IBM TECHNICAL INFORMATION EXCHANGE

TIE 4-0064
March 24, 1964
74 pages

BASIC CODING TECHNIQUES FOR THE 1401 COMPUTER

David A. Barrett
Associate Systems Engineer
IBM Israel Ltd.
15, Lincoln Street
Tel-Aviv, Israel

Distributed by
DPD Program Information Department
IBM Corporation
112 East Post Road
White Plains, New York

TABLE OF CONTENTS

CONTENTS PAGE

Abstract of Paper.

Basic Coding Techniques for the IBM 1401 Computer.

David A. Barrett

26th February 1964

Direct Inquiries to :

David A. Barrett,
IBM (Israel) Ltd.,
15, Lincoln st.,
Tel-Aviv,
Israel.

Tel-Aviv 31241

This paper is designed as an educational aid to programmers beginning to work on the IBM 1401 computer, and it also includes an analysis of more advanced techniques of program coding which may be of interest to more experienced programmers. It is divided into two sections, the first of which analyses the use of the 1401 instructions, the second of which describes various 1401 programming techniques. It could be used as a supplement to the 1401 machine manual.

Basic Coding Techniques for the 1401 Computer

INTRODUCTION

It is not always advisable to restrict programmers to simple techniques. A short-term advantage may be gained, in that inexperienced personnel will make fewer mistakes. They will tend not to over-reach themselves.

However, the 1401 computer offers a large number of alternative methods of carrying out any particular operation. In different circumstances, any one of these methods may turn out to be the most efficient solution, either because it uses up less of the core storage, or because it will be executed faster. When a programmer is able to condense a sequence of instructions into a very small section of core storage, he is making room for extra instructions and is effectively expanding the power of his machine. In the same way, time-saving techniques increase the effective speed of the machine. Such additions to the power and speed of the 1401 can be quite significant.

Tight programming will also have a very beneficial effect on the morale of the installation which employs it. Programmers gain a thorough understanding of their computer, and make fewer mistakes in the simpler types of operation. They also find themselves confronted with the problem of not only making a program work, but of making it work efficiently. This makes every program intrinsically more interesting, and adds to the concentration which they can bring to its solution.

Programmers should accustom themselves to the use of storage -- or time-saving techniques as often as possible, so that they may be entirely familiar with them when the need arises.

The potential IBM customer is advised to select a team of trainee programmers, on the basis of various aptitude and intelligence tests. These trainees are subjected to intensive IBM courses. Those who achieve high ratings on the course are retained as programmers by the customer, and are expected to begin writing actual programs for their installation immediately. Subsequent additions to the programming staff undergo a similar training, but benefit to a real if small degree from the advice of their by now more experienced colleagues. Naturally the customer requires that his installation be operational as soon as possible, and the programmer training is designed with this

...2

end in view. Therefore, graduates of the courses have not had time to absorb more than the essential principles. In time they will build up a fund of experience in the avoidance of errors and the creation of elegant coding techniques. This paper is intended to offer advice on the avoidance of the more common programming errors, and to introduce inexperienced programmers to some of the more complicated operations which are possible on the 1401.

Section I offers advice on the use of each machine instruction, and would provide a useful supplement to the standard 1401 machine manual.

Section II suggests solutions to several common coding problems.

SECTION I

1. The Add Instruction

It is never possible to half-chain an add instruction. The instruction 'A COUNT' will double the contents of the area called COUNT. Always use this form of the instruction to multiply a number by two, so that this use becomes second nature. It is a common mistake to try to add a series of scattered data fields to a series of sequential accumulators in storage using half-chained add instructions.

Let us suppose that COUNT contains an unsigned number. The instruction 'A *-6, COUNT' will not produce a zone on the junior position of COUNT. It will operate exactly the same as 'A @1@' COUNT.

If you are developing series of totals at different levels, major, intermediate, final, etc., do not add your data field to each level separately, but arrange to add the data fields to the lowest level of total only. When it is necessary to read out the contents of the minor totals, then add the minor totals before clearing the minor totals to zero. Where possible, arrange your totals so that the totals of different kinds are in the same sequence within each level, so that one level may be added to the next higher level using chained add instructions. For instance, using the suffix M to indicate a minor total, I to indicate intermediate, and F to indicate final, totals of Sales and Receipts should be organised as follows:

```
RECF   DCW # 10
SALEF  DCW # 10
RECI   DCW #  9
SALEI  DCW #  9
RECM   DCW #  8
SALEM  DCW #  8
```

so that the minor totals may be added to the intermediate by the phase:

```
A   SALEM,   SALEI
A
```

...4

Notice also that it is not usually a good idea to define a total which occurs as part of such a hierarchy with an area-defining literal.

```
(A SALEM, SALEI # 10) for instance.
```

For uses of the add instruction in address modification, see Section II paragrph 1 on address modification in general.

If you wish to remove the zones from a field, and replace all blanks with zeroes, this may be accomplished by the following instruction:

```
A       @0@, FIELD
```

This will not clear zones from the high-order or the low-order position of FIELD.

On machines with less than 16, 000 positions of memory it is not advisable to add into the highest accessible position of storage. If the sign of the B-field is changed by the add instruction, a process check will occur. For example, on an 8K machine, positions 7998 and 7999 contain 05 (ON). The instruction A@6@, 7999 will cause a process error under these circumstances, because the recomplement cycle will try to access position 8, 000, causing a wrap-around check.

If you use the common technique of using part of an instruction as a constant in an add operation, for instance, A *-6, COUNT, make sure that the constant has a word-mark in position. For instance, do not expect the following sequence of instructions to add five to a field called COUNT:-

```
A * + 8, COUNT # 5
BCE LOOP, FIELD, 5
```

2. ## The Zero and Add Instruction

The instruction ZA FIELD will remove zones from th(middle positions of FIELD and produce a zone on the junior position of FIELD. It will not produce zeroes in the high order positions of FIELD if these were originally blank. This can be accomplished by the instruction A @O@, FIELD.

The instruction ZA FIELD is often used before a divide instruction, to ensure that the field is in the correct format for the divide instruction (zone on junior position only).

3. The Subtract Instruction

Many of the comments made in Section I paragraph 1 also apply to the subtract instruction. Add and subtract are very similar in operation.

The instruction S TOTAL is often used to clear a field to zeroes. It should be noted that this instruction, besides clearing the field called TOTAL to zeroes, will also produce a sign on the junior position of that field, which may be plus or minus according as the previous state of the field was positive or negative. This can be significant in case the sign of the field has some special meaning in the program.

4. The Zero and Subtract Instruction

Do not be confused by the mnemonic.
ZS TOTAL will not clear the field 'TOTAL' to zeroes.
The instruction to accomplish is S TOTAL.

Remarks in Section I paragraph 2 are also applicable to this instruction.

5. The unconditional Branch Instruction

Remember that on a machine with the advanced programming feature, when any branch instruction is fulfilled, the B-address register will contain the address of the next sequential instruction after the branch. This is very important for the correct understanding of subroutine linkages on the 1401.

On machines of less than 16,000 memory positions, the fourth character of an unconditional branch instruction may not be in the highest accessible core location, as this will cause a wrap-around process error when the branch comes to be executed.

A branch instruction must be followed by a blank or by a word-mark. Any branch instruction which is not four, five, eight or one characters in length will either cause a process check (length 3 or 6) or act as a no-operation instruction.

6. The Branch if Indicator on Instruction

A subroutine may be called equally well from a conditional branch as from an unconditional. In all cases, where the condition tested for is fulfilled, then the B-address register will finally contain the location of the next sequential instruction after the branch instruction, and a subroutine will be able to exit to the next sequential instruction in the normal manner. However, be careful to avoid the following quite common mistake. Assume a situation where an error condition is tested for. If, for instance, two fields are unequal, there has been an error in the input data. If the fields are equal, then all is well and you now wish to call a subroutine. Be careful to use this sequence of instructions:-

```
C       FIELDA,  FIELDB
BE      * + 5
H       * - 3
B       SUBR
```

and not this one:-

```
C       FIELDA,  FIELDB
BE      SUBR
H       * - 3
```

Whereby the subroutine will exit to the error halt instruction!

7. The Branch if Character Equal Instruction

Notice that this instruction may also be used to call a subroutine in the normal manner.

Notice also the example in the 1401 manual whereby an entire field may be checked for the presence of a particular character, by using a fully chained branch instruction. This sequence of instructions will check whether there is a blank in any of the core locations 76 through 80. It can not be used to check that all core locations 78 through 80 do contain blanks. It is essentially a negative rather than a positive test, because the first fulfilled branch will cause an exit from the routine.

```
BCE   ERROR,   80,
CHAIN  4
```

If it is required to check that all the positions 76 through 80 <u>are</u> blank, then either use a compare, or with the column binary feature:-

```
BBE   ERROR,   80,⧧
CHAIN   4
```

Where the symbol ⧧ represents the bit configuration (BA 8421) - the group mark.

The fact that the B-address register chains down one location after a BCE that has not resulted in a branch can be very useful. The same applies to all eight-character branch instructions. - (BBE, BWZ).

One difficult thing to do on the 1401 is to check a single position of storage for various characters. This can be accomplished as follows: assuming that there is a word-mark under the CODE to be tested, and that we wish to test for A, 6, 9, Q or O.

```
MCW     CODE,  * + 8
BCE     YES, @A69QO@,
CHAIN   4
```

It is not possible to combine the BCE 8-position instruction and the BU 5-position instruction into a hybrid which says, for instance, BU ERROR, 80, 1 as I have sometimes seen done.

