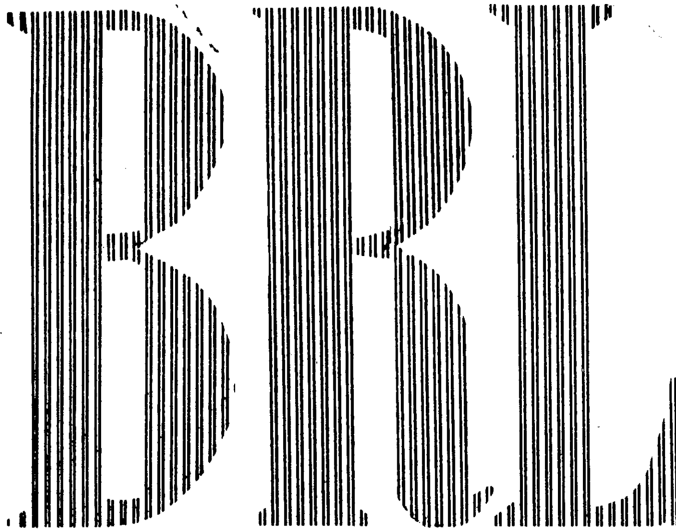


REPORT No. 997

OCTOBER 1956

Programming And Coding For ORDVAC

TADEUSZ LESER

MICHAEL ROMANELLI

DEPARTMENT OF THE ARMY PROJECT No. 5B0306002
ORDNANCE RESEARCH AND DEVELOPMENT PROJECT No. TB3-0007

BALLISTIC RESEARCH LABORATORIES

ABERDEEN PROVING GROUND, MARYLAND

Destroy when no longer needed. DO NOT RETURN

BALLISTIC RESEARCH LABORATORIES

REPORT NO. 997

OCTOBER 1956

PROGRAMMING AND CODING FOR ORDVAC

Tadeusz Leser

Michael Romanelli

Department of the Army Project No. 5B0306002
Ordnance Research and Development Project No. TB3-0007

ABERDEEN PROVING GROUND, MARYLAND

TABLE OF CONTENTS

		Page
ABSTRACT		3
PREFACE		5
ACKNOWLEDGEMENT		6
I.	BINARY AND SEXADECIMAL ARITHMETIC NECESSARY FOR ORDVAC	7
II.	INTRODUCTORY DESCRIPTION OF ORDVAC OPERATIONS. Flow Charts. Coding. Short List of Symbols.	25
III.	CARD PUNCHING AND TRANSCRIBING. Conversion. Reconversion. Putting a Problem on the Machine	47
IV.	SHIFT ORDERS. Scaling. Coding Scaled Problem in Straight Sequences.	65
V.	CONTROL UNIT. Front Panel. Operating Instructions	75
VI.	REPETITIVE SEQUENCES. Transfer Orders. Decision Box. Counters. Formation Formula. Address Modification. Extract Orders. Programming and Coding Loops of Repetitive Operations.	81
VII.	SUBROUTINES	127
VIII.	TRANSCRIBER ROUTINE AND INPUT ROUTINE	145
IX.	CODE CHECKING	161
X.	"IBM IN" AND "IBM OUT" ORDERS. "IBM IN" and "IBM OUT" Subroutines.	187
XI.	FLOATING POINT ROUTINE	205
XII.	THE ORDVAC MAGNETIC DRUM	233
	APPENDIX	237

B A L L I S T I C   R E S E A R C H   L A B O R A T O R I E S

REPORT NO. 997

TLeser/MRomanelli/jcw
Aberdeen Proving Ground, Md.
October 1956

PROGRAMMING AND CODING FOR ORDVAC

ABSTRACT

The installation of the new magnetic core memory in ORDVAC and the accompanying improvements in the machine's design have caused a change in the order types and in the address system. Furthermore, the high rate of personnel changes at the Computing Laboratory of the Ballistic Research Laboratories has also indicated a definite need for a manual or text which gives systematically the essentials necessary to code problems for high speed digital computers in general and for particular machines in detail. This text attempts to present to the beginner such basic fundamentals needed in the case of ORDVAC. The ideas presented here are not restricted exclusively to the ORDVAC, since they may be applied to a large family of digital computers which have been built with the same underlying pattern.

PREFACE

A computer is a "data processing machine". Indeed, raw numerical data are put into the computer, are operated on in the machine in a prescribed way, and thereupon yield the desired processed numerical results. Computers are of two types, "analog" and "digital". In an analog computer, numbers are represented by measures of certain physical quantities. For example, a slide rule is an analog computer, one in which numbers are represented by physical distances on a ruler. Or again, numbers in an analog computer may be represented by amounts of voltage or current. On the other hand, a digital computer counts discrete digits and handles number symbols themselves in an appropriate scale of numeration. ORDVAC (Ordnance Discrete Variable Automatic Computer) is an electronic digital computer.

A digital computer performs not only the four elementary arithmetic operations of addition, subtraction, multiplication and division, but also other basic operations, called "logical" operations. The most important of these logical operations are (i) duplicating numbers, (ii) moving numbers from one part of the machine to another, and, (iii) after testing two quantities for "equality" or for a "greater than" condition, choosing one of two paths to follow depending on the results of the comparison. Every problem to be computed on the machine must in its final analysis be expressed in terms of these basic operations.

An outstanding but nevertheless restrictive characteristic of an electronic digital computer is the speed at which it can perform the basic operations. ORDVAC can perform thousands of these operations per second. This fact makes feasible the solution of many problems not previously attempted because of the enormous length of time that would be required to do them by hand. Another notable characteristic of an electronic digital computer is the capacity of its storage device, called "memory". In a memory we can store data and information which in case of hand computations are kept in mind or recorded in notes, tables, etc. A simple machine such as a desk calculator which has a very small memory,

or no memory at all receives an order to perform a single operation, obtains a result which the operator records by hand and stores on paper, then it receives the next order, and so on. Each operational step requires direct human intervention. In a large memory we can store not merely the numerical results of all operations, but also a sequence of orders. A computer with such a memory can be automatically sequenced by a device called "control unit", in the sense that it can be made to perform a whole program of computations without any human intervention. This sequencing in turn requires a very detailed set of instrutions on what the computer has to do. Instructions must take into account all the acts of judgment and memory which in hand computation a person would perform (often automatically and perhaps without realizing their nature) and must be expressed in a language understood by the machine. The machine language is called "code", and the process of translating desired operations into code, is called "coding".

Experience with changing personnel and varied equipment at the Computing Laboratory of the Ballistic Research Laboratories has indicated a definite need for a manual or text which gives systematically the essentials necessary to code problems for high speed digital computers in general, and for one or more particular machines in detail. The aim of this text is to present to the beginner such basic fundamentals needed in case of the ORDVAC. The ideas here presented are not exclusively for ORDVAC, since they may be applied to a large family of digital computers designed with the same underlying pattern. The variations from this pattern called for added refinements only.

ACKNOWLEDGEMENT

The authors express acknowledgement to the referee Dr. Albert A. Bennett who had read the manuscript critically and made invaluable suggestions, and to Mr. George C. Francis who corrected most of the errors having used the report as a text in his course.

CHAPTER I

BINARY AND SEXADECIMAL ARITHMETIC NECESSARY FOR ORDVAC

ORDVAC can deal with numbers only in their binary representation (to the base 2), or for convenience in the essentially equivalent sexadecimal representation (base 16). Therefore an acquaintance with binary and sexadecimal arithmetic is essential for the ORDVAC coder.

Representation of Numbers. In every day life we use numbers in decimal representation, or numerals to the base ten, and these in connection with a position-value notation. For example the numeral 2489 means $2x10^3 + 4x10^2 + 8x10^1 + 9x10^0$. The digit 2, the most significant digit, is the fourth from the right and this position indicates that the digit thus placed is multiplied by 10^3, the digit 4 is the third from the right and is multiplied by 10^2, and so on. Analogously, the numeral 0.2489 means $2x10^{-1} + 4x10^{-2} + 8x10^{-3} + 9x10^{-4}$. A number expressed to the base ten is written as a sum of multiples of consecutive powers of this base. In decimal representation we have available ten different digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, the number of digits being equal to the base, ten. Each non-negative integer less than or equal to 9 is represented by a single corresponding digit, integers greater than 9 are represented by combinations of two or more digits. The numeral for the integer next greater than the greatest digit, 9, is "10", which represents the base, in decimal representation. The above examples illustrate the general principles of positional representation. These principles can be summarized as follows: 1) the number of distinct digit symbols equals the base, 2) the numeral, in positional notation, is written as a sum of digital multiples of consecutive powers of the base, 3) the symbol of the base is "10", 4) each non-negative integer less than or equal to the greatest digit is represented by a one-digit numeral, integers greater than the greatest digit are represented by combinations of two or more digit numerals. 5) The integer next greater than the greatest digit is the base. Thus a number, to the base "b", represented say by "$e_0e_1e_2e_3e_4$" is $e_0b^4 + e_1b^3 + e_2b^2 + e_3b^1 + e_4b^0$. The exponents 0, 1, 2, 3, 4 for the sake of simplicity are kept in decimal representation and each "e_i" is a digit less than b.

From now on whenever necessary we shall indicate the base of a given numeral by a decimal numeral in parentheses. For example 15(16) would mean the number sixteen plus five, represented in the base sixteen, namely twenty-one, (or 21 two-one to the base ten).

Binary Representation. In binary representation the base is two. The symbol of this base is (in binary notation) "10", and the only digits available are 0 and 1. For example "1111(2)" represents $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$, = 15(10). Numerals in binary representation are called "binary numerals", and the digits "0" and "1" are called "bits", a word coined by contracting the words "binary" and "digits". Examples of binary numerals and their ordinary verbal equivalents are shown below:

Binary	Verbal	Decimal
1	one	1
10	two	2
11	three	3
100	four	4
101	five	5
110	six	6
111	seven	7
1000	eight	8
1001	nine	9
1010	ten	10
1011	eleven	11
1100	twelve	12
1101	thirteen	13
1110	fourteen	14
1111	fifteen	15
10000	sixteen	16

Binary	Decimal
$0.1 = 1 \times 2^{-1}$	0.5
$0.11 = 1 \times 2^{-1} + 1 \times 2^{-2}$	0.75
$0.001 = 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$	0.125

The Basic Operations on Binary Numerals. The rules of básic operations for numbers in decimal representation may be so phrased as to remain valid for binary numerals. Addition, subtraction and multiplication are explained in the following examples. Division is omitted as we shall not need it.

$0+0=0 \quad 0+1=1+0=1 \quad 1+1=10$

$0 \cdot 0 = 0 \quad 0 \cdot 1 = 1 \cdot 0 = 0 \quad 1 \cdot 1 = 1$

Addition.

1011	augend	11(10)
111	addend	7(10)
10010	sum	18(10)

We add beginning from the extreme right column (the least significant bit): 1 plus 1 is 10; we write "0" and carry 1. The second column from the right: 1 plus 1 is 10 plus the carry bit 1 is 11; we write "1" and carry 1. The third column: 1 plus the carry bit 1 is 10; we write

"0" and carry 1. The fourth column: 1 plus 1 is 10; we write "0" and carry 1. The fifth column: 1 plus 0 is 1; we write "1". The sum is 10010(2) = 18(10).

The example is easy. There was never a carry into more than one column. But if a series of more than two binary numbers is added the "carry" problem becomes more difficult. Often three or four carries are involved for one column. Because of the increasing difficulty of adding longer and longer series of binary numbers, electronic computers add two numbers at a time and then add the third to the first sum, etc. Although more operations are involved, the additional time consumed is practically nil because of the extremely high speed of the machines.

An important case of addition is adding a binary 1 to a binary number N. If N is a number whose bits are not all ones, the next larger number N + 1 is obtained by changing the least significant 0 to 1 and changing all the 1's to the right of it to 0. For example:

10101 + 1 = 10110

10111 + 1 = 11000.

<u>Subtraction.</u>

1011	minuend	11(10)
111	subtrahend	7(10)
100	difference	4(10)

We begin subtracting from the extreme right column: 1 minus 1 is 0, we write "0". The second column from the right: 1 minus 1 is 0; we write "0". The third column: the digit 1 in the subtrahend is greater than the digit 0 in the minuend hence we borrow 1 from the fourth column (which makes 10 in the third column) in the minuend 1 from 10 is 1; we write "1". The fourth column: after borrowing 1 from the fourth column there is nothing left in the minuend; we do not write anything. The difference is 100(2) = 4(10).

<u>Multiplication.</u> (To find the product of 1011(2) the multiplicand, by 111(2), the multiplier.)

```
  1011 x 111            11 x 7
  ----------            ------
                        77 (10)
   1011
  1011
 1011
-------
1001101   (product)
```

Multiplication of binary numbers hardly needs explaining: We always multiply by 1 or by 0 which makes each single step much easier than in multiplying numbers in decimal representation.

An important case is multiplication of binary numbers by powers of 2. Remembering that $2^n(10) = 10^n(2)$ * (the exponent n is in decimal representation), multiplication of a binary number by 2^n is performed by shifting the binary point through n places to the right if n is positive, or to the left if n is negative. For example:

$$11.01011(2) \times 2^3(10) = 11010.11(2)$$

$$11.01011(2) \times 2^{-4}(10) = .001101011(2)$$

Conversion from Decimal to Binary Representation. The rule for converting an integer numeral N(10) to its binary equivalent is as follows: divide N by 2, write the quotient and remainder together in the next line below, with the quotient directly underneath and the remainder in a special column at the right; repeat this process with the first quotient, that is, divide this quotient by 2, write the new quotient, and the new remainder; continue thus until the last quotient becomes zero. The remainders which can be only 1 or 0 give, when read from bottom to top, the binary equivalent of "N(10)", the last remainder represents the first, the most significant bit, the first remainder represents the last, the least significant bit.

Example: Find the binary equivalent of 867(10).

Quotients	Remainders
867 ÷ 2	
433 ÷ 2	1
216 ÷ 2	1
108 ÷ 2	0
54 ÷ 2	0
27 ÷ 2	0
13 ÷ 2	1
6 ÷ 2	1
3 ÷ 2	0
1	1
0	1

↑ Read toward the point for integers

Thus 867(10) = 1101100011(2).

* Read, two to the nth (base ten) = one-zero to the n (base two)

The rule for converting a decimal fraction 0.N(10) to its binary equivalent is as follows: Multiply 0.N by 2, write the decimal part of product underneath and the integral part in the same line in a special column at the left; multiply only the fractional part of the product by 2 and proceed as before; continue until the fractional part of the product becomes zero, which would mean that the fraction terminates, or carry until the desired number of bits is obtained; the integral parts of the products give the binary equivalent of 0.N(10); the first integral part represents the first bit after the binary point.

Example 1. Find the binary equivalent of 0.671875(10).

Integral parts	Fractional parts
0 ,	671875x2
1	343750x2
0	687500x2
1	375000x2
0	750000x2
1	500000x2
1	000000

Read from the point for fractions.

Thus 0.671875(10) = 0.101011(2).

Example 2. Find the binary equivalent of 0.6(10).

Integral parts	Fractional parts
0 ,	6x2
1	2x2
0	4x2
0	8x2
1	6x2
1	2x2
0	4x2

In this example the binary fraction equivalent to 0.6(10) is a non-terminating recurring fraction. Thus

0.6(10) = 0.1001 1001 1001 . . . .(2) $0.\dot{1}\dot{0}\dot{0}\dot{1}(2)$.

Exercises: Convert the following numbers in decimal representation to binary representation. In case of non-terminating fractions find at least ten most significant bits.

1)3456 2)80012 3)10093 4)0.010203 5)345009 6)0.53125

Sexadecimal Representation. In sexadecimal representation the base is sixteen, the symbol of the base is "10" and the digits are 0,1,2,3,4, 5,6,7,8,9,K,S,N,J,F,L. The new digits, K,S,N,J,F,L whose decimal equivalents are 10,11,12,13,14,15, can be remembered from the mnemonic: "King Size Numbers Just for Laughs". For example "K8N(16)" represents $10 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 2700(10)$. Examples of sexadecimal numbers and their decimal equivalents are shown below:

Sexadecimal	Decimal
10	16
11	17
12	18
13	19
14	20
1F	30
28	40
32	50
3N	60
46	70
50	80
64	100
N8	200
1L4	500
3F8	1000

Addition and Subtraction. The rules of operations for numbers in decimal representation may be so phrased as to be valid for sexadecimal numbers. We shall need only addition and subtraction

Addition.	156K	augend	5482(10)
	1LS	addend	507(10)
	1765	sum	5989(10)

We begin adding from the extreme right column: S plus K is 15[decimally it means 11 plus 10 is 21 = 15(16)], we write "5" and carry 1. The second column: the carry digit 1 plus L is 10 plus 6 is 16; we write "6" and carry 1. The third column: the carry digit 1 plus

1 is 2 plus 5 is 7; we write "7". The fourth column: 0 plus 1 is 1; we write "1". The sum is 1765(16) = 5989(10).

Subtraction.	156K	Minuend	5482(10)
	1LS	subtrahend	507(10)
	136L	difference	4975(10)

We begin subtracting from the extreme right column: the digit S = 11(10) in the subtrahend is greater than the digit K = 10(10) in the minuend, hence we borrow 1 from the second digit in the minuend (which is 10(16) in the first column): S from 1K is L [decimally it means 11 from 26 is 15 = L(16)], we write "L". The second column: after borrowing, the second digit of the minuend is 5; the digit L in the subtrahend is greater than the digit 5 in the minuend hence we borrow 1 from the third digit in the minuend: L from 15 is 6 [decimally it means 15 from 21 is 6]; we write "6". The third column: after borrowing 1, the third digit in the minuend is 4; 1 from 4 is 3; we write "3". Fourth column: 0 from 1 is 1; we write "1". The difference is 136L(16) = 4975(10).

<u>Conversion from Binary to Sexadecimal Representation and from Sexadecimal to Binary.</u> Conversion from binary to sexadecimal representation is very simple. The rule for converting a four-bit binary numeral abcd to the equivalent sexadecimal numeral is as follows: make the following correspondence: the first digit from the right corresponds, in virtue of its position-value, to the multiplier 1(10), the second to 2(10), the third to 4(10), the fourth to 8(10), thus:

a is the multiplier of $8 = 2^3$

b is the multiplier of $4 = 2^2$

c is the multiplier of $2 = 2^1$

d is the multiplier of $1 = 2^0$.

The sexadecimal equivalent of abcd is then 8a + 4b + 2c + 1d. This sum is less than 16(10) because each of a, b, c, d is 0 or 1, therefore a

sexadecimal number equivalent to a four bit binary numeral consists of only one sexadecimal digit. For example 1011(2) = 8x1 + 4x0 + 2x1 + 1x1 = 11(10) = S(16).

The rule for converting a binary numeral to a sexadecimal numeral is as follows: divide the numeral in groups of four bits starting from the binary point, to the left and to the right, and write the corresponding sexadecimal digit for every group. For example to convert 1 1010 0111 .00111(2) we group as follows:

(1) (1010)(0111).(0011)(1) = 1K7.38(16)

1 K 7 . 3 8

When the number of bits on each side of the point is a multiple of four then the number of digits in equivalent sexadecimal number is only a quarter as great.

The rule for converting a sexadecimal numeral to its binary equivalent is as follows: Express each sexadecimal digit as a sum 8a + 4b + 2c + 1d, where each a, b, c, d, is 0 or 1, which will give the bits, a, b, c, d corresponding to the given sexadecimal digit. For example to convert KSL8(16) we proceed:

K = 8x1 + 4x0 + 2x1 + 1x0, giving 1010
S = 8x1 + 4x0 + 2x1 + 1x1, giving 1011
L = 8x1 + 4x1 + 2x1 + 1x1, giving 1111
8 = 8x1 + 4x0 + 2x0 + 1x0, giving 1000

Thus KSL8(16) = 1010101111111000(2).

Conversion from Decimal to Sexadecimal Representation. The rules for converting an integer N(10) or a decimal fraction 0.N(10) to its sexadecimal equivalent are very similar to the rules for converting to binary equivalents and need not be repeated. The only difference is that in case of an integer N(10) we divide by 16(10), and in the case of a fraction 0.N(10) we multiply by 16(10). Sometimes it is more convenient to convert first to binary equivalents and then to sexadecimal equivalents.

Examples. 1) Find the sexadecimal equivalent of 961(10)

Quotients	Remainders
961 ÷ 16	
60 ÷ 16	1
3 ÷ 16	N = 12(10)
0 ÷	3

Thus 961(10) = 3N1(16).

Converting first to binary equivalent and then to sexadecimal equivalent:

Quotient	Remainder
961 ÷ 2	
480 ÷ 2	1
240 ÷ 2	0
120 ÷ 2	0
60 ÷ 2	0
30 ÷ 2	0
15 ÷ 2	0
7 ÷ 2	1
3 ÷ 2	1
1 ÷ 2	1
0	1

Thus 961(10) = ..11 1100 0001(2) = 3N1(16)
3 N 1

2) Find the sexadecimal equivalent of 0.345(10)

Integral Part	Fractional Part
0	.345 x 16
5	.520 x 16
8	.320 x 16
5	.120 x 16
1	.920 x 16
F	.720 x 16
S	.520 x 16

Thus, 0.345(10) = 0.5851FS....(16)

Converting to binary and then to sexadecimal equivalent:

Integral Part	Fractional Part
0	.345 x 2
0	.690 x 2
1	.380 x 2
0	.760 x 2
1	.520 x 2
1	.040 x 2
0	.080 x 2
0	.160 x 2
0	.320 x 2
0	.640 x 2
1	.280 x 2
0	.560 x 2
1	.120 x 2
0	.240 x 2
0	.480 x 2
0	.960 x 2
1	.920 x 2
1	.840 x 2
1	.680 x 2
1	.360 x 2
0	.720 x 2
1	.440 x 2
0	.880 x 2
1	.760 x 2
1	520 x 2

Thus, 0.345(10) = ...0. 0101 1000 0101 0001 1110 1011 = 0.5851FS...(16),
0 5 8 5 1 F S

where 0.5851FS denotes the recurring fraction 0.5851FS 851FS.....

