot y| = |x| \cdot |y| > K,$$

where K is the maximum absolute value of the integers within the computer range, when z has the "proper" sign and when both

$$0 < |x| \leqslant |z| \leqslant K \quad \text{and} \quad 0 < |y| \leqslant |z| \leqslant K$$

are satisfied. To do this, we shall compare $|x|$ and $|y|$ with the (positive) square root of K.

Let k be the unique positive integer such that

$$k^2 \leqslant K < (k+1)^2, \tag{2.3}$$

let d be the non-negative integer such that

$$k^2 + d = K, \tag{2.4}$$

and let us assume for convenience that $|x| \geqslant |y|$. We may compare $|x|$ and $|y|$ with k by defining the integers a and b by

$$a = |x| - k \quad \text{and} \quad b = |y| - k. \tag{2.5}$$

Then, if $a \leqslant 0$, b must also be $\leqslant 0$ since $|x| \geqslant |y|$ implies $a \geqslant b$. In that case, we have

$$|x| \cdot |y| = (k+a)\cdot(k+b) \leqslant k^2 \leqslant K,$$

which means that overflow does not occur. Similarly, if $b > 0$, then $a > 0$ also and we have

$$|x| \cdot |y| = (k+a)\cdot(k+b) \geqslant (k+1)^2 > K,$$

which is precisely the condition for overflow. Therefore, if $a \leqslant 0$, there can be no multiplicative overflow since both $|x|$ and $|y|$ are less than the square root of K; and if $b > 0$, there must be overflow because both $|x|$ and $|y|$ are greater than the square root of K.

To determine whether or not overflow occurs when $a > 0$ and $b \leqslant 0$ (which is the remaining case), let us examine the equation

$$|x \cdot y| = |x| \cdot |y| = (k + a) \cdot (k + b)$$

$$= k^2 + (a + b)k + ab.$$

Substituting $K - d$ for k^2 in this equation gives

$$|x \cdot y| = K - d + (a + b)k + ab$$

from which it follows that overflow occurs - that is, $|x \cdot y| > K$ - if and only if

$$(a + b)k + ab > d. \qquad (2.6)$$

But since we are assuming that $a > 0 \geqslant b$, it follows that

$$(a + b)k + ab \leqslant (a + b)k,$$

so that if $a \leqslant |b|$, then

$$(a + b)k + ab \leqslant (a + b)k \leqslant 0 \leqslant d.$$

Combining this result with (2.6), we conclude that $a \leqslant |b|$ implies that overflow does not occur.

Finally, if $a > |b|$, (which is now the only remaining case), we add b^2 to both sides of (2.6) and apply the definitions (2.5) to the left side of the result. This gives

$$(a + b)k + ab + b^2 = (a + b) \cdot (k + b)$$

$$= (a + b)|y|,$$

which, when combined with (2.6), yields the conclusion that overflow occurs if and only if

$$(a + b) \cdot |y| > d + b^2.$$

Now, if no overflow occurs in multiplying $(a + b)$ by $|y|$,

we can readily compare $(a + b)\cdot|y|$ with $d + b^2$ to determine conclusively whether or not overflow occurs in calculating xy. On the other hand, if $(a + b)\cdot|y|$ overflows, then it follows from (2.6) that $|x \cdot y|$ must also overflow (and hence that xy overflows). These conclusions stem from the inequalities

$$0 \leqslant |b| = -b = k - |y| < k$$

and

$$(a + b)\cdot|y| > K = d + k^2 > d + b^2,$$

which follow from our definitions.

Therefore, if all other tests are inconclusive and if $a > |b|$ and $a > 0 \geqslant b$, then the detection of overflow in multiplying x and y depends upon the detection of overflow in multiplying $(a + b)$ by $|y|$. What we do in that case is define x_1 by

$$x_1 = a + b$$

and repeat the entire procedure to try to determine whether or not $|x_1 \cdot y| = |x_1|\cdot|y| = |a + b|\cdot|y|$ overflows. If necessary, we define an x_2, an x_3, or even an x_{k+1}, obtaining each x_i from the preceding one in exactly the same way as x_1 is obtained from x. Eventually, for some x_i, some test such as $a \leqslant |b|$ will halt this procedure since

$$|x| > |x_1| > |x_2| \quad \ldots, \text{ etc.}$$

This follows from

$$|x| = a + k > a \geq a + b = x_1 > 0$$

since $b \leq 0$; similar relations hold for x_1 and x_2, etc.

Thus, we have obtained an iterative procedure for detecting multiplicative overflow, a "flow chart" of which is given in Figure I. Let us now formalize this procedure for overflow detection by stating it as

Theorem 2.3 (Multiplicative Overflow Detection-Method I) - Let K be the largest of the absolute values of the integers in a residue number system. Let k and d be integers as defined above in (2.3) and (2.4) and assume for convenience that $|x| \geq |y|$. Let

$$x_0 = x, \quad b = |y| - k, \quad \text{and} \quad a_0 = |x_0| - k,$$

and let z_i be the computed product of x_i and y. Then xy overflows if $b > 0$ or if any of the conditions A.- D. holds for $i = 0$:

A. $z_i = 0$ while $x_i \neq 0 \neq y$;

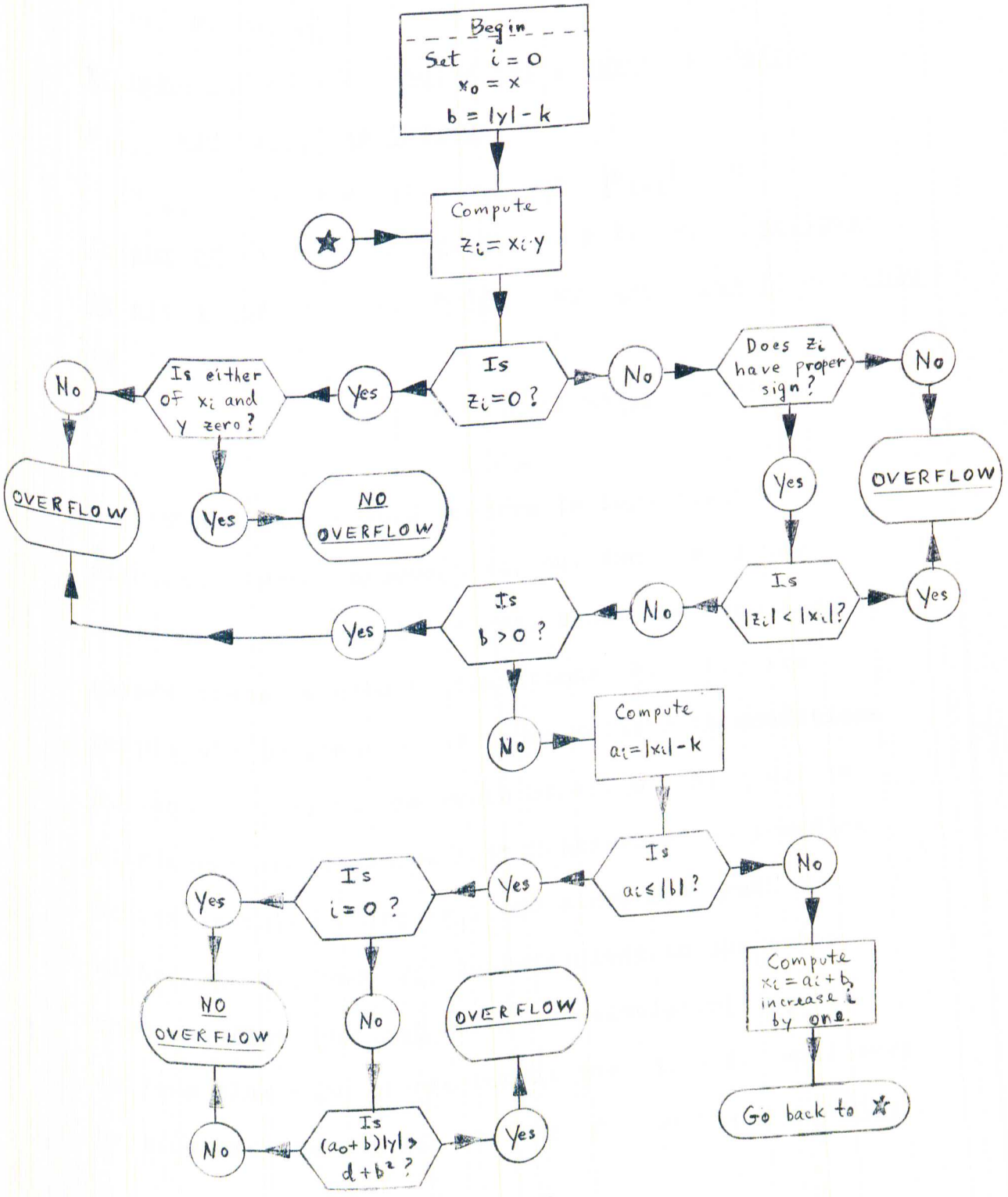
B. z_i has the wrong sign $(z_i \neq 0)$;

C. $0 < |z_i| < |x_i|$;

D. $0 < |z_i| < |y|$.

Moreover, xy does not overflow if either of the

Figure I - Multiplicative Overflow Detection - Method I



conditions E. - F. holds for $i = 0$:

E. $a_i \leqslant 0$;

F. $a_i \leqslant |b|$.

If none of A. - F. holds for a given i, define x_{i+1} and a_{i+1} as follows:

$x_{i+1} = a_i + b$ and $a_{i+1} = |x_{i+1}| - k$.

If any of A. - D. holds for $i \geqslant 1$, xy overflows; if either of E. - F. holds, xy overflows if and only if

$$|z_1| = (a_0 + b) \cdot |y| > d + b^2.$$

* * *

The proof of this theorem is implicit in the discussion which preceded it, but two clarifying statements are needed. First, since the sign and magnitude tests related to conditions A. - D. are completely independent of those related to conditions E. and F., since the truth of any of A. - D. is sufficient to guarantee that multiplicative overflow occurs in calculating z_i, and since the truth of either of E. and F. is sufficient to guarantee that overflow does not occur in calculating z_i, it follows that none of the conditions A. - D. will ever be true whenever either E. or F. is true, and vice

versa. (Note that condition E. is actually superfluous since $a_i \leqslant 0$ certainly implies $a_i \leqslant |b|$, which is condition F.) Second, since the algorithm is dependent upon the assumption that $|x_i| \geqslant |y|$, one might think that an interchange of x's and y's is necessary whenever none of A. - F. is satisfied for a given i and it happens that $|x_{i+1}|$ is less than $|y|$. However, since $b \leqslant 0$, it follows from the definition of a_{i+1} that, in that case, condition E. would be satisfied by a_{i+1} regardless of whether or not x_{i+1} and y are interchanged. In other words, if for some i we have

$|x_{i+1}| < |y|$ while $|x_i| \geqslant |y|$, then xy overflows if and only if $z_1 > d + b^2$, so that we needn't bother to interchange x_{i+1} and y. (In light of this, it also follows that condition D. is superfluous, since, by the assumption that $|x| \geqslant |y|$, condition D. implies the weaker condition C.)

To illustrate the use of this algorithm for detecting multiplicative overflow, let us consider the multiplication of x = 29 and y = 9 in a residue number system whose range is -104 through 105. Using the above definitions gives

$K = 105$, $k = 10$, $d = 5$,

$z_0 = 51$, $a_0 = 19$, and $b = -1$.

Since none of the conditions A. - F. is satisfied for these values of x $(=x_0)$, y, z_0, a_0, and b, we calculate

$$x_1 = a_0 + b = 18$$

and the (computed) product $z_1 = -48$ of x_1 and y. Since z_1 is negative while both x_1 and y are positive, we now find that condition B. is satisfied. Thus, we conclude that xy overflows.

It should be noted that when the computer range is not "symmetric" about zero, as in the preceding example, it is necessary to use two K's and two d's - e.g., $K = 105$, $K' = 104$, $d = 5$, and $d' = 4$ for the above example. One K and the corresponding d are to be used when the product of x and y should be positive - that is, when x and y have the same sign - and the other K and d (denoted by K' and d' for the above example) when the product should be negative. In this way, we may provide for the situation where xy overflows but $|x| \cdot |y|$ does not. (In the above example, this could happen only when the true product xy

is -105.) Clearly, this modification to the procedure given in Theorem 2.3 is not needed whenever the computer range is "symmetric" about zero - say, the integers $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ - or consists entirely of non-negative numbers - say, the integers 0 through M - 1.

After picking a few "sample" multiplications at random and using the above algorithm to determine whether or not multiplicative overflow occurs in each case, we begin to feel that instances in which it is necessary to calculate an x_2, an x_3, or even an x_{k+1} to determine conclusively whether or not multiplicative overflow occurs are probably quite rare. Nevertheless, the possibility of such cases does exist and motivates us to seek a faster method of detecting multiplicative overflow when the sign and magnitude tests, i.e., conditions A. - D. in Theorem 2.3, are inconclusive.

D. Multiplicative Overflow (continued). Let us now assume that the sign and magnitude tests A. - D. described above have been applied to x, y, and their

computed product z with the result that none of the conditions A. - D. is satisfied. Let us also assume that $|x| \geq |y|$ and that our computer range is either the set of integers $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ or the set of integers $-\frac{M}{2}+1$ through $\frac{M}{2}$. Thus, by assumption we have

$$0 < |y| \leq |x| \leq |z| \leq M/2.$$

Now, instead of comparing $|x|$ and $|y|$ with k as before, let us postulate the existence of a table of powers of two, stored within the computer, from which we can obtain unique integers p and q such that

$$2^{p-1} < |x| \leq 2^p \quad \text{and} \quad 2^{q-1} < |y| \leq 2^q. \tag{2.7}$$

If we define n to be the unique integer such that

$$2^n \leq M/2 < 2^{n+1}, \tag{2.8}$$

and if $p + q \leq n$, then it follows immediately that

$$|xy| = |x| \cdot |y| \leq 2^p \cdot 2^q = 2^{p+q} \leq 2^n \leq M/2,$$

which means that overflow - that is, $|xy| > M/2$ - does not occur. Similarly, if $p + q \geq n + 3$, then we have

$$M/2 < 2^{n+1} \leq 2^{p+q-2} = 2^{p-1} \cdot 2^{q-1} < |x| \cdot |y|,$$

which indicates that xy overflows.

If $p + q = n + 1$, then it follows that

$$2^{n-1} = 2^{p+q-2} < |x| \cdot |y| \leqslant 2^{p+q} = 2^{n+1} \leqslant M,$$

so that any overflow can be detected by the presence of the "wrong" sign on z or by the fact that $z = 0$ while $x \neq 0 \neq y$. Finally, if $p + q = n + 2$, then we have

$$2^{n} = 2^{p+q-2} < |x| \cdot |y| \leqslant 2^{p+q} = 2^{n+2} \leqslant 2M,$$

which means that the wrong sign on z (or $z = 0$) will have indicated any overflow such that

$$M/2 < |x| \cdot |y| \leqslant M$$

or such that

$$3M/2 < |x| \cdot |y| \leqslant 2M.$$

Therefore, if $p + q = n + 2$, it remains for us to distinguish between two cases:

$$\text{Case A} - M < |x| \cdot |y| \leqslant 3M/2;$$

$$\text{Case B} - 2^{n} < |x| \cdot |y| \leqslant M/2.$$

Clearly, in Case A there is overflow and in Case B there is no overflow.