8. The Branch if work-mark and/or Zone Instruction

The most common use of this instruction is to test for the presence of a work-mark, or to test for the presence of a minus zone. Both these operations have extended mnemonics in autocoder, so that it is not necessary to write the d-modifier. Thus

```
BWZ     LOOP1,  80,  K
BWZ     LOOP1,  80,  1
```

are equivalent to

```
BM      LOOP1,  80
BW      LOOP1,  80
```

respectively.

It is still necessary to write two addresses for the BM or BW.

It is not possible to half-chain a BWZ instruction. A four position BWZ, when followed by a work-mark, operates exactly as an unconditional branch under the same circumstances. Thus

```
B       LOOP
NOP
```

is equivalent to

```
BWZ     LOOP
NOP
```

A BWZ may of course be fully chained in the same way as a BCE or BBE.

There are nine possible d-modifiers to the BWZ instruction, and these are made up as follows:-

If it is required to test for a work-mark, a one-bit is included in the D-modifier. If it is required to test for a zone, a two bit is included in the D-modifier.

If a two bit is included in the d-modifier, then the A-B bit configuration of the zone to be tested for must be added to the d-modifier. If a one and a two bit occur simultaneously in a d-modifier, the presence of a word-mark *or* the presence of the required zone will cause the branch to become effective.

9. The no-operation instruction

Note that the NOP instruction may have any length *except* 3-characters and 6-characters. This, the following instructions in storage would both cause a process check:

N 1 1 N OOOOO

Notice also that the A- and B-address registers are changed by a NOP instruction, and therefore the A- and B-addresses, where they exist, must be valid addresses for the machine you are using.

. . . 14

10. The Compare Instruction

One of the most common programming slips is to follow a compare instruction by the wrong conditional branch. Be very sure to understand that:

BH tests for the B- field greater than the A- field.

BL tests for the B- field less than the A- field in the proceding compare instruction.

Numbers are not compared algebraically by the compare instruction. Thus, a negative number may be considered to be greater than a positive number. The result of the compare depends upon the alphabetic collating sequence, shown at the end of the 1401 manual. Remember that a number with a zone is equivalent to an alphabetic character, and it is this aspect of the number that is considered by the compare instruction.

This creates a problem when comparing one number with another, if the numbers involved may have different signs. The safest technique to use in this case is to compare one number with the other and test for the equal condition, and then subtract one number from the other and test for a minus zone on the result. For example, to compare the algebraic values of two numbers, C and D, the following routine could be used. It is assumed that the fields C, D and E are of the same length, and that both C & D have a sign and leading zeroes, not blanks.

```
                C       C, D
                BE      EQUAL
                ZA      C, E
                S       D, E
                BM      DBIG, E
CBIG            —
```

Notice that, if the test for equal compare is omitted initially, the program may go to either DBIG or CBIG when C = D, depending on whether the sign of C, D is negative or positive.

If it is not known whether C and D have a sign when positive or whether they have leading zeroes or blanks, then the following routine must precede the one shown above.

```
S      @O@,  C
S      @O@,  D
```

which will a) fill in leading zeroes

b) remove all zones except on the high and low order positions.

c) create a zone on the low-order position if it is absent.

If two fields of unequal length are compared, and the B-field is longer than the A-field, the result will always be B greater than A.

If it is necessary to compare a sub-field with no word-mark in its left-hand position with a field or constant with a word-mark, then the field with the word-mark must always be specified in the B-address of the compare instruction. Thus, unless there is a word-mark in position 80, the following instructions will always be a valid test for the number eleven in memory positions 79 and 80 :-

```
C      80,  @11@
BE     ELEVEN
```

Quite often a compare instruction may be used for purposes other than comparing two numbers. The compare instruction may be used to search for the word-mark in a variable length field without affecting that data in any way. For instance, the series of instructions:

```
C      FIELD
SBR    ADDR
```

will store the address of the memory location immediately to the left of the first word-mark, encountered in 'FIELD'. Consider the following applications.

Let us assume that is necessary to scan through a small file of cards, each of which contains a name in columns 11-80. Furthermore it is worth while storing these names variable length because there is only room in storage for all the significant characters of the names, and not for the blanks which pad out the shorter names.

The object of scanning this file is to check for names which occur on more than one card, and print out duplicates when they occur. Providing that the file is small enough, this could be done as follows:

Store all names in a table in storage, from the top of core downwards. When a new card is read in, find the last character of the name, and compare the name with every entry in the table. If an equal comparison occurs, print the name, if no equal comparison occurs before the end of the table, the name must be added to the table.

We shall use index register one to contain the address of the location immediately to the left of the last name stored in the table, index register two to contain the address of the name we are comparing against in the table, and index register three to contain the address of the rightmost character of the name in the card. Initially, index register one will be set to the topmost position of storage: Names may be from one to 70 characters long, and will contain only alphabetic characters.

```
START     SBR          89, 15999
          SW           11, 87
LOOP1     R
          SBR          94, 15999
          SBR          99, 80
LOOP2     C            0+X3,  BLANK
          SAR          99
          BU           FOUND
          C            99,  @10@
          BU        LOOP2
          H
BLANK     DCW       @b@
FOUND     SBR       99, 1+X3
LOOP3     C         94, 89
          BE        INSERT
          C         0+X3, 0+X2
          BU        NEXT
          C         0+X2, 0+X3
          BU        NEXT
          MCW       0+X3, 299
          W
          CS        299
          B         LOOP1
NEXT      C         0+X2
          SAR       94
          B         LOOP3
INSERT    LCA       0+X3, 0+X1
          SBR       89
          B         LOOP1
```

Do not attempt to compare two addresses to discover whether one is greater than the other, unless both addresses must be less than 1,000. One rather cumbersome method of comparing addresses is as follows:- To find out if ADDR A is greater than ADDR B. There are word-marks in ADDRA-2 and ADDRB-2.

The machine has 16,000 position of memory.

```
          MCW       ADDRA, LOOP + 3
          MCW       ADDRB, 89
          MCW       @⧧@, 15998
          MCW       0+X1, STORE # 1
          MCW       @  @, 0+X1
LOOP      MCW       0
          MN
          SBR       LOOP + 3
          MCW       STORE, 0+X1
          C         LOOP + 3, ADDRB
          BE        NO
          C         LOOP + 3, @I9H@
          BE        YES
          B         LOOP - 7
```

Notice that you may chain a compare across a B-field word-mark and still obtain a valid result for the whole double field. Thus

```
          C         FLDA, FLDB
          C
```

will compare all of FLDA with all of FLDB quite validly if FLDA and FLDB are of the same length, but there is an extra word-mark in the middle of FLDB. This is because a compare instruction does not reset the HLE indicators to equal until the 3rd position of a compare instruction is loaded into the registers.

There are two particular applications of the compare instruction that may be handled in slightly unusual ways. The first occurs when you wish to test a long field to see if it contains the same character in every location. More usually, if you wish to test an entire field to see if it is completely blank. It would be possible to compare the field with a blank constant of the same length, but in the case of long fields, this means defining a large blank constant. An elegant method of testing a whole field for blanks is as follows:-

```
          BCE       COMP, FIELD, b
          B         NOTBL
COMP      C         FIELD, FIELD-1
          BU        NOTBL
BLANK     --
```

The second occurs when you wish to discover whether a field has the algebraic value of zero. The field may contain plus zero, minus zero, blanks, or just plain zero. A good way of testing for this condition is as follows:

```
          MCS       FIELD, 332
          BCE       ZERO, 332,.
```

assuming that the B-field of the MCS contains no valuable information, and is at least one position longer than the A-field. If the field to be tested may contain a blank in the low-order position, but significant digits in other positions, then the following instruction should precede the above instructions:

```
A      @O@, FIELD
```

11. The Halt Instruction

The important thing to remember about the halt instruction is that the operator must be able to recognize why the halt occurred, and he must have some easy way of restarting the machine where possible.

Since most installations now use the setting of the I-address register to identify a halt, the same halt instruction should not be used for more than one error condition. The halt should normally be a halt and branch, which will branch back to the restart point.

Other methods of identifying halts are to include an A- and B-address to the halt, giving an identifying number in the A- and B-address register, which the operator may display, or by using a 2-position halt with an identifying d-modifier. These halts should be followed by an unconditional branch back to the restart position.

12. The Clear Storage Instruction

Notice that the B-address register after the operation of this instruction, is set to the nearest hundreds position minus one. This is especially significant when clearing storage from a location lower than 100. The B-address register after operation will contain the address of the highest memory location on the machine — 7999 for an 8K, 15999 for a 16K, etc. This means that clear storage instructions may be chained out of the bottom of memory round into the top, and, more important, it is possible to discover the size of the machine on which your program is operating, by clearing storage from an address less than 100, and storing the B-address register.

Be careful not to branch to a single-position clear-storage instruction. It is possible to make this mistake when patching. Instructions in core may read /_ 332/_ /_ from, say, locations 501-506.

To insert instruction at this point, the /_332 may be changed to an unconditional branch, the instructions inserted, and the following sequence take place.

/_332 B 505 to return to the main program.

However, this will normally result in the neat removal of your patches from memory, and a puzzling situation at the next program testing. The correct end to the insertion is, of course, /_332 // B 507.

Remember that the clear storage and branch instruction really means 'branch and clear storage' in that the operands appear in that order.

13. The Set and Clear Word-Mark Instructions

If you wish to set word-marks in two adjacent core storage locations, it is more efficient to give the instructions;-

```
SW      FIELD
SW
```

rather than the instruction SW FIELD, FIELD-1.