Exercises: Convert to sexadecimal equivalents:

1) 2358(10) 2) 0.2538(10) 3) 0.0110011(2) 4) 1.011110100011(2)

Convert to binary equivalents:

5) 0.KKLLSS(16) 6) 0.012345(16) 7) 12K.KSNJF32(16)

Representation of Negative Numbers by Complements. In order not to complicate the design of electronic computers a positive complement representation as described below was found for a negative number. In cases of subtraction the positive complement of the subtrahend can be added formally to the minuend to obtain the desired difference. Thus, the operation M - N can be done as M plus the complement of N. Let us consider for example a computer which holds in the "register" only three digits.

If a result of some operation has more digits than three the register will show the three least significant digits. (A register in a computer is a device which holds a number and records some or all of the operations on them). The digits beyond the left end of the register will be lost. For example multiplying the decimal number 231 by 11 we should obtain 2541, but the first digit "2" will be lost and the register will show 541. Let M and N be three-digit positive numbers. Using $(10^3 - N)$ as the complement representation for (-N) we get M plus $(10^3 - N) = (M - N)$ plus 10^3. Since 10^3 is 1 multiplied by 1000 the digit 1 exceeds the capacity of the register, being beyond the left end of the register; it will be lost and the number shown in the register will be M - N. For example if M = 324 and N = 135, then 135 is first subtracted from 1000 and the result is added to 324.

$$\begin{array}{r} 1000 \\ -135 \\ \hline 865 \end{array} \qquad \begin{array}{r} 324 \\ +865 \\ \hline 1189 \end{array} = M - N$$

Since 1, the leading digit, is lost the correct difference 189 stands in the register.

The number $(10^n - N)$ is called the "complement" of N (with respect to 10^n) where the exponent n equals the number of positions in the register. ORDVAC is a "fixed" point machine, in fact, it handles only numbers whose absolute values are less than 1 (save for the formal exception to be mentioned presently). If a result of some operation is greater than 1 the register in ORDVAC will normally hold only the fractional part and the integral part will be lost. The lost excess is called "the overflow". In ORDVAC the complement of a number + N(2) is [10(2) - N(2)]. Examples:

Numbers	Complements	
0.1	10 - 0.1	= 1.1
0.01	10 - 0.01	= 1.11
0.101	10 - 0.101	= 1.011

Note: Complement of a complement of N is N.

It is seen that the complement of any proper fraction has 1 in its integral part. In such a case the digit to the left of the binary point would be interpreted as representing a negative number. It must be stressed that the 1 in the integral part would be correctly interpreted by the computer only in the case of negative numbers represented by the complement.

An easy rule to convert a number - N(2) to the binary ORDVAC complement of N(2), or the binary complement of N(2) to -N(2) is as follows: Replace in N, "0"'s by "1"'s and "1"'s by "0"s, except (Counting from the left) the last "1", and leave the "0"'s following the last "1" unchanged. For example; if

	N = 0.01011100
then the complement of N	= 1.10100100, = representation of -N;
or, if the complement of N	= 1.01010110, = representation of -N,
then	N = 0.10101010.

Examples of binary subtraction performed by complement addition:

1) Subtract 0.0011 from 0.1011

```
By ordinary subt.     0.1011
                     -0.0011
                     -------
                      0.1000

By complement addition    0.1011
complement of 0.0011   = +1.1101
                         -------
                         10.1000
```

The minuend plus the complement of the subtrahend is 10.1000, but the ORDVAC register will hold 0.1000 which is the correct difference 0.1011 -0.0011.

2) Subtract 0.1101 from 0.0011

```
By Ordinary subtraction *    0.0011
                            -0.1101
                            -------
                            -0.1010

By complement addition       0.0011
                            +1.0011
                            -------
                             1.0110
```

The sum of the minuend and the complement of the subtrahend is 1.0110 which will stand in ORDVAC registers. It is interpreted as -(10.0000-1.0110) = -0.1010 which is the correct difference of 0.0011 - 0.1101. The result -0.1010 could be obtained directly by the rule of replacing 0's by 1's and 1's by 0's in 1.0110.

Machine Representation. ORDVAC handles forty-bit fractions of the form $e_0\ e_1\ e_2 \ldots\ldots\ldots e_{39}$. The binary point follows after the first bit, e_0, and e_0 is "0" (in case of a positive fraction), or "1" (for use with a negative fraction), where, as explained before, a negative fraction -N is replaced by the complement of N. The bit e_1 is the first significant bit and e_{39} is the last significant bit. If positive as well as negative numbers are taken into consideration, a thirty-nine significant bit fraction may assume $2^{40} - 1$ different values. To give the "machine representation" of a number, N, means to express N in the above described form. The binary point is self-understood and is usually omitted.

Examples:

1. Give the machine representation of 1/2.
   1/2 = 0.5(10) = 0.1(2), thus the machine representation of 1/2

   is 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000

2. Give the machine representation of -1/2.
   -1/2 = -0.5(10) = -0.1(2); the complement of 0.1 is 1.1, thus the machine representation of -1/2 is 1100 ......0 (thirty-eight zeros).

Exercises: Give the machine representation of: 3/4, -3/4, 0, -1, 3/16, -7/8, 1.

Machine Sexadecimal Representation. To avoid writing by hand all the forty bits of machine binary numerals we use special sexadecimal equivalents which represent machine binary numerals and have only ten sexadecimal digits. The "ORDVAC machine sexadecimal representation" of binary numeral N(2) is the sexadecimal equivalent of N(2) divided by 2. To convert a binary numeral N(2) to its machine sexadecimal representation, it is convenient to divide it by 2 before the conversion

by merely moving the binary point through one bit to the left. Thus, we first divide the numeral N(2) into ten groups of four bits. The first group containing e_0 as the first bit; and then we replace each group of four bits by its sexadecimal equivalent. For example, the proper sexadecimal equivalent of the binary numeral is,

0.1000	0100	0001	0100	1000	0110	0000	0000	0100	000
8	4	1	4	8	6	0	0	4	0

or 0.8414860040(16), but the machine sexadecimal representation of the binary numeral (binary point moved one left), is,

0100	0010	0000	1010	0100	0011	0000	0000	0010	0000
4	2	0	K	4	3	0	0	2	0

or 420K430020. A number in machine sexadecimal representation has ten sexadecimal digits. We shall repeat that the machine sexadecimal representation equals one half of the fractional part of the proper sexadecimal representation. For example, if the proper sexadecimal representation is 0.8414860040, the machine sexadecimal representation is 420K430020.

Conversion from Decimal to Machine Sexadecimal Representation. To convert a positive decimal fraction to its machine sexadecimal representation we multiply first by eight and after that we continue to multiply by sixteen. This process is carried only through the tenth sexadecimal digit. The zero in front of the sexadecimal point is omitted and it is not counted as one of the digits. Sometimes it is more convenient to convert first to binary machine representation and then to machine sexadecimal representation, or to convert first to the proper sexadecimal representation and divide by 2. For a negative fraction - N first convert + N and then complement with respect to 16.

Example. Give the machine sexadecimal representation of 0.5671(10).

Integral Part	Fractional Part
0	5671 x 8
4	5368 x 16
8	5888 x 16
9	4208 x 16
6	7328 x 16
S	7248 x 16
S	5968 x 16
9	5488 x 16
8	7808 x 16
N	4928 x 16
7	8848 x 16

Thus the machine sexadecimal representation of 0.5671(10) is 4896SS98N7. In this non-terminating process the conversion is carried only through the tenth sexadecimal digit, not counting the zero in front of the sexadecimal point.

Converting first to machine binary and then to machine sexadecimal equivalent:

Integral Part	Fractional Part
0	5671 x 2
1	1342
0	2684
0	5368
1	0736
0	1472
0	2944
0	5888
1	1776
0	3552
0	7104
1	4208
0	8416
1	6832
1	3664
0	7328
1	4656
0	9312
1	8624
1	7248

....
.... This is carried through the fortieth bit.

Thus, 0.5671(10) =	0100	1000	1001	0110	1011	1011	1001	1000	1100	0111
	4	8	9	6	S	S	9	8	N	7

and is represented sexadecimally by 4896SS98N7.

Conversion from Decimal to Machine Sexadecimal Representation On a Desk Calculator. The conversion from decimal to machine sexadecimal representation is carried through the tenth sexadecimal digit. A ten digit sexadecimal fraction corresponds to a twelve or thirteen digit decimal fraction, thus in order to obtain the maximum precision twelve or thirteen most significant decimal digits must be converted to

sexadecimal equivalents. The registers of a desk calculator hold only ten decimal digits and normally the maximum precision could not be obtained.

Dr. Harold K. Crowder from Case Institute of Technology, devised an ingenious method for converting twelve or thirteen digit decimal fractions into ten digit machine sexadecimal equivalents on a ten bank desk calculator. The decimal fraction to be converted is broken into two parts, the most significant ten decimal digits and the remainder. Consider the twelve digit decimal fraction .XXXXX XXXXX YZ. The portion .XXXXX XXXXX 00 can be converted in the usual manner shown in the preceding example carried through the eleventh sexadecimal digit. With the remainder .00000 00000 YZ we proceed as follows: Multiply .00000 00000 YZ by 3436, write the fractional part of the product underneath and integral part in the same line in a special column at the left. Record the integral part as a ninth digit of a machine sexadecimal number, the first eight digits being zeros, proceed multiplying by 16 and recording the integral parts as in the preceding example.

When the two machine sexadecimal numbers have been constructed they are to be added. The sum rounded to ten digits yields the complete converted number.

It may be noted that the number 3436 referred to above consists of the first four digits of 2^{35}, the described process being a short cut for multiplication by 8 once and by 16 eight times.

Example: Give machine sexadecimal representation of cos 1 = 0.54030 23058 68

Integral Part	Fractional Part
0	54030 23058 x 8
4	32241 84464 x 16
5	15869 51424 x 16
2	53912 22784 x 16
8	62595 64544 x 16
K	01530 32704 x 16
0	24485 23264 x 16
3	91763 72224 x 16
F	68219 55584 x 16
K	91512 89344 x 16
F	64206 29504 x 16
K	27300 72064 x 16
eight zeros	00068 00000 x 3436
2	33648 00000 x 16
5	38368 00000 x 16
6	13888 00000 x 16

Conversion of most significant part:	.4528K	03FKF	K
Conversion of least significant part:	.00000	00025	6
Converted number:	.4528K	03FJ4	0
Rounded converted number:	.4528K	03FJ4	

<u>Exercises</u>: Give the machine sexadecimal representation of

a) the binary machine numeral:

1) 010101000101...00, 2) 101010100110011...00.

b) the decimal fraction:

3) 0.3425, 4) 0.8132, 5) -0.8132 6) -0.1221.

Find the decimal equivalent of

c) the binary machine numeral:

7) 011010010...00 8) 1101000110...00

d) the ORDVAC sexadecimal:

9) 3K800 00000 10) J9L00 00000.

CHAPTER II

INTRODUCTORY DESCRIPTION OF ORDVAC OPERATIONS FLOW CHARTS. CODING. SHORT LIST OF SYMBOLS.

The Components. ORDVAC consists of three main parts or "units" namely: "the memory", "the arithmetic unit", and "the control unit". The memory is a storage device and consists currently of 4096 storage locations called "memory positions" which can be thought of as separate permanent storage boxes. In each memory position can be stored at any given time, one forty-bit number. The arithmetic unit is that part of the machine which performs the actual addition, subtraction, multiplication and division. This unit consists of several numbered registers, each capable of holding one forty-bit number for immediate use. The most important of these registers are: 1) "the accumulator register, R1," which is for addition and subtraction, and 2) "the arithmetic register, R2", which is mainly for multiplication and division. The control unit is the mechanism which schedules the performance of the operations so that they will occur in the desired sequence. A more detailed description of the main components of this unit will be given in Chapter V.

Any electronic digital computer operates at such speeds that no human operator can either supply the machine with data at an adequate rate for processing, or write down computed numbers as fast as the machine can supply them. This necessitates the employment of auxiliary terminal components called "input" and "output" devices.

Figure 1 shows the major units of the machine and the arrows indicate the possible directions of flow of information through the various units. Such information is in a numerical form.

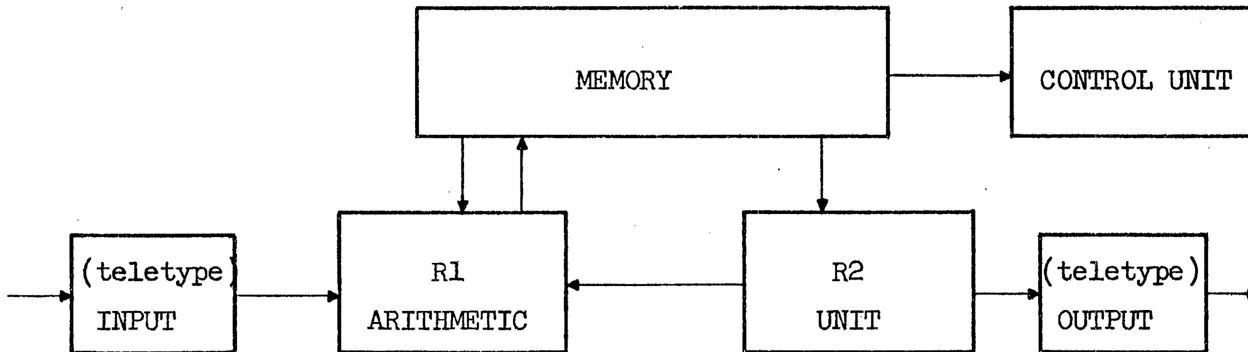


Figure 1

The greatly simplified diagram in Figure 1 shows that all information, external or internal, goes to the memory by way of R1. Information can be extracted from the memory by way of either R1 or R2. It is possible to move information directly from R2 to R1, but generally not vice versa. The numerical results recorded in the teletype output device must come from R2. (For IBM input and output see Chapter X). While the control unit receives information from the memory unit, it never transmits information in numerical form.

The Words and Orders. An ORDVAC "word" is an organized unit of information, consisting of forty-bits. Each memory position can contain at any given time, exactly one word. Similarly, each arithmetic register can contain, but only temporarily, exactly one word. A word can represent either:

1) One forty-bit "datum number" which is one of the numbers to be operated on (processed) by the machine.

2) One or two instructions that specify what the machine is to do with the datum numbers. In both cases a word looks like a forty-bit number. The machine is able to distinguish datum numbers from instructions in the following way: All words that enter the control unit are interpreted as instructions, all words that enter the arithmetic unit are interpreted as datum numbers. The important feature that a datum word and an instruction word have identical form, far from being an embarrassing disadvantage, is in fact, a great convenience. The same word can be interpreted in one connection as an instruction, in

another, as a datum number, depending upon which machine unit does the interpreting. We can bring an instruction word into the arithmetic unit and treat it as a datum number. We can change an instruction word by performing some operation on it and storing it back in a memory position ;it can then be interpreted as a different instruction word. In this way the machine can modify its own instructions, and with great flexibility.

The structure of words representing datum number was explained in the first chapter.

An instruction word consists of two distinct groups of twenty bits each called an "order". Figure 2 explains the structure of an order.

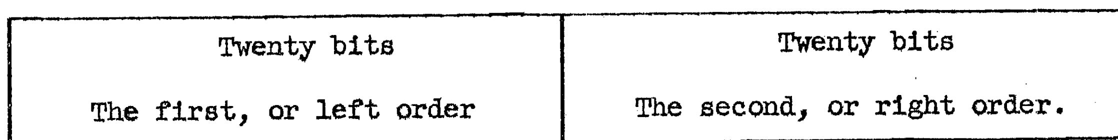
The Order

Instruction: 6 bits (Order type)	2 bits	Address: 12 bits
These six bits identify an operation (arithmetic or logical).	These two bits are both zero. They separate instruction from address.	These twelve bits identify one of the $4096 = 2^{12}$ memory positions.

Figure 2

The first six bits of an order identify the kind of operation to be performed, the last twelve bits together represent in binary notation the address of some particular one of the 4096 memory position (as will be discussed presently). We repeat that the forty separate bits in the two twenty-bit orders in an instruction word are strung in one row and have together exactly the same form as a datum number. The grouping and subgrouping of instruction words which we just described is purely a matter of interpretation, recognized by the machine. A diagram of an instruction word as a whole, in terms of orders, is shown in Figure 3.

The Instruction Word



Twenty bits	Twenty bits
The first, or left order	The second, or right order.

Figure 3

The first of the two orders in an instruction word is referred to as the "left order", the second as the "right order".

The Address. The 4096 memory positions are numbered serially, like safe deposit boxes in a bank vault. The number permanently identifying a memory position consists of twelve bits and is called "the address" of that memory position. Furthermore, a memory position has associated with it, at any one time, one word which is referred to as "the contents" of this memory position. Thus, each memory position has:

1) A unique permanent identifying address (12 bits) which can be compared to a box number or cell number.

2) The contents of this memory position (40 bits), which can be compared to the contents of the box or cell.

The Raster. The "raster" is a television-like screen on the panels of many computers where each memory position is represented by a green dot. The dots form a rectangular arrangement. By analogy, a similar rectangular arrangement drawn on a sheet of paper is also called "the raster" or the "raster sheet". We shall use the word "raster" in the later sense, because ORDVAC does not have a raster screen on its panels. Figure 4 shows a raster as a rectangular arrangement of 1024 addressed memory positions in a 32 x 32 matrix. The memory has 4096 positions needing four raster sheets to represent them. Figure 4 shows the left upper sheet, Figure 5 the diagram of all four raster sheets. Each small rectangular unit corresponds to one memory position. The identification of the 4096 positions in sexadecimal notation ranges from 000 to LLL.

The left part THE LEFT UPPER RASTER SHEET The right part

	0	1	2	3	4	5	6	7	8	9	K	S	N	J	F	L	0	1	2	3	4	5	6	7	8	9	K	S	N	J	F	L	
00																																	01
02																																	03
04																																	05
06																																	07
08																																	09
0K																																	0S
0N																																	0J
0F																																	0L
10																																	11
12																																	13
14																																	15
16																																	17
18																																	19
1K																																	1S
1N							X																										1J
1F																																	1L
20																																	21
22																																	23
24																																	25
26																																	27
28																								O									29
2K																																	2S
2N																																	2J
2F																																	2L
30																																	31
32																																	33
34																																	35
36																																	37
38																																	39
3K																																	3S
3N																																	3J
3F																																	3L
	0	1	2	3	4	5	6	7	8	9	K	S	N	J	F	L	0	1	2	3	4	5	6	7	8	9	K	S	N	J	F	L	

Figure 4

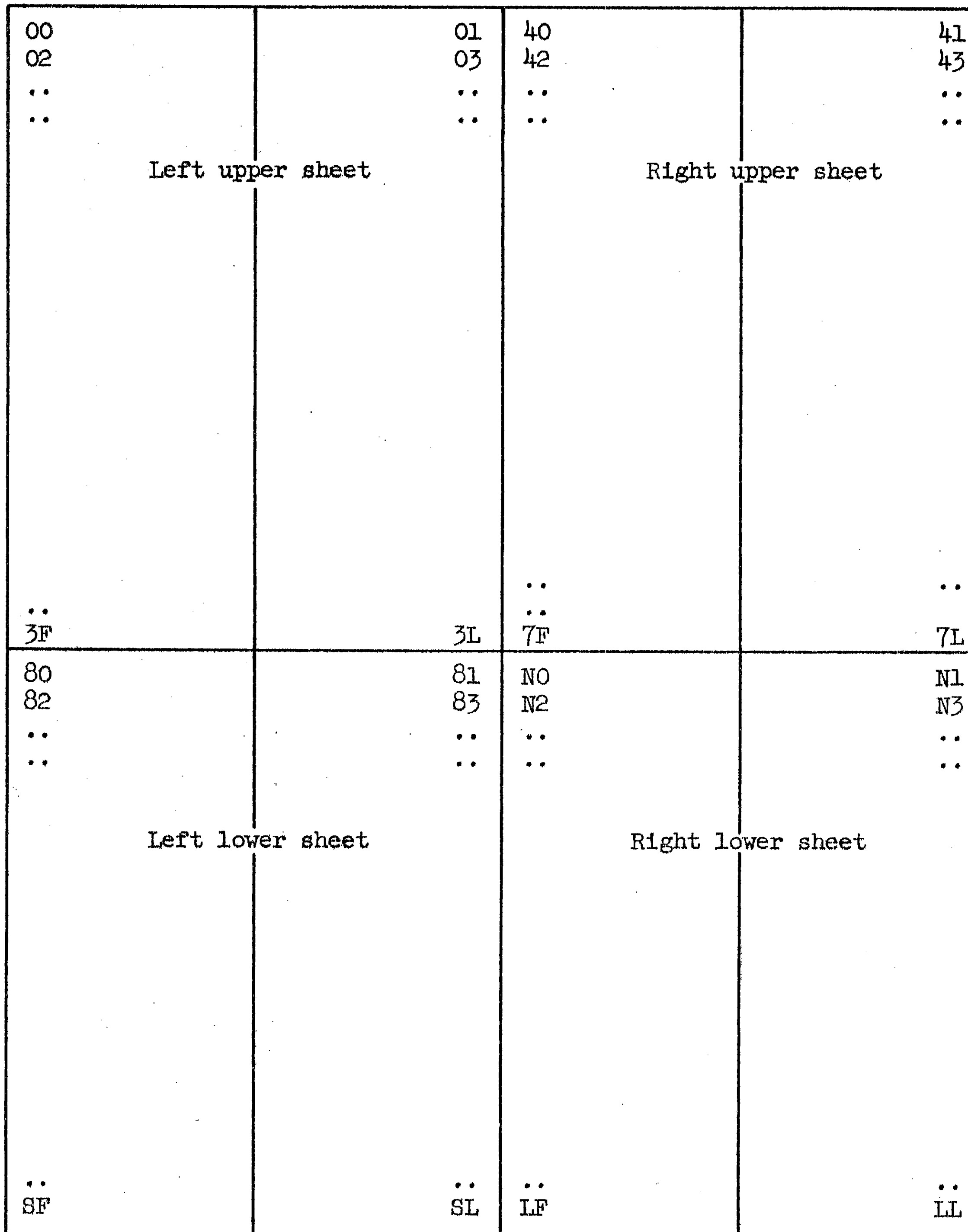


Figure 5

A vertical heavy line drawn in Figure 4 divides the raster sheet into two parts. The two digit sexadecimal numerals in the left margin mark the rows of the left part, the numerals in the right margin mark the rows of the right part. The numerals in the upper and lower margins mark the columns separately for each part. For example the little rectangle here marked with an "X" is in the left part of the raster sheet in the row labeled "1N" and the column labeled "6". It represents the address 1N6; the first two sexadecimal digits, 1N, being the number of the row, the last sexadecimal digit, 6, being the number of the column. The little rectangle marked with a "0" is in the right part of the raster sheet in the row labeled "29" and the column labeled "7". It represents the address 297.

