THE DESIGN AND IMPLEMENTATION OF A STRUCTURED PROGRAMMING LANGUAGE WITH FEW ARBITRARY SYNTACTIC RESTRICTIONS - THE INTERPRETIVE PHASE

A dissertation presented by

P.B. Gibbons

in partial fulfilment of the requirements

for the degree of

Master of Science in Computer Science

at

Massey University

August 1972

ACKNOWLEDGEMENTS

There are a number of people I wish to thank for the parts they have played, either directly or indirectly, in the production of this thesis.

Firstly, I would like to sincerely thank my supervisors, Professor G. Tate and Mr. R.W. Doran for all the guidance and encouragement they have given me during the preparation of this work. I would particularly like to thank Bob Doran for providing the original syntax of MUSSEL.

I am also extremely grateful to Miss Nola Gordon for the many long hours spent with her discussing and designing the implementation of MUSSEL; to Mr. P.J. Herbert for his co-operation in printing the structure diagrams; to the Massey University Computer Unit for absorbing much of the cost of the project; to the Mathematics Department for the use of its electric typewriter; and last, but by no means least, to the IBM 1620 II Computer at Massey, with profound apologies for the considerable amount of abuse it has quietly suffered during the development of the Interpreter.

I would also like to take this opportunity to thank my parents for their unfailing generosity and tolerance during my years of study at Massey, particularly while this thesis was being written.

Finally, I would like to say a very special 'thank you' to Noela for her constant patience, understanding, and untold assistance during the preparation of this work. For these qualities, alone, I shall always be grateful.

Massey University P.B. Gibbons
August 1972

TABLE OF CONTENTS

CHAPTER 0

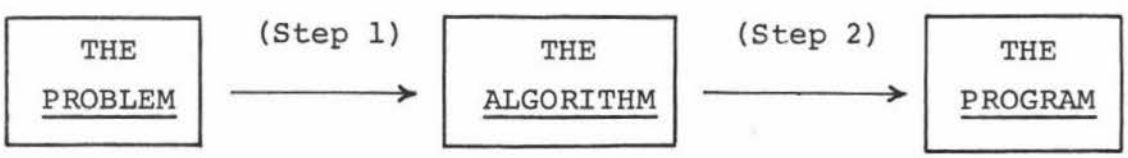
INTRODUCTION

THE CASE FOR A NEW LANGUAGE

The first and most important question that must be answered is, "Why in fact do we need a new programming language?". In order to answer this, we must really go back and try to answer the question, "What is programming?".

In designing MUSSEL, we have been very much influenced by the ideas of E.W. Dijkstra [35] and N. Wirth [7], [8], and [34]. Wirth, in particular, provides some strong criticism of present day programming courses, and in doing so, formulates some well-reasoned and constructive answers to the question, "What is programming?", or rather, "What should a programming course be?". His views, in fact, are representative of a growing dissatisfaction among many Computer Scientists with the conventional methods being taught to students as an aid to writing programs.

The process of writing a program to solve a particular problem on the computer may be divided into 2 fundamental steps. The first of these is the construction of a well-defined and efficient algorithm to solve the problem. The second is the translation of this algorithm to a well-structured, effective, and reliable program for the computer.



Wirth's main criticism is that too many present day courses concentrate on step 2 of this process, and therefore tend to ignore the fundamentals of step 1. In other words, while language details are being taught and examined at great length, the actual heart of the matter, the construction of algorithms, is largely left to the intuition of the student. Experience has shown that languages are not merely tools to communicate with the computer, but are the framework in whose terms the programmer thinks and designs. And as many of the languages used are extremely awkward to express algorithms in, it is not therefore surprising that the end product of such courses is more the knowledge of the details and idiosyncrasies of the languages used, rather than an appreciation of the disciplined reasoning needed to develop algorithms. In fact, as Wirth puts it, the course could almost be described as a 'computer disappreciation' course! This is particularly true of a language such as FORTRAN.

More emphasis, then, should be placed on step 1 of the programming process i.e. on teaching the student disciplined methods of constructing algorithms. Wirth believes that this discipline can be achieved only by emphasising the essential principles of program structuring from the very beginning. He believes that programming should be considered as an activity consisting of a succession of steps, each one breaking up a given task into a number of subtasks, starting with the original problem, and ending when all subtasks can be expressed by elementary statements of the underlying programming language. This process of successive refinement of tasks was originally proposed by Dijkstra, and has now come to be called "structured programming".

A good way of picturing the process is by the use of the "structure diagram" (R.W. Doran [3], [4], & [5]). In such a diagram, refinement of an algorithm is represented

pictorially by a 'tree', the nodes of which are boxes essentially containing statements of subalgorithms. (More strictly, such a diagram represents a List, rather than a tree, since the algorithms represented may be resursive.)

Having assumed, then, that the use of structure diagrams was a good method of formulating algorithms, we were then faced with the problem of choosing a suitable language which was compatible with these ideas, and which would therefore make step 2 as natural as possible for the student. FORTRAN, besides being unstructured, has a number of 'bad' features (see Chapter 1, and also Appendix E), and is therefore not particularly satisfactory. ALGOL, on the other hand, is indeed structured. Some of the concepts of the language, however, are probably a little sophisticated for instruction in a first year course. And among other things, we had no PL/I compiler.

There were, then, 2 options open to us. The first would be to modify an existing language so as to cover up or remove some of its 'bad' features. While this would at least have the advantage of teaching the student a language which is in wide use, it would not be entirely satisfactory, as it is unlikely that all of the bad features could be easily removed, if at all.

The other option, the one we chose, was to create a new language, entirely compatible with our ideas on structured programming, and yet simple enough for the average student to be able to grasp quickly and naturally. Although this approach has the disadvantage of teaching the student a language which is not in wide use, we believe that use of the language, in conjunction with a course on structured programming, would so train the student in the basic fundamentals of programming, as we have defined them, that he

should have no difficulty in adapting to other, more widely used languages in later, more advanced courses.

The language we have designed has been called MUSSEL (Massey University Structured Student Language). The language has 2 main features:

1. It is structured.

2. The syntax has been made as natural as possible for the student.

In fact, both the structure and syntax of MUSSEL resemble very closely the form of the structure diagram. In this way, we believe that the student will be able to translate his structure diagram quite easily to program form.

To summarise, then, we have created a language to fit in with what we believe is a good strategy for formulating and expressing algorithms. In doing so, we have essentially endeavoured to shift the emphasis in a programming course from the learning of a language, to the construction of algorithms.

CHAPTER 1

A DESCRIPTION OF THE LANGUAGE

1.1. Admissible Characters

MUSSEL uses a basic character set consisting of the 26 upper case letters of the alphabet, A through Z, the digits 0 through 9, and the special characters + - * / ! : . , b ( ) and $.

1.2. Constants

There are 3 types of constants in MUSSEL.

(i) Numbers

These may be real or integer, the latter being written without a decimal point or exponent.

examples: 5 -3000 278.3496 +1.0 -6.5E-7 E10
.00463E44

All numbers are represented internally by MUSSEL as a 7 digit mantissa (with the implied decimal point to the right) accompanied by a 2 digit exponent. Under this representation, integers are treated essentially as a subset of the reals and are distinguished internally by a zero exponent. The range of integers therefore is from -9999999 to +9999999, while reals may range from ± E99 to ± 9999999E99.

(ii) Booleans

These may be written as TRUE or FALSE, or in the abbreviated forms T or F. Internally they are represented by the digits 1 and 0 respectively.

(iii) Strings

A string in MUSSEL consists of zero or more characters from the basic character set, excluding the record mark. Constants are enclosed between exclamation (!) marks. Similar to the convention adopted in PL/I, each exclamation mark itself in a string is represented by 2 such marks.

```
examples:  !THIS IS A STRING!
           !!     (null string)
           !!!!   (single exclamation mark)
           !!!QUOTE!!!    (represents !QUOTE!)
```

1.3. Operators

The 4 types of operators used in MUSSEL expressions are as follows:

(i) Arithmetic

These include + - * / ./.(integer divide), and **(exponentiation).

(ii) Logical

Used within conditional expressions, these include .AND., .OR., and .NOT.

(iii) Relational

Written as in FORTRAN IV, these include .LT., .LE., .EQ., .NE., .GE., and .GT., being equivalent to the relational operators <≤=≠≥> as used in ALGOL.

(iv) String

Strings may be concatenated by the use of the operator .CAT. Other common string manipulations may be performed by the use of library functions.

The above operators have the following precedence:

Operator	Priority
.OR.	1
.AND.	2
.NOT.	3
.LT. .LE. .EQ. .NE. .GE. .GT.	4
.CAT.	5
+ -	6
* / ./. NEG	7
**	8

(NEG indicates unary minus)

1.4. Names

These are used in MUSSEL to identify variables, labels, procedures and functions. Although a name clearly should be as short as possible, in many languages it is often hard to find a good mnemonic to satisfy the restricted length requirements. For this reason MUSSEL has imposed no such restrictions, allowing names to be of any length, consisting

of a letter followed by a string of letters and/or digits.

examples: I SUM U124R902 VERYVERYVERYLONGNAME

1.5. Variables

All main variables used in a MUSSEL program must be reserved at the head of the program.

example: RESERVE X,SUM,MEAN,VARIANCE

The form of this instruction indicates clearly to students the process of setting aside storage space for variables. Another nice feature for the student is the absence, unlike FORTRAN, PL/I, ALGOL and COBOL, of explicit type declarations. Each variable's type is set implicitly through an assignment instruction and hence does not remain static throughout a program. The implementation of this is not a great drawback as MUSSEL, being an interpretive system, can easily check the types of variables during the execution phase.

1.6. Arrays and Subscripted Variables

Arrays in MUSSEL may be 1 or 2 dimensional only, which although restrictive is considered adequate for student requirements. Although the system does not provide for dynamic arrays (see procedures), main arrays, like outer block arrays in ALGOL, must have constant bounds.

example: RESERVE A(1:60,2:10),B(-1:1),(C,D,E(-4:4,0:5))

Subscripts in MUSSEL, like ALGOL and PL/I, may be any arithmetic expression with the convention that non-integral values are rounded to the nearest integer.

examples: A(I), B(J,K), C(A(4),2*SIN(4*PI*B(-4,0)))

1.7. Assignment Instructions

It is unfortunate that most languages, including FORTRAN and PL/I, use the equal (=) sign to specify the assignment operation.

example: I=I+1

The criticism here, of course, is that the meaning of this sign contrasts with its meaning in algebra. FORTRAN has the further disadvantage of not catering for multiple assignments.

To emphasise the assymmetry of the assignment relationship, ALGOL in fact replaces the equal sign by ':=', but even here the new operator resembles closely the equal sign. A further criticism of ALGOL is that the multiple assignment, although evaluated quite unambiguously according to the rules in the ALGOL report, is in a form which may cause trouble to students.

For these reasons MUSSEL has an assignment instruction of the form

SET...loc...TO exp

Analogous to the MOVE verb format in COBOL, this form expresses multiple assignments in a natural way and emphasizes quite clearly the operation of replacing the contents of a location (or locations) by the value of an expression.

examples: SET I TO I+1
SET A,B,C TO 0
SET DATE TO DAY.CAT.MONTH.CAT.YEAR

1.8. Program Structuring

Apart from procedures, which are discussed below, program structuring in MUSSEL is achieved through the use of instruction groups. These are equivalent to compound statements in ALGOL and similar to DO groups in PL/I i.e. they are optionally labelled sets of instructions grouped between the instruction parentheses DO and END, each group being treated syntactically as a single instruction. The absence (without procedures) of a more complex structure, such as the block in ALGOL, simplifies the structuring of MUSSEL a great deal. All main variables are reserved at the head of a program and are, together with labels, global to the whole of that program. In this way a student is able to structure and display his program clearly (indenting 5 columns per group) without having to master the more involved variable scope problem present in ALGOL.

1.9. Conditional Groups

A variety of conditional instruction groups is provided in MUSSEL. The IF-group, of the form

(i) IF condexp THEN simple instruction

or (ii) IF condexp
THEN instruction

corresponds to similar IF-THEN statements in ALGOL and PL/I. FORTRAN IV's logical IF statement corresponds to form (i) only.

examples: IF X.GT.Y .OR. A.LT.B THEN SET F TO 1

(other examples over page...)

```
IF N .GE. M
THEN DO
          SET MEAN TO SUM/M
          EXIT
     END
```

MUSSEL's conditional group, of the form

```
DO IF condexp
     THEN instruction
     ELSE instruction
END
```

corresponds to similar IF-THEN-ELSE statements in ALGOL,PL/I and COBOL, while the rather unnatural arithmetic IF statement is its counterpart in FORTRAN.

```
example:  DO IF I .EQ. 0
               THEN DO
                         SET S TO N+K
                         SET P TO P+S
                    END
               ELSE DO
                         SET S TO N-K
                         SET P TO P-S
                    END
          END
```

A feature of MUSSEL which is unavailable in the 4 mentioned languages is the general choice group. The use of this feature avoids the complicated nesting of conditional groups in some programs which would thereby obscure the simple structure of the program.

```
example:  DO CHOICE OF
               IF TEMP .LT.  0 THEN SET STATE TO !FREEZING!!!
               IF TEMP .LT. 10 THEN SET STATE TO !COLD!
               IF TEMP .LT. 15 THEN SET STATE TO !COOL!
               IF TEMP .LT. 25 THEN SET STATE TO !MILD!
               IF TEMP .LT. 30 THEN SET STATE TO !WARM!
               IF TEMP .LT. 50 THEN SET STATE TO !HOT!
               ELSE SET STATE TO !BOILING!!!
          END
```

The 'ELSE...' in the above form is optional.

Another feature is the CASE group which has its counterpart in Burrough's ALGOL. This is MUSSEL's equivalent of the computed GO TO in FORTRAN, PL/I and COBOL, and the switch in ALGOL. Its use may be seen in the following simple decoding example:

```
          DO CASE N OF
               SET STUDENT TO !FRESHER!
               SET STUDENT TO !2ND YEAR!
               SET STUDENT TO !3RD YEAR!
               SET STUDENT TO !4TH YEAR!
          END
```

An advantage in MUSSEL however is that by the insertion of bounds, the index need not necessarily range from 1.

```
example:  DO CASE OPCODE IN (22,26) OF
               SET OP TO !SUBTRACT!
               SET OP TO !MULTIPLY!
               SET OP TO !COMPARE!
               SET OP TO !TRANSMIT DIGIT!
               SET OP TO !TRANSMIT FIELD!
          END
```

1.10. Loops

The REPEAT-group of the form

```
DO REPEAT (control)
     ...instruction...
END
```

provides for extensive looping facilities in MUSSEL. The variety of controls available corresponds essentially to the PERFORM verb options available in COBOL.

examples: (i)
```
DO REPEAT FOR I FROM FIRST TO LAST BY STEP
     ...
END
```

(ii)
```
DO REPEAT FOR STRING SET TO !IT'S!,!AN!,!EXAMPLE!
     ...
END
```

The forms displayed above essentially correspond to those available in ALGOL (with its 'for' statement) and PL/I (with its 'DO variable = iteration list' statement), with the additional freedom of allowing set list elements in form (ii) to be of any type. (Note - FORTRAN's DO statement corresponds to form (i) above with the restriction that all controlling variables be positive at run time). As is allowed in these 3 languages, the STEP in form (i) may be omitted if 1. Additionally MUSSEL allows the test value also to be omitted.

examples: (iii)
```
DO REPEAT FOR I FROM 12*K**2
     ...
END
```

(iv)
```
DO REPEAT FOR I FROM 1 TO 100
     ...
END
```

(v)
```
DO REPEAT FOR I FROM N./.4 BY 2*C(U,V)
     ...
END
```

As in PL/I, the controlling expressions in the above forms are all called by a value rather than by name if the loop is regarded as a procedure i.e. the expressions are evaluated and the results stacked on entry to the loop at run time and are thus unaffected by any changes within the loop. This both simplifies the implementation and avoids some nasty features of ALGOL.

MUSSEL also provides for iteration without indexing a variable.

examples: (vi)
```
DO REPEAT
     READ X
     IF X .EQ. 9999 THEN EXIT
     SET SUM TO SUM+X
END
```

(vii)
```
DO REPEAT UNTIL P .EQ. Q
     ...
END
```

(viii)
```
DO REPEAT WHILE X .NE. 0
     ...
END
```

The last form only is available in PL/I.

Another feature of MUSSEL, unavailable in ALGOL, PL/I, or FORTRAN, but available using the PERFORM verb in COBOL, is the 'REPEAT valexp TIMES' loop.

```
example:  DO REPEAT N TIMES
               READ X
               SET SUM TO SUM+X
               SET SUMSQUARES TO SUMSQUARES+X*X
          END
```

Where REPEAT-groups are nested, the word EXIT specifies an exit from the innermost loop. If an exit from another loop is required, then the appropriate label must be specified also.

example:

```
  OUTLOOP: DO REPEAT FOR I FROM 1 TO M
                DO REPEAT FOR J FROM 1 TO N
                     IF B(I,J) .EQ. 0
                     THEN DO
                               SET STATE TO !ERROR!
                               EXIT FROM OUTLOOP
                          END
                     SET A(I,J) TO A(I,J)/B(I,J)
                END
           END OUTLOOP
```

It is important to note that MUSSEL has no equivalent of the GO TO statement of other languages. The reason for the absence of this unconditional branch instruction is to preserve the structure of a program. In this way a student is encouraged to refine his program achieving transfer of control by means of conditional groups and EXIT instructions,

or through the use of procedures and functions.

1.11. Procedures

As there are no explicit type declarations in MUSSEL, procedures and functions have similar declaration forms. These must be placed at the head of a program, immediately following the main variable reservations, and are of the following general form:

```
DEFINE procedure name (ON formal parameter list) AS
DO
     (RESERVE local variable list)
      ...instruction...
END
```

example:

```
DEFINE SWAP ON A,B AS
DO
     RESERVE HOLD
     SET HOLD TO A
     SET A TO B
     SET B TO HOLD
END
```

A function value assignment in MUSSEL is accomplished by use of the VALUE IS instruction.

example:

```
VALUE IS X**N/FACTORIAL(N)
```

The form of this instruction indicates to students the special nature of this assignment.

The MUSSEL translator distinguishes procedures and functions by their manner of call. A procedure call is specified by the EXECUTE statement.

```
examples:  EXECUTE SWAP(A,B)
           EXECUTE TRANSPOSE(A,N)
           EXECUTE P
```

A function call on the other hand is indicated by the appearance of the function name (and actual parameters, if any) in an expression, as in most other languages.

```
examples:  SET NIBBLE TO SUBSTRING(WORD,1,4)
           SET Y TO F(A**2,B*C)
```

Parameters in MUSSEL may be expressions or procedure names and are called by reference, as in PL/I. In particular, whole arrays may be altered by passing across array names as parameters to procedures. This has the additional advantage that space in a procedure need only be reserved for the array word address, rather than the array elements themselves.

As in ALGOL, the scope of variables reserved within a procedure does not extend beyond that procedure. However, variables reserved in the main program are global to all procedures, and any procedure may call any other, including itself. There is just one restriction - procedure declarations may not be nested. The purpose of this is to preserve the simple structure of the language. The block structure is essentially 2-level, as opposed to the multi-level structure of ALGOL which may be difficult for the novice programmer to comprehend.

1.12. Recursion

It is well known that recursion plays a powerful and essential role in the processing of many data structures. There are many areas (e.g. list processing) in which to

program iteratively, rather than recursively, would be quite awkward, if not impossible in practice. Yet a student learning FORTRAN for example, which does not allow a subroutine to be called recursively, is forced to think and program iteratively, and is thereby not encouraged to look for recursion. This may become a bad habit - when introduced to the concept at a later date, he may find it hard to understand.

For these reasons, recursion is considered an essential facility in MUSSEL. By programming in the language, a student may be introduced to the recursive way of thinking at an early stage and will thereby be in a position to weigh the respective merits of iterative and recursive programming.

1.13. Dynamic Arrays

These are reserved locally to a procedure. Bound expressions must be arithmetic, consisting of main variables and/or parameters to the procedure. As is the case with subscript expressions, the convention is that non-integral bound values are rounded to the nearest integer.

examples: RESERVE A(1:N),(B,C(BOT:TOP)),D(0:I*J/K,L:E(U,V))

These rules are similar to those applying to bound expressions in ALGOL. In FORTRAN IV however, where dynamic arrays are permitted in subprograms, subscripts and bounds must be (restricted) integer expressions ranging up from 1.

1.14. Input / Output

In FORTRAN, input/output, particularly the FORMAT statement, has long been a major bug for students. For example, a recent analysis (see appendix) of 511 first year

student programs at Massey University revealed that in fact nearly 30 percent of all programming errors were linked with the formatting of I/O.

A similar, more comprehensive examination of student programs has also been carried out under the DITRAN system at the University of Wisconsin. Here a total of 5,158 programs were analysed. From this analysis it was found that while I/O and FORMAT statements made up 17 percent of the total number of statements, they accounted for 10 percent of the total compilation time errors, and 64 percent of the total execution time errors.

Yet this is an area, we believe, which should not concern the student unduly at an early stage in his programming career - initially there are more fundamental concepts for him to grasp. For this reason MUSSEL has endeavoured not only to simplify I/O as much as possible for the novice, but also to allow him at a later stage to format his output in a clear and natural manner.

The following sections describe some of the features of the I/O system adopted by MUSSEL. Additional details may be obtained from section 3.15.

1.14.1. Input

MUSSEL has a simple unformatted stream input with data being separated on cards by a blank or a comma. String constants have the additional delimiter '!'. Arrays are read in row-wise by the specification of the array name.

```
example:  RES X,Y(1:10),Z(0:19,0:49)
          READ X,Y,Z
```

1.14.2. Output

For the novice programmer, MUSSEL provides a standard formatted output. (See section 3.15.2.). Here again arrays are output row-wise by specifying the array name in the PRINT list.

example: PRINT X,Y,Z

At a later stage, however, the student may optionally include a PICTURE specification with his output data, similar to the forms used in COBOL and PL/I. Although this may appear to be a primitive form of specification, we believe it will aid the student in visualising exactly how his O/P will appear on the printed page.

Characters used in the MUSSEL picture specification are as follows:

* The position indicated will contain an alphanumeric character. With a number, however, leading zeros will be automatically suppressed unless otherwise specified.

B A blank is inserted in the position indicated.

Those used specifically with numeric data are as follows:

. A decimal point will be printed in the position indicated.

, A comma is printed in this position if there is a printed digit to the left; otherwise a blank is printed.

9 A zero will not be suppressed in this position.

S If this character is placed at the head of a picture, the sign (+ or -) will be floated up to and printed at the head of the number concerned. Otherwise the sign of the number will be printed in the position indicated.

- Similar to S, except that the sign is printed only if the number is less than zero. Otherwise a blank is printed.

E The character 'E' is printed in the position indicated. This is used in the E - format specification of a number. It should be noted here that PIC specifications for exponents are subject to the same rules as those applying to mantissas. In particular, this means that leading exponent zeros are automatically suppressed unless otherwise specified.

In addition the following rules apply:

(i) Replication of the characters '*', 'B', and '9' may be specified by a bracketted integer following the character concerned.

example: *(4).*(2) means ****.**

(ii) The association of a picture with a variable remains optional, the output being in standard format if a picture is omitted.

(iii) Unlike COBOL, the picture accompanies the

variable name concerned in a PRINT list, and is of the form (PIC= 'picture').

example:

```
PRINT A,B(PIC=***.**),C(PIC=S***.**ES99)
```

(iv) Whole arrays may be output by specifying the array name and a single picture only. Each element is then printed out according to the picture with a standard 4 spaces between elements.

example:
```
RES A(0:M,0:N)
...
PRINT A(PIC=****.***)
```

(v) Character strings are left justified (or left adjusted) under a PICTURE specification i.e. if necessary a string is extended with blanks on the right or truncated on the right to the specified field width.

examples:

Internal String	PIC=	Output
HEADING	*(7)	HEADING
HEADING	*(10)	HEADINGbbb
HEADING	*(4)	HEAD

(vi) The required output form of Boolean constants may be specified by picture. If a picture is absent, the form TRUEb or FALSE is printed.

examples:

Boolean Value	PIC=	Output
TRUE	none	TRUEb
FALSE	none	FALSE
TRUE	*	T
FALSE	*	F
TRUE	****	TRUE
FALSE	****	FALS
TRUE	*(10)	TRUEbbbbbb
FALSE	*(10)	FALSEbbbbb

(vii) With pictures specifying numeric data no more than one of each of the characters . and E is allowed per picture.

examples:

Internal Representation		PIC=	Output
Mantissa	Exponent		
0001234	00	****	1234
		********	bbbb1234
		S*****.*ES99	bb+123.4E+01
		**999999	bb001234
-0012345	-03	****.***S	bb12.345-
		***.*	b12.3
		****.*E-**	1234.5E-b2

It can be seen that the PIC specification used in MUSSEL differs in a number of ways from that used in COBOL and PL/I. The main reason for these changes and simplifications are as follows:

(i) As MUSSEL is intended primarily as a student language, it is felt that a cheque protection character (e.g. * in COBOL) would not be used often enough to justify its inclusion in the picture specification. This leaves the asterisk as a reasonable character to use to specify an alphameric position in a picture. If, however, it should be required as a cheque protection character, then this could quite easily be programmed into the output by use of string processing subroutines. In a similar way the $ character, although not considered necessary as a special character, could quite easily be appended to output if necessary.

(ii) MUSSEL uses the picture specification to edit output only. The character 'V', specifying an implied decimal point, is therefore not required.

(iii) We believe it is more natural to have leading zeros in a number automatically suppressed unless otherwise specified.

(iv) We believe it is important that a picture correspond as closely as possible to the intended printed O/P. For this reason insertion of the sign of a number is not automatically implied in MUSSEL. A student using the language must specifically ask for the sign to be appended by the use of S or - and if this is not done, then the sign will be omitted.

(v) MUSSEL does not, as in COBOL, use the picture specification to define a variable's type.

Printer Controls

The following are used in MUSSEL:

NEWPAGE	skip to newpage
NEWLINE	skip to newline
NEWLINE(N)	skip N lines
SPACE	shift one space to right
SPACE(N)	shift N spaces to right
TAB(N)	shift to (N+1)th column

example:

```
PRINT NEWPAGE,TAB(30),!HEADING!,NEWLINE,TAB(30),!*******!,
      NEWLINE(2),TAB(28),!X= !,X(PIC=***.**)
```

The output with X containing 339.56 is as follows:

```
(30 spaces)...                  HEADING
                                *******

(28 spaces)...                X= 339.56
```

1.15. An Example

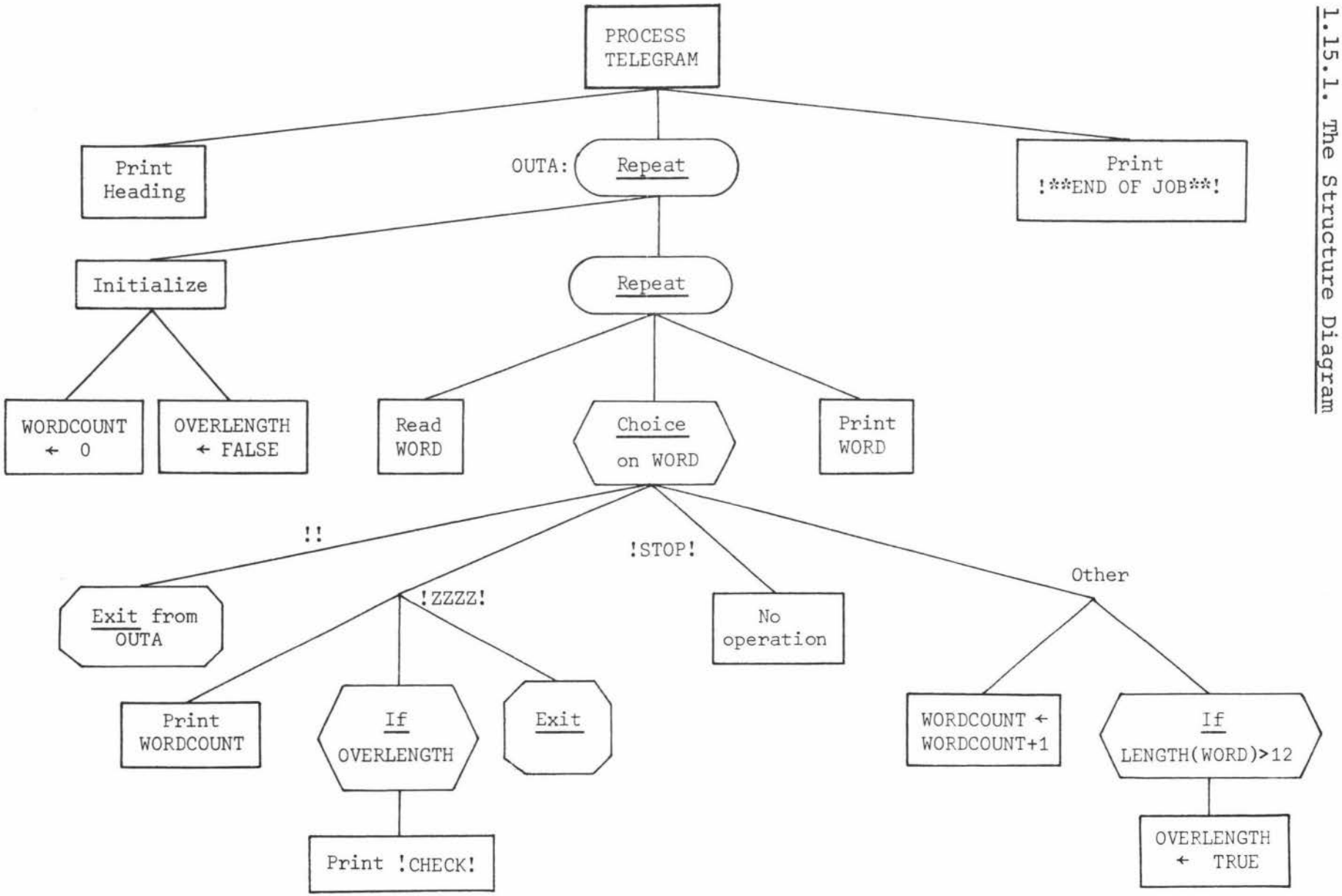
The following example[1] illustrates just how easily a MUSSEL program follows from a structure diagram.

Problem

A program is required to process a stream of telegrams. This stream is available as a sequence of words. Each telegram is delimited by the word !ZZZZ!. The stream of telegrams itself is terminated by the occurrence of the empty telegram i.e. the null word !!. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words !ZZZZ! and !STOP! are not chargeable and words of more than 12 letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.

---

[1] A simplified version of that used by Henderson and Snowdon in "An Experiment in Structured Programming." Technical Report Series, No.18, University of Newcastle (1971).


PROCESS TELEGRAM
Print Heading
OUTA:
Repeat
Print !**END OF JOB**!
Initialize
Repeat
WORDCOUNT ← 0
OVERLENGTH ← FALSE
Read WORD
Choice on WORD
Print WORD
!!
!STOP!
Other
Exit from OUTA
!ZZZZ!
No operation
Print WORDCOUNT
If OVERLENGTH
Exit
WORDCOUNT ← WORDCOUNT+1
If LENGTH(WORD)>12
Print !CHECK!
OVERLENGTH ← TRUE

1.15.2. The Program

```
#NAME  BROWN,CHARLES K.
*
*PROGRAM TO PROCESS STREAM OF TELEGRAMS
*
DO
     RESERVE WORDCOUNT,OVERLENGTH, WORD
     PRINT NEWPAGE,TAB(30),!TELEGRAM ANALYSIS!,
          NEWLINE,TAB(30),!*****************!,NEWLINE(3)
OUTA:DO REPEAT
          SET WORDCOUNT TO 0
          SET OVERLENGTH TO FALSE
          DO REPEAT
               READ WORD
               DO CHOICE OF
                    IF WORD .EQ. !! THEN EXIT FROM OUTA
                    IF WORD .EQ. !ZZZZ!
                    THEN DO
                              PRINT NEWLINE,!WORDCOUNT = !,
                                   WORDCOUNT(PIC=***)
                              IF OVERLENGTH
                              THEN PRINT SPACE(5),!**CHECK**!
                              EXIT
                         END
                    IF WORD .NE. !STOP!
                    THEN DO
                              SET WORDCOUNT TO WORDCOUNT+1
                              IF LENGTH(WORD) .GT. 12
                              THEN SET OVERLENGTH TO TRUE
                         END
               END
               PRINT WORD,SPACE
          END
     END OUTA
     PRINT NEWLINE(3),!**END OF JOB**!
END
```

CHAPTER 2

AN OVERVIEW OF THE IMPLEMENTATION

MUSSEL is being implemented on the IBM 1620 II computer at Massey University. This is an interim measure only, the system being designed to be as machine independent as possible to facilitate adaption to other computer systems. Clearly this ideal has its limitations. For instance, given the opportunity, one would obviously take advantage of a wider character set than that which is available on the 1620. Also, the 1620 is essentially a variable word length machine, and in this sense differs from the majority of computers in use today.

The system is an interpretive one i.e. during the compilation phase (Phase I) the source code is translated into an intermediate language which is then interpreted (Phase II). There are 2 main reasons for this type of implementation. The first is the need, with student jobs, to minimize compile time, ignoring execution time (which is usually trivial, in any case). The second is the need to maximize diagnostics, particularly during the execution phase. The use of an interpreter should enable us to achieve both these goals.

During the translation phase, when the source program is converted into the reverse polish intermediate language, 2 program listings are given. The first corresponds exactly to the submitted program, while the second is an amended form, with the line number appended, correct indenting, and error

messages inserted where necessary. The recogniser itself is syntax-directed, top-down, and uses a graph with links to the associated language productions. Because MUSSEL is designed as a student language, it is essential that the error diagnostics be satisfactory. Using a top-down parser it is difficult to keep track of exactly where an error has occurred. It is therefore conceivable that in this case a switch will have to be made to bottom-up parsing, thereby pinpointing the error more precisely. In fact, at a later stage it is hoped to implement a small amount of error correction with, for example, misspelt key words.

This translation phase of MUSSEL will be more fully described in Miss N.M. Gordon's M.Sc. thesis, entitled: "The Design and Implementation of a Structured Programming Language with Few Arbitrary Syntactic Restrictions - the Compilation Phase".

The Intermediate Language itself (see Appendix B) is similar to that used by Randell & Russell [1], being essentially a form of 'Reverse Polish' notation consisting of a set of object program operations which are obeyed by the Interpreter. The Intermediate Language here, of course, is much simpler, due to the simpler form of MUSSEL compared with ALGOL. For instance, the absence of explicit type declarations in MUSSEL means that a variable's type is not determined until the execution phase. TA (Take Address) and TR (Take Result) operators can therefore replace the operators TIA (Take Integer Address), TRA (Take Real Address), TBA (Take Boolean Address), TIR (Take Integer Result), TRR (Take Real Result), and TBR (Take Boolean Result), used by Randell & Russell [1].

The MUSSEL Interpreter, MUSINT, uses a run time Stack for expression evaluation and dynamic storage allocation.

During this phase, MUSINT has access not only to the names of variables and procedures, which have been stored on disk by the Translator, but also to the line number of the piece of original source code at which it is currently working. This line number updating is incorporated into the Intermediate Language. In this way, the Interpreter will be able, during execution, to keep a close check on such things as variable types and definition, subscript values, illegal operations, and improper procedure call sequences, and on the detection of an error will, in most cases, be in a position to give meaningful error messages, quoting not only the types of errors themselves, but also the names of the variables concerned and the appropriate line numbers.

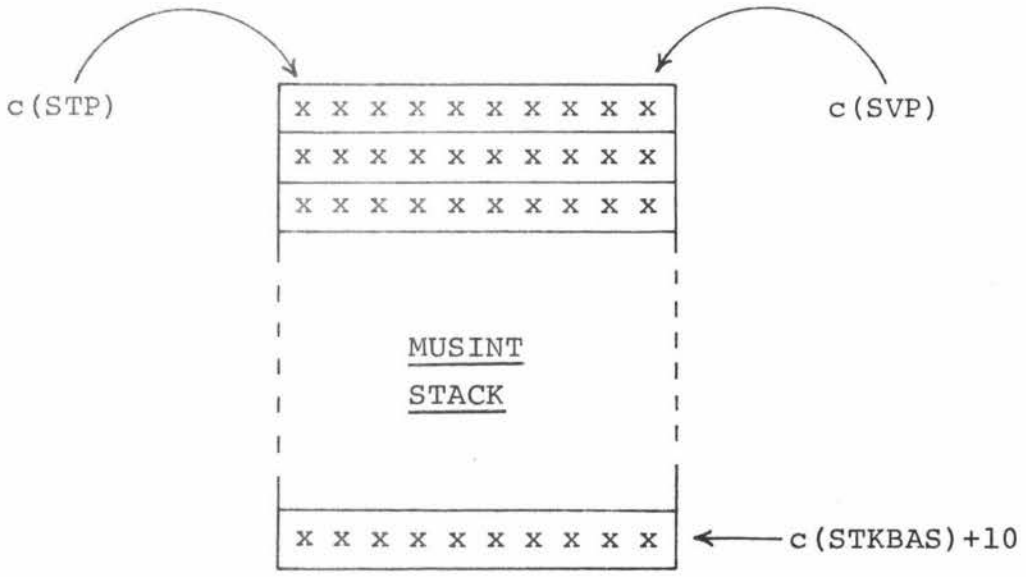
More of the potential diagnostic capabilities of MUSSEL will be discussed in Chapter 4.

CHAPTER 3

INTERPRETING THE INTERMEDIATE LANGUAGE

3.1. The Interpreter Stack

Each Stack unit (10 digits) consists of a type part (1 digit) followed by a value part (9 digits). Stack manipulation, consequently, is controlled by the 2 main Stack pointers STP (Stack Type Pointer) and SVP (Stack Value Pointer). The contents of STP, c(STP) (see structure diagram section, appendix, for notational explanation) in general point to the current top Stack type digit at any time, whereas c(SVP) points to the low-order digit of the corresponding top Stack value. The incrementing or decrementing of these pointers is achieved through the use of the MUSINT subroutine POPTOP.



The value part of a Stack unit may take on various forms depending on its particular type. The following is a complete summary of the Stack units used by the Interpreter:

	Type	Value
Null string	1	
Number	2	$\bar{x}\ x\ x\ x_{\uparrow}\ x\ x\ x\ \bar{x}_{\uparrow}\ x$ mantissa exponent
Boolean value	3	$\bar{x}\ x$ (01-TRUE 00-FALSE)
Address	6	$\bar{x}\ x\ x_{\uparrow}\ x\ x$ core address
String descriptor	7	$\bar{x}\ x_{\uparrow}\ x\ x\ \bar{x}\ x\ x_{\uparrow}\ x\ x$ length address
Array word (AW)	4	$\bar{x}_{\uparrow}\ \bar{x}\ x\ x_{\uparrow}\ x\ \bar{x}\ x\ x_{\uparrow}\ x$ dim. 1st array elt addr. SMF addr.
Return address word (RAW)	9	$\bar{x}\ x\ x_{\uparrow}\ x\ x\ \bar{x}\ x_{\uparrow}\ x\ x$ previous RAW addr. return I.L. address
Undefined Stack unit	0	
Repeat-times control word	$\bar{8}$	$\bar{x}\ x\ x\ x_{\uparrow}\ x\ x\ x\ \bar{x}\ x$ No. of loop iterations (integer)

The form of each of the above Stack units will be explained more fully in the appropriate section below.

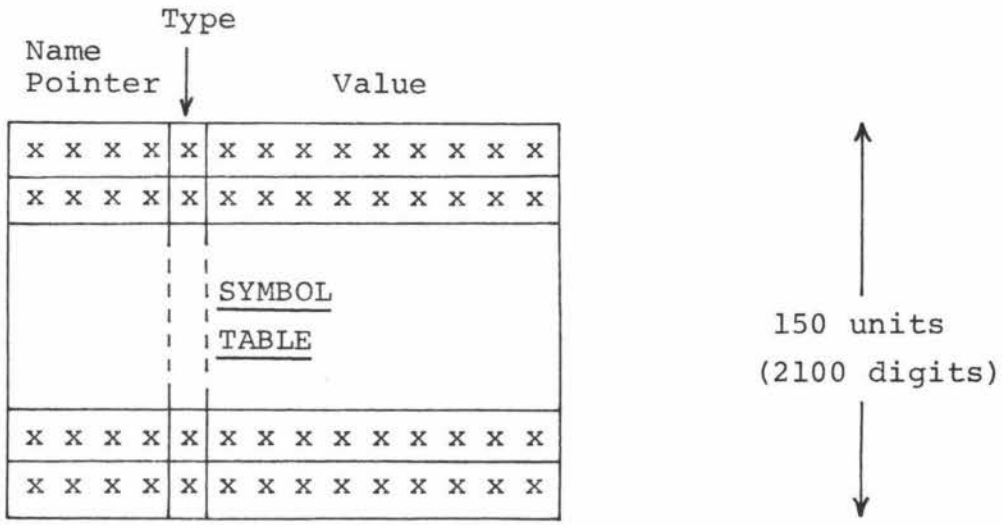
The above summary shows that with some units, particularly the Boolean and null string types, there is in fact a considerable amount of waste storage. In view of the limited amount of core storage available in the 1620, the possibility of using an alternative stack with variable length units was considered. Although this would save space in the Stack, it would require more complicated stack management routines. Execution would therefore be slower, with the added probability that the extra storage required for these routines would more than cancel out the space saved in the Stack. Adaption of the system to fixed word length machines would also be more difficult.

With these considerations in mind, then, it was decided to use a stack with fixed length units. As far as the actual length was concerned, ten seemed a good number to work with, and was sufficient to accommodate most pieces of stacked information, including a number in floating point form with a sufficient number of significant digits for student use.

3.2. The Symbol Table

The Symbol Table, during the interpretive phase, is of the form shown on the following page.

During the translation phase, each main variable, array, or procedure appearing in a MUSSEL program is assigned a unique unit in the table, the address of which is the 3-digit relative position of the particular unit in the table. A 4-digit pointer (relative to some base) to the name of each item in the table is also set up. This name area is stored on disk by the Translator and can be called into core and

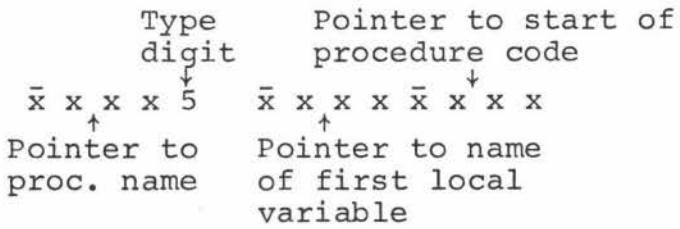


referenced by the Interpreter in the event of an error.

As mentioned earlier, a feature of MUSSEL is the absence of explicit type declarations, each variable's type being set implicitly through an assignment instruction. This means that a variable's type cannot be set during the translation phase and represented by I.L. code, as in the ALGOL 60 compiler of Randell & Russell. Instead the type must be stored with the value, remaining at '0' ('undefined') until the interpretive phase when it may be defined and changed during execution. In other words, whereas in the ALGOL compiler the type of a result is 'stored' with the I.L., in MUSSEL it must be stored with the result itself. Additionally, this dynamic type setting feature of MUSSEL implies that the Interpreter, on storing a result, need not check that the result type corresponds to the addressed location type (as must be done in the ALGOL compiler) - instead the old type is automatically overlayed.

Type and value parts for Symbol Table units representing array words, null strings, string descriptors, Boolean values, and numbers, are similar to the corresponding parts for Stack units. However, as procedure declarations may not be nested in MUSSEL, thereby ruling out the possibility of a procedure-type local variable, a unit which appears in the

Symbol Table, but not in the Stack, is the Procedure Address Word (PAW). This is of the following form:

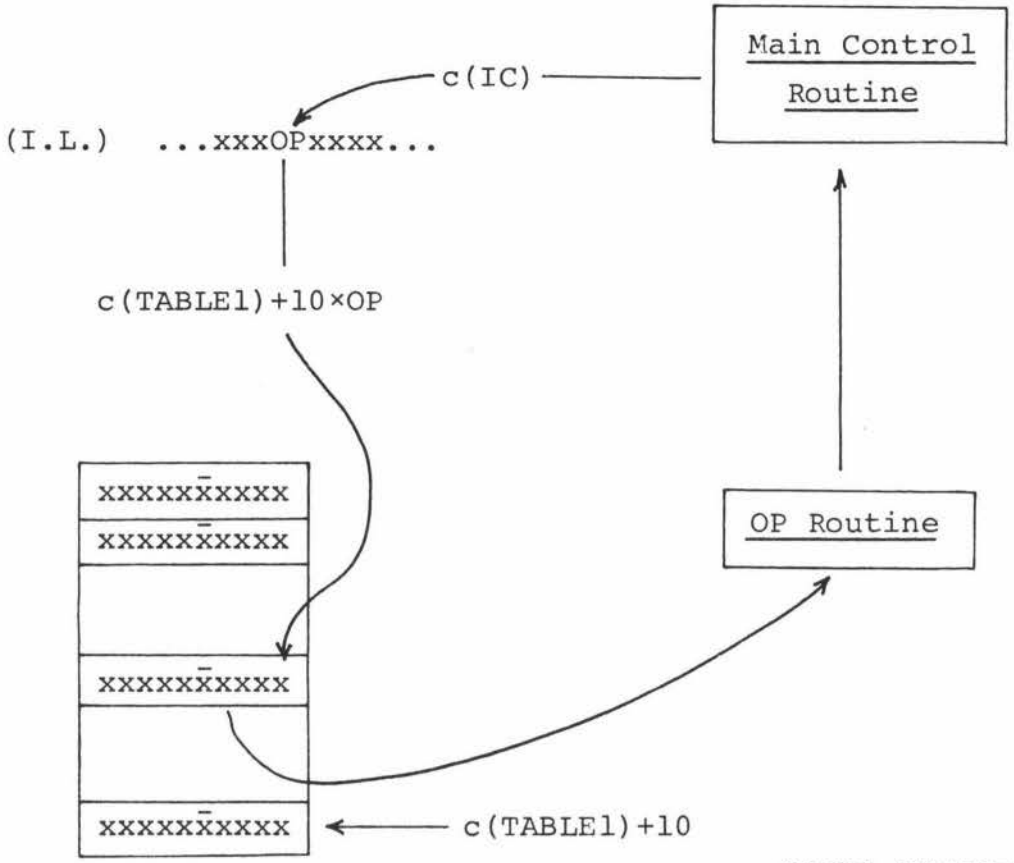


3.3. The Basic Structure of the Interpreter

The MUSSEL interpreter, MUSINT, consists essentially of a main control routine together with a number of sub-routines corresponding to the set of operators in the Intermediate Language. The function of this main control routine is to scan the Intermediate Language, picking up the individual operators and branching to the appropriate operator routines. On completion of each of these OP routines (with the exception of the END routine), control is in turn transferred back to the main control routine which will then proceed to pick up the next operator, and so on.

The I.L. operators themselves (see appendix) are coded sequentially, so that by use of a branching table (TABLE1) and suitable address modification, the main control routine can achieve a quick and efficient (indirect) branch to the appropriate OP routine. When the control routine is entered after the completion of each operator routine, the instruction counter, c(IC), points to the digit immediately to the left of the next instruction to be obeyed. Thus, by incrementing the counter by 2, the routine obtains a pointer to the right-most digit of the particular opcode, from which a branch to the corresponding routine can be obtained.

The basic structure of the Interpreter is displayed in the diagram shown on the following page.



BASIC STRUCTURE OF THE INTERPRETER

A similar table (TABLE2) is used by the expression operator routine, OP, to branch to each of its individual routines (viz. OR, AND, NOT, LT, LE, EQ, NE, GE, GT, CAT, +, -, *, /, ./., NEG, and **).

In designing MUSINT, prime consideration has been given to the saving of core, rather than speed of execution. This has meant, among other things, that subroutines have been created wherever possible. For instance, the subroutine POPTOP, which controls increments or decrements to the Stack pointers STP and SVP, essentially replaces only 2 instructions in-line. The subroutine is, however, called from about 30 different places in the Interpreter which implies that its use, in spite of increasing execution time, has achieved a considerable saving in core.

CORE STORAGE MAP OF 1620 DURING MUSSEL INTERPRETIVE PHASE

1620 Monitor and Tables	00000-02401
Interface variables	02402-02449
LN,TA,TR,TN,TN0,TN1,TB,TS,TNS, INDA,INDR,ST,STA,VI,UJ,BA,IFJ, and OP Routines	02450-06469
OP branching TABLE2	06470-06629
AND,OR,NOT,EQ,NE,GT,GE,LT,LE, CAT,ADD,SUB,MULT,DIV,INTDIV, NEG,RT,RF,RFWT,RS,CS,CSWB,RAP, RAF,CP,PE,TLA,TLR,MSF,RE,NP,NL, SP,TAB,I/O Load, & END Routines	06630-17600
Main Control Routine	17800-18000
Declarations	18200-20000
OP branching TABLE1	20005-20420
Interface routine & I/O overlay area (for RD,PR,& PRC routines)	20500-25799
Symbol Table	25800-27899
SMF's, Main Arrays, I.L., and Interpreter Stack ↓ ↓ ↓ ↑ ↑ ↑ String area	27900- -39999

The limited amount of core storage available has also necessitated the overlaying of the bulky I/O routines READ, PRINT, and PRINT CONTROL. These routines are stored on disk, and are called into core by a load routine when needed, the implementation of which feature will be discussed more fully in the appropriate section below.

A core storage map of the 1620 during the interpretive phase is shown on the preceding page. From this map it can be seen that the Interpreter, together with its various tables, occupies about 28,000 digits of core, leaving only 12,000 digits for the main arrays, I.L., interpreter Stack, and string area. Although this should provide sufficient space for testing the language, and should also cater for most elementary student programs, more space could readily be made available by overlaying some of the other more bulky routines such as the REPEAT and CASE routines.

3.4. Expression Evaluation

As indicated earlier, source language expressions are translated during Phase I into a reverse polish form similar to that used by Randell & Russell. The evaluation of such expressions is accomplished by using the Stack as a simple push-down store.

Corresponding to each simple main variable appearing in an expression an operator of the following form is generated:

TR (Take Result) $\quad\quad \bar{0}\ 3\ \bar{x}\ x\ x$

The parameter here specifies the Symbol Table address of the variable concerned. The function of the TR routine is to place the value at this address, together with its associated type, at the top of the Stack. The routine must of course

first check to see that the variable is in fact defined - if not, an appropriate error message is printed out. An added complication here is that if the result is in fact a string descriptor, then the corresponding string must be copied in the string area before a descriptor of the copy is placed at the top of the Stack. The significance of this will be explained more fully in the section on strings.

Corresponding to each constant appearing in an expression an appropriate "take constant" operator, accompanied by the constant itself, is generated in the Intermediate Language. An alternative method of setting up a special constant table, or of even storing constants in the Symbol Table during the translation phase was considered. However, although this method would indeed save storage space, as each constant would be stored only once, the added complication of setting up and maintaining such a table would imply a slower production of object code, which is contrary to one of the aims of the system. It was therefore decided to store constants in the stream of the I.L.

The following operators are used:

TN (Take Number) $\bar{0}\ 4\ \bar{x}\ x\ x\ x\ x\ x\ x\ \bar{x}\ x$

The parameter here specifies a number in floating point form. The function of the corresponding TN routine is to place this number at the top of the Stack, together with a 'number' type digit.

In order to save space, a special case has been made for the 2 commonly used constants '0' and '1'. Corresponding to these 2 constants the following 2 parameterless operators are generated:

TN0 (Take Number Zero) $\bar{0}$ 5

TN1 (Take Number One) $\bar{0}$ 6

For Boolean constants the following operator is used:

TB (Take Boolean) $\bar{0}$ 7 $\bar{x}$ x

The parameter here is a Boolean value (00 represents FALSE, 01 represents TRUE). 2 digits were used here to define a field - this is convenient when implementing the system on the 1620. The function of the TB routine is to place the Boolean value at the top of the Stack, accompanied by a corresponding 'Boolean' type digit.

TS (Take String) $\bar{0}$ 8 $\bar{x}$ x x x . . . x x 0 ≠

The parameter here is the alphameric code for the constant string itself, terminated by an alphameric record mark. The function of the TS routine is to pack the string away in the string area, assemble a corresponding string descriptor, and finally place this descriptor , together with a 'string descriptor' type digit, at the top of the Stack.

The following special parameterless operator is generated for a null string:

TNS (Take Null String) $\bar{0}$ 9

This merely places the type digit for a null string at the top of the Stack.

Corresponding to each operator appearing in the source code expression, object code of the following form is generated:

OP (Operator) $\bar{1}$ 9 $\bar{x}$ x

The parameter here specifies code for one of the following operations: OR, AND, NOT, LT, LE, EQ, NE, GE, GT, CAT, +, -, *, /, ./., NEG, and **. These operations are coded sequentially (see appendix) so that by use of a branching table (TABLE2), which is similar to TABLE1, the OP routine can transfer control quickly and efficiently to the appropriate routine.

In the case of the unary operation NOT or NEG, the associated routine applies the appropriate operation to the value at the top of the Stack, after first checking its type.

In the case of each of the remaining binary operations, the associated routine first checks the types of the top 2 Stack values. If they are in fact correct, it then applies the appropriate operation to these 2 operands, wipes them from the Stack, and finally stores the result and type in their place at the top of the Stack.

It should perhaps be noted here that the relational operations in MUSSEL can be applied to string expressions as well as arithmetic expressions (see string section).

Example:

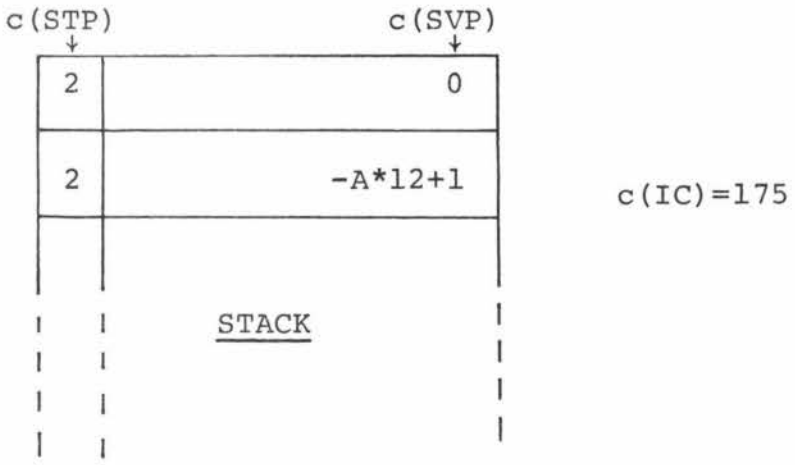
Source code

```
-A*12+1 .GE. 0  .OR.  STRING .EQ. !ABC!
```

Intermediate Language

Digit	OP	Parameter
...	..	...
142	TR	A
147	OP	'NEG'
151	TN	'12'
162	OP	'*'
166	TN1	
168	OP	'+'
172	TN0	
174	OP	'GE'
178	TR	STRING
183	TS	'ABC'
193	OP	'EQ'
197	OP	'OR'
201	..	...
...		

A 'snapshot' of the Stack immediately before the execution of the instruction 'OP 'GE'' would display the following situation:



3.5. Assignment Instructions

In the case of an assignment instruction, the Translator produces object code to first assemble the address (or addresses) of the variable (or variables) to be reset at the top of the Stack. To achieve this (for main variables), a 'take address' operator of the following form is used:

TA (Take Address) $\bar{0}\ 2\ \bar{x}\ x\ x$

The parameter here is a 3-digit Symbol Table address. The function of the TA routine is simply to convert this S. T. address to an absolute core address which is then placed, together with an 'address' type digit, at the top of the Stack.

The Translator follows this by object code to place the assigned value and type at the top of the Stack. One or more of the following store operators then follows:

ST (Store) $\bar{1}\ 2$

The function of this routine is to store the top Stack type and value at the address contained in the next Stack position. The top 2 Stack units are then wiped.

It should be mentioned here that if in fact the addressed location contains a string descriptor, then the corresponding string nodes must first be returned to the AVAIL list before storage takes place (see string section).

STA (Store Also) $\bar{1}\ 3$

This operator performs storage as for the ST operator. In this case, however, the assigned value and type is then moved down one position in the Stack to replace the stored-into address. This unit now becomes the top of the Stack.

The STA operator is used in the translation of multiple assignment instructions.

Example:

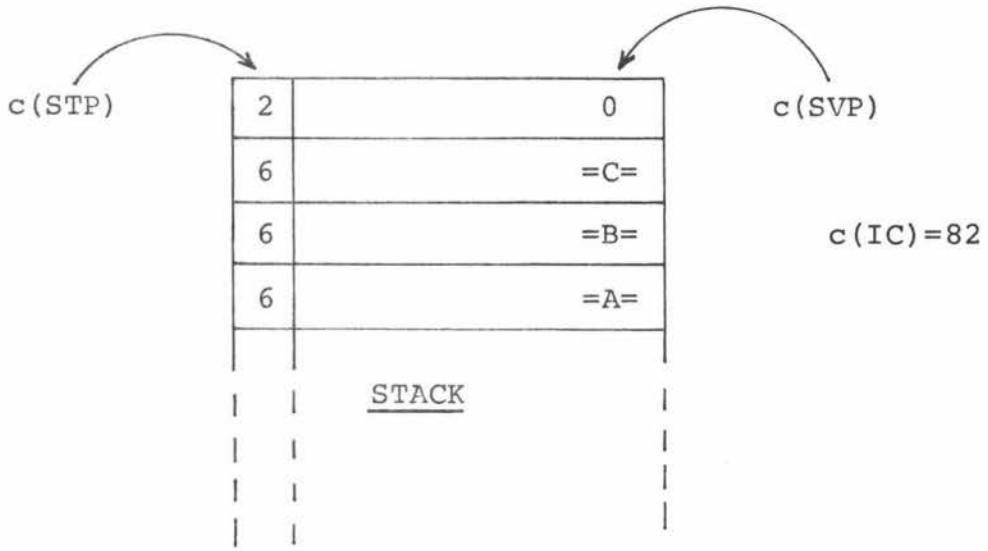
Source code

```
SET  A,B,C  TO  0
```

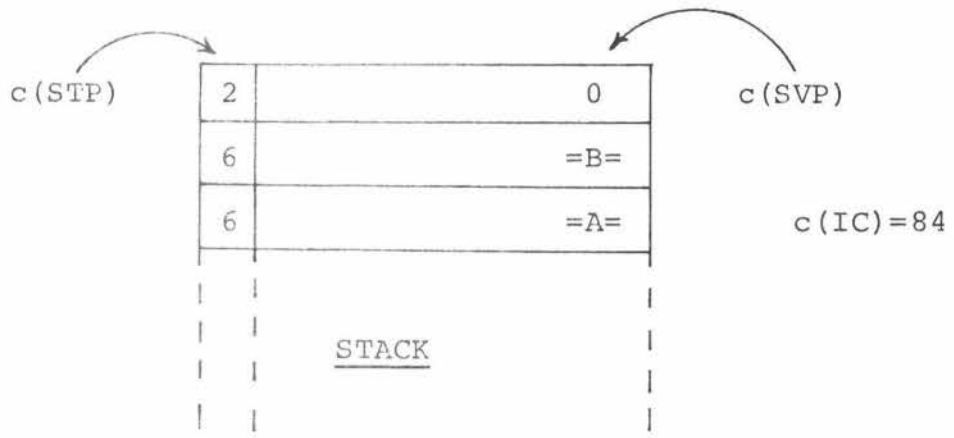
Intermediate Language

Digit	OP	Parameter
..	..	...
64	TA	A
69	TA	B
74	TA	C
79	TN0	
81	STA	
83	STA	
85	ST	
87	..	
..		

A 'snapshot' of the Stack immediately before the first store operation would display the following situation:



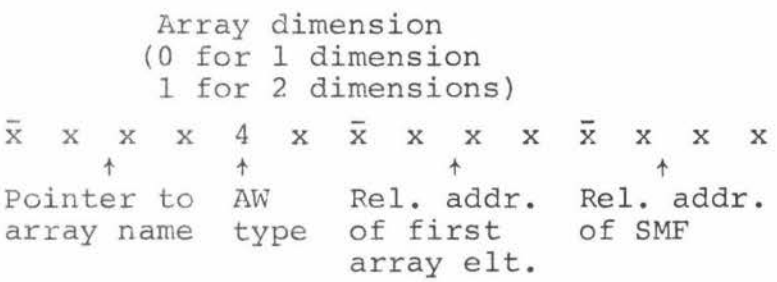
Just before the second store operation the situation would be as follows:



3.6. Main Arrays and Subscripted Variables

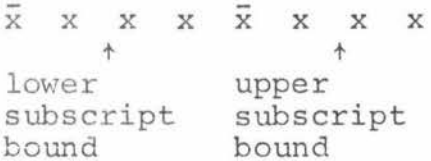
Although MUSSEL does provide for dynamic arrays (see procedures section), main arrays, like outer block arrays in ALGOL, must have constant bounds, and are reserved at the head of a program. This enables the system, during the translation phase, to set aside storage and assemble storage mapping functions for all main arrays. The space reserved is located immediately following the Symbol Table (see storage map), and is sufficient to contain the individual storage mapping functions together with the array elements themselves, stored row-wise. Each such element is similar in form to a Stack unit i.e. it contains a value part of 9 digits accompanied by a type part of 1 digit. In this way, not all elements of an array need necessarily be of the same type.

In the object program, all references to a main array refer to the appropriate array word, which is stored in the Symbol Table, and is of the following form:

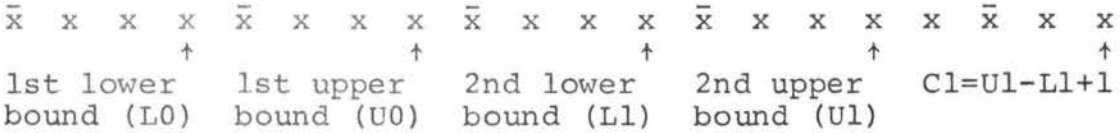


Each of the 2 relative addresses in the value part of the AW must be added to the Symbol Table base address (c(SYMBAS)) to give an absolute address.

Storage management for arrays in MUSSEL is greatly simplified by the restriction that they be 1- or 2-dimensional only. The storage mapping function for a 1-dimensional array is of the following form:



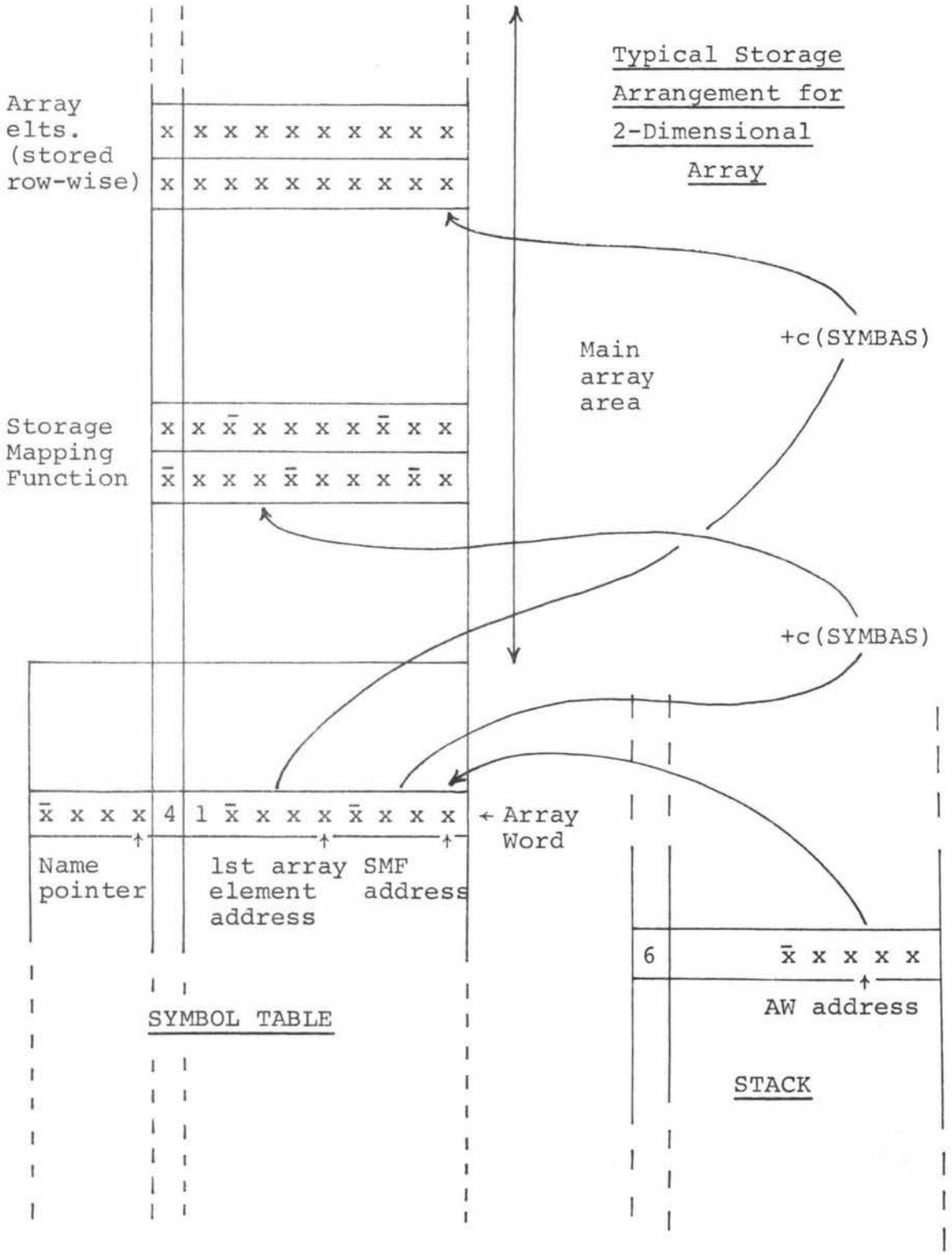
For a 2-dimensional array, the SMF is of the following form:



From the above, it can be seen that array bounds must be less than 10,000 in magnitude. An added restriction, again imposed by the limited amount of core available in the 1620, is that the size of an array, whether 1- or 2-dimensional, must be less than 1,000. In fact, the use of an array anywhere near this size would probably be disastrous in this implementation of MUSSEL. For example, an array consisting of 999

elements would gobble up almost 10,000 digits of core, leaving a mere 200 digits available for the Intermediate Language, Stack, and string area !

The storage arrangement for the main arrays, then, can be illustrated by the following diagram:



It should be noted that arrays reserved in the same segment, for example

```
RESERVE  (A,B,C(1:100))
```

in fact reference the same storage mapping function.

The method of representation of subscripted variables in the I.L. in MUSSEL is very similar to that employed by Randell & Russell. An 'indexing routine' uses the array word and storage mapping function, together with the values of the actual subscripts, to locate the element of the array to which the subscripted variable refers.

Thus, to obtain the address of an array element, the I.L. must assemble, as data for the indexing routine, the address of the array word, and the value of each subscript expression. This is done by using a TA operator to stack the address of the Symbol Table location which contains the array word, followed by the normal object program representation of each subscript expression. The operation INDA (Index Address) then follows. This routine works its way down through the Stack, beginning at the top, identifying the subscript values as numbers, keeping a count of their number, and checking that their magnitudes are in fact less than 10,000. An initial restriction, also, is that the subscript values be integers. Later the routine should be extended to cater for the rounding of real subscripts.

This process terminates when a Stack unit containing an address is reached. This address enables the array word, and hence the storage mapping function, to be located. (Note - a check is here made that the number of subscripts does in fact agree with the dimension of the array, as indicated in the array word - if not, an appropriate error message is printed

out.) The routine is then able to process the array word, storage mapping function, and stacked subscripts to obtain the address of the array element. During the process, a check is made to ensure that the subscript values do in fact lie between their prescribed bounds, which have been stored in the SMF.

Finally, the array element address is placed in the Stack unit which contained the address of the array word, and the Stack pointers decreased so as to make this now the top Stack unit.

Thus the total effect of the set of operations is just the same as a TA operation for a simple variable.

To produce the equivalent of a TR operation for a simple variable, the operation INDR (Index Result) is used. This works in the same way as the INDA routine, except that it finishes by fetching the actual array element type and value.

Example:

Source code

```
SET  A(I,J*K)  TO  B(C(L,M))
```

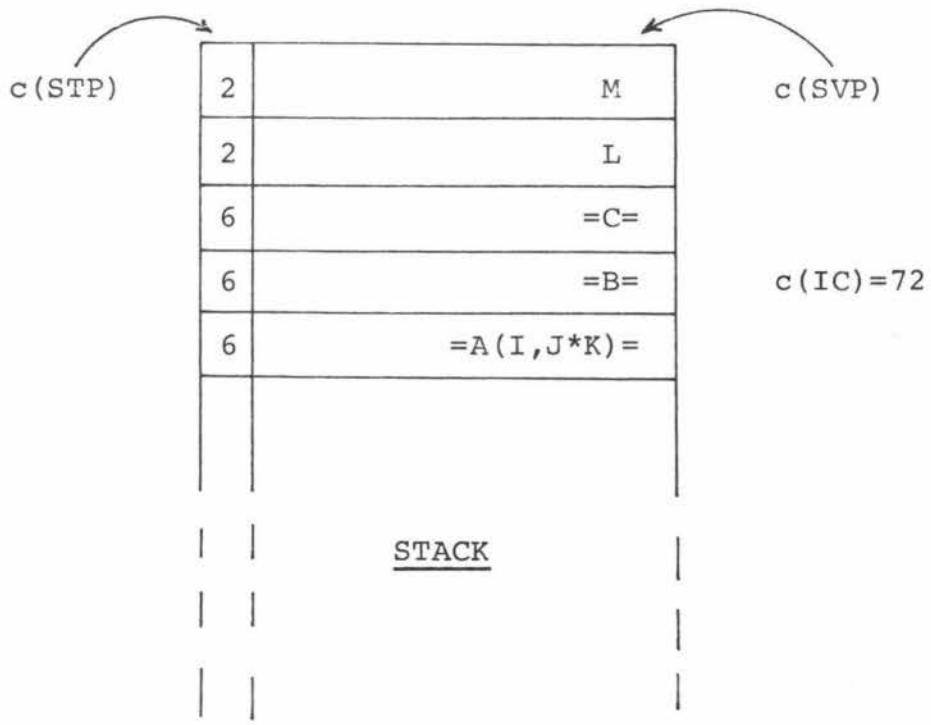
Intermediate Language

Digit	OP	Parameter
...	..	..
23	TA	A
28	TR	I
33	TR	J

(cont...)

Digit	OP	Parameter
38	TR	K
43	OP	'*'
47	INDA	
49	TA	B
54	TA	C
59	TR	L
64	TR	M
69	INDR	
71	INDR	
73	ST	
75	...	
..		

The Stack situation immediately before execution of the second INDR operation would be as follows:



3.7. Number Representation and Arithmetic

As indicated earlier, all numbers are represented internally by MUSSEL as a 7-digit mantissa accompanied by a 2-digit exponent. A minus sign, for the mantissa or exponent, is indicated by a flag over the low-order digit, the convention with all fields in the 1620. The reason for reducing the mantissa length from the customary 8 digits to 7 was to enable a number, together with its type, to be contained in a Stack unit of 10 digits. It was felt that this small loss in significance was justified in a student language.

The important feature, however, of the MUSSEL representation is that the implied decimal point for the mantissa is to the right. In this way, integers may be treated essentially as a subset of the reals, being distinguished internally by a zero exponent. The range of integers, therefore, is from -9999999 to +9999999, while reals may range from ±E-99 to ±9999999E99. The advantage with this representation, of course, is that it removes for the student the real - integer type conversion problem present in other languages, particularly the mixed-mode problem of FORTRAN.

The adoption of this convention in MUSSEL has meant the creation of the system's own independent arithmetic routines. These routines operate on integers as a special case, thereby bypassing the more involved processing required for real operands. This is to say that the routines ADD, SUBTRACT, MULTIPLY, and INTEGER DIVIDE, with integer operands, produce integer results, except in the case when the result overflows in 'integer format', and must therefore be converted to 'real format'.

Real numbers in MUSSEL are normalised in the sense that their mantissas are shifted right in order to remove low-order zeros. Zero itself is represented by a zero exponent.

Also, on the detection of overflow (i.e. when a number's exponent becomes greater than 99), the MUSINT rounding and normalising routine prints an appropriate warning message, and then returns the largest number (in magnitude) that it can accommodate - a mantissa (together with its particular sign) and exponent of 9's. And on the detection of underflow (i.e. when the exponent of a number becomes less than -99), the routine, having printed a warning message, continues in a similar forgiving manner by returning a result of zero (i.e. zero mantissa and exponent).

3.8. Logical Operations

A Boolean variable in MUSSEL is represented internally by a unit containing the type digit '3', together with a value part of '00' for the value FALSE, or '01' for the value TRUE.

Three Boolean operators are available in the language, the MUSINT routines for which are quite straightforward. The AND and OR routines work on the top 2 Stack units, checking that they are of type 'Boolean', and if so replacing them by the resulting Boolean type and value. The unary operation NOT merely confirms that the top type is 'Boolean', and then changes the top value from TRUE to FALSE, or vice-versa.

The relational operations LT, LE, EQ, NE, GE, and GT work on the top 2 units, which may be either numbers or strings (see string section). If the relation is satisfied, they replace the units by a Boolean type unit with value TRUE, otherwise with the value FALSE.

3.9. String Storage and Manipulation

Three basic methods of storing and processing character strings in MUSSEL were considered. They are described briefly as follows:

Method 1 (Sequential Allocation)

Using this method, strings are simply layed out sequentially in storage, character by character. To delete a section of a string, all characters to the right of the section to be deleted are moved left a number of places corresponding to the number of characters to be deleted. To add to a string, all the characters following the point of insertion are moved right the correct number of places, and the new characters inserted.

Method 2 (Dynamic Storage Allocation)

Here again, each string is layed out sequentially in storage in an unbroken fashion. The difference here, however, is that the base of a string remains fixed during processing. For instance, to delete a section of a string, the affected characters of that string only move left, thereby freeing an area of storage before the next string. Conversely, to add to a string which has insufficient free space to its right to accommodate the addition, the string must first be moved to an available area of storage which can accommodate the increased length. Here the extra characters can be inserted.

Method 3 (Linked Allocation)

Under this system, string characters are packed into fixed length nodes (or units), each node containing a link to the next. The terminal string node is indicated by a negative link pointer. To insert characters into a string,

a new node must be obtained from the available storage area, and the appropriate node link fields set up. The method is complicated by the fact that a certain amount of repacking must in general take place in order to keep the non-terminal node character fields full. Similarly, to delete a section of a string, a certain amount of link changing and repacking must take place.

What was wanted in MUSSEL was essentially a simple method with minimum storage requirements - speed was not a crucial factor.

As far as actual storage requirements are concerned, method 1 is clearly the most efficient. Here the "packing density", which is the percentage of storage containing character information, is in fact 100 percent. Yet although scanning a string is very easy, the insertion and deletion of characters has been found[1] to be not only very slow, but also quite involved, due mainly to the need to update the affected string descriptors each time.

Initially in method 2, all the characters will be stored linearly in memory, resulting in a packing density of 100 percent. As processing continues, however, the packing density is likely to be somewhat reduced. And although the use of a "garbage collector" to reorganise data periodically can keep the density as close to 100 percent as desired, it is very likely that this compacting routine will be quite lengthy and involved, and could also be required quite often, particularly in a small machine.

---

[1]Madnick, Stuart E. "String Processing Techniques." CACM Vol. 10, No. 7 (July 1967).

The operations of string assignment and substring selection, under method 2, can be confined to operations on string descriptors, and would therefore involve no change to the string area itself (as in the XCOM compiler[1]). On the other hand, however, the operations of concatenation and input, which use previously free space, could be quite complicated, depending on the particular storage management procedure being used.

Method 3, of course, implies a much less efficient use of storage than method 1. And while the packing density depends on the node size, it does remain constant, a disadvantage when compared with the potentially space saving method 2. The big advantage, however, with the system is that storage management is extremely simple. In fact, the storage lost here in actual string storage would more than likely be made up for by the space saved by such a simple storage management routine, compared with those required for methods 1 and 2. Also, scanning a string is quite easy, and although insertion and deletion of string sections may in general be difficult, the same processes applied to the ends of strings, which is mainly what MUSSEL allows for, is not so bad.

For these reasons, method 3 was finally adopted.

Now came the question of determining the actual node length. In doing so, there were 3 factors to consider:

1. The larger the node size, then the greater would be the packing density (assuming full nodes).

---

[1]McKeeman, Horning, & Wortman : A Compiler Generator (Englewood Cliffs, N.J. : Prentice-Hall Inc., 1970).

2. And the larger the node, then the less strings would be segmented. This would imply less work, in general, in manipulating strings.

3. However, the larger the node size, then the more likelihood there would be of core wastage due to only partially filled terminal node 'info' fields.

In other words, I required as large a node size as possible with as little storage wastage as possible.

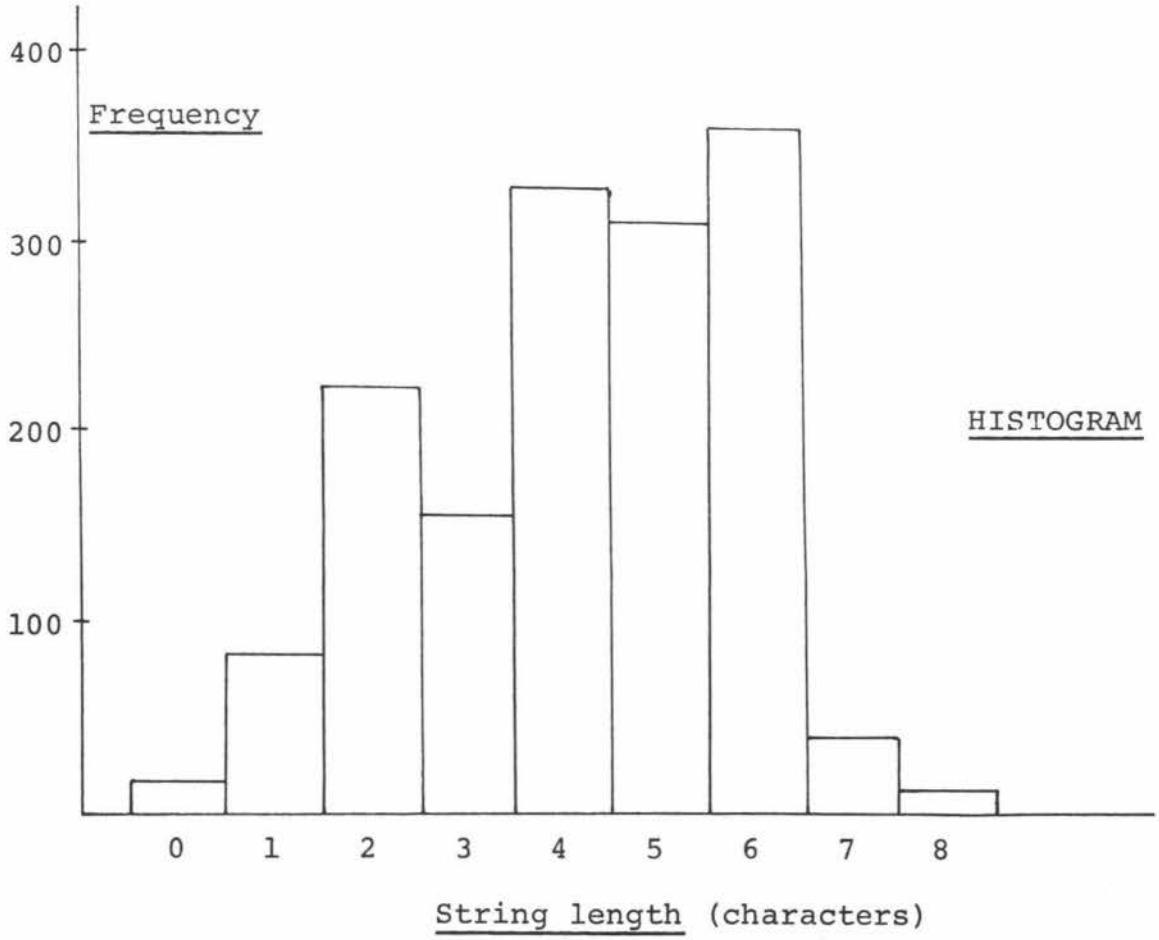
To help decide on an optimum node size, a number of SNOBOL programs were analysed. Although it could be argued that this was indeed a small sample from a small population, it did give me at least some basis on which to make a decision.

Each of the programs was stopped at random at some point during execution, and the lengths of the strings in use at the time recorded. The chart on the following page displays the results of this survey.

For this particular sample of strings the storage wastage for nodes of different lengths was then calculated. (Note - a link field of 4 digits was assumed here.) For example, for a node value field of 4 digits (i.e. 2 characters) the wastage would be as shown in the table on the following page.

String length (characters)	Frequency
0	19
1	85
2	227
3	157
4	330
5	312
6	362
7	40
8	13
9	1
15	1
TOTAL	1547

ANALYSIS OF CHARACTER STRINGS IN SNOBOL



HISTOGRAM

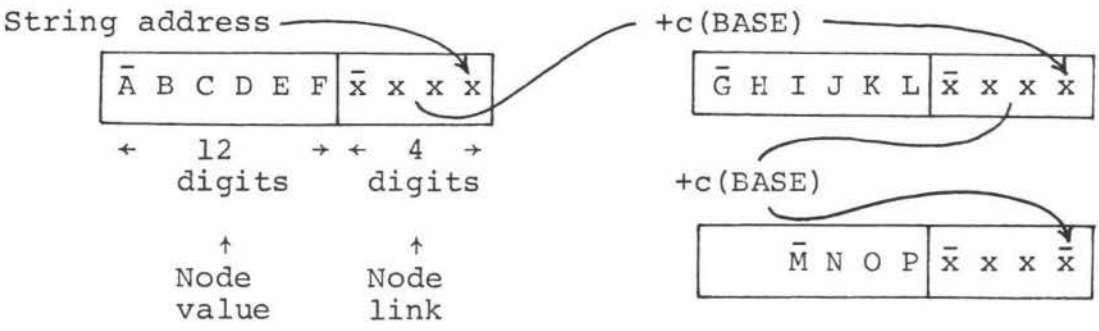
String length (characters)	Number of strings	Core wastage (in digits)
1	85	85×(4+2) = 510
2	227	227×4 = 908
3	157	157×(8+2) = 1570
4	330	330×8 = 2640
5	312	312×(12+2)= 4368
6	362	362×12 = 4344
7	40	40×(16+2)= 720
8	13	13×16 = 208
9	1	1×(20+2)= 22
15	1	1×(32+2)= 34
		TOTAL 15324

TABLE SHOWING CALCULATION OF CORE WASTAGE WITH NODE VALUE FIELD SIZE OF 4 DIGITS

A complete table of results of similar calculations for various other node sizes is as follows:

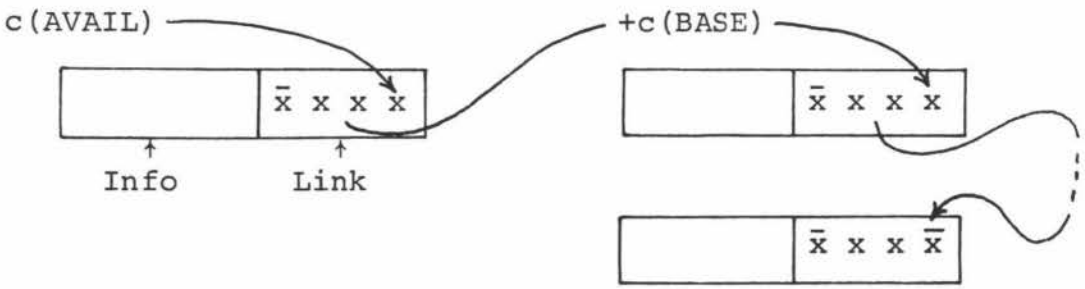
Node value field length (characters)	Storage wastage (digits)
1	25880
2	15324
3	13500
4	14180
5	14304
6	12404
7	12968
8	17660

From the above table, and the fact that only a small proportion of the strings analysed had a length greater than 6 (3.6 percent in fact), a node value field of 6 characters, or 12 digits, was decided on. As indicated earlier, the link field for each node is a 4-digit pointer, relative to some base (30,000). The implication here, of course, is that the string area is limited to 10,000 digits in size. Strings are therefore packed in the following manner:



The string area itself grows downwards from the top end of core in the 1620 towards the Stack. A pointer, c(BOUND), indicates the string area boundary at any stage. A list of available storage is maintained, the first node of which is pointed to by c(AVAIL). An empty AVAIL list is indicated by a negative value for c(AVAIL).

The AVAIL list would therefore appear as follows:

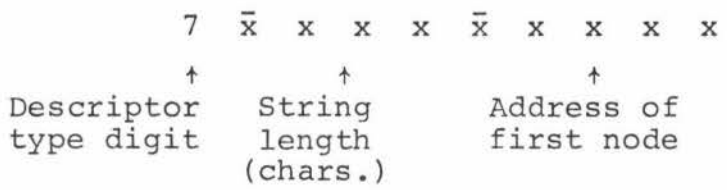


The subroutines for returning to (RETURN) and obtaining from (GRAB) the AVAIL list are quite straightforward.

In the latter, if a node is required, and the AVAIL list is empty, then c(BOUND) must be decreased to make a node available. Each time this is done, c(BOUND) must be checked against the top of the Stack, addressed by c(SVP), for the occurrence of overflow. Initially, c(BOUND) is set to 39,999, and c(AVAIL) to $0000\bar{0}$.

Operations on Strings

A string is represented in the Stack and Symbol Table by a descriptor of the following form:



A null string is simply represented by a unit with a type digit of '1'.

As a consequence of this remote representation, when a TR operator is used with a (non-null) string, the string must first be copied in the available area, and then a descriptor of the copy placed at the top of the Stack. And when a variable containing a string descriptor is reset, the corresponding string nodes must first be returned to the AVAIL list before storage takes place.

The TS (Take String) operator has a parameter which is the string itself in alphameric mode, terminated by an alphameric record mark. The function of this routine is to pack the string away in units of 6 characters, and then place a corresponding string descriptor, and type, at the top of the Stack.

The operator CAT works on the top 2 units of the Stack. After checking to see that these units do in fact contain string descriptors and/or null strings, it then proceeds to replace them by the concatenated result. The operation is simple if either of the strings is null. For 2 non-null strings, however, the routine must link the terminal node of the 'bottom' string (i.e. first operand) to the first node of the 'top' one (i.e. second operand), and reset the length field of the bottom descriptor to the sum of the 2 string lengths. Then, if the terminal node of the bottom string is not full, it must repack this node and the nodes of the top string. Finally, the top Stack unit is wiped.

As mentioned earlier, the relational operators LT, LE, EQ, NE, GE, and GT may be applied to strings as well as to numbers. The convention here is that characters are ordered by their alphameric coding. This defines a partial ordering on the set of strings if strings are compared character by character, beginning from the left. In fact, the ordering obtained is consistent with an alphabetic sort for strings containing only alphabetic characters.

The operation of the routines themselves is first to check for the occurrence of a null string in either of the operands. If so, a Boolean value can be immediately returned, since the null string is the greatest lower bound for the partially ordered set of strings. Otherwise, the lengths of the 2 strings are compared. If they are not equal, and the operation is EQ or NE, then an appropriate Boolean value can be returned. Otherwise, the 2 strings must be compared node by node until a difference is detected, or until either of the strings terminate. Then, depending on the OP and the lengths of the strings, an appropriate Boolean value and type can replace the top 2 Stack units.

Other common string operations can be performed in MUSSEL by the use of subroutines (see appendix).

3.10. The Line Number operator

This operator has been incorporated into the Intermediate language with the sole purpose of enabling the Interpreter to give meaningful error diagnostics at execution time. The operator is of the following form:

LN (Line Number) $\quad \bar{0}\ 1\ \bar{x}\ x\ x$

In general, the Translator inserts an LN operator at the start of the object code corresponding to each line of the source code. The 3-digit parameter of this operator is the appropriate line number.

The function of the MUSINT line number routine is simply to update a line counter (c(LN)) with the value of this parameter. Then, in the event of an execution-time error, the Interpreter will be able to quote the number of the line in the source program in which the error occurred.

An alternative method, whereby the program is divided in terms of its structure (i.e. by DO/END groups) rather than by lines, was also considered. It was felt, however, that, although the method provides a more natural division, the main disadvantage is that errors cannot be pinpointed closely enough, particularly with large groups of instructions.

3.11. General Choice-Groups, Conditional Groups, & If-Groups

A conditional group uses the truth values of Boolean expressions to choose between various instructions. The object program representation of instructions consists of sets of operations which are executed strictly in sequence. Two operations, IFJ (If False Jump) and UJ (Unconditional Jump), are used to break the normal sequencing of operations

in order to choose the set of operations corresponding to the required instruction. Each of these operators occupies 2 digits, and has a 4-digit parameter which is a pointer to the desired set of object code, relative to the base address of the I.L.

UJ $\bar{1}\ 6\ \bar{x}\ x\ x\ x$

IFJ $\bar{1}\ 8\ \bar{x}\ x\ x\ x$

The implication here, of course, is that the I.L. is ultimately limited to being less than 10,000 digits in length. However, it is extremely likely that an I.L. anywhere near this size would, in this implemention of MUSSEL, prove disastrous!

It should also be pointed out here that the DO's and END's appearing in the source program are essentially brackets, and do not therefore have any direct equivalent in the object program.

The function of the UJ routine is simply to add the value of its parameter to the I.L. base address (c(ILBASE)), and then reset the instruction counter (c(IC)) to this new value.

The function of the IFJ routine is to first check that the top Stack unit is of type Boolean. If not, an appropriate error message is printed out. Otherwise, the routine then checks the corresponding Boolean value. If TRUE, the instruction counter is set to point to the next I.L. instruction, and the top Stack unit 'popped'. If FALSE, the instruction counter is reset to the value of the parameter added to the I.L. base address (as with the UJ operator), and the top Stack unit then 'popped'.

It should be noted here that the parameter for each of the above 2 operators points to the leftmost digit of the appropriate opcode in the I.L. Thus the RESET subroutine used by the 2 routines in fact resets the instruction counter to point to this digit, minus one. This means that, after branching back to the control routine, MUSINT can increment c(IC) by 2, as is usual, so that it now points to the rightmost digit of the new opcode.

Example:

Source code

(Line no.)

```
5     DO  IF  B
6           THEN  SET  A  TO  1
7           ELSE  SET  A  TO  0
8     END
```

Intermediate Language

Digit	OP	Parameter
..	..	...
20	LN	'5'
25	TR	B
30	IFJ	'56'
36	LN	'6'
41	TA	A
46	TN1	
48	ST	
50	UJ	'70'
56	LN	'7'
61	TA	A

(cont...)

Digit	OP	Parameter
66	TNO	
68	ST	
70	...	
...		

3.12. Repeat-Group Instructions

The implementation of these instructions in MUSSEL is very much simplified by the fact that the controlling expressions (if any) for loops are all called by value rather than by name if the loop is regarded as a procedure. This means that the expressions can be evaluated and the results stacked by the Interpreter on entry to the loop so that they will be unaffected by any changes that may occur within the loop.

Before going on to discuss the implementation of each instruction in turn, I should first explain the mechanism employed for branching out of loops. At the beginning of the object code corresponding to a REPEAT instruction, the Translator inserts the branch instruction UJ or BA (see later for explanation) immediately preceded by a UJ operator to initially bypass the instruction on entry to the loop. The parameter for the first mentioned branch instruction is the address of the start of the I.L. immediately following the REPEAT-Group's code. Inside the loop, then, this branch instruction is consequently referenced by all instructions requiring an exit from the loop.

The reason for handling loop exits in this indirect manner, rather than directly, is primarily to facilitate the task of the Translator. After compiling object code

for each loop, the Translator need only fix up the address parameter for one instruction rather than several. Also, under this method the handling of labels associated with REPEAT groups is greatly simplified. For on striking a label in the object program, the Translator can immediately place it in a label table together with the address of the branch out instruction at the head of the loop.

The implementation of each REPEAT instruction will now be discussed in turn.

3.12.1. Simple REPEAT without control

The translation of this instruction from source code is quite straightforward, and involves no new I.L. operators. Because the REPEAT has no controls attached, exit must come from the loop body itself. The resulting object code from such an instruction is best displayed by an example.

Example:

Source code

(Line no.)

```
13      DO REPEAT
14           SET I TO I+1
15           IF I.GT.N
16           THEN EXIT
17      END
```

Intermediate Language

Digit	OP	Parameter
...	...	...
44	LN	'13'
49	UJ	'61'
55	UJ	'126'
61	LN	'14'
66	TA	I
71	TR	I
76	TN1	
78	OP	'+'
82	ST	
84	LN	'15'
89	TR	I
94	TR	N
99	OP	'GT'
103	IFJ	'120'
109	LN	'16'
114	UJ	'55'
120	UJ	'61'
126	...	...
...		

3.12.2. REPEAT-while instruction

The object code produced from this instruction is again quite straightforward, and involves no new I.L. operators. The execution of the loop each time is dependent on the truth value of a Boolean expression which is evaluated at the head of the loop. It should be pointed out that no execution of the loop takes place if the condition is initially FALSE.

Example:

Source code

```
4       DO REPEAT WHILE X .GT. N
5            SET X TO X-1
6            IF B THEN EXIT
7       END
```

Intermediate Language

Digit	OP	Parameter
...	...	...
10	UJ	'22'
16	UJ	'98'
22	LN	'4'
27	TR	X
32	TR	N
37	OP	'GT'
41	IFJ	'16'
47	LN	'5'
52	TA	X
57	TR	X
62	TN1	
64	OP	'-'
68	ST	
70	LN	'6'
75	TR	B
80	IFJ	'92'
86	UJ	'16'
92	UJ	'22'
98	...	...
...		

3.12.3. REPEAT-until instruction

The object code produced from this instruction is similar to that obtained from translation of the REPEAT-while instruction, the only difference being that the Boolean value obtained at the head of the loop each time must be complemented before it is tested (since the I.L. of MUSSEL contains no 'if true jump' operator). This implies that if the controlling condition is initially TRUE, then no execution of the loop takes place.

Example:

Source code

```
 9      DO REPEAT UNTIL X .GT. N
10           SET X TO X+1
11           IF B THEN EXIT
12      END
```

Intermediate Language

Digit	OP	Parameter
...	...	...
21	UJ	'33'
27	UJ	'113'
33	LN	'9'
38	TR	X
43	TR	N
48	OP	'GT'
52	OP	'NOT'
56	IFJ	'27'
62	LN	'10'

(cont...)

Digit	OP	Parameter
67	TA	X
72	TR	X
77	TN1	
79	OP	'+'
83	ST	
85	LN	'11'
90	TR	B
95	IFJ	'107'
101	UJ	'27'
107	UJ	'33'
113	...	...
...		

3.12.4. REPEAT-Times instruction

To control the execution of a REPEAT-Times loop, an RT (REPEAT-Times) instruction of the following form is used:

RT $\bar{2}$ 0 $\bar{x}$ x x x $\bar{x}$ x x

This instruction has 2 parameters. The first of these is the I.L. address of the branch out instruction for the loop; it is to this address that the RT routine must branch on completion of the loop. The second parameter is the source program line number of the start of the REPEAT instruction. This enables the RT routine to update the line number at the start of each execution of the loop.

In translating an instruction of this kind, the Translator first inserts 2 branch instructions, as described earlier, and then compiles object code to assemble the value

and type (a number) of the 'times' expression at the top of the Interpreter Stack. This is followed by an RT instruction, and then the code for the body of the loop. Thus when the Interpreter enters a loop and strikes the loop's RT operator for the first time, it should find a number type unit at the top of the Stack. The value part of this unit is, of course, the number of times the loop is to be executed.

The first function of the routine is to check that the number at the top of the Stack is in fact an integer. If so, it replaces the type digit (2) of the unit by '$\bar{8}$', thereby converting it into an RT control unit. The purpose of this is to indicate to the routine, on subsequent uses, that the particular loop has already been entered. The routine's next job (its first on subsequent traversals of the loop) is to check the actual value of the controlling integer. If it is negative, or zero, the instruction counter is reset to the address of the branch out instruction at the head of the loop. (Note that this implies that such a loop may indeed not be executed at all if the controlling expression is initially negative or zero.) Otherwise, the integer value is decremented by one, the line number reset, and the body of the loop entered.

At the end of the body of the loop, a branch is made back to the RT operator for the loop. At this stage the RT control unit will again be at the top of the Stack (since the REPEAT loop constitutes a complete instruction group), so that the operations described above can again be carried out.

And now to the question of exits from the loop. When this occurs, the RT control unit will be at the top of the Stack - to keep the Stack 'clean', it must be unstacked, a job which cannot be done by the RT routine as exits may come

from within the loop. Hence the need for a special branch out instruction for exits from loops with controlling expressions. This is the BA (Branch Address) operator which has the same format as the UJ operator:

BA $\bar{1}$ 7 $\bar{x}$ x x x

The BA routine first resets the instruction counter (as is done in the UJ routine), and then unstacks the loop control units. In this case only one unit, the RT control unit, need be unstacked.

Example:

Source code

```
10      DO REPEAT N TIMES
11           SET X TO X+1
12      END
```

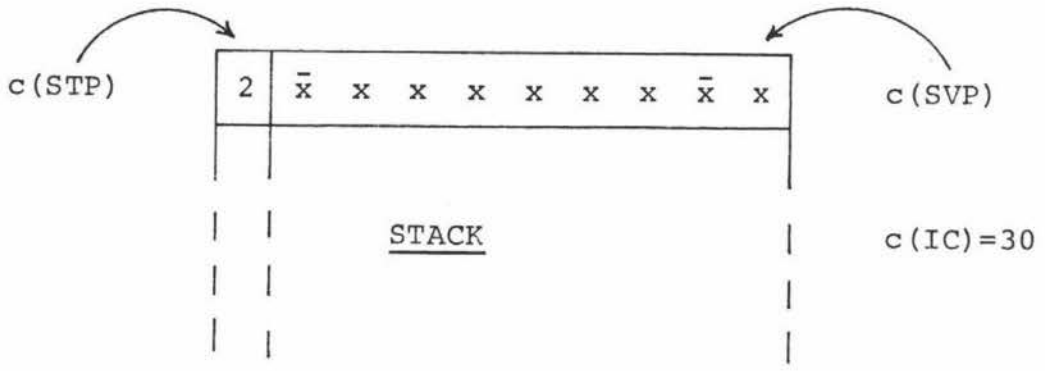
Intermediate Language

Digit	OP	Parameter
...	...	...
7	UJ	'19'
13	BA	'67'
19	LN	'10'
24	TR	N
29	RT	(13,0)
38	LN	'11'
43	TA	X
48	TR	X
53	TN1	
55	OP	'+'

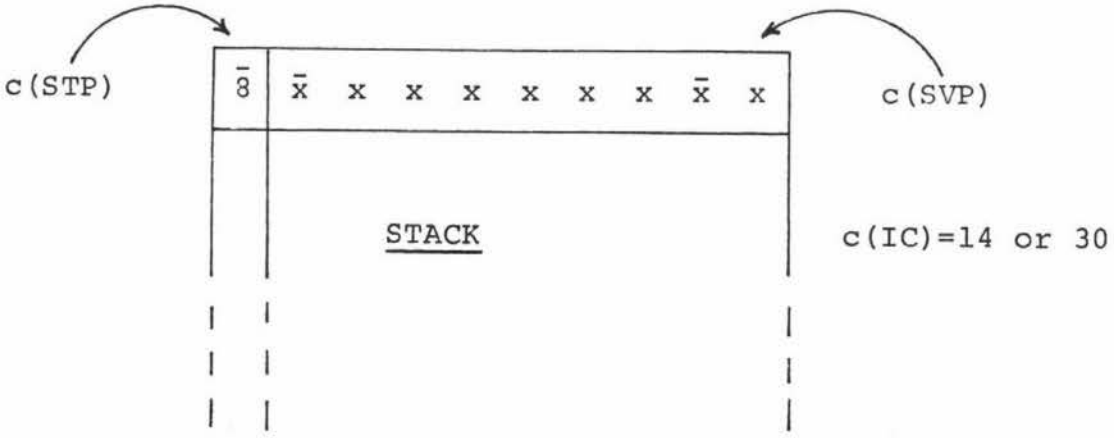
(cont...)

Digit	OP	Parameter
59	ST	
61	UJ	'29'
67	...	...
...		

The Stack situation on first meeting the RT operator on entry to the loop would be as follows:



On meeting the BA operator, or the RT operator after the first execution of the loop, the situation would be as follows:



3.12.5. REPEAT-For instruction group (with test)

This source code instruction group is of the following form:

```
DO REPEAT FOR loc FROM valexp TO valexp (BY valexp)[1]
     ...instruction...
END
```

In translating such an instruction group, the Translator compiles UJ and BA operators (as described earlier) followed by object code to assemble the address of the controlled variable, its starting value, its test value, and its incrementing value (note - if omitted, this value is assumed to be '1', so that a TN1 operator is automatically generated), at the top of the Stack. This is followed by an RF operator of the following form:

RF (REPEAT-For) $\bar{2}\ 1\ \bar{x}\ x\ x\ x\ \bar{x}\ x\ x$

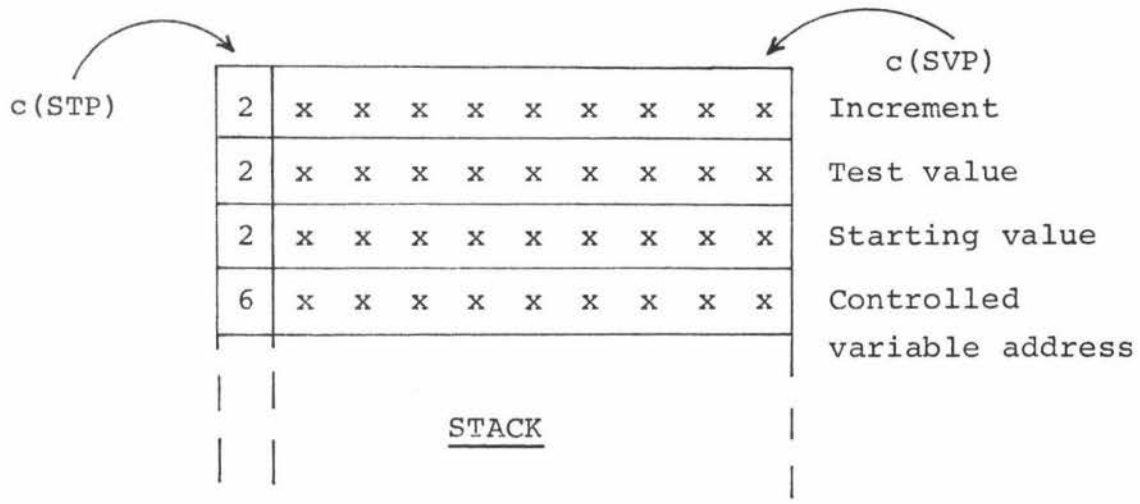
This operator, whose parameters are as for the RT operator, is followed in turn by code for the loop body itself, after which an unconditional branch back to the RT operator is inserted.

Thus, at the interpretive stage, when the loop has initially been entered, the RF routine would expect the top portion of the Stack to appear as shown at the top of the following page.

The first job of the RF routine is to check that the types of the top 4 Stack units are in fact as shown in the diagram (next page). It then initialises the loop variable,

---

[1] In syntax descriptions in this dissertation, material enclosed within round brackets () may be optionally omitted.



inserts a flag over the top Stack type digit to indicate to the routine on subsequent uses that this particular loop has already been entered, and finally returns to control. The implication here, of course, is that such a loop is always executed at least once, even though the test may have failed had it been applied before the first execution.

On subsequent traversals of the loop, the RF routine, on detecting a flagged top Stack type digit, will then proceed to use the stacked values to increment and test the loop variable. If the variable's value is greater than the test value, a branch is made to the BA operator for the loop. Otherwise the next instruction in sequence will be executed. As with the RT operator, the second parameter is used to reset the line number on each execution of the loop.

Here again, the BA operator must unstack the control units for the loop, a job which cannot be done by the RF routine since exits may come from the loop body itself, thereby overriding the loop controls. The operator simply does this by unstacking until an address type unit is reached. This unit is then unstacked, the instruction counter reset, and finally control returned to the main control routine.

Example:

Source code

```
5       DO REPEAT FOR A FROM 1 TO N BY 2
6            SET B(A) TO A*A
7       END
```

Intermediate Language

Digit	OP	Parameter
...	...	...
29	UJ	'41'
35	BA	'117'
41	LN	'5'
46	TA	A
51	TN1	
53	TR	N
58	TN	'2'
69	RF	(35,5)
78	LN	'6'
83	TA	B
88	TR	A
93	INDA	
95	TR	A
100	TR	A
105	OP	'*'
109	ST	
111	UJ	'69'
117	...	...
...		

3.12.6. REPEAT-For instruction group (without test)

This source code instruction group is of the following form:

```
DO REPEAT FOR loc FROM valexp (BY valexp)
     ...instruction...
END
```

Translation of such a group is very similar to that for a REPEAT-For instruction group with test. Instead of an RF operator, however, an RFWT operator of the following form is used:

RFWT (REPEAT-For without test) $\bar{2}\ 2\ \bar{x}\ x\ x$

The single parameter here is the line number of the start of the REPEAT instruction, and is used to update the line number on each execution of the loop. Unlike the RF operator, however, the RFWT operator need have no branch out address parameter, all exits coming from within the loop since the source program instruction provides no test value. As a consequence of this, the interpretive RFWT routine, although similar to the RF routine, is very much simplified by the fact that it need not test the controlled variable on each execution of the loop.

As before, a BA operator is used to unstack the control units on exit from the loop.

Example: Source code

```
 8      DO REPEAT FOR A FROM 1 BY 2
 9           SET B(A) TO A*A
10           IF C THEN EXIT
11      END
```

Intermediate Language

Digit	OP	Parameter
...	...	.....
102	UJ	'114'
108	BA	'203'
114	LN	'8'
119	TA	A
124	TN1	
126	TN	'2'
137	RFWT	'8'
142	LN	'9'
147	TA	B
152	TR	A
157	INDA	
159	TR	A
164	TR	A
169	OP	'*'
173	ST	
175	LN	'10'
180	TR	C
185	IFJ	'197'
191	UJ	'108'
197	UJ	'137'
203	...	.....
...		

3.12.7. REPEAT-Set instruction group

Translation of this group, which is of the following form

```
DO REPEAT FOR loc SET TO exp,exp,exp...
     ...instruction...
END
```

is very similar to that for the previous 3 instruction groups. The Translator compiles UJ and BA operators in the normal way at the head of the loop, followed by object code to assemble the address of the loop variable and the values of the set list elements at the top of the Stack. This is followed by an RS (REPEAT-Set) operator, code for the body of the loop, and finally an unconditional branch back to the RS operator itself.

The form of this RS operator is as follows:

RS (REPEAT-Set) $\bar{2}\ 3\ \bar{x}\ x\ \bar{x}\ x\ x\ x\ \bar{x}\ x\ x$

The operator has 3 parameters. The first of these is the number of set list elements; the second is the I.L. address of the BA operator for the loop; and the third is the source program line number of the start of the REPEAT instruction group. As for previous I.L. operators, this third parameter enables the RS routine to update the line number number at the start of each execution of the loop.

On initial entry to the loop at run time, the RS interpretive routine sets a flag at the top type position of the Stack, and initialises the loop variable. On successive traversals of the loop, the RS routine steps through the set list elements, resetting the loop variable accordingly.

Throughout this process a count of the list elements already processed is kept in the Stack position originally occupied by the first of these elements. Finally, on completion of the loop, the instruction counter is reset to the address of the BA operator for the loop.

It should be mentioned here that in this implementation, the set list elements themselves may be expressions of any type whose values are unaffected, as mentioned earlier, by any changes that may occur within the loop.

Again the BA operator, when used, unstacks the control units for the loop. This is done by unstacking units until an address type unit is met. The routine then pops this unit, resets the instruction counter, and finally returns to control.

Example:

Source code

```
19      DO REPEAT FOR A SET TO 1, 2, 3
20           SET B(A) TO A*A
21      END
```

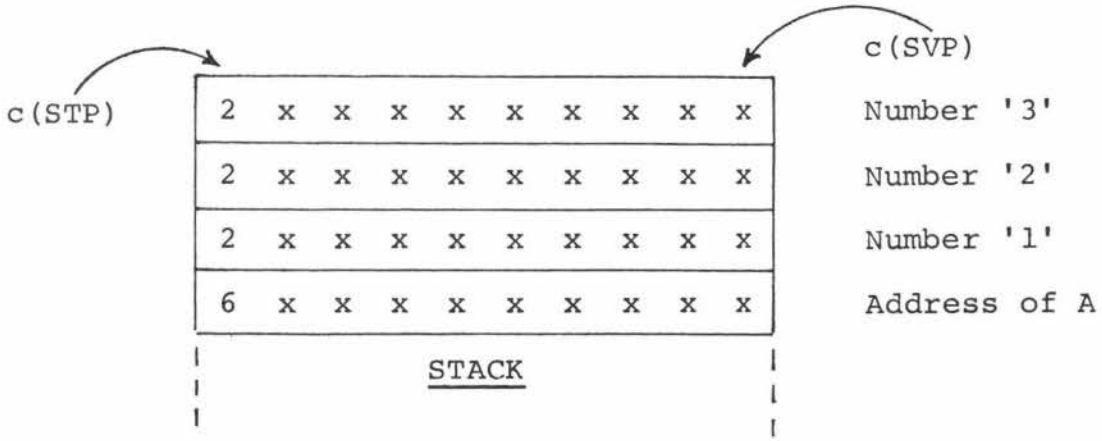
Intermediate Language

Digit	OP	Parameter
...	...	...
65	UJ	'77'
71	BA	'161'
77	LN	'19'
82	TA	A
87	TN1	

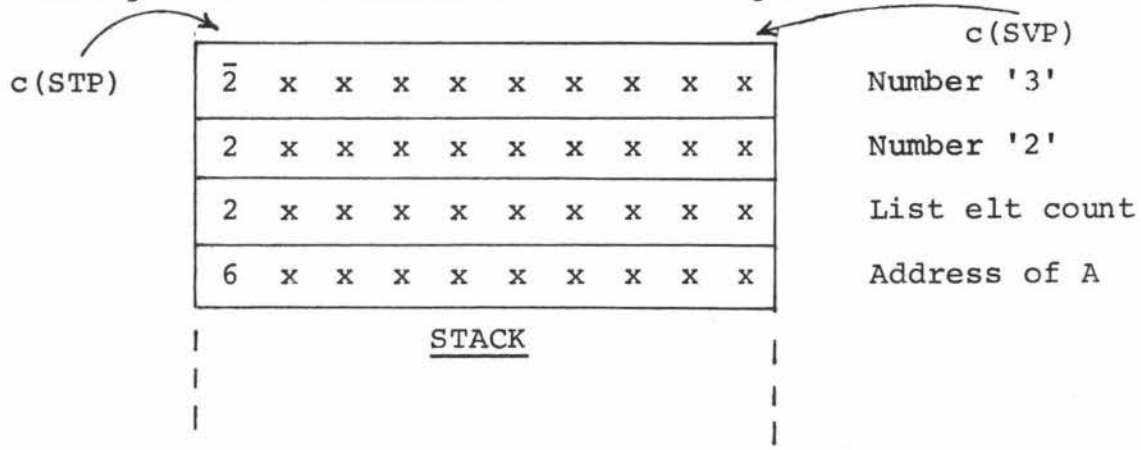
(cont...)

Digit	OP	Parameter
89	TN	'2'
100	TN	'3'
111	RS	(3,71,19)
122	LN	'20'
127	TA	B
132	TR	A
137	INDA	
139	TR	A
144	TR	A
149	OP	'*'
153	ST	
155	UJ	'111'
161	...	...
...		

The Stack, just before execution of the RS operator for the first time, would appear as follows:



And just before execution of the BA operator:



3.13. CASE-Group instruction

This is an instruction group of the following form:

```
DO CASE valexp (IN (integer,integer)) OF
     ...instruction...
END
```

The translation of such a group involves quite a bit of work for the Translator. In the case where the bounds have been inserted, the Translator first compiles object code to assemble the values of the CASE expression and bounds at the top of the Stack. Then comes an unconditional jump instruction to bypass the following object code for each instruction in the CASE-group. The code for each such instruction is terminated by an unconditional jump to the start of the code for the instruction immediately following the CASE-group. Finally, a CS (CASE) operator is inserted, followed by a series of unconditional jumps to the start of the object code for each case instruction.

The form of this CS operator is as follows:

CS (CASE) $\bar{2}\ 4\ \bar{x}\ x$

The single 2-digit parameter here is the number of CASE instructions in the CASE-group.

The form of the object code produced for the CASE-group is best illustrated by an example.

Example:

Source code

```
4       DO CASE I IN (3,5) OF
5            SET A TO 3
6            SET A TO 4
7            SET A TO 5
8       END
```

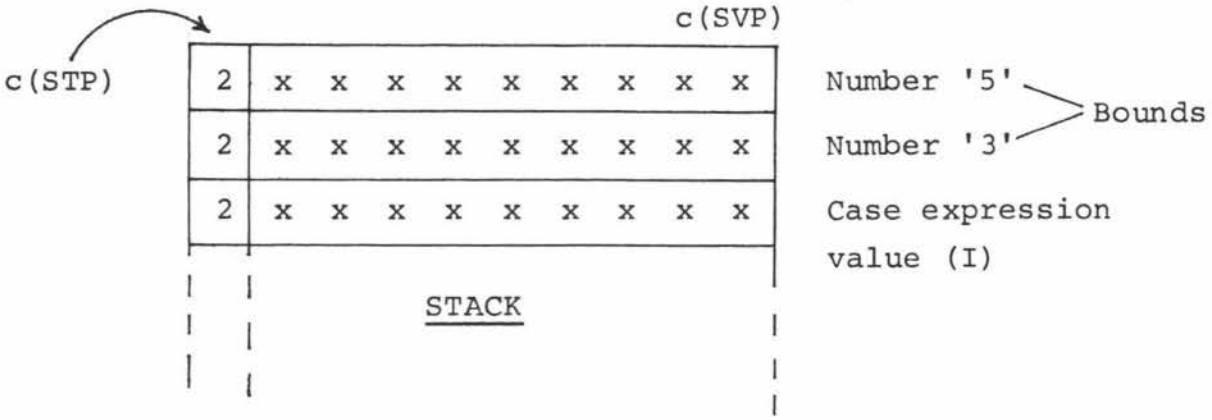
Intermediate Language

Digit	OP	Parameter
...	...	...
23	LN	'4'
28	TR	I
33	TN	'3'
44	TN	'5'
55	UJ	'148'
61	LN	'5'
66	TA	A
71	TN	'3'
82	ST	
84	UJ	'170'
90	LN	'6'
95	TA	A
100	TN	'4'
111	ST	
113	UJ	'170'
119	LN	'7'
124	TA	A
129	TN	'5'
140	ST	

(cont...)

Digit	OP	Parameter
142	UJ	'170'
148	CS	'3'
152	UJ	'61'
158	UJ	'90'
164	UJ	'119'
170	...	...
...		

Immediately before the CS operator is executed during the interpretive phase, the Stack situation would be as follows:



The function of the CS routine at the interpretive stage is to use the stacked expression and bound values to determine which following branch instruction to take. Having done so, and reset the instruction counter accordingly, it unstacks the 3 units and returns to control.

During this routine a check is made that the values of the CASE expression and bounds are all integer numbers (later the rounding of a non-integer CASE expression value could be implemented), and that the expression value does in fact lie between the prescribed bounds. The parameter for the CS operator is also used here to enable a check to be made that

the number of the statement being branched to is not greater than the total number of such statements.

Translation of a CASE instruction in which the bounds have been omitted is very similar to the method described above. The simplification here, of course, is that a lower bound of 1 is assumed, so that the CASE expression value only is used to determine which CASE instruction to take. To perform such a calculation a new operator, of the following form, is used:

CSWB (CASE without bounds) $\bar{2}\ 5\ \bar{x}\ x$

As with the CS operator, the parameter here is the actual number of CASE instructions in the group.

The function of the CSWB interpretive routine is clearly very similar to that of the CS routine. It takes the value of the CASE expression, which has been stacked, and uses it to reset the instruction counter to the address of the appropriate following branch instruction. Having done so, it unstacks the expression value and returns to control.

During the routine, the Interpreter checks that the CASE expression value is in fact an integer (which is not less than 1), and also, by use of the parameter, that the number of the statement branched to is not greater than the total number of statements in the group.