In the same way, the instruction SW FIELD, FIELD-2, may be replaced by the series:-

```
SW      FIELD
CW
SW                 which uses one core location less.
```

A fully chained set or clear word-mark is sometimes useful for purposes other than actually setting or clearing word-marks. It may be used (as the CW is in the above example) to reduce the A- and B-address registers by one.

14. The Move Characters to A- or B-word mark Instruction

A very useful application of the half-chaining capabilities of this instruction occurs in clearing a field so that every position in that field contains a particular character. It is usually employed with zeroes, blanks or nines. Here is an example which will clear a field to nines down as far as the first word-mark encountered:-

```
        MCW       @9@,  FIELD
        MCW       FIELD
```

A whole series of adjacent fields may be cleared in this manner by adding fully chained MCW instructions. However, remember that Two extra chained Moves are necessary for each extra field cleared, unless that extra field is only one position in length.

This should become obvious when you consider that you are moving FIELD to FIELD-1, FIELD-1 to FIELD-2 and so on. The first word-mark encountered will stop the operation when it is detected in the B-field. The B-star will address the word-mark minus one, but the A-star will address the word-mark. This word-mark in the A-field will restrict the next chained move to one position moved.

Notice that a field may be shifted up one position to the right by the instruction MCW FIELD, FIELD+1, but may not be shifted one position left by the instruction MCW FIELD, FIELD-1.

15. The Move Characters and Suppress Zeroes Instruction

For the use of this instruction to test a field for zeroes, see under the Compare Instruction.

MCS will suppress all zeroes in a field, including the junior position. It sometimes happens that one zero is required to be printed in case that the number to be edited is zero. This can be accomplished by substituting for:

```
MCS          ACCUM, 299
```

the instructions

```
MN           ACCUM, 299
MCS
```

which ensure that one zero will be printed if the entire field is zeroes.

Notice that MCS will remove all word-marks from its B-field. More especially, the 4-character MCS instruction may be used to MCS a field to itself. If you do use it to accomplish this, then remember that the defining word-mark of the field will be cleared.

16. The Move Numeric and Move Zone Instructions

Neither of these instructions can be half-chained. A four-position MN or MZ will not change the contents of core storage in any way whatever. Do not try to clear the zone from FIELD by giving the instruction:

```
MN        FIELD
```

which will merely take the numeric part of the location called FIELD and replace it in FIELD, it will not affect the zone position of FIELD in any way whatever.

If you wish to move characters from one part of memory to another, a combination of move numeric and move zone may be used. For instance, to move three characters from FIELDA to FIELDB regardless of word-marks, the following sequence may be used:-

```
MN        FIELDA,  FIELDB
CHAIN     2
MZ        FIELDA,  FIELDB
CHAIN     2
```

17. The Load Characters to A word-mark Instruction

This instruction is useful in loading variable sized fields into an area used in a subroutine, for instance. It also provides the neatest method of exactly duplicating one area of storage into another. Let us assume that it is necessary to duplicate locations 600 through 851 into locations 1600 through 1851, including word-marks. It may be accomplished as follows:-

```
            CW       SWITCH
            BW       *+8, 600
            SW       SWITCH. 600
            SBR      89, 0
LOOP        LCA      600+X1, 1600+X1
            C        89, @251@
            BE       FIN
            SBR      89, 1+X1
            B        LOOP
FIN         BW       *+5, SWITCH
            B        *+5
            CW       1600
```

18. The Move Characters and Edit Instructions

Remember that the edit word loaded into the output area before an MCE must contain at least enough room for the characters of the field to be edited. It is possible, however, to use a control word longer than absolutely necessary, and if many fields of different length have to be edited to approximately the same format, it is a good idea to define one edit word big enough to accommodate the largest of these fields, and use it for all of them. Care must be taken not to wipe out fields to the left of the field being edited when using a larger edit word than strictly necessary. In this case it is usually advisable to edit information into the (for instance) print area from right to left. That is filling up the print area from right to left, so that any overlap will be ignored.

19. Reader, Punch and Printer Instructions

Note that it is not possible to stacker select on the read side when using combination Read/Punch or Read/Write/Punch instruction. This is why many standard assembly programs, SPS and Basic Autocoder, for instance, use the inefficient separate read and punch instructions, with stacker selection on the read side, rather than using the more efficient combination instruction.

If a read and branch, write and branch, etc., instruction is given with no word-mark following, the machine will execute the I/O instruction and either come to a process check or scan down core storage until it encounters the next word-mark, which it will take as being underneath the operation code of the next instruction to be executed.

Instruction lengths of 3 and 6 characters will of course produce an immediate process error.

Note that there is no such animal as a start-read-feed and branch or a start-punch-feed and branch. These instructions always scan down for the next word-mark in storage to find the next instruction to be executed. They act exactly like a no-operation in this respect.

Note that the B-address register after a Punch instruction stands at 181, so that the punch area, including position 181, may be immediately cleared b the sequence:-

```
P
CS
```

Likewise, a print instruction leaves the B-address register at 333 when the machine has print storage,
335 or at 355 without print storage. So that the print area may immediately be cleared by the sequence:-

```
W
CS
CS
```

If you use this method of clearing the print area, be sure to place an ORG card in front of your program which will cause your program to start at 334 or 336 as the case may be.

ORG 334 ~~344~~ or ORG 336 immediately following the CTL card.

It is very dangerous to attempt to select cards into pocket 8/2 from both reader and punch in the same program. Very careful study must be given to the timing and to the positioning of the cards before this kind of merging is attempted.

20. The Multiply Instruction

When multiplying together two numbers A, B, the number of decimal places in the product AB will equal the sum of decimal places in A and B. It is often necessary to round off after a multiply instruction. If it is important to have the correct sign on the rounded product, add five to round off before moving the zone from the junior position of the product field to the junior position of the rounded product. If one position only is to be rounded off, it is possible to use a routine as follows:-

```
MZ      PRODCT,  FIVE
A       FIVE,  PRODCT
MZ      @ @,  FIVE
MZ      PRODCT,  PRODCT-1
```

to round up.

Although the product field length must be equal to the sum of the lengths of the multiplier and the multiplicand plus one, the high order position of the product will always be zero. Be careful when editing a product field to the print area that the edit word contains enough room to accommodate all the positions of the product field, including the high-order zero.

When multiplying a variable by a fixed constant, it is often not necessary to use a multiply at all. Many multiplications may be effected by the use of an add instruction. This is particularly useful when using a machine which lacks the multiply/divide special feature.

Obviously, to multiply by any power of ten, it is only necessary to insert the appropriate number of zeroes at the right hand end of the field.

To multiply by two, a four-position add instruction should always be used.

To multiply by four, two four-position add instructions should always be used.

To multiply by eight, three four-position add instructions should always be used.

Make sure that your field is long enough to take the increased size of the number when multiplying in this fashion.

To multiply field by eleven, set a zero in field + 1 and give the instruction A FIELD, FIELD + 1.

To multiply FIELD by nine, set a zero in FIELD + 1 and give the instruction S FIELD, FIELD + 1.

There are many other possibilities of this kind for multiplying a variable by a given constant.

Be careful of the sign control when using techniques similar to the above.

21. The Divide Instruction

This is the instruction which seems to cause inexperienced programmers the most trouble. Here are a few simple rules for division. We wish to divide a number A into a number B. A is the divisor, B the dividend, and Q the quotient we expect to obtain. We must set aside a work area for our division in the following manner. Let a1 be the length of the field "A" in storage, and b1 be the length of the number "B" in storage. Also let ad be the number of decimal places in the number A, bd be the number of decimal places in the number B, and gd be the number of decimal places we require in our result Q. In this case, the length of the work area we define for our division must be equal to or greater than:

$$a1 + b1 + gd - bd + ad + 3$$

and it must contain a word-mark at the left-hand end.

I shall separate the operation of division into two steps. The second step is always the same. The first step may take one of two possibilities, depending on the value of (gd - bd + ad + 2). Let d = (gd - bd + ad + 2), then if d is zero or negative, the first step of a division should be as follows:-

(NB:- Instructions marked with an asterisk should only be included if the sign of the result is important).

```
*       ZA        B
        ZA        B+(d), C
*       MZ        B, C
        D         A, C - (b1 + d - )
```

If d is positive, then the first step should be executed as follows:

```
*       ZA        B
        S         C
        MCS       B, C - (d)
*       MZ        B, C
        D         A, C - (b1 + d - 1)
```

The second step is always:-

```
        A       @5@,  C - (a1 + 2)
*       MZ      C - (a1 + 1),  C - (a1 + 3)
```

and the result, with the correct number of decimal places, and sign if required, is in the field C, with the low order position in the location C - (a1 + 3).

The remainder will be in the locations C through C - (a1 - 1).

To take a particular example, we wish to divide the six digit field B (format XXXX.XX) by the four digit field A (format XX.XX). We want the result to two decimal places, stored in the location Q. A and B may be positive or negative, but we are only interested in the magnitude, or absolute positive value, of B/A. We define a work-area C at least fourteen positions long, and proceed as follows:-

```
        S       C
        MCS     B,  C - 4
        D       A,  C - 9
        A       @5@,  C - 6
        MCW     C - 7,  Q
```

If we required the correct sign on Q, the instructions would have been:-

```
        ZA      B
        S       C
        MCS     B,  C - 4
        MZ      B,  C
        D       A,  C - 9
        A       @5@,  C - 6
        MZ      C - 5,  C - 7
        MCW     C - 7,  Q
```

22. The Modify Address Instruction

The modify address instruction ignores word-marks in the A- and B-fields; it operates on the three characters at the A-address and the three characters at the B-address, ignoring the zone at A-1 and B-1, leaving it unchanged. When you wish to increase a number by some amount less than 100, remember not to give instructions like:-

```
        MA        @10@,  ADDR
        MA        @5@,   ADDR
```

they must always be:-

```
        MA        @010@,  ADDR
        MA        @005@,  ADDR
```

When using a literal in the A-operand of a modify address instruction, always use a three character alphabetic literal, between @ signs.

