Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.
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 cooperation 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
August 1972

P.B. Gibbons
# TABLE OF CONTENTS

Chapter 0  INTRODUCTION - THE CASE FOR A NEW LANGUAGE . .  1

Chapter 1  A DESCRIPTION OF THE LANGUAGE . . . . . . . . . . . .  5

1.1. Admissible characters 5
1.2. Constants 5
1.3. Operators 6
1.4. Names 7
1.5. Variables 8
1.6. Arrays and subscripted variables 8
1.7. Assignment instructions 9
1.8. Program structuring 10
1.9. Conditional groups 10
1.10. Loops 13
1.11. Procedures 16
1.12. Recursion 17
1.13. Dynamic arrays 18
1.14.1. Input 19
1.14.2. Output 20
1.15. An example 26
1.15.1. The structure diagram 27
1.15.2. The program 28

Chapter 2  AN OVERVIEW OF THE IMPLEMENTATION . . . . . . .  29

- iii -
Chapter 3  INTERPRETING THE INTERMEDIATE LANGUAGE  . . .  32

3.1. The Interpreter Stack  32
3.2. The Symbol Table  34
3.3. The basic structure of the Interpreter  36
3.4. Expression evaluation  39
3.5. Assignment instructions  44
3.6. Main arrays & subscripted variables  46
3.7. Number representation & arithmetic  52
3.8. Logical operations  53
3.9. String storage & manipulation  54
3.10. The Line Number operator  63
3.11. General choice-groups, conditional groups, and if-groups  63
3.12. Repeat-group instructions  66
   3.12.1. Simple REPEAT without control  67
   3.12.2. REPEAT-while instruction  68
   3.12.3. REPEAT-until instruction  70
   3.12.4. REPEAT-times instruction  71
   3.12.5. REPEAT-for instr. grp. (with test)  75
   3.12.6. " " " (without test)  78
   3.12.7. REPEAT-set instruction group  80
3.13. CASE-group instruction  83
3.14. Functions and procedures  88
   3.14.1. Overview of their implementation  88
   3.14.2. The link data  90
   3.14.3. Fn. & proc. calls & declarations  91
   3.14.4. The use of local variables in expressions & assignments  99
   3.14.5. Dynamic arrays  100
3.15. Input/Output  105
   3.15.1. Input  105
   3.15.2. Output  110
      3.15.2.1. Unformatted output  111
      3.15.2.2. PIC controlled O/P  115
3.15.2.3. Printer controls 120
3.16. The END operator 121
3.17. The Translator-Interpreter interface 122

Chapter 4 POSSIBLE DEVELOPMENTS OF THE SYSTEM . . . . 124

APPENDICES AND REFERENCES . . . . . . . . . . . . . . . 128

APPENDIX A The Syntax of MUSSEL in BNF
APPENDIX B A Summary of the Intermediate Language
APPENDIX C The Interpreter Structure Diagrams
APPENDIX D The Interpreter SPS Program
APPENDIX E Error Analysis of First Year Student
FORTRAN Programs at Massey University
APPENDIX F 2 Sample MUSSEL Programs (with
    corresponding object code)

REFERENCES
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.

<table>
<thead>
<tr>
<th>THE PROBLEM</th>
<th>(Step 1)</th>
<th>THE ALGORITHM</th>
<th>(Step 2)</th>
<th>THE PROGRAM</th>
</tr>
</thead>
</table>

- 1 -
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 recursive.)

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 \(<=\geq\rangle\) 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>.OR.</td>
<td>1</td>
</tr>
<tr>
<td>.AND.</td>
<td>2</td>
</tr>
<tr>
<td>.NOT.</td>
<td>3</td>
</tr>
<tr>
<td>.LT. .LE. .EQ. .NE. .GE. .GT.</td>
<td>4</td>
</tr>
<tr>
<td>.CAT.</td>
<td>5</td>
</tr>
<tr>
<td>+ -</td>
<td>6</td>
</tr>
<tr>
<td>*/ ./ . NEG</td>
<td>7</td>
</tr>
<tr>
<td>**</td>
<td>8</td>
</tr>
</tbody>
</table>

(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 asymmetry 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

\[
\text{SET...loc...TO exp}
\]

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

examples: \( \text{SET I TO I+1} \)
\( \text{SET A,B,C TO 0} \)
\( \text{SET DATE TO DAY.CAT.MONTH.CAT.YEAR} \)
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 \text{ .GE. } M \\
THEN DO \\
\quad \text{SET MEAN TO SUM/M} \\
\quad \text{EXIT} \\
\quad \text{END} \\

MUSSEL's conditional group, of the form

DO IF condexp \\
\quad \text{THEN instruction} \\
\quad \text{ELSE instruction} \\
\quad \text{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 \text{ .EQ. } 0 \\
\quad \text{THEN DO} \\
\qquad \text{SET S TO N+K} \\
\qquad \text{SET P TO P+S} \\
\quad \text{END} \\
\quad \text{ELSE DO} \\
\qquad \text{SET S TO N-K} \\
\qquad \text{SET P TO P-S} \\
\quad \text{END} \\
\quad \text{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 CASEOpcode 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: \text{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.


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.

eample: 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 'I', '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:

<table>
<thead>
<tr>
<th>Internal String</th>
<th>PIC=</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADING</td>
<td>*(7)</td>
<td>HEADING</td>
</tr>
<tr>
<td>HEADING</td>
<td>*(10)</td>
<td>HEADINGbb</td>
</tr>
<tr>
<td>HEADING</td>
<td>*(4)</td>
<td>HEAD</td>
</tr>
</tbody>
</table>

(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:

<table>
<thead>
<tr>
<th>Boolean Value</th>
<th>PIC=</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>none</td>
<td>TRUEb</td>
</tr>
<tr>
<td>FALSE</td>
<td>none</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>*</td>
<td>T</td>
</tr>
<tr>
<td>FALSE</td>
<td>*</td>
<td>F</td>
</tr>
<tr>
<td>TRUE</td>
<td>****</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>****</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>*(10)</td>
<td>TRUEbbbbbb</td>
</tr>
<tr>
<td>FALSE</td>
<td>*(10)</td>
<td>FALSEbbbb</td>
</tr>
</tbody>
</table>

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

tables:

<table>
<thead>
<tr>
<th>Internal Representation</th>
<th>PIC=</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mantissa</td>
<td>Exponent</td>
<td>****</td>
</tr>
<tr>
<td>0001234</td>
<td>00</td>
<td>*******</td>
</tr>
<tr>
<td></td>
<td></td>
<td>******.*ES99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>**9999999</td>
</tr>
<tr>
<td>-0012345</td>
<td>-03</td>
<td>****.***S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>**<em>.</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>***<em>.<em>E-</em></em></td>
</tr>
</tbody>
</table>

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).
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 = I ,
            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
   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.
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.

\[ c(STP) \]
\[ c(SVP) \]

\[ \text{MUSINT STACK} \]

\[ c(STKBAS) + 10 \]
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null string</td>
<td>1</td>
</tr>
<tr>
<td>Number</td>
<td>2 (\bar{x} x \bar{x} x x \bar{x} x) mantissa exponent</td>
</tr>
<tr>
<td>Boolean value</td>
<td>3 (\bar{x} x) (01-TRUE 00-FALSE)</td>
</tr>
<tr>
<td>Address</td>
<td>6 (\bar{x} x \bar{x} x) core address</td>
</tr>
<tr>
<td>String descriptor</td>
<td>7 (\bar{x} x \bar{x} x \bar{x} x \bar{x} x x) length address</td>
</tr>
<tr>
<td>Array word (AW)</td>
<td>4 (\bar{x} \bar{x} x x \bar{x} x \bar{x} x x) dim. 1st array SMF elt addr. addr.</td>
</tr>
<tr>
<td>Return address word (RAW)</td>
<td>9 (\bar{x} x \bar{x} x x \bar{x} x x x) previous return I.L. RAW addr. address</td>
</tr>
<tr>
<td>Undefined Stack unit</td>
<td>0</td>
</tr>
<tr>
<td>Repeat-times control word</td>
<td>8 (\bar{x} x x x x x x x \bar{x} x) No. of loop iterations (integer)</td>
</tr>
</tbody>
</table>
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
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 'O' ('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:

<table>
<thead>
<tr>
<th>Type digit</th>
<th>Pointer to start of procedure code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x 5</td>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>

Pointer to proc. name  
Pointer to name of first local variable

3.3. The Basic Structure of the Interpreter

The MUSSEL interpreter, MUSINT, consists essentially of a main control routine together with a number of subroutines 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 rightmost 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.
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.
<table>
<thead>
<tr>
<th>Description</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface variables</td>
<td>00000-02401</td>
</tr>
<tr>
<td>TN, TA, TR, TN0, TN1, TB, TS, TNS, INDA, INDR, ST, STA, VI, UJ, BA, IFJ,</td>
<td>02402-02449</td>
</tr>
<tr>
<td>OP Routines</td>
<td>02450-06469</td>
</tr>
<tr>
<td></td>
<td>06470-06629</td>
</tr>
<tr>
<td>OP branching TABLE2</td>
<td>06630-17600</td>
</tr>
<tr>
<td>AND, OR, NOT, EQ, GE, LT, LE, CAT, ADD, SUB, MULT, DIV, INTDIV, NEG, RT,</td>
<td>17800-18000</td>
</tr>
<tr>
<td>RF, RFWT, RS, CS, CSWB, RAP, RAF, CP, PE, TLA, TLR, MSF, RE, NP, NL, SP,</td>
<td>18200-20000</td>
</tr>
<tr>
<td>TAB, I/O Load, &amp; END Routines</td>
<td>20005-20420</td>
</tr>
<tr>
<td>Main Control Routine</td>
<td>20500-25799</td>
</tr>
<tr>
<td>Declarations</td>
<td>25800-27899</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>27900-39999</td>
</tr>
<tr>
<td>SMF's, Main Arrays, I.L., and Interpreter Stack</td>
<td></td>
</tr>
<tr>
<td>String area</td>
<td></td>
</tr>
</tbody>
</table>
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:

\[ \text{TR (Take Result)} \quad 0 \ 3 \ \bar{x} \ x \ x \]

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:

\[ \text{TN (Take Number)} \quad \emptyset \ 4 \ \bar{x} \ x \ x \ x \ x \ x \ x \ x \ x \ x \ x \ x \]

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:
For Boolean constants the following operator is used:

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

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) \[\bar{0} 8 \bar{x} \bar{x} \bar{x} \ldots \bar{x} \bar{x} \bar{x} \bar{0} \bar{4}\]

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:
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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>142</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>147</td>
<td>OP</td>
<td>'NEG'</td>
</tr>
<tr>
<td>151</td>
<td>TN</td>
<td>'12'</td>
</tr>
<tr>
<td>162</td>
<td>OP</td>
<td>'*'</td>
</tr>
<tr>
<td>166</td>
<td>TN1</td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>OP</td>
<td>'+'</td>
</tr>
<tr>
<td>172</td>
<td>TN0</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>OP</td>
<td>'GE'</td>
</tr>
<tr>
<td>178</td>
<td>TR</td>
<td>STRING</td>
</tr>
<tr>
<td>183</td>
<td>TS</td>
<td>'ABC'</td>
</tr>
<tr>
<td>193</td>
<td>OP</td>
<td>'EQ'</td>
</tr>
<tr>
<td>197</td>
<td>OP</td>
<td>'OR'</td>
</tr>
<tr>
<td>201</td>
<td>..</td>
<td>...</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{c|c|c}
\hline
\text{c(STP)} & \text{c(SVP)} & \text{c(IC)}=175 \\
\hline
2 & 0 & \\
2 & -A*12+1 & \\
\hline
\end{array}
\]

STACK
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:

\[
\text{TA (Take Address)} \quad \overline{0} \ 2 \ \overline{x} \ x \ x
\]

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:

\[
\text{ST (Store)} \quad \overline{1} \ 2
\]

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).

\[
\text{STA (Store Also)} \quad \overline{1} \ 3
\]

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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>64</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>69</td>
<td>TA</td>
<td>B</td>
</tr>
<tr>
<td>74</td>
<td>TA</td>
<td>C</td>
</tr>
<tr>
<td>79</td>
<td>TNO</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>STA</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>STA</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>..</td>
<td></td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>c(STP)</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>=B=</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>=A=</td>
<td></td>
</tr>
</tbody>
</table>

STACK

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:
Array dimension
(0 for 1 dimension
1 for 2 dimensions)

\[
\begin{array}{ccccccccc}
\times & \times & \times & \times & 4 & \times & \times & \times & \times & \times \\
\uparrow & & & & & \uparrow & & & & \uparrow \\
\text{Pointer to AW} & \text{Rel. addr.} & \text{Rel. addr.} \\
\text{array name} & \text{type} & \text{of first} & \text{of SMF} & \text{array elt.}
\end{array}
\]

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:

\[
\begin{array}{ccccccccc}
\times & \times & \times & \times & \times & \times & \times & \times & \times \\
\uparrow & & & & & \uparrow \\
\text{lower} & \text{upper} \\
\text{subscript} & \text{subscript} \\
\text{bound} & \text{bound}
\end{array}
\]

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

\[
\begin{array}{cccccccccccccc}
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
\uparrow & & & & & & \uparrow & & & & & & \uparrow \\
\text{1st lower} & \text{1st upper} & \text{2nd lower} & \text{2nd upper} \\
\text{bound (L0)} & \text{bound (U0)} & \text{bound (L1)} & \text{bound (U1)} & \text{Cl=U1-Ll+1}
\end{array}
\]

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

```plaintext
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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>23</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>28</td>
<td>TR</td>
<td>I</td>
</tr>
<tr>
<td>33</td>
<td>TR</td>
<td>J</td>
</tr>
</tbody>
</table>

(cont...)
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 laid 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 laid 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\textsuperscript{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).

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.
<table>
<thead>
<tr>
<th>String length (characters)</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>85</td>
</tr>
<tr>
<td>2</td>
<td>227</td>
</tr>
<tr>
<td>3</td>
<td>157</td>
</tr>
<tr>
<td>4</td>
<td>330</td>
</tr>
<tr>
<td>5</td>
<td>312</td>
</tr>
<tr>
<td>6</td>
<td>362</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

**TOTAL** 1547

**Analysis of Character Strings in SNOBOL**
<table>
<thead>
<tr>
<th>String length (characters)</th>
<th>Number of strings</th>
<th>Core wastage (in digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>85</td>
<td>$85 \times (4+2) = 510$</td>
</tr>
<tr>
<td>2</td>
<td>227</td>
<td>$227 \times 4 = 908$</td>
</tr>
<tr>
<td>3</td>
<td>157</td>
<td>$157 \times (8+2) = 1570$</td>
</tr>
<tr>
<td>4</td>
<td>330</td>
<td>$330 \times 8 = 2640$</td>
</tr>
<tr>
<td>5</td>
<td>312</td>
<td>$312 \times (12+2) = 4368$</td>
</tr>
<tr>
<td>6</td>
<td>362</td>
<td>$362 \times 12 = 4344$</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>$40 \times (16+2) = 720$</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>$13 \times 16 = 208$</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>$1 \times (20+2) = 22$</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>$1 \times (32+2) = 34$</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Node value field length (characters)</th>
<th>Storage wastage (digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25880</td>
</tr>
<tr>
<td>2</td>
<td>15324</td>
</tr>
<tr>
<td>3</td>
<td>13500</td>
</tr>
<tr>
<td>4</td>
<td>14180</td>
</tr>
<tr>
<td>5</td>
<td>14304</td>
</tr>
<tr>
<td>6</td>
<td>12404</td>
</tr>
<tr>
<td>7</td>
<td>12968</td>
</tr>
<tr>
<td>8</td>
<td>17660</td>
</tr>
</tbody>
</table>
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:

![String address diagram]

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:

![AVAIL list diagram]

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 $c(\text{BOUND})$ must be decreased to make a node available. Each time this is done, $c(\text{BOUND})$ must be checked against the top of the Stack, addressed by $c(\text{SVP})$, for the occurrence of overflow. Initially, $c(\text{BOUND})$ is set to 39,999, and $c(\text{AVAIL})$ to 00000.

Operations on Strings

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

```
7   x x x x  x x x x x
  +     +     +
Descriptor  String  Address of
  type digit  length  first node
               (chars.)
```

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 alphanumeric 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:

\[ \text{LN (Line Number)} \quad 0 \ 1 \ x \ x \ x \]

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.

\[
\begin{align*}
\text{UJ} & \quad \overline{16xxx} \\
\text{IFJ} & \quad \overline{18xxx}
\end{align*}
\]

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 implementation 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 instruc-
tion 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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>20</td>
<td>LN</td>
<td>'5'</td>
</tr>
<tr>
<td>25</td>
<td>TR</td>
<td>B</td>
</tr>
<tr>
<td>30</td>
<td>IFJ</td>
<td>'56'</td>
</tr>
<tr>
<td>36</td>
<td>LN</td>
<td>'6'</td>
</tr>
<tr>
<td>41</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>46</td>
<td>TNL</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>UJ</td>
<td>'70'</td>
</tr>
<tr>
<td>56</td>
<td>LN</td>
<td>'7'</td>
</tr>
<tr>
<td>61</td>
<td>TA</td>
<td>A</td>
</tr>
</tbody>
</table>

(cont...)
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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>44</td>
<td>LN</td>
<td>'13'</td>
</tr>
<tr>
<td>49</td>
<td>UJ</td>
<td>'61'</td>
</tr>
<tr>
<td>55</td>
<td>UJ</td>
<td>'126'</td>
</tr>
<tr>
<td>61</td>
<td>LN</td>
<td>'14'</td>
</tr>
<tr>
<td>66</td>
<td>TA</td>
<td>I</td>
</tr>
<tr>
<td>71</td>
<td>TR</td>
<td>I</td>
</tr>
<tr>
<td>76</td>
<td>TNL</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>OP</td>
<td>'+'</td>
</tr>
<tr>
<td>82</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>LN</td>
<td>'15'</td>
</tr>
<tr>
<td>89</td>
<td>TR</td>
<td>I</td>
</tr>
<tr>
<td>94</td>
<td>TR</td>
<td>N</td>
</tr>
<tr>
<td>99</td>
<td>OP</td>
<td>'GT'</td>
</tr>
<tr>
<td>103</td>
<td>IFJ</td>
<td>'120'</td>
</tr>
<tr>
<td>109</td>
<td>LN</td>
<td>'16'</td>
</tr>
<tr>
<td>114</td>
<td>UJ</td>
<td>'55'</td>
</tr>
<tr>
<td>120</td>
<td>UJ</td>
<td>'61'</td>
</tr>
<tr>
<td>126</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>UJ</td>
<td>'22'</td>
</tr>
<tr>
<td>16</td>
<td>UJ</td>
<td>'98'</td>
</tr>
<tr>
<td>22</td>
<td>LN</td>
<td>'4'</td>
</tr>
<tr>
<td>27</td>
<td>TR</td>
<td>X</td>
</tr>
<tr>
<td>32</td>
<td>TR</td>
<td>N</td>
</tr>
<tr>
<td>37</td>
<td>OP</td>
<td>'GT'</td>
</tr>
<tr>
<td>41</td>
<td>IFJ</td>
<td>'16'</td>
</tr>
<tr>
<td>47</td>
<td>LN</td>
<td>'5'</td>
</tr>
<tr>
<td>52</td>
<td>TA</td>
<td>X</td>
</tr>
<tr>
<td>57</td>
<td>TR</td>
<td>X</td>
</tr>
<tr>
<td>62</td>
<td>TNL</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>OP</td>
<td>'-'</td>
</tr>
<tr>
<td>68</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>LN</td>
<td>'6'</td>
</tr>
<tr>
<td>75</td>
<td>TR</td>
<td>B</td>
</tr>
<tr>
<td>80</td>
<td>IFJ</td>
<td>'92'</td>
</tr>
<tr>
<td>86</td>
<td>UJ</td>
<td>'16'</td>
</tr>
<tr>
<td>92</td>
<td>UJ</td>
<td>'22'</td>
</tr>
<tr>
<td>98</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>21</td>
<td>UJ</td>
<td>'33'</td>
</tr>
<tr>
<td>27</td>
<td>UJ</td>
<td>'113'</td>
</tr>
<tr>
<td>33</td>
<td>LN</td>
<td>'9'</td>
</tr>
<tr>
<td>38</td>
<td>TR</td>
<td>X</td>
</tr>
<tr>
<td>43</td>
<td>TR</td>
<td>N</td>
</tr>
<tr>
<td>48</td>
<td>OP</td>
<td>'GT'</td>
</tr>
<tr>
<td>52</td>
<td>OP</td>
<td>'NOT'</td>
</tr>
<tr>
<td>56</td>
<td>IFJ</td>
<td>'27'</td>
</tr>
<tr>
<td>62</td>
<td>LN</td>
<td>'10'</td>
</tr>
</tbody>
</table>

(cont...)
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:

\[
\text{RT} \quad 2 \ 0 \ \bar{x} \ x \ x \ x \ \bar{x} \ x \ x
\]

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 '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:

\[
\text{BA } \text{... } \text{... x x x x x}
\]

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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>UJ</td>
<td>'19'</td>
</tr>
<tr>
<td>13</td>
<td>BA</td>
<td>'67'</td>
</tr>
<tr>
<td>19</td>
<td>LN</td>
<td>'10'</td>
</tr>
<tr>
<td>24</td>
<td>TR</td>
<td>N</td>
</tr>
<tr>
<td>29</td>
<td>RT</td>
<td>(13,0)</td>
</tr>
<tr>
<td>38</td>
<td>LN</td>
<td>'11'</td>
</tr>
<tr>
<td>43</td>
<td>TA</td>
<td>X</td>
</tr>
<tr>
<td>48</td>
<td>TR</td>
<td>X</td>
</tr>
<tr>
<td>53</td>
<td>TNL</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>OP</td>
<td>'+'</td>
</tr>
</tbody>
</table>

(cont...)
<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>UJ</td>
<td>'29'</td>
</tr>
<tr>
<td>67</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

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

```
c(STP) 2 \times \times \times \times \times \times \times \times \times x
         |   |   STACK   | c(IC)=30
         |   |           |
         |   |           |
         |   |           |

c(SVP) |   |           |   |
```

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

```
c(STP) \overline{3} \times \times \times \times \times \times \times \times \times x
c(SVP)          |   |           | c(IC)=14 or 30
         |   |   STACK   |   |
         |   |           |   |
         |   |           |   |
         |   |           |   |
```
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)  \(2 \ 1 \ \bar{x} \ \bar{x} \ \bar{x} \ \bar{x} \ \bar{x} \ \bar{x} \ \bar{x} \ \bar{x} \)

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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>29</td>
<td>UJ</td>
<td>'41'</td>
</tr>
<tr>
<td>35</td>
<td>BA</td>
<td>'117'</td>
</tr>
<tr>
<td>41</td>
<td>LN</td>
<td>'5'</td>
</tr>
<tr>
<td>46</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>51</td>
<td>TNL</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>TR</td>
<td>N</td>
</tr>
<tr>
<td>58</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>69</td>
<td>RF</td>
<td>(35,5)</td>
</tr>
<tr>
<td>78</td>
<td>LN</td>
<td>'6'</td>
</tr>
<tr>
<td>83</td>
<td>TA</td>
<td>B</td>
</tr>
<tr>
<td>88</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>93</td>
<td>INDA</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>100</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>105</td>
<td>OP</td>
<td>'*'</td>
</tr>
<tr>
<td>109</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>UJ</td>
<td>'69'</td>
</tr>
<tr>
<td>117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
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)  2 2 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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>UJ</td>
<td>'114'</td>
</tr>
<tr>
<td>108</td>
<td>BA</td>
<td>'203'</td>
</tr>
<tr>
<td>114</td>
<td>LN</td>
<td>'8'</td>
</tr>
<tr>
<td>119</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>124</td>
<td>TN1</td>
<td>'2'</td>
</tr>
<tr>
<td>126</td>
<td>TN</td>
<td>'8'</td>
</tr>
<tr>
<td>137</td>
<td>RFWT</td>
<td>'9'</td>
</tr>
<tr>
<td>142</td>
<td>LN</td>
<td>B</td>
</tr>
<tr>
<td>147</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>152</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>157</td>
<td>INDA</td>
<td>A</td>
</tr>
<tr>
<td>159</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>164</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>169</td>
<td>OP</td>
<td>'*'</td>
</tr>
<tr>
<td>173</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>LN</td>
<td>'10'</td>
</tr>
<tr>
<td>180</td>
<td>TR</td>
<td>C</td>
</tr>
<tr>
<td>185</td>
<td>IFJ</td>
<td>'197'</td>
</tr>
<tr>
<td>191</td>
<td>UJ</td>
<td>'108'</td>
</tr>
<tr>
<td>197</td>
<td>UJ</td>
<td>'137'</td>
</tr>
<tr>
<td>203</td>
<td>...</td>
<td>.....</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)  2 3 \overline{x} \times \overline{x} \times \times \overline{x} \times \times

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 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...)
<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>100</td>
<td>TN</td>
<td>'3'</td>
</tr>
<tr>
<td>111</td>
<td>RS</td>
<td>(3, 71, 19)</td>
</tr>
<tr>
<td>122</td>
<td>LN</td>
<td>'20'</td>
</tr>
<tr>
<td>127</td>
<td>TA</td>
<td>B</td>
</tr>
<tr>
<td>132</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>137</td>
<td>INDA</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>144</td>
<td>TR</td>
<td>A</td>
</tr>
<tr>
<td>149</td>
<td>OP</td>
<td>'∗'</td>
</tr>
<tr>
<td>153</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>UJ</td>
<td>'111'</td>
</tr>
<tr>
<td>161</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

```
<table>
<thead>
<tr>
<th>2</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
```

STACK

And just before execution of the BA operator:

```
<table>
<thead>
<tr>
<th>2</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
```

STACK
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:

\[
\text{CS (CASE)} \quad \overline{2} \ 4 \ \bar{x} \ x
\]

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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>LN</td>
<td>'4'</td>
</tr>
<tr>
<td>28</td>
<td>TR</td>
<td>I</td>
</tr>
<tr>
<td>33</td>
<td>TN</td>
<td>'3'</td>
</tr>
<tr>
<td>44</td>
<td>TN</td>
<td>'5'</td>
</tr>
<tr>
<td>55</td>
<td>UJ</td>
<td>'148'</td>
</tr>
<tr>
<td>61</td>
<td>LN</td>
<td>'5'</td>
</tr>
<tr>
<td>66</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>71</td>
<td>TN</td>
<td>'3'</td>
</tr>
<tr>
<td>82</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>UJ</td>
<td>'170'</td>
</tr>
<tr>
<td>90</td>
<td>LN</td>
<td>'6'</td>
</tr>
<tr>
<td>95</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>100</td>
<td>TN</td>
<td>'4'</td>
</tr>
<tr>
<td>111</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>UJ</td>
<td>'170'</td>
</tr>
<tr>
<td>119</td>
<td>LN</td>
<td>'7'</td>
</tr>
<tr>
<td>124</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>129</td>
<td>TN</td>
<td>'5'</td>
</tr>
<tr>
<td>140</td>
<td>ST</td>
<td></td>
</tr>
</tbody>
</table>

(cont...)
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:

\[
\text{CSWB (CASE without bounds) } \bar{2} \ 5 \bar{x} \ x
\]

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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>35</td>
<td>LN</td>
<td>'4'</td>
</tr>
<tr>
<td>40</td>
<td>TR</td>
<td>I</td>
</tr>
<tr>
<td>45</td>
<td>UJ</td>
<td>'136'</td>
</tr>
<tr>
<td>51</td>
<td>LN</td>
<td>'5'</td>
</tr>
<tr>
<td>56</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>61</td>
<td>TN</td>
<td>'1'</td>
</tr>
<tr>
<td>72</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>UJ</td>
<td>'160'</td>
</tr>
<tr>
<td>80</td>
<td>LN</td>
<td>'6'</td>
</tr>
<tr>
<td>85</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>90</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>101</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>UJ</td>
<td>'160'</td>
</tr>
<tr>
<td>109</td>
<td>LN</td>
<td>'7'</td>
</tr>
<tr>
<td>114</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>119</td>
<td>TN</td>
<td>'3'</td>
</tr>
<tr>
<td>130</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>UJ</td>
<td>'160'</td>
</tr>
<tr>
<td>138</td>
<td>CSWB</td>
<td>'3'</td>
</tr>
</tbody>
</table>

(cont...)

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

<table>
<thead>
<tr>
<th>2</th>
<th>ℬ</th>
<th>ℬ</th>
<th>ℬ</th>
<th>ℬ</th>
<th>ℬ</th>
<th>ℬ</th>
<th>ℬ</th>
<th>ℬ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 across 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:

```
9 x x x x x x x x
^   ^   ^
RAW Previous Return I.L.
type RAW address
digit address
```

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 top-most RAW at any time is indicated by an RAW pointer, \( c(\text{RAW}) \), 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) 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) 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').

\[
\text{CP (Call Procedure)} \quad 2 \quad 8 \quad x \quad x \quad x \quad x
\]

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:

\textbf{Example:}

\textbf{Source code}

\begin{verbatim}
 12 SET M TO MEAN(U,V*2)
\end{verbatim}
Intermediate Language

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>422</td>
<td>LN</td>
<td>'12'</td>
</tr>
<tr>
<td>427</td>
<td>TA</td>
<td>M</td>
</tr>
<tr>
<td>432</td>
<td>RAF</td>
<td></td>
</tr>
<tr>
<td>434</td>
<td>TA</td>
<td>U</td>
</tr>
<tr>
<td>439</td>
<td>TR</td>
<td>V</td>
</tr>
<tr>
<td>444</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>455</td>
<td>OP</td>
<td>'*'</td>
</tr>
<tr>
<td>459</td>
<td>CP</td>
<td>(MEAN,2)</td>
</tr>
<tr>
<td>466</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The Stack situation:

(i) Immediately before execution of the RAF operator:

\[
\begin{array}{c|c|c|c}
\text{c(STP)} & 6 & \overline{x} & \overline{x} & \overline{x} & \overline{x} \\
\text{c(SVP)} & \text{=M=} & & & & \\
\end{array}
\]

(ii) Immediately before execution of the CP operator:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
\text{Actual parameters} & 2 & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} \\
\text{c(STP)} & 6 & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} & \overline{x} \\
\text{c(SVP)} & \text{=U=} & & & & & & & & & & \\
\text{c(RAW)} & \text{=M=} & V*2 & =U= & RAW & function val. & =M= \\
\end{array}
\]
(iii) Immediately after execution of the CP operator:

<table>
<thead>
<tr>
<th>c(STP)</th>
<th>c(SVP)</th>
<th>c(RAW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x x x x x x x x x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x x x x x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>x x x x x x</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>Ret^addr</td>
<td>RAW</td>
</tr>
<tr>
<td>6</td>
<td>x x x x x</td>
<td>fn. value</td>
</tr>
</tbody>
</table>

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

\[ \text{LN} \quad \text{L} \quad \text{PE} \quad (\text{FP,LI}) \quad \ldots \quad \ldots \quad \text{RE} \]

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:

\[ \text{PE (Procedure Entry)} \quad 3 \quad 0 \quad x \quad x \quad x \quad x \]

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:

\[ \text{PE (Procedure Entry)} \quad 3 \quad 0 \quad x \quad x \quad x \quad x \]
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:

\[
\text{VI (Value Is) } \overline{1} \overline{5}
\]

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.
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

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

The Stack situation:

(i) Immediately before execution of the PE operator:

\[
\begin{array}{|c|c|c|}
\hline
\text{c(STP)} & \text{c(SVP)} & \text{c(RAW)} \\
\hline
\text{Actual parameters} & 2 \boxed{x} \boxed{x} \boxed{x} \boxed{x} \boxed{x} \boxed{x} \boxed{x} & 6 \boxed{x} \boxed{x} \boxed{x} \boxed{x} \\
\text{c(IC)=170} & 9 \boxed{x} \boxed{x} \boxed{x} \boxed{x} \boxed{x} \boxed{x} \boxed{x} & 0 \text{Ret}^+\text{addr} \\
\text{(Previous RAW address)} & 6 \boxed{x} \boxed{x} \boxed{x} \boxed{x} & \text{STACK} \\
\hline
\end{array}
\]

\(1\text{OP to be explained later.}\)
(ii) Immediately after the execution of the PE operator:

<table>
<thead>
<tr>
<th>c(STP)</th>
<th>c(SVP)</th>
<th>c(WVP)</th>
<th>c(RAW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c(IC)=174)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 \bar{x} x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 \bar{x} x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 \bar{x} x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Space for 1st possible SMF TEMP V*2 =U= c(BVP) 

(Previous RAW address) 

Stack 

(iii) Immediately after execution of the VI operator:

<table>
<thead>
<tr>
<th>c(STP)</th>
<th>c(SVP)</th>
<th>c(WVP)</th>
<th>c(RAW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c(IC)=223)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 \bar{x} x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 \bar{x} x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 \bar{x} x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 \bar{x} x x x x x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Space for 1st possible SMF TEMP(=(A+B)/2) V*2 =U= c(BVP) 

Stack
(iv) Immediately after execution of the RE operator:

\[
\begin{array}{c}
c(\text{STP}) \\
\hline
(\text{c(IC)}=\text{ret addr} = 465) \\
\hline
c(\text{BTP}) \\
\hline
\hline
2 \quad x \quad x \quad x \quad x \quad x \quad x \quad x \\
\hline
6 \quad x \quad x \quad x \quad x
\end{array}
\]

\[
\text{c(SVP)} \quad \text{c(RAW)} \quad \text{c(BVP)}
\]

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 runtime. 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) \[\overline{3} \ \overline{1} \ \overline{x} \ \overline{x} \]

TLR (Take Local Result) \[\overline{3} \ \overline{2} \ \overline{x} \ \overline{x} \]

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:

\[ \text{MSF (Make Storage Function)} \quad \overline{3} \quad \overline{3} \quad \overline{x} \quad \overline{x} \quad \overline{x} \]

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 Cl (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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>TN1</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>TR</td>
<td>N</td>
</tr>
<tr>
<td>14</td>
<td>TN0</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TR</td>
<td>M</td>
</tr>
<tr>
<td>21</td>
<td>MSF</td>
<td>(5,3,1)</td>
</tr>
<tr>
<td>27</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

\[(c(\text{IC})=22)\]
(ii) Immediately after execution of the MSF operator:

```
(c(IC)=26)

Space for
SMF
0

Elements
of array
A
0
0
0

Elements
of array
B
0
0
0

Elements
of array
C
0
0

SMF
x x x x x x
x x x x x

Local
Variables
Array
Words
{(C) 4 l x x x x x x x x x}
{(B) 4 l x x x x x x x x x}
{(A) 4 l x x x x x x x x x}

RAW
9 x x x x x x x x x

STACK
```
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 already in 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:

\[ \text{RD (Read)} \quad 3 \quad 6 \]

Example:

**Source code**

19 READ X,Y,Z

**Intermediate Language**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>29</td>
<td>LN</td>
<td>'19'</td>
</tr>
<tr>
<td>34</td>
<td>TA</td>
<td>X</td>
</tr>
<tr>
<td>39</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>TA</td>
<td>Y</td>
</tr>
<tr>
<td>46</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>TA</td>
<td>Z</td>
</tr>
<tr>
<td>53</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

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 (BUF MAN) 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:

\[ \text{PR (Print)} \quad 3 \, 7 \]

Example:

Source code

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

Intermediate Language

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>LN</td>
<td>'20'</td>
</tr>
<tr>
<td>69</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>74</td>
<td>PR</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>TR</td>
<td>B</td>
</tr>
<tr>
<td>81</td>
<td>TR</td>
<td>C</td>
</tr>
<tr>
<td>86</td>
<td>OP</td>
<td>'*'</td>
</tr>
<tr>
<td>90</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>101</td>
<td>OP</td>
<td>'/'</td>
</tr>
<tr>
<td>105</td>
<td>PR</td>
<td>D</td>
</tr>
<tr>
<td>107</td>
<td>TA</td>
<td></td>
</tr>
</tbody>
</table>

(continue...)
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Implied PICTURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>-******9BBBBBBBB</td>
</tr>
<tr>
<td>F-form</td>
<td>-*******.........</td>
</tr>
<tr>
<td>E-form</td>
<td>BBB-*.*******ES99</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>944</td>
<td>LN</td>
<td>'15'</td>
</tr>
<tr>
<td>949</td>
<td>TA</td>
<td>A</td>
</tr>
<tr>
<td>954</td>
<td>TR</td>
<td>B</td>
</tr>
<tr>
<td>959</td>
<td>TR</td>
<td>C</td>
</tr>
<tr>
<td>964</td>
<td>OP</td>
<td>'*'</td>
</tr>
<tr>
<td>968</td>
<td>TA</td>
<td>D</td>
</tr>
<tr>
<td>973</td>
<td>TR</td>
<td>I</td>
</tr>
<tr>
<td>978</td>
<td>TR</td>
<td>J</td>
</tr>
<tr>
<td>983</td>
<td>INDR</td>
<td></td>
</tr>
<tr>
<td>985</td>
<td>PRC</td>
<td>(3,***)</td>
</tr>
<tr>
<td>1001</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 alphanemic record mark as a second parameter). The Interpreter recognizes 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 rescans 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:

\[
NP \text{ (Newpage)} \quad \overline{3\ 9}
\]

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:

\[
NL \text{ (Newline)} \quad \overline{4\ 0\ x\ x}
\]

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

\[ \text{SP (Space)} \quad \bar{4} \ 1 \bar{x} \quad x \]

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.

\[ \text{TAB} \quad \bar{4} \ 2 \bar{x} \quad x \quad x \]

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:

\[ \text{END} \quad \bar{3} \ 5 \]

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.
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:

<table>
<thead>
<tr>
<th>Core Address</th>
<th>Symbolic Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>2402 - 2406</td>
<td>ARBASE</td>
<td>Addresses type digit of 1st array element</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(if any)</td>
</tr>
<tr>
<td>2407 - 2411</td>
<td>ILBASE</td>
<td>Addresses 1st digit of I.L.</td>
</tr>
<tr>
<td>2412 - 2416</td>
<td>STKBAS</td>
<td>Addresses digit immediately following end of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the I.L.</td>
</tr>
<tr>
<td>2417</td>
<td></td>
<td>Switch for postinterpretation S.T. listing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>option ('1' - on, '0' - off)</td>
</tr>
<tr>
<td>2418 - 2431</td>
<td></td>
<td>DCF for name area</td>
</tr>
</tbody>
</table>
The set up in core at this stage is as follows:

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>SMF's</th>
<th>Array elt. space</th>
<th>I.L.</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>25800</td>
<td>c(ARBASE)</td>
<td>c(ILBASE)</td>
<td>c(STKBAS)</td>
<td></td>
</tr>
</tbody>
</table>

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.
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

<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> ::= (]|}I|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 spec>
          <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.>

<read> ::= READ <listpart>

<print> ::= PRINT <print list>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$string exp$ ::= $string term|$string exp$.CAT.$string term$
<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> | <integer>
   | -<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> | <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> ::= !l!

<special character> ::= : | | * | + | - | / | | $ | @ | + | | |

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

<eol> ::= end of line - not represented externally
## APPENDIX B

### A SUMMARY OF THE INTERMEDIATE LANGUAGE

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

(Continued...)
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Opcode</th>
<th>Parameter(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>CALL PROCEDURE</td>
<td>28</td>
<td>*<strong>,</strong></td>
</tr>
<tr>
<td>CLP</td>
<td>CALL LIBRARY PROCEDURE</td>
<td>29</td>
<td><strong>,</strong></td>
</tr>
<tr>
<td>PE</td>
<td>PROCEDURE ENTRY</td>
<td>30</td>
<td><strong>,</strong></td>
</tr>
<tr>
<td>TLA</td>
<td>TAKE LOCAL ADDRESS</td>
<td>31</td>
<td>**</td>
</tr>
<tr>
<td>TLR</td>
<td>TAKE LOCAL RESULT</td>
<td>32</td>
<td>**</td>
</tr>
<tr>
<td>MSF</td>
<td>MAKE STORAGE FUNCTION</td>
<td>33</td>
<td><strong>,</strong>,*</td>
</tr>
<tr>
<td>RE</td>
<td>RETURN</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>END</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>RD</td>
<td>READ</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>PR</td>
<td>PRINT</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>PRC</td>
<td>PRINT CONTROL</td>
<td>38</td>
<td><strong>,</strong>,**...,**0#</td>
</tr>
<tr>
<td>NP</td>
<td>NEWPAGE</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>NL</td>
<td>NEWLINE</td>
<td>40</td>
<td>**</td>
</tr>
<tr>
<td>SP</td>
<td>SPACE</td>
<td>41</td>
<td>**</td>
</tr>
<tr>
<td>TAB</td>
<td>TAB</td>
<td>42</td>
<td>***</td>
</tr>
</tbody>
</table>

### SECONDARY OPCODES

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

**LINE NUMBER ROUTINE**

- Increment $c(IC)$ to point to line number parameter: $c(IC) + 3$
- Place 'LN' code & parameter in PBUF (for trace)
- Update line number: $c(LN) + c(c(IC))$
- Return to Control

**Opcode parameter**