Example:

Source code

```
4       DO CASE I OF
5            SET A TO 1
6            SET A TO 2
7            SET A TO 3
8       END
```

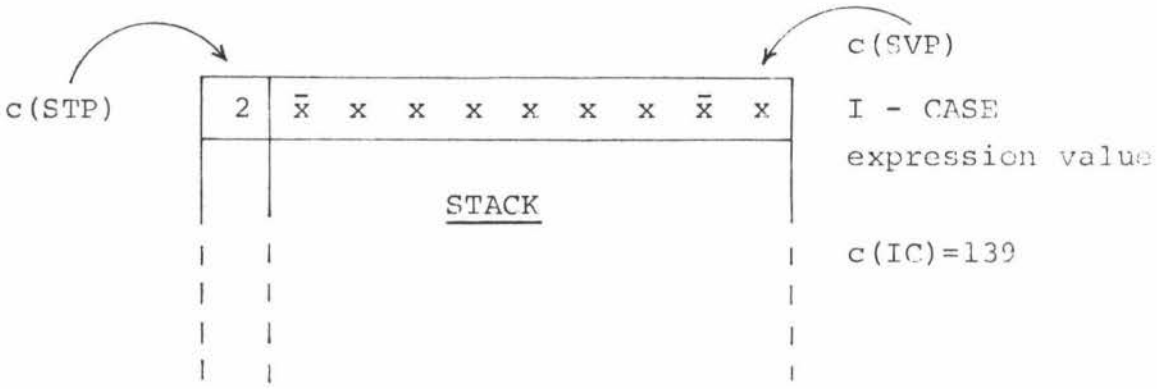
Intermediate Language

Digit	OP	Parameter
...	...	...
35	LN	'4'
40	TR	I
45	UJ	'138'
51	LN	'5'
56	TA	A
61	TN	'1'
72	ST	
74	UJ	'160'
80	LN	'6'
85	TA	A
90	TN	'2'
101	ST	
103	UJ	'160'
109	LN	'7'
114	TA	A
119	TN	'3'
130	ST	
132	UJ	'160'
138	CSWB	'3'

(cont...)

Digit	OP	Parameter
142	UJ	'51'
148	UJ	'80'
154	UJ	'109'
160	...	.....
...		

The Stack situation immediately before the CSWB operator is executed would be as follows:



3.14. Functions and Procedures

3.14.1. An overview of their implementation

The scheme used in the translation and implementation of functions and procedures in MUSSEL is very similar to that used by Randell & Russell, the Interpreter stack being used for the dynamic allocation of storage for local variables during the running of the object program.

As each procedure is entered, its working storage, i.e. the storage space required for its local variables and link data, is added to the top of the Stack. When the procedure is left, this working storage is deleted from the Stack. If a procedure body is re-entered by means of a recursive

procedure call, before it has been left, then a further set of working storage is created on top of the Stack, thereby rendering the original set inaccessible for the duration of the recursive call. In this way, the use of the Stack allows the same system to be used for the allocation of space for working storage of both recursive and non-recursive procedures.

Dynamic arrays are also implemented using the Stack. On entry to a procedure, the expressions in an array declaration are evaluated and the amount of storage space necessary for the array word, the SMF, and the array elements themselves, is reserved on top of the Stack. Thus only the exact amount of space required by an array for the current activation of the procedure in which it is declared is set aside in the Stack.

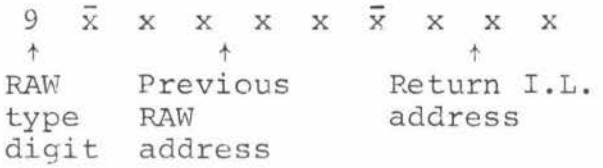
The mechanism required for this implementation in MUSSEL is very much simplified by the fact that procedure declarations in the language may not be nested. Under this restriction, the block structure is essentially 2-level, as opposed to the multi-level structure of ALGOL, so that inside any procedure, the only identifiers (apart from labels) that may be referenced are those which are local to the procedure itself (i.e. formal parameters and variables reserved within the procedure), and those which have been reserved at the head of the main program. As far as the Interpreter is concerned, this means that the creation and maintenance of the 'static chain' of Randell & Russell is unnecessary, and hence that the link data required on leaving each procedure can be simplified a great deal.

As in PL/I, parameters for procedures in MUSSEL are all called by reference. This is essentially a combination of the 'call-by-name' and 'call-by-value' facilities of

ALGOL, and means that, whereas addresses for arrays, procedures, and simple variables are passed accross to the procedure concerned, expressions on the other hand are evaluated first, so that their results only are accessible to the procedure. One of the main advantages for the Interpreter here is that in the case of an array, only the array word address need be copied in the Stack, rather than the array word, SMF, and array elements themselves.

3.14.2. The link data

When a procedure (or function) is called and entered during the execution of the object program, appropriate link data must be set up in the Stack to enable the program to resume correctly on completion of the procedure. This link data in MUSSEL is contained within a single unit, the Return Address Word (RAW), which is of the following form:



The RAW for any activation of a procedure is situated in the Stack immediately below the local variable units (if any) for the particular activation. The address of the topmost RAW at any time is indicated by an RAW pointer, c(PAW), which is initially 0. This pointer effectively forms a base from which the absolute address of a local variable in use during the current procedure activation can be calculated.

The information contained in an RAW enables the Interpreter to perform 2 important functions on completion of the procedure. The first field in the RAW value part allows the RAW pointer to be reset to the address of the previous RAW, thereby effectively creating access to a new set of local

variables. The second field allows the instruction counter to be reset so that execution of the object program can be resumed from the operator immediately following the point at which the procedure was called.

3.14.3. Function and procedure calls and declarations

As mentioned earlier, the only real difference between functions and procedures as they appear in the source program is in their manner of call. This similarity is reflected in the resulting object code, the only difference at call being that in the case of a function, code is produced to reserve an extra space in the Stack for a function value. The pieces of object code produced from procedure and function declarations are, like their source, identical in form.

In translating a procedure or function call, the Translator compiles an RAP or RAF operator respectively, followed by object code to assemble the addresses, or values if constants or proper expressions, of the actual parameters at the top of the Stack. Finally a CP (Call Procedure) operator is inserted. The forms and functions of the new operators just introduced are as follows:

RAP (Return Address Procedure) $\bar{2}\ 6$

The function of this operator is to begin filling in a Return Address Word at the top of the Stack. At this stage an RAW type digit ('9') is filled in together with the address of the previous RAW. Finally, the RAW pointer is updated to point to the low-order digit of this new RAW.

RAF (Return Address Function) $\bar{2}\ 7$

This operator begins filling in an RAW, as for the RAP

operator. In this case, however, the associated interpretive routine reserves a unit in the Stack, immediately below the RAW, for the function value. The type digit for this unit is, at this stage, set to '0' ('undefined').

CP (Call Procedure) 2 8 $\bar{x}$ x x $\bar{x}$ x

The first parameter for this operator is the Symbol Table address of the Procedure Address Word. This word contains a pointer to the start of the object code corresponding to the procedure body. (Note - this pointer in fact addresses the digit to the immediate left of the first operator for the procedure). The second parameter is the number of actual parameters used in the procedure call.

The job of the CP interpretive routine is to first check that the unit contained at the indicated S.T. address is in fact a Procedure Address Word. If so, it then fills in the return I.L. address in the Return Address Word and, using the PAW, resets the instruction counter to the start of the object code corresponding to the procedure body. Finally, it stores the number of actual parameters to be checked later against the number of formal parameters (by the PE operator), and then returns to control.

The following example illustrates the process which has been described:

Example:

Source code

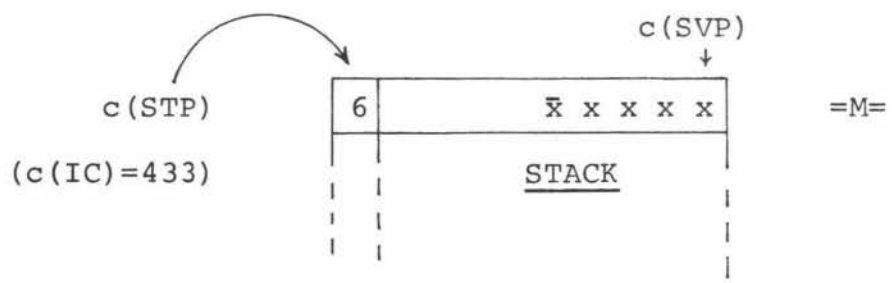
```
12      SET M TO MEAN(U,V*2)
```

Intermediate Language

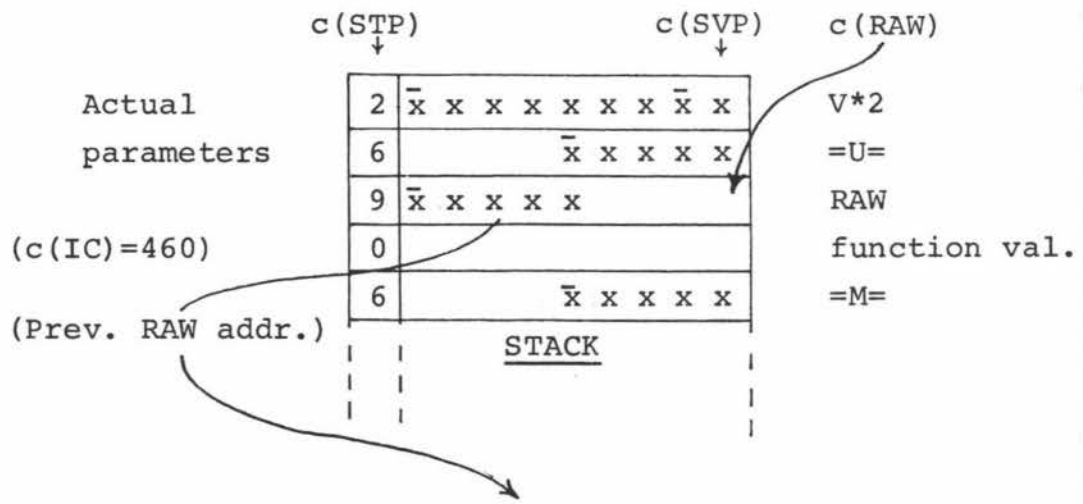
Digit	OP	Parameter
...	...	...
422	LN	'12'
427	TA	M
432	RAF	
434	TA	U
439	TR	V
444	TN	'2'
455	OP	'*'
459	CP	(MEAN,2)
466	...	...
...		

The Stack situation:

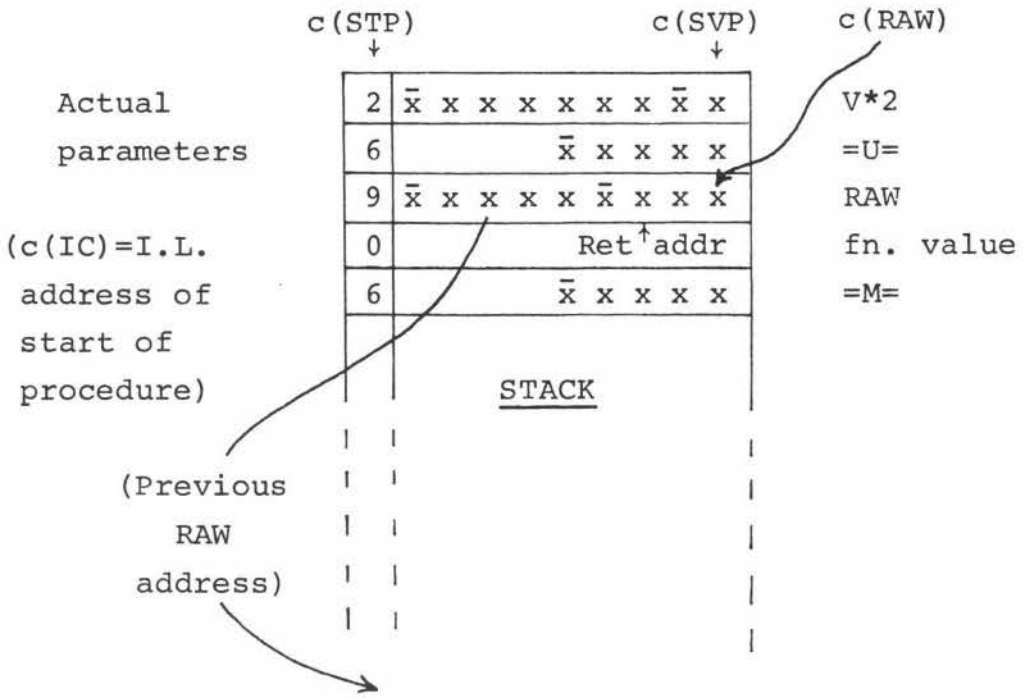
(i) Immediately before execution of the RAF operator:



(ii) Immediately before execution of the CP operator:



(iii) Immediately after execution of the CP operator:



The general layout of the object program corresponding to a procedure (or function) declaration is as follows:

LN	L
PE	(FP,LI)
...	...
RE	

The first operator here indicates the line number for the start of the procedure declaration, and it is to this operator that the Interpreter branches on entry to a procedure. Having updated the line number, the Interpreter then calls on a PE (Procedure Entry) routine whose main function is to set up pointers and allocate working storage for the current procedure activation. The form of this PE operator, as it appears in the I.L., is as follows:

PE (Procedure Entry) $\bar{3}\ 0\ \bar{x}\ x\ \bar{x}\ x$

The first parameter here is the number of formal parameters for the procedure; the second the number of local identifiers. The first job of the PE routine is to check that the number of formal parameters for the procedure does in fact correspond to the number of actual parameters used in the procedure call. If so, the routine sets up base value and type pointers for the procedure (c(BVP) and c(BTP) respectively) to address the RAW value and type parts respectively. These pointers are used to assist in the addressing of the types and values of local variables. The routine then checks the second parameter to determine the number of local identifiers for the procedure. If zero, no further work need be done, and the routine is able to return immediately to control. If non-zero, however, the routine must allocate a corresponding number of units in the Stack for these identifiers, followed by 2 units for the first possible Storage Mapping Function. The top of these 2 units is addressed by the contents of a Working Value Pointer, c(WVP). Having done this, the routine finally branches to a CLEAR subroutine which sets the types of the local variables to '0' ('undefined').

For the storage of a function value, an operator VI (Value Is) of the following form is used:

VI (Value Is) $\overline{1}$ 5

This routine simply stores in the normal way the value and type at the top of the Stack in the reserved unit immediately below the RAW for the function. The top Stack unit is then deleted.

For exits from a procedure (or function) an RE operator (of the form shown below) is used to reset the appropriate variables and pointers.

RE (Return) $\bar{3}$ 4

The RE routine begins by resetting the Stack value and type pointers to address the value and type parts respectively of the unit immediately below the topmost RAW. In other words, this unit now becomes the top of the Stack. Next, the RAW mentioned is used to reset the instruction counter to the return I.L. address for the procedure. Finally, using the high-order field in this RAW, the RAW pointer is chained down to address the next RAW (if any), and the procedure base type and value pointers reset accordingly.

From the point of view of storage allocation, then, the net effect for a procedure call and activation has been a return to the original Stack position. For the case of a function, the net effect has been the return of a type and value to the top of the Stack.

As an example, let us consider a declaration for the procedure used in the previous example. In this declaration, the use of the variable TEMP, although inefficient, has been intentional so as to illustrate the mechanism for allocating space for local variables.

Example:

Source code

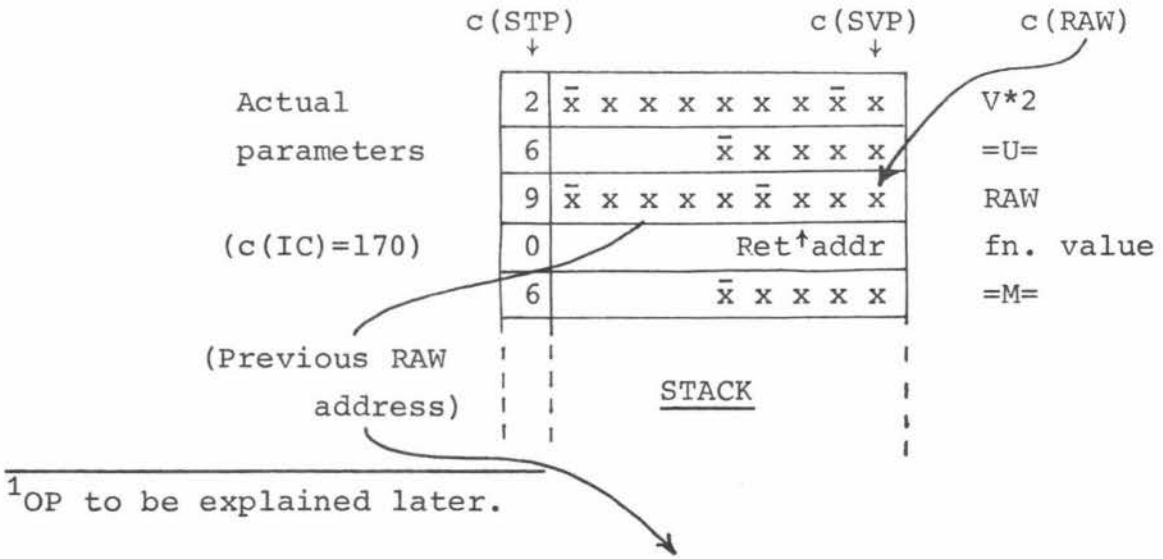
```
3       DEFINE MEAN ON A,B AS
4       DO
5            RESERVE TEMP
6            SET TEMP TO (A+B)/2
7            VALUE IS TEMP
8       END
```

Intermediate Language

Digit	OP	Parameter
...	...	...
164	LN	'3'
169	PE	(2,1)
175	LN	'6'
180	TLA[1]	TEMP
184	TLR[1]	A
188	TLR	B
192	OP	'+'
196	TN	'2'
207	OP	'/'
211	ST	
213	LN	'7'
218	TLR	TEMP
222	VI	
224	RE	
226	...	
...		

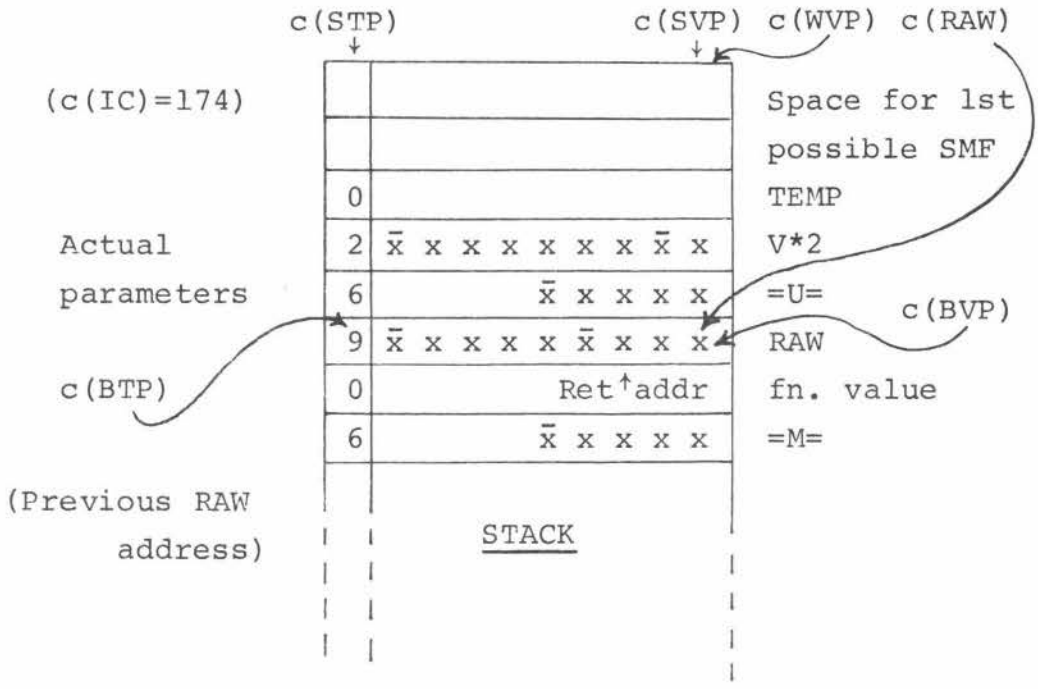
The Stack situation:

(i) Immediately before execution of the PE operator:

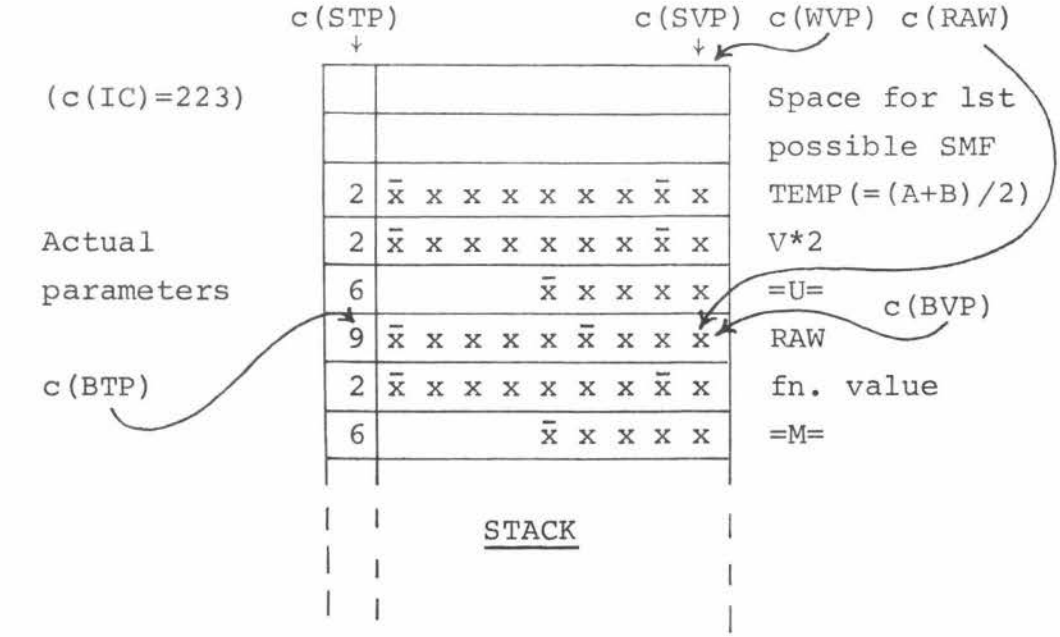


[1]OP to be explained later.

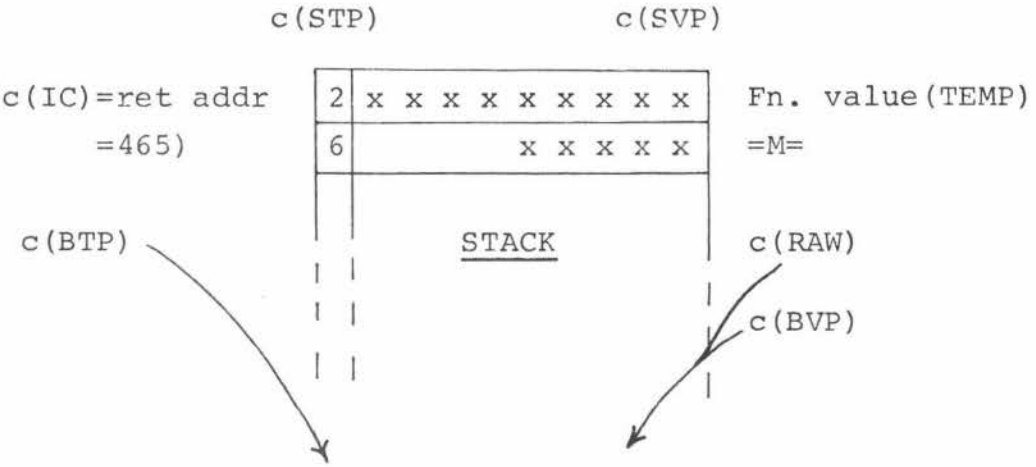
(ii) Immediately after the execution of the PE operator:



(iii) Immediately after execution of the VI operator:



(iv) Immediately after execution of the RE operator:



3.14.4. The use of local variables in expressions and assignments

During the translation of a procedure (or function) declaration, local variables (i.e. formal parameters and identifiers declared within the procedure) are allocated addresses (in sequence) corresponding to their relative positions in the Stack above the Return Address Word at run time. Consequently, the use of these variables in source code expressions or assignments requires the generation of special 'take local address' and 'take local result' operators in the object code. These new operators have the following form:

TLA (Take Local Address)	$\bar{3}\ 1\ \bar{x}\ x$
TLR (Take Local Result)	$\bar{3}\ 2\ \bar{x}\ x$

In each case, the single 2-digit parameter is the relative address of the variable concerned. As for the TA and TR operators described earlier, the function of these operators is to place the absolute address or value, respectively, of the variable concerned at the top of the Stack. In this case, however, the address of the topmost RAW, rather than the base address of the Symbol Table, is used in the calcu-

lation of the variable's absolute address. The situation here is further changed by the fact that, in the case of the variable being a parameter, the 'value' at the addressed location may in fact be the address of the actual parameter which has been passed across to the procedure. In such a case, the TLR routine must 'chain down' to locate the result concerned and 'bubble' it to the top of the Stack. Similarly, to avoid a complicated chaining process when local variables themselves are used as parameters to procedures, the address contained within the location, rather than the address of the location itself, must be placed at the top of the Stack.

3.14.5. Dynamic arrays

Having entered a procedure and allocated space in the Stack for its local identifiers, the Interpreter must then proceed to set up storage mapping functions and allocate storage for the arrays (if any) that are reserved locally to the procedure. The technique used here is again very similar to that used by Randell & Russell.

During the translation phase, each array declaration, simple or multiple, is translated into a set of object program representations of the subscript bound expressions. This is followed by an MSF (Make Storage Function) operator of the following form:

MSF (Make Storage Function) $\bar{3}\ 3\ \bar{x}\ x\ x\ x$

This operator has 3 parameters. The first is the 2-digit dynamic address allocated to the last array word; the second the number of identifiers in the array declaration; and the third the dimension ('0' for 1 dimension, '1' for 2 dimensions) of each array. The task of the MSF interpretive routine is to use the subscript bound values which have been placed at the top of the Stack to set up a storage mapping

function, and an array word, for each array in the declaration, at the same time allocating space in the Stack for the elements of each array. The key steps in this rather lengthy routine are as follows:

Step 1

This step involves the setting up of a SMF for the set of arrays. First, the values of the subscript bounds are loaded into the space indicated by the contents of the Working Value Pointer, c(WVP). Then, for a set of 2-dimensional arrays, the value C1 (the difference between the second set of bounds, plus 1) is calculated and filled into the SMF. During this step a check is made that the bounds are integers which do not exceed the value 9999 in magnitude. A check is also made that the number of elements in each array does not exceed 999.

Step 2

Here the array words themselves are set up in the space previously allocated in the Stack for the associated array identifiers. In conjunction with this process, space is set aside for the array elements themselves.

Step 3

Having allocated space for the current set of arrays, the Interpreter now reserves 2 units at the top of the Stack for another possible Storage Mapping Function. The Working Value Pointer is then reset to address these locations.

Step 4

Finally, the type digits for the units reserved for the array elements are each set to '0' ('undefined').

Note: Array words for dynamic arrays have a flag over the dimension digit. This is to indicate to the Interpreter that, when calculating absolute addresses from the relative addresses

in the array words, the base address for the Stack, rather than the Symbol Table, is to be used.

Example:

Source code

```
RESERVE  (A,B,C(1:N,0:M))
```

Intermediate Language

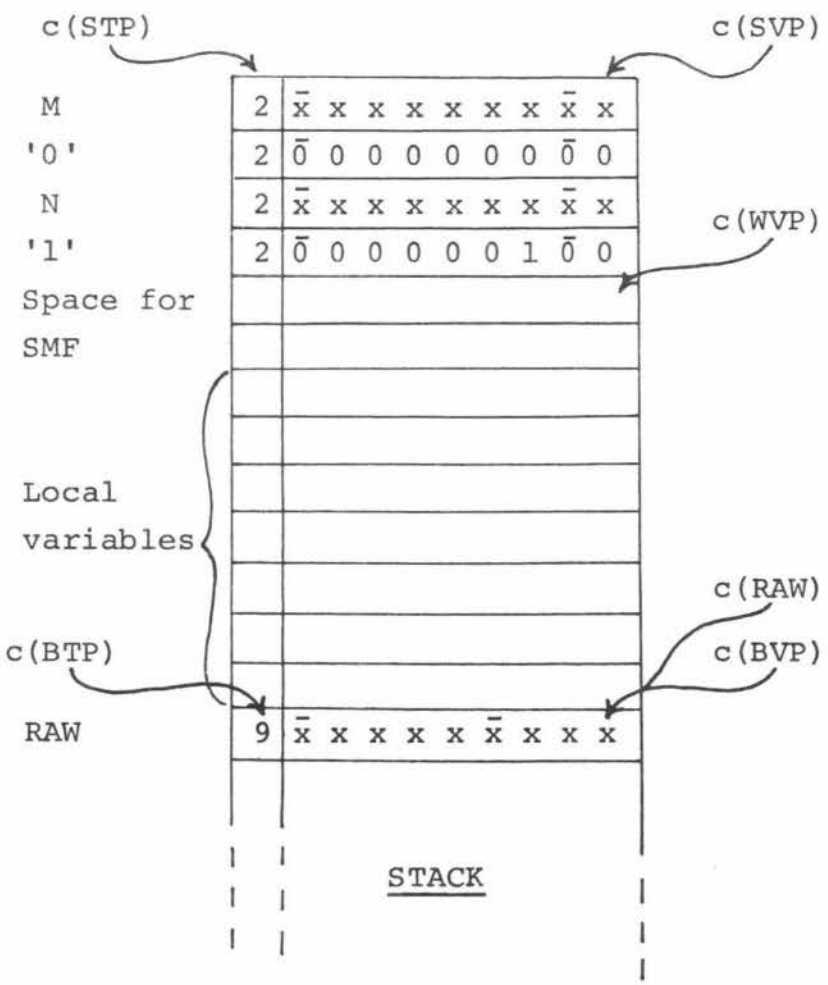
Digit	OP	Parameter
...	...	...
7	TN1	
9	TR	N
14	TN0	
16	TR	M
21	MSF	(5,3,1)
27	...	...
...		

Note: It is assumed here that the array words A, B, C have been allocated local addresses 3, 4, 5 respectively.

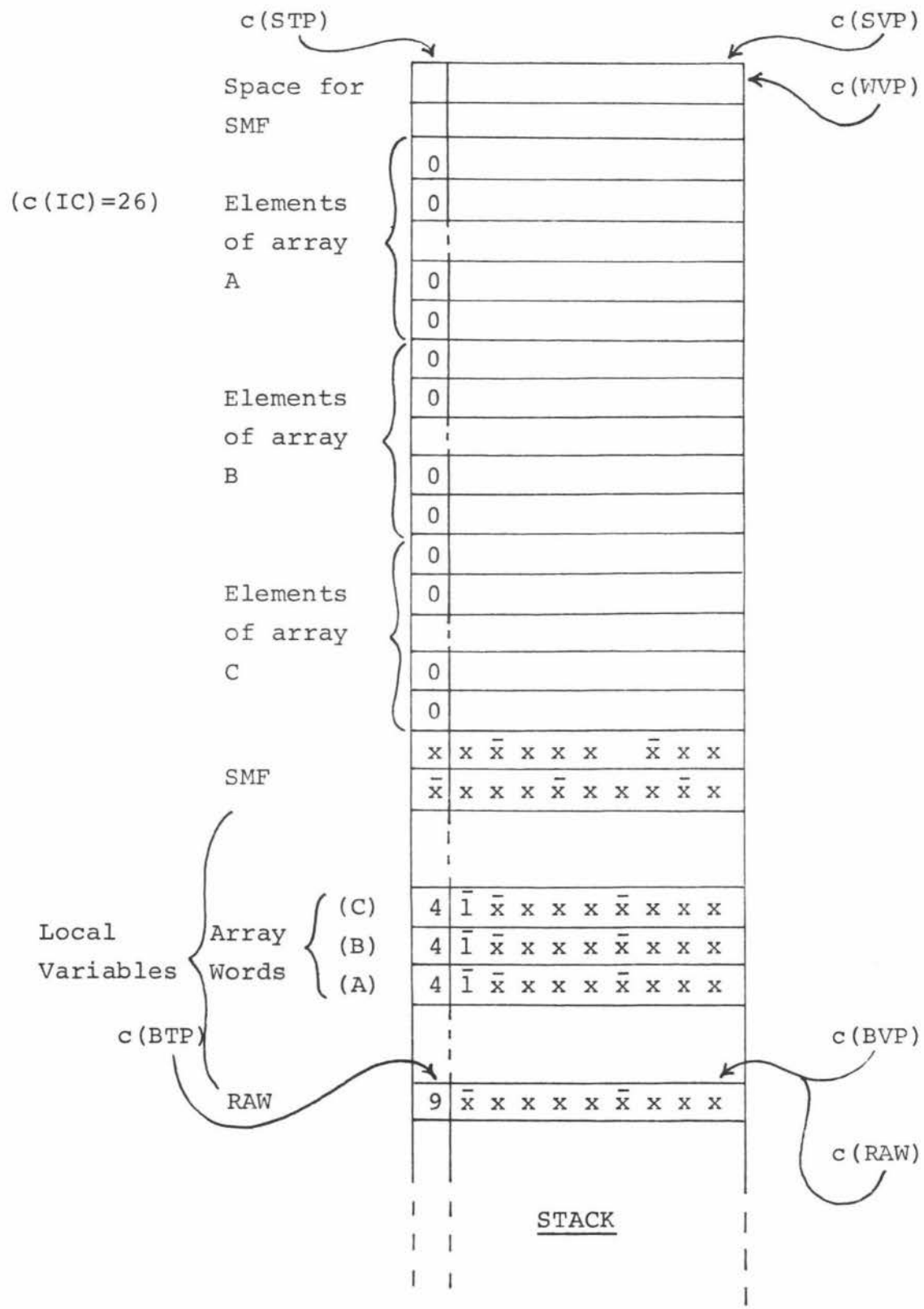
The Stack situation:

(i) Immediately before execution of the MSF operator:

(c(IC)=22)



(ii) Immediately after execution of the MSF operator:



3.15. Input/Output

The 3 main I/O routines used by MUSINT, namely RD (Read), PR (Print), and PRC (Print Control), are by far the most bulky of those used by the Interpreter, occupying in total almost 10,000 digits of core storage. In view of the storage problem faced in implementing the system on the 1620, these routines clearly cannot all occupy core simultaneously - an overlaying technique must be used. Each of the routines is therefore stored on disk and called into a reserved area of core when required. This area must be able to accommodate the largest of the routines, PRC, which occupies almost 4,800 digits, so that effectively this scheme achieves a saving of approximately 5,000 digits of core at the expense of increased execution time.

To control the overlaying of these routines in core, a special I/O overlaying routine is used. This routine resides permanently in core, and maintains a switch for each routine indicating whether or not that routine is currently in core. Thus when a particular routine is required its switch is first checked. If on, the routine is alreadyin core so that control can therefore be immediately transferred to it. If off, however, the routine must first be called in off disk and the 3 switches reset. In this way references to disk storage are kept to a minimum.

3.15.1. Input

As mentioned earlier, MUSSEL has a simple unformatted stream input with data being separated on cards by a blank or a comma. String constants have the additional delimiter '!'. Arrays are read in row-wise by the specification of the array name.

As a result of this method of input, the generation of object code for a READ instruction is a particularly easy job for the Translator. For each variable in a READ list, code is produced to place the variable's address at the top of the Stack. Each of these 'take address' operators is followed by an RD (Read) operator of the following form:

RD (Read) $\bar{3}$ 6

Example:

Source code

19 READ X,Y,Z

Intermediate Language

Digit	OP	Parameter
...	...	...
29	LN	'19'
34	TA	X
39	RD	
41	TA	Y
46	RD	
48	TA	Z
53	RD	
55	...	
...		

The function of the RD interpretive routine is, for a simple variable, to convert the next card element to internal form, and then place the converted result, together with an appropriate type digit, at the address contained at the top of the Stack. In the case of an array, the routine

must store the following appropriate number of card elements, together with type digits, in the locations reserved for the array elements. In both cases, the top Stack unit is then deleted.

The actual working of this RD routine is best explained in a series of steps.

Step 1 (Prepare for array input, if necessary)

The routine checks whether the unit addressed by the pointer at the top of the Stack is an array word. If so, an ARSUB subroutine is called upon to prepare for an array input. This subroutine first sets an ARRAY flag and initialises an array element counter to 1. It then uses the addressed array word and associated SMF to calculate the array size, which is stored. Finally, ARSUB replaces the address at the top of the Stack by the address of the location reserved for the 1st array element.

Step 2 (Input next card element to address at top of Stack)

The type of the next card element can be determined from its first character. An appropriate routine is then used to pack and store this value.

(i) Boolean constants

MUSINT in fact allows the Boolean constants TRUE and FALSE each to be input as any set of characters beginning with 'T' or 'F' respectively. Thus, having picked up the actual Boolean value, the routine stores this value, together with a Boolean type digit of '3', at the address at the top of the Stack, and then skips over the remaining characters in the string until a delimiting comma

or blank is found.

(ii) Strings

These are detected by the appearance of the string delimiter '!'. The function of the routine is very similar to that of the TS (Take String) routine described earlier. That is, it must scan the string and pack it away in the string area in units of 6 characters. In this case, however, the occurrence of the null string must be detected (the string operand for the TS routine is non-null). Another difference here is that the routine must convert 2 successive '!' characters appearing in a string into 1 such character, a job which has already been done by the Translator in the case of the TS routine. (Note - also here the Interpreter must convert the punched code for '!' into a different printer code).

Having packed the string away, the routine then stores a corresponding descriptor (for a non-null string) and type at the address contained at the top of the Stack.

(iii) Numbers

MUSINT allows numbers to be input in integer, decimal, or exponent form.

examples: 0 000 -1234 +0.0000000123456
1.000000 -7.90E-24 .2 123.4567
E10

As a result of this freedom, the MUSINT number

packing routine is quite lengthy. Briefly, the routine scans the number concerned and packs it into mantissa - exponent form. For a number that is input with a mantissa containing more than 7 significant digits the routine takes only the first 8, rounds them to a 7-digit mantissa, and prints an appropriate warning message. While the mantissa is being assembled, the routine maintains an exponent count, which is added to the given exponent for a number in exponent form. At this stage, a check is made that the exponent does not exceed 2 digits in length - if so, an appropriate error message is printed. Otherwise the routine can then go ahead and store the 7-digit mantissa and 2-digit exponent (with signs), together with a 'number' type digit of '2', at the address contained at the top of the Stack.

Step 3 (Array input check)

Here the routine checks for array input, as indicated by an array flag. If there is no flag, the top Stack unit is popped and control returned to the Main Control Routine. Otherwise, the routine increments the array count by one, and checks it against the array size. If not greater, the routine increments the address at the top of the Stack by 10 so that it now points to the next array element location, and then repeats the process described above. Otherwise the routine terminates the array input by clearing the ARRAY flag, popping the top Stack unit, and finally returning to control.

Some notes on the above

1. During the above process a pointer marks the current digit to which the READ buffer has been scanned. The

incrementing of this pointer is controlled by a READ Buffer Manager subroutine (BUFMAN) which checks for the case when the pointer goes past the end of the buffer. When this occurs, a new card is read into the buffer, and the pointer reset accordingly.

2. Instead of allowing array input to be delimited by the slash character, '/', as in PL/I, MUSSEL in fact allows for a null input, which is specified by the appearance of 2 adjacent input delimiters (i.e. blanks or commas). On detection of such an input, the RD routine simply inserts a type digit of '0' ('undefined') at the specified location. In this way, not all elements of an array need be input at one time. And, unlike PL/I, the portion of the array which is input does not necessarily have to be sequential, beginning with the first element. For instance, using this method, one could input the 2nd and 4th rows of a 2-dimensional array only, without having to specify the others.

3.15.2. Output

A full description of the output facilities available in MUSSEL has been given earlier in Chapter 1 in a description of the language itself. There it was explained that the language actually provides for 2 forms of output i.e. simple unformatted output and PICTURE controlled output. Corresponding to these 2 forms are the interpretive routines PR (Print) and PRC (Print Control) respectively, the workings of which will now be explained.

3.15.2.1. Unformatted output

The translation of a PRINT instruction in which there appear no PIC specifications is very similar to that of the READ instruction described above. Here the Translator compiles object code to place the address, or value if a constant or proper expression, of each PRINT list element at the top of the Stack (as in the case of actual parameters to procedures). Each such piece of object code is followed by a PR operator of the following form:

PR (Print) $\bar{3}$ 7

Example:

Source code

```
20        PRINT  A,B*C/2,D(I,J)
```

Intermediate Language

Digit	OP	Parameter
...	...	...
64	LN	'20'
69	TA	A
74	PR	
76	TR	B
81	TR	C
86	OP	'*'
90	TN	'2'
101	OP	'/'
105	PR	
107	TA	D

(cont...)

Digit	OP	Parameter
112	TR	I
117	TR	J
122	INDR	
124	PR	
126	...	
...		

The function of the PR routine is, in the case of a single result, to output the value in a standard format, depending on its type. In the case of an array, the routine must output the elements row-wise.

The general working of the routine is best explained by a series of steps, as was done for the RD routine.

Step 1 (If necessary, place the value, with type, to be o/p at the top of the Stack.)

If an address is contained in the top Stack unit, the routine checks the type of the addressed unit. If this is not an array word, then the type and value of the unit replace the contents of the top Stack unit. Otherwise, the routine calls on the subroutine ARSUB (see RD routine) to prepare for array input, and then replaces the top Stack unit by the type and value of the first array element.

Step 2 (Output the value at the top of the Stack.)

Depending on the type of the unit at the top of the Stack, an appropriate routine is used to convert the associated value to standard form and then place it in the PRINT buffer.

(i) Boolean values

This routine simply places the character string 'TRUE ' or 'FALSE', followed by a spacing of 4 blanks, in the PRINT buffer.

(ii) Strings

This routine unpacks a non-null string into the PRINT buffer, following it by a spacing of 4 blanks. For a null string, the 4 blanks spacing only is inserted.

(iii) Numbers

The routine for converting numbers to standard form is rather more involved than those for Boolean values and strings, mainly due to the fact that numbers can be output in the 3 following forms:

Type	Implied PICTURE
Integer	-******9BBBBBBBB
F-form	-*******.*******
E-form	BBB-*.******ES99

To decide under which of the above formats a number is to be output, the Interpreter first calls on a subroutine, LSUB, to calculate the number's mantissa length. Then, if this length, plus the number's exponent value, lies in the interval [-7,7], the number is output in F-form (of which

the integer form is just a special case). Essentially, this is done by matching to an F-format pattern. Otherwise, the number is output in E-format. In this case, the decimal point is fixed to the right of the first significant digit, so that the exponent value must be altered accordingly. (A fuller description of the working of this routine can be obtained from the appropriate structure diagram in the appendix.)

Having inserted the number in the required form into the PRINT buffer, the routine follows it by the standard spacing of 4 blanks.

Step 3 (Array output check)

Here the routine checks for array output, as indicated by an ARRAY flag. If there is no such flag, the top Stack unit is popped and control returned to the Main Control Routine. Otherwise, the routine increments the array count by 1, and checks it against the array size. If not greater, the routine replaces the contents of the top Stack unit by the value and type of the next array element, and then repeats the above process. Otherwise the routine terminates the array input by clearing the ARRAY flag, popping the top Stack unit, and finally returning to control.

Note: During the above process, a pointer marks the current position to which the PRINT buffer has been filled. All increments to this pointer are controlled by a Print Buffer Manager subroutine (PBFMAN) which checks for the case when the pointer meets the buffer end. When this occurs, and there is no ARRAY flag, the routine prints a line overflow warning message - programmers in MUSSEL must specifically ask for a new line when one is required - and then calls on another

routine, OUTPUT, to print and clear the print buffer, and reset the buffer pointer to the start of the buffer. Finally, PBFMAN checks to see whether the line overflow has occurred as a result of space being required in the PRINT buffer for a non-blank value. If so, it increments the now reset buffer marker to leave space for the insertion of this value. Otherwise, if only a blank space had been required, the buffer pointer remains at the start of the buffer.

3.15.2.2. PICTURE controlled output

In translating the elements involved in a simple, or multiple, PICTURE specification in MUSSEL, the Translator compiles object code to assemble the addresses, or values if constants or proper expressions, of the concerned PRINT list elements at the top of the Stack. This is followed by a PRC (Print Control) operator of the following form:

PRC (Print Control) 3 8 x x x x . . . x x 0 ≠

This operator has 2 parameters. The first is the (2-digit) number of list elements appearing in the PIC specification. The second is the PICTURE itself in alphameric mode, terminated by an alphameric record mark.

The job of the PRC interpretive routine is to output the values of each of the list elements according to the specified PICTURE, and then delete the list addresses, or values, from the top of the Stack.

Example:

Source code

```
15      PRINT  (A,B*C,D(I,J)(PIC=***.*))
```

Intermediate Language

Digit	OP	Parameter
...	...	...
944	LN	'15'
949	TA	A
954	TR	B
959	TR	C
964	OP	'*'
968	TA	D
973	TR	I
978	TR	J
983	INDR	
985	PRC	(3,***.*)
1001	...	...
...		

Again the workings of this PRC routine is best explained by a series of steps.

Step 1 (Initialise)

The routine marks the beginning of the PICTURE as it appears in the I.L. stream, and then, having marked the top of the Stack, decreases the Stack pointers SVP and STP to address the unit corresponding to the first element in the PIC specification.

Step 2 (Bubble up value)

The routine checks the type of the unit addressed by the Stack pointers. If an address, the routine then checks the type of the addressed unit. If this is not an array word, then the type and value of the unit replace the contents of the Stack unit addressed by the Stack pointers. Otherwise,

the routine calls on the subroutine ARSUB to prepare for an array input, and then replaces the addressed Stack unit by the type and value of the first array element.

Step 3 (Output value)

Depending on the type of the addressed value in the Stack, an appropriate routine is used to convert the value to the form specified and then place it in the PRINT buffer.

(i) Boolean values

Asterisks only are allowed in a PICTURE associated with a Boolean value. The routine here simply counts the number of these asterisks and, depending on their number, either chops the particular constant TRUE or FALSE to insert it in the PRINT buffer, or else left justifies it and pads the buffer out with an appropriate number of blanks.

(ii) Strings

This routine first lays the string out at the top of the Stack, and then matches the string with the PICTURE, character by character, filling the PRINT buffer as it goes. Depending on the number of asterisks in the PICTURE, and on the length of the string, the string itself will either be chopped or padded out with an appropriate number of blanks. For a null string, the routine simply counts the number of characters in the PICTURE, and then inserts a corresponding number of blanks in the PRINT buffer.

A special case exists for constant strings

appearing in a PRINT list. To save space, the Translator in this instance compiles a PRC operator with a 'null' PIC parameter (i.e. an alphameric record mark as a second parameter). The Interpreter recognises this and simply outputs the string with no following spaces.

(iii) Numbers

The routine here is quite involved. Basically, it performs 2 scans of the given PICTURE. During the preliminary scan the routine picks up the general form in which the number is to be output i.e. it counts the number of spaces reserved for the number before the decimal point (if given), after the decimal point, and also for the exponent. For a number to be output in F-format, the routine checks that the number will 'fit', and then rounds if necessary. During the rescan of the PICTURE, the routine matches the number with the PICTURE, filling the PRINT buffer as it goes. Points to note here are:

(i) Leading zeros before a decimal point are automatically suppressed unless specified by a '9'. This implies that trailing zeros after a decimal point are in fact inserted.

example: a value of 2.79 would be output under a PICTURE of ****.**** as ' 2.7900'.

(ii) A sign (S or -) at the head of a PICTURE is automatically floated up. A sign elsewhere in a PICTURE, however, is printed in the position specified.

(iii) The rules applying to exponent PICTURE's are the same as those applying to mantissa PICTURE's.

Again, a fuller explanation of the workings of this routine can be obtained from the appropriate structure diagrams in the appendix.

Step 4 (Check for array output)

Here the routine checks for array output, as indicated by the presence of an ARRAY flag. If there is no such flag, a check is made that all the list elements have been output i.e. c(SVP) is compared with c(TOPV), which marks the top of the Stack. If not equal, the Stack pointers c(STP) and c(SVP) are increased to point to the next Stack unit (corresponding to the next list element), and the above process repeated from step 2. Otherwise, the PRC routine terminates by popping the Stack units corresponding to the list elements, and returning to control.

If there is an ARRAY flag, the routine increments the array element count by 1, and checks it against the array size. If not greater, the routine replaces the contents of the addressed Stack unit by the value and type of the next array element, and then repeats the above process from step 2. Otherwise, the routine terminates the array input by clearing the ARRAY flag, and then checks for another list element as described in the above paragraph.

Notes:

1. A Print Buffer Manager subroutine is used as in the case of the PR routine.

2. Unlike the case for unformatted o/p, in array output under a PIC control, spacing between the elements is not implied, and must therefore be specified by the programmer in the PICTURE itself.

3.15.2.3. Printer controls

Corresponding to the NEWPAGE control used in the source language is the object program operator NP (Newpage), which is of the following form:

NP (Newpage) $\bar{3}$ 9

The function of the NP interpretive routine is simply to call on the subroutine OUTPUT to print and clear the contents (if any) of the PRINT buffer, and then to skip to a newpage.

Corresponding to the NEWLINE control used in the source language is the object program operator NL (Newline), which is of the following form:

NL (Newline) $\bar{4}$ 0 $\bar{x}$ x

The parameter here is the number of lines to be skipped. (Note - if the parameter is omitted in the source program, the Translator inserts an implied parameter of '1'.) The NL interpretive routine is, here again, quite straightforward. It simply calls on the subroutine OUTPUT to print and clear the contents (if any) of the PRINT buffer, and then skips the appropriate number of lines.

The SP (Space) operator in the Intermediate Language corresponds to the SPACE control used in the source language.

This operator is of the form

SP (Space) $\bar{4}\ 1\ \bar{x}\ x$

where the parameter specifies the number of spaces to be left in the PRINT buffer. (Here again, if the parameter is omitted in the source program, the Translator inserts an implied parameter of '1'.) The SP interpretive routine just uses the Print Buffer Manager subroutine (PBFMAN) to increment the buffer pointer by twice the SP operator's parameter.

TAB $\bar{4}\ 2\ \bar{x}\ x\ x$

The 3-digit parameter here is the printer position immediately to the left of the position in which the next printed character is to be placed. The TAB interpretive routine first checks that this parameter is not greater than 120, the printer width being used in MUSSEL. It then resets the buffer pointer to the start of the buffer, plus twice the value of the given TAB parameter, and finally returns to control.

3.16. The END Operator

The parameterless END operator merely indicates the end of the object program. It is of the following form:

END $\bar{3}\ 5$

The function of the operator is to signal to the Interpreter to print out the contents (if any) of the PRINT buffer (see section 3.15.2.), and then cease execution. A

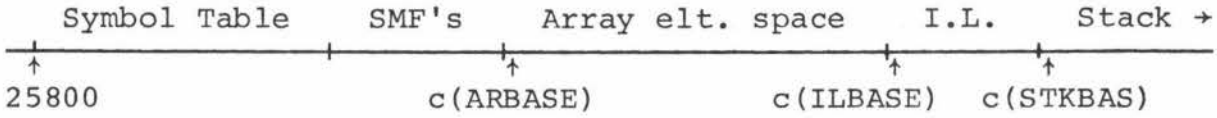
useful diagnostic option at this stage would be a listing of the contents of the Symbol Table together with the variables' names. Having done so, the Interpreter, in a batch processing mode, would then call in and link to the Translator to begin processing the next (student) program.

3.17. The Translator-Interpreter Interface

Before calling in the Interpreter to commence the execution phase of the system, the Translator first loads the Symbol Table, Storage Mapping Functions (if any), and Intermediate Language into core, and then stores a number of values at the bottom end of the available core. These values, which will be required by the Interpreter's interface routine, are stored as follows:

Core Address	Symbolic Address	Function
2402 - 2406	ARBASE	Addresses type digit of lst array element (if any)
2407 - 2411	ILBASE	Addresses lst digit of I.L.
2412 - 2416	STKBAS	Addresses digit immediately following end of the I.L.
2417		Switch for postinterpretation S.T. listing option ('1' - on, '0' - off)
2418 - 2431		DCF for name area

The set up in core at this stage is as follows:



The Translator then calls in the Interpreter from the disk, and links to its interface routine. This routine has 3 main tasks:

1. To clear the PRINT buffer, and initialise the READ and PRINT buffer markers.

2. To initialise the instruction counter and Stack pointers.

3. To clear the array area i.e. to set the type digits of the array elements (if any) to '0' ('undefined').

Having performed these tasks, the routine then branches to the Main Control Routine to commence interpreting the Intermediate Language.

CHAPTER 4

POSSIBLE DEVELOPMENTS OF THE SYSTEM

What has been presented so far in this dissertation is only the basis for what could be a far more sophisticated Interpreter. Ideally, we would like to develop the MUSSEL system in 3 main areas.

4.1. Diagnostics

As mentioned earlier, the use of an interpreter enables quite extensive error diagnostics to be produced. In addition to the usual variety of compile-time syntax/semantics diagnostic checks, one would like to develop in MUSSEL such execution-time diagnostics as detection and recovery of undefined variables, improper array references (such as subscripts omitted or out of range), improper call sequences (such as calling a function or subroutine with an improper number of arguments), and illegal operations. The basis for such facilities has been laid, and it would not therefore be too difficult to develop them and create meaningful error messages. For example, the line number operator has been incorporated into the Intermediate Language so that the Interpreter can quote the line number of an error during execution. Also, a mechanism has been set up for the Interpreter to have access to identifier names during execution.

We would also like to adopt a philosophy similar to that of SPLINTER (R.L. Glass, [2]), which is that of a 'forgiving processor'. In SPLINTER, diagnostics are voluminous, almost none is fatal, and every attempt is made to

continue a compilation/interpretation past a detected error. A few types of errors are fatal, mostly those involving overflow of a vital table, and an accumulation of 15 less significant errors is also fatal. In a checkout system, it is especially important to obtain a maximum amount of information from each computer run. The cost of this in the time wasted while a program 'runs on' is usually (debatably) insignificant.

4.2. Debugging facilities

Diagnostics, discussed in the section above, is a passive form of user checkout aid; it requires no active effort on the user's part, but in fact arises normally from involuntary errors.

Debugging aids, on the other hand, allow active user participation in the checkout process. We believe that good debugging aids are quite important, particularly in a student environment, and for this reason intend to make them an essential part of the MUSSEL system.

Features that we would like to incorporate into MUSSEL are similar to those provided in SPLINTER. Some of the possibilities are as follows:

1. A 'statement trace' to print out the line numbers of statements as they are executed.

2. A 'name trace' to list the names and values of specified variables whenever they are referenced during execution, as well as the line number of the statement in which the reference occurs.

3. A 'store name' trace to list the names and values of specified variables whenever their values change during execution, as well as the line number of the statement in which the reference occurs.

These trace modes could be requested or suspended, as in SPLINTER, by the use of control statements inserted at key points in the user's program. One could also, perhaps, allow such statements to be embedded in other statements, such as

```
IF  A.LT.B  THEN  *TRACE A
```

thereby obtaining a conditional trace.

Other possibilities include an optional postinterpretation printout of the contents of the Symbol Table, and an optional listing of specified sections of the Intermediate Language.

4.3. Monitoring the performance of MUSSEL

It would be nice to monitor the performance and use of MUSSEL, particularly during the initial stages of its development. The accounting system employed could be similar to that of the DITRAN system (Moulton & Muller, [13]), which records, for each submitted program, the following information:

- user and run identification
- memory requirements of the program
- size of program
- number of statements of each type
- information describing the size of system tables needed during compilation

- library functions referenced
- compilation time
- execution time
- number of records input and output during execution
- a code describing condition of termination
- number of errors in each of 21 categories.

An analysis of information of this type should help in identifying those areas where improvement of compiler performance, or even of the language itself, should be of greatest value. One area of special interest would be the identifying of the types of errors most frequently made by the users. Having identified these errors, we could then make an effort to instruct users or to improve the diagnostics in order to provide stronger cues to aid the correction of errors.

APPENDIX A

THE SYNTAX OF MUSSEL IN BNF[1]

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|
U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<logical value> ::= TRUE|FALSE|T|F

<delimiter> ::= <operator>|<bracket>|<separator>|<key>

<operator> ::= <arithmetic operator>|<string operator>|
<relational operator>|<logical operator>|
<sequential operator>

<arithmetic operator> ::= +|-|*|/|./.|**

<string operator> ::= .CAT.

<relational operator> ::= .LT.|.LE.|.EQ.|.NE.|.GE.|.GT.

<logical operator> ::= .OR.|.AND.|.NOT.

<sequential operator> ::= THEN|ELSE

<bracket> ::= (|)|!|DO|END

<separator> ::= ,|.|:|TO| |UNTIL|WHILE|FROM|BY|ON|AS|IN|
|IS|FOR|TIMES

<key> ::= RES|RESERVE|SET|READ|PRINT|EXIT|REPEAT|CASE|IF|
CHOICE|DEFINE|VALUE|EXECUTE

---

[1]As provided by Miss N.M. Gordon.

```
<program> ::= <instruction>

<instruction> ::= <simple instruction>|<group instruction>|
                  <label>: <group instruction>

<simple instruction> ::= <reserve>|<assign>|<read>|<print>|
                         <exit>|<subroutine definition>|
                         <subroutine execute>|
                         <function value assign>

<group instruction> ::= <simple group>|<repeat group>|
                        <choice group>|<case group>|
                        <conditional group>|<if group>

<reserve> ::= RESERVE <loclist>| RES <loclist>

<loclist> ::= <simple variable>|<array spec>|<multiple array sp>
              <loclist>,<simple variable>|<loclist>,<array spec>
              <loclist>,<multiple array spec>

<simple variable> ::= <variable identifier>

<variable identifier>  ::= <identifier>

<identifier> ::= <letter>|<identifier><letter>|
                 <identifier><digit>

<array spec> ::= <array identifier>(<bound pair list>)

<bound pair list> ::= <bound pair>|<bound pair>,<bound pair>

<bound pair> ::= <valexp>:<valexp>

<array identifier> ::= <identifier>
```

```
<multiple array spec> ::= (<array segment>)

<array segment> ::= <array list>,<array spec>

<array list> ::= <array identifier>|<array list>,<array ident.>

<assign> ::= SET <list part> TO <expression>

<list part> ::= <variable>|<list part>,<variable>

<variable> ::= <simple variable>|<array variable>

<array variable> ::= <array identifier>(<valexp>)|<array id.>
                     <valexp>,<valexp>)|<array identifier>

<read> ::= READ <listpart>

<print> ::= PRINT <print list>

<print list> ::= <print el>|<print control>|
                 <print list>,<print el>|
                 <print list>,<print control>

<print el> ::=  (<expression>,<print el>)|
                <expression><picture control>

<picture control> ::= (PIC=<picture>)|<empty>

<picture> ::= <alpha picture>|<numeric picture>

<alpha picture> ::= <first alpha>|<alpha picture><first alpha>|
                    <alpha picture>,<first alpha>|
                    <alpha picture>(<integer>)

<first alpha> ::= *|B
```

```
<numeric picture> ::= <sign><integer part><decimal part>
                      <exponent part><sign>

<sign> ::= -|S|<empty>

<integer part> ::= <first num>|<integer part><first num>|
                   <integer part>,<first num>|
                   <integer part>(<integer>)|<empty>

<decimal part> ::= .<integer part>|<empty>

<exponent part> ::= E<sign>**|E<sign>99|<empty>

<first num> ::= *|9|B

 empty  ::=

<print control> ::= NEWPAGE|NEWLINE|NEWLINE(<valexp>)
                    SPACE(<valexp>)|TAB(<valexp>)

<exit> ::= EXIT|EXIT FROM <label>

<label> ::= <identifier>

<subroutine definition> ::= DEFINE <subroutine identifier> AS|
          DEFINE <subroutine identifier> ON <parameterlist> AS

<subroutine identifier> ::= <identifier>

<parameterlist> ::= <parameter identifier>|<parameterlist>,
                                          <parameter identifier>

<parameter identifier> ::= <identifier>

<subroutine execute> ::=  EXECUTE <subroutine identifier>|
                 EXECUTE <subroutine identifier>(<explist>)
```

```
<expression> ::= <valexp>|<condexp>|<string exp>

<valexp> ::= <term>|<adding op><term>|<valexp><adding op><term>

<term> ::= <factor>|<term><mult op><factor>

<factor> ::= <primary>|<factor>**<primary>

<primary> ::= <unsigned number>|<variable>|<function call>
              (<valexp>)

<mult op> ::= *|/|./.

<adding op> ::= +|-

<condexp> ::= <logical term>|<condexp>.OR.<logical term>

<logical term> ::= <logical factor>|<logical term>.AND.
                   <logical factor>

<logical factor> ::= <logical primary>|.NOT.<logical primary>

<logical primary> ::= <logical value>|<variable>|<function call>
                      |<relation>|(<condexp>)

<relation> ::= <valexp><relational op><value exp>|
               <string exp><relational op><string exp>

<string exp> ::= <string term>|<string exp>.CAT.<string term>

<string term> ::= <variable>|<string>|<function call>
                  <string exp>

<function call> ::= <function identifier>|<function identifier>
                    (<explist>)
```

```
<explist> ::= <expression>|<explist>,<expression>

<function value assign> ::= VALUE IS <expression>

<simple group> ::= DO <eol><instructions><group end>

<instructions> ::= <instruction>|<instructions><instruction>

<group end> ::= END <eol>|END <label><eol>

<repeat group> ::= DO REPEAT<eol><instructions><group end>
                  |DO REPEAT <control><eol><instructions>
                                          <group end>

<control>  ::= UNTIL <condexp>|WHILE<condexp>
               <valexp>TIMES|FOR<variable>FROM <valexp><for end>
               |FOR<variable>SET TO<explist>

<for end> ::= TO<valexp>BY<valexp>|BY<valexp>|
              TO<valexp>|<empty>

<choice group> ::= DO CHOICE OF<eol><if group><group end>

<case group> ::= <casehead><instructions><group end>

<casehead> ::= DO CASE<valexp>OF<eol>|
               DO CASE<valexp>IN(<integer>,<integer>)OF<eol>

<if group> ::= IF<condexp>THEN<simple instruction>|
               IF<condexp><eol>THEN<instruction>

<conditional group> ::= DO IF<condexp><eol>THEN<instruction>
             <group end>|DO IF<condexp><eol>THEN<instruction>
              ELSE<instruction><group end>
```

```
<function identifier> ::= <identifier>|<library function>

<library function> ::= LOG|EXP|ABS|SIN|COS|SQRT|STRING|NUMBER
                      |INTEGER|CONDITION|LENGTH|POSITION|
                       SUBSTRING|COUNT|MAXIMUM|MINIMUM

<number> ::= <unsigned number>|+<unsigned number>|
             -<unsigned number>

<unsigned number> ::= <decimal number><exponent>|
                      <decimal number>

<decimal number> ::= <unsigned integer>|<decimal fraction>|
                     <unsigned integer><decimal fraction>

<exponent> ::= E<integer>

<decimal fraction> ::=  .<unsigned integer>

<integer> ::= <unsigned integer>|+<unsigned integer>|
              -<unsigned integer>

<unsigned integer> ::= <digit>|<unsigned integer><digit>

<proper string> ::= <string character>|<proper string>
                                       <string character>
<string character> ::= <letter>|<digit>|<quote>|<special
                        character>|<empty>

<quote> ::= !!

<special character> ::= :|.|*|+|-|/|$|@|≠|)|(

<string> ::= !<proper string>!

<eol> ::=  end of line - not represented externally
```

APPENDIX B

A SUMMARY OF THE INTERMEDIATE LANGUAGE

OPERATOR		INSTRUCTION FORM	
Mnemonic	Meaning	Opcode	Parameter(s)
LN	LINE NUMBER	01	***
TA	TAKE ADDRESS	02	***
TR	TAKE RESULT	03	***
TN	TAKE NUMBER	04	*********
TN0	TAKE NUMBER ZERO	05	
TN1	TAKE NUMBER ONE	06	
TB	TAKE BOOLEAN	07	**
TS	TAKE STRING	08	****...**0≠
TNS	TAKE NULL STRING	09	
INDA	INDEX ADDRESS	10	
INDR	INDEX RESULT	11	
ST	STORE	12	
STA	STORE ALSO	13	
VI	VALUE IS	15	
UJ	UNCONDITIONAL JUMP	16	****
BA	BRANCH ADDRESS	17	****
IFJ	IF FALSE JUMP	18	****
OP	OPERATOR	19	**
RT	REPEAT-TIMES	20	****,***
RF	REPEAT-FOR	21	****,***
RFWT	REPEAT-FOR-WITHOUT-TEST	22	***
RS	REPEAT-SET	23	**,****,***
CS	CASE	24	**
CSWB	CASE-WITHOUT-BOUNDS	25	**
RAP	RETURN ADDRESS (PROCEDURE)	26	
RAF	RETURN ADDRESS (FUNCTION)	27	

(cont...)

OPERATOR		INSTRUCTION FORM	
Mnemonic	Meaning	Opcode	Parameter(s)
CP	CALL PROCEDURE	28	***,**
CLP	CALL LIBRARY PROCEDURE	29	**,**
PE	PROCEDURE ENTRY	30	**,**
TLA	TAKE LOCAL ADDRESS	31	**
TLR	TAKE LOCAL RESULT	32	**
MSF	MAKE STORAGE FUNCTION	33	**,*,*
RE	RETURN	34	
END	END	35	
RD	READ	36	
PR	PRINT	37	
PRC	PRINT CONTROL	38	**,****...**0≠
NP	NEWPAGE	39	
NL	NEWLINE	40	**
SP	SPACE	41	**
TAB	TAB	42	***

SECONDARY OPCODES

Operator	Code	Operator	Code
OR	01	GT	09
AND	02	CAT	10
NOT	03	+	11
LT	04	-	12
LE	05	*	13
EQ	06	/	14
NE	07	./.	15
GE	08	NEG	16

APPENDIX C

THE INTERPRETER STRUCTURE DIAGRAMS

Note: To be consistent with the SPS assembly language, where, for example, the name 'A' is in fact the symbolic address of a location, we have used, in the following structure diagrams, the notation c(A) to refer to the contents of the location at this address. In this way, the indirect addressing feature of the 1620 can be easily represented. For example, c(c(A)) would refer to the "contents of the contents of the address A".

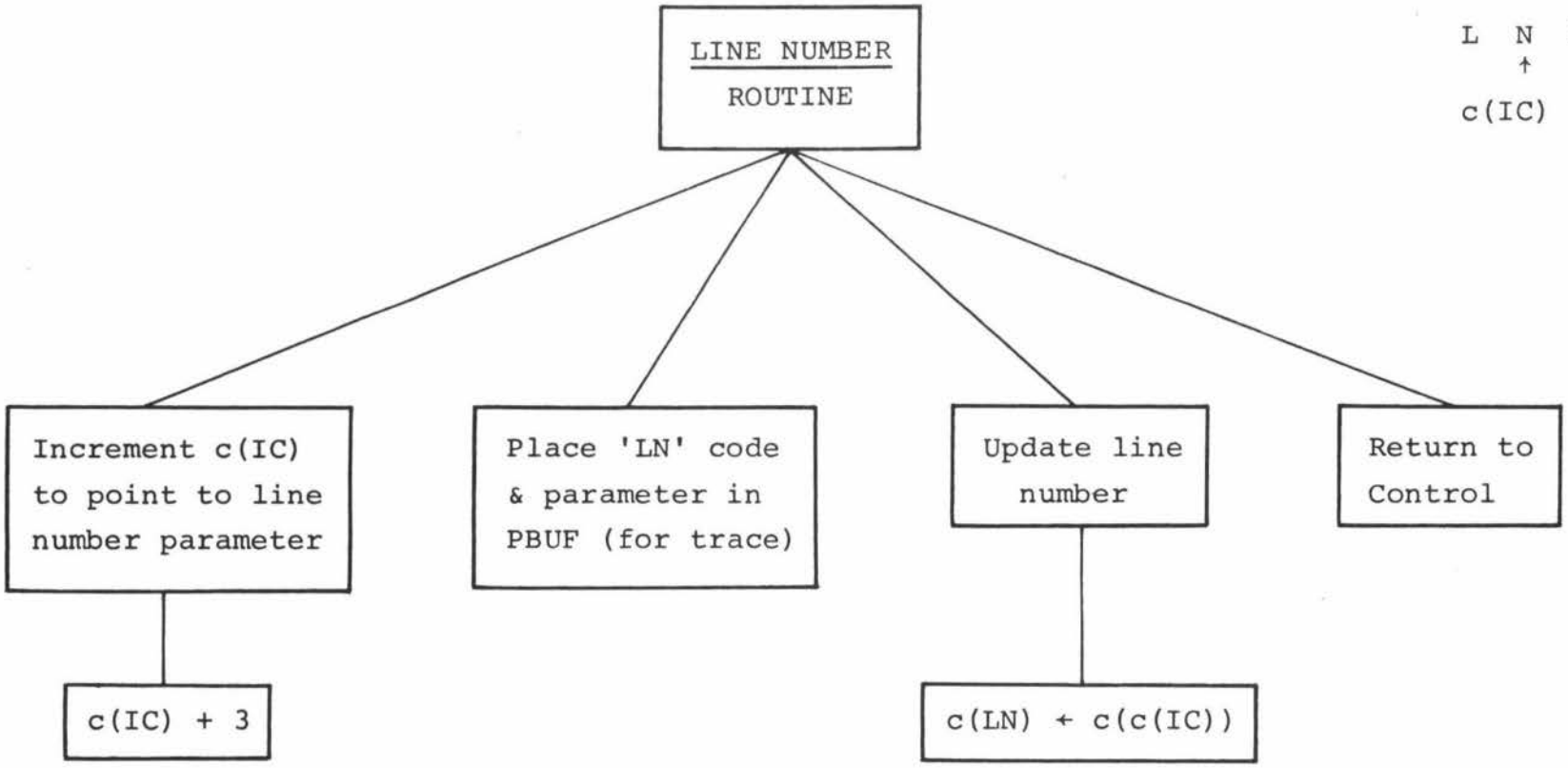
OP 01

opcode parameter

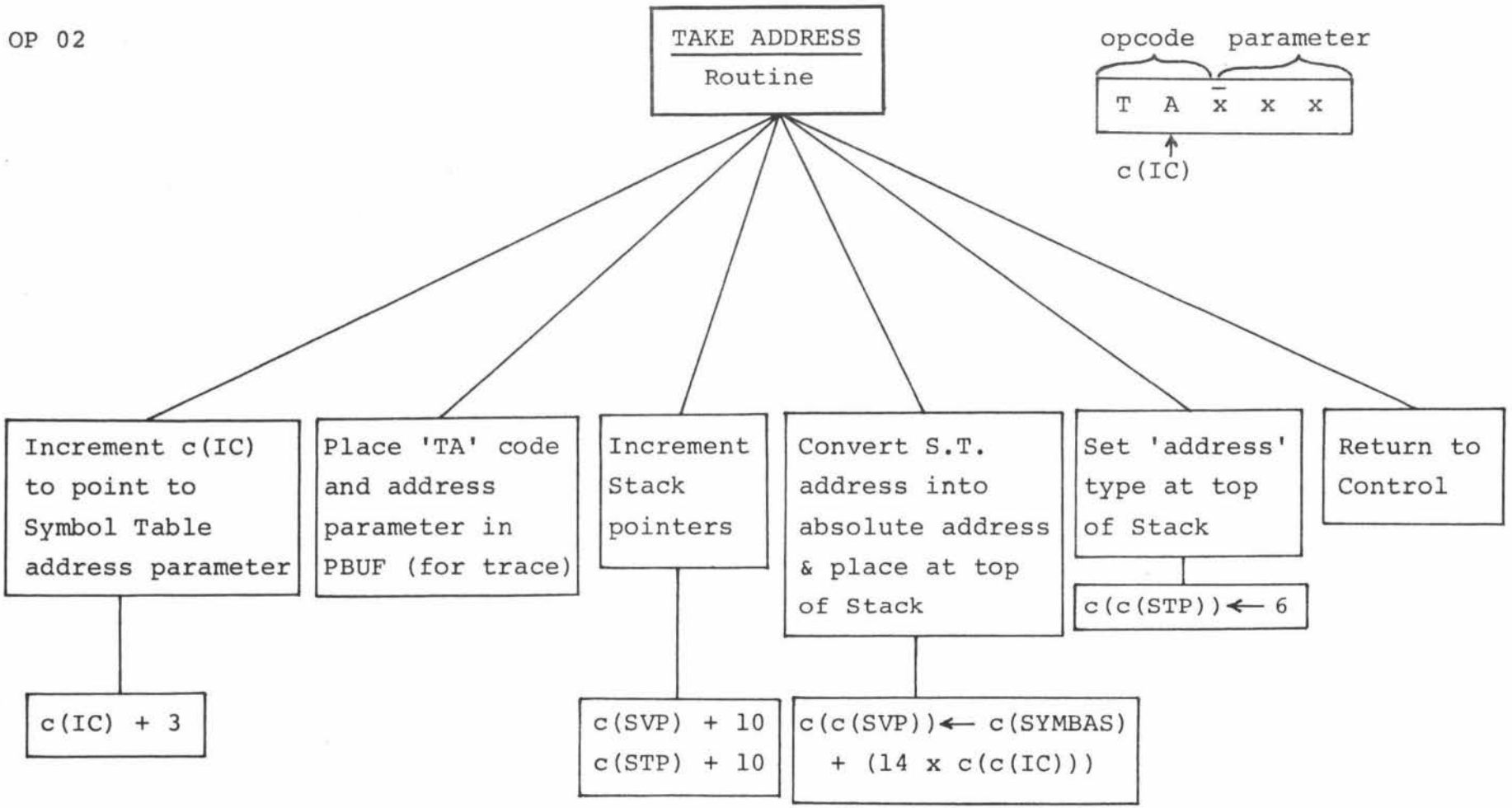
L N $\bar{x}$ x x

↑

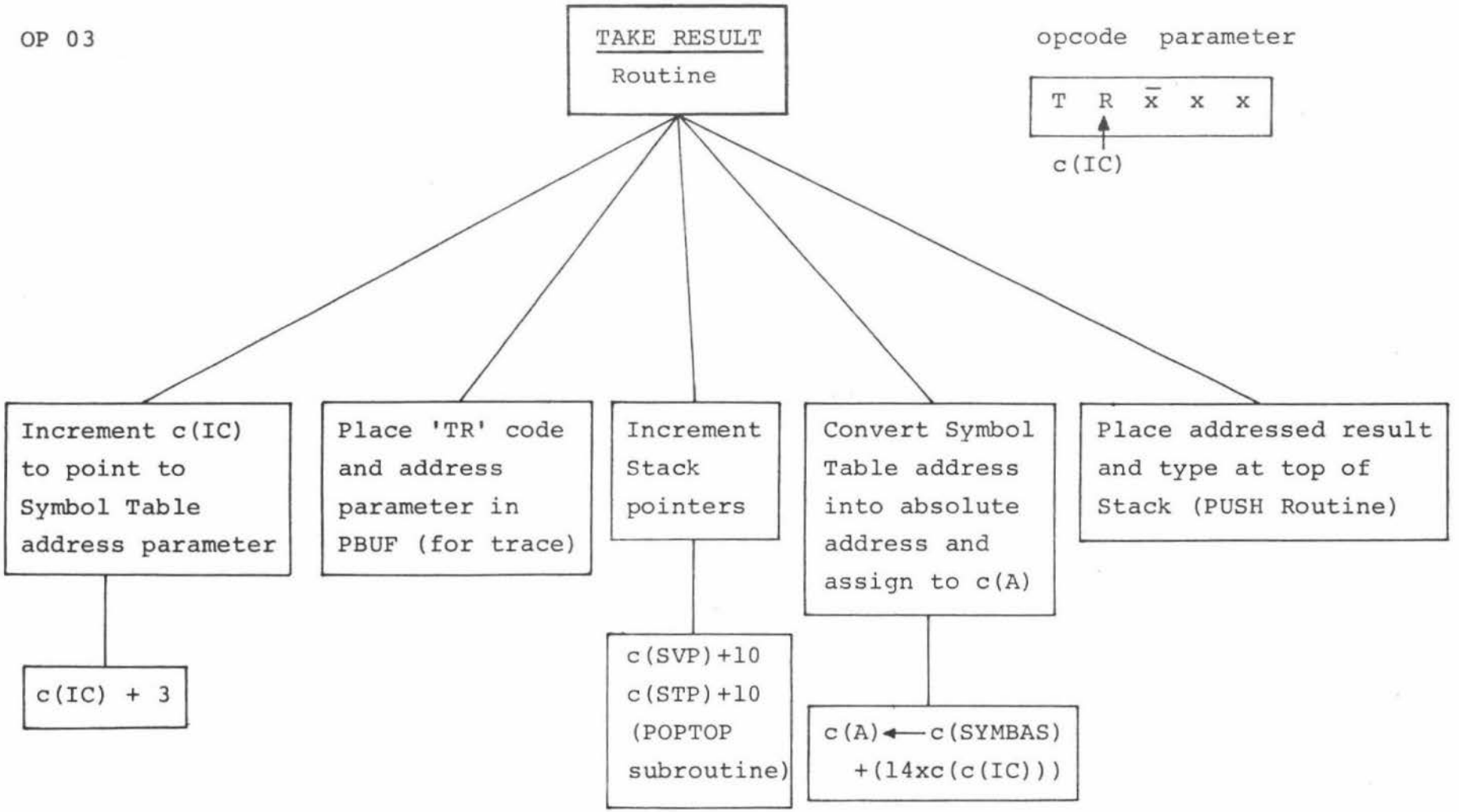
c(IC)

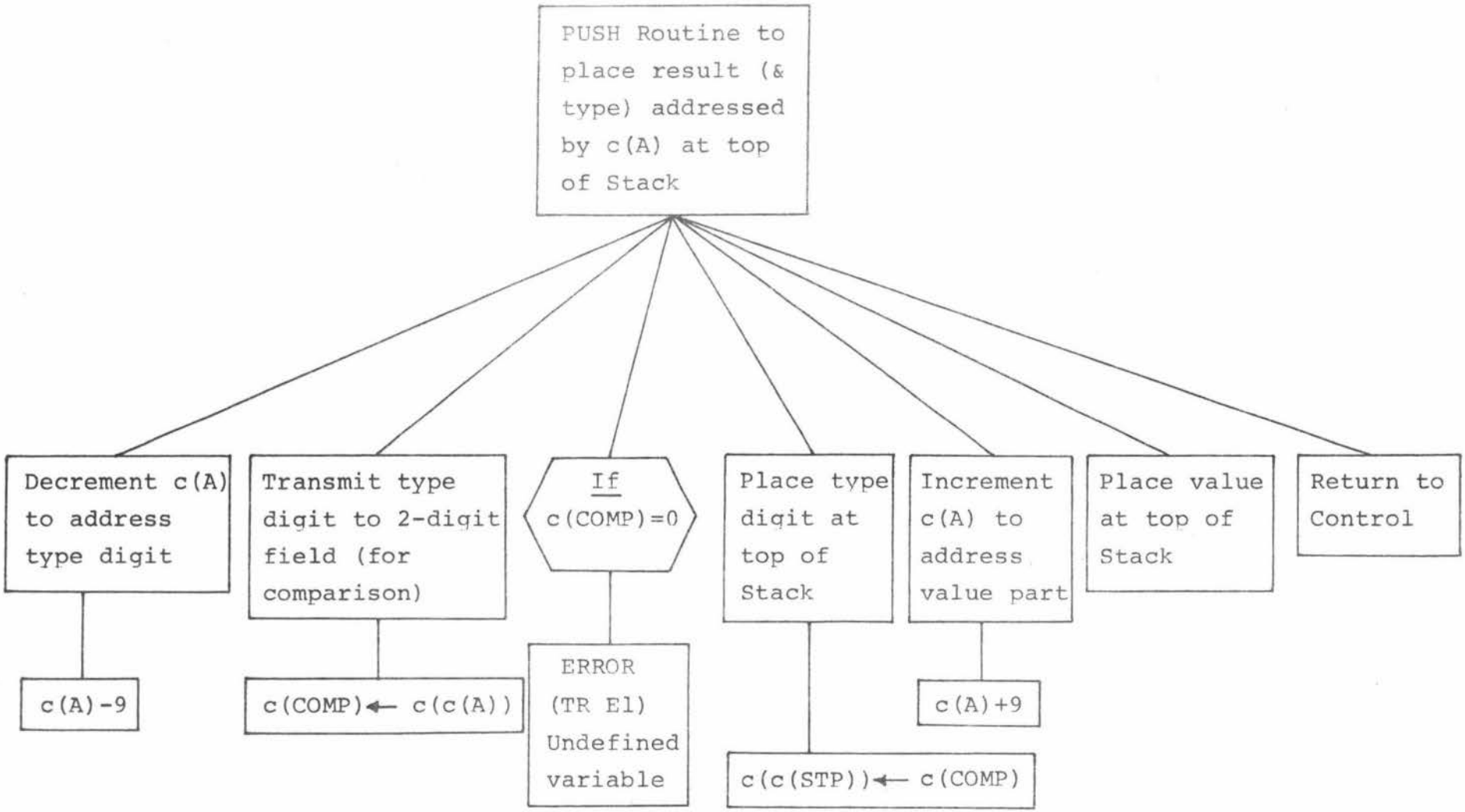


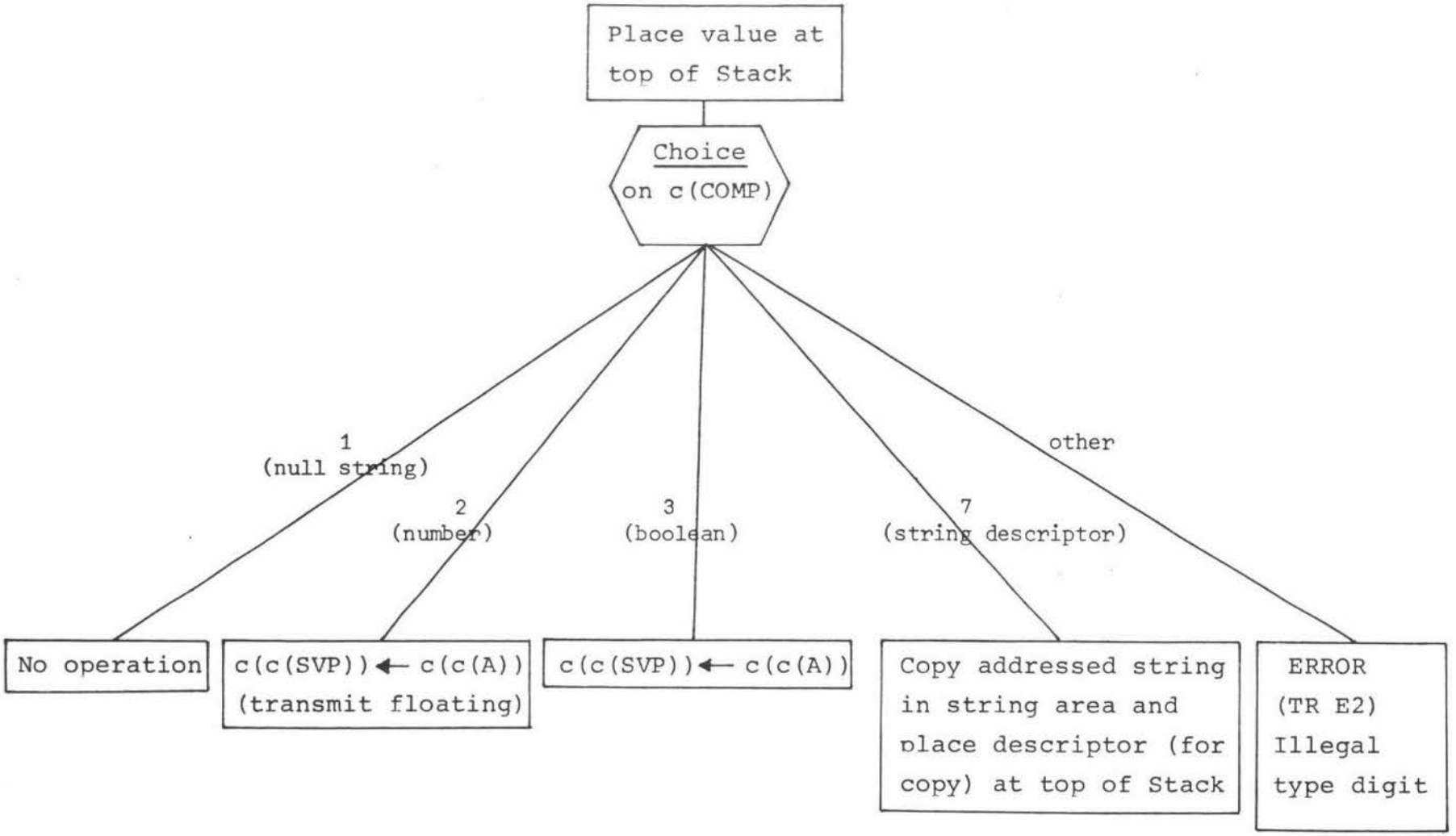
OP 02

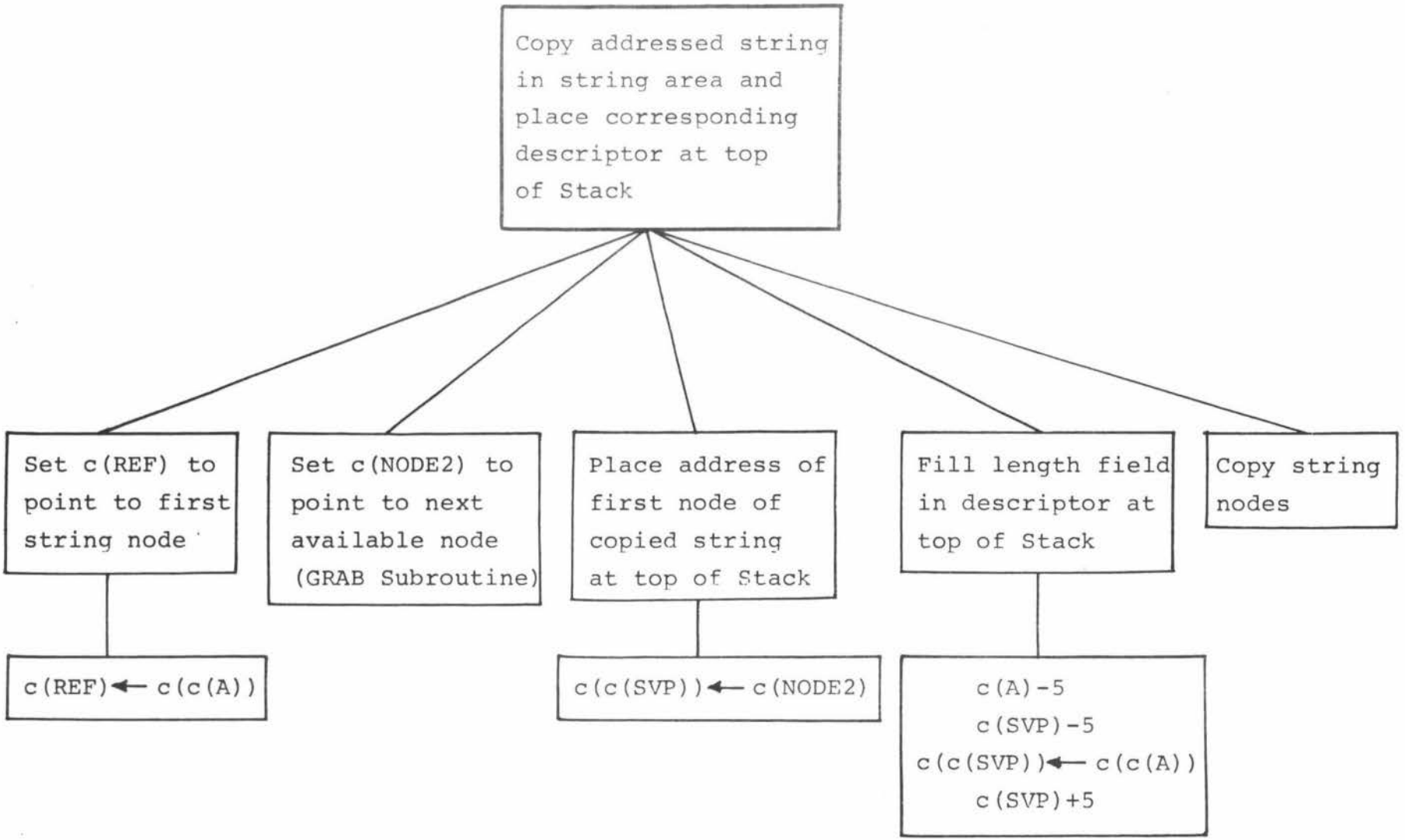

TAKE ADDRESS
Routine
opcode parameter
T A x̄ x x
c(IC)
Increment c(IC) to point to Symbol Table address parameter
Place 'TA' code and address parameter in PBUF (for trace)
Increment Stack pointers
Convert S.T. address into absolute address & place at top of Stack
Set 'address' type at top of Stack
Return to Control
c(IC) + 3
c(SVP) + 10
c(STP) + 10
c(c(SVP)) ← c(SYMBAS) + (14 x c(c(IC)))
c(c(STP)) ← 6

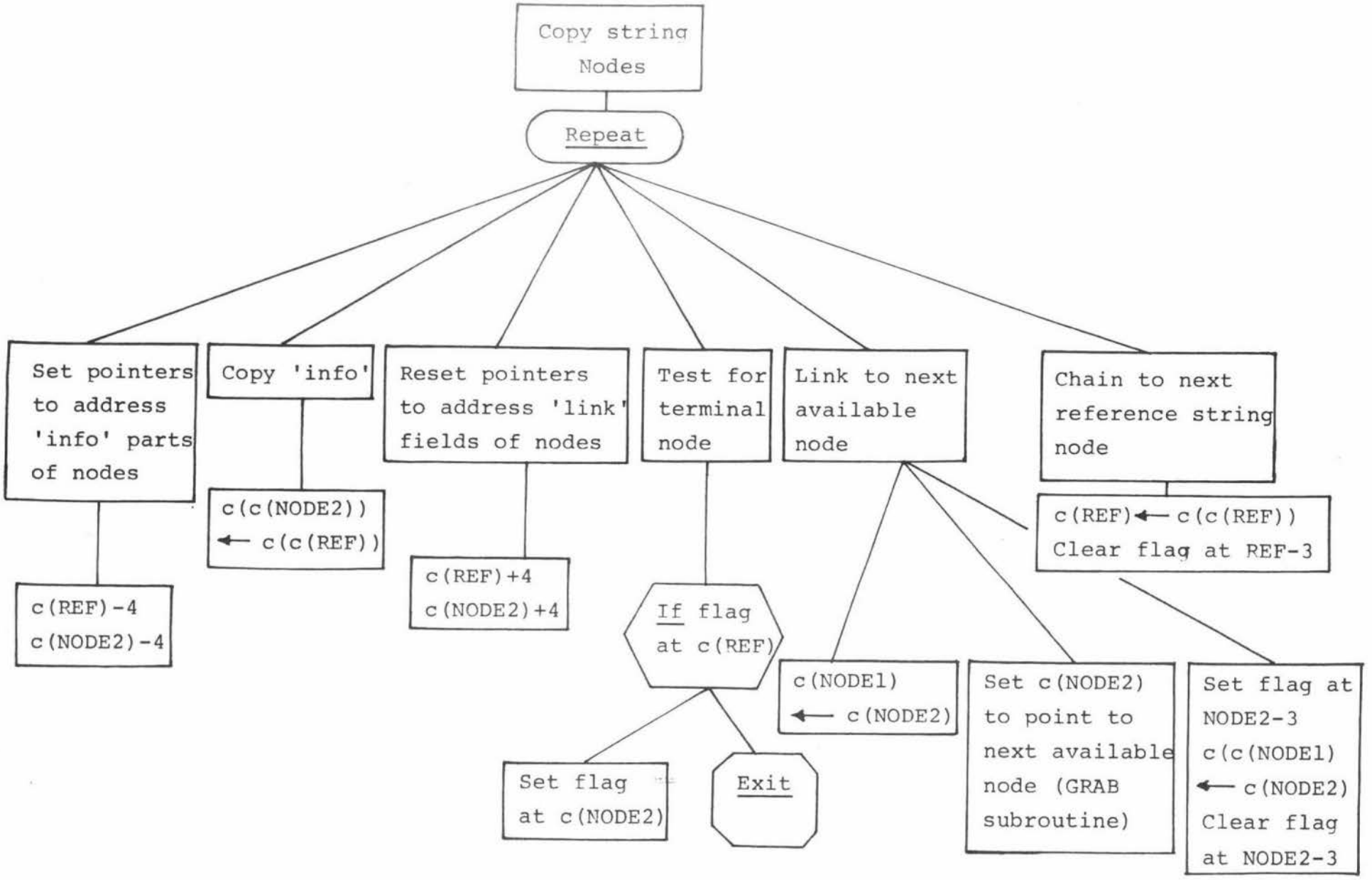
OP 03


TAKE RESULT
Routine
opcode parameter
T R x̄ x x
c(IC)
Increment c(IC) to point to Symbol Table address parameter
Place 'TR' code and address parameter in PBUF (for trace)
Increment Stack pointers
Convert Symbol Table address into absolute address and assign to c(A)
Place addressed result and type at top of Stack (PUSH Routine)
c(IC) + 3
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
c(A)←c(SYMBAS) +(14xc(c(IC)))

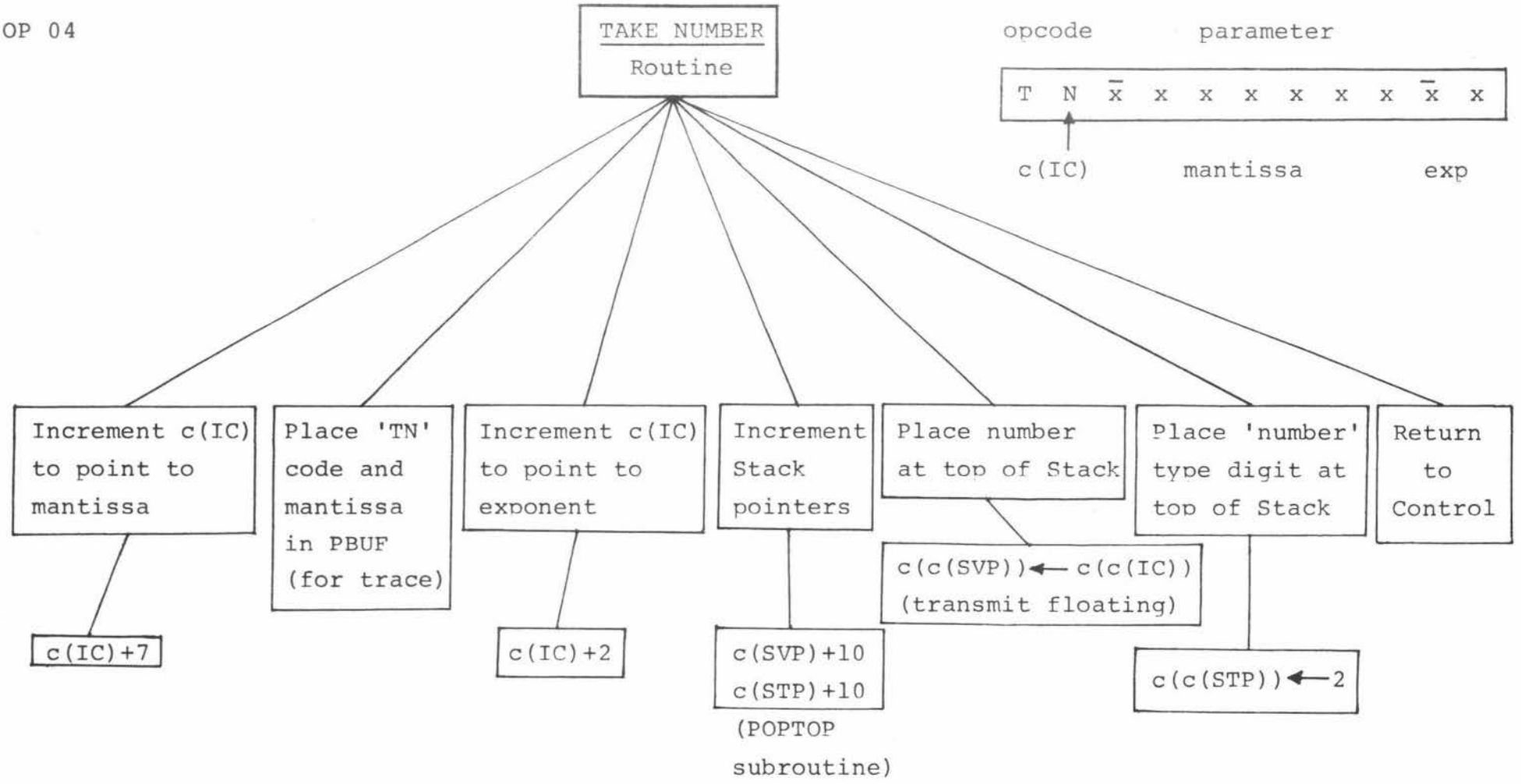

PUSH Routine to place result (& type) addressed by c(A) at top of Stack
Decrement c(A) to address type digit
Transmit type digit to 2-digit field (for comparison)
If c(COMP)=0
Place type digit at top of Stack
Increment c(A) to address value part
Place value at top of Stack
Return to Control
c(A)-9
c(COMP)← c(c(A))
ERROR (TR E1) Undefined variable
c(c(STP))← c(COMP)
c(A)+9


Place value at top of Stack
Choice
on c(COMP)
1
(null string)
2
(number)
3
(boolean)
7
(string descriptor)
other
No operation
c(c(SVP)) ← c(c(A))
(transmit floating)
c(c(SVP)) ← c(c(A))
Copy addressed string in string area and place descriptor (for copy) at top of Stack
ERROR
(TR E2)
Illegal type digit

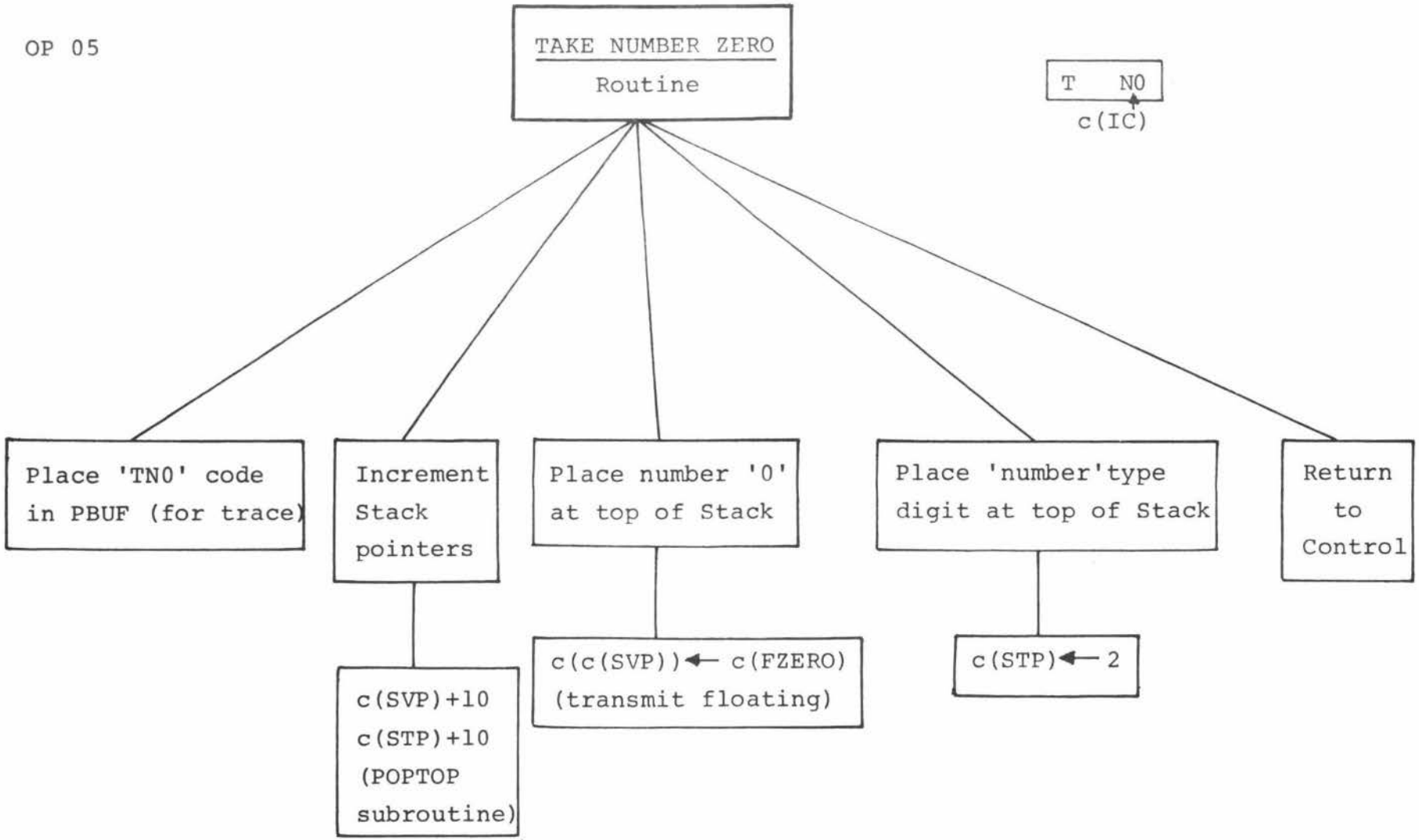

Copy addressed string in string area and place corresponding descriptor at top of Stack
Set c(REF) to point to first string node
Set c(NODE2) to point to next available node (GRAB Subroutine)
Place address of first node of copied string at top of Stack
Fill length field in descriptor at top of Stack
Copy string nodes
c(REF) ← c(c(A))
c(c(SVP)) ← c(NODE2)
c(A)-5
c(SVP)-5
c(c(SVP)) ← c(c(A))
c(SVP)+5


Copy string Nodes
Repeat
Set pointers to address 'info' parts of nodes
c(REF)-4
c(NODE2)-4
Copy 'info'
c(c(NODE2)) ← c(c(REF))
Reset pointers to address 'link' fields of nodes
c(REF)+4
c(NODE2)+4
Test for terminal node
If flag at c(REF)
Set flag at c(NODE2)
Exit
Link to next available node
c(NODE1) ← c(NODE2)
Set c(NODE2) to point to next available node (GRAB subroutine)
Chain to next reference string node
c(REF) ← c(c(REF))
Clear flag at REF-3
Set flag at NODE2-3
c(c(NODE1) ← c(NODE2)
Clear flag at NODE2-3

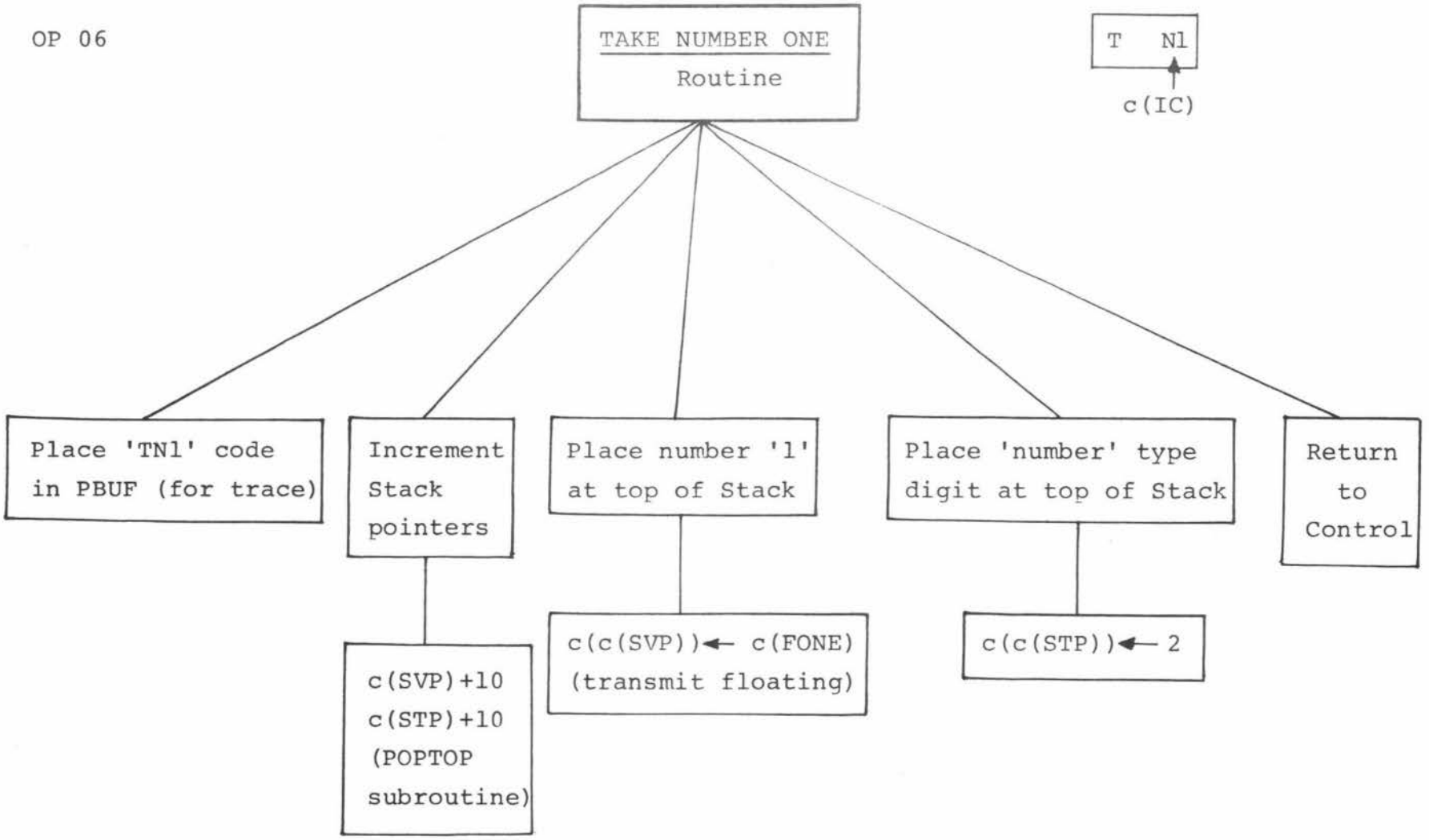
OP 04


TAKE NUMBER
Routine
opcode
parameter
T N x̄ x x x x x x x̄ x
c(IC)
mantissa
exp
Increment c(IC) to point to mantissa
Place 'TN' code and mantissa in PBUF (for trace)
Increment c(IC) to point to exponent
Increment Stack pointers
Place number at top of Stack
Place 'number' type digit at top of Stack
Return to Control
c(IC)+7
c(IC)+2
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
c(c(SVP)) ← c(c(IC))
(transmit floating)
c(c(STP)) ← 2

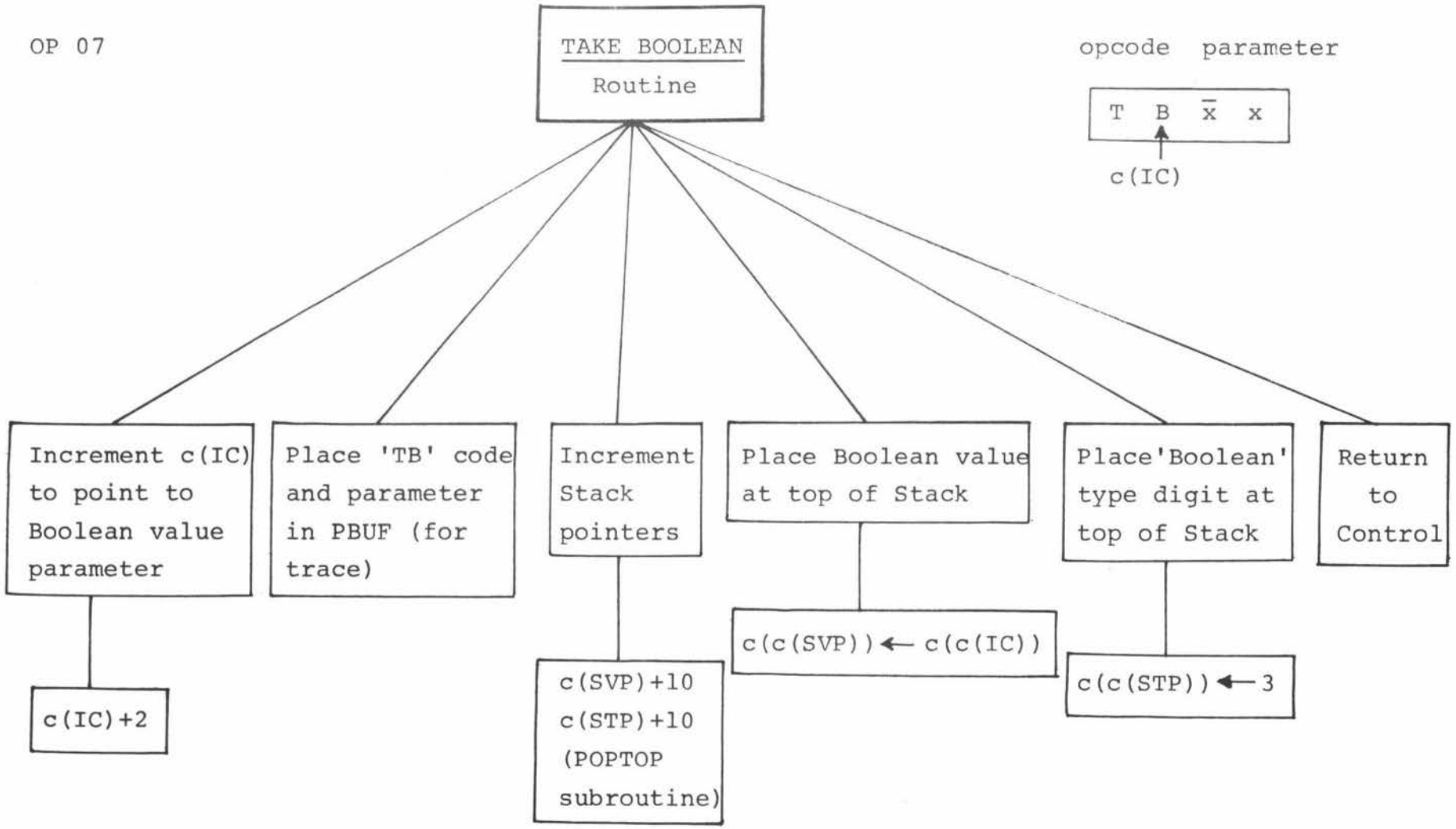
OP 05

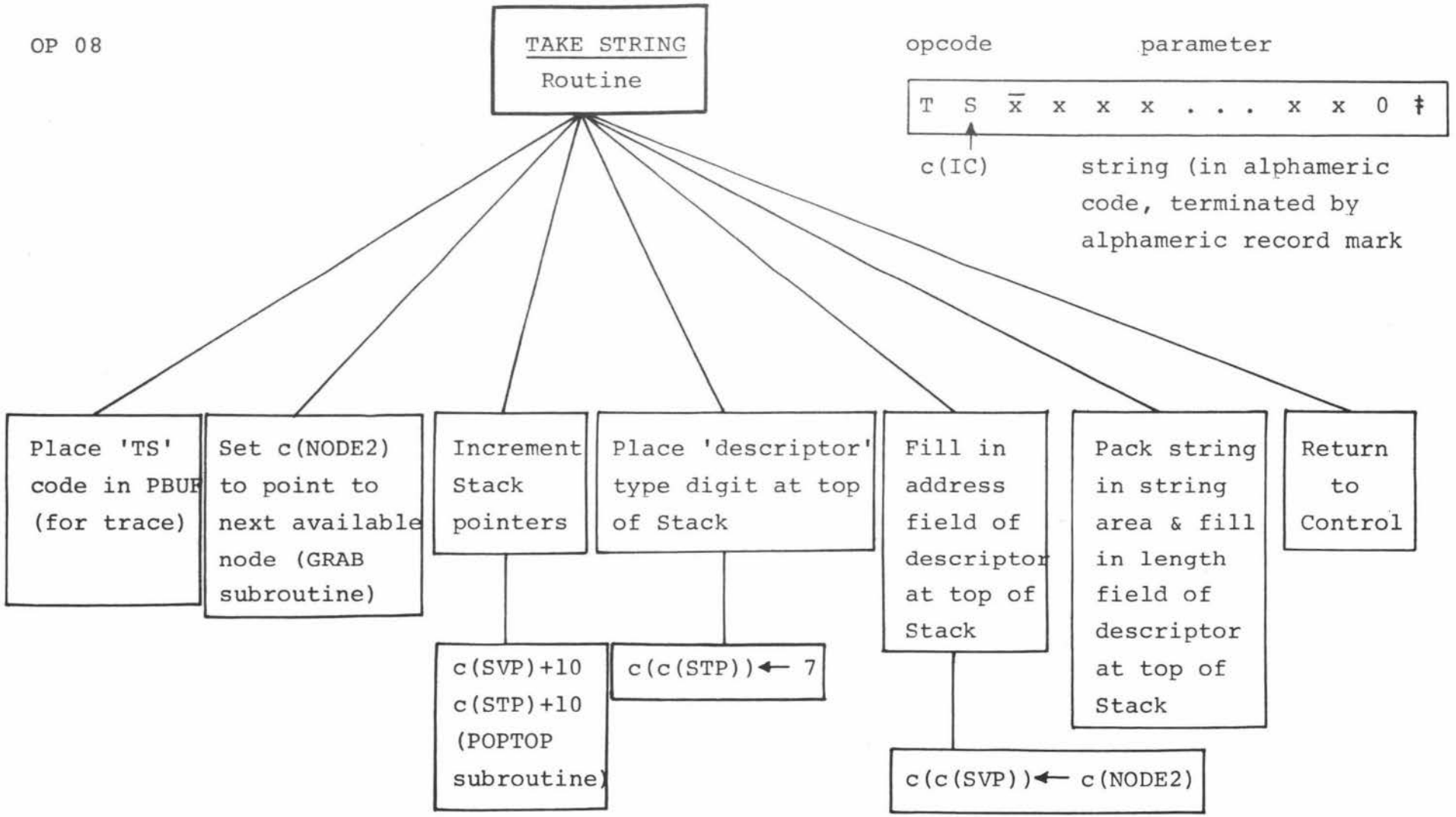

TAKE NUMBER ZERO
Routine
T N0
c(IC)
Place 'TN0' code in PBUF (for trace)
Increment Stack pointers
Place number '0' at top of Stack
Place 'number'type digit at top of Stack
Return to Control
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
c(c(SVP)) ← c(FZERO)
(transmit floating)
c(STP) ← 2