The rows and the columns in the three raster sheets representing the remaining 3072 memory positions (1024 positions in each sheet) are numbered as follows:

<u>The right upper sheet</u>:

Left part: Rows; 40, 42, 44, ..., 7F; Columns; 0, 1, 2, ..., L.
Right part: Rows; 41, 43, 45, ..., 7L; Columns; 0, 1, 2, ..., L.

<u>The left lower sheet</u>:

Left part: Rows; 80, 82, 84, ..., SF; Columns; 0, 1, 2, ..., L.
Right part: Rows; 81, 83, 85, ..., SL; Columns; 0, 1, 2, ..., L.

<u>The right lower sheet</u>:

Left part: Rows; N0, N2, N4, ..., LF; Columns; 0, 1, 2, ..., L.
Right Part: Rows; N1, N3, N5, ..., LL; Columns; 0, 1, 2, ..., L.

<u>Programming and Coding</u>. "The Program" is a plan for solving a given problem. Programs may therefore range in complexity from a tentative sketchy outline to a complete elaborate working system of directions. Planning a method for solving a problem is called "programming". This term is somewhat flexible because the programming may vary with difficulty of the problem or with the experience of the programmer and of course may reflect the special restrictions of the

prospectively available computer. The process of programming or planning a solution is distinguished from the "coding" which is the translation of a program into the language of a specific machine. The "coded program", or "routine", although often also loosely referred to as the "program", is a sequence of machine words (explained before) instructing the machine to perform specified operations which will lead to a solution.

We learned in Chapter I that ORDVAC handles binary fractions and we shall assume in this chapter that all datum numbers and the results of all operations on them are such fractions. We shall consider now a very simple example of programming and coding a problem: given two numbers a_1 and a_2, to program and to code the computation and the printing of the sum $a_1 + a_2$. The machine can start the operations when the numbers a_1 and a_2 and the coded program are stored in the memory. The sequence of words which constitutes a program will usually be stored in consecutive memory positions (at consecutive addresses) in their proper order. Storing a coded program (a routine) in the memory is called "reading in" a program. Reading in a routine directly is cumbersome and seldom done. In practice, before reading in a routine we store an auxiliary coded program called the "input routine". Due to special features the input routine can be stored very easily. When once stored in the memory the input routine has the facility of automatically reading in a given computation program, as will be explained later. The input routine can also perform the necessary task of storing the numerical data which will have to be processed in the course of computation. Hence the storing of initial datum numbers need not be a part of the computation program. In our present program and in most of the problems which follow we shall assume that the input routine with all the input datum numbers is already available prior to introducing the coded computation program. In our problem then the input routine will store the a_1 and a_2 mentioned above at some addresses, say A1 and A2 respectively.

Programming. The programming for our problem would consist of the following outline:

I. Compute the sum $a_1 + a_2$

II. Print the sum and stop.

In I are grouped the operations associated with the computations of $a_1 + a_2$, in II are grouped the operations associated with printing the result and stopping the machine. As a rule the outline is prepared in the form of a diagram called the "flow chart", having numbered "boxes" connected by arrows indicating the flow of information. Thus, for our simple problem, programming is reduced to preparing a flow chart. The flow chart of our problem is shown in Figure 6.

FLOW CHART, $a_1 + a_2$

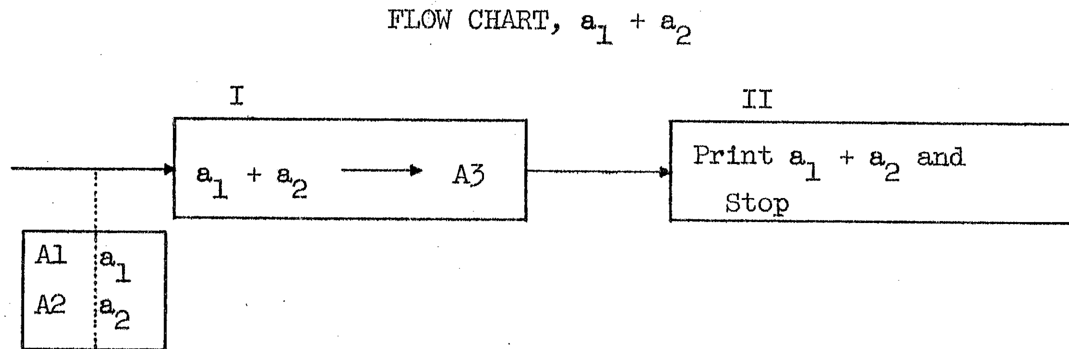


Figure 6

Notes on the Flow Chart.

a) Groups I and II of the outline are shown in Boxes I and II on the flow chart. It is customary to use Roman numerals to designate major sections of a program. The orders corresponding to operations in Boxes I and II are often referred to as "Sequence" I and II. There are no strict rules as to how to group operations in a box: experience will eventually show the coder that it is natural to end boxes at "transfer orders" (explained later). The orders representing individual operations are numbered with Arabic numerals following the common Roman numeral, thus: in Box I: I,1; I,2; I,3; etc; in Box II: II,1; II,2; II,3; and so on.

b) The arrows on the lines connecting the boxes indicate the order in which the groups of operations will take place, which group will come first and which next, etc.

c) The box connected to the flow chart by a broken line is not numbered because the operations grouped in it are not a part of the program. This box is called the "Storage Box" and the operation in it consist of storing the numbers a_1 and a_2 in memory positions A1 and A2. We have assumed that these operations would be performed by the input routine; the presence of the storage box on the flow chart reminds the coder where the datum numbers are stored.

d) The arrow inside Box I indicates that the sum $a_1 + a_2$ is to be stored in memory position A3.

Preliminary Coding.

Before the example can be coded it will be necessary to digress upon the ORDVAC's order structure. The ORDVAC can execute fifty different orders. A description of the complete list of orders is given in the Appendix. p. 240 A short list of orders sufficient for the early simple examples is presented below.

Short List of Orders. Two forms of representation of orders are commonly used: the "preliminary representation" and the "sexadecimal representation". For final coding, for actual use on the machine the machine sexadecimal representation must be used. However, in the early stages of coding, when many changes have to be made and before addresses have actually been assigned, it is convenient to use "a preliminary order representation", a notation which is generally more easily understood by the coder. In this short list a combination of a capital letter and a numeral, like A1, represents an address of a memory position, save for two exceptions to be mentioned. Parentheses placed about an address symbol are used to represent at any stage in the flow chart, the contents of the given address at that stage, thus "(A1)" represents the contents of memory position A1. The exceptions to this interpretation of letter-numeral combinations are "R1 and "R2" which stand for accumulation and arithmetic registers (explained before). However, the use of parentheses still holds: "(R1)" and "(R2)" represent contents of R1 and R2 (a 40-bit word in each case). An arrow, " $\rightarrow$ " is

read "goes to". For example "(R1) → A3" is read: "the content of R1 goes to the memory position A3", and it means that the 40-bit word in R1 is duplicated in memory position A3. The previous contents of a memory position or of a register are cancelled only when replaced by another word, save for a few exceptions which will be explained later.

A Short List of Orders

No	Preliminary Representation of Orders	Sexadecimal Representation of Orders	Description of Orders	Verbal Name
1	+B1	K4...	(B1) → R1	Clear, add
2	(+)B1	N4...	(R1)+(B1) → R1	Hold, add
3	-B1	24...	-(B1) → R1	Clear, Subtract
4	(-)B1	04...	(R1)-(B1) → R1	Hold, Subtract
5	M B1	10...	(R1) → B1	Store
6	XuB1	K8...	(R2)x(B1) → R1	Exact multiply ~~multiplication~~
7	÷ B1	78...	(R1)÷(B1) → R2	Divide
8	R B1	S4...	(B1) → R2	
9	P	L4028	Print (R2) on teleprinter in sexadecimal form; erases ~~R1 to zeros~~, R2 to 1's.	Print
10	Z u	00000	Stop (for the machine)	Stop
11	U B1	NO...	Transfer control to the left order of B1	Transfer

Figure 7

A+ (next pair) KN... (R2) → R1

Notes on the List of Orders:

a) The two sexadecimal digits in each line of the third column of the list are the sexadecimal equivalents of a six-bit instruction followed by two binary zeros (explained before). The three dots used in the representation in the third column are to be filled later by three sexadecimal digits representing the sexadecimal equivalent of the twelve-bit address, B1.

b) The preliminary representation of Orders No. 2 and No. 4 stand respectively with "(+)" and "(-)". The parentheses are used to distinguish them from Orders No. 1 and No. 3, which start with ", +", and "-". The Orders No. 9, P, and No. 10, Zu, differ in structure from other orders, in not containing address parts. They consist of twenty-bit instructions represented by five sexadecimal digits.

c) Order No. 6, Xu, is the only order giving a "double precision result". The first thirty-nine most significant bits of the product are in the register R1, the thirty-nine least significant bits are in R2. The register R2 holds forty bits, but the first bit, which is always zero, is disregarded. Thus the multiplication of two numbers gives a seventy-eight-bit product. Addition, subtraction, and division give thirty-nine-bit results.

d) Order No. 7, ÷ , gives the quotient in R2. The last bit (e_{39}, the least significant bit) of the quotient is always "1" (binary one).

For example, 0.01 ÷ 0.1 = 0.1, but the register R2 will show, 0100 0000 0000 0000 0000 0000 0000 0000 0000 0001, introducing an error of 2^{-39}. The division is said to be "a round-off" division. The remainder, is "shifted left one", which is equivalent to being multiplied by 2, = 10(2), and is held in R1.

Order No. 11 belongs to the category of "transfer orders". A transfer order is not automatically followed by the next order in sequence but by the order in the memory position specified in the address part of the transfer order. For example, Order No. 11, U BI, directs control to

the left (the first) order which is stored in memory position B1. The Orders No. 1 through No. 10 are not transfer orders and any one of these orders is automatically followed by the next order in the sequence. No order except a transfer order specifies the address of the order to follow.

Now let us return to the example. The programming results in a flow chart. When the flow chart is completed we can begin the preliminary coding, patterned directly upon the flow chart.

Preliminary Coding for Computing $a_1 + a_2$.

Sequence (Box on the flow chart)	Code (sexadecimal order)	Word (address of the word)	Order (preliminary symbol)	R1 (contents of)	R2 (contents of)	Memory (contents of)	Description
I, 1	word { K4...		+A1	a_1			(A1)→R1
2	N4...		(+)A2	a_1+a_2			(R1)+(A2)→R1
3	word { 10...		M A3			a_1+a_2	(R1)→A3
II, 1	S4...		R A3		a_1+a_2		(A3)→R2
2	word { L4028		P				Print (R2)
3	00000		Zu				Stop

Notes on the Preliminary Coding.

a) The above program can be explained in words as follows:

Order I, 1; (A1) = a_1 is duplicated in R1, after this order is performed (R1) = a_1.

Order I, 2, (R1) = a_1 is added to (A2) = $+a_2$; after this order is performed (R1) = a_1+a_2.

Order I, 3, (R1) = a_1+a_2 is duplicated in A3, after this order is performed (A3) = a_1+a_2.

Order II, 1, (A3) = a_1+a_2 is duplicated in R2, after this order is performed (R2) = a_1+a_2.

Order II, 2, (R2) = a_1+a_2 is printed in the teleprinter (the output device) in sexadecimal form.

Order II, 3, The machine stops operation.

b) The orders are written in pairs : I, 1 and I, 2 ; I, 3 and II, 1, II, 2 and II, 3. Each pair forms one instruction word to be stored in one memory position. The whole program consists of three words.

c) The column under the heading "Sequence" lists the numerical sequences of orders of the corresponding boxes. The Roman numerals refer to boxes on the flow chart.

The second column under the heading "Code" contains sexadecimal representations of the orders. The instruction part of an order consisting of the first two sexadecimal digits is immediately available from the list. The address part, the last three sexadecimal digits, is to be inserted in the final coding after the addresses have been assigned.

The third column under the heading "Word" has the addresses of the instruction word and is also filled in the final coding after the addresses have been assigned. The columns under the heading "R1", "R2", "Memory" have the contents of R1, R2 and Memory respectively, where the results of the corresponding orders are recorded.

d) The sum $a_1 + a_2$ was actually obtained after the order I, 2, but we needed three more orders to accomplish its printing, since the design of ORDVAC permits the printing of results in the teletype output device only from the register R2 (see ORDVAC diagram).

The Assignment of Addresses. The next step after the preliminary coding is to assign addresses for instruction words, initial datum numbers, and temporary positions. As each address is assigned it will prove convenient to record this fact by shading the appropriate space on the raster sheet to remind the coder that it is no longer available. The space alloted to the input routine must also be shaded. In the normal method of operation by the ORDVAC, when transfer orders do not interrupt, the left order of an instruction word at address B1 of the memory is

executed, followed by the right order of that word, then the left order of the word at the next address B2 (next in the sequence of addresses), etc. The ordinary procedure in assigning addresses would be to start from some convenient unused address and choose the following addresses in their natural increasing order until it becomes necessary, as in more complicated problems than the present example, to skip out of this ordering by means of a transfer. Figure 8 shows a part of a raster with assigned memory positions for the input routine, numerical data and the words of the routine for computing a_1+a_2.

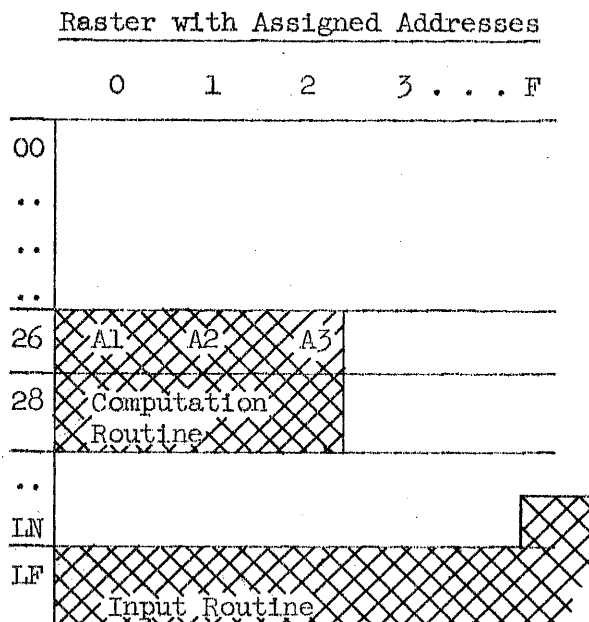


Figure 8

Notes on Figure 8.

a) The memory positions LNF through LLL are used by the input routine. (Fifty memory positions).

b) The number a_1 is stored at the address 260(16), the number a_2 at 261(16). In the preliminary coding the addresses of a_1 and a_2 were represented by symbols A1 and A2 respectively.

c) In the preliminary coding we used a storage position whose address was represented by the symbol A3. For this position we assign the address 262(16).

d) The three instruction words of the computation routine are assigned the following addresses in sexadecimal representation: 280, 281, 282.

The Final Coding. Having assigned the sexadecimal addresses we can now go back to the preliminary coding, Figure 7, and complete the final coding by filling the second and the third columns under the headings "Code" and "Word".

Final Coding for Computing a_1+a_2

Sequence	Code	Word	Order	R1	R2	Memory	Description
I, 1	K4260	280	+ A1	a_1			(A1) → R1
2	N4261		(+)A2	a_1+a_2			(R1) + (A2) → R.
3	10262	281	M A3			a_1+a_2	(R1) → A3
II, 1	S4262		R A3		a_1+a_2		
2	L4028	282	P				Print (R2)
3	00000		Zu				Stop

Figure 9

a) The first instruction word of the routine consisting of the first two orders is K4260N4261, and it is to be stored at the address 280. The second instruction word, 10262S4262, is to be stored at the address 281. The third instruction word, L402800000, is to be stored at the address 282.

Key-Words. Coded routines are inserted into ORDVAC's memory by means of an input routine which has several optional modes of operation. In general, a coded program can be subdivided in a natural way into groups of words at consecutive addresses such that all members of the same group require the same treatment. The input routine has been so constructed that a group of consecutive words can be subjected to a common input operation by inserting a special "key word" for that operation just ahead of its group. It should be remarked that this insertion is actually necessary only on the tape or punched card input: if a key-word is written on the final coding sheets, no address is to be assigned to it, that is, the key-word itself is not to be placed in the memory.

In this Chapter we shall require only the two following key-words. A complete list is given in the Appendix.

Short List of Key Words

	Symbol	Explanation of the "pseudo"-instruction
1.	80000 00... (A1)	Store the first word of the computation routine at the address A1 and continue storing in consecutive memory positions the words which follow, until the next key-word appears.
2.	80001 00... (B1)	Start operations beginning ~~from~~ with the left order of the instruction word stored at the address B1.

Figure 10

The first key word from the above list precedes the first word of the computation routine, the second key word follows the last word of the computation routine. The computation routine of our problem of computing a_1+a_2 together with the key words would look as follows:

Coded Program for Computing a_1+a_2

Key word	80000 00280	key word
1st word	K4260 N4261	Computation routine
2nd word	10262 S4262	
3rd word	L4028 00000	
Key word	80001 00280	Key word

Figure 11

The coded program in Figure 11 reads as follows:

Store the first word of the computation routine at the address 280, and continue storing the words which follow at succeeding addresses, 281 and 282 for the second and third words respectively. When the third instruction word is stored, then the machine is instructed by the key word which follows the last word to start operations beginning ~~from~~ with the left order of the first instruction word stored at the address 280. Operations will then continue, and the machine receives and executes orders from consecutive addresses. The orders will be executed in the following sequence: left and then right from the address 280, left and right from 281, left and right from 282.

Card Punching. The coded program in Figure 11 is not yet in a form acceptable by the machine. The last steps consist of punching the coded program on cards or on teletype tape. In ORDVAC practice the tape is very seldom used and we shall discuss only the card input. A stack of cards in their proper order ready for the machine is referred to as the "deck". The deck, arranged with the computation routine on top of the input routine (input routine comes first) is the final form of the coded program ready for the input device. The preparation of a deck will be explained in the next chapter.

Preparing a Problem for Machine Computation. Summary. Suppose that a problem has been formulated mathematically and that a numerical method has been selected to solve it. Then the preparation of the problem for the machine consists of the following steps:

1. Programming. The course of the computations is planned and the coding procedures to be used are selected. The final result of the programming is a flow chart. In principle the general outlines of programming will not be limited to any one particular machine. In practice, the special features of a given machine may influence the details of the planning.

2. Coding. In coding, or in translating the flow chart into the language of the particular machine, a sequence of machine words is formulated to represent a sequence of operations. Coding can be subdivided into three distinct steps:

a) Preliminary coding, using preliminary representation of orders and symbolic addresses.

b) Assigning memory addresses.

c) Final coding, using sexadecimal representation of orders and actual addresses.

3. Card Punching and Transcribing. These will be explained in the next chapter. Except for the card punching and transcribing, every step listed above was described for the problem of computing the sum of a_1+a_2. We shall illustrate the preparation of a problem for machine computation through another example, that of finding the value of : z = xy-b.

1. Programming. Assume that the datum numbers, x, y, b, xy and z are already proper fractions; hence their reduction to fractional form does not require special attention. The input routine will store x, y, and b. The problem is very simple and programming involves only the preparation of a flow chart.

Flow Chart for Computing, xy-b.

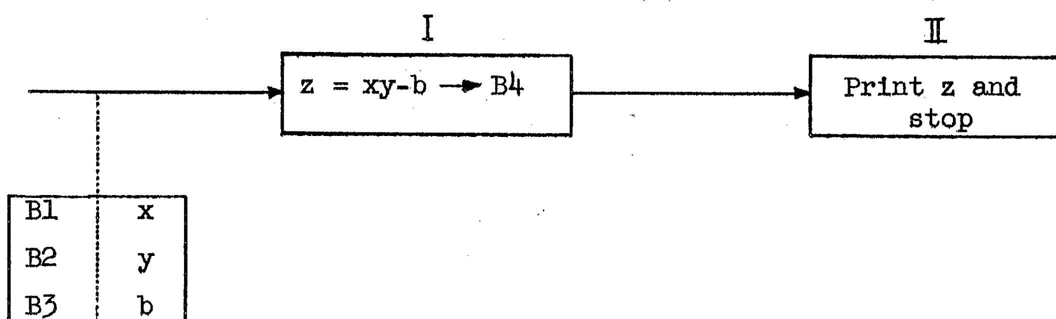


Figure 12

2. Coding.

a) Preliminary coding for computing, xy-b

Sequence	Code	Word	Order	R1	R2	Memory	Description
I, 1			R B1		x		(B1) → R2
2			Xu B2	xy			(R2)x(B2) → R1
3			(-) B3	xy-b			(R1)-(B3) → R1
4			M B4			xy-b	(R1) → B4
II, 1			R B4		xy-b		(B4) → R2
2			P				Print (R2)
3			Zu				Stop

b) Assigning Addresses

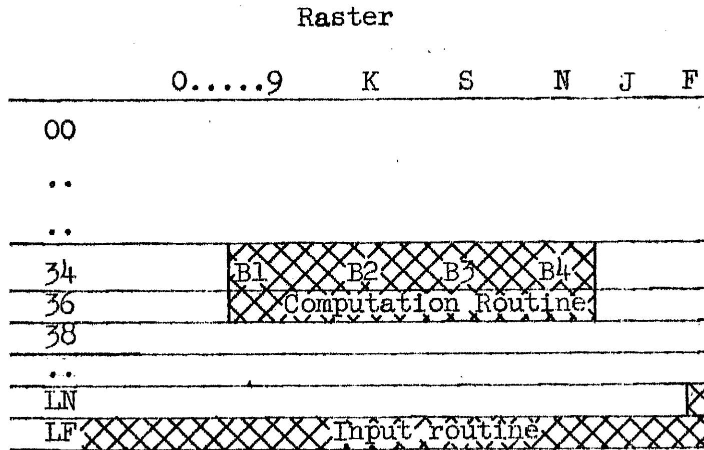


Figure 13

c) Final Coding for Computing xy-b.