```
LN xxx +
c(IC)
```
OP 02 TAKE ADDRESS

Routine

- Increment \( c(\text{IC}) \) 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

\[
\begin{align*}
\text{c(\text{IC})} + 3 \\
\text{c(SVP)} + 10 \\
\text{c(STP)} + 10 \\
\text{c(SVP)} \leftarrow \text{c(SYMBAS)} + (14 \times \text{c(\text{IC})})
\end{align*}
\]
OP 03

TAKE RESULT Routine

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)

opcode parameter

TR XXXX

c(IC)

\[ c(IC) + 3 \]

\[ c(SVP) + 10 \]

\[ c(STP) + 10 \]

\[ \text{POPTOP subroutine} \]

\[ c(A) \leftarrow c(SYMBAS) + (14 \times c(IC)) \]
PUSH Routine to place result (& type) addressed by c(A) at top of Stack

Decrement c(A) to address type digit
- c(A) - 9

Transmit type digit to 2-digit field (for comparison)
- c(COMP) ← c(c(A))

If c(COMP) = 0
- ERROR (TR E1)
- Undefined variable

Place type digit at top of Stack
- c(c(STP)) ← c(COMP)

Increment c(A) to address value part
- c(A) + 9

Place value at top of Stack

Return to Control
Place value at top of Stack

**Choice on c(COMP)**

1. (null string)
   - No operation
2. (number)
   - c(c(SVP)) ← c(c(A)) (transmit floating)
3. (boolean)
   - c(c(SVP)) ← c(c(A))
4. (string descriptor)
   - Copy addressed string in string area and place descriptor (for copy) at top of Stack
5. Other
   - ERROR (TR E2) Illegal type digit
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

Copy addressed string in string area and place corresponding descriptor at top of Stack.
Copy string
Nodes

Repeat

Set pointers to address 'info' parts of nodes
\[ c(c(NODE2)) \rightarrow c(c(REF)) \]
\[ c(REF) - 4 \quad c(NODE2) - 4 \]

Copy 'info'
\[ c(c(NODE2)) \rightarrow c(c(REF)) \]
\[ c(REF) + 4 \quad c(NODE2) + 4 \]

Reset pointers to address 'link' fields of nodes

Test for terminal node

Link to next available node

Chain to next reference string node
\[ c(REF) \rightarrow c(c(REF)) \]
Clear flag at REF-3

If flag at c(REF)

Set c(NODE2) to point to next available node (GRAB subroutine)
\[ c(NODE1) \rightarrow c(NODE2) \]
\[ c(REF) \rightarrow c(c(REF)) \]
Set flag at NODE2-3
Clear flag at NODE2-3

Set flag at c(NODE2)

Exit
TAKE NUMBER Routine

opcode parameter

\[ \text{TN} \ x \ x \ x \ x \ x \ x \ x \ x \ x \]

\[ \text{c(IC)} \ \text{mantissa} \ \text{exp} \]

- Increment \( c(IC) \) to point to mantissa
- Place 'TN' code and mantissa in PBUF (for trace)
- Increment \( c(IC) \) to point to exponent
- Increment \( c(IC) \) 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)
\[ c(c(STP)) \leftarrow 2 \]
OP 05

TAKE NUMBER ZERO Routine

Place 'TNO' code in PBUF (for trace)

Increment Stack pointers

c(SVP)+10
c(STP)+10
(POPTOP subroutine)

Place number '0' at top of Stack

c(c(SVP)) \rightarrow c(FZERO)
(transmit floating)

Place 'number'type digit at top of Stack

c(STP) \rightarrow 2

Return to Control
OP 06

Place 'TNl' code in PBUF (for trace)

Increment Stack pointers

c(SVP) + 10

c(STP) + 10

(POPTOP subroutine)

Place number '1' at top of Stack

c(c(SVP)) ← c(FONE)

(transmit floating)

Place 'number' type digit at top of Stack

c(c(STP)) ← 2

Return to Control

TAKE NUMBER ONE Routine

T

TNl

c(IC)
OP 07

**TAKE BOOLEAN Routine**

- **Opcode parameter**
  - \( \text{T B X X} \)
  - \( c(\text{IC}) \)

- **Increment \( c(\text{IC}) \)**
  - to point to Boolean value parameter
  - \( c(\text{IC}) + 2 \)

- **Place 'TB' code and parameter in PBUF** (for trace)
  - \( c(\text{SVP}) + 10 \)
  - \( c(\text{STP}) + 10 \)
  - (POPTOP subroutine)
  - \( c(c(\text{SVP})) \leftarrow c(c(\text{IC})) \)

- **Increment Stack pointers**
  - \( c(\text{IC}) \)

- **Place Boolean value at top of Stack**
  - \( c(c(\text{STP})) \leftarrow 3 \)

- **Place 'Boolean' type digit at top of Stack**

- **Return to Control**
**TAKE STRING Routine**

<table>
<thead>
<tr>
<th>opcode</th>
<th>parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSxxxx</td>
<td>string (in alphameric code, terminated by alphameric record mark)</td>
</tr>
</tbody>
</table>

- 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
Initialize length counter to 0

OUTA:

Repeat

Peel off next String unit

Pack unit away in string area

Test for terminal node

Pack string in string area & fill in length field of descriptor at top of Stack

Repeat

if (c(COUNT) = 0)
    exit

if (c(COUNT) = 5)
   _set flag at c(NODE1)

if (c(COUNT) = 5)
    Return node addressed by c(NODE2) to AVAIL list (RETURN subroutine)

Exit from OUTA
Test for terminal node

Choice
Flag at IND?

yes

Increment c(IC)
Set terminal node link
Move flag at IND to c(NODE2)
c(SVP)-5
Set length field in descriptor at top of Stack

no

Exit c(NODE1) ← c(NODE2)
Set c(NODE2) to point to next available node (GRAB subroutine)
Link

Link to next available node

Clear flag at NODE2-3
Set flag at NODE2-3
c(SVP)+5
c(SVP) ← c(LENGTH)
c(SVP)-5
Place 'TNS' code in PBUF (for trace)

Increment Stack pointers

Place 'null string' type digit at top of Stack

Return to Control

\[ \text{c(SVP)} + 10 \]

\[ \text{c(STP)} + 10 \] (POPTOP subroutine)

\[ \text{c(c(STP))} \leftarrow 1 \]
Place 'IN' code in PB Buf (for trace)

Initialize dimension counter

Check that indices are integers

Check that c(A) addresses Array Word

Set relative address base

Remainder of Routine

Repeat

Check that indices are integers

Inc. c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Choice

Local or main array?

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

Exit

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?

Yes

No

ERROR (IND E2)

Invalid subscript type

ERROR (IND E1)

Non-integral subscript

Increment c(A) to address array dimension

Transmit to 2-digit field (for comparison)

Local or main array?

Set relative address base

Remainder of Routine

Set flag at COMP

Clear flag at COMP

Choice

Flag at COMP?
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

If c(W) ← c(c(A)) +c(BASE)

Set c(A) to address storage mapping function

If c(A) ← c(c(A)) +c(BASE)

Set c(SVP) to address first subscript value

If c(SVP) +8

Check subscript values against array bounds

If c(c(SVP)) > c(c(A))

ERROR (IND E5)
Subscript greater than upper bound

If c(c(SVP)) ≤ c(c(A))

REMINDER OF ROUTINE

c(L0) ← c(c(SVP)) -c(c(A))

If c(L0) < 0

ERROR (IND E5)
Subscript less than lower bnd.

c(A) +4

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

Set c(A) to address array elt

Test OP

c(SVP)-8

Return to Control

c(A)+4

c(SVP)+10

Check array bounds

c(A)+4

c(SVP)-18

Set c(A) to address array elt

Test OP

Place 'DA' code in PBUF (for trace)

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

2

Choice on c(c(IC))

10(INDA)

11(INDR)

If c(L)<0

If c(c(SVP))>c(c(A))

If c(c(SVP))<c(A)

If c(c(SVP)) = c(A)

If c(c(SVP)) = c(A)

If c(c(SVP)) = c(A)

If c(c(SVP)) = c(A)
OPS 12,13

STORE
STORE ALSO
Routines

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

If
c(c(STP))=6

ERROR
(ST E1)

\[ \text{c(TOPV)} + \text{c(SVP)} \]
\[ \text{c(TOPT)} + \text{c(STP)} \]
\[ \text{c(SVP)} - 10 \]
\[ \text{c(STP)} - 10 \]
\[ \text{POPTOP subroutine} \]
\[ \text{c(A)} + \text{c(c(SVP))} \]
Check OP & return to Control

Choice on \(c(c(IC))\)

Decrement Stack pointers
- \(c(SVP) - 10\)
- \(c(STP) - 10\)
  (POPTOP subroutine)

Place 'A' code in PBUF (for trace)
- \(c(c(STP)) \leftarrow c(COMP)\)

Shift down type
- \(c(c(STP)) \leftarrow c(COMP)\)

Shift down value

Choice on \(c(COMP)\)

'2'
- \(c(c(SVP)) \leftarrow c(c(TOPV))\)
  (transmit floating)

'1'
- No operation

Other
- \(c(c(SVP)) + c(c(TOPV))\)

Return to Control
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

- Transmit to 2-digit field (for comparison)
  - c(c(A)) + c(c(STOR-1))

Check stored value type

Choice on c(COMP)

- 0 or 1
  - Return string units to AVAIL list
  - c(c(A)) + c(c(TOPT))

- 2
  - No operation
  - c(c(A)) + c(c(STOR-1)) (transmit fltg)

- 3
  - c(c(A)) + c(c(STOR-1))

- 7
  - ERROR (ST E2)
  - Invalid type digit
  - c(c(A)) + c(c(STOR-1))

- Other
  - c(A) - 5
  - c(STOR-1) - 5
  - c(c(A)) + c(c(STOR-1))
  - c(A) + 1
Set \( c(\text{NODE2}) \) to address first node

\[ c(\text{NODE2}) + c(c(\text{A})) \]

Set flag at \( \text{AVAIL-3} \)

Repeat

Last node?

If Flag at \( c(\text{NODE2}) \)

Exit

Chain down to next node

\[ c(\text{NODE1}) + c(\text{NODE2}) \]

Exit

Return string units to \( \text{AVAIL list} \)

Return last node to \( \text{AVAIL list} \)

\[ c(c(\text{NODE2})) + c(\text{AVAIL}) \]

\[ c(\text{AVAIL}) + c(\text{NODE2}) \]

Return node to \( \text{AVAIL list} \)

\[ c(c(\text{NODE1})) + c(\text{AVAIL}) \]

\[ c(\text{AVAIL}) + c(\text{NODE1}) \]

Clear flag at \( \text{NODE2-3} \)
OP 15

VALUE IS Routine

- Place 'VI' code in PBUF (for trace)
- Set c(A) to address function value location (immediately below Return Address Word)
- 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

opcode
V I + c(IC)
UNCONDITIONAL JUMP Routine

Increment c(IC) to point to I.L. address parameter

c(IC) + 4

Place 'UJ' code and parameter in PBUF (for trace)

Reset instruction counter to new I.L. address (RESET subroutine)

Reset instruction counter to new I.L. address (RESET subroutine)

c(IC) + c(c(IC)) + c(ILBASE) - 1

Return to Control

c(IC) - 1

c(IC) + c(ILBASE)

c(IC) + c(L0)

c(L0) + c(c(IC))
Increment $c(\text{IC})$ to point to I.L. address parameter

Place 'BA' code and parameter in PBUF (for trace)

Reset $c(\text{IC})$ to new I.L. address (RESET subroutine)

Unstack loop control variables

If $c(c(\text{STP})) + 8$

Repeat

Unstack POPTOP subroutine

If $c(c(\text{STP})) = 6$

Exit

Return to Control

If $c(c(\text{STP})) = 6$

Exit

If $c(c(\text{STP})) = 6$

Exit

If $c(c(\text{STP})) = 6$

Exit
OP 18

Check for Boolean type unit at top of Stack

If \( c(c(\text{STP})) \neq 3 \)

Error (IFJ El)
Conditional expression type error

Increment \( c(\text{IC}) \) to point to I.L. address parameter

Place 'IFJ' code and parameter in PBUF (for trace)

Jump if top Boolean value FALSE

Pop \( c(\text{SVP})-10 \), \( c(\text{STP})-10 \)
(POPTOP subroutine)

If \( c(c(\text{SVP})) = 0 \)

Reset instruction counter to new I.L. address (RESET sub.)

opcode parameter

IF \( J \times \times \times \times \times \)

\[ c(\text{IC}) \]
OP 19

**OP Routine to control branches to secondary operator routines**

- Increment \( c(IC) \) to point to secondary op parameter
  - \( c(IC) + 2 \)

- Place 'OP' code and parameter in PBUF (for trace)
  - \( c(TABLE2) + -.TAB2 \)

- Branch to appropriate op routine
  - Branch indirectly to op routine
  - Branch to \( c(c(TABLE2)) \)

**Opcode parameter**

- \( O P \times x \)
- \( \uparrow c(IC) \)
Secondary ops 1 & 2

Check that top 2 Stack units are of type 'Boolean' and set pointers (CHEC subroutine)

OR op?

If $c(c(IC)) = 1$

$D(DIG) + 1$

Check top Stack value

If $c(c(TOPV)) = c(DIG)$

$c(c(SVP)) \leftarrow c(DIG)$

Reset $c(DIG)$

$c(DIG) + 0$

Return to Control

Opcode parameter $OP \xrightarrow{x} c(IC)$
Secondary op 3

NOT Routine

Check that top Stack unit is of type 'Boolean'

If \( c(c(\text{STP})) = 3 \)
ERROR (NOT EI)
Non-Boolean
NOT operand

Complement top Boolean value

Choice on \( c(c(\text{SVP})) \)

0 (FALSE)
\( c(c(\text{SVP})) + 1 \)

1 (TRUE)
\( c(c(\text{SVP})) + 0 \)

Return to Control

opcode parameter

\( \text{OP} \quad \bar{x} \quad x \quad + \quad c(\text{IC}) \)
Secondary ops 4 – 9 (relationals)

- Store relational operator
  - Check operand type ('number' or 'string')
    - Choice on top Stack type - c(c(STP))=2?
      - No: String relational routine
        - Scale and compare mantissas (c(EDIF) high)
      - Yes: Check that operands (i.e. top 2 Stack units) are of type 'number' and set pointers (CHEC subroutine)
        - Exponents equal - compare mantissas
        - Scale and compare mantissas (c(EDIF) low)

- Set 'Boolean' result type
  - Calculate exponent difference
    - Comparison
      - Choice: Compare c(EDIF) with 0
        - >: Exponents equal - compare mantissas
        - <: Scale and compare mantissas (c(EDIF) low)

- Compare operands
  - Return

- opcode parameter
  - O P X X
    - c(IC)
Set large positive exponent difference

Scale and compare mantissas (c(EDIF) high)

Test T2

Choice on op

Choice

Scale and compare mantissas (c(EDIF) low)

Set large negative exponent difference

EQ, GE, LE

c(c(SVP)) + 01 (TRUE result)

NE, GT, LT

c(c(SVP)) + 00 (FALSE result)
Check for EQ or NE op (EQNE subroutine)

SCALE & COMPARE MANTISSAS (c(EDIF) LOW)

Large difference in exponents?

Yes

Test T3

Choice

On op

GT,GE

LT,LE

No

Scale

c(LOD) + MAC - c(EDIF)
c(MAC) + c(c(TOPV))
Move flag from MAC to c(LOD)

Compare

Choice

Compare c(c(BOTV)) with c(c(LOD))

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

c(c(SVP)) + 00 (FALSE result)

NE

c(c(SVP)) + 01 (TRUE result)

Return

Large difference in exponents

Choice

c(EDIF) ≥ 7?

Yes

Test T1

Choice on op

GT, GE

c(c(SVP)) + 01 (TRUE result)

LT, LE

c(c(SVP)) + 00 (FALSE result)

No

Choice on op

Scale

c(LOD) + MAC + c(EDIF)

c(MAC) + c(c(BOTV))

Move flag from MAC to c(LOD)

Compare

Choice

c(c(LOD)) with c(c(TOPV))

Test T1

Test T2

Test T3
STRING RELATIONAL ROUTINE

Set pointers
\[ \text{c(TOPV)} + \text{c(SVP)} \]
\[ \text{c(SVP)} - 10 \]
\[ \text{c(STP)} - 10 \]

Check for occurrence of null string operand

Set pointers:
\[ \text{c(NODE1)} \text{ to bottom string} \]
\[ \text{c(NODE2)} \text{ to top string} \]
\[ \text{c(BOTV)} \text{ to bot str length} \]
\[ \text{c(TOPV)} \text{ to top str length} \]

Choice on top Stack type

If \[ \text{c(c(BOTV))} \neq \text{c(c(TOPV))} \]
Check for EQ or NE op (EQNE subroutine)

String lengths not equal?

Save next Stack type

Set 'Boolean' result type on next Stack (i.e. c(COMP))

Test T2
Test T1

Decision:
- Test T3
- No operation
- Other

Choice on next Stack type (i.e. c(COMP))

Test T3

Other

Invalid operand

String length not equal?

Invalid operand
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 string & compare end bits

Exit

Compare node values

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
Set pointers to end bits

Yes
Compare bits (Test T12)

c(NODE1)-4
c(NODE2)-16+c(BOTREM)

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
Set pointers to end bits

Yes
Test T12

Test T13

c(NODE2)-4
c(NODE1)-16+c(TOPREM)

Test T13

Choice
Compare c(BOTREM) with c(TOPREM)

<
c(NODE2)-4
c(NODE1)-4
c(BOTREM)

>
c(NODE2)

Test T12

> c(TOPREM)

<
c(BOTREM)

+ c(TOPREM)

> c(TOPREM)

+ c(BOTREM)

Yes
Secondary op 10

CAT ROUTINE

Place 'CAT' code in PBUF (for trace)

Transmit top Stack type to 2-digit field (for comparison)

Set pointers

Check top Stack type

Set pointers

Check next Stack type

Set pointers

POPTOP subroutine

Return to Control

No operation

ERROR (CAT E1) Invalid CAT operand type

Concatenate Strings

Shift down top string descr. & type

Return to Control

Choice on c(STP)

Choice on c(BOTV)

Choice on c(TOPV)

opc ode parameter

O P x x

+ c(IC)
CONCATENATE STRINGS

- Chain down to last node of bottom string
  - If flag at c(NODE1)
    - Exit
  - Repeat
    - Link terminal node of bottom string to first node of top string
      - c(NODE2) + c(c(TOPSTR))
        - Set flag at NODE2-3
      - c(NODE1) + c(c(NODE1))
        - Clear flag at NODE2-3
      - c(NODE1) + c(c(BOTSTR))

- Divide bottom string length by 6
  - Load dividend (c(c(BOTV)))
    - c(99) + c(c(BOTV)) (load dividend)
    - Divide c(96) by 6

- Add string lengths
  - c(c(BOTV)) + c(c(TOPV))

- Check division remainder against 0
  - Choice
    - Remainder(c(99)) = 0?
      - Yes
        - Return to Control
      - No
        - Repack top string
REPACK TOP STRING

Set \( c(\text{BOTREM}) \) to bottom string length remainder (in digits)

Set \( c(\text{TOPREM}) \) to top string length remainder (in digits)

\[ c(\text{BOTREM}) + c(99) \times 2 \]

Divide \( c(\text{TOPV}) \) by 6

\[ c(\text{TOPREM}) + '\text{remainder}' \times 2 \]

End of top string?

If Flag at \( c(\text{NODE2}) \)

Exit

Repeat

Repack \( \text{NODE1} \) (REPACK subroutine)

Reset \( c(\text{NODE2}) \) to address node's link field

Chain down

\[ \text{c(NODE2)} + \text{c(BOTREM)} + 3 \]

\[ \text{c(NODE1)} + \text{c(NODE2)} \]

\[ \text{c(NODE2)} + \text{c(c(NODE2))} \]

Clear flag at \( \text{NODE2-3} \)

Tidy up end nodes

\[ \text{c(NODE2)} \]

\[ \text{c(NODE1)} + \text{c(NODE2)} \]

\[ \text{c(NODE2)} + \text{c(c(NODE2))} \]

\[ \text{c(NODE1)} + \text{c(NODE2)} \]

\[ \text{c(NODE2)} + \text{c(c(NODE2))} \]

\[ \text{c(NODE1)} + \text{c(NODE2)} \]

\[ \text{c(NODE2)} + \text{c(c(NODE2))} \]
TIDY UP
END NODES

\[ c(\text{DIF}) + c(\text{BOTREM}) + c(\text{TOPREM}) - 12 \]

**Choice**

\[ c(\text{DIF}) \leq 0? \]

- **Yes**
  - Remove minus sign of \( c(\text{DIF}) \)
  - Clear flag at \( c(\text{DIF}) \)
  - Terminate concatenated string at NODE1

- **No**
  - Terminate concatenated string at NODE2

- **Return to Control**

**Remove minus sign of \( c(\text{DIF}) \)**

**Clear flag at \( c(\text{DIF}) \)**

**Set terminal link field**

**Return NODE2 to AVAIL list (RETURN subroutine)**

**Pack NODE1**

**Set terminal link field**

**Set flag at \( c(\text{NODE2}) \)**

**Repack NODE1 (REPACK subroutine)**

**Set flag at \( c(\text{NODE1}) \)**

\[ c(\text{TEMP}) + c(\text{c(NODE1)}) \]
\[ c(\text{c(NODE1)}) + c(\text{c(NODE2)}) \]
\[ c(\text{NODE1}) - c(\text{TOPREM}) \]
\[ c(\text{c(NODE1)}) + c(\text{TEMP}) \]
\[ c(\text{NODE1}) + 1 \]

**Clear flag at \( c(\text{NODE1}) \)**
REPACK subroutine to repack NODE1

Save NODE1 info

- \( c(NODE1) - 4 \)
- \( c(TEMP) + c(c(NODE1)) \)

Fill portion of info from NODE2

- \( c(NODE2) - 4 - c(REPACK - 1) \)
- \( c(c(NODE1)) + c(c(NODE2)) \)

Replace NODE1 info

- \( c(NODE1) + c(BOTREM) - 12 \)
- \( c(c(NODE1)) + c(TEMP) \)

Tidy up

- \( c(NODE1) + 1 \)
- \( c(NODE2) + 1 \)
- Move flag from \( c(NODE1) \) to \( c(NODE2) \)
Decrement string area boundary to make node available

AVAIL list empty?

Choice
Flag at AVAIL?

Yes

Decrement string area boundary to make node available

No

Take node from AVAIL list

c(NODE2) + c(BOUND)
c(BOUND)-16
Check for o'flow
If c(BOUND)<c(SVP)
ERROR (STR E1)
Core overflow

Grab

c(NODE2) + c(AVAIL)

Reset AVAIL pointer

c(AVAIL) + c(c(AVAIL))
Clear flag at AVAIL-3

GRAB subroutine to set c(NODE2) to point to next available string node

AVAIL list empty?

Choice
Flag at AVAIL?
RETURN subroutine to return NODE2 to the AVAIL list

- Link node to AVAIL list
  - Set flag at AVAIL-3
  - \( c(c(\text{NODE2})) + c(\text{AVAIL}) \)

- Reset AVAIL pointer
  - \( c(\text{AVAIL}) + c(\text{NODE2}) \)
Secondary op 11

ADD ROUTINE

Check that operands (i.e. top 2 Stack units) are of type 'number' and set pointers (CHEC subroutine)

If $c(c(TOPV)) = 0$ or $c(c(SVP)) = 0$

Floating add routine

Integer operands?

Add integers

If 0'flow

Set $c(TOPV)$ to address top mantissa

Turn off overflow

Add to bottom mantissa

If Choice Flag at IND?

Return to Control

Return to Increment Set Variable subroutine

Return

Shift result to accumulator

Set pointers to high and low order digits

Set exponent accumulator

Round, normalize & store result

opcode parameter

$I O p x x$

c(IC)
FLOATING ADD ROUTINE

Set accumulators

Choice
\[ c(c(SVP)) > c(c(TOPV))? \]

Calculate exponent difference

\[ c(EDIF) = c(EP) - c(EQ) \]

Check exponent magnitude

Choice
\[ c(EDIF) < 8? \]

No

Large difference in exponents implies one operand relatively insignificant - store other

\[ c(c(BOTV)) + c(MAC) \]
\[ c(c(SVP)) + c(EP) \]

Yes

Scale and add numbers

Round, normalize, and store result

\[ c(c(TOPV)) + c(MQ) \]
SCALE AND ADD NUMBERS

Scale

Turn off overflow

Add, and store sign of result

Choice

O'flow?

Yes

Result zero?

Choice
c(c(LOD))=0?

No

Yes

Scale up

Set pointers to high & low order digits of result in accumulator

Set pointers to high & low order digits of result in accumulator

Store zero

Return

Clear flag at MAC-6
c(MAC-7) + -1

c(EAC)+c(EDIF)+1

Clear flag at EAC-1
c(EAC-2) + -0
c(LOD) + MAC+c(EDIF)

Move flag from MAC to c(LOD)

Move flag from c(LOD) to IND

Add, and store sign of result

c(c(LOD))+c(MQ)

c(c(MAC-7))+c(MAC-1)

c(HOD) + MAC-7

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

Repeat

Set flag at c(HOD)
c(LOD)+1
Turn off o'flow
c(c(LOD))+5

Choice

O'flow?

Yes

Scale up

c(c(LOD))

No

Exit

c(c(HOD))

c(LOD)+1
c(EAC)+1

Check for exponent o'flow or u'flow

Store result & tidy up

Choice

Digit at c(LOD)?

Yes

Clear flag at c(HOD)
c(HOD)-1
c(c(HOD)) ← 0
c(LOD)-1
c(EAC)+1

No

Exit

c(c(HOD))

c(LOD)+1

Check for exponent o'flow or u'flow

Store result & tidy up

Choice

Flag at EAC-2?

Yes

Print "exponent underflow"

Replace with zero result

Tidy up & return

Print "exponent overflow"

Replace with 9's

No

c(c(SVP)) ← c(FZERO)
(transmit floating)

c(c(LOD)) ← c(ALL9S)
c(EAC) ← 99
Store result & tidy up

Store mantissa

\[ c(c(\text{BOTV})) + c(c(\text{LOD})) \]

Append sign

Move flag from IND to \( c(SVP) \)

Set flag at EAC-1

Store exponent

\[ c(c(\text{SVP})) + c(\text{EAC}) \]

Tidy up and return

\[ c(MAC+15) + c(\text{ZERO}-3) \]

Clear flag at MAC+1

Return
Secondary op 12

SUBTRACT Routine

Negate top
Stack value
(NEG subroutine)

Add operands
(ADD Routine)

opcode parameter

\[ O P X X \]
\[ \uparrow \]
\[ c(IC) \]
Secondary op 13

MULTIPLY ROUTINE

Check that operands (i.e. top 2 Stack units) are of type 'number' and set pointers (CHEC subroutine)

Add exponents
- c(EAC) + c(c(TOPV))
- Clear flag at EAC-1
- c(EAC-2) + -0
- c(EAC) + c(c(SVP))

Multiply mantissas

Result zero?
- If c(99)=0
  - Store zero
  - Return to Control

Integer result?
- If c(EAC)=0
  - Load accs and set pointers

Real result
  - c(MAC+7) + c(99)
  - c(EAC) + 7
  - c(LOD) + MAC
  - c(HOD) + MAC - 6

Multiply
- c(c(BOTV)) x c(c(TOPV))
- (result in product area 80-99)

Store sign of result

Store integer

Set flag at 93

Store mantissa

Store exponent
- c(c(BOTV)) + c(99)
- c(c(SVP)) + 00

opcode parameter
- 0 P x x
  - c(IC)
Secondary op 14

**DIVIDE ROUTINE**

- **Check that operands (i.e. top 2 Stack units) are of type 'number' and set pointers (CHEC subroutine)**
  - \( c(EAC) + c(c(SVP)) \)
  - Clear flag at EAC-1
  - \( c(EAC-2) + 0 \)
  - \( c(EAC) - c(c(TOPV)) \)
  - \( c(TOPV) - 2 \)

- **Subtract exponents**

- **Dividend zero?**
  - If \( c(c(TOPV)) = 0 \)
    - **ERROR (DIV E1)** zero divisor
  - If \( c(c(BOTV)) = 0 \)
    - **Store zero result**
    - **Return to Control**

- **Divisor zero?**
  - **If**
    - **Load accs. and set pointers**
    - \( c(MAC+8) + c(92) \)
    - \( c(HOD) + MAC-6 \)
    - \( c(LOD) + MAC \)
    - **Round, normalize & store result**

- **Divide mantissas**
  - **Load dividend (bottom mantissa) to product area**
    - \( c(99-8) + c(c(BOTV)) \) (load dividend)
  - **Divide by top mantissa**
    - \( c(100-15) \) by \( c(c(TOPV)) \)
  - **Store sign of result**
    - **Move flag from 92 to IND**
Secondary op 15

INTEGER DIVIDE
ROUTINE

Check that operands (i.e. top 2 Stack units) are of type 'number' and set pointers (CHEC subroutine)

Check that operands are integers

If c(c(TOPV))≠0 or c(c(SVP))≠0

ERROR (IDIV E1) Non-integer operand

Dividend zero?

If c(c(BOTV))=0

Store zero result

Divisor zero?

If c(c(TOPV))=0

ERROR (DIV E1) Zero divisor

Divide c(c(BOTV)) by c(c(TOPV)) & store integer part of result

Divide by c(c(TOPV))

Load dividend, c(c(BOTV))

Divide c(100-7) by c(c(TOPV))

Store integer part

c(99) + c(c(BOTV))

c(SVP)+2

Opcode parameter

0 P × x

c(IC)
Secondary op 16

NEG Routine

Check that top Stack unit contains number

If \[
c(c(STP)) \neq 2
\]

ERROR (NEG El) Non-numeric NEG operand

Negate top Stack value (NEG subroutine)

Return to Control

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

Clear flag at \(c(TOPV)\)

No

Set flag at \(c(TOPV)\)

opcode parameter

\[
O \hat{P} \hat{X} x
\]

\(c(IC)\)
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)

If
\[ c(\text{COMP}) + c(\text{CHEC}) = 1 \]

ERROR (CHEC E1)
Operand type error

\[ c(\text{TOPV}) + c(\text{SVP}) \]
\[ c(\text{SVP}) - 10 \]
\[ c(\text{BOTV}) + c(\text{SVP}) - 2 \]
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

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

Yes

Set RT control unit type

c(c(STP)) + -8

Test control unit value i.e. check loop count

Choice

If c(c(TOPV))≤0?

Yes

Branch out of loop

Reset instruction counter to address of BA op for loop (RESET subroutine)

No

Loop again

Dec loop count

Reset line number

c(c(TOPV))-1

+ c(IC) + 3
c(c(TOPV))+c(c(IC))
REPEAT-FOR ROUTINE

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
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

Yes
Loop already entered:
Increment & test loop variable

opcode parameters
\[RF \overline{x} \overline{x} \overline{x} \overline{x} \overline{x} \overline{x} \overline{x}\]
\[c(IC)\]

Increment loop variable (ISV subroutine)
Test loop variable
Choice (rel routine)
Loop variable \(\leq\) test value?
Yes
Loop again
Reset line no.
\[c(IC) + 3\]
\[c(LN) + c(IC)\]
No
Branch out
Reset \(c(IC)\) to address of BA op for loop (RESET subroutine)
Increment c(IC) to address line number parameter

Place 'RFWT' code & c(c(IC)) in PBUF (for trace)

First traversal of loop?

Choice
Flag at c(STP)?

No
Enter loop

Check types of loop control variables & initialize loop variable (CHKSET subroutine)

Reset Stack pointers

c(STP) + 20

Set flag at c(STP) to indicate loop already traversed

c(SVP) + 10

Yes
Loop again

Increment loop variable (ISV subroutine)

Reset Stack value pointer

c(SVP) + c(TOPV) + 2
OP 23

REPEAT-SET ROUTINE

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

First traversal of loop?

Choice
Flag at c(STP)?

Yes
Reset and test loop variable

No

Enter loop

Set flag at c(STP) to indicate loop entered

c(SVP)+10

Set c(SVP) to address first set list elt.

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)

Initialize loop count in first set list element location

Reset Stack value pointer

c(SVP) + c(STP)+9

c(IC)+3
Place c(c(IC)) in PBUF

c(IC)+4
Place c(c(IC)) in PBUF

c(c(SVP)) + 0

opcode parameters

R S x x x x x x x x x x + c(IC)
RESET & TEST LOOP VARIABLE

Increment loop counter

Test count

Choice
\[ c(c(SVP)) = c(c(IC)) \]?

Yes

Branch out of loop

Reset Stack value pointer
\[ c(SVP)+c(c(SVP)+10) \]

Inc c(IC) & place remaining 2 parameters in PBUF (for trace) (FILL subroutine)
\[ c(SVP)+c(c(SVP)) \]

Dec c(IC)
\[ c(IC)-3 \]

Reset c(IC) to address of BA op for loop (RESET subroutine)
\[ c(SVP)+c(c(SVP)) \]

No

Loop again

Reset Stack value pointer
\[ c(SVP)+c(c(SVP))+c(STP)+9 \]

Set c(SVP) to address next set list elt.
\[ c(SVP)+c(c(SVP)) \]

Reset loop variable i.e. store type & value addressed by c(SVP) at c(A) (STOR subroutine)
\[ c(SVP)+c(c(SVP)) \]

Inc c(IC) & place remaining 2 parameters in PBUF (for trace) (FILL subroutine)
\[ c(LN)+c(c(SVP)) \]

Reset line number
\[ c(SVP)+c(c(SVP)+10) \]

Reset Stack value pointer
\[ c(SVP)+c(c(SVP))+c(STP)+9 \]
ISV subroutine to increment loop variable

Save pointer to top of Stack

Set pointers to loop variable location

Increment loop variable

Set flag at IND

Set c(SVP) to loop variable address

Set c(BOTV) to address loop variable mantissa

Dec c(SVP) to address loop variable address

Set c(SVP) to loop variable address

Set flag at IND

c(c(SVP)) + c(c(TOPV)) (ADD routine)

Clear flag at IND

c(BOTV) + c(SVP) - 2

Subroutine parameter

c(SVP) - c(ISV-1) +
Set \( c(\text{COUNT}) \) to number of control variables (parameter to subroutine)

\[ c(\text{COUNT}) + c(\text{CHKSET-1}) \]

CHKSET subroutine to check types of loop control variables & to initialize loop variable

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

\[ c(A) + c(c(SVP)) \]

\[ c(SVP)+10 \]

\[ c(c(A)) + c(c(SVP)) \]

\[ c(A)-9 \]

\[ c(c(A)) + 2 \]

Repeat

Decrement Stack pointers

If \( c(c(\text{STP})) \neq 2 \)

ERROR (RF E1)
Repeat For loop control variable not number

If \( c(\text{COUNT}) = 0 \)

\[ c(\text{SVP})-10 \]

\[ c(\text{STP})-10 \]

(POPTOP subroutine)

Exit

\[ c(c(A)) + c(c(SVP)) \]
**CASE ROUTINE**

- **Increment c(IC)** to address op parameter (no. of case statements)
  - $c(IC)+2$

- **Place 'CS' code and parameter in PBUF (for trace)**

- **Check that values of case expression & bounds are integers**

- **Set pointers:**
  - $c(TOPV)$ to upper bound
  - $c(BOTV)$ to lower bound
  - $c(SVP)$ to expression value

- **Check bounds and branch to appropriate statement**

- **Repeat 3 times**

- **If** $c(c(STP)) 
eq 2$
  - **ERROR (CS E1)**
  - Case parameter not number

- **If** $c(c(SVP)) 
eq 0$
  - **ERROR (CS E2)**
  - Case parameter not integer

- **Dec Stack pointers**
  - $c(SVP)-10$
  - $c(STP)-10$
  (POPTOP subroutine)

- **opcode parameter**
  - $CS \times x$
  - $c(IC)$
If \( c(SVP) > c(TOPV) \), ERROR (CS E3)
Case expression value exceeds upper bound

Subtract lower bound from case control variable

If \( c(SVP) < 0 \)
ERROR (CS E4)
Case range too large

Turn off o'flow

If \( c(SVP) > c(BOTV) \)
ERROR (CS E5)
Case expression value less than lower bound

Check bounds against number of case statements

Check case number against number of case statements

If \( c(SVP) \geq c(IC) \)
ERROR (CS E6)
Case number outside range

Reset instruction counter

Reset Stack value pointer (i.e. pop case expression value & bounds)

Return to Control

If \( c(SVP) = c(IC) + 6 \times c(SVP) \)
Reset Stack value pointer (i.e. pop case expression value & bounds)

If \( c(SVP) - 8 \)
Return to Control
Increment c(IC) to address op parameter (no. of case statements)

Place 'CSWB' code & parameter in PBUF (for trace)

Check case expression type

Decrement Stack pointers

Case number less than 1?

Case number greater than no. of case statements

Reset c(IC) and c(SVP)

Return to Control

If

If

c(c(SVP)) ≠ 0

ERROR (CSWB E2)
Case number not integer

ERROR (CSWB E3)
Case number less than implied lower bound of 1

ERROR (CSWB E4)
Case number greater than no. of case statements

c(IC) + 2

c(STP) - 10
(c(SVP)) - 2 (POPTOP subroutine)

If

If

c(c(SVP)) < 1

ERROR (CSWB E3)
Case number less than implied lower bound of 1

ERROR (CSWB E4)
Case number greater than no. of case statements

If

If

c(c(SVP)) > c(c(IC))

c(IC) + (6 x c(c(SVP)))

c(SVP) - 8

opcode parameter

CS WB x x
↑
c(IC)
RETURN ADDRESS (PROCEDURE) ROUTINE

Place 'RAP' code in PBUF (for trace)

Increment Stack pointers to leave space for Return Address Word

Fill Return Address Word and reset RAW pointer

Return to Control

Increment Stack pointers to leave space for Return Address Word

Fill Return Address Word and reset RAW pointer

Reset RAW pointer

Set type

Link to previous Return Address Word

Reset RAW pointer

c(STP)+10

Set type

Link to previous Return Address Word

Reset RAW pointer

c(SVP)+6

c(c(STP)) + 9

Op code

RA P

+ c(IC)
OP 27

RETURN ADDRESS (FUNCTION)
ROUTINE

Place 'RAF' code in PBUF (for trace)
Increment Stack pointers to leave space for function value
Set function value to 'undefined'
Increment Stack pointers to leave space for RAW
Fill RAW & reset RAW pointer (see RAP routine)
Return to control

opcode
RA F
↑
c(IC)

\[
c(\text{STP}) + 10
c(\text{SVP}) + 10
\]
\[
c(\text{STP}) + 0
\]
\[
c(\text{STP}) + 10
c(\text{SVP}) + 6
\]
CALL PROCEDURE ROUTINE

- Increment c(IC) to address first parameter (S.T. address of PAW)
  \[ c(IC) + 3 \]

- Place 'CP' code & first parameter in PBUF (for trace)

- Set c(A) to core address of Procedure Address Word (type digit)
  \[ c(A) + c(SY~BAS) + (14\times c(c(IC))) - 9 \]

- Check addressed type

  - If \[ c(c(A)) \times 5 \]
    - ERROR (CP E1)
      - CP parameter does not address PAW
  - Increment c(A) to address value part of PAW
  - Inc c(A) to rest of Routine

- Opcode parameters
  \[ CP \bar{X} \bar{X} \bar{X} \bar{X} x + c(IC) \]
Save no. of actual parameters (to be checked against no. of formal parameters on entry to procedure)

Place second op parameter (no. of A.P.s) in PBUF (for Trace)

Store return I.L. address in RAW

Reset instruction counter to address start of procedure

Return to Control

\[ c(\text{IC}) + 2 \]

\[ c(c(\text{RAW})) + c(\text{IC}) - c(\text{ILBASE}) \]

\[ c(\text{IC}) + c(c(A)) + c(\text{ILBASE}) \]

\[ c(\text{AP}) + c(c(\text{IC})) \]

REST OF ROUTINE
PROCEDURE ENTRY
ROUTINE

Increment c(IC) to address first parameter (no. of formal parameters)

c(IC)+2

Place 'PE' code & first parameter in PBUF (for trace)

Place parameter in PBUF (for trace)

Check no. of actual parameters against no. of formal parameters

If c(AP) = c(c(IC))

ERROR (PE E1)
No. of actual parameters not equal to no. of formal parameters

If c(c(IC)) = 0
Increment Stack value pointer to leave req'd space

Set procedure base pointers

Set local variables to 'undefined' (CLEAR routine)

Leave space for local variables (if any) & first possible SMF

Place parameter in PBUF (for trace)

Inc. c(IC) to address second parameter (no. of local identifiers)

c(IC)+2

c(BVP) + c(RAW)
c(BTP) + c(RAW) - 9

Set procedure base pointers

opcode parameters

P E x x x x
+
c(IC)

c(SVP) + (c(c(IC)) x 10) + 20

Return to Control
Check for core overflow

If \( c(SVP) > c(BOUND) \)

ERROR - CORE OVERFLOW

Set upper limit pointer

\[ c(TOPT) + c(WVP) - 29 \]

Repeat

[0] \( c(STP)+10 \) \( c(c(STP)) + 0 \)

If \( c(STP) = c(TOPT) \)

Exit

CLEAR routine to set type codes of Stack variables to 0 ('undefined')

Set upper limit pointer

\[ c(TOPT) + c(WVP) - 29 \]

Clear

Repeat

[0] \( c(STP)+10 \) \( c(c(STP)) + 0 \)