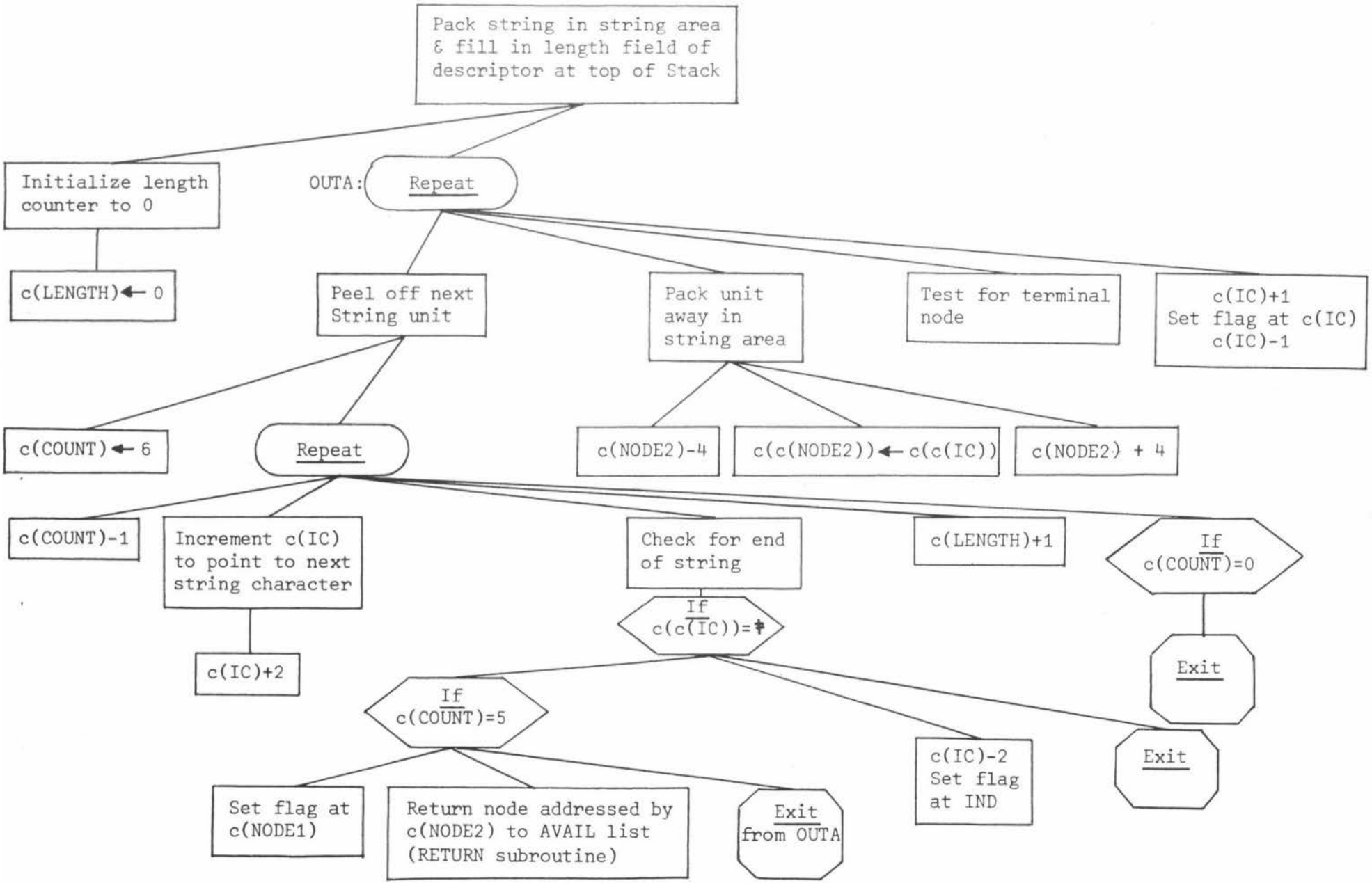
OP 06

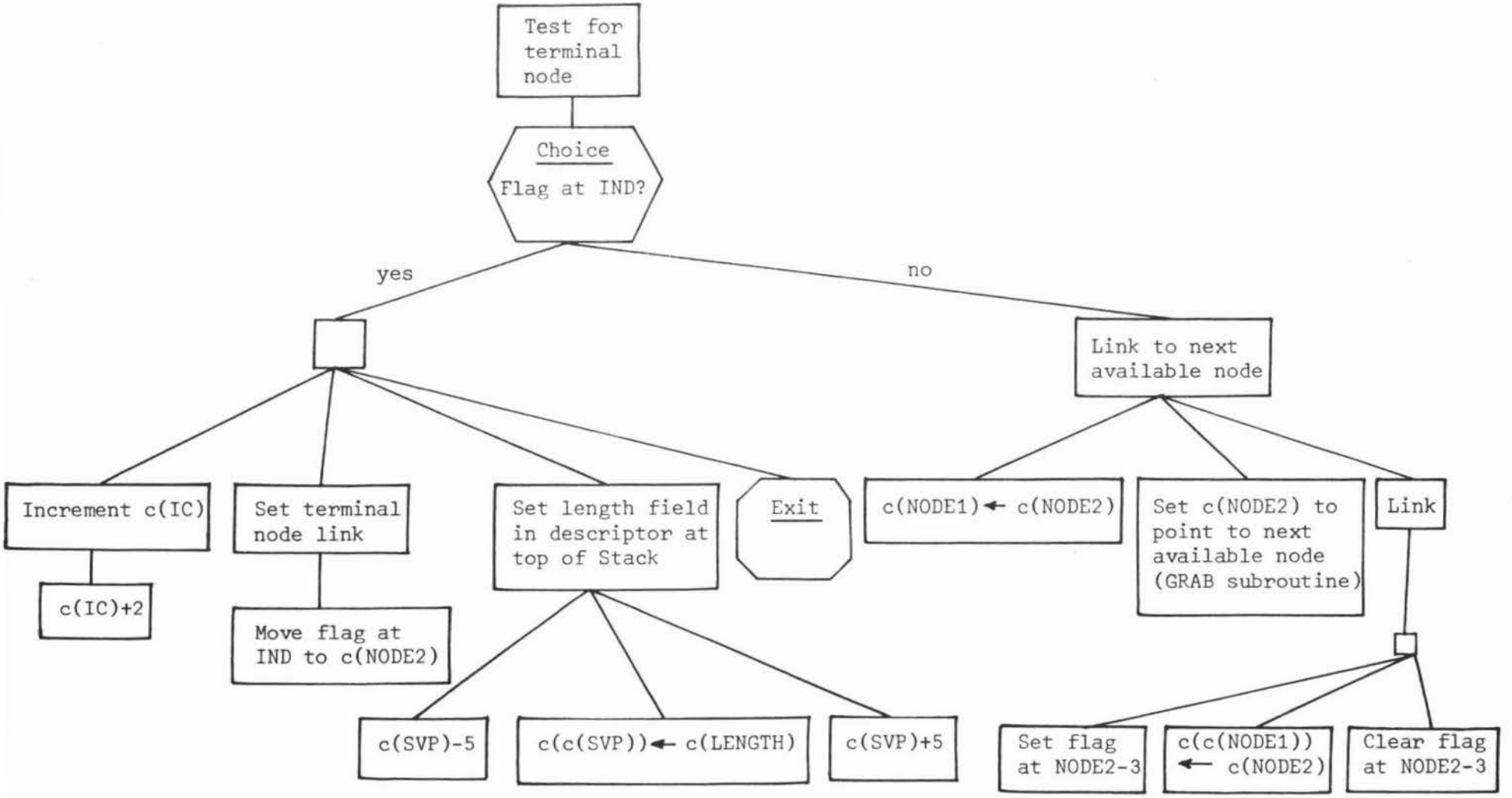

TAKE NUMBER ONE
Routine
T Nl
c(IC)
Place 'TNl' code in PBUF (for trace)
Increment Stack pointers
Place number '1' at top of Stack
Place 'number' type digit at top of Stack
Return to Control
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
c(c(SVP))← c(FONE)
(transmit floating)
c(c(STP))← 2

OP 07

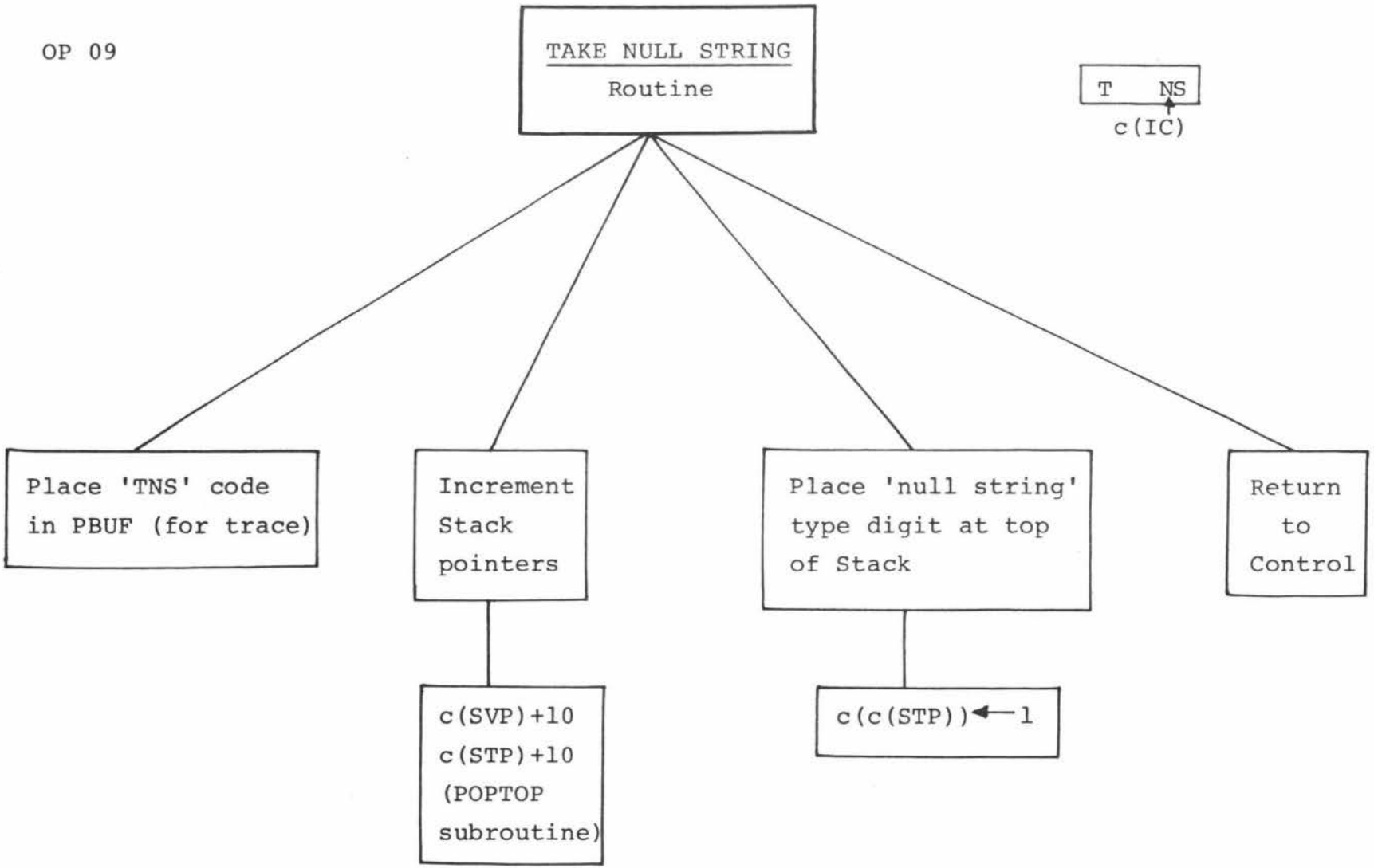

TAKE BOOLEAN
Routine
opcode parameter
T B x̄ x
c(IC)
Increment c(IC) to point to Boolean value parameter
c(IC)+2
Place 'TB' code and parameter in PBUF (for trace)
Increment Stack pointers
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
Place Boolean value at top of Stack
c(c(SVP)) ← c(c(IC))
Place'Boolean' type digit at top of Stack
c(c(STP)) ← 3
Return to Control


OP 08
TAKE STRING
Routine
opcode
parameter
T S x̄ x x x . . . x x 0 ǂ
c(IC)
string (in alphameric code, terminated by alphameric record mark
Place 'TS' code in PBUR (for trace)
Set c(NODE2) to point to next available node (GRAB subroutine)
Increment Stack pointers
Place 'descriptor' type digit at top of Stack
Fill in address field of descriptor at top of Stack
Pack string in string area & fill in length field of descriptor at top of Stack
Return to Control
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
c(c(STP))← 7
c(c(SVP))← c(NODE2)


Pack string in string area & fill in length field of descriptor at top of Stack
Initialize length counter to 0
OUTA:
Repeat
c(LENGTH)← 0
Peel off next String unit
Pack unit away in string area
Test for terminal node
c(IC)+1
Set flag at c(IC)
c(IC)-1
c(COUNT)← 6
Repeat
c(NODE2)-4
c(c(NODE2))← c(c(IC))
c(NODE2) + 4
c(COUNT)-1
Increment c(IC) to point to next string character
c(IC)+2
Check for end of string
If c(c(IC))=#
c(LENGTH)+1
If c(COUNT)=0
Exit
If c(COUNT)=5
Set flag at c(NODE1)
Return node addressed by c(NODE2) to AVAIL list (RETURN subroutine)
Exit from OUTA
c(IC)-2
Set flag at IND
Exit


Test for terminal node
Choice
Flag at IND?
yes
no
Increment c(IC)
c(IC)+2
Set terminal node link
Move flag at IND to c(NODE2)
Set length field in descriptor at top of Stack
c(SVP)-5
c(c(SVP))← c(LENGTH)
c(SVP)+5
Exit
Link to next available node
c(NODE1)← c(NODE2)
Set c(NODE2) to point to next available node (GRAB subroutine)
Link
Set flag at NODE2-3
c(c(NODE1)) ← c(NODE2)
Clear flag at NODE2-3

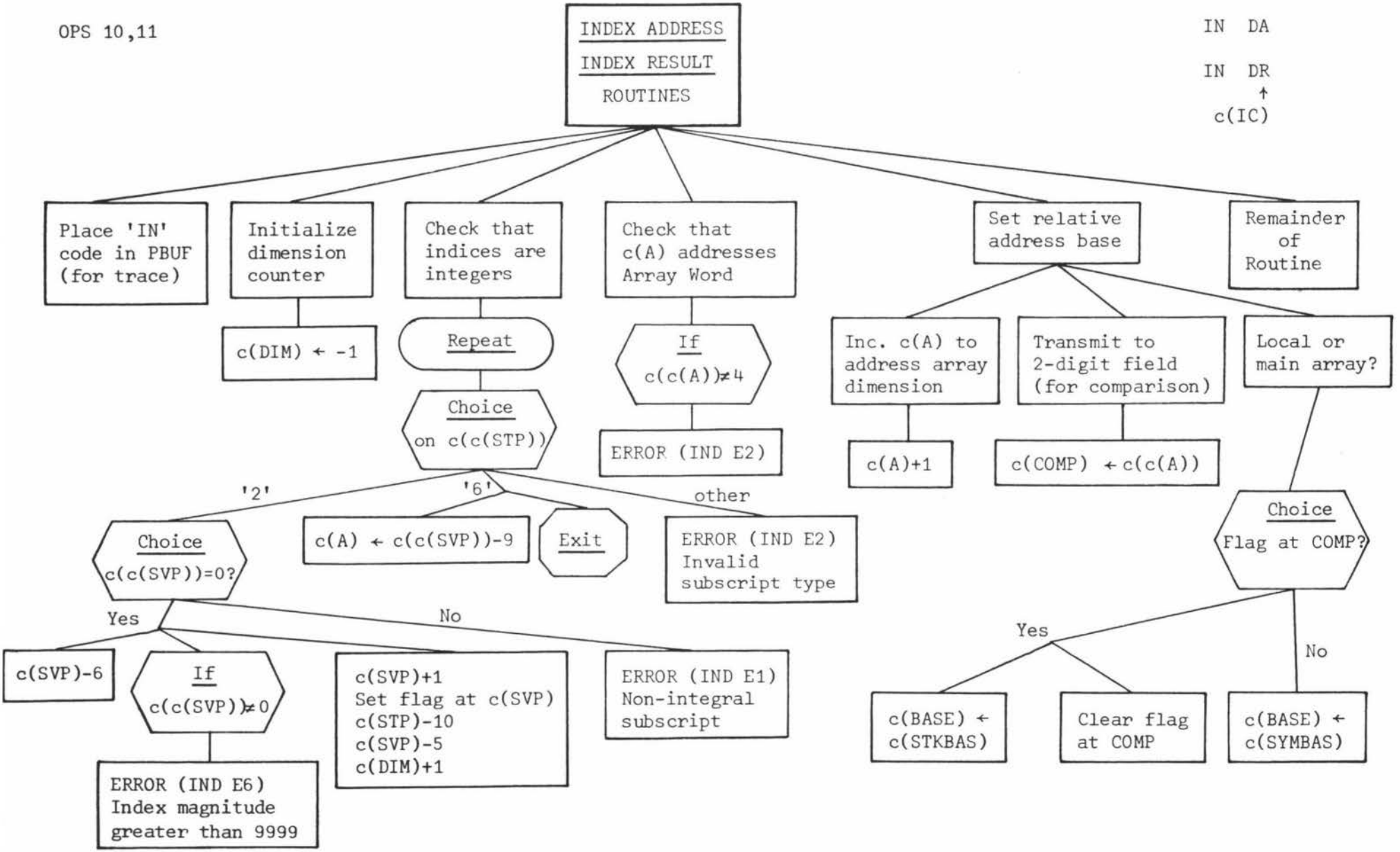
OP 09

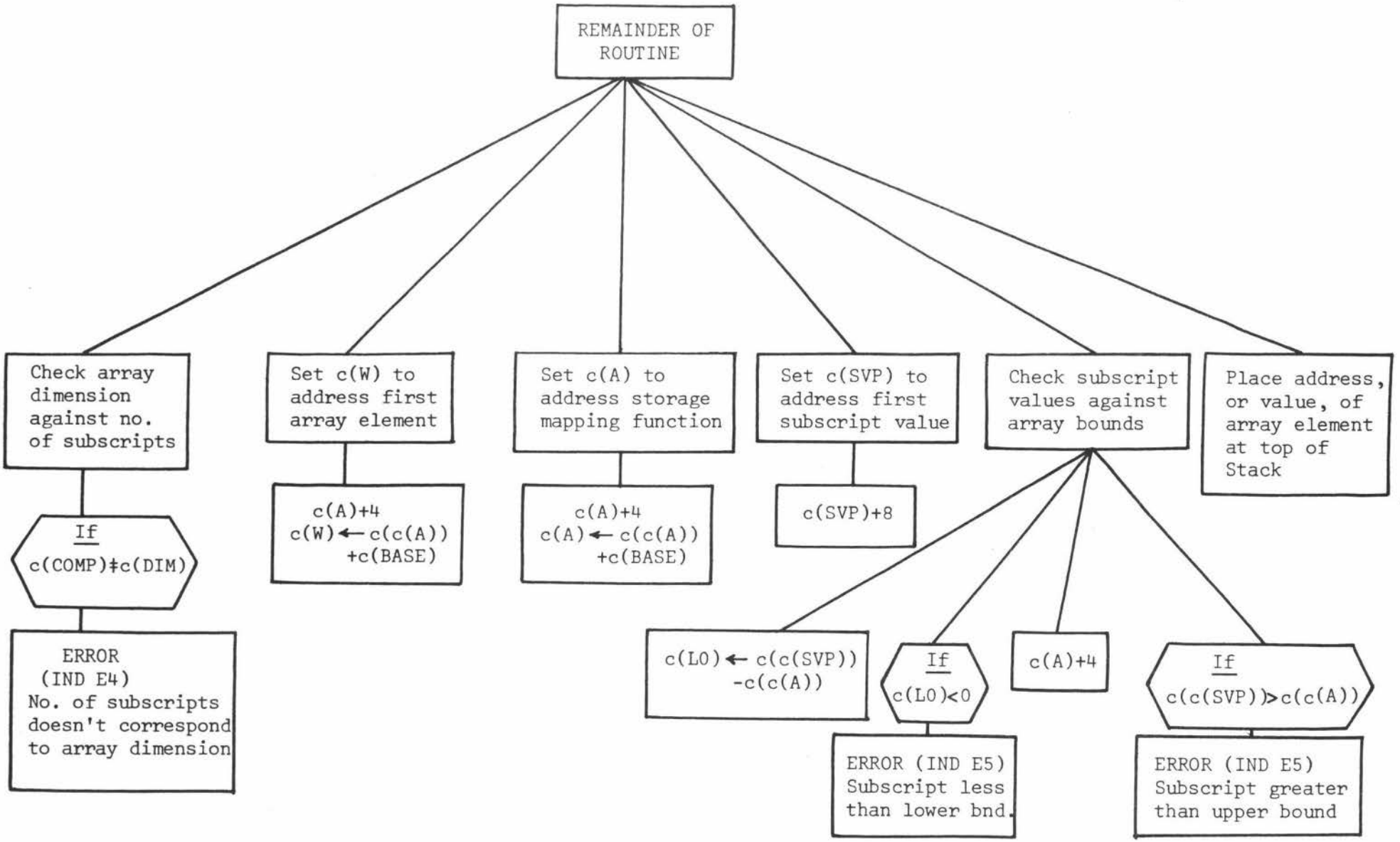

TAKE NULL STRING
Routine
T NS
c(IC)
Place 'TNS' code in PBUF (for trace)
Increment Stack pointers
Place 'null string' type digit at top of Stack
Return to Control
c(SVP)+10
c(STP)+10
(POPTOP subroutine)
c(c(STP))←1

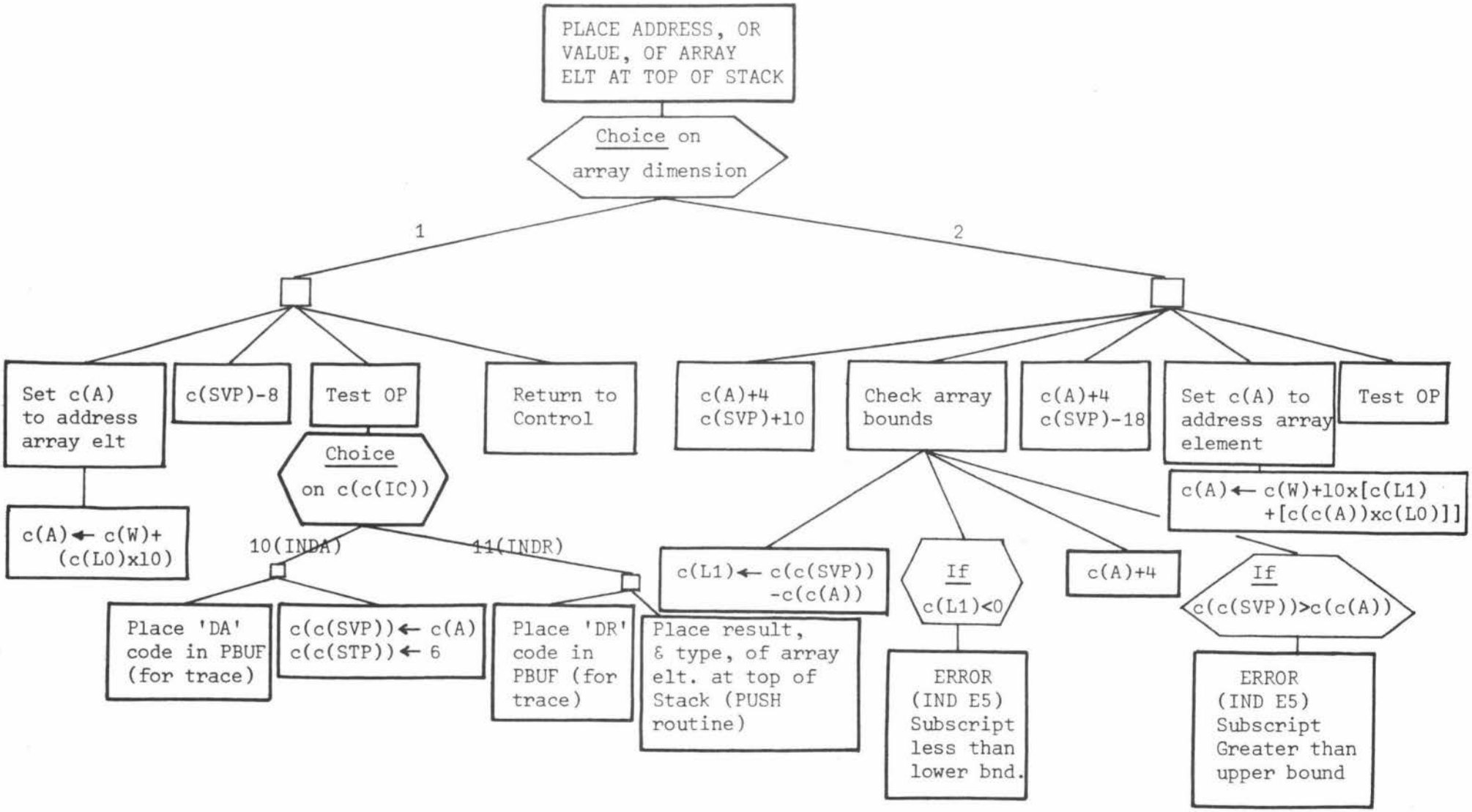
OPS 10,11

IN DA

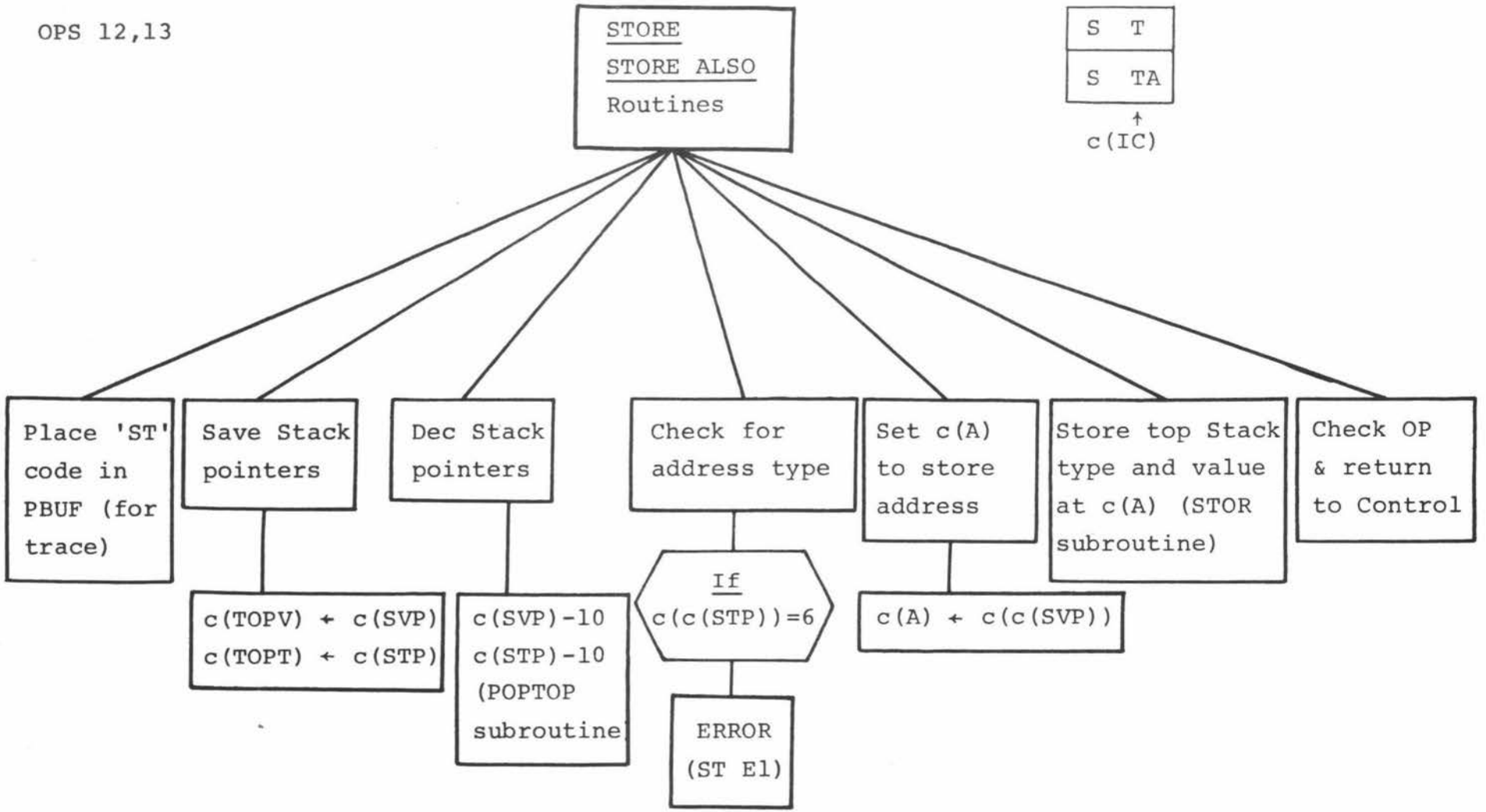
IN DR
↑
c(IC)

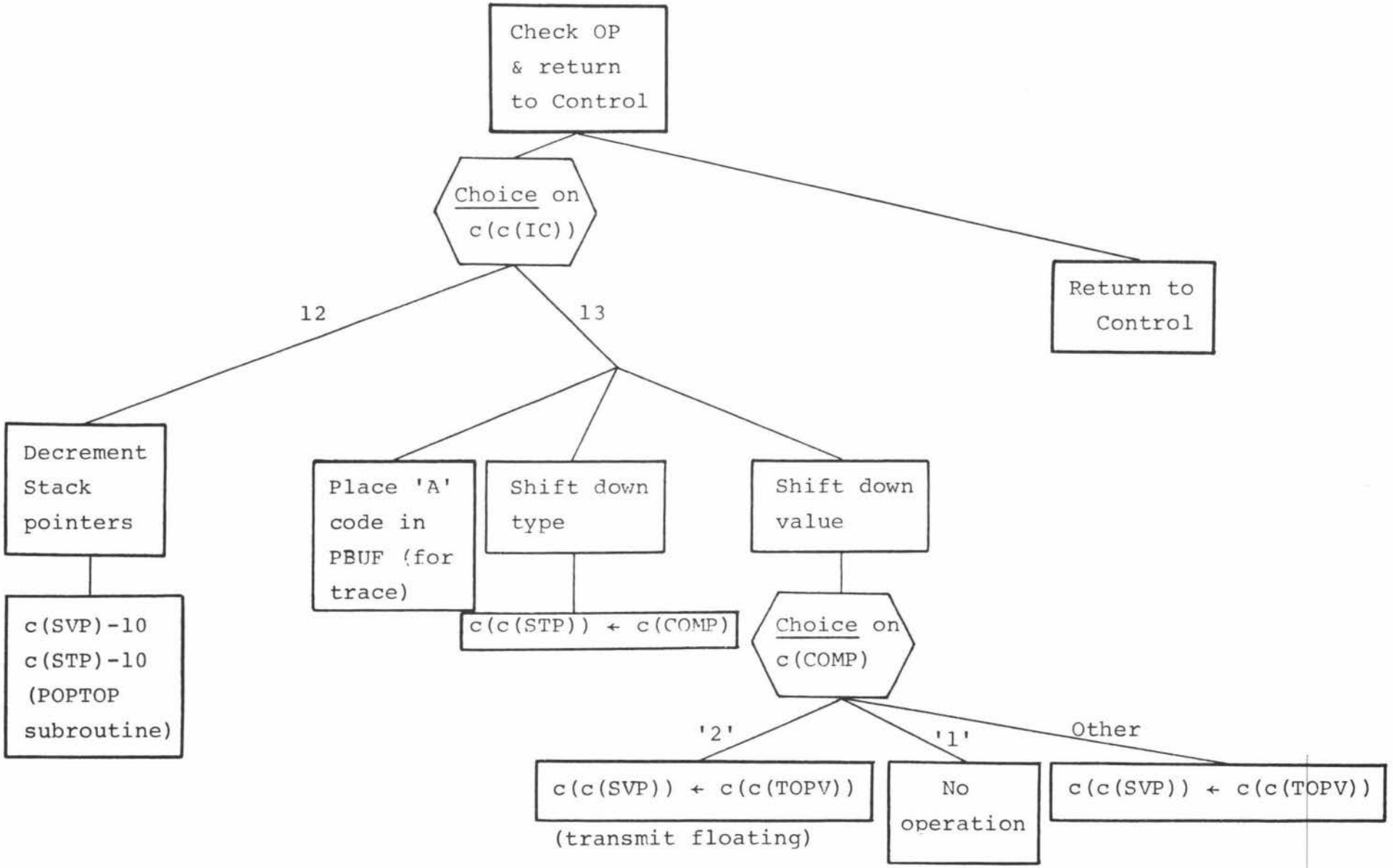


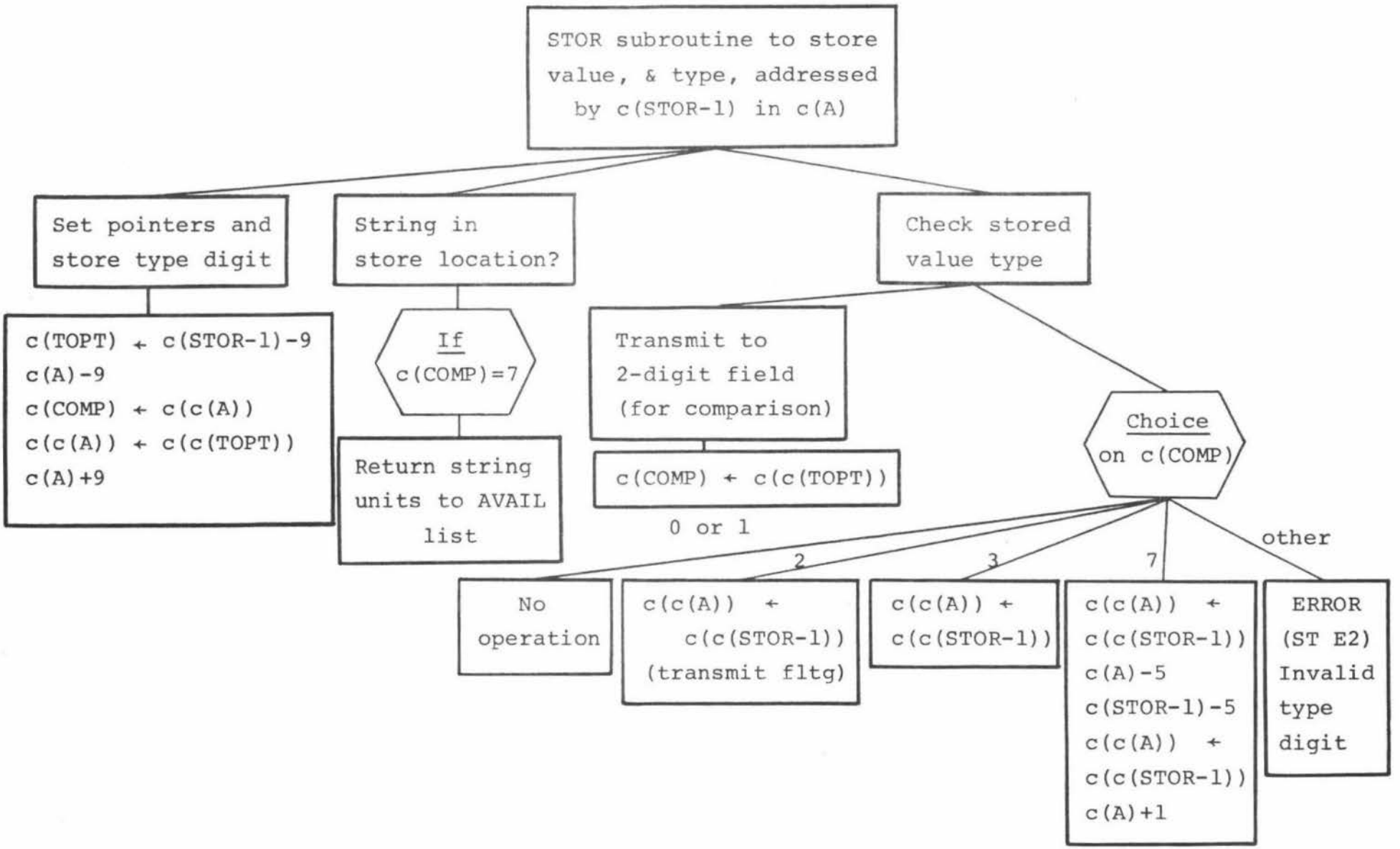

REMAINDER OF ROUTINE
Check array dimension against no. of subscripts
If c(COMP)≠c(DIM)
ERROR (IND E4) No. of subscripts doesn't correspond to array dimension
Set c(W) to address first array element
c(A)+4 c(W)←c(c(A)) +c(BASE)
Set c(A) to address storage mapping function
c(A)+4 c(A)←c(c(A)) +c(BASE)
Set c(SVP) to address first subscript value
c(SVP)+8
Check subscript values against array bounds
c(L0)←c(c(SVP)) -c(c(A))
If c(L0)<0
ERROR (IND E5) Subscript less than lower bnd.
c(A)+4
If c(c(SVP))>c(c(A))
ERROR (IND E5) Subscript greater than upper bound
Place address, or value, of array element at top of Stack

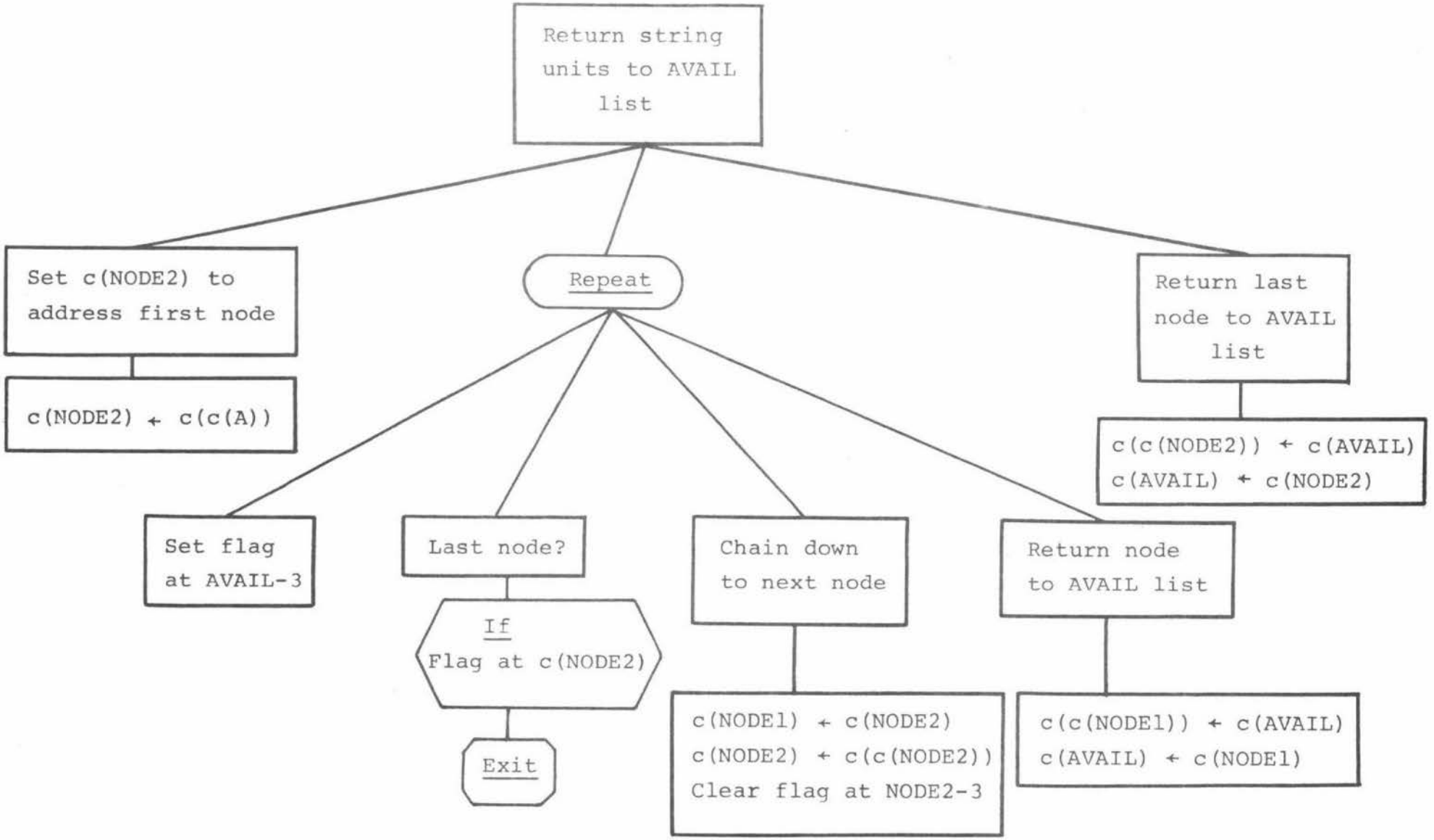

PLACE ADDRESS, OR VALUE, OF ARRAY ELT AT TOP OF STACK
Choice on array dimension
1
2
Set c(A) to address array elt
c(SVP)-8
Test OP
Return to Control
c(A)+4 c(SVP)+10
Check array bounds
c(A)+4 c(SVP)-18
Set c(A) to address array element
Test OP
c(A)← c(W)+ (c(L0)x10)
Choice on c(c(IC))
c(A)← c(W)+10x[c(L1) +[c(c(A))xc(L0)]]
10(INDA)
11(INDR)
c(L1)← c(c(SVP)) -c(c(A))
If c(L1)<0
c(A)+4
If c(c(SVP))>c(c(A))
Place 'DA' code in PBUF (for trace)
c(c(SVP))← c(A) c(c(STP))← 6
Place 'DR' code in PBUF (for trace)
Place result, & type, of array elt. at top of Stack (PUSH routine)
ERROR (IND E5) Subscript less than lower bnd.
ERROR (IND E5) Subscript Greater than upper bound

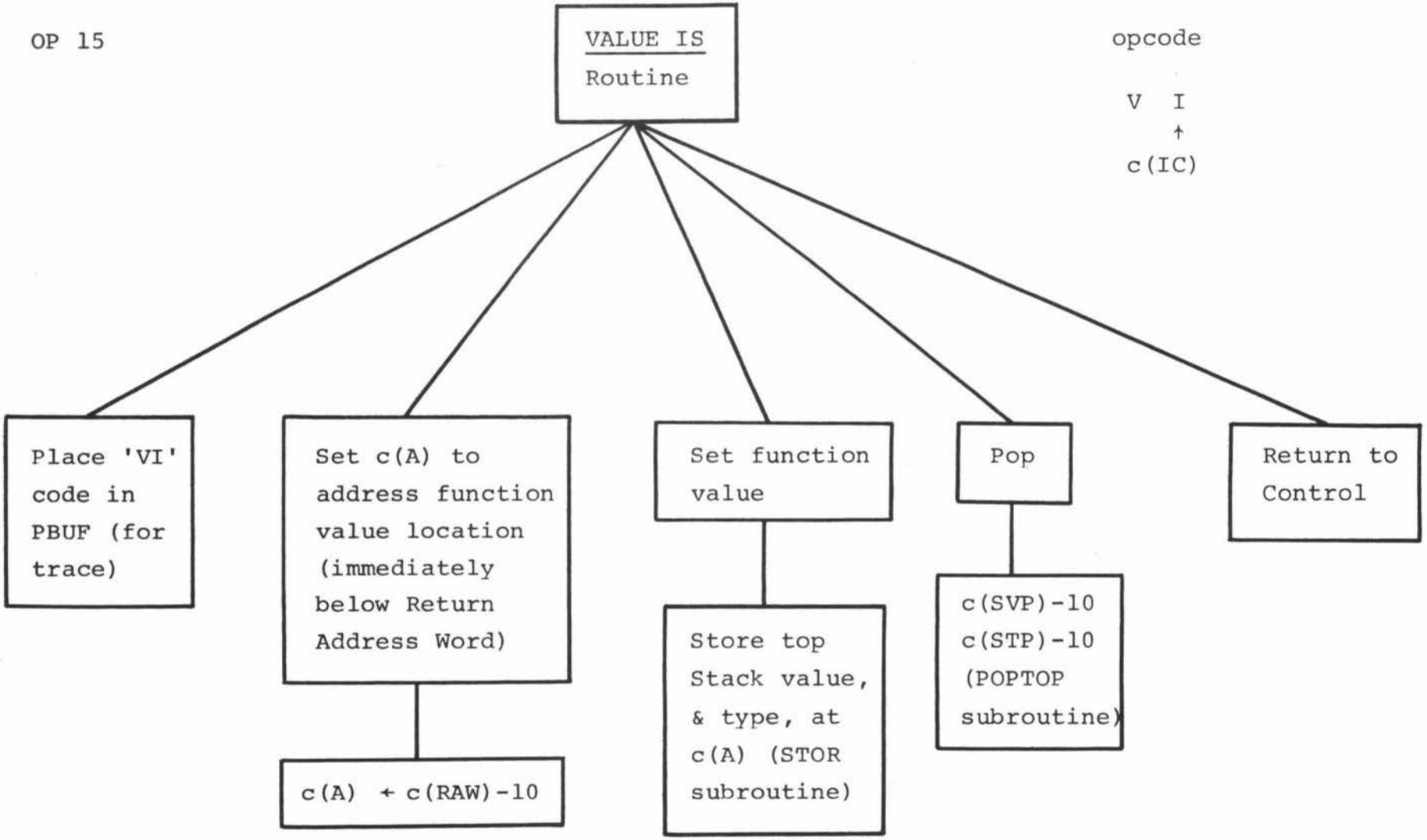
OPS 12,13


STORE
STORE ALSO
Routines
S T
S TA
c(IC)
Place 'ST' code in PBUF (for trace)
Save Stack pointers
Dec Stack pointers
Check for address type
Set c(A) to store address
Store top Stack type and value at c(A) (STOR subroutine)
Check OP & return to Control
c(TOPV) ← c(SVP)
c(TOPT) ← c(STP)
c(SVP)-10
c(STP)-10
(POPTOP subroutine
If
c(c(STP))=6
c(A) ← c(c(SVP))
ERROR
(ST E1)

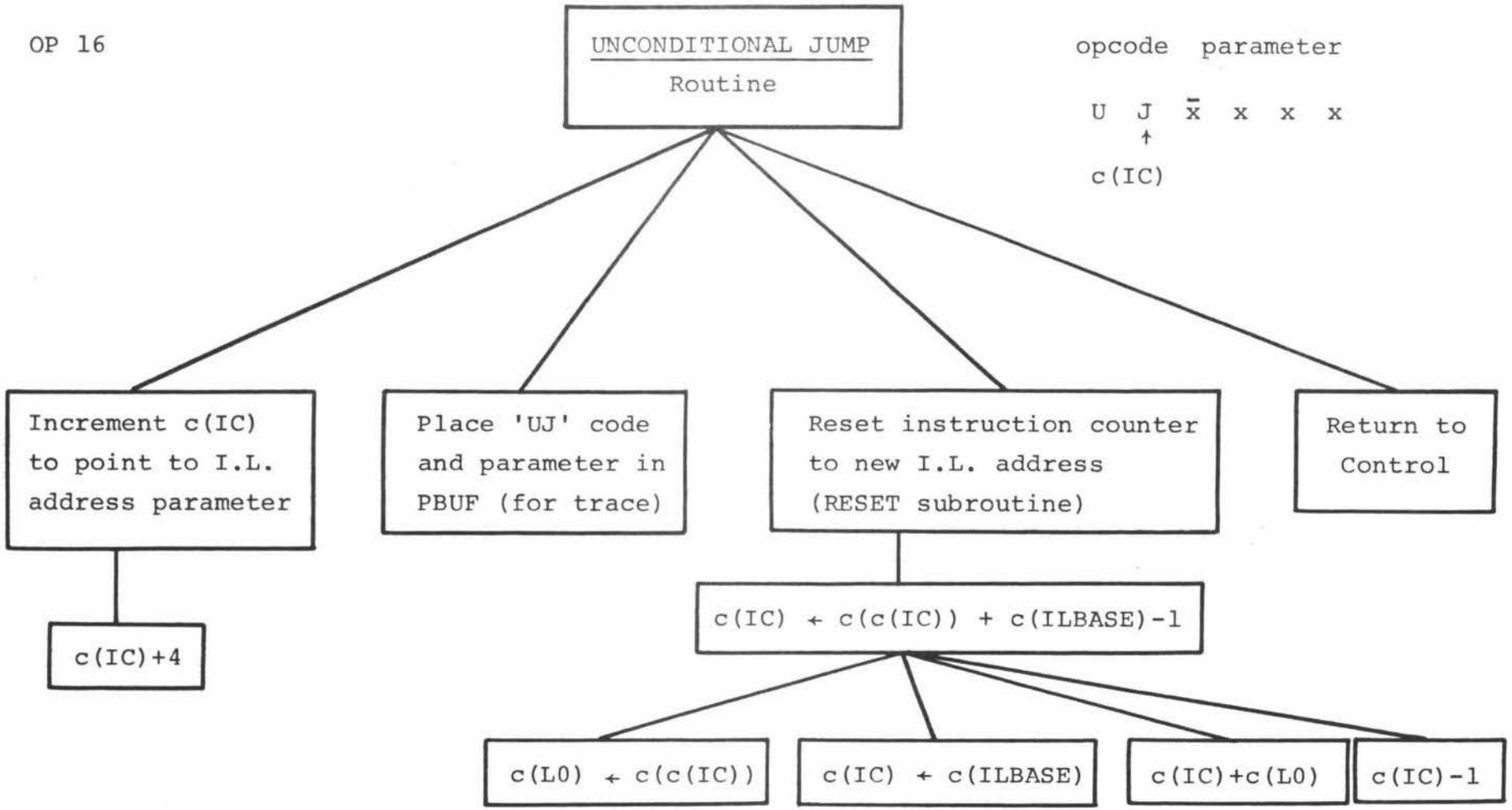

Check OP & return to Control
Choice on c(c(IC))
Return to Control
12
13
Decrement Stack pointers
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
Place 'A' code in PBUF (for trace)
Shift down type
c(c(STP)) ← c(COMP)
Shift down value
Choice on c(COMP)
'2'
'1'
Other
c(c(SVP)) ← c(c(TOPV))
(transmit floating)
No operation
c(c(SVP)) ← c(c(TOPV))


STOR subroutine to store value, & type, addressed by c(STOR-1) in c(A)
Set pointers and store type digit
c(TOPT) ← c(STOR-1)-9
c(A)-9
c(COMP) ← c(c(A))
c(c(A)) ← c(c(TOPT))
c(A)+9
String in store location?
If c(COMP)=7
Return string units to AVAIL list
Check stored value type
Transmit to 2-digit field (for comparison)
c(COMP) ← c(c(TOPT))
0 or 1
Choice on c(COMP)
2
3
7
other
No operation
c(c(A)) ← c(c(STOR-1)) (transmit fltg)
c(c(A)) ← c(c(STOR-1))
c(c(A)) ← c(c(STOR-1))
c(A)-5
c(STOR-1)-5
c(c(A)) ← c(c(STOR-1))
c(A)+1
ERROR (ST E2) Invalid type digit

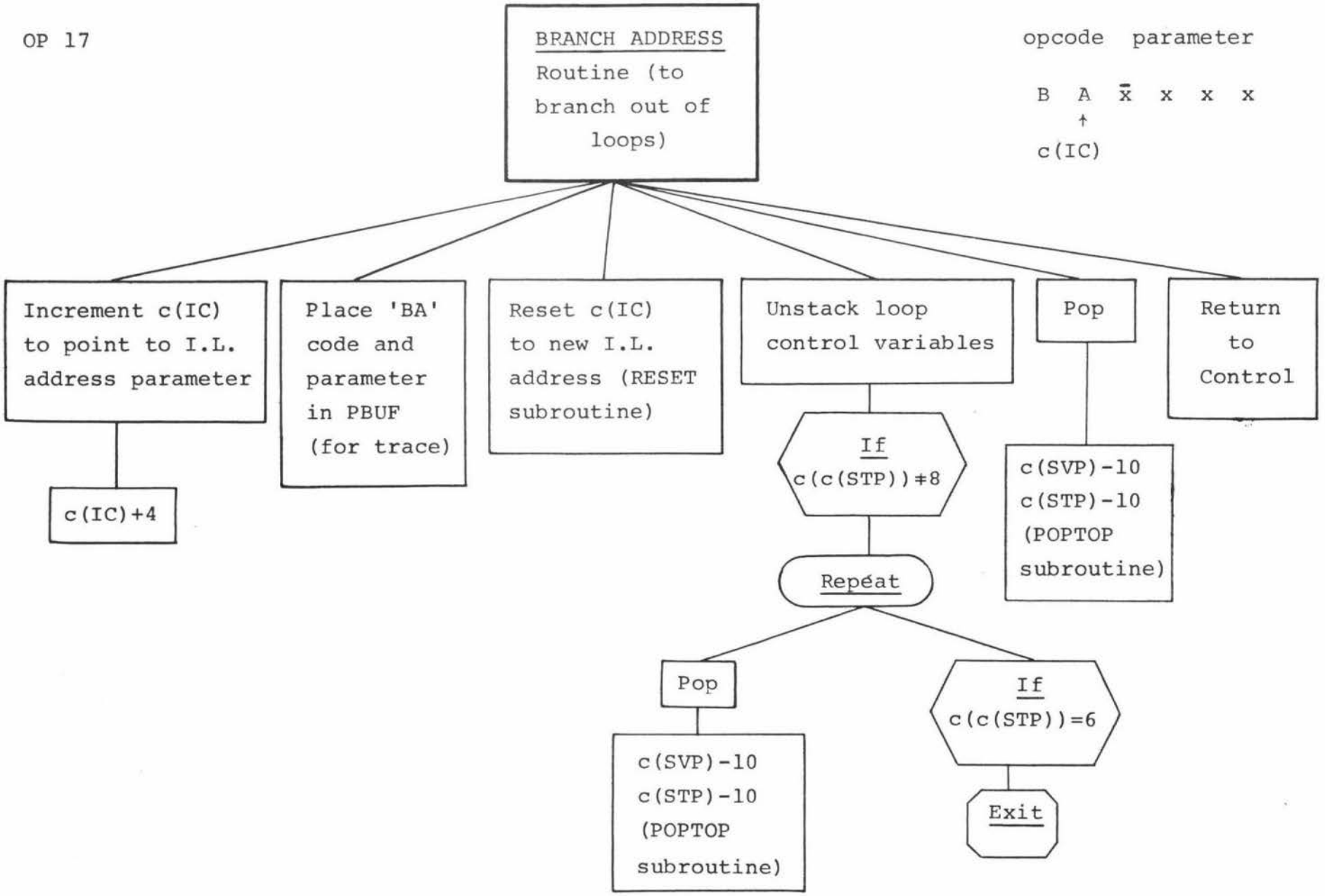

Return string units to AVAIL list
Set c(NODE2) to address first node
c(NODE2) ← c(c(A))
Repeat
Return last node to AVAIL list
c(c(NODE2)) ← c(AVAIL)
c(AVAIL) ← c(NODE2)
Set flag at AVAIL-3
Last node?
If Flag at c(NODE2)
Exit
Chain down to next node
c(NODE1) ← c(NODE2)
c(NODE2) ← c(c(NODE2))
Clear flag at NODE2-3
Return node to AVAIL list
c(c(NODE1)) ← c(AVAIL)
c(AVAIL) ← c(NODE1)


OP 15
VALUE IS
Routine
opcode
V I
c(IC)
Place 'VI' code in PBUF (for trace)
Set c(A) to address function value location (immediately below Return Address Word)
c(A) ← c(RAW)-10
Set function value
Store top Stack value, & type, at c(A) (STOR subroutine)
Pop
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
Return to Control

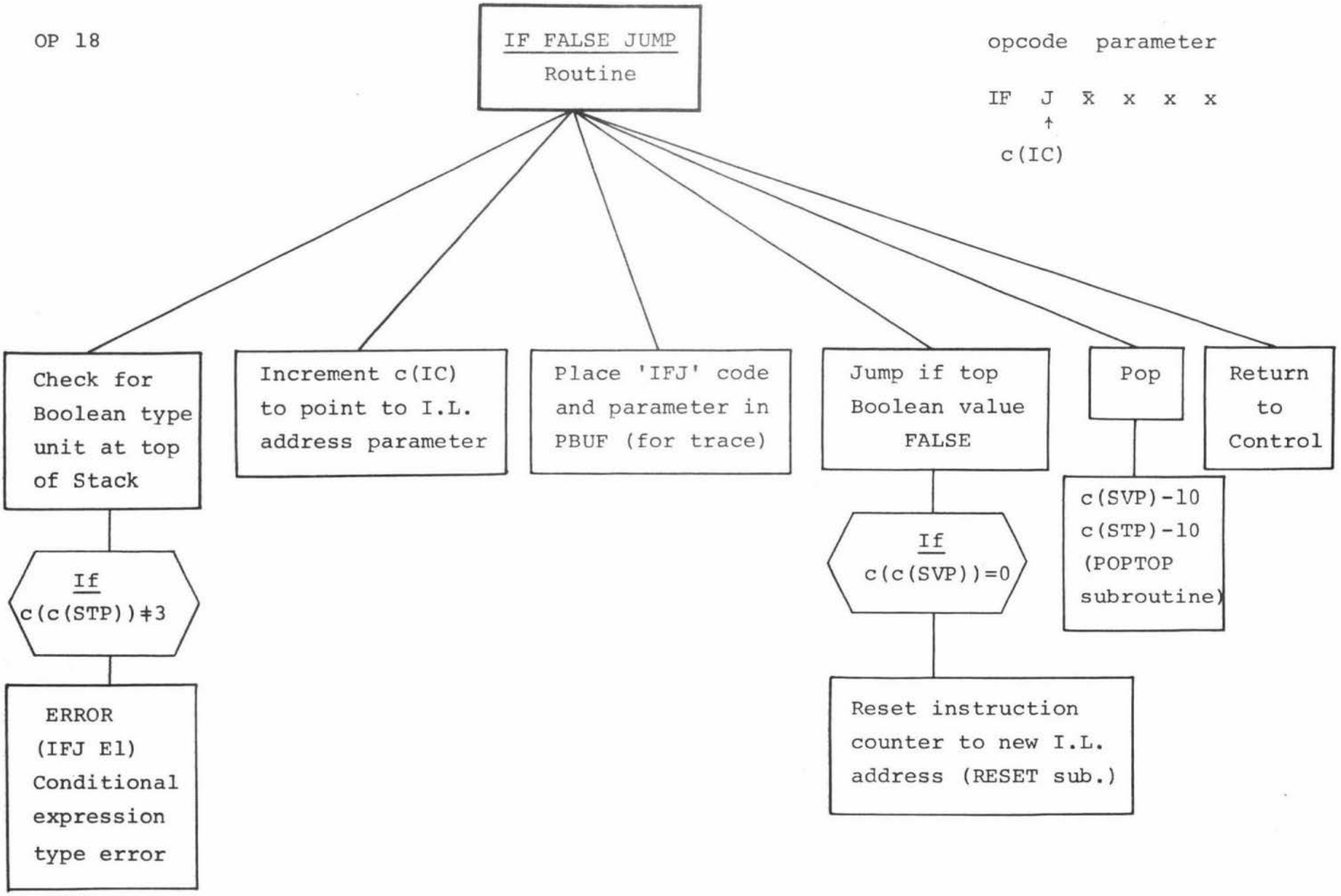
OP 16


UNCONDITIONAL JUMP
Routine
opcode parameter
U J x̄ x x x
c(IC)
Increment c(IC) to point to I.L. address parameter
Place 'UJ' code and parameter in PBUF (for trace)
Reset instruction counter to new I.L. address (RESET subroutine)
Return to Control
c(IC)+4
c(IC) ← c(c(IC)) + c(ILBASE)-1
c(L0) ← c(c(IC))
c(IC) ← c(ILBASE)
c(IC)+c(L0)
c(IC)-1

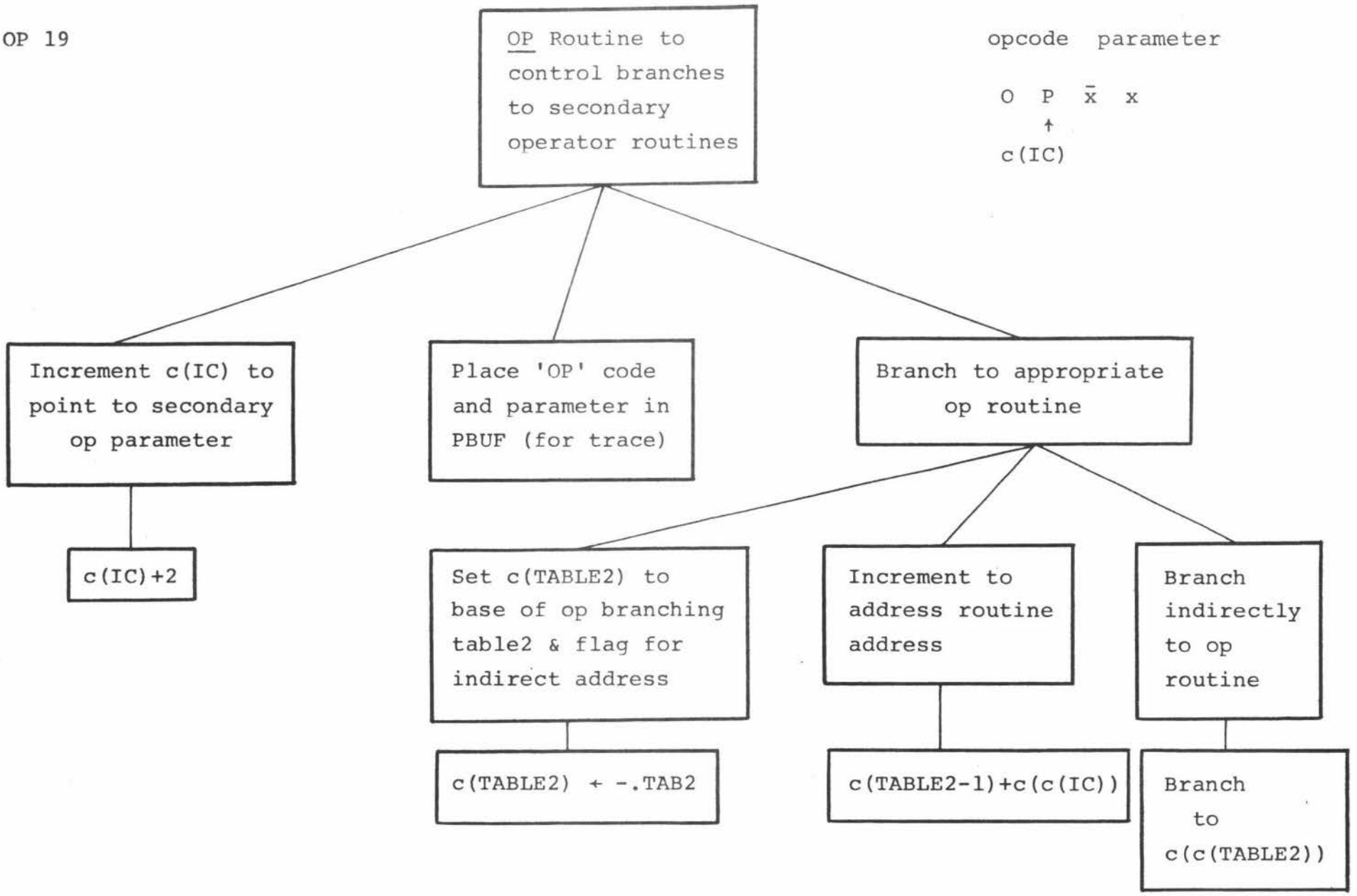
OP 17


BRANCH ADDRESS
Routine (to branch out of loops)
opcode parameter
B A x̄ x x x
c(IC)
Increment c(IC) to point to I.L. address parameter
c(IC)+4
Place 'BA' code and parameter in PBUF (for trace)
Reset c(IC) to new I.L. address (RESET subroutine)
Unstack loop control variables
If c(c(STP))≠8
Repeat
Pop
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
If c(c(STP))=6
Exit
Pop
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
Return to Control

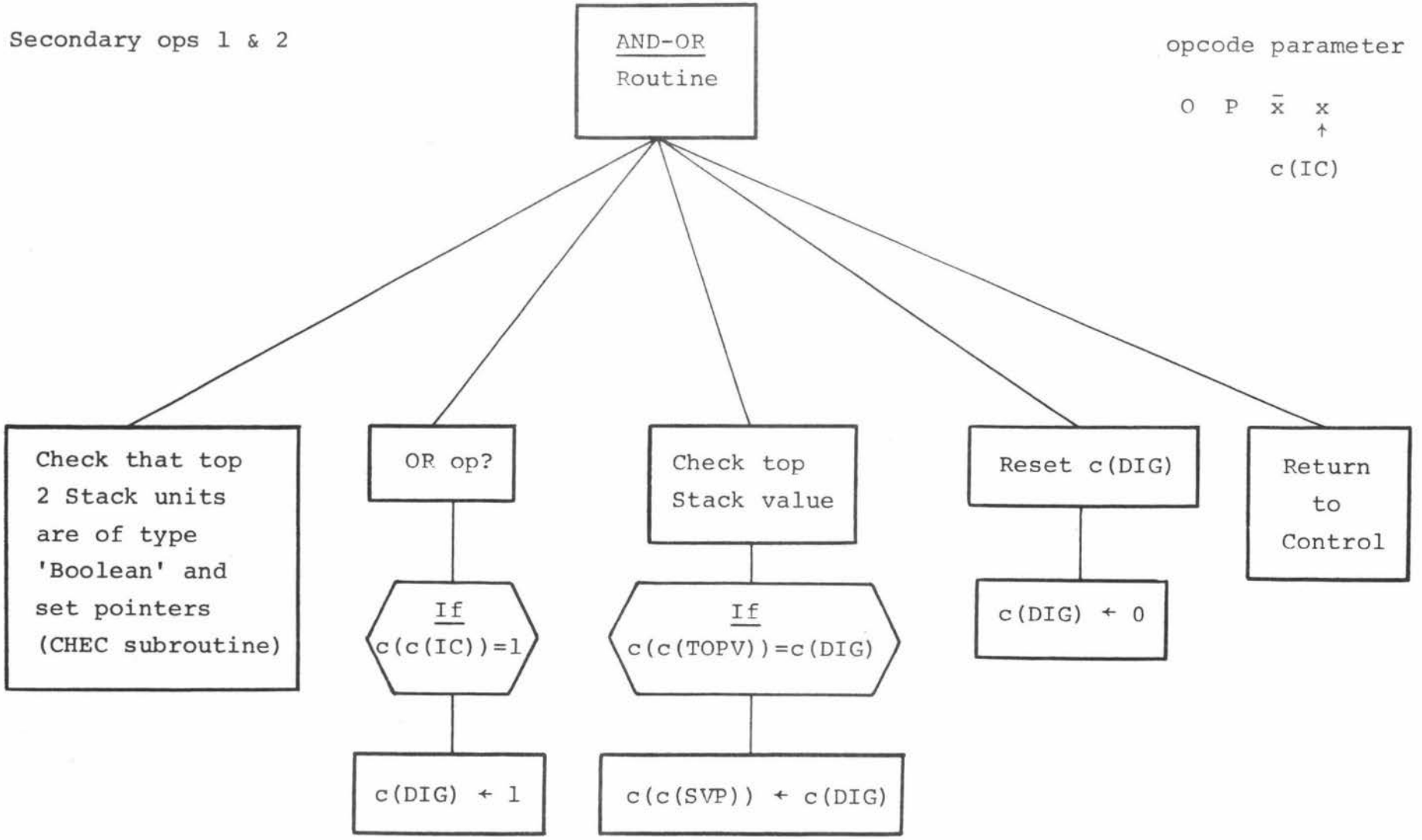
OP 18


IF FALSE JUMP
Routine
opcode parameter
IF J x̄ x x x
c(IC)
Check for Boolean type unit at top of Stack
Increment c(IC) to point to I.L. address parameter
Place 'IFJ' code and parameter in PBUF (for trace)
Jump if top Boolean value FALSE
Pop
Return to Control
If c(c(STP))≠3
ERROR (IFJ E1) Conditional expression type error
If c(c(SVP))=0
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
Reset instruction counter to new I.L. address (RESET sub.)

OP 19


OP Routine to control branches to secondary operator routines
opcode parameter
O P x̄ x
c(IC)
Increment c(IC) to point to secondary op parameter
Place 'OP' code and parameter in PBUF (for trace)
Branch to appropriate op routine
c(IC)+2
Set c(TABLE2) to base of op branching table2 & flag for indirect address
Increment to address routine address
Branch indirectly to op routine
c(TABLE2) ← -.TAB2
c(TABLE2-1)+c(c(IC))
Branch to c(c(TABLE2))

Secondary ops 1 & 2



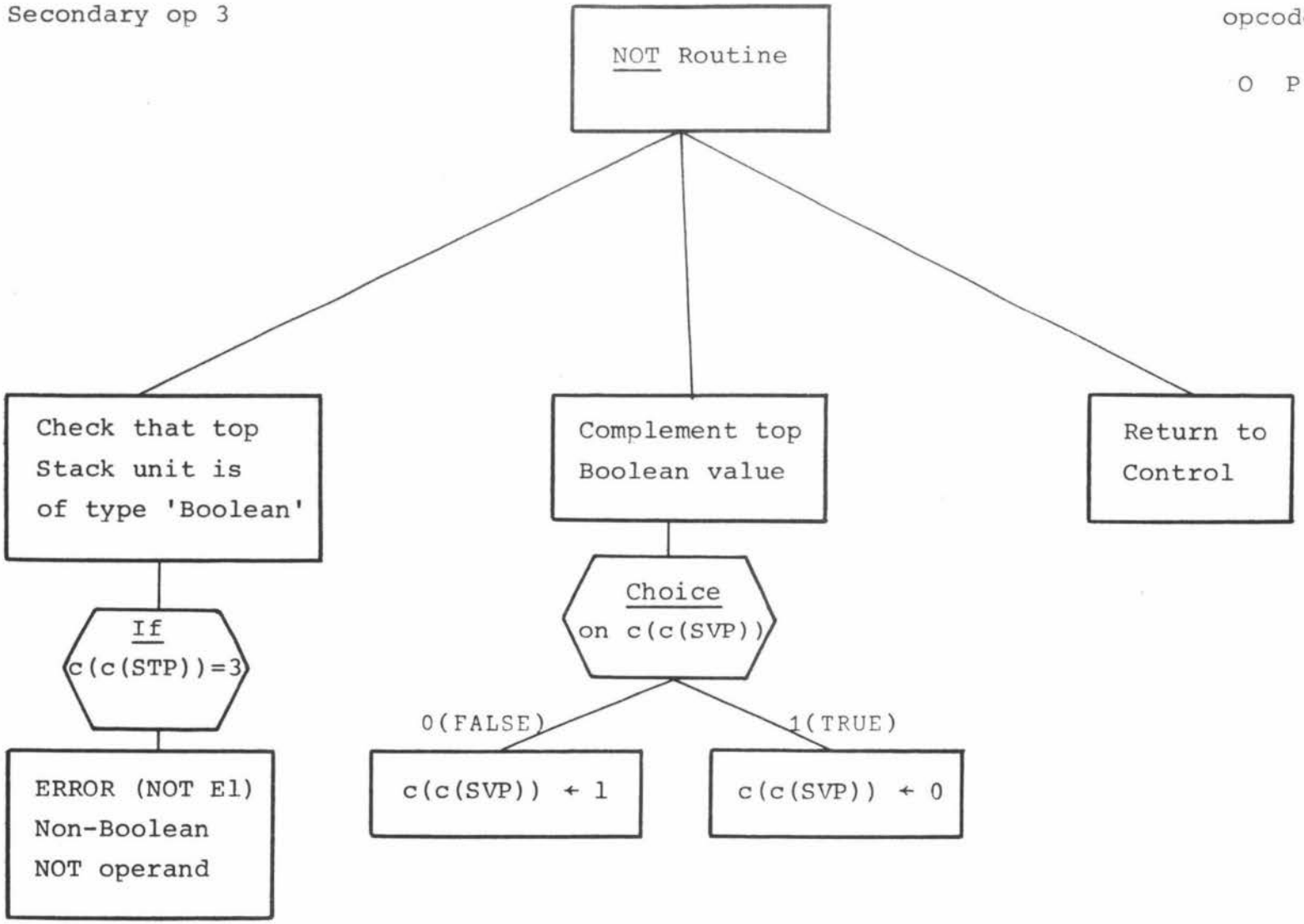
Secondary op 3

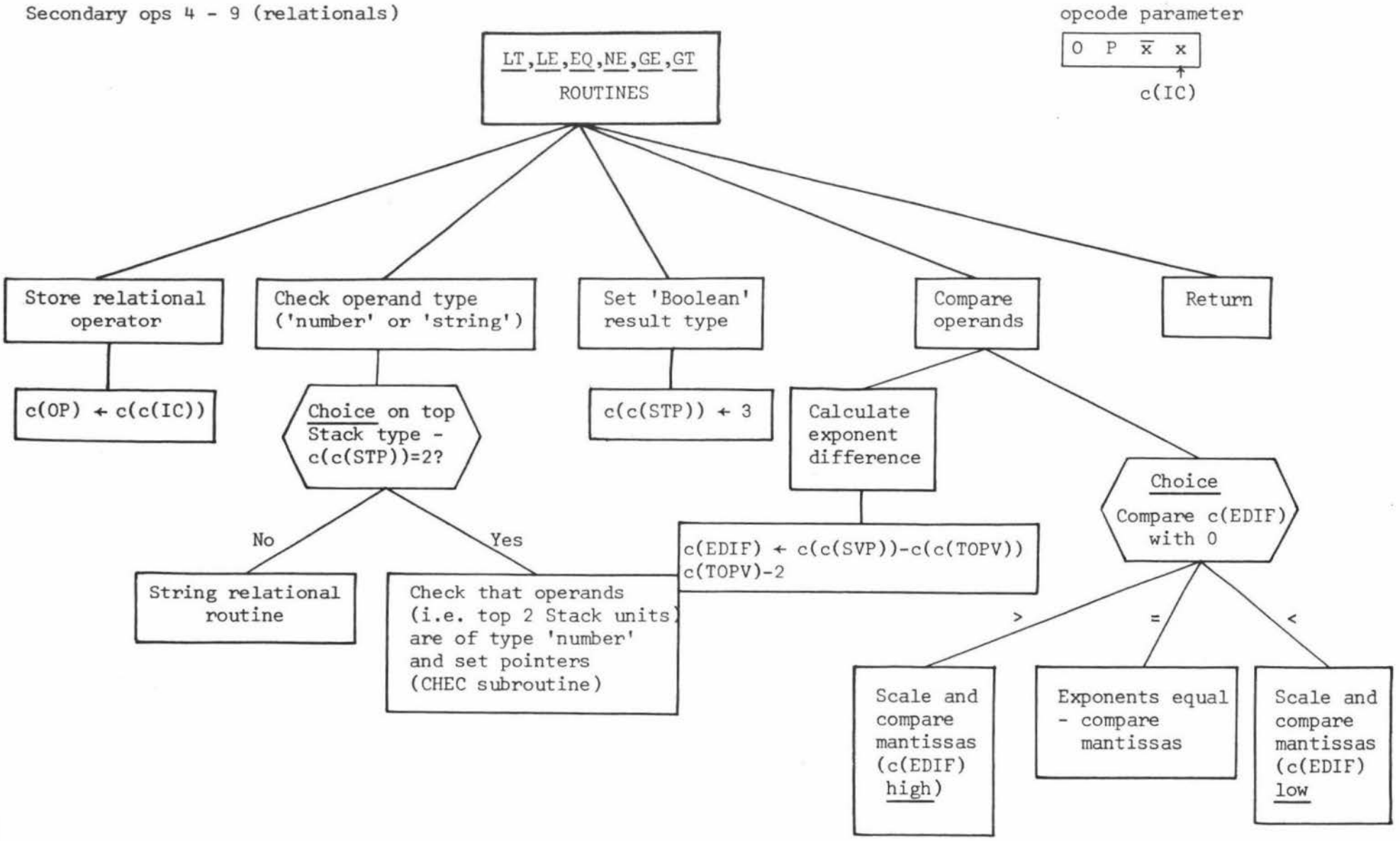
opcode parameter

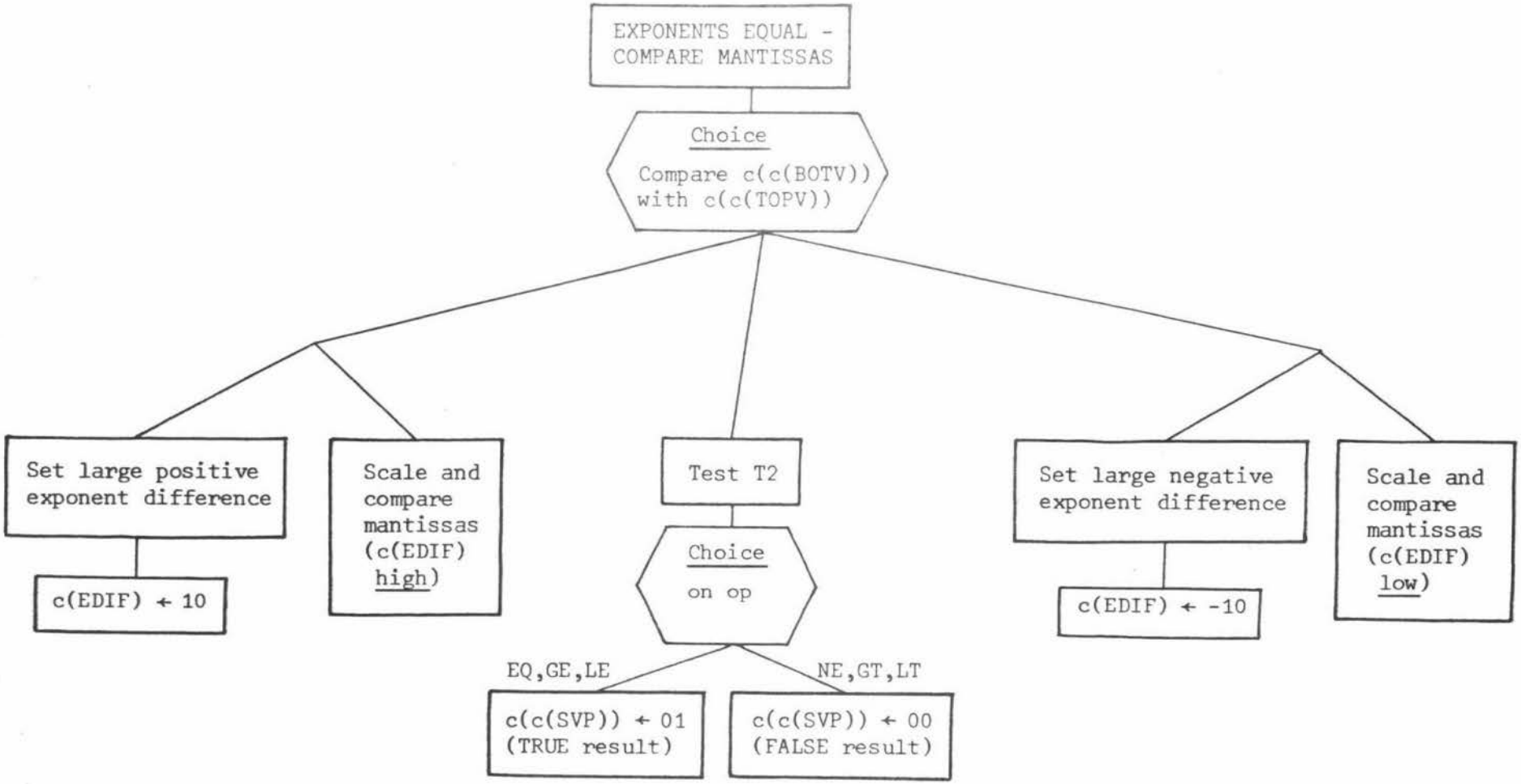
O P $\bar{x}$ x

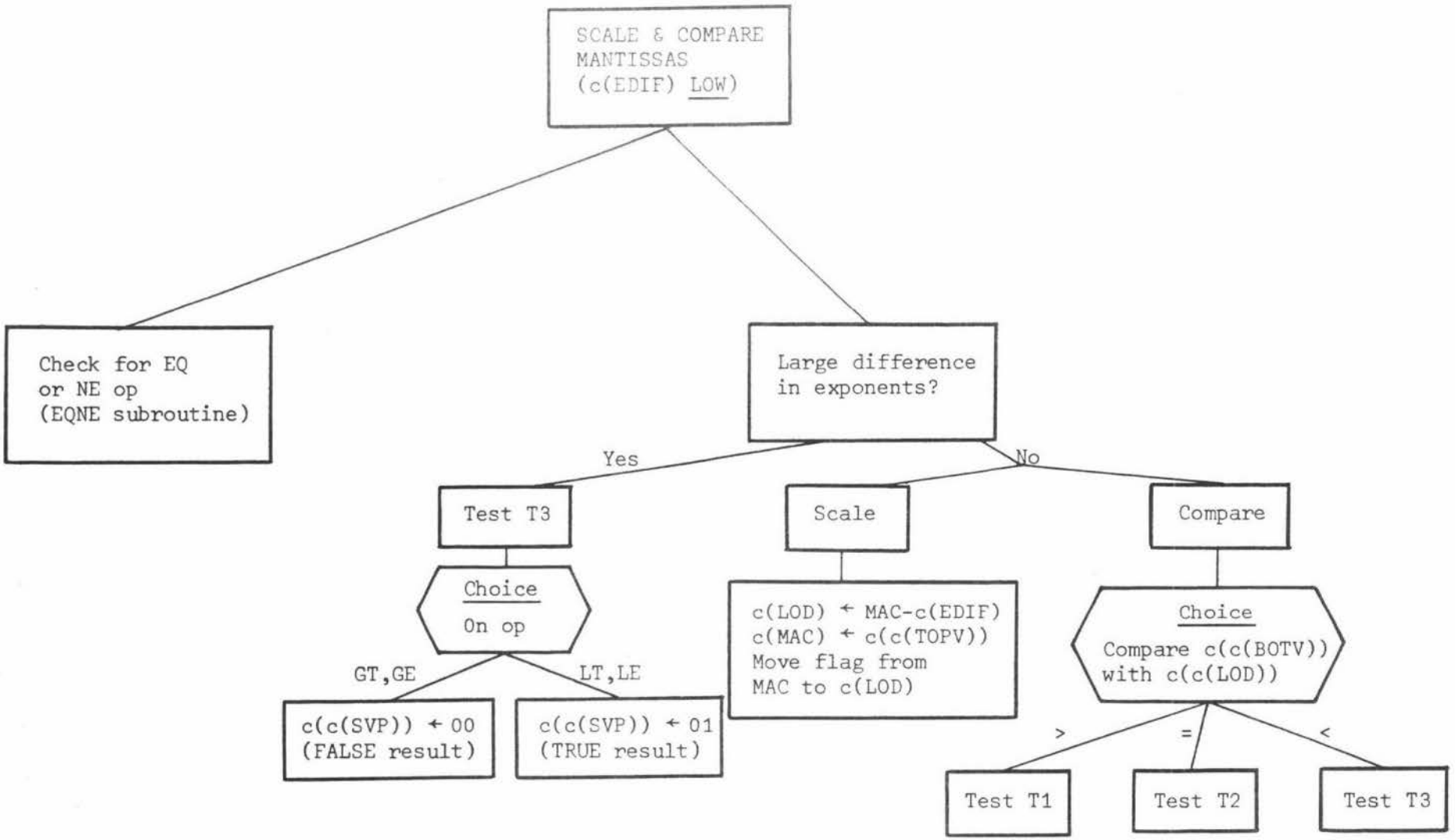
↑

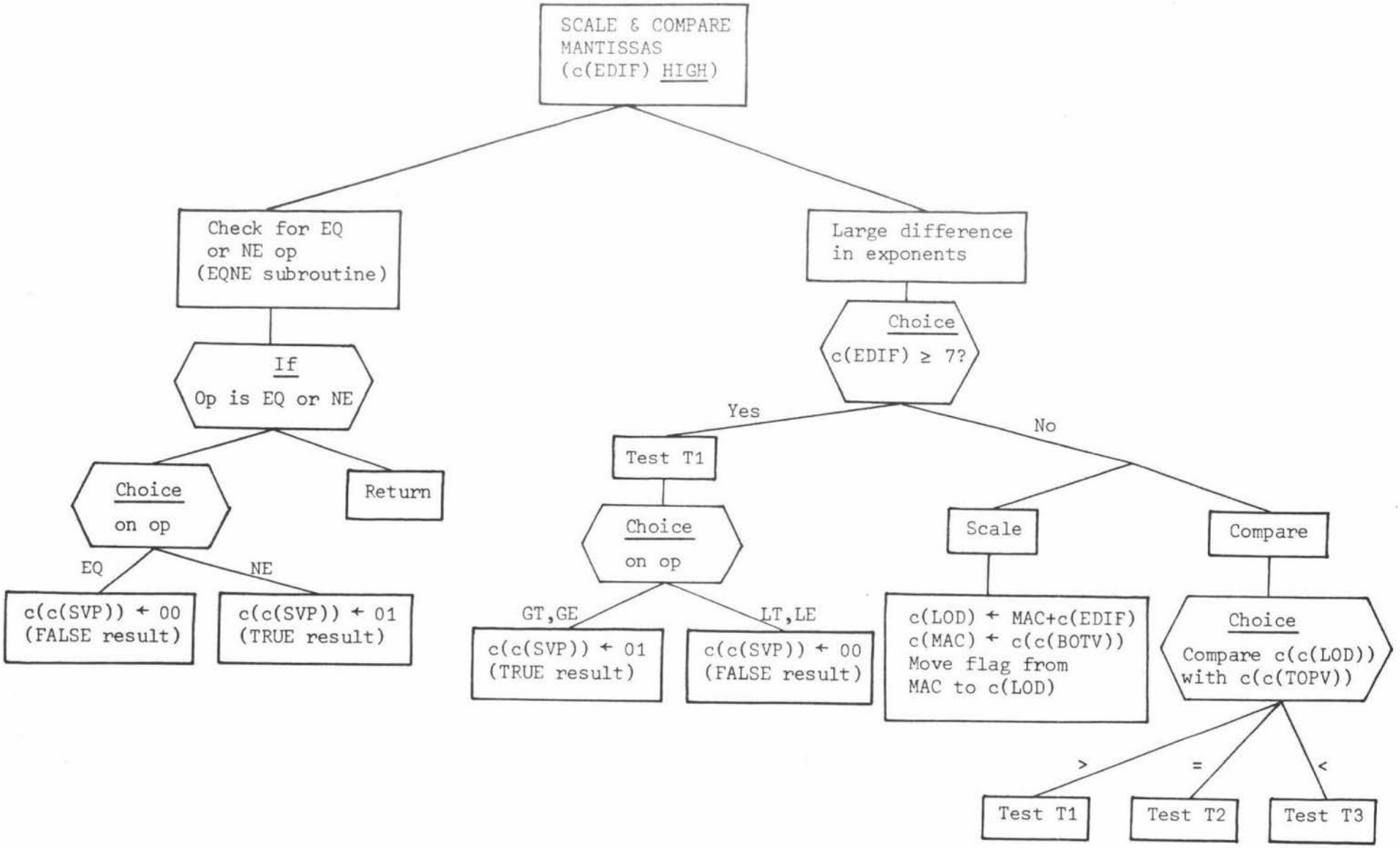
c(IC)

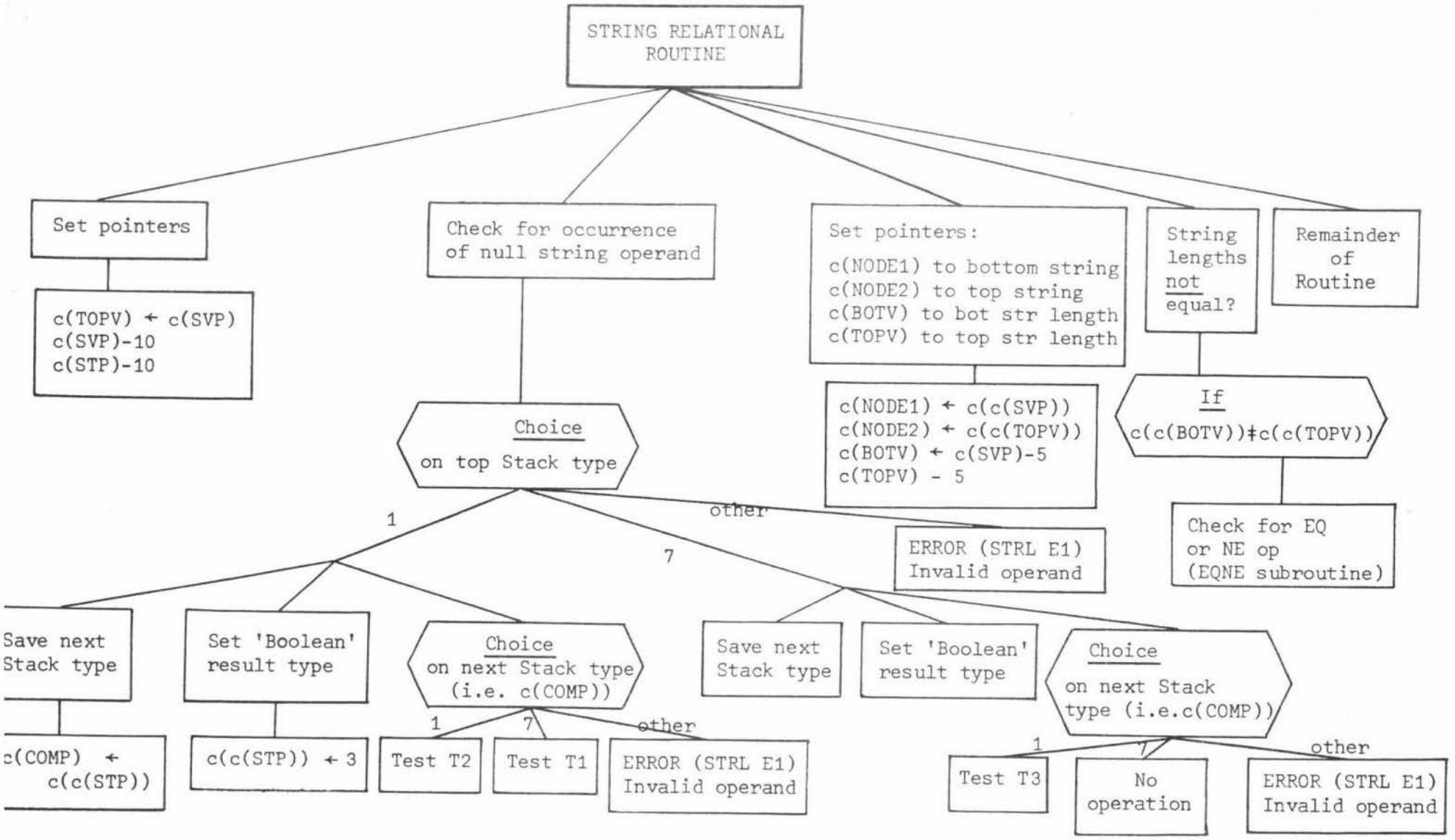


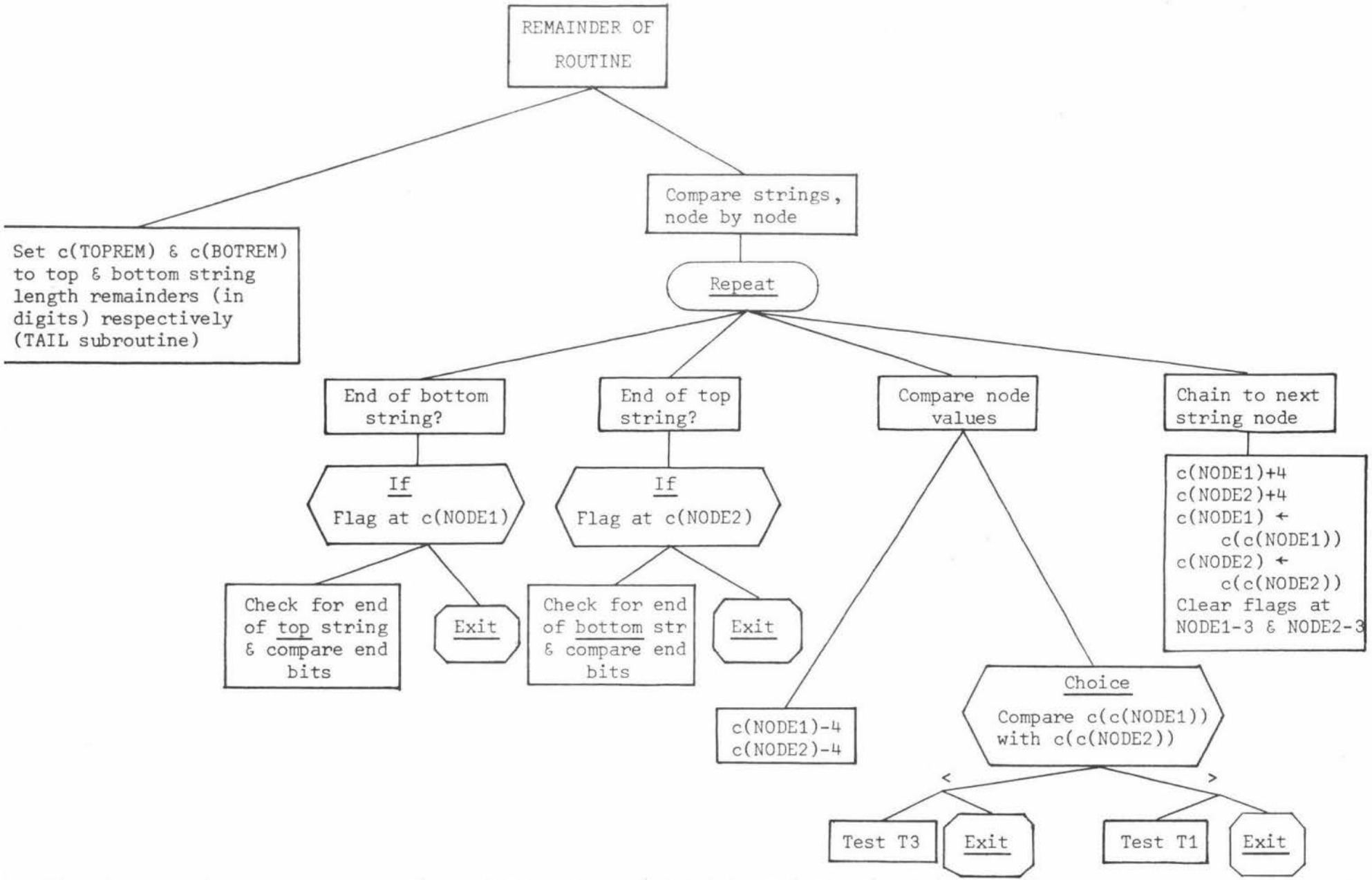

Secondary ops 4 - 9 (relationals)
opcode parameter
O P x̄ x
c(IC)
LT,LE,EQ,NE,GE,GT
ROUTINES
Store relational operator
Check operand type ('number' or 'string')
Set 'Boolean' result type
Compare operands
Return
c(OP) ← c(c(IC))
Choice on top Stack type - c(c(STP))=2?
c(c(STP)) ← 3
Calculate exponent difference
Choice
Compare c(EDIF) with 0
No
Yes
String relational routine
Check that operands (i.e. top 2 Stack units) are of type 'number' and set pointers (CHEC subroutine)
c(EDIF) ← c(c(SVP))-c(c(TOPV))
c(TOPV)-2
>
=
<
Scale and compare mantissas (c(EDIF) high)
Exponents equal - compare mantissas
Scale and compare mantissas (c(EDIF) low

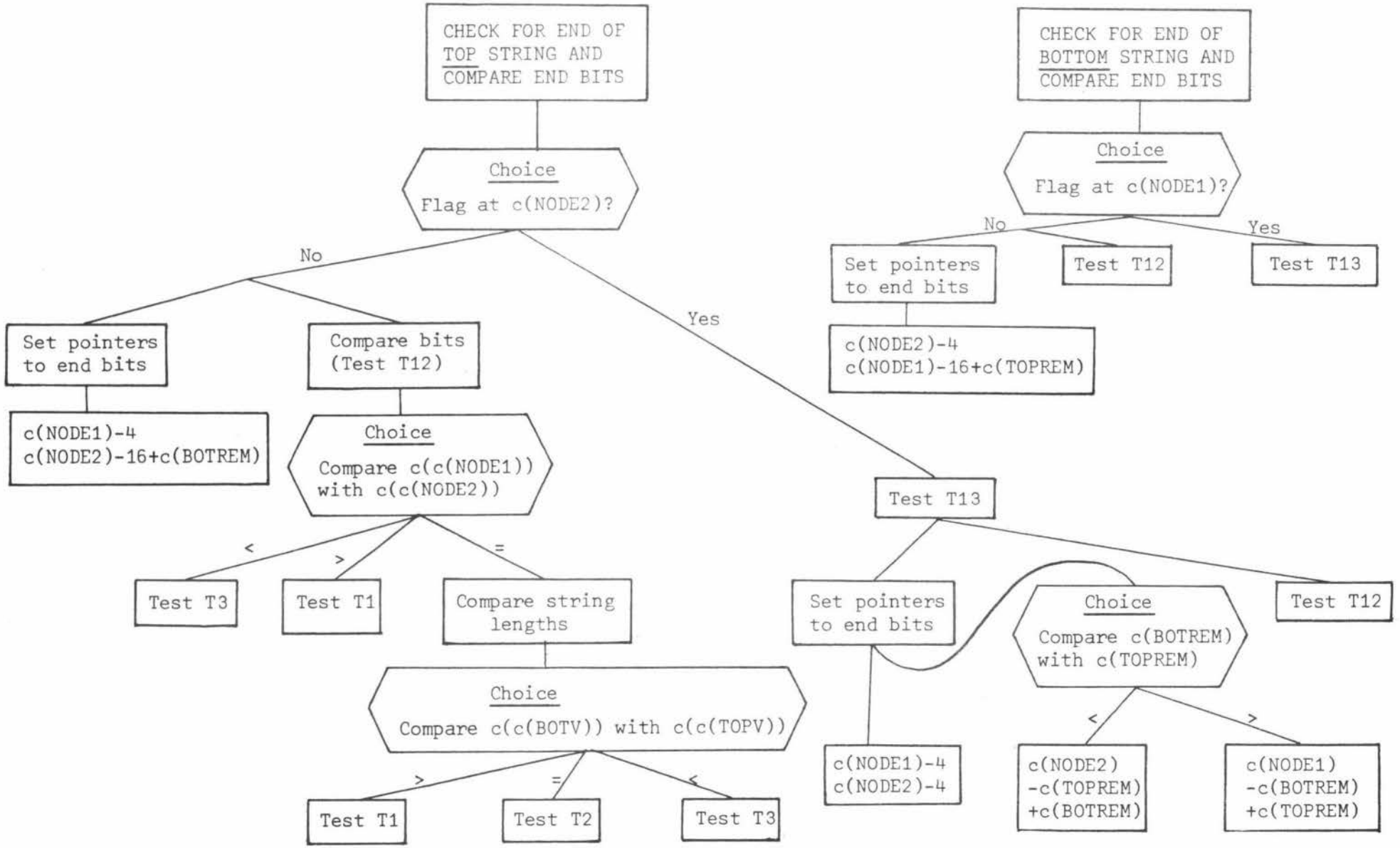

EXPONENTS EQUAL - COMPARE MANTISSAS
Choice
Compare c(c(BOTV)) with c(c(TOPV))
Set large positive exponent difference
c(EDIF) ← 10
Scale and compare mantissas (c(EDIF) high)
Test T2
Choice
on op
EQ,GE,LE
NE,GT,LT
c(c(SVP)) ← 01 (TRUE result)
c(c(SVP)) ← 00 (FALSE result)
Set large negative exponent difference
c(EDIF) ← -10
Scale and compare mantissas (c(EDIF) low)


SCALE & COMPARE
MANTISSAS
(c(EDIF) LOW)
Check for EQ
or NE op
(EQNE subroutine)
Large difference
in exponents?
Yes
No
Test T3
Scale
Compare
Choice
On op
c(LOD) ← MAC-c(EDIF)
c(MAC) ← c(c(TOPV))
Move flag from
MAC to c(LOD)
Choice
Compare c(c(BOTV))
with c(c(LOD))
GT,GE
LT,LE
c(c(SVP)) ← 00
(FALSE result)
c(c(SVP)) ← 01
(TRUE result)
>
=
<
Test T1
Test T2
Test T3


SCALE & COMPARE MANTISSAS (c(EDIF) HIGH)
Check for EQ or NE op (EQNE subroutine)
If Op is EQ or NE
Choice on op
EQ
NE
c(c(SVP)) ← 00 (FALSE result)
c(c(SVP)) ← 01 (TRUE result)
Return
Large difference in exponents
Choice c(EDIF) ≥ 7?
Yes
No
Test T1
Choice on op
GT,GE
LT,LE
c(c(SVP)) ← 01 (TRUE result)
c(c(SVP)) ← 00 (FALSE result)
Scale
c(LOD) ← MAC+c(EDIF)
c(MAC) ← c(c(BOTV))
Move flag from MAC to c(LOD)
Compare
Choice Compare c(c(LOD)) with c(c(TOPV))
>
=
<
Test T1
Test T2
Test T3


STRING RELATIONAL ROUTINE
Set pointers
c(TOPV) ← c(SVP)
c(SVP)-10
c(STP)-10
Check for occurrence of null string operand
Choice
on top Stack type
1
7
other
ERROR (STRL E1)
Invalid operand
Save next Stack type
c(COMP) ← c(c(STP))
Set 'Boolean' result type
c(c(STP)) ← 3
Choice
on next Stack type (i.e. c(COMP))
1
7
other
Test T2
Test T1
ERROR (STRL E1)
Invalid operand
Set pointers:
c(NODE1) to bottom string
c(NODE2) to top string
c(BOTV) to bot str length
c(TOPV) to top str length
c(NODE1) ← c(c(SVP))
c(NODE2) ← c(c(TOPV))
c(BOTV) ← c(SVP)-5
c(TOPV) - 5
Save next Stack type
Set 'Boolean' result type
Choice
on next Stack type (i.e.c(COMP))
1
7
other
Test T3
No operation
ERROR (STRL E1)
Invalid operand
String lengths not equal?
If
c(c(BOTV))≠c(c(TOPV))
Check for EQ or NE op
(EQNE subroutine)
Remainder of Routine


REMAINDER OF ROUTINE
Set c(TOPREM) & c(BOTREM) to top & bottom string length remainders (in digits) respectively (TAIL subroutine)
Compare strings, node by node
Repeat
End of bottom string?
If Flag at c(NODE1)
Check for end of top string & compare end bits
Exit
End of top string?
If Flag at c(NODE2)
Check for end of bottom str & compare end bits
Exit
Compare node values
c(NODE1)-4
c(NODE2)-4
Choice
Compare c(c(NODE1)) with c(c(NODE2))
<
Test T3
Exit
>
Test T1
Exit
Chain to next string node
c(NODE1)+4
c(NODE2)+4
c(NODE1) ← c(c(NODE1))
c(NODE2) ← c(c(NODE2))
Clear flags at NODE1-3 & NODE2-3


CHECK FOR END OF TOP STRING AND COMPARE END BITS
Choice
Flag at c(NODE2)?
No
Yes
Set pointers to end bits
c(NODE1)-4
c(NODE2)-16+c(BOTREM)
Compare bits (Test T12)
Choice
Compare c(c(NODE1)) with c(c(NODE2))
<
>
=
Test T3
Test T1
Compare string lengths
Choice
Compare c(c(BOTV)) with c(c(TOPV))
>
=
<
Test T1
Test T2
Test T3
CHECK FOR END OF BOTTOM STRING AND COMPARE END BITS
Choice
Flag at c(NODE1)?
No
Yes
Set pointers to end bits
c(NODE2)-4
c(NODE1)-16+c(TOPREM)
Test T12
Test T13
Test T13
Set pointers to end bits
c(NODE1)-4
c(NODE2)-4
Choice
Compare c(BOTREM) with c(TOPREM)
<
>
c(NODE2) -c(TOPREM) +c(BOTREM)
c(NODE1) -c(BOTREM) +c(TOPREM)
Test T12

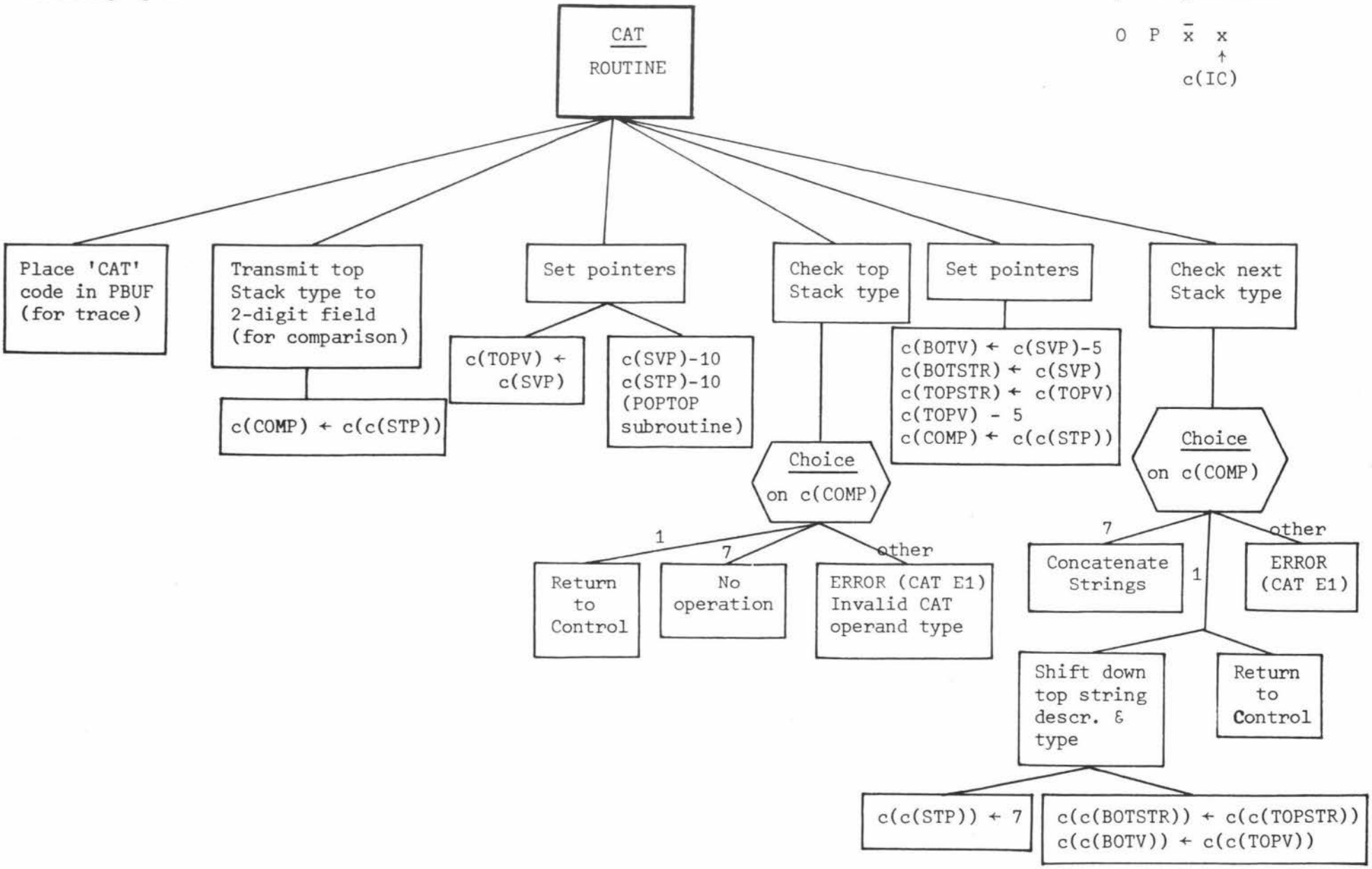
Secondary op 10

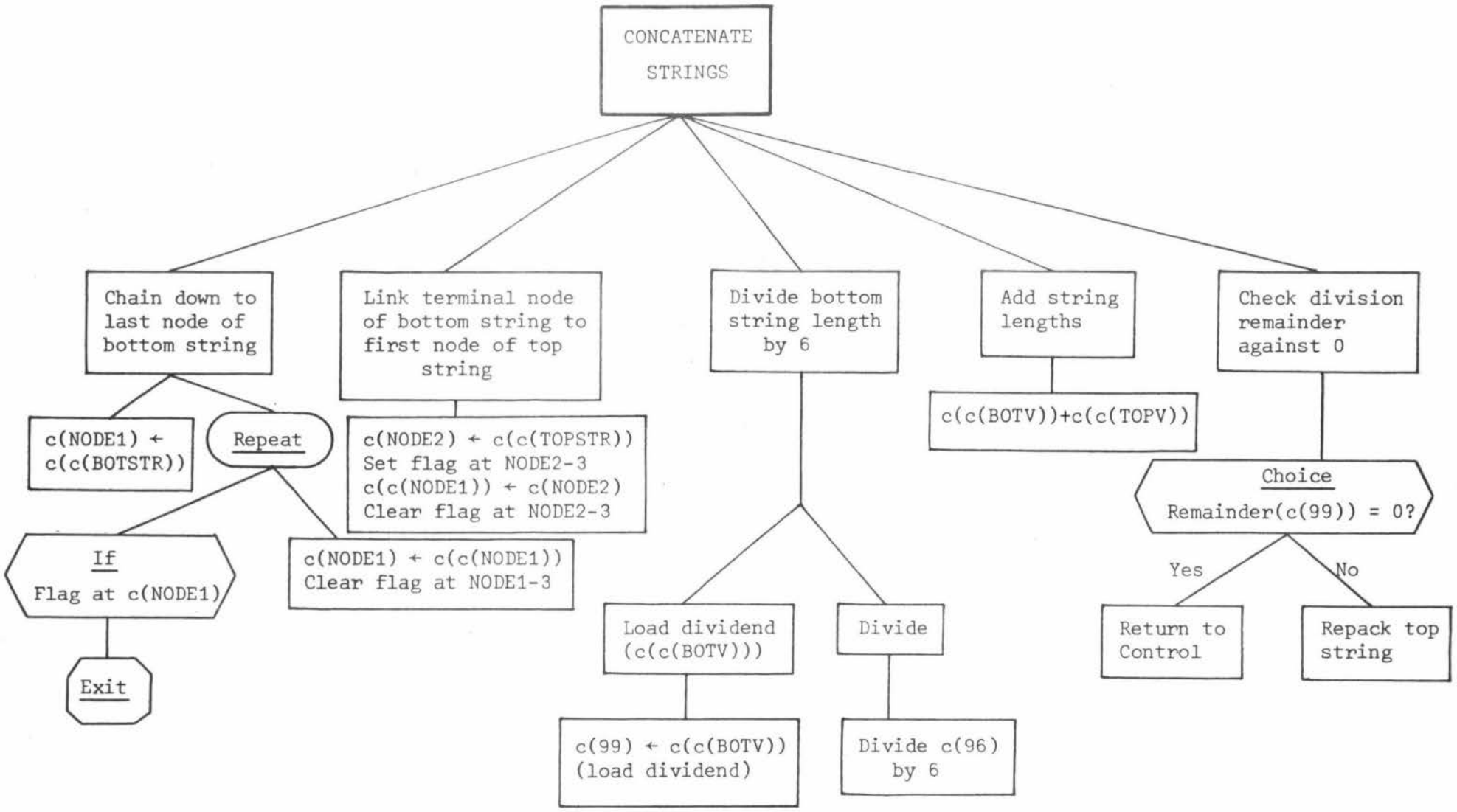
opcode parameter

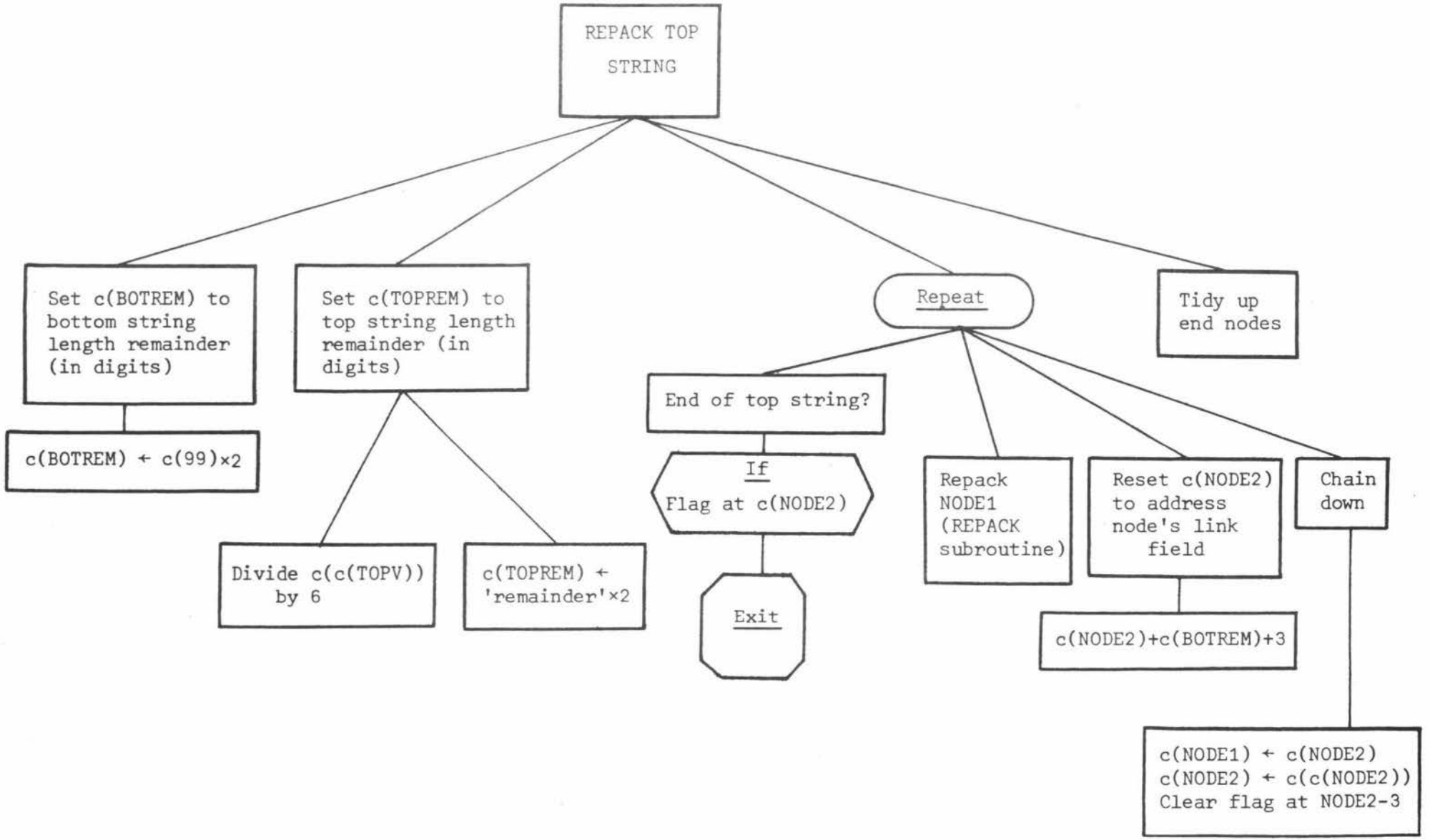
O P $\bar{x}$ x

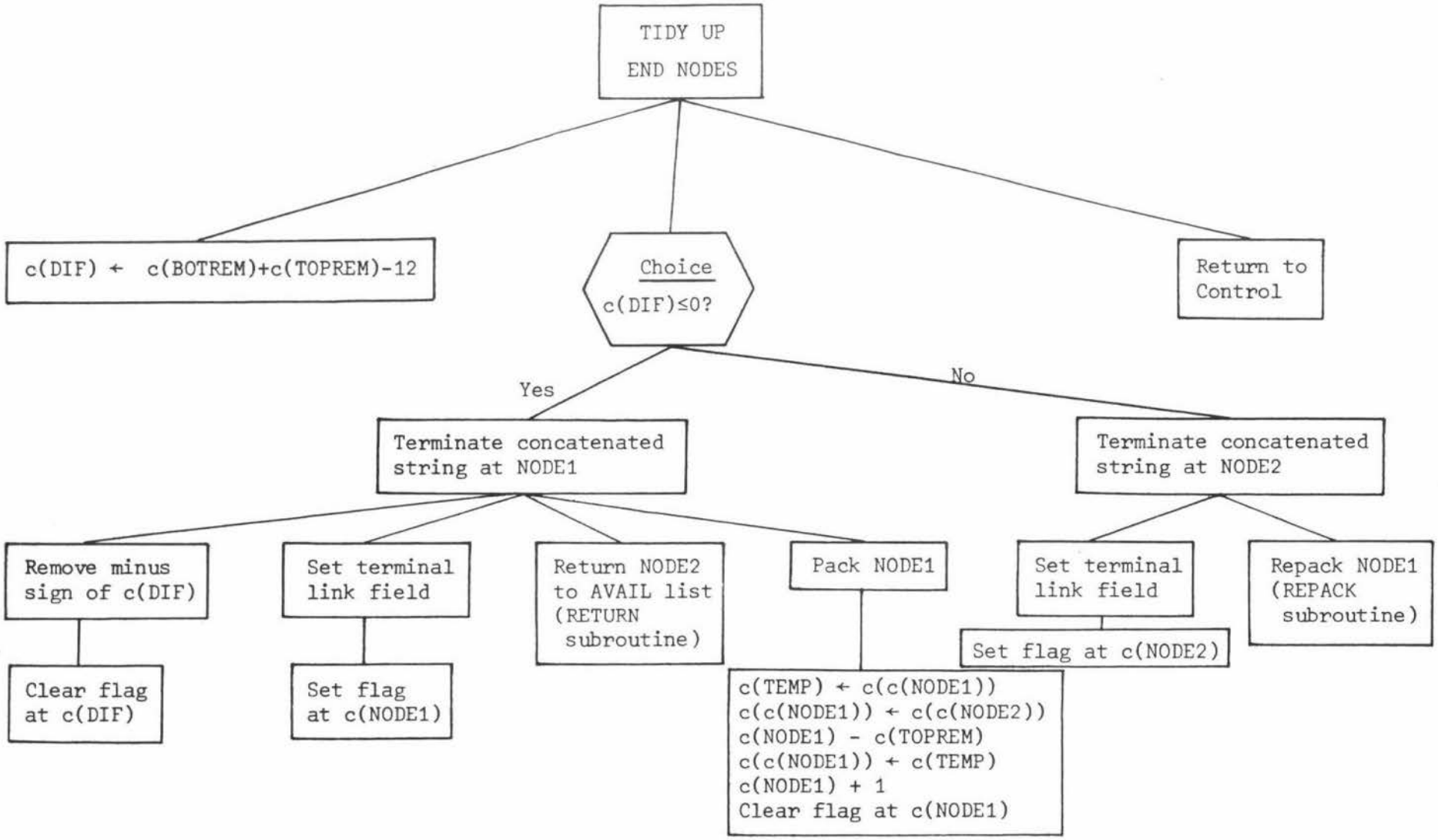
↑

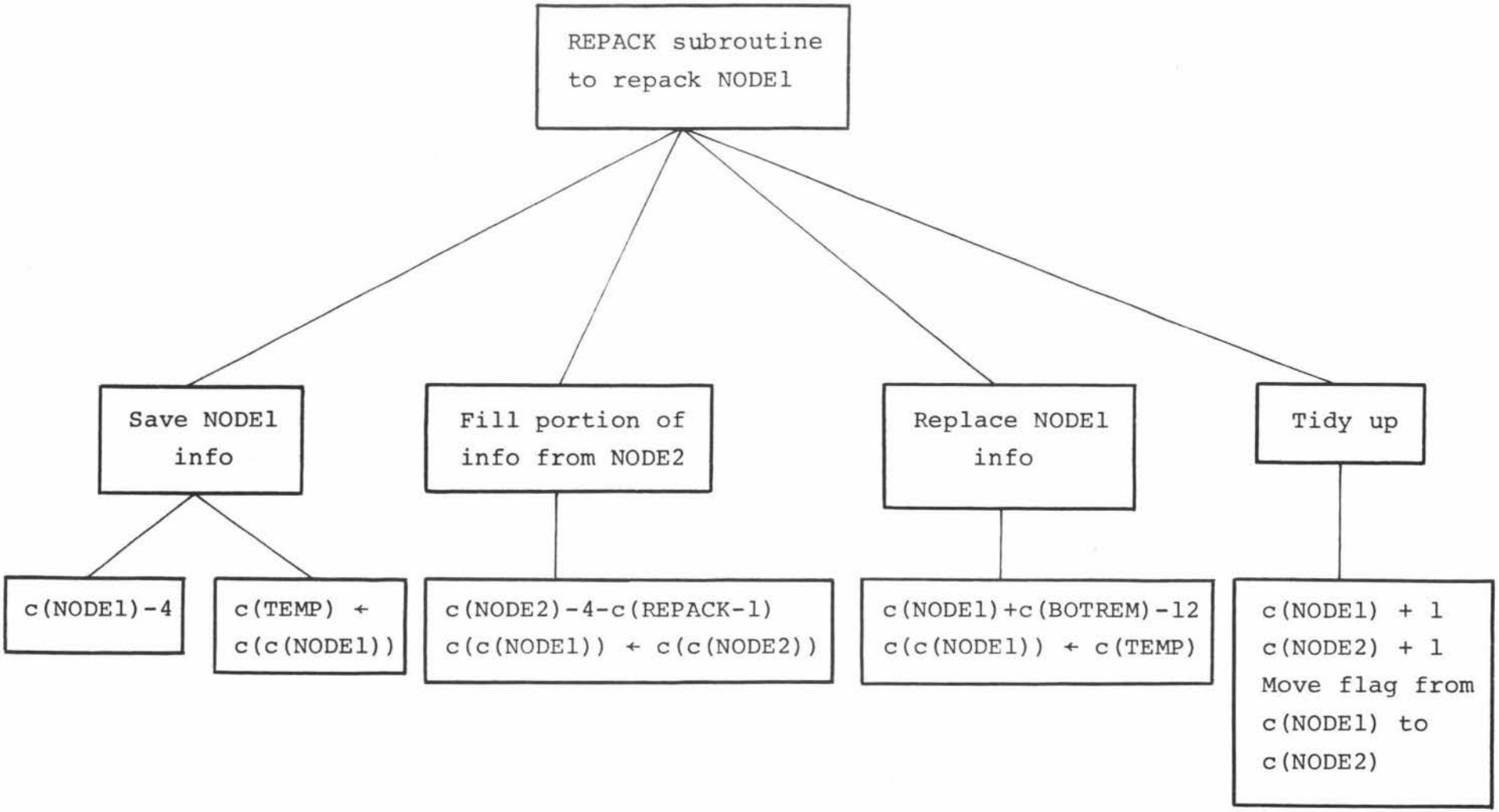
c(IC)