To distinguish between Cases A and B, let us define z_1 to be the computed product of 2^{p-1} and $|y|$. Then, in Case A, we have

$$M/2 < |x/2| \cdot |y| \leqslant 2^{p-1} \cdot |y| \leqslant 2^{p-1} \cdot 2^{q} = 2^{n+1} \leqslant M,$$

which means that z_1 will be negative or zero. But, in

Case B, we have

$$2^{n-1} = 2^{p+q-3} < |x/2| \cdot |y| \leq 2^{p-1} \cdot |y|$$

$$< |x| \cdot |y| \leq M/2,$$

which indicates that z_1 will be positive. Therefore, when $p + q = n + 2$, xy overflows if and only if z_1, the computed product of 2^{p-1} and $|y|$, is negative or zero. This completes the proof of

Theorem 2.4 (Multiplicative Overflow Detection - Method II) - Let z be the computed product of x and y, where $|x| \geq |y|$, in a residue number system in which the absolute value of all integers is no greater than $M/2$. If $x \neq 0 \neq y$, let p, q, and n be the positive integers satisfying (2.7) and (2.8), and let z_1 be the computed product of 2^{p-1} and $|y|$. Then, xy overflows if and only if one or more of the following conditions is satisfied:

A. $z = 0$ while $x \neq 0 \neq y$;

B. z has the wrong sign $(z \neq 0)$;

C. $0 < |z| < |x|$;

D. $p + q = n + 3$;

E. $p + q = n + 2$ and $z_1 \leq 0$.

* * *

Interestingly enough, it is also possible to determine whether or not multiplicative overflow occurs when

$n + 1 \leqslant p + q \leqslant n + 2$ by using the additive overflow detection procedure. If we define c and d to be integers such that

$$c = |x| - 2^{p-1} \quad \text{and} \quad d = |y| - 2^{q-1}, \tag{2.9}$$

then we have

$$|x| \cdot |y| = 2^{p+q-2} + c \cdot 2^{q-1} + d \cdot 2^{p-1} + cd. \tag{2.10}$$

From (2.7) and (2.9) if follows that

$$0 < c \leqslant 2^{p-1} \quad \text{and} \quad 0 < d \leqslant 2^{q-1},$$

so that each of the four terms on the right side of (2.10) is not greater than

$$2^{p+q-2} \leqslant 2^{n} \leqslant M/2.$$

Hence, it follows that xy overflows if and only if additive overflow occurs in calculating the sum in the right side of equation (2.10).

While this technique of using additive overflow to detect multiplicative overflow seems simpler than using z_1 as prescribed in Theorem 2.4, it turns out that one to three magnitude comparisons (two to six conversions from residue to mixed-radix notation) are required to determine whether or not any additive overflow occurs in (2.10), while only one sign test (one residue to mixed-radix conversion) is necessary to check condition E. in Theorem

2.4. Hence, using z_1 as prescribed in Theorem 2.4 is "faster" than using equation (2.10).

It should be mentioned that the requirement of a stored table such as is needed for the overflow detection procedure given in Theorem 2.4 is quite reasonable. The table itself would not be very large since it need contain only those (positive integer) powers of two within the computer range. Furthermore, the integers p and q would be obtained easily from the table by a simple "look-up" procedure in which the mixed-radix coefficients of $|x|$ and $|y|$ would be compared with those of the powers of two stored in the table, and the mixed-radix coefficients of $|x|$ and $|y|$ would already have been computed in order to perform the sign and magnitude tests A. - D.

To illustrate the use of Theorem 2.4, let us consider the multiplication of $x = -31$ and $y = 8$ in the residue number system in which $M = 210$ and the range is -104 through 105. First, we note that, since the computed product of x and y in this system is $z = -38$, none of the conditions A. - C. in Theorem 2.4 is satisfied.

Second, we obtain $p = 5$ and $q = 3$, from which we find that

$$p + q = 8 = n + 2,$$

since $n = 6$. Next, calculating z_1 gives

$$2^{p-1} \cdot |y| = 16 \cdot 8 = 128 \equiv -82 \pmod{210}$$

or $z_1 = -82$. Since condition E. is now satisfied, multiplicative overflow is indicated.

For comparison, we note that using (2.10) to detect overflow in the above example requires at least one magnitude comparison (meaning two residue to mixed-radix conversions) to detect additive overflow in

$$\begin{aligned} |x| \cdot |y| &= 2^{p+q-2} + c \cdot 2^{q-1} + d \cdot 2^{p-1} + cd \\ &= 2^6 + 15 \cdot 2^2 + 4 \cdot 2^4 + 15 \cdot 4 \\ &= 64 + 60 + 64 + 60. \end{aligned}$$

Only one residue to mixed-radix conversion was required to find the sign of z_1.

We have now shown how to detect overflow resulting from addition, subtraction, and multiplication in modular arithmetic computers - or, at least, in those having certain "select" computer ranges. In all cases concerned, two or more mixed-radix conversions are necessary to

determine whether or not overflow has occurred, and in some instances, considerably more computation than that is necessary to confirm the presence or absence of overflow. This means that overflow detection in modular arithmetic computers will always be somewhat slower than in comparable conventional digital computers and that, in general, more complicated circuitry will be needed for overflow detection in modular arithmetic computers. However, this handicap is not as great as it might seem, since overflow detection tests need not be used as often in modular arithmetic computers as in conventional computers. The reason for this is that, unlike the truncation in conventional computers, the truncation used in residue number systems often permits the correct answers to be obtained even though overflows may have occurred at many intermediate steps in the calculations. For example, in calculating the partial sum of an alternating series, the programmer of the modular arithmetic computer may completely ignore the fact that the individual terms in the series overflow if he is certain that the partial sum itself will be within the computer range. In fact, this particular property of residue number systems will be used extensively in performing some of the important calculations needed in

the division and square root methods described in the next two chapters.

Therefore, although overflow detection in modular arithmetic computers is somewhat more cumbersome than might be desired, we have shown that it is possible to detect such overflow and we have given methods whereby the detection can be accomplished in a reasonable amount of computing time. Although we have found it necessary to introduce a small table of powers of two in order to allow a more efficient method - namely, that of Theorem 2.4 - for detecting multiplicative overflow, we shall find in the following two chapters that this same table can also be used to facilitate other very important operations in modular arithmetic computers.

CHAPTER III

DIVISION

A. Division in Residue Number Systems. Normally, when we speak of the division of, say, x by y in any number system, we are referring to the process of obtaining the solution z of the linear equation

$$yz = x.$$

Assuming that multiplication is commutative and associative in the number system, the existence of such a z (for all x) is equivalent to the existence of a multiplicative inverse y^{-1} of y such that

$$y \cdot y^{-1} = y^{-1} \cdot y = 1,$$

where 1 denotes the multiplicative identity. (See Jacobson [14], p. 24.) Clearly, if such an inverse y^{-1} exists, then $z = xy^{-1}$.

In a commutative ring, the existence of a multiplicative inverse for any element y is dependent in part upon whether or not y is a zero divisor - that is, whether or not there exists a $w \neq 0$ in the ring such that

$yw = 0$, where 0 denotes the additive identity in the ring. In particular, if we assume there exists a multiplicative inverse y^{-1} for the zero divisor y, then we have

$$w = 1 \cdot w = (y^{-1} \cdot y)w = y^{-1}(yw) = y^{-1} \cdot 0 = 0,$$

which contradicts the definition of w. Hence, if y is a zero divisor, then it has no multiplicative inverse, and "division" by y is not possible.

It is not hard to verify that, under addition and multiplication modulo M, the product of the moduli, residue number systems are always commutative rings. However, unless M is a prime, in which case it is the only modulus, all residue number systems contain non-zero elements which are zero divisors. (See Jacobson [14], pp. 66-68.) In particular, if y is any non-zero integer in a residue number system and if y is not relatively prime to all the moduli for the system, then y is a zero divisor and has no multiplicative inverse. Hence, unless y is relatively prime to all the moduli, division by y is impossible - that is, for each x in the residue number system,

$$yz \equiv x \pmod{M} \tag{3.1}$$

either has no solution z or has several different solutions. (For example, in the residue number system based on the moduli 2, 3, 5, and 7, there exists no integer z such that

$$36 \cdot z \equiv 59 \pmod{M},$$

but there are <u>five</u> solutions to

$$-95 \cdot z \equiv 20 \pmod{M}:$$

$z = 2$, $z = 44$, $z = 86$, $z = -82$, and $z = -40$.) Furthermore, even if the multiplicative inverse of an integer y does exist in a residue number system, the solution z to (3.1) is not the quotient one would expect from most computers unless x is an exact (integer) multiple of y. The reason for this is that the multiplication in (3.1) is performed modulo M. (For example, in the residue number system used above, the solution z to (3.1) for $x = 78$ and $y = 37$ is $z = -66$.) Hence, even when division is possible in a residue number system, the quotient obtained in many cases - in fact, in most cases - is not suitable for use in most computer applications.

There are also zero divisors in the number systems used in conventional digital computers, but there the problem discussed above is avoided by using a different

definition of division. In particular, when a number x is divided by a non-zero number y in a conventional digital computer, the "quotient" which results is usually the "integral portion" of the true quotient - that is, the greatest integer not exceeding $|x/y|$, preceded by the proper sign. To obtain this "quotient" in conventional digital computers, "division" is usually performed by a sequence of subtractions and "shifts" which amounts to generating the quotient by counting the number of times the divisor can be subtracted from the dividend before a sign change occurs.

There is no reason why we cannot carry over this new definition of quotient for use in residue number systems or, for that matter, why we cannot use some other definition of quotient such as, say, the "nearest" integer to x/y. However, we do encounter considerable difficulty in carrying over to residue number systems the method of division by subtracting and "shifting." In particular, since sign determination is more difficult in modular arithmetic computers than in other digital computers (the results of Chapter I notwithstanding), the method of simple repeated subtraction of the divisor from the dividend

is prohibitively time consuming and inefficient. Furthermore, if we try to speed up this procedure by using the technique of "shifting" used in conventional computers, we find that performing a "shift" in residue number systems is equivalent to performing the division itself.

Therefore, we shall now seek some other procedure whereby we can conveniently calculate some reasonable approximation to the quotient of the (non-zero) integers x and y in a residue number system. We shall present a new method for finding the nearest integer to (or the integral portion of) the quotient x/y, and then show how this method can be extended to give a much better approximation to that quotient. Finally, we shall apply these new "division" methods to enable modular arithmetic computers to perform "floating-point" arithmetic - a capability heretofore possessed only by conventional digital computers.

B. Division Algorithms for Residue Number Systems.

Let us now assume that x and y are non-zero integers in a residue number system whose range consists of the integers $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ (if M is odd) or $-\frac{M}{2}+1$

through $\frac{M}{2}$ (if M is even). Let us assume further that there exists a table of powers of two from 2^1 through 2^n where, as in (2.8), n is the integer such that

$$2^n \leqslant M/2 < 2^{n+1}. \tag{3.2}$$

As we explained in Chapter II, we may obtain from this table the non-negative integers p and q such that

$$2^{p-1} < |x| \leqslant 2^p \quad \text{and} \quad 2^{q-1} < |y| \leqslant 2^q. \tag{3.3}$$

From these inequalities it then follows that

$$2^{p-q-1} = 2^{p-1}/2^q < |x|/|y| = |x/y| < 2^p/2^{q-1} = 2^{p-q+1}, \tag{3.4}$$

so that it seems reasonable to choose

$$z_1 = 2^{p-q}$$

as a first approximation to $|x/y|$. However, if $|x| < |y|$, then the nearest integer z to $|x/y|$ must be either zero or one. In that case, we may ignore z_1 and choose between the two possible values for z by calculating $2|x|$ and comparing it with $|y|$. If

$$0 < 2|x| < |y|,$$

then

$$|x/y| < 1/2,$$

so that we should set $z = 0$. But if $2|x| \geqslant |y|$, then

$$1/2 \leqslant |x/y| < 1,$$

so we should set $z = 1$. Moreover, since

$$|x| < |y| \leqslant M/2,$$

we have $2|x| < M$, so that the computed product $2|x|$ will be negative if multiplicative overflow occurs. Hence, if the computed product $2|x| < 0$, then the true product satisfies

$$2|x| > M/2 \geqslant |y|,$$

and we should again set $z = 1$.

On the other hand, if $|x| \geqslant |y|$, then the nearest integer to $|x/y|$ is greater than or equal to one. Also, in this case, $p-q \geqslant 0$ so that z_1 is an integer. If we define the integer e_1 by

$$e_1 = |x| - |y| \cdot z_1 \tag{3.5}$$

and combine this definition with (3.3) and with the definition of z_1, we have

$$2^{p-1} - 2^q 2^{p-q} < e_1 < 2^p - 2^{q-1}2^{p-q} \tag{3.6}$$

or

$$|e_1| < 2^{p-1}.$$

But, since $|x| \leqslant M/2$, it follows that $p \leqslant n+1$. Hence,

$$|e_1| < 2^{p-1} \leqslant 2^n \leqslant M/2,$$

which means that e_1 is within the computer range.