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

Increment c(IC) to address local address parameter

Place 'TL' code and parameter in PBUF (for trace)

Set local variable pointers

Increment Stack pointers

Place 'TL' code and parameter in PBUF (for trace)

Set local variable pointers

Increment Stack pointers

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

Result type - push local variable addr.

Chain down

Locate and push result

Place result and type addressed by c(A) at top of Stack (PUSH ROUTINE)

opcode parameter

TL A ⨷ ⨷

TL R ⨷ ⨷

+ c(IC)
MAKE STORAGE FUNCTION
ROUTINE

opcode parameters

\[ MSF \quad x \quad x \quad x \quad x \]
\[ \uparrow \]
\[ c(IC) \]

- 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

Choice on \( c(c(IC)) \) (array dim)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c(MFP) - 12 )</td>
<td>( c(MFP) - 4 )</td>
</tr>
</tbody>
</table>

- Set flag at IND to indicate 2-dimensional array
- Clear flag at SIZE-3
- \( c(SIZE) + c(c(MFP)) \)

- \( c(MFP) \)
- \( c(SVP) - 8 \)
- \( c(STP) - 10 \)

- \( c(MFP) + c(WVP) \)
Check & load lower array bound value into SMF (LOAD subroutine)

Calculate & check bound range

If \( c(\text{SIZE}) < 999 \)

Set flag at \( \text{SIZE} - 2 \)

Dec. pointers (DEC sub)

Check if load 1st upper bound value into SMF (LOAD subroutine)

Save bound value (DEC sub)

Dec. pointers (DEC sub)

Check if load lower bound value into SMF (LOAD subroutine)

Fill \( c_1 \) in SMF

2-dimensional array?

Fill in array words & reserve space for array elements

If Flag at IND

If \( c(\text{SIZE}) > 999 \)

ERROR (MSF E1) Array size greater than 999

If \( c(\text{SIZE}) > 999 \)

Set flag at \( \text{SIZE} - 2 \)

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 (IC) + 1
- Set COUNT + (IC)
- [c(IC)]

Set count of no. of AW's in array segment
- Set WVP to address of first array element location
- Set MFP to address of SMF
- Set MFP + WVP - 26

Repeat until COUNT = 0
- Fill in AW
- Space for array elts.
- Dec pointers and count
- Tidy up

- Set flag at WVP - 3
- Fill flag at IND?

- Choice
- [c(STP) + 4]
- [c(SVP) + c(MFP)]
- [c(SVP) - 4]
- [c(SVP) + c(WVP)]
- [c(SVP) - 4]
- [c(SVP) + 1]
- [c(SVP) + 0]
- [c(SVP) - 9]

- [c(SVP) + c(BVP) + (c(IC) * 10)]

- [c(STP) + c(SVP)]
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

Clear flag at WVP-3

\[ c(WVP) + c(STKBAS) + 10 \]

\[ c(SVP) + c(WVP) \\
   c(IC) + 1 \]

\[ c(STP) + c(STKBAS) \\
   c(STP) + c(MFP) + 7 \]

Set array element types to 'undefined' (CLEAR routine)
LOAD subroutine to check & load array bound into Storage Mapping Function

Check bound type

If $c(c(\text{STP})) \neq 2$
- ERROR (MSF E2)
- Bound not number

If $c(c(\text{SVP})) \neq 0$
- ERROR (MSF E3)
- Non-integral bound value

Check magnitude

If $c(c(\text{SVP})) \neq 0$
- ERROR (MSF E4)
- Bound magnitude greater than 9999

Load bound value

Define field

$\text{c(SVP)} + 1$
Set flag at $c(SVP)$
$c(SVP) + 3$

Load

$c(c(MFP)) + c(c(SVP))$
OP 34

RETURN ROUTINE

Place 'RE' code in PBUF (for trace)

Reset Stack pointers to address unit below RAW

Reset Instruction Counter to return I.L. address

Reset RAW pointer to address next Return Address Word

Reset Procedure base pointers

\[ \text{opcode} = R E + c(IC) \]

\[ c(SVP) + c(BVP) - 10 \]
\[ c(STP) + c(BTP) - 10 \]

\[ c(IC) + c(c(RAW)) + c(ILBASE) \]

\[ c(RAW) - 4 \]

\[ c(RAW) + c(c(RAW)) \]

\[ c(BVP) + c(RAW) \]
\[ c(BTP) + c(RAW) - 9 \]
OP 35

END ROUTINE

Place 'END' code in PBUF (for trace)

Print and clear contents (if any) of PBUFF (OUTPUT subroutine)

Print 'PHASE COMPLETE'

Print \texttt{c(PBUF)} (for trace)

If Switch 2 on

CALL EXIT

END

c(IC)
OP 36

Place 'RD' code in PBUF (for trace)

Set c(A) to address (?) at top of Stack

Check addressed type

Input (next) card element

Return to Control

If c(c(STP)) ≠ 6

ERROR (RD E1)

c(A) + c(c(SVP))

Choice c(c(A))=4?

Yes

Prepare to input array i.e. set c(A) to address first array elt. location, etc. (ARSUB subroutine)

No

Simple input - inc. c(A) to address value part of input location

c(A) + 9
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

Check next READ buffer character

Number?

If \[ c(c(RBUF MK)) = 7 \]

Assemble & store number

Check next READ buffer character

Set flag at \[ c(RBUF MK) \]

Set flag at \[ c(RBUF MK) + 1 \]

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

Store at \[ c(A) \]

(STOR subroutine)

\[ c(c(STP)) + 0 \]
ASSEMBLE AND STORE NUMBER

Place 'number' type at top of Stack

\[ c(c(STP)) + 2 \]

Skip over non-significant zeros

Repeat

Character zero?

\[ c(RBUFMK) + 1 \]

IF \[ c(c(RBUFMK)) \neq 0 \]

Exit (first sig dig)

Next character digit?

\[ c(RBUFMK) + 1 \]

IF \[ c(c(RBUFMK)) \neq 7 \]

If \[ c(c(RBUFMK)) \leq 7 \]

Number zero accumulated so far - check next character

Set flag at \[ c(RBUFMK) \]

Choice on \[ c(c(RBUFMK)) \]

Assemble & store real number

'.'

'E'

or blank

other

Skip exponent & pack zero result

Store zero result

ERROR (RD E2)

Invalid input form
Skip exponent and store zero result

Repeat

Check next character

- \( c(\text{RBUFMK}) + 1 \) (BUFMAN subroutine)
- Set flag at \( c(\text{RBUFMK}) \)
- \( c(\text{RBUFMK}) + 1 \)

Comma or blank?

- If \( c(c(\text{RBUFMK})) = 23 \) or 0
- Exit

Store zero result

Place value '0' at top of Stack

Store at \( c(A) \) (STOR subroutine)

- \( c(c(SVP)) + c(\text{FZERO}) \)
Pack up to next 8 digits into number accumulator - then store

Repeat

Pack next digit

- Increment accumulator pointer
  - \( c(\text{NUMBER}) + 1 \)

- Pack digit
  - \( c(\text{RBUF MK}) + 1 \)
  - \( \text{BUFMAN subroutine} \)

- Increment mantissa count
  - \( c(\text{MCOUNT}) + 1 \)

- Clear (possible) flag at \( c(\text{NUMBER}) \)
  - \( c(c(\text{NUMBER})) + c(c(\text{RBUF MK})) \)

Check mantissa count

- If \( c(\text{MCOUNT}) = 8 \)
  - Skip over remaining mantissa digits - then round and store result

Next character digit?

- Choice \( c(c(\text{RBUF MK})) = 7? \)
  - Yes
    - \( c(\text{RBUF MK}) + 1 \)
  - No
    - Terminate integer phase

- \( \text{c(MCOUNT) + 1} \)
**SKIPP OVER REMAINING MANTISSA DIGITS - THEN ROUND & STORE RESULT**

**Repeat**

Next character digit?

Choice

$c(c(RBUFMK))=7$?

- **Yes**
  - Inc. buffer marker
  - $c(RBUFMK)+2$ (BUFMAN subroutine)

- **No**
  - Inc. exponent accumulator
  - $c(EAC)+1$

Exit

**Terminate Integer Phase**

Check next character

Set flag at $c(RBUFMK)$ $c(RBUFMK)+1$

Choice on $c(RBUFMK)$

- $'$, $.$, or blank
- $'$
- 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

Place integer at top of Stack
- \(c(SVP) + c(FZERO)\)
- \(c(BOTV) + c(NUMBER)\)
- Move flag from IND to \(c(BOTV)\)

Store at \(c(A)\) (STOR subroutine)

No

Print warning error message - then round, normalize, and store number

Print warning error message - too many significant digits in input number

Load rounding accumulator & set pointers
- \(c(MAC+1) + c(NUMBER)\)
- \(c(EAC) + 1\)
- \(c(HOD) + MAC-6\)
- \(c(LOD) + MAC\)
- Set INFLAG

Round, normalize, & store number (RNS routine)
If \( c(MCOUNT) = 0 \)

Skip over leading zeros

Repeat

Next character digit?

If \( c(c(RBUFMK)) \neq 7 \)

Number zero accumulated so far - check next character

If \( c(c(RBUFMK)) = 0 \)

Exit (first sig dig)

If \( c(c(RBUFMK)) \neq 7 \)

Decrement exponent accumulator

\( c(EAC) = 1 \)

Check mantissa count

If \( c(MCOUNT) = 8 \)

Skip remaining decimal digits - then round & store result

Next character digit?

If \( c(c(RBUFMK)) \neq 0 \)

\( c(RBUFMK) + 1 \)

(BUFMAN subroutine)

Pack up to 8 mantissa digits in number accumulator - then store result

Repeat


Set flag at c(RBUFMK) + 1 (BUFMAN subroutine)

Repeat

Next character digit?

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\'

Assemble & store floating point number

\',\', or 6

Pack real number

ERROR (RD E2)

Invalid input form

\'\', or 6

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))

'+' '−'

Check first exponent character

Set flag at IND-1

Check first exponent character

ERROR (RD E2) Invalid input form

Digit zero?

IF c(c(RBUFMK)) ≠ 0

IF c(c(RBUFMK)) ≠ 7

Zero exponent

Exit

Next character digit?

Clear flag at IND-1

Set flag at c(RBUFMK)
c(RBUFMK)+1

Choice on c(c(RBUFMK))

'.' on bl. '−'

Pack real number

ERROR (RD E2) Invalid input form

Skip over non-significant exponent zeros

Repeat

Fill exponent accumulator & terminate exponent phase
FILL EXPONENT
ACCUMULATOR &
TERMINATE
EXPONENT PHASE

Repeat

Pack latest
exponent digit

Next character
digit?

c(EXPR) + 1

If

Check exponent
size

c(EXPR) = XAREA+1

ERROR (RD E3)
Exponent
overflow

Terminates
exponent
phase

c(EXPR) = XAREA+1

If

c(EXPR) + 1

c(EXPR) = c(EXPR) + c(EXPR)
Clear (possible) flag
at c(EXPR)

c(EXPR) + 1

If

BUFMAN
subroutine

c(EXPR) + 1

If

BUFMAN
subroutine

c(EXPR) + 1

If

BUFMAN
subroutine

c(EXPR) + 1
TERMINATE EXPONENT PHASE

Check next character

Set flag at \( c(RBUFMK) \)
\( c(RBUFMK) + 1 \)

Choice on \( c(RBUFMK) \)

',,' or blank

Calculate exponent

Append sign to accumulator
Add to decimal point count

Move flag from IND-1 to \( c(EXP) \)
\( c(EAC) + c(c(EXP)) \)

Zero mantissa?

If \( c(MCOUNT) = 0 \)

Place mantissa value '1' at top of Stack
\( c(c(SVP)) + c(FONE) \)

Fill in exponent at top of Stack
\( c(c(SVP)) + c(EAC) \)

Append mantissa sign
Move flag from IND to \( c(BOTV) \)

Store TOS number at \( c(A) \) (STOR subroutine)

ERROR (RD E2)
Invalid input form
INPUT STRING

Place 'string descriptor' type digit at top of Stack
\[ c(c(STP)) \rightarrow 7 \]

Set c(NODE2) to address next available string node
(�GRAB subroutine)
\[ c(c(SVP)) + c(NODE2) \]
\[ c(LENGTH) \rightarrow 0 \]

Fill address field in string descriptor and initialize length counter

Pack string away in string area

Terminate string input

OUTLOOP:
Repeat

Pack a node

Peel off up to next 6 string characters
\[ c(STRING) +1 \]
\[ c(NODE2) -4 \]
Set flag at SAREA-12
\[ c(c(NODE2)) \rightarrow c(c(STRING)) \]

Pack in string area

Save
\[ c(NODE2) +4 \]
\[ c(NODE1) \rightarrow c(NODE2) \]

Link to next available node

Grab
\[ Set c(NODE2) to address next available node (�GRAB subroutine) \]
\[ c(c(NODE1)) \rightarrow c(NODE2) \]
Clear flag at NODE2-3

Set flag at NODE2-3

Link
Initialize accumulator pointer & loop counter

Repeat

PEEL OFF UP TO NEXT 6 STRING CHARACTERS

Check next character

Pack character in accumulator

Inc. length counter

Exit?

Check next character

If \( c(c(RBUF MK)) = '' \)

c(STRING) + 3

If \( c(c(STRING)) + c(c(RBUF MK)) = c(STRING) - 1 \)

Clear flag at c(STRING)

c(LENGTH) + 1

Exit

If \( c(STRING) + 3 + c(c(RBUF MK)) = c(STRING) \)

Convert to internal form

ERROR (RED2)

Invalid input form

Exit from OUTLOOP

Exit

c(RBUF MK) + 1

(BUFMAN subroutine)

Set flag at c(RBUF MK) c(RBUF MK) + 1

c(RBUF MK) + 1

(BUFMAN subroutine)

Set flag at c(RBUF MK) c(RBUF MK) + 1

' ' or blank

Exit

'!!'

'!!'

other

Convert to internal form

ERROR (RED2)

Invalid input form

Exit

c(RBUF MK) + 1

(BUFMAN subroutine)

Set flag at c(RBUF MK) c(RBUF MK) + 1

Check next character

c(RBUF MK) + 1

(BUFMAN subroutine)

Set flag at c(RBUF MK) c(RBUF MK) + 1

Check next character

Choice on c(c(RBUF MK))

""
**TERMINATE STRING INPUT**

Null string?

**Choice**
c(LENGTH)=0?

Yes

Place 'null str.' type digit at TOS  
c(c(STP)) + 1

No

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

Store TOS descriptor at c(A) (STOR subroutine)

No

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

Store TOS descriptor at c(A) (STOR subroutine)

**Terminate string at NODE1**

Set flag at c(NODE1)
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))

'F'

Place value FALSE at TOS

'F'

Place value TRUE at TOS

Skip remaining characters until comma or blank

Check next character

Place value FALSE at TOS

Store TOS Boolean value and type at c(A) (STOR sub.)

Comma or blank?

If c(c(RBUFMK)) = 23 or 0

Exit

Set Boolean type digit at top of Stack

Place value TRUE at TOS

Place value FALSE at TOS

Repeat

Check next character

Set flag at c(RBUFMK) + 1 (BUFMAN subroutine)

c(RBUFMK) + 1

01

c(SVP) + 01

00

c(SVP) + 00

c(SVP) + 00

c(SVP) + 00

01
Increment array element counter
\[ c(ACCOUNT) + 1 \]

Check count against array size

Choice
\[ c(ACCOUNT) > c(SIZE)? \]

Yes
Terminate array input
Clear flag at ARRAY

No
Input next card element

Pop top of Stack
\[ c(SVP) - 10 \]
\[ c(STP) - 10 \]
(POPTOP subroutine)
Increment buffer marker

Check against buffer end

If
\[ c(\text{RBUFMK}) \geq \text{RBUF}+158 \]

Read new card into READ buffer

Reset marker

\[ c(\text{RBUFMK}) - 160 \]

BUFMAN subroutine to control increments to READ buffer marker \((c(\text{RBUFMK}))\)
PRINT0 routine to print 'zero' in standard F-format

- Leave 7 spaces in PRINT buffer
  - c(PBUFMK) + 16 (PBFMAN subroutine)

- Insert '0'
  - c(c(PBUFMK)) + 70

- Set 'space' flag (see SPACE routine for explanation)

- Leave 12 spaces in PRINT buffer
  - c(PBUFMK) + 24 (PBFMAN subroutine)

- Check for array O/P (BACHEC routine)
OP 37

**PRINT ROUTINE**

- Place 'PR' code in PBUF (for trace)
- Mark top of Stack and set count
  - \(c(\text{TOPV}) + c(\text{SVP})\)
  - \(c(\text{COUNT}) + 1\)
- Check top Stack type and print
- Transmit to 2-digit field (for comparison)
  - \(c(\text{COMP}) + c(\text{c(STP)})\)
- Check for array output (BACHEC routine)
- **Choice** on \(c(\text{COMP})\)
  - '1': Output null string
  - '2': Leave 4 blank spaces (PBFMAN subroutine)
  - '3': Bubble addressed type & value to top of Stack (BUBBLE subroutine)
  - '6': Check top Stack type and print
  - '17': Output string
  - Other: Incorrect PRINT operand
Set $c(A)$ to address type of addressed unit

Check type

If $c(\text{COMP}) = 4$

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

Place copy of addressed type & value at top of Stack

Repeat until $c(P) > c(SVP)$

Add $c(P) + c(\text{STP})$

Add $c(c(P)) + c(c(A))$ (transmit digit)

Add $c(P) + 1$

Add $c(A) + 1$

Add $c(A) - 9$
Set `c(LENGTH)` to length of number's mantissa (LSUB subroutine)

Leave 16 spaces in PRINT buffer for number

Check size of number and determine output format

- `c(PBUPMK) + 32` (PBFMAN subroutine)
- `c(LENGTH) + c(c(SVP))`

- More than 7 digits before or after decimal point

  Choice

  \[
  \text{Choice:} \quad \begin{cases} 
  c(LENGTH) > 7 \\
  c(c(SVP)) < -7
  \end{cases}
  \]

  Yes

  Output number in standard E-format

  No

  Clear SIGFLG

  Output number in standard F-format
Output number in standard F-format

Set pointer to first pattern digit & initialize count of places before decimal point

\[ \text{c(POINT)} + \text{FPAT-16} \]
\[ \text{c(BEFORE)} + 8 \]

Dec. PRINT buffer marker to address first output digit position

\[ \text{c(PBUFMK)} - 30 \]
\[ \text{c(BEFORE)} - 1 \]
\[ \text{c(POINT)} + 1 \]

Check pattern digit

Choice on \( \text{c(c(POINT))} \)

\[ 1 \]
\[ 0 \]
\[ 1 \]

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

(Note: \( \text{c(FPAT)} = 1111111101111111 \)

-standard F-format pattern)

\( \text{c(PBUFMK)} + 8 \)

(PBFMAN subroutine)
Set SIGFLG

Any significant digits before decimal point

If no SIGFLG

Set SIGFLG

Insert '0' before decimal point

Insert sign (if minus)

If IND flag

Insert decimal point if not integer

If c(c(SVP)) ≠ 0

c(c(PBUFMK)) + 03

c(c(PBUFMK)) + 70

Insert decimal point

c(PBUFMK) + 2

If c(c(PBUFMK)) + 2070

c(c(PBUFMK)) + 2070

c(BEFORE) + 1

c(PBUFMK) + 2

INSERT DECIMAL POINT IF NON-INTEGRAL NUMBER
Insert digit or blank in print buffer

**Choice**

\[ c(\text{BEFORE}) > c(\text{LENGTH})? \]

- **Yes**
  - Leave blank space if SIGFLG not on - otherwise insert zero
  - \[ c(\text{PBUFMK}) + 2 \]

- **If** SIGFLG on
  - \[ c(c(\text{PBUFMK})) + 70 \]

- **If** no SIGFLG
  - Set SIGFLG (if minus)
    - **If** IND flag
      - \[ c(c(\text{PBUFMK})) + 20 \]
    - **If** no IND flag
      - Append sign
      - \[ c(c(\text{PBUFMK})) + 70 \]

- **No**
  - Place next digit in PRINT buffer
    - \[ c(\text{PBUFMK}) + 2 \]
    - Any mantissa digits left?
      - **Choice**
        - \[ c(P) \leq c(\text{BOTV})? \]
          - **Yes**
            - Fill next digit
            - \[ c(c(\text{PBUFMK})) + 70 \]
          - **If** c(\text{BEFORE}) > 0
            - \[ c(c(\text{PBUFMK})) + c(c(P)) \]
            - (transmit digit)
            - \[ c(P) + 1 \]
            - \[ c(c(\text{PBUFMK})) + 70 \]
          - **No**
            - Insert trailing zeros (if before decimal pt.)
            - \[ c(c(\text{PBUFMK})) + 70 \]
Dec. PRINT buffer marker to address first output digit position

\[ c(\text{PBUFMK}) = 22 \]

Calculate exponent part

\[ c(\text{LENGTH}) - c(\text{SVP}) - 1 \]
\[ c(\text{EAC}) = 000 \]
\[ c(\text{EAC}) + c(\text{SVP}) + c(\text{LENGTH}) \]

Insert sign (if minus), first digit, & decimal point

\[ c(\text{PBUFMK}) + 70 \]

Append sign (if minus)

Insert first digit

\[ c(\text{PBUFMK}) + (c(\text{P})) \]
[transmit digit]
\[ c(\text{P}) + 1 \]

If IND flag

\[ c(\text{PBUFMK}) + 2070 \]

Insert decimal point

\[ c(\text{PBUFMK}) + 2 \]
\[ c(\text{PBUFMK}) + 03 \]
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 \( c(P) \neq c(BOTV) \)

- \( c(c(PBUFMK)) + c(c(P)) \) (transmit digit)

- \( c(c(PBUFMK)) + c(P) \) (transmit digit)

If Flag at EAC

- Insert 'E+'
- Insert minus if negative exponent
- Fill exponent digits
- Set 'space' flag (see SPACE routine for expl.)
- Leave 4 blank spaces

If \( c(PBUFMK) + 4 \), \( c(c(PBUFMK)) + 4510 \)

Clear flag

Insert minus

- \( c(c(PBUFMK)) \) (transmit numeric fill)

Leave space after number

\( c(PBUFMK) + 8 \) (PBFMAN subroutine)

\( c(c(PBUFMK)) + 20 \)
OUTPUT BOOLEAN VALUE

Inc. PRINT buffer marker

\[ c(PBUFMK) + 10 \]
(PBFMAN subroutine)

Fill in Boolean value

Choice on \[ c(c(SVP)) \]

0

\[ c(c(PBUFMK)) + c(FALBUF) \]
('FALSE')

1

\[ c(c(PBUFMK)) + c(TRUBUF) \]
('TRUE ')

Leave space

Set SPFLG (see SPACE routine for explanation)

Leave 4 spaces

\[ c(PBUFMK) + 8 \]
(PBFMAN subroutine)
Set \( c(\text{NODE1}) \) to address first string node and set \( c(\text{TOPREM}) \) to number of digits in terminal node.

STREM subroutine

Last node?

If Flag at \( c(\text{NODE1}) \)

Exit

Reset Stack value pointer

\( c(\text{SVP}) + 5 \)

Pack string in PRINT buffer

Pack terminal node

Set SPFLG

Leave space after string

Leave 4 bl spaces

\( c(\text{PBUFMK}) + 8 \) (PBFMAN sub.)

Repeat

\( c(\text{PBUFMK}) + (\text{PBFMAN sub.}) \)

\( c(\text{NODE1}) - 4 \)

\( c(c(\text{PBUFMK})) + c(c(\text{NODE1})) \)

\( c(\text{NODE1}) + 4 \)

\( c(\text{NODE1}) + c(c(\text{NODE1})) \)

Clear flag at \( \text{NODE1} - 3 \)
STREM subroutine to set \( c(\text{NODE1}) \) to address first string node, & to set \( c(\text{TOPREM}) \) to no. of digits in terminal node

Set \( c(\text{NODE1}) \)

\[ c(\text{NODE1}) + c(\text{SVP}) \]

Find remainder

Divide string length by 6

\[ c(\text{SVP}) - 5 \]

Divide \( c(\text{SVP}) \) by 6

\[ c(\text{TOPREM}) + 2 \times \text{'remainder'} \]

Store remainder

If \( c(\text{TOPREM}) = 0 \)

\[ c(\text{TOPREM}) + 12 \]
OP 38

Inc. c(IC) to address first parameter (no. of list elts. in PIC spec.)

Place 'PRC' code & first parameter in PBUF (for trace)

Mark beginning of PICTURE and store count of no. of list elts in PIC spec.

Mark Stack top & then dec. Stack pointers to address first PIC parameter

Output next PIC controlled element

Check for array output or multiple PIC spec. (BACHEC routine)

opcode parameters

PR C x x x x . . . x 0 ↑
c(IC)

'7'
Output (formatted) string

'11'
Output (formatted) null string

'12'
Output (formatted) number

'13'
Output (formatted) Boolean value

'6'
Bubble addressed type & value to unit addressed by c(SVP) (BUBBLE subroutine)

Other

Output next PIC controlled element

ERROR (PRC E1)
Invalid PRC operand
Set before decimal, after decimal, and exponent counts to zero

\[ c(\text{BEFORE}) + 0 \]
\[ c(\text{AFTER}) + 0 \]
\[ c(\text{XC}) + 0 \]

Clear exponent, negative, and positive flags

\[ \text{Clear EFLG, NFLG, & PFLG} \]

\[ \text{c(P)} \leftarrow \text{BEFORE} \]

Preliminary format scan

\[ \text{Inc. c(IC) to address next PIC character} \]

\[ \text{c(IC)+2} \]
Set flag at c(IC)-1

\[ \text{Add to I I} \]
Check PIC character

\[ \text{Rescan PIC & output number} \]

Check for array O/P or multiple PIC spec. (BACHEC routine)

Choice on c(c(IC))

\[ \text{Switch to counting digits after d.p.} \]
\[ \text{(Continue) preliminary format scan} \]

\[ \text{c(P)} \leftarrow \text{AFTER} \]

Ignore - (continue) preliminary format scan

Add to total

\[ \text{c(c(P))} + \text{c(TOT)} \]
Check PIC character

\[ \text{c(P)} \leftarrow \text{XC} \]

\[ \text{Switch to counting digits in exponent} \]

\[ \text{(Continue) preliminary format scan} \]

\[ \text{Choice EFLG?} \]

\[ \text{Output no. in E-format} \]
\[ \text{Output no. in F-format} \]

\[ \text{ERROR (PRC E2) Invalid PIC char.} \]

Choice on 0*

\[ \text{Count next sequence of asterisks (or nines) (KOUNT sub.)} \]

\[ \text{Set EFLG} \]

\[ \text{Add to total} \]

\[ \text{Check PIC character} \]

\[ \text{Switch to counting digits in exponent} \]

\[ \text{(Continue) preliminary format scan} \]

\[ \text{Choice EFLG?} \]

\[ \text{Output no. in E-format} \]
\[ \text{Output no. in F-format} \]

Choice on 'F'

\[ \text{Other} \]

Choice on '*' or 'g'

Choice on 'E'

Choice on '0+'

Choice on 'B', 'S', '-'

(Continue) preliminary format scan
OUTPUT NUMBER IN F-FORMAT

Set c(LENGTH) to length of mantissa (LSUB subroutine)

Add exponent (to give digits before decimal point)

Check against BEFORE count

IF c(LENGTH) > c(BEFORE)

ERROR (PRC E3)
Overflow - number too large for specified PICTURE

Else

Assemble zero for output

c(LENGTH) + 1

Round number (if necessary)

c(P) ← c(BOTV)
c(c(P)) ← 0

Choice c(LENGTH) < 0?

Yes

No
Set $c(BOTV)$ to "add 5" position

If $c(c(SVP)) + c(AFTER) < 0$

- Enlarge field
- Set $c(BOTV)$
- $c(P) - 1$
- Clear flag at $c(P)$
- Assemble zero for output

Add

- $c(c(BOTV)) + 5$
- $c(BOTV) - 1$
- $c(P) + 1$
- $c(L) = c(BEFORE)$
- If $c(BOTV) < c(P)$
- Assemble zero for output
- If $c(L) = c(BEFORE)$
- ERROR (PRC E3)
- Overflow - number too large for PIC

If $c(c(SVP)) + c(AFTER) < 0$

- ROUND NUMBER (IF NECESSARY)

Add 5

Choice

Digit at $c(P)$?

No

Yes
Set OFLG
Set c(LENGTH) to length of mantissa (LSUB sub.)
Clear OFLG
Calculate exponent part
Set c(BEFORE) to total O/P mantissa length
Round mantissa if necessary
c(LENGTH) + c(BEFORE)
Find exponent length

OUTPUT NUMBER
IN E-FORMAT

If c(LENGTH) > c(BEFORE)

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
Assemble zero exponent for output

\[
\begin{align*}
\text{c(XL)} &= 1 \\
\text{c(POINT)} &= 0 \\
\text{c(c(POINT))} &= 0
\end{align*}
\]

Set \text{c(POINT)} to address first significant digit of exponent

\[
\text{c(POINT)} + \text{EAC-2}
\]

Check for overflow

\[
\text{If } \text{c(XL)}>\text{c(XC)}
\]

Overflow - exponent too large for specified PIC

Calculate exponent length

\[
\text{c(XL)} + \text{EAC-2} - \text{c(POINT)+1}
\]

If digit at \text{c(POINT)}

\[
\text{c(POINT)+1}
\]

Exit

FIND EXPONENT LENGTH

Choice \text{c(EAC)=0?}

Yes

No

Repeat
RESCAN PICTURE & OUTPUT NUMBER

- Reset c(IC) to address first PIC character
  \[ c(IC) + (ICM) + 2 \]

- Check first Character
  \[ \text{Choice } c(c(IC)) = 'S' ? \]
  - Yes
    - \[ c(\text{PBUFMK}) + 2 \]
    - \[ (\text{PBFMAN subroutine}) \]
    - \[ c(IC) + 2 \]
    - Yes
      - Set NFLG
    - No
      - Set PFLG
  - No
    - \[ c(\text{PBUFMK}) + 2 \]
    - \[ (\text{PBFMAN subroutine}) \]
    - \[ c(IC) + 2 \]
    - If \[ c(c(IC)) = '-' \]
      - If Flag at IND
        - Set NFLG
FILL NEXT DIGIT

Choice

$c(\text{BEFORE}) > c(\text{LENGTH})$?

Yes

$c(\text{BEFORE}) - 1$

No

Choice

SIGFLG off?

Yes

Leave blank

$c(\text{PBUFMK}) + 2$

(PBFMAN subroutine)

No

Insert floating sign (if any) & '0' (FLOATS subroutine)

Set SIGFLG

If $c(\text{P}) \leq c(\text{BOTV})$

Check next PIC character in rescan

Fill digit

$c(c(\text{PBUFMK})) + c(c(\text{P}))$

$c(c(\text{P})) + 1$
OUTPUT EXPONENT PART

Insert 'E'
- c(PBFMK) + 2
- (PBFMAN subroutine)

Clear PFLG & NFLG
- c(c(PBFMK)) + 'E'

Check for zero exponent
- Shift exponent sign
- Move flag from EAC to IND

Choice OFLG?
- No
  - Shift exponent sign
  - Move flag from EAC to IND
- Yes
  - Switch variables and pointers
    - c(BEFORE) ← c(XC)
    - c(LENGTH) ← c(XL)
    - c(P) ← c(POINT)
    - c(BOTV) ← c(SVP)

Rescan & output exponent (as for mantissa)
- Rescan PIC and output number
  - (But DO NOT first reset c(IC))

Clear IND flag
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

Repeat

Lay out terminal node

If

Lay out node

Leave space for node

Check for o'flow to str. area

If

ERROR - CORE OVERFLOW

Last node?

If

Flag at c(NODE1)

Exit

If

c(SBUF MK) to address TOS

Initialize

If

c(NODE1)-4 c(P)+c(SBUF MK)
c(SBUF MK)+12 c(c(SBUF MK))
c(c(NODE1))
c(P) + 1 Clear flag at c(P)

If

c(SBUF MK)>

If

If

c(BOUND)
c(c(NODE1))
c(P) + 1 Clear flag at c(P)
Check for 'null' PIC

Inc. c(IC)

If Record mark at c(IC)

c(IC)+2 Set flag at c(IC)-1

c(TOT2) + c(LENGTH) Output next chunk of string

Fill next portion

c(PBUFMK) + c(TOT2) (PBFMAN subroutine)

PATTERN MATCH STRING TO PRINT BUFFER

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 last portion & then pad out with blanks

Fill next portion

If c(LENGTH) ≠ 0

Pad out with blanks

Exit

'OF'

'OB'

 other

Count no. of B's (blanks) (KOUNT sub.)

c(PBUFMK)+ c(TOT2) (PBFMAN subroutine)

ERROR (PRC E2) Invalid PIC character

Count no. of asterisks (KOUNT sub.)

Output next chunk of string

Fill last portion & then pad out with blanks

Pad out with blanks

Exit
OUTPUT (FORMATTED)
NULL STRING

Set c(TOT2) to zero

c(TOT2) = 0

Increment c(IC) to address first PIC character

c(IC) = c(IC) + 2

Pad out with blanks

Insert trailing blanks

c(PBUF MK) + c(TOT2) (PBFMAN sub.)

Skip over remaining PIC characters

Repeat

End of PIC?

If Record mark at c(IC)

Exit

Increment c(TOT2)

c(TOT2) = c(TOT2) + 2

Increment c(IC)

c(IC) = c(IC) + 2
Determine Boolean value

Choice on c(c(SVP))

01

Set pointer to 'TRUE' buffer

c(BOOBUF) + TRUBUF

00

Set pointer to 'FALSE' buffer

c(BOOBUF) + FALBUF

Inc. c(IC) to address first PIC character

c(IC) + 2

Count number of asterisks in PICTURE (KOUNT subroutine)

If no record mark at c(IC)

ERROR (PRC E2) Invalid PIC character

'Chop' Boolean value

Choice c(TOT2)<10?

Yes

Fill 'chopped' value

Fill value

Pad out with blanks

No

Count number of asterisks in PICTURE (KOUNT subroutine)

Fill 'chopped' value

Fill value

Pad out with blanks

Set record mark at c(IC)

Fill 'chopped' value

Fill value

Pad out with blanks

Set 'space' flag (see SPACE routine for explanation)
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

Choice on print routine

PRC

Assemble zero for output

Rescan PIC and output number

PR

Repeat until digit at $c(P)$

$c(P) + c(BOTV) - 6$

Print '0' in standard F-format (PRINT0 routine)

$c(P) + 1$

No

Move flag from $c(BOTV)$ to IND

$c(LENGTH) + c(BOTV) - c(P) + 1$
FLOATS subroutine to insert floating sign (if any) and zero

Floating plus?
If PFLG on
Clear PFLG
c(c(PBUFMK)) + '+'

Floating minus?
If NFLG on
Clear NFLG
c(c(PBUFMK)) + '-'

Insert alphameric zero

c(c(PBUFMK)) + 2
(PBFMAN subroutine)
c(c(PBUFMK)) + 70
REPSUB subroutine to set up or update REP count

If no REPFLG
- Set REPFLG
- Accumulate and store REP count
  - \( \text{c(REPCNT)} + \text{NACC-9} \)
  - \( \text{c(RCOUNT)} + 1 \)
  - \( \text{c(IC)} + 1 \)

Repeat

Choice
  - \( \text{c(c(IC))} = 7? \)
    - Yes
      - \( \text{c(IC)} + 1 \)
      - \( \text{c(REPCNT)} + 1 \)
      - \( \text{c(c(REPCNT))} + \text{c(c(IC))} \)
      - \( \text{c(RCOUNT)} + 2 \)
    - No
      - Clear (possible) flag at \( \text{c(REPCNT)} \)

Dec. and test REP count
  - \( \text{c(c(REPCNT))} - 1 \)
  - \( \text{c(IC)} - 2 \)
  - If \( \text{c(c(REPCNT))} = 0 \)
    - Clear REPFLG
    - \( \text{c(IC)} + \text{c(RCOUNT)} + 5 \)
  - No
    - \( \text{c(IC)} - \text{c(RCOUNT)} \)
    - Exit
BACHEC routine to check for array output or multiple PIC spec.

Choice
Flag at ARRAY?

Yes

Inc. array elt. count

No

Check for end of PIC control

Choice

Check against array size
c(ACCOUNT) > c(SIZE)?

Yes

c(ACCOUNT)+1

No

Inc. Stack pointers to addr. next PRC parameter
c(SVP)+10
c(STP)+10

Output next PIC controlled element

Yes

Return to Control

No

Check top Stack type and print

c(SVP-1)-c(COUNT)
c(STP-1)-c(COUNT)

Pop print parameters

Check for end of PIC control

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

Bubble array elt. type & value to top of Stack (BUBBLE subroutine)
ARSUB subroutine to prepare for array input or output

- Set ARRAY flag
- Initialize array element counter
  - $c(ACCOUNT) + 1$
  - Inc. $c(A)$ to address dimension digit in AW
    - $c(A) + 1$
- Set base
- Set $c(W)$ to address first array element
  - $c(A) + 4$
  - $c(W) \leftarrow c(c(A)) + c(BASE)$
- Local or main array?
  - Yes
    - $c(BASE) + c(STKBAS)$
  - No
    - $c(BASE) \leftarrow c(SYMBAS)$
- Choice
  - Flag at COMP?
    - Yes
      - $c(BASE) + c(STKBAS)$
    - No
      - Clear flag at COMP
- Remainder of Subroutine
  - Set $c(A)$ to address Storage Mapping Function
    - $c(A) + 4$
    - $c(A) \leftarrow c(c(A)) + c(BASE)$
REMAINDER OF ARSUB SUBROUTINE

Calculate array size

Calculate range of first bound pair

Store upper bound
- $c(A) + 4$
- $c(SIZE) + c(c(A))$

Subtract lower bound
- $c(SIZE) - c(c(A))$

Add one
- $c(SIZE) + 1$

Set $c(A)$ to address first array element
- $c(A) + c(W)$

2-dimensional array?
- If $c(COMP) = 1$
  - Multiply by $C1$
    - $c(A) + 16$
    - $c(SIZE) \times c(c(A))$ (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)

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

- Choice \( c(c(IC)) = 24 \)?
  - No: Exit
  - Yes: Left bracket - accumulate REP count
    - \( c(REPCNT) \) + NACC-9

Yes: Inc. totals

- \( c(TOT) + 1 \)
- \( c(TOT2) + 2 \)

Set flag at \( c(IC) - 1 \)

- \( c(IC) + 2 \)

Repeat

Choice

- \( c(c(IC)) = 7 \)?
  - Yes: Inc. c(IC)
    - \( c(IC) + 3 \)
    - SF c(IC)-1
    - Check next PIC character
  - No:
    - \( c(IC) + 1 \)
    - \( c(REPCNT) + c(c(c(IC))) \)
    - Clear (possible) c(REPCNT) flag
    - \( c(TOT) + c(c(REPCNT)) - 1 \)
    - \( c(TOT2) + 2xc(c(REPCNT)) - 2 \)
    - \( c(IC) + 3 \)
NEWPAGE ROUTINE

- Place 'NP' code in PBUF (for trace)
- Print & clear contents (if any) of output buffer (OUTPUT sub.)
- Skip to newpage
- Return to Control

opcode
N P +
c(IC)
Increment c(IC) to address op parameter (no. of lines to be skipped)

Place 'NL' code and parameter in PBUF (for trace)

Print & clear contents (if any) of output buffer (OUTPUT subroutine)

Skip lines

Repeat c(c(IC)) times

Skip a line

Return to Control

Opcode parameter

N L × x

c(IC)
Increment $c(IC)$
to address op parameter (no. of spaces to be left)

Place 'SP' code and parameter in PBUF (for trace)

Leave spaces

Return to Control

Set 'space' flag (SPFLG)
(NOTE - this signifies to the PBFMAN subroutine that, in the event of buffer overflow, when the contents of the buffer should be printed & cleared, the buffer marker should be reset to the start of the buffer i.e. no spaces should be left at the start of the newline)

Inc. output buffer marker, $c(PBUFMK)$, by $c(c(IC))$
(PBFMAN subroutine)
Increment \( c(IC) \) to address op parameter (TAB column position) 

Place 'TAB' code and parameter in PBUF (for trace) 

Check TAB parameter 

Reset output buffer marker 

Return to Control 

If \( c(c(IC)) \geq 120 \) 

\[ c(PBUFMK) + PBUFF + (c(c(IC)) \times 2) - 2 \]

ERROR (TAB E1) 
TAB position greater than printer width

opcode parameter 

\( \text{TAB} \times \times \times \) 

\[ + c(IC) \]
OUTPUT subroutine to print & clear contents (if any) of output buffer (PBUFF)

Anything in PBUFF?

If
\[ c(PBUFKM) > PBUFF - 2 \]

- Print out contents of PBUFF
- Reset buffer marker
- Clear buffer

Repeat until
\[ c(P) = PBUFF + 238 \]

\[ c(c(P)) + c(ZERO-8) \]
Increment marker

\[ c(PBUFMK) + c(PBFMAN-1) \]

 subroutine PBFMAN subroutine to control increments to print buffer marker, \( c(PBUFMK) \)

Check marker against buffer end

\[ If \quad c(PBUFMK) > PBUFF + 238 \]

If no array flag i.e. not array O/P

Buffer overflow - print warning error (PR El)

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

\[ c(PBUFMK) + c(PBFMAN-1) \]
Define DEC field

Set flag at POPTOP-2

Dec. Stack value pointer

Subtract \( c(SVP) - c(POPTOP-1) \)

Dec. Stack type pointer

Append sign to DEC field

Move flag from POPTOP-1 to POPTOP-3

POPTOP subroutine to decrement (or increment) Stack pointers

Subroutine parameter

\[ x \quad x \quad x \quad x \quad x \]

STP  SVP  \( \hat{\text{POPTOP}} \)
DEC  DEC

Subtract \( c(STP) - c(POPTOP-3) \)
Place \( c(IC) \) & \( c(c(IC)) \) in PBUF (for trace)

Choice on \( c(c(IC)) \)

If Switch 2 on
Print \( c(PBUF) \) (for trace)

If Switch 3 on
Print Stack, Symbol Table, & key variables (for trace)

Increment \( c(IC) \) to address next opcode

Repeat

Line Number Routine

Take Address Routine

Tab Routine
Set array base
- \( c(ARBASE) + 28,000 \)

Set I.L. base
- \( c(ILBASE) + 29,000 \)

Set Stack base
- \( c(STKBAS) + 30,000 \)

Read in Symbol Table

Read in I.L.

Call in and link to MUSINT
APPENDIX D

THE INTERPRETER SPS PROGRAM
**MUSINHT LISTING**

* MUSSEL INTERPRETER VERSION 1 (1972)
* -----------------------------------

**LINE NUMBER ROUTINE**