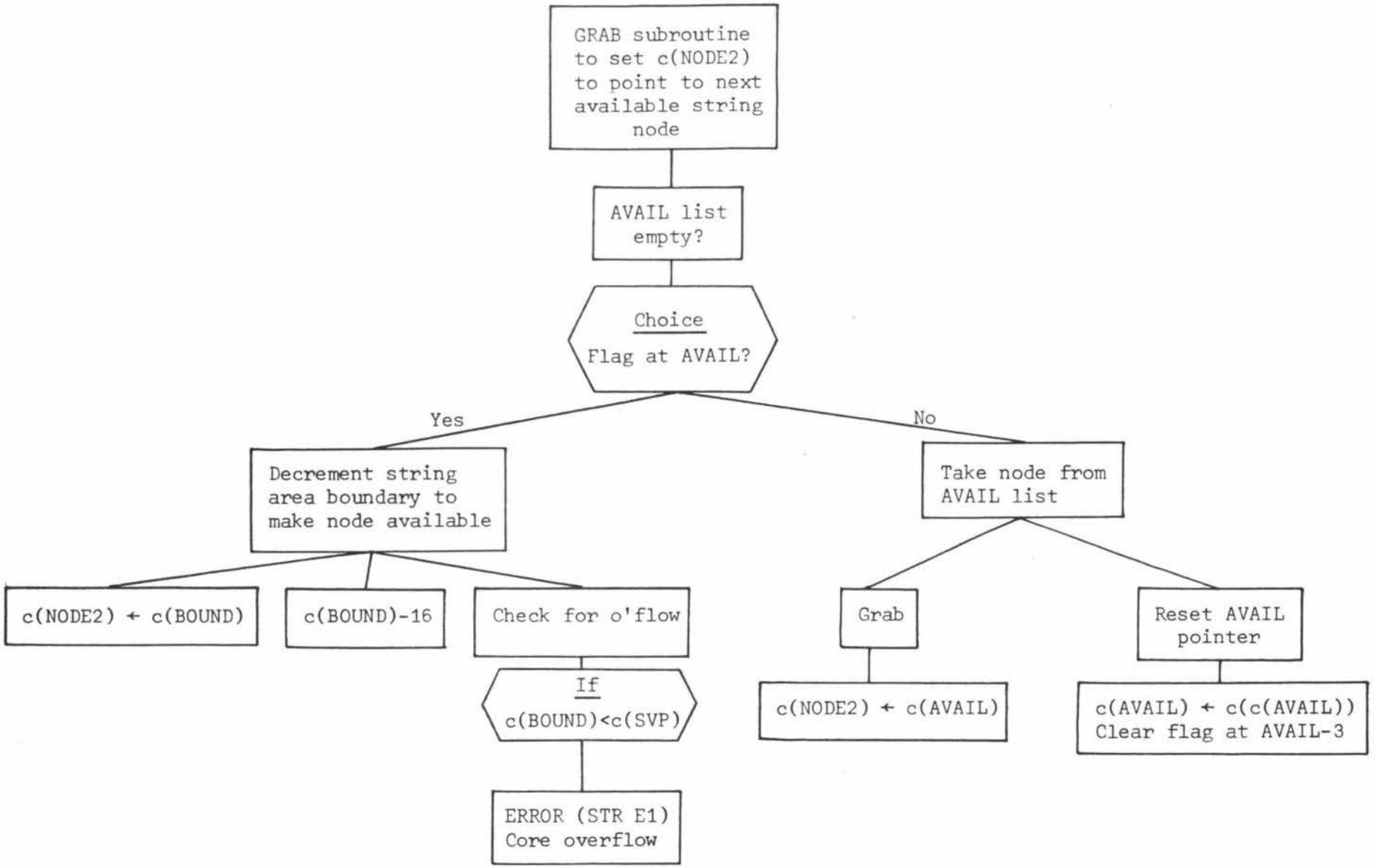


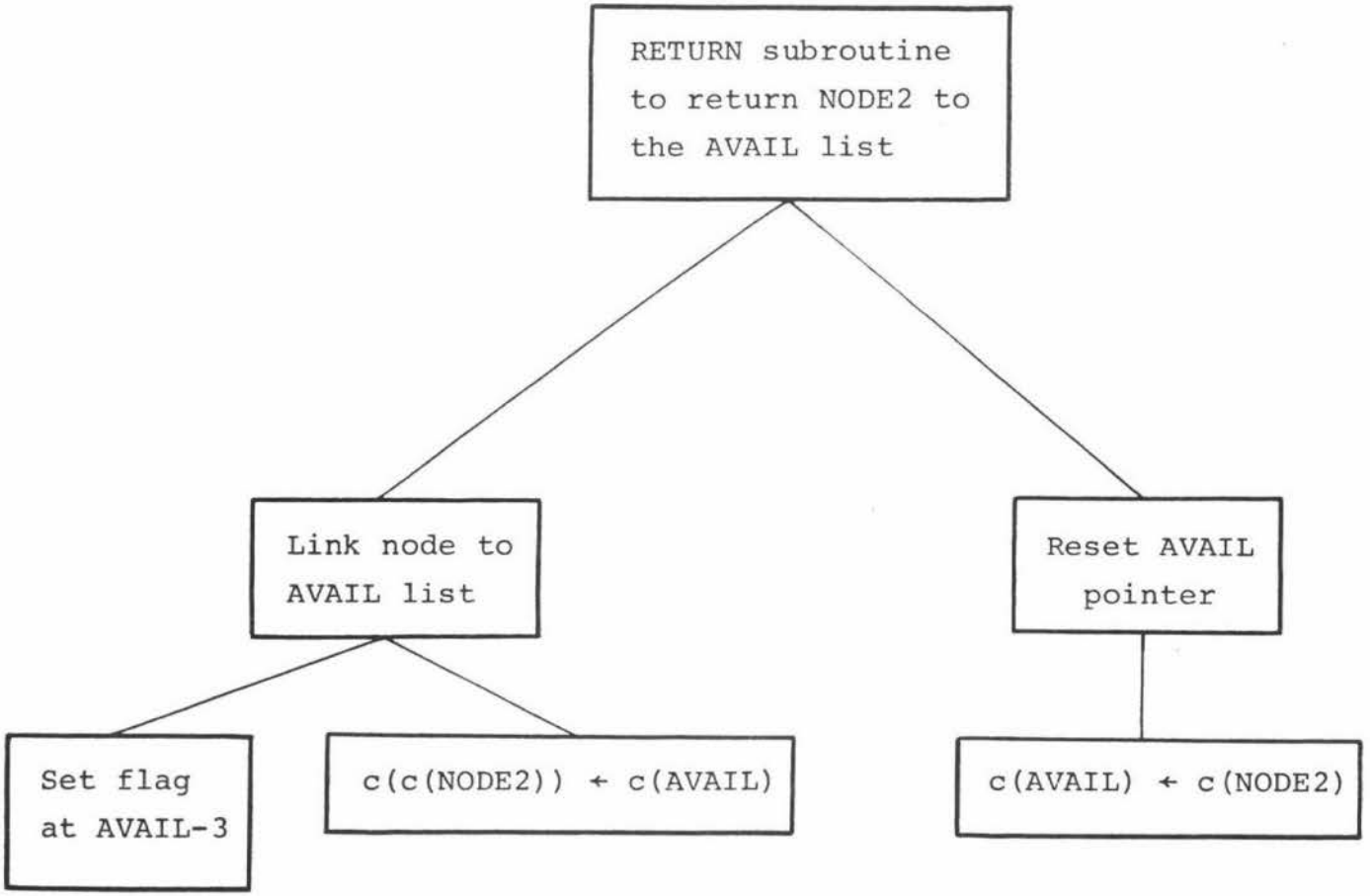

CONCATENATE STRINGS
Chain down to last node of bottom string
Link terminal node of bottom string to first node of top string
Divide bottom string length by 6
Add string lengths
Check division remainder against 0
c(NODE1) ← c(c(BOTSTR))
Repeat
c(NODE2) ← c(c(TOPSTR))
Set flag at NODE2-3
c(c(NODE1)) ← c(NODE2)
Clear flag at NODE2-3
c(c(BOTV))+c(c(TOPV))
Choice
Remainder(c(99)) = 0?
If
Flag at c(NODE1)
c(NODE1) ← c(c(NODE1))
Clear flag at NODE1-3
Load dividend (c(c(BOTV)))
Divide
Yes
No
Exit
c(99) ← c(c(BOTV))
(load dividend)
Divide c(96) by 6
Return to Control
Repack top string


REPACK TOP STRING
Set c(BOTREM) to bottom string length remainder (in digits)
c(BOTREM) ← c(99)×2
Set c(TOPREM) to top string length remainder (in digits)
Divide c(c(TOPV)) by 6
c(TOPREM) ← 'remainder'×2
Repeat
End of top string?
If Flag at c(NODE2)
Exit
Repack NODE1 (REPACK subroutine)
Reset c(NODE2) to address node's link field
c(NODE2)+c(BOTREM)+3
Chain down
c(NODE1) ← c(NODE2)
c(NODE2) ← c(c(NODE2))
Clear flag at NODE2-3
Tidy up end nodes

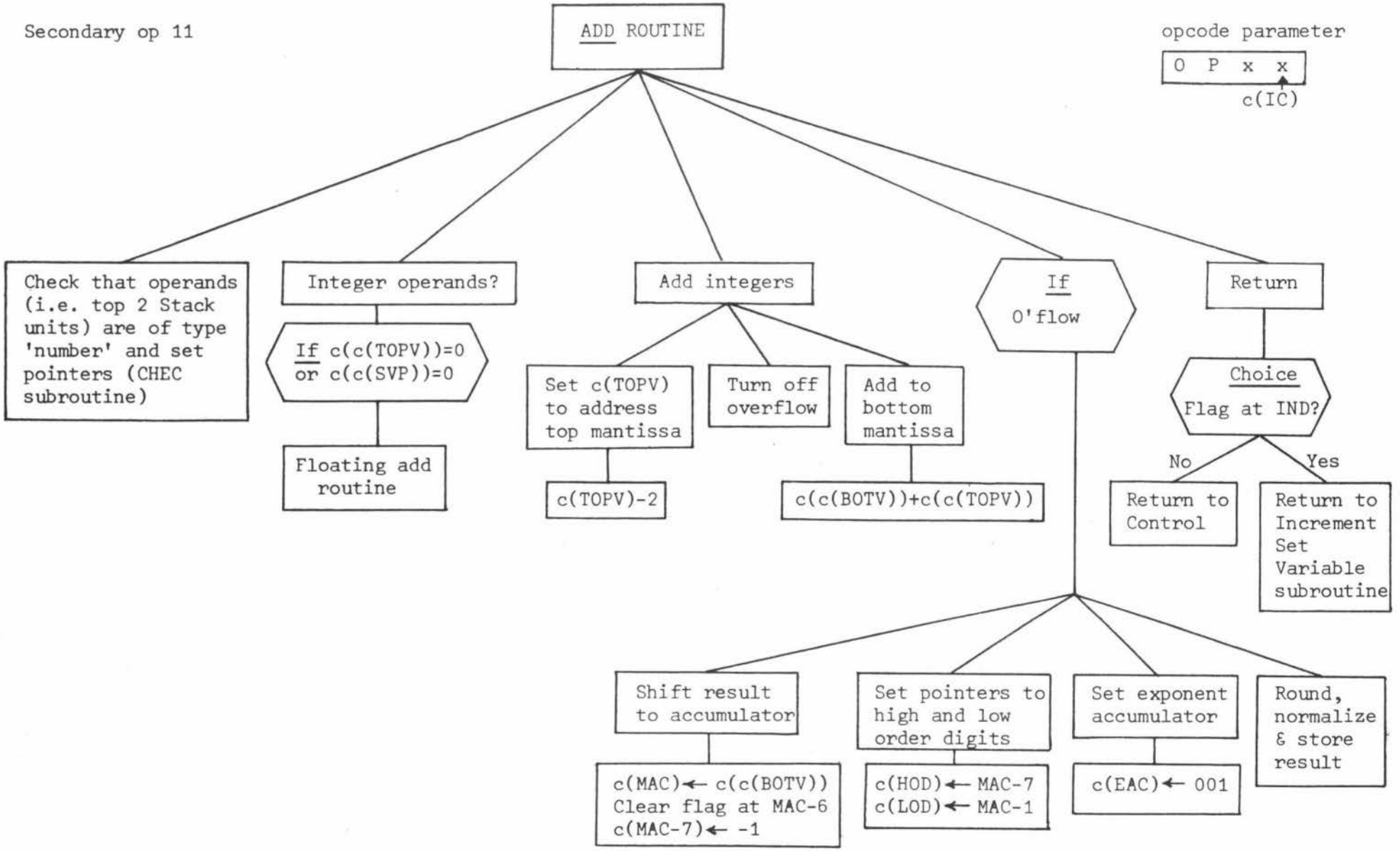

TIDY UP
END NODES
c(DIF) ← c(BOTREM)+c(TOPREM)-12
Choice
c(DIF)≤0?
Return to Control
Yes
No
Terminate concatenated string at NODE1
Terminate concatenated string at NODE2
Remove minus sign of c(DIF)
Set terminal link field
Return NODE2 to AVAIL list (RETURN subroutine)
Pack NODE1
Set terminal link field
Repack NODE1 (REPACK subroutine)
Clear flag at c(DIF)
Set flag at c(NODE1)
Set flag at c(NODE2)
c(TEMP) ← c(c(NODE1))
c(c(NODE1)) ← c(c(NODE2))
c(NODE1) - c(TOPREM)
c(c(NODE1)) ← c(TEMP)
c(NODE1) + 1
Clear flag at c(NODE1)

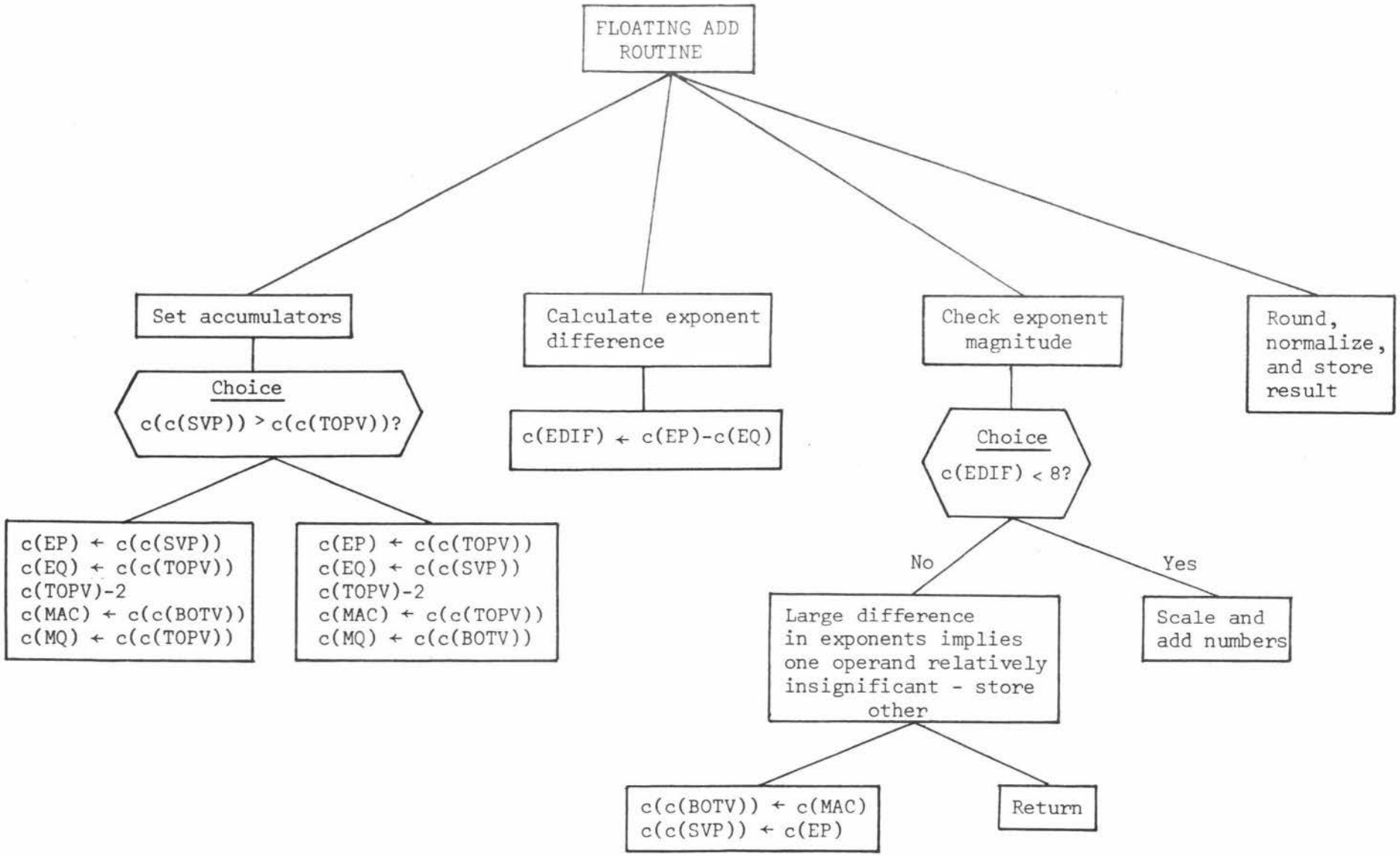

REPACK subroutine to repack NODE1
Save NODE1 info
Fill portion of info from NODE2
Replace NODE1 info
Tidy up
c(NODE1)-4
c(TEMP) ← c(c(NODE1))
c(NODE2)-4-c(REPACK-1)
c(c(NODE1)) ← c(c(NODE2))
c(NODE1)+c(BOTREM)-12
c(c(NODE1)) ← c(TEMP)
c(NODE1) + 1
c(NODE2) + 1
Move flag from c(NODE1) to c(NODE2)

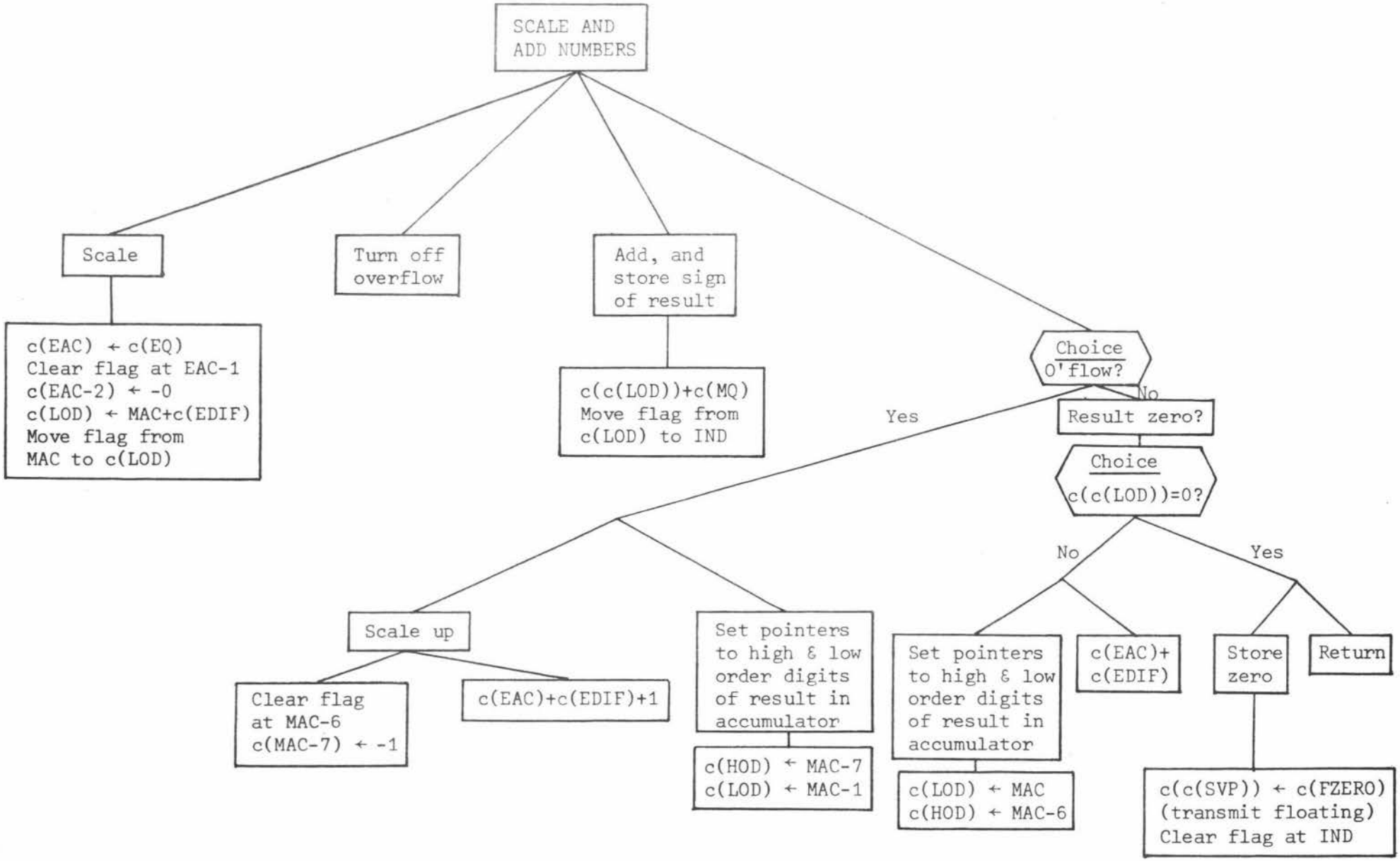

GRAB subroutine to set c(NODE2) to point to next available string node
AVAIL list empty?
Choice
Flag at AVAIL?
Yes
No
Decrement string area boundary to make node available
c(NODE2) ← c(BOUND)
c(BOUND)-16
Check for o'flow
If
c(BOUND)<c(SVP)
ERROR (STR E1)
Core overflow
Take node from AVAIL list
Grab
c(NODE2) ← c(AVAIL)
Reset AVAIL pointer
c(AVAIL) ← c(c(AVAIL))
Clear flag at AVAIL-3

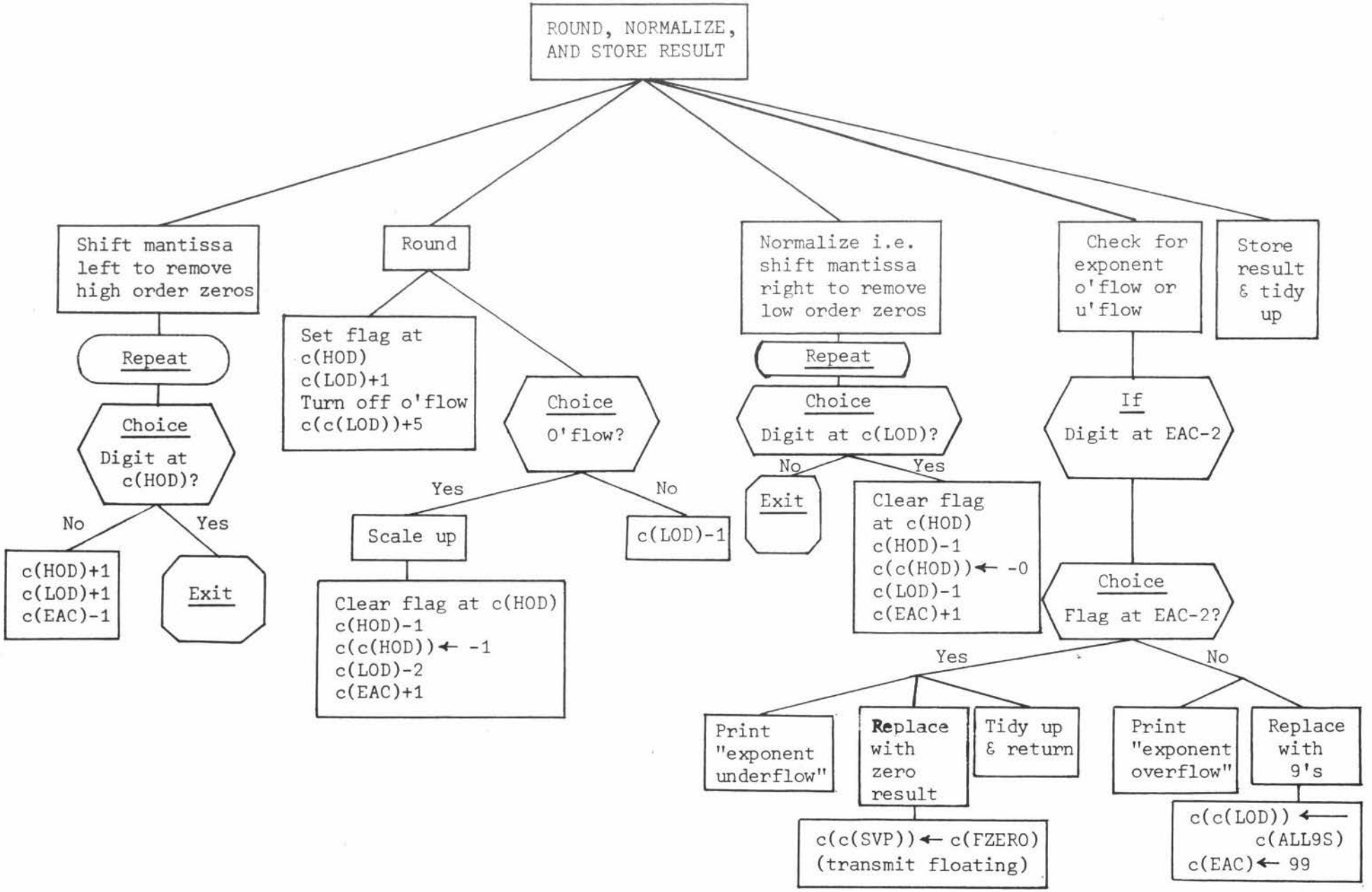

RETURN subroutine to return NODE2 to the AVAIL list
Link node to AVAIL list
Reset AVAIL pointer
Set flag at AVAIL-3
c(c(NODE2)) ← c(AVAIL)
c(AVAIL) ← c(NODE2)

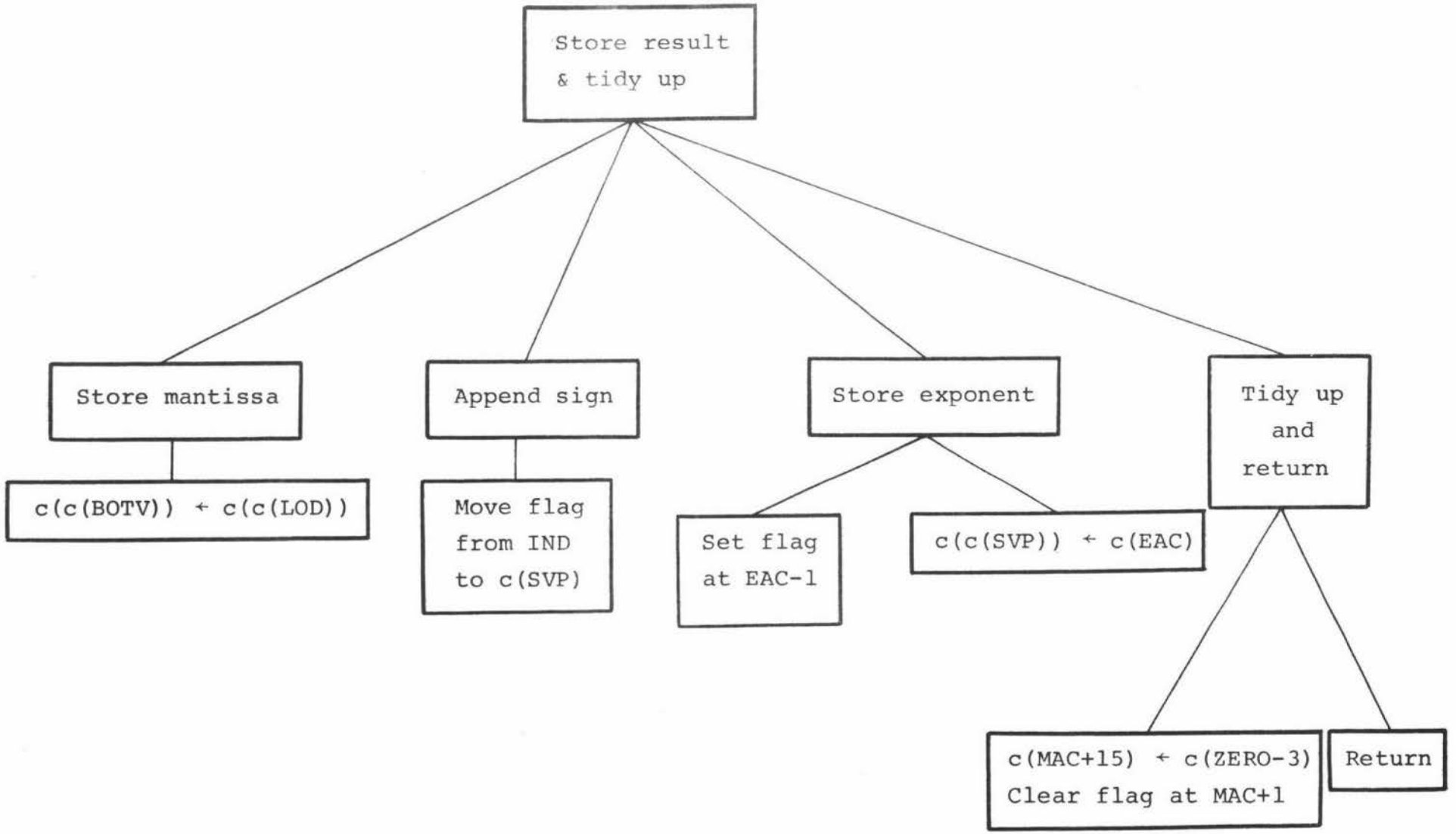
Secondary op 11




FLOATING ADD ROUTINE
Set accumulators
Choice
c(c(SVP)) > c(c(TOPV))?
c(EP) ← c(c(SVP))
c(EQ) ← c(c(TOPV))
c(TOPV)-2
c(MAC) ← c(c(BOTV))
c(MQ) ← c(c(TOPV))
c(EP) ← c(c(TOPV))
c(EQ) ← c(c(SVP))
c(TOPV)-2
c(MAC) ← c(c(TOPV))
c(MQ) ← c(c(BOTV))
Calculate exponent difference
c(EDIF) ← c(EP)-c(EQ)
Check exponent magnitude
Choice
c(EDIF) < 8?
No
Yes
Large difference in exponents implies one operand relatively insignificant - store other
Scale and add numbers
c(c(BOTV)) ← c(MAC)
c(c(SVP)) ← c(EP)
Return
Round, normalize, and store result


SCALE AND ADD NUMBERS
Scale
c(EAC) ← c(EQ)
Clear flag at EAC-1
c(EAC-2) ← -0
c(LOD) ← MAC+c(EDIF)
Move flag from MAC to c(LOD)
Turn off overflow
Add, and store sign of result
c(c(LOD))+c(MQ)
Move flag from c(LOD) to IND
Choice
O'flow?
Yes
No
Result zero?
Choice
c(c(LOD))=0?
No
Yes
Scale up
Clear flag at MAC-6
c(MAC-7) ← -1
c(EAC)+c(EDIF)+1
Set pointers to high & low order digits of result in accumulator
c(HOD) ← MAC-7
c(LOD) ← MAC-1
Set pointers to high & low order digits of result in accumulator
c(LOD) ← MAC
c(HOD) ← MAC-6
c(EAC)+ c(EDIF)
Store zero
Return
c(c(SVP)) ← c(FZERO)
(transmit floating)
Clear flag at IND


ROUND, NORMALIZE, AND STORE RESULT
Shift mantissa left to remove high order zeros
Repeat
Choice
Digit at c(HOD)?
No
Yes
c(HOD)+1
c(LOD)+1
c(EAC)-1
Exit
Round
Set flag at c(HOD)
c(LOD)+1
Turn off o'flow
c(c(LOD))+5
Choice
O'flow?
Yes
No
Scale up
c(LOD)-1
Clear flag at c(HOD)
c(HOD)-1
c(c(HOD))← -1
c(LOD)-2
c(EAC)+1
Normalize i.e. shift mantissa right to remove low order zeros
Repeat
Choice
Digit at c(LOD)?
No
Yes
Exit
Clear flag at c(HOD)
c(HOD)-1
c(c(HOD))← -0
c(LOD)-1
c(EAC)+1
Check for exponent o'flow or u'flow
If
Digit at EAC-2
Choice
Flag at EAC-2?
Yes
No
Print "exponent underflow"
Replace with zero result
Tidy up & return
c(c(SVP))← c(FZERO)
(transmit floating)
Print "exponent overflow"
Replace with 9's
c(c(LOD))← c(ALL9S)
c(EAC)← 99
Store result & tidy up


Store result & tidy up
Store mantissa
Append sign
Store exponent
Tidy up and return
c(c(BOTV)) ← c(c(LOD))
Move flag from IND to c(SVP)
Set flag at EAC-1
c(c(SVP)) ← c(EAC)
c(MAC+15) ← c(ZERO-3)
Clear flag at MAC+1
Return

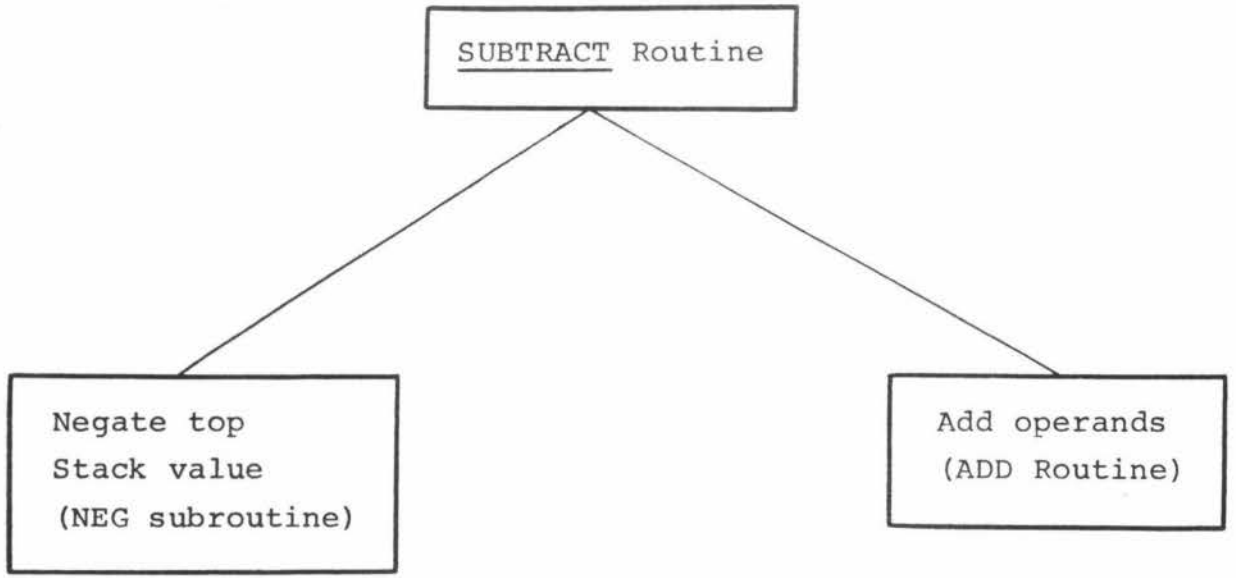
Secondary op 12

opcode parameter

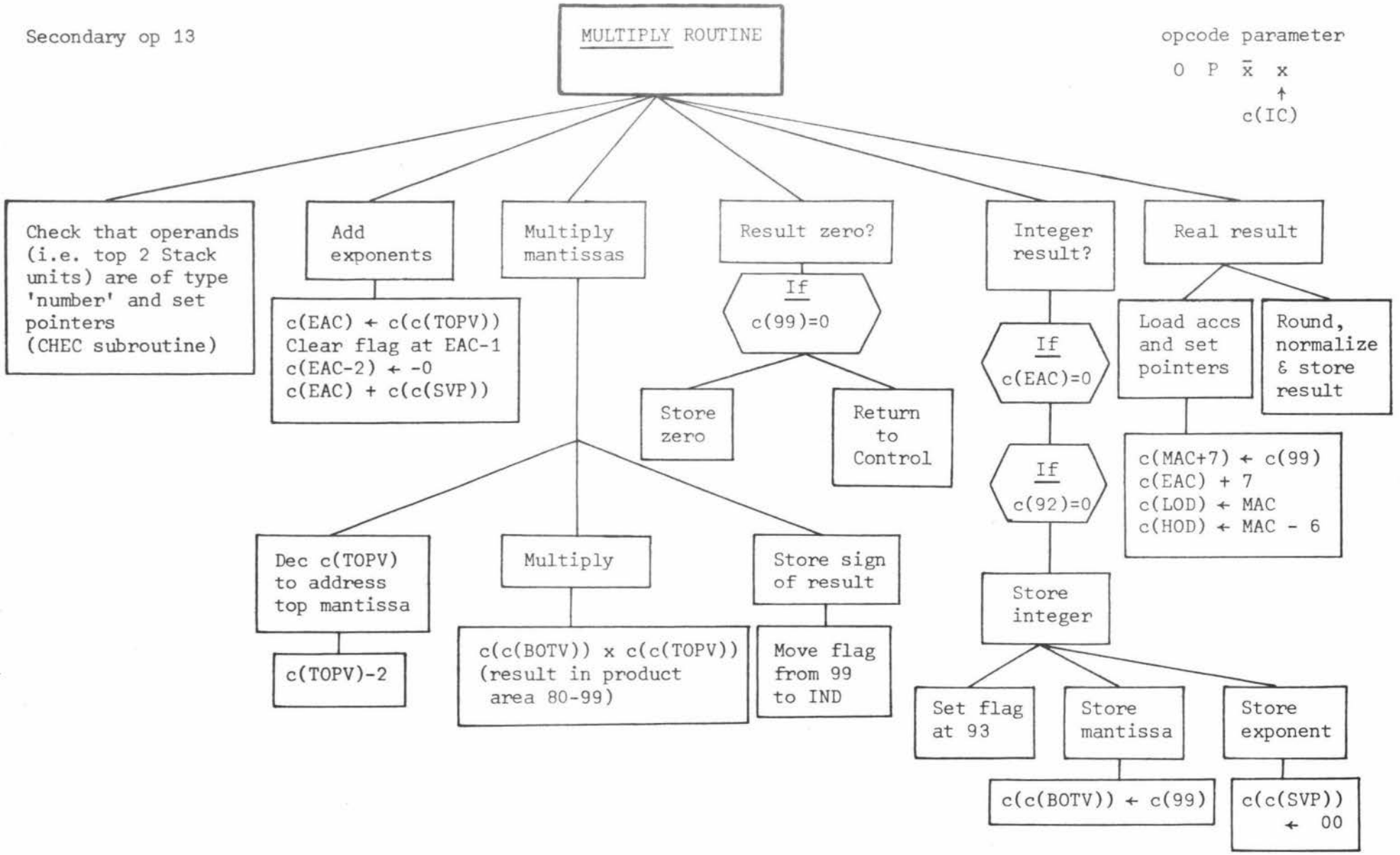
O P $\bar{x}$ x

↑

c(IC)



Secondary op 13



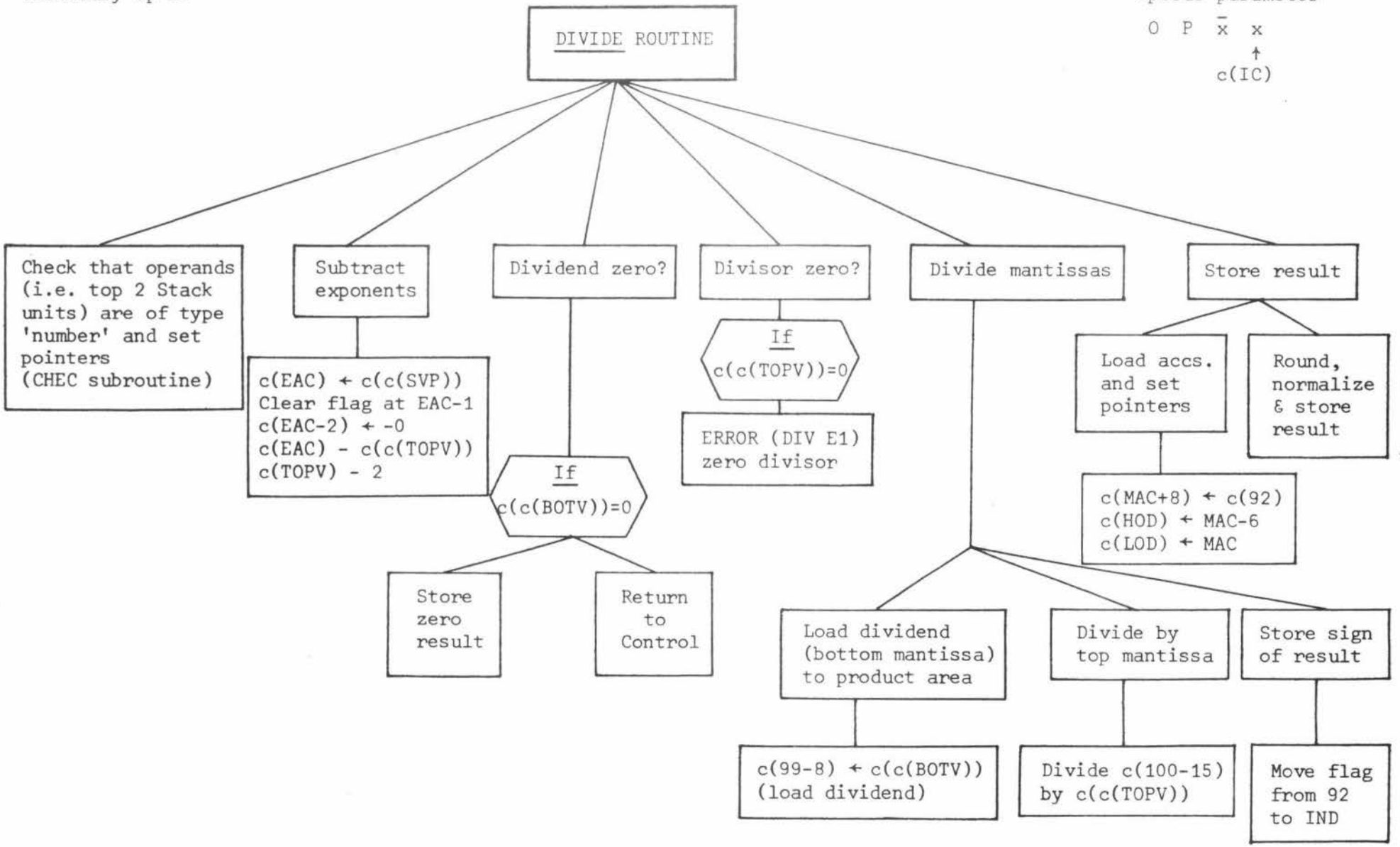
Secondary op 14

opcode parameter

O P $\bar{x}$ x

↑

c(IC)



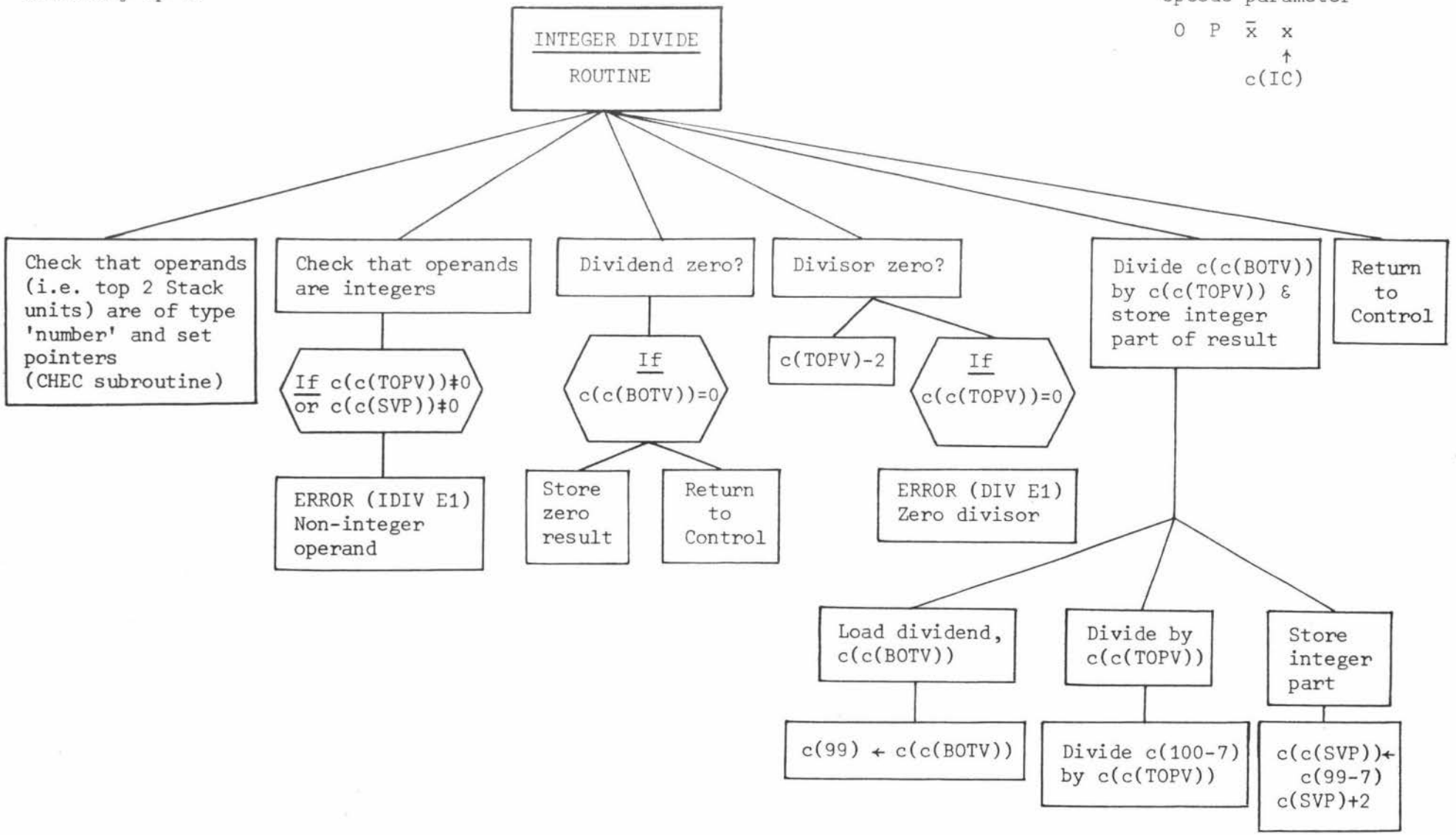
Secondary op 15

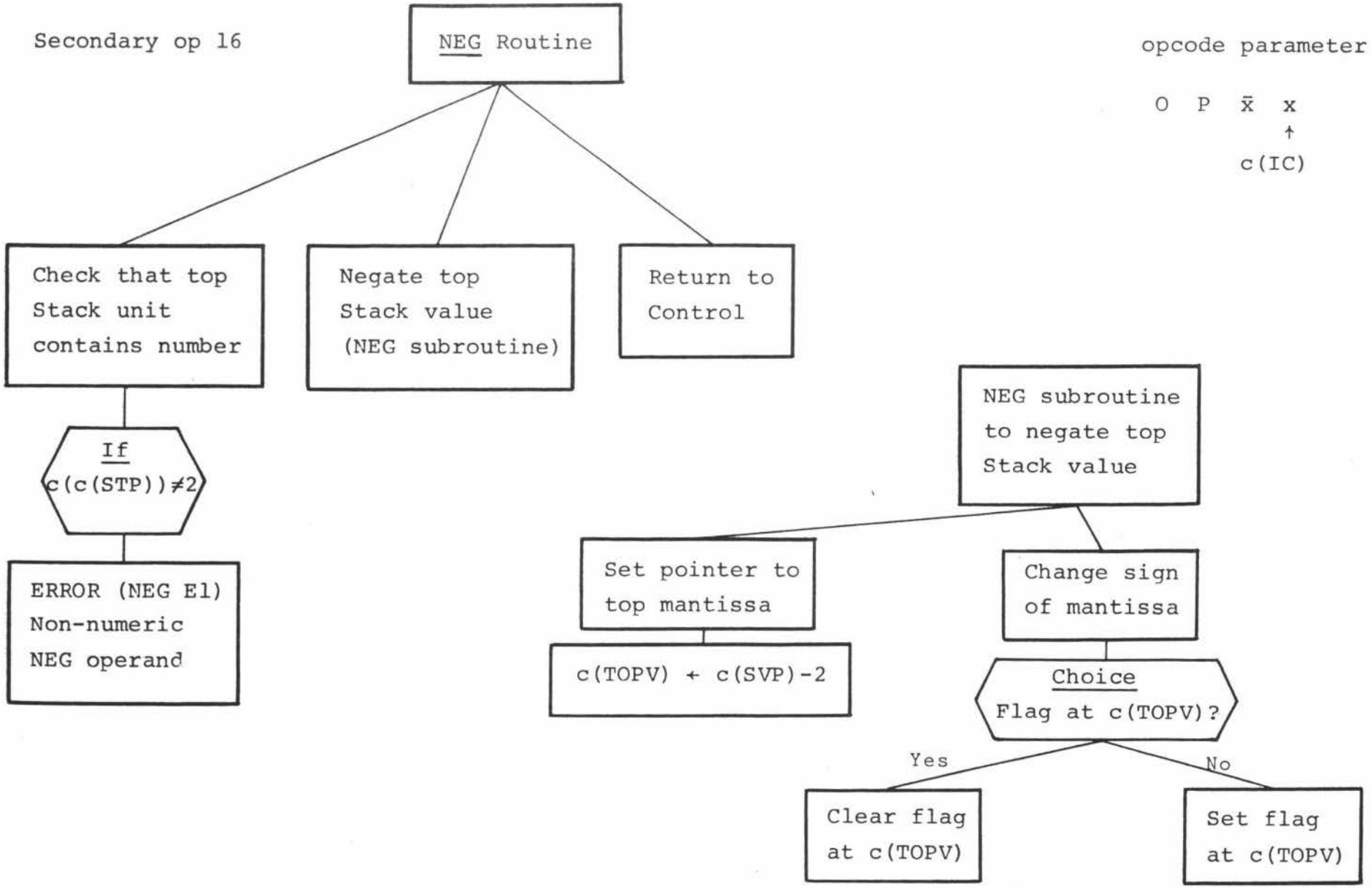
opcode parameter

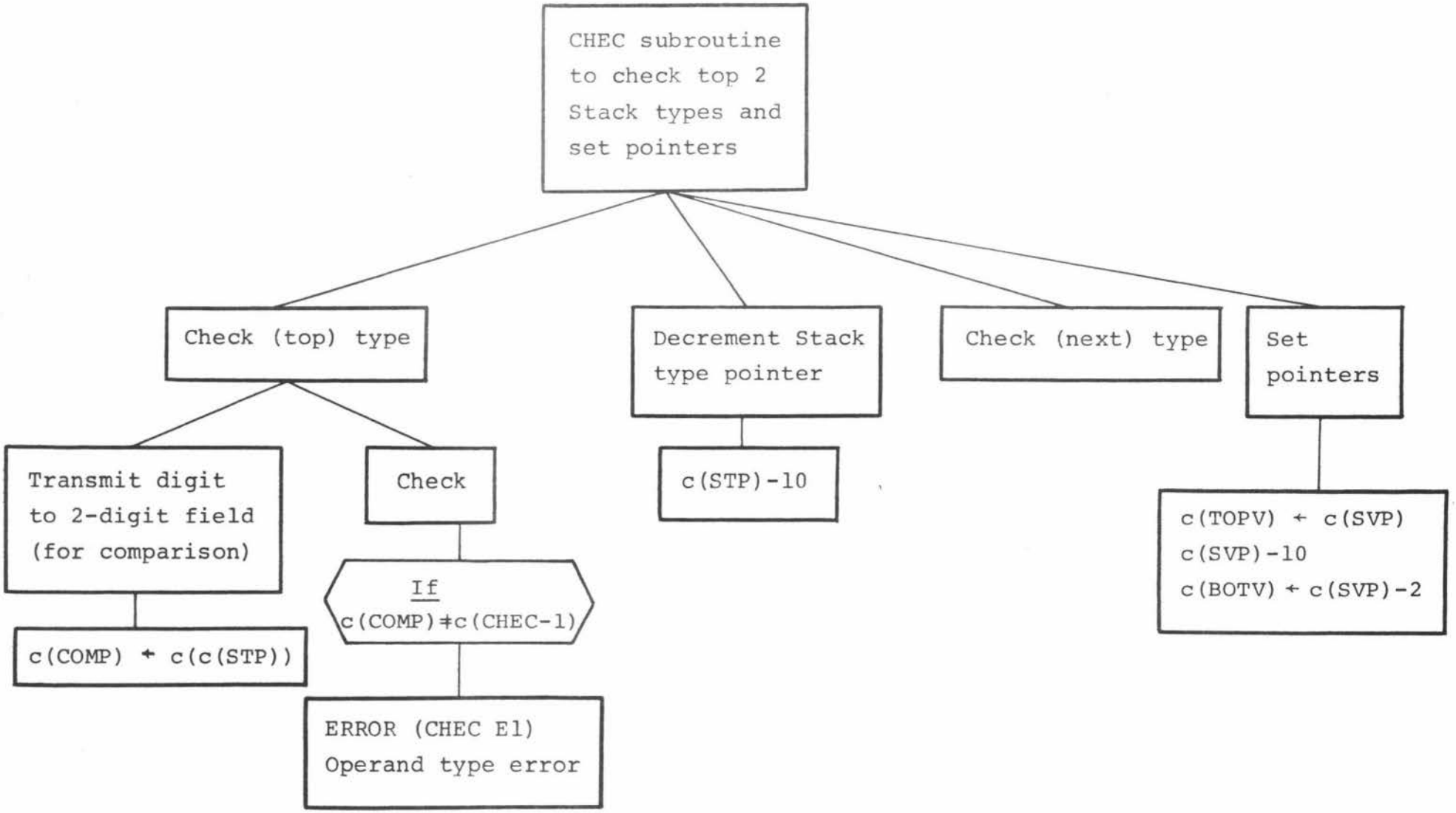
O P $\bar{x}$ x

↑

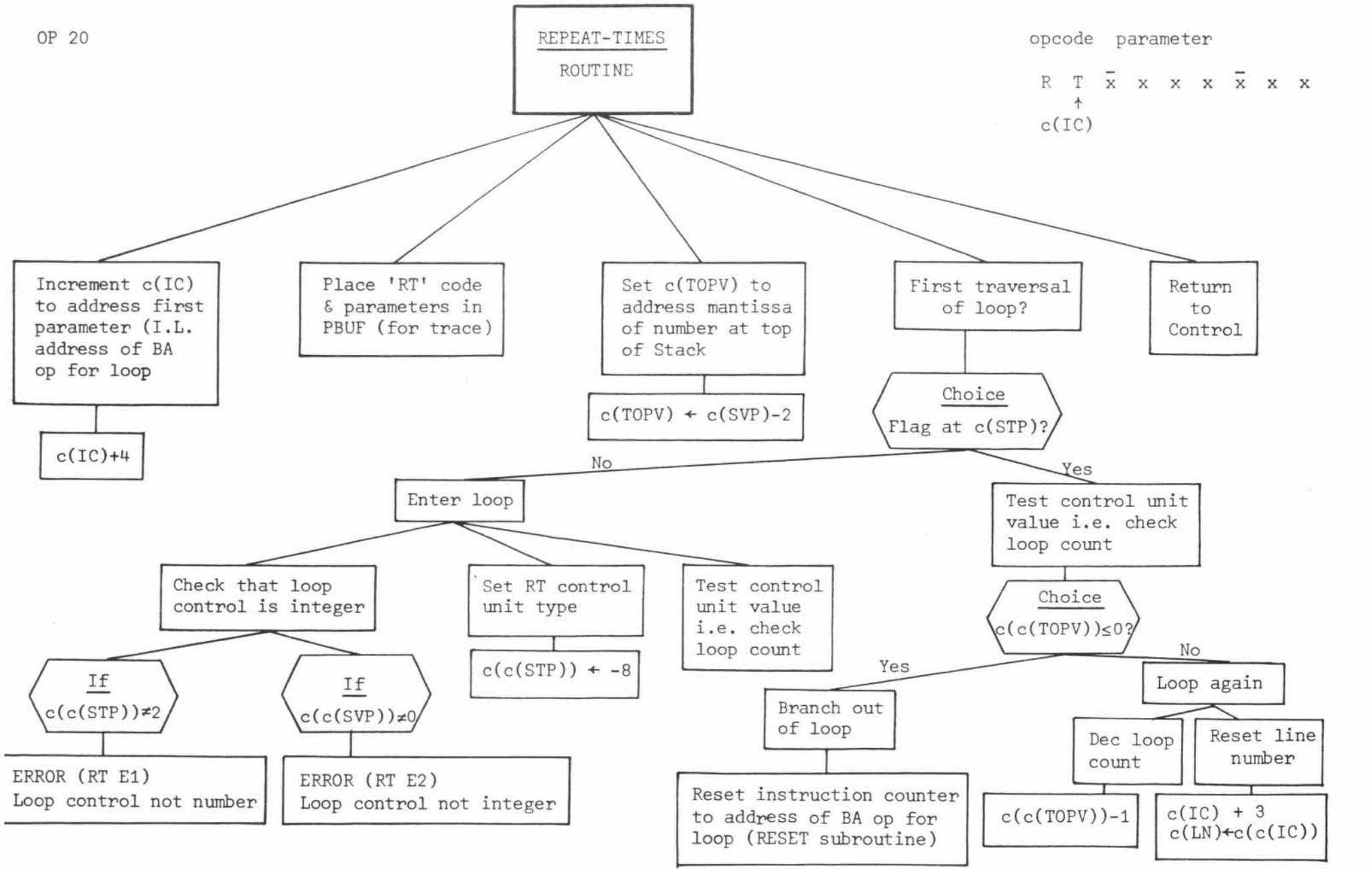
c(IC)



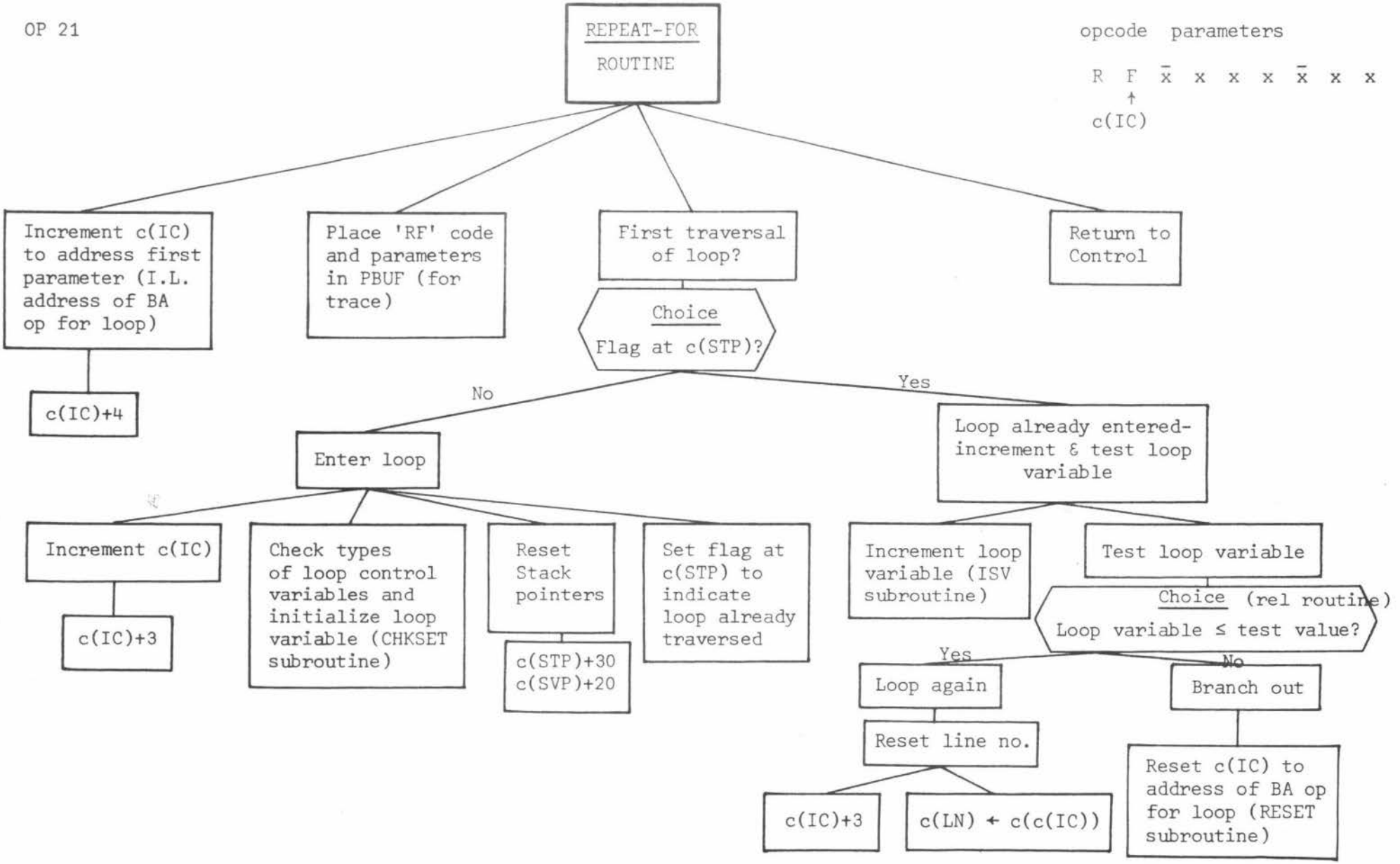

Secondary op 16
NEG Routine
opcode parameter
O P x̄ x
c(IC)
Check that top Stack unit contains number
Negate top Stack value (NEG subroutine)
Return to Control
If c(c(STP))≠2
ERROR (NEG E1) Non-numeric NEG operand
NEG subroutine to negate top Stack value
Set pointer to top mantissa
c(TOPV) ← c(SVP)-2
Change sign of mantissa
Choice Flag at c(TOPV)?
Yes
No
Clear flag at c(TOPV)
Set flag at c(TOPV)


CHEC subroutine to check top 2 Stack types and set pointers
Check (top) type
Decrement Stack type pointer
Check (next) type
Set pointers
Transmit digit to 2-digit field (for comparison)
Check
c(STP)-10
c(TOPV) ← c(SVP)
c(SVP)-10
c(BOTV) ← c(SVP)-2
c(COMP) ← c(c(STP))
If
c(COMP)≠c(CHEC-1)
ERROR (CHEC E1)
Operand type error

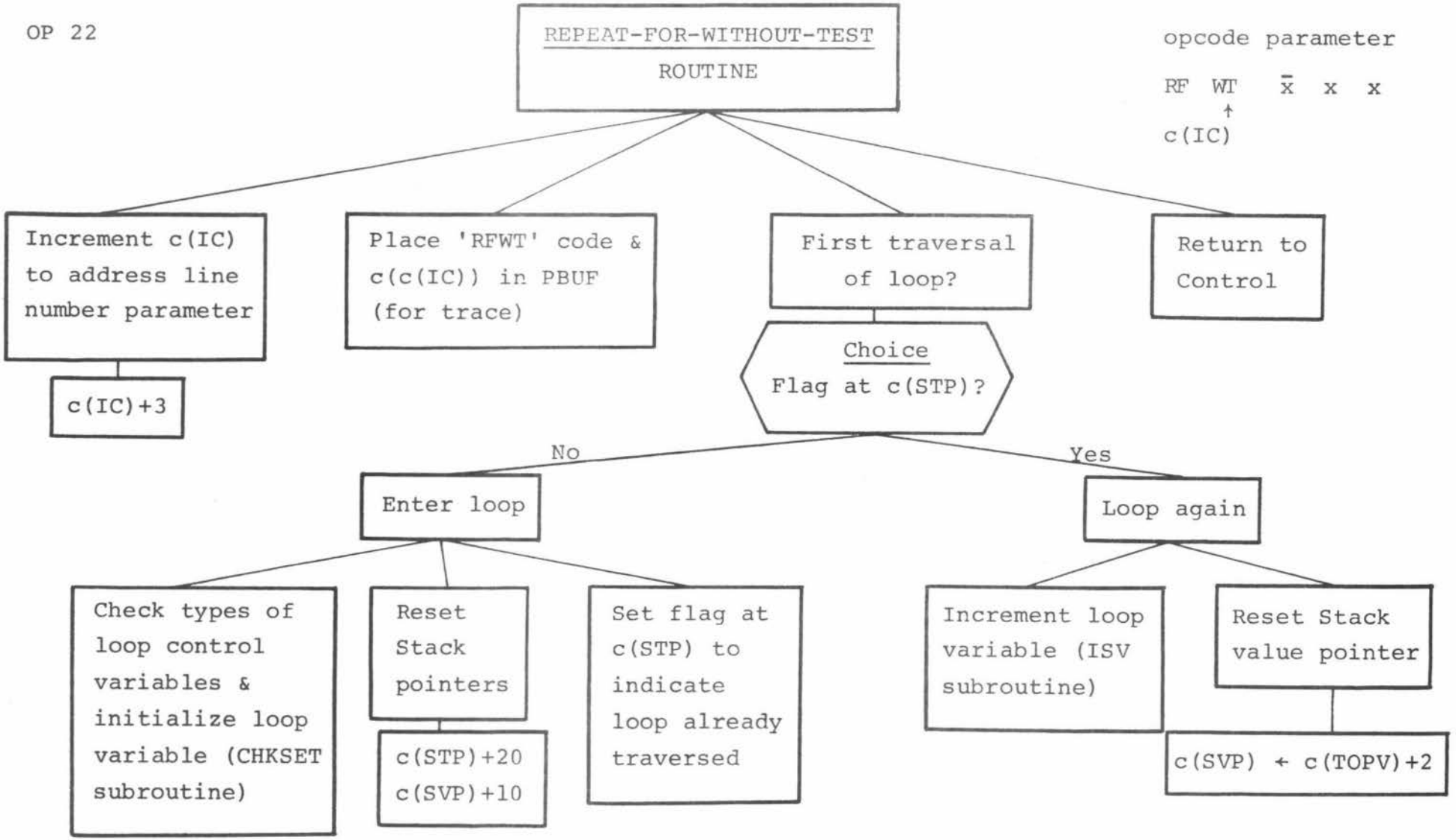
OP 20


REPEAT-TIMES
ROUTINE
opcode parameter
R T x̄ x x x x̄ x x
c(IC)
Increment c(IC) to address first parameter (I.L. address of BA op for loop
c(IC)+4
Place 'RT' code & parameters in PBUF (for trace)
Set c(TOPV) to address mantissa of number at top of Stack
c(TOPV) ← c(SVP)-2
First traversal of loop?
Choice
Flag at c(STP)?
No
Yes
Return to Control
Enter loop
Check that loop control is integer
If
c(c(STP))≠2
ERROR (RT E1)
Loop control not number
If
c(c(SVP))≠0
ERROR (RT E2)
Loop control not integer
Set RT control unit type
c(c(STP)) ← -8
Test control unit value i.e. check loop count
Test control unit value i.e. check loop count
Choice
c(c(TOPV))≤0?
Yes
No
Branch out of loop
Reset instruction counter to address of BA op for loop (RESET subroutine)
Loop again
Dec loop count
c(c(TOPV))-1
Reset line number
c(IC) + 3
c(LN)←c(c(IC))

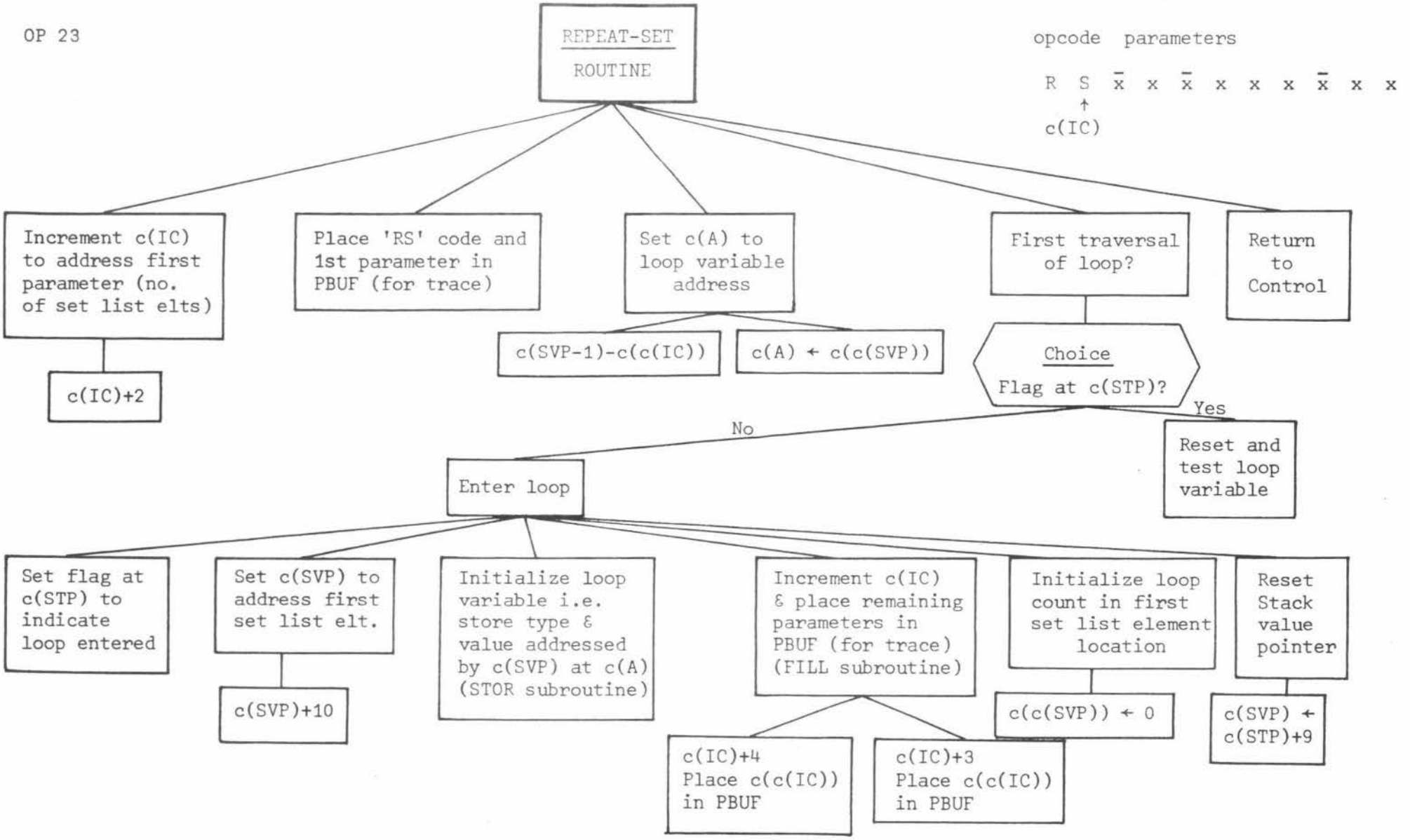
OP 21

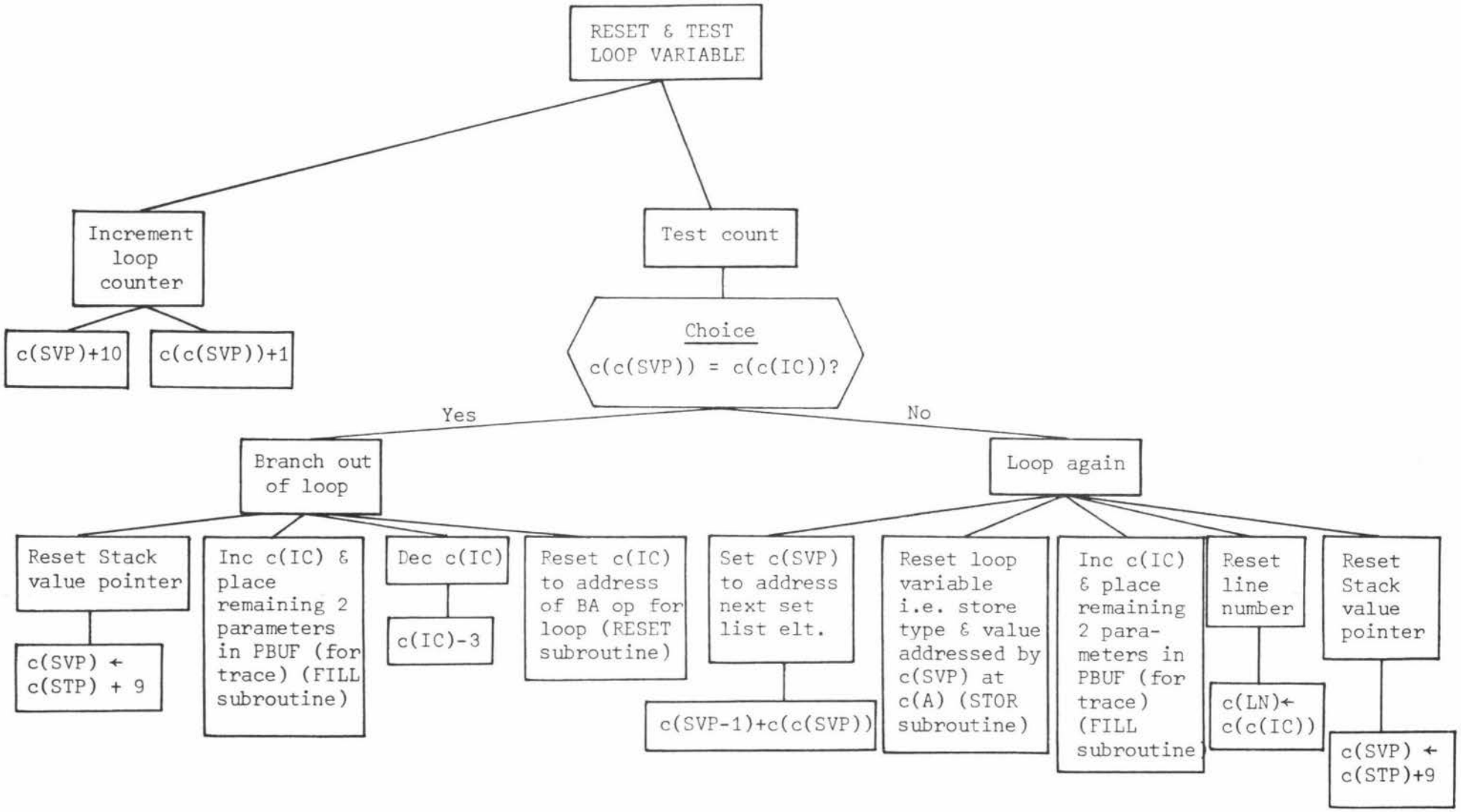

REPEAT-FOR ROUTINE
opcode parameters
R F x̄ x x x x̄ x x
c(IC)
Increment c(IC) to address first parameter (I.L. address of BA op for loop)
c(IC)+4
Place 'RF' code and parameters in PBUF (for trace)
First traversal of loop?
Choice
Flag at c(STP)?
No
Yes
Return to Control
Enter loop
Increment c(IC)
c(IC)+3
Check types of loop control variables and initialize loop variable (CHKSET subroutine)
Reset Stack pointers
c(STP)+30
c(SVP)+20
Set flag at c(STP) to indicate loop already traversed
Loop already entered-increment & test loop variable
Increment loop variable (ISV subroutine)
Test loop variable
Choice (rel routine)
Loop variable ≤ test value?
Yes
No
Loop again
Reset line no.
c(IC)+3
c(LN) ← c(c(IC))
Branch out
Reset c(IC) to address of BA op for loop (RESET subroutine)

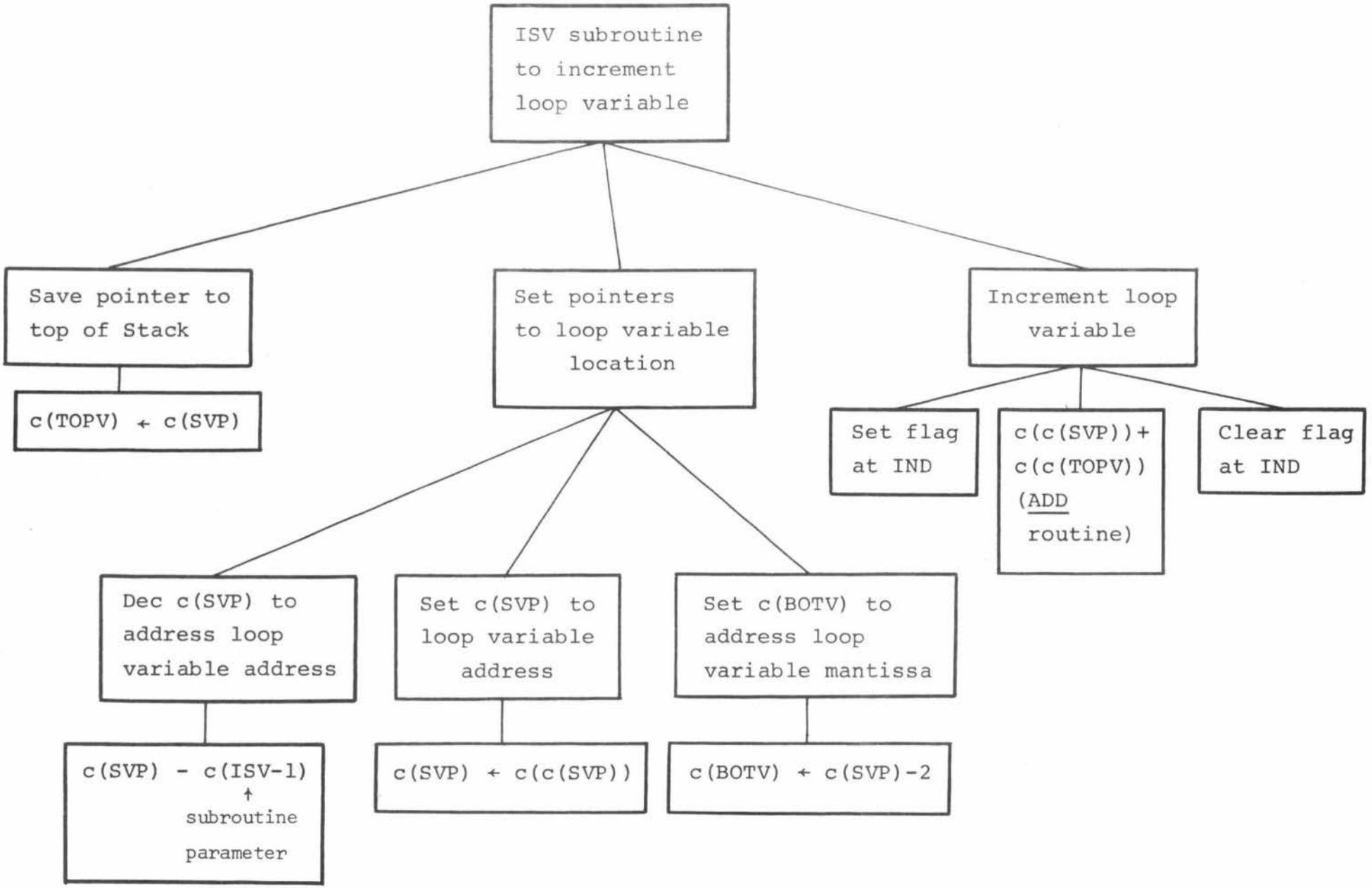
OP 22

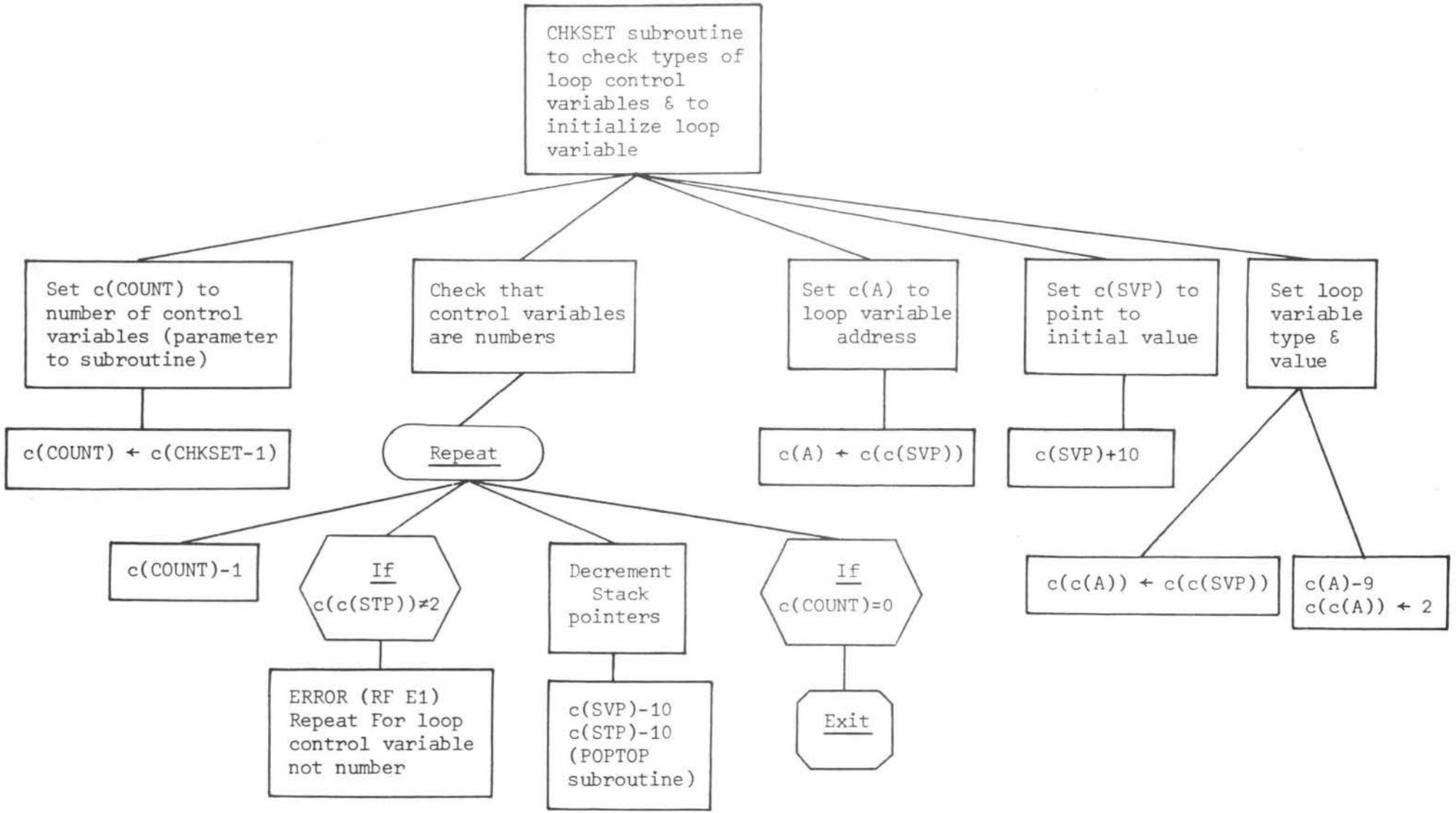

REPEAT-FOR-WITHOUT-TEST
ROUTINE
opcode parameter
RF WT x̄ x x
c(IC)
Increment c(IC) to address line number parameter
c(IC)+3
Place 'RFWT' code & c(c(IC)) in PBUF (for trace)
First traversal of loop?
Return to Control
Choice
Flag at c(STP)?
No
Yes
Enter loop
Loop again
Check types of loop control variables & initialize loop variable (CHKSET subroutine)
Reset Stack pointers
c(STP)+20
c(SVP)+10
Set flag at c(STP) to indicate loop already traversed
Increment loop variable (ISV subroutine)
Reset Stack value pointer
c(SVP) ← c(TOPV)+2

OP 23


REPEAT-SET
ROUTINE
opcode parameters
R S x̄ x x̄ x x x x̄ x x
c(IC)
Increment c(IC) to address first parameter (no. of set list elts)
c(IC)+2
Place 'RS' code and 1st parameter in PBUF (for trace)
Set c(A) to loop variable address
c(SVP-1)-c(c(IC))
c(A) ← c(c(SVP))
First traversal of loop?
Return to Control
Choice
Flag at c(STP)?
Yes
No
Reset and test loop variable
Enter loop
Set flag at c(STP) to indicate loop entered
Set c(SVP) to address first set list elt.
c(SVP)+10
Initialize loop variable i.e. store type & value addressed by c(SVP) at c(A) (STOR subroutine)
Increment c(IC) & place remaining parameters in PBUF (for trace) (FILL subroutine)
c(IC)+4 Place c(c(IC)) in PBUF
c(IC)+3 Place c(c(IC)) in PBUF
Initialize loop count in first set list element location
c(c(SVP)) ← 0
Reset Stack value pointer
c(SVP) ← c(STP)+9

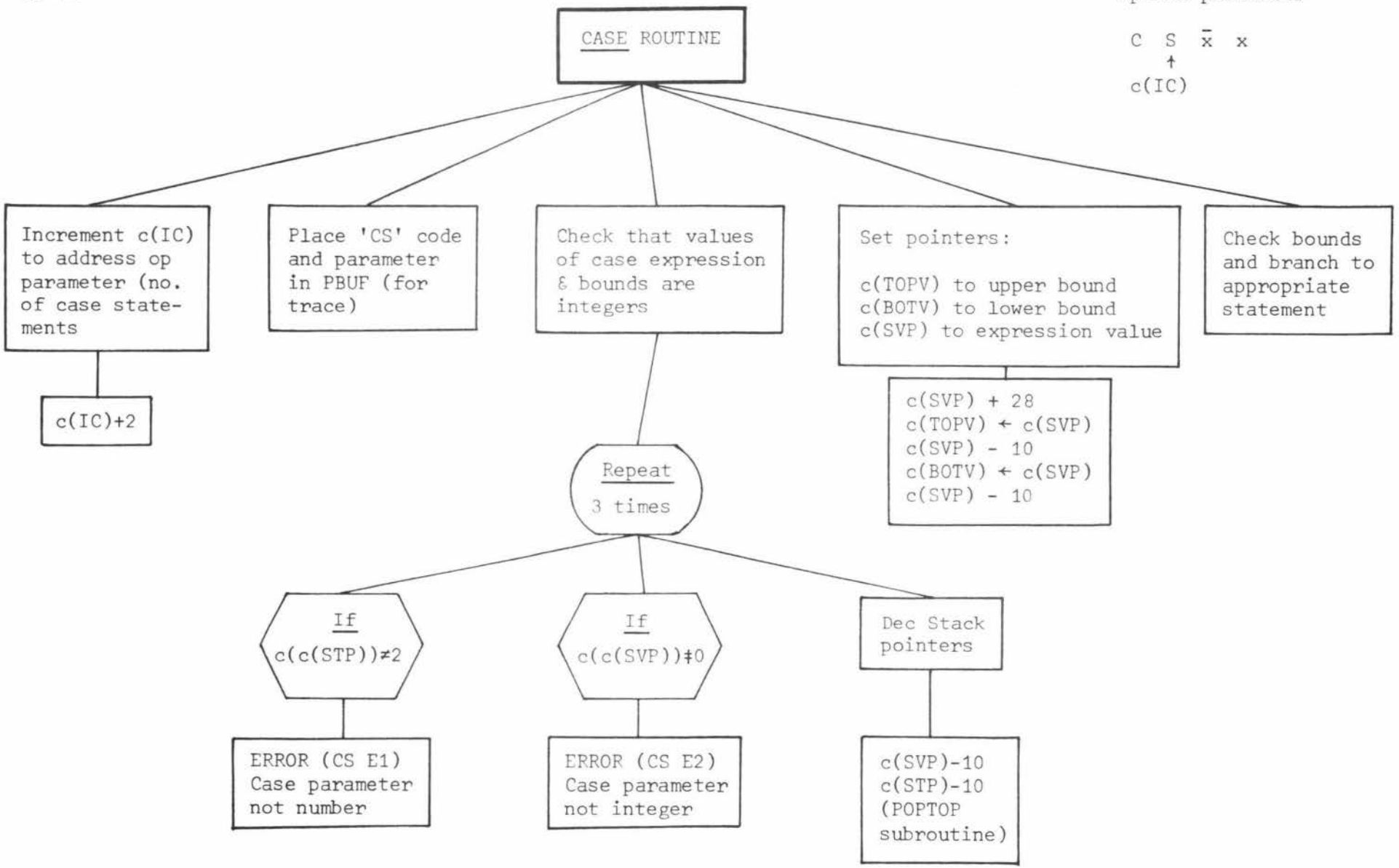

RESET & TEST LOOP VARIABLE
Increment loop counter
c(SVP)+10
c(c(SVP))+1
Test count
Choice
c(c(SVP)) = c(c(IC))?
Yes
No
Branch out of loop
Reset Stack value pointer
c(SVP) ← c(STP) + 9
Inc c(IC) & place remaining 2 parameters in PBUF (for trace) (FILL subroutine)
Dec c(IC)
c(IC)-3
Reset c(IC) to address of BA op for loop (RESET subroutine)
Loop again
Set c(SVP) to address next set list elt.
c(SVP-1)+c(c(SVP))
Reset loop variable i.e. store type & value addressed by c(SVP) at c(A) (STOR subroutine)
Inc c(IC) & place remaining 2 parameters in PBUF (for trace) (FILL subroutine)
Reset line number
c(LN)← c(c(IC))
Reset Stack value pointer
c(SVP) ← c(STP)+9

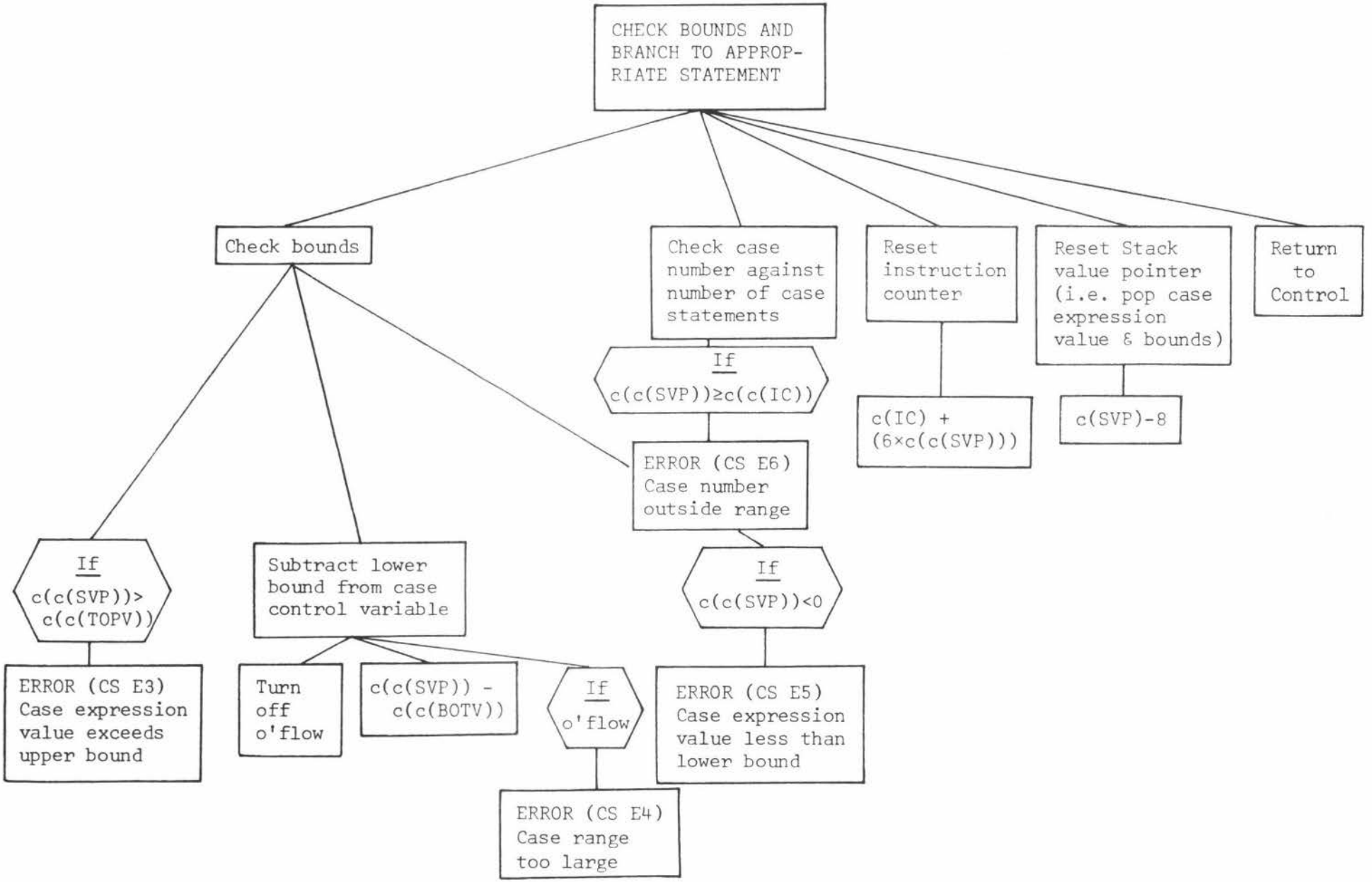

ISV subroutine to increment loop variable
Save pointer to top of Stack
c(TOPV) ← c(SVP)
Set pointers to loop variable location
Increment loop variable
Set flag at IND
c(c(SVP))+ c(c(TOPV)) (ADD routine)
Clear flag at IND
Dec c(SVP) to address loop variable address
Set c(SVP) to loop variable address
Set c(BOTV) to address loop variable mantissa
c(SVP) - c(ISV-1) ↑ subroutine parameter
c(SVP) ← c(c(SVP))
c(BOTV) ← c(SVP)-2


CHKSET subroutine to check types of loop control variables & to initialize loop variable
Set c(COUNT) to number of control variables (parameter to subroutine)
Check that control variables are numbers
Set c(A) to loop variable address
Set c(SVP) to point to initial value
Set loop variable type & value
c(COUNT) ← c(CHKSET-1)
Repeat
c(A) ← c(c(SVP))
c(SVP)+10
c(COUNT)-1
If c(c(STP))≠2
Decrement Stack pointers
If c(COUNT)=0
c(c(A)) ← c(c(SVP))
c(A)-9
c(c(A)) ← 2
ERROR (RF E1) Repeat For loop control variable not number
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
Exit

opcode parameter

C S $\bar{x}$ x
↑
c(IC)



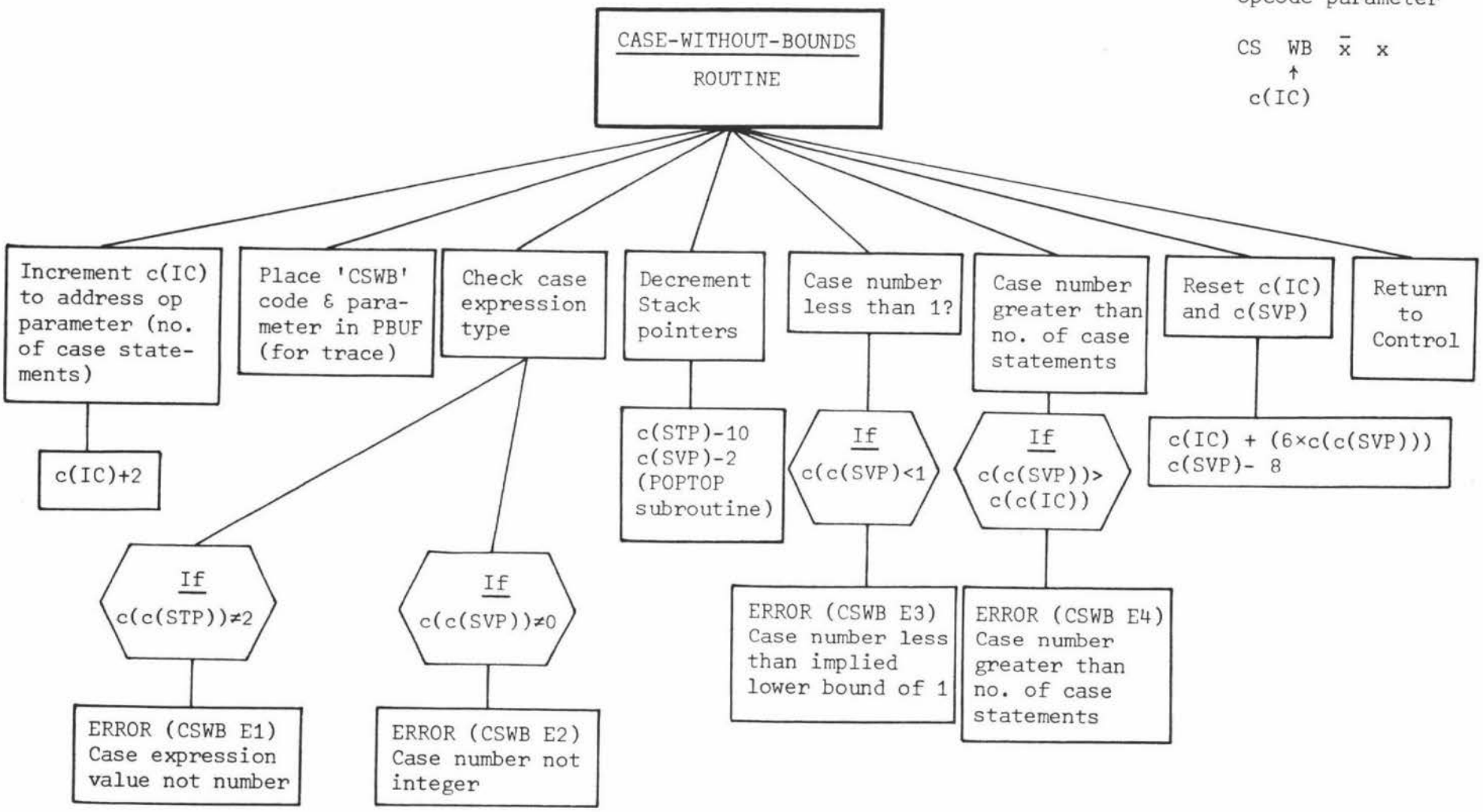

CHECK BOUNDS AND BRANCH TO APPROP-RIATE STATEMENT
Check bounds
Check case number against number of case statements
Reset instruction counter
Reset Stack value pointer (i.e. pop case expression value & bounds)
Return to Control
If c(c(SVP))≥c(c(IC))
c(IC) + (6×c(c(SVP)))
c(SVP)-8
ERROR (CS E6) Case number outside range
If c(c(SVP))<0
If c(c(SVP))> c(c(TOPV))
Subtract lower bound from case control variable
ERROR (CS E5) Case expression value less than lower bound
ERROR (CS E3) Case expression value exceeds upper bound
Turn off o'flow
c(c(SVP)) - c(c(BOTV))
If o'flow
ERROR (CS E4) Case range too large

opcode parameter

CS WB $\bar{x}$ x

↑

c(IC)

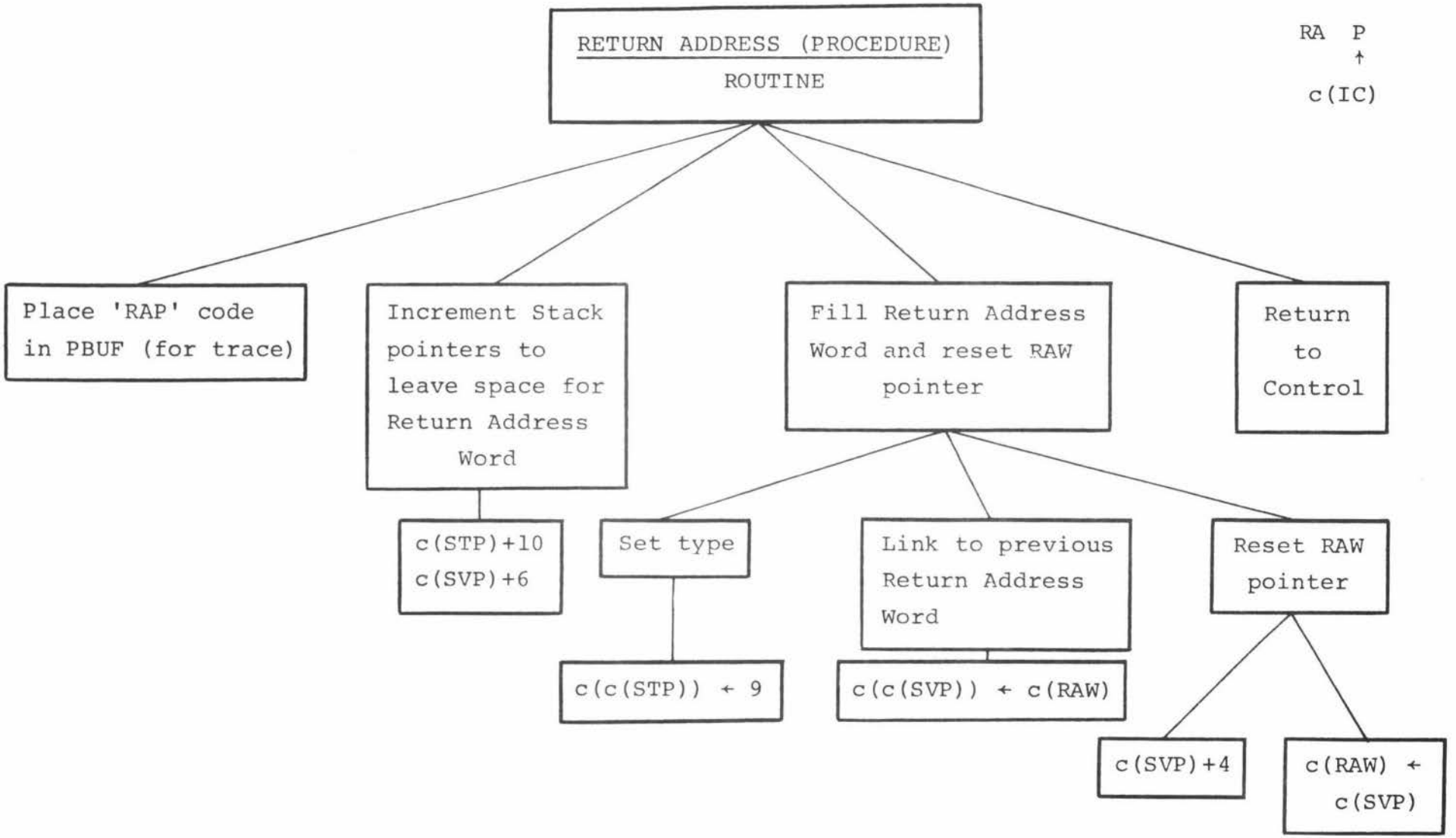


OP 26

opcode

RA P

c(IC)



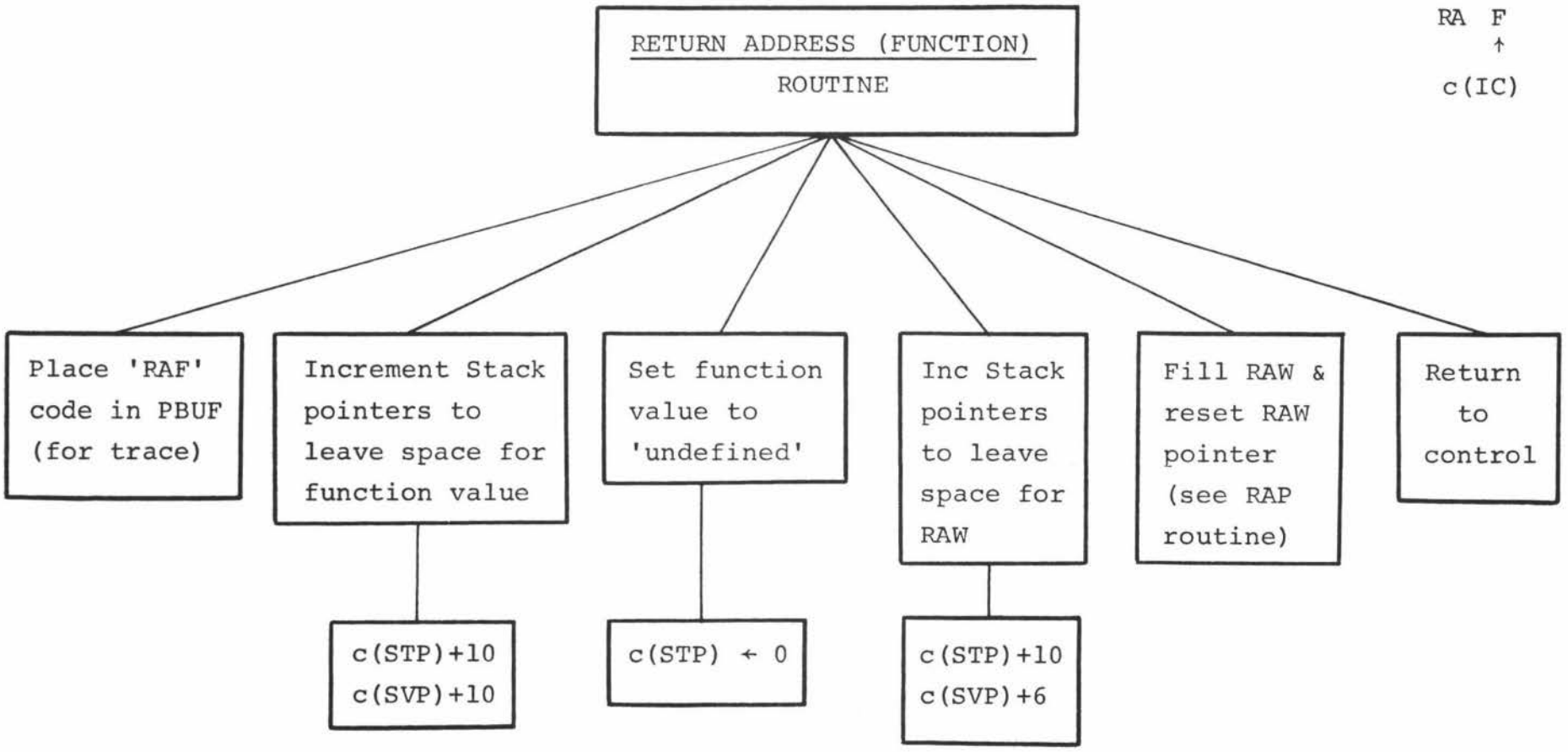
OP 27

opcode

RA F

↑

c(IC)



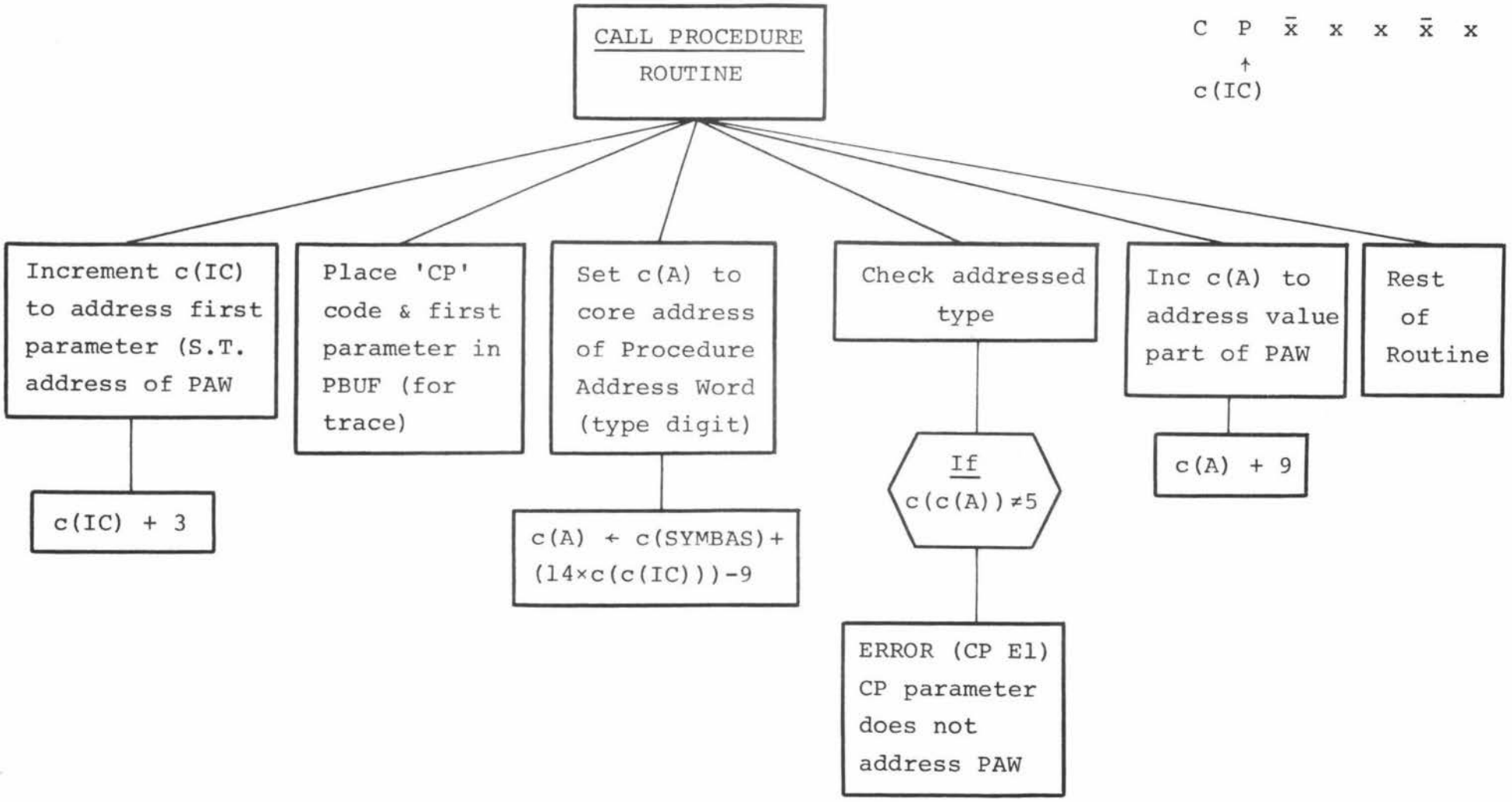
OP 28

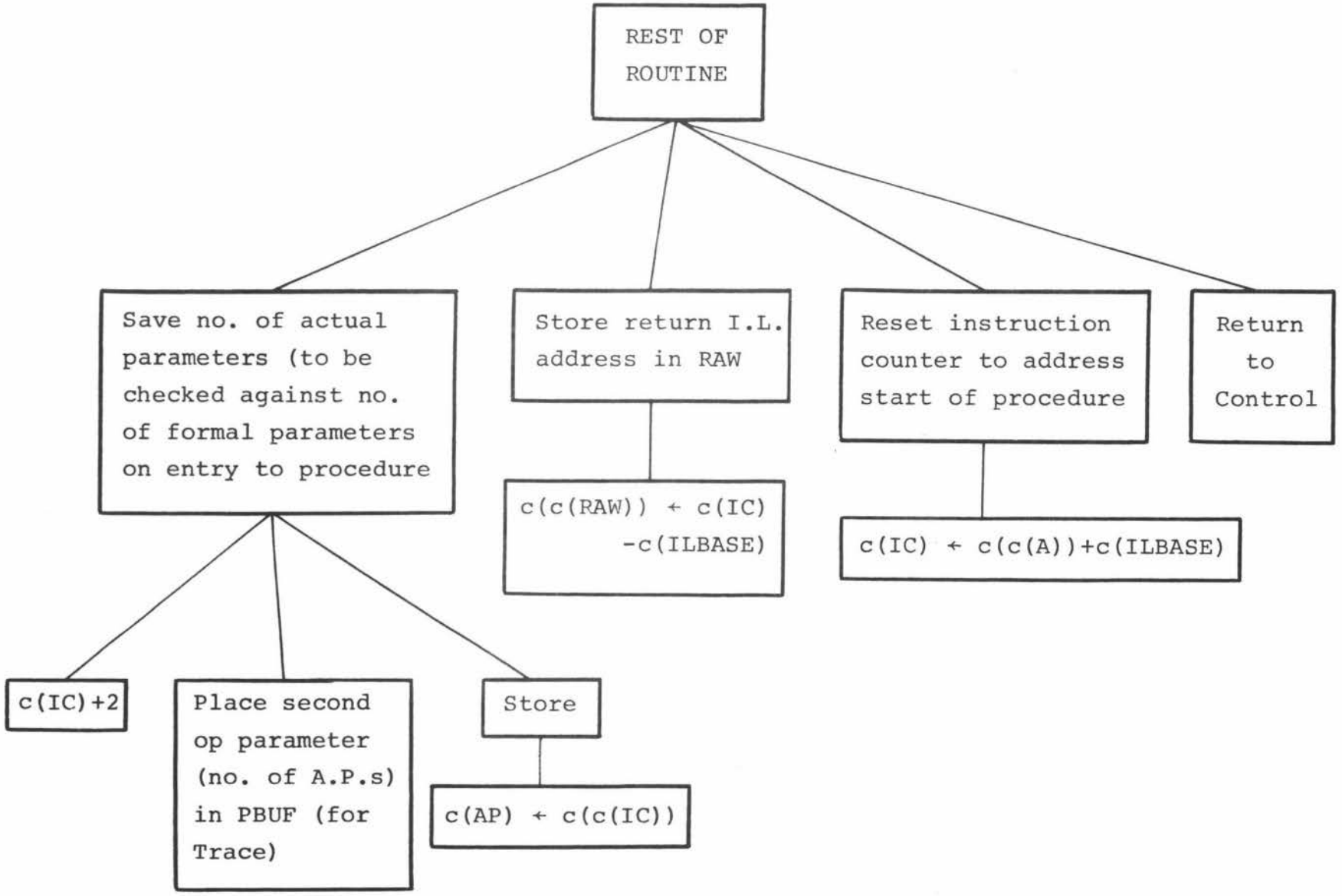
opcode parameters

C P $\bar{x}$ x x $\bar{x}$ x

↑

c(IC)