Likewise you must write

```
        MA        @S51@,  ADDR
NOT
        MA        @1251@,  ADDR
```

23. The Store A- and B-address Register Instructions

These instructions are used when you wish to store the contents of the A- or B-address registers at a particular point in the program.

They may often be used in a kind of chaining routine, where, for instance, a long series of accumulators must be set to zero. Supposing that there are one hundred ten-position accumulators sequentially in storage to be cleared, the first of these is labelled FIRST, and the last LAST. It would be possible to give the instructions:-

```
S         LAST
CHAIN     99
```

which are rather wasteful of core storage. One method, using a store register instruction would be:

```
MCW       +LAST,  *+4
S         O
SAR       *-4
C         *-11,   + FIRST
BU        *-19
```

Notice, however, that it is not possible to store both the A- and B-address registers after an instruction, as the SBR will destroy the contents of the A-register and the SAR will destroy the contents of the B-address register. In this case it is necessary to repeat the instruction, storing one register after each repeat:-

```
MCW       A, B
SAR       ADDR1
MCW       A. B
SBR       ADDR2
```

or, in case this is not possible, for instance after an add or subtract instruction, the word-mark in the field whose register setting was not stored may be found by a compare instruction, thus

```
        A        A, B
        SAR      ADDR1
        C        B
        SAR      ADDR2
```

Apart from these obvious uses, the SBR in particular has some very useful applications.

A seven-character SBR instruction will act in the normal SBR manner, except that the B-address of the SBR will fill the B-address register. Thus, if you want to store a particular address somewhere, it is very useful to be able to employ the SBR, which reduces the necessity for address constants.
In the example above, to clear 100 accumulators in storage, I used an instruction to set up the address of the first accumulator to be cleared as follows:-

```
        MCW      +LAST,  *+4
```

This could be equally well replaced by the instruction:

```
        SBR      *+4,  LAST
```

which does not neet an address constant, and does not need to worry about word-marks.

The B-address of the SBR does not have to be a symbolic address. If you wish to set an index register to, say, 10, the instruction

```
        SBR      89, 10
```

will do this for index 1.

Remember, however, that the B-address of the 7-character SBR must indicate a valid machine address for your machine.

One extremely useful application of the 7-character SBR is in incrementing index registers. To increase index register 1 by one, you need only give the instruction

```
        SBR      89,  1+X1
```

This works as follows, the contents of index 1 are added to the B-address in the normal indexing manner, and the results of this are stored back into locations 87 through 89, which are, of course, index 1.

On a 16, 000 position machine this technique may also be used to decrease an index register.

SBR 89, 15999+X1 will decrease the contents of index register one by one.

This technique may not be used for decreasing on index register on machines of less than 16, 000 positions, because an invalid address would be created within the B-address register. On such machines, a modify address instruction should be used instead.

A 4-character SBR does not alter the contents of the B-address register, so that a 7-character SBR followed by a series of 4-character SBRS will store the same address in many places. For instance, one way of setting all three indexes to zero would be as follows:-

```
SBR     89, 0
SBR     94
SBR     99
```

Be careful not to enter such a routine at any point other than the first instruction.

The most common use of the 4-character SBR instruction is as the first instruction of a subroutine. Any branch to that subroutine will contain the location of the next sequential instruction after the branch - i.e., the return address - in the B-address register. This must be stored immediately by an SBR, as the exit address after the subroutine has finished.

For example, here is a subroutine intended to skip to printer channel 1, Print a leading sequential page number, and space 2.

```
OVSUB      SBR        OVEX + 3
           CC         1
           A          *-6,  PAGE
           MCW        HEDLIN, 300
           CC         T
           MCS        PAGE,  332
           W
           CS
           CS
OVEX       B          0
```

At any point, the programmer may write BCV OVSUB, which will cause the machine to do nothing except on page overflow, where it will cause the machine to skip to a new page, and exit to the next instruction after the BCV OVSUB instruction.

24. The Branch if Bit Equal Instruction

This instruction is part of the column binary feature. Apart from its use in testing and coding binary bit structures, it may have extensive application in testing for a range of valid codes in a particular location.

The most obvious use of this instruction is to test for an odd or even number, thus

```
        BBE     ODD, NUMBER, 1
EVEN    ___
        ___
```

When using the BBE to test for a valid range of codes, great economies in core storage can be made, but great care must be taken. For instance, if a particular card column must contain one of the odd numbers - 9 through 9 (X-overpunch when negative, no zone when positive), it could be effected by a string of ten BCE instructions or as follows:

```
        MZ      *-6, *+8
        BBE     ERROR, CODE,
        BBE     OK, CODE, 1
ERROR   H
```

However, this would let through a series of special characters, such as the full stop, number sign, etc., which also contain a 1-bit. Make sure that your tests are necessary and sufficient. A full series of tests for the above would be as follows:-

```
        MZ      *-6, *+8
        BBE     ERROR, CODE,
        BBE     TEST1, CODE, 8
        BBE     OK, CODE, 1
ERROR   H
TEST1   BCE     OK, CODE, 9
        BCE     OK, CODE, R
        B       ERROR
OK      --
```

which is still more economical than ten BCE instructions.

Notice the difficulty illustrated above in testing for an A-bit. There is no punched character which can represent a solitary A-bit, and the correct d-modifier has to be created by programming. If you are pushed for storage, this can of course be done during an overlay.

Be careful when testing a code for negative characteristics only. In other words, if the code contains a particular bit it is not valid. Suppose that a particular code must be one of the digits 1 through 7, and must not have a zone. The series:-

```
        BBE       ERROR, CODE, 8
        BBE       ERROR, CODE, &
```

would seem to be an accurate test of the validity of the code.

However, we have implied that a blank would be an error in this position, and a blank would easily pass the tests shown above. When using BBE to test for invalid characters, be careful to ascertain how a blank will be treated by the series of test instructions that you devise.

Note also that the instruction:

```
        BBE       LOOP, CODE, 3
```

for instance, will branch to loop if the code contains either a two-bit or a one bit or both.

Characters in the d-modifier which consist of more than one bit cause the instruction to test for the presence of any combination of those bits in the B-field.

Note also that a zero has the bit structure (8-2).

SECTION II

1. Address Modification and Index Registers

A considerable problem is created by the 3-character address structure used by the 1401. Only in limited circumstances can one increase an address by a specified amount just by adding that quantity to it. Addresses greater than 3999 have a zone on the junior position which is meaningful, in that it determines whether that address refers to the first, second, third or fourth sector of storage - defining a sector as 4,000 memory positions - so that a subtract operation, which produces a zone, and may act as an addition if the address lies between 8,000 and 11,999, is obviously out of the question.

There are four main methods of address modification. The first of these and the safest, is to employ the special Modify address instruction. This is safe whether you wish to effect addition or subtraction. Although subtraction can only be possible when you wish to subtract a certain fixed constant from an address (it must be represented in its 16,000 complement form). Thus, decrease ADDR by 5, can be effected as follows:-

```
        MA        @I9E@,  ADDR
```

(where I9E is the address equivalent of 15,995).

It is not possible to subtract one variable address from another without using some special routine to obtain the 16,000 complement of one of the addresses. Unless, of course, both addresses must be less than ≠00 (1,000).

The second method is to use an add instruction. This has a disadvantage against the modify address in that the address to be modified must have a word-mark under the left-hand character, if the result is to be greater than 1,000 and in any case the add instruction will only work for addresses less than 4,000.

If the word-mark under the left-hand position is missing, the zone on that position which indicates which band of 1,000 is to be addressed within the sector will be cleared, and the formation of an overflow bit on passing from one band to the next will not occur.

The third method is to use some instruction followed by a store-register instruction. For instance, to decrease an address by 1, the following routine could be used, assuming that there is a word mark under the left hand position of ADDR.

```
MCW      ADDR,  *+4
MN       O
SBR      ADDR
```

Or to decrease it by six

```
MCW      ADDR,  *+4
MN       O
CHAIN    5
SBR      ADDR
```

This technique is usually inferior to the modify address instruction.

The fourth method can only be used with index registers, and is explained under the store B-address register instruction. One method of adding a number N to an index register is to use the instruction

```
SBR      89,  N + X1
```

and on a 16K machine only of subtracting a number N from an index register:

```
SBR      89, (16000 - N) + X1.
```

This, of course, could be used to modify an address as follows:-

```
MCW      ADDR,  89
SBR      ADDR,  4 +X 1
```

to add 4 to ADDR.

This is obviously inferior to

```
        MA        @004@,  ADDR
```

When you require to set an address to a certain value, it is nearly always preferable to use a store B-address register instruction. Thus to set the Address ADDR to the address of A, use

```
        SBR       ADDR,  A
```

The main exception to this is when you wish to set an index register to zero.