Sequence	Code	Word	Order	R1	R2	Memory	Description
I, 1	S4349	369	R B1				
2	K834K		Xu B2				
3	0434S	36K	(-)B3				
4	1034N		M B4				
II, 1	S434N	36S	R B4				
2	L4028		P				
3	00000	36N	Zu				
4	00000						

Notes on the final coding.

1. For the sake of exposition we used separate forms (coding sheets) for preliminary and final coding. In practice the same form is used for both.

2. The last order, II, 3, is a left order not followed by the right order. In this case we make the right order 00000, or anything else we like.

The computation routine, with the key words, would appear as follows:

80000	00369
S4349	K834K
0434S	1034N
S434N	L4028
00000	00000
80001	00369

The above sequence can of course be read directly from the coded program form and does not need to be re-written in the manner shown above. We presented it here for clarity.

3. Card Punching and Transcribing. Preparation of the deck will be explained in the next chapter.

Exercises: Prepare the machine computation for each of the following problems: (the results should be printed).

1. a + b - c (each of the numbers, a, b, and c and all partial sums in abs. value are less than 1)
2. xyz (each of the numbers, x, y, and z in abs. value is less than 1)
3. x + y - xy(each of the numbers x and y in abs. value is less than 1/2)

CHAPTER III

CARD PUNCHING AND TRANSCRIBING. CONVERSION. RECONVERSION. TABULATION. PUTTING A PROBLEM ON THE MACHINE

Preparing a Routine for Card Punching. In the example of computing a_1+a_2, which we used in the second chapter, the computation routine with the key words was as follows:

	Left order	Right order	
1st word	80000	00280	Key word
2nd word	K4260	N4261	Instruction words
3rd word	10262	S4262	Instruction words
4th word	L4028	00000	Instruction words
5th word	80001	00280	Key word

To prepare the routine for card punching we insert, in every word, a "0" after the second and the seventh digit. In case of an instruction word we insert a "0" after the second digit of each order. Thus, every order will have six digits instead of five, and every word will have twelve digits instead of ten. The reason for doing this will be explained in Chapter VII. The twelve-digit word routine is shown below:

1st word	800000 000280	Key word
2nd word	K40260 N40261	Instruction words
3rd word	100262 S40262	Instruction words
4th word	L40028 000000	Instruction words
5th word	800001 000280	Key word

Sexadecimal Cards. An IBM card, shown in Figure 1, has twelve horizontal rows labeled Y,X,0,1,2,3,4,5,6,7,8,9, and eighty vertical columns labeled at the bottom of the card by numerals 1 through 80 respectively. For sexadecimal punching, the eighty columns are considered as divided into six groups of twelve columns and one group of eight columns. The last eight column group is not used. As is shown by the labels inserted on the upper margin of the card in Figure 2, the first twelve column group corresponds to the first twelve digit word, the second group to the second word and so on. An IBM card with the above grouping and correspondence is called "a sexadecimal card". The twelve-

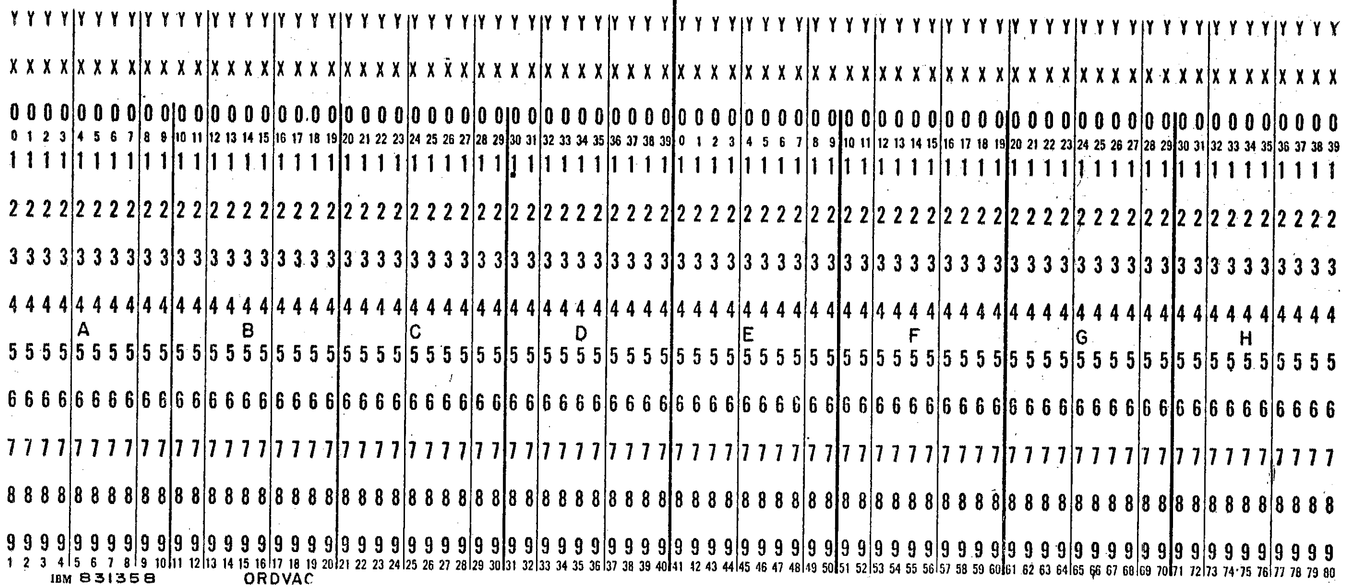


Fig. 1

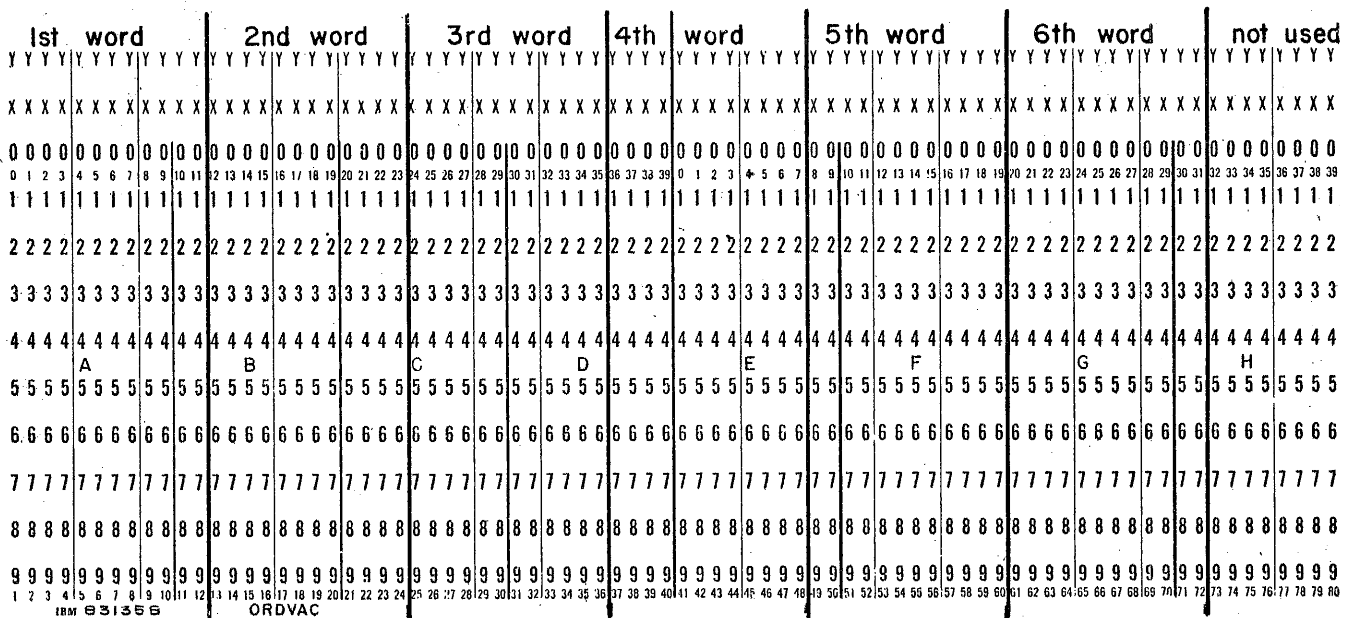


Fig. 2

columns of a group correspond respectively to the twelve digits of a word. In counting columns and digits from the left, the first column corresponds to the first digit, the second column to the second digit, and so on. If the digit is 0,1,2,3,4,5,6,7,8, or 9,we punch 0,1,2,3,4, 5,6,7,8, or 9 in the corresponding column. For example,if the first digit of a word is 0, we punch "0" in the first column, or again if the twelfth digit is 5, we punch "5" in the twelfth column. However,if a digit is a K, S, N, J, F, or L, we punch two characters in the corresponding column, using the following key:

K(=10)	is	represented	by	punches	of	X and 2
S(=11)	"	"	"	"	"	0 and 2
N(=12)	"	"	"	"	"	X and 5
J(=13)	"	"	"	"	"	X and 1
F(=14)	"	"	"	"	"	Y and 6
L(=15)	"	"	"	"	"	X and 3

Figure 3 shows the representations of the digits K through L, and also the digits 3 and 7.

Y	Y	Y	Y	[Y]	Y	Y	Y
[X]	X	[X]	[X]	X	[X]	X	X
0	[0]	0	0	0	0	0	0
1	1	1	[1]	1	1	1	1
[2]	[2]	2	2	2	2	2	2
3	3	3	3	3	[3]	[3]	3
4	4	4	4	4	4	4	4
5	5	[5]	5	5	5	5	5
6	6	6	6	[6]	6	6	6
7	7	7	7	7	7	7	[7]
8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9
K	S	N	J	F	L	3	7

Figure 3

Figure 4 shows the sexadecimal card of the coded program for computing a_1+a_2. For the sake of illustration the sexadecimal words are typed at the top of each twelve column group.

Sexadecimal Card for Computing a_1+a_2

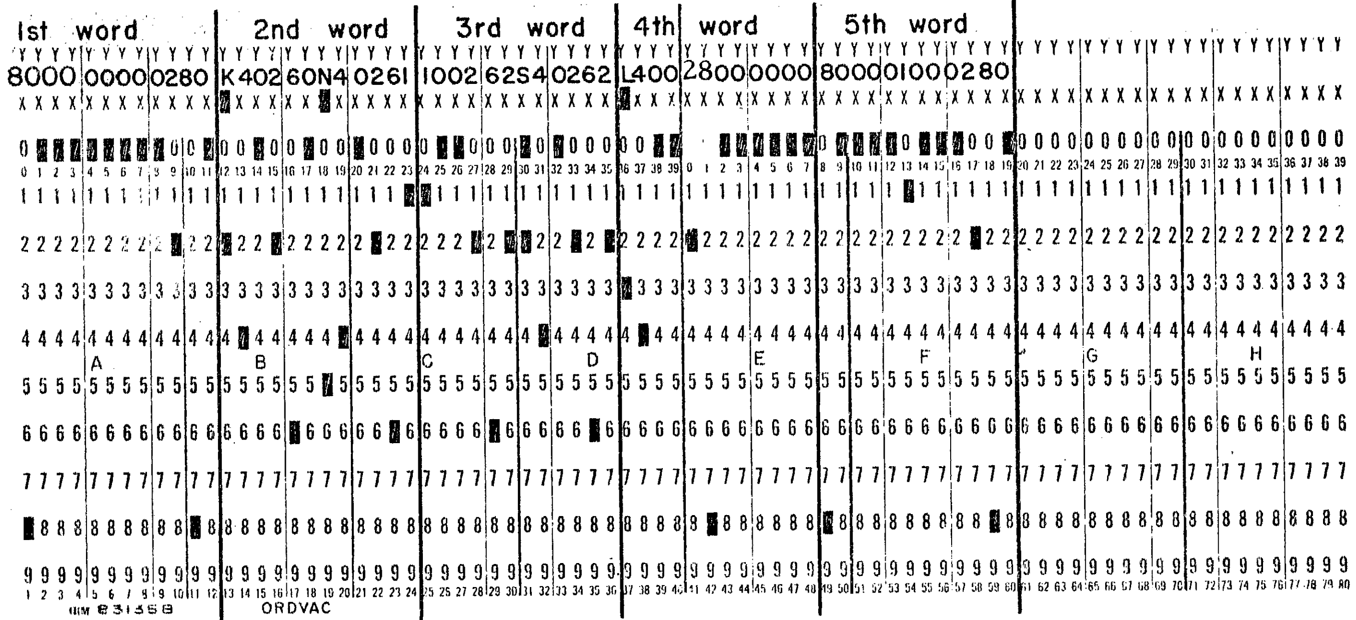


Figure 4

The sexadecimal cards are punched by an operator using an IBM machine called "the manual punch".

Transcribing. One of the input devices is an IBM machine called the "card reader". The reader "reads" the cards by translating the punched holes into pulses of electric current which enter the computer. One of the output devices is the teleprinter, referred to previously. Another output device is an "automatic card punch". The output results can be printed on the teleprinter or punched on IBM cards.

The reader could "read" the sexadecimal cards, but the input routine does not interpret sexadecimal cards because it is designed to interpret the information in binary form. Thus, the sexadecimal cards have to be replaced by "binary cards" with binary word representation. For this

reason the routine on sexadecimal cards is translated into binary form and punched automatically on binary cards by "transcribing". The punching of binary cards is automatic in the sense that it is carried out by the computer itself. A special routine called "the transcriber routine" is stored in the memory and directs the computer to interpret the pulses from sexadecimal cards in such a way that the automatic punch in the output device will punch the computation routine on binary cards. The transcriber routine will be explained in Chapter VII. The method of transcribing can be summarized in the following instructions:

1. Stack the following in the reader: (a) the deck of the transcriber routine, (b) the deck of the sexadecimal computation routine and (c) a blank card. (All cards are stacked face down, y edge in).

2. Read in both decks. (See Operating instructions, next chapter).

3. Collect the deck of punched binary cards from the automatic punch output device.

The deck of binary cards represents the final form of the coded program, the form acceptable by the input routine.

Binary Cards. For binary punching, the eighty vertical columns of an IBM card are regarded as divided into two groups of forty columns (See Figure 5). In each group, each row of forty characters corresponds to a forty-bit word. In Figure 5 on the extreme left and right margins of the card, the rows in each group are labeled K, M, W1, W2, and so on, showing the ordering of the words. The first word, W1, corresponds to the "X"s row in the first group, the second word, W2, corresponds to the "X"s row in the second group, and so on. In the "Y"s row the word labeled K is a "key word", the word labeled M is the "modifying word". These two words will be explained later. An IBM card with the above grouping and correspondence is called a "binary card". On a binary card we can represent twenty-four 40-bit words (K, M, W1-W22).

Binary Card

1st group | 2nd group

K ... M
W1 ... W2
W3 ... W4
W5 ... W6
W7 ... W8
W9 ... W10
W11 ... W12
W13 ... W14
W15 ... W16
W17 ... W18
W19 ... W20
W21 ... W22

IBM 831356 ORDVAC

Figure 5

A punch of a character in a given row represents a binary "1", no perforation represents a binary "0". For example, if the "4"s row in the 1st group is punched in the following way:

4444 4444 4444 4444 4444 4444 4444 4444 4444 4444, then it represents the binary word, W11

0100 1000 0100 1100 1000 0000 0000 0010 0000 0100.

Figure 6 shows the binary card of the coded program for computing a_1+a_2.

Binary Card for Computing a_1+a_2.

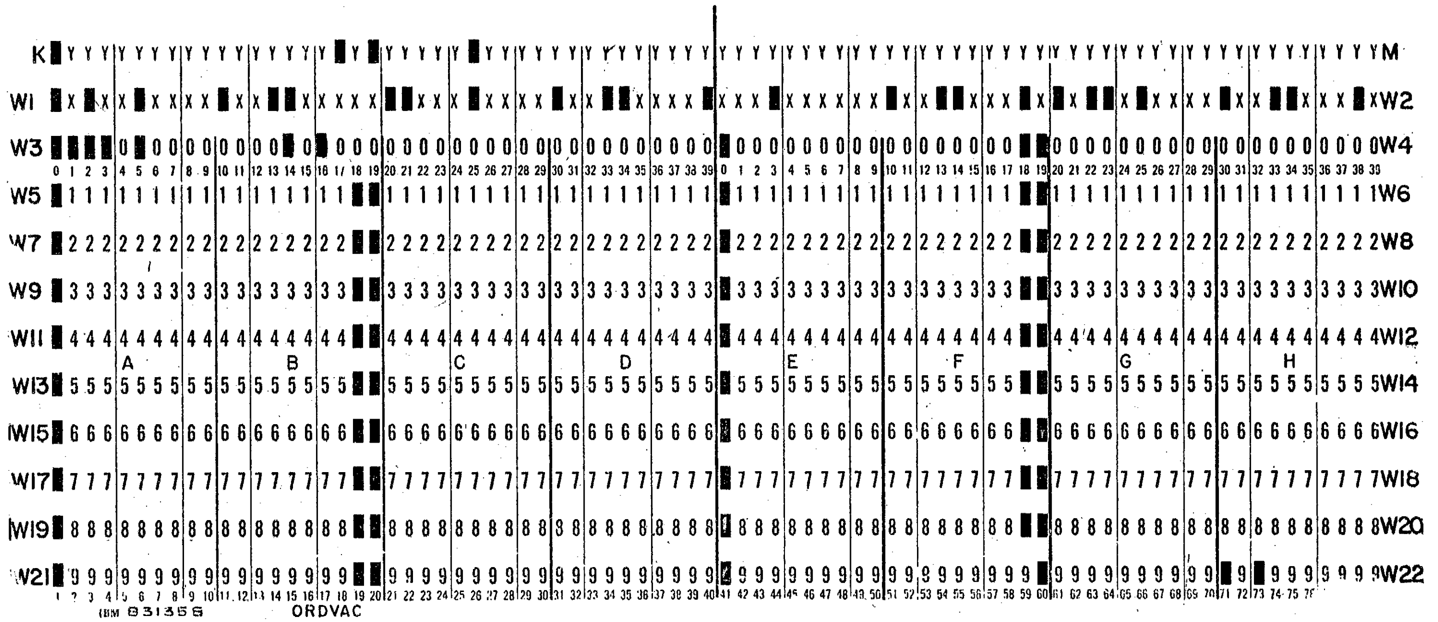


Figure 6

Notes on Figure 6.

a) The binary words on the card read as follows:

	Binary	Sexadecimal Equivalent
K	1000 0000 0000 0000 0101 0000 0100 0000 0000 0000	80005 04000
M	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	00000 00000
W1	1010 0100 0010 0110 0000 1100 0100 0010 0110 0001	K4260 N4261
W2	0001 0000 0010 0110 0010 1011 100 0010 0110 0010	10262 S4262
W3	1111 0100 ~~0010 1000 0000~~ (0000 0010 1000) 0000 0000 0000 0000 0000	L4028 00000
W4	1000 0000 0000 0000 0011 0000 0000 0000 0000 0000	80003 00000
	same as W4	
	same as W4	
W21	same as W4	
W22	1000 0000 0000 0000 0001 0000 0000 0010 1000 0000	80001 00280

b) In Figure 7 both the sexadecimal and binary cards for computing $a_1 + a_2$ are displayed together to compare them, to verify where there is a correspondence between them, and to discover where the correspondence may be lacking.

1st word | 2nd Word | 3rd word | 4th word | 5th word | (SEXADECIMAL CARD)

8000 0000 0280 | K402 60N4 0261 | 1002 62S4 0262 | L400 2800 0000 | 8000 0100 0280

IBM 831558 ORDVAC

(BINARY CARD)

IBM 831369 ORDVAC

Figure 7

There is a correspondence between the following words:

Sexadecimal Card				Binary Card
2nd word	K40260 N40261	corresponds to	W1	K4260 N4261
3rd word	100262 S40262	corresponds to	W2	10262 S4262
4th word	L40028 000000	corresponds to	W3	L4028 00000
5th(key)word	800001 000280	corresponds to	W22	80001 00280

There is no correspondence:

On sexadecimal card: 1st (key) word is 800000 000280 and does not correspond to any word on our binary card. The key word 80000 00280 instructs the machine to store the routine words at consecutive addresses beginning from the address 280, therefore it must have somewhere a binary correspondent. It would be found in W22 row (the last row) on a binary card which would immediately precede our card.

On binary card: K (key word) is 80005 04000, and does not correspond to any word on the sexadecimal card. When a computation routine has more cards than one, then every transcribed binary card would have K of the same "5" type, that is, 80005 .... (See Chapter VIII).

M (modifier word) is 00000 00000, and does not correspond to any word on the sexadecimal card. In our case the modifier word has no meaning.