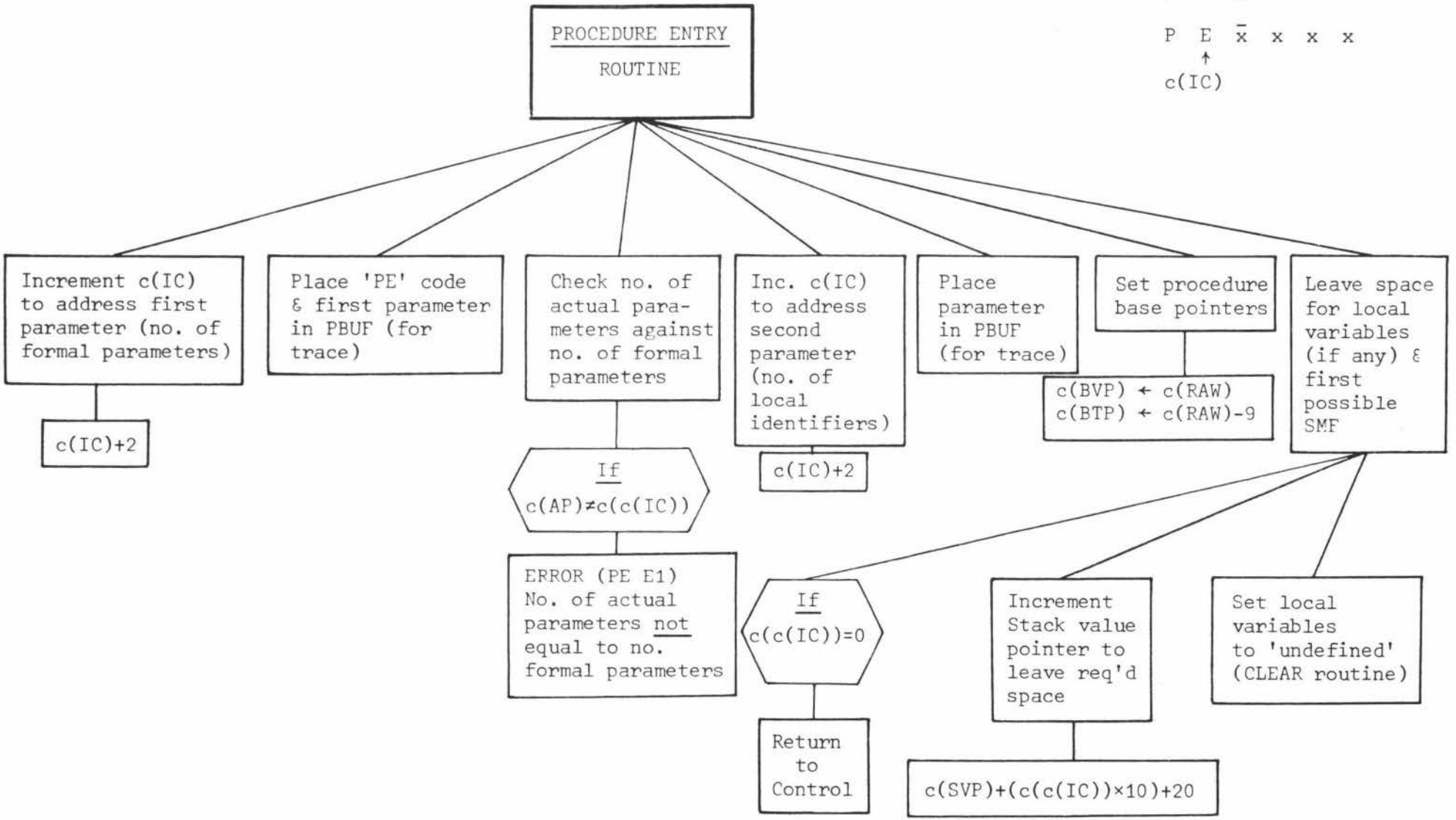

REST OF ROUTINE
Save no. of actual parameters (to be checked against no. of formal parameters on entry to procedure
Store return I.L. address in RAW
Reset instruction counter to address start of procedure
Return to Control
c(c(RAW)) ← c(IC) -c(ILBASE)
c(IC) ← c(c(A))+c(ILBASE)
c(IC)+2
Place second op parameter (no. of A.P.s) in PBUF (for Trace)
Store
c(AP) ← c(c(IC))

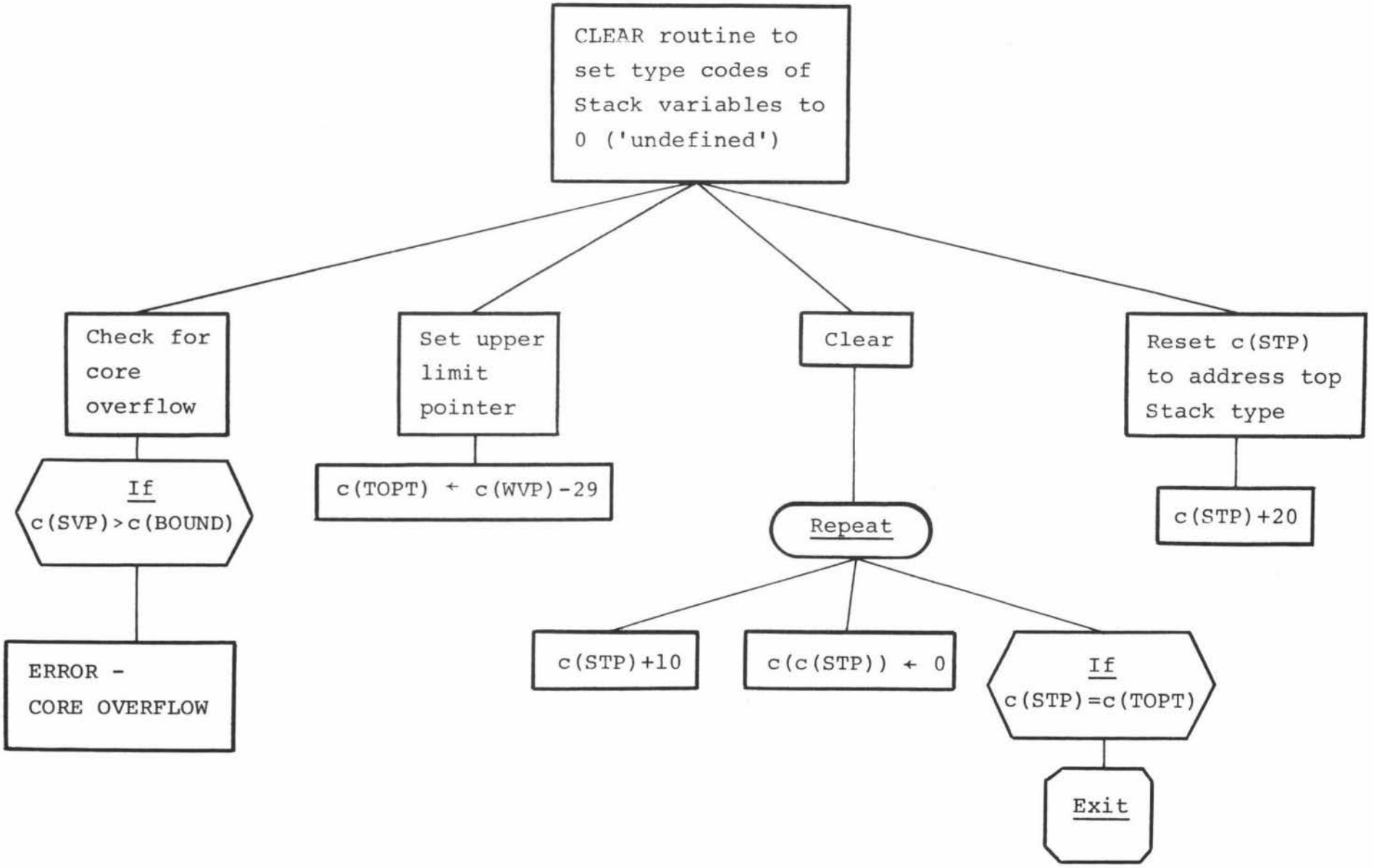
opcode parameters

P E $\bar{x}$ x x x

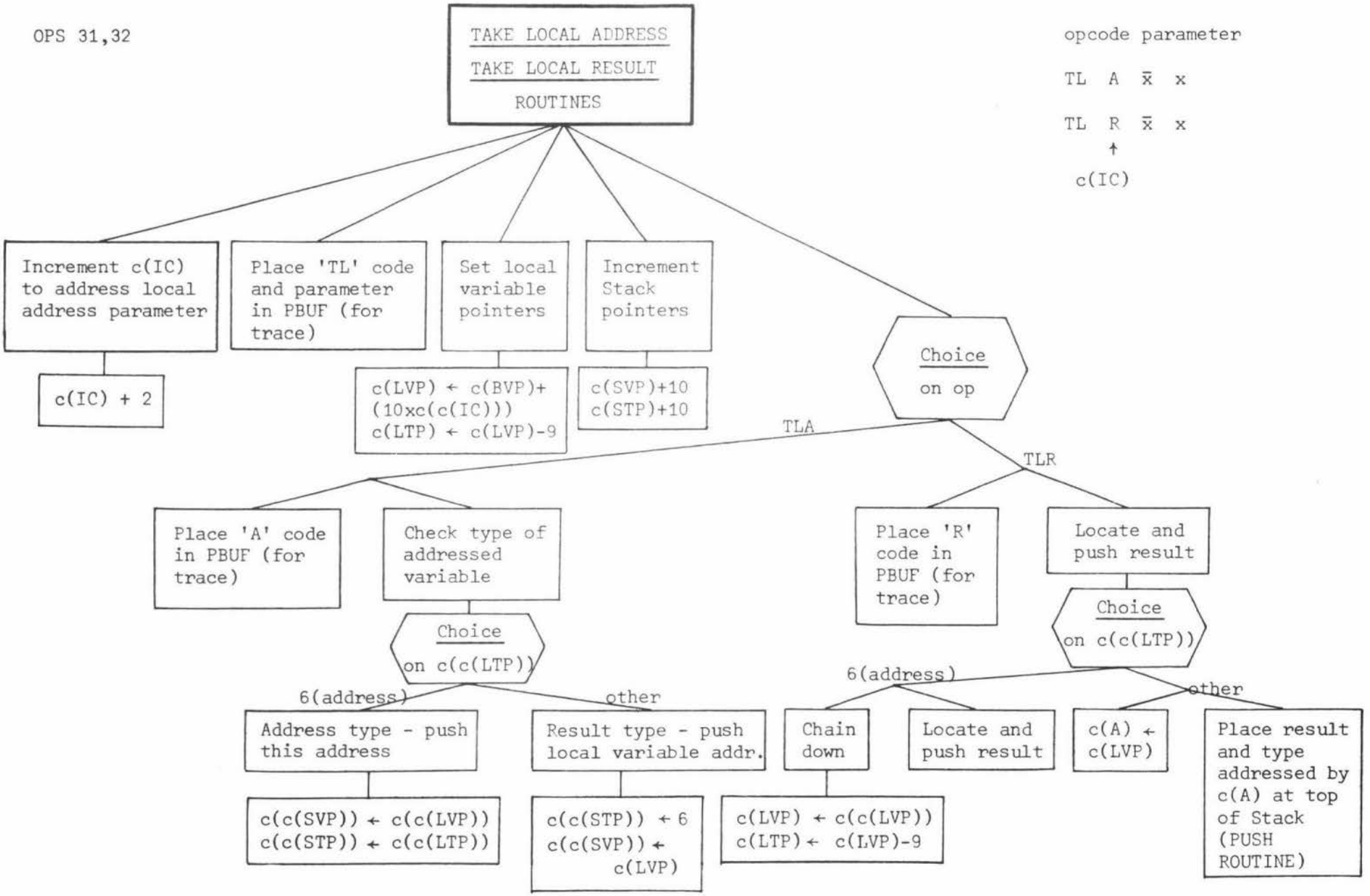
↑

c(IC)

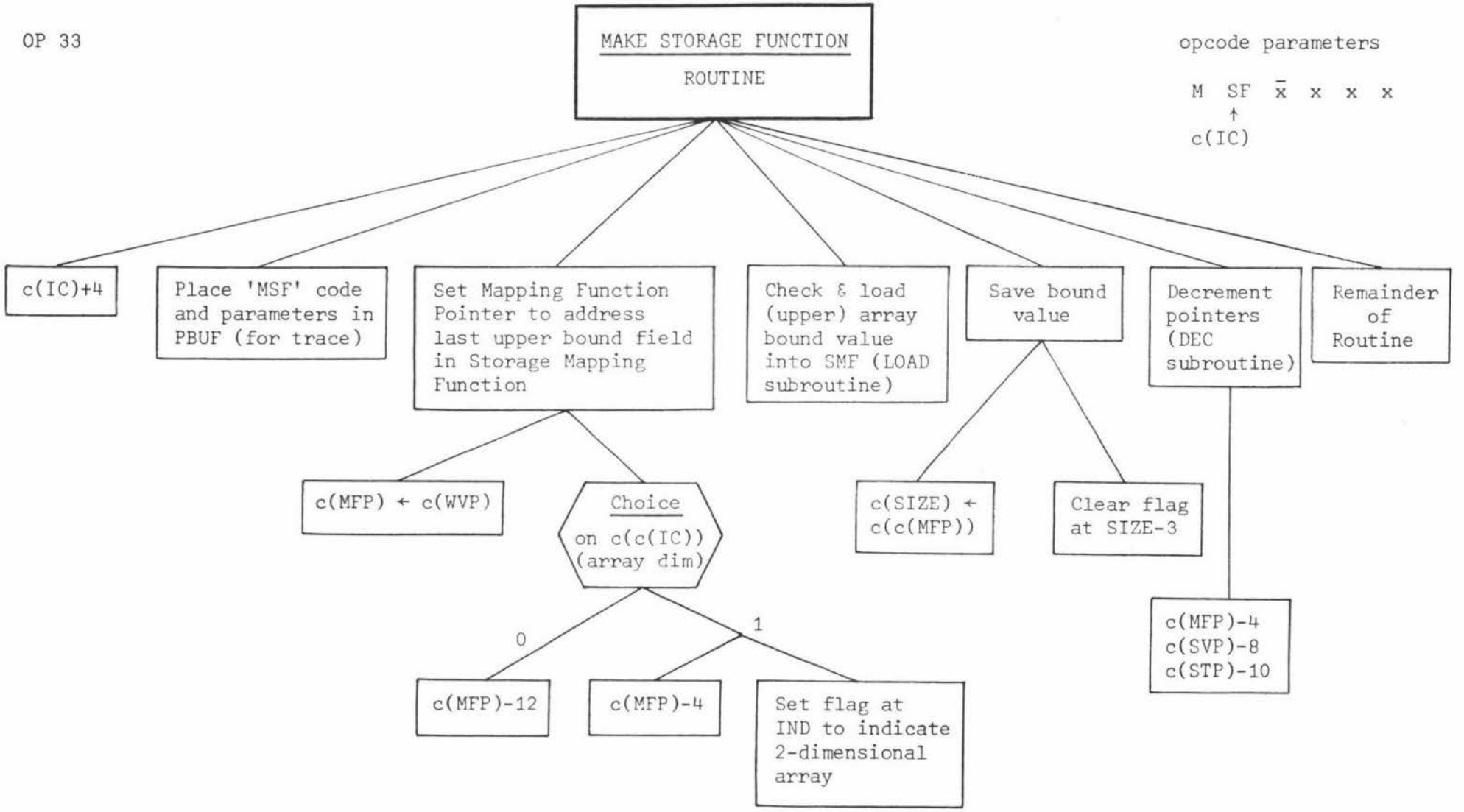


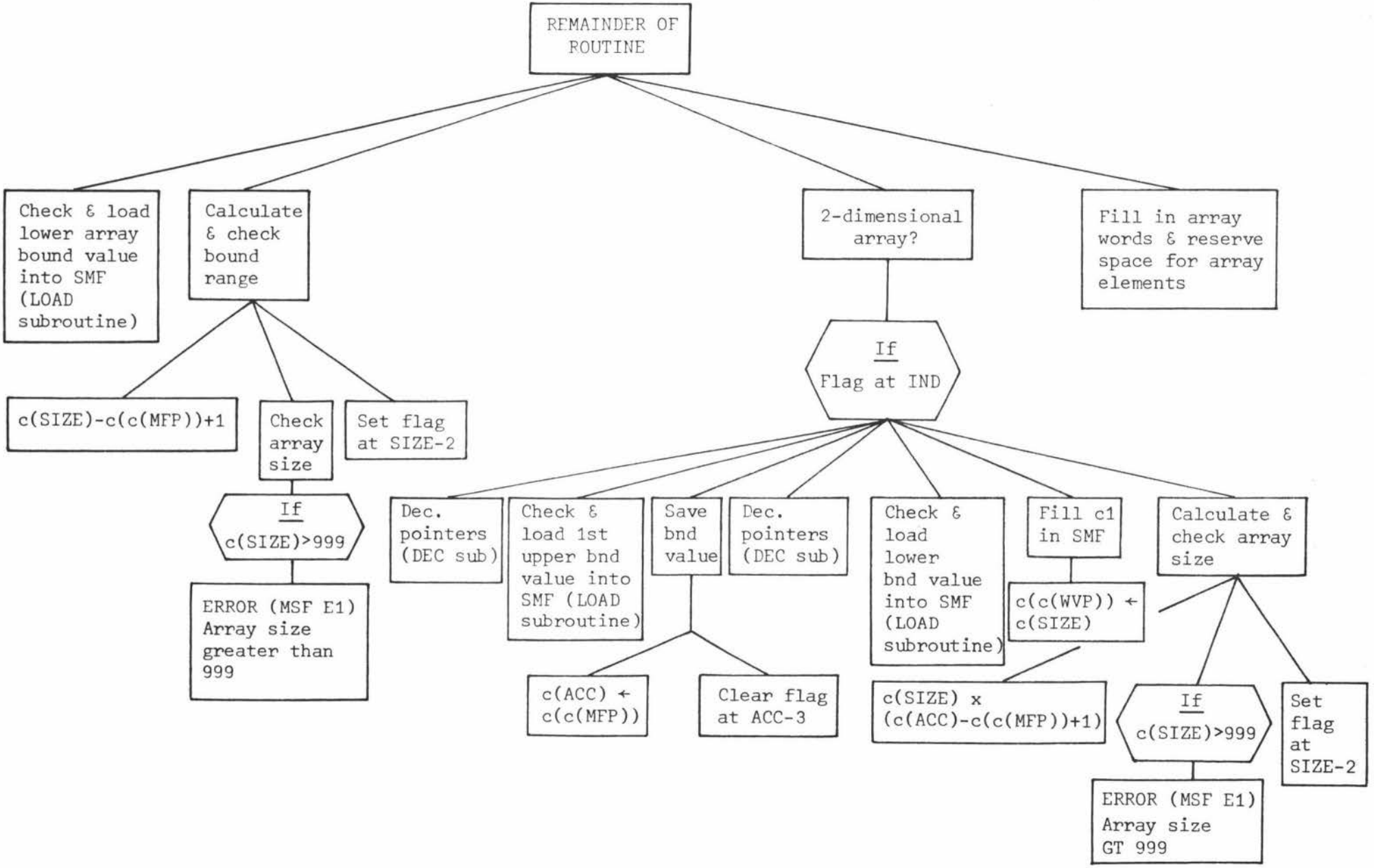

CLEAR routine to set type codes of Stack variables to 0 ('undefined')
Check for core overflow
If c(SVP)>c(BOUND)
ERROR - CORE OVERFLOW
Set upper limit pointer
c(TOPT) ← c(WVP)-29
Clear
Repeat
c(STP)+10
c(c(STP)) ← 0
If c(STP)=c(TOPT)
Exit
Reset c(STP) to address top Stack type
c(STP)+20

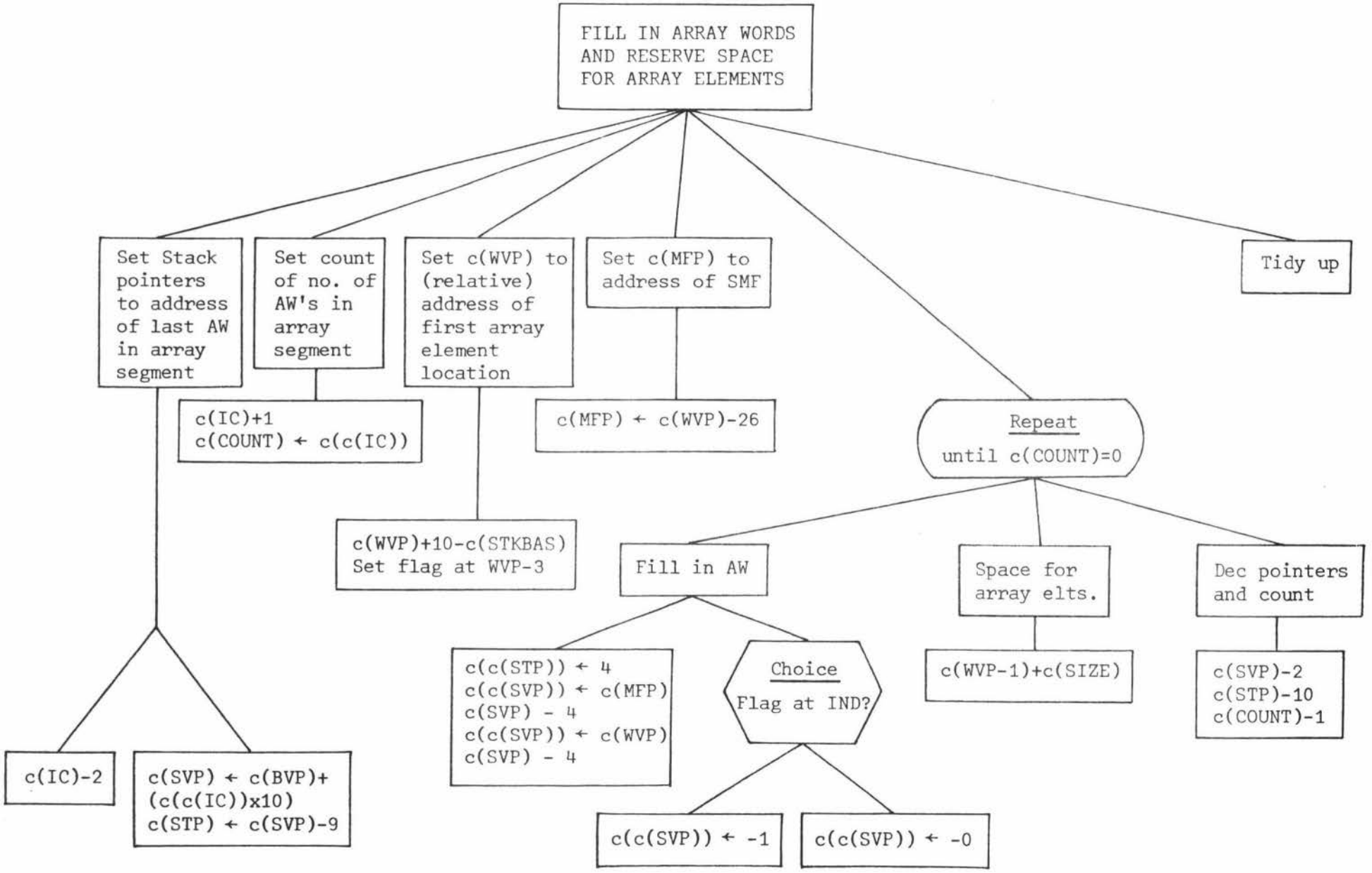
OPS 31,32

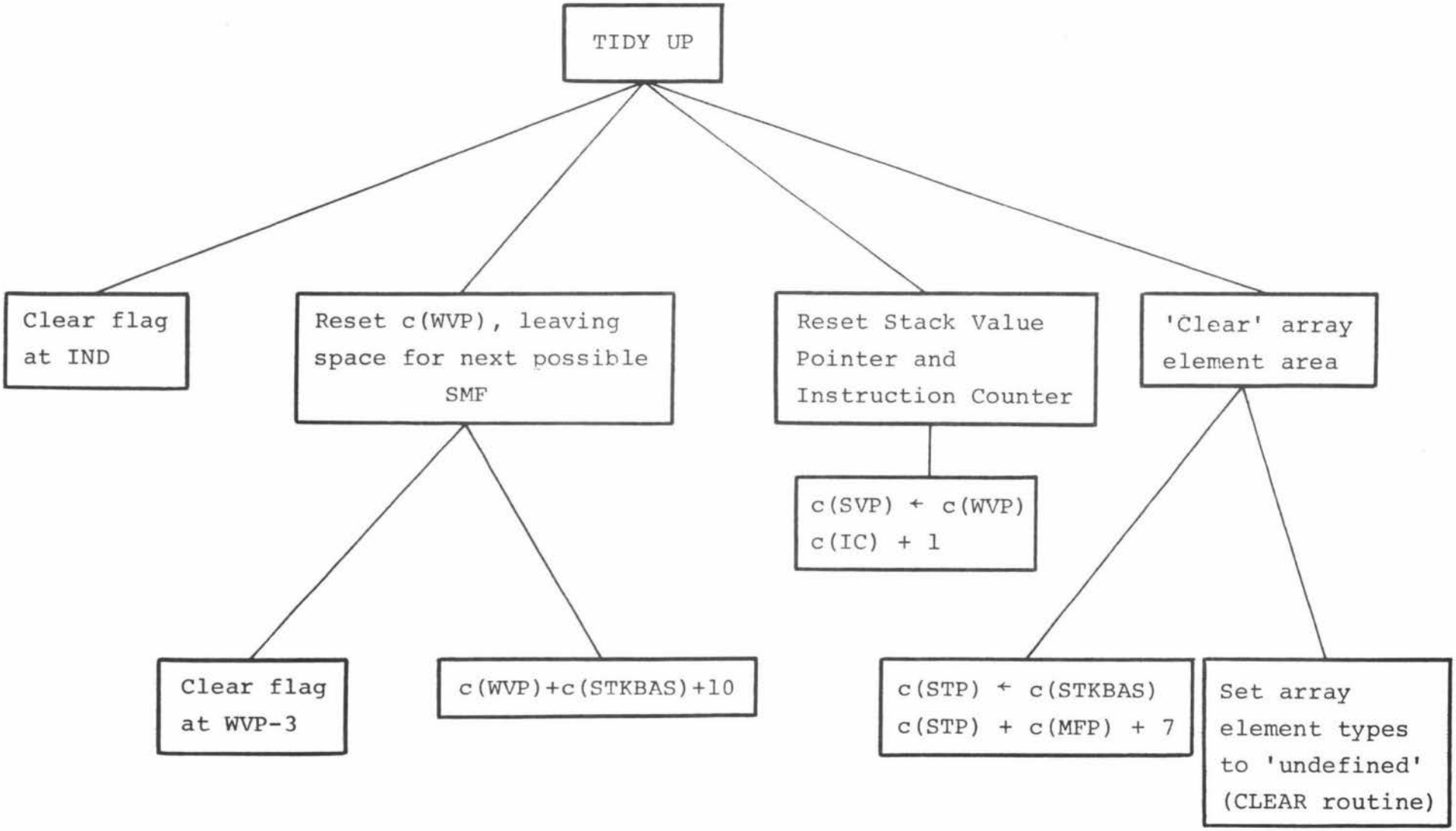

TAKE LOCAL ADDRESS
TAKE LOCAL RESULT
ROUTINES
opcode parameter
TL A x̄ x
TL R x̄ x
c(IC)
Increment c(IC) to address local address parameter
c(IC) + 2
Place 'TL' code and parameter in PBUF (for trace)
Set local variable pointers
c(LVP) ← c(BVP)+ (10xc(c(IC)))
c(LTP) ← c(LVP)-9
Increment Stack pointers
c(SVP)+10
c(STP)+10
Choice on op
TLA
TLR
Place 'A' code in PBUF (for trace)
Check type of addressed variable
Choice on c(c(LTP))
6(address)
other
Address type - push this address
c(c(SVP)) ← c(c(LVP))
c(c(STP)) ← c(c(LTP))
Result type - push local variable addr.
c(c(STP)) ← 6
c(c(SVP)) ← c(LVP)
Place 'R' code in PBUF (for trace)
Locate and push result
Choice on c(c(LTP))
6(address)
other
Chain down
c(LVP) ← c(c(LVP))
c(LTP) ← c(LVP)-9
Locate and push result
c(A) ← c(LVP)
Place result and type addressed by c(A) at top of Stack (PUSH ROUTINE)

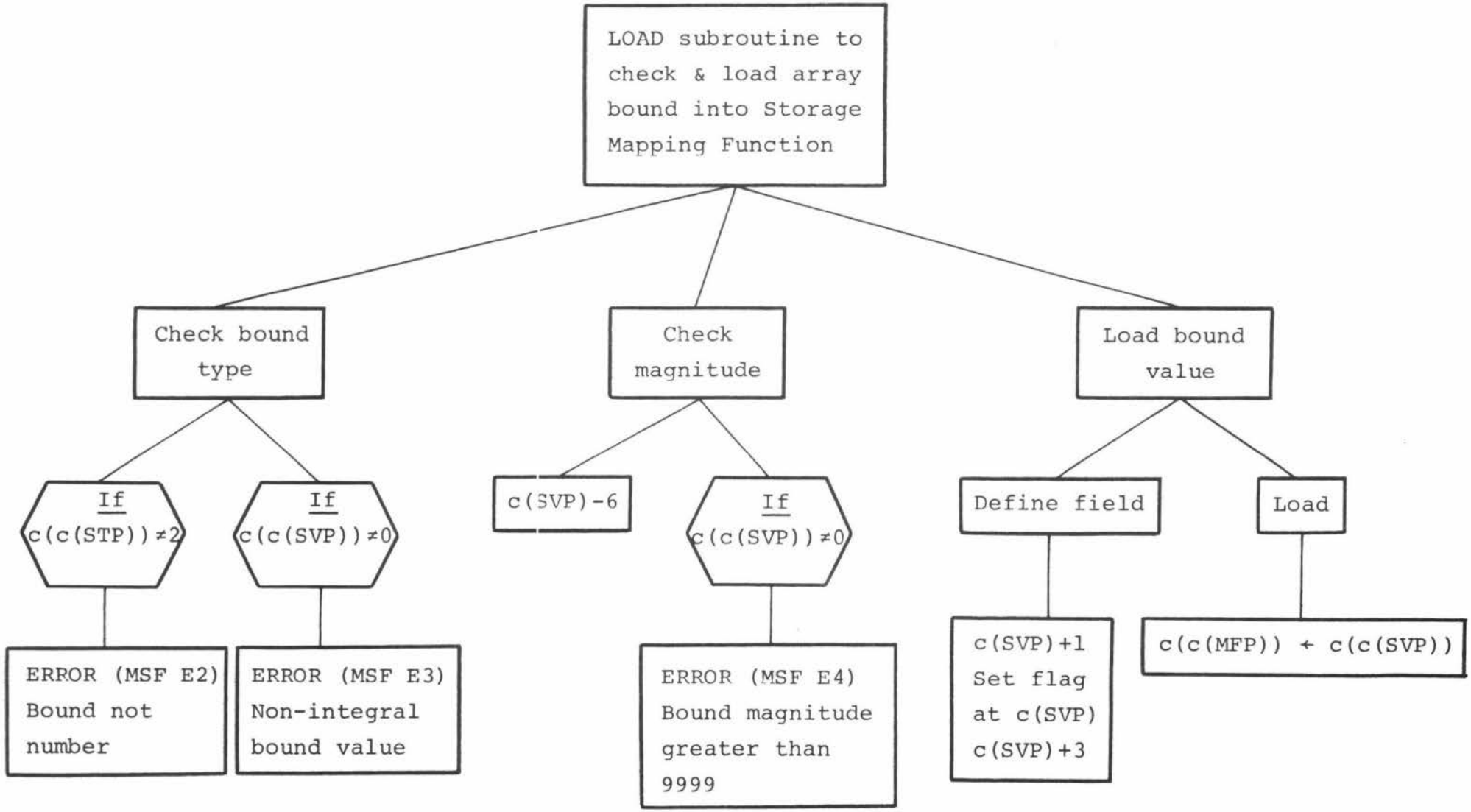
OP 33


MAKE STORAGE FUNCTION
ROUTINE
opcode parameters
M SF x̄ x x x
c(IC)
c(IC)+4
Place 'MSF' code and parameters in PBUF (for trace)
Set Mapping Function Pointer to address last upper bound field in Storage Mapping Function
Check & load (upper) array bound value into SMF (LOAD subroutine)
Save bound value
Decrement pointers (DEC subroutine)
Remainder of Routine
c(MFP) ← c(WVP)
Choice
on c(c(IC))
(array dim)
c(SIZE) ← c(c(MFP))
Clear flag at SIZE-3
c(MFP)-4
c(SVP)-8
c(STP)-10
0
1
c(MFP)-12
c(MFP)-4
Set flag at IND to indicate 2-dimensional array


REMAINDER OF ROUTINE
Check & load lower array bound value into SMF (LOAD subroutine)
Calculate & check bound range
2-dimensional array?
Fill in array words & reserve space for array elements
c(SIZE)-c(c(MFP))+1
Check array size
Set flag at SIZE-2
If c(SIZE)>999
ERROR (MSF E1) Array size greater than 999
If Flag at IND
Dec. pointers (DEC sub)
Check & load 1st upper bnd value into SMF (LOAD subroutine)
Save bnd value
Dec. pointers (DEC sub)
Check & load lower bnd value into SMF (LOAD subroutine)
Fill c1 in SMF
Calculate & check array size
c(ACC) ← c(c(MFP))
Clear flag at ACC-3
c(c(WVP)) ← c(SIZE)
c(SIZE) x (c(ACC)-c(c(MFP))+1)
If c(SIZE)>999
Set flag at SIZE-2
ERROR (MSF E1) Array size GT 999


FILL IN ARRAY WORDS AND RESERVE SPACE FOR ARRAY ELEMENTS
Set Stack pointers to address of last AW in array segment
Set count of no. of AW's in array segment
Set c(WVP) to (relative) address of first array element location
Set c(MFP) to address of SMF
Tidy up
c(IC)+1
c(COUNT) ← c(c(IC))
c(MFP) ← c(WVP)-26
Repeat
until c(COUNT)=0
c(WVP)+10-c(STKBAS)
Set flag at WVP-3
Fill in AW
Space for array elts.
Dec pointers and count
c(IC)-2
c(SVP) ← c(BVP)+ (c(c(IC))x10)
c(STP) ← c(SVP)-9
c(c(STP)) ← 4
c(c(SVP)) ← c(MFP)
c(SVP) - 4
c(c(SVP)) ← c(WVP)
c(SVP) - 4
Choice
Flag at IND?
c(WVP-1)+c(SIZE)
c(SVP)-2
c(STP)-10
c(COUNT)-1
c(c(SVP)) ← -1
c(c(SVP)) ← -0

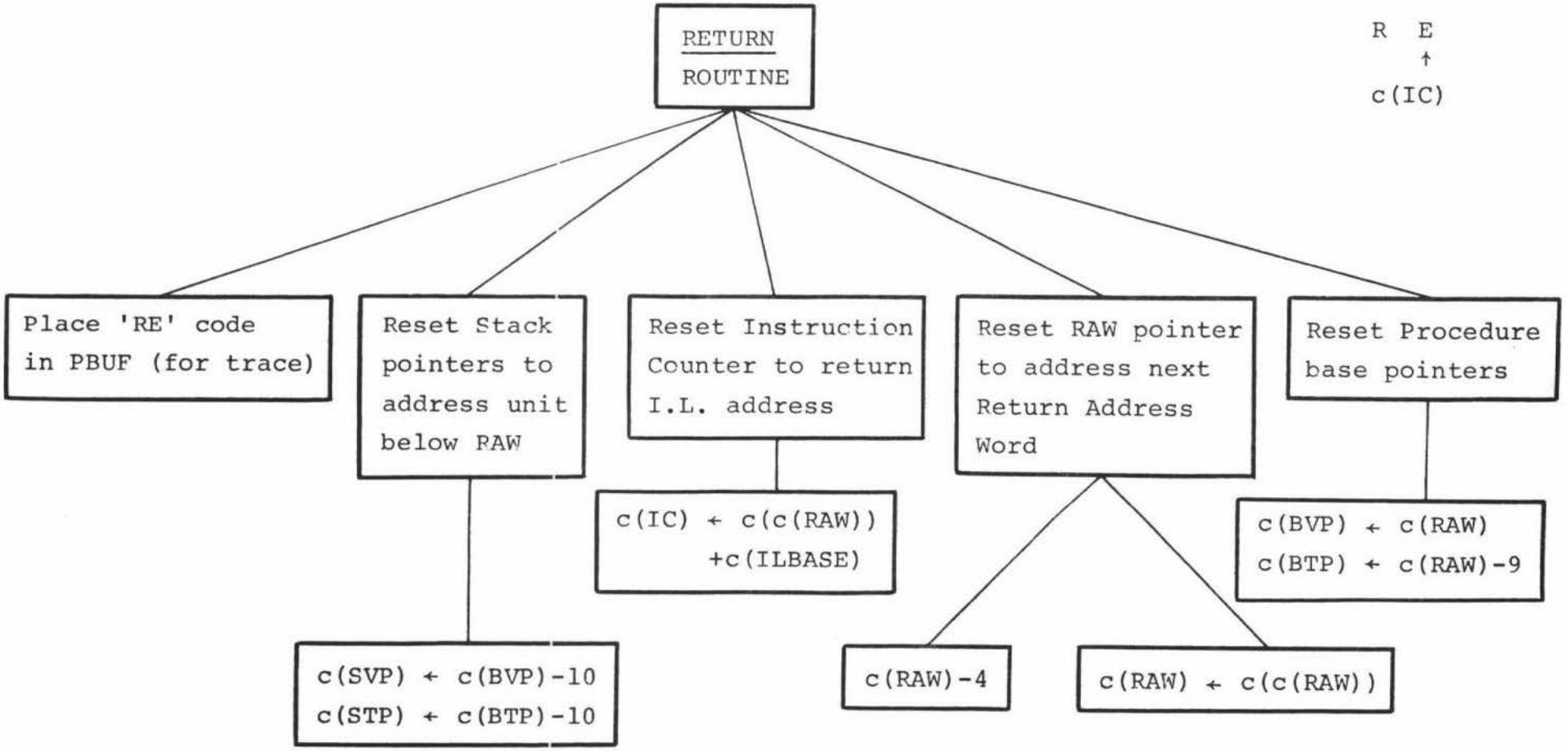

TIDY UP
Clear flag at IND
Reset c(WVP), leaving space for next possible SMF
Reset Stack Value Pointer and Instruction Counter
'Clear' array element area
c(SVP) ← c(WVP)
c(IC) + 1
Clear flag at WVP-3
c(WVP)+c(STKBAS)+10
c(STP) ← c(STKBAS)
c(STP) + c(MFP) + 7
Set array element types to 'undefined' (CLEAR routine)


LOAD subroutine to check & load array bound into Storage Mapping Function
Check bound type
Check magnitude
Load bound value
If c(c(STP))≠2
If c(c(SVP))≠0
c(SVP)-6
If c(c(SVP))≠0
Define field
Load
ERROR (MSF E2) Bound not number
ERROR (MSF E3) Non-integral bound value
ERROR (MSF E4) Bound magnitude greater than 9999
c(SVP)+1 Set flag at c(SVP) c(SVP)+3
c(c(MFP)) ← c(c(SVP))

OP 34

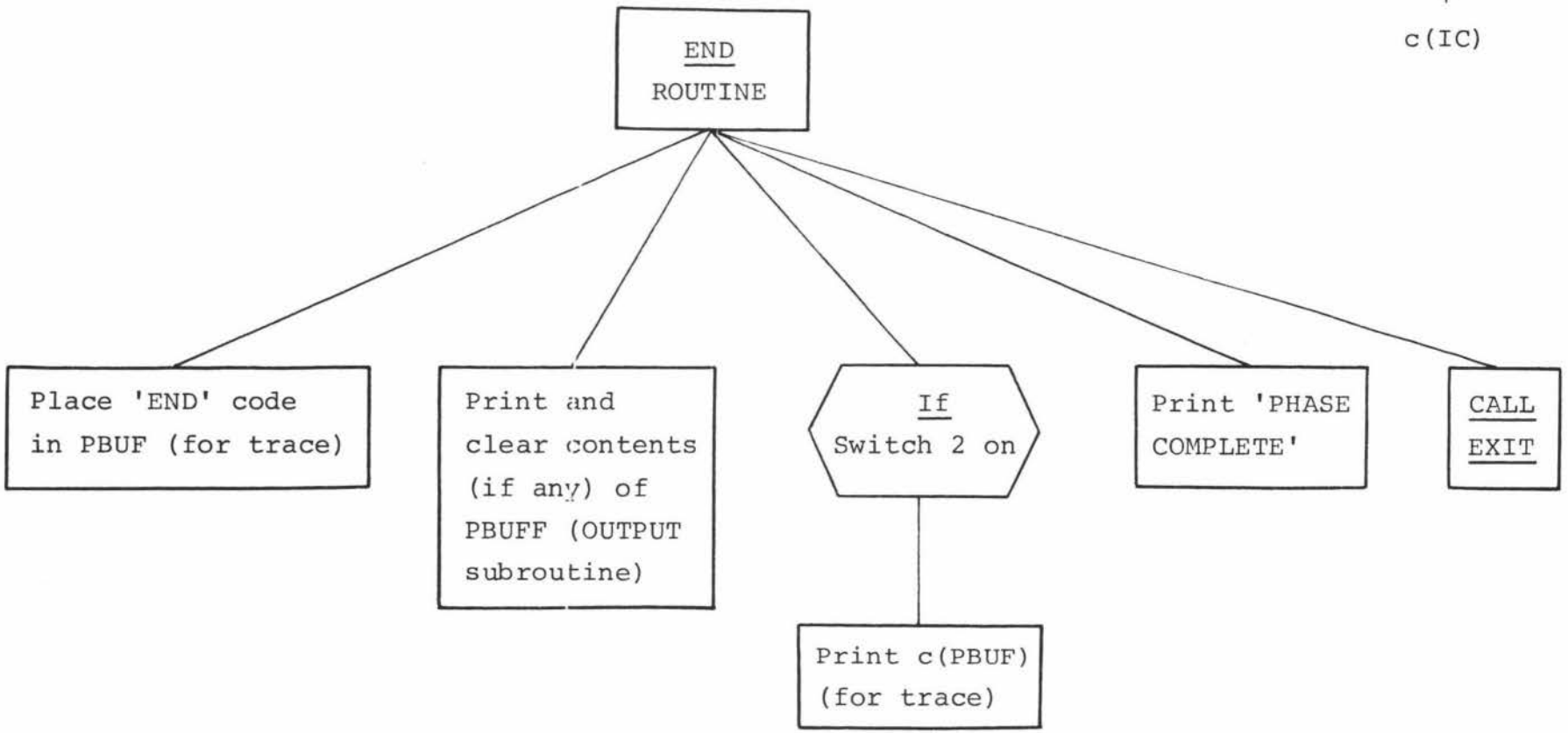
opcode

R E
↑
c(IC)



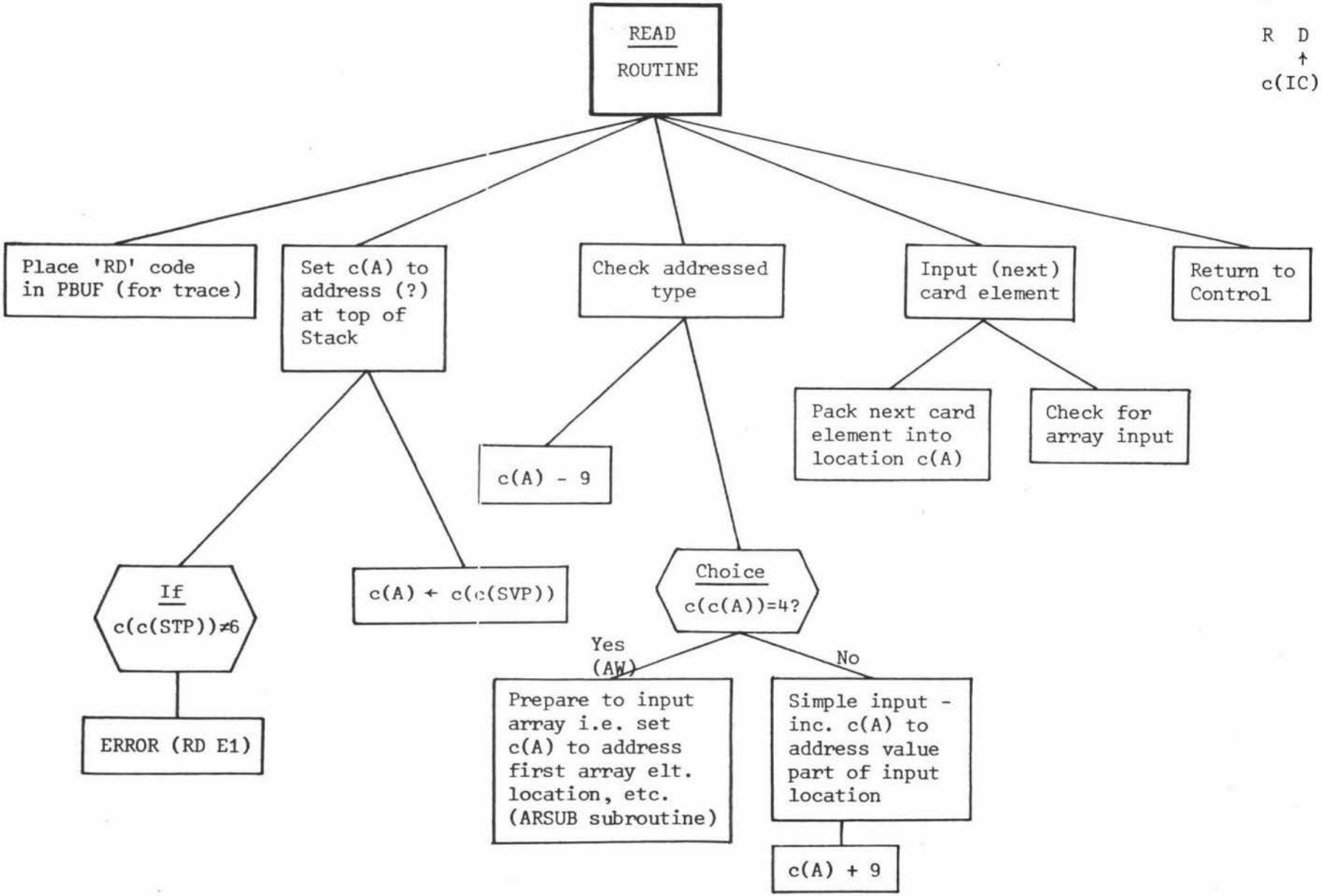
OP 35

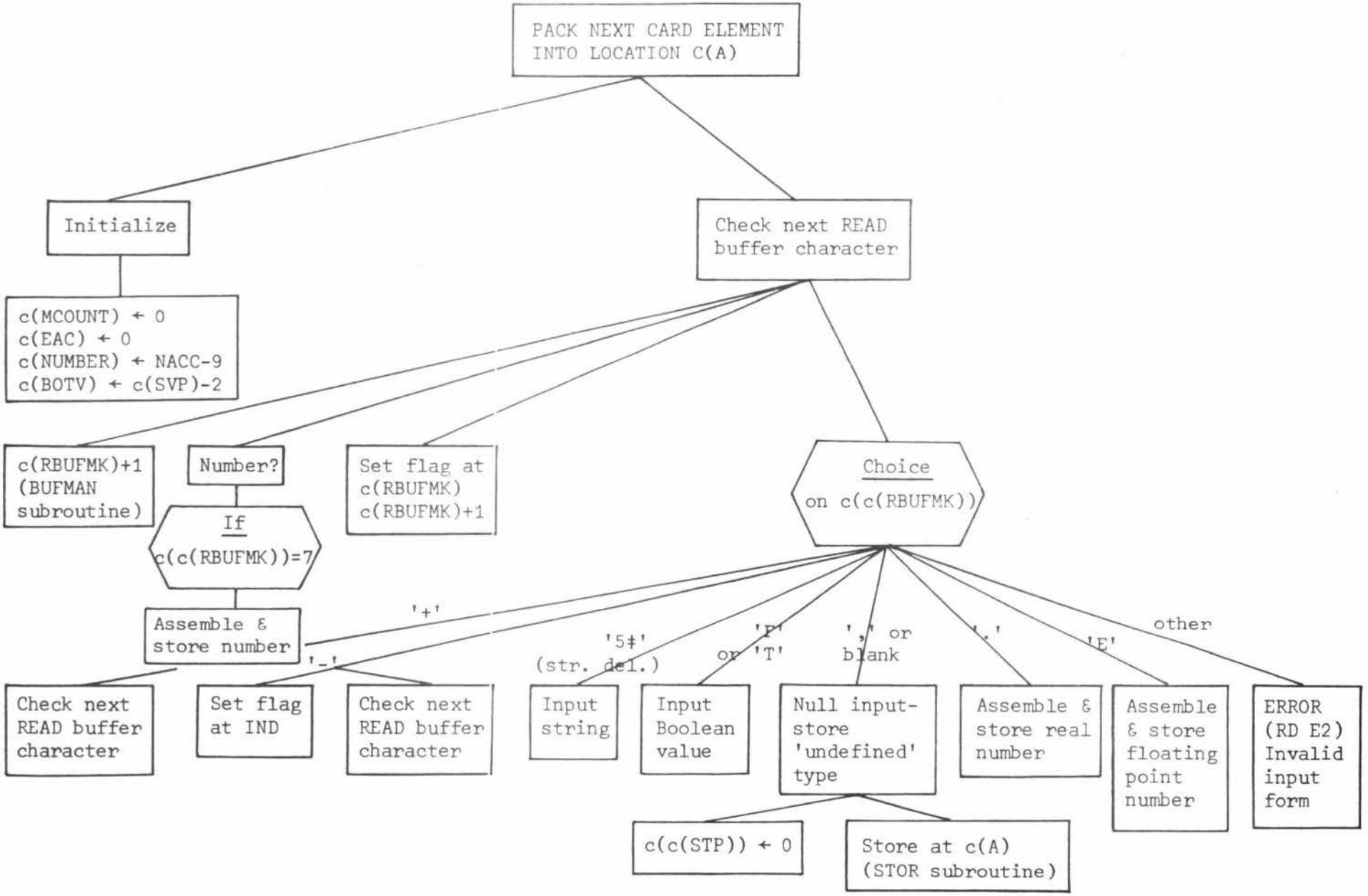
E ND
↑
c(IC)

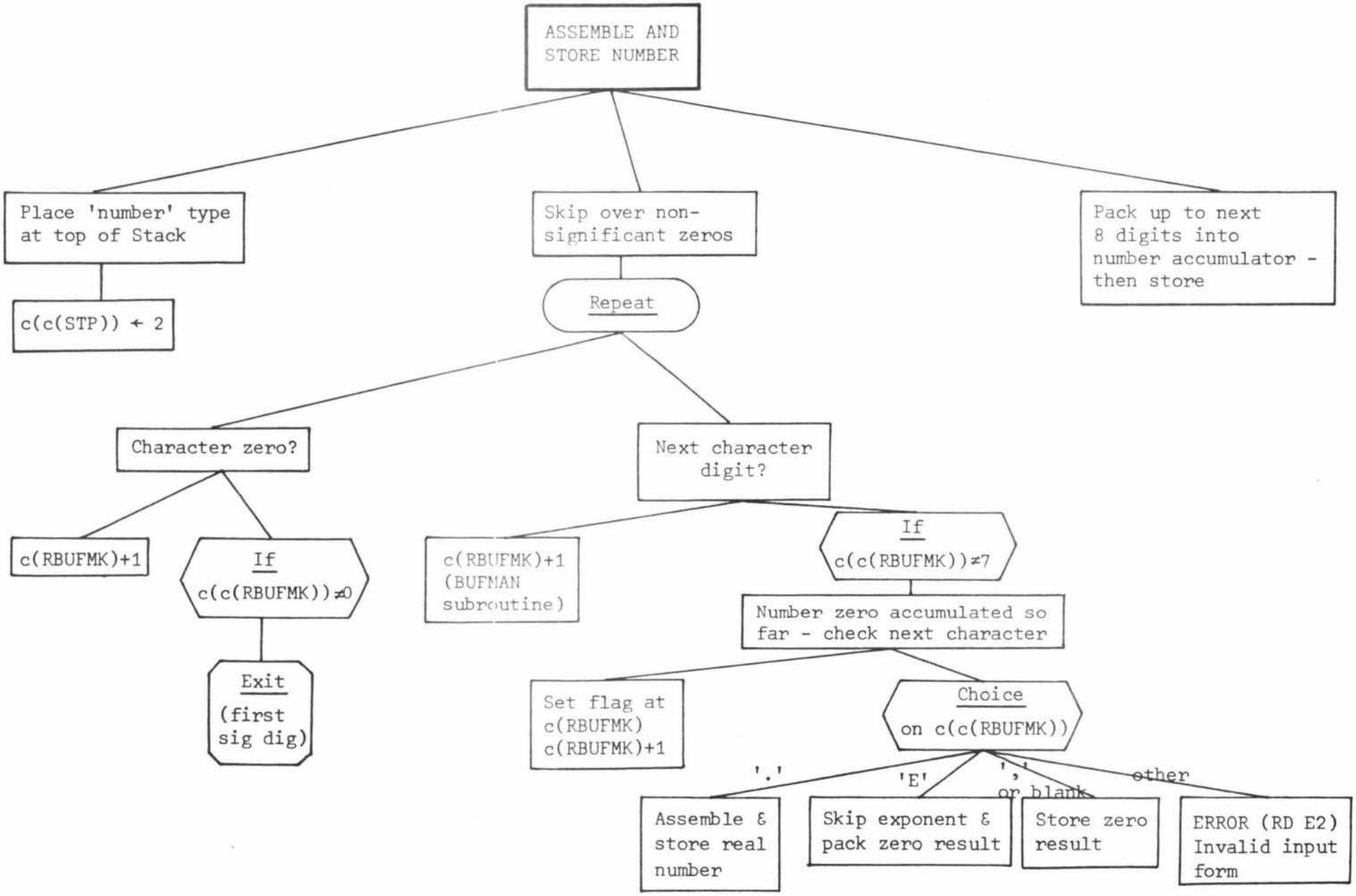

END
ROUTINE
Place 'END' code in PBUF (for trace)
Print and clear contents (if any) of PBUFF (OUTPUT subroutine)
If
Switch 2 on
Print c(PBUF) (for trace)
Print 'PHASE COMPLETE'
CALL
EXIT

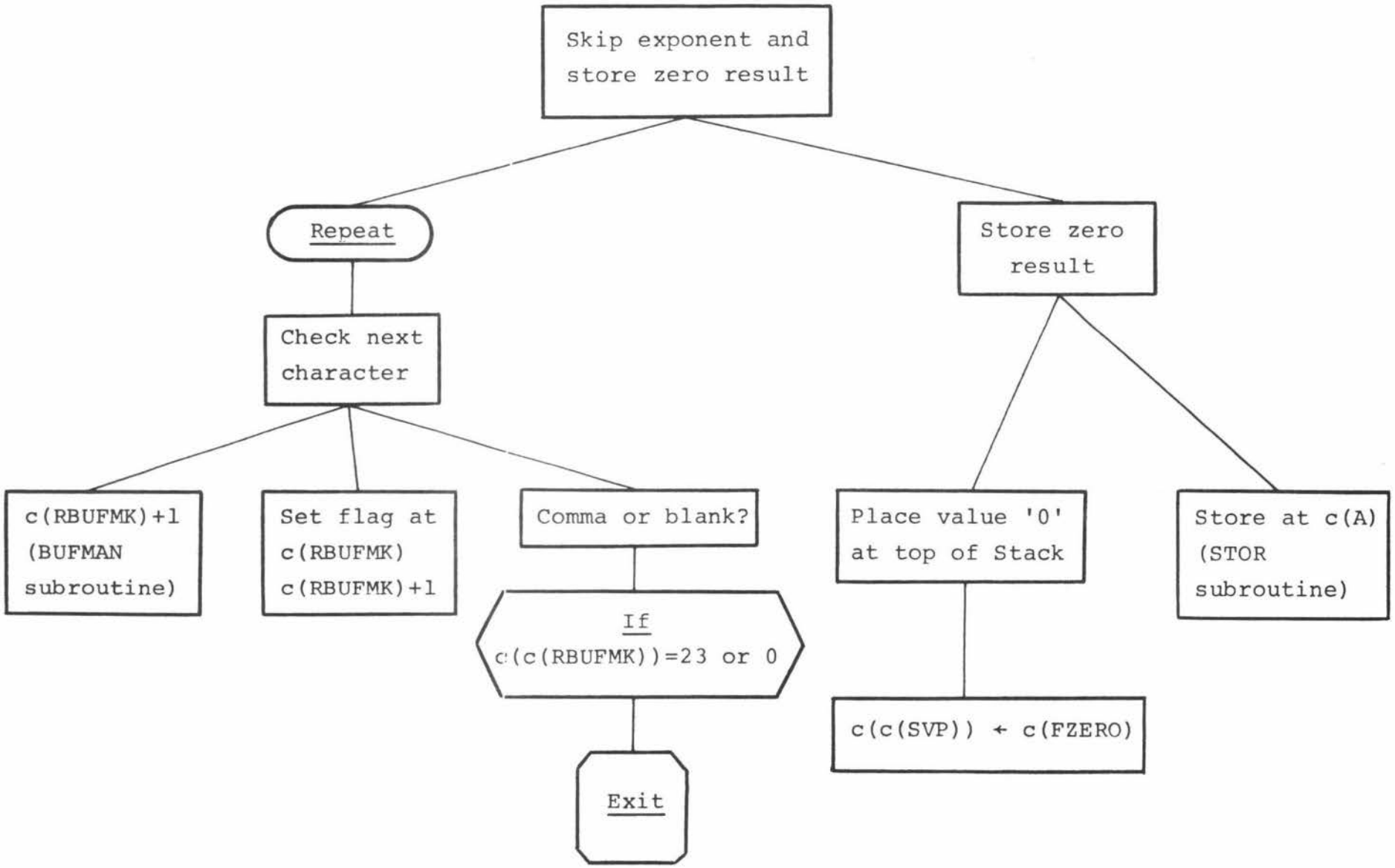
opcode

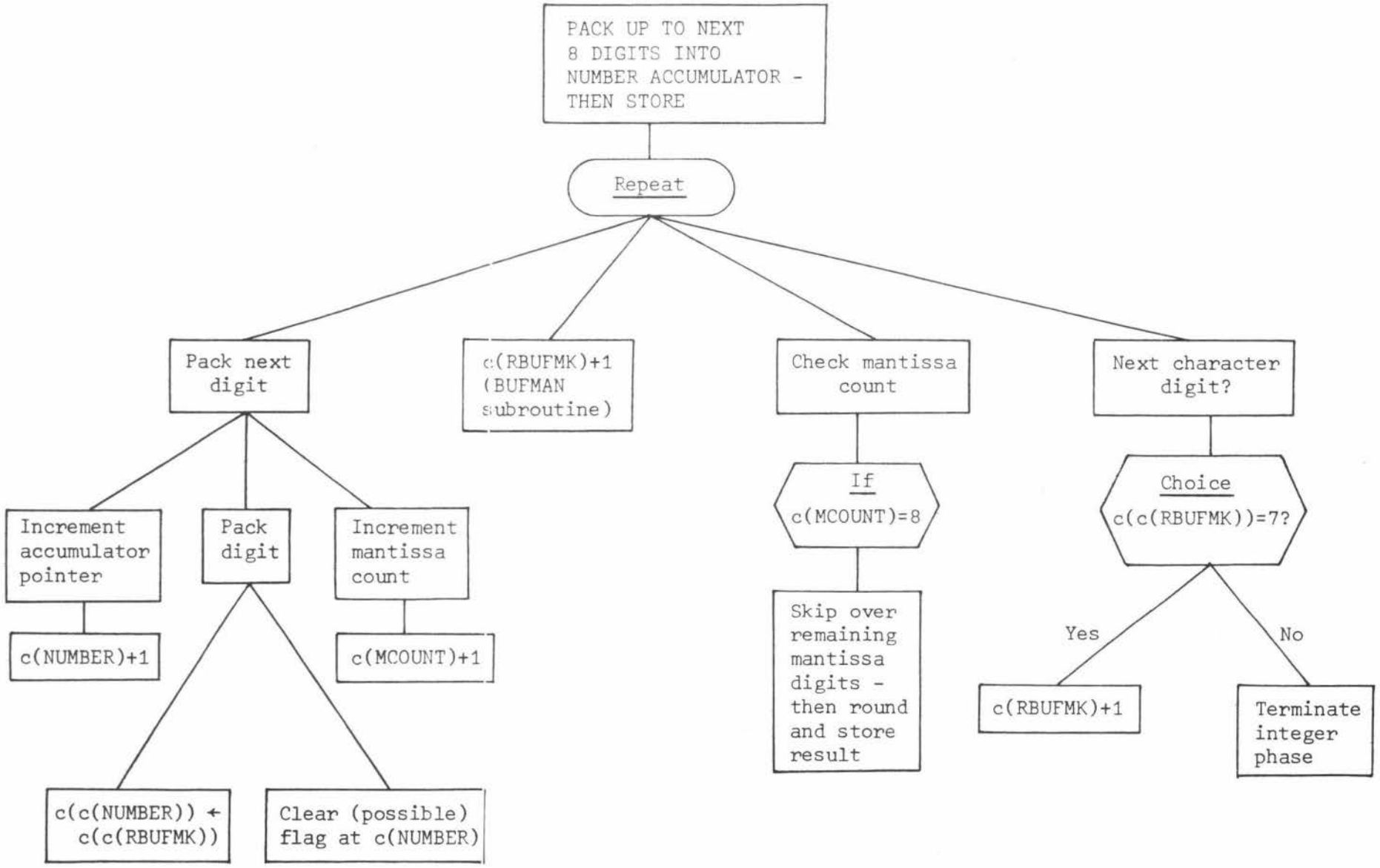
R D
↑
c(IC)

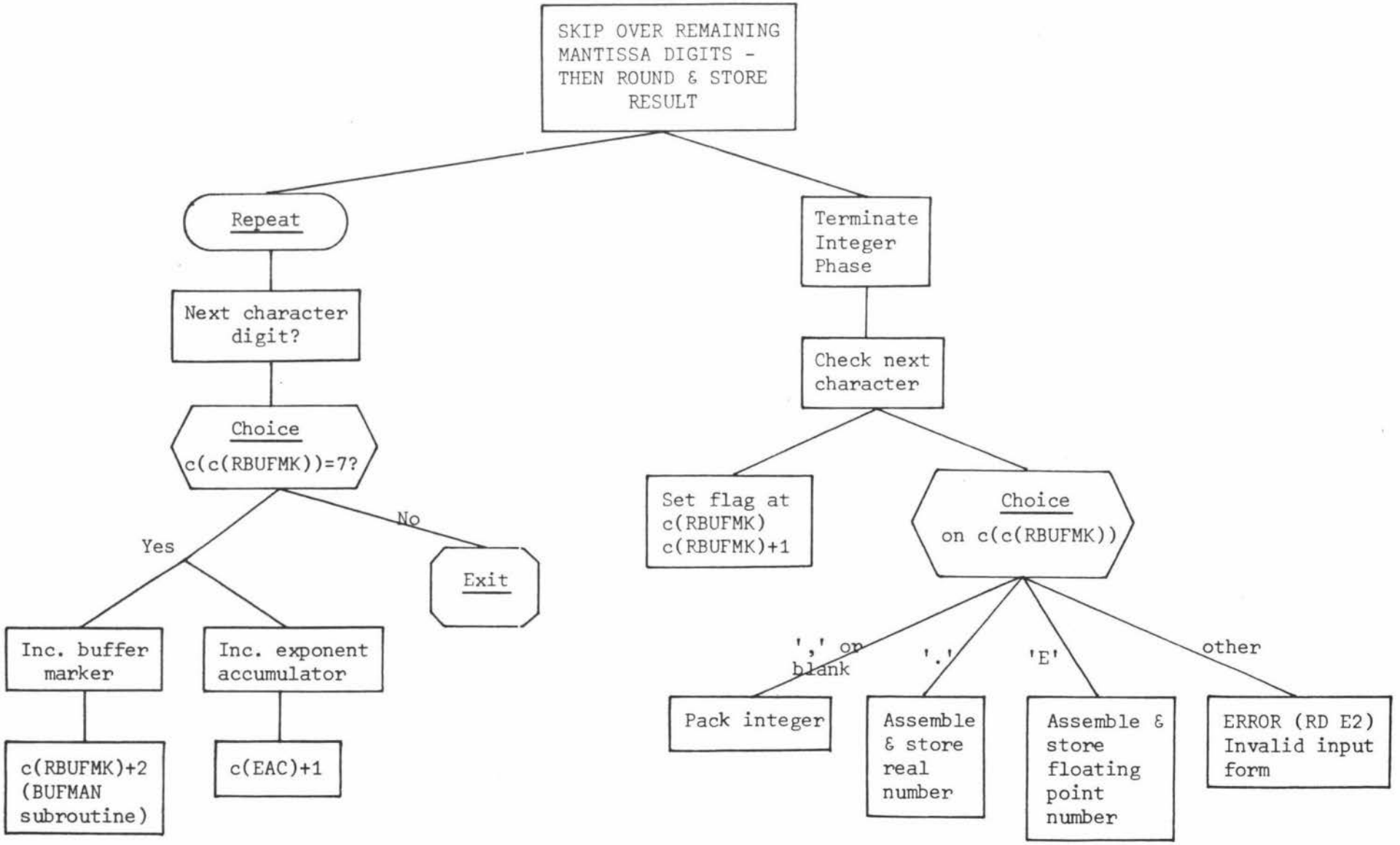


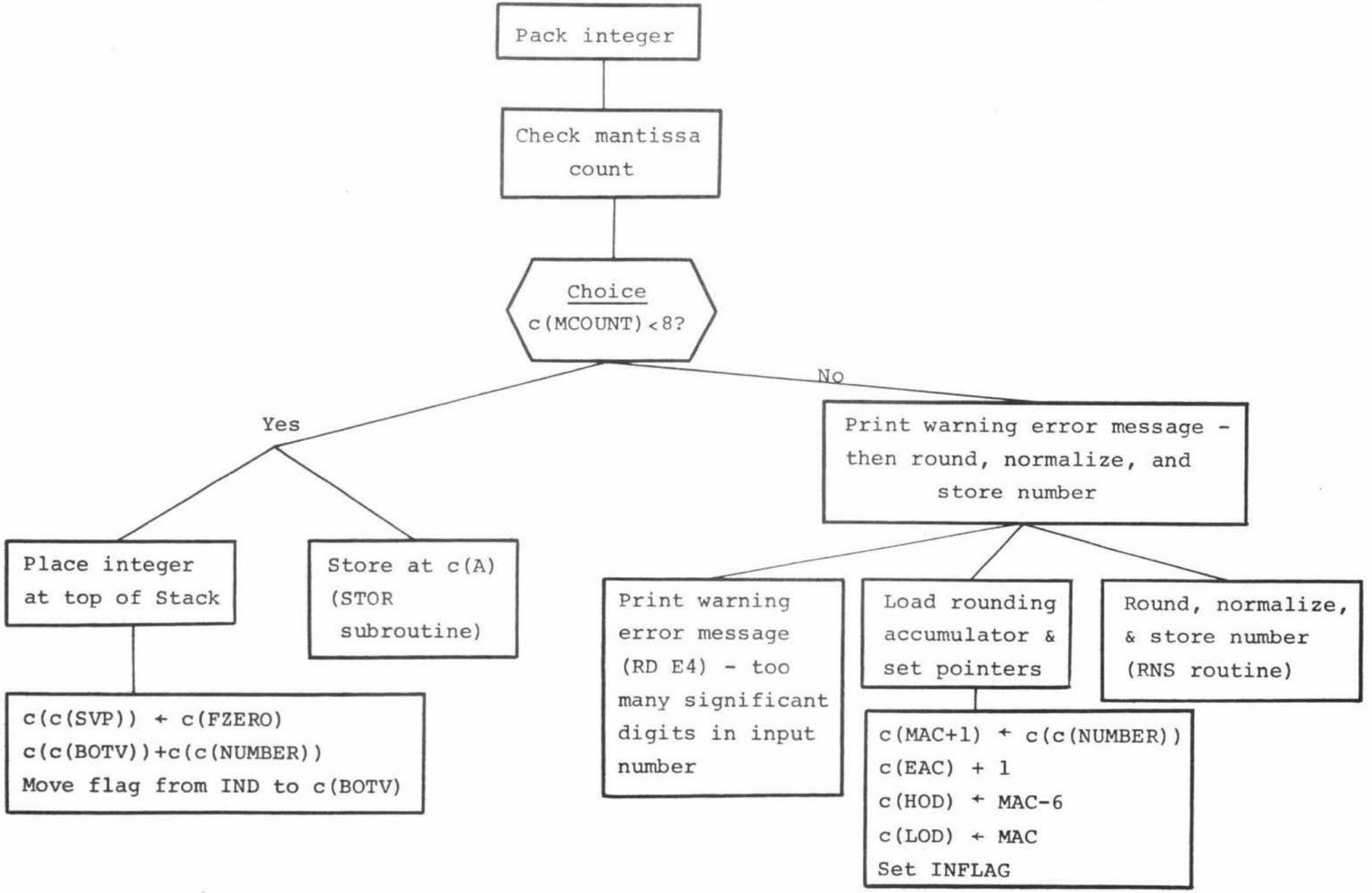

PACK NEXT CARD ELEMENT INTO LOCATION C(A)
Initialize
c(MCOUNT) ← 0
c(EAC) ← 0
c(NUMBER) ← NACC-9
c(BOTV) ← c(SVP)-2
Check next READ buffer character
c(RBUFMK)+1 (BUFMAN subroutine)
Number?
If c(c(RBUFMK))=7
Assemble & store number
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice on c(c(RBUFMK))
'+'
'-'
'5‡' (str. del.)
'F' or 'T'
',' or blank
'.'
'E'
other
Check next READ buffer character
Set flag at IND
Check next READ buffer character
Input string
Input Boolean value
Null input-store 'undefined' type
Assemble & store real number
Assemble & store floating point number
ERROR (RD E2) Invalid input form
c(c(STP)) ← 0
Store at c(A) (STOR subroutine)

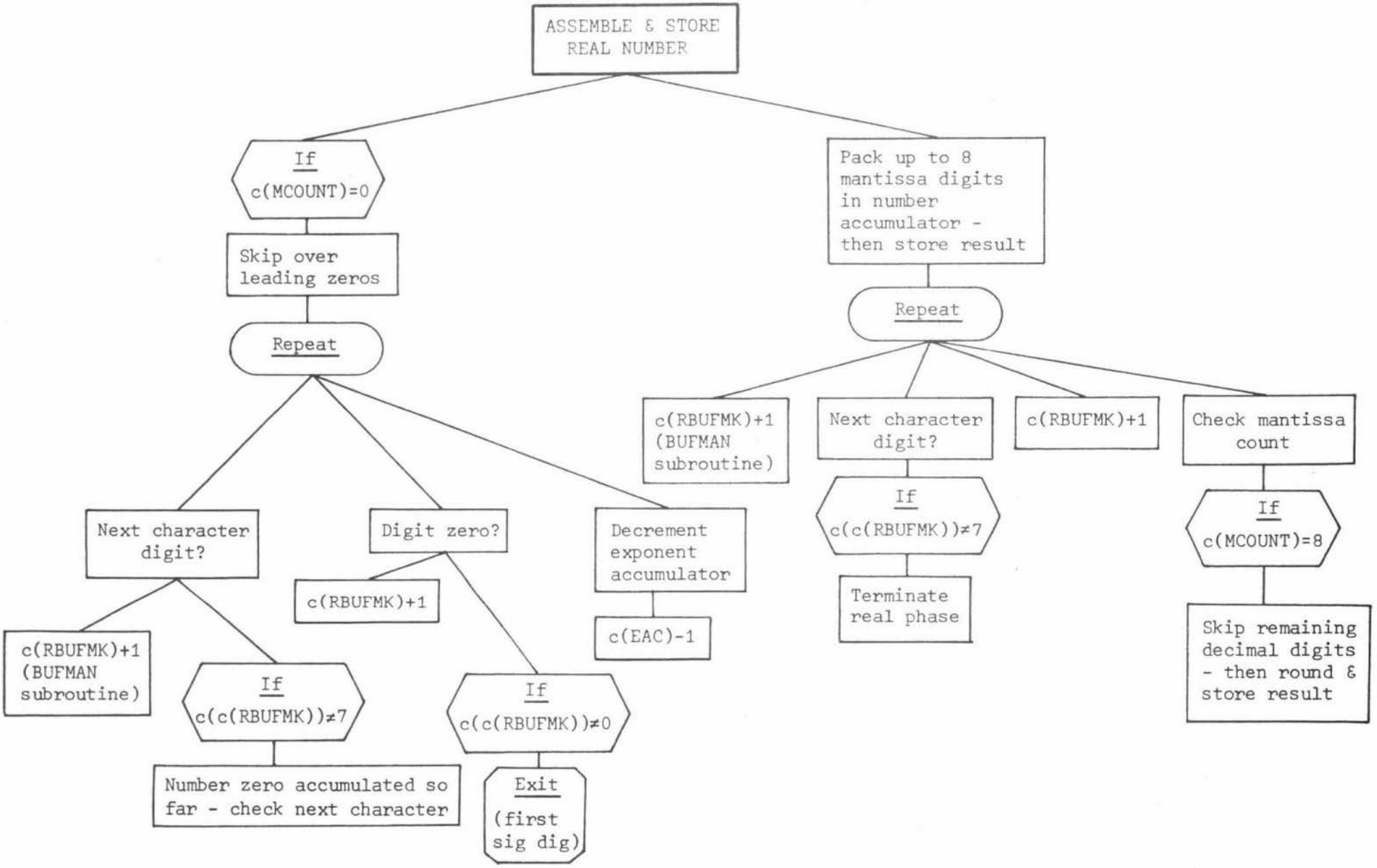

ASSEMBLE AND STORE NUMBER
Place 'number' type at top of Stack
c(c(STP)) ← 2
Skip over non-significant zeros
Repeat
Pack up to next 8 digits into number accumulator - then store
Character zero?
Next character digit?
c(RBUFMK)+1
If c(c(RBUFMK))≠0
Exit (first sig dig)
c(RBUFMK)+1 (BUFMAN subroutine)
If c(c(RBUFMK))≠7
Number zero accumulated so far - check next character
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice on c(c(RBUFMK))
'.'
'E'
',' or blank
other
Assemble & store real number
Skip exponent & pack zero result
Store zero result
ERROR (RD E2) Invalid input form

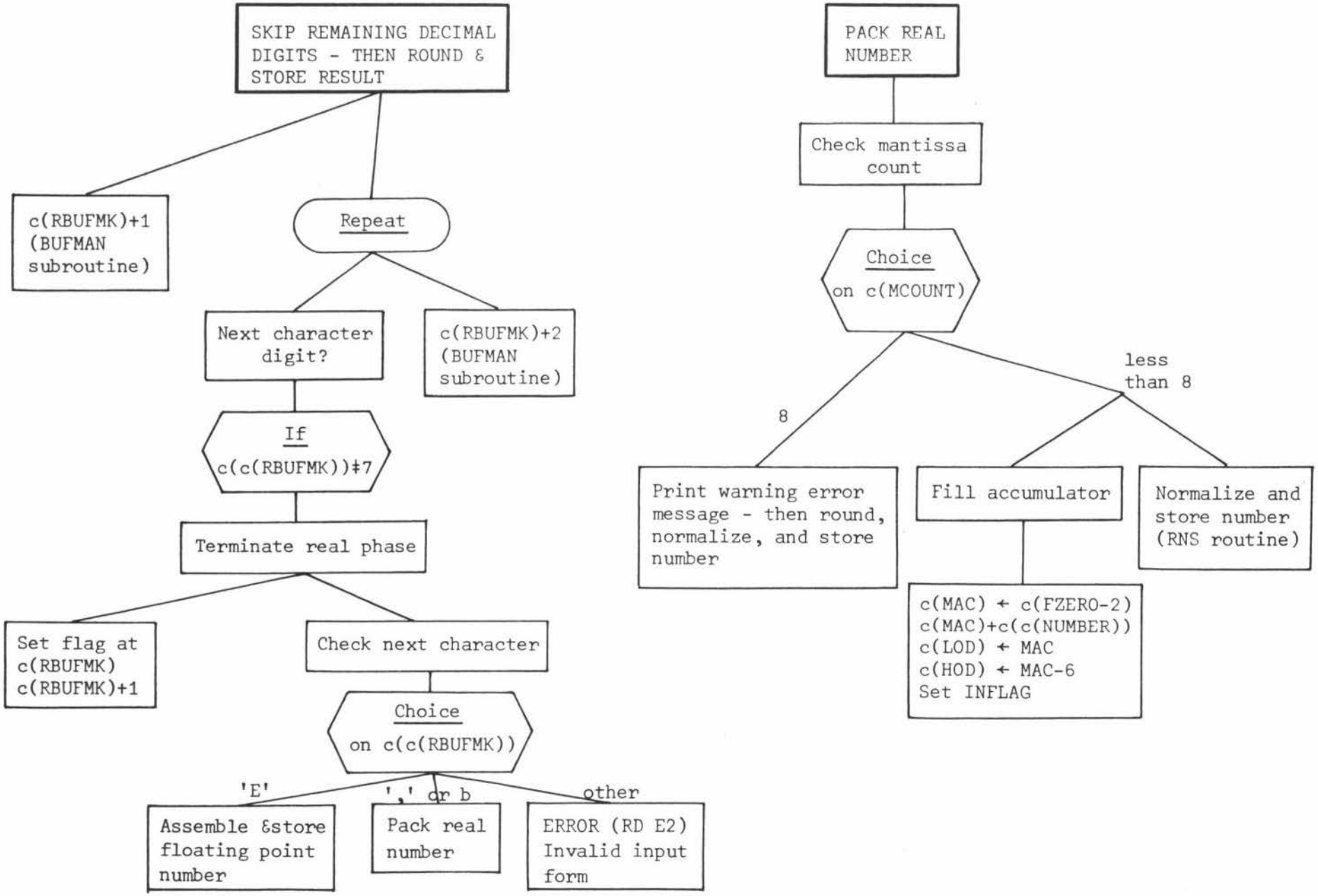

Skip exponent and store zero result
Repeat
Store zero result
Check next character
c(RBUFMK)+1 (BUFMAN subroutine)
Set flag at c(RBUFMK) c(RBUFMK)+1
Comma or blank?
Place value '0' at top of Stack
Store at c(A) (STOR subroutine)
If c(c(RBUFMK))=23 or 0
c(c(SVP)) ← c(FZERO)
Exit

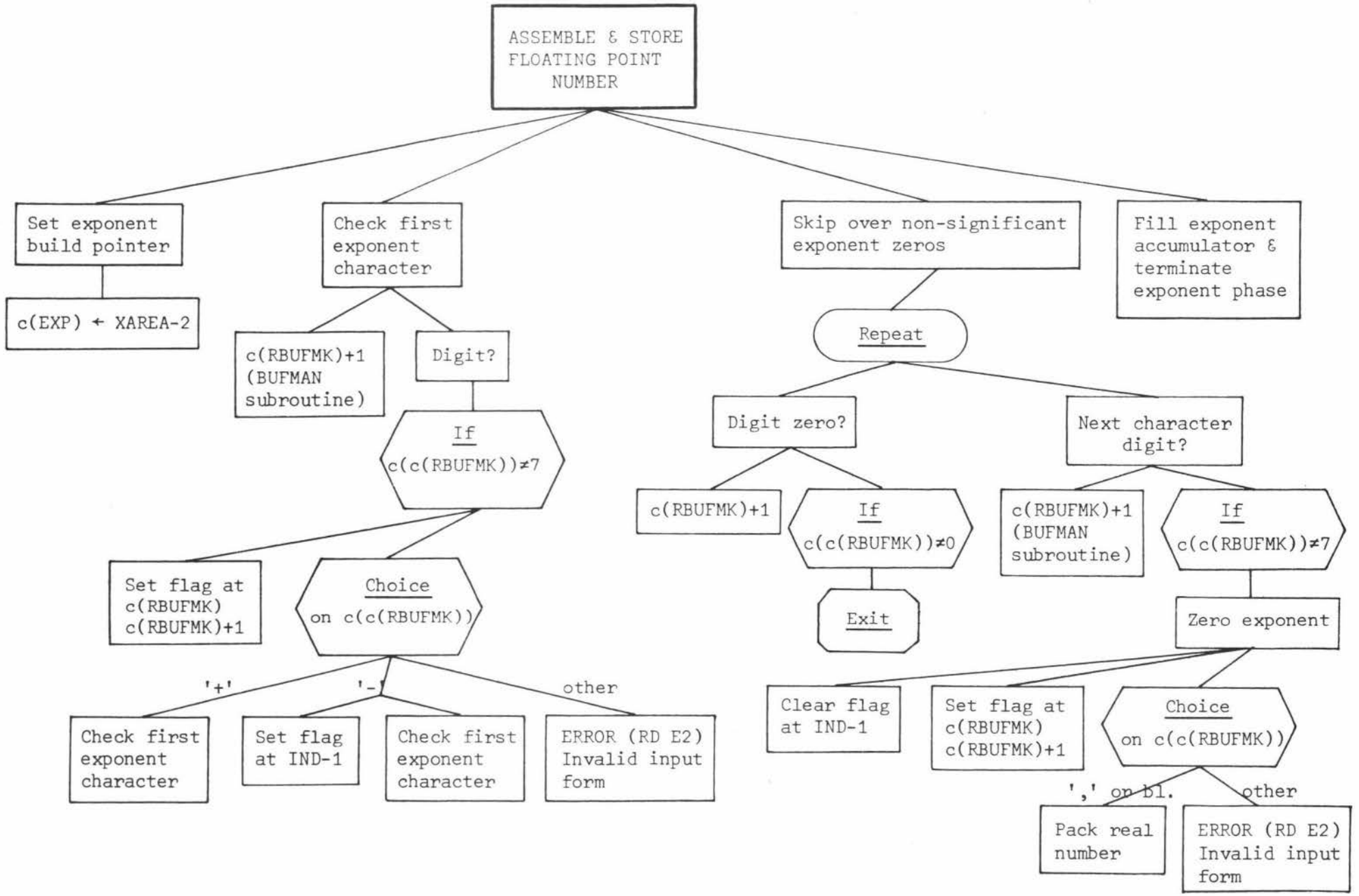

PACK UP TO NEXT 8 DIGITS INTO NUMBER ACCUMULATOR - THEN STORE
Repeat
Pack next digit
c(RBUFMK)+1 (BUFMAN subroutine)
Check mantissa count
Next character digit?
Increment accumulator pointer
Pack digit
Increment mantissa count
c(NUMBER)+1
c(MCOUNT)+1
c(c(NUMBER)) ← c(c(RBUFMK))
Clear (possible) flag at c(NUMBER)
If c(MCOUNT)=8
Skip over remaining mantissa digits - then round and store result
Choice c(c(RBUFMK))=7?
Yes
No
c(RBUFMK)+1
Terminate integer phase

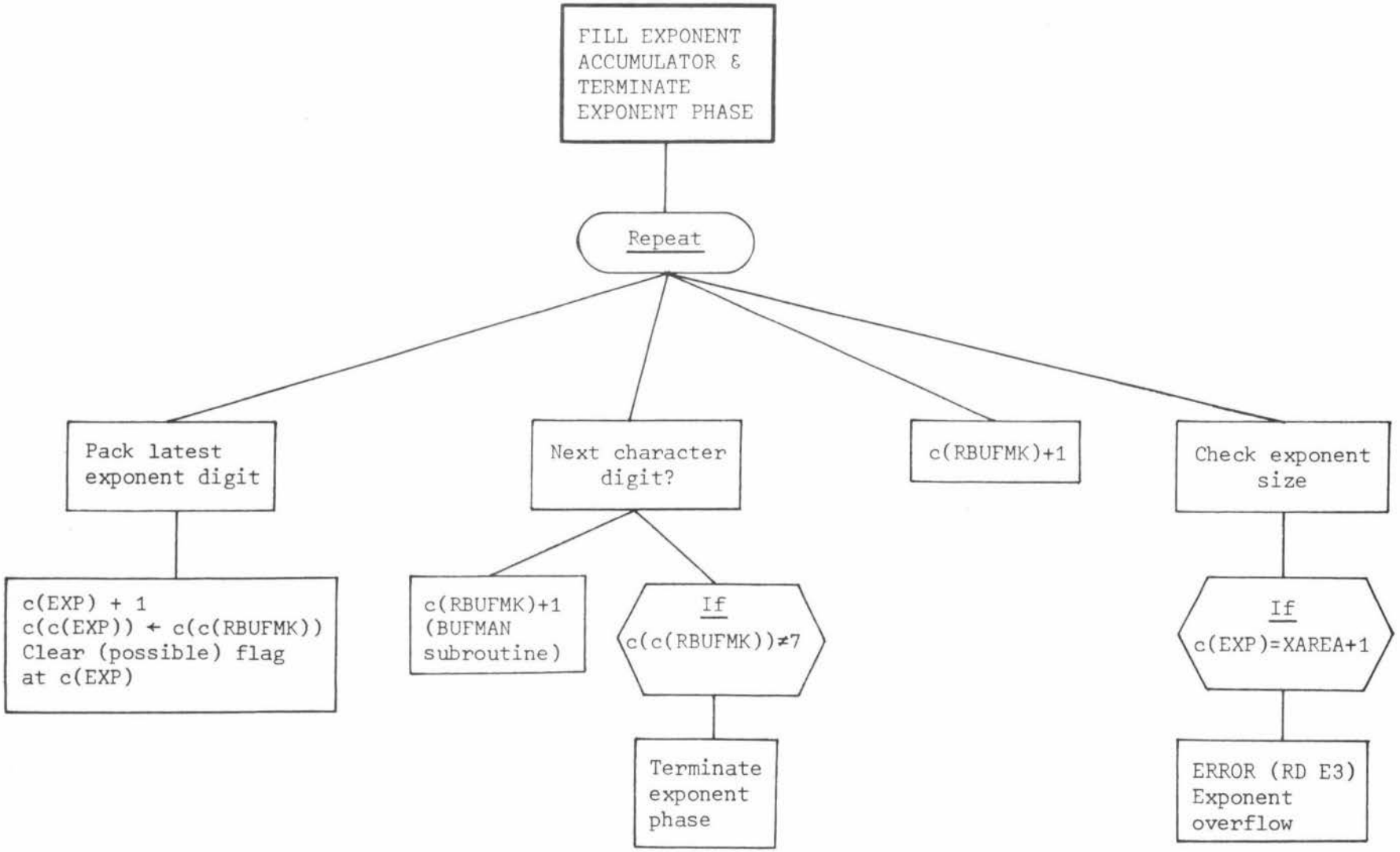

SKIP OVER REMAINING MANTISSA DIGITS - THEN ROUND & STORE RESULT
Repeat
Next character digit?
Choice
c(c(RBUFMK))=7?
Yes
No
Exit
Inc. buffer marker
Inc. exponent accumulator
c(RBUFMK)+2 (BUFMAN subroutine)
c(EAC)+1
Terminate Integer Phase
Check next character
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice
on c(c(RBUFMK))
',' or blank
'.'
'E'
other
Pack integer
Assemble & store real number
Assemble & store floating point number
ERROR (RD E2) Invalid input form

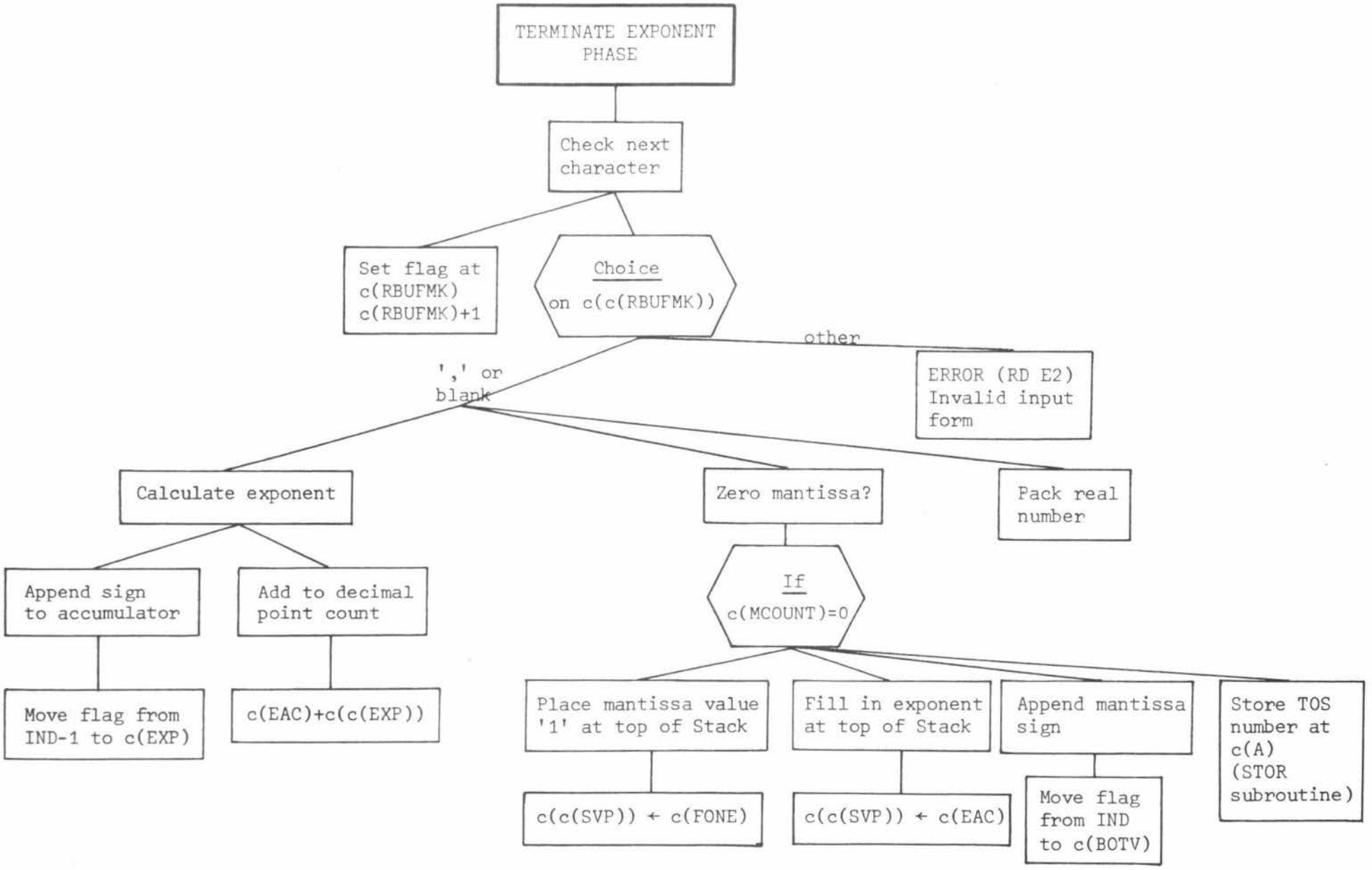

Pack integer
Check mantissa count
Choice
c(MCOUNT)<8?
Yes
No
Place integer at top of Stack
Store at c(A) (STOR subroutine)
c(c(SVP)) ← c(FZERO)
c(c(BOTV))+c(c(NUMBER))
Move flag from IND to c(BOTV)
Print warning error message - then round, normalize, and store number
Print warning error message (RD E4) - too many significant digits in input number
Load rounding accumulator & set pointers
Round, normalize, & store number (RNS routine)
c(MAC+1) ← c(c(NUMBER))
c(EAC) + 1
c(HOD) ← MAC-6
c(LOD) ← MAC
Set INFLAG

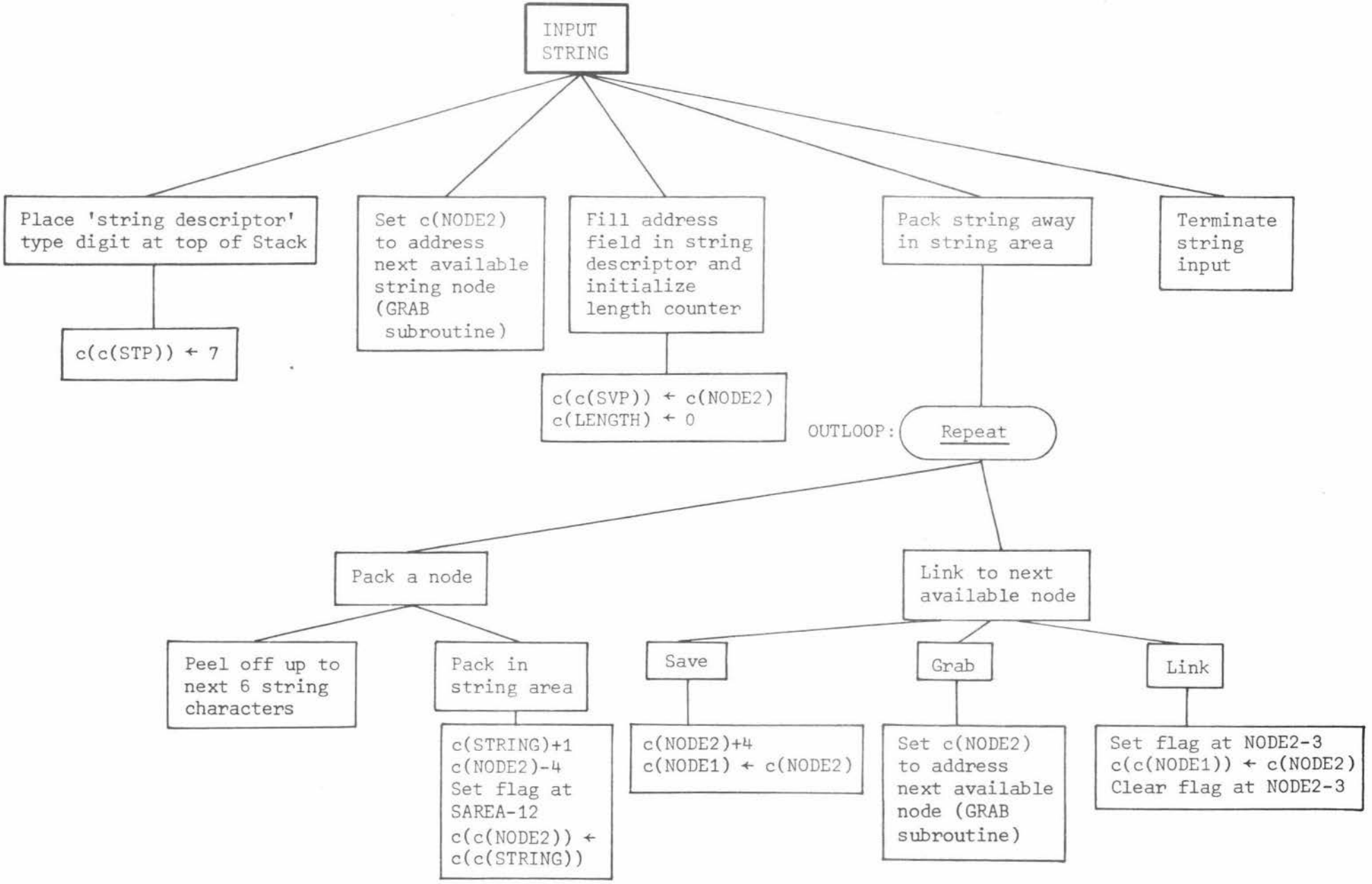

ASSEMBLE & STORE REAL NUMBER
If c(MCOUNT)=0
Skip over leading zeros
Repeat
Next character digit?
c(RBUFMK)+1 (BUFMAN subroutine)
If c(c(RBUFMK))≠7
Number zero accumulated so far - check next character
Digit zero?
c(RBUFMK)+1
If c(c(RBUFMK))≠0
Exit (first sig dig)
Decrement exponent accumulator
c(EAC)-1
Pack up to 8 mantissa digits in number accumulator - then store result
Repeat
c(RBUFMK)+1 (BUFMAN subroutine)
Next character digit?
If c(c(RBUFMK))≠7
Terminate real phase
c(RBUFMK)+1
Check mantissa count
If c(MCOUNT)=8
Skip remaining decimal digits - then round & store result

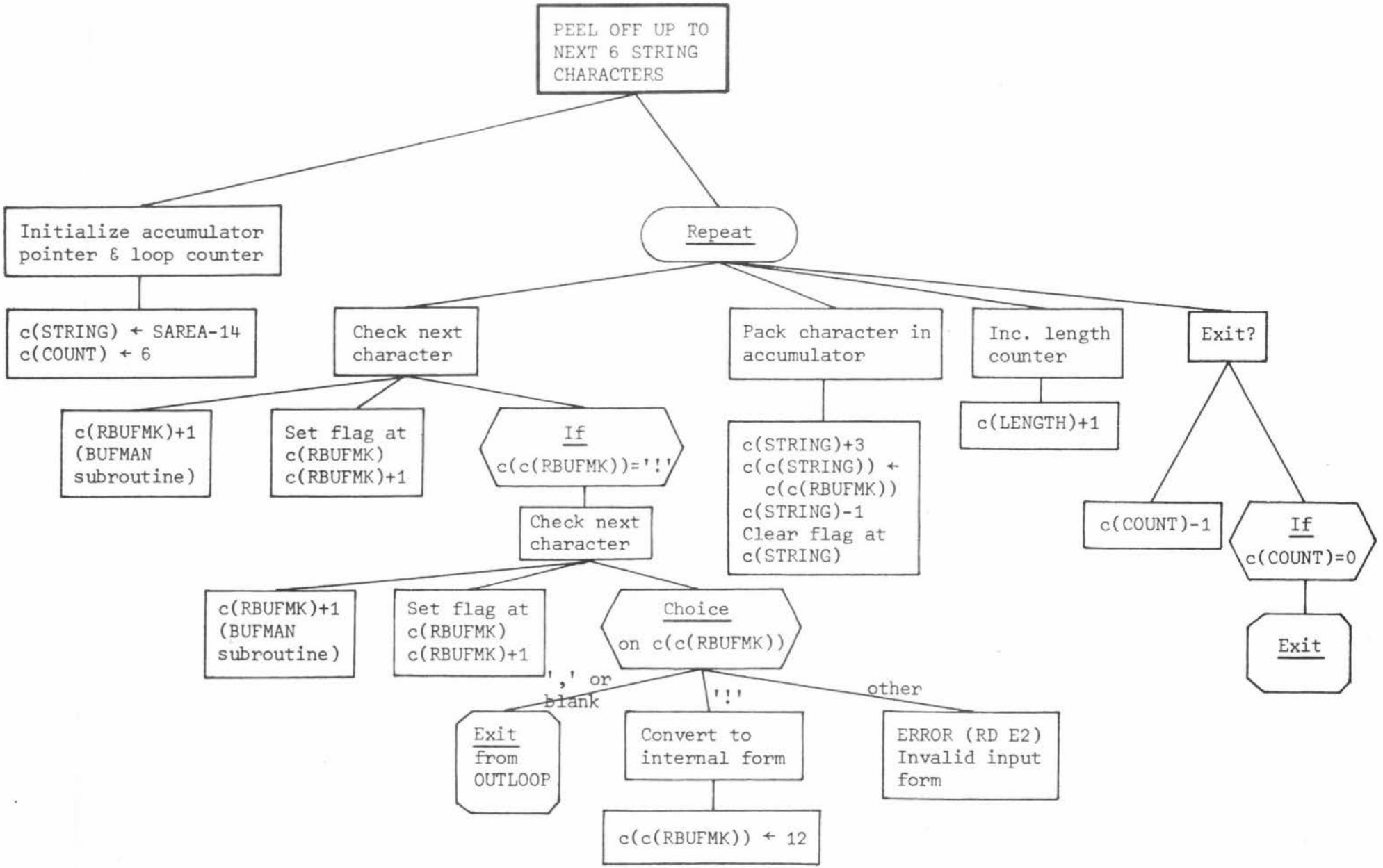

SKIP REMAINING DECIMAL DIGITS - THEN ROUND & STORE RESULT
c(RBUFMK)+1 (BUFMAN subroutine)
Repeat
Next character digit?
c(RBUFMK)+2 (BUFMAN subroutine)
If c(c(RBUFMK))≠7
Terminate real phase
Set flag at c(RBUFMK) c(RBUFMK)+1
Check next character
Choice on c(c(RBUFMK))
'E'
',' or b
other
Assemble &store floating point number
Pack real number
ERROR (RD E2) Invalid input form
PACK REAL NUMBER
Check mantissa count
Choice on c(MCOUNT)
8
less than 8
Print warning error message - then round, normalize, and store number
Fill accumulator
Normalize and store number (RNS routine)
c(MAC) ← c(FZERO-2)
c(MAC)+c(c(NUMBER))
c(LOD) ← MAC
c(HOD) ← MAC-6
Set INFLAG

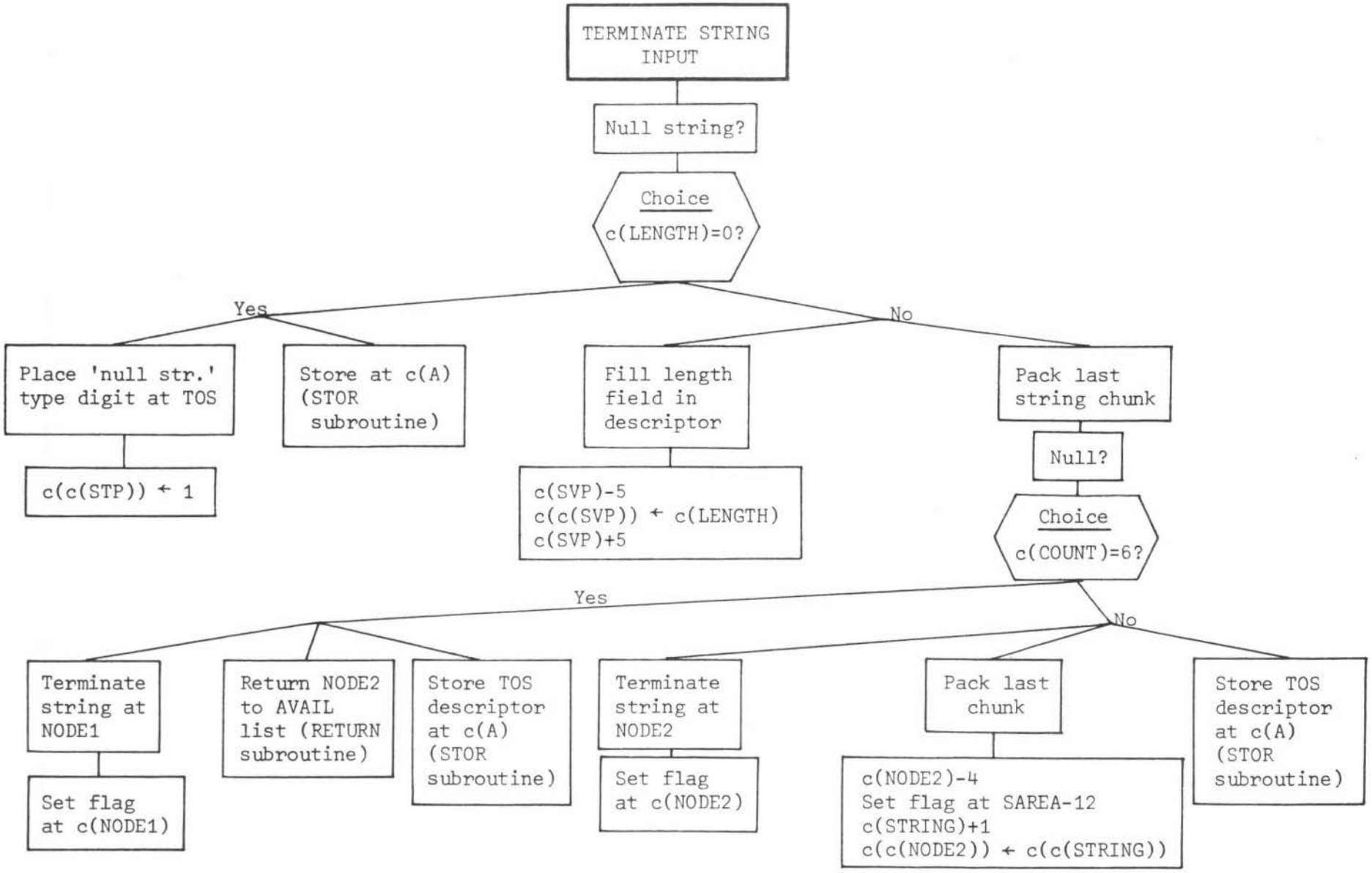

ASSEMBLE & STORE FLOATING POINT NUMBER
Set exponent build pointer
c(EXP) ← XAREA-2
Check first exponent character
c(RBUFMK)+1 (BUFMAN subroutine)
Digit?
If c(c(RBUFMK))≠7
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice on c(c(RBUFMK))
'+'
'-'
other
Check first exponent character
Set flag at IND-1
Check first exponent character
ERROR (RD E2) Invalid input form
Skip over non-significant exponent zeros
Repeat
Digit zero?
c(RBUFMK)+1
If c(c(RBUFMK))≠0
Exit
Next character digit?
c(RBUFMK)+1 (BUFMAN subroutine)
If c(c(RBUFMK))≠7
Zero exponent
Clear flag at IND-1
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice on c(c(RBUFMK))
',' or bl.
other
Pack real number
ERROR (RD E2) Invalid input form
Fill exponent accumulator & terminate exponent phase

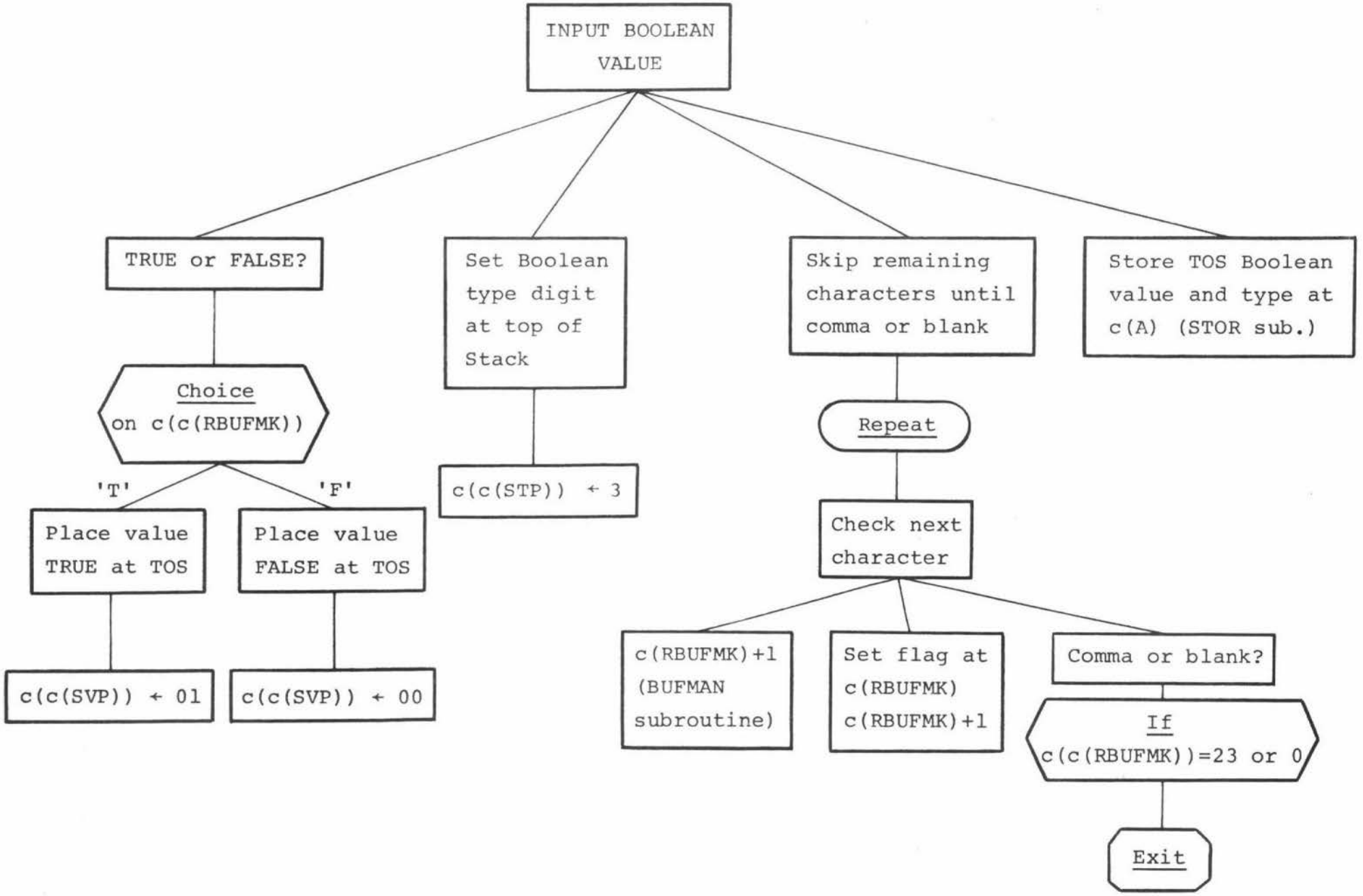

FILL EXPONENT ACCUMULATOR & TERMINATE EXPONENT PHASE
Repeat
Pack latest exponent digit
c(EXP) + 1
c(c(EXP)) ← c(c(RBUFMK))
Clear (possible) flag at c(EXP)
Next character digit?
c(RBUFMK)+1 (BUFMAN subroutine)
If c(c(RBUFMK))≠7
Terminate exponent phase
c(RBUFMK)+1
Check exponent size
If c(EXP)=XAREA+1
ERROR (RD E3) Exponent overflow

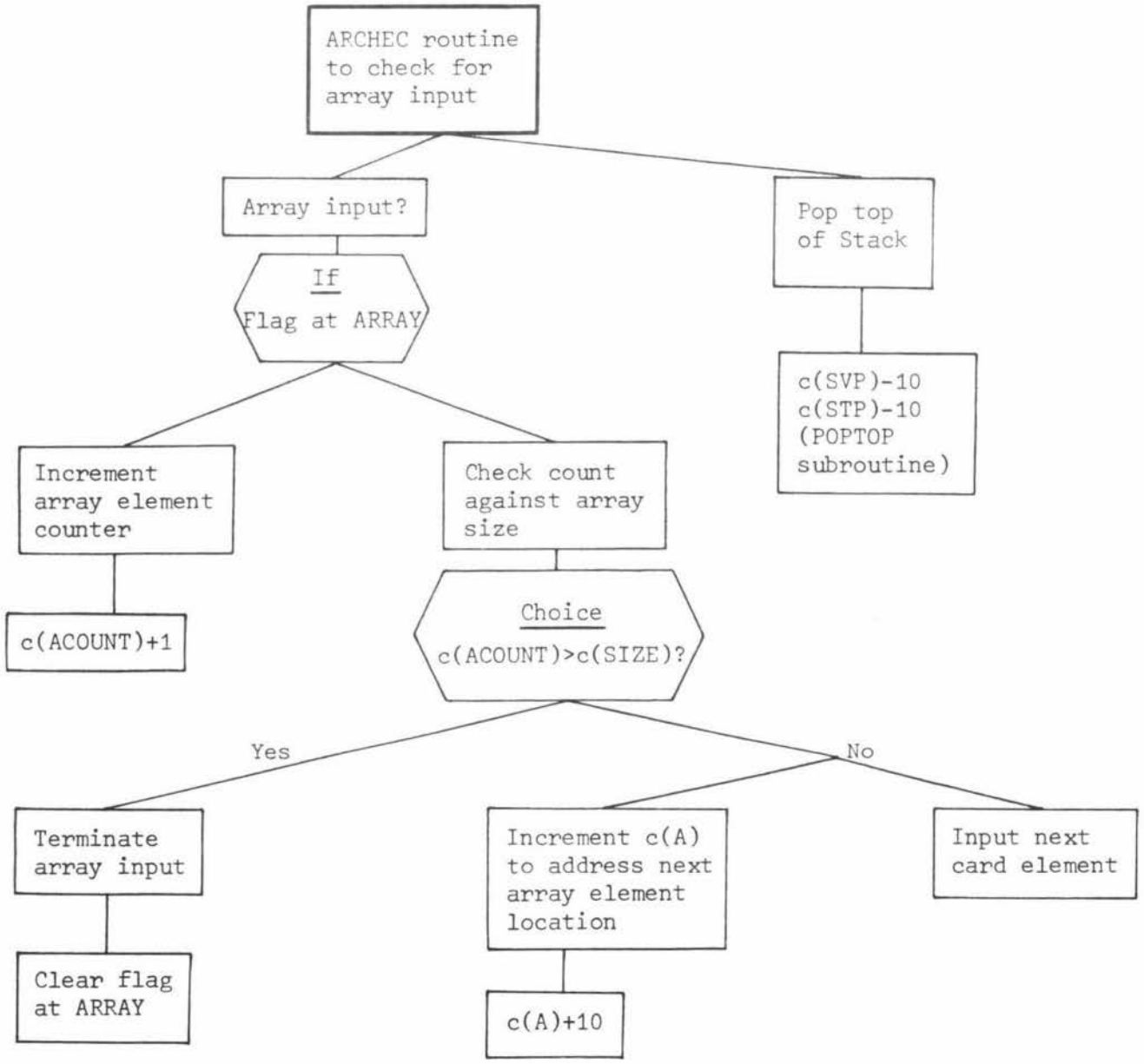

TERMINATE EXPONENT PHASE
Check next character
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice on c(c(RBUFMK))
other
',' or blank
ERROR (RD E2) Invalid input form
Calculate exponent
Zero mantissa?
Pack real number
Append sign to accumulator
Add to decimal point count
If c(MCOUNT)=0
Move flag from IND-1 to c(EXP)
c(EAC)+c(c(EXP))
Place mantissa value '1' at top of Stack
Fill in exponent at top of Stack
Append mantissa sign
Store TOS number at c(A) (STOR subroutine)
c(c(SVP)) ← c(FONE)
c(c(SVP)) ← c(EAC)
Move flag from IND to c(BOTV)