A word-mark should nearly always be set in locations 87, 92, 97 and the index register cleared by the instructions

```
        S          90
        S          95
        S         100
```

respectively. Notice that S 89 is not valid as it will produce a zone on position 89 and probably set XR 1 to the address 12,000. Also all XRs may be cleared simultaneously by the instructions

```
        S          100
        CHAIN      2
```

Note that the contents of any indexed address must be valid both before and after indexing. Thus, on an 8K machine, the instruction

```
        SBR       89, 15999 + X1
```

is not possible, even if XR 1 contains, for instance, 002 which will result after indexing in 001 in the B-address register.

It is, however, possible on a machine of less than 16K to have an invalid address in the XR itself.

Thus on an 8K machine

```
          MCW       10, 80 + X1
```

will, if locations 87 through 89 contain for instance I9I, work perfectly well and produce the same result as the instruction

```
          MCW       10, 79
```

under those circumstances.

2. Subroutines

Subroutines should be used wherever possible. They save coding effort, make program design more logical and elegant, and often enable the programmer to avoid the use of program switches, which are a common source of confusion and error.

If a particular sequence occurs more than once within a program, it is normally of advantage to write that series of instructions only once, as a subroutine, and to branch to it at the points where the routine formerly occurred in the main program.

If a routine occurs several times during a program with slight variations, this routine may be coded as a subroutine embodying all the variations. When the subroutine is called from the main program, the branch to subroutine should be followed by a parameter or parameters which indicate which variation of the subroutine is required at this particular time.

The Store B-address register instruction allows a subroutine to return to the correct point in the main program, and may also allow the subroutine to examine the memory at the point immediately following the branch to subroutine.

Any branch to subroutine, or any sequence of instructions branching to a subroutine and producing variations in the operation of the subroutine will be referred to from now on as a 'call' routine.

Subroutines may be roughly divided into four classes:

a) Normal entry exit subroutines.

b) Subroutines which analyze the preceding instruction to produce same variation in their own operation.

c) Subroutines which analyze a series of parameters following the branch instruction to produce some variation in their own operation.

d)   The in-line subroutine.

Examples of types (a) through (c) follow. Type (d) is a somewhat unusual application, and will be explained separately at the end of this section.

Let us assume that the instructions A@1@, COUNT and S@1@, COUNT occur many times in a program. They occur enough times to make it worth while to make a subroutine of each instruction. The unusual case of making a subroutine out of one instruction is chosen for the sake of simplicity.

Then, using type (a) subroutines, every time A@1@, COUNT occurs it is replaced by the instruction B SUBA, and every time S@1@, COUNT occurs, it is replaced by the instruction B SUBS. Two subroutines are then inserted into the program as follows:-

```
SUBA        SBR       SUBAX + 3
            A         @1@,  COUNT
SUBAX       B         O
SUBS        SBR       SUBSX + 3
            S         @1@, COUNT
SUBSX       B         O
```

To take a particular instance, SUBA may be assembled to occur in locations 501 through 515. The location of the field COUNT may be 2304, and a constant 1 may be available in location 3901.

SUBA will assemble as follows:-

```
H515A101L04B000
!   !      !
501 505    512
```

If one of the calls for SUBA occurs in the main program, in location 901 through 904, say, it will assemble as follows:-

```
B 5 0 1 N
!       !       where N is the next op. code.
9 0 1   9 0 5
```

as soon as the machine reaches location 901, it will branch to 501. The B address register after the branch will contain 905. This address will be stored in locations 513 through 515. The machine will then execute the add one to count instruction and arrive at the branch, which will now be in the form B 905.

The machine will therefore exit to the next instruction after the call routine.

Using type (b) subroutines, only one subroutine need be written. Every time the instruction A @1@, COUNT occurs, it will be replaced by the sequence:-

```
NOP
DC          @A@
B           SUB
```

and S @1@, COUNT will be replaced by:

```
NOP
DC          @S@
B           SUB
```

and the subroutine will take this form:-

```
SUB         SBR         EXIT + 3
            SBR         INSTR + 3
            MA          @19E@, INSTR + 3
INSTR       MCW         0, * + 1
            NOP         @1@, COUNT
EXIT        B           0
```

this might be assembled in locations 501 through 533 as follows:

```
H533H519  #  104519M000523N101L04B000
!   !     !        !      !      !
501 505   509      516    523    530
```

and a particular call for this routine, specifying subtract one from acount might be assembled in locations 901 through 906 as follows:

```
N S B 5 0 1 N
-   -       -
!   !       !
901 903     907
```

This will operate as follows: as soon as the branch instruction in 903 is reached, the machine will branch to 501, the B-address register at this point will contain 907. This B-address register will then be stored in locations 531 through 533, and in locations 517 through 519. The modify address instruction will then reduce the address in 517 through 519 by five. The subroutine will then look like this:

```
H 5 3 3 H 5 1 9 # 1 0 4 5 1 9 M 9 0 2 5 2 3 N 1 0 1 L 0 4 B 9 0 7
-       -       -             -             -             -
                              !
                              *
```

The operation code of the instruction in 523 will then be changed to a subtract, it will then be executed as specified, and the branch instruction in 530 will cause the subroutine to exit to the correct location in the main program.

Using a type (c) subroutine, the same procedure can be effected, using again only one subroutine. Every time S @1@, COUNT occurs, it is replaced by the instructions:-

```
B       SUB
S
```

likewise A @1@, COUNT will be replaced by

```
B       SUB
A
```

The subroutine will then be written as follows:

```
SUB     SBR     89
MCW     MOW     0 + X1, * + 1
        NOP     @1@, COUNT
        B       1 + X1
```

which might assemble as follows in locations 501 through 522

```
    H 0 8 9 M 0 ‡ 0 5 1 2 N 1 0 1 L 0 4 B 0 ‡ 1
```

and a particular call for the subroutine, specifying Add one to count, might be assembled in locations 901 through 905 as follows:-

```
    B 5 0 1 A  N
    !       !  !
    901     905 906
```

This will operate as follows:-

From location 901, the computer will branch to location 501. At this point the B-address register will contain 905. This address will be stored in index register 1 (positions 87 through 89). Then the contents of location 0 + X 1, i.e., the contents of location 905 will be moved to the op. code of the instruction which increases or decreases COUNT, this instruction will be executed and the machine will branch to location (1 modified by index 1) = location 906, which is the correct re-entry point.

The above techniques should be understood to understand the remainder of this discussion.

It should be noted that time is lost when a subroutine is used. Every time a subroutine is entered, instead of re-writing the subroutine in the main program, the execution time of the branch to subroutine, store B-address register, and exit branch is added to the natural execution time of the routine. This is not normally very significant, but with more complicated routines with several parameters, a large number of extra instructions may be executed on each entry. The time log can become significant. An extreme example of this is the program produced by the 1401 Fortran processor. Every operation in the object program is compiled as a subroutine entry with parameters, and execution speed is exceedingly slow. This is the price that must be paid for the tremendous programming convenience of the FORTRAN language.

It is easy to decide whether it is worth while from the point of view of storage space saved to turn a repeated routine into a subroutine. It may be computed as follows:-

Let N be the number of times the routine is used. Let Ln be the length of the series of instructions within the subroutine, excluding linkage instructions. Let Lc be the length of the call instructions for the subroutine. Let A be the amount of storage saved by turning the given routine into a subroutine, and Ls be the length of the linkage instructions within the subroutine.

Then $A = NLn - NLc - Ls - Ln$

and the number of times a given group of instructions must occur to make it worth while to make them a subroutine:

$$= \frac{Ls + Ln + 1}{Ln - Lc}$$

As an example of the value of subroutines in simplifying the logic of a program, two block diagrams are enclosed. The problem to be solved is one which occurs very frequently in commercial applications -- that of producing a tabulation. That is, a series of fields in input cards must be accumulated at various levels, and totals must be printed out to correspond to a series of control fields in the card.

Normally these totals are within a hierarchy. For example, a particular application may involve a company's sales in different districts. And a breakdown of the sales may be required for each country in which the company operates. This total should be further broken down into a series of separate totals for each region within the country. And each region total may be broken down to give a total for each city within a region.

The normal way of solving this problem is to sort the input cards into order by city number within region number within country number. Minor total are printed at the end of each city, an intermediate level total at the end of each region -- which will be the sum of all the minor totals printed out within that region, and a major total is printed at the end of each country, which will be equal to the sum of all the intermediate region totals within that country. A final grand total also might be required.

Notice that the machihe must test for a change in the highest level of total first, and if there is a change, it must print out first the minor total it is accumulating, followed by the intermediate, followed by the major, etc.

A problem is created by the fact that the control fields must be tested in the order major, intermediate, minor, but when a change is detected, totals should be printed in the, order minor, intermediate, major.

This is usually accomplished by means of a series of program switches, which make for rather complex logic, and are very error-prone. A possible solution of the standard tabulation problem using switches is shown in figure 1.

If one uses subroutines in the coding instead of switches, the solution becomes much more elegant, straightforward and logical. Figure 2 is the block diagram of the same problem using subroutines instead of switches. The advantages of the second method should be obvious. Note particularly, that the test for a major change, for instance, can be very simply coded as follows:-

```
C           CARD,  MAJOR
BH          SEQERR
BH          SUB1
C           CARD,  INTER
```

The only test which cannot be coded by a direct branch if indicator on instruction straight to the subroutine is the last card test which should be coded as follows:-

```
BLC         * + 5
R           LOOP
B           SUB1
MCW         FINAL,  PRINT
```

etc.

This is a good example of the use of type (a) subroutines to simplify the logic of the program, as opposed to their normal use in cutting down the amount of storage used and the amount of coding effort by preventing the repetition of frequently occurring sequences of instructions in the main program.