W4 through W21 (dummy key words) are 80003 00000 each and do not correspond to any words on the sexadecimal card. They mean "reject, read the next word", and they fill the gap (if there is any) between the last instruction word,W3, and the key word at the end, which always occupies the last, W22,row on the last binary card.

Remarks on Card Punching and Transcribing. A student learning the complicated procedure of card punching and transcribing may ask why we do not punch words directly on binary cards, instead of punching them on sexadecimal cards and then transcribing them. The reason is, that manual punching of a forty-bit word routine would be inconvenient, would

take too much time, and would lead easily to mistakes. Another question is also pertinent: If the transcriber routine can interpret a sexadecimal computation routine in such a way that the output device punches the computation routine on binary cards, why not design another interpretative routine which would interpret the sexadecimal computation routine as a computation routine and cause the computer to perform the desired operations? Such an interpretative routine, a kind of a special input routine (it would precede the computation routine), would eliminate the need for transcribing, and can be easily designed. The objection against such an input routine is that it must necessarily be long (would have many words) and would take a considerable number of memory positions. In many problems the storage space (number of available memory positions) is critical, and the routines of such problems still would have to be transcribed. Besides, a sexadecimal card contains only 6 words, a binary card 24 words; hence a sexadecimal deck has nearly four times as many cards as a binary deck (nearly four times, and not exactly four times, because a binary card always has the key and the modifier words) and the reading in of a sexadecimal deck would take much more time.

Exercises: Punch and transcribe the routines of the problems 1, 2, and 3 from the second chapter.

Punching Datum Numbers. The punching of datum numbers, to be stored separately from the computation routine, differs from the punching of the computation routine. A decimal datum punch card, shown in Figure 8, looks like a sexadecimal IBM card. It uses six groups of twelve columns and leaves unused one group of eight columns. Each group of twelve columns corresponds to one datum number. The first group in the first card is reserved for a special key word.

The First Decimal Datum Card

Key Word | 1st Number | 2nd Number | 3rd Number | 4th Number | 5th Number

A B C D E F G H

IBM 831358 ORDVAC

Figure 8

The numbers to be punched on a card are expressed as decimal fractions represented by 12 characters, the zero to the left of the decimal point is replaced by:

K, if the number is positive, or by

S, if the number is negative.

For example, the number, 5/13 = 0.384615384615, is represented by, K38461 538461; the number, -5/13 = -0.384615384615, is represented by, S38461 538461. In a given group on a card, every column corresponds to a decimal digit except the first column. For each digit, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, the same numeral in the corresponding column is punched out. For K as before, we punch in the same corresponding column, X and 2, for S, 0 and 2.

For example: K 3 8 4 6 1 5 3 8 4 6 1 , is punched

K	3	8	4	6	1	5	3	8	4	6	1
Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
[X]	X	X	X	X	X	X	X	X	X	X	X
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	[1]	1	1	1	1	1	[1]
[2]	2	2	2	2	2	2	2	2	2	2	2
3	[3]	3	3	3	3	3	[3]	3	3	3	3
4	4	4	[4]	4	4	4	4	4	[4]	4	4
5	5	5	5	5	5	[5]	5	5	5	5	5
6	6	6	6	[6]	6	6	6	6	6	[6]	6
7	7	7	7	7	7	7	7	7	7	7	7
8	8	[8]	8	8	8	8	8	[8]	8	8	8
9	9	9	9	9	9	9	9	9	9	9	9

Transcribing Datum Cards. The words on decimal datum cards are transcribed on binary cards, where all the numbers are converted to the binary machine equivalents (negative numbers are represented by complements). The same transcriber routine transcribes words from sexadecimal cards and from decimal datum cards. The transcriber routine distinguishes sexadecimal words from decimal datum words through the key-words. On the first datum card, the first group of twelve columns is reserved for the following sexadecimal key-word: 800003 $000\overset{C1}{\ldots}$, where C1 is the three digit sexadecimal address indicating where the number which immediately follows the key word will be stored. We shall call this key word "the 30-type." Words which follow the 30-type key word are identified by the transcriber routine as decimal datum numbers as long as no new key-word is encountered. In the problem of computing a_1+a_2, let $a_1 = 5/13 = 0.384615384 61$, represented for punching by K38461 538461, and $a_2 = -2/13 = -0.1538461538461$, represented for punching by S15384 615385. The addresses assigned for the datum numbers a_1 and a_2 were respectively 260 and 261. (See chapter II). The decimal datum card for a_1 and a_2 is shown in Figure 9.

Decimal Datum Card for Storage of a_1 and a_2

Key word | $a_1 = 5/13$ | $a_2 = -2/13$

8000 0300 0260 K384 6153 8461 S153 8461 5385

IBM 831358 ORDVAC

Figure 9

Figure 10 shows the binary datum card for storage of a_1 and a_2 transcribed from the sexadecimal datum card. In transcription, the datum card was immediately followed by the sexadecimal card for computing a_1 and a_2.

IBM 831358 ORDVAC

Figure 10

The binary card reads:

		Sexadecimal Equivalent
K	1000 0000 0000 0000 0101 0000 0000 0010 0110 0000	80005 00260
M	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	00000 00000
W1	0011 0001 0011 1011 0001 0011 1011 0001 0011 1001	313S1 3S139
W2	1110 1100 0100 1110 1000 0100 1110 1100 0100 1101	FN4F8 4FN4J
W3	1000 0000 0000 0000 0011 0000 0000 0000 0000 0000	80003 00000
.	same as W3	
.	same as W3	
.	same as W3	
W21	same as W3	
W22	1000 0000 0000 0000 0000 0000 0000 0010 1000 0000	80000 00280

Comparing the decimal and the binary cards we find the following correspondences:

Decimal Card				Binary Card
1st (Key) word	800003 000260	corresponds to	K	80005 00260
2nd word, a_1	K38461 538461	corresponds to	W1	313S1 3S139
3rd word, a_2	S15384 615385	corresponds to	W2	FN4F8 4FN4J

There is no correspondence:

M (modifier word) is 00000 00000, and does not correspond to any word on the decimal card.

W3 through W21 (dummy key words) are each 80003 00000 and do not correspond to any words on the decimal card. The meaning of dummy key words was explained before. W22 (key word) is 80000 00280 and does not correspond to any word on the decimal datum card. It corresponds though, to the first (key) word on the sexadecimal card for computing a_1+a_2 which in transcription immediately followed the decimal card.

Conversion. In ORDVAC procedures "conversion" means only conversion from decimal to binary representation, and "reconversion" means conversion from binary to decimal representation. In this sense the transcriber routine is a conversion routine with respect to decimal numbers on decimal cards, but it is not a conversion routine with respect to sexadecimal cards although it transcribes the sexadecimal words to binary

words. In addition to the conventional input routine we have also a special conversion input subroutine called "IBM card input subroutine", which converts the decimal datum numbers to equivalent binary words and stores them in the memory. When we use the IBM card input subroutine we do not transcribe the decimal datum cards. The IBM card input subroutine cannot be used for sexadecimal or binary cards. (See Chapter X).

Reconversion. The teleprinter, as an output device, prints the results in sexadecimal representation which is very inconvenient. The automatic card punch, as an output device, punches the binary results on cards, which is also inconvenient. To obtain results in a convenient decimal form we have a reconversion subroutine called "IBM output subroutine". The IBM output subroutine (stored in memory) directs the reconversion, and the results are punched on cards in decimal form.

Tabulation. Reading IBM cards, even the decimal cards, is very cumbersome. For this reason the results are copied from the decimal cards onto paper in neat printed form. The copying of results from the decimal cards, performed automatically, is called "tabulation" and is done by an IBM machine called a "tabulator". Figure 11 shows a sample of tabulated results.

Putting a Problem on the Machine. "Putting a problem on a machine" means making the machine perform the computations which lead to the desired solution of the problem. We shall give all the steps of putting a problem on the machine, together with the steps which precede and which follow.

Preceding Steps:

1. Programming, coding the computation routine, assigning addresses.

2. Checking computation routine and correcting errors which were found.

3. Punching sexadecimal cards for the computation routine and the decimal cards for the datum numbers.

Tabulated Results

20800000	420500000	2252200170	15545580
7200000	389600000	2252200170	23480244
388500000		2252300170	
385900000	31900000	2252300170	1794
373000000	62400000	2252300170	16478
360700000	91900000	2252300170	44111
348900000	120500000	2252300170	82647
337500000	148200000	2252300170	131594
326400000	174900000	2252300170	190831
315600000	200600000	2252300170	260422
304900000	225400000	2252300170	340552
294300000	249100000	2252300170	431486
283800000	271800000	2252300170	533548
271700000	291500000	2252300170	649002
259100000	309300000	2252300170	782821
246800000	326000000	2252300170	934456
234700000	341400000	2252300170	1103234
223000000	355700000	2252300170	1288713
211600000	368800000	2252300170	1490452
200500000	380700000	2252300170	1708410

Figure 11

4. Transcribing the routine and the decimal datum cards (if the use of the IBM input subroutine is not planned).

5. Having ready all the auxiliary subroutines, that is, the decks of the conventional input routine, IBM card input subroutine, IBM output subroutine.

Putting a Problem on the Machine:

If the decimal datum cards have been transcribed to binary cards, then the following cards, in the given order, are stacked in the reader, face down, top edge of the card first:

1. the binary deck corresponding to the conventional input routine;
2. the binary deck corresponding to the transcribed datum numbers;
3. the binary deck corresponding to the IBM output subroutine;
4. the binary deck corresponding to the computation routine;
5. a blank card on top.

If the decimal datum cards have not been transcribed to binary cards, and the use of the IBM input subroutine is planned, the following cards in the given order are stacked in the reader, face down, top edge of the card first:

1. the binary deck corresponding to the conventional input routine;
2. the binary deck corresponding to the IBM input subroutine;
3. the binary deck corresponding to the IBM output subroutine;
4. the binary deck corresponding to the computation routine;
5. the decimal deck corresponding to the decimal data;
6. a blank card on top.

Following Steps:

1. Remove the deck of punched results from the automatic punch output device.

2. Tabulate the results.

Exercises: Prepare to put on the machine with all the preceding and following steps the problems 1, 2, and 3 from the Exercises at the end of the second Chapter. Assume that the datum numbers in decimal representation are:

$$a = 0.1234567$$
$$b = 0.3247687$$
$$c = 0.7869401$$
$$x = 0.9999812$$
$$y = 0.0000074$$
$$z = 0.9191916$$

CHAPTER IV

SHIFT ORDERS
SCALING. CODING SCALED PROBLEMS IN STRAIGHT SEQUENCES.

Scaling. ORDVAC, a fixed point machine, handles only proper fractions (with the formal exception of complements in case of negative numbers). Thus, a programmer has to make a careful analysis of the problem to be computed and, if necessary, to introduce such modifications that all the datum numbers and the results of operations on them are fractions. The above procedure is called "scaling" and the numbers modified in the procedure are said to be "scaled", or "scaled down". The scaling will be explained in an example.

Example. Scale $f(x,y,z) = ax^2 + by^2 + cz^2$, if $a = 1.1$, $b = 0.51$, $c = 2.3$, and the values of x, y, and z vary within the following limits: $0.1 \leq x \leq 1$, $0.2 \leq y \leq 0.9$, $0.3 \leq z \leq 0.8$.

We start from a preliminary examination of the datum numbers and notice that a and c are greater than one, and x may equal one; then we write for each of these datum numbers the greatest factor called the "scaling factor", of the form, $2^{-n} = 0.00...01(2)$, (with n successive digits, 0), which would make each of these datum numbers less than one. Thus:

Datum Number	Initial scaling factor
$a = 1.1$	2^{-1}
$c = 2.3$	2^{-2}
$x = 1$	2^{-1}

The next step is to examine the possible results of operations on the datum numbers. In our problem every term of the polynomial, f, and the sum of terms must be less than one. We write for the greatest value of each term, and for the sum of the terms, the necessary scaling factors. For the time being we do not take into account the initial scaling of a, c, and x. Thus:

Term	Maximum value of a term	Secondary scaling factor
ax^2	$(1.1)(1) = 1.1$	2^{-1}
by^2	$(0.51)(0.81) = 0.413$	none
cz^2	$(2.3)(0.64) = 1.472$	2^{-1}
sum	2.985	2^{-2}

In the final step we must reconcile the initial requirement that a, c, and x have to be scaled and the second requirement that every term must be multiplied by at most 2^{-2} in order to make the sum less than one. (Every term as a whole must have the same eventual scaling factor, otherwise the formal sum would be meaningless). We shall examine now every term, taking each requirement into account simultaneously. Thus:

Unscaled term	Needed Scaling
ax^2	$(2^{-1}a)(2^{-1}x)(2^{-1}x) = 2^{-3}ax^2$
by^2	$= by^2$
cz^2	$(2^{-2}c)(z)(z) = 2^{-2}cz^2$

This last analysis shows that each term has to be multiplied by 2^{-3}. To achieve this several alternatives are open. For example, in the second term we can multiply b by 2^{-3} or we can multiply b and y by 2^{-1}. The general rul is to keep the scaled numbers as close to one as possible. If we choose the scaled datum numbers to be: $a_1 = 2^{-1}a$, $b_1 = 2^{-1}b$, $c_1 = 2^{-3}c$, $x_1 = 2^{-1}x$, $y_1 = 2^{-1}y$, $z_1 = z$, then the problem reduces to: $2^{-3}f(x,y,z) = a_1x_1^2 + b_1y_1^2 + c_1z_1^2$. If we decide that the datum numbers b and y are to be left unscaled, the problem would then be: $2^{-3}f(x,y,z) = a_1x_1^2 + 2^{-3}by^2 + c_1z_1^2$.

There are valid reasons to write the scaled numbers, a_1, b_1, c_1, x_1, y_1, z_1 in terms of the original numbers, a, b, c, x, y, z. Thus, instead of a_1 we write $2^{-1}a$, instead of b_1 we write $2^{-1}b$, instead of c_1 we write $2^{-3}c$, and so on, and the problem is written, $2^{-3}f(x,y,z) = 2^{-3}ax^2 + 2^{-3}by^2 + 2^{-3}cz^2$.

The rule to keep the scaled numbers as close to one as possible stems, of course, from the necessity to keep as many significant bits as possible. For example if due to scaling, the machine number k = 0100 1100 1100 1100 1100 1100 1100 1100 1100 1101 = 0.6(10) has to be multiplied by 2^{-6} = 0.000001(2), then the scaled number $k_1 = 2^{-6}k$ would be 0000 0001 0011 0011 0011 0011 0011 0011 0011 0011 and the six least significant bits would be lost. Such a loss may become so critical that the results obtained are very inaccurate, or even completely misleading.

Because of the alternatives presented, scaling a problem in the most efficient manner is often difficult, but the procedure is essentially the same as that which was demonstrated in the trivial example given above. Skill in scaling can only be acquired in practice, although of course a systematic procedure could be developed to handle problems all of the same structure.

Shift Orders. Scaling involves multiplication by scaling factors which are of the type $2^{-n} = 0.0\overbrace{\ldots\ \ldots}^{n}01(2)$, where $n = 1, 2, 3, \ldots$. Multiplying a binary number by 2^{-n} or by 2^{n} shifts the binary point through n places to the left or to the right, respectively, therefore such a multiplication is called "shifting". For example, 0.875×2^{-5} = 0.111(2) x 0.00001(2) = 0.00000111(2); the binary point was shifted five places to the left. In ORDVAC the shifting takes place in the registers R1 and R2. Let a word, $e_0e_1\ldots e_{39}$, stand in the register R1. (It is understood that the binary point is always after the first bit, e_0). After multiplying this word by 2^{-n} the register R1 will show $e_0e_0\ldots e_0e_1e_2\ldots e_{39-n}$; the most significant bits $e_1e_2\ldots e_{39-n}$ were shifted n places to the right and the n least significant bits were lost from R1. We say that a number, $e_0e_1\ldots e_{39}$, was "shifted right n", if it was multiplied by 2^{-n}, or "shifted left n" if it was multiplied by 2^{n}. The orders for shifting machine numbers are called "shift orders". The most important shift orders are listed and explained below.

List of Shift Orders

Let the initial contents of R1 be $e_o e_1 \ldots e_{39}$, of R2 be $d_o d_1 d_2 \ldots d_{39}$.
$n = 1, 2, 3 \ldots 63$.

No.	Preliminary representation of order	Sexadecimal representation of order	The contents after the shift of R1	The contents after the shift of R2
1	$\longrightarrow$ n	$08\overset{n}{\ldots}$	$e_o \overbrace{e_o \ldots e_o}^{n} e_1 e_2 \ldots e_{39-n}$	$d_o \overbrace{e_{40-n} \ldots e_{39}}^{n} d_1 d_2 \ldots d_{39-n}$
2	$\longleftarrow$ n	$18\overset{n}{\ldots}$	$e_o e_{n+1} \ldots e_{39} \overbrace{0 \ldots 0}^{n}$	$d_n d_{n+1} . d_{39} e_1 e_2 . . e_n$
3	$\xrightarrow{1}$ n	$28\overset{n}{\ldots}$	$\overbrace{00 \ldots 0}^{n+1} 10 . . 0$	$d_o \overbrace{00 \ldots 0}^{n} d_1 d_2 \ldots d_{39-n}$

Notes: The shift orders will be illustrated in two examples of important special cases when the initial contents of R2 is zero.

Let the initial contents of R1 and R2 be:

Example a) (R1) = 0100 1100 1100 1100 1100 1100 1100 1100 1100 1100, equivalent to 0.6(10)

(R2) = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000, that is zero.

$n = 8$

Example b) (R1) = 1011 0011 0011 0011 0011 0011 0011 0011 0011 0100, the binary complement representing -0.6(10).

(R2) = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000, that is zero.

$n = 8$.

Order No. 1, $\longrightarrow$ n, shifts right n the contents of R1; the n least significant bits are lost from R1. The original bits $e_1, e_2, \ldots, e_n$ are replaced by the first bit e_o. Shifts right n the contents of R2, the n least significant bits are lost from R2. The initial bits d_1, $d_2, \ldots, d_n$ are replaced by the bits $e_{40-n}, e_{41-n}, \ldots, e_{39}$, which are the least significant bits lost from R1. The order $\longrightarrow 8$ will change the contents of R1 and R2 as follows:

a) (R1) = $\overbrace{0000\ 0000}^{8}$ 0100 1100 1100 1100 1100 1100 1100 1100 = $0.6x2^{-8}$

last 8 bits from R1

(R2)= $\overbrace{0110\ 0110}$ 0000 0000 0000 0000 0000 0000 0000 0000

b) (R1) = $\overbrace{1111\ 1111}^{8}$ 1011 0011 0011 0011 0011 0011 0011 0011, the binary complement representing $(-0.6)(2^{-8})$.

last 8 bits from R1

(R2) = $\overbrace{0001\ 1010}$ 0000 0000 0000 0000 0000 0000 0000 0000

Order No. 2. ← n, shifts left n the contents of R1, the n most significant bits, $e_1, e_2, \ldots e_n$, are lost from R1. The original bits $e_{39-n}, e_{40-n}, \ldots,$ e_{39} are replaced by zeros. Shifts left n the contents of R2; the first original bit, d_o, is also replaced; the n-1 most significant bits are lost from R2. The original bits $d_{39-n}, d_{40-n}, \ldots, d_{39}$ are replaced by the bits $e_1, e_2, \ldots, e_n$, which are the most significant bits lost from R1. The order ← 8 will change the original contents of R1 and R2 as follows

a) (R1) = 0100 1100 1100 1100 1100 1100 1100 1100 $\overbrace{0000\ 0000}^{8}$ = The binary fractional part of, $(0.6)(2^8)$

(R2) = 0000 0000 0000 0000 0000 0000 0000 0000 $\overbrace{1001\ 1001}^{8}$

b) (R1) = 1011 0011 0011 0011 0011 0011 0011 0100 $\overbrace{0000\ 0000}^{8}$ = the binary fractional part representing the complement of $(-0.6)(2^8)$.

(R2) = 0000 0000 0000 0000 0000 0000 0000 0000 $\overbrace{0110\ 0110}^{8}$

Order No. 3. →1→ n, clears to zeros the register R1 and puts a binary one in the (n+1)st place after the binary point, that is, it generates the number $2^{-(n+1)}$ in R1. Shifts right n the contents of R2, the n least significant bits are lost from R2. The original bits $d_1, d_2 \ldots . d_n$, are replaced by zeros.. The order →1→ 8 will result in:

a) (R1) = 0000 0000 0100 0000 0000 0000 0000 0000 0000 0000 = 2^{-9}
(R2) = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

b) (R1) = 0000 0000 0100 0000 0000 0000 0000 0000 0000 0000 = 2^{-9}
(R2) = 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

General Remarks on Shift Orders. Shift orders contain no address-parts. In the place usually filled by the address we put the sexadecimal equivalent of n. Examples:

Preliminary representation of order		sexadecimal representation of order
$\longrightarrow$ n		08...^{n}
$\longrightarrow$ 9	9(10) = 9(16)	08009
$\longrightarrow$ 10	10(10) = K(16)	0800K
$\longrightarrow$ 16	16(10) = 10(16)	08010
$\longrightarrow$ 63	63(10) = 3L(16)	0803L

Exercises: Given initial contents in sexadecimal representation of:

(R1) = L5202 K03L1, (R2) = L2000 00000, (A1) = L4121 40211

Find (R1) and (R2) after the completion of the following orders, or sequences of orders:
a) $\longleftarrow$ 2
b) $\longrightarrow$ 1
c) $\longleftarrow$ 1, $\longrightarrow$ 1, M A1

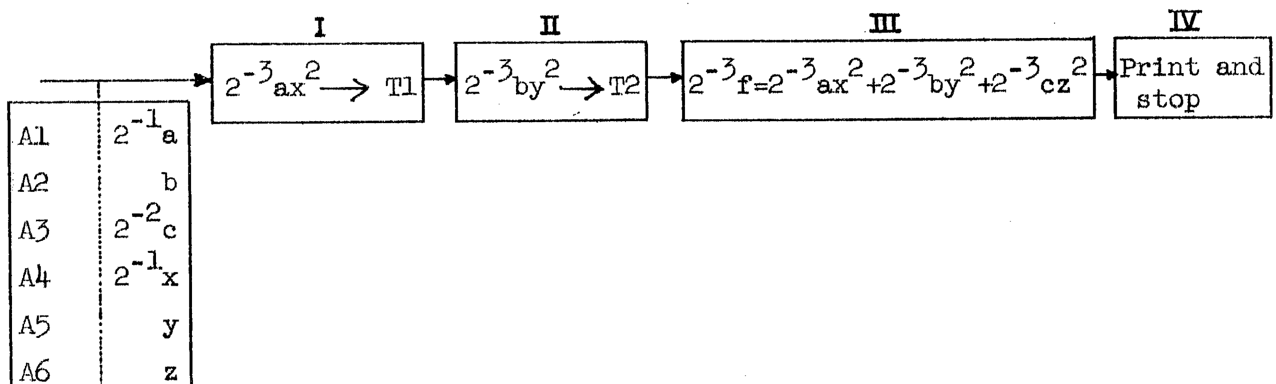
Coding Scaled Problems.