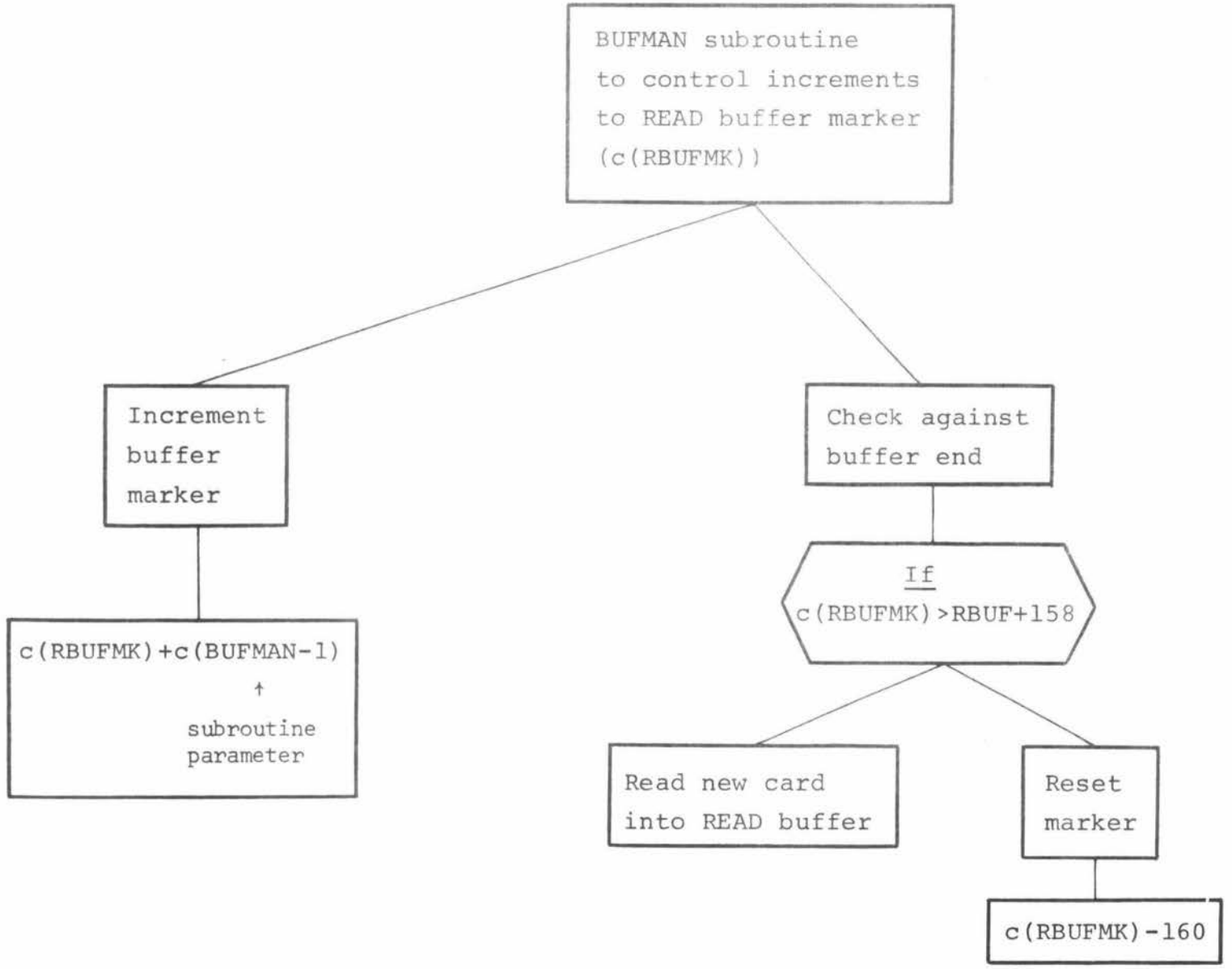

INPUT STRING
Place 'string descriptor' type digit at top of Stack
c(c(STP)) ← 7
Set c(NODE2) to address next available string node (GRAB subroutine)
Fill address field in string descriptor and initialize length counter
c(c(SVP)) ← c(NODE2)
c(LENGTH) ← 0
Pack string away in string area
Terminate string input
OUTLOOP:
Repeat
Pack a node
Peel off up to next 6 string characters
Pack in string area
c(STRING)+1
c(NODE2)-4
Set flag at SAREA-12
c(c(NODE2)) ← c(c(STRING))
Link to next available node
Save
c(NODE2)+4
c(NODE1) ← c(NODE2)
Grab
Set c(NODE2) to address next available node (GRAB subroutine)
Link
Set flag at NODE2-3
c(c(NODE1)) ← c(NODE2)
Clear flag at NODE2-3

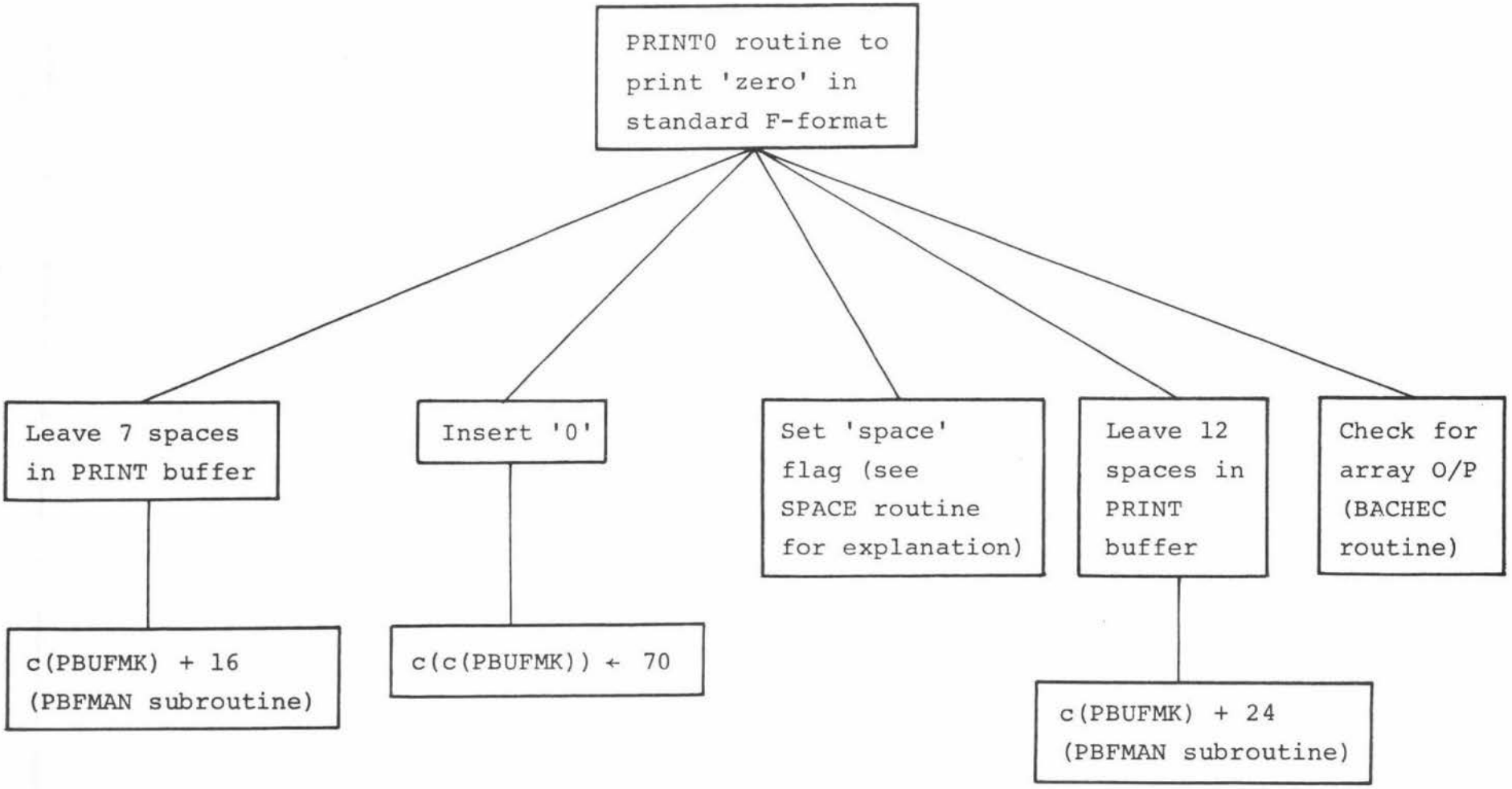

PEEL OFF UP TO NEXT 6 STRING CHARACTERS
Initialize accumulator pointer & loop counter
c(STRING) ← SAREA-14
c(COUNT) ← 6
Repeat
Check next character
c(RBUFMK)+1 (BUFMAN subroutine)
Set flag at c(RBUFMK) c(RBUFMK)+1
If c(c(RBUFMK))='!'
Check next character
c(RBUFMK)+1 (BUFMAN subroutine)
Set flag at c(RBUFMK) c(RBUFMK)+1
Choice on c(c(RBUFMK))
',' or blank
Exit from OUTLOOP
'!'
Convert to internal form
c(c(RBUFMK)) ← 12
other
ERROR (RD E2) Invalid input form
Pack character in accumulator
c(STRING)+3
c(c(STRING)) ← c(c(RBUFMK))
c(STRING)-1
Clear flag at c(STRING)
Inc. length counter
c(LENGTH)+1
Exit?
c(COUNT)-1
If c(COUNT)=0
Exit

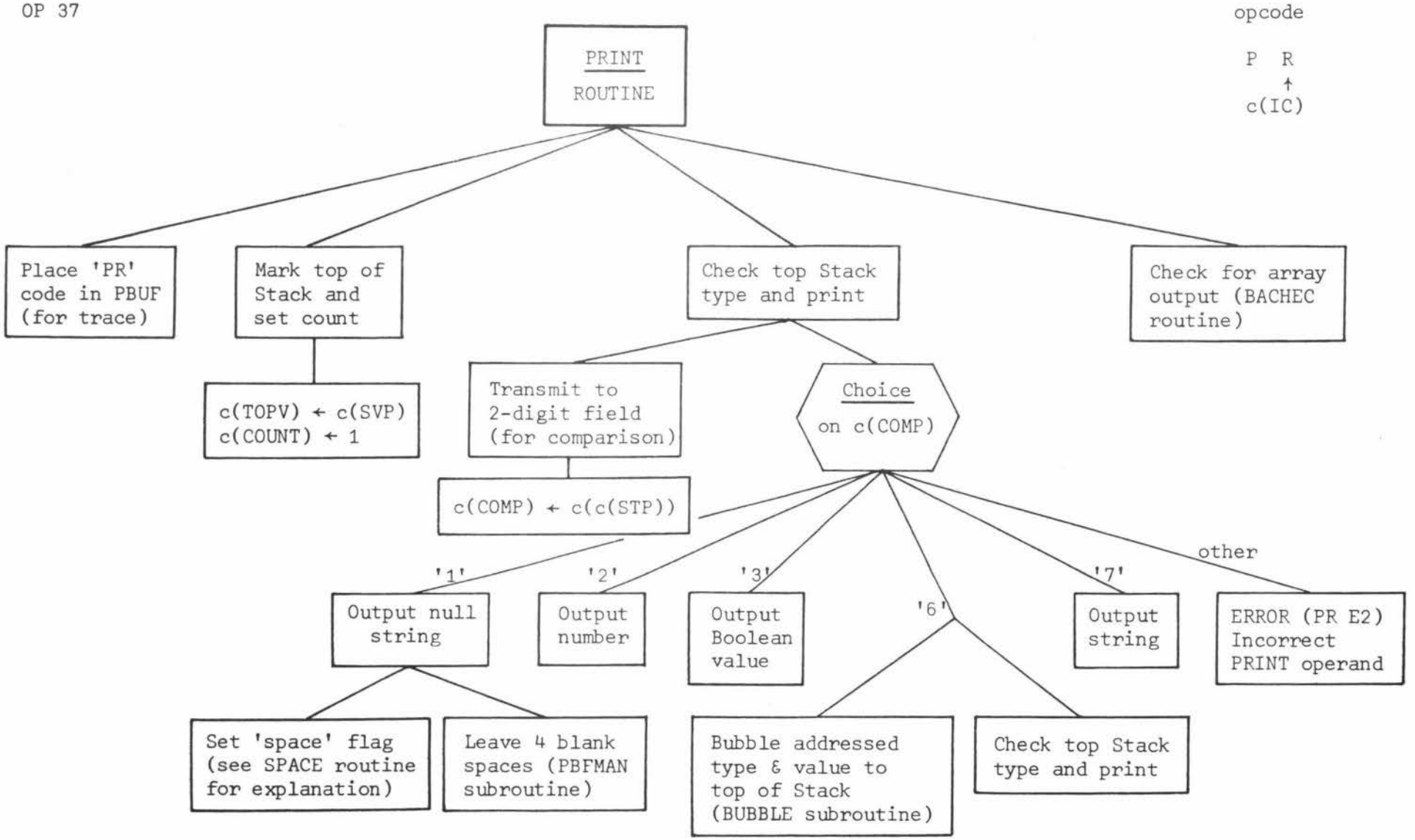

TERMINATE STRING INPUT
Null string?
Choice
c(LENGTH)=0?
Yes
No
Place 'null str.' type digit at TOS
c(c(STP)) ← 1
Store at c(A) (STOR subroutine)
Fill length field in descriptor
c(SVP)-5
c(c(SVP)) ← c(LENGTH)
c(SVP)+5
Pack last string chunk
Null?
Choice
c(COUNT)=6?
Yes
No
Terminate string at NODE1
Set flag at c(NODE1)
Return NODE2 to AVAIL list (RETURN subroutine)
Store TOS descriptor at c(A) (STOR subroutine)
Terminate string at NODE2
Set flag at c(NODE2)
Pack last chunk
c(NODE2)-4
Set flag at SAREA-12
c(STRING)+1
c(c(NODE2)) ← c(c(STRING))
Store TOS descriptor at c(A) (STOR subroutine)

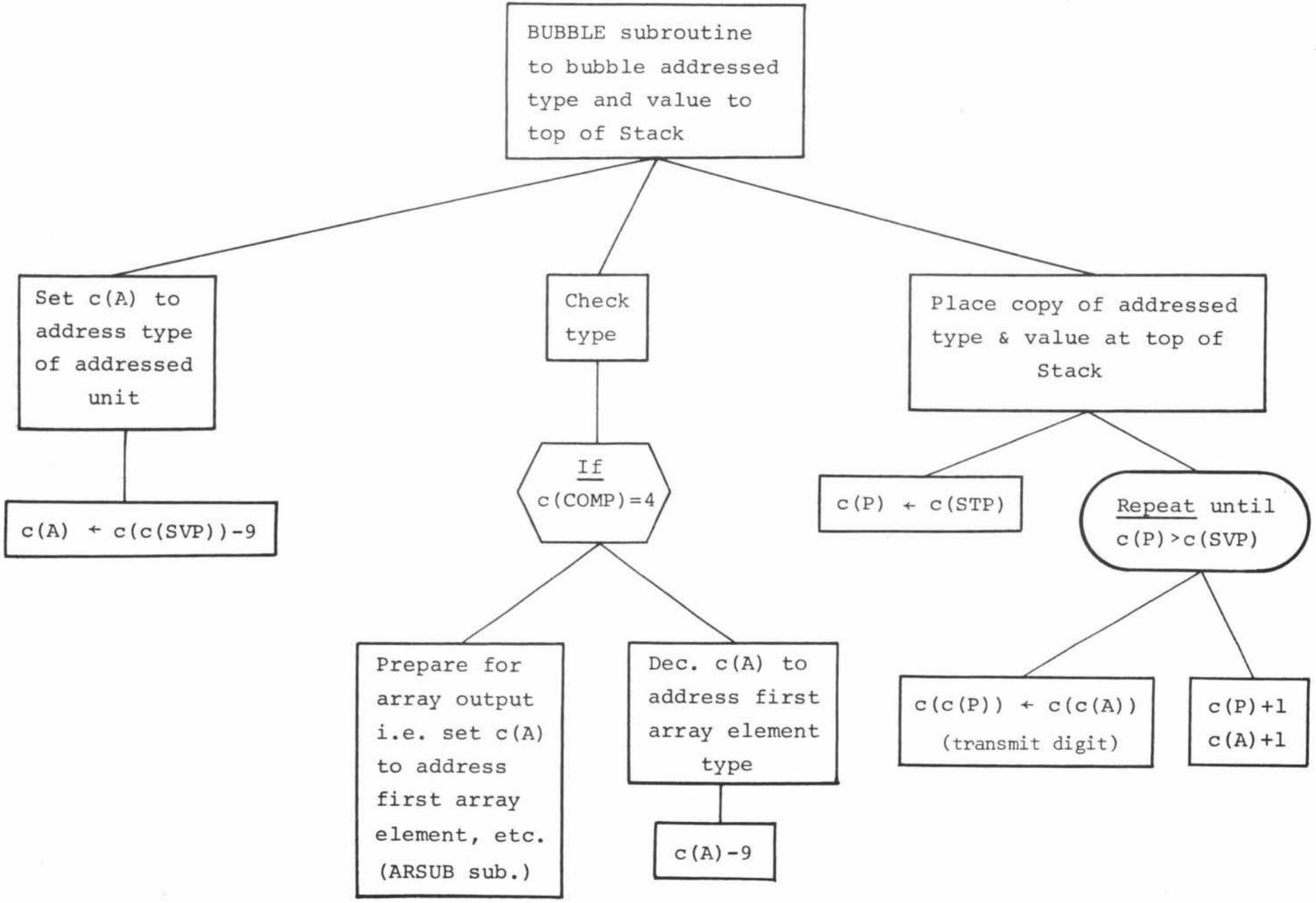

INPUT BOOLEAN VALUE
TRUE or FALSE?
Set Boolean type digit at top of Stack
Skip remaining characters until comma or blank
Store TOS Boolean value and type at c(A) (STOR sub.)
Choice on c(c(RBUFMK))
'T'
'F'
Place value TRUE at TOS
Place value FALSE at TOS
c(c(SVP)) ← 01
c(c(SVP)) ← 00
c(c(STP)) ← 3
Repeat
Check next character
c(RBUFMK)+1 (BUFMAN subroutine)
Set flag at c(RBUFMK) c(RBUFMK)+1
Comma or blank?
If c(c(RBUFMK))=23 or 0
Exit

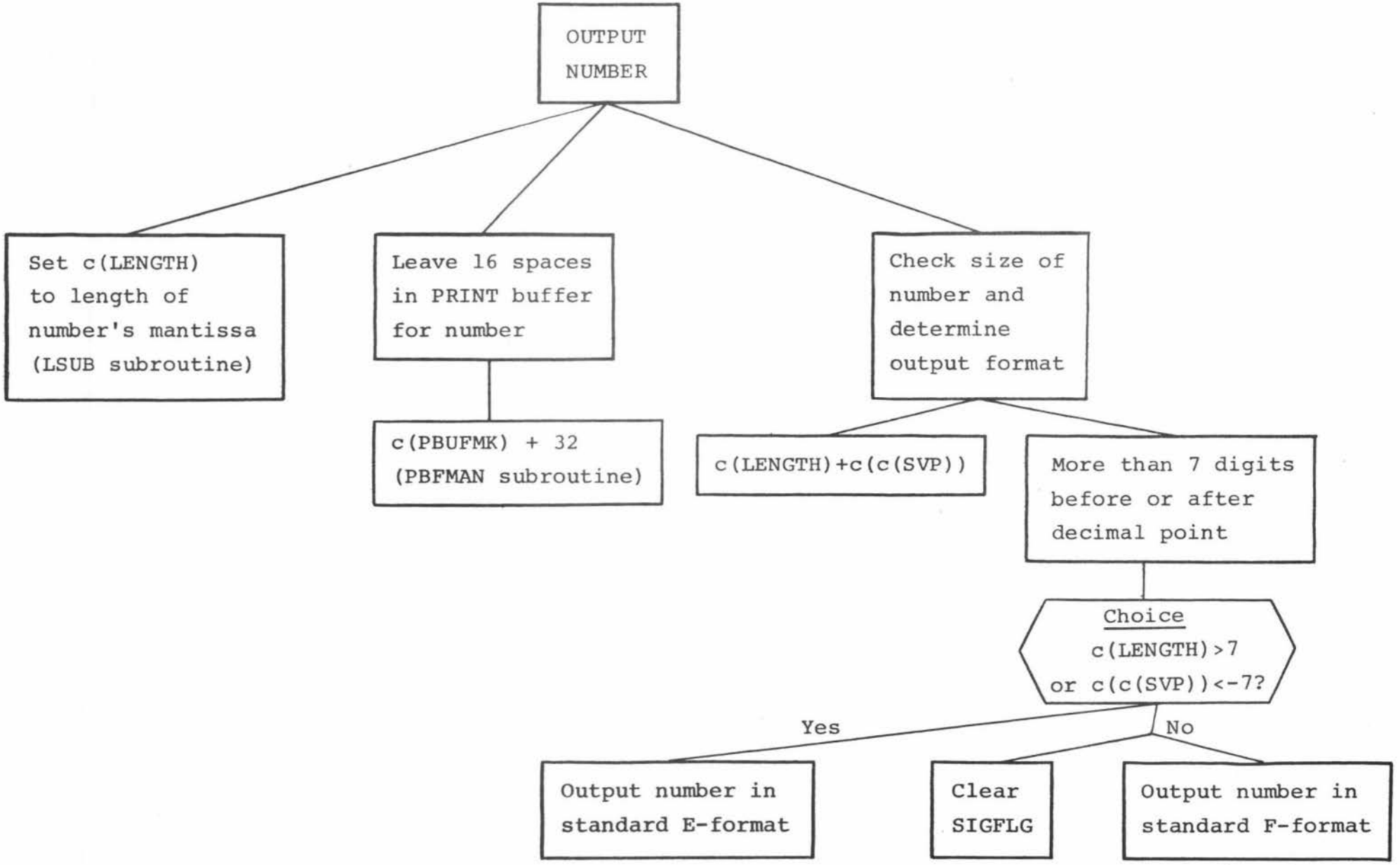

ARCHEC routine to check for array input
Array input?
If
Flag at ARRAY
Pop top of Stack
c(SVP)-10
c(STP)-10
(POPTOP subroutine)
Increment array element counter
c(ACOUNT)+1
Check count against array size
Choice
c(ACOUNT)>c(SIZE)?
Yes
No
Terminate array input
Clear flag at ARRAY
Increment c(A) to address next array element location
c(A)+10
Input next card element

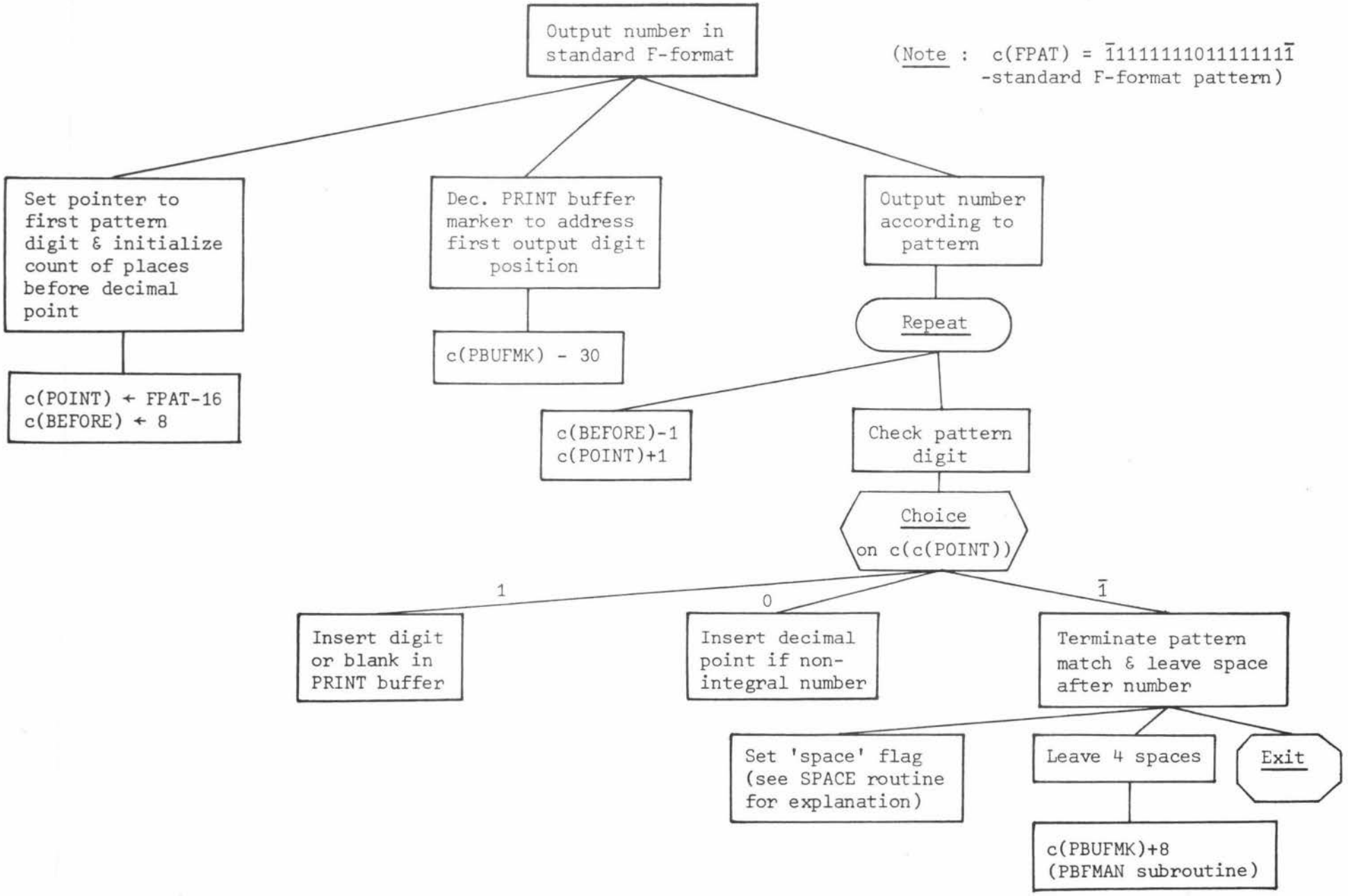

BUFMAN subroutine
to control increments
to READ buffer marker
(c(RBUFMK))
Increment
buffer
marker
c(RBUFMK)+c(BUFMAN-1)
subroutine
parameter
Check against
buffer end
If
c(RBUFMK)>RBUF+158
Read new card
into READ buffer
Reset
marker
c(RBUFMK)-160

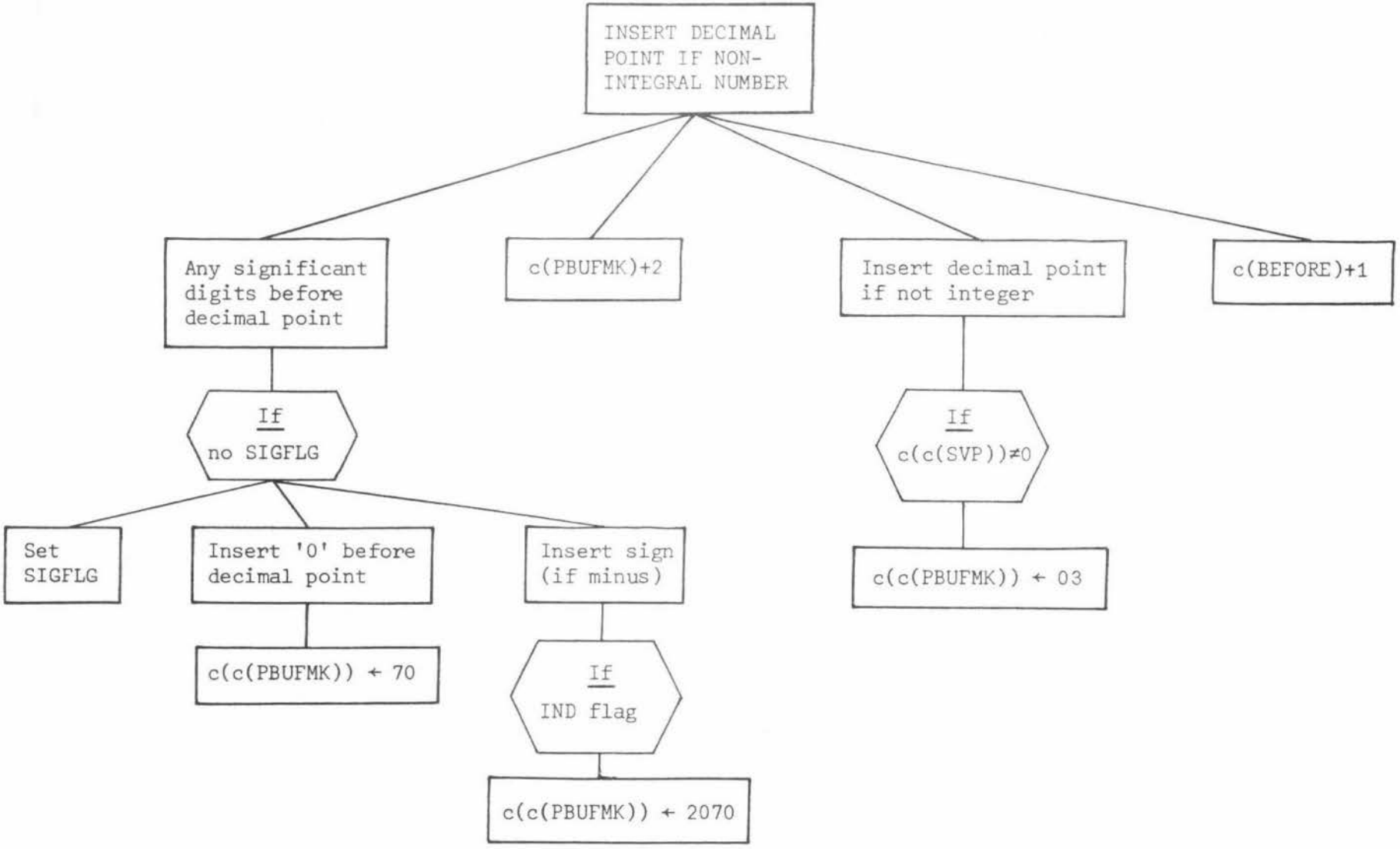

PRINT0 routine to print 'zero' in standard F-format
Leave 7 spaces in PRINT buffer
Insert '0'
Set 'space' flag (see SPACE routine for explanation)
Leave 12 spaces in PRINT buffer
Check for array O/P (BACHEC routine)
c(PBUFMK) + 16 (PBFMAN subroutine)
c(c(PBUFMK)) ← 70
c(PBUFMK) + 24 (PBFMAN subroutine)

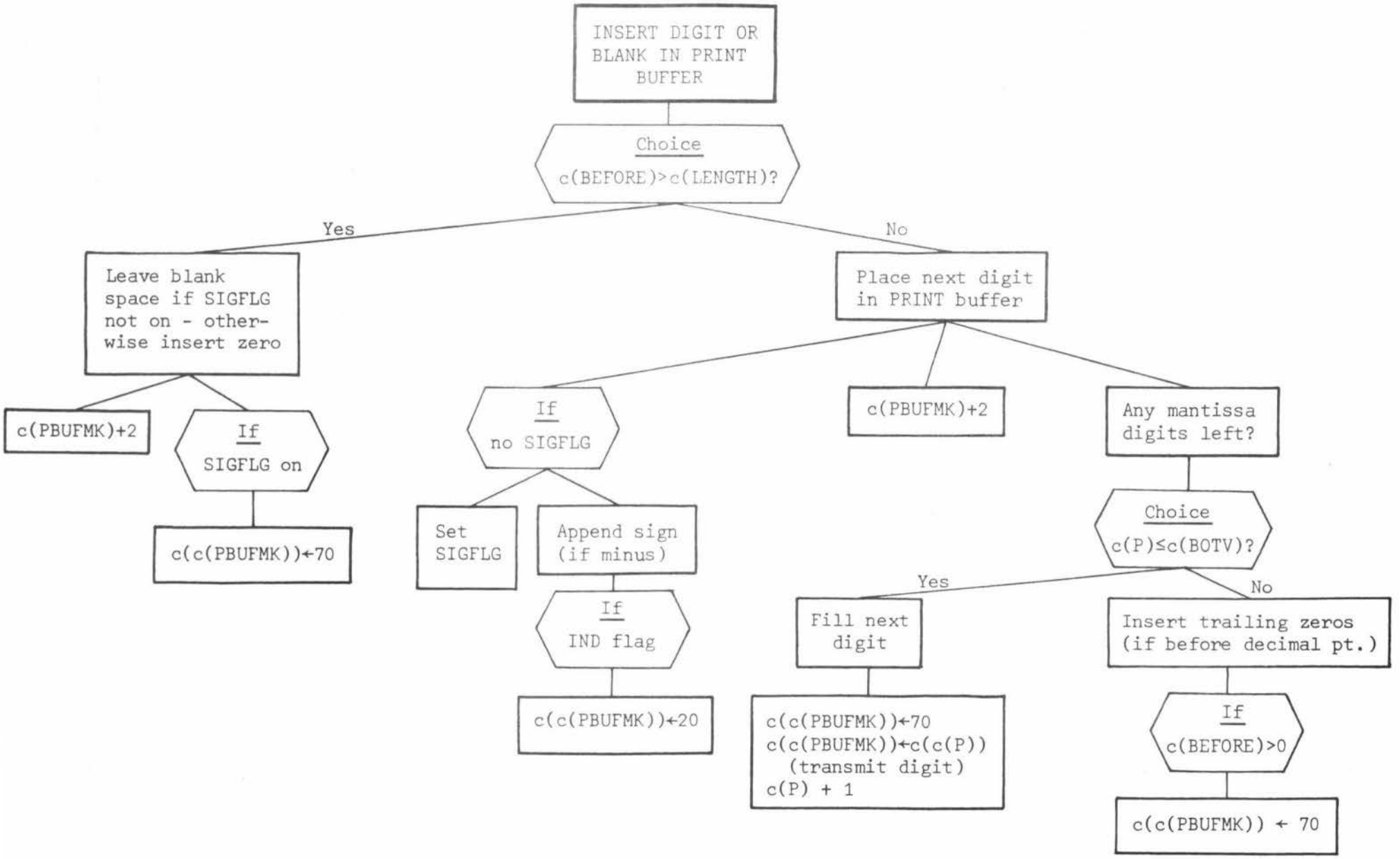

opcode
P R
c(IC)
PRINT ROUTINE
Place 'PR' code in PBUF (for trace)
Mark top of Stack and set count
c(TOPV) ← c(SVP)
c(COUNT) ← 1
Check top Stack type and print
Transmit to 2-digit field (for comparison)
c(COMP) ← c(c(STP))
Choice on c(COMP)
'1'
'2'
'3'
'6'
'7'
other
Output null string
Output number
Output Boolean value
Output string
ERROR (PR E2) Incorrect PRINT operand
Set 'space' flag (see SPACE routine for explanation)
Leave 4 blank spaces (PBFMAN subroutine)
Bubble addressed type & value to top of Stack (BUBBLE subroutine)
Check top Stack type and print
Check for array output (BACHEC routine)

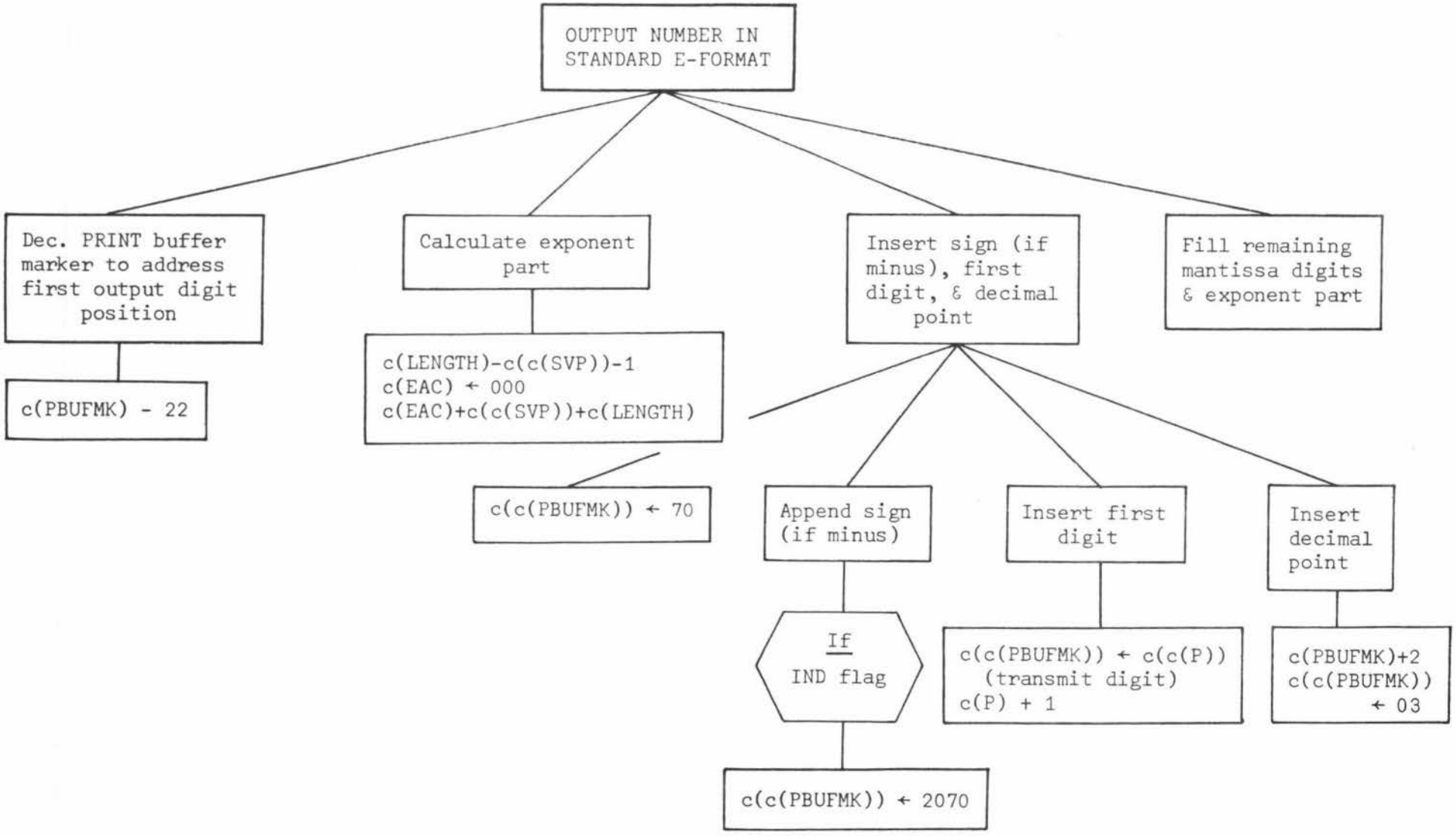

BUBBLE subroutine to bubble addressed type and value to top of Stack
Set c(A) to address type of addressed unit
c(A) ← c(c(SVP))-9
Check type
If c(COMP)=4
Prepare for array output i.e. set c(A) to address first array element, etc. (ARSUB sub.)
Dec. c(A) to address first array element type
c(A)-9
Place copy of addressed type & value at top of Stack
c(P) ← c(STP)
Repeat until c(P)>c(SVP)
c(c(P)) ← c(c(A)) (transmit digit)
c(P)+1
c(A)+1

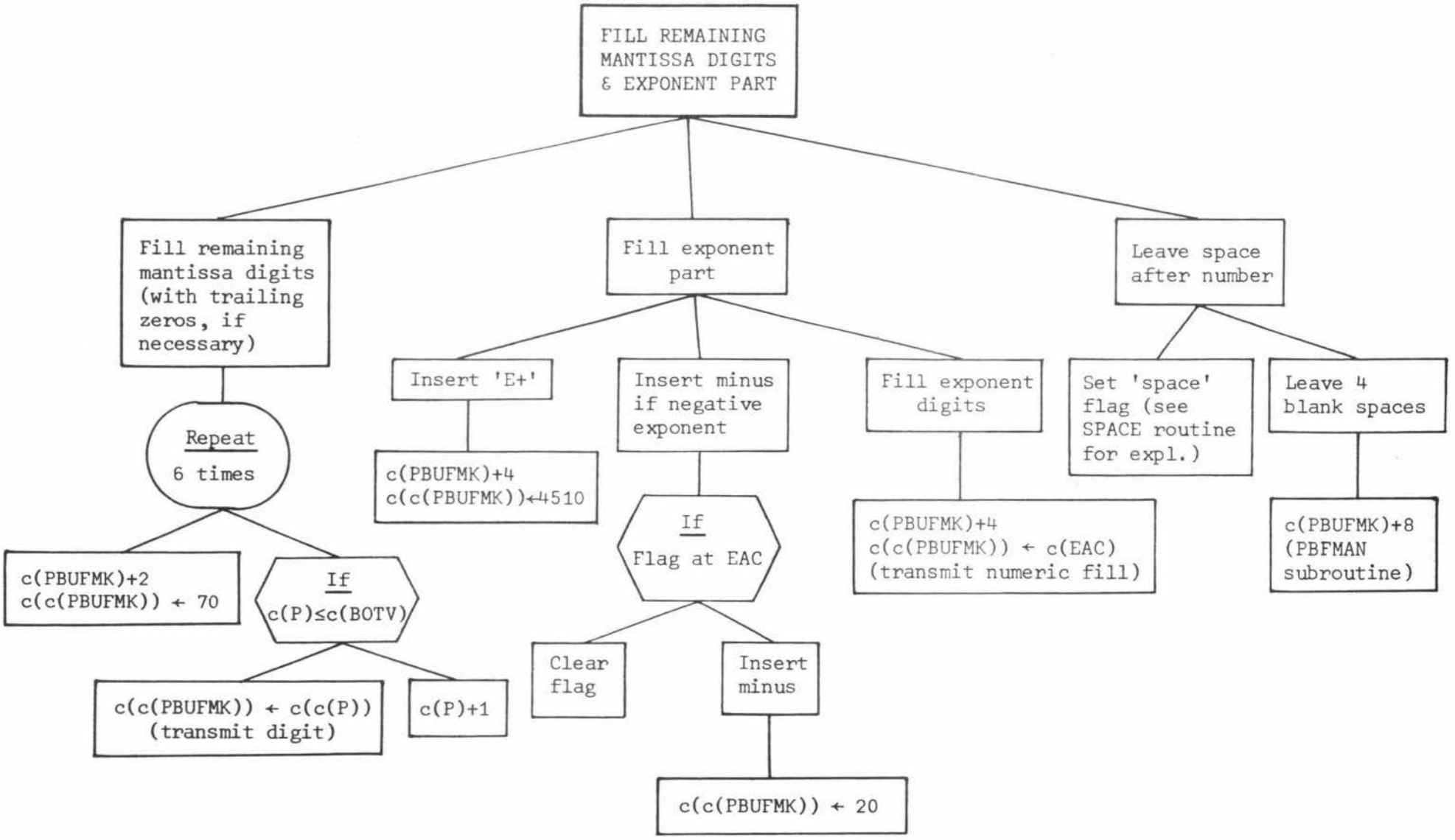

OUTPUT NUMBER
Set c(LENGTH) to length of number's mantissa (LSUB subroutine)
Leave 16 spaces in PRINT buffer for number
c(PBUFMK) + 32 (PBFMAN subroutine)
Check size of number and determine output format
c(LENGTH)+c(c(SVP))
More than 7 digits before or after decimal point
Choice
c(LENGTH)>7 or c(c(SVP))<-7?
Yes
No
Output number in standard E-format
Clear SIGFLG
Output number in standard F-format

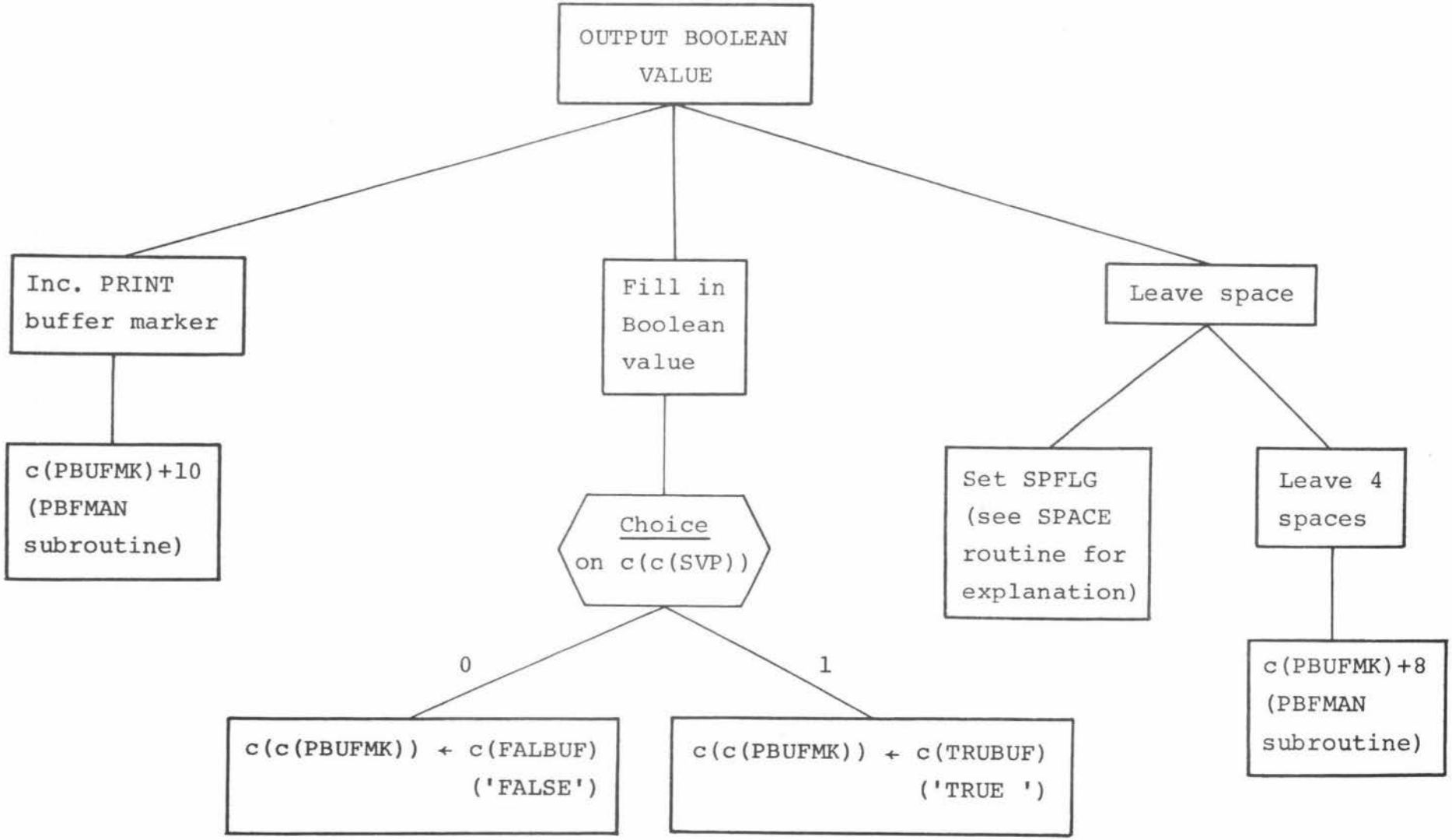

Output number in standard F-format
(Note : c(FPAT) = 1̄111111011111111̄
-standard F-format pattern)
Set pointer to first pattern digit & initialize count of places before decimal point
c(POINT) ← FPAT-16
c(BEFORE) ← 8
Dec. PRINT buffer marker to address first output digit position
c(PBUFMK) - 30
Output number according to pattern
Repeat
c(BEFORE)-1
c(POINT)+1
Check pattern digit
Choice
on c(c(POINT))
1
0
1̄
Insert digit or blank in PRINT buffer
Insert decimal point if non-integral number
Terminate pattern match & leave space after number
Set 'space' flag (see SPACE routine for explanation)
Leave 4 spaces
Exit
c(PBUFMK)+8
(PBFMAN subroutine)

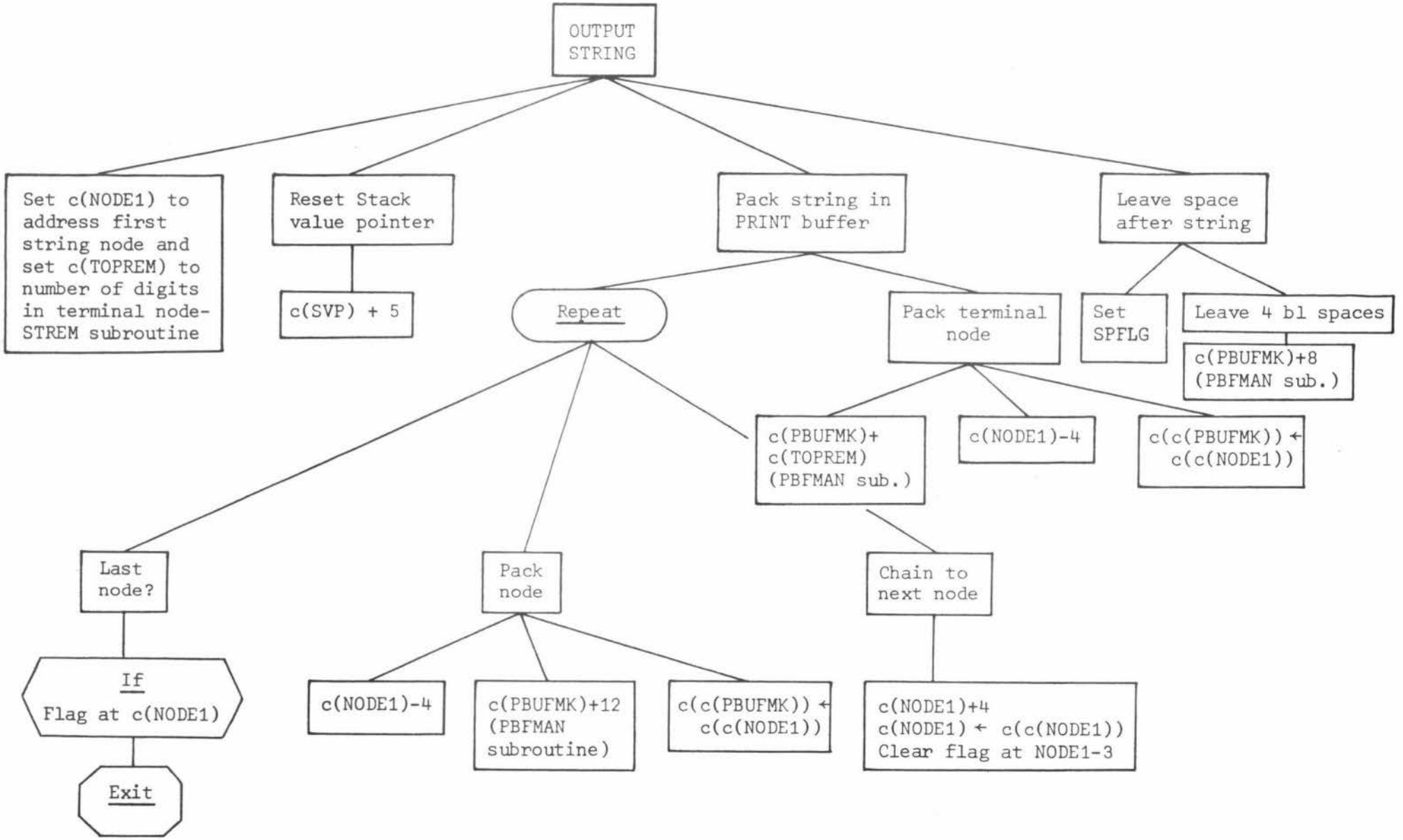

INSERT DECIMAL POINT IF NON-INTEGRAL NUMBER
Any significant digits before decimal point
c(PBUFMK)+2
Insert decimal point if not integer
c(BEFORE)+1
If no SIGFLG
If c(c(SVP))≠0
Set SIGFLG
Insert '0' before decimal point
Insert sign (if minus)
c(c(PBUFMK)) ← 03
c(c(PBUFMK)) ← 70
If IND flag
c(c(PBUFMK)) ← 2070

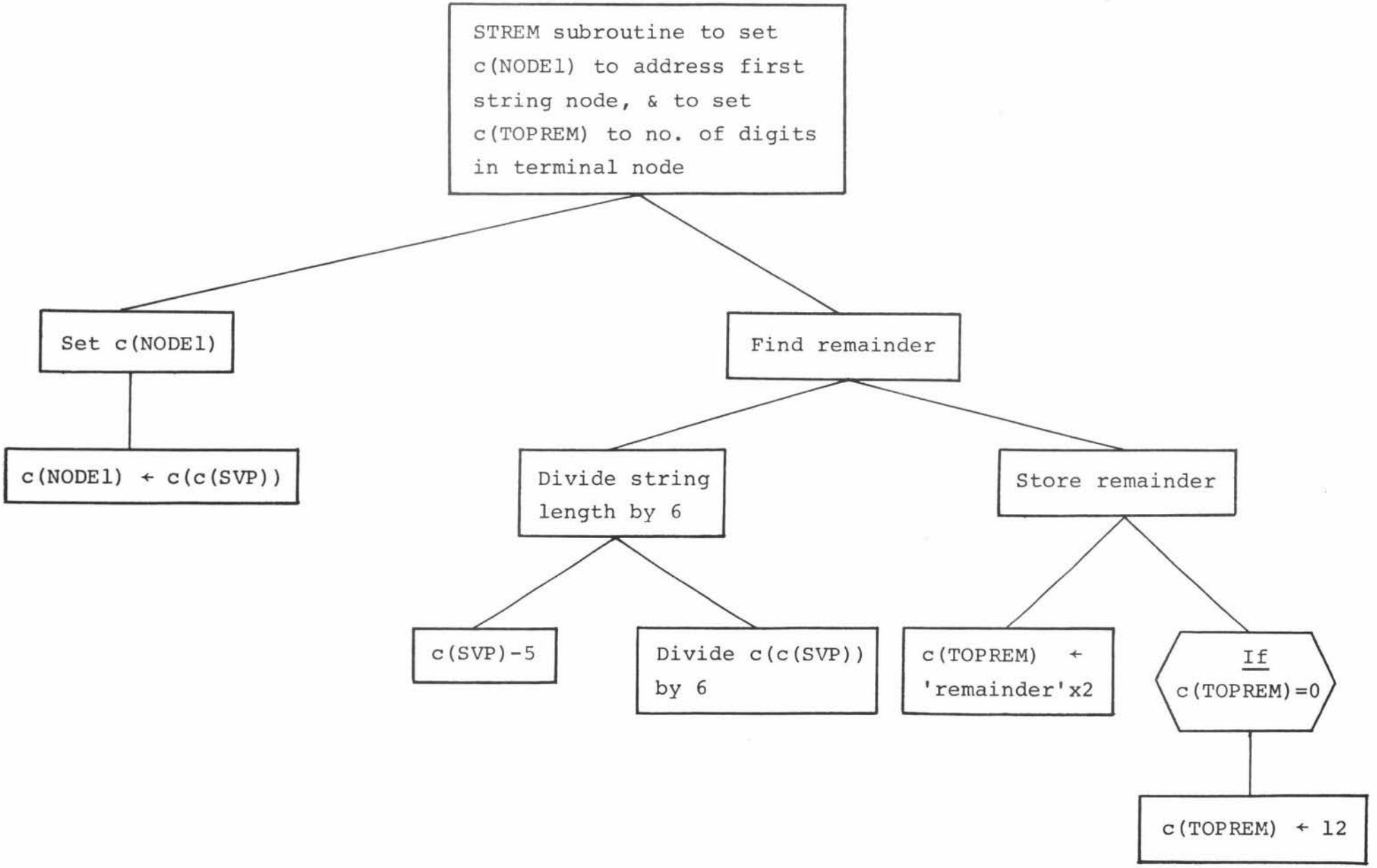

INSERT DIGIT OR BLANK IN PRINT BUFFER
Choice
c(BEFORE)>c(LENGTH)?
Yes
No
Leave blank space if SIGFLG not on - otherwise insert zero
c(PBUFMK)+2
If
SIGFLG on
c(c(PBUFMK))←70
Place next digit in PRINT buffer
If
no SIGFLG
Set SIGFLG
Append sign (if minus)
If
IND flag
c(c(PBUFMK))←20
c(PBUFMK)+2
Any mantissa digits left?
Choice
c(P)≤c(BOTV)?
Yes
No
Fill next digit
c(c(PBUFMK))←70
c(c(PBUFMK))←c(c(P))
(transmit digit)
c(P) + 1
Insert trailing zeros (if before decimal pt.)
If
c(BEFORE)>0
c(c(PBUFMK)) ← 70


OUTPUT NUMBER IN STANDARD E-FORMAT
Dec. PRINT buffer marker to address first output digit position
c(PBUFMK) - 22
Calculate exponent part
c(LENGTH)-c(c(SVP))-1
c(EAC) ← 000
c(EAC)+c(c(SVP))+c(LENGTH)
c(c(PBUFMK)) ← 70
Insert sign (if minus), first digit, & decimal point
Append sign (if minus)
If
IND flag
c(c(PBUFMK)) ← 2070
Insert first digit
c(c(PBUFMK)) ← c(c(P))
(transmit digit)
c(P) + 1
Insert decimal point
c(PBUFMK)+2
c(c(PBUFMK))
← 03
Fill remaining mantissa digits & exponent part

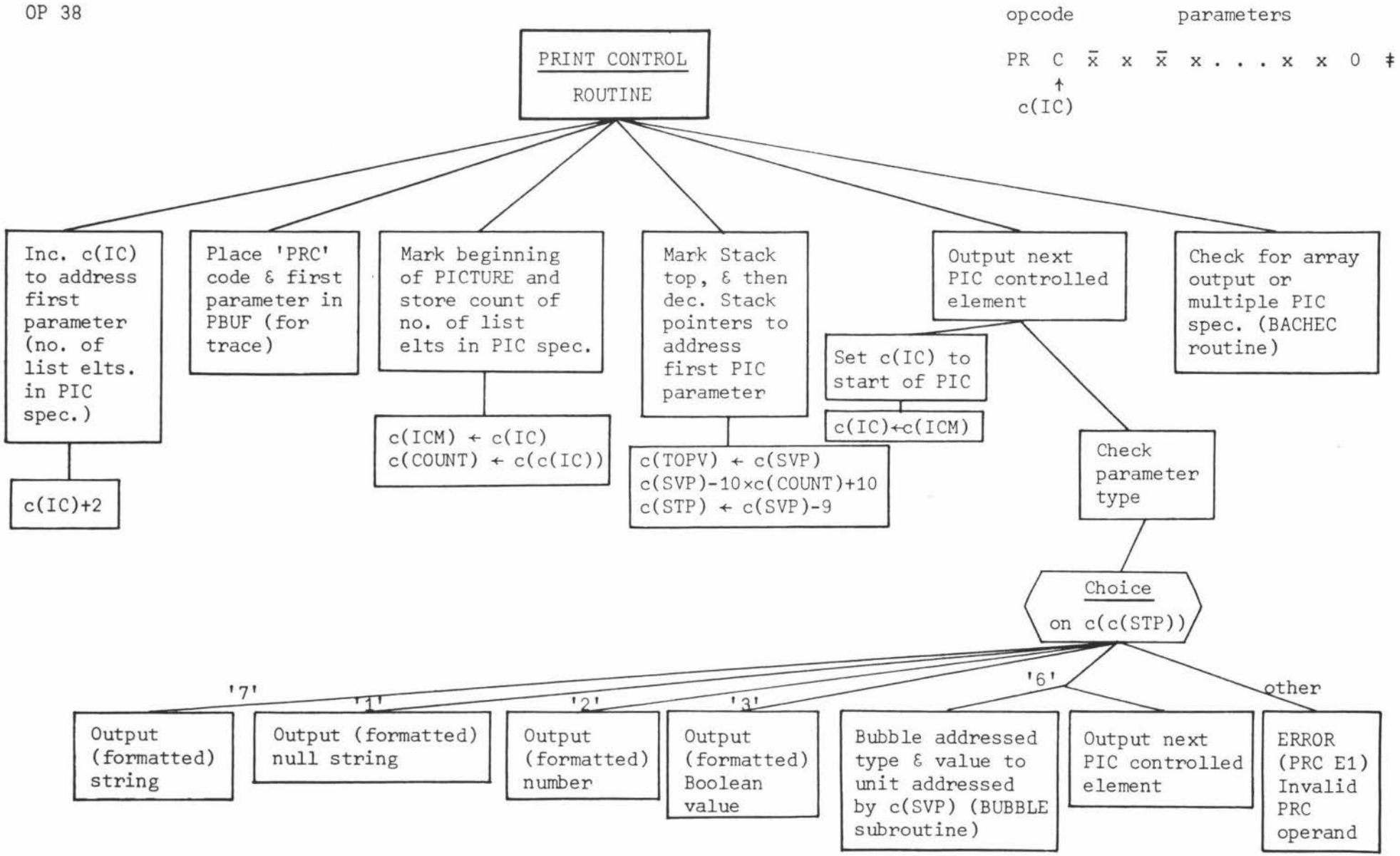

FILL REMAINING MANTISSA DIGITS & EXPONENT PART
Fill remaining mantissa digits (with trailing zeros, if necessary)
Repeat 6 times
c(PBUFMK)+2
c(c(PBUFMK)) ← 70
If c(P)≤c(BOTV)
c(c(PBUFMK)) ← c(c(P)) (transmit digit)
c(P)+1
Fill exponent part
Insert 'E+'
c(PBUFMK)+4
c(c(PBUFMK))←4510
Insert minus if negative exponent
If Flag at EAC
Clear flag
Insert minus
c(c(PBUFMK)) ← 20
Fill exponent digits
c(PBUFMK)+4
c(c(PBUFMK)) ← c(EAC)
(transmit numeric fill)
Leave space after number
Set 'space' flag (see SPACE routine for expl.)
Leave 4 blank spaces
c(PBUFMK)+8
(PBFMAN subroutine)

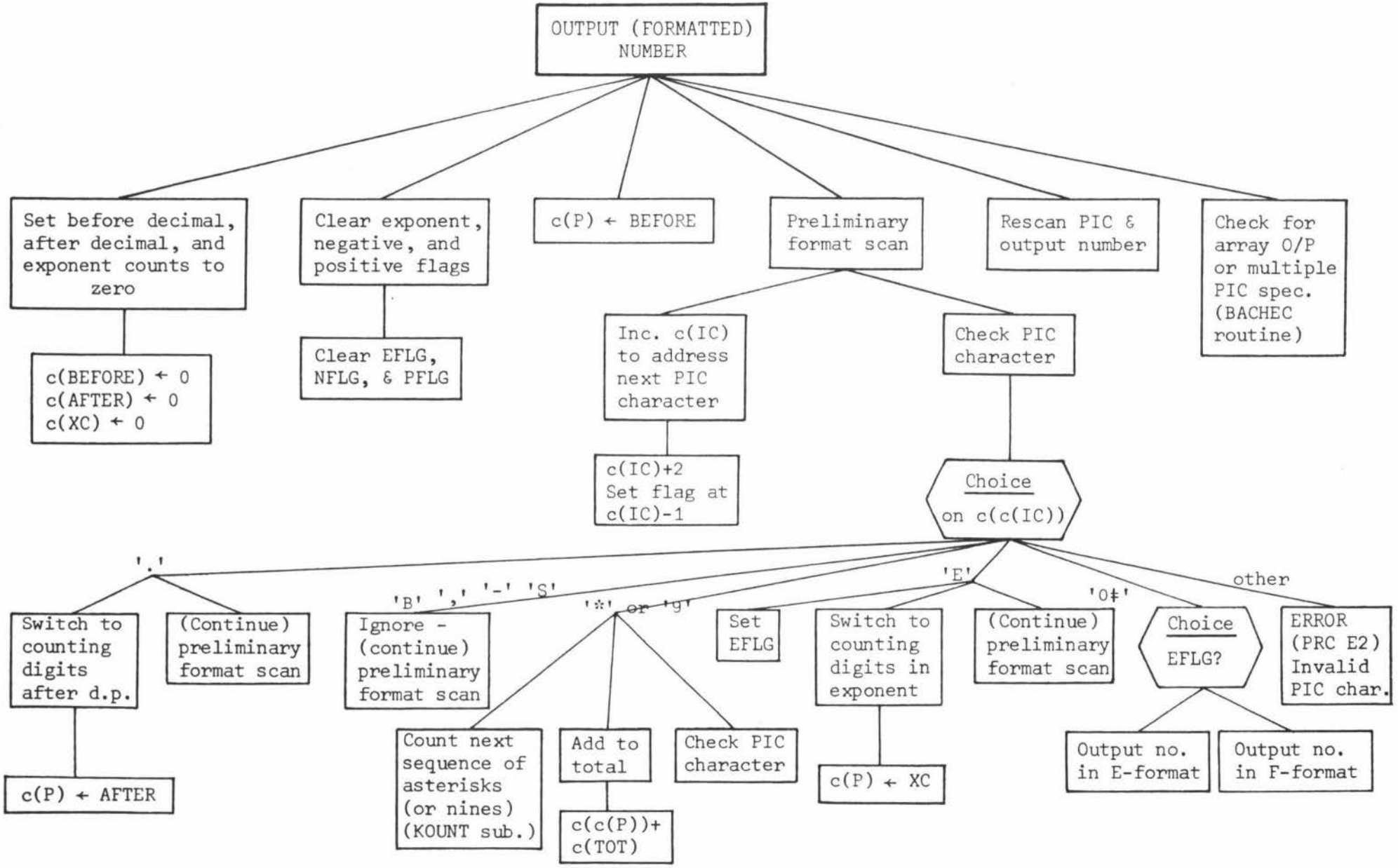

OUTPUT BOOLEAN VALUE
Inc. PRINT buffer marker
c(PBUFMK)+10 (PBFMAN subroutine)
Fill in Boolean value
Choice on c(c(SVP))
0
1
c(c(PBUFMK)) ← c(FALBUF) ('FALSE')
c(c(PBUFMK)) ← c(TRUBUF) ('TRUE ')
Leave space
Set SPFLG (see SPACE routine for explanation)
Leave 4 spaces
c(PBUFMK)+8 (PBFMAN subroutine)

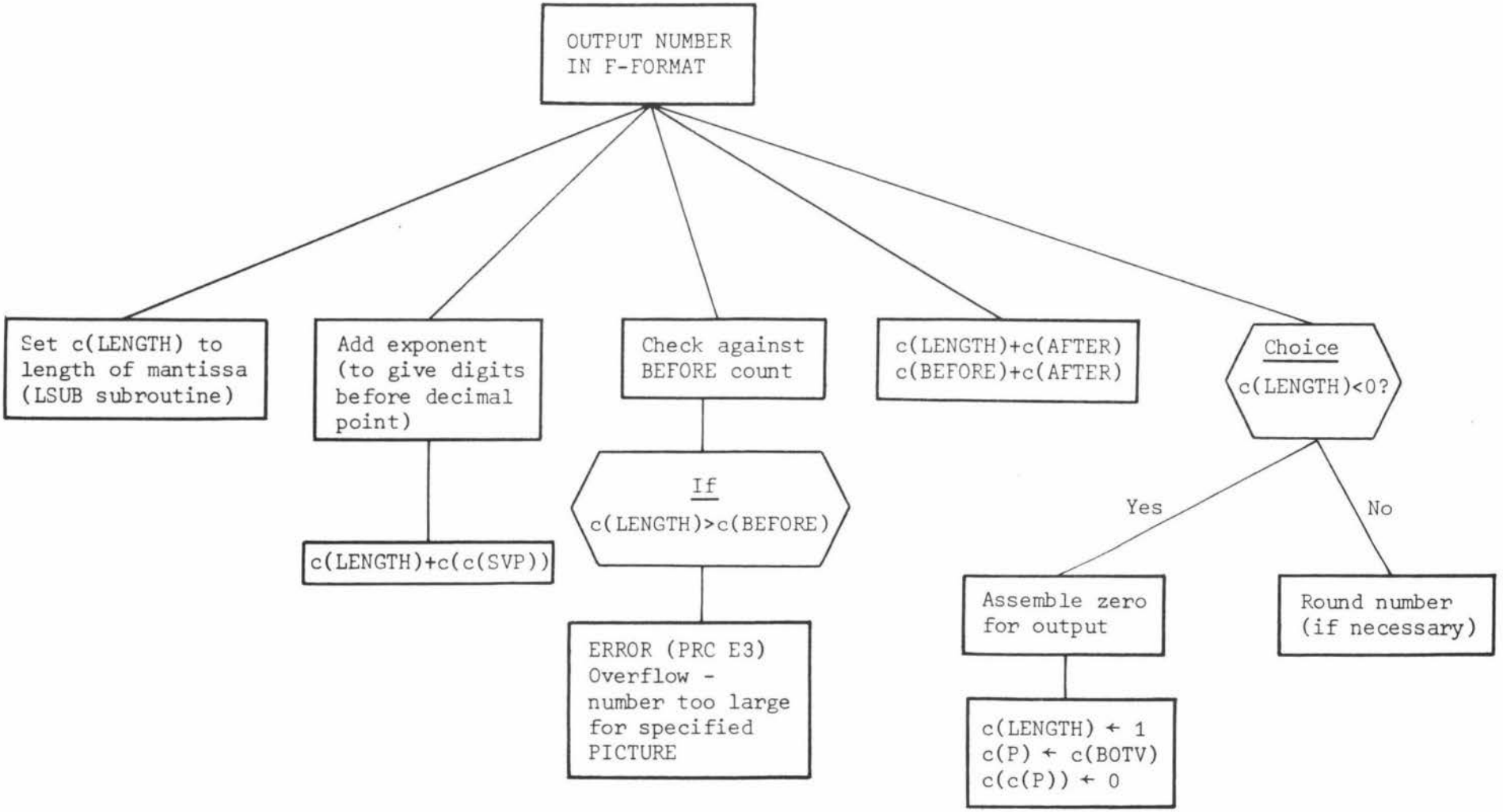

OUTPUT STRING
Set c(NODE1) to address first string node and set c(TOPREM) to number of digits in terminal node-STREM subroutine
Reset Stack value pointer
c(SVP) + 5
Pack string in PRINT buffer
Repeat
Pack terminal node
c(PBUFMK)+ c(TOPREM) (PBFMAN sub.)
c(NODE1)-4
c(c(PBUFMK)) ← c(c(NODE1))
Leave space after string
Set SPFLG
Leave 4 bl spaces
c(PBUFMK)+8 (PBFMAN sub.)
Last node?
If Flag at c(NODE1)
Exit
Pack node
c(NODE1)-4
c(PBUFMK)+12 (PBFMAN subroutine)
c(c(PBUFMK)) ← c(c(NODE1))
Chain to next node
c(NODE1)+4
c(NODE1) ← c(c(NODE1))
Clear flag at NODE1-3

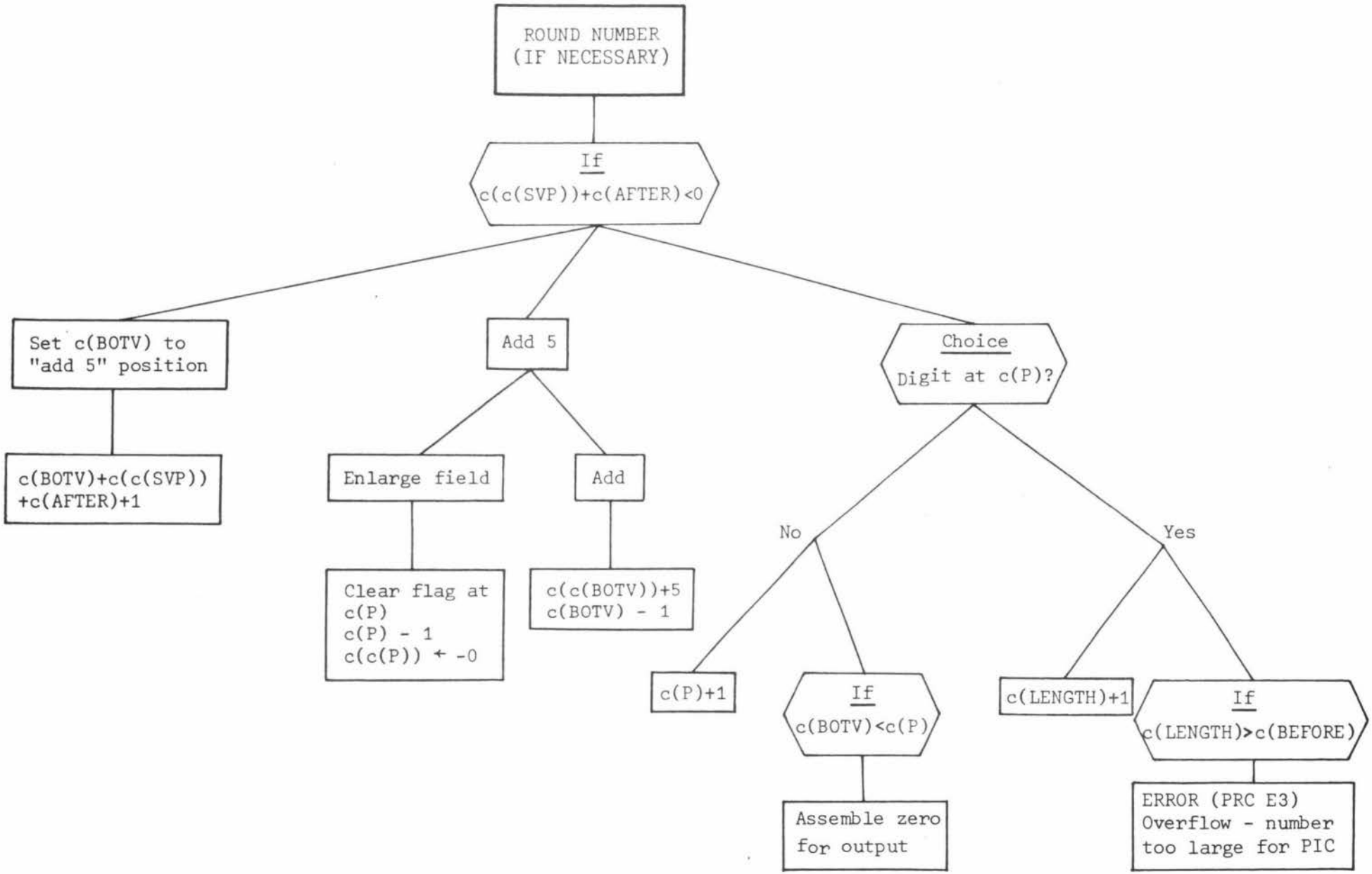

STREM subroutine to set c(NODE1) to address first string node, & to set c(TOPREM) to no. of digits in terminal node
Set c(NODE1)
c(NODE1) ← c(c(SVP))
Find remainder
Divide string length by 6
c(SVP)-5
Divide c(c(SVP)) by 6
Store remainder
c(TOPREM) ← 'remainder'x2
If c(TOPREM)=0
c(TOPREM) ← 12

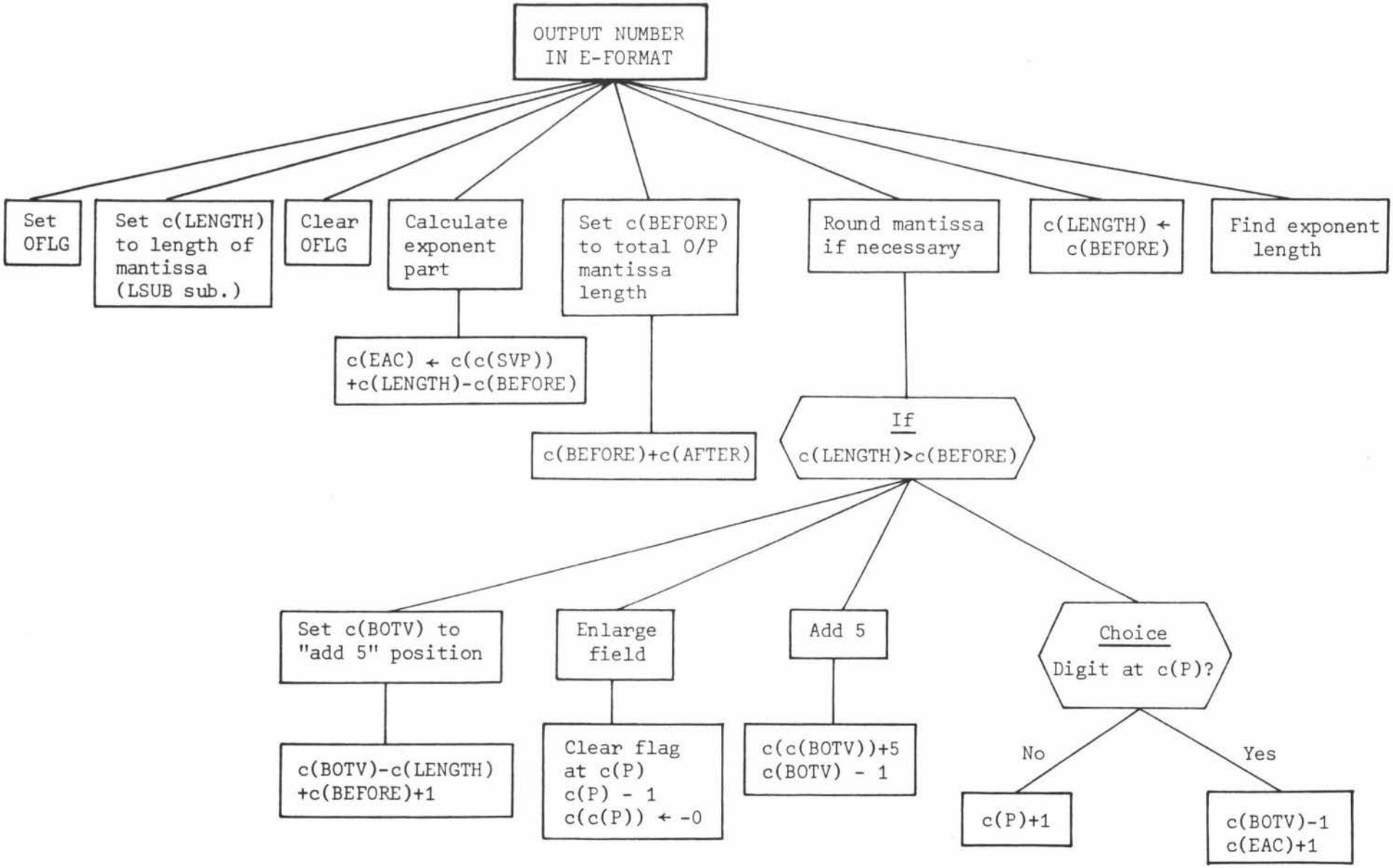
OP 38

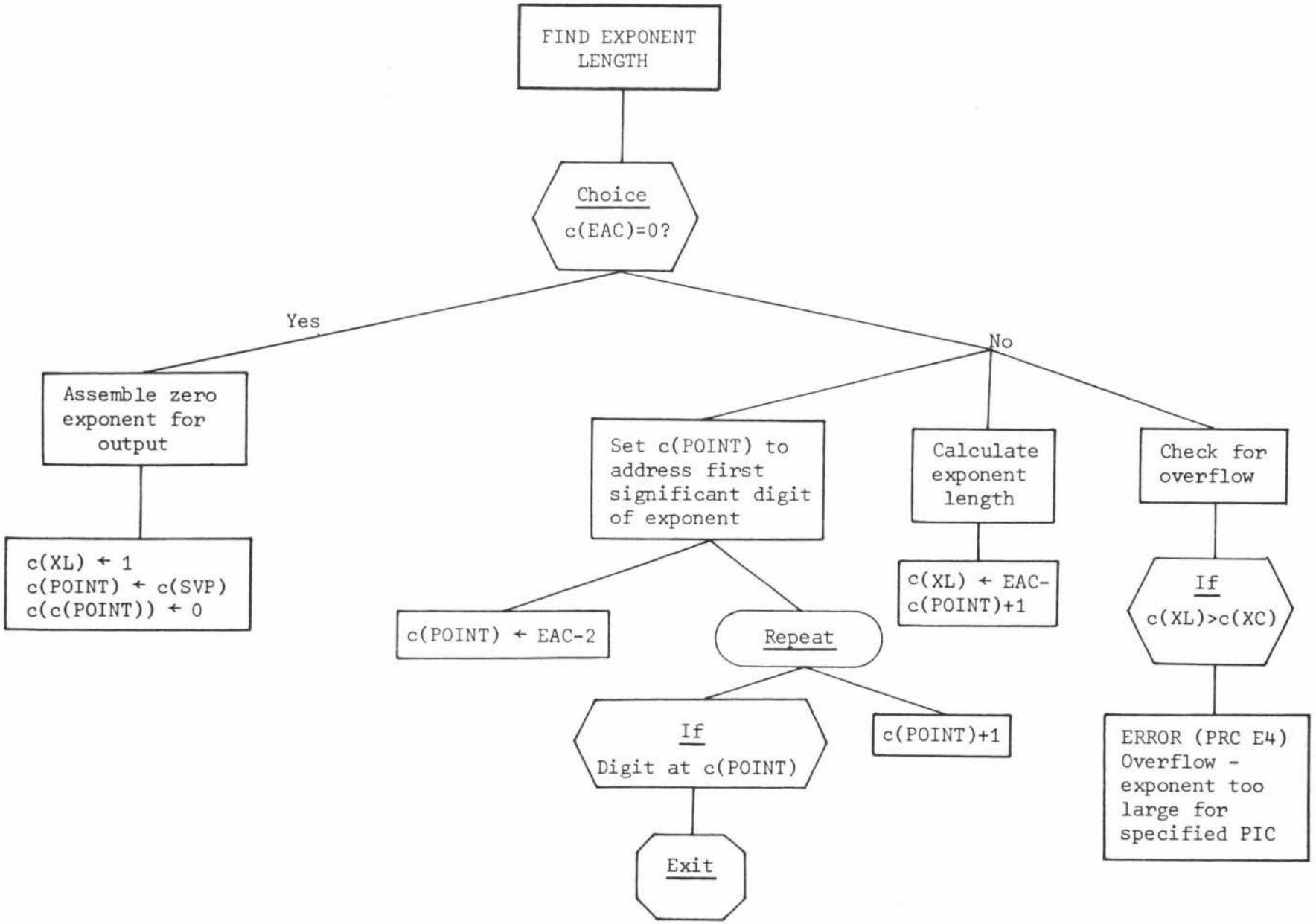

opcode
parameters
PR C x̄ x x̄ x . . . x x 0 ‡
c(IC)
PRINT CONTROL ROUTINE
Inc. c(IC) to address first parameter (no. of list elts. in PIC spec.)
c(IC)+2
Place 'PRC' code & first parameter in PBUF (for trace)
Mark beginning of PICTURE and store count of no. of list elts in PIC spec.
c(ICM) ← c(IC)
c(COUNT) ← c(c(IC))
Mark Stack top, & then dec. Stack pointers to address first PIC parameter
c(TOPV) ← c(SVP)
c(SVP)-10×c(COUNT)+10
c(STP) ← c(SVP)-9
Output next PIC controlled element
Set c(IC) to start of PIC
c(IC)←c(ICM)
Check parameter type
Choice on c(c(STP))
'7'
'1'
'2'
'3'
'6'
other
Output (formatted) string
Output (formatted) null string
Output (formatted) number
Output (formatted) Boolean value
Bubble addressed type & value to unit addressed by c(SVP) (BUBBLE subroutine)
Output next PIC controlled element
ERROR (PRC E1) Invalid PRC operand
Check for array output or multiple PIC spec. (BACHEC routine)

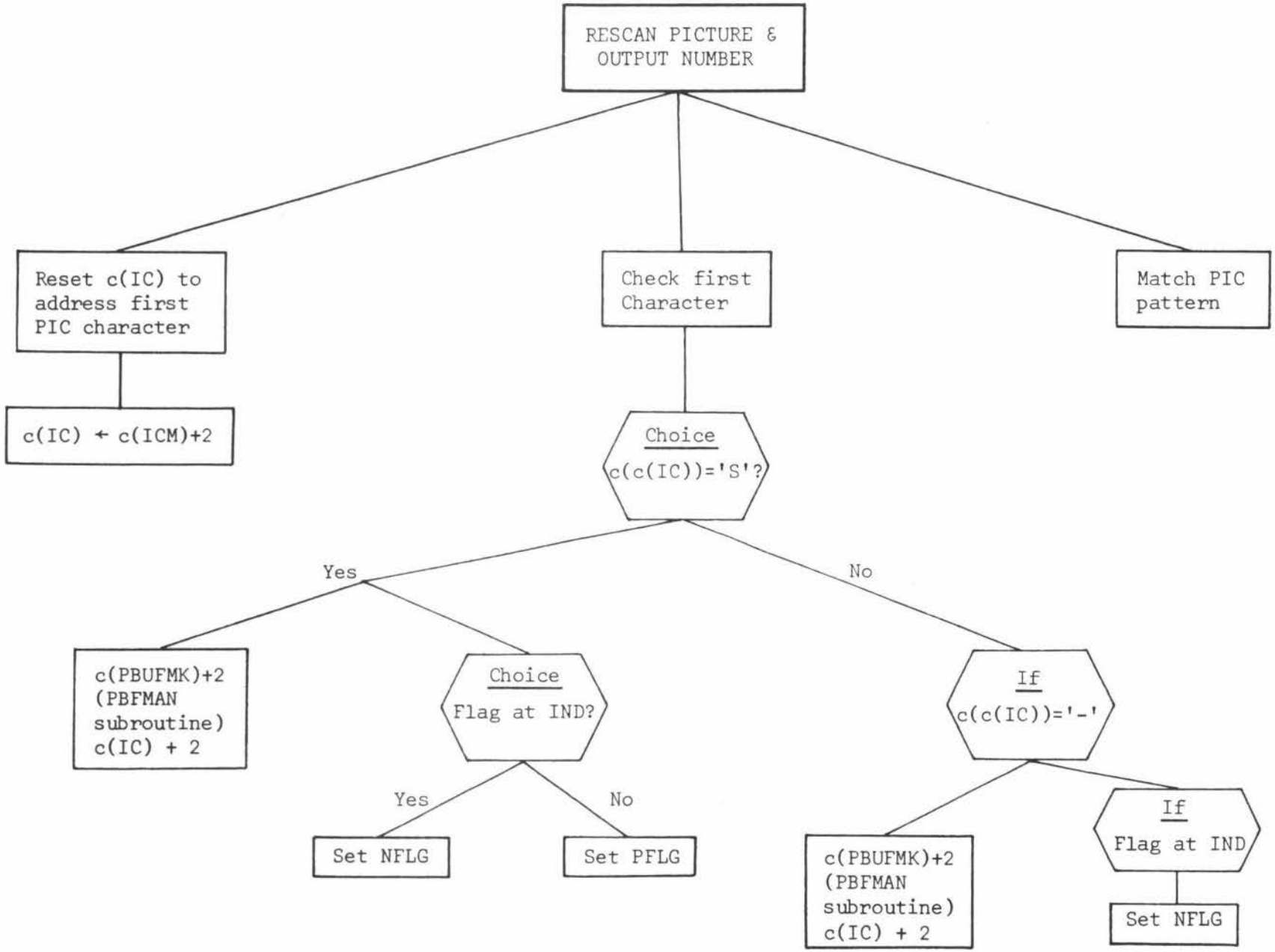

OUTPUT (FORMATTED) NUMBER
Set before decimal, after decimal, and exponent counts to zero
c(BEFORE) ← 0
c(AFTER) ← 0
c(XC) ← 0
Clear exponent, negative, and positive flags
Clear EFLG, NFLG, & PFLG
c(P) ← BEFORE
Preliminary format scan
Inc. c(IC) to address next PIC character
c(IC)+2
Set flag at c(IC)-1
Check PIC character
Choice on c(c(IC))
Rescan PIC & output number
Check for array O/P or multiple PIC spec. (BACHEC routine)
'.'
Switch to counting digits after d.p.
c(P) ← AFTER
(Continue) preliminary format scan
'B' ',' '-' 'S'
Ignore - (continue) preliminary format scan
'*' or '9'
Count next sequence of asterisks (or nines) (KOUNT sub.)
Add to total
c(c(P))+ c(TOT)
Check PIC character
'E'
Set EFLG
Switch to counting digits in exponent
c(P) ← XC
'0≠'
(Continue) preliminary format scan
other
Choice EFLG?
Output no. in E-format
Output no. in F-format
ERROR (PRC E2) Invalid PIC char.

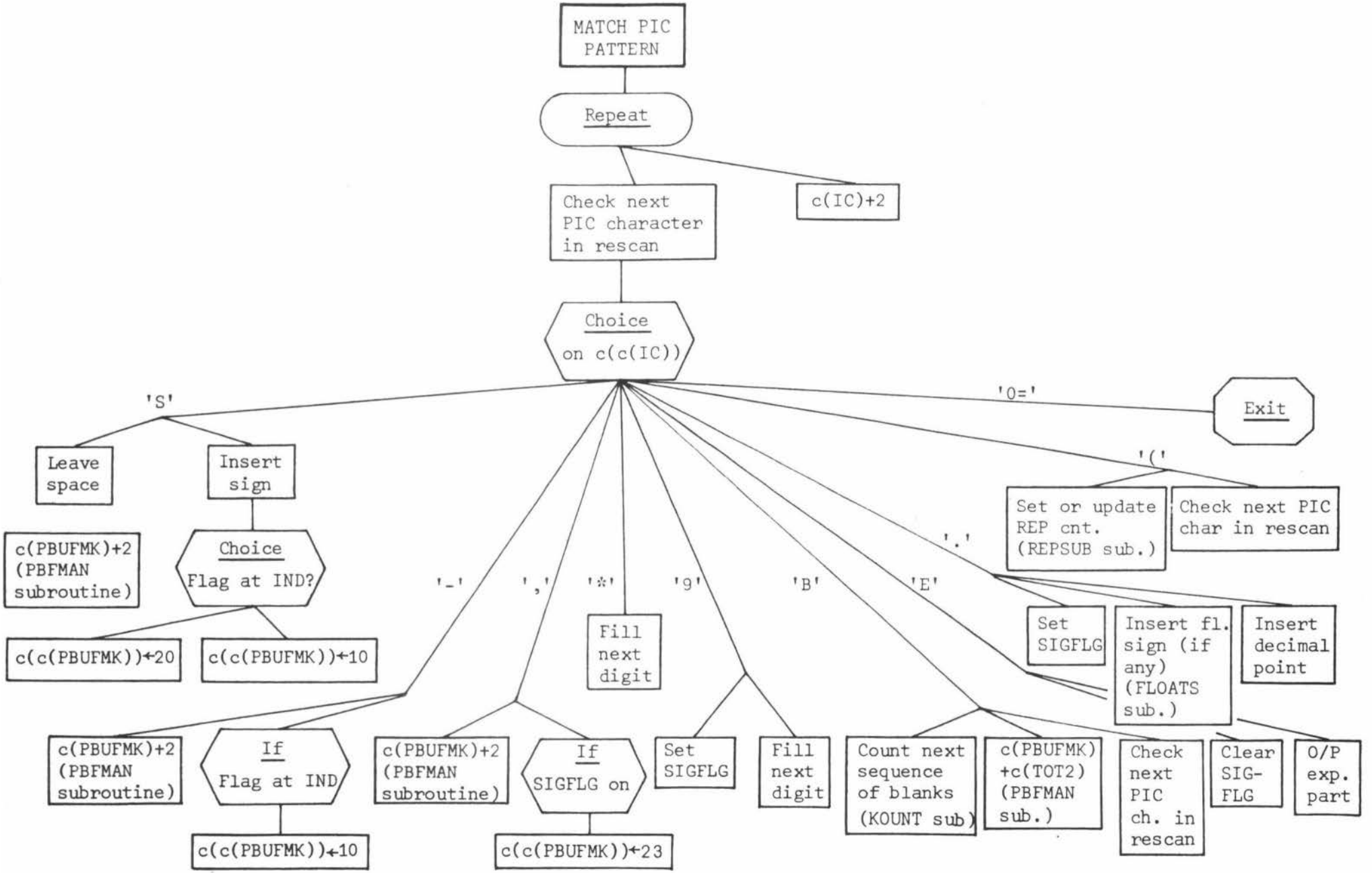

OUTPUT NUMBER IN F-FORMAT
Set c(LENGTH) to length of mantissa (LSUB subroutine)
Add exponent (to give digits before decimal point)
c(LENGTH)+c(c(SVP))
Check against BEFORE count
If
c(LENGTH)>c(BEFORE)
ERROR (PRC E3) Overflow - number too large for specified PICTURE
c(LENGTH)+c(AFTER)
c(BEFORE)+c(AFTER)
Choice
c(LENGTH)<0?
Yes
No
Assemble zero for output
c(LENGTH) ← 1
c(P) ← c(BOTV)
c(c(P)) ← 0
Round number (if necessary)

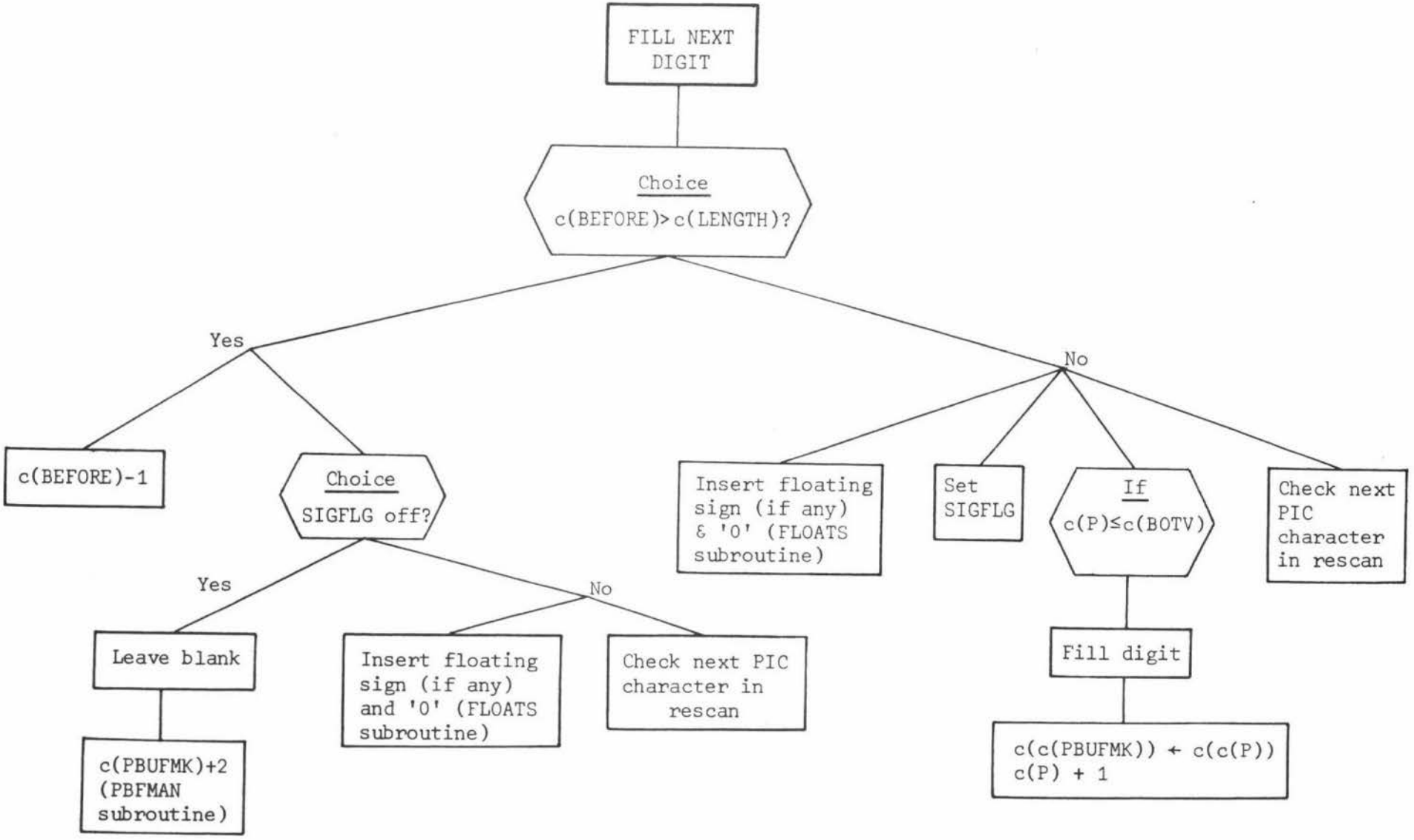

ROUND NUMBER
(IF NECESSARY)
If
c(c(SVP))+c(AFTER)<0
Set c(BOTV) to
"add 5" position
c(BOTV)+c(c(SVP))
+c(AFTER)+1
Add 5
Enlarge field
Clear flag at
c(P)
c(P) - 1
c(c(P)) ← -0
Add
c(c(BOTV))+5
c(BOTV) - 1
Choice
Digit at c(P)?
No
Yes
c(P)+1
If
c(BOTV)<c(P)
Assemble zero
for output
c(LENGTH)+1
If
c(LENGTH)>c(BEFORE)
ERROR (PRC E3)
Overflow - number
too large for PIC

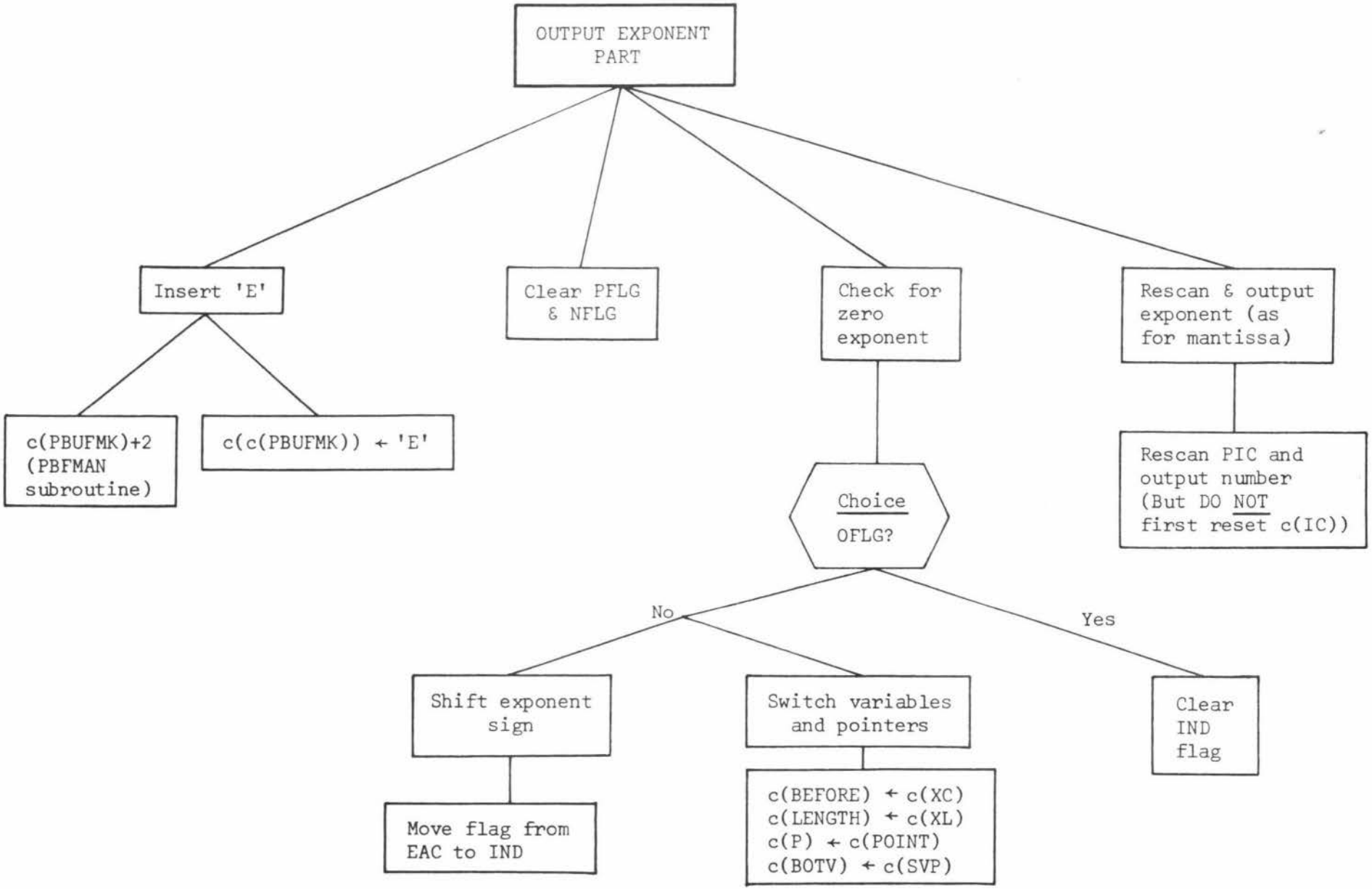

OUTPUT NUMBER IN E-FORMAT
Set OFLG
Set c(LENGTH) to length of mantissa (LSUB sub.)
Clear OFLG
Calculate exponent part
c(EAC) ← c(c(SVP)) +c(LENGTH)-c(BEFORE)
Set c(BEFORE) to total O/P mantissa length
c(BEFORE)+c(AFTER)
Round mantissa if necessary
If c(LENGTH)>c(BEFORE)
c(LENGTH) ← c(BEFORE)
Find exponent length
Set c(BOTV) to "add 5" position
c(BOTV)-c(LENGTH) +c(BEFORE)+1
Enlarge field
Clear flag at c(P) c(P) - 1 c(c(P)) ← -0
Add 5
c(c(BOTV))+5 c(BOTV) - 1
Choice Digit at c(P)?
No
c(P)+1
Yes
c(BOTV)-1 c(EAC)+1

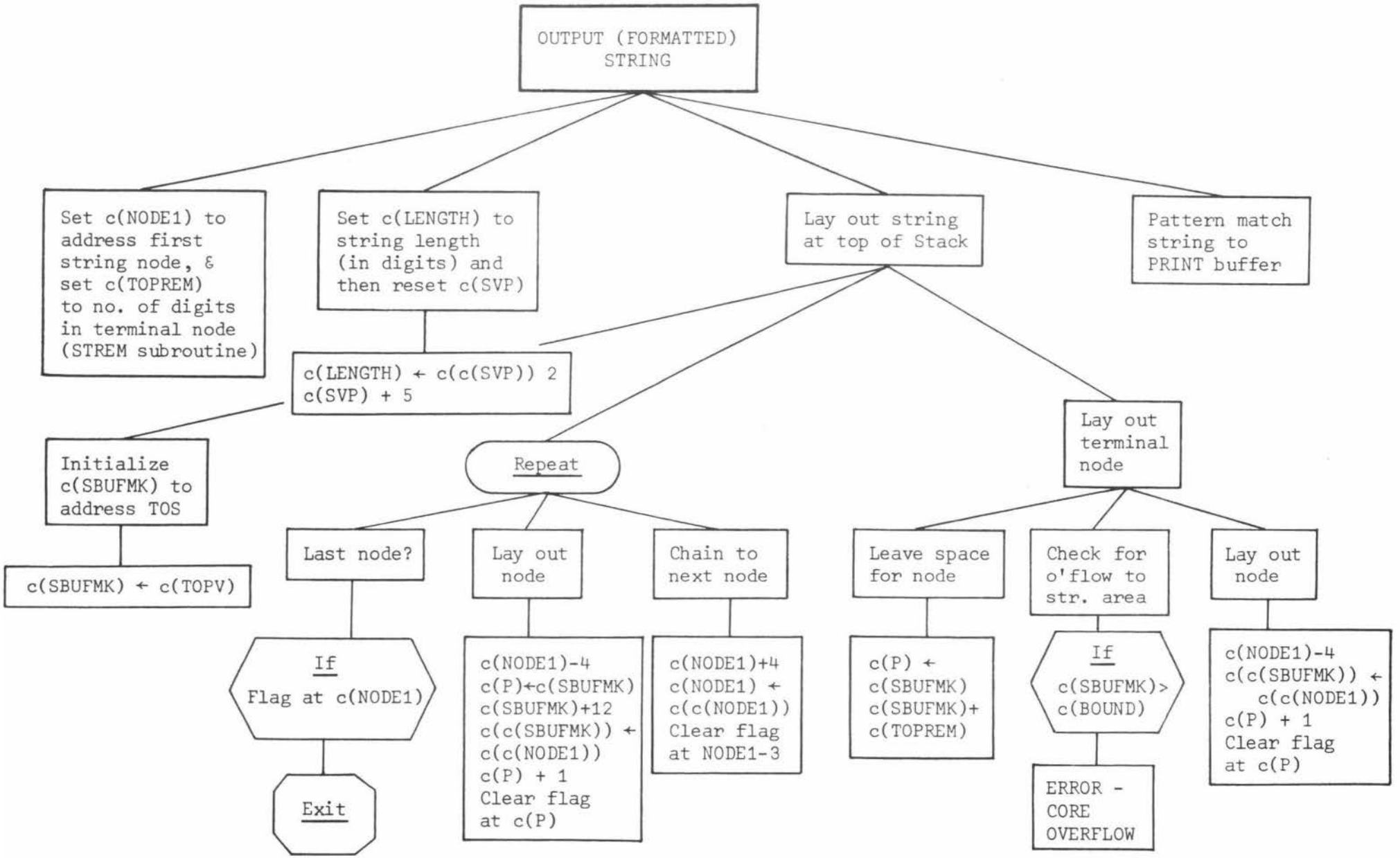

FIND EXPONENT LENGTH
Choice
c(EAC)=0?
Yes
No
Assemble zero exponent for output
c(XL) ← 1
c(POINT) ← c(SVP)
c(c(POINT)) ← 0
Set c(POINT) to address first significant digit of exponent
c(POINT) ← EAC-2
Repeat
If
Digit at c(POINT)
Exit
c(POINT)+1
Calculate exponent length
c(XL) ← EAC-c(POINT)+1
Check for overflow
If
c(XL)>c(XC)
ERROR (PRC E4) Overflow - exponent too large for specified PIC

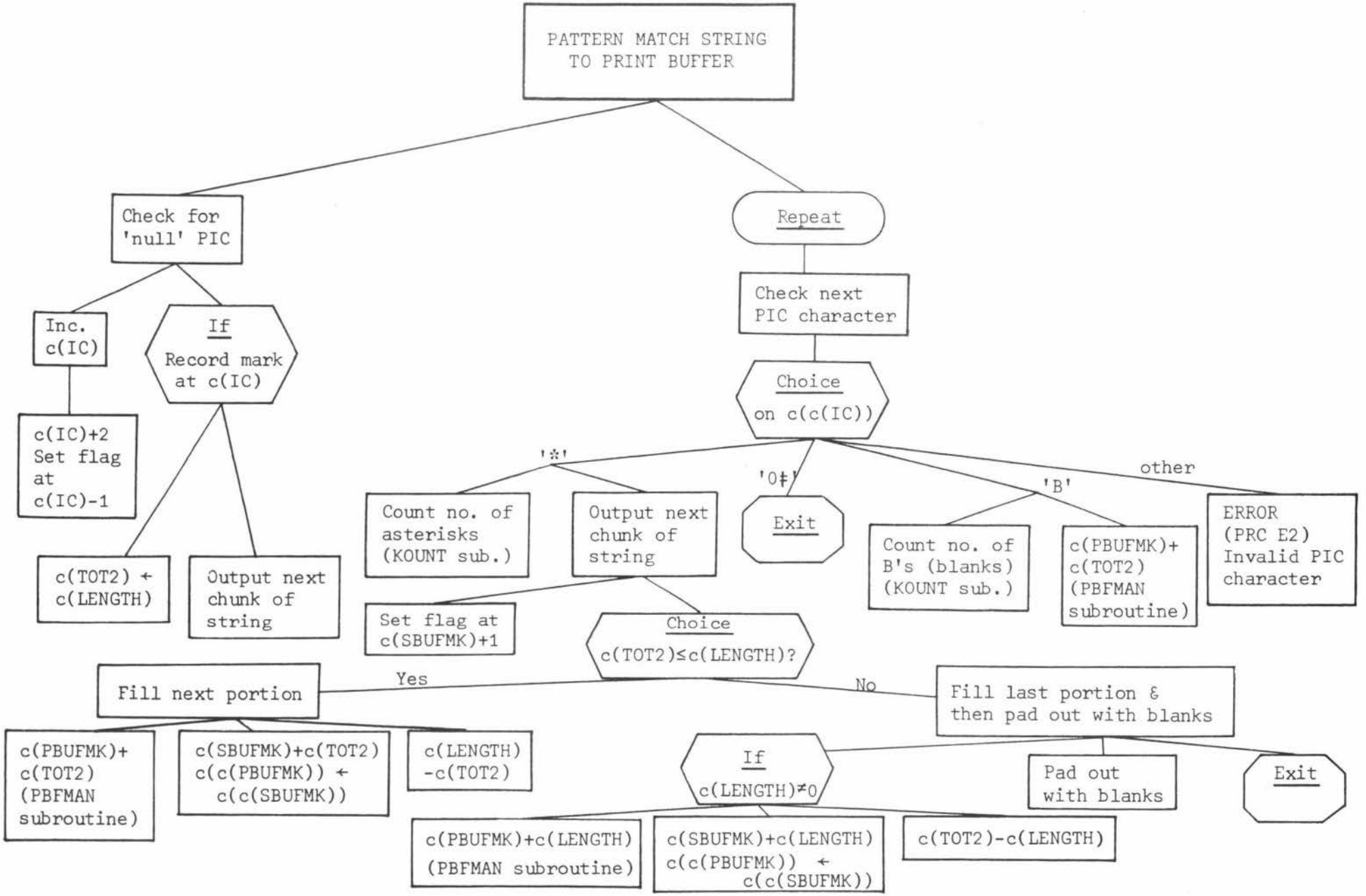

RESCAN PICTURE & OUTPUT NUMBER
Reset c(IC) to address first PIC character
Check first Character
Match PIC pattern
c(IC) ← c(ICM)+2
Choice
c(c(IC))='S'?
Yes
No
c(PBUFMK)+2 (PBFMAN subroutine) c(IC) + 2
Choice
Flag at IND?
If
c(c(IC))='-'
Yes
No
Set NFLG
Set PFLG
c(PBUFMK)+2 (PBFMAN subroutine) c(IC) + 2
If
Flag at IND
Set NFLG

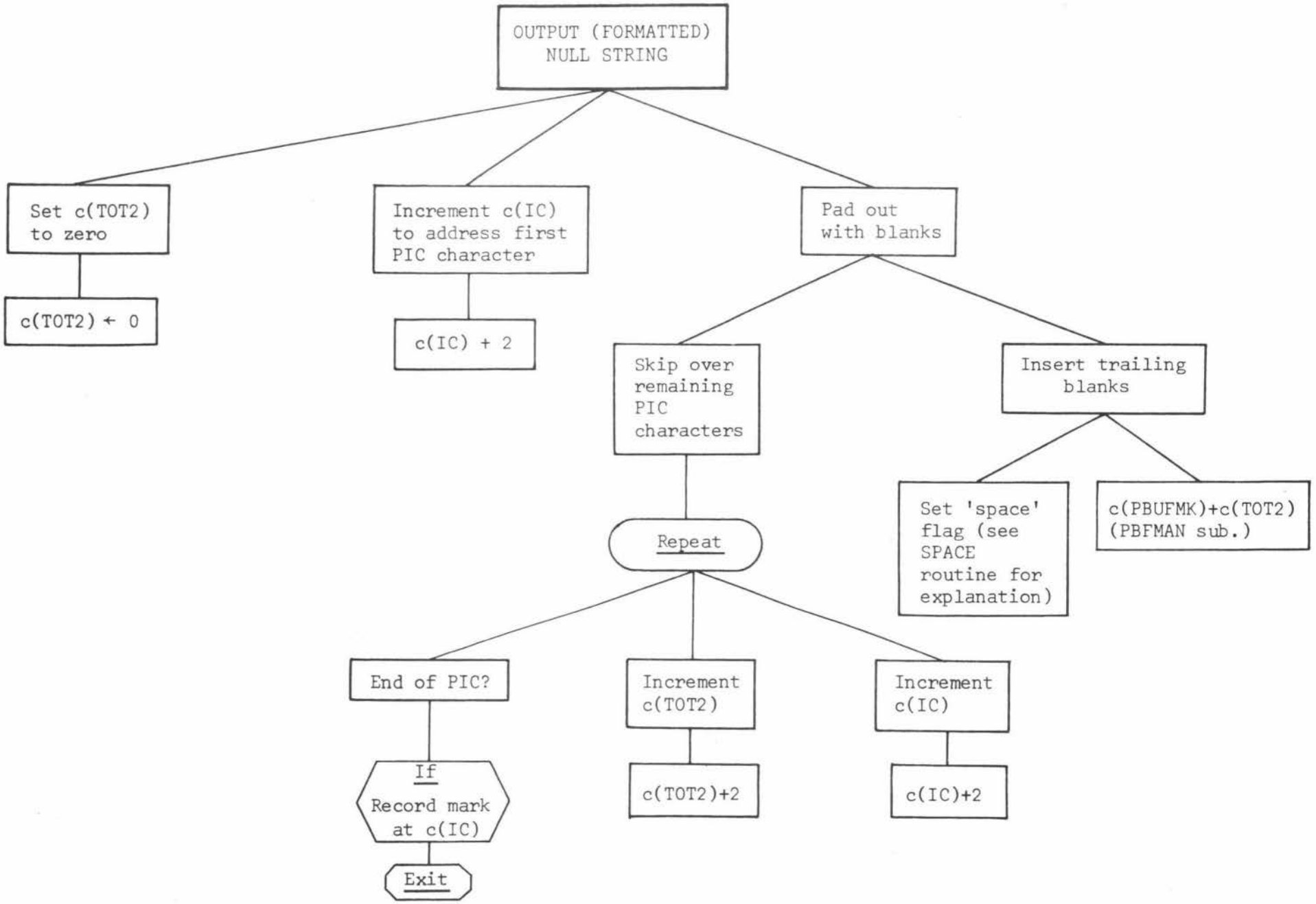

MATCH PIC PATTERN
Repeat
Check next PIC character in rescan
c(IC)+2
Choice on c(c(IC))
'S'
'0='
Exit
'('
Leave space
Insert sign
Set or update REP cnt. (REPSUB sub.)
Check next PIC char in rescan
c(PBUFMK)+2 (PBFMAN subroutine)
Choice Flag at IND?
'.'
'-'
','
'*'
'9'
'B'
'E'
Set SIGFLG
Insert fl. sign (if any) (FLOATS sub.)
Insert decimal point
Fill next digit
c(c(PBUFMK))←20
c(c(PBUFMK))←10
c(PBUFMK)+2 (PBFMAN subroutine)
If Flag at IND
c(PBUFMK)+2 (PBFMAN subroutine)
If SIGFLG on
Set SIGFLG
Fill next digit
Count next sequence of blanks (KOUNT sub)
c(PBUFMK) +c(TOT2) (PBFMAN sub.)
Check next PIC ch. in rescan
Clear SIG-FLG
O/P exp. part
c(c(PBUFMK))←10
c(c(PBUFMK))←23