```
DDRG 2450,,
*LN AM IC,3,10, INC IC
TFM PBUF+54,5355,8, FILL PBUF
TNF PBUF+30,-IC,, RESET LN
TF LN,-IC,, RETURN TO CONTROL
B7 BACK,, *
```

**TAKE ADDRESS ROUTINE**

```
*TA AM IC,3,10, INC IC
TNF PBUF+30,-IC,, FILL PBUF
TFM PBUF+54,6341,8, INC STACK POINTERS
BTM POPTOP,1010,811, CONVERT SYMTAB ADDRESS TO ABS ADDRESS
MM -IC,14,10, AND PLACE AT TOP OF STACK
A 99,SYMBAS,,
TF -SVP,99,,
TDM -STP,6,,
B7 BACK,, RETURN TO CONTROL
```

**TAKE RESULT ROUTINE**

```
*TR AM IC,3,10, INC IC
TNF PBUF+30,-IC,, FILL PBUF
TFM PBUF+54,6359,8, INC STACK POINTERS
BTM POPTOP,1010,811, SET A TO ADDRESS VARIABLE
MM -IC,14,10, 
A 99,,
TF A,9,10,
A A,SYMBAS,,
PUSH,, PUSH RESULT AND TYPE
```

**PUSH ROUTINE**

```
PUSH SM A,9,10, CHECK ADDRESSED VARIABLE TYPE
TD COMP,-A,, ERROR - VARIABLE UNDEFINED
CM COMP,0,10, HALT TO CHECK ERROR
BNE *=+32,, PUSH TYPE
PRA TREL,,
B7 HALT,,
TD -STP,COMP,,
CM COMP,1,10,
BE BACK,,
AM A,9,10,
```

**NCR IC**
- FILL PBUF
- INC STACK POINTERS
- CONVERT SYMTAB ADDRESS TO ABS ADDRESS
- AND PLACE AT TOP OF STACK
- RETURN TO CONTROL

**PUSH ROUTINE**
- CHECK ADDRESSED VARIABLE TYPE
- ERROR - VARIABLE UNDEFINED
- HALT TO CHECK ERROR
- PUSH TYPE
- NULL STRING - RETURN TO CONTROL
**TAKE NUMBER ROUTINE**

* .TN TFM PBUF+54, 6355, 8, 
  FILL PBUF
  AM IC, 7, 10,
  TBF PBUF+38, -IC, 
  AM IC, 2, 10,
  TBF PBUF+42, -IC, 
  BBT POPTOP, 1010, 811, 
  INC STACK POINTERS
  TFL -SVP, -IC, 
  TDM -STP, 2, 
  B7 BACK, 

* .TN TFM PBUF+56, 70, 10, 
  FILL PBUF
  TFM PBUF+54, 6355, 8, 
  INC STACK POINTERS
  BTM POPTOP, 1010, 811, 

*TAKE NUMBER ZERO ROUTINE**

*
*MUSINT LISTING*

TFL -SVP,FZERO,,
TDM -STP,2,,
B7 BACK,,

* 

***************
**TAKE NUMBER ONE ROUTINE**
***************

* 

TN1 TFM PBUF+56,71,10,
TFM PBUF+54,6355,8,
BTM POPTOP,1010,811,
TFL -SVP,FONE,,
TDM -STP,2,,
B7 BACK,,

* 

***************
**TAKE BOOLEAN ROUTINE**
***************

* 

TB AM IC,2,10,
TNF PBUF+28,-IC,,
TFM PBUF+54,6342,8,
BTM POPTOP,1010,811,
TF -SVP,-IC,,
TDM -STP,3,,
B7 BACK,,

* 

***************
**TAKE STRING ROUTINE**
***************

* 

TS TFM PBUF+54,6362,8,
BTM GRAB,0,10,
BTM POPTOP,1010,811,
TDM -STP,7,,
TF -SVP,NODE2,,
TFM LENGTH,0,8,
TFM COUNT,6,10,
LOOP6 SM COUNT,1,10,
AM IC,2,10,
BMR ADD1,-IC,,
CM COUNT,5,10,
BNE **+44,,
SF -NODE1,,
BTM RETURN,0,10,
B7 SETLTH,,
SM IC,2,10,
SF IND,,
B7 OUT6,,
ADD1 AM LENGTH,1,10,
CM COUNT,0,10,
BNE LOOP6,,

OUT6 SM NODE2,4,10,
TF -NODE2,-IC,,
AM NODE2,4,10,
BNE LINK,IND,,
MF -NODE2,IND,,

PUSH VALUE
PUSH TYPE
RETURN TO CONTROL

FILL PBUF
INC STACK POINTERS
PUSH VALUE
PUSH TYPE
RETURN TO CONTROL

INCRC IC
FILL PBUF
INC STACK POINTERS
PUSH STRING DESCRIPTOR TYPE
PUSH DESCRIPTOR ADDRESS FIELD
RETURN TO CONTROL

FILL PBUF
GRAB A NODE
INC STACK POINTERS
PUSH STRING DESCRIPTOR TYPE
PUSH DESCRIPTOR ADDRESS FIELD
RETURN TO CONTROL

SELECT NEXT STRING CHUNK
RECORD MARK TERMINATES STRING
CHECK COUNT
ZERO CHUNK LENGTH - FLAG TERMINAL NODE1
RETURN NODE2 TO AVAIL LIST
BRANCH TO SET DESCRIPTOR LENGTH FIELD
UP LENGTH
PACK CHUNK
SET TERMINAL NODE LINK FIELD
*MUSINT LISTING*

```
AM  IC,2,10,
SETLTH SM  SVP,5,10,
TF  -SVP,LENGTH,,
AM  SVP,5,10,
B7  BACK,,

LINK TF  NODE1,NODE2,,
BTM  GRAB,0,10,
SF  NODE2-3,,
TF  -NODE1,NODE2,,
CF  NODE2-3,,
AM  IC,1,10,
SF  -IC,,
SM  IC,1,10,
B7  LOOP6-12,,

SET STRING DESCRIPTOR LENGTH FIELD
RETURN TO CONTROL

GRAB A NODE
SET LINK FIELD

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
TOM  -STP,1,,
B7  BACK,,

SET NULL STRING TYPE
RETURN TO CONTROL

*******************************************************************************
**SUBROUTINE TO RETURN C(NODE2) TO AVAIL LIST**
*******************************************************************************

DS 2,,
RETURN SF  AVAIL-3,,
TF  -NODE2,AVAIL,,
TF  AVAIL,NODE2,,
BB2  ,,  BRANCH BACK

*******************************************************************************
**SUBROUTINE TO SET C(NODE2) TO POINT TO NEXT AVAILABLE NODE**
*******************************************************************************

DS 2,,
GRAB BNF  POPNOD,AVAIL,,
TF  NODE2,BOUND,,
SM  BOUND,16,10,
CM  BOUND,SVP,,
BL  ++14,,
BB2  ,,  BRANCH BACK
PRA  STRE1,,
B7  HALT,,  HALT TO CHECK ERROR
POPNOD TF  NODE2,AVAIL,,
TF  AVAIL-,AVAIL,,
CF  AVAIL-3,,
BB2  ,,  BRANCH BACK

*******************************************************************************
**INDEX ADDRESS, INDEX RESULT ROUTINES**
*******************************************************************************

*
*MUSINT LISTING*  PAGE 5

.IND  TFM  PBUF+54,4955,8,  FILL PBUF
       TDM  DIM,1,11,  SET DIM TO -1

TEST2  TD  COMP,-SVP,,  
       CM  COMP,2,10,,  
       BNE  ADDR,,,  
       CM  -SVP,0,10,,  
       BE  *+32,,  
       PRA  INDE1,,,  
       B7  HALT,,,  
       SM  SVP,6,10,,  
       CM  -SVP,0,10,,  
       BE  *+32,,  
       PRA  INDE6,,  
       B7  HALT,,,  
       AM  SVP,1,10,,  
       SF  -SVP,,,  
       BTM  POPTOP,1005,8,,  
       AM  DIM,1,10,,  
       B7  TEST2,,  

ADDR  CM  COMP,6,10,,  
       BE  *+32,,  
       PRA  INDE2,,,  
       B7  HALT,,,  
       TF  A,-SVP,,,  
       SM  A,9,10,,  
       TD  COMP,-A,,,  
       CM  COMP,4,10,,  
       BE  *+32,,  
       PRA  INDE3,,  
       B7  HALT,,,  
       AM  A,1,10,,  
       TD  COMP,-A,,,  
       BNF  *+44,COMP,,,  
       TF  BASE,STKBAS,,  
       CF  COMP,,,  
       B7  *+20,,  
       TF  BASE,SYMBAS,,  
       C  COMP,DIM,,  
       BE  *+32,,  
       PRA  INDE4,,  
       B7  HALT,,,  
       AM  A,4,10,,  
       TF  W,BASE,,  
       A  W,-A,,,  
       AM  SVP,8,10,,  
       AM  A,4,10,,  
       TF  A,-A,,,  
       CF  A,-3,,  
       TDM  A-4,0,11,,  
       A  A,BASE,,,  
       TF  L0,-SVP,,,  
       S  L0,-A,,,  
       CM  L0,0,10,,  
       BNL  *+32,,  

OUTBND  PRA  INDE5,,  
       B7  HALT,,,  
       AM  A,4,10,,  

ERROR - INDEX NOT AN INTEGER  
HALT TO CHECK ERROR  
ERROR - INDEX MAGNITUDE GT 9999  
HALT TO CHECK ERROR  
DEC STACK POINTERS  
CHECK TYPE  
ERROR - NOT ADDRESS  
HALT TO CHECK ERROR  
SET A TO ADDR APPROPRIATE TYPE DIGIT  
CHECK TYPE DIGIT  
ERROR - NOT AW  
HALT TO CHECK ERROR  
SET BASE  
CHECK DIMENSION  
ERROR - DIMENSION DISCREPANCY  
HALT TO CHECK ERROR  
SET W TO ADDRESS FIRST ARRAY ELT  
SET A TO ADDRESS SMF  
CALCULATE LO  
CHECK LOWER BOUND  
ERROR - INDEX OUTSIDE BOUNDS  
HALT TO CHECK ERROR
*MUSINT LISTING*

PAGE 6

C -SVP,-A,,
BH OUTBN,B,,
BD 2D,DIM,,
TF A,W,,
A A-1,L0,,
SM SVP,8,10,,

**TEST OP**

BD INDR,-IC,,
TFM PBUF+58,4441,8,
TF -SVP,A,,
TDM -STP,6,,
B7 BACK,,

**INDR**

TFM PBUF+58,4445,8,
B7 PUSH,,

**TEST**

AM A,4,10,,
AM SVP,10,10,,
TF L1,-SVP,,
S L1,-A,,
CM L1,0,10,,
BL OUTBD,,
AM A,4,10,,
C -SVP,-A,,
BH OUTBD,,
AM A,4,10,,
SM SVP,18,10,,
M -A,L0,,
A 99,L1,,
SF 96,,
TF A,W,,
A A-1,99,,
B7 TESTOP,,

**ST, STA ROUTINES**

****

*ST TFM PBUF+54,6263,8,,
TF TOPV,SVP,,
TF TOPT,STP,,
BTM POPTOP,1010,8,
TD COMP,-STP,,
CM COMP,6,10,,
BE *+32,,
PRA STEI,,
B7 HALT,,
TF A,-SVP,,
BT STOR,TOPV,,
CM -IC,13,10,,
BE STA,,
BTM POPTOP,1010,8,,
B7 BACK,,

**STA**

TFM PBUF+56,41,10,
B7 BACK,,

CHECK UPPER BOUND

BYPASS IF 2 DIMENSIONAL ARRAY
SET A TO ADDRESS ARRAY ELT

PLACE ARRAY ELT ADDRESS ON STACK
PLACE AA TYPE DIGIT ON STACK
RETURN TO CONTROL

PUSH RESULT AND TYPE

CALCULATE L1

CHECK LOWER BOUND

CHECK UPPER BOUND

SET A TO ADDRESS ARRAY ELT

*----------*
**ERROR - NOT ABSOLUTE ADDRESS**

HALT TO CHECK ERROR

SET A TO ADDRESS STORE TYPE POSITION
STORE TOP STACK TYPE AND VALUE

CHECK OP CODE

DEC Stack Pointers

Fill PBUF

Dec Stack Pointers

Return to Control

STA OP

Shift down Type

Shift down Value

Return to Control
**SUBROUTINE TO STORE VALUE (+ TYPE) ADDRESSED BY C(STOR-1) IN C(A)**

*DS 5,,  
STOR TF TOPT,STOR-1,,  
SM TOPT, 9,10,  
SM A,9,10,  
TD COMP,-A,,  
TD -A,-TOPT,,  
AM A,9,10,  
CM COMP,7,10,  
BNE NOTSTR,,  
TF NODE2,-A,,  
B7 IN9,,  
LOOP9 TF NODE1,NODE2,,  
TF NODE2,-NODE2,,  
CF NODE2-3,,  
TF -NODE1,AVAL,,  
TF AVAL,NODE1,,  
IN9 SF AVAL-3,,  
BNF LOOP9,-NODE2,,  
TF -NODE2,AVAL,,  
TF AVAL,NODE2,,  
NOTSTR TD COMP,-TOPT,,  
CM COMP,1,10,  
BH +14,,  
BB2 ,,  
CM COMP,2,10,  
BNE +26,,  
TFL -A,-STOR+1,,  
BB2 ,,  
CM COMP,3,10,  
BNE +26,,  
BOOL TF -A,-STOR+1,,  
BB2 ,,  
CM COMP,7,10,  
BE +32,,  
PRA STE2,,  
B7 HALT,,  
TF -A,-STOR+1,,  
SM A,5,10,  
SM STOR-1,5,10,  
TF -A,-STOR+1,,  
AM A,5,10,  
BB2 ,,  

***VALUE IS ROUTINE***

*VI TFM PBUF+54,6549,8,,  
TF A,RAW,,  
SM A,10,10,  
BT STOR,SV,,  

*FILL PBUF  
SET A TO ADDRESS FUNCTION VALUE POSN  
STORE FUNCTION VALUE*
**UNCONDITIONAL JUMP ROUTINE**

```
*UJ AM IC,4,10,  
TFM PBUF+54,6451,8,  
TNF PBUF+32,-IC,,  
BTM RESET,BACK,,  
```

**BRANCH ADDRESS ROUTINE**

```
*BA TFM PBUF+54,4241,8,  
AM IC,4,10,  
TNF PBUF+32,-IC,,  
TD COMP,-STP,,  
CM COMP,-8,10,  
BE POPOUT,,  
POP BTM POPTOP,1010,8,  
TD COMP,-STP,,  
CM COMP,6,10,  
BNE POP,,  
POPOUT BTM POPTOP,1010,8,  
BTM RESET,BACK,,  
```

**IF FALSE JUMP ROUTINE**

```
*IFJ TD COMP,-STP,,  
CM COMP,3,10,,  
BE <+32,,  
PRA IFJE1,,  
B7 HALT,,  
AM IC,4,10,  
TFM PBUF+56,51,10,  
TFM PBUF+54,4946,8,,  
TNF PBUF+32,-IC,,  
CM -SVP,1,10,  
BE <+24,,  
BTM RESET,++12,,  
BTM POPTOP,1010,8,  
B7 BACK,,  
```

**SUBROUTINE TO RESET IC**

```
RESET DS 5,,  
TF LO,-IC,,  
TF IC,ILBASE,,  
A IC,LO,,  
SM IC,1,10,  
```

PAGE 8
**MUSINT LISTING**

B7 -RESET+1,,  BRANCH BACK

***************
**OP ROUTINES**
***************

*OP AM IC,2,10,  INCR IC
TFM PBUF54,5657,BU, FILL PBUF
TNF PBUF28,-IC,,
TFM TABLE2, TABLE2,711,  RESET TABLE2
A TABLE2-1,-IC,,  SET UP TABLE2 BRANCH ADDRESS
B7 -TABLE2,,  BRANCH

**OP BRANCHING TABLE2**************************

*TAB2 DS *,*=-9,,
DS A ANDOR,0,,AND OR,0,,NOT,0,,REL,0,,REL
DS A 0,,REL,0,,REL,0,,REL,0,,REL,0,,CAT
DS A 0,,ADD,0,,SUB,0,,MULT,0,,DIV,0,,INTDV
DS A 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 NOT1,,
B7 HALT,,
BD +32,-SVP,,
TDM -SVP,1,,
B7 BACK,,
TDM -SVP,0,,
B7 BACK,,