Example 1. Prepare for machine computation, $f(x,y,z) = ax^2 + by^2 + cz^2$. for a : 1.1, b = 0.51, c = 2.3, x = 1, y = 0.9, z = 0.8.

1. Programming.

a) Scaling. This problem was scaled in the previous article with the following results: datum numbers are: $2^{-1}a$, b, $2^{-2}c$, $2^{-1}x$, y, z, the formula is: $2^{-3}f(x,y,z) = 2^{-3}ax^2 + 2^{-3}by^2 + 2^{-3}cz^2$.

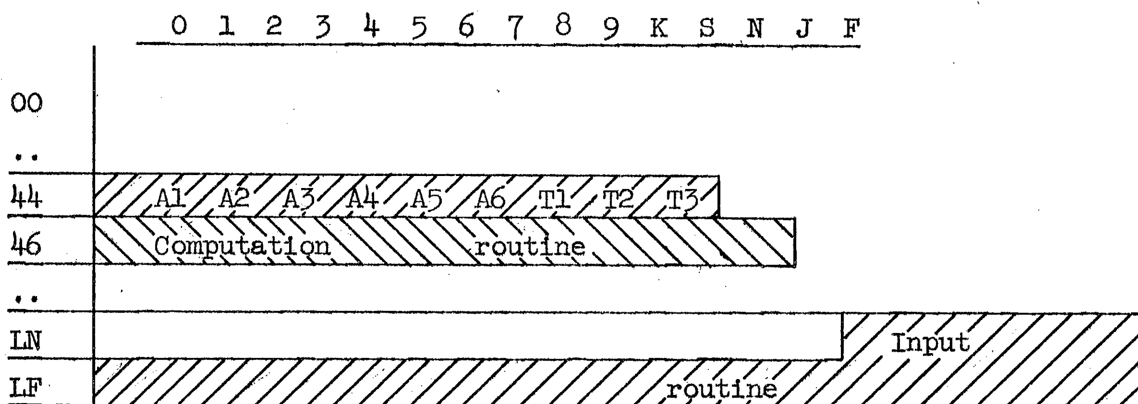
b) Flow chart.



2. Preliminary and final coding.

Sequence	Code	Word	Order	R1	R2	Memory	Description
I, 1	S4440	460	R A1		$2^{-1}a$		(A1) → R2
2	K8443		Xu A4	$2^{-2}ax$			(R2)(A4) → R1
3	10446	461	M T1			$2^{-2}ax$	(R1) → T1
4	S4446		R T1		$2^{-2}ax$		(T1) → R2
5	K8443	462	Xu A4	$2^{-3}ax^2$			(R2)(A4) → R1
6	10446		M T1			$2^{-3}ax^2$	(R1) → T1
II, 1	S4441	463	R A2		b		(A2) → R2
2	K8444		Xu A5	by			(R2)(A5) → R1
3	10447	464	M T2			by	(R1) → T2
4	S4447		R T2		by		(T2) → R2
5	K8444	465	Xu A5	by^2			(R2)(A5) → R1
6	08003		→ 3	$2^{-3}by^2$			(R1)(2^{-3}) → R1
7	10447	466	M T2			$2^{-3}by^2$	(R1) → T2
III, 1	S4442		R A3		$2^{-2}c$		(A3) → R2
2	K8445	467	Xu A6	$2^{-2}cz$			(R2)(A6) → R1
3	10448		M T3			$2^{-2}cz$	(R1) → T3
4	S4448	468	R T3		$2^{-2}cz$		(T3) → R2
5	K8445		Xu A6	$2^{-2}cz^2$			(R2)(A6) → R1
6	08001	469	→ 1	$2^{-3}cz^2$			(R1)(2^{-1}) → R1
7	N4446		(+) T1	$2^{-3}cz^2+2^{-3}ax^2$			(R1)+(T1) → R1
8	N4447	46K	(+) T2	$2^{-3}f(x,y,z)$			(R1)+(T2) → R1
IV, 1	10448		M T3			$2^{-3}f(x,y,z)$	(R1) → T3
2	S4448	46S	R T3		$2^{-3}f(x,y,z)$		(T3) → R2
3	I4028		P				Print (R2)
4	00000	46N	Zu				Stop
	00000						

3. Assigning addresses.



3. Punching and Transcribing. This is left to the student, as an exercise.

Example 2. Prepare for machine computation, $f(x,y) = (x^2 + xy + y^2) \div (x-y)$, if the datum numbers x and y vary as follows: $0.5 \leq x \leq 1.1$, $0 \leq y \leq 2.1$, and, $1 \leq |x-y| \leq 1.6$.

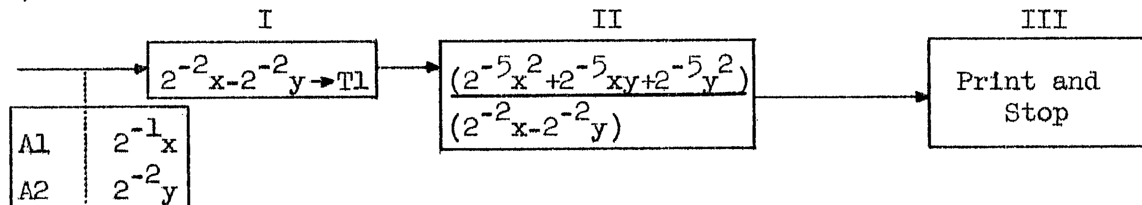
1. Programming.

a) Scaling:

Datum numbers, terms, dividend and divisor	Greatest Scaling Factor	Scaled datum numbers and Scaled Formula
maximum $x = 1.1$	2^{-1}	$2^{-1}x$
maximum $y = 2.1$	2^{-2}	$2^{-2}y$, hence, $2^{-2}(x-y)$
maximum $x^2 = 1.21$	2^{-1}	
maximum $xy = 2.31$	2^{-2}	
maximum $y^2 = 4.41$	2^{-3}	
using the scaled datum		
$(2^{-2}y)(2^{-2}y) = 2^{-4}y^2$	2^{-4}	
maximum dividend, $x^2+xy+y^2=7.93$	2^{-3}	$2^{-4}(x^2+xy+y^2)$, although the greatest scaling factor is 2^{-3}, since the y^2 term alone must be multiplied by 2^{-4} and thus the whole dividend is multiplied by 2^{-4}.
maximum divisor $\|x-y\| = 1.6$	2^{-1}	
minimum divisor $\|x-y\| = 1$	2^{-1}	
max $f(x,y)$=(max divid) $\div$ (min divis) = 7.93	2^{-3}	

The above analysis shows that $f(x,y)$ must have the scaling factor 2^{-3}, the divisor must have the scaling factor 2^{-2}, hence the scaling factor of the dividend cannot be greater than 2^{-5}. The scaled problem is as follows:
Datum numbers are: $2^{-1}x$, $2^{-2}y$,
the formula is: $2^{-3}f(x,y) = (2^{-5}x^2+2^{-5}xy + 2^{-5}y^2) \div (2^{-2}x-2^{-2}y)$.

b) Flow chart.



Note: As a rule it is convenient to form and store the divisor first, which we decided to show in a separate Box I. The coding procedure will explain why.

2. Preliminary Coding.

Sequence	Order	R1	R2	Memory	Description
I, 1	+ A1	$2^{-1}x$			(A1) $\rightarrow$ R1
2	$\longrightarrow$ 1	$2^{-2}x$			(R1)(2^{-1}) $\rightarrow$ R1
3	(-)A2	$2^{-2}x-2^{-2}y$			(R1)-(A2) $\longrightarrow$ R1
4	M T1			$2^{-2}(x-y)$	(R1) $\rightarrow$ T1
II, 1	R A1		$2^{-1}x$		(A1) $\longrightarrow$ R2
2	Xu A1	$2^{-2}x^2$			(R2)(A1) $\longrightarrow$ R1
3	$\longrightarrow$ ·3	$2^{-5}x^2$			(R1)(2^{-3}) $\longrightarrow$ R1
4	M T2			$2^{-5}x^2$	(R1) $\longrightarrow$ T2
5	R A1		$2^{-1}x$		(A1) $\longrightarrow$ R2
6	Xu A2	$2^{-3}xy$			(R2)(A2 $\longrightarrow$ R1
7	$\longrightarrow$ 2	$2^{-5}xy$			(R1)(2^{-2}) $\longrightarrow$ R1
8	M T3			$2^{-5}xy$	(R1) $\longrightarrow$ T3
9	R A2		$2^{-2}y$		(A2) $\longrightarrow$ R2
10	Xu A2	$2^{-4}y^2$			(R2)(A2) $\longrightarrow$ R1
11	$\longrightarrow$ 1	$2^{-5}y^2$			(R1)(2^{-1}) $\longrightarrow$ R1
12	(+) T2	$2^{-5}y^2+2^{-5}xy$			(R1)+(T2) $\longrightarrow$ R1
13	(+) T3	$2^{-5}(x^2+xy+y^2)$			(R1)+(T3) $\longrightarrow$ R1
14	÷ T1		$2^{-3}f(x,y)$		(R1)÷(T1) $\longrightarrow$ R2
III, 1	P				Print (R2)=$2^{-3}f(x,y)$
2	Zu				Stop

3 and 4 The assignment of addresses and the final coding, card punching and transcribing. This is left to the student as an exercise.

Exercises. Prepare for machine computation the following problems:

1. $f(x,y,z)=(2x^3+0.8y^3-0.9xyz)\div(3x^2-y^2)$

2. $f(x,y,z)=(xy+2yz-3xz-2)(x^2+y^2-3)$, in both cases, $1\leq x\leq 2$, $-1\leq y\leq 1.5$, $0\leq z\leq 0.9$.

CHAPTER V

CONTROL UNIT FRONT PANEL OPERATING INSTRUCTIONS.

"To put a coded problem on the machine", or "to read-in a routine", means to make the machine perform operations prescribed by a routine, (see Chapter III). The operating procedures of reading-in a routine require reference to the control unit. The control unit, which is the mechanism causing the operations to occur in the desired sequence, consists of three components:

1. the control counter,
2. the order register, R_3 (read, R lower three),
3. the control panel.

The neon display of R_3 and the control panel are on the ORDVAC front panel. Figure 1 shows a diagram of the ORDVAC front panel. The front panel has also neon displays of R1, R2, and R^3, as well as the panels of: "Address 'A' Halt", "Memory Display", and "Neon Gating", which will be explained later. Figure 2 shows the control panel with all the switches, neons, and in particular the switches S5 and neon displays N5 of the control counter.

Control counter. "The control counter" is actually at the back of the machine but we shall deal only with the switches S5 and neon displays N5 of the control counter, and these are on the control panel. In referring to the control counter we shall actually mean the switches and neon displays of the control panel. The twelve neon displays N5 of the control counter automatically record at any instant, the address of the instruction word which the machine is set to perform next. A "neon on" means a binary "1", a "neon off" means a binary "0". For example the following combination of lights: o o ● o ● o o o o o o o, represents the binary number 0 0 1 0 1 0 0 0 0 0 0 0, which sexadecimally is 280, indicates that the next order pair is stored at the address 280. When the machine is not running the control counter can be set manually by the switches S5. A "switch up" lights the associated neon, thus setting a binary "1", a "switch down" extinguishes the associated neon, thus setting a binary "0". The twelve neons can be cleared (extinguished)

ORDVAC FRONT PANEL

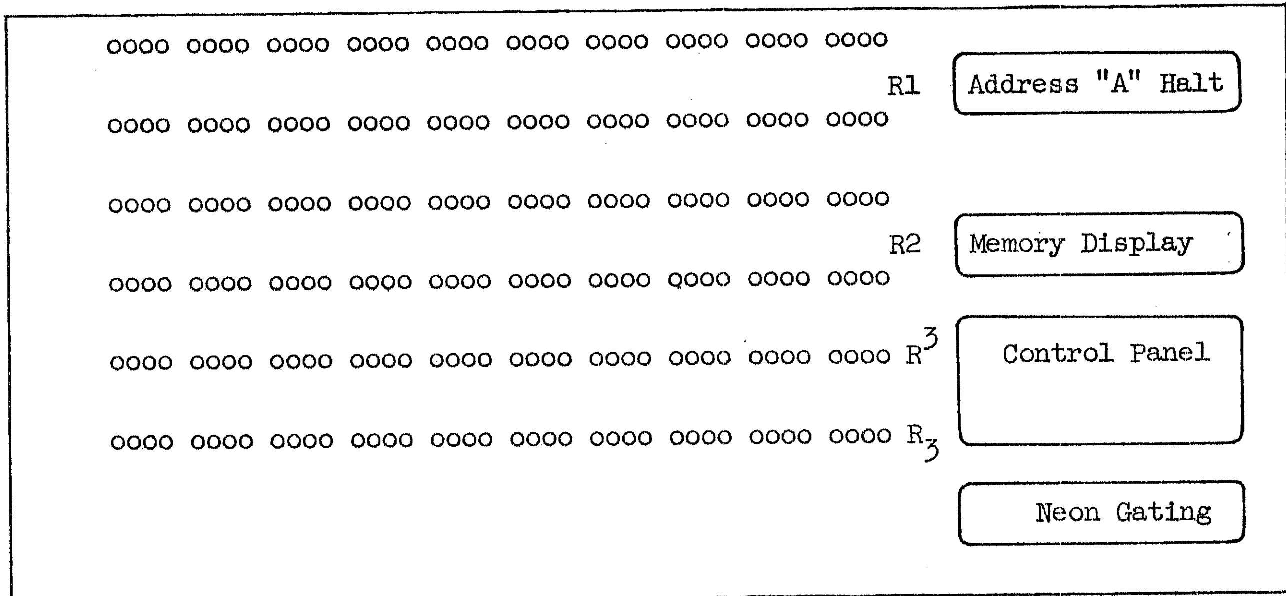


Figure 1

CONTROL PANEL

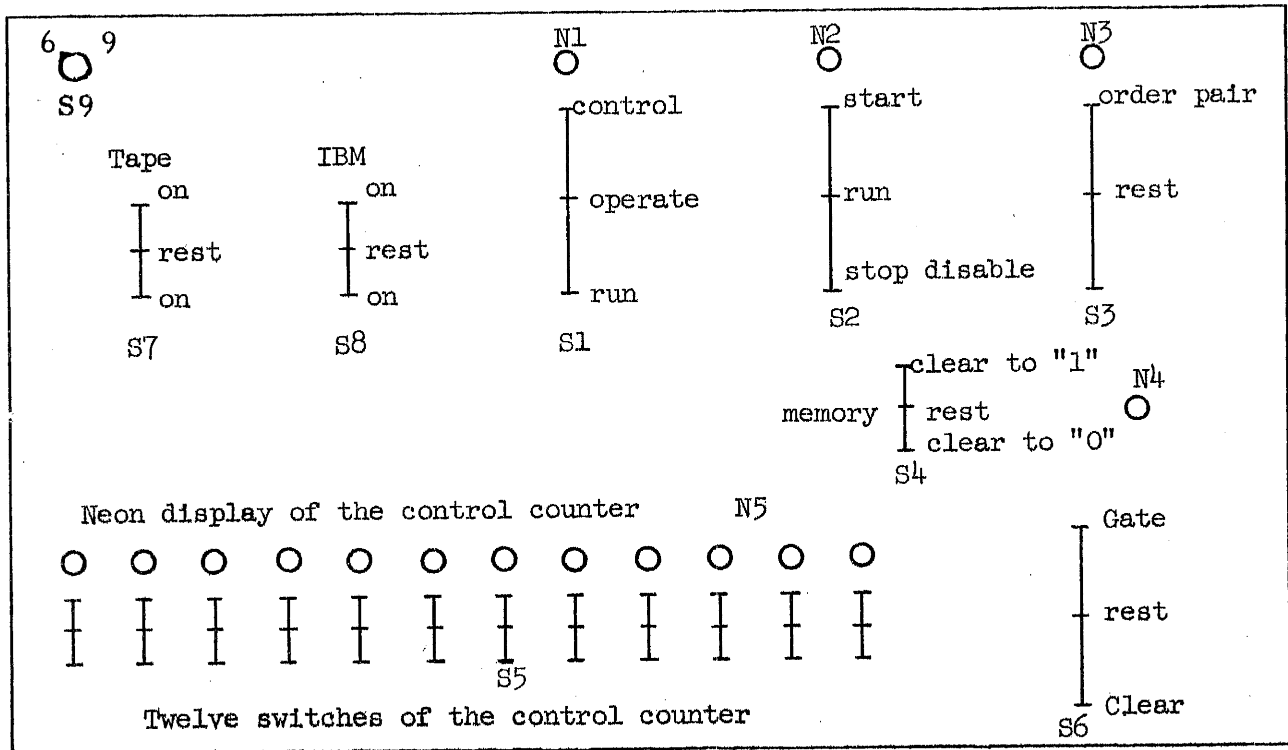


Figure 2

simultaneously by moving the switch S6 down. Moving the switch S6 up to the "gate" position transfers (gates) the setting of the switches S5 and neons N5 to the control counter at the back of the machine.

The order register R_3. The order register R_3 is actually inside the machine but we shall deal only with the forty neon display of R_3 on the front panel (see Figure 1). In referring to the register R_3 we shall mean the forty neon display. The forty neon display at any instant, records automatically the instruction word (an order pair) which the machine is to perform. It must be clearly understood that while the control counter records the address of the instruction word, the register R_3 records the contents of this address. For example, the combination of lights of the R_3 neon display;

●O●O O●OO OO●O O●●O OOOO ●●OO O●OO OO●O O●●O OOO●

represents the machine number

1010 0100 0010 0110 0000 1100 0100 0010 0110 0001 whose sexadecimal equivalent is K4260 N4261. If the control counter display at the same time shows the binary number 0010 1000 0000 whose sexadecimal equivalent is 280, it means that the memory position, addressed 280, contains the order pair K4260 N4261. The neon N1 on the control panel (see Figure 2) indicates which order, left or right, is to be performed. The "neon off" means the left-order, the "neon on" means the right-order. When the machine is not running any desired order pair can be set manually in R3 by the switches on the "Neon Gating" panel (see Figure 1).

The control panel. In the diagram of the control panel (Figure 2) the switches and neons are labelled for the purpose of reference. They are not labelled on the actual control panel. A neon associated with a given switch is labelled in this diagram by the same numeral; for example, the neon N4 is associated with the switch S4. The switches with the associated neons perform the following functions:

Switch S1 and neon N1. Switch S1 in the position "run" starts computing operations making the machine execute sequences of orders, one after another. If the programmer wants the machine to stop he sets the switch to the position "operate". When the machine stops and the programmer wants it to execute the next order and stop, he switches S1 to the "control" position and then back to the "operate" position. The neon N1 on (lit) indicates that the right-order is to be executed, the neon off indicates that the left-order is to be executed.

Switch S2 and neon N2. The switch S2 is a "conditional stop" switch. In the position "run" it will stop the machine when the control encounters a conditional stop order (see Appendix). Moving the switch to the "start" position makes the machine run again and stop at the next conditional stop order. In the position "stop disable" conditional stop orders in the computed routine will be disregarded and the machine will not stop. The neon N2 lights when the machine has stopped after encountering a conditional stop order.

Switch S3 and neon N3. The switch S3 in the position "rest" makes the machine execute orders in a sequence, one after another. The switch in the position "order pair" makes the machine execute only the order pair which is at the time in the order register R_3. The neon N3 on indicates that the switch is in the position "order pair".

Switch S4 and neon N4. The switch S4 is the "memory clear" switch. The switch in the middle position "rest" disconnects the memory from the control panel. The "rest" position is the normal position and the switch automatically returns to it from other positions. The "up" position clears the memory to "1"s (that is stores in every memory position a word consisting of forty "1"s), the "down" position clears the memory to "0"s. While the memory is being cleared the neon N4 is lit. When the process of clearing is finished (approximately 1.5 seconds) the neon is off.

Switches S5 and neons N5. The switches S5 and the neons N5 belong to the control counter and were described before.

Switch S6. Switch S6 also belongs to the control counter.

Switch S7. The switch S7, whether in the "up" or"down" position, inserts the "tape read order" (See Appendix) in order register R_3, in the case when the input is on tape instead of on IBM cards.

Switch S8. The switch S8, whether in the "up" or "down" position, inserts the "IBM" card read order" (see Appendix) in the order register R_3.

Switch S9. For 6-bit instruction code this should always point to "6".

Beside the control panel there are three more auxiliary panels which are shown in Figure 1. These panels will be shortly described.

Address "A" Halt panel. The Address "A" Halt panel has twelve switches in a row associated with a twelve neon display. The switches set a memory address and the neons display it in the same way as in the control counter. The "halt-ignore" switch in the position "halt" will stop the machine before the execution of the pair of orders which are stored at the address set by the twelve switches; in the position "ignore" the machine will not stop.