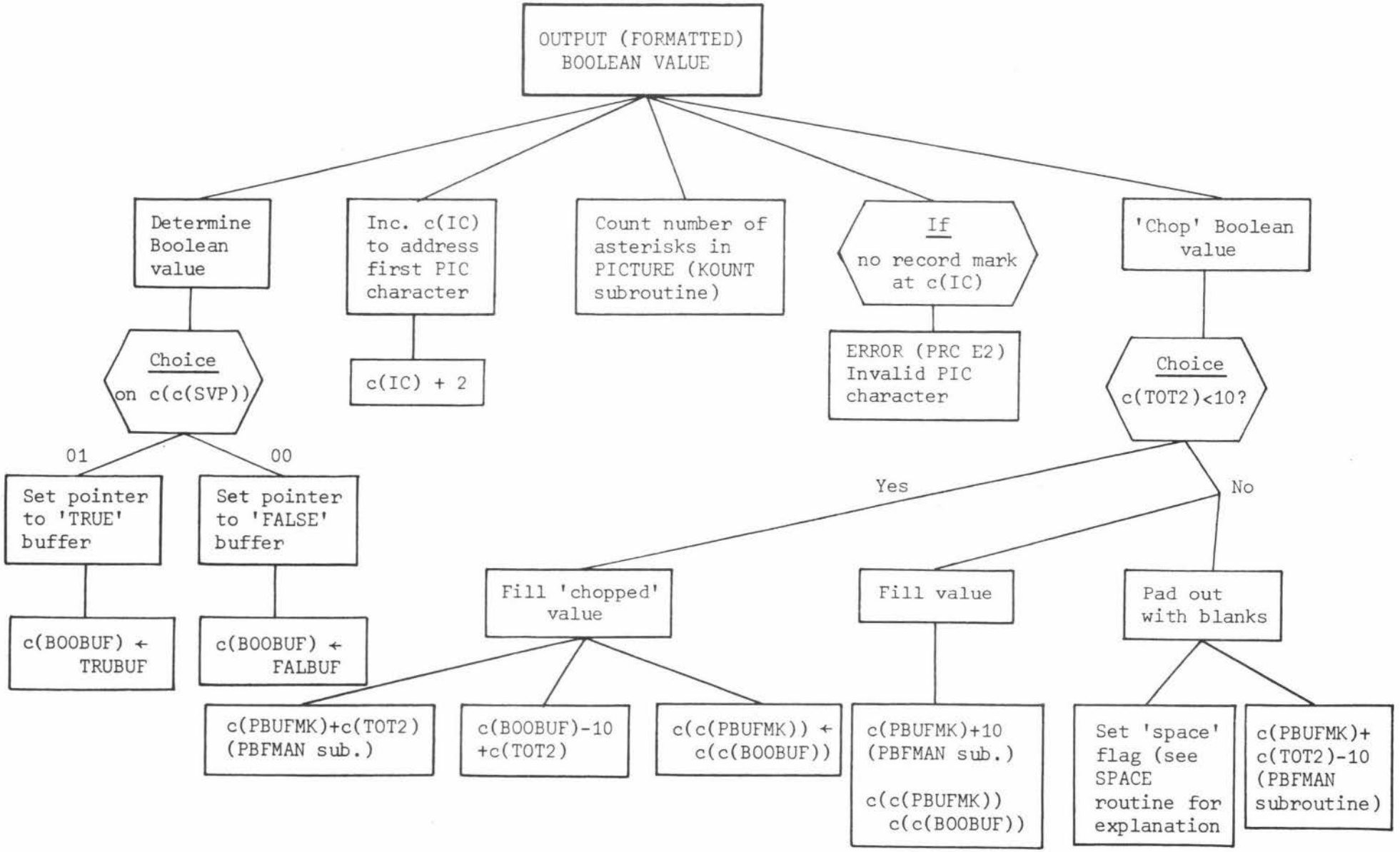

FILL NEXT DIGIT
Choice
c(BEFORE)>c(LENGTH)?
Yes
No
c(BEFORE)-1
Choice
SIGFLG off?
Yes
No
Leave blank
c(PBUFMK)+2 (PBFMAN subroutine)
Insert floating sign (if any) and '0' (FLOATS subroutine)
Check next PIC character in rescan
Insert floating sign (if any) & '0' (FLOATS subroutine)
Set SIGFLG
If
c(P)≤c(BOTV)
Check next PIC character in rescan
Fill digit
c(c(PBUFMK)) ← c(c(P))
c(P) + 1

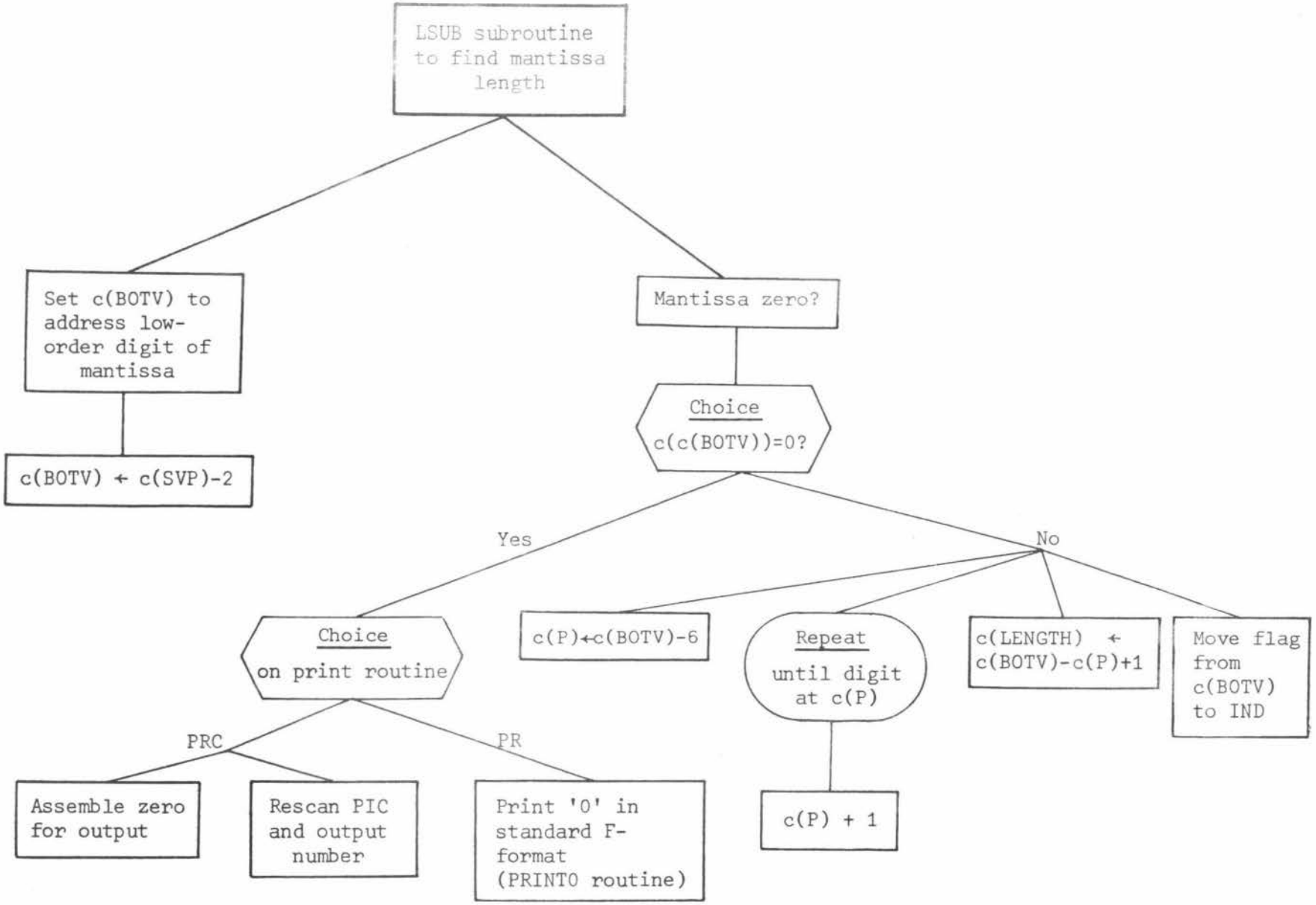

OUTPUT EXPONENT PART
Insert 'E'
Clear PFLG & NFLG
Check for zero exponent
Rescan & output exponent (as for mantissa)
c(PBUFMK)+2 (PBFMAN subroutine)
c(c(PBUFMK)) ← 'E'
Rescan PIC and output number (But DO NOT first reset c(IC))
Choice
OFLG?
No
Yes
Shift exponent sign
Switch variables and pointers
Clear IND flag
Move flag from EAC to IND
c(BEFORE) ← c(XC)
c(LENGTH) ← c(XL)
c(P) ← c(POINT)
c(BOTV) ← c(SVP)

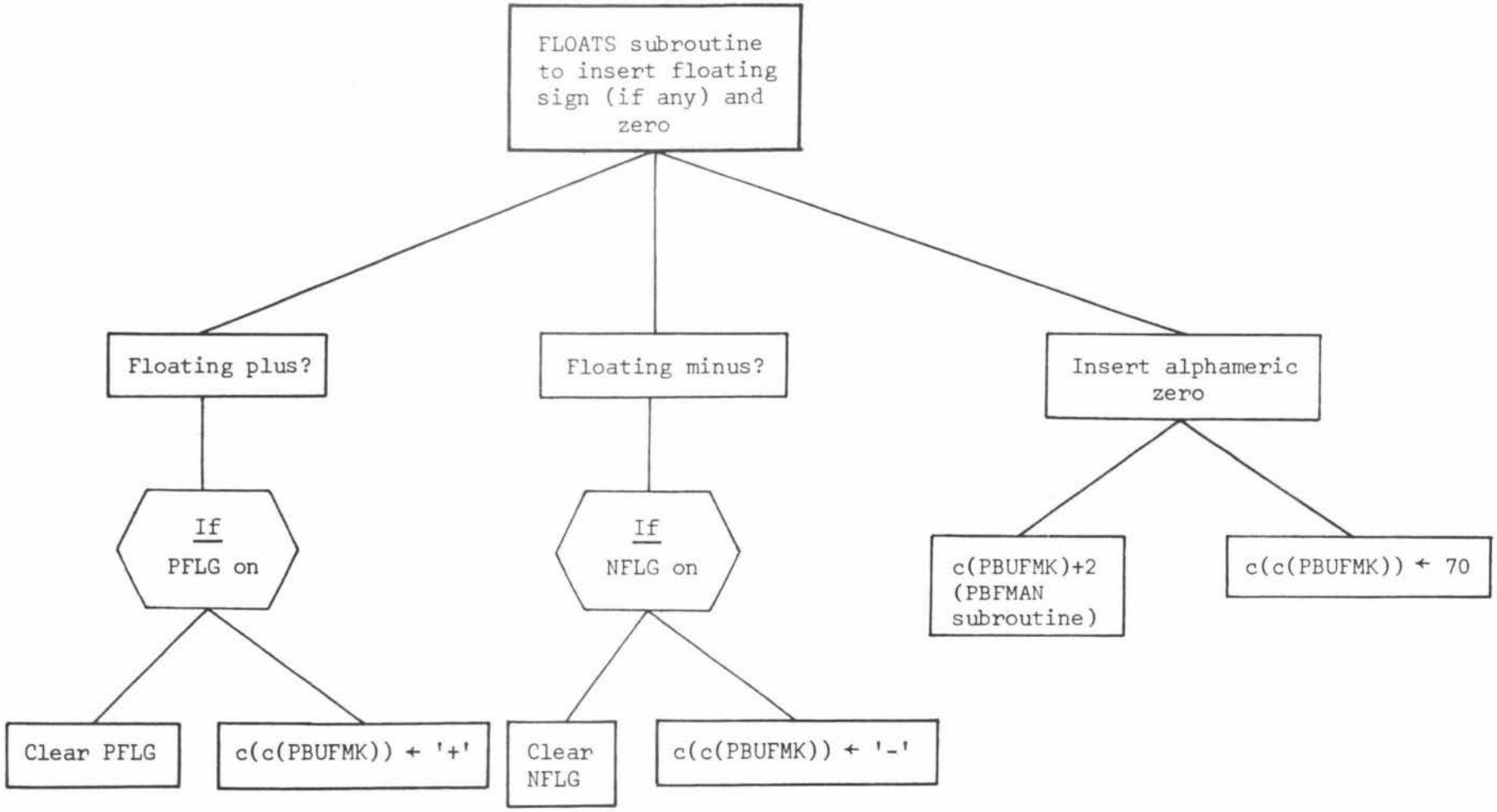

OUTPUT (FORMATTED) STRING
Set c(NODE1) to address first string node, & set c(TOPREM) to no. of digits in terminal node (STREM subroutine)
Set c(LENGTH) to string length (in digits) and then reset c(SVP)
Lay out string at top of Stack
Pattern match string to PRINT buffer
c(LENGTH) ← c(c(SVP)) 2
c(SVP) + 5
Initialize c(SBUFMK) to address TOS
c(SBUFMK) ← c(TOPV)
Repeat
Lay out terminal node
Last node?
Lay out node
Chain to next node
Leave space for node
Check for o'flow to str. area
Lay out node
If
Flag at c(NODE1)
Exit
c(NODE1)-4
c(P)←c(SBUFMK)
c(SBUFMK)+12
c(c(SBUFMK)) ← c(c(NODE1))
c(P) + 1
Clear flag at c(P)
c(NODE1)+4
c(NODE1) ← c(c(NODE1))
Clear flag at NODE1-3
c(P) ← c(SBUFMK)
c(SBUFMK)+ c(TOPREM)
If
c(SBUFMK)> c(BOUND)
ERROR - CORE OVERFLOW
c(NODE1)-4
c(c(SBUFMK)) ← c(c(NODE1))
c(P) + 1
Clear flag at c(P)

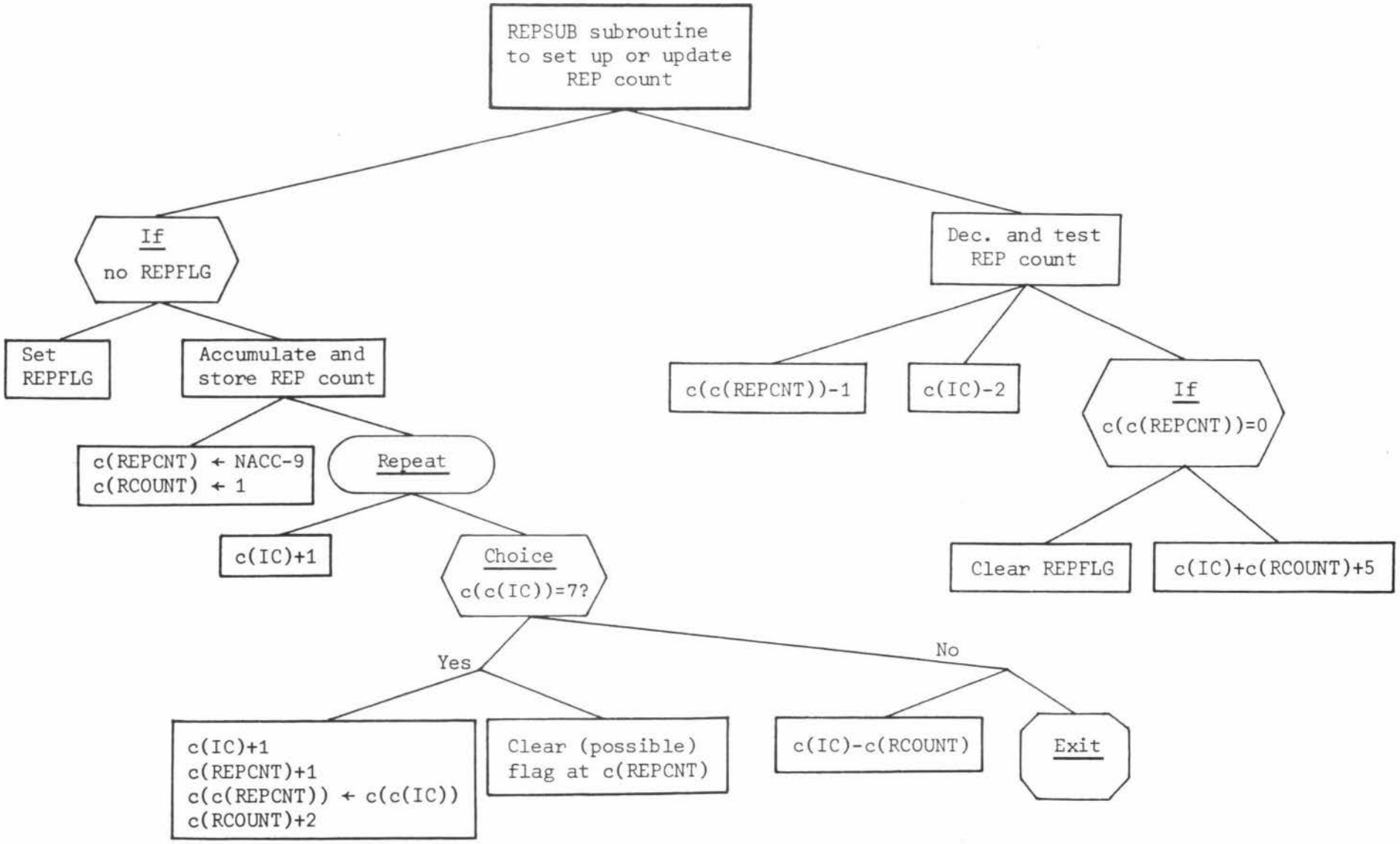

PATTERN MATCH STRING TO PRINT BUFFER
Check for 'null' PIC
Inc. c(IC)
c(IC)+2 Set flag at c(IC)-1
If Record mark at c(IC)
c(TOT2) ← c(LENGTH)
Output next chunk of string
Repeat
Check next PIC character
Choice on c(c(IC))
'*'
Count no. of asterisks (KOUNT sub.)
Output next chunk of string
Set flag at c(SBUFMK)+1
Choice c(TOT2)≤c(LENGTH)?
Yes
Fill next portion
c(PBUFMK)+ c(TOT2) (PBFMAN subroutine)
c(SBUFMK)+c(TOT2) c(c(PBUFMK)) ← c(c(SBUFMK))
c(LENGTH) -c(TOT2)
No
Fill last portion & then pad out with blanks
If c(LENGTH)≠0
c(PBUFMK)+c(LENGTH) (PBFMAN subroutine)
c(SBUFMK)+c(LENGTH) c(c(PBUFMK)) ← c(c(SBUFMK))
c(TOT2)-c(LENGTH)
Pad out with blanks
Exit
'0≠'
Exit
'B'
Count no. of B's (blanks) (KOUNT sub.)
c(PBUFMK)+ c(TOT2) (PBFMAN subroutine)
other
ERROR (PRC E2) Invalid PIC character

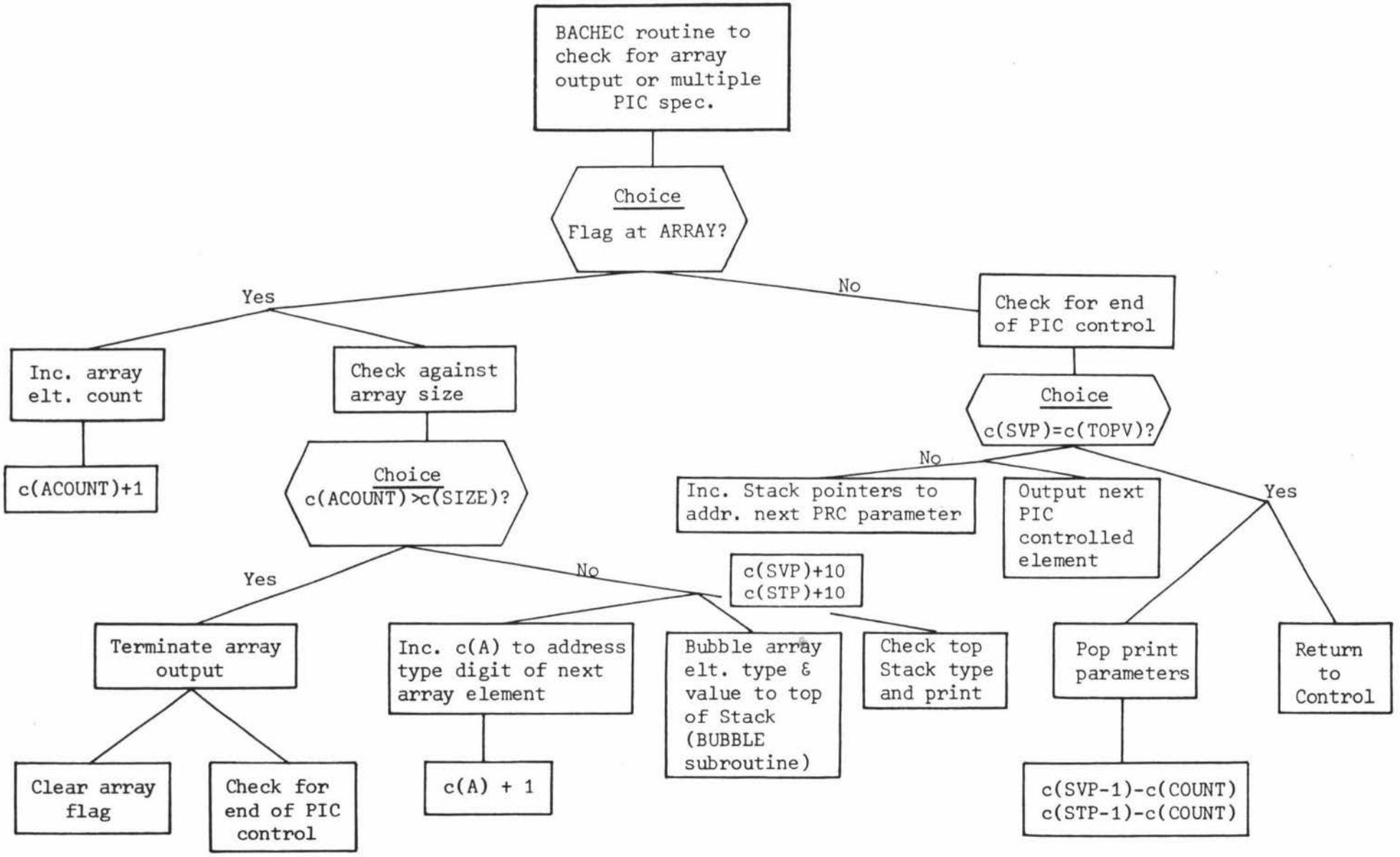

OUTPUT (FORMATTED) NULL STRING
Set c(TOT2) to zero
c(TOT2) ← 0
Increment c(IC) to address first PIC character
c(IC) + 2
Pad out with blanks
Skip over remaining PIC characters
Repeat
End of PIC?
If Record mark at c(IC)
Exit
Increment c(TOT2)
c(TOT2)+2
Increment c(IC)
c(IC)+2
Insert trailing blanks
Set 'space' flag (see SPACE routine for explanation)
c(PBUFMK)+c(TOT2) (PBFMAN sub.)

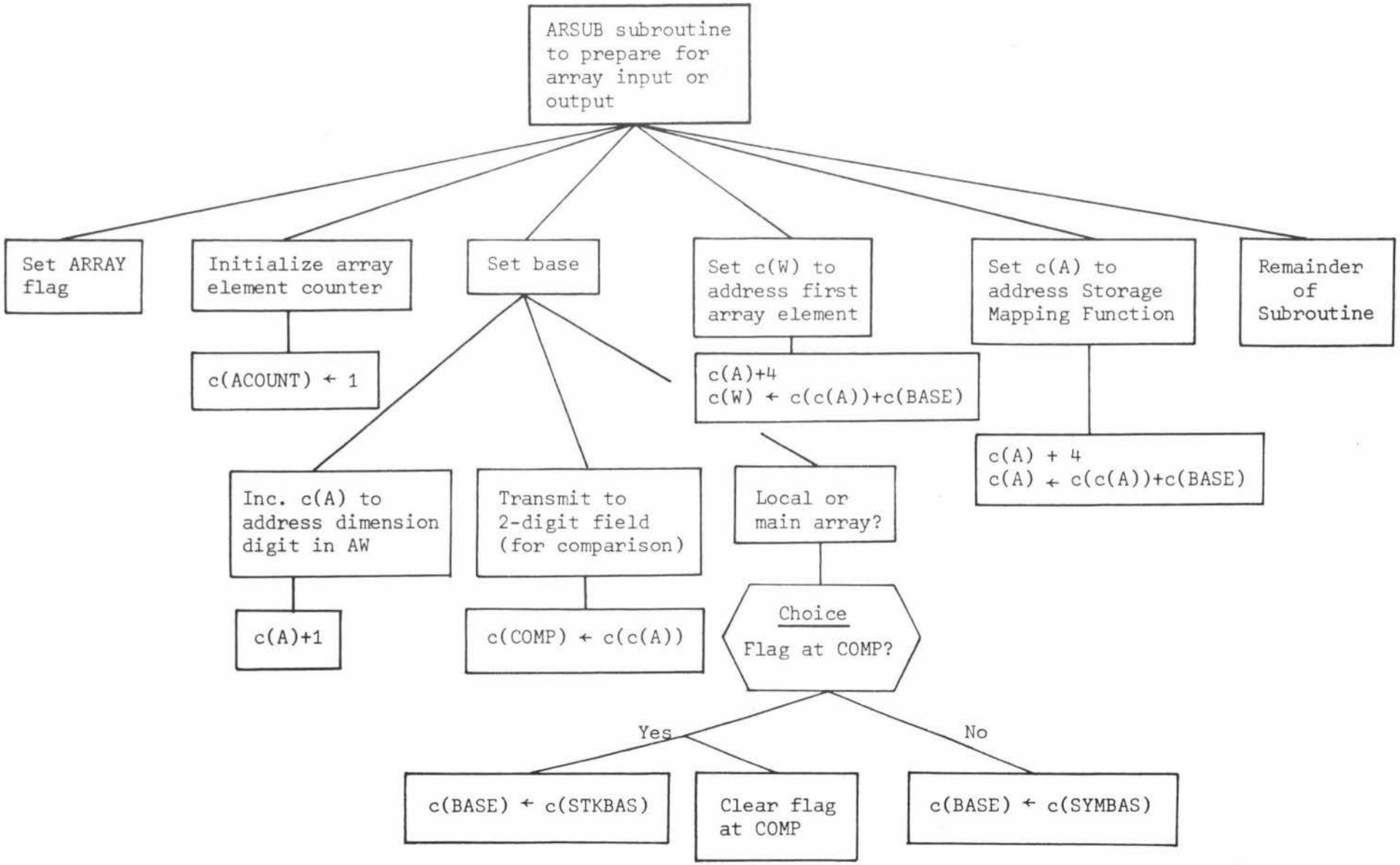

OUTPUT (FORMATTED) BOOLEAN VALUE
Determine Boolean value
Inc. c(IC) to address first PIC character
Count number of asterisks in PICTURE (KOUNT subroutine)
If no record mark at c(IC)
'Chop' Boolean value
Choice on c(c(SVP))
c(IC) + 2
ERROR (PRC E2) Invalid PIC character
Choice c(TOT2)<10?
01
00
Yes
No
Set pointer to 'TRUE' buffer
Set pointer to 'FALSE' buffer
Fill 'chopped' value
Fill value
Pad out with blanks
c(BOOBUF) ← TRUBUF
c(BOOBUF) ← FALBUF
c(PBUFMK)+c(TOT2) (PBFMAN sub.)
c(BOOBUF)-10 +c(TOT2)
c(c(PBUFMK)) ← c(c(BOOBUF))
c(PBUFMK)+10 (PBFMAN sub.)
c(c(PBUFMK)) c(c(BOOBUF))
Set 'space' flag (see SPACE routine for explanation
c(PBUFMK)+ c(TOT2)-10 (PBFMAN subroutine)

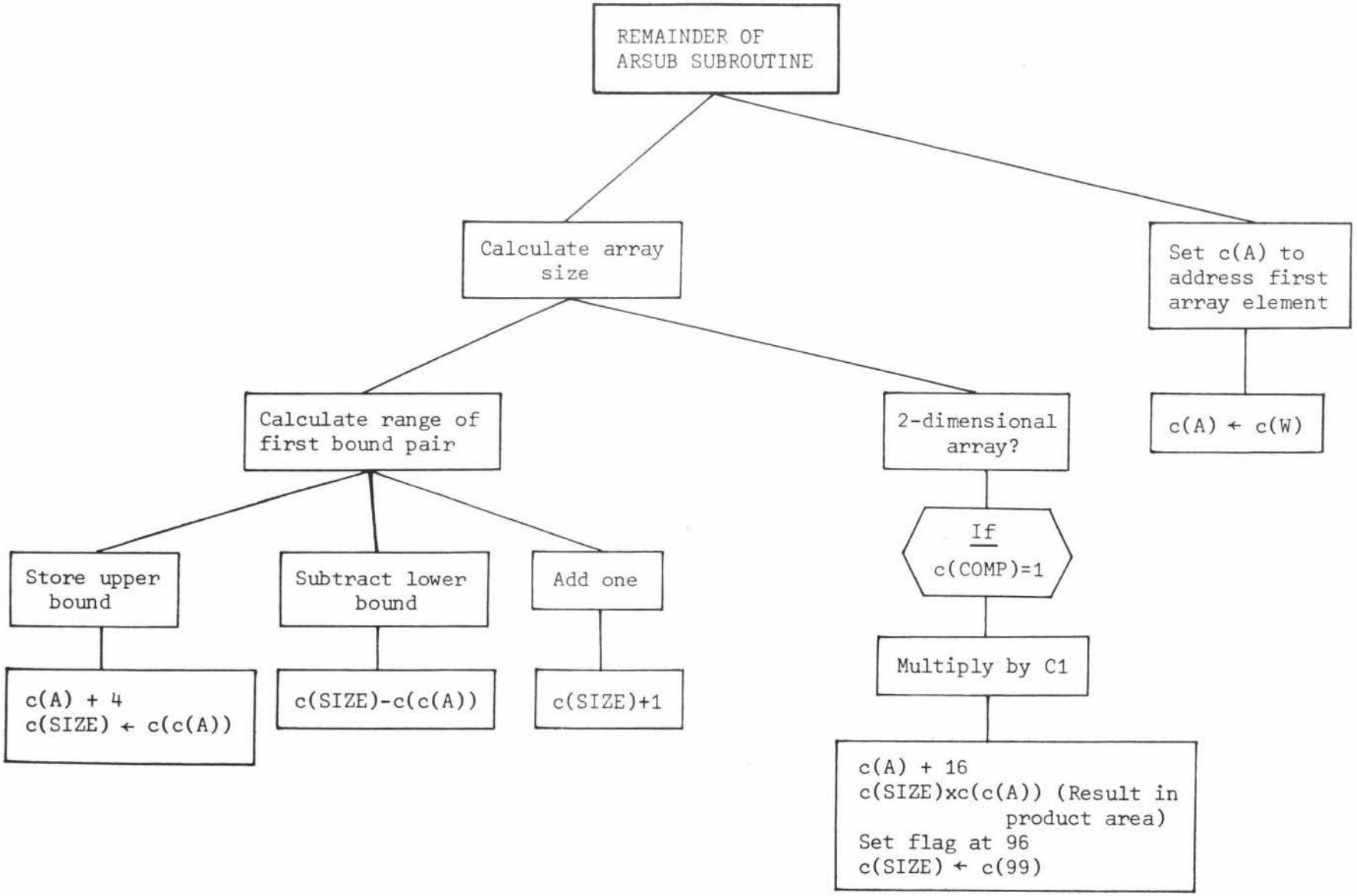

LSUB subroutine to find mantissa length
Set c(BOTV) to address low-order digit of mantissa
c(BOTV) ← c(SVP)-2
Mantissa zero?
Choice
c(c(BOTV))=0?
Yes
No
Choice
on print routine
PRC
PR
Assemble zero for output
Rescan PIC and output number
Print '0' in standard F-format (PRINT0 routine)
c(P)←c(BOTV)-6
Repeat
until digit at c(P)
c(P) + 1
c(LENGTH) ← c(BOTV)-c(P)+1
Move flag from c(BOTV) to IND

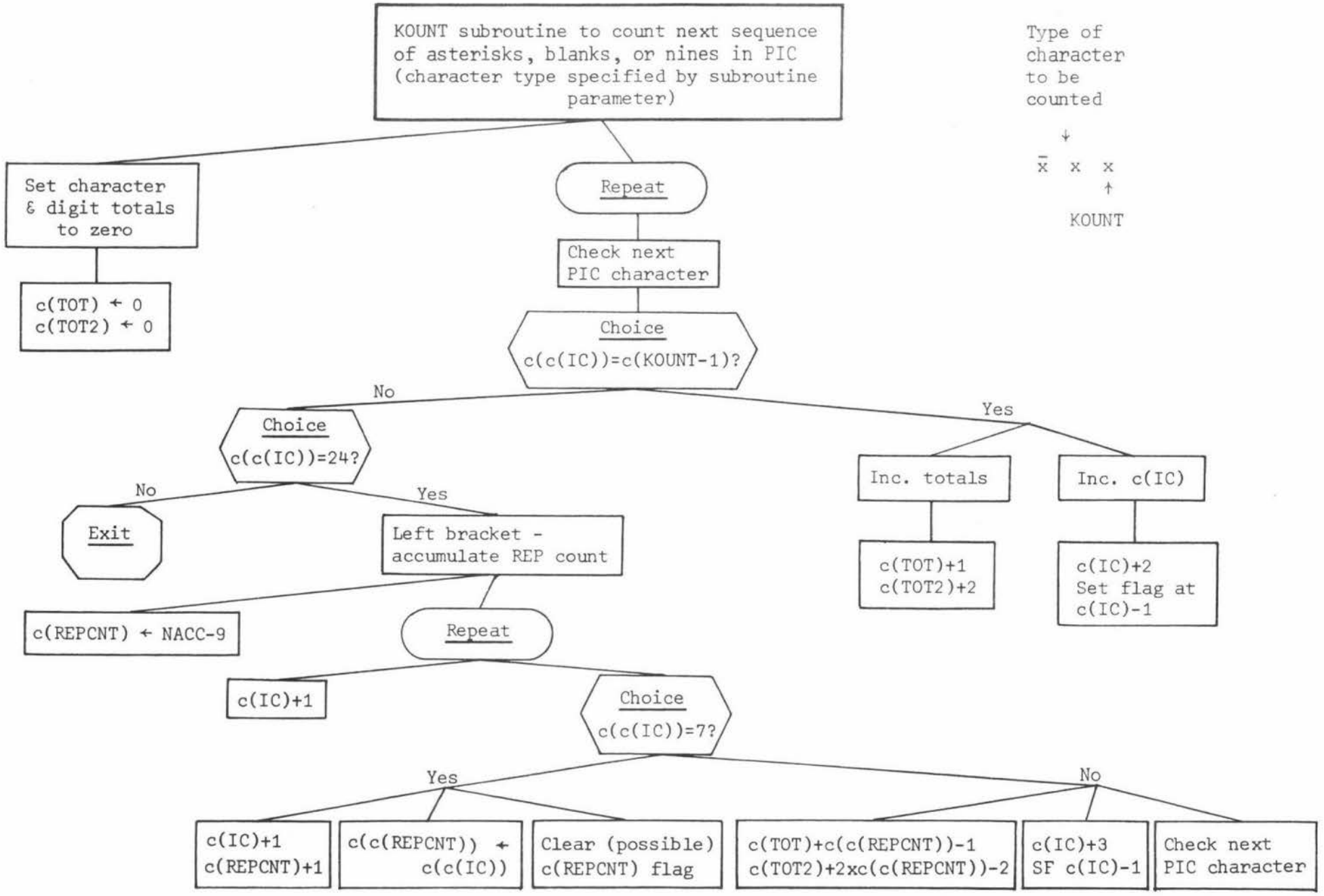

FLOATS subroutine to insert floating sign (if any) and zero
Floating plus?
Floating minus?
Insert alphameric zero
If PFLG on
If NFLG on
c(PBUFMK)+2 (PBFMAN subroutine)
c(c(PBUFMK)) ← 70
Clear PFLG
c(c(PBUFMK)) ← '+'
Clear NFLG
c(c(PBUFMK)) ← '-'


REPSUB subroutine to set up or update REP count
If no REPFLG
Set REPFLG
Accumulate and store REP count
c(REPCNT) ← NACC-9
c(RCOUNT) ← 1
Repeat
c(IC)+1
Choice c(c(IC))=7?
Yes
No
c(IC)+1
c(REPCNT)+1
c(c(REPCNT)) ← c(c(IC))
c(RCOUNT)+2
Clear (possible) flag at c(REPCNT)
c(IC)-c(RCOUNT)
Exit
Dec. and test REP count
c(c(REPCNT))-1
c(IC)-2
If c(c(REPCNT))=0
Clear REPFLG
c(IC)+c(RCOUNT)+5


BACHEC routine to check for array output or multiple PIC spec.
Choice
Flag at ARRAY?
Yes
No
Inc. array elt. count
c(ACOUNT)+1
Check against array size
Choice
c(ACOUNT)>c(SIZE)?
Yes
No
Terminate array output
Clear array flag
Check for end of PIC control
Inc. c(A) to address type digit of next array element
c(A) + 1
c(SVP)+10
c(STP)+10
Bubble array elt. type & value to top of Stack (BUBBLE subroutine)
Check top Stack type and print
Check for end of PIC control
Choice
c(SVP)=c(TOPV)?
No
Yes
Inc. Stack pointers to addr. next PRC parameter
Output next PIC controlled element
Pop print parameters
c(SVP-1)-c(COUNT)
c(STP-1)-c(COUNT)
Return to Control


ARSUB subroutine to prepare for array input or output
Set ARRAY flag
Initialize array element counter
c(ACOUNT) ← 1
Set base
Inc. c(A) to address dimension digit in AW
c(A)+1
Transmit to 2-digit field (for comparison)
c(COMP) ← c(c(A))
Set c(W) to address first array element
c(A)+4
c(W) ← c(c(A))+c(BASE)
Local or main array?
Choice
Flag at COMP?
Yes
No
c(BASE) ← c(STKBAS)
Clear flag at COMP
c(BASE) ← c(SYMBAS)
Set c(A) to address Storage Mapping Function
c(A) + 4
c(A) ← c(c(A))+c(BASE)
Remainder of Subroutine


REMAINDER OF ARSUB SUBROUTINE
Calculate array size
Set c(A) to address first array element
Calculate range of first bound pair
2-dimensional array?
c(A) ← c(W)
Store upper bound
Subtract lower bound
Add one
If c(COMP)=1
c(A) + 4
c(SIZE) ← c(c(A))
c(SIZE)-c(c(A))
c(SIZE)+1
Multiply by C1
c(A) + 16
c(SIZE)xc(c(A)) (Result in product area)
Set flag at 96
c(SIZE) ← c(99)

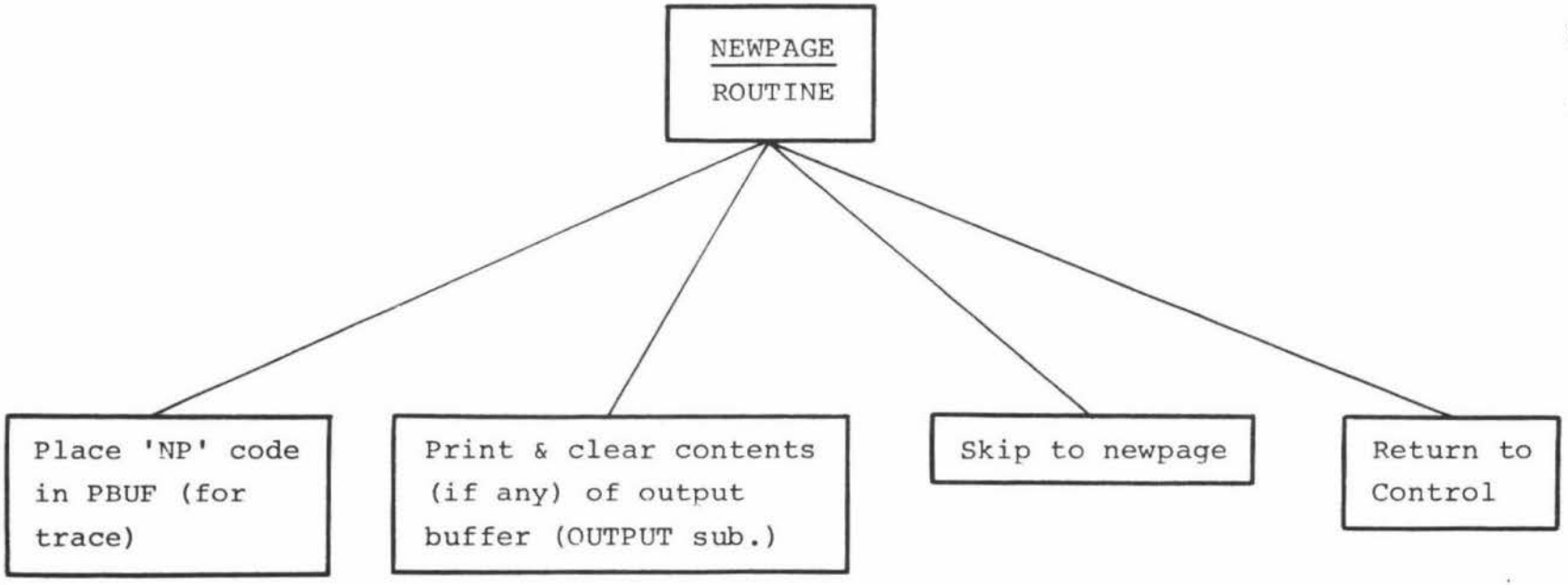

KOUNT subroutine to count next sequence of asterisks, blanks, or nines in PIC (character type specified by subroutine parameter)
Type of character to be counted
x̄ x x
KOUNT
Set character & digit totals to zero
c(TOT) ← 0
c(TOT2) ← 0
Repeat
Check next PIC character
Choice
c(c(IC))=c(KOUNT-1)?
No
Yes
Choice
c(c(IC))=24?
No
Exit
Yes
Left bracket - accumulate REP count
c(REPCNT) ← NACC-9
Repeat
c(IC)+1
Choice
c(c(IC))=7?
Yes
c(IC)+1
c(REPCNT)+1
c(c(REPCNT)) ← c(c(IC))
Clear (possible) c(REPCNT) flag
No
c(TOT)+c(c(REPCNT))-1
c(TOT2)+2xc(c(REPCNT))-2
c(IC)+3
SF c(IC)-1
Check next PIC character
Inc. totals
c(TOT)+1
c(TOT2)+2
Inc. c(IC)
c(IC)+2
Set flag at c(IC)-1

OP 39

opcode

N P

c(IC)



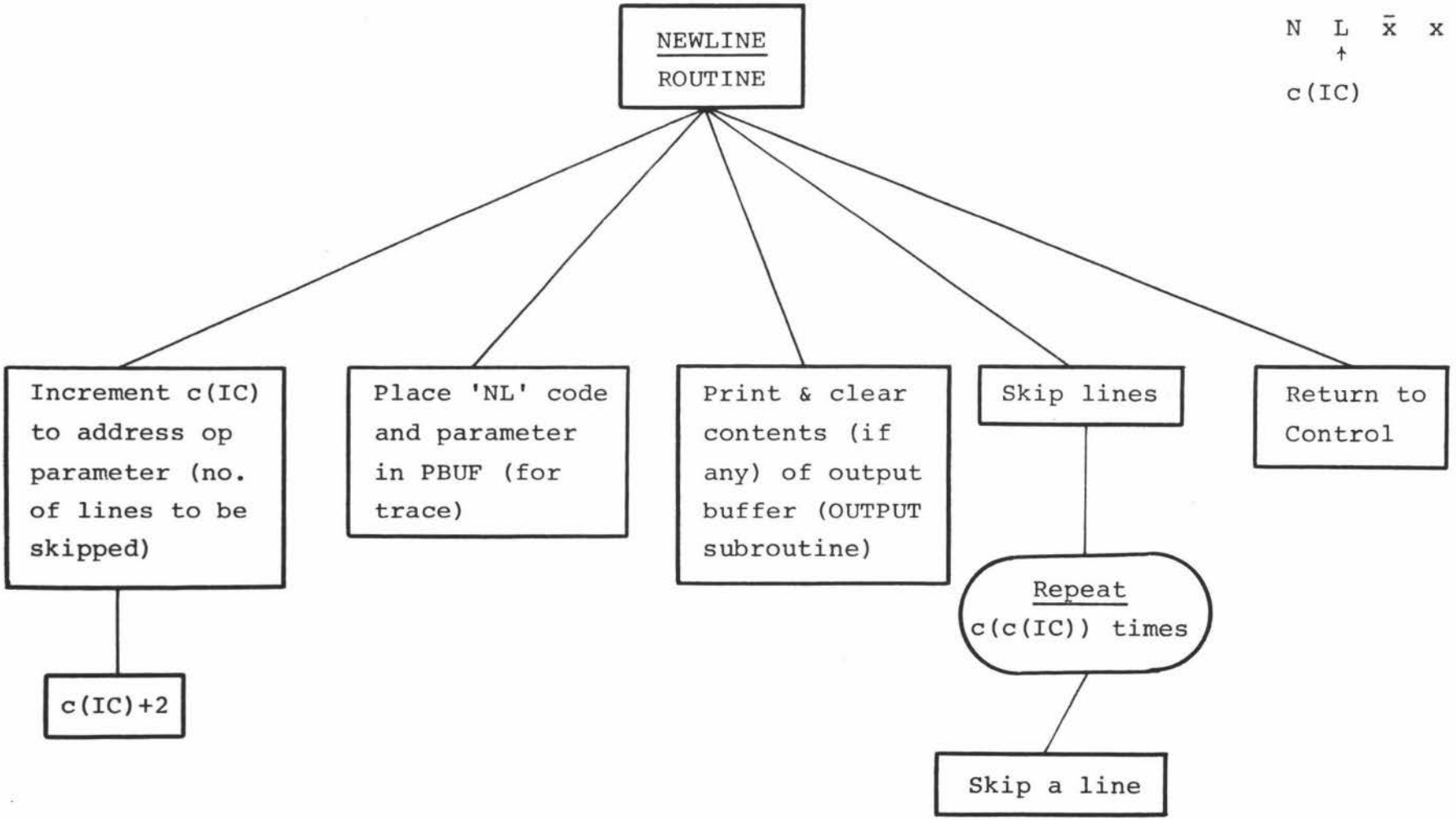
OP 40

opcode parameter

N L $\bar{x}$ x

↑

c(IC)

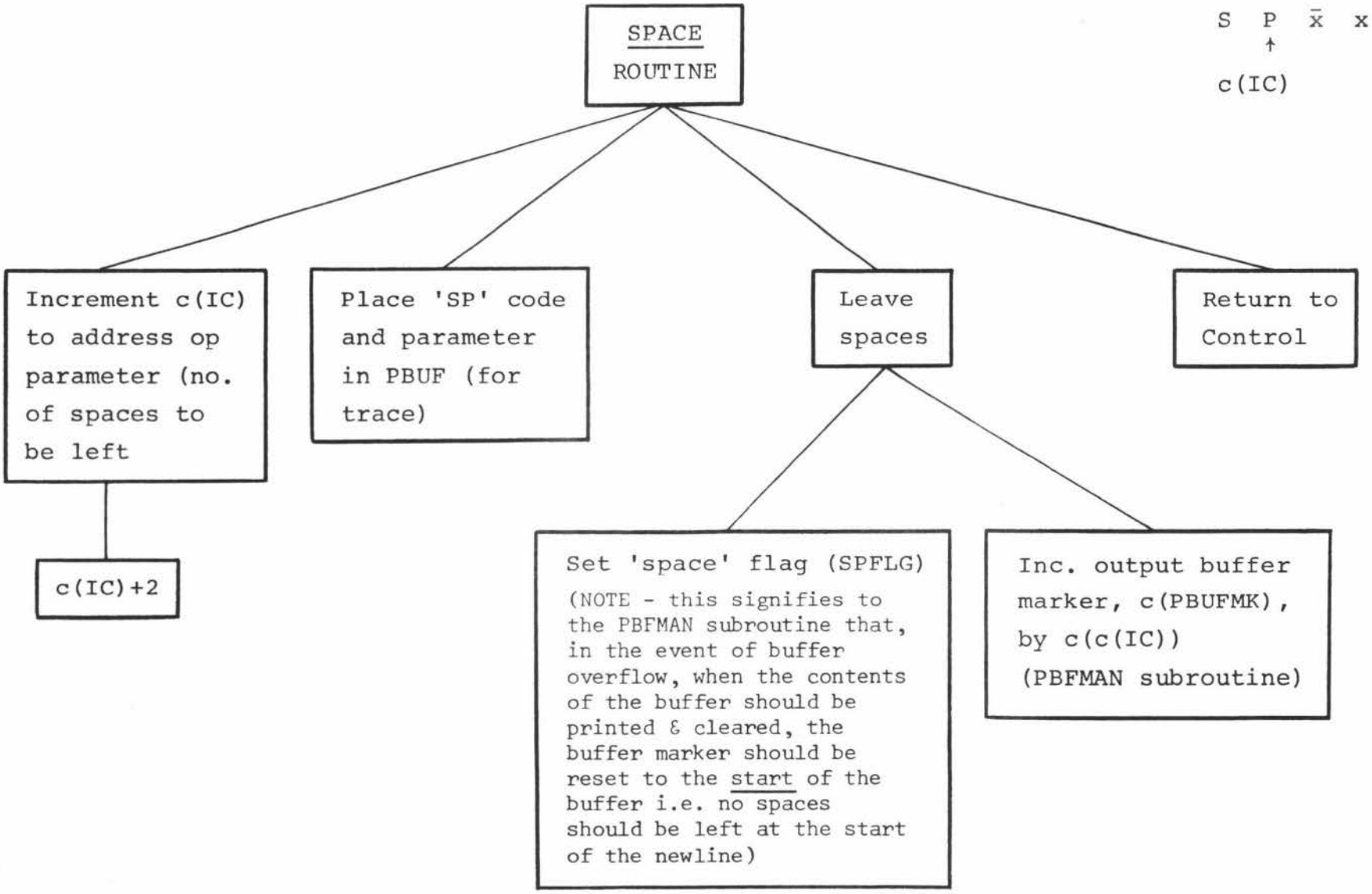


OP 41

opcode parameter

S P $\bar{x}$ x

c(IC)



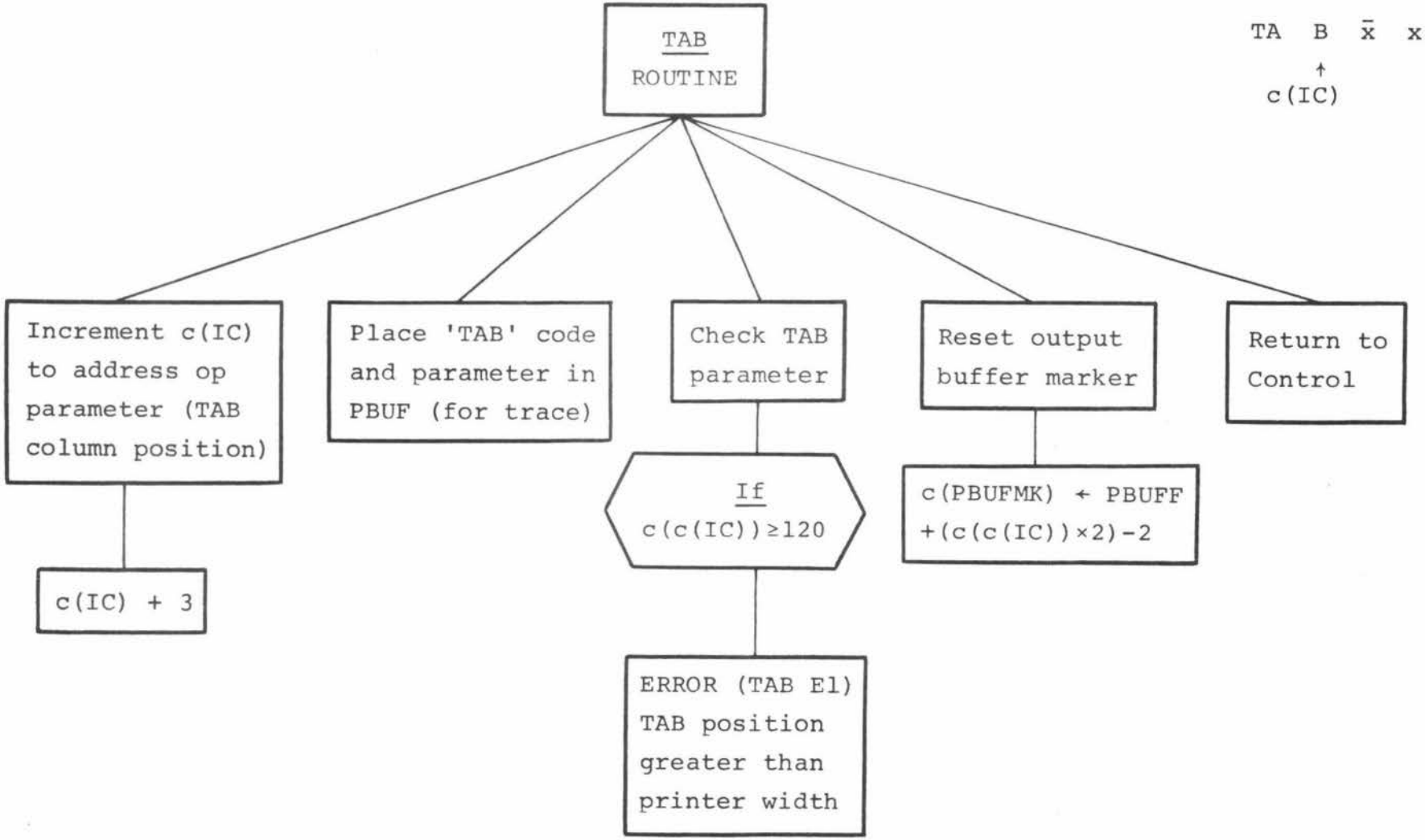
OP 42

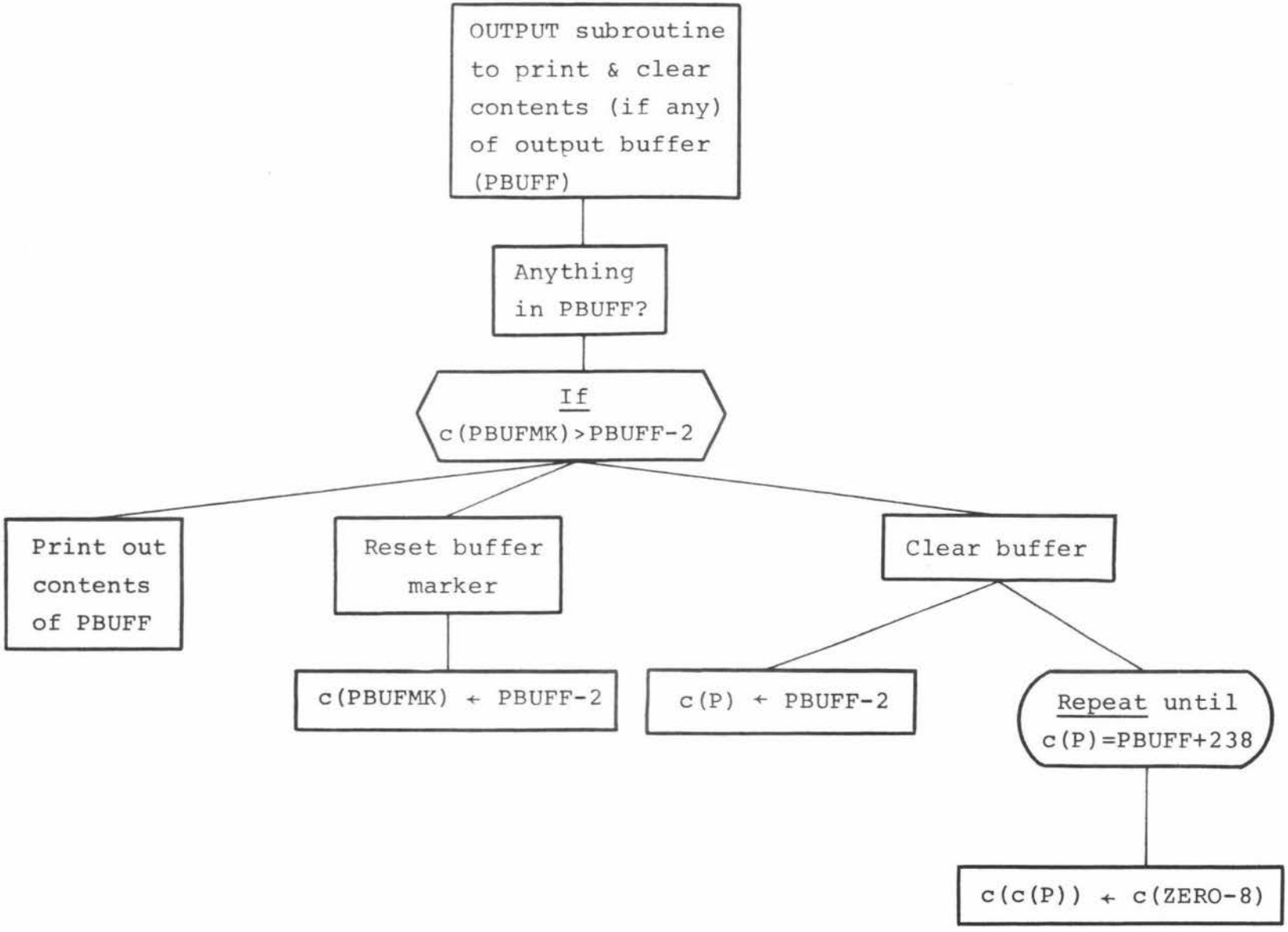
opcode parameter

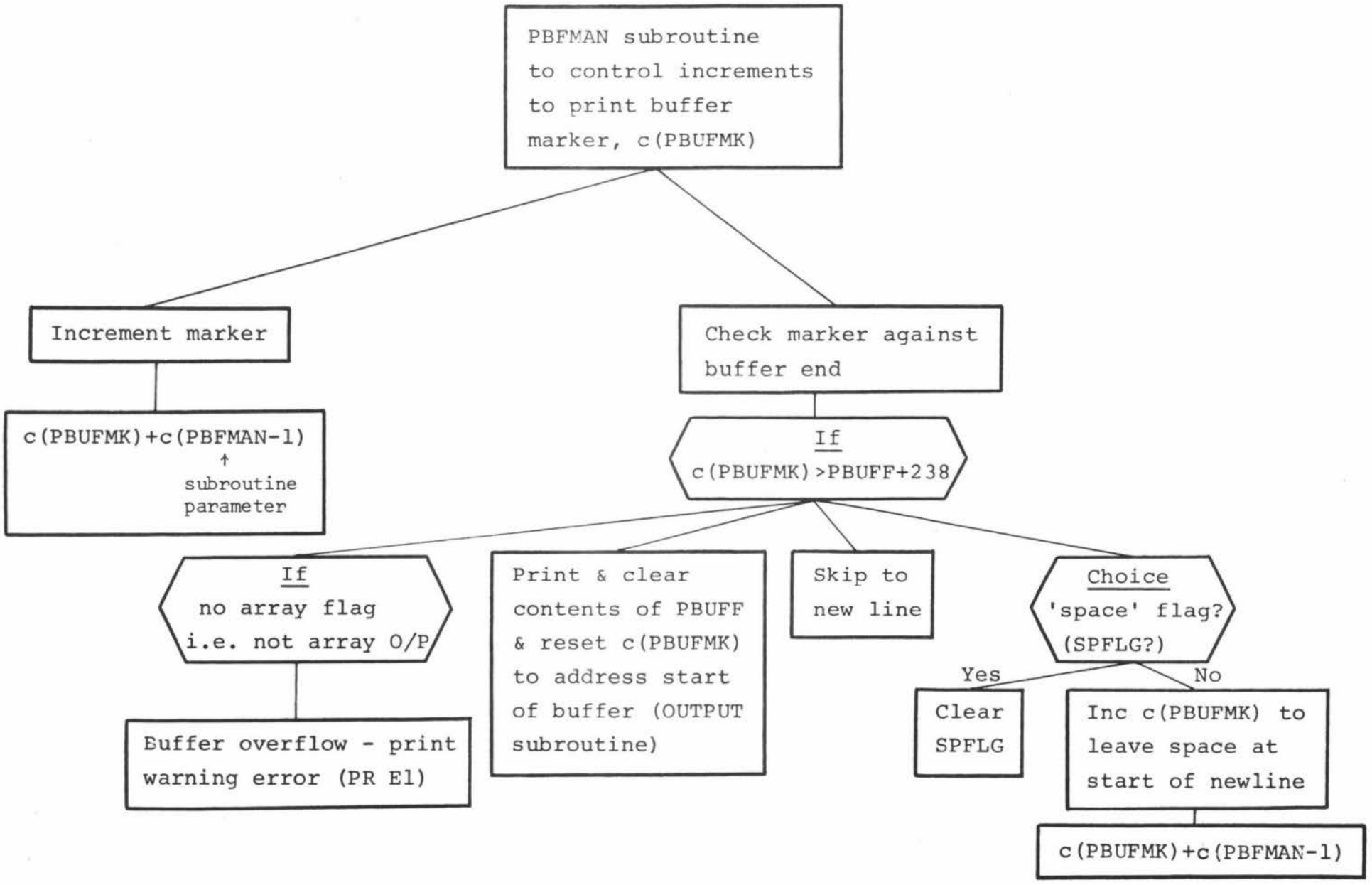
TA B $\bar{x}$ x x

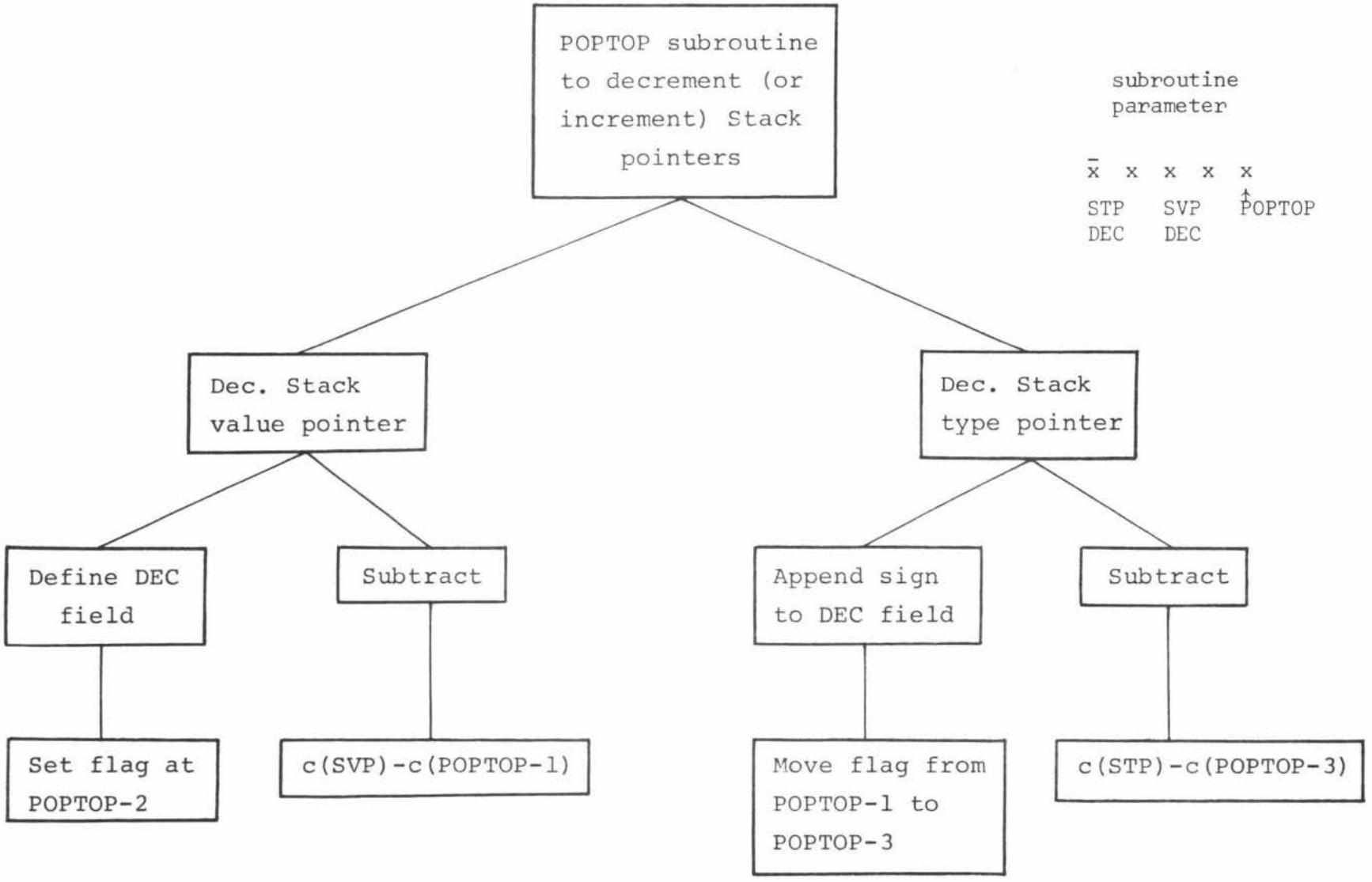
↑

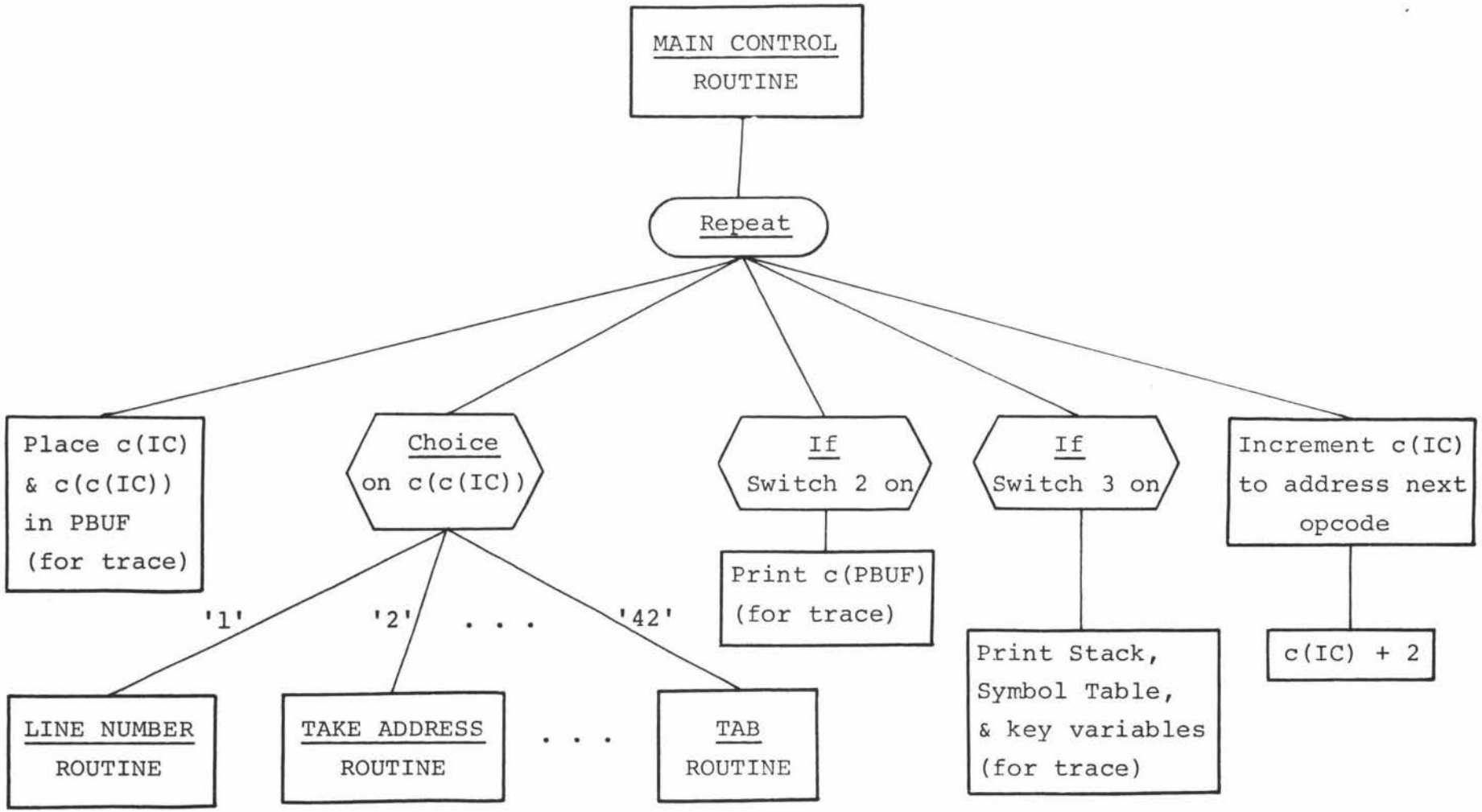
c(IC)

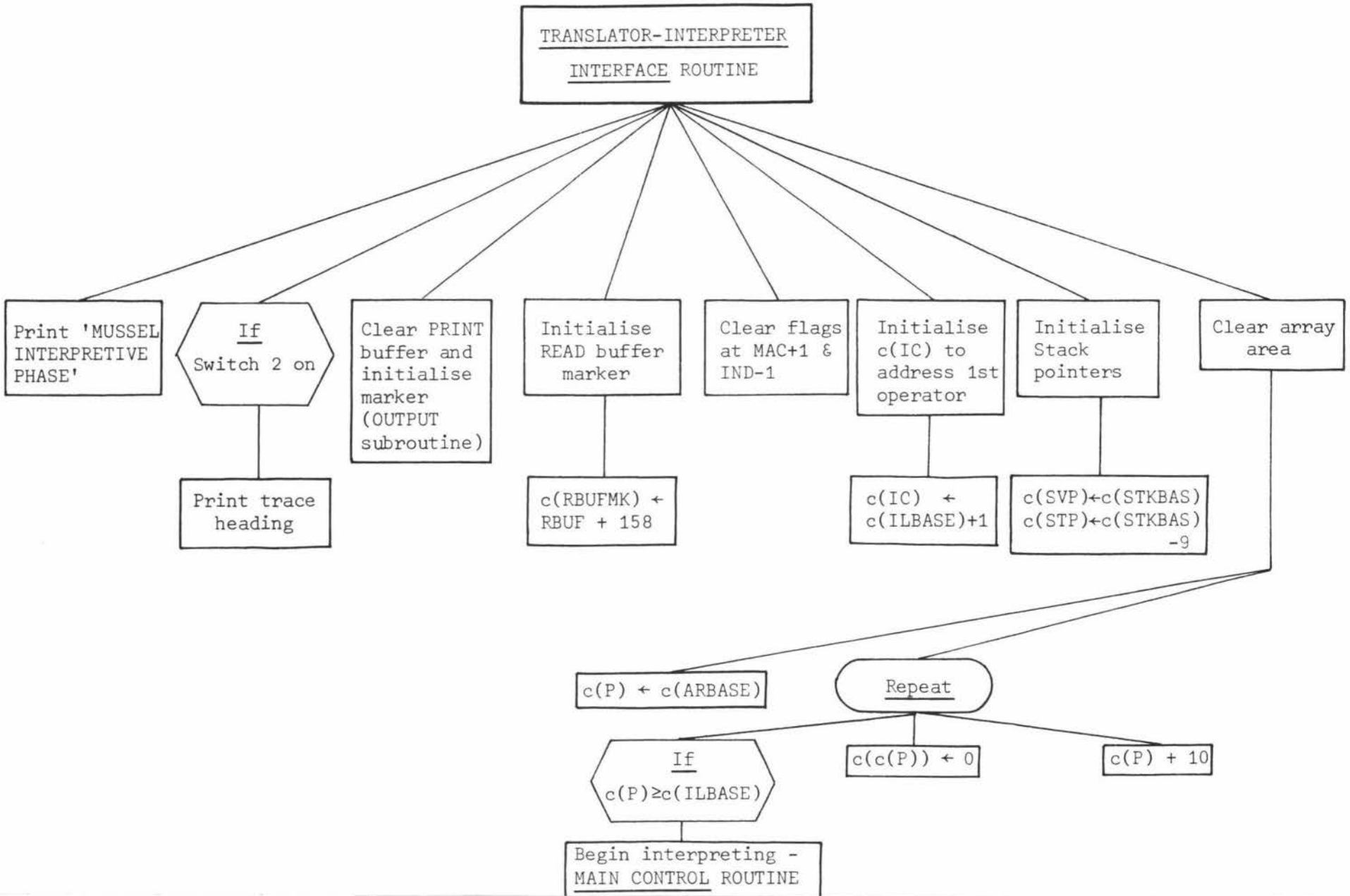


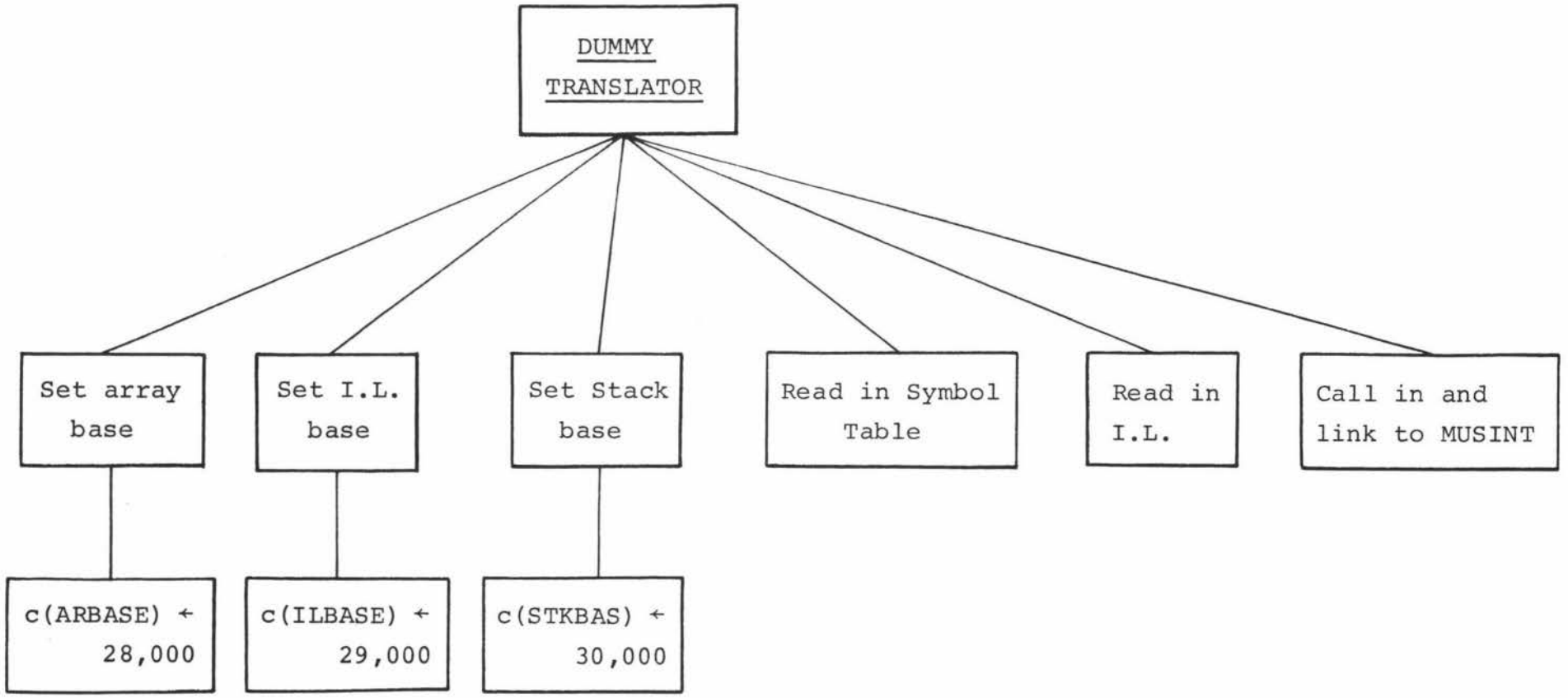

OUTPUT subroutine to print & clear contents (if any) of output buffer (PBUFF)
Anything in PBUFF?
If c(PBUFMK)>PBUFF-2
Print out contents of PBUFF
Reset buffer marker
Clear buffer
c(PBUFMK) ← PBUFF-2
c(P) ← PBUFF-2
Repeat until c(P)=PBUFF+238
c(c(P)) ← c(ZERO-8)


PBFMAN subroutine to control increments to print buffer marker, c(PBUFMK)
Increment marker
c(PBUFMK)+c(PBFMAN-1)
subroutine parameter
Check marker against buffer end
If c(PBUFMK)>PBUFF+238
If no array flag i.e. not array O/P
Buffer overflow - print warning error (PR E1)
Print & clear contents of PBUFF & reset c(PBUFMK) to address start of buffer (OUTPUT subroutine)
Skip to new line
Choice 'space' flag? (SPFLG?)
Yes
Clear SPFLG
No
Inc c(PBUFMK) to leave space at start of newline
c(PBUFMK)+c(PBFMAN-1)


POPTOP subroutine to decrement (or increment) Stack pointers
subroutine parameter
x̄ x x x x
STP DEC
SVP DEC
POPTOP
Dec. Stack value pointer
Dec. Stack type pointer
Define DEC field
Subtract
Append sign to DEC field
Subtract
Set flag at POPTOP-2
c(SVP)-c(POPTOP-1)
Move flag from POPTOP-1 to POPTOP-3
c(STP)-c(POPTOP-3)


MAIN CONTROL ROUTINE
Repeat
Place c(IC) & c(c(IC)) in PBUF (for trace)
Choice on c(c(IC))
If Switch 2 on
If Switch 3 on
Increment c(IC) to address next opcode
Print c(PBUF) (for trace)
'1'
'2'
. . .
'42'
LINE NUMBER ROUTINE
TAKE ADDRESS ROUTINE
. . .
TAB ROUTINE
Print Stack, Symbol Table, & key variables (for trace)
c(IC) + 2


TRANSLATOR-INTERPRETER INTERFACE ROUTINE
Print 'MUSSEL INTERPRETIVE PHASE'
If Switch 2 on
Print trace heading
Clear PRINT buffer and initialise marker (OUTPUT subroutine)
Initialise READ buffer marker
c(RBUFMK) ← RBUF + 158
Clear flags at MAC+1 & IND-1
Initialise c(IC) to address 1st operator
c(IC) ← c(ILBASE)+1
Initialise Stack pointers
c(SVP)←c(STKBAS)
c(STP)←c(STKBAS) -9
Clear array area
c(P) ← c(ARBASE)
Repeat
If c(P)≥c(ILBASE)
Begin interpreting - MAIN CONTROL ROUTINE
c(c(P)) ← 0
c(P) + 10


DUMMY
TRANSLATOR
Set array base
Set I.L. base
Set Stack base
Read in Symbol Table
Read in I.L.
Call in and link to MUSINT
c(ARBASE) ← 28,000
c(ILBASE) ← 29,000
c(STKBAS) ← 30,000