A typical example of a type (b) subroutine is the tape error routine. This is a subroutine which is called immediately after a tape read, tape write, or write tape mark instruction to test for a tape transmission error, Normally, if a read tape instruction was given before the entry to the subroutine, it should backspace and re-read many times, and than halt if the tape error is persistent. If the preceding instruction was an output instruction, the routine should backspace and rewrite a few times, then erase the bad section of tape, and try again further on. Thus the subroutine has two different sets of actions depending upon whether it was entered after a write or read tape instruction, it must analyse the preceding instruction.

Here is a simplified tape error routine, which does not check for noise records. It is called by an unconditional branch immediately following the tape instruction, thus:-

```
RT          1,  INPUT
B           ERSUB
                        or
WTW         2,  OUTPUT
B           ERSUB
                        or
WTM         1
B           ERSUB
                        etc.
```

Example of a Type (b) Subroutine   Tape Error Routine

```
ERSUB       SBR         EXIT + 3
            C
            C
            SBR         * + 4
            LCA         0, INSTR + 7
            S           ERCNT
            S
ERB         BER         EXIT + 4
            LCA         NOP. INSTR + 2
            LCA         NOP - 1
EXIT        B           0
            C           INSTR + 7
            SAR         * + 11
            MA          FOUR  * + 4
            MN          0, * + 4
BSP         BSP         1
            BCE         ERA. INSTR + 7, W
            BCE         ERHT. ERCNT - 1, 2
INSTR       RT          0, 0
            A           * - 6, ERCNT
            B           ERB
ERA         BCE         * + 5, ERCNT. C
            B           INSTR
            BCE         ERHT. ERSKP - 1, 1
            MN          BSP + 3, * + 4
            SKP         1
            S           ERCNT
            A           * - 6, ERSKP
            B           INSTR
ERHT        H           INSTR
NOP         DCW         @N1@
ERSKP       DCW         #2
ERCNT       DCW         #2
FOUR        DCW         @004@
```

Notice the use of the chained compares after the initial store register instruction, in order to scan down to the right hand character of the preceding instruction. Remember that a compare will move the registers down as far as the first word-mark which is encountered in either field. In this case, the setting of the registers after the SBR will be:-

A - The next instruction after the branch
B - The op. code of EXIT

both these should contain word-marks, so that the compare will chain down the registers one position, and the registers will be as follows:-

A - The last digit of the branch instruction which called the subroutine.
B - The last digit of the 4-character load instruction before exit.

The second chained compare takes over the registers at this point. Both A- and B-fields are 4-characters long, and their setting after this compare is:-

A - The last character of the instruction LCA NOP, INSTR + 2
B - The character immediately preceding the branch to subroutine instruction - i.e., the last position of the tape instruction which preceded the subroutine entry.

This setting of the B-address is what we have been aiming at, we may now pull out the tape instruction and use it at the appropriate place in the subroutine.

A type (c) subroutine is usually used in conjunction with at least one index register, and the parameters following the branch are usually a string of addresses. It will be noted that the autocoder call macro with parameters works in this way, for instance, the autocoder instruction:-

```
    CALL      SUBR, FLDA, FLDB, FLDC
```

will assemble as

```
          B        SUBR
          DSA      FLDA
          DSA      FLDB
          DSA      FLDC
```

or

```
B         SUBR
DCW       FLDA
DCW       FLDB
DCW       FLDC
```

depending on what version of autocoder you have. Both the above routines cause exactly the same things to be assembled.

There is no reason why ones own subroutines should not also take this form where necessary.

Consider the following examples:-

It is necessary to print a set of totals, add them to the next level, and clear them. The totals are stored in sequence in storage as follows:-

```
               etc.
MAJOR 2        DCW       # N1
MAJOR 1        DCW       # N2
               etc.
INTER  3       DCW       # N3
INTER  2       DCW       # N4
               etc.
MINOR 2        DCW       # N5
MINOR 1        DCW       # N6
```

and so on.

It will be seen that every total may be of any length less than 10. Major totals are stored sequentially before intermediate totals immediately before minor totals etc. Totals of the same amount are defined in the same sequence number of different totals within each level.

These totals are to be printed 10 positions apart, starting at print position 10.

These specifications allow of considerable flexibility in the relative sizes of the totals.

The call routine is as follows:-

```
B      SUBR       Where LEVELA is the address
DSA    LEVELA     of the first total to be printed, LEVELB
DSA    LEVELB     the address of the total into which the
                  LEVELA total must be rolled.
```

<u>The subroutine is as follows:-</u>

```
SUBR      SBR      89
          MCW      2 + X1, 94
          MCW      5 + X1, 99
          SBR      PRINT + 6, 210
PRINT     MCS      0 + X2, 0
          A        0 + X2, 0 + X3
          SBR      99
          S        0 + X2
          SBR      94
          C        94, 5 + X1
          BE       EXIT
          MA       @010@, PRINT + 6
          B        PRINT
EXIT      W
          CS
          CS
          B        6 + X1
```

This subroutine could be applied to the coding of the sample tabulation in figure 2.

In this case, the entire program could be coded as follows: Assume for the sake of simplicity that there are only two types of total A, and B. Card format might be as follows:-

Card Columns	Contents
1 - 3	Country (major)
4 - 6	Region (Inter)
7 - 9	City (minor)
10 - 15	Amount B
16 - 20	Amount A

The coding might be:-

```
                ORG       334
                DCW       # 1
DUMMY           DCW       # 1
                DCW       # 10
FINAL           DCW       # 9
                DCW       # 9
INTER           DCW       # 8
                DCW       # 8
MINOR           DCW       # 7
PAGE            DCW       # 3
COUNTER         DCW       # 3
REGION          DCW       # 3
CITY            DCW       # 3
START           S         PAGE
                CHAIN     6
                R         OVSUB
                SW        10, 16
                MCW       9, CITY
                CHAIN     2
LOOP            C         3, COUNTER
                BH        SQERR
                BU        SUB1
                C         6, REGION
                BH        SQERR
                BU        SUB2
                C         9, CITY
                BH        SQERR
                BU        SUB3
                A         20, MINOR
                A
                BLC       * + 5
                R         LOOP
                B         SUBR
                DSA       FINAL
                DSA       DUMMY
                H         * - 3
SUB 3           SBR       EXIT 3 + 3
                BCV       OVSUB
                MCS       CITY, 290
                B         SUBR
                DSA       MINOR
                DSA       INTER
                MCW       9, CITY
```

```
EXIT 3          B          0
SUB  2          SBR        EXIT 2 + 3
                B          SUB3
                MCS        REGION, 285
                B          SUBR
                DSA        INTER
                DSA        MAJOR
                MCW        6, REGION
EXIT 2          B          0
SUB 1           SBR        EXIT 1 + 3
                B          SUB 2
                MCS        COUNTR, 280
                B          SUBR
                DSA        MAJOR
                DSA        FINAL
                MCW        3, COUNTR
EXIT 1          B          0
OVSUB           SBR        OVEX + 3
                CC         1
                A          * - 6, PAGE
                CC         T
                MCS        PAGE, 332
                W
                CS
OVEX            B          0
                END        START
```

The type (d) subroutine offers an interesting method of avoiding the difficulties of a count-controlled loop. If the same sequence of instructions must be repeated several times one after the other in the main program, the normal way of doing this is as follows:

```
                MCW        @1@, COUNT
LOOP             -
                 -
                 -
                 -
                C          COUNT, @N@
                BE         OUT
                A          * - 6, COUNT
                B          LOOP
OUT              -
                 -
```

Where N is the number of times the routine is to be executed. A useful way of producing this effect using subroutine techniques is as follows. To execute a loop twice, give the instructions

```
                     -
                     B           * + 1
LOOP                 SBR         SWITCH + 3
                     -
                     -
SWITCH               B           0
OUT                  -
```

In this case, the computer will execute the B * + 1 instruction initially, and, as is normal, the B-address register will contain the address of the next sequential instruction after the branch. In this case, it will be the address of LOOP. This address will now be stored in the SWITCH instruction, and the routine to be repeated will be executed. After this, the SWITCH instruction will be executed. The B-address register after the branch will contain the address of the next sequential instruction after the branch, which will be the address of OUT. This address will then be stored into the switch instruction, which will now say B OUT. The routine to be repeated will now be executed for the second time, and at the end the switch instruction will cause an exit to OUT.

If it is desirable to execute the loop more than twice, extra iterations may be obtained by adding a series of B LOOP instructions after the switch instruction.

Thus the following routine will execute the loop four times:-

```
                     B           * + 1
LOOP                 SBR         SWITCH + 3
                     -
                     -
                     -
SWITCH               B           0
                     B           LOOP
                     B           LOOP
OUT                  -
                     -
```

This method has the added advantage that small variations in the operation of the routine to be repeated may be accomplished, by inserting modification instructions in between the B LOOP instruction.

Thus, if a particular routine must be executed four times in succession, with index register settings of 0, 4, 9 and 20 respectively on the different iterations, this may be accomplished as follows:-

```
                 SBR          89, 0
                 B            * + 1
LOOP             SBR          SWITCH + 3
                 -
                 -
                 -
                 SBR          89, 4 + X 1
SWITCH           B            0
                 SBR          89, 1 + X 1
                 B            LOOP
                 SBR          89, 7 + X 1
                 B            LOOP
OUT              -
                 -
                 -
```

I am indebted to Mr. C. Purvis of IBM (U.K.) for bringing to my attention this extremely elegant technique.