Now, if $e_1 = 0$, it then follows from $y \neq 0$ and

$$e_1 = |x| - |y| \cdot z_1 = (|x/y| - z_1) \cdot |y|$$

that z_1 is exactly equal to $|x/y|$, so that we may set $z = z_1$ to obtain the nearest integer to $|x/y|$ - namely, $|x/y|$ itself. If, however, $e_1 \neq 0$, we then turn to the table of powers of two to obtain the non-negative integer r_1 such that

$$2^{r_1 - 1} < |e_1| \leqslant 2^{r_1}. \tag{3.7}$$

Since

$$|e_1| = \big||x/y| - z_1\big| \cdot |y|,$$

it follows from this definition of r_1 that

$$\begin{aligned} 2^{r_1-q-1} = 2^{r_1-1}/2^q &< \big||x/y| - z_1\big| \\ &< 2^{r_1}/2^{q-1} = 2^{r_1-q+1}. \end{aligned} \tag{3.8}$$

Hence, if $r_1 \geqslant q$, then

$$\big||x/y| - z_1\big| > 2^{r_1-q-1} \geqslant 2^{-1} = 1/2,$$

so z_1 is surely not the nearest integer to $|x/y|$. Since $r_1 \geqslant q$ also implies that 2^{r_1-q} is an integer, it seems reasonable, in view of (3.8), to use this quantity as a "correction" to z_1. Furthermore, inasmuch as e_1 is negative if $z_1 > |x/y|$ and positive if $z_1 < |x/y|$, we may use this correction to obtain a second approximation z_2 to $|x/y|$ by defining the integer z_2 by

$$z_2 = z_1 + (\operatorname{sign} e_1)2^{r_1-q}.$$

As we did for z_1, we calculate e_2 from z_2 by taking $i = 2$ in

$$e_i = |x| - |y| \cdot z_i. \tag{3.9}$$

If $e_2 \neq 0$, we define r_2 to be the non-negative integer satisfying

$$2^{r_i-1} < |e_i| \leqslant 2^{r_i}. \tag{3.10}$$

Repeating this procedure for $i = 2, 3, \ldots$, we define the next approximation to $|x/y|$ whenever $e_i \neq 0$ and $r_i \geqslant q$ by

$$z_{i+1} = z_i + (\text{sign } e_i) \cdot 2^{r_i-q}. \tag{3.11}$$

Then if $e_i > 0$ for any i, we have

$$\begin{aligned} e_{i+1} = |x| - |y| \cdot z_{i+1} &= |x| - |y| \cdot (z_i + 2^{r_i-q}) \\ &= e_i - |y| \cdot 2^{r_i-q}. \end{aligned}$$

Combining this with (3.10) gives

$$\begin{aligned} -2^{r_i-1} = 2^{r_i-1} - 2^q 2^{r_i-q} &< e_{i+1} \\ &< 2^{r_i} - 2^{q-1} 2^{r_i-q} = 2^{r_i-1} \end{aligned} \tag{3.12}$$

or

$$|e_{i+1}| < 2^{r_i-1}.$$

Similarly, if $e_i < 0$, then we have

$$e_{i+1} = e_i + |y| \cdot 2^{r_i-q},$$

from which we obtain

$$\begin{aligned} -2^{r_i-1} = -2^{r_i} + 2^{q-1} 2^{r_i-q} &< e_{i+1} \\ &< -2^{r_i-1} + 2^q 2^{r_i-q} = 2^{r_i-1} \end{aligned} \tag{3.13}$$

or

$$|e_{i+1}| < 2^{r_i - 1}.$$

Hence, either $e_{i+1} = 0$ or else it follows from (3.10) that

$$r_{i+1} \leqslant r_i - 1.$$

Thus, eventually for some i, either $e_i = 0$, in which case $z = z_i$ is exactly $|x/y|$, or else we have $r_i < q$. If, in the latter case, $r_i \leqslant q-2$, then it follows from (3.9) that

$$\begin{aligned} \left| |x/y| - z_i \right| &= |e_i| / |y| < 2^{r_i} / 2^{q-1} \\ &= 2^{r_i - q + 1} \leqslant 2^{-1} = 1/2, \end{aligned}$$

so that setting $z = z_i$ makes z the nearest integer to $|x/y|$. On the other hand, if $r_i = q - 1$, then (3.9) gives

$$\begin{aligned} 1/4 = 2^{-2} = 2^{r_i - 1} / 2^q &< |e_i| / |y| \\ = \left| |x/y| - z_i \right| &< 2^{r_i} / 2^{q-1} = 2^0 = 1, \end{aligned}$$

in which case z_i differs from the nearest integer z to $|x/y|$ by at most one. In this case, we calculate $2e_i$ and compare it to e_i and $|y|$. If $2e_i$ and e_i have opposite signs, then multiplicative overflow must have occurred in calculating $2e_i$, so that $2|e_i|$ is clearly greater than $|y|$. (Overflow occurs

in $2e_i$ if and only if $2e_i$ and e_i have opposite signs since

$$2|e_i| \leqslant 2 \cdot 2^{r_i} = 2 \cdot 2^{q-1} = 2^q \leqslant 2^{n+1} \leqslant M.)$$

On the other hand, if $2e_i$ and e_i have the same sign, then no overflow has occurred, so we may compare $2e_i$ and $|y|$ directly. If

$$-|y| \leqslant 2e_i < |y|,$$

then it follows that

$$-1/2 \leqslant |x/y| - z_i < 1/2,$$

so that setting $z = z_i$ again makes z the nearest integer to $|x/y|$. (If $|x/y| = k + 1/2$ for some non-negative integer k, we round the $1/2$ "upward" to obtain $z = k + 1$.) If e_i is negative and $2e_i$ overflows or if both $2e_i$ and e_i are negative and $2e_i < -|y|$, then we set $z = z_i - 1$ to get

$$-1/2 \leqslant |x/y| - z < 1/2. \qquad (3.14)$$

And if e_i is positive and either $2e_i$ overflows or $2e_i \geqslant |y|$, we then set $z = z_i + 1$ to arrive at (3.14). Thus, we have obtained a division algorithm, a "flow chart" of which appears in Figure II, and we have established

Theorem 3.1 (Division Algorithm) - Let x and y be non-zero integers in a residue number system in which the

absolute values of all integers are not greater than $M/2$. Let p and q be the non-negative integers defined by (3.3) and let z be the integer determined as follows:

1. If $|x| < |y|$, set $z = 0$ if $0 < 2|x| < |y|$, and set $z = 1$ if $2|x| < 0$ or $2|x| \geq |y|$.
2. If $|x| \geq |y|$, set $z_1 = 2^{p-q}$ and calculate e_1 by (3.9). If $e_i = 0$ for any i, set $z = z_i$.
3. If $e_i \neq 0$ for any i, let r_i be the non-negative integer satisfying (3.10). If $r_i \geq q$, calculate z_{i+1} by (3.11) and go back to step 2 above to calculate e_{i+1}. If $r_i = q - 1$, calculate $2e_i$. If $2e_i$ and e_i have opposite signs or if $2e_i$ and e_i have the same sign and either of

$$2e_i < -|y| \quad \text{or} \quad 2e_i \geq |y|$$

is true, set

$$z = z_i + (\text{sign } e_i) \cdot 1.$$

Otherwise, set $z = z_i$.

Then, z satisfies

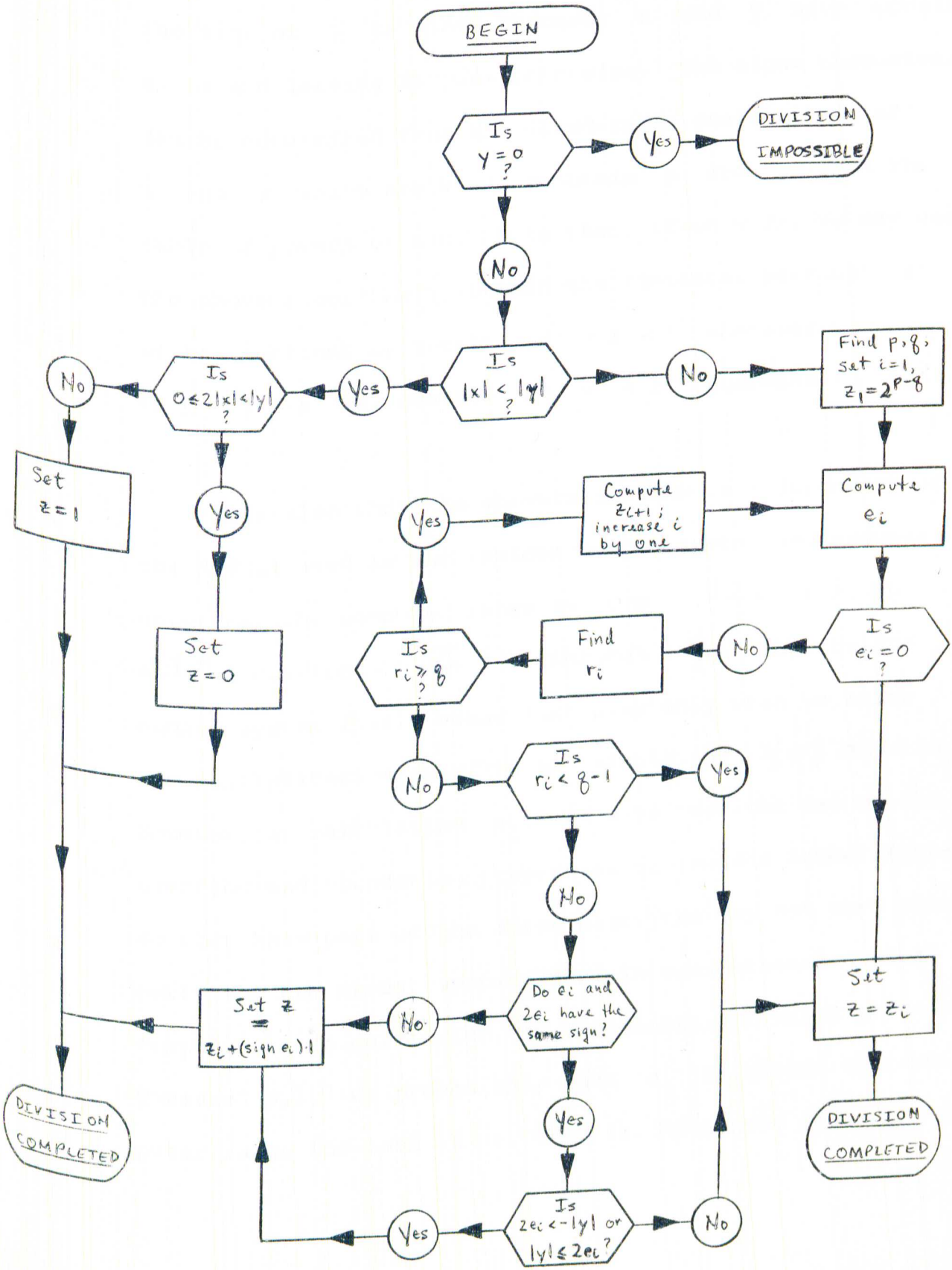
$$-1/2 \leq |x/y| - z < 1/2;$$

that is, z is the nearest integer to $|x/y|$.

* * *

Once having obtained the nearest integer z to $|x/y|$ by the procedure given in the above theorem, we

Figure II - Division Algorithm



can easily obtain the nearest integer to x/y by changing the sign of z to minus whenever x and y have opposite signs and leaving it plus otherwise. The signs themselves can be determined from the mixed-radix coefficients of x and y which are used to obtain p and q from the table of powers of two. Note that, if we wish, we may use the above algorithm to obtain the "integral portion" z' of the quotient by setting $z' = z - 1$ whenever $|x| - |y| \cdot z$ is negative and $z' = z$ otherwise.

Note also that the above algorithm is independent of the moduli used in the residue number system and assumes only that the computer range is from $-\frac{M-1}{2}$ through $+\frac{M-1}{2}$ or from $-\frac{M}{2}+1$ through $\frac{M}{2}$. The residue number system itself comes into play only when we allow for multiplicative overflow in calculating $2|x|$ and $2e_i$. However, in calculating e_i, we make implicit use of the overflow and truncation properties of residue number systems, so that this part of the above algorithm may not work properly for the number systems used in conventional digital computers. In particular, in the discussion which preceded Theorem 3.1, we proved only that e_i is within the computer range for each i - that is, we proved that e_1 is

within the computer range and that, for each i, $|e_{i+1}| < |e_i|$ - but we did not prove anything at all about overflow in the intermediate results used in calculating e_i. In general, these intermediate results are not within the computer range so that multiplicative overflow usually occurs in the calculation of $|y| \cdot z_i$. But, because all operations are performed modulo M in the residue number system, the correct result is still obtained when we subtract the computed product of $|y|$ and z_i from $|x|$. Hence, although few assumptions are made about the particular residue number system being used, the algorithm given in Theorem 3.1 makes rather important use of the fact that the calculations are performed in a residue number system (as opposed to the number systems used in conventional digital computers).

Although we can prove only that $r_{i+1} \leq r_i - 1$ for each i, the algorithm given in Theorem 3.1 usually converges quite rapidly to z. For example, if $x = 136,047$ and $y = 85$, we apply Theorem 3.1 as follows to obtain the nearest integer z to $|x/y| = 1600.5529\ldots$:

First, since $|x| > |y|$, we obtain $p = 18$ and $q = 7$ from the table of powers of two. Setting

$$z_1 = 2^{p-q} = 2^{11} = 2048$$

gives

$$e_1 = 136,047 - 85 \cdot 2048 = -38,033.$$

Next, since $r_1 = 16$ is greater than $q = 7$, we calculate z_2 by

$$z_2 = 2048 - 2^9 = 1536.$$

Then, from $e_2 = 5487$ and $r_2 = 13$, we obtain

$$z_3 = 1536 + 2^6 = 1600,$$

$$e_3 = 47, \quad \text{and} \quad r_3 = 6.$$

Since $r_3 = q-1$, we calculate $2e_3 = 94$, which is clearly greater than $|y| = 85$. Hence, we set

$$z = z_3 + 1 = 1601,$$

which is indeed the nearest integer to $|x/y|$.

Perhaps the most interesting feature of the above division procedure is that it can be extended to provide a much better approximation to $|x/y|$ than the nearest integer obtained in Theorem 3.1. In particular, the algorithm of Theorem 3.1 can be modified to yield an approximation to $|x/y|$ in the form $w \cdot 2^j$, where w is an integer in the residue number system and j is a negative integer. Using $z_1 = 2^{p-q}$ and the definition (3.5) for e_1, we were able to show in the proof of Theorem 3.1 that

$$|e_1| < 2^{p-1}.$$

Hence, if we define w_1 and f_1 respectively by

$$w_1 = z_1 \cdot 2^{-j} = 2^{p-q-j} \quad \text{and} \quad f_1 = 2^{-j}\left|x\right| - \left|y\right| \cdot w_1,$$

then we have

$$w_1 = 2^{p-q-j} < 2^{n+1-q} \leqslant 2^n \leqslant M/2$$

whenever

$$0 \geqslant j \geqslant p - n - 1 \tag{3.15}$$

and $q \geqslant 1$. If $q = 0$, then $\left|y\right| = 1$ and no elaborate division procedure is necessary; that is, we may set $w = \left|x\right| \cdot 2^{-j}$, where

$$0 \geqslant j \geqslant p-n,$$

to obtain

$$w \cdot 2^j = \left|x/y\right| = \left|x\right|.$$

Also, if j satisfies (3.15), then we have, as in (3.6),

$$2^{-j}2^{p-1} - 2^q 2^{p-q-j} < f_1 < 2^{-j}2^p - 2^{q-1}2^{p-q-j}$$

which gives

$$\left|f_1\right| < 2^{p-j-1} \leqslant 2^n \leqslant M/2.$$

Hence, by choosing j to satisfy (3.15), we are assured that the integers w_1 and f_1 are within the computer range.