APPENDIX D

THE INTERPRETER SPS PROGRAM

```
*
* MUSSEL INTERPRETER VERSION 1 (1972)
* ===================================
*
*
**********************
**LINE NUMBER ROUTINE**
**********************
*
        DORG 2450,,,
 .LN    AM   IC,3,10,              INCR IC
        TFM  PBUF+54,5355,8,       FILL PBUF
        TNF  PBUF+30,-IC,,
        TF   LN,-IC,,              RESET LN
        B7   BACK,,,               RETURN TO CONTROL
*
**********************
**TAKE ADDRESS ROUTINE**
**********************
*
 .TA    AM   IC,3,10,              INCR IC
        TNF  PBUF+30,-IC,,         FILL PBUF
        TFM  PBUF+54,6341,8,
        BTM  POPTOP,1010,811,      INC STACK POINTERS
        MM   -IC,14,10,            CONVERT SYMTAB ADDRESS TO ABS ADDRESS
        A    99,SYMBAS,,           AND PLACE AT TOP OF STACK
        TF   -SVP,99,,
        TDM  -STP,6,,
        B7   BACK,,,               RETURN TO CONTROL
*
**********************
**TAKE RESULT ROUTINE**
**********************
*
 .TR    AM   IC,3,10,              INCR IC
        TNF  PBUF+30,-IC,,         FILL PBUF
        TFM  PBUF+54,6359,8,
        BTM  POPTOP,1010,811,      INC STACK POINTERS
        MM   -IC,14,10,            SET A TO ADDRESS VARIABLE
        TF   A,99,,
        A    A,SYMBAS,,
        B7   PUSH,,,               PUSH RESULT AND TYPE
*
***************
**PUSH ROUTINE**
***************
*
 PUSH   SM   A,9,10,
        TD   COMP,-A,,
        CM   COMP,0,10,            CHECK ADDRESSED VARIABLE TYPE
        BNE  *+32,,,
        PRA  TRE1,,,               ERROR - VARIABLE UNDEFINED
        B7   HALT,,,               HALT TO CHECK ERROR
        TD   -STP,COMP,,           PUSH TYPE
        CM   COMP,1,10,
        BE   BACK,,,               NULL STRING - RETURN TO CONTROL
        AM   A,9,10,
```

```
         CM    COMP,2,10,
         BNE   *+32,,,
         TFL   -SVP,-A,,           PUSH NUMBER
         B7    BACK,,,             RETURN TO CONTROL
         CM    COMP,3,10,
         BNE   *+32,,,
         TF    -SVP,-A,,,          PUSH BOOLEAN VALUE
         B7    BACK,,,             RETURN TO CONTROL
         CM    COMP,7,10,
         BE    *+32,,,
         PRA   TRE2,,,             ERROR IN TYPE CODE
         B7    HALT,,,             HALT TO CHECK ERROR
         TF    REF,-A,,            STRING DESCRIPTOR
         BTM   GRAB,0,10,          GRAB NEXT AVAILABLE NODE
         TF    -SVP,NODE2,,        PUSH DESCRIPTOR ADDRESS
         SM    A,5,10,
         SM    SVP,5,10,
         TF    -SVP,-A,,           PUSH DESCRIPTOR LENGTH
         AM    SVP,5,10,           RESET STACK VALUE POINTER
LOOP10   SM    REF,4,10,           COPY ADDRESSED STRING IN STRING AREA
         SM    NODE2,4,10,
         TF    -NODE2,-REF,,       COPY STRING UNIT VALUE
         AM    REF,4,10,
         AM    NODE2,4,10,
         BNF   *+32,-REF,,         CHECK REFERENCE NODE LINK
         SF    -NODE2,,,           TERMINATE COPY
         B7    BACK,,,             RETURN TO CONTROL
         TF    NODE1,NODE2,,
         BTM   GRAB,0,10,          GRAB NEXT AVAILABLE NODE
         SF    NODE2-3,,,          SET LINK FIELD
         TF    -NODE1,NODE2,,
         CF    NODE2-3,,,
         TF    REF,-REF,,          CHAIN DOWN REFERENCE STRING
         CF    REF-3,,
         B7    LOOP10,,,           LOOP AGAIN
*
**********************
**TAKE NUMBER ROUTINE**
**********************
*
 .TN     TFM   PBUF+54,6355,8,     FILL PBUF
         AM    IC,7,10,
         TNF   PBUF+38,-IC,,
         AM    IC,2,10,
         TNF   PBUF+42,-IC,,
         BTM   POPTOP,1010,811,    INC STACK POINTERS
         TFL   -SVP,-IC,,          PUSH VALUE
         TDM   -STP,2,,            PUSH TYPE
         B7    BACK,,,             RETURN TO CONTROL
*
****************************
**TAKE NUMBER ZERO ROUTINE**
****************************
*
 .TN0    TFM   PBUF+56,70,10,      FILL PBUF
         TFM   PBUF+54,6355,8,
         BTM   POPTOP,1010,811,    INC STACK POINTERS
```

```
         TFL   -SVP,FZERO,,          PUSH VALUE
         TDM   -STP,2,,              PUSH TYPE
         B7    BACK,,,               RETURN TO CONTROL
*
**************************
**TAKE NUMBER ONE ROUTINE**
**************************
*
 .TN1    TFM   PBUF+56,71,10,        FILL PBUF
         TFM   PBUF+54,6355,8,
         BTM   POPTOP,1010,811,      INC STACK POINTERS
         TFL   -SVP,FONE,,           PUSH VALUE
         TDM   -STP,2,,              PUSH TYPE
         B7    BACK,,,               RETURN TO CONTROL
*
***********************
**TAKE BOOLEAN ROUTINE**
***********************
*
 .TB     AM    IC,2,10,              INCR IC
         TNF   PBUF+28,-IC,,         FILL PBUF
         TFM   PBUF+54,6342,8,
         BTM   POPTOP,1010,811,      INC STACK POINTERS
         TF    -SVP,-IC,,            PUSH VALUE
         TDM   -STP,3,,              PUSH TYPE
         B7    BACK,,,               RETURN TO CONTROL
*
***********************
**TAKE STRING ROUTINE**
***********************
*
 .TS     TFM   PBUF+54,6362,8,       FILL PBUF
         BTM   GRAB,0,10,            GRAB A NODE
         BTM   POPTOP,1010,811,      INC STACK POINTERS
         TDM   -STP,7,,              PUSH STRING DESCRIPTOR TYPE
         TF    -SVP,NODE2,,          PUSH DESCRIPTOR ADDRESS FIELD
         TFM   LENGTH,0,8,
         TFM   COUNT,6,10,
 LOOP6   SM    COUNT,1,10,           SELECT NEXT STRING CHUNK
         AM    IC,2,10,
         BNR   ADD1,-IC,,            RECORD MARK TERMINATES STRING
         CM    COUNT,5,10,           CHECK COUNT
         BNE   *+44,,,
         SF    -NODE1,,,             ZERO CHUNK LENGTH - FLAG TERMINAL NODE1
         BTM   RETURN,0,10,          RETURN NODE2 TO AVAIL LIST
         B7    SETLTH,,,             BRANCH TO SET DESCRIPTOR LENGTH FIELD
         SM    IC,2,10,
         SF    IND,,,
         B7    OUT6,,,
 ADD1    AM    LENGTH,1,10,          UP LENGTH
         CM    COUNT,0,10,
         BNE   LOOP6,,,
 OUT6    SM    NODE2,4,10,           PACK CHUNK
         TF    -NODE2,-IC,,
         AM    NODE2,4,10,
         BNF   LINK,IND,,
         MF    -NODE2,IND,,          SET TERMINAL NODE LINK FIELD
```

```
         AM    IC,2,10,
 SETLTH  SM    SVP,5,10,
         TF    -SVP,LENGTH,,       SET STRING DESCRIPTOR LENGTH FIELD
         AM    SVP,5,10,
         B7    BACK,,,             RETURN TO CONTROL
 LINK    TF    NODE1,NODE2,,
         BTM   GRAB,0,10,          GRAB A NODE
         SF    NODE2-3,,,          SET LINK FIELD
         TF    -NODE1,NODE2,,
         CF    NODE2-3,,,
         AM    IC,1,10,
         SF    -IC,,,
         SM    IC,1,10,
         B7    LOOP6-12,,,         LOOP BACK TO PACK NEXT CHUNK
*
**************************
**TAKE NULL STRING ROUTINE**
**************************
*
  .TNS   TFM   PBUF+56,62,10,      FILL PBUF
         TFM   PBUF+54,6355,8,
         BTM   POPTOP,1010,811,    INC STACK POINTERS
         TDM   -STP,1,,            SET NULL STRING TYPE DIGIT
         B7    BACK,,,             RETURN TO CONTROL
*
*****************************************
**SUBROUTINE TO RETURN C(NODE2) TO AVAIL LIST**
*****************************************
*
         DS    2,,,
 RETURN  SF    AVAIL-3,,,
         TF    -NODE2,AVAIL,,      RESET LINK FIELD
         TF    AVAIL,NODE2,,       RESET AVAIL
         BB2   ,,,                 BRANCH BACK
*
*************************************************************
**SUBROUTINE TO SET C(NODE2) TO POINT TO NEXT AVAILABLE NODE**
*************************************************************
*
         DS    2,,,
 GRAB    BNF   POPNOD,AVAIL,,
         TF    NODE2,BOUND,,       SET NODE2
         SM    BOUND,16,10,        DEC STRING AREA BOUNDARY
         CM    BOUND,SVP,,         CHECK BOUND
         BL    *+14,,,
         BB2   ,,,                 BRANCK BACK
         PRA   STRE1,,,            ERROR - CORE OVERFLOW
         B7    HALT,,,             HALT TO CHECK ERROR
 POPNOD  TF    NODE2,AVAIL,,       SET NODE2
         TF    AVAIL,-AVAIL,,      POP AVAIL STACK
         CF    AVAIL-3,,,
         BB2   ,,,                 BRANCH BACK
*
***************************************
**INDEX ADDRESS, INDEX RESULT ROUTINES**
***************************************
*
```

```
.IND    TFM   PBUF+54,4955,8,,     FILL PBUF
        TDM   DIM,1,11,            SET DIM TO -1
TEST2   TD    COMP,-STP,,,
        CM    COMP,2,10,,
        BNE   ADDR,,,
        CM    -SVP,0,10,,
        BE    *+32,,,
        PRA   INDE1,,,             ERROR - INDEX NOT AN INTEGER
        B7    HALT,,,              HALT TO CHECK ERROR
        SM    SVP,6,10,,
        CM    -SVP,0,10,,
        BE    *+32,,,
        PRA   INDE6,,,             ERROR - INDEX MAGNITUDE GT 9999
        B7    HALT,,,              HALT TO CHECK ERROR
        AM    SVP,1,10,,
        SF    -SVP,,,
        BTM   POPTOP,1005,8,       DEC STACK POINTERS
        AM    DIM,1,10,,
        B7    TEST2,,,
ADDR    CM    COMP,6,10,           CHECK TYPE
        BE    *+32,,
        PRA   INDE2,,,             ERROR - NOT ADDRESS
        B7    HALT,,,              HALT TO CHECK ERROR
        TF    A,-SVP,,,            SET A TO ADDR APPROPRIATE TYPE DIGIT
        SM    A,9,10,,
        TD    COMP,-A,,,           CHECK TYPE DIGIT
        CM    COMP,4,10,,
        BE    *+32,,,
        PRA   INDE3,,,             ERROR - NOT AW
        B7    HALT,,,              HALT TO CHECK ERROR
        AM    A,1,10,,
        TD    COMP,-A,,,
        BNF   *+44,COMP,,          SET BASE
        TF    BASE,STKBAS,,,
        CF    COMP,,,
        B7    *+20,,,
        TF    BASE,SYMBAS,,,
        C     COMP,DIM,,,          CHECK DIMENSION
        BE    *+32,,,
        PRA   INDE4,,,             ERROR - DIMENSION DISCREPANCY
        B7    HALT,,,              HALT TO CHECK ERROR
        AM    A,4,10,,
        TF    W,BASE,,,            SET W TO ADDRESS FIRST ARRAY ELT
        A     W,-A,,,
        AM    SVP,8,10,
        AM    A,4,10,,
        TF    A,-A,,               SET A TO ADDRESS SMF
        CF    A-3,,,
        TDM   A-4,0,11,,
        A     A,BASE,,,
        TF    L0,-SVP,,,           CALCULATE L0
        S     L0,-A,,
        CM    L0,0,10,
        BNL   *+32,,,              CHECK LOWER BOUND
OUTBND  PRA   INDE5,,,             ERROR - INDEX OUTSIDE BOUNDS
        B7    HALT,,,              HALT TO CHECK ERROR
        AM    A,4,10,,
```

```
         C     -SVP,-A,,,          CHECK UPPER BOUND
         BH    OUTBND,,,
         BD    2D,DIM,,            BYPASS IF 2 DIMENSIONAL ARRAY
         TF    A,W,,,              SET A TO ADDRESS ARRAY ELT
         A     A-1,L0,,,
         SM    SVP,8,10,,
 TESTOP  BD    INDR,-IC,,
         TFM   PBUF+58,4441,8,
         TF    -SVP,A,,            PLACE ARRAY ELT ADDRESS ON STACK
         TDM   -STP,6,,            PLACE AA TYPE DIGIT ON STACK
         B7    BACK,,,             RETURN TO CONTROL
 INDR    TFM   PBUF+58,4459,8,
         B7    PUSH,,,             PUSH RESULT AND TYPE
 2D      AM    A,4,10,,
         AM    SVP,10,10,
         TF    L1,-SVP,,           CALCULATE L1
         S     L1,-A,,
         CM    L1,0,10,,           CHECK LOWER BOUND
         BL    OUTBND,,,
         AM    A,4,10,,
         C     -SVP,-A,,           CHECK UPPER BOUND
         BH    OUTBND,,,
         AM    A,4,10,,
         SM    SVP,18,10,,
         M     -A,L0,,             SET A TO ADDRESS ARRAY ELT
         A     99,L1,,,
         SF    96,,,
         TF    A,W,,,
         A     A-1,99,,,
         B7    TESTOP,,,
*
******************
**ST,STA ROUTINES**
******************
*
 .ST     TFM   PBUF+54,6263,8,,    FILL PBUF
         TF    TOPV,SVP,,
         TF    TOPT,STP,,
         BTM   POPTOP,1010,8,      DEC STACK POINTERS
         TD    COMP,-STP,,         CHECK STORE ADDRESS TYPE
         CM    COMP,6,10,
         BE    *+32,,,
         PRA   STE1,,,             ERROR - NOT ABSOLUTE ADDRESS
         B7    HALT,,,             HALT TO CHECK ERROR
         TF    A,-SVP,,            SET A TO ADDRESS STORE TYPE POSITION
         BT    STOR,TOPV,,         STORE TOP STACK TYPE AND VALUE
         CM    -IC,13,10,          CHECK OP CODE
         BE    STA,,,
         BTM   POPTOP,1010,8,      DEC STACK POINTERS
         B7    BACK,,,             RETURN TO CONTROL
 STA     TFM   PBUF+56,41,10,      STA OP
         TD    -STP,COMP,,         SHIFT DOWN TYPE
         CM    COMP,2,10,,         SHIFT DOWN VALUE
         BNE   *+32,,
         TFL   -SVP,-TOPV,,
         B7    BACK,,,             RETURN TO CONTROL
         TF    -SVP,-TOPV,,
```

```
         B7    BACK,,,                 RETURN TO CONTROL
*
*****************************************************************
**SUBROUTINE TO STORE VALUE (+ TYPE) ADDRESSED BY C(STOR-1) IN C(A)**
*****************************************************************
*
         DS    5,,,
 STOR    TF    TOPT,STOR-1,,           SET TOP TYPE POINTER
         SM    TOPT,9,10,
         SM    A,9,10,
         TD    COMP,-A,,
         TD    -A,-TOPT,,              STORE TYPE
         AM    A,9,10,
         CM    COMP,7,10,              CHECK REPLACED TYPE
         BNE   NOTSTR,,,
         TF    NODE2,-A,,              STRING TYPE - RETURN STRING UNITS
         B7    IN9,,,                                TO AVAIL LIST
 LOOP9   TF    NODE1,NODE2,,
         TF    NODE2,-NODE2,,
         CF    NODE2-3,,,
         TF    -NODE1,AVAIL,,
         TF    AVAIL,NODE1,,
 IN9     SF    AVAIL-3,,,
         BNF   LOOP9,-NODE2,,
         TF    -NODE2,AVAIL,,
         TF    AVAIL,NODE2,,
 NOTSTR  TD    COMP,-TOPT,,
         CM    COMP,1,10,              CHECK TYPE OF VALUE TO BE STORED
         BH    *+14,,,
         BB2   ,,,                     TYPE UNDEF OR NULL STRING - BRANCH BACK
         CM    COMP,2,10,
         BNE   *+26,,,
         TFL   -A,-STOR+1,,            NUMBER - STORE (TFL)
         BB2   ,,,                     BRANCH BACK
         CM    COMP,3,10,
         BNE   *+26,,,
 BOOL    TF    -A,-STOR+1,,            BOOLEAN VALUE - STORE (TF)
         BB2   ,,,                     BRANCH BACK
         CM    COMP,7,10,
         BE    *+32,,,
         PRA   STE2,,,                 ERROR IN TYPE CODE
         B7    HALT,,,                 HALT TO CHECK ERROR
         TF    -A,-STOR+1,,            STRING DESCRIPTOR - STORE ADDRESS
         SM    A,5,10,
         SM    STOR-1,5,10,
         TF    -A,-STOR+1,,            STORE LENGTH
         AM    A,5,10,
         BB2   ,,,                     BRANCH BACK
*
********************
**VALUE IS ROUTINE**
********************
*
 .VI     TFM   PBUF+54,6549,8,,        FILL PBUF
         TF    A,RAW,,                 SET A TO ADDRESS FUNCTION VALUE POSN
         SM    A,10,10,
         BT    STOR,SVP,,              STORE FUNCTION VALUE
```

```
         BTM   POPTOP,1010,8,      POP STACK TYPE AND VALUE
         B7    BACK,,,             RETURN TO CONTROL
*
*****************************
**UNCONDITIONAL JUMP ROUTINE**
*****************************
*
 .UJ     AM    IC,4,10,            INCR IC
         TFM   PBUF+54,6451,8,     FILL PBUF
         TNF   PBUF+32,-IC,,
         BTM   RESET,BACK,,        RESET IC AND RETURN TO CONTROL
*
*************************
**BRANCH ADDRESS ROUTINE**
*************************
*
 .BA     TFM   PBUF+54,4241,8,     FILL PBUF
         AM    IC,4,10,
         TNF   PBUF+32,-IC,,
         TD    COMP,-STP,,         CHECK TOP STACK TYPE
         CM    COMP,-8,10,
         BE    POPOUT,,,
 POP     BTM   POPTOP,1010,8,      POP STACK TYPE AND VALUE
         TD    COMP,-STP,,         CHECK TOP STACK TYPE
         CM    COMP,6,10,
         BNE   POP,,,              EXIT LOOP IF ADDRESS TYPE
 POPOUT  BTM   POPTOP,1010,8,      POP STACK TYPE AND VALUE
         BTM   RESET,BACK,,        RESET IC AND RETURN TO CONTROL
*
************************
**IF FALSE JUMP ROUTINE**
************************
*
 .IFJ    TD    COMP,-STP,,         CHECK TOP STACK TYPE
         CM    COMP,3,10,,
         BE    *+32,,,
         PRA   IFJE1,,,            ERROR - NOT BOOLEAN
         B7    HALT,,,             HALT TO CHECK ERROR
         AM    IC,4,10,,           INCR IC
         TFM   PBUF+56,51,10,      FILL PBUF
         TFM   PBUF+54,4946,8,,
         TNF   PBUF+32,-IC,,
         CM    -SVP,1,10,          CHECK TOP BOOLEAN VALUE
         BE    *+24,,,
         BTM   RESET,*+12,,        FALSE - RESET IC
         BTM   POPTOP,1010,8,      DEC STACK POINTERS
         B7    BACK,,,             RETURN TO CONTROL
*
*************************
**SUBROUTINE TO RESET IC**
*************************
*
         DS    5,,,                SPACE FOR BRANCH OUT ADDRESS
 RESET   TF    L0,-IC,,            SAVE IL PARAMETER
         TF    IC,ILBASE,,         TRANSMIT BASE
         A     IC,L0,,             ADD PARAMETER
         SM    IC,1,10,            SUBTRACT ONE
```

```
         B7    -RESET+1,,,          BRANCH BACK
*
***************
**OP ROUTINES**
***************
*
 .OP     AM    IC,2,10,             INCR IC
         TFM   PBUF+54,5657,8,,     FILL PBUF
         TNF   PBUF+28,-IC,,
         TFM   TABLE2,,TAB2,711,    RESET TABLE2
         A     TABLE2-1,-IC,,,      SET UP TABLE2 BRANCH ADDRESS
         B7    -TABLE2,,,           BRANCH
*
**OP BRANCHING TABLE2*****************************
*
 .TAB2   DS    ,*-5,,
         DSA   .ANDOR,0,,ANDOR,0,,NOT,0,,REL,0,,REL
         DSA   0,,REL,0,,REL,0,,REL,0,,REL,0,,CAT
         DSA   0,,ADD,0,,SUB,0,,MULT,0,,DIV,0,,INTDV
         DSA   0,,NEG
*
*******************
**AND,OR ROUTINES**
*******************
*
 .ANDOR  BTM   CHEC,3,10,           CHECK TOP 2 STACK TYPES
         CM    -IC,2,10,            RESET NEXT TO TOP STACK VALUE AND POP
         BE    *+24,,,
         TDM   DIG,1,,
         C     -TOPV,DIG,,
         BNE   *+24,,,
         TF    -SVP,DIG,,
         TDM   DIG,0,,              RESET DIG
         B7    BACK,,,              RETURN TO CONTROL
*
***************
**NOT ROUTINE**
***************
*
 .NOT    TD    COMP,-STP,,          CHECK TOP STACK TYPE
         CM    COMP,3,10,,
         BE    *+32,,,
         PRA   NOTE1,,,             ERROR - NOT BOOLEAN
         B7    HALT,,,              HALT TO CHECK ERROR
         BD    *+32,-SVP,,          COMPLEMENT TOP STACK VALUE
         TDM   -SVP,1,,
         B7    BACK,,,              RETURN TO CONTROL
         TDM   -SVP,0,,
         B7    BACK,,,              RETURN TO CONTROL
*
******************************
**EQ,NE,GT,GE,LT,LE ROUTINES**
******************************
*
 .REL    TFM   TESTSV-1,BACK,,      FILL BRANCH OUT ADDRESS
         TF    OP,-IC,,             STORE OP
         TD    COMP,-STP,,
```

```
         CM    COMP,2,10,          CHECK TOP STACK TYPE
         BNE   STREL,,,            NOT NUMBER - TO STR RELATIONAL ROUTINE
         BTM   CHEC,2,10,          OTHERWISE CHECK TOP 2 STACK TYPES
         TDM   -STP,3,,            FILL BOOLEAN TYPE
         TFM   EDIF,0,9,           CALCULATE EXPONENT DIFFERENCE
         A     EDIF,-SVP,,
         B7    *+14,,,
         DS    6,,,                SPACE FOR BRANCH OUT ADDRESS
TESTSV   S     EDIF,-TOPV,,
         SM    TOPV,2,10,          TOPV NOW POINTS TO TOP MANTISSA
         CM    EDIF,0,10,          COMPARE EXP DIFFERENCE WITH ZERO
         BL    LOW,,,              LESS
         BE    EQUAL,,,            EQUAL
HIGH     BT    EQNE,TESTSV-1,,     GREATER - CHECK FOR EQ OR NE
         CM    EDIF,7,10,
         BL    SCALE,,,
T1       CM    OP,7,10,            CHECK OP
         BL    FALSE,,,            FALSE IF LT,LE,OR EQ - OTHERWISE TRUE
TRUE     TFM   -SVP,1,10,          SET TOP STACK VALUE TO TRUE
         B7    -TESTSV+1,,,        BRANCH OUT
FALSE    TFM   -SVP,0,10,          SET TOP STACK VALUE TO FALSE
         B7    -TESTSV+1,,,        BRANCH OUT
SCALE    TFM   LOD,MAC,,           SCALE AND COMPARE MANTISSAS
         A     LOD,EDIF,,
         TF    MAC,-BOTV,,
         MF    -LOD,MAC,,
         C     -LOD,-TOPV,,
         BH    T1,,,               BOT GREATER - OP TEST 1
         BL    T3,,,               TOP GREATER - OP TEST 3
         B7    T2,,,               EQUAL - OP TEST 2
EQUAL    C     -BOTV,-TOPV,,       COMPARE MANTISSAS
         BL    LOW-12,,,           BOT LESS
         BE    T2,,,               EQUAL - OP TEST 2
         TFM   EDIF,10,10,         BOT GREATER
         B7    HIGH,,,
T2       CM    OP,8,10,            CHECK OP
         BE    TRUE,,,             TRUE IF GE
         CM    OP,7,10,
         BNL   FALSE,,,            FALSE IF NE OR GT
         CM    OP,4,10,
         BH    TRUE,,,             TRUE IF LE OR EQ
         B7    FALSE,,,            FALSE IF LT
         TFM   EDIF,-10,10,
LOW      BT    EQNE,TESTSV-1,,     CHECK FOR EQ OR NE
         CM    EDIF,-7,10,
         BH    SCOMP,,,
T3       CM    OP,6,10,            CHECK OP
         BL    TRUE,,,             TRUE IF LT OR LE
         BE    FALSE,,,            FALSE IF EQ
         CM    OP,7,10,
         BE    TRUE,,,             TRUE IF NE
         B7    FALSE,,,            FALSE IF GT OR GE
SCOMP    TFM   LOD,MAC,,           SCALE AND COMPARE MANTISSAS
         S     LOD,EDIF,,
         TF    MAC,-TOPV,,
         MF    -LOD,MAC,,,
         C     -BOTV,-LOD,,
```

```
         BH    T1,,,                    BOT GREATER - OP TEST 1
         BL    T3,,,                    BOT LESS - OP TEST 3
         B7    T2,,,                    EQUAL - OP TEST 2
*
**STRING RELATIONAL OPS**
*
 STREL   TF    TOPV,SVP,,               SAVE TOP STACK VALUE POINTER
         BTM   POPTOP,1010,8,           DEC STACK TYPE AND VALUE POINTERS
         CM    COMP,1,10,               CHECK TOP STACK TYPE
         BNE   DESCR,,,
         TD    COMP,-STP,,              NULL STRING - SAVE NEXT STACK TYPE
         TDM   -STP,3,,                 FILL BOOLEAN TYPE
         CM    COMP,1,10,               CHECK SAVED TYPE
         BE    T2,,,                    NULL STRING - STRINGS EQUAL - OP TEST 2
         CM    COMP,7,10,
         BE    T1,,,                    STR DESCRIPTOR - BOT STR GT - OP TEST 1
         PRA   STRLE1,,,                ERROR - INVALID STACK TYPE
         B7    HALT,,,                  HALT TO CHECK ERROR
 DESCR   CM    COMP,7,10,
         BNE   DESCR-20,,,
         TD    COMP,-STP,,              STR DESCRIPTOR - SAVE NEXT STACK TYPE
         TDM   -STP,3,,                 FILL BOOLEAN TYPE
         CM    COMP,1,10,               CHECK SAVED TYPE
         BE    T3,,,                    NULL STR - TOP STR GREATER - OP TEST 3
         CM    COMP,7,10,
         BNE   DESCR-20,,,,
         TF    NODE1,-SVP,,             DESCRIPTOR - CHECK STRINGS NODE BY NODE
         TF    NODE2,-TOPV,,            NODE1 POINTS TO BOT STR - NODE2 TO TOP
         SM    TOPV,5,10,               TOPV POINTS TO TOP STR LENGTH
         TF    BOTV,SVP,,
         SM    BOTV,5,10,               BOTV POINTS TO BOT STR LENGTH
         C     -BOTV,-TOPV,,            COMPARE STRING LENGTHS
         BE    *+24,,,
         BTM   EQNE,BACK,,              IF NOT EQUAL CHECK FOR EQ OR NE OP
         BT    TAIL,BOTV,,
         TF    BOTREM,TOPREM,,
         BT    TAIL,TOPV,,
         B7    TEST95,,,                AND THEN BRANCH INTO COMPARE LOOP
 LOOP95  SM    NODE1,4,10,              DEC NODE1 POINTER
         SM    NODE2,4,10,              DEC NODE2 POINTER
         C     -NODE1,-NODE2,,          COMPARE NODE VALUES
         BH    T1,,,                    BOT STR GREATER - OP TEST 1
         BL    T3,,,                    TOP STR GREATER - OP TEST 3
         AM    NODE1,4,10,              CHAIN DOWN STRING LISTS
         AM    NODE2,4,10,
         TF    NODE1,-NODE1,,
         TF    NODE2,-NODE2,,
         CF    NODE1-3,,,
         CF    NODE2-3,,,
 TEST95  BNF   *+20,-NODE2,,
         B7    OUT95,,,                 NODE2 FLAGGED - CHECK FOR NODE1 FLAG
         BNF   LOOP95,-NODE1,,          LOOP AGAIN IF NODE1 UNFLAGGED
         BNF   T12-36,-NODE2,,          OTHERWISE CHECK FOR NODE2 FLAG
 T13     SM    NODE1,4,10,              IF FLAG, DEC NODE1 POINTER
         SM    NODE2,4,10,              DEC NODE2 POINTER
         C     BOTREM,TOPREM,,          COMPARE STRING LENGTH REMAINDERS
         BE    T12,,,                   EQUAL - COMPARE NODE VALUES
```

```
         BH    *+44,,,
         S     NODE2,TOPREM,,      TOPREM GREATER - ADJUST NODE2
         A     NODE2,BOTREM,,
         B7    T12,,,              AND THEN COMPARE NODE VALUES
         S     NODE1,BOTREM,,      BOTREM GREATER - ADJUST NODE1
         A     NODE1,TOPREM,,
         B7    T12,,,              AND THEN COMPARE NODE VALUES
         SM    NODE1,4,10,         NO NODE2 FLAG - DEC NODE1 POINTER
         SM    NODE2,16,10,        ADJUST NODE2 POINTER
         A     NODE2,BOTREM,,
 T12     C     -NODE1,-NODE2,,     COMPARE NODE VALUES
         BL    T3,,,               BOT STRING LESS - OP TEST 3
         BH    T1,,,               BOT STRING GREATER - OP TEST 1
         C     -BOTV,-TOPV,,       EQUAL - COMPARE STRING LENGTHS
         BE    T2,,,               EQUAL - OP TEST 2
         B7    *-48,,,             OTHERWISE CHECK LESS OR GREATER
 OUT95   BNF   *+20,-NODE1,,
         B7    T13,,,              NODE1 FLAGGED - BRANCH TO TEST 13
         SM    NODE2,4,10,         OTHERWISE DEC NODE2 POINTER
         SM    NODE1,16,10,        ADJUST NODE1 POINTER
         A     NODE1,TOPREM,,
         B7    T12,,,,             AND THEN BRANCH TO COMPARE NODE VALUES
*
**SUBROUTINE TO FIND STRING LENGTH REMAINDER**
*
         DS    5,,,,               SPACE FOR STRING LENGTH FIELD ADDRESS
 TAIL    LD    99,-TAIL+1,,
         DM    96,6,10,            DIVIDE STRING LENGTH BY 6
         TF    TOPREM,99,,         MULTIPLY REMAINDER BY 2
         A     TOPREM,99,,         AND STORE IN TOPREM
         CM    TOPREM,0,10,
         BNE   *+24,,,
         TFM   TOPREM,12,10,       SET TO 12 IF ZERO
         BB2   ,,,                 BRANCH BACK
*
**SUBROUTINE TO CHECK FOR EQ OR NE OP**
*
         DS    5,,,                SPACE FOR BO ADDRESS IF EQ OR NE
 EQNE    CM    OP,6,10,            CHECK OP
         BNE   *+32,,,
         TFM   -SVP,0,10,          FALSE IF EQ
         B7    -EQNE+1,,,          BRANCH OUT
         CM    OP,7,10,
         BE    *+14,,
         BB2   ,,,
         TFM   -SVP,1,10,          TRUE IF NE
         B7    -EQNE+1,,,          BRANCH OUT
*
***************
**CAT ROUTINE**
***************
*
 .CAT    TFM   PBUF+56,63,10,      FILL PBUF
         TFM   PBUF+54,4341,8,
         TD    COMP,-STP,,         STORE TOP STACK TYPE DIGIT
         TF    TOPV,SVP,,          SAVE TOP VALUE POINTER
         BTM   POPTOP,1010,8,      DEC STACK POINTERS
```

```
         CM    COMP,1,10,          CHECK TOP STACK TYPE
         BE    BACK,,,             NULL STRING - RETURN TO CONTROL
         CM    COMP,7,10,
         BE    *+32,,,
STRERR   PRA   CATE1,,,            ERROR - NOT STRING TYPE
         B7    HALT,,,             HALT TO CHECK ERROR
         TF    BOTV,SVP,,          SET POINTERS
         SM    BOTV,5,10,
         TF    BOTSTR,SVP,,
         TF    TOPSTR,TOPV,,
         SM    TOPV,5,10,
         TD    COMP,-STP,,
         CM    COMP,1,10,          CHECK NEXT STACK TYPE
         BNE   *+56,,,
         TDM   -STP,7,,            NULL STRING - SHIFT DOWN TOP STACK TYPE
         TF    -BOTSTR,-TOPSTR,,   SHIFT DOWN STRING DESCRIPTOR ADDRESS
         TF    -BOTV,-TOPV,,       SHIFT DOWN STRING DESCRIPTOR LENGTH
         B7    BACK,,,             RETURN TO CONTROL
         CM    COMP,7,10,
         BNE   STRERR,,,
         TF    NODE1,-BOTSTR,,     SET NODE1
         B7    TEST7,,,
CHAIN    TF    NODE1,-NODE1,,      CHAIN DOWN TO TERMINAL NODE
         CF    NODE1-3,,,
TEST7    BNF   CHAIN,-NODE1,,
         TF    NODE2,-TOPSTR,,     SET NODE2
         SF    NODE2-3,,,          LINK 2 STRINGS
         TF    -NODE1,NODE2,,
         CF    NODE2-3,,,
         LD    99,-BOTV,,          DIVIDE BOTTOM STRING LENGTH BY 6
         DM    96,6,10,,
         SF    98,,,
         A     -BOTV,-TOPV,,       ADD STRING LENGTHS
         CM    99,0,10,            CHECK DIVISION REMAINDER AGAINST 0
         BE    BACK,,,             EQUAL - RETURN TO CONTROL
         TF    BOTREM,99,,         STORE REMAINDER (DIGITS) IN BOTREM
         MM    BOTREM,2,10,
         SF    98,,,
         TF    BOTREM,99,,
         LD    99,-TOPV,,          DIVIDE TOP STRING LENGTH BY 6
         DM    96,6,10,
         SF    98,,,
         TF    TOPREM,99,,         STORE REMAINDER (DIGITS) IN TOPREM
         MM    TOPREM,2,10,
         SF    98,,,
         TF    TOPREM,99,,
         B7    TEST8,,,            REPACK TOP STRING
LOOP8    BT    REPACK,BOTREM,,     REPACK NODE1
         AM    NODE2,3,10,
         A     NODE2,BOTREM,,
         TF    NODE1,NODE2,,
         TF    NODE2,-NODE2,,
         CF    NODE2-3,,,
TEST8    BNF   LOOP8,-NODE2,,
         TF    DIF,BOTREM,,        TIDY UP END NODES
         A     DIF,TOPREM,,
         SM    DIF,12,10,
```

```
         CM    DIF,0,10,
         BH    HI,,,
         CF    DIF,,,
         SF    -NODE1,,,              NODE1 TERMINATES STRING
         BTM   RETURN,0,10,           RETURN NODE2 TO AVAIL LIST
         SM    NODE1,4,10,            REPACK NODE1
         SM    NODE2,4,10,
         TF    TEMP,-NODE1,,
         TF    -NODE1,-NODE2,,
         S     NODE1,TOPREM,,
         TF    -NODE1,TEMP,,
         AM    NODE1,1,10,
         CF    -NODE1,,,
         B7    BACK,,,                RETURN TO CONTROL
 HI      SF    -NODE2,,,
         BT    REPACK,DIF,,           REPACK NODE1 - NODE2 TERMINATES STRING
         B7    BACK,,,                RETURN TO CONTROL
*
**********************
**REPACK SUBROUTINE**
**********************
*
         DS    2,,,
 REPACK  SM    NODE1,4,10,
         SM    NODE2,4,10,
         S     NODE2,REPACK-1,,
         TF    TEMP,-NODE1,,
         TF    -NODE1,-NODE2,,
         A     NODE1,BOTREM,,
         SM    NODE1,12,10,
         TF    -NODE1,TEMP,,
         AM    NODE1,1,10,
         AM    NODE2,1,10,
         MF    -NODE2,-NODE1,,
         BB2   ,,,                    BRANCH BACK
*
***************
**ADD ROUTINE**
***************
*
 .ADD    BTM   CHEC,2,10,             CHECK TOP 2 STACK TYPES
         CM    -TOPV,0,10,
         BNE   FLOAT,,,
         CM    -SVP,0,10,,
         BNE   FLOAT,,,
         SM    TOPV,2,10,             INTEGER OPERANDS
         BV    *+12,,,
         A     -BOTV,-TOPV,,,
         BV    *+32,,,
         BNF   BACK,IND-1,,           INTEGER RESULT - RETURN TO CONTROL
         B7    ISVSUB,,,              RETURN TO ISV SUBROUTINE
         TF    MAC,-BOTV,,            OVERFLOW - FLOATING RESULT
         CF    MAC-6,,,
         TDM   MAC-7,1,11,
         TFM   HOD,MAC-7,,            SET POINTERS B4 BRANCHING TO
         TFM   LOD,MAC-1,,            RNS ROUTINE
         TFM   EAC,1,9,,
```

```
         B7    ROUND,,,          BRANCH TO ROUND
*
**********************
**FLOATING ADD ROUTINE**
**********************
*
 FLOAT   C     -SVP,-TOPV,,      FLOATING POINT OPERANDS
         BNE   *+80,,,           EP GE EQ
         TF    EP,-TOPV,,
         TF    EQ,-SVP,,
         SM    TOPV,2,10,
         TF    MAC,-TOPV,,
         TF    MQ,-BOTV,,
         B7    *+68,,,
         TF    EP,-SVP,,
         TF    EQ,-TOPV,,
         SM    TOPV,2,10,
         TF    MAC,-BOTV,,
         TF    MQ,-TOPV,,
         TF    EDIF,EP,,         CALCULATE EXPONENT DIFFERENCE
         CF    EDIF-1,,,
         TDM   EDIF-2,0,11,
         S     EDIF,EQ,,
         CM    EDIF,8,10,,
         BL    *+56,,,
         TF    -BOTV,MAC,,       LARGE DIFFERENCE IN EXPONENTS
         TF    -SVP,EP,,
         BNF   BACK,IND-1,,      RETURN TO CONTROL
         B7    ISVSUB,,,         RETURN TO ISV SUBROUTINE
         TF    EAC,EQ,,          SCALE
         CF    EAC-1,,,
         TDM   EAC-2,0,11,
         TFM   LOD,MAC,,
         A     LOD,EDIF,,
         MF    -LOD,MAC,,
         BV    *+12,,,
         A     -LOD,MQ,,         ADD
         MF    IND,-LOD,,        STORE SIGN OF SUM
         BNV   *+92,,,
         CF    MAC-6,,,          OVERFLOW
         TDM   MAC-7,1,11,
         TFM   HOD,MAC-7,,       SET POINTERS BEFORE BRANCHING TO
         TFM   LOD,MAC-1,,       RNS ROUTINE
         A     EAC,EDIF,,
         AM    EAC,1,10,
         B7    RNS,,,
         CM    -LOD,0,10,        NO OVERFLOW
         BE    STZ,,,
         A     EAC,EDIF,,
 SETPTS  TFM   LOD,MAC,,         SET POINTERS B4 BRANCHING TO
         TFM   HOD,MAC-6,,       RNS ROUTINE
         B7    RNS,,,
 STZ     TFL   -SVP,FZERO,,      STORE ZERO RESULT
         CF    IND,,,
         BNF   BACK,IND-1,,      RETURN TO CONTROL
         B7    ISVSUB,,,         RETURN TO ISV SUBROUTINE
*
```

```
*************************************
**ROUND, NORMALISE, AND STORE ROUTINE**
*************************************
*
 RNS     BD    *+56,-HOD,,          REMOVE HIGH ORDER ZEROS
         AM    HOD,1,10,            SHIFT LEFT
         AM    LOD,1,10,
         SM    EAC,1,10,
         B7    RNS,,,
         SF    -HOD,,,
 ROUND   AM    LOD,1,10,            ROUND
         BV    *+12,,,
         AM    -LOD,5,10,,
         BV    *+32,,,
         SM    LOD,1,10,            NO OVERFLOW
         B7    NORM,,,
         CF    -HOD,,,              OVERFLOW
         SM    HOD,1,10,
         TDM   -HOD,1,11,
         SM    LOD,2,10,
         AM    EAC,1,10,
 NORM    BD    EXPON,-LOD,,         NORMALISE - REMOVE LOW ORDER ZEROS
         CF    -HOD,,,              SHIFT RIGHT
         SM    HOD,1,10,
         TDM   -HOD,0,11,
         SM    LOD,1,10,
         AM    EAC,1,10,
         B7    NORM,,,
 EXPON   BD    *+20,EAC-2,,         CHECK FOR EXPONENT OVERFLOW OR UNDERFLO
         B7    STORE,,,
         BNF   OFLOW,EAC,,
         PRA   UFLO,,,              UNDERFLOW - PRINT WARNING MESSAGE
         TFL   -SVP,FZERO,,,        FILL WITH ZEROS
         B7    STORE+48,,,
 OFLOW   PRA   OFLO,,,,             OVERFLOW - PRINT WARNING MESSAGE
         TFM   -LOD,ALL9S,,         FILL WITH NINES
         TFM   EAC,99,10,
 STORE   TF    -BOTV,-LOD,,         STORE RESULT
         MF    -BOTV,IND,,          STORE SIGN
         SF    EAC-1,,,
         TF    -SVP,EAC,,
         TF    MAC+15,ZERO-3,,      TIDY UP
         CF    MAC+1,,,
         BNF   *+20,IND-1,,
         B7    ISVSUB,,,            RETURN TO ISV SUBROUTINE IF NO IND-1 FL
         BNF   BACK,INFLAG,,        RETURN TO CONTROL IF NO INPUT FLAG
         CF    INFLAG,,,            OTHERWISE CLEAR INPUT FLAG
         B7    INSTOR,,,            AND BRANCH TO STORE
*
********************
**SUBTRACT ROUTINE**
********************
*
 .SUB    BTM   NEG,0,10,            NEGATE TOP STACK VALUE
         B7    .ADD,,,              BRANCH TO ADD ROUTINE
*
********************
```

```
**MULTIPLY ROUTINE**
********************
*
 .MULT   BTM   CHEC,2,10,          CHECK TOP 2 STACK TYPES
         TF    EAC,-TOPV,,         ADD EXPONENTS
         CF    EAC-1,,,
         TDM   EAC-2,0,11,
         A     EAC,-SVP,,
         SM    TOPV,2,10,
         M     -BOTV,-TOPV,,       MULTIPLY MANTISSAS
         MF    IND,99,,            STORE SIGN OF PRODUCT
         CM    99,0,10,
         BE    STZ,,,              ZERO RESULT
         CM    EAC,0,10,
         BNE   *+80,,,
         CM    92,0,10,
         BNE   *+56,,,
         SF    93,,,               STORE INTEGER RESULT
         TF    -BOTV,99,,
         TFM   -SVP,0,10,
         B7    BACK,,,             RETURN TO CONTROL
         TF    MAC+7,99,,          FLOATING RESULT - LOAD ACCUMULATOR
         AM    EAC,7,10,
         B7    SETPTS,,,           SET POINTERS
*
********************
**DIVIDE ROUTINE**
********************
*
 .DIV    BTM   CHEC,2,10,,         CHECK TOP 2 STACK TYPES
         TF    EAC,-SVP,,          SUBTRACT EXPONENTS
         CF    EAC-1,,,
         TDM   EAC-2,0,11,
         S     EAC,-TOPV,,
         SM    TOPV,2,10,
         CM    -BOTV,0,10,
         BE    STZ,,,              ZERO DIVIDEND - ZERO RESULT
         CM    -TOPV,0,10,
         BNE   *+32,,,
 ZDIV    PRA   DIVE1,,,            ERROR - ZERO DIVISOR
         B7    HALT,,,             HALT TO CHECK ERROR
         LD    91,-BOTV,,          LOAD DIVIDEND
         TFM   79,0,9,,
         D     85,-TOPV,,          DIVIDE
         MF    IND,92,,            STORE SIGN OF QUOTIENT
         TF    MAC+8,92,,          TRANSFER RESULT TO ACCUMULATOR
         B7    SETPTS,,,           SET POINTERS
*
**************************
**INTEGER DIVIDE ROUTINE**
**************************
*
 .INTDV  BTM   CHEC,2,10,          CHECK TOP 2 STACK TYPES
         CM    -TOPV,0,10,         CHECK INTEGER OPERANDS
         BNE   *+36,,,
         CM    -SVP,0,10,
         BE    *+32,,,
```

```
         PRA   IDIVE1,,,          ERROR - NON INTEGER OPERAND
         B7    HALT,,,            HALT TO CHECK ERROR
         SM    TOPV,2,10,
         CM    -BOTV,0,10,
         BE    STZ,,,             ZERO DIVIDEND - ZERO RESULT
         CM    -TOPV,0,10,
         BE    ZDIV,,,            ERROR - ZERO DIVISOR
         LD    99,-BOTV,,         LOAD DIVIDEND
         D     93,-TOPV,,         DIVIDE
         TF    -BOTV,92,,         STORE RESULT
         B7    BACK,,,            RETURN TO CONTROL
*
***************
**NEG ROUTINE**
***************
*
 .NEG    TD    COMP,-STP,,        CHECK TOP STACK TYPE
         CM    COMP,2,10,,
         BE    *+32,,,
         PRA   NEGE1,,,           ERROR - NOT NUMBER
         B7    HALT,,,            HALT TO CHECK ERROR
         BTM   NEG,0,10,          NEGATE TOP STACK VALUE
         B7    BACK,,,            RETURN TO CONTROL
*
*************************************
**SUBROUTINE TO NEGATE TOP STACK VALUE**
*************************************
*
         DS    2,,,
 NEG     TF    TOPV,SVP,,
         SM    TOPV,2,10,
         BNF   *+26,-TOPV,,       NEGATE TOP STACK VALUE
         CF    -TOPV,,,           CLEAR FLAG
         BB2   ,,,                BRANCH BACK
         SF    -TOPV,,,           SET FLAG
         BB2   ,,,                BRANCH BACK
*
*****************************************************
**SUBROUTINE TO CHECK TOP 2 STACK TYPES AND SET POINTERS**
*****************************************************
*
         DS    2,,,
 CHEC    TD    COMP,-STP,,
         C     COMP,CHEC-1,,      CHECK TOP STACK TYPE
         BNE   *+60,,,
         SM    STP,10,10,
         TD    COMP,-STP,,
         C     COMP,CHEC-1,,      CHECK NEXT STACK TYPE
         BE    *+32,,,
         PRA   CHECE1,,,          ERROR - INCORRECT TYPE
         B7    HALT,,,            HALT TO CHECK ERROR
         TF    TOPV,SVP,,
         SM    SVP,10,10,
         TF    BOTV,SVP,,
         SM    BOTV,2,10,
         BB2   ,,,                BRANCH BACK
*
```

```
*****************************************
**SUBROUTINE TO INC OR DEC STACK POINTERS**
*****************************************
*
         DS    4,,,                    SPACE FOR SVP,STP DECS
 POPTOP  SF    POPTOP-2,,,             SPLIT
         S     SVP,POPTOP-1,,          POP VALUE
         MF    POPTOP-3,POPTOP-1,,
         S     STP,POPTOP-3,,          POP TYPE
         C     SVP,BOUND,,             CHECK C(SVP) AGAINST C(BOUND)
         BNH   *+32,,,
         PRA   CORFLO,,,               IF GREATER CORE OVERFLOW
         B7    HALT,,,                 HALT
         BB2   ,,,                     OTHERWISE BRANCH BACK
*
***********************
**REPEAT-TIMES ROUTINE**
***********************
*
 .RT     TFM   PBUF+54,5963,8,         FILL PBUF
         AM    IC,4,10,
         TNF   PBUF+32,-IC,,
         AM    IC,3,10,
         TNF   PBUF+38,-IC,,
         SM    IC,3,10,
         TF    TOPV,SVP,,
         SM    TOPV,2,10,
         BNF   NF1,-STP,,              FLAG INDICATES LOOP ALREADY TRAVERSED
 CHECNT  CM    -TOPV,0,10,             CHECK LOOP COUNT AGAINST ZERO
         BH    RSETLN,,,               IF HIGH RESET LINE NUMBER
         BTM   RESET,BACK,,            ELSE RESET IC TO BA ADDR AND RET TO CON
 RSETLN  SM    -TOPV,1,10,             DEC LOOP COUNTER
         AM    IC,3,10,                RESET LINE NUMBER
         TF    LN,-IC,,
         B7    BACK,,,                 RETURN TO CONTROL
 NF1     TD    COMP,-STP,,             NO FLAG INDICATES 1ST TRAVERSAL OF LOOP
         CM    COMP,2,10,              CHECK LOOP CONTROL TYPE
         BE    *+32,,,
         PRA   RTE1,,,                 ERROR - NOT NUMBER
         B7    HALT,,,
         CM    -SVP,0,10,
         BE    *+32,,,
         PRA   RTE2,,,                 ERROR - NOT INTEGER
         B7    HALT,,,
         TDM   -STP,8,11,
         B7    CHECNT,,,               BRANCH TO CHECK LOOP COUNT
*
**********************
**REPEAT-FOR ROUTINE**
**********************
*
 .RF     TFM   PBUF+54,5946,8,         FILL PBUF
         AM    IC,4,10,
         TNF   PBUF+32,-IC,,
         AM    IC,3,10,
         TNF   PBUF+38,-IC,,
         SM    IC,3,10,
```

```
         BNF   NF2,-STP,,          FLAG INDICATES LOOP ALREADY TRAVERSED
         BTM   ISV,30,10,          INCREMENT LOOP VARIABLE
         TFM   OP,5,10,            TEST LOOP VARIABLE
         SM    TOPV,8,10,
         TFM   EDIF,0,9,
         A     EDIF,-SVP,,
         TF    SVP,TOPV,,
         SM    SVP,10,10,
         BTM   TESTSV,*+12,,
         TF    TOPV,SVP,,
         AM    SVP,20,10,
         BD    RSETLN+12,-TOPV,,   NOT END OF LOOP - RESET LINE NUMBER
         BTM   RESET,BACK,,        ELSE RESET IC TO BA ADDR AND RET TO CON
 NF2     AM    IC,3,10,            NO FLAG INDICATES 1ST TRAVERSAL OF LOOP
         BTM   CHKSET,3,10,        CHK LOOP CONTROL TYPES + INIT LOOP VAR
         BTM   POPTOP,3020,811,    RESET STACK POINTERS
         SF    -STP,,,             SET FLAG AT TOP STACK TYPE POSITION
         B7    BACK,,,             RETURN TO CONTROL
*
**********************************
**REPEAT-FOR-WITHOUT-TEST ROUTINE**
**********************************
*
 .RFWT   TFM   PBUF+54,5946,8,     FILL PBUF
         TFM   PBUF+58,6663,8,
         AM    IC,3,10,
         TNF   PBUF+30,-IC,,
         BNF   NF3,-STP,,          FLAG INDICATES LOOP ALREADY TRAVERSED
         BTM   ISV,20,10,          INCREMENT LOOP VARIABLE
         TF    SVP,TOPV,,          RESET SVP
         AM    SVP,2,10,
         B7    BACK,,,             RETURN TO CONTROL
*                                  NO FLAG INDICATES 1ST LOOP TRAVERSAL
 NF3     BTM   CHKSET,2,10,        CHK LOOP CONTROL TYPES + INIT LOOP VAR
         BTM   POPTOP,2010,811,    RESET STACK POINTERS
         SF    -STP,,,             SET FLAG AT TOP STACK TYPE POSITION
         B7    BACK,,,             RETURN TO CONTROL
*
**********************************************************************
**SUBROUTINE TO CHECK LOOP CONTROL TYPES + INITIALISE LOOP VARIABLE**
**********************************************************************
*
         DS    2,,,                SPACE FOR COUNT PARAMETER
 CHKSET  TF    COUNT,CHKSET-1,,    SET COUNT
 TEST4   CM    COUNT,0,10,         TEST COUNT
         BE    OUT4,,,             EXIT LOOP IF ZERO
         SM    COUNT,1,10,         DEC COUNT
         TD    COMP,-STP,,         CHECK TOP STACK TYPE
         CM    COMP,2,10,
         BE    *+32,,,
         PRA   RFE1,,,             ERROR - NOT NUMBER
         B7    HALT,,,
         SM    STP,10,10,          DEC STACK POINTERS
         SM    SVP,10,10,
         B7    TEST4,,,            LOOP BACK
 OUT4    TF    A,-SVP,,            SET A TO ADDRESS LOOP VARIABLE
         AM    SVP,10,10,          SET SVP TO ADDRESS INITIAL VALUE
```

```
         TFL   -A,-SVP,,          INITIALISE VALUE
         SM    A,9,10,
         TDM   -A,2,,             INITIALISE TYPE
         BB2   ,,,                BRANCH BACK
*
*****************************************
**SUBROUTINE TO INCREMENT LOOP VARIABLE**
*****************************************
*
         DS    2,,,               SPACE FOR PARAMETER
 ISV     TF    TOPV,SVP,,         SET POINTERS
         S     SVP,ISV-1,,
         TF    SVP,-SVP,,
         TF    BOTV,SVP,,
         SM    BOTV,2,10,
         SF    IND-1,,,           SET FLAG AT IND-1
         B7    .ADD+12,,,         BRANCH TO ADD ROUTINE
 ISVSUB  CF    IND-1,,,           CLEAR FLAG AT IND-1
         BB2   ,,,                BRANCH BACK
*
**********************
**REPEAT-SET ROUTINE**
**********************
*
 .RS     AM    IC,2,10,
         TFM   PBUF+54,5962,8,    FILL PBUF
         TNF   PBUF+28,-IC,,
         S     SVP-1,-IC,,        SET A TO ADDRESS LOOP VARIABLE
         TF    A,-SVP,,
         BNF   NF4,-STP,,
         AM    SVP,10,10,         FLAG INDICATES LOOP ALREADY TRAVERSED
         AM    -SVP,1,10,         INCREMENT SET LIST COUNT
         C     -SVP,-IC,,         CHECK AGAINST NO. OF SET LIST ELTS
         BNE   NEXTEL,,,
         TF    SVP,STP,,          EQUAL
         AM    SVP,9,10,
         BTM   FILL,0,10,         FILL PBUF
         SM    IC,3,10,
         BTM   RESET,BACK,,       RESET IC TO BA ADDRESS AND RET TO CONTR
 NEXTEL  A     SVP-1,-SVP,,       SET SVP TO ADDRESS NEXT SET LIST ELT
         BT    STOR,SVP,,         RESET LOOP VARIABLE
         BTM   FILL,0,10,         FILL PBUF AND INCR IC
         TF    LN,-IC,,           RESET LINE NUMBER
         TF    SVP,STP,,          RESET SVP
         AM    SVP,9,10,
         B7    BACK,,,            RETURN TO CONTROL
 NF4     SF    -STP,,,            NO FLAG INDICATES 1ST TRAVERSAL OF LOOP
         AM    SVP,10,10,
         BT    STOR,SVP,,         INITIALISE LOOP VARIABLE
         BTM   FILL,0,10,         FILL PBUF AND INCR IC
         TFM   -SVP,0,10,         SET LIST COUNT TO ZERO
         B7    NF4-32,,,          RESET SVP AND RETURN
*
***********************************************
**REPEAT-SET SUBROUTINE TO FILL PBUF AND INCR IC**
***********************************************
*
```

```
         DS    2,,,
 FILL    AM    IC,4,10,
         TNF   PBUF+36,-IC,,
         AM    IC,3,10,
         TNF   PBUF+42,-IC,,
         BB2   ,,,                    BRANCH BACK
*
***************
**CASE ROUTINE**
***************
*
 .CS     TFM   PBUF+54,4362,8,        FILL PBUF
         AM    IC,2,10,
         TNF   PBUF+28,-IC,,
         TFM   COUNT,3,10,            SET COUNT
 TEST5   CM    COUNT,0,10,            CHECK COUNT
         BE    OUT5,,,                EXIT IF ZERO
         TD    COMP,-STP,,            CHECK TOP STACK TYPE
         CM    COMP,2,10,
         BE    *+32,,,
         PRA   CSE1,,,                ERROR - NOT NUMBER
         B7    HALT,,,
         CM    -SVP,0,10,
         BE    *+32,,,
         PRA   CSE2,,,                ERROR - NOT INTEGER
         B7    HALT,,,
         SM    COUNT,1,10,            DEC COUNT
         BTM   POPTOP,1010,8,         DEC STACK POINTERS
         B7    TEST5,,,               LOOP BACK
 OUT5    AM    SVP,28,10,             SET POINTERS
         TF    TOPV,SVP,,
         SM    SVP,10,10,
         TF    BOTV,SVP,,
         SM    SVP,10,10,
         C     -SVP,-TOPV,,           CHECK UPPER CASE BOUND
         BNH   *+32,,,
         PRA   CSE3,,,                ERROR - CASE CONTROL VALUE EXCEEDS U BN
         B7    HALT,,,
         BV    *+12,,,                TURN OFF OVERFLOW
         S     -SVP,-BOTV,,           SUBTRACT L BND FROM CASE CONTROL VAR
         BNV   *+32,,,
         PRA   CSE4,,,                ERROR - OVERFLOW
         B7    HALT,,,
         CM    -SVP,0,10,             CHECK AGAINST ZERO
         BNL   *+32,,,
         PRA   CSE5,,,                ERROR - CASE CONTROL VALUE BELOW L BND
         B7    HALT,,,
         C     -SVP,-IC,,             CHECK AGAINST NO. OF CASE STATEMENTS
         BL    *+32,,,
         PRA   CSE6,,,                ERROR - EXCEEDS NO. OF CASE STATEMENTS
         B7    HALT,,,
 RSETIC  MM    -SVP,6,10,             RESET IC
         SF    96,,,
         A     IC,99,,
         SM    SVP,8,10,
         B7    BACK,,,                RETURN TO CONTROL
*
```

```
*****************************
**CASE-WITHOUT-BOUNDS ROUTINE**
*****************************
*
 .CSWB   TFM   PBUF+54,4362,8,      FILL PBUF
         TFM   PBUF+58,6642,8,
         AM    IC,2,10,
         TNF   PBUF+28,-IC,,
         TD    COMP,-STP,,          CHECK TOP STACK TYPE
         CM    COMP,2,10,
         BE    *+32,,,
         PRA   CSWBE1,,,            ERROR - NOT NUMBER
         B7    HALT,,,
         CM    -SVP,0,10,
         BE    *+32,,,
         PRA   CSWBE2,,,            ERROR - NOT INTEGER
         B7    HALT,,,
         BTM   POPTOP,1002,8,       DEC STACK POINTERS
         CM    -SVP,1,10,           CHECK CONTROL VARIABLE AGAINST ONE
         BNL   *+32,,,
         PRA   CSWBE3,,,            ERROR - CASE CONTROL VALUE LESS THAN ON
         B7    HALT,,,
         C     -SVP,-IC,,           CHK CONTROL VAR AGAINST NO. OF CS STMTS
         BNH   *+32,,,
         PRA   CSWBE4,,,            ERROR - CONTROL VALUE GT NO. OF CS STMT
         B7    HALT,,,
         SM    -SVP,1,10,
         B7    RSETIC,,,            RESET IC
*
***********************************
**RETURN ADDRESS (PROCEDURE) ROUTINE**
***********************************
*
 .RAP    TFM   PBUF+56,57,10,       FILL PBUF
         TFM   PBUF+54,5941,8,,
         BTM   POPTOP,1006,811,     INC STACK POINTERS
 FILRAW  TDM   -STP,9,,             FILL RAW AND SET RAW POINTER
         TF    -SVP,RAW,,
         AM    SVP,4,10,
         TF    RAW,SVP,,
         B7    BACK,,,              RETURN TO CONTROL
*
************************************
**RETURN ADDRESS (FUNCTION) ROUTINE**
************************************
*
 .RAF    TFM   PBUF+56,46,10,       FILL PBUF
         TFM   PBUF+54,5941,8,,
         BTM   POPTOP,1010,811,     LEAVE SPACE FOR FUNCTION VALUE
         TDM   -STP,0,,             SET TO UNDEFINED
         B7    FILRAW-12,,,         FILL RAW AND SET RAW POINTER
*
************************
**CALL PROCEDURE ROUTINE**
************************
*
 .CP     AM    IC,3,10,
```

```
         TFM   PBUF+54,4357,8,      FILL PBUF
         TNF   PBUF+30,-IC,,
         MM    -IC,14,10,           CONVERT SYMTAB ADDR TO ABSOLUTE ADDR
         TF    A,99,,
         A     A,SYMBAS,,
         SM    A,9,10,
         TD    COMP,-A,,
         CM    COMP,5,10,
         BE    *+32,,,
         PRA   CPE1,,,              ERROR - NOT PAW
         B7    HALT,,,              HALT TO CHECK ERROR
         AM    A,9,10,,
         AM    IC,2,10,,
         TNF   PBUF+34,-IC,,
         TF    AP,-IC,,             STORE NO. OF ACTUAL PARAMETERS
         S     IC,ILBASE,,          STORE RETURN I.L. ADDRESS IN RAW
         SF    IC-3,,,
         TF    -RAW,IC,,
         TF    IC,ILBASE,,          SET I.C. TO START OF PROCEDURE
         A     IC,-A,,
         B7    BACK,,,              RETURN TO CONTROL
*
**************************
**PROCEDURE ENTRY ROUTINE**
**************************
*
 .PE     AM    IC,2,10,
         TNF   PBUF+28,-IC,,        FILL PBUF
         TFM   PBUF+54,5745,8,,
         C     AP,-IC,,             COMPARE NO. OF A.P.S. WITH NO. OF F.P.S
         BE    *+32,,,
         PRA   PEE1,,,              ERROR - NO. A.P.S NE NO. F.P.S
         B7    HALT,,,              HALT TO CHECK ERROR
         AM    IC,2,10,
         TNF   PBUF+32,-IC,,
         TF    BVP,RAW,,            SET BASE VALUE POINTER
         TF    BTP,RAW,,            AND BASE TYPE POINTER
         SM    BTP,9,10,,
         CM    -IC,0,10,
         BE    BACK,,,
         A     SVP-1,-IC,,          LEAVE SPACE FOR LOCAL VARIABLES
         AM    SVP,20,10,,          AND STORAGE MAPPING FUNCTION
         TF    WVP,SVP,,            SET WORKING VALUE POINTER
         B7    CLEAR,,,             SET LOCAL VARIABLE TYPES TO UNDEFINED
*
***********************************************
**TAKE LOCAL ADDRESS, TAKE LOCAL RESULT ROUTINES**
***********************************************
*
 .TL     CM    -IC,32,10,
         BE    *+24,,,
         SF    IND,,,               IND FLAG INDICATES TLA OP
         AM    IC,2,10,
         TFM   PBUF+54,6353,8,      FILL PBUF
         TNF   PBUF+28,-IC,,
         TF    LVP,BVP,,            SET LOCAL VARIABLE POINTERS
         A     LVP-1,-IC,,,
```

```
         TF    LTP,LVP,,
         SM    LTP,9,10,
         BTM   POPTOP,1010,811,     INC STACK POINTERS
         TD    COMP,-LTP,,
         BNF   TLR,IND,,
         CF    IND,,,
         TFM   PBUF+56,41,10,       TLA OP
         CM    COMP,5,10,           CHECK TYPE OF ADDRESSED VARIABLE
         BL    *+44,,,
         TF    -SVP,-LVP,,,         ADDRESS TYPE - PUSH THIS ADDRESS VALUE
         TD    -STP,-LTP,,
         B7    BACK,,,              RETURN TO CONTROL
         TDM   -STP,6,,             RESULT TYPE - PUSH LOCAL VARIABLE ADDR
         TF    -SVP,LVP,,
         B7    BACK,,,              RETURN TO CONTROL
 TLR     TFM   PBUF+56,59,10,       TLR OP
 TEST3   CM    COMP,6,10,           CHECK TYPE OF ADDRESSED VARIABLE
         BNE   OUT3,,,
         TF    LTP,-LVP,,           AA TYPE - CHAIN DOWN
         TF    LVP,LTP,,
         SM    LTP,9,10,
         TD    COMP,-LTP,,
         B7    TEST3,,,
 OUT3    TF    A,LVP,,
         B7    PUSH,,,              PUSH RESULT AND TYPE
*
**********************************
**MAKE STORAGE FUNCTION ROUTINE**
**********************************
*
 .MSF    AM    IC,4,10,,
         TFM   PBUF+56,46,10,       FILL PBUF
         TFM   PBUF+54,5462,8,
         TNF   PBUF+32,-IC,,
         TF    MFP,WVP,,            SET MFP DEPENDING ON ARRAY DIMENSION
         BD    *+32,-IC,,
         SM    MFP,12,10,
         B7    *+32,,,
         SM    MFP,4,10,,
         SF    IND,,,               IND FLAG INDICATES 2 DIMENSIONAL ARRAY
         BTM   LOAD,0,10,           CHECK AND LOAD INDEX VALUE
         TF    SIZE,-MFP,,
         CF    SIZE-3,,,
         BTM   DEC,0,10,            DECREMENT POINTERS
         BTM   LOAD,0,10,           CHECK AND LOAD INDEX VALUE
         S     SIZE,-MFP,,
         AM    SIZE,1,10,
         CM    SIZE-3,0,10,
         BE    *+32,,,
 2BIG    PRA   MSFE1,,,             ERROR - ARRAY SIZE GT 999
         B7    HALT,,,              HALT TO CHECK ERROR
         SF    SIZE-2,,,
         BNF   AROUND,IND,,         BYPASS IF 1 DIMENSIONAL ARRAY
         BTM   DEC,0,10,            DECREMENT POINTERS
         BTM   LOAD,0,10,           CHECK AND LOAD INDEX VALUE
         TF    ACC,-MFP,,,
         BTM   DEC,0,10,            DECREMENT POINTERS
```

```
         BTM   LOAD,0,10,          CHECK AND LOAD INDEX VALUE
         TF    -WVP,SIZE,,         FILL IN C1 IN SMF
         CF    ACC-3,,,
         S     ACC,-MFP,,          CALCULATE ARRAY SIZE
         AM    ACC,1,10,
         M     SIZE,ACC,,
         CM    96,0,10,,
         BNE   2BIG,,,
         SF    97,,,
         TF    SIZE,99,,,
AROUND   SM    IC,2,10             DECREMENT IC
         TF    SVP,BVP,,           CALCULATE TOP AW ADDRESS
         A     SVP-1,-IC,,
         TF    STP,SVP,,
         SM    STP,9,10,,
         AM    IC,1,10,,
         TD    COUNT,-IC,,         SET COUNT TO NUMBER OF ARRAY IDENTIFIER
         AM    WVP,10,10,
         S     WVP,STKBAS,,
         SF    WVP-3,,,
         TF    MFP,WVP,,           SET MFP TO 4 DIGIT RELATIVE ADDRESS
         SM    MFP,26,10,          OF SMF
TEST1    CM    COUNT,0,10,
         BE    OUT1,,,             EXIT IF COUNT ZERO
         TDM   -STP,4,,            FILL IN ARRAY WORD
         TF    -SVP,MFP,,,
         SM    SVP,4,10,
         TF    -SVP,WVP,,
         SM    SVP,4,10,
         BNF   *+32,IND,,
         TDM   -SVP,1,11,
         B7    *+20,,,
         TDM   -SVP,0,11,
         A     WVP-1,SIZE,,        RESERVE SPACE FOR ARRAY ELTS
         SM    COUNT,1,10,         DEC COUNT
         SM    SVP,2,10,,          DEC STACK POINTERS TO ADDRESS NEXT AW
         SM    STP,10,10,
         B7    TEST1,,,
OUT1     CF    IND,,,              CLEAR IND FLAG
         CF    WVP-3,,,
         A     WVP,STKBAS,,
         AM    WVP,10,10,          LEAVE SPACE FOR SMF
         TF    SVP,WVP,,
         TF    STP,STKBAS,,
         A     STP,MFP,,
         AM    STP,7,10,
         AM    IC,1,10,
         B7    CLEAR,,,            SET ARRAY ELT TYPES TO UNDEFINED
*
*****************************************************************
**MSF SUBROUTINE TO DECREMENT MFP BY 4, SVP BY 8, AND STP BY 10**
*****************************************************************
*
         DS    2,,,
DEC      SM    MFP,4,10,,          DEC MFP
         SM    SVP,8,10,,          DEC SVP
         SM    STP,10,10,,         DEC STP
```

```
         BB2   ,,,                    BRANCH BACK
*
*****************************************
**MSF SUBROUTINE TO CHECK AND LOAD INDEX VALUE**
*****************************************
*
         DS    2,,,
 LOAD    TD    COMP,-STP,,
         CM    COMP,2,10,
         BE    *+32,,,
         PRA   MSFE2,,,               ERROR - INDEX NOT NUMBER
         B7    HALT,,,                HALT TO CHECK ERROR
         CM    -SVP,0,10,,
         BE    *+32,,,
         PRA   MSFE3,,,               ERROR - INDEX NOT INTEGER
         B7    HALT,,,                HALT TO CHECK ERROR
         SM    SVP,6,10,,
         CM    -SVP,0,10,,
         BE    *+32,,,
         PRA   MSFE4,,,               ERROR - INDEX MAGNITUDE GT 9999
         B7    HALT,,,                HALT TO CHECK ERROR
         AM    SVP,1,10,,
         SF    -SVP,,,
         AM    SVP,3,10,,
         TF    -MFP,-SVP,,            LOAD INDEX VALUE
         BB2   ,,,                    BRANCH BACK
*
*****************
**RETURN ROUTINE**
*****************
*
 .RE     TFM   PBUF+54,5945,8,,       FILL PBUF
         TF    SVP,BVP,,              RESET STACK POINTERS
         TF    STP,BTP,,
         BTM   POPTOP,1010,8,
         TF    IC,ILBASE,,            RESET IC
         A     IC,-RAW,,
         SM    RAW,4,10,,
         TF    RAW,-RAW,,             RESET RAW POINTER
         TF    BVP,RAW,,              RESET BASE VALUE POINTER
         TF    BTP,RAW,,              RESET BASE TYPE POINTER
         SM    BTP,9,10,,
         B7    BACK,,,                RETURN TO CONTROL
*
*************************************************************
**SUBROUTINE TO SET TYPE CODES OF STACK VARIABLES TO UNDEFINED**
*************************************************************
*
 CLEAR   C     SVP,BOUND,,            CHECK C(SVP) AGAINST C(BOUND)
         BNH   *+32,,,
         PRA   CORFLO,,,              IF GREATER, CORE OVERFLOW
         B7    HALT,,,                HALT
         TF    TOPT,WVP,,             OTHERWISE SET TEST POINTER
         SM    TOPT,29,10,
 LOOP11  AM    STP,10,10,             INCR TYPE POINTER
         TDM   -STP,0,,               CLEAR TYPE POSITION
         C     STP,TOPT,,             TEST
```

```
         BNE   LOOP11,,,
         AM    STP,20,10,          SET STP TO ADDR TOP STACK TYPE POSITION
         B7    BACK,,,             RETURN TO CONTROL
*
*******************
**NEWPAGE ROUTINE**
*******************
*
 .NP     TFM   PBUF+54,5557,8,     FILL PBUF
         BTM   OUTPUT,*+12,,       PRINT AND CLEAR PBUFF
         SKIP  ,1,,                SKIP TO NEWPAGE
         B7    BACK,,,             RETURN TO CONTROL
*
*******************
**NEWLINE ROUTINE**
*******************
*
 .NL     TFM   PBUF+54,5553,8,     FILL PBUF
         AM    IC,2,10,
         TNF   PBUF+28,-IC,,
         BTM   OUTPUT,*+12,,       PRINT AND CLEAR PBUFF
         TF    COUNT,-IC,,         SET SKIP COUNT
 SKIPL   SPIM  ,1,,                SKIP A LINE
         SM    COUNT,1,10,         DEC COUNT
         CM    COUNT,0,10,         CHECK
         BE    BACK,,,             RETURN TO CONTROL IF ZERO
         B7    SKIPL,,,            OTHERWISE LOOP AGAIN
*
*****************
**SPACE ROUTINE**
*****************
*
 .SP     TFM   PBUF+54,6257,8,     FILL PBUF
         AM    IC,2,10,
         TNF   PBUF+28,-IC,,
         SF    SPFLG,,,            SET SPACE FLAG
         MM    -IC,2,10,
         BT    PBFMAN,99,,         INC PBUFF MARKER
         B7    BACK,,,             RETURN TO CONTROL
*
***************
**TAB ROUTINE**
***************
*
 .TAB    AM    IC,3,10,            INC IC
         TFM   PBUF+54,6342,8,     FILL PBUF
         TNF   PBUF+30,-IC,,
         CM    -IC,120,9,          CHECK TAB PARAMETER
         BL    *+32,,,             OK IF LESS THAN 120
         PRA   TBE1,,,             OTHERWISE ERROR
         B7    HALT,,,             HALT TO CHECK ERROR
         TFM   PBUFMK,PBUFF-2,,    RESET PBUFF MARKER
         A     PBUFMK,-IC,,
         A     PBUFMK,-IC,,
         B7    BACK,,,             RETURN TO CONTROL
*
*************************************
```

```
**SUBROUTINE TO PRINT AND CLEAR PBUFF**
**************************************
*
         DS    5,,,                    SPACE FOR RETURN ADDRESS
 OUTPUT  CM    PBUFMK,PBUFF-2,,        CHECK PBUFF CONTENTS
         BNH   -OUTPUT+1,,,            BRANCH BACK IF EMPTY
         PRAS  PBUFF,,,                OTHERWISE PRINT CONTENTS
         TFM   PBUFMK,PBUFF-2,,        RESET PBUFF MARKER
         TFM   P1,PBUFF-2,,            SET WORKING POINTER
 LOOP30  AM    P1,10,10,               CLEAR PBUFF
         TF    -P1,ZERO-8,,
         CM    P1,PBUFF+238,,
         BE    -OUTPUT+1,,,
         B7    LOOP30,,,
*
**********************************
**PRINT BUFFER MANAGER SUBROUTINE**
**********************************
*
         DS    4,,,                    SPACE FOR PARAMETER (PBUFMK INCREMENT)
 PBFMAN  A     PBUFMK,PBFMAN-1,,       INC BUF MARKER
         CM    PBUFMK,PBUFF+238,,      CHECK AGAINST BUFFER END
         BNH   *+68,,,
         BNF   WARN,ARRAY,,            WARNING IF LINE OFLO AND NO ARRAY FLAG
         TFM   OUTPUT-1,*+20,,
         B7    OUTPUT,,,               PRINT AND CLEAR PBUFF AND RESET BUF MAR
         SPIM  ,1,,                    SKIP A LINE
         BNF   PBFMAN,SPFLG,,
         CF    SPFLG,,,
         BB2   ,,,                     BRANCH BACK
 WARN    PRA   PRE1,,,
         B7    PBFMAN+48,,,
*
******************************
**INPUT-OUTPUT LOAD ROUTINES**
******************************
*
 RDSW    DC    3,0,,                   READ SWITCH
 PRSW    DS    ,RDSW-1,,               PRINT SWITCH
 PRCSW   DS    ,RDSW-2,,               PRINT CONTROL SWITCH
 .RD     BD    20500,RDSW,,            READ ROUTINE IN CORE IF RDSW ON
         TFM   RDSW,1,9,               OTHERWISE SET RDSW AND CALL IN
         CALL  LINK,MUSRD,20500
 .PR     BD    20500,PRSW,,            PRINT ROUTINE IN CORE IF PRSW ON
         TFM   RDSW,10,9,              OTHERWISE SET PRSW AND CALL IN
         CALL  LINK,MUSPR,20500
 .PRC    BD    20500,PRCSW,,           PRC ROUTINE IN CORE IF PRCSW ON
         TFM   RDSW,100,9,             OTHERWISE SET PRCSW AND CALL IN
         CALL  LINK,MUSPRC,20500
*
***************
**END ROUTINE**
***************
*
         DORG  17500,,,
 .END    TFM   PBUF+56,44,10,          FILL PBUF
         TFM   PBUF+54,4555,8,,
```

```
         BTM   OUTPUT,*+12,,        PRINT AND CLEAR PBUFF
         BNC2  *+24,,,
         PRA   PBUF,,,
         WATY  MUSEND,,,
         RCTY  ,,,
         CALL  EXIT,,,
 MUSEND  DAC   15,PHASE COMPLETE',,
*
**********************
**MAIN CONTROL ROUTINE**
**********************
*
         DORG  17800,,,
 BACK    BNC2  *+24,,,              PRINT PBUF IF SWITCH 2 ON
         PRA   PBUF,,,
         BNC3  START-12,,,          BYPASS IF SWITCH 3 OFF
         PRN   30001,,,             PRINT STACK
         PRN   30141,,,
         PRN   -SYMBAS,,,           PRINT SYMBOL TABLE
         PRN   PRINT,,,             PRINT VARIABLES
         PRN   PRINT+144,,,
         AM    IC,2,10,             INCR IC
 START   TNF   PBUF+10,IC,,
         TNF   PBUF+22,-IC,,
         TF    TABLE1-1,-IC,,       SET UP TABLE1 BRANCH ADDRESS
         CF    TABLE1-2,,,
         TF    PBUF+42,ZERO,,
         TFM   PBUF+58,0,8,
         B7    -TABLE1,,,           BRANCH
*
***************
**DECLARATIONS*****************************************************
***************
*
         DORG  18200,,,
 HALT    DS    ,,END+24,,
 PRINT   DS    ,,,,
 PBUFF   DAS   120,,,               PRINT BUFFER
         DAC   1,',,
 PBUFMK  DS    5,,,                 PRINT BUFFER MARKER
 P       DS    5,,,                 WORKING POINTER
 P1      DS    5,,,                 WORKING POINTER
 SPFLG   DC    2,0,,                SPACE FLAG
 FPAT    DC    17,-11111111011111111,,  FREE F-FORMAT PATTERN
 POINT   DS    5,,,                 WORKING POINTER
 I       DS    2,,,                 WORKING INDEX
 FALBUF  DC    10,4641536245,,      FALSE BUFFER
 TRUBUF  DC    10,6359644500,,      TRUE BUFFER
 BOOBUF  DS    5,,,                 BOOLEAN BUFFER POINTER
 REPFLG  DC    2,0,,                REPLICATE FLAG
 RCOUNT  DS    4,,,                 REPLICATE COUNT
 ICM     DS    5,,,                 INSTRUCTION COUNTER MARKER
 TOT     DS    3,,,                 TOTAL
 TOT2    DS    3,,,                 TWICE TOTAL
 REPCNT  DS    5,,,                 REP COUNT POINTER
 BEFORE  DS    3,,,                 BEFORE DECIMAL COUNT
 AFTER   DS    3,,,                 AFTER DECIMAL COUNT
```

```
XC      DS    3,,,          EXPONENT COUNT
SIGFLG  DC    2,0,,         SIGNIFICANCE FLAG
EFLG    DC    2,0,,         EXPONENT FLAG
NFLG    DC    2,0,,         NEGATIVE FLAG
PFLG    DC    2,0,,         PLUS FLAG
OFLG    DC    2,0,,         ZERO FLAG
XL      DS    5,,,          EXPONENT LENGTH
SBUFMK  DS    5,,,          STRING BUFFER MARKER
INFLAG  DC    2,0,,         INPUT FLAG
ARRAY   DC    2,0,,         ARRAY INPUT FLAG
RBUF    DAS   80,,,         READ BUFFER
RBUFMK  DS    5,,,          READ BUFFER MARKER
MCOUNT  DS    2,,,          MANTISSA COUNT
NUMBER  DS    5,,,
NACC    DC    10,0,,        NUMBER ACCUMULATOR
ACOUNT  DS    4,,,          ARRAY COUNT
EXP     DS    5,,,          EXPONENT BUILD POINTER
XAREA   DC    3,0,,         EXPONENT BUILD AREA
SAREA   DC    14,0,,        STRING BUILD AREA
STRING  DS    5,,,          STRING BUILD POINTER
IC      DS    5,,,          INSTRUCTION COUNTER
SVP     DS    5,,,          STACK VALUE POINTER
STP     DS    5,,,          STACK TYPE POINTER
TOPV    DS    5,,,          POINTER TO TOP STACK VALUE
TOPT    DS    5,,,          POINTER TO TOP STACK TYPE
LN      DC    5,0,,         LINE NUMBER
BOTV    DS    5,,,          BOTTOM VALUE
A       DS    5,,,          ABSOLUTE ADDRESS
RAW     DC    5,0,,,        RETURN ADDRESS WORD
AP      DS    2,,,          ACTUAL PARAMETERS
BVP     DS    5,,,          BASE VALUE POINTER
BTP     DS    5,,,          BASE TYPE POINTER
LVP     DS    5,,,          LOCAL VALUE POINTER
LTP     DS    5,,,          LOCAL TYPE POINTER
WVP     DS    5,,,,         WORKING VALUE POINTER
MFP     DS    5,,,          MAPPING FUNCTION POINTER
LENGTH  DS    5,,,          STRING LENGTH
NODE1   DC    5,30000,,     NODE POINTERS
NODE2   DC    5,30000,,             - LINK(C(NODE1)) = C(NODE2)
REF     DS    5,,,          REFERENCE NODE
AVAIL   DC    5,-0,,        AVAILABLE LIST POINTER
BOUND   DC    5,39999,,     LOWER STRING AREA BOUNDARY
TOPSTR  DS    5,,,          TOP STRING
BOTSTR  DS    5,,,          BOTTOM STRING
TOPREM  DS    2,,,          TOP STRING LENGTH REMAINDER
BOTREM  DS    2,,,          BOTTOM STRING LENGTH REMAINDER
DIF     DS    2,,,
TEMP    DS    12,,,         TEMPORARY STORE
COUNT   DC    2,0,,
SYMBAS  DC    5,25800,,     SYMBOL TABLE BASE
ARBASE  DS    ,2406,,       ARRAY BASE
ILBASE  DS    ,2411,,       INTERMEDIATE LANGUAGE BASE
STKBAS  DS    ,2416,,       STACK BASE
BASE    DS    5,,,
SIZE    DC    5,0,,         ARRAY SIZE
ACC     DC    5,0,,         ACCUMULATOR
DIM     DC    2,0,,         ARRAY DIMENSION
```

```
 W       DS    5,,,                    ADDRESS OF FIRST ARRAY ELT
 L0      DS    4,,,
 L1      DS    4,,,
 TABLE1  DC    5,-20000,,              BRANCHING TABLE 1
 TABLE2  DS    5,,                     BRANCHING TABLE 2
 COMP    DC    2,0,,                   COMPARE
 OP      DS    2,,,                    OPERATOR CODE
 IND     DC    2,0,,                   INDICATOR
         DC    7,0,
 MAC     DS    7,,                     MANTISSA ACCUMULATOR
         DC    15,0,,
 HOD     DS    5,,                     HIGH ORDER DIGIT
 LOD     DS    5,,                     LOW ORDER DIGIT
 EAC     DC    3,0,,                   EXPONENT ACCUMULATOR
 EP      DS    2,,,                    EXPONENT P
 EQ      DS    2,,,                    EXPONENT Q
 MQ      DS    7,,                     MANTISSA Q
 EDIF    DS    3,,                     EXPONENT DIFFERENCE
 UFLO    DAC   10,UNDERFLOW',,
 OFLO    DAC   9,OVERFLOW',,
 DIG     DC    2,0,,                   DIGIT
         DC    7,0,,
 FZERO   DC    2,0,,                   FLOATING POINT ZERO
         DC    7,1,,
 FONE    DC    2,0,,                   FLOATING POINT ONE
 PBUF    DAC   31,                                 ',,
 ZERO    DC    18,0,,,
 ALL9S   DC    7,9999999,,
 CORFLO  DAC   14,CORE OVERFLOW',,
 HEAD    DAC   28,  IC          INSTR          OP',,
*
**ERRORS**************************
*
 TRE1    DAC   6,TR E1',,
 TRE2    DAC   6,TR E2',,
 STE1    DAC   6,ST E1',,
 STE2    DAC   6,ST E2',,
 IFJE1   DAC   7,IFJ E1',,
 NOTE1   DAC   7,NOT E1',,
 NEGE1   DAC   7,NEG E1',,
 DIVE1   DAC   7,DIV E1',,
 IDIVE1  DAC   8,IDIV E1',,
 STRLE1  DAC   8,STRL E1',,
 CHECE1  DAC   9,CHECK E1',,
 CPE1    DAC   6,CP E1',,
 PEE1    DAC   6,PE E1',,
 MSFE1   DAC   7,MSF E1',,
 MSFE2   DAC   7,MSF E2',,
 MSFE3   DAC   7,MSF E3',,
 MSFE4   DAC   7,MSF E4',,
 INDE1   DAC   7,IND E1',,
 INDE2   DAC   7,IND E2',,
 INDE3   DAC   7,IND E3',,
 INDE4   DAC   7,IND E4',,
 INDE5   DAC   7,IND E5',,
 INDE6   DAC   7,IND E6',,
 TLE1    DAC   6,TL E1',,
```

```
 RTE1    DAC   6,RT E1',,
 RTE2    DAC   6,RT E2',,
 RFE1    DAC   6,RF E1',,
 CSE1    DAC   6,CS E1',,
 CSE2    DAC   6,CS E2',,
 CSE3    DAC   6,CS E3',,
 CSE4    DAC   6,CS E4',,
 CSE5    DAC   6,CS E5',,
 CSE6    DAC   6,CS E6',,
 CSWBE1  DAC   8,CSWB E1',,
 CSWBE2  DAC   8,CSWB E2',,
 CSWBE3  DAC   8,CSWB E3',,
 CSWBE4  DAC   8,CSWB E4',,
 STRE1   DAC   7,STR E1',,
 CATE1   DAC   7,CAT E1',,
 RDE1    DAC   6,RD E1',,
 RDE2    DAC   6,RD E2',,
 RDE3    DAC   6,RD E3',,
 RDE4    DAC   6,RD E4',,
 TBE1    DAC   6,TB E1',,
 PRE1    DAC   6,PR E1',,
 PRE2    DAC   6,PR E2',,
 PRE3    DAC   6,PR E3',,
 PRCE1   DAC   7,PRC E1',,
 PRCE2   DAC   7,PRC E2',,
 PRCE3   DAC   7,PRC E3',,
 PRCE4   DAC   7,PRC E4',,
*
**OP BRANCHING TABLE1**********************************
*
         DORG  20006,,,
         DSA   .LN,0,.TA,0,.TR,0,.TN,0,.TN0
         DSA   0,.TN1,0,.TB,0,.TS,0,.TNS,0,.IND
         DSA   0,.IND,0,.ST,0,.ST,0,0,0,.VI
         DSA   0,.UJ,0,.BA,0,.IFJ,0,.OP,0,.RT
         DSA   0,.RF,0,.RFWT,0,.RS,0,.CS,0,.CSWB
         DSA   0,.RAP,0,.RAF,0,.CP,0,0,0,.PE
         DSA   0,.TL,0,.TL,0,.MSF,0,.RE,0,.END
         DSA   0,.RD,0,.PR,0,.PRC,0,.NP,0,.NL
         DSA   0,.SP,0,.TAB
*
         DEND  BEGIN
```

```
*
********************
**INTERFACE ROUTINE**
********************
*
        DORG  20500,,,
 BEGIN  WATY  MUSGO,,,
        RCTY  ,,,
*
**PRINT DIAGNOSTIC HEADING*
*
        SKIP  ,1,,
        BNC2  SETUP,,,
        PRA   HEAD,,,                PRINT HEADING
        SPIM  ,3,,
*
**INITIALIZE**
*
 SETUP  TFM   OUTPUT-1,*+20,,
        B7    OUTPUT+36,,,           CLEAR PRINT BUFFER AND INITIALISE MARKE
        TFM   RBUFMK,RBUF+158,,      INITIALISE READ BUFFER MARKER
        CF    MAC+1,,,
        CF    IND-1,,,
        TF    IC,ILBASE,,            INITIALISE IC
        AM    IC,1,10,
        TF    SVP,STKBAS,,           SET STACK POINTERS
        TF    STP,STKBAS,,
        SM    STP,9,10,
*
**CLEAR ARRAY AREA**
*
        TF    P,ARBASE,,             INITIALIZE WORKING POINTER
 TEST85 C     P,ILBASE,,             TEST P AGAINST I.L. BASE
        BNL   START,,,               IF GE, BRANCH TO START INTERPRETING
        TDM   -P,0,,                 OTHERWISE SET ARRAY ELT TO UNDEFINED
        AM    P,10,10,               INC POINTER
        B7    TEST85,,,              AND LOOP AGAIN
 MUSGO  DAC   26,MUSSEL INTERPRETIVE PHASE',,
```

```
*
****************
**READ ROUTINE**
****************
*
         DORG  20500,,,
         TFM   PBUF+54,5944,8,      FILL PBUF
         TD    COMP,-STP,,
         CM    COMP,6,10,           CHECK TOP STACK TYPE
         BE    *+32,,,
         PRA   RDE1,,,              ERROR - NOT ADDRESS
         B7    HALT,,,              HALT TO CHECK ERROR
         TF    A,-SVP,,             SAVE ADDRESS
         SM    A,9,10,
         TD    COMP,-A,,
         CM    COMP,4,10,           CHECK ADDRESSED LOCATION CONTENT TYPE
         BNE   READIN-12,,,         BRANCH TO INPUT IF NOT ARRAY WORD
         BTM   ARSUB,READIN,,       OTHERWISE PREPARE FOR ARRAY INPUT
         AM    A,9,10,
 READIN  TFM   MCOUNT,0,10,         INITIALISE - MANTISSA COUNT
         TFM   EAC,0,9,                          EXPONENT ACCUMULATOR
         TFM   NUMBER,NACC-9,,                   MANTISSA BUILD POINTER
         TF    BOTV,SVP,,
         SM    BOTV,2,10,
         BTM   BUFMAN,1,10,         INC BUF MARKER
         BT    DIG7,RBUFMK,,        CHECK FOR DIGIT
         BE    NUM,,,               DIGIT - BRANCH TO ASSEMBLE AND STORE NU
         BTM   COMCHA,10,10,        OTHERWISE CHECK NEXT CHARACTER
         BE    READIN+36,,,         + - BACK TO READIN
         CM    -RBUFMK,20,10,
         BNE   *+32,,,
         SF    IND,,,               MINUS - STORE
         B7    READIN+36,,,         THEN BACK TO READIN
         BNR   *+20,-RBUFMK,,
         B7    STR,,,               STRING DELIMITER - BRANCH TO PACK STRIN
         CM    -RBUFMK,46,10,
         BE    BOO+20,,,            BOOLEAN FALSE - TO BOOLEAN ROUTINE
         CM    -RBUFMK,63,10,
         BE    BOO,,,               BOOLEAN TRUE - TO BOOLEAN ROUTINE
         CM    -RBUFMK,23,10,
         BNE   *+32,,,
         TDM   -STP,0,,             COMMA - NULL INPUT - SET TYPE UNDEFINED
         B7    INSTOR,,,                                 AND BRANCH TO STOR
         CM    -RBUFMK,0,10,
         BE    *-32,,,              BLANK - NULL INPUT
         TDM   -STP,2,,             STORE NUMBER TYPE
 DECHEC  CM    -RBUFMK,3,10,
         BE    REAL,,,              DECIMAL POINT - BRANCH TO REAL ROUTINE
         CM    -RBUFMK,45,10,
         BE    EXPO,,,              E - BRANCH TO EXPONENT ROUTINE
 INCHEC  PRA   RDE2,,,              ERROR - INCORRECT INPUT FORM
         B7    HALT,,,              HALT TO CHECK ERROR
*
**********************************
**ASSEMBLE AND STORE NUMBER ROUTINE**
**********************************
*
```

```
NUM     TDM   -STP,2,,          STORE NUMBER TYPE
LOOP20  AM    RBUFMK,1,10,      LOOP TO SKIP OVER NON SIG ZEROS
        BD    OUT20,-RBUFMK,,   1ST SIG DIG - BRANCH OUT
        BTM   BUFMAN,1,10,      INC BUF MARKER
        BT    DIG7,RBUFMK,,     CHECK FOR DIGIT
        BE    LOOP20,,,         LOOP AGAIN IF DIGIT
ZERO    BTM   COMCHA,3,10,      OTHERWISE CHECK NEXT CHARACTER
        BE    REAL,,,           DECIMAL POINT - BRANCH TO REAL
        CM    -RBUFMK,45,10,
        BE    SKIPEE,,,         E - SKIP EXPONENT PART
        CM    -RBUFMK,23,10,
        BE    PACK0,,,
        CM    -RBUFMK,0,10,
        BNE   INCHEC,,,         ERROR IF NOT COMMA OR BLANK
PACK0   TFL   -SVP,FZERO,,      OTHERWISE PACK ZERO
        B7    INSTOR,,,         THEN BRANCH TO STORE
SKIPEE  BTM   BUFMAN,1,10,      SKIP OVER EXPONENT PART
        BTM   COMCHA,23,10,     CHECK NEXT CHARACTER
        BE    PACK0,,,          PACK ZERO IF COMMA
        CM    -RBUFMK,0,10,
        BE    PACK0,,,          PACK ZERO IF BLANK
        B7    SKIPEE,,,         OTHERWISE LOOP AGAIN
        AM    RBUFMK,1,10,      INC BUF MARKER
OUT20   AM    NUMBER,1,10,      PACK NEXT DIGIT
        TD    -NUMBER,-RBUFMK,,
        CF    -NUMBER,,,
        AM    MCOUNT,1,10,      INC MANTISSA COUNT
        BTM   BUFMAN,1,10,      INC BUF MARKER
        CM    MCOUNT,8,10,      CHECK MANTISSA COUNT
        BE    LOOP21,,,         IF 8 IGNORE REMAINING DIGITS
        BT    DIG7,RBUFMK,,     OTHERWISE CHECK FOR DIGIT
        BE    OUT20-12,,,       LOOP AGAIN IF DIGIT
        B7    OUT21,,,          OTHERWISE TERMINATE INTEGER PHASE
LOOP21  BT    DIG7,RBUFMK,,     CHECK FOR DIGIT
        BNE   OUT21,,,          TERMINATE INTEGER PHASE IF NOT
        BTM   BUFMAN,2,10,      OTHERWISE INC BUF MARKER
        AM    EAC,1,10,         INC EXPONENT ACCUMULATOR
        B7    LOOP21,,,         AND LOOP AGAIN
OUT21   BTM   COMCHA,23,10,     TERMINATE INTEGER PHASE - CHK NEXT CHAR
        BE    PACKIN,,,
        CM    -RBUFMK,0,10,
        BNE   DECHEC,,,         CHECK FOR DECIMAL IF NOT COMMA OR BLANK
PACKIN  CM    MCOUNT,8,10,      OTHERWISE PACK INTEGER - CHECK MCOUNT
        BL    INT,,,            STORE INTEGER IF LESS THAN 8
        PRA   RDE4,,,           OTHERWISE PRINT WARNING ERROR
        TF    MAC+1,-NUMBER,,   THEN ROUND AND STORE REAL
        AM    EAC,1,10,
        TFM   HOD,MAC-6,,
        TFM   LOD,MAC,,
        SF    INFLAG,,,         SET INPUT FLAG
        B7    RNS,,,            BRANCH TO ROUND AND NORMALISE
INSTOR  BT    STOR,SVP,,        BRANCH TO STORE
        B7    ARCHEC,,,         BRANCH TO ARRAY CHECK
INT     TFL   -SVP,FZERO,,      STORE INTEGER
        A     -BOTV,-NUMBER,,   PACK MANTISSA
        MF    -BOTV,IND,,       APPEND SIGN
        B7    INSTOR,,,         BRANCH TO STORE
```

```
*
**********************************
**ASSEMBLE AND STORE REAL ROUTINE**
**********************************
*
 REAL    CM    MCOUNT,0,10,          CHECK MANTISSA COUNT AGAINST 0
         BNE   OUT22+60,,,           IF ZERO SKIP OVER LEADING ZEROS
 LOOP22  BTM   BUFMAN,1,10,          INC BUF MARKER
         BT    DIG7,RBUFMK,,         CHECK FOR DIGIT
         BNE   ZERO,,,               EXIT IF NOT
         AM    RBUFMK,1,10,          OTHERWISE CHECK DIGIT
         BD    OUT22,-RBUFMK,,       BRANCH OUT IF NON ZERO
         SM    EAC,1,10,             OTHERWISE DEC EXPONENT ACCUMULATOR
         B7    LOOP22,,,             AND LOOP AGAIN
 LOOP23  CM    MCOUNT,8,10,          CHECK MANTISSA COUNT AGAINST 8
         BE    SKIPO,,,              IF EQUAL SKIP OVER REMAINING DIGITS
 OUT22   AM    NUMBER,1,10,          OTHERWISE PACK NEW DIGIT
         AM    MCOUNT,1,10,
         TD    -NUMBER,-RBUFMK,,
         CF    -NUMBER,,,
         SM    EAC,1,10,
         BTM   BUFMAN,1,10,          INC BUF MARKER
         BT    DIG7,RBUFMK,,         CHECK FOR DIGIT
         BNE   ENDRE,,,              IF NOT TERMINATE REAL PHASE
         AM    RBUFMK,1,10,          OTHERWISE INC BUF MARKER
         B7    LOOP23,,,             AND LOOP AGAIN
 SKIPO   BTM   BUFMAN,1,10,          IGNORE REMAINING DIGITS
         BT    DIG7,RBUFMK,,         CHECK FOR DIGIT
         BNE   ENDRE,,,              IF NOT TERMINATE REAL PHASE
         BTM   BUFMAN,2,10,          OTHERWISE INC BUF MARKER
         B7    SKIPO+12,,,           AND LOOP AGAIN
 ENDRE   BTM   COMCHA,45,10,         TERMINATE REAL PHASE - CHECK NEXT CHAR
         BE    EXPO,,,               IF E BRANCH TO EXPONENT ROUTINE
 COMCHK  CM    -RBUFMK,23,10,        IF COMMA OR BLANK PACK REAL
         BE    PACKRE,,,
         CM    -RBUFMK,0,10,
         BNE   INCHEC,,,             OTHERWISE ERROR
 PACKRE  CM    MCOUNT,8,10,          PACK REAL
         BE    PACKIN+24,,,
         TF    MAC,FZERO-2,,
         A     MAC,-NUMBER,,
         TFM   HOD,MAC-6,,
         TFM   LOD,MAC,,
         SF    INFLAG,,,             SET INPUT FLAG
         B7    NORM,,,               BRANCH TO NORMALISE AND STORE ROUTINE
*
********************
**EXPONENT ROUTINE**
********************
*
 EXPO    TFM   EXP,XAREA-2,,         SET EXPONENT BUILD POINTER
         BTM   BUFMAN,1,10,          INC BUF MARKER
         BT    DIG7,RBUFMK,,         CHECK FOR DIGIT
         BE    LOOP24,,,             BRANCH AROUND IF DIGIT
         BTM   COMCHA,10,10,         OTHERWISE CHECK CHARACTER
         BE    EXPO+12,,,            IGNORE +
         CM    -RBUFMK,20,10,
```

```
         BNE   INCHEC,,,          ERROR IF NOT MINUS
         SF    IND-1,,,           OTHERWISE STORE MINUS SIGN
         B7    EXPO+12,,,         THEN BRANCH BACK TO LOOK FOR DIGIT
 LOOP24  AM    RBUFMK,1,10,       LOOP TO SKIP OVER LEADING (NS) ZEROS
         BD    LOOP25,-RBUFMK,,   BRANCH OUT ON SIG DIG
         BTM   BUFMAN,1,10,       OTHERWISE INC BUF MARKER
         BT    DIG7,RBUFMK,,      CHECK FOR DIGIT
         BE    LOOP24,,,          LOOP AGAIN IF DIGIT
         CF    IND-1,,,           OTHERWISE CHECK NEXT CHARACTER
         BTM   COMCHA,23,10,
         B7    COMCHK+12,,,
 LOOP25  AM    EXP,1,10,          INC EXPONENT COUNTER
         TD    -EXP,-RBUFMK,,     PACK LATEST EXPONENT DIGIT
         CF    -EXP,,,
         BTM   BUFMAN,1,10,       INC BUF MARKER
         BT    DIG7,RBUFMK,,      CHECK FOR DIGIT
         BNE   OUT25,,,           IF NOT TERMINATE EXPONENT PHASE
         AM    RBUFMK,1,10,       OTHERWISE INC BUF MARKER
         CM    EXP,XAREA+1,,,     CHECK EXPONENT SIZE
         BNE   LOOP25,,,
         PRA   RDE3,,,            ERROR - EXPONENT OVERFLOW
         B7    HALT,,,            HALT TO CHECK ERROR
 OUT25   BTM   COMCHA,23,10,      TERMINATE EXPO PHASE - CHECK NEXT CHAR
         BE    *+36,,,
         CM    -RBUFMK,0,10,
         BNE   INCHEC,,,          ERROR IF NOT COMMA OR BLANK
         MF    -EXP,IND-1,,       OTHERWISE APPEND EXPONENT SIGN
         A     EAC,-EXP,,
         CM    MCOUNT,0,10,       CHECK MANTISSA COUNT
         BNE   PACKRE,,,          IF NON ZERO BRANCH TO PACK REAL
         TFL   -SVP,FONE,,        OTHERWISE PACK MANTISSA VALUE ONE
         A     -SVP,EAC,,         PACK EXPONENT VALUE
         MF    -BOTV,IND,,        APPEND MANTISSA SIGN
         B7    INSTOR,,,          BRANCH TO STORE
*
**********************
**PACK BOOLEAN ROUTINE**
**********************
*
  BOO    TFM   -SVP,1,10,         PACK VALUE TRUE
         B7    *+20,,,
         TFM   -SVP,0,10,         PACK VALUE FALSE
         TDM   -STP,3,,           PACK BOOLEAN TYPE DIGIT
 BOLOOP  BTM   BUFMAN,1,10,       INC BUF MARKER
         BTM   COMCHA,23,10,      CHECK NEXT CHARACTER
         BE    INSTOR,,,          BRANCH TO STORE IF COMMA
         CM    -RBUFMK,0,10,
         BE    INSTOR,,,          BRANCH TO STORE IF BLANK
         B7    BOLOOP,,,          OTHERWISE LOOP AGAIN
*
**********************
**PACK STRING ROUTINE**
**********************
*
  STR    TDM   -STP,7,,           STORE STRING DESCRIPTOR TYPE DIGIT
         BTM   GRAB,0,10,         GRAB A NODE
         TF    -SVP,NODE2,,       PACK ADDRESS FIELD IN STRING DESCRIPTOR
```

```
         TFM   LENGTH,0,8,          SET STRING LENGTH COUNTER TO ZERO
 LOOP27  TFM   STRING,SAREA-14,,    SET STRING ASSEMBLY POINTER
         TFM   COUNT,6,10,          SET LOOP COUNTER
 LOOP28  BTM   BUFMAN,1,10,         INC BUF MARKER
         SF    -RBUFMK,,,
         AM    RBUFMK,1,10,
         BNR   PAST,-RBUFMK,,       CHECK NEXT CHARACTER - BYPASS IF NOT RM
         BTM   BUFMAN,1,10,         OTHERWISE CHECK NEXT CHARACTER
         BTM   COMCHA,23,10,
         BE    OUT27,,,             TERMINATE STRING INPUT IF COMMA
         CM    -RBUFMK,0,10,
         BE    OUT27,,,             TERMINATE STRING INPUT IF BLANK
         BNR   INCHEC,-RBUFMK,,     ERROR IF NOT STRING DELIMITER
         TFM   -RBUFMK,12,10,       OTHERWISE CONVERT TO INTERNAL FORM
 PAST    AM    STRING,3,10,         OTHERWISE PACK CHARACTER
         TF    -STRING,-RBUFMK,,
         SM    STRING,1,10,
         CF    -STRING,,,
         AM    LENGTH,1,10,         INC LENGTH COUNTER
         SM    COUNT,1,10,          DEC LOOP COUNTER
         BD    LOOP28,COUNT,,       EXIT IF COUNT ZERO
         SM    NODE2,4,10,          STORE STRING CHUNK
         AM    STRING,1,10,
         SF    SAREA-12,,,
         TF    -NODE2,-STRING,,
         AM    NODE2,4,10,
         TF    NODE1,NODE2,,        SET LINK TO NEXT NODE
         BTM   GRAB,0,10,
         SF    NODE2-3,,,
         TF    -NODE1,NODE2,,
         CF    NODE2-3,,,
         B7    LOOP27,,,            LOOP AGAIN
 OUT27   CM    LENGTH,0,10,         CHECK STRING LENGTH
         BNE   NONNUL,,,
         TDM   -STP,1,,             PACK NULL STRING TYPE IF ZERO
         B7    INSTOR,,,            THEN BRANCH TO STORE
 NONNUL  SM    SVP,5,10,            OTHERWISE TERMINATE STRING
         TF    -SVP,LENGTH,,        SET LENGTH FIELD IN STRING DESCRIPTOR
         AM    SVP,5,10,
         CM    COUNT,6,10,
         BNE   *+44,,,              CHECK LATEST CHUNK
         SF    -NODE1,,,            IF ZERO TERMINATE AT NODE1
         BTM   RETURN,0,10,         RETURN NODE2 TO AVAIL
         B7    INSTOR,,,            BRANCH TO STORE
         SF    -NODE2,,,            OTHERWISE TERMINATE AT NODE2
         SM    NODE2,4,10,          STORE FINAL STRING CHUNK
         AM    STRING,1,10,
         SF    SAREA-12,,,
         TF    -NODE2,-STRING,,
         B7    INSTOR,,,            BRANCH TO STORE
*
**********************
**ARRAY CHECK ROUTINE**
**********************
*
 ARCHEC  BNF   POPO,ARRAY,,         RETURN TO CONTROL IF NO ARRAY INPUT
         AM    ACOUNT,1,10,         OTHERWISE INC ARRAY ELT COUNT
```

```
         C     ACOUNT,SIZE,,        CHECK AGAINST ARRAY SIZE
         BNH   *+32,,,
         CF    ARRAY,,,             TERMINATE ARRAY INPUT IF HIGH
         B7    POPO,,,
         AM    A,10,10,             OTHERWISE INC A
         B7    READIN,,,            AND BRANCH TO INPUT NEXT ELEMENT
 POPO    BTM   POPTOP,1010,8,       POP TYPE AND VALUE
         B7    BACK,,,              RETURN TO CONTROL
*
**********************************
**READ BUFFER MANAGER SUBROUTINE**
**********************************
*
         DS    2,,,                 SPACE FOR PARAMETER (RBUFMK INCREMENT)
 BUFMAN  A     RBUFMK,BUFMAN-1,,    INC BUF MARKER
         CM    RBUFMK,RBUF+158,,    CHECK AGAINST BUFFER END
         BNH   *+36,,,
         RACD  RBUF,,,              READ IN NEW CARD IF HIGH
         SM    RBUFMK,160,9,        RESET BUF MARKER
         BB2   ,,,                  BRANCH BACK
*
**********************************************
**SUBROUTINE TO COMPARE NEXT RBUF CHARACTER**
**********************************************
*
         DS    2,,,                 SPACE FOR PARAMETER (COMPARE CHARACTER)
 COMCHA  SF    -RBUFMK,,,           SET FLAG
         AM    RBUFMK,1,10,         INC RBUF MARKER
         CF    -RBUFMK,,,
         C     -RBUFMK,COMCHA-1,,   COMPARE
         BB2   ,,,                  BRANCH BACK
*
************************************************************
**SUBROUTINE TO PREPARE FOR ARRAY INPUT OR OUTPUT**
************************************************************
*
         DS    5,,,                 SPACE FOR BRANCH OUT ADDRESS
 ARSUB   SF    ARRAY,,,             SET ARRAY FLAG
         TFM   ACOUNT,1,8,          SET ARRAY ELT COUNTER
         AM    A,1,10,
         TD    COMP,-A,,
         BNF   *+44,COMP,,          SET BASE
         TF    BASE,STKBAS,,
         CF    COMP,,,
         B7    *+20,,,
         TF    BASE,SYMBAS,,
         AM    A,4,10,
         TF    W,BASE,,             SET W TO ADDRESS 1ST ARRAY ELT
         A     W,-A,,
         AM    A,4,10,
         TF    A,-A,,               SET A TO ADDRESS SMF
         CF    A-3,,,
         TDM   A-4,0,11,
         A     A,BASE,,
         AM    A,4,10,
         TF    SIZE,-A,,,           CALCULATE ARRAY SIZE
         SM    A,4,10,
```

```
         S     SIZE,-A,,
         AM    SIZE,1,10,
         CM    COMP,1,10,
         BNE   *+60,,,
         AM    A,16,10,
         M     SIZE,-A,,
         SF    96,,,
         TF    SIZE,99,,,
         TF    A,W,,                  SET A TO ADDRESS 1ST ARRAY ELT
         B7    -ARSUB+1,,,
*
*****************************************************
**SUBROUTINE TO CHECK NEXT RBUF DIGIT AGAINST 7**
*****************************************************
*
         DS    5,,,                   SPACE FOR BRANCH OUT ADDRESS
 DIG7    TD    COMP,-DIG7+1,,         TRANSMIT DIGIT TO COMP
         CF    COMP,,,                CLEAR (POSSIBLE) FLAG
         CM    COMP,7,10,             COMPARE WITH 7
         BB2   ,,,                    BRANCH BACK
```

```
*
*****************
**PRINT ROUTINE**
*****************
*
         DORG  20500,,,
         TFM   PBUF+54,5759,8,      FILL PBUF
         TF    ICM,IC,,             MARK IC
         TFM   COUNT,1,10,
         TF    TOPV,SVP,,
         TF    IC,ICM,,
CHOICE   TD    COMP,-STP,,
         CM    COMP,1,10,           CHECK TOP STACK TYPE
         BE    NULL1,,,             NULL STRING
         CM    COMP,2,10,
         BE    NUM1,,,              NUMBER
         CM    COMP,3,10,
         BE    BOOL1,,,             BOOLEAN
         CM    COMP,6,10,
         BNE   *+24,,,
         BTM   BUBBLE,CHOICE-12,,   ADDRESS - BUBBLE ADDRESSED TYPE + VALUE
         CM    COMP,7,10,
         BE    STR1,,,              STRING
         PRA   PRE2,,,              ERROR - INVALID STACK TYPE
         B7    HALT,,,              HALT TO CHECK ERROR
NULL1    SF    SPFLG,,,             SET SPACE FLAG
         BTM   PBFMAN,8,10,         INC BUF MARKER
         B7    BACHEC,,,            THEN BRANCH TO BACK CHECK
NUM1     BTM   LSUB,PRINTO,,        FIND MANTISSA LENGTH
         BTM   PBFMAN,32,10,        CLEAR SPACE FOR NUMBER
         A     LENGTH,-SVP,,        CHECK NUMBER FORM
         CM    LENGTH,7,10,
         BH    EFORM,,,
         CM    -SVP,7,1011,
         BL    EFORM,,,
         CF    SIGFLG,,,            CLEAR SIG FLAG
FFORM    TFM   POINT,FPAT-16,,      SET PATTERN POINTER
         TFM   BEFORE,8,10,
         SM    PBUFMK,30,10,
LOOP41   SM    BEFORE,1,10,         DEC BEFORE COUNT
         AM    POINT,1,10,          INC PATTERN POINTER
         BD    IFFLAG,-POINT,,      CHECK PATTERN DIGIT
         BNF   SIGON1,SIGFLG,,      0 - INDICATES DECIMAL POINT
         AM    PBUFMK,2,10,
         CM    -SVP,0,10,
         BE    LOOP41+12,,,         NO DECIMAL POINT FOR INTEGER
         TFM   -PBUFMK,3,10,
         B7    LOOP41+12,,,
SIGON1   SF    SIGFLG,,
         TFM   -PBUFMK,70,10,
         BNF   LOOP41+48,IND,,
         TFM   -PBUFMK,2070,8,
         B7    LOOP41+48,,,
IFFLAG   BNF   *+20,-POINT,,
         B7    NULL1,,,             FLAGGED 1 - TERMINATES PATTERN
         C     BEFORE,LENGTH,,      UNFLAGGED 1 - DIGIT POSITION
         BNH   *+56,,,
```

```
         AM    PBUFMK,2,10,
         BNF   LOOP41,SIGFLG,,
         TFM   -PBUFMK,70,10,
         B7    LOOP41,,,
         BNF   SIGON2,SIGFLG,,
BUFINC   AM    PBUFMK,2,10,
         C     P,BOTV,,
         BNH   *+56,,,
         CM    BEFORE,0,10,
         BNH   LOOP41,,,
         TFM   -PBUFMK,70,10,
         B7    LOOP41,,,
         TFM   -PBUFMK,70,10,
         TD    -PBUFMK,-P,,
         AM    P,1,10,
         B7    LOOP41,,,
SIGON2   SF    SIGFLG,,,
         BNF   BUFINC,IND,,,
         TFM   -PBUFMK,20,10,
         B7    BUFINC,,,
PRINTO   BTM   PBFMAN,16,10,        INC BUF MARKER
         TFM   -PBUFMK,70,10,       INSERT ZERO
         SF    SPFLG,,,             SET SPACE FLAG
         BTM   PBFMAN,24,10,        INC BUF MARKER
         B7    BACHEC,,,            THEN BRANCH TO BACK CHECK
EFORM    SM    PBUFMK,22,10,
         S     LENGTH,-SVP,,
         SM    LENGTH,1,10,
         TFM   EAC,0,9,             SET EXPONENT ACCUMULATOR
         A     EAC,-SVP,,
         A     EAC,LENGTH,,
         BD    *+24,EAC-2,,
         SF    EAC-1,,,
         TFM   I,6,10,              SET LOOP COUNT
         TFM   -PBUFMK,70,10,
         BNF   *+24,IND,,
         TFM   -PBUFMK,2070,8,      FILL SIGN IF MINUS
         TD    -PBUFMK,-P,,         FILL FIRST DIGIT
         AM    P,1,10,              INC MANTISSA POINTER
         AM    PBUFMK,2,10,
         TFM   -PBUFMK,3,10,        FILL DECIMAL POINT
LOOP42   AM    PBUFMK,2,10,         LOOP TO FILL REMAINING DIGIT POSITIONS
         TFM   -PBUFMK,70,10,
         C     P,BOTV,,
         BH    *+36,,,
         TD    -PBUFMK,-P,,
         AM    P,1,10,
         SM    I,1,10,
         BD    LOOP42,I,,
         AM    PBUFMK,4,10,         FILL EXPONENT PART
         TFM   -PBUFMK,4510,8,
         BNF   *+36,EAC,,
         CF    EAC,,,
         TFM   -PBUFMK,20,10,
         AM    PBUFMK,4,10,
         TNF   -PBUFMK,EAC,,
         B7    NULL1,,,
```

```
BOOL1   BTM   PBFMAN,10,10,        CLEAR SPACE FOR BOOLEAN VALUE
        BD    *+32,-SVP,,          CHECK VALUE
        TF    -PBUFMK,FALBUF,,     INSERT FALSE
        B7    NULL1,,,
        TF    -PBUFMK,TRUBUF,,     INSERT TRUE
        B7    NULL1,,,             THEN BRANCH TO SPACE AND BACK CHECK
STR1    BTM   STREM,0,10,          FIND STRING LENGTH REMAINDER
        AM    SVP,5,10,
        B7    TEST45,,,            BRANCH TO TEST FIRST NODE LINK
LOOP45  SM    NODE1,4,10,          DEC NODE POINTER
        BTM   PBFMAN,12,10,        INC PBUFF MARKER
        TF    -PBUFMK,-NODE1,,     FILL NODE VALUE
        AM    NODE1,4,10,          SET NODE1 TO LINK(NODE1)
        TF    NODE1,-NODE1,,
        CF    NODE1-3,,,
TEST45  BNF   LOOP45,-NODE1,,      LOOP AGAIN IF NOT TERMINAL NODE
        BT    PBFMAN,TOPREM,,      OTHERWISE INC BUF MARKER
        SM    NODE1,4,10,          DEC NODE POINTER
        TF    -PBUFMK,-NODE1,,     FILL TERMINAL NODE VALUE
        B7    NULL1,,,
```

```
*
**************************
**PRINT CONTROL ROUTINE**
**************************
*
         DORG 20500,,,
         AM   IC,2,10,
         TFM  PBUF+54,5759,8,      FILL PBUF
         TFM  PBUF+56,43,10,
         TNF  PBUF+28,-IC,,
         TF   ICM,IC,,             MARK IC
         TF   COUNT,-IC,,          SET MULT SPEC COUNT
         TF   TOPV,SVP,,           MARK STACK TOP
         BTM  POPTOP,1010,811,     SET STACK POINTERS TO FIRST O/P ELEMENT
         S    SVP-1,COUNT,,
         S    STP-1,COUNT,,
         TF   IC,ICM,,
 CHOOSE  TD   COMP,-STP,,          CHECK ELT TYPE
         CM   COMP,1,10,
         BE   NULL2,,              NULL STRING
         CM   COMP,2,10,
         BE   NUM2,,,              NUMBER
         CM   COMP,3,10,
         BE   BOOL2,,,             BOOLEAN
         CM   COMP,6,10,
         BNE  *+24,,,
         BTM  BUBBLE,CHOOSE-12,,   ADDRESS - BUBBLE ADDRESSED TYPE + VALUE
         CM   COMP,7,10,
         BE   STR2,,,              STRING
         PRA  PRCE1,,,             ERROR - INVALID STACK TYPE
         B7   HALT,,,              HALT TO CHECK ERROR
 NUM2    TFM  BEFORE,0,9,          SET BEFORE,AFTER,AND EXPONENT COUNTS
         TFM  AFTER,0,9,
         TFM  XC,0,9,
         TF   PFLG,ZERO-10,,       CLEAR EFLG,PFLG,NFLG,SIGFLG
         TFM  P,BEFORE,,           PRELIMINARY FORMAT SCAN
 INCINC  BTM  INCI,*+12,,          INC IC
         CM   -IC,42,10,           CHECK NEXT PIC CHARACTER
         BE   INCINC,,,            B - SKIP OVER
         CM   -IC,62,10,
         BE   INCINC,,,            S - SKIP OVER
         CM   -IC,23,10,
         BE   INCINC,,,            , - SKIP OVER
         CM   -IC,20,10,
         BE   INCINC,,,            MINUS - SKIP OVER
         CM   -IC,3,10,
         BNE  *+32,,,
         TFM  P,AFTER,,            DECIMAL POINT - SWITCH TO AFTER COUNT
         B7   INCINC,,,
         CM   -IC,14,10,
         BNE  *+44,,,
         BTM  KOUNT,14,10,         ASTERISK - COUNT
         A    -P,TOT,,,            ADD TO TOTAL
         B7   INCINC+12,,,
         CM   -IC,79,10,
         BNE  *+32,,,
         BTM  KOUNT,79,10,         9 - COUNT
```

```
         B7    *-56,,,
         CM    -IC,45,10,
         BNE   *+44,,,
         SF    EFLG,,,              E - SET EXPONENT FLAG
         TFM   P,XC,,               SWITCH TO EXPONENT COUNT
         B7    INCINC,,,
         BNR   *+32,-IC,,,          RECORD MARK TERMINATES PIC
         BNF   FMAT,EFLG,,,         IF NO EFLG BRANCH TO F-FORMAT
         B7    EMAT,,,              OTHERWISE BRANCH TO E -FORMAT
PICERR   PRA   PRCE2,,,             ERROR - INVALID PIC CHARACTER
         B7    HALT,,,              HALT TO CHECK ERROR
FMAT     BTM   LSUB,LNGTHO,,        FIND MANTISSA LENGTH
         A     LENGTH,-SVP,,        ADD EXPONENT
         C     LENGTH,BEFORE,,      CHECK AGAINST BEFORE COUNT
         BNH   *+32,,,
FERR     PRA   PRCE3,,,             ERROR IF HIGH
         B7    HALT,,,              HALT TO CHECK ERROR
         A     LENGTH,AFTER,,       OTHERWISE ADD DECIMAL PART LENGTH
         A     BEFORE,AFTER,,,
         CM    LENGTH,0,10,         COMPARE WITH ZERO
         BL    LNGTHO,,,            ZERO IF LOW
         SF    AFTER,,,
         C     -SVP,AFTER,,
         BNL   RESCAN,,,
         A     BOTV,-SVP,,          ROUND IF NECESSARY
         S     BOTV,AFTER,,
         BTM   ADD5,0,10,
         BD    *+92,-P,,
         AM    P,1,10,
         C     BOTV,P,,
         BNL   RESCAN,,,
LNGTHO   TFM   LENGTH,1,9,          ASSEMBLE ZERO
         TF    P,BOTV,,
         TDM   -P,0,,
         B7    RESCAN,,,            BRANCH TO RESCAN PICTURE
         AM    LENGTH,1,10,
         C     LENGTH,BEFORE,,
         BH    FERR,,,
         B7    RESCAN,,,
EMAT     SF    OFLG,,,
         BTM   LSUB,LNGTHO,,        FIND MANTISSA LENGTH
         CF    OFLG,,,
         TF    EAC,LENGTH,,         SET EXPONENT PART
         A     EAC,-SVP,,
         S     EAC,BEFORE,,
         A     BEFORE,AFTER,,       BEFORE IS NOW TOTAL OUTPUT MANTISSA LEN
         C     LENGTH,BEFORE,,
         BNH   ELNGTH-12,,,         ROUND MANTISSA IF NECESSARY
         S     BOTV,LENGTH,,
         A     BOTV,BEFORE,,
         BTM   ADD5,0,10,
         BD    *+32,-P,,
         AM    P,1,10,
         B7    ELNGTH-12,,,
         SM    BOTV,1,10,
         AM    EAC,1,10,
         TF    LENGTH,BEFORE
```

```
ELNGTH  CM    EAC,0,10,          FIND EXPONENT LENGTH
        BNE   LOOP70-12,,,
        TFM   XL,1,9,
        TF    POINT,SVP,,
        TDM   -POINT,0,,
        B7    RESCAN,,,
        TFM   POINT,EAC-2,,
LOOP70  BD    OUT70,-POINT,,
        AM    POINT,1,10,
        B7    LOOP70,,,
OUT70   TFM   XL,EAC,,
        S     XL,POINT,,
        AM    XL,1,10,
        C     XL,XC,,            CHECK FOR FIT
        BNH   RESCAN,,,          OK - BRANCH TO RESCAN PICTURE
        PRA   PRCE4,,,           ERROR - EXPONENT TOO LARGE
        B7    HALT,,,            HALT TO CHECK ERROR
RESCAN  TF    IC,ICM,,           RESCAN PIC - RESET IC
        AM    IC,2,10,
        CM    -IC,62,10,         CHECK FIRST CHARACTER
        BNE   CHECM,,,
        BNF   *+32,IND,,         S - FLOATING SIGN
        SF    NFLG,,,            IF NEGATIVE NUMBER SET NFLG
        B7    CHECHA-24
        SF    PFLG,,,            OTHERWISE SET PFLG
        B7    CHECHA-24,,,
CHECM   CM    -IC,20,10,
        BNE   CHECHA,,,
        BNF   *+24,IND,,         FLOATING MINUS
        SF    NFLG,,,            SET NFLG IF NEGATIVE NUMBER
        BTM   PBFMAN,2,10,       INC BUF MARKER
        AM    IC,2,10,           INC IC
CHECHA  CM    -IC,62,10,         CHOICE ON NEXT CHARACTER
        BNE   MCHEC,,,
        BTM   PBFMAN,2,10,       S - INSERT SIGN
        BNF   *+32,IND,,
        TFM   -PBUFMK,20,10,
        B7    CHECHA-12,,,
        TFM   -PBUFMK,10,10,
        B7    CHECHA-12,,,
MCHEC   CM    -IC,20,10,
        BNE   CHECOM,,,
        BTM   PBFMAN,2,10,       MINUS - INSERT SIGN IF NEG NUMBER
        BNF   CHECHA-12,IND,,
        B7    MCHEC-40,,,
CHECOM  CM    -IC,23,10,
        BNE   CHECAS,,,
        BTM   PBFMAN,2,10,       COMMA - INSERT IF SIGFLG ON
        BNF   CHECHA-12,SIGFLG,,
        TFM   -PBUFMK,23,10,
        B7    CHECHA-12,,,
CHECAS  CM    -IC,14,10,
        BNE   CHEC9,,,
        C     BEFORE,LENGTH,,    ASTERISK - COMPARE BEFORE WITH LENGTH
        BNH   *+68,,,
        SM    BEFORE,1,10,       DEC BEFORE IF HIGH
        BNF   *+24,SIGFLG,,      IF SIG FLAG
```

```
        BTM   FLOATS,CHECHA-12,,   CHECK FOR FLOATING SIGN AND INSERT ZERO
        BTM   PBFMAN,2,10,         OTHERWISE LEAVE BLANK
        B7    CHECHA-12,,,
        BTM   FLOATS,*+12,,        IF BEFORE LE LENGTH CHECK FOR FLOATING
        SF    SIGFLG,,,            SET SIGNIFICANCE FLAG
        C     P,BOTV,,,
        BH    CHECHA-12,,,         IF P GT BOTV INSERT ZERO
        TD    -PBUFMK,-P,          OTHERWISE INSERT NEXT DIGIT
        AM    P,1,10,              AND INC P
        B7    CHECHA-12,,,
CHEC9   CM    -IC,79,10,
        BNE   CHECB,,,
        SF    SIGFLG,,,            9 - SET SIG FLAG
        B7    CHECAS+24,,,         BRANCH TO ASTERISK ROUTINE
CHECB   CM    -IC,42,10,
        BNE   DOT,,,
        BTM   KOUNT,42,10,         B - COUNT
        BT    PBFMAN,TOT2,,        INC PBUF MARKER
        B7    CHECHA,,,
DOT     CM    -IC,3,10,
        BNE   *+56,,,
        SF    SIGFLG,,,            DECIMAL POINT - SET SIG FLAG
        BTM   FLOATS,*+12,,        FLOATING SIGN CHECK
        TFM   -PBUFMK,3,10,        INSERT DECIMAL POINT
        B7    CHECHA-12,,,
        CM    -IC,24,10,
        BNE   *+24,,,
        BTM   REPSUB,CHECHA,,      LEFT BRACKET - BRANCH TO REPSUB
        CM    -IC,45,10,
        BNE   BACHEC,,,            IF NOT E BRANCH TO BACK CHECK
        CF    SIGFLG,,,            OTHERWISE CLEAR SIG FLAG
        BTM   PBFMAN,2,10,         INSERT E
        TFM   -PBUFMK,45,10,
        TF    PFLG,ZERO-10,,       CLEAR PFLG,NFLG
        BNF   *+32,OFLG,,
        CF    IND,,,               ZERO EXPONENT IF ZERO MANTISSA
        B7    RESCAN+12,,,
        MF    IND,EAC,,            OTHERWISE STORE EXPONENT SIGN
        TF    BEFORE,XC,,          THEN SWITCH COUNTS - BEFORE IS EXP COUN
        TF    LENGTH,XL,,          LENGTH IS EXPONENT LENGTH
        TF    P,POINT,,            SWITCH POINTERS - P IS EXPONENT POINTER
        TF    BOTV,SVP,,           BOTV POINTS TO LOW ORDER EXPONENT DIGIT
        B7    RESCAN+12,,,         BRANCH TO RESCAN PICTURE
STR2    BTM   STREM,0,10,          FIND STRING LENGTH REMAINDER
        TF    LENGTH,-SVP,,        SET LENGTH TO TWICE STRING LENGTH
        A     LENGTH,-SVP,,
        AM    SVP,5,10,
        TF    SBUFMK,TOPV,,
        B7    TEST65,,,
LOOP65  SM    NODE1,4,10,          LAYOUT STRING AT TOP OF STACK
        TF    P,SBUFMK,,
        AM    SBUFMK,12,10,
        TF    -SBUFMK,-NODE1,,
        AM    P,1,10,
        CF    -P,,,
        AM    NODE1,4,10,
        TF    NODE1,-NODE1,,
```

```
         CF    NODE1-3,,,
TEST65   BNF   LOOP65,-NODE1,,,
         TF    P,SBUFMK,,,
         A     SBUFMK,TOPREM,,
         C     SBUFMK,BOUND,,          CHECK C(SBUFMK) AGAINST C(BOUND)
         BNH   *+32,,,
         PRA   CORFLO,,,               IF GREATER, CORE OVERFLOW
         B7    HALT,,,                 HALT
         SM    NODE1,4,10,
         TF    -SBUFMK,-NODE1,,
         AM    P,1,10,
         CF    -P,,,
         TF    SBUFMK,TOPV,,
         BTM   INCI,*+12,,             BRANCH TO MATCH OUTPUT PATTERN
         BNR   LOOP67,-IC,,            IF NON NULL PICTURE
         TF    TOT2,LENGTH,,           OTHERWISE SET TOT2
         B7    LOOP67+36,,,            AND BRANCH TO O/P CONSTANT STRING
LOOP67   CM    -IC,14,10,              CHOICE ON NEXT PIC CHARACTER
         BNE   BCHECK,,,
         BTM   KOUNT,14,10,            * - COUNT
         AM    SBUFMK,1,10,            SET SBUF FLAG
         SF    -SBUFMK,,,
         SM    SBUFMK,1,10,
         C     TOT2,LENGTH,,           CHK * COUNT AGAINST REMAINING STR LNGTH
         BH    *+68,,,
         A     SBUFMK,TOT2,,           IF LE LAYOUT STRING CHUNK IN PBUF
         BT    PBFMAN,TOT2,,
         TF    -PBUFMK,-SBUFMK,,
         S     LENGTH,TOT2,,           DEC LENGTH
         B7    LOOP67,,,               LOOP AGAIN
         CM    LENGTH,0,10,            OTHERWISE LAYOUT REMAINDER OF STRING
         BE    PADOUT,,,
         A     SBUFMK,LENGTH,,
         BT    PBFMAN,LENGTH,,
         TF    -PBUFMK,-SBUFMK,,
         S     TOT2,LENGTH,,
         B7    PADOUT,,,               THEN BRANCH TO FILL BLANK PADDING
BCHECK   CM    -IC,42,10,
         BNE   *+44,,,
         BTM   KOUNT,42,10,            B - COUNT BLANKS
         BT    PBFMAN,TOT2,,           INC BUF MARKER
         B7    LOOP67,,,               LOOP AGAIN
         BNR   PICERR,-IC,,            ERROR IF NOT RECORD MARK
         B7    BACHEC,,,               OTHERWISE BRANCH TO BACK CHECK
LOOP63   AM    IC,2,10,                PADDING LOOP - INC IC
         AM    TOT2,2,10,              INC TOT2
PADOUT   BNR   LOOP63,-IC,,            LOOP AGAIN IF NO RECORD MARK
         SF    SPFLG,,,                OTHERWISE
         BT    PBFMAN,TOT2,,           INC BUF MARKER
         B7    BACHEC,,,               AND BRANCH TO BACK CHECK
NULL2    AM    IC,2,10,                NULL STRING - INC IC
         TFM   TOT2,0,10,              SET TOT2 TO 0
         B7    PADOUT,,,               BRANCH TO FILL PADDING
BOOL2    BD    *+32,-SVP,,             CHECK BOOLEAN TYPE
         TFM   BOOBUF,FALBUF,,         SET BOOBUF TO POINT TO FALSE BUF
         B7    *+20,,,
         TFM   BOOBUF,TRUBUF,,         SET BOOBUF TO POINT TO TRUE BUF
```

```
         BTM   INCI,*+12,,          INC IC
         BTM   KOUNT,14,10,         COUNT ASTERISKS
         BNR   PICERR,-IC,,         ERROR IF NEXT CHARACTER IS NOT RM
         CM    TOT2,10,10,          OTHERWISE CHECK ASTERISK COUNT
         BL    BOOBIT,,,            IF LESS THAN 10, BRANCH TO CHOP BOOLEAN
         BTM   PBFMAN,10,10,        OTHERWISE INSERT VALUE IN PBUF
         TF    -PBUFMK,-BOOBUF,,
         SF    SPFLG,,,
         SM    TOT2,10,10
         BT    PBFMAN,TOT2,,        PAD OUT WITH BLANKS
         B7    BACHEC,,,            THEN BRANCH TO BACK CHECK
BOOBIT   BT    PBFMAN,TOT2,,        CUT BOOLEAN DOWN TO SIZE
         SM    TOT2,10,10,
         A     BOOBUF,TOT2,,
         TF    -PBUFMK,-BOOBUF,,    INSERT INTO PBUF
         B7    BACHEC,,,            THEN BRANCH TO BACK CHECK
*
**********************************************************************
** SUBROUTINE TO COUNT NUMBER OF ASTERISKS,BLANKS,OR NINES IN PIC**
**********************************************************************
*
         DS    2,,,                 SPACE FOR CHARACTER CODE
KOUNT    TFM   TOT,0,9,             SET TOTALS
         TFM   TOT2,0,9,
LOOP60   C     -IC,KOUNT-1,,        CHECK NEXT CHARACTER
         BNE   OUT60,,,             OUT IF NOT PARAMETER CHARACTER
         AM    TOT,1,10,            OTHERWISE INC TOTS
         AM    TOT2,2,10,
         TFM   INCI-1,LOOP60,,      INC IC AND LOOP AGAIN
         B7    INCI,,,
OUT60    CM    -IC,24,10,           CHECK FOR LEFT BRACKET
         BE    *+14,,,
         BB2   ,,,                  BRANCH BACK IF NOT
         TFM   REPCNT,NACC-9,,      OTHERWISE ASSEMBLE REP COUNT
LOOP61   AM    IC,1,10,
         TD    COMP,-IC,,           CHECK FOR DIGIT
         CF    COMP,,,
         CM    COMP,7,10,
         BE    ADDON,,,
         SM    -REPCNT,1,10,        IF NOT INC TOTALS
         A     TOT,-REPCNT,,
         A     TOT2,-REPCNT,,
         A     TOT2,-REPCNT,,
         AM    IC,1,10,             INC IC
         B7    OUT60-20,,,          THEN BRANCH TO CHECK NEXT CHARACTER
ADDON    AM    IC,1,10,             OTHERWISE ADD ON NEXT REP DIGIT
         AM    REPCNT,1,10,
         TD    -REPCNT,-IC,,
         CF    -REPCNT,,,
         B7    LOOP61,,,            AND THEN LOOP AGAIN
*
**********************************
**FLOATING SIGN CHECK SUBROUTINE**
**********************************
*
         DS    5,,,                 SPACE FOR BRANCH OUT ADDRESS
FLOATS   BNF   *+44,PFLG,,          IF PFLG
```

```
         CF    PFLG,,,                 CLEAR
         TFM   -PBUFMK,10,10,          AND INSERT PLUS SIGN
         B7    *+44,,,
         BNF   *+36,NFLG,,             IF NFLG
         CF    NFLG,,,                 CLEAR
         TFM   -PBUFMK,20,10,          AND INSERT MINUS SIGN
         BTM   PBFMAN,2,10,            INC BUF MARKER
         TFM   -PBUFMK,70,10,          INSERT ALPHAMERIC ZERO
         B7    -FLOATS+1,,,            BRANCH BACK
*
*********************************
**REPLICATE CONTROL SUBROUTINE**
*********************************
*
         DS    5,,,                    SPACE FOR BRANCH OUT ADDRESS
 REPSUB  BNF   REPSET,REPFLG,,         IF REP FLAG
         SM    -REPCNT,1,10,           DEC REP COUNT
         SM    IC,2,10,                DEC IC
         CM    -REPCNT,0,10,           CHECK REP COUNT AGAINST ZERO
         BNE   -REPSUB+1,,,            BRANCH BACK IF POSITIVE
         CF    REPFLG,,,               OTHERWISE FINISH REPLICATION
         A     IC,RCOUNT,,             INC IC
         AM    IC,5,10,
         B7    -REPSUB+1,,,            BRANCH BACK
 REPSET  SF    REPFLG,,,               OTHERWISE ENTER REP STATE - SET REPFLG
         TFM   REPCNT,NACC-9,,         ASSEMBLE REP COUNT
         TFM   RCOUNT,1,9,
 LOOP90  AM    IC,1,10
         BT    DIG7,IC,,
         BE    *+32,,,
         S     IC,RCOUNT,,
         B7    REPSUB+12,,,            BRANCH TO COUNT CHECK
         AM    IC,1,10,
         AM    REPCNT,1,10,
         TD    -REPCNT,-IC,,
         AM    RCOUNT,2,10,
         CF    -REPCNT,,,
         B7    LOOP90,,,
*
*************************************************
**SUBROUTINE TO INC IC FOR NEXT PIC CHARACTER CHECK**
*************************************************
*
         DS    5,,,                    SPACE FOR BRANCH OUT ADDRESS
 INCI    AM    IC,1,10,                INC IC
         SF    -IC,,,                  SET FLAG
         AM    IC,1,10,                INC IC
         B7    -INCI+1,,,              BRANCH BACK
*
*********************
**ROUND SUBROUTINE**
*********************
*
         DS    2,,,
 ADD5    AM    BOTV,1,10,
         CF    -P,,,
         SM    P,1,10,
```

```
         TDM   -P,0,11,
         AM    -BOTV,5,10,
         SM    BOTV,1,10,
         BB2   ,,,
*
************************************
**SUBROUTINE TO FIND MANTISSA LENGTH**
************************************
*
         DS    5,,,                  BRANCH ADDRESS FOR ZERO MANTISSA
 LSUB    TF    BOTV,SVP,,
         SM    BOTV,2,10,            BOTV POINTS TO MANTISSA
         CM    -BOTV,0,10,
         BE    -LSUB+1,,,            BRANCH OUT IF ZERO MANTISSA
         TF    P,BOTV,,,             OTHERWISE SET WORKING POINTER
         SM    P,6,10,
 LOOP40  BD    OUT40,-P,,            BRANCH OUT ON DIGIT
         AM    P,1,10,               OTHERWISE INC P
         B7    LOOP40,,,             AND LOOP AGAIN
 OUT40   TF    LENGTH,BOTV,,         CALCULATE MANTISSA LENGTH
         S     LENGTH,P,,
         AM    LENGTH,1,10,
         SF    LENGTH-2,,,
         MF    IND,-BOTV,,           STORE MANTISSA SIGN
         BB2   ,,,                   BRANCH BACK
*
*********************************************
**SUBROUTINE TO FIND STRING LENGTH REMAINDER**
*********************************************
*
         DS    2,,,
 STREM   TF    NODE1,-SVP,,          NODE1 POINTS TO FIRST STRING NODE
         SM    SVP,5,10,
         LD    99,-SVP,,             DIVIDE STRING LENGTH BY 6
         DM    96,6,10,
         TF    TOPREM,99,,           MULTIPLY REMAINDER BY 2
         A     TOPREM,99,,           AND STORE IN TOPREM
         CM    TOPREM,0,10,
         BNE   *+24,,,
         TFM   TOPREM,12,10,         SET TO 12 IF ZERO
         BB2   ,,,                   BRANCH BACK
*
*************************************************
**SUBROUTINE TO BUBBLE UP ADDRESSED TYPE AND VALUE**
*************************************************
*
         DS    5,,,                  SPACE FOR BRANCH OUT ADDRESS
 BUBBLE  TF    A,-SVP,,              SET A TO ADDRESS TYPE
         SM    A,9,10,
         TD    COMP,-A,,             CHECK TYPE
         CM    COMP,4,10,
         BNE   *+36,,,               IF  AW PREPARE FOR ARRAY OUTPUT
         BTM   ARSUB,*+12,,
         SM    A,9,10,
         TF    P,STP,,               OTHERWISE COPY ADDRESSED TYPE
 LOOP50  TD    -P,-A,,               AND VALUE TO ORIGINAL STACK POSITION
         C     P,SVP,,
```

```
         BE    -BUBBLE+1,,,          THEN BRANCH OUT
         AM    P,1,10,
         AM    A,1,10,
         B7    LOOP50,,,
*
*****************************************************
**SUBROUTINE TO CONTROL ARRAY OR MULTIPLE PIC SPEC**
*****************************************************
*
 BACHEC  BNF   FIN,ARRAY,,           IF  ARRAY FLAG
         AM    ACOUNT,1,10,          INC ARRAY COUNT
         C     ACOUNT,SIZE,,         CHECK AGAINST SIZE
         BNH   *+32,,,
         CF    ARRAY,,,              TERMINATE ARRAY OUTPUT IF HIGH
         B7    FIN,,,                THEN BRANCH TO MULT SPEC CHECK
         AM    A,1,10,               OTHERWISE INC A
         B7    LOOP50-12,,,          AND BRANCH TO BUBBLE NEXT ARRAY ELT
 FIN     C     SVP,TOPV,,            COMPARE SVP WITH TOPV
         BE    *+32,,,               IF NOT EQUAL
         BTM   POPTOP,1010,811,      INC STACK POINTERS
         B7    CHOOSE-12,,,          AND PRINT AGAIN
         S     SVP-1,COUNT,,         OTHERWISE POP VALUE(S)
         S     STP-1,COUNT,,                 POP TYPE(S)
         B7    BACK,,,               RETURN TO CONTROL
*
```

```
*
*****************************
**MUSSEL - DUMMY TRANSLATOR**
*****************************
*
         DORG  2450,,,
 ILBASE  DS    ,2411,,
 COUNT   DS    2,,,
         TFM   2406,28000,,          SET ARBASE
         TFM   2411,29000,,          SET ILBASE
         TFM   2416,30000,,          SET STKBAS
*
**SET SYMTAB**
*
         RNCD  25801,,,              SET SYMBOL TABLE
*
**READ I.L.*
*
         TFM   COUNT,0,10,
 READIL  RNCD  -ILBASE,,,
         AM    ILBASE,80,10,
         AM    COUNT,1,10,
         CM    COUNT,5,10,
         BNH   READIL,,,
         SM    ILBASE,480,9,
         H     ,,,
*
**CALL MUSINT**
*
         CALL  LINK,MUSINT,2450
         DEND  2452,,,
```

APPENDIX E

ERROR ANALYSIS OF FIRST YEAR STUDENT FORTRAN PROGRAMS

AT MASSEY UNIVERSITY (1971)

	Prob. 1, run 1	Prob. 2, run 1	Prob. 3, run 1	Prob. 3, run 2	Prob. 3, run 3	TOTAL	
Programs analysed	173	172	86	52	28	511	
Number that 'went'	88	51	4	5	2	150	
% that 'went'	51	30	5	10	7	29.4	
Error types							Percentage
(1) Loops	15	9	16	6	3	49	8.3
(2) Format	31	34	64	25	14	168	28.6
(3) IF statement	8	18	8	3	1	38	6.5
(4) Statement numbers	12	21	27	8	2	70	11.9
(5) I/O list	5	4	30	10	0	49	8.3
(6) Mixed mode	3	6	0	1	0	10	1.7
(7) C spec.	3	10	9	3	0	25	4.3
(8) Name length	3	8	2	0	0	13	2.2
(9) Assignment	3	2	0	0	0	5	0.9
(10) Undefined variable	2	12	1	2	1	18	3.1
(11) Undimensioned array	0	2	15	2	0	19	3.2
(12) Array name as simple variable name	1	1	14	1	0	17	2.9
(13) Invalid subscript	0	0	3	1	0	4	0.7
(14) Miscellaneous	20	47	18	9	9	103	17.5
TOTAL ERRORS	106	174	207	71	30	588	

THE PROBLEMS

1. Write a FORTRAN program to construct a table of squares of the integers 1,2,3,...,10 by making use of the fact that the differences form the sequence of odd integers 3,5,7,...

2. Write a FORTRAN program to read a set of data cards of the type described on page 11 of the duplicated notes and compute the following:

   (a) the number of men with incomes ≥ $4000,
   (b) the number of women with incomes < $2000,
   (c) the age of the youngest man,
   (d) the age of the oldest women.

   The required 4 values are to be printed out on one line in the above order.

3. Write a FORTRAN program to read a set of cards, each of which has punched in it the following data:

   (a) in columns 1 and 2: a decimal integer N ≤ 50;
   (b) in columns 3 to N+2: a string of N binary digits (bits: 0's or 1's);

   and then print out the following information:

   (a) for each card: the length of the longest run of consecutive 1's, the length of the longest run of consecutive 0's, and the string of bits;
   (b) the length of the longest run of consecutive 1's on any card, and the length of the longest run of consecutive 0's on any card.

NOTES ON THE ERROR CATEGORIES

Category (1) includes the following:

- DO statement
- incorrect number of iterations
- infinite loops
- DO loop ended with transfer statement
- un-numbered CONTINUE statement

Category (2) includes the following:

- missing FORMAT statement number, or comma after 'READ' or 'PRINT'
- syntax error in FORMAT statement
- invalid FORMAT specification
- FORMAT statement incorrectly referenced
- O/P not as instructed in the problem
- incorrect reading of data

Note: A large proportion of errors were due to a missing comma after 'READ' or 'PRINT', hence not specifying free format. Also, a large proportion of students in problem 3 used a specification such as 'FORMAT(NI1)'.

Category (3) includes the following IF statement errors:

- invalid expression (e.g. IF(J=100)1,2,3)
- statement numbers in wrong order
- statement numbers not separated by commas
- extra comma after last statement number
- incorrect number of statement numbers
- missing 'IF'
- other syntax errors

Category (4) includes the following errors concerning statement numbers:

- in wrong columns
- repeated
- undefined
- no statement number for next executable statement following transfer statement

Category (5) includes syntax errors in I/O lists. A large proportion were concerned with syntax errors in array lists such as (L(I),I=1,100), and a few wrote (L(I),I=1,N).

Category (7) (errors through misuse of comment spec.) includes errors caused by:

- comments without 'C' in column 1
- comments with 'C' in wrong column
- actual program statements with 'C' in column 1

Category (8) - variable name > 6 alphameric characters. This type of error actually arose for a variety of reasons.

Category (9) - errors from misunderstanding of assignment operator ('='). Includes errors of the form:

(i) I**2=1
(ii) KOUNT+1=KOUNT

Category (14) - miscellaneous. A large proportion of these were logical rather than syntactical errors.

Comments:

While the above method of classifying errors is somewhat arbitrary, it is evident from the analysis that transfer of control (both conditional and unconditional), iteration, and formatting, in particular, account for a large proportion (55.3%) of student errors in FORTRAN. The analysis in fact would support the following beliefs:

1. That the IF statement is unnatural for students.

2. That the unstructured nature of FORTRAN makes it awkward for students to picture the ideas of iteration and transfer of control i.e. to 'structure' their problems.

3. That the method of formatting I/O in FORTRAN is not clear to students. I/O is in fact an independent part of programming, and should not over-burden the student at an early stage.

APPENDIX F

2 SAMPLE MUSSEL PROGRAMS (WITH CORRESPONDING OBJECT CODE)

EXAMPLE 1

Source Code

```
          #NAME  DODSON,TED
          $CODE
          $XREF
          *
          * PROGRAM TO FIND HCF OF 2 NUMBERS USING
          * A RECURSIVE PROCEDURE
          *
    1     DO
    2          RESERVE N,M,A
    3          DEFINE HCF ON I,J AS
    4          DO IF  J.EQ.0
    5               THEN VALUE IS I
    6               ELSE VALUE IS HCF(J,I-I./.J*J)
    7          END
    8          READ N,M
    9          PRINT HCF(N,M)
   10     END
```

Note: $CODE is a control statement specifying that a listing of the Intermediate Language is to be given at the end of the translation phase. Similarly, $XREF is a control statement requesting a postinterpretation listing of the Symbol Table.

Intermediate Language

Digit	OP	Parameter
0	UJ	'105'
6	LN	'3'
11	PE	(2,0)
17	LN	'4'
22	TLR	J
26	TN0	
28	OP	'EQ'
32	IFJ	'55'
38	LN	'5'
43	TLR	I
47	VI	
49	UJ	'103'
55	LN	'6'
60	RAF	
62	TLA	J
66	TLR	I
70	TLR	I
74	TLR	J
78	OP	'./.'
82	TLR	J
86	OP	'*'
90	OP	'-'
94	CP	(HCF,2)
101	VI	
103	RE	
105	LN	'8'
110	TA	N
115	RD	
117	TA	M
122	RD	

(cont...)

Digit	OP	Parameter
124	LN	'9'
129	RAF	
131	TA	N
136	TA	M
141	CP	(HCF,2)
148	PR	
150	END	

EXAMPLE 2

Source Code

```
        #NAME  GORDON,N.M.
        $CODE
        $XREF
        *
        *  SELECTION SORT EXAMPLE
        *
 1      DO
 2           RESERVE N
 3           DEFINE SORTARRAY ON N AS
 4           DO
 5                RESERVE I,LIST(1:N),TOP,MAXPOS,T
 6                READ LIST
 7                DO REPEAT FOR TOP FROM N TO 2 BY -1
 8                     SET MAXPOS TO 1
 9                     DO REPEAT FOR I FROM 2 TO TOP
10                          IF LIST(I).GT.LIST(MAXPOS)
11                          THEN SET MAXPOS TO I
12                     END
13                     SET T TO LIST(TOP)
14                     SET LIST(TOP) TO LIST(MAXPOS)
15                     SET LIST(MAXPOS) TO T
16                END
17                PRINT LIST
18           END
19           READ N
20           EXECUTE SORTARRAY(N)
21      END
```

Intermediate Language

Digit	OP	Parameter
0	UJ	'289'
6	LN	'3'
11	PE	(1,5)
17	LN	'6'
22	TLA	LIST
26	RD	
28	LN	'7'
33	UJ	'45'
39	BA	'276'
45	TLA	TOP
49	TR	N
54	TN	'2'
65	TN	'-1'
76	RF	(39,7)
85	LN	'8'
90	TLA	MAXPOS
94	TN1	
96	ST	
98	LN	'9'
103	UJ	'115'
109	BA	'201'
115	TLA	I
119	TN	'2'
130	TLR	TOP
134	TN1	
136	RF	(109,9)
145	LN	'10'
150	TLA	LIST
154	TLR	I
158	INDR	
160	TLA	LIST

(cont...)

Digit	OP	Parameter
164	TLR	MAXPOS
168	INDR	
170	OP	'GT'
174	IFJ	'195'
180	LN	'11'
185	TLA	MAXPOS
189	TLR	I
193	ST	
195	UJ	'136'
201	LN	'13'
206	TLA	T
210	TLA	LIST
214	TLR	TOP
218	INDR	
220	ST	
222	LN	'14'
227	TLA	LIST
231	TLR	TOP
235	INDA	
237	TLA	LIST
241	TLR	MAXPOS
245	INDR	
247	ST	
249	LN	'15'
254	TLA	LIST
258	TLR	MAXPOS
262	INDA	
264	TLR	T
268	ST	
270	UJ	'76'
276	LN	'17'
281	TLA	LIST

(cont...)

Digit	OP	Parameter
285	PR	
287	RE	
289	LN	'19'
294	TA	N
299	RD	
301	LN	'20'
306	RAP	
308	TA	N
313	CP	(SORTARRAY,1)
322	END	

REFERENCES

1. Randell, B., and Russell, L.J. ALGOL 60 Implementation. Academic Press, New York, 1964.

2. Glass, R.L. An Elementary Discussion of Compiler/Interpreter Writing. Computing Surveys, Vol.1, No.1, March 1969.

3. Doran, R.W. Structural Programming. Computer Education, No.9, October 1971.

4. Doran, R.W., and Tate, G. An Approach to Structured Programming. Massey University Computer Unit, Publication No.6, 1972.

5. Doran, R.W. Teaching Structured Programming using an Unstructured Programming Language. Massey University Computer Unit, Publication No.5, May 1972.

6. Gordon, N.M., and Gibbons, P.B. The Design and Implementation of a Structured Teaching Language. Massey University Computer Unit, Publication No.8, May 1972.

7. Wirth, N. Programming and Programming Languages. Presented to 1st International Computing Symposium, Bonn, May 1970.

8. Wirth, N. Program Development by Stepwise Refinement. C.A.C.M. Vol.14, No.4, April 1971.

9. Henderson, P., and Snowdon, R. An Experiment in Structured Programming. Technical Report Series, No.18, University of Newcastle upon Tyne, 1971.

10. Higman, B. A Comparative Study of Programming Languages. Computer Monographs, No.2, MacDonald/Elsevier (4th ed., 1970).

11. Hopgood, F.R.A. Compiling Techniques. Computer Monographs, No.8, MacDonald/Elsevier (4th ed., 1970).

12. Sprowls, R. Clay. Computers - A Programming Problem Approach. Rev. ed., Harper & Row, New York, 1968.

13. Moulton, P.G., and Muller, M.E. DITRAN - A Compiler Emphasising Diagnostics. C.A.C.M. Vol.10, No.1, 1967.

14. Knuth, D.E. An Empirical Study of FORTRAN Programs. Software - Practice and Experience, Vol.1, 1971, pp. 105-133.

15. Knuth, D.E. The Art of Computer Programming. Fundamental Algorithms, Vol.I, Addison-Wesley, Massachusetts, 1968.

16. Radin, G., and Rogoway, H.P. NPL : Highlights of a New Programming Language. C.A.C.M. Vol.8, No.1, January 1965.

17. Naur, P. (Ed.) Revised Report on the Algorithmic Language ALGOL 60. C.A.C.M. Vol.6, p.1, January 1963.

18. Bottenbruch, H. Structure and Use of ALGOL 60. Journal of the A.C.M., Vol.9, pp. 161-221, April 1962.

19. Rosen, S. The ALGOL Programming Language. Chapter 2B of "Programming Systems and Languages", McGraw-Hill, 1967.

20. Sammet, J.E. Basic Elements of COBOL 61. C.A.C.M. Vol.5, No.5, May 1962.

21. Heising, W.P. FORTRAN vs. Basic FORTRAN. C.A.C.M. Vol.7, No.10, October 1964.

22. Sprowls, R. Clay. Introduction to PL/I Programming. Harper & Row, New York, 1969.

23. Conway, R.W., and Maxwell, W.L. CORC - The Cornell Computing Language. C.A.C.M. Vol.6, No.6, June 1963.

24. Smith, L.B. A Comparison of Batch Processing and Instant Turnaround. C.A.C.M. Vol.10, No.8, August 1967.

25. Snowdon, R.A. PEARL: An Interactive System for the Preparation and Validation of Structured Programs. Technical Report Series, No.28, University of Newcastle upon Tyne, November 1971.

26. Rosen, S.; Spurgeon, R.A.; and Donnelly, J.K. PUFFT - The Purdue University Fast FORTRAN Translator. C.A.C.M. Vol.8, No.11, November 1965.

27. Doran, R.W., and White, D. Simulator for a Stack Machine Plus Phrase Structure Compiler. Research Memorandum (Course Project), City University, London, May 1970.

28. Dewar, R.B.K.; Hochsprung, R.R.; and Worley, W.S. The IITRAN Programming Language. C.A.C.M. Vol.12, No.10, October 1969.

29. Madnick, S.E. String Processing Techniques. C.A.C.M. Vol.10, No.7, July 1967.

30. McKeeman, W.M.; Horning, J.J.; and Wortman, D.B. A Compiler Generator. Prentice-Hall Inc., N.J., 1970.

31. IBM. 1620 Monitor I System, IBM Systems Reference Library, Form C26-5739-4.

32. IBM. 1620 Central Processing Unit, Model 2. IBM Systems Reference Library, Form No. A26-5781-2.

33. Germain, C.B. Programming the IBM 1620. 2nd ed., Prentice-Hall Inc., N.J., 1965.

34. Wirth, N. The Programming Language Pascal. Acta Informatica 1, 35-63, 1971.

35. Dijkstra, E.W. Structured Programming. Pp. 84-88, Software Engineering Techniques, Nato Science Committee, 1969.