Subroutines may also be used on machines which lack the store B-address register instruction. In such a case they will be used less often, as the call routine becomes longer.

```
                 MCW          DSA 1, EXIT + 3
                 B            SUBR
ENTRY 1          -
                 -
                 -
                 -
DSA 1            DSA          ENTRY 1
```

Usually a slightly more acceptable method, which avoids the necessity of defining a whole series of addresses with DSAs is to use the instruction:

```
                        MCW             * - 3, ADDR
                        B               SUBR
```

and within the subroutine, to have the sequence:-

```
    SUBR                A               @8@, ADDR
                        MCW             ADDR, EXIT + 3
                        -
                        -
                        -
    EXIT                B               0
    ADDR                DCW             # 3
```

3. Magnetic Tapes

There are many input/output packages available which will test for tape errors and for noise records, and which will also handle the unblocking of blocked tape records. The IOCS system is the most comprehensive of these. However, such systems often have disadvantages in that they prevent the programmer from kaing the best use of his machine, either because the routines are very bulky, or because they do not allow him to take into account critical timing considerations.

It is a very simple matter to program the unblocking of a tape record. Normally an index will be set aside to keep the place within the record. And assuming a fixed length blocked record, routines like the following can be used:-

```
LOOP 1          SBR            89, 0
                RT             1, INPUT
                B              ERSUB
                BEF            END
LOOP 2          C              RECORD, @PADDING@
                BE             LOOP 1
                -
                -
                -  BCE         LOOP1, ENDRES, ≢
                ~~C~~              ~~89, @ MAX @~~
                ~~BE~~             ~~LOOP 1~~
                SBR            89, L + X1
                B              LOOP 2
```

where L is the length of the individual record within the block, PADDING is a constant containing whatever padding character is being employed, and MAX is a three character machine address equivalent to L(B - 1) where B is the blocking factor. The label RECORD will appear as part of a define area statement which will be as follows:-

```
INPUT           DA             BXL, X 1, G
RECORD                         X, Y
```

One particular instance where a large saving in running time may be made is when a card to tape operation must be performed using 7330 tapes. Even with early card read, the speed of the card to tape may drop considerably, because every time a tape record is written, the reader clutch point is missed, and the reader shows to 600 cards per minute. A constant speed of 800 cards per minute may be attained by the careful use of the start read feed instruction.

The principle of this is to ensure that the tape record is written during the read start time. This may be accomplished, for a card to tape using unblocked card image records as follows:-

```
               R
               LCA          GPMK, 81
     LOOP      SRF
               WT           1, 1
               R            ERSUB
               B            LOOP
```

However, single card image records have several disadvantages - mainly that they take up a large amount of room on the tape, and are inefficient to read later on. A blocked record will only cause the reader to slow down every Bth Card, where B is the blocking factor.

This time loss can still be significant, and, under some circumstances fill reading speed can still be maintained. There are 21 ms. available of read start time, during which a 7330 tape unit can write a 300 character record. Providing the blocks of data on tape are kept to a maximum of 300 characters, this technique may still be used.

If a test is made for tape write error immediately after giving the write tape instruction, the processor is unnecessarily interlocked for 8.7 ms. Most of the normal read cycle processing time, so that the reader may be slowed down if the movement of data from the card area to the tape area, and any checking that may be performed use up more than 1.3 ms. This will normally be the case.

To gain full efficiency on a card/tape program with 7330 tapes, it is advisable to use two separate output areas, alternately, and to test for tape error immediately before writing a new tape record instead of immediately after the tape write.

This means using this kind of routine. Assume that cards are to be put on to tape in card image form, blocked 3 toa tape record.

```
AREA  1         DA          3 X 80, X 1, G
CARD  1                     1, 80
AREA  2         DA          3 X 80, X 2, G
CARD  2                     1, 80
START           SW          1
                R
                SBR         89, 0
                SBR         94
                CW          SWITCH
LOOP            Insert here any checking etc.
                BW          RTN 2, SWITCH
                MCW         80, CARD 1
                BCE         WRITE 1, 87, 1
                SBR         89, 80 + X 1
                B           LOOP 1
WRITE 1         BER         ERR 2
                SRF
                WT          1, AREA 1 + X 0
                SBR         89, 0
                SW          SWITCH
LOOP 1          R           LOOP
RTN  2          MCW         80, CARD2
                BCE         WRITE 2, 92, 1
                SBR         94, 80 + X 2
                B           LOOP 1
WRITE 2         BER         ERR 1
                SRF
                WT          1, AREA 2 + X 0
                SBR         94, 0
                CW          SWITCH
                B           LOOP 1
```

ERR 2 will try to rewrite area 2 and return to WRITE 1 + 5 after a successful operation.

ERR 1 will try to rewrite area 1 and return to WRITE 2 + 5 after a successful operation.

The same limitation applies to 729 II tape units, with a maximum record length of 600 characters, and to 729 IVs with a maximum record length of about 1,000 characters.

4. Tables and Table-Look-Up

- Not too good.

It is often valuable to be able to compute a direct address to find ones position in a table in storage. Suppose that there are four hundred five-digit counters in storage, and an amount must be added into one of those counters, depending upon a code which will vary between one and four hundred in value, then the obvious approach is to multiply the code by five, and place the result in an index register to determine the address to which you wish to add. However, the result of your multiplication will be a four digit address, which must be converted to three digits to reference the correct core storage location. Here is a useful routine to convert a five-digit numeric address to a three digit machine address. Assume that your numeric address is in ADDR, with no zone on the junior position.

```
                     MCW          ADDR,  WORK # 5
LOOP                 S            @1@,  WORK - 3
                     BM           OUT,  WORK - 3
                     MA           @ ‡ 00@,  WORK
                     B            LOOP
OUT                  -
                     -
                     -
```

Alternatively:-

```
                     LCA          ADDR,  89
LOOP                 S            @1@,  86
                     BM           OUT,  86
                     SBR          89,  1000 + X1
                     B            LOOP
OUT                  SW           87
                     -
                     -
```

If a short table of fields must be selected on the basis of a series of 'random' codes, this can sometimes be best effected by a string of 7-character store B-address register instructions. For instance, supposing that a particular note must be printed alongside a number, depending on the card code in column 80, say. The line-up is as follows:

Column	Print
7	X X
A	X Y
4	Z Z
N	Z Y
any other	blank

and the instructions to select the appropriate entry from the table:

```
TABLE           DCW             @ZY@
                DCW             @ZZ@
                DCW             @XY@
                DCW             @XX@
```

are

```
                SBR             89, 0
                BCE             LOOP, 80, 7
                BCE             LOOP + 7, 80, A
                BCE             LOOP + 14, 80, 4
                BCE             LOOP + 21, 80, N
                B               LOOP + 28
LOOP            SBR             89, 2 + X1
                SBR             89, 2 + X1
                SBR             89, 2 + X1
                MCW             TABLE + X1, PRINT
                -
                -
```

Consider the normal table look-up situation where a short card code is used to select a particular factor in a calculation. The factor may be reduced to a code to save space in the card, or for various other reasons.

Take the example where a 3-digit code in the card will decide which of a possible 500 seven-digit factors will be used in a calculation. All the codes are spread over the range 000-999, so that there are gaps, and no direct addressing system is possible.

A large table will be created in storage, containing all possible codes, together with their corresponding factors, looking something like this:-

0 1 0 6 7 1 4 5 7 6 0 4 3 3 1 3 1 3 1 2 1 9 0 6 6 6 6 6 6 7

and so on. If some codes occurred more frequently than others, it might be of advantage to organize the table with the most frequently occurring codes at the beginning and use a step by step search.

Thus, uspposing that the card code is in positions 78 through 80 of the card, that the first code in the table is labelled CODE, and its corresponding factor FACTOR, a routine like the one shown below might be used:

```
                    SBR             89, 0
LOOP                C               80, CODE + X1
                    BE              FOUND
                    C               89, @99 ⧧ @
                    BE              ERROR
                    SBR             89, 10 + X1
                    B               LOOP
FOUND               A               FACTOR + X 1, AMOUNT
                    -
                    -
```

But if the codes have a more or less random occurrence, it might be of more advantage to organise the table sequentially by code in storage, and search in the following manner:-

```
                    C               80, CODE
                    BH              ERROR
                    C               80, CODE + 4990
                    BL              ERROR
                    SBR             89, 0
LOOP 1              C               80, CODE + 490 + X1
                    BH              NEXT
                    SBR             89, 500 + X1
                    B               LOOP 1
NEXT                C               80, CODE + X1
                    BE              FOUND
                    C               89, @99 ⧧ @
                    BE              ERROR
                    SBR             89, 10 + X1
                    B               NEXT
FOUND               A               FACTOR + X1, AMOUNT
```

Such a splitting down of the table into sections will usually use more storage, but be much faster in execution.

~~It is very difficult to use the famous 'binary search' routine on the 1401 because of the slow speed of the divide instruction, and the difficulty encountered with addresses greater than 999.~~

Tables may also be used to great advantage in solving complicated logical tests or, for instance, a series of card codes.

If you are given the kind of problem where a valid code combination is asked for, i.e., valid card codes might be

	X	on	80,	2, 3 or 4 in column 79
or	4	on	80,	1 in column 79 and 5 in column 3
or	2	on	80,	0 through 9 in column 79 and 2 in column 1.

and so on, it is usually a good idea to design a table which embodies the information that is required, and write a routine which will analyze this table, rather than writing out in full the complicated series of character, zone and number range tests that we required. Be careful in this case, though, as the table plus analyzing routine will normally be much slower in execution than the whole series of interconnected tests written out in full. The latter, of course, will eat up much more storage space.