If $f_1 = 0$, then $\left|x/y\right| = w_1 \cdot 2^j$ and our division is completed. Otherwise, as in (3.7), we define the non-negative integer s_1 by

$$2^{s_1-1} < \left|f_1\right| < 2^{s_1},$$

and if $s_1 \geqslant q$, we obtain a better approximation $w_2 \cdot 2^j$ to $|x/y|$ by setting

$$w_2 = w_1 + (\text{sign } f_1) \cdot 2^{s_1 - q}.$$

Clearly, we may continue this procedure as before, defining the integer f_i by

$$f_i = 2^{-j}|x| - |y| \cdot w_i \tag{3.16}$$

and, if $f_i \neq 0$, defining the non-negative integer s_i by

$$2^{s_i - 1} < |f_i| \leqslant 2^{s_i}. \tag{3.17}$$

If $s_i \geqslant q$, the next approximation $w_{i+1} \cdot 2^j$ to $|x/y|$ is defined by

$$w_{i+1} = w_i + (\text{sign } f_i) \cdot 2^{s_i - q}. \tag{3.18}$$

Thus, if $f_i > 0$, then

$$f_{i+1} = f_i - |y| \cdot 2^{s_i - q}$$

and it follows as before in (3.12) that

$$-2^{s_i - 1} = 2^{s_i - 1} - 2^{s_i - q} 2^q < f_{i+1}$$
$$< 2^{s_i} - 2^{q-1} 2^{s_i - q} = 2^{s_i - 1}.$$

And if $f_i < 0$, then

$$f_{i+1} = f_i + |y| \cdot 2^{s_i - q}$$

and it follows as in (3.13) that

$$-2^{s_i - 1} = -2^{s_i} + 2^{q-1} 2^{s_i - q} < f_{i+1}$$
$$< -2^{s_i - 1} + 2^q 2^{s_i - q} = 2^{s_i - 1}.$$

Therefore, in either case,

$$|f_{i+1}| < 2^{s_i - 1},$$

so that it follows from (3.17) that

$$s_{i+1} \leqslant s_i - 1. \tag{3.19}$$

Hence, eventually for some i we must have that $f_i = 0$, in which case $w_i \cdot 2^j = |x/y|$, or else we must have $s_i < q$. If $s_i \leqslant q - 2$, then

$$\left| |x/y| - w_i \cdot 2^j \right| = 2^j |f_i| / |y| < 2^j 2^{s_i} / 2^{q-1} \leqslant 2^{j-1}.$$

If $s_i = q - 1$, we compare $2|f_i|$ with $|y|$ and set

$$w = w_i + (\text{sign } f_i) \cdot 1 \tag{3.20}$$

if $2|f_i| > |y|$. Otherwise, if $2|f_i| \leqslant |y|$, we set $w = w_i$. This again gives

$$\left| |x/y| - w \cdot 2^j \right| \leqslant 2^{j-1}.$$

Thus, we obtain a division algorithm whose "flow chart" is essentially the same as that given in Figure II, but which is capable of giving a more accurate approximation to the quotient than the algorithm of Theorem 3.1. This completes the proof of

Theorem 3.2 (Division Algorithm - Extended Form) - Let x and y be non-zero integers in a residue number system in which the absolute values of the integers are not greater than $M/2$. Let p and q be the non-negative integers defined by (3.3), let n be the positive integer satisfying (3.2), let j be any integer satisfying (3.15), and let w be the integer determined as follows:

1. If $q = 0$, set $w = |x| \cdot 2^{-j}$.

2. If $q \neq 0$, set $w = 0$ if $p-q-j+1 < 0$ or if $p-q-j+1 = 0$ and $2^{-j+1}|x| < |y|$. If $p-q-j+1 = 0$ and $2^{-j+1}|x| \geq |y|$, set $w = 1$.

3. If $p-q-j \geq 0$, set $w_1 = 2^{p-q-j}$ and calculate f_1 by (3.16). If $f_i = 0$ for some i, set $w = w_i$.

4. If $f_i \neq 0$ for any i, let s_i be the non-negative integer satisfying (3.17). If $s_i \geq q$, define w_{i+1} by (3.18) and go back to step 3 to calculate f_{i+1}. If $s_i = q - 1$, calculate $2|f_i|$ and obtain w from w_i by (3.20) if $2|f_i| > |y|$. Otherwise, set $w = w_i$.

Then, w satisfies

$$\left| |x/y| - w \cdot 2^j \right| \leq 2^{j-1};$$

that is, w differs from $2^{-j}|x/y|$ by at most $1/2$.

* * *

One of the more desirable features of the algorithm given in this theorem is that j may be changed in successive "iterations" - that is, j may decrease as i increases. In particular, for any i, f_{i+1} will still be within the computer range if we decrease j by as much as $t_i = n - s_i + 1$ for the $(i + 1)$st iteration.

Doing this increases w_{i+1} and f_{i+1} by a factor of 2^{t_i}, but, by (3.19) and the definition of t_i, we have

$$|f_{i+1}| \leqslant 2^{s_i+1} \leqslant 2^{s_i-1}2^{t_i} \leqslant 2^n \leqslant M/2.$$

Hence, we may decrease j with each iteration in order to obtain increased accuracy in the approximation $w.2^j$ of $|x/y|$. However, in order to keep w_i within the computer range as we decrease j, we must also require that

$$0 \geqslant j \geqslant (p - q - n + 1) = j_{min}. \tag{3.21}$$

Since

$$|x/y| < 2^{p-q+1},$$

keeping j in the interval specified in (3.21) gives

$$|w_i| < 2^{p-q-j+1} \leqslant 2^n \leqslant M/2.$$

And if $f_1 \leqslant 0$, then $|x/y| \leqslant 2^{p-q}$, so that we may further decrease j_{min} to $p - q - n$. Note that, as for the e_i's in Theorem 3.1, the integers f_i in Theorem 3.2 are always within the computer range, but the intermediate results - the products $2^{-j}|x|$ and $|y| \cdot w_i$, in (3.16) - may be outside the range and may thus cause multiplicative overflow. However, if the multiplications and the subtraction are performed modulo M in a residue number system, the computed difference f_i of the computed products $2^{-j}|x|$ and $|y| \cdot w_i$ will still be correct.

As before in Theorem 3.1, our proof of Theorem 3.2 guarantees - in (3.19) - only that the error in the approximation $w_i \cdot 2^j$ to $|x/y|$ will diminish by approximately a factor of two in each successive iteration. Cases actually exist - namely, when $|x/y| = 1/3$ or $2/3$ - in which the algorithm of Theorem 3.2 converges in exactly this way - that is, $s_{i+1} = s_i - 1$ - so that we cannot hope to obtain a "sharper" estimate of convergence than that given in the above proof. However, as the following example suggests, the algorithm of Theorem 3.2 actually gives, in many cases, a rate of convergence considerably greater than the minimal rate given in the proof.

To illustrate the use of Theorem 3.2, let us assume our residue number system to be based on the eight moduli 2, 3, 5, 7, 11, 13, 17, and 19. Since this gives $M/2 = 4,849,845$ and since $2^{22} = 4,194,304$, we have $n = 22$. If we now apply Theorem 3.2 and the remarks following it to approximate the quotient of $x = 829,314$ and $y = 6,057$ with the maximal accuracy permitted by our residue number system, we obtain first, from (3.3) and (3.15),

$$p = 20,\ q = 13,\ \text{and}\ j = -3.$$

Hence, for the first "iteration" we set

$$w_1 = 2^{20-13-(-3)} = 2^{10} = 1024$$

and $j = -3$. Calculating f_1 from (3.16) gives

$$f_1 = 2^3 \cdot 829,314 - 6057 \cdot 1024 = 432,144$$

from which we get $s_1 = 19$ using (3.17). For the second iteration, we set

$$w_2 = w_1 + 2^{s_1 - q} = 1024 + 2^6 = 1088,$$

and since we wish to obtain maximal accuracy, we decrease j by

$$t_1 = n - s_1 + 1 = 4.$$

This gives $j = -7$ and, by multiplying w_2 by 2^{t_1} to compensate for the change in j, $w_2 = 17,408$. Proceeding with the second and following iterations as prescribed in Theorem 3.2, we obtain the results given in Table IV. From (3.21), we calculate that $j_{min} = -14$, so that we halt the algorithm when $j = -14$ and $s_i < q = 13$. The resulting approximation,

$$2,243,269 \cdot 2^{-14} = 136.91827392\ldots$$

differs from

$$|x/y| = 136.91827637\ldots$$

by approximately 0.00000245, which is considerably less than the maximum error

$$2^{j-1} = 2^{-15} = 0.00003052\ldots$$

predicted in Theorem 3.2.

Table IV - Application of Theorem 3.2: Sample Division Problem

$x = 829,314$ $\quad y = 6,057$ $\quad \lfloor x/y \rfloor = 136.91827637...$

i	w_i	j	f_i	s_i	$w_i \cdot 2^j$	Error = $x/y - w_i \cdot 2^j$
1	1,024	-3	432,144	19	128.00000000...	+8.91827637...
2	17,408	-7	711,936	20	136.00000000...	+0.91827637...
3	140,288	-10	-506,880	19	137.00000000...	-0.08172363...
4	2,243,584	-14	-1,907,712	21	136.93750000...	-0.01922363...
5	2,243,328	-14	-357,120	19	136.92187500...	-0.00359863...
6	2,243,264	-14	30,528	15	136.91796875...	+0.00030762...
7	2,243,268	-14	6,300	13	136.91821289...	+0.00006348...
8	2,243,269	-14	243	8	136.91827392...	+0.00000245...

Note that, in this example, w would still be within the computer range if j were decreased to -15. However, if this were done, w would not be less than or equal to $2^n = 2^{22}$, which is the criteria by which we determined j_{min}. Hence, it will happen occasionally that we can still keep w within the computer range when we make j (one) less than j_{min}. In that case we have

$$2^n < w \leqslant M/2.$$

The probability of these cases, in which we obtain slightly less than the "maximal" accuracy consistent with the range of the residue number system, can be minimized by choosing the moduli of that system such that their product is as close as possible, but not less than, a power of two — that power being 2^{n+1}, where n satisfies (3.2).

Now we have shown how to perform "division" in residue number systems. The algorithms we have given may be used to give either an integer approximation to the quotient or a closer approximation in the form $w*2^j$, where w is an integer in the residue number system and j is a negative integer. In particular, we have illustrated in the above example how the "exponent" j can be manipulated so that the algorithm of Theorem 3.2 yields the most accurate

approximation consistent with keeping w within the computer range. Next, we shall show how these division algorithms can be used to help perform other useful operations in modular arithmetic computers.

C. Floating-Point Arithmetic. In most digital computers provision is made for representing very large integers and very small fractions by what is essentially the equivalent of "scientific notation." That is, instead of representing "six-hundred billion" by 600,000,000,000, the more compact notation 6×10^{11} is used, and instead of representing "minus two millionths" by -0.000002, -2×10^{-6} is used. In digital computers, the equivalent of this scientific notation is achieved by reserving a certain number of digits in each "number" for the "mantissa" - 6 and -2 in the above examples - and using the other digits for the "exponent" - 11 and -6 in the above examples. The computer programmer then has the option of regarding the numbers within the machine as being in the ordinary radix notation, which he calls "fixed-point" notation, or in the above equivalent of scientific notation, which he calls "floating-point" notation.

Hence, he has a choice of two sets of rules by which additions, subtractions, multiplications, and divisions are performed within the computer: "fixed-point" arithmetic and "floating-point" arithmetic.

For example, in the IBM 7090 computer, a widely-used large-scale computer, a number may be regarded as a 35-bit binary integer, preceded by a "sign bit," or as a 27-bit binary fraction, preceded by a sign bit and followed by an 8-bit binary exponent. The programmer may instruct the 7090 to add two numbers as signed 35-bit binary integers by using an "ADD" instruction, or he may use a "FAD" (floating add) instruction, which causes the 7090 to "shift" the 27-bit fraction in one of the addends until the corresponding exponent agrees with that of the other addend before adding the two 27-bit fractions. In the same manner, other instructions may be used to cause the 7090 to perform similar operations in subtracting, multiplying, and dividing fixed- and floating-point numbers.

Clearly, addition, subtraction, and multiplication modulo M, and the "nearest integer" division of

Theorem 3.1 are the modular arithmetic computer's equivalents of the 7090's fixed-point addition, subtraction, multiplication, and division. But because of the difficulties in performing the "shifts" necessary to align the exponents of the operands in modular arithmetic computers, these computers previously had no equivalents of the 7090's floating-point arithmetic operations. Now, however, we find that, by using the division procedures of Theorems 3.1 and 3.2, floating-point arithmetic operations can be performed in modular arithmetic computers in a relatively straightforward manner. In particular, we may represent very large integers or very small fractions in modular arithmetic computers in the form $x \cdot 2^j$, where x is an integer within the "fixed-point" computer range and represented in residue form and where j is another integer, probably represented in the usual binary form.

To simplify the "shifting" processes necessary to align properly the two operands for floating-point arithmetic operations, the 7090 computer assumes that the operands are given in "normalized" form - that is, the exponent in the floating-point representation of a non-zero number is adjusted so that the absolute value of the binary fraction in

in the representation is less than 1 but not less than 1/2. (Zero is represented in floating-point form by a zero exponent and a zero fraction.) After performing each floating-point arithmetic operation, the 7090 automatically adjusts the fraction and the exponent in the result to put it back into "normalized" form.

In modular arithmetic computers, too, floating-point arithmetic operations may be simplified by assuming that the operands have been "normalized" in some way. One possible normalization of the non-zero floating-point number $x \cdot 2^j$ might be to adjust the exponent j so that x satisfies

$$2^{n-1} < |x| \leqslant 2^n,$$

where n is the integer defined above in (3.2). Expressing a floating-point number in this way would provide the maximum number of significant digits consistent with keeping x within the (fixed-point) computer range. However, we feel that another form of normalization for modular arithmetic computers leads to simpler floating-point arithmetic operations. In particular, for a non-zero number in the form $x \cdot 2^j$, we prefer to adjust the exponent j in such a way that x satisfies

$$2^{m-1} < |x| \leq 2^m,$$

where m is the positive integer defined by

$$m = \left[\frac{n-1}{2}\right]. \tag{3.22}$$

In addition to simplifying, in particular, the operation of floating-point multiplication, this form of normalization yields reasonably simple floating add and subtract operations as well and it has the added advantage that the integer x can be represented by its residues with respect to each of the moduli in some (proper) subset of moduli whose product exceeds 2^m, while the exponent j is carried as a "residue" for one (or more) of the moduli.

We shall now describe how the division algorithms given above in Theorems 3.1 and 3.2 can be used to perform floating-point arithmetic operations in modular arithmetic computers with numbers normalized as described above. Unless stated otherwise, all divisions in the following will be understood to result in the nearest integer to the quotient as explained in Theorem 3.1.

1. Fixed-Point to Floating-Point Conversion. Converting a number from fixed- to floating-point is

essentially a "normalize" operation. If the fixed-point number v is zero, setting the floating-point number $u \cdot 2^j$ to zero - that is, setting $u = j = 0$ - completes the conversion. Otherwise, we obtain from the table of powers of two the non-negative integer p such that

$$2^{p-1} < |v| \leqslant 2^p$$

and compare p with the integer m defined above in (3.22). If $p < m$, we set $u = v \cdot 2^{m-p}$ and $j = p-m$; if $p = m$, we set $u = v$ and $j = 0$; and if $p > m$, we divide v by 2^{p-m}, set u equal to the result, and set $j = m-p$.

2. Floating-Point to Fixed-Point Conversion. If $j < 0$ for the number $u \cdot 2^j$ in normalized floating-point form, we convert $u \cdot 2^j$ to the fixed-point number v by dividing u by 2^{-j} and setting v equal to the result. If $j = 0$, we simply set $v = u$; and if $j > 0$, we multiply u by 2^j to obtain v, checking for multiplicative overflow if we wish.