***************
**EQ,NE,GT,GE,LT,LE ROUTINES**
***************

*REL TFM TESTSV-1,BACK,,  FILL BRANCH OUT ADDRESS
TF OP,-IC,,  STORE OP
TD COMP,-STP,
*MUSINT LISTING*

**PAGE 10**

- **CM**  COMP,2,10,
- **BNE**  STREL,,
- **BTM**  CHEC,2,10,
- **TDM**  -STP,3,,
- **TFM**  EDIF,0,9,
- **A**  EDIF,-SVP,,
- **B7**  **+14**, DS 6,,
- **TESTSV S**  EDIF,-TOPV,,
- **SM**  TOPV,2,10,,
- **CM**  EDIF,0,10,,
- **BL**  LOW,,
- **BE**  EQUAL,,
- **HIGH**  BT EONE,TESTSV-1,,-
- **CM**  EDIF,7,10, BL SCALE,,
- **T1**  CM OP,7,10, BL FALSE,,
- **TRUE**  TFM -SVP,1,10, B7 -TESTSV+1,,
- **FALSE**  TFM -SVP,0,10, B7 -TESTSV+1,,
- **SCALE**  TFM LOD,MAC,, A LOD,EDIF,,
- **TF**  MAC,-BOTV,,
- **MF**  -LOD,MAC,,
- **C**  -LOD,-TOPV,,
- **BH**  T1,,,
- **BL**  T3,,,
- **B7**  T2,,,
- **EQUAL**  C -BOTV,-TOPV,,
- **BL**  LOW-12,,,
- **BE**  T2,,,
- **TFM**  EDIF,10,10,
- **B7**  HIGH,,,
- **T2**  CM OP,8,10, BE TRUE,,
- **CM**  OP,7,10, BNL FALSE,..
- **CM**  OP,4,10, BH TRUE,,
- **B7**  FALSE,,
- **TFM**  EDIF,-10,10,
- **LOW**  BT EONE,TESTSV-1,,
- **CM**  EDIF,-7,10,,
- **BH**  SCOMP,,
- **T3**  CM OP,6,10, BL TRUE,,
- **BE**  FALSE,,
- **CM**  OP,7,10,
- **BE**  TRUE,,
- **B7**  FALSE,,
- **SCOMP**  TFM LOD,MAC,,
- **S**  LOD,EDIF,,
- **TF**  MAC,-TOPV,,
- **MF**  -LOD,MAC,,
- **C**  -BOTV,-LOD,,

**CHECK TOP STACK TYPE**

**NOT NUMBER - TO STR RELATIONAL ROUTINE**

**OTHERWISE CHECK TOP 2 STACK TYPES**

**FILL BOOLEAN TYPE**

**CALCULATE EXPONENT DIFFERENCE**

**SPACE FOR BRANCH OUT ADDRESS**

**TOPV NOW POINTS TO TOP MANTISSA**

**COMPARE EXP DIFFERENCE WITH ZERO**

**LESS**

**EQUAL**

**GREATER - CHECK FOR EQ OR NE**

**CHECK OP**

**FALSE IF LT,LE,OR EQ - OTHERWISE TRUE**

**SET TOP STACK VALUE TO TRUE**

**BRANCH OUT**

**SET TOP STACK VALUE TO FALSE**

**BRANCH OUT**

**SCALE AND COMPARE MANTISSAS**

**BOT GREATER - OP TEST 1**

**TOP GREATER - OP TEST 3**

**EQUAL - OP TEST 2**

**COMPARE MANTISSAS**

**BOT LESS**

**EQUAL - OP TEST 2**

**BOT GREATER**

**CHECK OP**

**TRUE IF GE**

**FALSE IF NE OR GT**

**TRUE IF DE OR EQ**

**FALSE IF LT**

**CHECK FOR EQ OR NE**

**CHECK OP**

**TRUE IF LT OR LE**

**FALSE IF EQ**

**TRUE IF NE**

**FALSE IF GT OR GE**

**SCALE AND COMPARE MANTISSAS**
**STRING RELATIONAL OPS**

* STREL
  TF TOPV, SVP,,
  BTM POPTOP, 1010, 8,
  CM COMP, 1, 10,
  BNE DESCR,,
  TD COMP, -STP,,
  TDM -STP, 3,,
  CM COMP, 1, 10,
  BE T2,,,
  CM COMP, 7, 10,
  BE T1,,
  PRA STRLE1,,,
  B7 HALT,,

* DESCR
  CM COMP, 7, 10,
  BNE DESCR-20,,
  TD COMP, -STP,,
  TDM -STP, 3,,
  CM COMP, 1, 10,
  BE T3,,
  CM COMP, 7, 10,
  BNE DESCR-20,,
  TF NODE1, -SVP,,
  TF NODE2, -TOPV,,
  SM TOPV, 5, 10,
  TF BOTV, SVP,,
  SM BOTV, 5, 10,
  C -BOTV, -TOPV,,
  BE **+24,,
  BMT EQNE, BACK,,
  BT TAIL, BOTV,,
  TF BOTREM, TOPREM,,
  BT TAIL, TOPV,,
  B7 TEST95,,,

* LOOP95
  SM NODE1, 4, 10,
  SM NODE2, 4, 10,
  C -NODE1, NODE2,,
  BH T1,,
  BL T3,,
  AM NODE1, 4, 10,
  AM NODE2, 4, 10,
  TF NODE1, -NODE1,,
  TF NODE2, -NODE2,,
  CF NODE1-3,,
  CF NODE2-3,,

* TEST95
  BNF **+20, -NODE2,,
  B7 OUT95,,,
  BNF LOOP95, -NODE1,,
  BNF T12-36, -NODE2,,

* T13
  SM NODE1, 4, 10,
  SM NODE2, 4, 10,
  C BOTREM, TOPREM,,
  BE T12,,
**SUBROUTINE TO FIND STRING LENGTH REMAINDER**

```
  BH  *+44,,
S    NODE2, TOPREM,,
A    NODE2, BOTREM,,
B7   T12,,
S    NODE1, BOTREM,,
A    NODE1, TOPREM,,
B7   T12,,
SM   NODE1, 4, 10,
SM   NODE2, 16, 10,
A    NODE2, BOTREM,,
T12  C  -NODE1, -NODE2,,
BL   T3,,
BH   T1,,
C    -BOTV, -TOPV,,
BE   T2,,
B7   *-48,,
OUT95 BNF  *+20, -NODE1,,
B7   T13,,
SM   NODE2, 4, 10,
SM   NODE1, 16, 10,
A    NODE1, TOPREM,,
B7   T12,,
```

TOPREM GREATER - ADJUST NODE2
AND THEN COMPARE NODE VALUES
BOTREM GREATER - ADJUST NODE1
AND THEN COMPARE NODE VALUES
NO NODE2 FLAG - DEC NODE1 POINTER
ADJUST NODE2 POINTER
COMPARE NODE VALUES
BOT STRING LESS - OP TEST 3
BOT STRING GREATER - OP TEST 1
EQUAL - COMPARE STRING LENGTHS
EQUAL - OP TEST 2
OTHERWISE CHECK LESS OR GREATER
NODE1 FLAGGED - BRANCH TO TEST 13
OTHERWISE DEC NODE2 POINTER
ADJUST NODE1 POINTER
AND THEN BRANCH TO COMPARE NODE VALUES

**SUBROUTINE TO CHECK FOR EQ OR NE OP**

```
  DS  5,,
TLD  99, -TAIL+1,,
DM   96, 6, 10,
TF   TOPREM, 99,,
A    TOPREM, 99,,
CM   TOPREM, 0, 10,
BNE  *+24,,
TFM  TOPREM, 12, 10,
BB2  ,,;
```

SPACE FOR STRING LENGTH FIELD ADDRESS
DIVIDE STRING LENGTH BY 6
MULTIPLY REMAINDER BY 2
AND STORE IN TOPREM
SET TO 12 IF ZERO
BRANCH BACK

**CAT ROUTINE**

```
  DS  5,,
CM   OP, 6, 10,
BNE  *+32,,
TFM  -SVP, 0, 10,
B7   -EQNE+1,,
CM   OP, 7, 10,
BE   *+14,,
BB2  ,,;
TFM  -SVP, 1, 10,
B7   -EQNE+1,,
```

SPACE FOR BD ADDRESS IF EQ OR NE
CHECK OP
FALSE IF EQ
BRANCH OUT
TRUE IF NE
BRANCH OUT

*CAT

```
  TFM  PBUF+56, 63, 10,
TFM  PBUF+54, 4341, 8,
TD   COMP, -STP,,
TF   TOPV, SVP,,
BTM  POPTOP, 1010, 8,
```

FILL PBUF
STORE TOP STACK TYPE DIGIT
SAVE TOP VALUE POINTER
DEC STACK POINTERS
CM COMP,1,10, CHECK TOP STACK TYPE
BE BACK,,, NULL STRING - RETURN TO CONTROL
CM COMP,7,10, ERROR - NOT STRING TYPE
BE +32,,, HALT TO CHECK ERROR
STRERR PRA CATE1,,, SET POINTERS
B7 HALT,,, TF BOTV,SVP,,, 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,,,, SET NODE1
TF NODE1,-BOTSTR,, B7 TEST7,,, CHAIN DOWN TO TERMINAL NODE
CHAIN TF NODE1,-NODE1,, CF NODE1-3,,, TEST7 BNF CHAIN,-NODE1,,
TF NODE2,-TOPSTR,, SET NODE2
SF NODE2-3,,, TF -NODE1,NODE2,, LINK 2 STRINGS
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,, BE BACK,,, CHECK DIVISION REMAINDER AGAINST 0
TF BOTREM,99,,, EQUAL - RETURN TO CONTROL
MM BOTREM,2,10,
SF 98,,, TF BOTREM,99,, STORE REMAINDER (DIGITS) IN BOTREM
LD 99,-TOPV,, DM 96,6,10,
SF 98,,, TF TOPREM,99,, DIVIDE TOP STRING LENGTH BY 6
MM TOPREM,2,10,
SF 98,,, TF TOPREM,99,, STORE REMAINDER (DIGITS) IN TOPREM
B7 TEST8,,, 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,, TIDY UP END NODES
TF DIF,BOTREM,, A DIF, TOPREM,,
SM DIF,12,10,
*MUSINT LISTING*

CM DIF,0,10,
BH HI,,
CF DIF,,
SF -NODE1,,
BTM RETURN*0,10,
SM NODE1,4,10,
SM NODE2,4,10,
TF TEMP,-NODE1,,
TF -NODE1,-NODE2,,
S NODE1,TEMP,,
TF -NODE1,TEMP,,
AM NODE1,1,10,
CF -NODE1,,
B7 BACK,,

HI SF -NODE2,,
BT REPACK,DIF,,
B7 BACK,,

*
********************
**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 ,,,

*
***************
**ADD ROUTINE**
***************
*
ADD BTM CHEC,2,10,
CM -TOPV,0,10,
BNE FLOAT,,, CM -SVP,0,10,,
BNE FLOAT,,, SM TOPV,2,10,
BV */+12,,
A -BOTV,-TOPV,,, BV */+32,,
BNF BACK,IND-1,,
B7 ISVSUB,,, TF MAC,-BOTV,,
CF MAC-6,,
TDM MAC-7,1,11,
TFM MOD,MAC-7,,
TFM LOD,MAC-1,,
TFM EAC,1,9,,

* NODE1 TERMINATES STRING
RETURN NODE2 TO AVAIL LIST
REPACK NODE1

RETURN TO CONTROL
REPACK NODE1 - NODE2 TERMINATES STRING
RETURN TO CONTROL

BRANCH BACK

INTEGER OPERANDS
INTEGER RESULT - RETURN TO CONTROL
RETURN TO ISV SUBROUTINE
OVERFLOW - FLOATING RESULT
SET POINTERS B4 BRANCHING TO RNS ROUTINE
FLOATING ADD ROUTINE

FLOAT C
  SVP, -TOPV,,
  BNE +80,,
  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,,
  CF EDIF-1,,
  TDM EDIF-2, 0, 11,
  S EDIF, EQ,,
  CM EDIF, 8, 10,,
  BL +56,,
  TF BOTV, MAC,,
  TF -SVP, EP,,
  BNF BACK, IND-1,,
  B7 ISVSUB,,
  TF EAC, EQ,,
  CF EAC-1,,
  TDM EAC-2, 0, 11,
  TFM LOD, MAC,,
  A LOD, EDIF,,
  MF -LOD, MAC,,
  BV +12,,
  A LOD, MQ,,
  MF IND, -LOD,,
  BNV +92,,
  CF MAC-6,,
  TDM MAC-7, 1, 11,
  TFM HOD, MAC-7,,
  TFM LOD, MAC-1,,
  A EAC, EDIF,,
  AM EAC, 1, 10,
  B7 RNS,,
  CM -LOD, 0, 10,
  BE STZ,,
  A EAC, EDIF,,
  SETPTS TFM LOD, MAC,,
  TFM HOD, MAC-6,,
  B7 RNS,,
  STZ TFL -SVP, FZERO,,
  CF IND,,
  BNF BACK, IND-1,,
  B7 ISVSUB,,

*
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,
BV *+12,,, ROUND
BV *+32,,, NO OVERFLOW
SM LOD,1,10,,
B7 NRM,,, OVERFLOW
CF -HOD,,,
SM HOD,1,10,
TDM -HOD,1,11,
SM LOD,2,10,
AM EAC,1,10,

NORMALISE 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 NRM,,,

EXPON BD *+20, EAC-2,,, CHECK FOR EXPONENT OVERFLOW OR UNDERFLOW
B7 STORE,,,
BNF OFLOW,EAC,,, UNDERFLOW - PRINT WARNING MESSAGE
PRA UFLO,,, FILL WITH ZEROS
TFL -SVP,FZERO,,, B7 STORE+$48,,, OVERFLOW - PRINT WARNING MESSAGE
OFLOW PRA OFLO,,, FILL WITH NINES
TFL -LOD,ALL9S,,, STORE RESULT
TFL EAC,99,10,, STORE SIGN
STORE TF -BOTV,-LOD,,
MF -BOTV,IND,,
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,
TF EAC -TOPV,
CF EAC -1,,
TDM EAC -2,0,11,
A EAC -SVP,,
SM TOPV,2,10,
M -BOTV,-TOPV,,
MF IND 99,,
CM 99,0,10,
BE STZ,,
CM EAC 0,10,
BNE +80,,
CM 92,0,10,
BNE +56,,
SF 93,,
TF -BOTV 99,,
TFM -SVP,0,10,
B7 BACK,,
TF MAC+7,99,,
AM EAC 7,10,
B7 SETPTS,,

**DIVIDE ROUTINE**

*.

DIV BTM CHEC 2,10,,
TF EAC -SVP,,
CF EAC -1,,
TDM EAC -2,0,11,
S EAC -TOPV,,
SM TOPV,2,10,
CM -BOTV,0,10,
BE STZ,,
CM -TOPV,0,10,
BNE +32,,
ZDIV PRA DIV1,,
B7 HALT,,
LD 91,-BOTV,,
TFM 79,0,9,,
D 85,-TOPV,,
MF IND 92,,
TF MAC+8,92,,
B7 SETPTS,,

***INTEGER DIVIDE ROUTINE***

*.

INTDV BTM CHEC 2,10,,
CM -TOPV,0,10,
BNE +36,,
CM -SVP,0,10,
BE +32,,

Check top 2 stack types
Add exponents
Multiply mantissas
Store sign of product
Zero result
Store integer result
Return to control
Floating result - load accumulator
Set pointers

Check top 2 stack types
Subtract exponents
Zero dividend - zero result
Error - zero divisor
Halt to check error
Load dividend
Divide
Store sign of quotient
Transfer result to accumulator
Set pointers

Check top 2 stack types
Check integer operands
*MUSINT LISTING*

```
PRA IDIVE1,,, ERROR – NON INTEGER OPERAND
B7 HALT,,,, HALT TO CHECK ERROR
SM TOPV,2,10,
CM -BOTV,0,10,
BE STZ,,,,
CM -TOPV,0,10,
BE ZDIV,,
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,,
PR A 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,, CHECK TOP STACK TYPE
C COMP,CHEC-1,,
BNE *+60,,
SM STP,10,10,
TD COMP,-STP,, CHECK NEXT STACK TYPE
C COMP,CHEC-1,,
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**

**REPEAT-TIMES ROUTINE**

**REPEAT-FOR ROUTINE**
*MUSINT LISTING*

BNF NF2,-STP, ,
BTM ISV,30,10, 
TFM OP,5,10, 
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, ,
BTM RESET,BACK, ,
NF2 AM IC,3,10, 
BTM CHKSET,3,10, 
BTM POPTOP,3020,811, 
SF -STP, ,
B7 BACK, ,

*FLAG INDICATES LOOP ALREADY TRAVERSED*
*INCREMENT LOOP VARIABLE*
*TEST LOOP VARIABLE*

**REPEAT-FOR-WITHOUT-TEST ROUTINE**

*RFWT TFM PBUF+54,5946,8, FILL PBUF*
TFM PBUF+58,6663,8,  
AM IC,3,10, 
TFM PBUF+30,-IC, ,  
BNF NF3,-STP, ,
BTM ISV,20,10, 
TF SVP,TOPV, , 
SM SVP,2,10, 
B7 BACK, ,

*FLAG INDICATES LOOP ALREADY TRAVERSED*
*INCREMENT LOOP VARIABLE*
*RESET SVP*

**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,,  ERROR - NOT NUMBER
PRA RFE1,, 
B7 HALT,,  DEC STACK POINTERS
SM SVP,10,10, 
B7 TEST4,, LOOP BACK
AM SVP,10,10, SET SVP TO ADDRESS INITIAL VALUE

NO FLAG INDICATES 1ST TRAVERSAL OF LOOP
CHK LOOP CONTROL TYPES + INIT LOOP VAR
RESET STACK POINTERS
SET FLAG AT TOP STACK TYPE POSITION
RETURN TO CONTROL

NO END OF LOOP - RESET LINE NUMBER
ELSE RESET IC TO BA ADDR AND RET TO CON
NO FLAG INDICATES 1ST TRAVERSAL OF LOOP
CHK LOOP CONTROL TYPES + INIT LOOP VAR
RESET STACK POINTERS
SET FLAG AT TOP STACK TYPE POSITION
RETURN TO CONTROL

*********************************************************************
**SUBROUTINE TO CHECK LOOP CONTROL TYPES + INITIALISE LOOP VARIABLE**
*********************************************************************

SPACE FOR COUNT PARAMETER
SET COUNT
TEST COUNT
EXIT LOOP IF ZERO
DEC COUNT
CHECK TOP STACK TYPE
ERROR - NOT NUMBER
DEC STACK POINTERS
LOOP BACK
SET SVP TO ADDRESS INITIAL VALUE
*MUSINT LISTING*

TFL -A,-SVP,, INITIALISE VALUE
SM A,9,10, INITIALISE TYPE
TDM -A,2,, BRANCH BACK
BB2 ,, *

*******************************************************************************
**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
ISV SUB CF IND-1,, CLEAR FLAG AT IND-1
BB2 ,, BRANCH BACK

*******************************************************************************
**REPEAT-SET ROUTINE**
*******************************************************************************

RS AM IC,2,10, FILL PBUF
TFM PBUF+54,5962,B, SET A TO ADDRESS LOOP VARIABLE
TNF PBUF+2B,-IC,,
S SVP-1,-IC,,
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,
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,
TF LN,-IC,, FILL PBUF AND INCR IC
TF SVP,STP,, RESET LINE NUMBER
AM SVP,9,10,
B7 BACK,,, RESET SVP
B7 NF4-32,,, RETURN TO CONTROL

*******************************************************************************
**REPEAT-SET SUBROUTINE TO FILL PBUF AND INCR IC**
*******************************************************************************

*
```
* MUSINT LISTING *

<table>
<thead>
<tr>
<th>DS</th>
<th>2,..</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL</td>
<td>AM IC,4,10,</td>
</tr>
<tr>
<td></td>
<td>TNP PBUF+36,-IC,</td>
</tr>
<tr>
<td></td>
<td>AM IC,3,10,</td>
</tr>
<tr>
<td></td>
<td>TNP PBUF+42,-IC,</td>
</tr>
<tr>
<td></td>
<td>BB2 ,..</td>
</tr>
<tr>
<td></td>
<td>BRANCH BACK</td>
</tr>
</tbody>
</table>
|      | *
|      | *************** |
|      | **CASE ROUTINE** |
|      | *************** |
|      | *
| CS  | TFM PBUF+54,4362,8, |
|     | AM IC,2,10, |
|     | TNP PBUF+28,-IC, |
|     | TFM COUNT,3,10, |
| TEST5 | CM COUNT,0,10, |
|     | BE OUTS, ,.. |
|     | TD COMP,-STP,, |
|     | CM COMP,2,10, |
|     | BE *+32,,, |
|     | PRA CSE1,,, |
|     | B7 HALT,,, |
|     | CM -SVP,0,10, |
|     | BE *+32,,, |
|     | PRA CSE2,,, |
|     | B7 HALT,,, |
|     | SM COUNT,1,10, |
|     | BTM POP TOP,10,10,8, |
|     | B7 TEST5,,.. |
| OUTS | AM SVP,28,10, |
|      | TF TOPV,SVP,, |
|      | SM SVP,10,10, |
|      | TF BOTV,SVP,, |
|      | SM SVP,10,10, |
|      | C -SVP,-TOPV,, |
|      | BNH *+32,,, |
|      | PRA CSE3,,, |
|      | B7 HALT,,, |
|      | BV *+12,,, |
|      | S -SVP,-BOTV,, |
|      | BNV *+32,,, |
|      | PRA CSE4,,, |
|      | B7 HALT,,, |
|      | CM -SVP,0,10, |
|      | BNL *+32,,, |
|      | PRA CSE5,,, |
|      | B7 HALT,,, |
|      | C -SVP,-IC,, |
|      | BL *+32,,, |
|      | PRA CSE6,,, |
|      | B7 HALT,,, |
| RSETIC | MM -SVP,6,10, |
|       | SF 96,,, |
|       | A IC,99, |
|       | SM SVP,8,10, |
|       | B7 BACK,,, |
|       | RESET IC |
|       | 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,
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,
BNL *+32,,
PRA CSWBE3,,, ERROR - CASE CONTROL VALUE LESS THAN ONE
B7 HALT,,,
C -SVP, -IC,,
BNH *+32,,
PRA CSWBE4,,, ERROR - CONTROL VALUE GT NO. OF CS STMTS
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, 1002, 8,, 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
FILRAW TDM -STP, 0,, SET TO UNDEFINED
TF -SVP, RAW,,
AM SVP, 4, 10,
TF RAW, SVP,,
B7 BACK,,, FILL RAW AND SET RAW POINTER

CALL PROCEDURE ROUTINE*

*****

*CP* AM IC, 3, 10,
FILL PBUF

CONVERT SYMTAB ADDR TO ABSOLUTE ADDR

ERROR - NOT PAW

HALT TO CHECK ERROR

STORE NO. OF ACTUAL PARAMETERS
STORE RETURN I.L. ADDRESS IN RAW

SET I.C. TO START OF PROCEDURE

RETURN TO CONTROL

FILL PBUF

COMPARE NO. OF A.P.S. WITH NO. OF F.P.S

ERROR - NO. A.P.S NE NO. F.P.S

HALT TO CHECK ERROR

SET BASE VALUE POINTER
AND BASE TYPE POINTER

LEAVE SPACE FOR LOCAL VARIABLES
AND STORAGE MAPPING FUNCTION

SET WORKING VALUE POINTER
SET LOCAL VARIABLE TYPES TO UNDEFINED

IND FLAG INDICATES TLA OP

FILL PBUF

SET LOCAL VARIABLE POINTERS
TF LTP,LVP,,
SM LTP9,10,
BTM POPTOP1010,811,
TD COMP-LTP,,
BNF TLR,IND,,
CF IND,,
TFM PBUF+56,41,10,
CM COMP,5,10,
BL *+44,,
TF -SVP,-LVP,,,
TD -STP,-LTP,,
B7 BACK,,
B7 BACK,,
TLR TFM PBUF+56,59,10,
TEST3 CM COMP,6,10,
BNE OUT3,,
TF LTP,-LVP,,
TF LVP,LTP,,
SM LTP9,10,
TD COMP,-LTP,,
B7 TEST3,,
OUT3 TF A,LVP,,
B7 PUSH,,

*MSF AM IC,4,10,,
TFM PBUF+56,46,10,
TFM PBUF+54,5462,8,
TNF PBUF+32,-IC,,
TF MFP,WVP,,
BD *+32,-IC,,
SM MFP12,10,
B7 *+32,,
SM MFP4,10,,
SF IND,,
BTM LOAD,0,10,
TF SIZE,-MFP,,
CF SIZE-3,,
BTM DEC0,10,
BTM LOAD,0,10,
S SIZE,-MFP,,
AM SIZE1,10,
CM SIZE-3,0,10,
BE *+32,,
2BIG PRA MSFE1,,,
B7 HALT,,
SF SIZE-2,,
BNF AROUND,IND,,
BTM DEC0,10,
BTM LOAD0,10,
TF ACC,-MFP,,
BTM DEC0,10,

INC STACK POINTERS
TLA OP
CHECK TYPE OF ADDRESSED VARIABLE
ADDRESS TYPE - PUSH THIS ADDRESS VALUE
RESULT TYPE - PUSH LOCAL VARIABLE ADDR
RETURN TO CONTROL
RETURN TO CONTROL
TLR OP
CHECK TYPE OF ADDRESSED VARIABLE
AA TYPE - CHAIN DOWN
FILL PBUF
SET MFP DEPENDING ON ARRAY DIMENSION
IND FLAG INDICATES 2 DIMENSIONAL ARRAY
CHECK AND LOAD INDEX VALUE
DECREMENT POINTERS
CHECK AND LOAD INDEX VALUE
DECREMENT POINTERS
ERROR - ARRAY SIZE GT 999
HALT TO CHECK ERROR
BYPASS IF 1 DIMENSIONAL ARRAY
DECREMENT POINTERS
CHECK AND LOAD INDEX VALUE
DECREMENT POINTERS
*MUSINT LISTING*

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

```
DS 2,,
LOAD TD COMP, -STP,,
CM  COMP, 2, 10,,
BE  *+32,,
PRA MS=E2,,, ERROR - INDEX NOT NUMBER
B7  HALT,,, HALT TO CHECK ERROR
CM  -SVP, 0, 10,,
BE  *+32,,
PRA MS=E3,,, ERROR - INDEX NOT INTEGER
B7  HALT,,, HALT TO CHECK ERROR
SM  SVP, 5, 10,,
CM  -SVP, 0, 10,,
BE  *+32,,
PRA MS=E4,,, ERROR - INDEX MAGNITUDE GT 9999
B7  HALT,,, HALT TO CHECK ERROR
AM  SVP, 1, 10,,
SF  -SVP,,, LOAD INDEX VALUE
AM  SVP, 3, 10,,
TF  -M=FP, -SVP,,, BRANCH BACK
BB2 ,, *
```