Memory Display panel. This panel has a forty neon display in two rows, twenty neons in each row, capable of displaying one forty bit word, and has also twelve address switches with associated twelve neons, which set one memory address. The forty neons display the contents of the memory address which is set by the address switches.

Neon Gating panel. This panel has forty switches with forty associated neons. The switches set a forty bit word which is displayed by the neons. A word set by the forty switches can be gated (inserted) to the register R1 or R_3 by two switches, the "R1-R_3" selecting switch and the gating switch. This gating can only be done when the machine is not running. Indeed, this fact is so critical that the operation engineer keeps emphasizing to the novice who is using the machine, the following warning:
DO NOT TOUCH THE FORTY SWITCHES WHEN THE GATING SWITCH IN ON "GATE"

Neon displays of the accumulation register R1, of the arithmetic register R2 and of the registers R^3 and R_3. The two topmost rows of neons, forty neons in each row on the front panel (see Figure 1) are the register R1 display neons. The important row is the lower row, which shows the contents of R1. The upper row displays the results of certain minor operations and will not be described.

The two middle rows of neons are the register R2 display neons, the lower row shows the contents of R2. The display of the upper row will not be described.

The upper of the bottom two rows of neon displays the contents of the R^3 (R upper three) register which plays a minor role only in coding and will not be described. The lower row displays the contents of the order register R_3, which was described before.

Reading-in a routine. Operating Instructions. In the following operating instructions the letter-numeral labels will refer to the switches on the control panel shown in Figure 2. We shall assume that the power is on and the machine is warmed and ready for operations. The instructions for reading-in a routine consist of the following:

1. Move S1 up to the position "control" extinguishing the neon N1.
2. Move S3 to the position "rest", which is the normal position.
3. Move S2 to a) the position "run" when the routine has conditional stop orders to be obeyed by the machine, or
   b) the position "stop disable" when the routine has no conditional stop orders, or when the conditional stop orders are to be ignored.
4. Move S4 down to the "clear memory to zero" position, which would clear the memory to "0"s and insert the IBM card read order in R_3 as well.
5. When using the tape input, move S7 to the position "on" which inserts the tape read order in R_3. For IBM cards input this step must be ignored.
   If neon N1 is lit, repeat steps 1 to 5.
6. Stack the deck in the reader.
7. Press the second button from the left on the reader, causing this reader to "read down" the first card.
8. Move S1 down to the "run" position, which makes the machine start the computations.

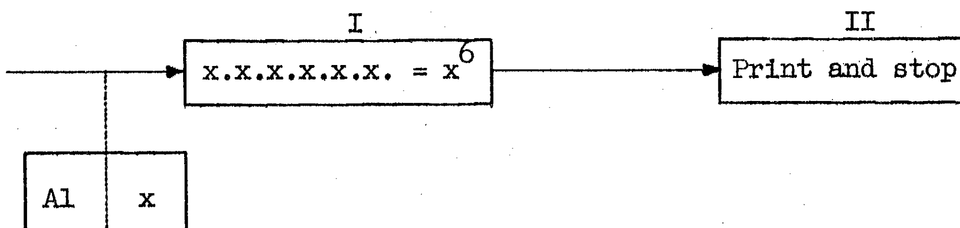
CHAPTER VI

REPETITIVE SEQUENCES. TRANSFER ORDERS. DECISION BOX. COUNTERS FORMATION FORMULA. ADDRESS MODIFICATION. EXTRACT ORDERS. PROGRAMMING AND CODING LOOPS OF REPETITIVE OPERATIONS.

Repetitive Sequences. Let us consider the following problem: To program and code $f(x) = x^6$, where $|x| < 1$.

1. Programming.

Flow Chart



2. Preliminary Coding

Seq.		Order	R1	R2	Memory	Description
I,1		R A1		x		(A1) ⟶ R2
2	1st Group	XuA1	$x.x=x^2$			(R2)(A1) ⟶ R1
3		M A2			x^2	(R1) ⟶ A2
4		R A2		x^2		(A2) ⟶ R2
5	2nd Group	XuA1	$x^2.x=x^3$			(R2)(A1) ⟶ R1
6		M A2			x^3	(R1) ⟶ A2
7		R A2		x^3		(A2) ⟶ R2
8	3rd Group	XuA1	$x^3.x=x^4$			(R1)(A1) ⟶ R1
9		M A2			x^4	(R1) ⟶ A2
10		R A2		x^4		(A2) ⟶ R2
11	4th Group	XuA1	$x^4.x=x^5$			(R2)(A1) ⟶ R1
12		M A2			x^5	(R1) ⟶ A2
13		R A2		x^5		(A2) ⟶ R2
14	5th Group	XuA1	$x^5.x=x^6$			(R2)(A1) ⟶ R1
15		M A2			x^6	(R1) ⟶ A2
II,1		R A2		x^6		(A2) ⟶ R2
2		P				Print (R2)
3		Zu				Stop

3. 4. Assignment of Addresses and final coding. This is left to the student as an exercise.

The sequence (Box) I of this routine has fifteen orders. These fifteen orders are in five groups , of which four are identical, the first group differs only in the address part in the first order. In general, a routine for computing x^n ($n = 1,2,\ldots.$) would have in the operation Box I (sequence I), n - 1 similar groups of three orders. Routines of many problems with repeating operations have very often "repetitive sequences". Approximation methods which employ iterative formulas, also lead to repetitive sequences. The routine for x^6 was coded in a "straight sequence", that is, every repetitive group of orders was written down one after another. Instead of coding repeating operations in a straight sequence, we can program and code such operations using "loops", that is, instructing the machine to repeat a certain cycle of operations automatically. The coding (explained presently) of a repetitive operation in the form of a loop has many advantages. The routine is much shorter, thus saving memory positions (as we already know, every order-pair makes an instruction word which is stored in one memory position); the machine can be instructed to repeat a cycle of operations either a specified number of times or as many times as is necessary to give the result which will satisfy a desired condition.

The programming and coding of loops require techniques which we will now explain. These are: using transfer orders, setting "counters", "modifying the addresses", and setting "formation formulas".

Transfer orders. The transfer order "U A1" was introduced in Chapter II. We shall repeat that a transfer order is an order which instructs the machine to break the initial sequence of orders and to start a new sequence which begins from the order specified in the address-part, say A1, of the transfer order. We say that "the control was transferred to A1", and "transfer of control" is the technical name for breaking the initial sequence of orders and starting a new one which begins from a specified order. Other names for transferring of control are: "directing the control", and "jumping". A transfer order specifies the address, A1, (preliminary symbols do not specify addresses , see Notes.), where an instruction word containing two orders, the left and the right, is stored. The new sequence begins from one of those two orders and a transfer order also specifies which one.

Most Important Transfer Orders

No	Preliminary Symbol	Sexadecimal Symbol	Description
1	U x	NO.$\overset{x}{..}$.	Transfer control to x. Start from the left order.
2	U' x	14.$\overset{x}{..}$.	Transfer control to x. Start from the right order.
3	oU x	KO.$\overset{x}{..}$.	Clear R1 to zeros. Transfer control to x. Start from the left order
4	oU' x	34.$\overset{x}{..}$.	Clear R1 to zeros. Transfer control to x. Start from the right order
5	C x	20.$\overset{x}{..}$.	Transfer control to x (left order) if (R1) $\geqq$ 0, do not transfer (continue in sequence) if (R1) $<$ 0.
6	C' x	40.$\overset{x}{..}$.	Transfer control to x (right order) if (R1) $\geqq$ 0, do not transfer (continue in sequence) if (R1) $<$ 0.
7	A + x	KN.$\overset{x}{..}$.	(R2) $\longrightarrow$ R1. Transfer control to x. Start from the left order. (See notes).
8	A(+) x	NN.$\overset{x}{..}$.	(R1)+(R2) $\longrightarrow$ R1. Transfer control to x. Start from the left order. (See notes)

Notes:

a. Transfer orders have the following peculiarity:

In the address-part of the preliminary representation we write the symbol of the order itself (its sequence number) instead of writing its address. For example, if we want to transfer control to the order II,2, stored at some address, say A1, we write U II,2, instead of writing U A1. The rule that an order refers to an address and not to the contents must be broken in this case, because in preliminary coding we do not know the addresses of the instruction words. In the final coding, the sexadecimal symbols must have, of course, the proper address-part. For example, if a left order II,2 is stored at the address 284(16) the preliminary symbol of an order directing control to II,2 is U II,2 and the sexadecimal symbol is N0284. In the above list a two digit sexadecimal instruction (third column) is followed by "x" which represents a three digit sexadecimal address.

b. The orders No. 1 through No. 4 belong to the category of "unconditional transfer orders" and are fully explained in the list.

c. The orders No. 5 and No. 6 are "conditional transfer orders", called "compare orders" or "branch orders".

d. The orders No. 7 and No. 8 are called "secondary transfer orders". Their primary purpose is to move the contents of R2 to R1. They transfer control under the following conditions: a) always, when A+ x or A(+) x is the right-order, b) when A+ x or A(+) x is the left-order and the following right-order is not another transfer order. In these cases control is transferred after the following right-order is completed. When A+ x or A(+) x is the left-order and the following right-order is another transfer order, then A+ x or A(+) x does not transfer control. [(R2) ⟶ R1 or (R2) + (R1) ⟶ R1 whether control is transferred or not].

Decision Box. In the example $f(x) = x^6$, the operations represented by a three order group are performed five times. If we want to code a loop for such a computation we have to devise an arrangement which allows for exactly five repetitions. This arrangement consists of a "decision box" and a "counter". A decision box (called also comparison, alternative or discrimination box) is a sequence of a few orders shown on a flow chart in a separate box. Both the box on a flow chart and the sequence of orders which it represents are referred to as "the decision box". The decision box will be explained in the following example: A group of operations represented by Box II have to be performed five times and after the fifth performance the result should be printed in Box IV. Between the Boxes II and IV is inserted the Decision Box III, which together with the counter will direct such a course. We shall not discuss the counter now; we shall assume that it is there and performs the assigned function. The diagram is as follows:

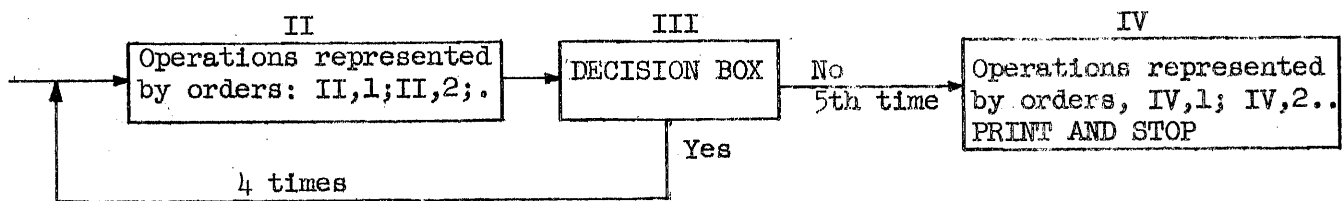


Figure 1.

When the cycle of operations in Box II is performed for the first time the Decision Box III does not direct control through the "No" branch to Box IV, but directs control to Box II, to perform the cycle of operations in Box II for the second time. This is indicated by "the loop", that is, by the branch "Yes". After the cycle of operations in Box II is performed for the second time, the Decision Box III does the same as before, and likewise after the third and the fourth time. After the fifth time, the Decision Box III directs control through the "No" branch to Box IV. A decision box on a flow chart can be recognized by the three branches, the incoming branch and the two exit branches labeled "Yes", and "No". We shall describe now the operations represented by the sequence of orders in the decision Box III. Let $i = 1, 2, 3, 4, 5$, represent the number of times that the operations in Box II were performed. When $4-i$ equals 3 it means that the cycle of operations in Box II was performed once ($i = 1$), when $4-i = 2$, the cycle was performed twice, $4-i = 1$, the cycle was performed three times, $4-i = 0$, the cycle was performed four times, $4-i = -1$, the cycle was performed five times. This can be summarized as follows: when $4-i \geq 0$, the cycle has been performed four times or less, when $4-i < 0$, (namely, $4-i = -1$) the cycle has been performed five times. Let the number 4 (scaled by 2^{-3}) be stored at an address D1, and the number i (scaled also by 2^{-3}) be stored at an address D2. Before the first run of the cycle of operations in Box II, i would equal 1; before the second run the counter would substitute 2 for i; before the third run, 3 for i; before the fourth run 4 for i; before the fifth run, 5 for i. The orders grouped in the Decision Box III would be as follows:

Sequence	Order	R1	Description
III,1	+ D1	$4x2^{-3}$	(D1) $\longrightarrow$ R1
2	(-) D2	$2^{-3}(4-i)$	(R1)-(D2) $\longrightarrow$ R1
3	C V,1		Transfer control to V,1 when (R1) = $2^{-3}(4-i) \geq 0$. Do not transfer (continue in sequence) which means that the following order would be IV,1) when (R1) = $2^{-3}(4-i) < 0$.

The Decision Box III on the flow chart would symbolically represent this sequence of orders as follows:

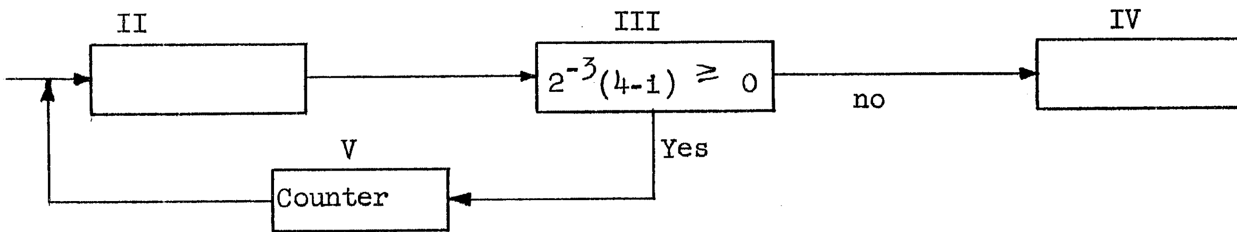


Figure 2

<u>Counter</u>. The sequence of orders which instructs the machine to increase i by 1 every time after the completion of the cycle of operations in Box II is called a "counter". On the flow chart the box which groups the operations of the counter is called a "substitution box". The substitution box is inserted in the "Yes" branch between the Operation Box II and the Decision Box III, shown in Figure 3. Inside the substitution box we write the symbol $i+1 \rightarrow i \rightarrow D2$ which reads: i+1 replaces i at the address D2.

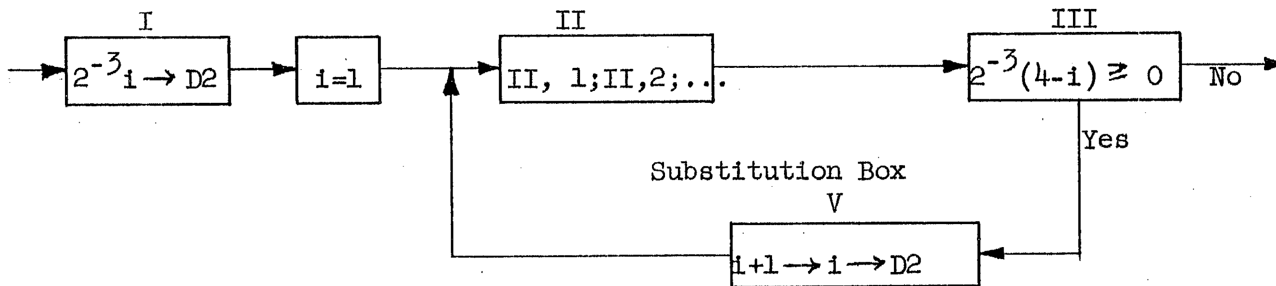


Figure 3

The storage of the initial value of i (i=1) at D2 is an operation which precedes the operations in Box II and is shown in Box I (Figure 3). The little box (not labeled) immediately following Box I is called an "assertion box". This assetion box states the initial value of i, which in our case is 1. The orders in the substitution Box V instruct the machine to add 1 to i and to store i+1 at the same address where i was stored previously,

that is at D2, and then transfer control to the order II, 1. The totality of operations in Boxes II, III, and V form a "loop". The orders in the substitution Box V would be as follows:

Sequence	Order	R1	Memory	Description
V, 1	~~1~~→ 2	$1x2^{-3}$		$2^{-3} \longrightarrow$ R1
2	(+) D2	$2^{-3}(i+1)$		(R1)+(D2) $\longrightarrow$ R1
3	M D2		$2^{-3}(i+1)$	(R1) $\longrightarrow$ D2
4	U II,1			Transfer control to the order II,1.

Exercise: Sketch and code the decision box and the substitution box for a loop to perform a cycle fifteen times (for a fifteen-cycle loop).

Formation Formula. When programming a loop we must set a "formation formula" which would allow identical operations in each cycle. A formation formula is of the form $x_{i+1} = F(x_i)$, where $i = 1,2,3,\ldots,n$, and x_1 is an initially given value. F(x) is a function which has the following properties: a sequence $x_2, x_3, \ldots, x_{n+1}$ can be formed (term by term) by repeated application [n times] of the function F, and the last value, x_{n+1}, is the final result. The values $x_2, x_3, \ldots, x_n$ are called the "partial results" or the "intermediate results". The indices "i" are not necessarily subscripts. They may be exponents, factors, or terms. Operations represented by $x_{i+1} = F(x_i)$ are identical in each cycle, but the indices are different in different cycles. In the first cycle $i = 1$, and the operations give the first partial result, $x_2 = F(x_1)$; in the second cycle $i = 2$, the operations giving the second partial result, $x_3 = F(x_2)$, and so on; in the last, the nth cycle, $i = n$, and the operations give the final, the n-th result, $x_{n+1} = F(x_n)$.

Programming and Coding the Loop for the Problem $f(x) = x^6$.

Programming. We shall set a formation formula whose final result is x^6. Observing the straight sequence routine of our problem (at the beginning of this chapter) we notice that:

in the first group of orders, I,1; I,2; I,3; the operations lead to $x^1.x=x^2$

in the second group of orders, I,4; I,5; I,6; " " " " $x^2.x=x^3$

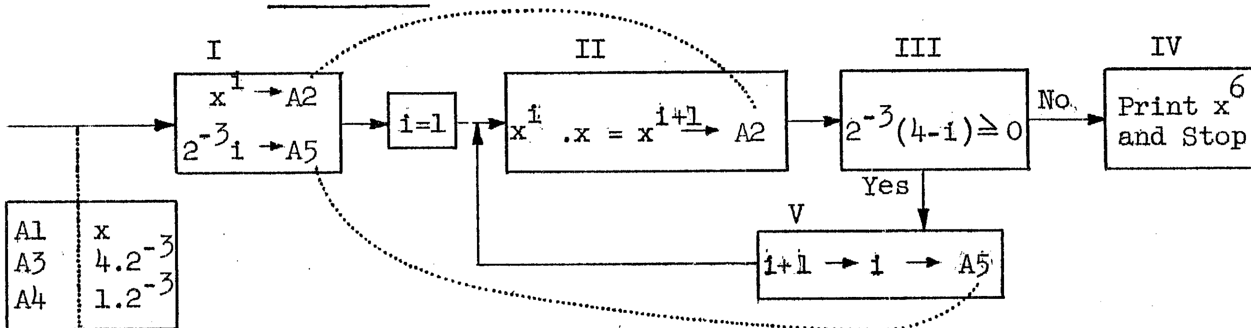
in the third group of orders, I,7; I,8; I,9; " " " " $x^3.x=x^4$

in the fourth group of orders, I,10; I,11; I,12; " " " $x^4.x=x^5$

in the fifth group of orders, I, 13; I,14; I,15; " " " $x^5.x=x^6$.

This suggests a formation formula of the form, $x^{i+1} = x^i.x$, where $i = 1,2,3,4,5$. The multiplication of x^i by x is performed five times; after each multiplication the exponent i is increased by 1. The partial result (product) becomes the final result (product) when $i = 5$. Other features of programming, like the counter, storage, etc., are explained on the flow chart.

Flow Chart



Notes:

a. We remembered to store in A3 the number $4x2^{-3}$, which we need in the Decision Box III, as is shown in the Storage Box.

b. Box I is immediately followed by the non-labeled Assertion Box. Inside the Assertion Box is written the initial value of the index (exponent) i, which in our case is 1.

c. Box I is an operation box where we store the initial value $2^{-3}\ i = 1x2^{-3}$, which we need in the Decision Box III and in the Substitution Box V. In Box I we also store the initial value of $x^i = x^1$. (The storage of these two numbers is an essential part of the computation routine and cannot be performed advantageously with the input routine since this latter is available only once.) In Box II, when the partial product x^{i+1} is formed, it is stored back at the address A2, where the previous x^i was stored. In Box V, i+1 is stored back at the address A5 where i was

previously stored. Replacing at the same address an old value by a new value with an index increased by 1, is an important feature and is indicated on the flow chart by dotted lines.