3. Floating-Point Magnitude Comparison. Given the distinct normalized floating-point numbers

$$a = u \cdot 2^j \quad \text{and} \quad b = v \cdot 2^k, \tag{3.23}$$

we determine the signs of a and b from the signs of u and v, respectively, which are obtained in turn from the two-sided mixed-radix coefficients of u and v. If a and b have different signs, whichever of a and b is positive is obviously the greater. If a and b have the same sign, we compare j and k and conclude that, since a and b are both normalized, $a > b$ if a and b are negative and $j < k$ or if a and b are positive and $j > k$, and vice versa. If $j = k$, then $a > b$ if and only if $u > v$.

4. Floating-Point Addition and Subtraction. Given the normalized floating-point numbers a and b defined above in (3.23), we shall calculate the normalized floating-point number $c = t \cdot 2^i$ such that $c = a + b$. The difference of the numbers a and b can be obtained in a similar manner by adding a and b', where $b' = -b$. First, we compute $r = |j - k|$. If $r > m$, where m is defined in (3.22), then the smaller of $|a|$ and $|b|$ is too small to affect the floating-point representation of the larger when the addition is performed. Hence, we simply set $c = a$ - that is, we set $t = u$ and $i = j$ - if $j > k$ and $c = b$ if $j < k$ when $r > m$. Otherwise, if $r \leq m$, we set

$w = u + v \cdot 2^r$ and $h = j$ if $j < k$; we set $w = u \cdot 2^r + v$ and $h = k$ if $j > k$; and we set $w = u + v$ and $h = j = k$ if $j = k$. Then, $w \cdot 2^h$ is the floating-point sum of a and b, but since $w \cdot 2^h$ is not normalized, we must now perform a "normalization" operation to obtain c. From the table of powers of two we obtain the integer p such that

$$2^{p-1} < |x| \leqslant 2^p \tag{3.24}$$

and we compare p with the integer m defined in (3.22). If $p > m$, we set $t = w \cdot 2^{p-m}$ and $i = h + p - m$; if $p = m$, we set $t = w$ and $i = h$; and if $p < m$, we set t equal to the quotient obtained by dividing w by 2^{p-m} and set $i = h + p - m$.

5. Floating-Point Multiplication. Given the normalized floating-point numbers a and b as in (3.23), we calculate their product $c = t \cdot 2^i$ as follows. We multiply u and v to obtain w. If $w = 0$, we set $t = i = 0$; otherwise, we find the integer p satisfying (3.23) and set t equal to the quotient of w and 2^{p-m} and $i = j + k + p - m$.

6. Floating-Point Division. Given the normalized floating-point numbers a and b as in (3.23), we shall find the normalized floating-point number $c = t \cdot 2^i$ such that $c = a/b$. If $a = 0$, we set $t = i = 0$. If $b = 0$, we do not proceed with the division, but rather we give some indication - such as turning on an error indicator in the computer - that division by zero was attempted. Otherwise, we know that u and v are both non-zero and that their absolute values are greater than 2^{m-1} but less than or equal to 2^m, where m is the integer satisfying (3.22). Hence, we may divide u by v as prescribed in Theorem 3.2, taking the j in that theorem equal to $-m+1$. If w is the result of that division, we set $t = i = 0$ if $w = 0$, and if $w \neq 0$, we obtain from the table of powers of two an integer r such that

$$2^{r-1} < |w| \leqslant 2^r.$$

If $r \neq p$ - that is, $r < p$ - we set $t = 2w$ and $i = j - k - m$; if $r = p$, we set $t = w$ and $i = j - k - m + 1$.

* * *

These procedures show how the division algorithms of Theorems 3.1 and 3.2 can be combined with a suitable definition of "normalization" to provide modular arithmetic computers with the capability of performing floating-point arithmetic operations. Because of the "normalization" used, no overflow is possible in these operations except where a very large floating-point number is converted to fixed-point. Clearly, however, the floating-point arithmetic operations described above are somewhat slower and more complicated than their fixed-point counterparts, but the same can be said of the floating-point operations in conventional digital computers. At least, floating-point arithmetic operations are now possible in modular arithmetic computers, whereas, to the best of the author's knowledge, they had not even been attempted with previously known division methods. (To illustrate the workings of floating-point arithmetic operations in modular arithmetic computers, we shall use some of the above procedures to perform some floating-point operations in an illustrative computation near the end of the next chapter.)

This presentation of floating-point arithmetic concludes our discussion of division methods in residue number systems. Although we have been able to prove only

that the division algorithms given above in Theorems 3.1 and 3.2 decrease the error in approximating the quotient by a factor of two in each "iteration", we shall see in Chapter V that, in practice, these procedures usually converge much faster than that. In the meantime, we shall devote our attention to showing how these division procedures can be modified to approximate the square roots of integers in residue number systems.

CHAPTER IV

SQUARE ROOTS

A. Square Root Calculations in Digital Computers.

Since the arithmetic operations executable by digital computers are restricted to the "rational" operations, add, subtract, multiply, and divide, irrational quantities such as square roots must be approximated in these computers through the use of only rational operations. The most common method used for calculating an approximation to the square root of a positive number in digital computers is the Newton-Raphson iteration. For a positive number x, this iterative method yields a sequence of approximations y_i to the positive square root y of x as follows: from any approximation y_i to y, the next approximation y_{i+1} is calculated by

$$y_{i+1} = \tfrac{1}{2}(y_i + x/y_i). \qquad (4.1)$$

If the first approximation y_0 is any positive number, then it can be shown that the sequence y_i converges to

y and that the convergence, in general, is rather rapid. (See Hildebrand [12], pp. 447-448.)

In modular arithmetic computers, however, calculating square roots by the Newton-Raphson method presents some problems, primarily because at least one division is required in each "iteration." If we rewrite (4.1) in the equivalent form

$$y_{i+1} = \frac{(y_i^2 + x)}{2y_i}, \tag{4.2}$$

and if we use the algoritm of Theorem 3.1 to perform the division by $2y_i$, then we obtain only the nearest integer to y_{i+1}, which raises the unpleasant question of how the convergence of the Newton-Raphson method is affected by introducing these rather sizeable round-off errors. Furthermore, we must also worry about possible overflow in calculating the numerator of the fraction in (4.2), and if we try to avoid this overflow by using (4.1), then we must perform two divisions per iteration, which introduces possibly a greater error in y_{i+1} than that arising from the one "nearest integer" division needed in (4.2). Our other alternative is to minimize the round-off error by using the algorithm of Theorem 3.2 to perform the divisions in either

(4.1) or (4.2), but then we are forced to calculate the succeeding y_i's in floating-point arithmetic, which results in a rather excessive amount of computation just to approximate a square root in a modular arithmetic computer.

Thus, we are led to seek a method by which we can approximate square roots in modular arithmetic computers without using division. In the method we shall give below, we shall avoid division simply by modifying the division algorithm itself to yield a new algorithm by which we can calculate directly the successive approximations to a square root.

B. A Square Root Algorithm for Residue Number Systems.

Let x be any positive integer and let y be its positive square root. We shall now describe a procedure whereby we can obtain an approximation to y in the form $z \cdot 2^j$, where z is an integer in the residue number system and j is either zero or a negative integer. If $j = 0$, our procedure will parallel the division procedure of Theorem 3.1 and will yield the nearest integer to y, and if $j < 0$, our procedure will be more like the division procedure of Theorem 3.2 in that it will yield a closer approximation to y than the nearest integer.

As in both of the division procedures given in Theorems 3.1 and 3.2, we begin our square root approximation algorithm by using the stored table of powers of two to determine the non-negative integer p such that

$$2^{p-1} < x \leqslant 2^{p}. \tag{4.3}$$

From this definition of p it follows that

$$2^{(p-1)/2} < y \leqslant 2^{p/2} \tag{4.4}$$

so that, since one of the numbers $(p-1)/2$ and $p/2$ must be an integer, we pick $z_0 = 2^q$ as our first approximation to y, where

$$q = \left[\frac{p}{2}\right], \tag{4.5}$$

the largest integer not exceeding $p/2$. Following the pattern used to calculate the e_i's in Theorem 3.1, we now define g_0 by

$$g_0 = x - z_0^2.$$

From this definition and from the definition of z_0 it follows that

$$2^{p-1} - (2^{p/2})^2 < g_0 \leqslant 2^{p} - (2^{(p-1)/2})^2,$$

so that

$$|g_0| \leqslant 2^{p-1}.$$

Recalling now the definition of w_1 in Theorem 3.2, we replace our first approximation z_0 to y with $z_1 = 2^{q-j}$,

where j is an integer satisfying

$$0 \geqslant j \geqslant (p-n-1)/2. \tag{4.6}$$

(As before, we define n to be the positive integer such that

$$2^n \leqslant M/2 < 2^{n+1}, \tag{4.7}$$

where M is the product of the moduli in the residue number system.) Replacing also g_0 by g_1, where g_1 is defined by

$$g_1 = 2^{-2j}.x - z_1^2,$$

we have

$$g_1 = 2^{-2j}.|g_0| \leqslant 2^{p-1}2^{n-p+1} = 2^n \leqslant M/2$$

and

$$z_1 \leqslant 2^{p/2 - j} \leqslant 2^{(n+1)/2} \leqslant 2^n \leqslant M/2,$$

since n is clearly greater than one. Hence, both z_1 and g_1 are integers within the range of our residue number system.

If $g_1 = 0$, then $z_1 = x \cdot 2^{-2j}$ so that $z_1 \cdot 2^j$ is exactly equal to y, the square root of x. If $g_1 \neq 0$, we note that the differential dg_1 of g_1 is $-2z_1 \cdot dz_1$. Since we should correct g_1 by dg_1 to reduce g_1 to zero to obtain the desired approximation to y, we should have

$$dg_1 = -g_1$$

or equivalently,

$$dz_1 = g_1/2z_1,$$

which is essentially how y_2 is obtained from y_1 in the Newton-Raphson iteration. Here, however, we avoid the division by $2z_1$ by approximating z_1 with 2^{s-j}, where

$$s = \left\lceil \frac{p+1}{2} \right\rceil , \tag{4.8}$$

by approximating $|g_1|$ with 2^{t_1}, where t_1 is the non-negative integer satisfying

$$2^{t_1-1} < |g_1| \leqslant 2^{t_1},$$

and by approximating dz_1 with

$$2^{t_1}/2 \cdot 2^{s-j} = 2^{t_1+j-s-1}.$$

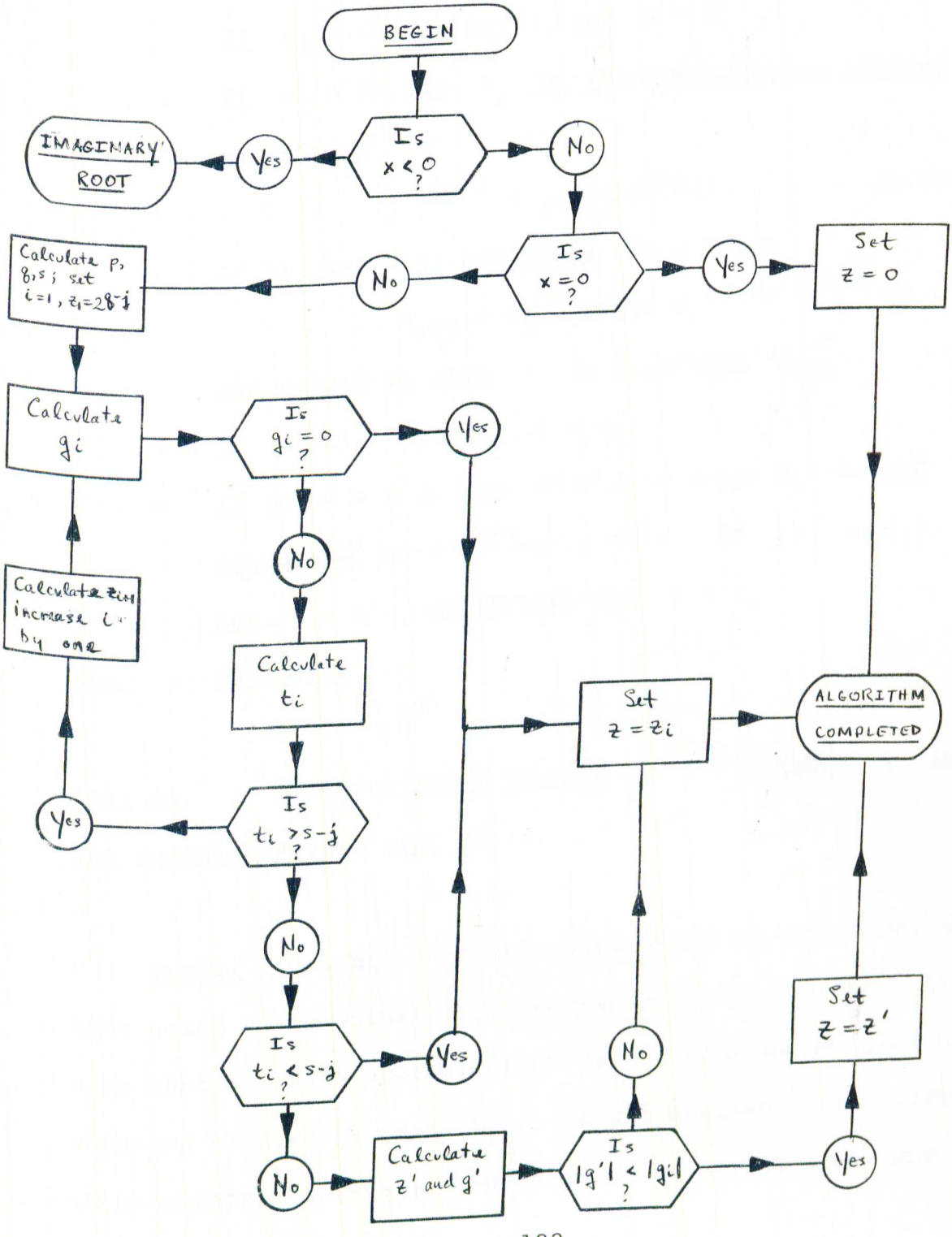
Thus, we define our second approximation z_2 to y by

$$z_2 = z_1 + (\text{sign } g_1) \cdot 2^{t_1+j-s-1}.$$

As for Theorems 3.1 and 3.2, repeating this reasoning for z_2, z_3, and so forth leads to an iterative procedure, a "flow chart" of which is given in Figure III. We now summarize this procedure in

Theorem 4.1 (Square Root Algorithm) - Let x be a positive integer in a residue number system in which all integers are between $-M/2$ and $+M/2$. Let p, q, j, n, and s be integers satisfying (4.3), (4.5), (4.6), (4.7), and (4.8), respectively. Let z be the integer defined as follows:

Figure III - Square Root Algorithm



1. Set $z_1 = 2^{q-j}$ and calculate g_1 by

$$g_i = 2^{-2j}.x - z_i^2. \tag{4.9}$$

If $g_i = 0$ for any i, set $z = z_i$.

2. If $g_i \neq 0$, let t_i be the non-negative integer satisfying

$$2^{t_i-1} < |g_i| \leq 2^{t_i}. \tag{4.10}$$

If $t_i > s - j$, calculate z_{i+1} from

$$z_{i+1} = z_i + (\text{sign } g_i)\cdot 2^{t_i+j-s-1} \tag{4.11}$$

and return to step 1 to calculate g_{i+1}.