**RETURN ROUTINE**

```
RE    TFM PBUF+54, 5945, B,, FILL PBUF
TF    SVP, BVP,, RESET STACK POINTERS
TF    STP, BTP,,
BTM  POPTOP, 1010, B,
TF    IC, ILCASE,, RESET IC
A     IC, -RAW,,
SM    RAW, 4, 10,,
TF    RAW, -RAW,,
TF    BVP, RAW,,
TF    BTP, RAW,,
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  CORFLOW,,, 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,
B7 BACK,,

SET STP TO ADDR TOP STACK TYPE POSITION
RETURN TO CONTROL

***************
**NEWPAGE ROUTINE**
***************

* NP TFM PBUF+54, 5557, 8, FILL PBUF
BTM OUTPUT, **+12,,
SKIP,1,,
B7 BACK,,

PRINT AND CLEAR PBUF
SKIP TO NEWPAGE
RETURN TO CONTROL

***************
**NEWLINE ROUTINE**
***************

* NL TFM PBUF+54, 5553, 8, FILL PBUF
AM IC, 2, 10,
TNF PBUF+28, -IC,,
BTM OUTPUT, **+12,,
TF COUNT, -IC,,
SKIP SPIM,,
SM COUNT, 1, 10,
CM COUNT, 0, 10,
BE BACK,,
B7 SKIPL,,

SET SKIP COUNT
SKIP A LINE
DEC COUNT
CHECK
RETURN TO CONTROL IF ZERO
OTHERWISE LOOP AGAIN

***************
**SPACE ROUTINE**
***************

* SP TFM PBUF+54, 6257, 8, FILL PBUF
AM IC, 2, 10,
TNF PBUF+28, -IC,,
SF SPFLG,,
MM -IC, 2, 10,
BT PBFMAN+99,,
B7 BACK,,

SET SPACE FLAG
INC PBUFF MARKER
RETURN TO CONTROL

***************
**TAB ROUTINE**
***************

* TAB AM IC, 3, 10,
TFM PBUF+54, 6342, 8, FILL PBUF
TNF PBUF+30, -IC,,
CM -IC, 120, 9,
BL **+32,,
PRA TBE1,,
B7 HALT,,
TFM PBUFMK, PBUFF-2,,
A PBUFMK, -IC,,
A PBUFMK, -IC,,
B7 BACK,,

INC IC
CHECK TAB PARAMETER
OK IF LESS THAN 120
OTHERWISE ERROR
HALT TO CHECK ERROR
RESET PBUFF MARKER
RETURN TO CONTROL
**SUBROUTINE TO PRINT AND CLEAR PBUF**

***************

* DS 5,,
OUTPUT CM PBUFMK,PBUFF-2,,
BNH -OUTPUT+1,,
PRAS PBUFF,,
TFM PBUFMK,PBUFF-2,,
TFM P1,PBUFF-2,,
LOOP30 AM P1,10,10,
   TF -P1,ZERO-8,,
   CM P1,PBUFF+238,,
   BE -OUTPUT+1,,
   B7 LOOP30,,
*

*******

PRINT BUFFER MANAGER SUBROUTINE

**************

* DS 4,,
PBFMAN A PBUFMK,PBFMAN-1,,
CM PBUFMK,PBUFF+238,,
BNH #+58,,
BNF WARN ARRAY,,
TFM OUTPUT-1,*+20,,
B7 OUTPUT,,
SPIM 1,,
BNF PBFMAN,SPFLG,,
CF SPFLG,,
BB2,,
WARN PRA PRE1,,
B7 PBFMAN+48,,
*

*******

INPUT-OUTPUT LOAD ROUTINES

**************

* RDSW DC 3,0,,
PRSW DS ,RDSW-1,,
PRCSW DS ,RDSW-2,,
*RD BD 20500,RDSW,,
   TFM RDSW,1,9,,
   CALL LINK,MUSRD,20500
*PR BD 20500,PRSW,,
   TFM RDSW,10,9,,
   CALL LINK,MUSPR,20500
*PRC BD 20500,PRCSW,,
   TFM RDSW,100,9,,
   CALL LINK,MUSPRC,20500
*

*******

END ROUTINE

**********

* DORG 17500,,
   END TFM PBUF+56,44,10,
   TFM PBUF+54,4555,8,
**MAIN CONTROL ROUTINE**

* DORG 17800,,

BACK BNC2 ++24,,

PRINT PBUF IF SWITCH 2 ON

BNC3 START-12,,

BYPASS IF SWITCH 3 OFF

PRN 30001,,

PRINT STACK

PRN 30141,,

PRINT SYMBOL TABLE

PRN PRNT,,

PRINT VARIABLES

PRN PRNT+144,,

INCR IC

START TNF PBUF+10,IC,,

SET UP TABLE1 BRANCH ADDRESS

TNF PBUF+22,-IC,,

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,,

PRINT BUFFER MARKER

PBUFMK DS 5,,

WORKING POINTER

P DS 5,,

WORKING POINTER

P1 DS 5,,

WORKING POINTER

SPFLG DC 2,0,,

SPACE FLAG

FPAT DC 17,-11111110111111,, 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

REPLFLG 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
**MUSINT LISTING**

<table>
<thead>
<tr>
<th>DS</th>
<th>VALUE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC</td>
<td>3</td>
<td>EXPONENT COUNT</td>
</tr>
<tr>
<td>.SIGFLG DC</td>
<td>2,0</td>
<td>SIGNIFICANCE FLAG</td>
</tr>
<tr>
<td>EFLG DC</td>
<td>2,0</td>
<td>EXPONENT FLAG</td>
</tr>
<tr>
<td>NFLG DC</td>
<td>2,0</td>
<td>NEGATIVE FLAG</td>
</tr>
<tr>
<td>PFLG DC</td>
<td>2,0</td>
<td>PLUS FLAG</td>
</tr>
<tr>
<td>QFLG DC</td>
<td>2,0</td>
<td>ZERO FLAG</td>
</tr>
<tr>
<td>XL</td>
<td>5</td>
<td>EXPONENT LENGTH</td>
</tr>
<tr>
<td>SBUFMK DS</td>
<td>5</td>
<td>STRING BUFFER MARKER</td>
</tr>
<tr>
<td>INFLAG DC</td>
<td>2,0</td>
<td>INPUT FLAG</td>
</tr>
<tr>
<td>ARRAY DC</td>
<td>2,0</td>
<td>ARRAY INPUT FLAG</td>
</tr>
<tr>
<td>RBUF DS</td>
<td>80</td>
<td>READ BUFFER</td>
</tr>
<tr>
<td>RBUFMK DS</td>
<td>5</td>
<td>READ BUFFER MARKER</td>
</tr>
<tr>
<td>MCOUNT</td>
<td>2</td>
<td>MANTISSA COUNT</td>
</tr>
<tr>
<td>NUMBER DS</td>
<td>5</td>
<td>NUMBER ACCUMULATOR</td>
</tr>
<tr>
<td>NACC DC</td>
<td>10,0</td>
<td>ARRAY COUNT</td>
</tr>
<tr>
<td>ACOUNT DC</td>
<td>4</td>
<td>EXPONENT BUILD POINTER</td>
</tr>
<tr>
<td>EXP DS</td>
<td>5</td>
<td>EXPONENT BUILD AREA</td>
</tr>
<tr>
<td>XAREA DC</td>
<td>3,0</td>
<td>STRING BUILD AREA</td>
</tr>
<tr>
<td>SAREA DC</td>
<td>14,0</td>
<td>STRING BUILD POINTER</td>
</tr>
<tr>
<td>STRING DS</td>
<td>5</td>
<td>STRING LENGTH</td>
</tr>
<tr>
<td>IC</td>
<td>5</td>
<td>INSTRUCTION COUNTER</td>
</tr>
<tr>
<td>SVP DS</td>
<td>5</td>
<td>STACK VALUE POINTER</td>
</tr>
<tr>
<td>STP DS</td>
<td>5</td>
<td>STACK TYPE POINTER</td>
</tr>
<tr>
<td>TOPV DS</td>
<td>5</td>
<td>POINTER TO TOP STACK VALUE</td>
</tr>
<tr>
<td>TOPT DS</td>
<td>5</td>
<td>POINTER TO TOP STACK TYPE</td>
</tr>
<tr>
<td>LN DC</td>
<td>5,0</td>
<td>LINE NUMBER</td>
</tr>
<tr>
<td>BOTV DS</td>
<td>5</td>
<td>BOTTOM VALUE</td>
</tr>
<tr>
<td>A DS</td>
<td>5</td>
<td>ABSOLUTE ADDRESS</td>
</tr>
<tr>
<td>RAW DC</td>
<td>5,0</td>
<td>RETURN ADDRESS WORD</td>
</tr>
<tr>
<td>AP DS</td>
<td>2</td>
<td>ACTUAL PARAMETERS</td>
</tr>
<tr>
<td>BVP DS</td>
<td>5</td>
<td>BASE VALUE POINTER</td>
</tr>
<tr>
<td>BTP DS</td>
<td>5</td>
<td>BASE TYPE POINTER</td>
</tr>
<tr>
<td>LVP DS</td>
<td>5</td>
<td>LOCAL VALUE POINTER</td>
</tr>
<tr>
<td>LTP DS</td>
<td>5</td>
<td>LOCAL TYPE POINTER</td>
</tr>
<tr>
<td>WVP DS</td>
<td>5</td>
<td>WORKING VALUE POINTER</td>
</tr>
<tr>
<td>MFP DS</td>
<td>5</td>
<td>MAPPING FUNCTION POINTER</td>
</tr>
<tr>
<td>LENGTH DS</td>
<td>5</td>
<td>STRING LENGTH</td>
</tr>
<tr>
<td>NODE1 DC</td>
<td>5,30000</td>
<td>NODE POINTERS</td>
</tr>
<tr>
<td>NODE2 DC</td>
<td>5,30000</td>
<td>- LINK(C(NODE1)) = C(NODE2)</td>
</tr>
<tr>
<td>REF DS</td>
<td>5</td>
<td>REFERENCE NODE</td>
</tr>
<tr>
<td>AVAIL DC</td>
<td>5,-0</td>
<td>AVAILABLE LIST POINTER</td>
</tr>
<tr>
<td>BOUND DC</td>
<td>5,39999</td>
<td>LOWER STRING AREA BOUNDARY</td>
</tr>
<tr>
<td>TOPSTR DS</td>
<td>5</td>
<td>TOP STRING</td>
</tr>
<tr>
<td>BOTSTR DS</td>
<td>5</td>
<td>BOTTOM STRING</td>
</tr>
<tr>
<td>TOPREM DS</td>
<td>2</td>
<td>TOP STRING LENGTH REMAINDER</td>
</tr>
<tr>
<td>BOTREM DS</td>
<td>2</td>
<td>BOTTOM STRING LENGTH REMAINDER</td>
</tr>
<tr>
<td>DIF DS</td>
<td>2</td>
<td>TEMPORARY STORE</td>
</tr>
<tr>
<td>TEMP DS</td>
<td>12</td>
<td>TEMPORARY STORE</td>
</tr>
<tr>
<td>COUNT DC</td>
<td>2,0</td>
<td>TEMPORARY STORE</td>
</tr>
<tr>
<td>SYMBAS DC</td>
<td>5,25800</td>
<td>SYMBOL TABLE BASE</td>
</tr>
<tr>
<td>ARBASE DS</td>
<td>2406</td>
<td>ARRAY BASE</td>
</tr>
<tr>
<td>ILBASE DS</td>
<td>2411</td>
<td>INTERMEDIATE LANGUAGE BASE</td>
</tr>
<tr>
<td>STKBAS DS</td>
<td>2416</td>
<td>STACK BASE</td>
</tr>
<tr>
<td>BASE DS</td>
<td>5</td>
<td>ARRAY SIZE</td>
</tr>
<tr>
<td>SIZE DC</td>
<td>5,0</td>
<td>ACCUMULATOR</td>
</tr>
<tr>
<td>ACC DC</td>
<td>5,0</td>
<td>ARRAY DIMENSION</td>
</tr>
<tr>
<td>DIM DC</td>
<td>2,0</td>
<td>ARRAY DIMENSION</td>
</tr>
</tbody>
</table>
W      DS  5'',
L0     DS  4'',
L1     DS  4'',
TABLE1 DC 5,-20000,,
TABLE2 DS 5,,
COMP   DC 2,0,,
OP     DS  2,,
IND    DC 2,0,,
      DC 7,0,,
MAC    DS  7,,
      DC 15,0,,
HOD    DS  5,,
LOD    DS  5,,
EAC    DC 3,0,,
EP     DS  2,,
EQ     DS  2,,
MQ     DS  7,,
EDIF   DS  3,,
UFLO   DAC 10,UNDERFLOW'',
OFLO   DAC 9,OVERFLOW'',
DIG    DC  2,0,,
      DC  7,0,,
FZERO  DC  2,0,,
      DC  7,1,,
FONE   DC  2,0,,
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'',
PPE1  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'',

ADDRESS OF FIRST ARRAY ELT
BRANCHING TABLE 1
BRANCHING TABLE 2
COMPARE
OPERATOR CODE
INDICATOR
MANTISSA ACCUMULATOR
HIGH ORDER DIGIT
LOW ORDER DIGIT
EXPONENT ACCUMULATOR
EXPONENT P
EXPONENT Q
MANTISSA Q
EXPONENT DIFFERENCE
DIGIT
FLOATING POINT ZERO
FLOATING POINT ONE
RTE1  DAC  6,RT E1',
RTE2  DAC  6,RT E2',
RF1  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,RE E1',
RDE2  DAC  6,RE E2',
RDE3  DAC  6,RE E3',
RDE4  DAC  6,RE E4',
TBE1  DAC  6,TB E1',
PRE1  DAC  6,PR E1',
PRE2  DAC  6,PR E2',
PRE3  DAC  6,PR E3',
PRE4  DAC  7,PRC E1',
PRE5  DAC  7,PRC E2',
PRE6  DAC  7,PRC E3',
PRE7  DAC  7,PRC E4',

**OP BRANCHING TABLE1********************************************************************
*

DORG 00005,

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,*DP,0,*RT
DSA  0,*RF,0,*RFRT,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,,
SPIM ,3,,
```

**INITIALIZE**

```
SETUP TFM OUTPUT-1,**+20,,
B7 OUTPUT+36,,
TFM RBUF MK,RBUF+158,,
CF MAC+1,,
CF IND-1,,
TF IC,ILBASE,,
AM IC,1,10,
TF SVP,STKBAS,,
TF STP,STKBAS,,
SM STP,9,10,
```

**CLEAR ARRAY AREA**

```
TF P,ARBASE,,
TEST85 C P,ILBASE,,
BNL START,,
TDM -P,0,,
AM P,10,10,
B7 TEST85,,
MUSGO DAC 26,MUSSEL INTERPRETIVE PHASE,,
```

**PRINT HEADING**

```
PRINT HEADING
```

CLEAR PRINT BUFFER AND INITIALISE MARKER

INITIALISE READ BUFFER MARKER

INITIALISE IC

SET STACK POINTERS

INITIALIZE WORKING POINTER

TEST P AGAINST I,L, BASE

IF GE, BRANCH TO START INTERPRETING

OTHERWISE SET ARRAY ELT TO UNDEFINED

INC POINTER

AND LOOP AGAIN

MUSGO DAC 26,MUSSEL INTERPRETIVE PHASE,
**ASSEMBLE AND STORE NUMBER ROUTINE**

DORG 20500,
TFM PBUF+54,5944,8,
TD COMP,-STP,,
CM COMP,6,10,
BE *+32,,
PRA RDE1,,
B7 HALT,,
TF A,-SVP,,
SM A,9,10,
TD COMP,-A,,
CM COMP,4,10,
BNE READIN-12,,
BTM ARSUB,READIN,,
AM A,9,10,
READIN TFM MCOUNT,0,10,
TFM EAC,0,9,
TFM NUMBER,NACC-9,,
TF BOTV,SVP,,
SM BOTV,2,10,
BTM BUFDMAN,1,10,
BT DIG7,RBUF MK,,
BE NUM,1,
BTM COMCHA,10,10,
BE READIN+36,,
CM -RBUF MK,20,10,
BNE *+32,,
SF IND,,
B7 READIN+36,,
BNR *+20,-RBUF MK,,
B7 STR,1,
CM -RBUF MK,46,10,
BE BOO+20,,
CM -RBUF MK,63,10,
BE BOO,,
CM -RBUF MK,23,10,
BNE *+32,,
TDM -STP+0,,
B7 INSTOR,,
CM -RBUF MK,0,10,
BE *+32,,
TDM -STP+2,,
DECHEC CM -RBUF MK,3,10,
BE REAL,,
CM -RBUF MK,45,10,
BE EXP,,
INCHEC PRA RDE2,,
B7 HALT,,

* FILL PBUF
* CHECK TOP STACK TYPE
* ERROR - NOT ADDRESS
* HALT TO CHECK ERROR
* SAVE ADDRESS
* CHECK ADDRESSED LOCATION CONTENT TYPE
* BRANCH TO INPUT IF NOT ARRAY WORD
* OTHERWISE PREPARE FOR ARRAY INPUT
* INITIALISE - MANTISSA COUNT
* EXPONENT ACCUMULATOR
* MANTISSA BUILD POINTER
* INC BUF MARKER
* CHECK FOR DIGIT
* DIGIT - BRANCH TO ASSEMBLE AND STORE NUMBER
* OTHERWISE CHECK NEXT CHARACTER
* + - BACK TO READIN
* MINUS - STORE
* THEN BACK TO READIN
* STRING DELIMITER - BRANCH TO PACK STRING
* BOOLEAN FALSE - TO BOOLEAN ROUTINE
* BOOLEAN TRUE - TO BOOLEAN ROUTINE
* COMMA - NULL INPUT - SET TYPE UNDEFINED
* AND BRANCH TO STORE NUMBER TYPE
* BLANK - NULL INPUT
* STORE NUMBER TYPE
* DECIMAL POINT - BRANCH TO REAL ROUTINE
* E - BRANCH TO EXPONENT ROUTINE
* ERROR - INCORRECT INPUT FORM
* HALT TO CHECK ERROR
**NUM**
- TDM -STP,2,,
- LOOP20 AM RBUF MK,1,10,
- BD OUT20,-RBUF MK,,
- B TM BUFMAN,1,10,
- BT DIG7,RBUF MK,,
- BE LOOP20,,
- ZE RO BTM COMCHA,3,10,
- BE REAL,,
- CM -RBUF MK,45,10,
- BE SKIPEE,,
- CM -RBUF MK,23,10,
- BE PACK0,,
- CM -RBUF MK,0,10,
- BNE INCH EC,,

**PACK0**
- TFL -SVP,FZERO,,
- B7 INSTOR,,

**SKIPEE**
- BTM BUFMAN,1,10,
- BTM COMCHA,23,10,
- BE PACK0,,
- CM -RBUF MK,0,10,
- BE PACK0,,
- B7 PACK0,,
- AM RBUF MK,1,10,

**OUT20**
- AM NUMBER,1,10,
- TD -NUMBER,-RBUF MK,,
- CF -NUMBER,,
- AM MCOUNT,1,10,
- BTM BUFMAN,1,10,
- CM MCOUNT,8,10,
- BE LOOP21,,
- BT DIG7,RBUF MK,,
- BE OUT20-12,,
- B7 OUT21,,

**LOOP21**
- BT DIG7,RBUF MK,,
- BNE OUT21,,
- BTM BUFMAN,2,10,
- AM EAC,1,10,
- B7 LOOP21,,

**OUT21**
- BTM COMCHA,23,10,
- BE PACKIN,,
- CM -RBUF MK,0,10,
- BNE DECHEC,,

**PACKIN**
- CM MCOUNT,8,10,
- BL INT,,
- PRA RDE4,,
- TF MAC+1,-NUMBER,,
- AM EAC,1,10,
- TFM HOD,MAC-6,,
- TFM LOD,MAC,,
- SF INFLAG,,
- B7 RNS,,

**INSTOR**
- BT STOR,SVP,,
- B7 ARCH EC,,

**INT**
- TFL -SVP,FZERO,,
- A -BOTV,-NUMBER,,
- MF -BOTV,IND,,
- B7 INSTOR,,

*STORE NUMBER TYPE*

*LOOP TO SKIP OVER NON SIG ZEROS*

*1ST SIG DIG - BRANCH OUT*

*INC BUF MARKER*

*CHECK FOR DIGIT*

*LOOP AGAIN IF DIGIT*

*OTHERWISE CHECK NEXT CHARACTER*

*DECIMAL POINT - BRANCH TO REAL*

*E - SKIP EXPONENT PART*

*ERROR IF NOT COMMA OR BLANK*

*OTHERWISE PACK ZERO*

*THEN BRANCH TO STORE*

*SKIP OVER EXPONENT PART*

*CHECK NEXT CHARACTER*

*PACK ZERO IF COMMA*

*PACK ZERO IF BLANK*

*OTHERWISE LOOP AGAIN*

*INC BUF MARKER*

*PACK NEXT DIGIT*

*INC MANTISSA COUNT*

*INC BUF MARKER*

*CHECK MANTISSA COUNT*

*IF 8 IGNORE REMAINING DIGITS*

*OTHERWISE CHECK FOR DIGIT*

*LOOP AGAIN IF DIGIT*

*OTHERWISE TERMINATE INTEGER PHASE*

*CHECK FOR DIGIT*

*TERMINATE INTEGER PHASE IF NOT*

*OTHERWISE INC BUF MARKER*

*INC EXPONENT ACCUMULATOR*

*AND LOOP AGAIN*

*TERMINATE INTEGER PHASE - CHK NEXT CHAR*

*CHECK FOR DECIMAL IF NOT COMMA OR BLANK*

*OTHERWISE PACK INTEGER - CHECK MCOUNT*

*STORE INTEGER IF LESS THAN 8*

*OTHERWISE PRINT WARNING ERROR*

*THEN ROUND AND STORE REAL*

*SET INPUT FLAG*

*BRANCH TO ROUND AND NORMALISE*

*BRANCH TO STORE*

*BRANCH TO ARRAY CHECK*

*STORE INTEGER*

*PACK MANTISSA*

*APPEND SIGN*

*BRANCH TO STORE*
REAL CM MCOUNT,0,10,
BNE OUT22+60,,
BT DIG7,RBUFMK,,
BNE ZERO,,
AM RBUFMK,1,10,
BD OUT22,-RBUFMK,,
SM EAC,1,10,
B7 LOOP22,,

LOOP23 CM MCOUNT,8,10,
BE SKIPO,,
OUT22 AM NUMBER,1,10,
AM MCOUNT,1,10,
TD -NUMBER,-RBUFMK,,
CF -NUMBER,,
SM EAC,1,10,
BTM BUFMAN,1,10,
BT DIG7,RBUFMK,,
BNE ENDRE,,,
AM RBUFMK,1,10,
B7 LOOP23,,

SKIPO BTM BUFMAN,1,10,
BT DIG7,RBUFMK,,
BNE ENDRE,,,
BTM BUFMAN,2,10,
B7 LOOP23,,

ENDRE BTM COMCHA,45,10,
BE EXPO,,

COMCHK CM -RBUFMK,23,10,
BE PACKRE,,
CM -RBUFMK,0,10,
BNE INCHEC,,,
PACKRE CM MCOUNT,8,10,
BE PACKIN+24,,
TF MAC,FZERO-2,,
A MAC,-NUMBER,,
TFM HOD,MAC-6,,
TFM LOD,MAC,,
SF INFLAG,,
B7 NORM,,

* CHECK MANTISSA COUNT AGAINST 0
* IF ZERO SKIP OVER LEADING Zeros
* INC BUF MARKER
* CHECK FOR DIGIT
* EXIT IF NOT
* OTHERWISE CHECK DIGIT
* BRANCH OUT IF NON ZERO
* OTHERWISE DEC EXPONENT ACCUMULATOR
* AND LOOP AGAIN
* CHECK MANTISSA COUNT AGAINST 8
* IF EQUAL SKIP OVER REMAINING DIGITS
* OTHERWISE PACK NEW DIGIT
* INC BUF MARKER
* CHECK FOR DIGIT
* IF NOT TERMINATE REAL PHASE
* OTHERWISE INC BUF MARKER
* AND LOOP AGAIN
* IGNORE REMAINING DIGITS
* IF NOT TERMINATE REAL PHASE
* OTHERWISE INC BUF MARKER
* AND LOOP AGAIN
* TERMINATE REAL PHASE - CHECK NEXT CHAR
* IF E BRANCH TO EXPONENT ROUTINE
* IF COMMA OR BLANK PACK REAL
* CHECK NEXT CHAR
* IF ERRANCH TO EXPONENT ROUTINE
* IF COMMA OR BLANK PACK REAL
* SET INPUT FLAG
* BRANCH TO NORMALISE AND STORE ROUTINE

* EXPONENT ROUTINE*
* EXPONENT ROUTINE*

* EXP +
BNE INCHEC,,
SF IND-1,,
B7 EXP+12,,
LOOP24 AM RBUFMK,,1,10,,
BD LOOP25,-RBUFMK,,
BTM BUFMAN,,1,10,,
BT DIG7,-RBUFMK,,
BE LOOP24,,
CF IND-1,,
BTM COMCHA,,23,10,,
B7 COMCHK+12,,
LOOP25 AM EXP,,1,10,,
TD -EXP,-RBUFMK,,
CF -EXP,,
BTM BUFMAN,,1,10,,
BT DIG7,RBUFMK,,
BNE OUT25,,
AM RBUFMK,,1,10,,
CM EXP,XAREA+1,,
BNE LOOP25,,
PRA RDE3,,
B7 HALT,,
OUT25 BTM COMCHA,,23,10,,
BE **+36,,
CM -RBUFMK,0,10,,
BNE INCHEC,,
MF -EXP,IND-1,,
A EAC,-EXP,,
CM M COUNT,,0,10,,
BNE PACKRE,,
TFL -SVP,FONE,,
A -SVP,EAC,,
MF -BOTV,IND,,
B7 INSTOR,,

ERROR IF NOT MINUS
OTHERWISE STORE MINUS SIGN
THEN BRANCH BACK TO LOOK FOR DIGIT
LOOP TO SKIP OVER LEADING (NS) ZEROS
BRANCH OUT ON SIG DIG
OTHERWISE INC BUF MARKER
CHECK FOR DIGIT
LOOP AGAIN IF DIGIT
OTHERWISE CHECK NEXT CHARACTER

INC EXPONENT COUNTER
PACK LATEST EXPONENT DIGIT
INC BUF MARKER
CHECK FOR DIGIT
IF NOT TERMINATE EXPONENT PHASE
OTHERWISE INC BUF MARKER
CHECK EXPONENT SIZE
ERROR - EXPONENT OVERFLOW
HALT TO CHECK ERROR
TERMINATE EXPO PHASE - CHECK NEXT CHAR
ERROR - EXPONENT OVERFLOW
OTHERWISE IF NOT COMMA OR BLANK
OTHERWISE APPEND EXPONENT SIGN
CHECK MANTISSA COUNT
IF NON ZERO BRANCH TO PACK REAL
OTHERWISE PACK MANTISSA VALUE ONE
PACK EXPONENT VALUE
APPEND MANTISSA SIGN
BRANCH TO STORE

PACK VALUE TRUE
PACK VALUE FALSE
PACK BOOLEAN TYPE DIGIT
INC BUF MARKER
CHECK NEXT CHARACTER
BRANCH TO STORE IF COMMA
BRANCH TO STORE IF BLANK
OTHERWISE LOOP AGAIN