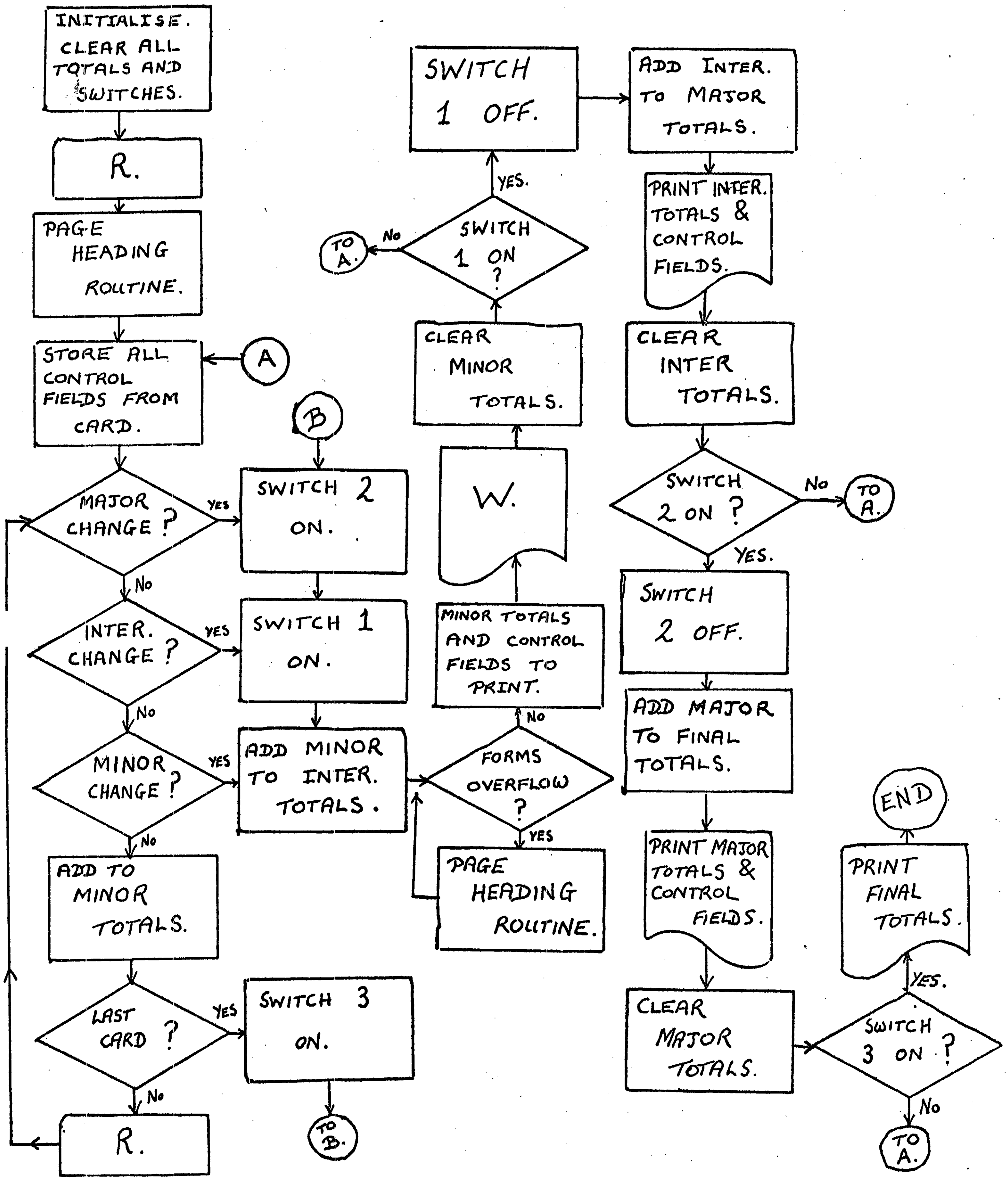

FIGURE 1. SKELETON TABULATION USING SWITCHES.

Can be done better.

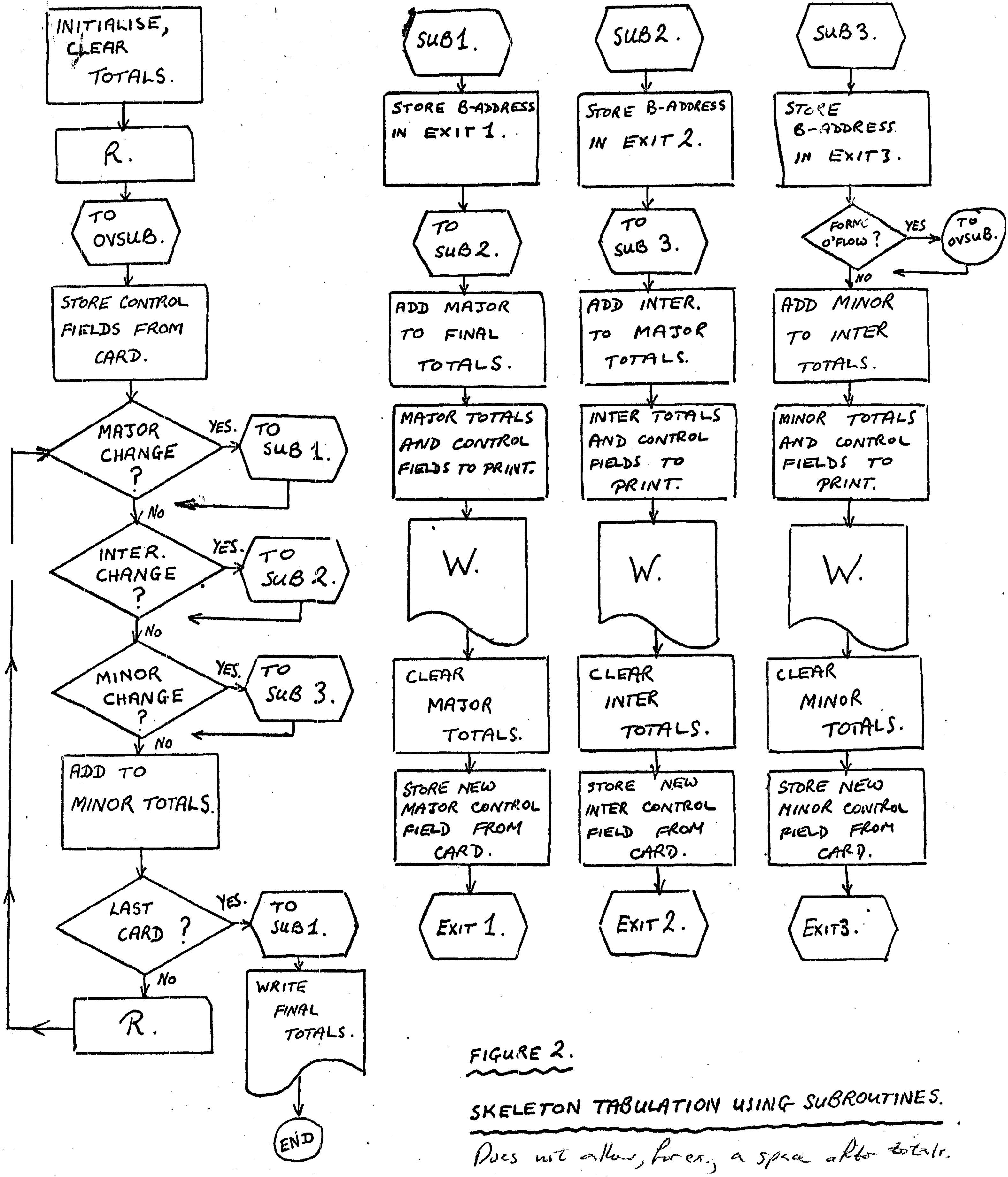
INITIALISE, CLEAR TOTALS.
R.
TO OVSUB.
STORE CONTROL FIELDS FROM CARD.
MAJOR CHANGE ?
YES.
TO SUB 1.
No
INTER. CHANGE ?
YES.
TO SUB 2.
No
MINOR CHANGE ?
YES.
TO SUB 3.
No
ADD TO MINOR TOTALS.
LAST CARD ?
YES.
TO SUB1.
No
R.
WRITE FINAL TOTALS.
END
SUB1.
STORE B-ADDRESS IN EXIT 1.
TO SUB 2.
ADD MAJOR TO FINAL TOTALS.
MAJOR TOTALS AND CONTROL FIELDS TO PRINT.
W.
CLEAR MAJOR TOTALS.
STORE NEW MAJOR CONTROL FIELD FROM CARD.
EXIT 1.
SUB 2.
STORE B-ADDRESS IN EXIT 2.
TO SUB 3.
ADD INTER. TO MAJOR TOTALS.
INTER TOTALS AND CONTROL FIELDS TO PRINT.
W.
CLEAR INTER TOTALS.
STORE NEW INTER CONTROL FIELD FROM CARD.
EXIT 2.
SUB 3.
STORE B-ADDRESS IN EXIT 3.
FORM O'FLOW ?
YES
TO OVSUB.
NO
ADD MINOR TO INTER TOTALS.
MINOR TOTALS AND CONTROL FIELDS TO PRINT.
W.
CLEAR MINOR TOTALS.
STORE NEW MINOR CONTROL FIELD FROM CARD.
EXIT3.

FIGURE 2.

SKELETON TABULATION USING SUBROUTINES.

Does not allow, for ex., a space after totals.