3. If $t_i < s - j$, set $z = z_i$.

4. If $t_i = s - j$, set $z' = z_i + (\text{sign } g_i)\cdot 1$ and calculate $g' = 2^{-2j}.x - z'^2$. If $|g'| < |g_i|$, set $z = z'$; otherwise, set $z = z_i$.

Then z satisfies

$$|2^{-2j}.x - z^2| < 1/2;$$

that is, z is the nearest integer to $y\cdot 2^j$, where y is the positive square root of x.

Proof: The proof of this theorem is by induction, the idea being to show that t_i decreases as i increases or else that an exact approximation z_i to the square root is obtained (in which case t_{i+1} is not defined). In order to show this, it is also necessary to establish upper and

lower bounds on the integers z_i. In particular, our induction hypotheses are

$$t_i \leqslant t_{i-1} \leqslant p - 1 \tag{4.12}$$

and

$$2^{-j}2^{s-1} \leqslant z_i \leqslant 2^{-j}2^{s}, \tag{4.13}$$

where s is defined by (4.8). Since we have already shown in the remarks preceding the theorem that

$$|g_1| \leqslant 2^{p-1}$$

and since, by definition,

$$z_1 = 2^{-j}2^{s-1} \quad \text{or} \quad z_1 = 2^{-j}2^{s}$$

according as p is odd or even, it follows immediately that the induction hypotheses are satisfied when $i = 1$. Let us now assume that these hypotheses are satisfied when i is some integer $k \geqslant 1$ and let us show that this implies that (4.12) and (4.13) hold for $i = k+1$.

We begin by obtaining bounds on $|2^{-j}y - z_k|$. First, from (4.4), (4.8), and (4.13), we have

$$\begin{aligned} 2^{-j}2^{s} &= 2^{-j}2^{(p+1)/2} = 2^{-j}[2^{(p-1)/2} + 2^{(p-1)/2}] \\ &= 2^{-j}2^{(p-1)/2} + 2^{-j}2^{s-1} < 2^{-j}y + z_k \\ &\leqslant 2^{-j}2^{p/2} + 2^{-j}2^{s} = 2^{-j}[2^{p/2} + 2^{(p+1)/2}] \\ &< 2^{-j}2^{(p+3)/2} = 2^{-j}2^{s+1}, \end{aligned}$$

when p is odd; and similarly,

$$2^{-j}2^{s} = 2^{-j}2^{p/2} < 2^{-j}[2^{(p-1)/2} + 2^{(p-2)/2}]$$
$$= 2^{-j}2^{(p-1)/2} + 2^{-j}2^{s-1} < 2^{-j}y + z_k$$
$$\leqslant 2^{-j}2^{p/2} + 2^{-j}2^{s} = 2^{-j}[2^{p/2} + 2^{p/2}]$$
$$= 2^{-j}2^{(p+2)/2} = 2^{-j}2^{s+1},$$

when p is even. Hence, we have

$$2^{-j}2^{s} < 2^{-j}y + z_k \leqslant 2^{-j}2^{s+1}. \qquad (4.14)$$

From this, (4.9), and (4.10), it follows that

$$2^{t_k} > |2^{-2j}x - z_k^{2}| = (2^{-j}y + z_k)\cdot|2^{-j}y - z_k|$$
$$> 2^{-j}2^{s}\cdot|2^{-j}y - z_k|$$

and that

$$2^{t_k-1} < |2^{-2j}x - z_k^{2}| = (2^{-j}y + z_k)\cdot|2^{-j}y - z_k|$$
$$\leqslant 2^{-j}2^{s+1}\cdot|2^{-j}y - z_k|.$$

Hence, we have

$$2^{t_k+j-s-2} < |2^{-j}y - z_k| < 2^{t_k+j-s}. \qquad (4.15)$$

At this point it is necessary to split our induction proof into two cases. In the first of these cases - Case A - we shall assume p to be an odd integer, and in the second case - Case B - we shall assume p to be even. In Case A we shall show first that z_{k+1} is less than the upper bound given in (4.13) and then that t_{k+1} is less than t_k whenever z_{k+1} and z_k are on "opposite" sides of $2^{-j}y$. Next in Case A, we shall show that $t_{k+1} \leqslant t_k$

whenever z_{k+1} and z_k are on the same side of $2^{-j}y$, which will complete the proof of (4.12) for $i = k+1$. Finally, by establishing a slightly stronger result than (4.12) for $i < k$, we shall complete the induction for Case A by proving that z_{k+1} is greater than the lower bound in (4.13). In Case B, our proof will be essentially the same as in Case A, but because the different value of s in Case B yields different bounds on z_k in our induction hypothesis (4.13), it will be necessary to rearrange the steps of the proof somewhat. In particular, we shall show first in Case B that z_{k+1} is greater than the lower bound given in (4.13) and then we shall establish (4.12) for the situation in which $z_k > 2^{-j}y$. Next, we shall show that (4.12) holds for $i = k+1$ whenever z_k and z_{k+1} are both less than $2^{-j}y$ and then, by again establishing a slightly stronger result than (4.12) for $i < k$, we shall show that z_{k+1} is less than the upper bound given in (4.13). Finally, we shall complete the induction for Case B by showing that (4.12) also holds when $z_k < 2^{-j}y < z_{k+1}$. We proceed now with

Case A. If $z_k \cdot 2^j > y$, then $g_k < 0$ and

$$z_{k+1} = z_k - 2^{t_k+j-s-1} < z_k \leqslant 2^{-j}2^s$$

by (4.11) and the induction hypothesis (4.13). And if

$z_k \cdot 2^j < y$, then by (4.4) and (4.11) we have

$$\begin{aligned} z_{k+1} &= z_k + 2^{t_k+j-s-1} < 2^{-j}y + 2^{p-1+j-(p+3)/2} \\ &\leqslant 2^{-j}2^{p/2} + 2^{-2j}2^{j+(p-5)/2} \\ &= 2^{-j}2^{(p-5)/2}\cdot(2^{5/2} + 1) < 2^{-j}2^{(p-5)/2}\cdot(2^3) \\ &= 2^{-j}2^{(p+1)/2} = 2^{-j}2^s. \end{aligned} \tag{4.16}$$

Hence, $z_{k+1} < 2^{-j}2^s$, which shows that the upper bound in (4.13) holds for $i = k+1$.

Let us now consider what happens when $(2^{-j}y - z_k)$ and $(2^{-j}y - z_{k+1})$ have different signs - that is, when $z_k \cdot 2^j$ and $z_{k+1} \cdot 2^j$ are on opposite sides of y. It then follows from (4.11) that

$$2^{t_k+j-s-1} > |2^{-j}y - z_k|$$

so that, by (4.15), we have

$$\begin{aligned} |2^{-j}y - z_{k+1}| &= 2^{t_k+j-s-1} - |2^{-j}y - z_k| \\ &< 2^{t_k+j-s-1} - 2^{t_k+j-s-2} \\ &= 2^{t_k+j-s-2}. \end{aligned}$$

Hence, since $z_{k+1} < 2^{-j}2^s$, it follows that

$$\begin{aligned} |g_{k+1}| &= (2^{-j}y + z_{k+1})\cdot|2^{-j}y - z_{k+1}| \\ &< 2^{-j}2^{s+1}\cdot 2^{t_k+j-s-2} = 2^{t_k-1}. \end{aligned} \tag{4.17}$$

Therefore, if $z_k \cdot 2^j$ and $z_{k+1} \cdot 2^j$ are on opposite sides of y, then it follows that $t_{k+1} \leqslant t_k - 1$.

On the other hand, if $(2^{-j}y - z_k)$ and $(2^{-j}y - z_{k+1})$ have the same sign, then by (4.11) and (4.15) we have

$$\begin{aligned} \left|2^{-j}y - z_{k+1}\right| &= \left|2^{-j}y - z_k\right| - 2^{t_k+j-s-1} \\ &< 2^{t_k+j-s} - 2^{t_k+j-s-1} \\ &= 2^{t_k+j-s-1}. \end{aligned}$$

Hence, from this and from $z_{k+1} \leq 2^{-j}2^s$, it follows that

$$\begin{aligned} \left|g_{k+1}\right| &= (2^{-j}y + z_{k+1})\cdot\left|2^{-j}y - z_{k+1}\right| \\ &< 2^{-j}2^{s+1}\cdot 2^{t_k+j-s-1} = 2^{t_k}. \end{aligned} \tag{4.18}$$

It now follows that $t_{k+1} \leq t_k$ whenever $z_k\cdot 2^j$ and $z_{k+1}\cdot 2^j$ are on the same side of y. Coupling this with the result of the preceding paragraph completes the proof that (4.12) is satisfied for $i = k+1$.

Now all that remains to be shown for Case A is that the lower bound in (4.13) holds for $i = k+1$. In order to show this we first note that if we have $t_{i+1} = t_i$ for any $i < k$, then it follows from (4.11) that

$$z_{i+2} = z_i + (\text{sign } g_i)\cdot 2^{t_i+j-s}.$$

But then by (4.15) it follows that z_{i+2} and z_i (and z_{i+1}) are on opposite sides of $2^{-j}y$, so that by (4.17) we have $t_{i+2} \leq t_i - 1$. Hence, we have established a slightly stronger result than (4.12): namely, that for any positive integer $i < k$,

$$t_{i+2} \leqslant t_{i+1} \quad \text{and} \quad t_{i+2} \leqslant t_i - 1, \tag{4.19}$$

where $t_i \leqslant p - 1$. It is this stronger form of (4.12) that enables us to prove that $z_{k+1} \geqslant 2^{-j}2^{s-1}$ and thus complete the induction.

If $z_{k+1} > z_k$ - that is, $z_k \cdot 2^j < y$ - then by the induction hypothesis (4.13) we have

$$z_{k+1} > z_k \geqslant 2^{-j}2^{s-1}.$$

But if $z_{k+1} < z_k$, then it follows from (4.11) that z_k must have been greater than $2^{-j}y$. Since, by definition, z_1 is less than $2^{-j}y$, there must exist a largest integer $m < k$ such that $z_m < 2^{-j}y$. Then, by (4.17) we have

$$t_{m+1} \leqslant t_m - 1,$$

so that

$$\begin{aligned} z_{k+1} &= z_k - 2^{t_k+j-s-1} \\ &= z_m + 2^{t_m+j-s-1} - 2^{t_{m+1}+j-s-1} - \ldots \\ &\quad \ldots - 2^{t_k+j-s-1} \\ &\geqslant z_m + 2^{t_m+j-s-1} - 2^{t_m+j-s-1} = z_m. \end{aligned}$$

This follows from the fact that $t_{m+1}, t_{m+2}, \ldots, t_{k-1}$ must be a strictly decreasing sequence, since otherwise, if two of these t_i's were equal, some z_h would be less than $2^{-j}y$, by (4.19), for $m < h < k$. Since this would contradict the definition of m, it follows immediately from

the induction hypothesis applied to z_m that

$$z_{k+1} \geqslant z_m \geqslant 2^{-j}2^{s-1}.$$

This completes the proof of (4.13) for $i = k+1$ and ends our consideration of Case A.

Case B. If $z_k \cdot 2^j < y$, then

$$z_{k+1} > z_k \geqslant 2^{-j}2^{s-1}$$

by the induction hypothesis (4.13). And if $z_k \cdot 2^j > y$, then

$$\begin{aligned} z_{k+1} &= z_k - 2^{t_k+j-(p+2)/2} \\ &> 2^{-j}y - 2^{-2j}2^{t_k+j-(p+2)/2} \\ &> 2^{-j}[2^{(p-1)/2} - 2^{(p-4)/2}] \\ &> 2^{-j}2^{(p-2)/2} = 2^{-j}2^{s-1}. \end{aligned}$$

Hence, the lower bound in (4.13) holds for $i = k+1$.

Let us now assume momentarily that $z_k \cdot 2^j > y$, so that $z_{k+1} < z_k \leqslant 2^{-j}2^s$ by the induction hypothesis (4.13). Then if $z_{k+1} \cdot 2^j$ is also greater than y, it follows from (4.11) and (4.15) that

$$\begin{aligned} \left|2^{-j}y - z_{k+1}\right| &= \left|2^{-j}y - z_k\right| - 2^{t_k+j-s-1} \\ &< 2^{t_k+j-s} - 2^{t_k+j-s-1} \\ &= 2^{t_k+j-s-1}. \end{aligned}$$

Hence, we have

$$\begin{aligned} |g_{k+1}| &= (2^{-j}y + z_{k+1}) \cdot |2^{-j}y - z_{k+1}| \\ &< 2^{-j} 2^{s+1} \cdot 2^{t_k+j-s-1} \\ &= 2^{t_k}. \end{aligned} \tag{4.20}$$

Therefore, if $z_k > z_{k+1} > 2^{-j}y$, then $t_{k+1} \leqslant t_k$.

Similarly, if $z_k \cdot 2^j > y > z_{k+1} \cdot 2^j$, then it follows from (4.11) and (4.15) that

$$2^{t_k+j-s-1} > |2^{-j}y - z_k|$$

and that

$$\begin{aligned} |2^{-j}y - z_{k+1}| &= 2^{t_k+j-s-1} - |2^{-j}y - z_k| \\ &< 2^{t_k+j-s-1} - 2^{t_k+j-s-2} \\ &= 2^{t_k+j-s-2}. \end{aligned}$$

From this we have

$$\begin{aligned} |g_{k+1}| &= (2^{-j}y + z_{k+1}) \cdot |2^{-j}y - z_{k+1}| \\ &< 2^{-j} 2^{s+1} \cdot 2^{t_k+j-s-2} \\ &= 2^{t_k-1}. \end{aligned} \tag{4.21}$$

Hence, if $z_k > 2^{-j}y > z_{k+1}$, then $t_{k+1} \leqslant t_k - 1$.

Now let us assume that z_k is less than $2^{-j}y$. Then, if $z_k < z_{k+1} < 2^{-j}y$, we have

$$\begin{aligned} |2^{-j}y - z_{k+1}| &= |2^{-j}y - z_k| - 2^{t_k+j-s-1} \\ &< 2^{t_k+j-s-1} \end{aligned}$$

and

$$|g_{k+1}| = (2^{-j}y + z_{k+1}) \cdot |2^{-j}y - z_{k+1}|$$
$$< 2^{-j} 2^{s+1} \cdot 2^{t_k+j-s-1}$$
$$= 2^{t_k}.$$

By combining this result with (4.20), it follows that $t_{k+1} \leqslant t_k$ whenever z_k and z_{k+1} are on the same side of $2^{-j}y$. However, as before in Case A, if this happens and if $t_{i+1} = t_i$ for some integer $i < k$, then it follows from (4.11) that

$$z_{i+2} = z_i + (\text{sign } g_i) \cdot 2^{t_k+j-s}.$$

Hence, by (4.15), it follows that z_{i+2} and z_i are on opposite sides of $2^{-j}y$ so that (4.19) also holds for Case B whenever $z_i > 2^{-j}y$ and $i < k$.