STORE STRING DESCRIPTOR TYPE DIGIT
GRAB A NODE
PACK ADDRESS FIELD IN STRING DESCRIPTOR
*MUSINT LISTING*

**ARRAY CHECK ROUTINE**

```
LENGTH,0,8,
TFM LENGTH,0,8,
SET STRING LENGTH COUNTER TO ZERO
STRING,SAREA-14,,
TFM STRING,SAREA-14,,
SET STRING ASSEMBLY POINTER
COUNT,6,10,
TFM COUNT,6,10,
SET LOOP COUNTER

LOOP27

INC BUF MARKER

LOOP28

PTM

PAST

BTM

BUFMAN,1,10,
SF

-RBUF Mk,,
AM

RBUMK,1,10,
BNR

PAST,-RBUF Mk,,
BTM

BUFMAN,1,10,
BTM

COMCHA,23,10,
BE

OUT27,,
CM

- RBUF Mk,0,10,
BE

OUT27,,
BNR

INCHEC,-RBUF MK,,
TFM

-RBUF MK,12,10,

PAST

AM

STRING,3,10,
TF

-STRING,-RBUF MK,,
SM

STRING,1,10,
CF

-STRING,,
AM

LENGTH,1,10,
SM

COUNT,1,10,
BD

LOOP28,COUNT,,
SM

NODE2,4,10,
AM

STRING,1,10,
SF

SAREA-12,,
TF

-NO DE2,-STRING,,
AM

NODE2,4,10,
TF

NODE1,NODE2,,
BTM

GRAB,0,10,
SF

NODE2-3,,
TF

-NO DE1,NODE2,,
CF

NODE2-3,,
B7

LOOP27,,

OUT27

CM

LENGTH,0,10,
BNE

NONUL,,
TDM

-STP,1,,
B7

INSTOR,,
NONUL

SM

SVP,5,10,
TF

-SVP,LENGTH,,
AM

SVP,5,10,
CM

COUNT,6,10,
BNE

**44,,
SF

-NO DE1,,
BTM

RETURN,0,10,
B7

INSTOR,,
SF

-NO DE2,,
SM

NODE2,4,10,
AM

STRING,1,10,
SF

SAREA-12,,
TF

-NO DE2,-STRING,,
B7

INSTOR,,

* RETURN TO CONTROL IF NO ARRAY INPUT

***************
**ARRAY CHECK ROUTINE**
***************

ARC H EC

BNF

PO PO ,ARRAY,,
AM

ACCOUNT,1,10,

RETURN TO CONTROL IF NO ARRAY INPUT

OTHERWISE INC ARRAY ELT COUNT
*MUSINT LISTING*

C ACOUNT,SIZE,, CHECK AGAINST ARRAY SIZE
BNH #+32,,
CF ARRAY,, TERMINATE ARRAY INPUT IF HIGH
B7 POPO,, OTHERWISE INC A
AM A+10,10,, AND BRANCH TO INPUT NEXT ELEMENT
B7 READIN,,
POPO BTM POPTOP,10,10,8,, POP TYPE AND VALUE
B7 BACK,, RETURN TO CONTROL

* 

******************************************************
**READ BUFFER MANAGER SUBROUTINE**
******************************************************

* 

DS 2,, SPACE FOR PARAMETER (RBUMK INCREMENT)
BUFMAN A RBUMK,BUFMAN-1,, INC BUF MARKER
CM RBUMK,RBUF+15B,, CHECK AGAINST BUFFER END
BNH #+36,,
RACD RBUF,,, READ IN NEW CARD IF HIGH
SM RBUMK,160,9,, RESET BUF MARKER
BB2 ,, BRANCH BACK

* 

******************************************************
**SUBROUTINE TO COMPARE NEXT RBUF CHARACTER**
******************************************************

* 

DS 2,, SPACE FOR PARAMETER (COMPARE CHARACTER)
COMCHA SF -RBUMK,,, SET FLAG
AM RBUMK,1,10,, INC RBUF MARKER
CF -RBUMK,,, COMPARE
C -RBUMK,COMCHA-1,,
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,, SET A TO ADDRESS SMF
TF A,-A,,
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,7,,
B7  -ARSUB+1,,, 

SET A TO ADDRESS 1ST ARRAY ELT

******************************************************************************
**SUBROUTINE TO CHECK NEXT RBUF DIGIT AGAINST 7**
******************************************************************************

******************************************************************************
**SPACE FOR BRANCH OUT ADDRESS**
******************************************************************************

DS  5,,, 
DIG7  TD  COMP,-DIG7+1,,, 
     CF  COMP,,, 
     CM  COMP,7,10, 
     BB2  ,,,
*MUSINT LISTING*

**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,, CHECK TOP STACK TYPE
CM COMP,1,10,, NULL STRING
BE NULL1,,
CM COMP,2,10,, NUMBER
BE NUM1,,
CM COMP,3,10,, BOOLEAN
BE BOOL1,,
CM COMP,6,10,,
BNE *+24,,
BTM BUBBLE,CHOICE-12,, ADDRESS - BUBBLE ADDRESSED TYPE + VALUE
CM COMP,7,10,, STRING
BE STR1,,
PRA PRE2,,
B7 HALT,,
NULL1 SF SPFLG,, ERROR - INVALID STACK TYPE
B7 BACHEC,,
NUM1 BTM LSUB,PRINTO,, FIND MANTISSA LENGTH
BTM PBFMAN,8,10,, CLEAR SPACE FOR NUMBER
B7 BACHEC,,
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 PBUF MK,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 PBUF MK,2,10,,
CM -SVP,0,10,,
BE LOOP41+12,,, NO DECIMAL POINT FOR INTEGER
TFM ->PBUF MK,3,10,,
B7 LOOP41+12,,,
SIGON1 SF SIGFLG,,
TFM ->PBUF MK,70,10, FLAGGED 1 - TERMINATES PATTERN
BNF LOOP41+48,IND,,
TFM ->PBUF MK,2070,8,,
B7 LOOP41+48,,
IFFLAG BNF *+20,-POINT,, UNFLAGGED 1 - DIGIT POSITION
B7 NULL1,,
C BEFORE,LENGTH,,
BNH *+56,,
```
AM  PBUF MK,2,10,
BNF  LOOP41,SIGFLG,,
TFM  =PBUF MK,70,10,
B7  LOOP41,,,
BNF  SIGON2,SIGFLG,,
BUF INC AM  PBUF MK,2,10,
C  P,BOTV,,
BNH  #*+56,,
CM  BEFORE,0,10,
BNH  LOOP41,,,
TFM  =PBUF MK,70,10,
B7  LOOP41,,,  INC BUF MARKER
TFM  =PBUF MK,70,10,
TD  =PBUF MK, -P,,
AM  P,1,10,
B7  LOOP41,,,
SIGON2 SF  SIGFLG,,
BNF  BUF INC,IND,,,  INSERT ZERO
TFM  =PBUF MK,20,10,
B7  BUF INC,,,
PRINTO BTM  PBF MAN,16,10,  SET SPACE FLAG
TFM  =PBUF MK,70,10,
SF  SPFLG,,
BTM  PBF MAN,24,10,
B7  BACHEC,,,  INC BUF MARKER
TFM  =PBUF MK,20,10,
EFORM SM  PBUF MK,22,10,
S  LENGTH,-SVP,,
SM  LENGTH,1,10,
TFM  EAC,0,9,,
A  EAC,-SVP,,
A  EAC,LENGTH,,
BD  #*+24,EAC-2,,
SF  EAC-1,,
TFM  I*5,10,
TFM  =PBUF MK,70,10,
BNF  #*+24,IND,,,  THEN BRANCH TO BACK CHECK
TFM  =PBUF MK,2070,8,
TD  =PBUF MK, -P,,
AM  P,1,10,
AM  PBUF MK,2,10,
TFM  =PBUF MK,3,10,
LOOP42 AM  PBUF MK,2,10,
TFM  =PBUF MK,70,10,
C  P,BOTV,,
BH  #*+36,,
TD  =PBUF MK, -P,,
AM  P,1,10,
SM  I*1,10,
BD  LOOP42,I,,
AM  PBUF MK,4,10,
TFM  =PBUF MK,4510,8,
BNF  #*+36,EAC,,
CF  EAC,,,  SET EXPONENT ACCUMULATOR
TFM  =PBUF MK,20,10,
AM  PBUF MK,4,10,
TFN  =PBUF MK,EAC,,
B7  NULL1,,
B7  NULL1,,
B7 NULL1,,
BOOL1 BTM PBFMAN,10,10, CLEAR SPACE FOR BOOLEAN VALUE
BD *+32,-SVP,, CHECK VALUE
TF -PBUFMK,FALBUF,, INSERT FALSE
B7 NULL1,,, INSERT TRUE
TF -PBUFMK,TRUBUF,, THEN BRANCH TO SPACE AND BACK CHECK
B7 NULL1,,, STR1 BTM STREM,0,10, FIND STRING LENGTH REMAINDER
AM SVP,5,10,,
B7 TEST45,,, LOOP45 SM NODE1,4,10, BRANCH TO TEST FIRST NODE LINK
BTM PBFMAN,12,10, DEC NODE POINTER
TF -PBUFMK,-NODE1,, INC PBUFF MARKER
AM NODE1,4,10, FILL NODE VALUE
TF NODE1,-NODE1,, SET NODE1 TO LINK(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,
TFM PBUF+56,43,10,
TNF PBUF+28,-IC,,
TF ICM,IC,,
TF COUNT,-IC,,
TF TOPV,SVP,,
BTM POPTOP,1010,811,
S SVP-1,COUNT,,
S SVP-1,COUNT,,
TF IC,ICM,,

CHOOSE TD COMP,-STP,,
CM COMP,1,10,
BE NULL2,,
CM COMP,2,10,
BE NUM2,,
CM COMP,3,10,
BE BOOL2,,
CM COMP,6,10,
BNE *+24,,

BTM BUBBLE,CHOOSE-12,,
CM COMP,7,10,
BE STR2,,
PRA PRCSt,,
B7 HALT,,

NUM2 TFM BEFORE,0,9,
TFM AFTER,0,9,
TFM XC,0,9,
TF PFLG,ZERO-10,,
TFM P,BEFORE,,

INCINC BTM INCI,**+12,,
CM -IC,42,10,
BE INCINC,,
CM -IC,62,10,
BE INCINC,,
CM -IC,23,10,
BE INCINC,,
CM -IC,20,10,
BE INCINC,,
CM -IC,3,10,
BNE *+32,,

TFM P,AFTER,,
B7 INCINC,,
CM -IC,14,10,
BNE *+44,,

BTM KOUNT,14,10,
A -P,TOT,,
B7 INCINC+12,,
CM -IC,79,10,
BNE *+32,,

BTM KOUNT,79,10, 9 - COUNT
E - SET EXPONENT FLAG
SWITCH TO EXPONENT COUNT
RECORD MARK Terminates PIC
IF NO EFLG Branch to F-FORMAT
OTHERWISE Branch to E-FORMAT
ERROR - INVALID PIC CHARACTER
HALT TO CHECK ERROR
ADD EXPONENT
CHECK AGAINST BEFORE COUNT
ERROR IF HIGH
HALT TO CHECK ERROR
OTHERWISE ADD DECIMAL PART LENGTH
COMPARE WITH ZERO
ZERO IF LOW
ROUND IF NECESSARY
ASSEMBLE ZERO
BRANCH TO RESCAN PICTURE
FIND MANTISSA LENGTH
SET EXPONENT PART
BEFORE IS NOW TOTAL OUTPUT MANTISSA LENGTH
ROUND MANTISSA IF NECESSARY
<table>
<thead>
<tr>
<th>ELNGTH</th>
<th>CM</th>
<th>EAC,0,10*</th>
<th>FIND EXPONENT LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td>LOOP70-12,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFM</td>
<td>XL,1,9,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TF</td>
<td>POINT,SVP,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDM</td>
<td>-POINT,0,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>RESCAN,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFM</td>
<td>POINT,EAC-2,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP70</td>
<td>BD</td>
<td>OUT70,-POINT,,</td>
<td></td>
</tr>
<tr>
<td>AM</td>
<td>POINT,1,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>LOOP70,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT70</td>
<td>TFM</td>
<td>XL,EAC,,</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>XL,POINT,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM</td>
<td>XL,1,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>XL,XC,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNH</td>
<td>RESCAN,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRA</td>
<td>PRCE4,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>HALT,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESCAN</td>
<td>TF</td>
<td>IC,ICM,,</td>
<td></td>
</tr>
<tr>
<td>AM</td>
<td>IC,2,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td>-IC,62,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>CHECM,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNF</td>
<td>*+32,IND,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>NFLG,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>CHECHA-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>PFLG,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>CHECHA-24,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHECM</td>
<td>CM</td>
<td>-IC,20,10,</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>CHECHA,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNF</td>
<td>*+24,IND,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF</td>
<td>NFLG,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTM</td>
<td>PBF=MAN,2,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM</td>
<td>IC,2,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHECHA</td>
<td>CM</td>
<td>-IC,62,10,</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>MCHEC,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNF</td>
<td>*+32,IND,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFM</td>
<td>-PBUFMK,20,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>CHECHA-12,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFM</td>
<td>-PBUFMK,10,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>CHECHA-12,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCHEC</td>
<td>CM</td>
<td>-IC,20,10,</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>CHECOM,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTM</td>
<td>PBF=MAN,2,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNF</td>
<td>CHECHA-12,IND,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>MCHEC-40,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHECOM</td>
<td>CM</td>
<td>-IC,23,10,</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>CHECAS,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTM</td>
<td>PBF=MAN,2,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNF</td>
<td>CHECHA-12,SIGFLG,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFM</td>
<td>-PBUFMK,23,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>CHECHA-12,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHECAS</td>
<td>CM</td>
<td>-IC,14,10,</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>CHEC9,,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>BEFORE,LENGTH,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNH</td>
<td>*+68,,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM</td>
<td>BEFORE,1,10,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNF</td>
<td>*+24,SIGFLG,,</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Find Exponent Length**
- **Check for fit**
- **OK - Branch to Rescan Picture**
- **Error - Exponent too large**
- **Halt to check error**
- **Rescan picture - reset IC**
- **Check first character**
- **S - Floating sign**
- **If negative number set NFLG**
- **Otherwise set PFLG**
- **Floating minus**
- **Set NFLG if negative number**
- **Inc buf marker**
- **Inc IC**
- **Choice on next character**
- **S - Insert sign**
- **Minus - Insert sign if neg number**
- **Comma - Insert if SIGFLG on**
- **Asterisk - Compare before with length**
- **Dec before if high**
- **If sig flag**
BTM FLOATS, CHECHA-12,, CHECK FOR FLOATING SIGN AND INSERT ZERO  
BTM PBFMAN, 2, 10, OTHERWISE LEAVE BLANK  
B7 CHECHA-12,,, IF BEFORE LE LENGTH CHECK FOR FLOATING  
BTM FLOATS,**+12,,, SET SIGNIFICANCE FLAG  
SF SIGFLG,,,  
C P, BOTV,,, IF P GT BOTV INSERT ZERO  
BH CHECHA-12,,, OTHERWISE INSERT NEXT DIGIT  
TD ~PBUFMK, ~P, AND INC P  
AM P, 1, 10,  
B7 CHECHA-12,,, CHECK FOR FLOATING SIGN AND INSERT ZERO  
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,,, DECIMAL POINT - SET SIG FLAG  
BTM FLOATS,**+12,,, FLOATING SIGN CHECK  
TFM ~PBUFMK, 3, 10, INSERT DECIMAL POINT  
B7 CHECHA,,,  
CM ~IC, 24, 10,  
BNE *24,,, LEFT BRACKET - BRANCH TO REPSUB  
BTM REPSUB, CHECHA,,,  
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, ZER0-10,,, CLEAR PFLG,NFLG  
BNE *32, OFLG,,, ZERO EXPONENT IF ZER0 MANTISSA  
CF IND,,,  
B7 RESCAN+12,,, OTHERWISE STORE EXPONENT SIGN  
MF IND, EAC,,, THEN SWITCH COUNTS - BEFORE IS EXP COUN  
TF BEFORE, Xc,,, LENGTH IS EXPONENT LENGTH  
TF LENGTH, XL,,, SWITCH POINTERS - P IS EXPONENT POINTER  
TF P, POINT,,, BOTV POINTS TO LOW ORDER EXPONENT DIGIT  
TF BOTV, SVP,,, BRANCH TO RESCAN PICTURE  
B7 RESCAN+12,,, FIND STRING LENGTH REMAINDER  
STR2 BTM STREM, 0, 10, SET LENGTH TO TWICE STRING LENGTH  
TF LENGTH,-SVP,,,  
A LENGTH,-SVP,,,  
AM SVP, 5, 10,  
TF SBUFMK, TOPV,,,  
B7 TEST65,,, LAYOUT STRING AT TOP OF STACK  
LOOP65 SM NODE1, 4, 10,  
TF P, SBUF MK,,,  
AM SBUF MK, 12, 10,  
TF ~SBUF MK, ~NODE1,,,  
AM P, 1, 10,  
CF ~P,,,  
AM NODE1, 4, 10,  
TF NODE1, ~NODE1,,,
**MUSINT LISTING**

**PAGE 49**

**CF** NODE1-3...
**TEST65** BNF LOOP65,-NODE1...
**TF** P,SBUF,MARK,, A SBUF,MARK, TOP,PREM, C SBUF,MARK,BOUND,,
**BNH** *+32,, PRA CORFLO,, B7 HALT,,
**SM** NODE1,4,10, TF -SBUF,MARK,-NODE1,, AM P,1,10, CF -P,, TF SBUF,MARK, TOP,REM,, B7 LOOP67,-IC,,
**SM** BOUND,,
**BTM** INC#*+12,,
**BNR** LOOP67,-IC,, TF TOT2,LENGTH,, B7 LOOP67+36,,
**LOOP67** CM -IC,14,10,,
**BNE** BCHECK,,
**BTM** KOUNT,14,10,, AM SBUF,MARK,1,10,,
**SF** -SBUF,MARK,, SM SBUF,MARK,1,10,, C TOT2,LENGTH,,
**BH** ++68,, A SBUF,MARK,TOT2,,
**BT** PB#FMAN,TOT2,,
**TF** -PBUF,MARK,-SBUF,MARK,, S LENGTH,TOT2,,
**B7** LOOP67,, CM LENGTH,0,10,,
**BE** PADOUT,,
**A** SBUF,MARK,LENGTH,,
**BT** PB#FMAN,LENGTH,,
**TF** -PBUF,MARK,-SBUF,MARK,, S TOT2,LENGTH,,
**B7** PADOUT,,
**BCHECK** CM -IC,42,10,,
**BNE** *+44,,
**BTM** KOUNT,42,10,,
**BT** PB#FMAN,TOT2,,
**B7** LOOP67,,
**BNR** PICERR,-IC,,
**B7** BACHEC,,
**LOOP63** AM IC,2,10,, AM TOT2,2,10,,
**PADOUT** BNR LOOP63,-IC,,
**SF** SPFLG,,
**BT** PB#FMAN,TOT2,,
**B7** BACHEC,,
**NULL2** AM IC,2,10,,
**TFM** TOT2,0,10,,
**B7** PADOUT,,
**BOOL2** BD *+32,-SVP,,
**TFM** BOOBUF,FALBUF,,
**B7** *+20,,
**TFM** BOOBUF,TRUBUF,,

CHECK C(SBUF,MARK) AGAINST C(BOUND)

IF GREATER, CORE OVERFLOW

HALT

BRANCH TO MATCH OUTPUT PATTERN

IF NON NULL PICTURE

OTHERWISE SET TOT2

AND BRANCH TO 0/P CONSTANT STRING

CHOICE ON NEXT PIC CHARACTER

* - COUNT

SET SBUF FLAG

CHK * COUNT AGAINST REMAINING STR LENGTH

IF LE LAYOUT STRING CHUNK IN PBUF

DEC LENGTH

LOOP AGAIN

OTHERWISE LAYOUT REMAINDER OF STRING

THEN BRANCH TO FILL BLANK PADDING

B - COUNT BLANKS

INC BUF MARKER

LOOP AGAIN

ERROR IF NOT RECORD MARK

OTHERWISE BRANCH TO BACK CHECK

LOOP AGAIN IF NO RECORD MARK

OTHERWISE

INC BUF MARKER

AND BRANCH TO BACK CHECK

NULL STRING - INC IC

SET TOT2 TO 0

BRANCH TO FILL PADDING

CHECK BOOLEAN TYPE

SET BOOBUF TO POINT TO FALSE BUF

SET BOOBUF TO POINT TO TRUE BUF
**SUBROUTINE TO COUNT NUMBER OF ASTERISKS, BLANKS, OR NINES IN PIC**

```
**********
*DS KOUNT TFM* 
**SPACE FOR CHARACTER CODE**
*DS 2,, 
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,, BRANCH BACK IF NOT 
BB2 ,, 
TFM REPCNT,NAACC-9,, OTHERWISE ASSEMBLE REP COUNT 
LOOP61 AM IC,1,10, CHECK FOR DIGIT 
TD COMP,-IC,, 
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,, 
FLOATS BNF *+44,PFLG,, SPACE FOR BRANCH OUT ADDRESS 
IF PFLG 
*
*MUSINT LISTING*

CF PFLG,, CLEAR
TFM →PBUMK,10,10, AND INSERT PLUS SIGN
B7 ++44,, IF NFLG
BNF ++36,NFLG,, CLEAR
CF NFLG,,
TFM →PBUMK,20,10, AND INSERT MINUS SIGN
BTM PBFAN,2,10, INC BUF MARKER
TFM →PBUMK,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 REPPLICATION
A IC,RCOUNT,,, INC IC
AM IC,5,10, BRANCH BACK
B7 →REPSUB+1,,, SPACE FOR BRANCH OUT ADDRESS
REPSUB SF REPFLG,,, IF REP FLAG
TFM REPCNT,NACC-9,,, ASSEMBLE REP COUNT
TFM RCOUNT,1,9,,
LOOP90 AM IC,1,10, SPACE FOR BRANCH OUT ADDRESS
BT DIG7,IC,,
BE ++32,, BRANCH BACK
S IC,RCOUNT,,, SPACE FOR BRANCH OUT ADDRESS
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,
SUBROUTINE TO FIND MANTISSA LENGTH

```
* DS S,, BRANCH ADDRESS FOR ZERO MANTISSA
LSUB TF BOTV,SVP,, BOTV POINTS TO MANTISSA
 SM BOTV,2,10,
 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,
 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,,, NODE1 POINTS TO FIRST STRING NODE
STREM TF NODE1,-SVP,, DIVIDE STRING LENGTH BY 6
 SM SVP,5,10,
 LD 99,-SVP,, MULTIPLY REMAINDER BY 2
 DM 96,6,10,
 TF TOPREM,99,, AND STORE IN TOPREM
 A TOPREM,99,,
 CM TOPREM,0,10,
 BNE *+24,,, SET TO 12 IF ZERO
 TFM TOPREM,12,10,, BRANCH BACK
 BB2 ,,*
* 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,, OTHERWISE COPY ADDRESSED TYPE
 SM A,9,10,
 TF P,STP,, AND VALUE TO ORIGINAL STACK POSITION
 LOOP50 TD -P,-A,,
 C P,SVP,,
```
THEN BRANCH OUT

* ******************************************************
**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,, TERMINATE ARRAY OUTPUT IF HIGH
CF ARRAY,, THEN BRANCH TO MULT SPEC CHECK
B7 FIN,, OTHERWISE INC A
AM A,1,10, AND BRANCH TO BUBBLE NEXT ARRAY ELT
B7 LOOP50-12,,, 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
*MUSINT LISTING*

* ****************************
** 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**
*  
RNC 25801,, SET SYMBOL TABLE
*  
**READ I.L.**
*  
TFM COUNT,0,10,
READIL RNC -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)

<table>
<thead>
<tr>
<th>Error types</th>
<th>Prob. 1, run 1</th>
<th>Prob. 2, run 1</th>
<th>Prob. 3, run 1</th>
<th>Prob. 3, run 2</th>
<th>Prob. 3, run 3</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs analysed</td>
<td>173</td>
<td>172</td>
<td>86</td>
<td>52</td>
<td>28</td>
<td>511</td>
</tr>
<tr>
<td>Number that 'went'</td>
<td>88</td>
<td>51</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>150</td>
</tr>
<tr>
<td>% that 'went'</td>
<td>51</td>
<td>30</td>
<td>5</td>
<td>10</td>
<td>7</td>
<td>29.4</td>
</tr>
</tbody>
</table>

(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     | 1              | 1              | 14             | 1              | 0              | 17    | 2.9  |
variable name                 |                |                |                |                |                |       |
(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 $\geq 4000$,
   (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 $N \leq 50$;
   (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(Nil)'.

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

<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UJ</td>
<td>105</td>
</tr>
<tr>
<td>6</td>
<td>LN</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>PE</td>
<td>(2,0)</td>
</tr>
<tr>
<td>17</td>
<td>LN</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>TLR</td>
<td>J</td>
</tr>
<tr>
<td>26</td>
<td>TNO</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>OP</td>
<td>EQ</td>
</tr>
<tr>
<td>32</td>
<td>IFJ</td>
<td>55</td>
</tr>
<tr>
<td>38</td>
<td>LN</td>
<td>5</td>
</tr>
<tr>
<td>43</td>
<td>TLR</td>
<td>I</td>
</tr>
<tr>
<td>47</td>
<td>VI</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>UJ</td>
<td>103</td>
</tr>
<tr>
<td>55</td>
<td>LN</td>
<td>6</td>
</tr>
<tr>
<td>60</td>
<td>RAF</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>TLA</td>
<td>J</td>
</tr>
<tr>
<td>66</td>
<td>TLR</td>
<td>I</td>
</tr>
<tr>
<td>70</td>
<td>TLR</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>TLR</td>
<td>J</td>
</tr>
<tr>
<td>78</td>
<td>OP</td>
<td>./.</td>
</tr>
<tr>
<td>82</td>
<td>TLR</td>
<td>J</td>
</tr>
<tr>
<td>86</td>
<td>OP</td>
<td>*</td>
</tr>
<tr>
<td>90</td>
<td>OP</td>
<td>-</td>
</tr>
<tr>
<td>94</td>
<td>CP</td>
<td>(HCF,2)</td>
</tr>
<tr>
<td>101</td>
<td>VI</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>RE</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>LN</td>
<td>8</td>
</tr>
<tr>
<td>110</td>
<td>TA</td>
<td>N</td>
</tr>
<tr>
<td>115</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>TA</td>
<td>M</td>
</tr>
<tr>
<td>122</td>
<td>RD</td>
<td>(cont...)</td>
</tr>
<tr>
<td>Digit</td>
<td>OP</td>
<td>Parameter</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>---------------</td>
</tr>
<tr>
<td>124</td>
<td>LN</td>
<td>'9'</td>
</tr>
<tr>
<td>129</td>
<td>RAF</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>TA</td>
<td>N</td>
</tr>
<tr>
<td>136</td>
<td>TA</td>
<td>M</td>
</tr>
<tr>
<td>141</td>
<td>CP</td>
<td>(HCF,2)</td>
</tr>
<tr>
<td>148</td>
<td>PR</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UJ</td>
<td>'289'</td>
</tr>
<tr>
<td>6</td>
<td>LN</td>
<td>'3'</td>
</tr>
<tr>
<td>11</td>
<td>PE</td>
<td>(1,5)</td>
</tr>
<tr>
<td>17</td>
<td>LN</td>
<td>'6'</td>
</tr>
<tr>
<td>22</td>
<td>TLA</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>LN</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>UJ</td>
<td>'45'</td>
</tr>
<tr>
<td>39</td>
<td>BA</td>
<td>'276'</td>
</tr>
<tr>
<td>45</td>
<td>TLA</td>
<td>TOP</td>
</tr>
<tr>
<td>49</td>
<td>TR</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>65</td>
<td>TN</td>
<td>'-1'</td>
</tr>
<tr>
<td>76</td>
<td>RF</td>
<td>(39,7)</td>
</tr>
<tr>
<td>85</td>
<td>LN</td>
<td>'8'</td>
</tr>
<tr>
<td>90</td>
<td>TLA</td>
<td>MAXPOS</td>
</tr>
<tr>
<td>94</td>
<td>TN1</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>LN</td>
<td>'9'</td>
</tr>
<tr>
<td>103</td>
<td>UJ</td>
<td>'115'</td>
</tr>
<tr>
<td>109</td>
<td>BA</td>
<td>'201'</td>
</tr>
<tr>
<td>115</td>
<td>TLA</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>TN</td>
<td>'2'</td>
</tr>
<tr>
<td>130</td>
<td>TLR</td>
<td>TOP</td>
</tr>
<tr>
<td>134</td>
<td>TN1</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>RF</td>
<td>(109,9)</td>
</tr>
<tr>
<td>145</td>
<td>LN</td>
<td>'10'</td>
</tr>
<tr>
<td>150</td>
<td>TLA</td>
<td>LIST</td>
</tr>
<tr>
<td>154</td>
<td>TLR</td>
<td>I</td>
</tr>
<tr>
<td>158</td>
<td>INDR</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>TLA</td>
<td>LIST</td>
</tr>
</tbody>
</table>

(cont...)
<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>164</td>
<td>TLR</td>
<td>MAXPOS</td>
</tr>
<tr>
<td>168</td>
<td>INDR</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>OP</td>
<td>'GT'</td>
</tr>
<tr>
<td>174</td>
<td>IFJ</td>
<td>'195'</td>
</tr>
<tr>
<td>180</td>
<td>LN</td>
<td>'11'</td>
</tr>
<tr>
<td>185</td>
<td>TLA</td>
<td>MAXPOS</td>
</tr>
<tr>
<td>189</td>
<td>TLR</td>
<td>I</td>
</tr>
<tr>
<td>193</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>UJ</td>
<td>'136'</td>
</tr>
<tr>
<td>201</td>
<td>LN</td>
<td>'13'</td>
</tr>
<tr>
<td>206</td>
<td>TLA</td>
<td>T</td>
</tr>
<tr>
<td>210</td>
<td>TLA</td>
<td>LIST</td>
</tr>
<tr>
<td>214</td>
<td>TLR</td>
<td>TOP</td>
</tr>
<tr>
<td>218</td>
<td>INDR</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>LN</td>
<td>'14'</td>
</tr>
<tr>
<td>227</td>
<td>TLA</td>
<td>LIST</td>
</tr>
<tr>
<td>231</td>
<td>TLR</td>
<td>TOP</td>
</tr>
<tr>
<td>235</td>
<td>INDA</td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>TLA</td>
<td>LIST</td>
</tr>
<tr>
<td>241</td>
<td>TLR</td>
<td>MAXPOS</td>
</tr>
<tr>
<td>245</td>
<td>INDR</td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>249</td>
<td>LN</td>
<td>'15'</td>
</tr>
<tr>
<td>254</td>
<td>TLA</td>
<td>LIST</td>
</tr>
<tr>
<td>258</td>
<td>TLR</td>
<td>MAXPOS</td>
</tr>
<tr>
<td>262</td>
<td>INDA</td>
<td></td>
</tr>
<tr>
<td>264</td>
<td>TLR</td>
<td>T</td>
</tr>
<tr>
<td>268</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>UJ</td>
<td>'76'</td>
</tr>
<tr>
<td>276</td>
<td>LN</td>
<td>'17'</td>
</tr>
<tr>
<td>281</td>
<td>TLA</td>
<td>LIST</td>
</tr>
</tbody>
</table>

(cont...)
<table>
<thead>
<tr>
<th>Digit</th>
<th>OP</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>285</td>
<td>PR</td>
<td></td>
</tr>
<tr>
<td>287</td>
<td>RE</td>
<td></td>
</tr>
<tr>
<td>289</td>
<td>LN</td>
<td>'19'</td>
</tr>
<tr>
<td>294</td>
<td>TA</td>
<td>N</td>
</tr>
<tr>
<td>299</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>LN</td>
<td>'20'</td>
</tr>
<tr>
<td>306</td>
<td>RAP</td>
<td></td>
</tr>
<tr>
<td>308</td>
<td>TA</td>
<td>N</td>
</tr>
<tr>
<td>313</td>
<td>CP</td>
<td>(SORTARRAY,1)</td>
</tr>
<tr>
<td>322</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
REFERENCES


32. IBM. 1620 Central Processing Unit, Model 2. IBM Systems Reference Library, Form No. A26-5781-2.