Now if $z_k < 2^{-j}y < z_{k+1}$, let m be the largest integer such that $z_m > 2^{-j}y$. (We know that such an m exists since $z_1 > 2^{-j}y$ by definition.) Then, by (4.21) we have

$$t_{m+1} \leqslant t_m - 1$$

so that

$$z_{k+1} = z_k + 2^{t_k+j-s-1}$$
$$= z_m - 2^{t_m+j-s-1} + 2^{t_{m+1}+j-s-1} + \ldots$$
$$\ldots + 2^{t_k+j-s-1}. \qquad (4.22)$$

Since $t_{m+1}, t_{m+2}, \ldots, t_{k-1}$ must be a strictly decreasing

sequence - otherwise for some h such that $m < h < k$ we would have $z_h > 2^{-j}y$, contradicting the definition of m - it follows from (4.22) that $z_{k+1} \leqslant z_m$, which gives

$$z_{k+1} \leqslant 2^{-j}2^s$$

by the induction hypothesis (4.13) applied to z_m. Thus, as in (4.21) we have

$$\left|2^{-j}y - z_{k+1}\right| < 2^{t_k+j-s-2}$$

and

$$\left|g_{k+1}\right| < 2^{-j}2^{s+1}.2^{t_k+j-s-2} = 2^{t_k-1}.$$

Therefore, $t_{k+1} \leqslant t_k - 1$ whenever z_k and z_{k+1} are on opposite sides of $2^{-j}y$. This proves (4.12) - and also (4.19) - for Case B.

Finally, since z_{k+1} is clearly less than $2^{-j}2^s$ whenever $z_{k+1} < z_k$ or whenever $z_k < z_{k+1} < 2^{-j}y$, and since we have already shown in (4.22) that $z_{k+1} \leqslant 2^{-j}2^s$ whenever $z_k < 2^{-j}y < z_{k+1}$, it follows that the upper bound in (4.13) holds for $i = k+1$, which completes the proof for Case B.

By the induction principle it follows that (4.12) and (4.13) are satisfied for every positive integer i. Since we proved the result (4.19) in both Cases A and B, we have actually shown that the z_i's converge to y with a

decrease in the error by at least a factor of two for every two "iterations." Finally, if $t_i < s-j$, then it follows from (4.15) that

$$\left|2^{-j} y - z_i\right| < 2^{t_i+j-s} \leq 2^{-1} = 1/2.$$

But if $t_i = s-j$, then we have

$$\left|2^{-j} y - z_i\right| < 1,$$

so that z_i differs from the desired result z by at most 1. The final comparison between g_i and g' as prescribed in step 4 of the theorem assures us that whichever of z_i and z' is closer to y is the final value assigned to z. Note that since we are approximating the square root of an integer, the difference between z and $2^{-j} y$ will always be strictly less than $1/2$, and hence $z \cdot 2^j$ will differ from y by less than 2^{j-1}. This completes the proof of Theorem 4.1.

* * *

As in the division algorithm of Theorem 3.2, the exponent j in Theorem 4.1 may be decreased in successive "iterations." In particular, since $t_{i+1} \leq t_i$ for every i and since a decrease of one in j causes an increase in g_{i+1} by a factor of four, j may be decreased in the $(i+1)$st iteration by as much as

$$w_i = \left[\frac{n - t_i}{2}\right]. \tag{4.23}$$

However, since $z_i \leqslant 2^{s-j}$ by (4.13), it follows that

$$z_i \leqslant 2^n \leqslant M/2 \quad -$$

that is, z_i is within the computer range - whenever j is not less than

$$j_{min} = s - n. \tag{4.24}$$

Thus, we set $j = 0$ in Theorem 4.1 to obtain the nearest integer to the positive square square root of a positive integer or we may make j negative and obtain a more accurate approximation to the root. If we decrease j to j_{min}, we obtain the most accurate approximation to the root that is consistent with keeping the integer z in Theorem 4.1 within the range of the residue number system. And regardless of the j we use - as long as it satisfies $0 \geqslant j \geqslant j_{min}$ - Theorem 4.1 guarantees that the error in the final approximation to the root is less than 2^{j-1}.

As an example of using this procedure to obtain an approximation to a square root of an integer, let us now calculate the most accurate approximation $z \cdot 2^j$ to the positive square root Y of $x = 627,323$ consistent with keeping z within the range of the residue number system whose moduli are 2, 3, 5, 7, 11, 13, 17, and 19. From

(4.3) and (4.7), respectively, we have

$$p = 20 \quad \text{and} \quad n = 22,$$

and by (4.5), (4.6), (4.8), and (4.24), respectively, we have

$$p = 10, \quad j = -1, \quad s = 10, \quad \text{and} \quad j_{min} = -12.$$

Then, our first approximation to $2^{-j}y$ is

$$z_1 = 2^{q-j} = 2^{11} = 2048$$

and, calculating g_1 from (4.9), we have

$$g_1 = 2^2 \cdot 627,323 - (2048)^2 = -1,685,012.$$

From this and (4.10), (4.23), and (4.11), we obtain

$$t_1 = 21, \quad w_1 = 0, \quad \text{and} \quad z_2 = 2048 - 2^9 = 1536.$$

The results of the calculations for the remaining iterations are given in Table V.

Note that in the fourth iteration of this calculation, we have

$$t_4 = 14 = s - j.$$

However, since

$$j = -4 > -12 = j_{min}$$

at this point, we do not proceed with step 4 of Theorem 4.1. For, when we decrease j by $w_4 = 4$, we find that $2^{w_2 t_4 + j - s - 1}$ is indeed an integer, so that we may continue with a fifth iteration. Moreover, again in the seventh

Table V - Application of Theorem 4.1: Sample Square Root Calculation

$x = 627{,}323$ $\qquad$ $y = 792.0372465...$

i	z_i	j	g_i	t_i	$z_i \cdot 2^j$	$(z_i \cdot 2^j)^2$
1	2,048	-1	-1,685,012	21	1024.0000000	1,048,576.0000
2	1,536	-1	149,996	18	768.0000000	589,824.0000
3	6,400	-3	-811,328	20	800.0000000	640,000.0000
4	12,672	-4	15,104	14	792.0000000	627,264.0000
5	202,760	-8	622,528	20	792.0312500	627,313.5009...
6	405,524	-9	-754,064	20	792.0390625	627,325.8765...
7	811,046	-10	227,932	18	792.0371093...	627,322.7826...
	3,244,185	-12	-2,841,157	--	792.0373535...	627,323.1693...

$2^{j-1} = 2^{-13} = 0.0001221...$ $\qquad$ Error = 0.0001070...

iteration we have

$$t_7 = 18 < 20 = s = j,$$

but since

$$t_7 + w_7 = s - j$$

we proceed as in step 4 of Theorem 4.1 to calculate $z' = 3,244,185$. Since

$$|g'| = 2,841,157 < 3,646,912 = 2^{2w_7}|g_7|,$$

we set $z = z'$ to obtain the final approximation

$$z \cdot 2^j = 3,244,185 \cdot 2^{-12} = 792.0373535\ldots$$

which differs from $y = 792.0372465\ldots$ by less than

$$2^{j-1} = 2^{-13} = 0.0001221\ldots.$$

Thus, when j is being decreased from iteration to iteration, the comparisons made between t_i and $s - j$ in steps 2, 3, and 4 of Theorem 4.1 should be replaced by comparisons between $t_i + w'$ and $s - j$, where w' is the value of j in the ith iteration minus the value of j desired in the final result.

As the above example suggests, the square root algorithm of Theorem 4.1 often yields successive approximations converging to the square root y at a rate considerably faster than the minumum of one binary "bit" of accuracy per two iterations established in the proof. And as we

shall see in the next chapter, this algorithm does indeed produce in practice a sequence of approximations converging at a rate several times faster than is predicted in the proof. Moreover, it is interesting to note that an estimate of the number of operations necessary in using floating-point arithmetic in conjunction with the Newton-Raphson method in a modular arithmetic computer to calculate an approximation to the square root of 627,323 with the same accuracy as obtained in the above example indicates that the Newton-Raphson method requires nearly three times as much computational effort as the above algorithm.

C. Floating-Point Operations in a Residue Number System.

Clearly, the algorithm of Theorem 4.1 can be utilized to calculate approximations to the positive square root of a positive number given in floating-point form in a modular arithmetic computer. In particular, if the positive number $x = u \cdot 2^k$ is given in the normalized floating-point form specified in the preceding chapter (pp. 108-109), then an approximation to the square root of x may be calculated as follows: If k is odd, set $v = 2u$ and $h = (k-1)/2$; if k is even, set $v = u$ and $h = k/2$. Next, calculate an approximation z to the square root of v, using Theorem 4.1 and $j = \lceil m/2 \rceil$,

where m is the integer satisfying (3.22). Then, $y = z \cdot 2^{i}$, where $i = j + h$, is the desired floating-point approximation to the positive square root of x.

To illustrate the use of this procedure in approximating a square root of a number in floating-point form as well the use of some of the other floating-point operations described in the preceding chapter (pp. 109-113), let us now show how these operations can be used to calculate the greater root of

$$x^2 - 5x - 7 = 0$$

in a residue number system whose moduli are 2, 3, 5, 7, 11, 13, 17, and 19. From (3.22) we have $m = 10$, so that the normalized floating-point representations of the coefficients in the above equation are

$$a = 1 = 1024 \cdot 2^{-10}, \qquad b = -5 = -640 \cdot 2^{-7},$$

$$\text{and} \quad c = -7 = -896 \cdot 2^{-7}.$$

Using floating-point arithmetic operations to evaluate

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a},$$

we first calculate $2a$. Since the constant "two" in normalized floating-point form is $1024 \cdot 2^{-9}$, we multiply $1024 \cdot 2^{-9}$ by $1024 \cdot 2^{-10}$ as outlined in the description of floating-point multiplication in the preceding chapter (p.112). We obtain $1{,}048{,}576 \cdot 2^{-17}$ which, when normalized, gives

$$2a = 1024 \cdot 2^{-9}.$$

In a similar manner we calculate

$$2c = -917{,}504 \cdot 2^{-16} = -896 \cdot 2^{-6},$$

$$(2a) \cdot (2c) = 4ac = 917{,}504 \cdot 2^{-14} = -896 \cdot 2^{-5}$$

and

$$b^2 = 409{,}600 \cdot 2^{-14} = 800 \cdot 2^{-5}.$$

Next, we subtract $4ac$ from b^2 as outlined in the preceding chapter under floating-point addition and subtraction (pp. 111-112) and obtain

$$b^2 - 4ac = 1696 \cdot 2^{-5} = 848 \cdot 2^{-4}.$$

Approximating the square root of $b^2 - 4ac$ as outlined above gives

$$\sqrt{b^2 - 4ac} = 932 \cdot 2^{-7}.$$

Subtracting b from this, we obtain

$$-b + \sqrt{b^2 - 4ac} = 1572 \cdot 2^{-7} = 786 \cdot 2^{-6}.$$

Finally, dividing this result by $2a$ as outlined in the preceding chapter (p. 113), we obtain

$$x = 786 \cdot 2^{-7} = 6.140625,$$

which is our computed approximation to the solution

$$x = \frac{5 + \sqrt{53}}{2} = 6.140055\ldots.$$

Thus, we have shown how modular arithmetic computers can be used to add, subtract, multiply, divide, and approximate square roots in either fixed-point or floating-point arithmetic.

While the square root approximation procedure given in Theorem 4.1 above is somewhat complicated and while it may converge rather slowly in some instances, the results of extensive trial calculations using this procedure indicate that it converges sufficiently rapidly to be more efficient - on the average - than using floating-point arithmetic and the Newton-Raphson method and more accurate than using fixed-point operations with the Newton-Raphson method in modular arithmetic computers. To examine in more detail the practical behavior of this square root algorithm, let us now turn our attention to the results of those trial calculations.

CHAPTER V

COMPUTER SIMULATION

A. Simulation Programs. In order to obtain a better idea about how the division and square root procedures of Theorems 3.2 and 4.1, respectively, might behave in practice, two simulation programs were written to perform those procedures on the IBM 7090 computer. Under the control of the input data, these programs perform "typical" divisions and square root approximations, record the amount of computation required for each, and check the accuracy of each approximation obtained. Through the use of these programs, it is possible to compute several thousand quotients and square roots in a rather short time, so that detailed information about the practical behavior of the division and square root procedures can be obtained without resorting to hours of laborious hand calculations. For simplicity in programming, most of the calculations in both the division and the square root simulation programs are performed in

normal 7090 (floating-point) binary arithmetic. The simulation programs use residue arithmetic only when the error estimates f_i and g_i defined respectively by (3.16) and (4.11) are to be calculated, since the special truncation and overflow properties of residue number systems are necessary to obtain the correct values for these quantities. To calculate f_i and g_i in the specified residue number systems, the simulation programs use special subroutines. Other subroutines are also used to simulate the use of the stored table of powers of two.

The division simulation program, written partly in FORTRAN II and partly in FAP, accepts as input data the moduli to be used and the number of divisions to be performed. For each division, it obtains a dividend and divisor by using random digits from a "random number" generating subroutine to give the number of digits in the dividend, the number of digits in the divisor, then the dividend itself, and finally the divisor itself. (Since the division procedure behaves no differently for positive or negative numbers, only positive dividends and divisors are used.) Using

Table VI - Division Simulation Program Results

Moduli are 7 11 13 23 29 31
M = 20,697,677 M/2 = 10,348,838 n = 23

Dividend	Divisor	Quotient	j		Iterations	Binary bits of accuracy	Bits of accuracy per iteration
2,993,174	625,186	5,020,219	-20 =	4.78765392	10	24	2.400
502,614	6,759	4,873,400	-16 =	74.36218262	10	26	2.600
5	40,290	4,263,381	-35 =	0.00012407	8	37	4.625
2	92	5,934,180	-28 =	0.02210654	11	30	2.727
4	855,024	5,157,832	-40 =	0.00000469	9	41	4.556
7,786,191	3	5,190,794	- 1 =	2,595,397.	9	Exact	
835,659	171	5,004,180	-10 =	4,886.89453125	9	24	2.667
846	20,079	5,655,072	-27 =	0.04213357	8	28	3.500
675,797	59	5,864,543	- 9 =	11,454.18554687	9	24	2,667
49,722	13	7,833,127	-11 =	3,824.76904297	7	24	3.429
48,827	424	7,546,996	-16 =	115.15802002	10	26	2.600
6	25	8,053,064	-25 =	0.24000001	7	26	3.714
9,176	13	5,782,292	-13 =	705.84619141	9	24	2.667
93,846	69,368	5,674,355	-22 =	1.35287166	12	24	2.000
72,946	7,091,970	5,522,101	-29 =	0.01028571	13	31	2.385

Average number of bits per iteration = 2.836 - 1 exact solution.

the "randomly generated" dividend and divisor, the program next begins the division procedure described in Theorem 3.2 and "iterates" with that procedure until it obtains the most accurate approximation to the quotient consistent with the range of the residue number system being used. Finally, the program checks the accuracy of the approximation obtained and the number of "iterations" which were required to attain it. After printing out the dividend, the divisor, the approximation obtained, and the information about the iterations required and the accuracy obtained, the division program returns to the "random number" subroutine to calculate the dividend and the divisor for the next division. Table VI contains a sample of the output generated by this simulation program.

The square root simulation program operates in much the same way as the division program, except that the "random number" generator is not used. Instead, the program reads from punched cards the smallest and largest positive numbers whose square roots are to be calculated and the increment to be used in obtaining other numbers which are between the smallest and largest and whose square roots are also to be calculated. For

Table VII - Square Root Simulation Program Results

Moduli are 8 25 27 29 37 47
M = 272,327,400 M/2 = 136,163,700 n = 27

Number	Root	j		$Root^2$	Iterations	Binary bits of accuracy	Bits of accuracy per iteration
136,164	96,732,230	-18	= 369.00417328	136,164.078	11	28	2.545
340,751	76,511,857	-17	= 583.73914337	340,751.387	11	34	3.091
545,338	96,792,703	-17	= 738.46971893	545,337.523	12	29	2.417
749,925	113,505,966	-17	= 865.98179626	749,924.469	11	28	2.545
954,512	128,056,202	-17	= 976.99128723	954,511.969	13	28	2.154
1,159,099	70,557,007	-16	= 1076.61448669	1,159,098.750	13	28	2.154
1,363,686	76,530,948	-16	= 1167.76959228	1,363,685.812	8	29	3.625
1,568,273	82,071,200	-16	= 1252.30712891	1,568,273.141	14	28	2.000
1,772,860	87,260,388	-16	= 1331.48785400	1,772,859.891	13	28	2.154
1,977,447	92,157,855	-16	= 1406.21726990	1,977,447.000	9	29	3.222
2,182,034	96,807,874	-16	= 1477.17092895	2,182,033.937	11	29	2.636
2,386,621	101,244,557	-16	= 1544.86933899	2,386,621.250	10	28	2.800
2,591,208	105,494,789	-16	= 1609.72273254	2,591,207.250	12	29	2.417
2,795,795	109,580,335	-16	= 1672.06321716	2,795,795.375	9	27	3.000
3,000,382	113,518,906	-16	= 1732.16104126	3,000,381.844	13	28	2.154

Average number of bits per iteration = 2.594 - 0 exact solutions.

each of these numbers, the program approximates the positive square root by the procedure described above in Theorem 4.1, "iterating" over and over until the best approximation to the square root consistent with the range of the number system is obtained. After checking the accuracy of the approximation and the number of "iterations" required, the program prints out the number whose square root was approximated, the approximation itself, the square of the approximation (for comparison with the original number whose root was calculated), and the information about the accuracy obtained and the iterations required. Finally the program adds the aforementioned increment to the number whose root was just approximated and obtains the next number whose square root it is to calculate. Table VII contains a sample of the output generated by this simulation program.

B. Simulation Results. The simulation programs were run on the IBM 7090 computer at the University of Maryland's Computer Science Center. Eleven different sets of moduli, ranging from 2, 3, 5, and 7 to 8, 25, 27, 29, 37, and 47, were tried to determine whether or not changing the residue number system - that is,

the computer range - has any effect on the behavior of the division and square root procedures. In general, changing the moduli produced no noticeable effect, at least in the average rates of convergence for the two procedures. The accuracy of the approximations increased as the computer range increased; but then, so did the number of iterations.

In all, over 6400 divisions and 6400 square roots were calculated by the simulation programs. The total computing time was 35-40 minutes. For the divisions, from 1 to 25 iterations were required for each approximation, while for the square roots the number of iterations ranged from 2 to 21. The accuracy attained in the approximations was, in general, higher for the divisions than for the square roots. For example, slightly over 10% of the division approximations were exactly equal to the quotient, while only 0.33% of the square root approximations were exact. Also the accuracy of the "non-exact" approximations was greater for the divisions than for the square roots, the approximations being often as great as 20 binary bits "more accurate" than predicted

in Theorem 3.2 for division while seldom more than 6 or 7 bits more accurate than predicted in Theorem 4.1 for square roots.

The most significant results obtained from the simulation programs were that the division and square root procedures converge, on the average, considerably more rapidly than is suggested by the proofs of Theorems 3.2 and 4.1, respectively. In particular, for all of the more than 6400 divisions performed, the average rate of convergence for the division procedure was 3.021 binary bits of accuracy per iteration, and for about the same number of square roots, the square root procedure converged at an average rate of 2.617 bits per iteration. For the division program, the rate of convergence obtained in the simulation runs ranged from as low as the minimal 1 binary bit of accuracy per iteration predicted in the proof of Theorem 3.2 to as high as 8.2 bits per iteration. In the square root simulation, the rate of convergence was as low as 1.2 binary bits per iteration and as high as 12 bits.

Clearly, these results from the simulation programs emphasize the practical value of the division and square root procedures developed in Theorems 3.2 and 4.1. Not only do these procedures converge considerably more rapidly in practice than is proved in the above theorems, but also the computational effort they require to obtain the approximations is considerably less than for any other division or square root procedure yet devised for residue number systems.

CONCLUSION

In this thesis we have treated four problems: how to compare the magnitudes of two numbers, how to detect additive and multiplicative overflow, how to divide, and how to approximate square roots in residue number systems. In Chapter I, we showed how the ordinary positional notation for integers can be extended to a *mixed-radix notation* which can then be used to determine the larger and smaller of two numbers in a residue number system. In Chapter II, we used this comparison technique to help determine whether or not overflow occurs in addition, subtraction, and multiplication in a residue number system. We gave simple necessary and sufficient conditions for additive overflow and we presented two methods for detecting multiplicative overflow. For the latter multiplicative overflow detection procedure, we introduced the use of a table of powers of two, which we then also used in Chapters III and IV to implement respectively a division algorithm and a square root algorithm for residue number systems. In Chapter III, we showed how the division algorithm can be used to provide

approximations to a quotient ranging from the nearest integer to the most accurate approximation possible for the residue number system being used. We also showed how the division algorithm may be applied to provide modular arithmetic computers with the capability for performing floating-point arithmetic operations. In Chapter IV, we presented an algorithm in which division can be avoided while approximating the square root of a number in a residue number system, and we showed how this algorithm can be used to obtain an approximation to the square root with any degree of accuracy from the nearest integer to most accurate approximation possible for the residue number system used.

In each instance, we have provided examples illustrating how the procedures given are used in actual computations and we have explained how the necessary computations for these procedures can be performed conveniently in a modular arithmetic computer. Finally, in Chapter V, we described how a conventional digital computer was programmed to simulate the use of the division and square root algorithms in a modular arithmetic computer in performing trial calculations. From the sample calculations performed by the simulation programs, we found that the convergence of these methods is considerably faster in practice than was indicated by the proofs of

the pertinent theorems in Chapters III and IV. Thus, we have not only presented solutions to the four problems we considered, but we have also shown that these solutions are workable in practical applications.

At this point it might be well to ask what problems related to the use of residue number systems in digital computers have we not solved. In addition to the many problems related to the electronic engineering and design of modular arithmetic computers, there are still numerous open "theoretical" questions, of which we shall mention just a few. First, we have not considered in the preceding chapters whether or not a table of powers of three or four or some other positive integer can be substituted for the table of powers of two which we used in the multiplicative overflow detection, the division, and the square root procedures. Because of the reliance on the specific properties of the powers of two at various critical points in the proofs related to these procedures, it is the author's opinion that using a table of powers of an integer greater than two would complicate considerably any extensions of the procedures given. Nevertheless, such extensions, or entirely different methods, are no doubt possible and would probably converge

faster than the methods given above for division and square root approximation. Next, it might be inquired whether or not the square root algorithm given in Chapter IV can be extended to provide approximations to real roots of degree higher than two or, more generally, whether the algorithm can be extended to approximate real roots of polynomial equations. Such extensions, or methods entirely different from that given above, are obviously quite desirable, but in view of the complexity of the proof of Theorem 4.1, the author feels that finding them would be rather difficult. Finally, instead of trying to force residue number systems to perform calculations for problems based in the real or rational number systems, it might be asked whether or not there exist problems - in particular, in number theory - which can be stated directly in terms of residues and congruences and for which a digital computer using a residue number system would be better suited than conventional digital computers. If such problems exist, the author is presently unaware of them, but he feels that learning a sufficient amount of number theory to carry out such an inquiry should be rewarding enough to make the whole effort worthwhile.

BIBLIOGRAPHY

Since much of the work on the application of residue number systems to digital computers has been published in rather obscure journals and reports, the author has attempted to include in this bibliography all references known to him and pertaining to the use of residue number systems in digital computers. To those interested in examining some of these references, the following information may be of assistance.

References [1], [11], [13], [17], [19], and [32] below are United States Air Force Technical Reports which were submitted under contracts with the Electronic Technology Laboratory, Aeronautical Systems Division, United States Air Force, Wright-Patterson Air Force Base, Ohio. Qualified requesters may obtain copies of these reports from the Defense Documentation Center, Cameron Station, Alexandria, Virginia 22314. These reports are not, in general, part of the "open" literature.

The journal, Stroje Na Zpracování Informací, referred to in References [23], [24], [28], [30], and [31] below, is published by the Laboratoř Matematičkých Strojů, Československá Akademie Věd (Laboratory of Mathematical Machines, Czechoslovakian Academy of Sciences), Prague, Czechoslovakia. The title means "Machines for Processing Information." Sborník I-VIII (Volumes 1-8) of this journal are available at the Library of Congress under call number QA76.S84.

— — —

[1] Aiken, H.H. and Semon, W. Advanced Digital Computer Logic. ASD Technical Report No. 59-472. Wright-Patterson Air Force Base, Ohio: Aeronautical Systems Division, United States Air Force, 1959.

[2] Cheney, P.W. "A Digital Correlator Based On the Residue Number System," IRE Transactions on Electronic Computers, EC-10 (March, 1961), pp. 63-70.

[3] Dickson, L.E. History of the Theory of Numbers, Vol. II. New York: Chelsea Publishing Company, 1952.

[4] Eastman, W.L. "Sign Determination in a Modular Number System," Proceedings of a Harvard Symposium on Digital Computers and Their Applications, 1961, pp. 136-162. Cambridge, Massachusetts: Harvard University Press, 1962.

[5] Garner, H.L. "Error Checking and the Structure of Binary Addition." Ph.D. Dissertation. Ann Arbor, Michigan: University of Michigan, 1958.

[6] _____. "The Residue Number System," Proceedings of the Western Joint Computer Conference, 1959, pp. 143-153.

[7] _____. "The Residue Number System," IRE Transactions on Electronic Computers, EC-8 (June, 1959), pp. 140-147.

[8] Griffin, H. Elementary Theory of Numbers. New York: McGraw-Hill Book Company, 1954.

[9] Guffin, R.M. "A Computer For Solving Linear Simultaneous Equations Using the Residue Number System," IRE Transactions on Electronic Computers, EC-11 (April, 1962), pp. 164-173.

[10] Hardy, G.H. and Wright, E.M. An Introduction to the Theory of Numbers, 4th ed. London: Oxford University Press, 1960.

[11] Harvard Computation Laboratory, Harvard University. Notes on Modular Number Systems. ASD Technical Report No. 61-12. Wright-Patterson Air Force Base, Ohio: Aeronautical Systems Division, United States Air Force, 1961.

[12] Hildebrand, F.B. Introduction to Numerical Analysis. New York: McGraw-Hill Book Company, 1956.

[13] Information Systems Laboratory, University of Michigan. Residue Number Systems for Computers. ASD Technical Report No. 61-483. Wright-Patterson Air Force Base, Ohio: Aeronautical Systems Division, United States Air Force, 1961.

[14] Jacobson, N. Lectures in Abstract Algebra, Vol. I. Princeton, New Jersey: D. Van Nostrand Company, 1951.

[15] Keir, Y.A., Cheney, P.W., and Tannenbaum, M. "Division and Overflow Detection in Residue Number Systems," IRE Transactions on Electronic Computers, EC-11 (August, 1962), pp. 501-507.

[16] Lindamood, G.E. and Shapiro, G. "Magnitude Comparison and Overflow Detection in Modular Arithmetic Computers," SIAM Review, V (October, 1963), pp. 342-350.

[17] Lockheed Missiles and Space Company. Modular Arithmetic Techniques. ASD Technical Report No. 62-686. Wright-Patterson Air Force Base, Ohio: Aeronautical Systems Division, United States Air Force, 1962.

[18] Rozenberg, D.P. "An Investigation of the Algebraic Properties of the Residue Number System." Ph.D. Dissertation. Ann Arbor, Michigan: University of Michigan, 1961.

[19] Scope, Incorporated. Computer Applications of Residue Class Notations. ASD Technical Report No. 61-189. Wright-Patterson Air Force Base, Ohio: Aeronautical Systems Division, United States Air Force, 1961.

[20] Shapiro, G. "Gauss Elimination for Singular Matrices," Mathematics of Computation, XVII (October, 1963), pp. 441-445.

[21] Shapiro, H.S. "Some Notes on Modular Arithmetic and Parallel Computation," Mathematics of Computation, XVI (April, 1962), pp. 218-222.

[22] _____. Private communication. May, 1962.

[23] Svoboda, A. "Application of the Korobov Sequence in Mathematical Machines," Stroje Na Zpracování Informací, Sborník III (1955), pp. 61-76 (1956).

[24] _____. "The Rational Number System of Residual Classes," Stroje Na Zpracování Informací, Sborník V (1957), pp. 9-37.

[25] _____. "The Numerical System of Residual Classes in Mathematical Machines," Proceedings: Congreso Internacionale de Automática, Madrid, 13-18 October 1958, pp. 388-397. Madrid: Instituto de Electricidad y Automática, Consejo Superior de Investigaciones Científicas, 1961.

[26] _____. "The Numerical System of Residual Classes in Mathematical Machines," Information Processing: Proceedings of the International Conference on Information Processing, Paris, 15-20 April, 1959, pp. 419-422. Paris: UNESCO, 1960.

[27] Szabo, N. "Sign Detection in Nonredundant Residue Number Systems," IRE Transactions on Electronic Computers, EC-11 (August, 1962), pp. 494-500.

[28] Valach, M. "Vznik Kodu A Číselné Soustavy Zbytkových Tríd," Stroje Na Zpracování Informací, Sborník III (1955), pp. 211-245 (1956).

[29] _____. "Abbildung der Zahlen und der Arithmetischen Operationen in Restklassen," Aktuelle Probleme im Rechnentechnik: Bericht über das Internationale Mathematiker-Kolloquium, Dresden, 22-27 November 1955, pp. 57-59. Berlin: VEB Deutscher Verlag der Wissenschaften, 1957.

[30] _____. "The Translation of Numbers from the System of Remainder Classes to a Polyadic System by Change of Scale of Period," Stroje Na Zpracování Informací, Sborník IV (1956), pp. 53-64.

[31] _____ and Svoboda, A. "Operátorové Obvody," Stroje Na Zpracování Informací, Sborník III (1955), pp. 247-295 (1956).

[32] Westinghouse Electric Corporation. Modular Arithmetic Computing Techniques. ASD Technical Report No. 63-280. Wright-Patterson Air Force Base, Ohio: Aeronautical Systems Division, United States Air Force, 1963.