2. Preliminary and Final Coding.

Seq.	Code	Word	Order	R1	R2	Memory	Description
I,1	28002	140	—1→ 2	2^{-3}x1			2^{-3}x1 ⟶ R1
2	10124		M A5			$ix2^{-3}=1x2^{-3}$	(R1) ⟶ A5
3	K4120	141	+ A1	x			(A1) ⟶ R1
4	10121		M A2			$x^i = x^1$	(R1) ⟶ A2
II,1	S4121	(142)	R A2		x^i		(A2) ⟶ R2
2	K8120		Xu A1	x^{i+1}			(R2)(A1) ⟶ R1
3	10121	143	M A2			x^{i+1}	(R1) ⟶ A2
III,1	K4122		+ A3	$4x2^{-3}$			(A3) ⟶ R1
2	04124	144	(-)A5	$2^{-3}(4-i)$			(R1)-(A5) ⟶ R1
3	40146		C'V,1				Transfer control to V,1 right order, when $(4-i) \geq 0$
IV,1	S4121	145	R A2		x^6		(A2) ⟶ R2
2	L4028		P				Print (R2)
3	00000	(146)	Zu				Stop
V,1	K4124		+ A5	$ix2^{-3}$			(A5) ⟶ R1
2	N4123	147	(+) A4	$2^{-3}(i+1)$			(R1)+(A4) ⟶ R1
3	10124		M A5			$2^{-3}(i+1)$	(R1) ⟶ A5
4	N0142	148	U II,1				Transfer control to II,1; left order.
	00000						

Notes:

a) Instruction words in the sequences I, II, III, and IV must be stored at consecutive addresses in the above order; the two instruction words in the counter sequence, V, can be stored in a different part of the memory at two consecutive addresses. To avoid errors, it is advisable to label boxes and write instructions in their proper order so that they are

assigned to consecutive addresses. In this connection every box with three branches must be carefully examined and the rule should be followed that boxes connected by a "no" branch must be labeled by consecutive numerals.

b) When we explained and showed how to set the transfer orders of the decision and substitution boxes, treating them apart, we never specified left- or right-order, because we were unable to tell. This we learn only from the context of the computation routine as a whole, when the order pairs are grouped together. The importance of carefully writing order pairs must be stressed. An error in grouping order pairs may result in directing control to the right-order instead of to the left-order, or vice-versa.

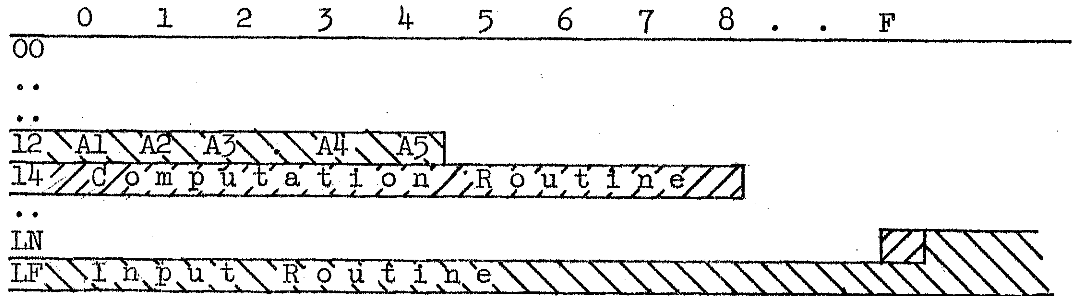
c) The address part in the sexadecimal representation of the transfer orders, II, 3 and V, 4, was put in the final coding after the assigned addresses of instruction words were written. The "long arrows" indicate where the addresses in the transfer orders come from. The address part in the preliminary representations is filled by the sequence number which is a symbol of the order itself instead of its address, which we explained before.

d) The counter in Box V has the following alternative form:

Sequence	Order		R1
V,1	$\xrightarrow{1}$	2	2^{-3}
2	(+)	A5	$2^{-3}(i+1)$
3	the same as before		
4	the same as before		

The alternative has a small advantage over the previous arrangement. The storage of $1x2^{-3}$ at the address A4 (see storage box) is not necessary and one memory position is saved.

3. Assignment of addresses.



4. Card Punching and Transcribing. This is left to the student as an exercise.

Programming with a Decision Box for Desired Accuracy.

Successive approximation formulas (iteration formulas) are of the form $x_{i+1} = F(x_i)$, where $i = 1, 2, 3, 4, \ldots$; x_1 is an initial estimate of the true value, x, and F is called the "improvement function", or the "improvement operator". A sequence, $x_2, \ldots, x_n$, of improved estimates (second, third, ..., nth approximations) can be formed by repeated application of the improvement function F. The magnitude of the error of an nth approximation, x_n, is the absolute value of the difference between the true value, x, and x_n, that is, the error $e_n = |x-x_n|$. The smaller the error the greater the accuracy. Unfortunately, we do not know the true value (otherwise we would not use the iteration formula). However under some conditions, as a satisfactory measure of accuracy one may use the absolute value of the difference between two successive approximations, that is, $|x_{n+1}-x_n|$. For example, Newton's iteration formula for $x = \sqrt{N}$, $(N > 0)$, is $x_{i+1} = \frac{1}{2}(x_i + N/x_i)$. When $x = \sqrt{2}$, and the initial estimate of x is $x_1 = 1.5$ $(1 < \sqrt{2} < 2)$, then

$x_2 = \frac{1}{2}(1.5 + 2/1.5) = 0.75 + 0.67 = 1.42;$ $\qquad |x_2 - x_1| = 0.08$

$x_3 = \frac{1}{2}(1.42 + 2/1.42) = 0.71 + 0.704 = 1.414$ $\qquad |x_3 - x_2| = 0.006$

If we assume that $\sqrt{2} = 1.41421$ (accurate to the fifth decimal place) is, x, the true value of $\sqrt{2}$, then the errors of each successive approximation

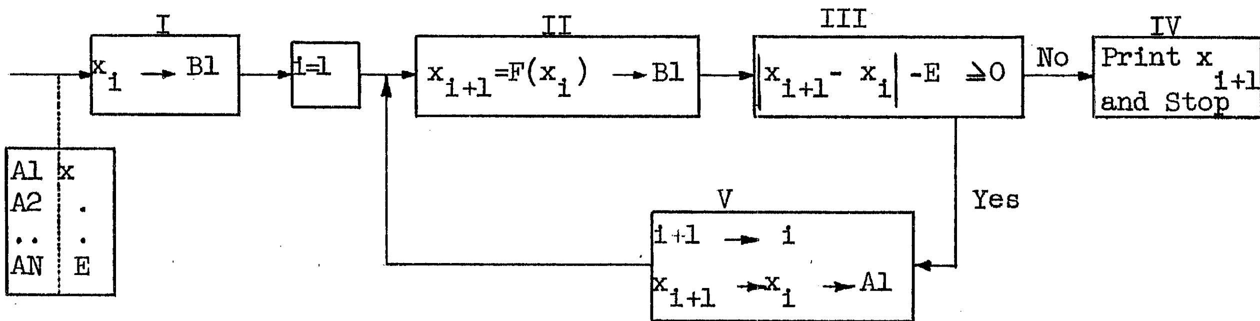
compared with the differences between two successive approximations are as follows:

$$e_1 = |1.41421-1.5| = 0.08579$$

$$e_2 = |1.41421-1.42| = 0.00579 \qquad |x_2 - x_1| = 0.08$$

$$e_3 = |1.41421-1.414| = 0.00021 \qquad |x_3 - x_2| = 0.006.$$

When the error, e_n, decreases, so does $|x_{n+1} - x_n|$. When we say that we want an accurate result we mean at least that we require that $|x_{n+1} - x_n|$ be small. An iteration formula is very convenient for loop programming and coding because it has the form of a formation formula. Let us program a loop for $x_{i+1} = F(x_i)$, with an accuracy such that $|x_{i+1} - x_i| < E$, E being an assigned, usually small , number. Assume that scaling is already done. The flow chart would be as follows:



The first run of operations in Box II give and store x_2 at the address B1. The difference, $(|x_2 - x_1| - E)$,is tested in the Decision Box III. If $(|x_2 - x_1| - E) \geqslant 0$, that is,if $|x_2 - x_1| \geqslant E$, then the control is directed to the Box V, which is not a counter. The indices i in the formation formula are replaced automatically by i+1 without any special arrangement. We write inside Box V, $i+1 \longrightarrow i$, but the absence of storage indicates that it is not coded; it is only a reminder that after each run of operations in Box II the indices were replaced. The operations in Box V store x_2 at the address A1. This would erase x_1, which was previously stored at A1 (see storage box), but we do not need x_1 after the first run of operations in Box II. Thus, x_2 is stored at two addresses, B1 and A1; the second storage is necessary because we shall need x_2 later on for the decision Box III and the x_2 at B1 will be erased by the storage

of x_3 (after the second run in Box II). The last operation in Box V directs control to Box II, where the next approximation, x_3, is obtained and stored at B1. The difference $(|x_3 - x_2| - E)$ is tested in the decision Box III, and the operations will cycle in Boxes II, III, and V, until, let us say, after the nth cycle the difference $(|x_{n+1} - x_n| - E) < 0$ for the first time. The decision Box III, then, will not transfer control to Box V, but to Box IV, where the desired approximation, x_{n+1} is printed. The machine performs as many cycles as are necessary to obtain the desired accuracy.

Exercises.

1) Program and code the loop for computing $x = \sqrt{3}$, with the accuracy measured by the condition that $|x_{i+1} - x_i| < 2^{-30}$.

2) Newton's iteration formula for a root of an equation, $f(x) = 0$, is:

$$x_{i+1} = x_i - f(x_i)/f'(x_i).$$

Program and code the loop for computing both roots of the equation:

$$f(x) = x^2 - 2x - 2 = 0$$

Orders instructing operations on absolute values are in the list in the Appendix.

Programming and Coding a Loop for the Sum of Hundred Terms.

Let us program and code a loop for computing $S_{100} = a_1 + a_2 + \cdots + a_{100} = \sum_{k=1}^{100} a_k$. To make it simpler we shall assume that scaling is not necessary.

Programming. We shall begin from the formation formula. If the sum of the first i terms is $S_i = \sum_{k=1}^{i} a_k$, then the sum of the first i+1 terms is $S_{i+1} = S_i + a_{i+1}$ and this is the formation formula. The successive partial sums, $S_{i+1} = S_i + a_{i+1}$, are as follows:

$$S_1 = a_1$$

$$S_2 = S_1 + a_2$$

$$S_3 = S_2 + a_3$$

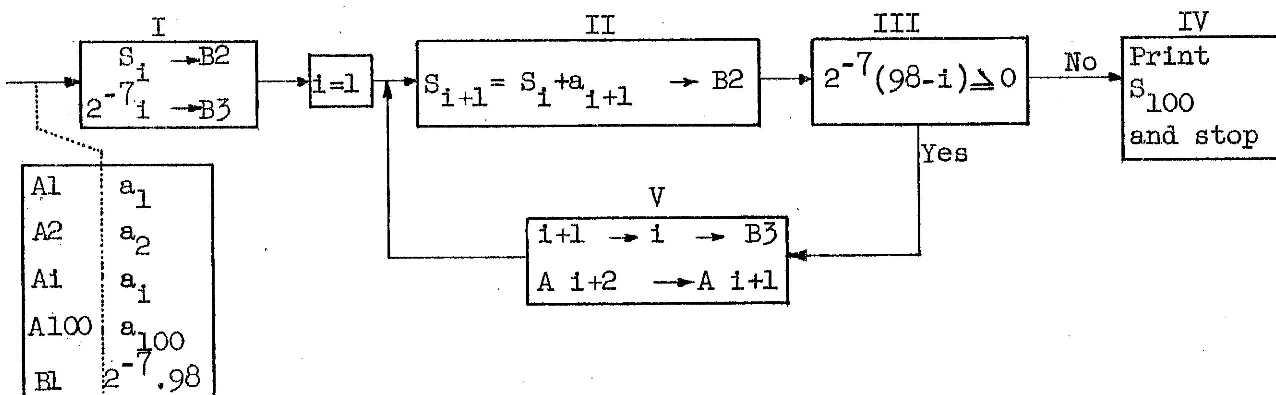
. . . .

. . . .

$$S_{100} = S_{99} + a_{100}.$$

The first partial sum $S_1 = a_1$ is given, hence the machine must perform 99 cycles computing successively $S_2, S_3, \ldots, S_{100}$. After the 99th cycle the final result, S_{100}, is obtained. The Decision Box will test the difference $2^{-7}(98-i)$; if $2^{-7}(98-i) \geqq 0$, then the control will be directed to the counter, and from there to the operation box, where $S_{i+1} = S_i + a_{i+1}$ is computed. When $2^{-7}(98-i) < 0$ (namely, when $98-i = -1$) then the control will go to "print and stop" Box, where S_{100} will be printed. The flow chart of the loop for this computation is very similar to the previous flow charts.

Flow Chart



Notes.

a. In previous examples the storage of datum numbers at consecutive addresses was not necessary. Here it is essential. Storage Box shows that a_1 is stored at A1, a_2 at A2, etc. The reasons will be obvious presently.

b. After each run of operations in Box II the indices i have to be increased by 1. We shall examine carefully where the change of indices must take place in the following table:

i	Box II a_{i+1}	Box II $S_{i+1} = S_i + a_{i+1}$	Box III (98-i)
1	a_2	$S_2 = S_1 + a_2$	98-1
2	a_3	$S_3 = S_2 + a_3$	98-2
.	.	.	.
.	.	.	.
98	a_{99}	$S_{99} = S_{98} + a_{99}$	98-98
99	a_{100}	$S_{100} = S_{99} + a_{100}$	98-99

The table shows that the indices, i, have to be increased by 1 in three places:

1. In Box II ; after completing the operations in Box II, the partial sum, S_i, must be replaced by S_{i+1}. This does not require special arrangements, because the operation $S_i + a_{i+1}$ gives automatically S_{i+1}.

2. In Box III, after the control is directed to Box V, i has to be increased by 1 in the difference 98-i. This is done by the counter in Box V and we already know how to arrange for that.

3. In Box II, after completing the operations in Box II, the number a_{i+1} has to be replaced by the number a_{i+2}, so that the order, which initially instructed the machine to add a_{i+1} to S_i would instruct the machine next time to add a_{i+2} to S_{i+1}. How to arrange for that is explained in the following article.

Modification of Addresses. We remind the reader that a machine order does not refer to a number itself but to the address where the number is stored. The number a_{i+1} is stored at A(i+1), and the number a_{i+2} is stored at A(i+2). The order (preliminary symbol) to add a_{i+1} is (+) A(i+1), and the order to add a_{i+2} is (+) A(i+2). The "add order" is in Box II and we shall call it by its sequence number, that is II,2 (see flow chart). To instruct the machine to add a_{i+2} instead of a_{i+1} we must change the address-part of the "add order", II, 2, in such a way that instead of (+) A(i+1) we shall have (+) A(i+2). Changing address parts in machine orders is called "modification of addresses" or "address modifications". The operations

which perform address modifications are grouped in the substitution Box V (we remember that the operations replacing i by i+1 are also in Box V). For the purpose of an illustration let us assume that

the address A(i+1) is 281(16) = 0010 1000 0001(2)

and the address A(i+2) is 282(16) = 0010 1000 0010(2).

Now we want to change the order II,1, initially (+) A(i+1) into an order, (+) A(i+2), which in sexadecimal representation means to change K4281 into K4282, and in binary representation means to change 1010 0100 0010 1000 0001 into 1010 0100 0010 1000 0010, or in general to change an address into an address which is next in the sequence of addresses (change address n into an address n+1). We shall speak now of the binary representation because this will render more simple the explanation. Observe the binary representations of the orders (+) A(i+1) and (+) A(i+2) and notice that to change an address-part n into an address part n+1 we must add a binary 1 to the last, the twentieth,bit of the order. Thus,

The last four bits of (+) Ai+1 are 0001
+1

The last four bits of (+) Ai+2 are 0010.

The "add order" II,2 may be the left- or the right-order of instruction word stored at some address, say B3. We shall discuss each case separately.

Case 1. The order II,2 is the left-order and its companion, the right-order, is II,3. At the moment we are not interested in II,3 whose digits (or bits) we shall indicate by x's. The whole instruction word consisting of the order-pair II,2 and II,3 and stored at B3, before the modification, in sexadecimal representation is K4281 xxxxx, and in binary representation is 1010 0100 0010 1000 0001 xxxx xxxx xxxx xxxx xxxx

if we add to it 2^{-19} = 0000 0000 0000 0000 0001 0000 0000 0000 0000 0000

we obtain 1010 0100 0010 1000 0010 xxxx xxxx xxxx xxxx xxxx

which is the modified instruction word with the address of II,2 modified to an address which is next in the sequence (of addresses). The above example illustrates the following rule: to modify the address-part of a left-order to an address-part which is next in the sequence (of addresses), add 2^{-19} to the instruction word.

Case 2. The order II,2 is the right-order and its companion, the left-order is II,1. The instruction word consisting of the order-pair II,1 and II,2, before the modification, in sexadecimal representation is, xxxxx K4281 and in binary representation is

	xxxx xxxx xxxx xxxx xxxx 1010 0100 0010 1000 0001;
if we add to it 2^{-39} =	0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
we obtain	xxxx xxxx xxxx xxxx xxxx 1010 0100 0010 1000 0010,

which is the modified instruction word with the address-part of II,2 modified to an address-part which is next in the sequence (of addresses). The above example illustrates the following rule: to modify an address-part of a right-order into an address-part which is next in the sequence of addresses, add 2^{-39} to the instruction word.

Coding the Operations Which Modify Addresses. We said that the order II,2 is stored with its right- or left-companion at some address B3. In preliminary coding we do not know yet the addresses of the instruction words, hence we do not know what B3 is. The situation is similar to the one which we encountered in transfer orders. We cannot refer to the address, of II,2 because we do not know it, therefore we put II,2 in the address-part of the preliminary symbol. In the final coding the sexadecimal symbols must have, of course, their proper address-part. The modification of II,2 is coded as follows:

Seq.	Order	R1	Memory	Description
1st case; II,2 is the left order				
V, 1	$\xrightarrow{1}$ 18	2^{-19}		
2	(+) II,2	(II,2)+2^{-19}		2^{-19} is added to the instruction word containing the order-pair II,2 and II,3
3	M II,2			The modified instruction word, that is the order-pair II,2 and II,3, is stored at the same address where it was before.
2nd case; II,2 is the right order				
V,1	$\xrightarrow{1}$ 38	2^{-39}		
2	(+) II,2	(II, 2) + 2^{-39}		
3	M II, 2	II,1 and II,2 at the same address.		

2. Preliminary and Final Coding of the Loop for $S_{100} = \sum_{k=1}^{100} a_k$.

Seq.	Code	Word	Order	R1	R2	Memory	Description
I,1	K4000	KN0	+ A1	$S_1 = a_1$		$S_i = a_1$	
2	10065		M B2				$S_i \rightarrow B2$
3	28006	KN1	$\rightarrow$ 6 (1 struck through)	1.2^{-7}			
4	10066		M B3			1.2^{-7}	$1.2^{-7} \rightarrow B3$
II,1	K4065	KN2	+ B2	S_i			
2	N4001		(+) [A2]	$S_i + a_{i+1}$			
3	10065	KN3	M B2		S_{i+1}		$S_{i+1} \rightarrow B2$
III,1	K4064		+ B1	$2^{-7}.98$			
2	04066	KN4	(-) B3	$2^{-7}(98-i)$			
3	40KN6		C'V,1				Transfer to V,1 if $98-i \geq 0$
IV,1	S4065	KN5	R B2		S_{100}		
2	L4028		P				Print S_{100}
3	00000	KN6	Zu				Stop
V,1	28026		$\rightarrow$ 38 (1 struck through)	1.2^{-39}			
2	N4KN2	KN7	(+)II,2	$II,2+2^{-39}$			
3	10KN2		M II,2		Modified word, II,1 and II,2, back at its address.		
4	28006	KN8	$\rightarrow$ 6 (1 struck through)	1.2^{-7}			
5	N4066		(+) B3	$2^{-7}(i+1)$			
6	10066	KN9	M B3			$2^{-7}(1+i) \rightarrow B3$	
7	N0KN2		U II,1				Transfer to II,1

Notes.

a) The orders V,1; V,2; V,3 instruct the machine to modify the address-part of II,2, which is the right-order. The modification is accomplished by adding 2^{-39} to the order pair II,1 and II,2. The preliminary

symbol of the address-part of II,2 is [A2]. Taking A2 in square brackets reminds the coder that A2 will be modified into A3, A4, ..., Ai+1 after every consecutive run of operations in the loop.

b) The sexadecimal address-part of the shift order V,1 is 026, because 38(10) = 26(16). (See paragraph on shift orders in preceding chapter).

c) The long arrows show where the sexadecimal addresses of the orders III,3; V,2; V,3; and V,7 come from.

d) In the "Description" column we marked only the explanations which we may need later on. The detailed descriptions in previous examples were merely for the purpose of instruction.

4. Assigning Addresses.

0 1 2 4 5 6 7 8 9 K S N J F L | 0 1 3 4 5 6 7 8

00 A1 A2 A3 . . . A16 | 01 | A17 A18 . . .
02 A33 . . . A48 | 03 | A49 A50 . . .
04 A65 . . . A80 | 05 | A81 . . .
06 A97 A98 A100 B1 B2 B3
..
..
KN Computation Routine
KF
..
LN
LF Input Routine

Card Punching. Review Let us review shortly the topic of Chapter III: how to prepare a computation routine for card punching. In that Chapter we inserted in each sexadecimal instruction word (of ten sexadecimal digits) a "0" after the second and the seventh digit, thus obtaining a twelve sexadecimal digit word. For example, in our handwritten routine the word containing the order-pair I,1 and I,2 is:

I,1 I,2
K4000 10065

which would be changed so that in punched form it would appear:

K4 0 000 10 0 065.

The spacings in each of these two representations of words are only to show the two orders and to indicate where the "O"'s were inserted. Let us review again the structure of an order. In the sexadecimal representation in the left-order and the right-order respectively the first two digits represent the instruction-part, the last three digits the address-part. For example in the order I,l, the digits K4 are the instruction-part, and the digits 000 are the address-part. The procedure of inserting "O"'s for card punching can be restated as follows: The "O"'s are inserted immediately after the instruction-part (or before the address-part) in each order. There are exceptions in this procedure (of inserting "O"'s) which are explained in Chapter VII.

All sexadecimal words in the routine are transcribed on binary cards. Figure 4 shows a binary card to remind the student of the grouping and correspondences on the card.

K	Y Y Y Y	Y Y Y Y	Y Y	Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y	Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y	Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y Y Y	Y Y	Y Y	Y Y Y Y	Y Y Y Y	M
W1	X X X X	X X X X	X X	X X	X X X X	X X X X	X X X X	X X X X	X X	X X	X X X X	X X X X	X X X X	X X X X	X X	X X	X X X X	X X X X	X X X X	X X X X	X X	X X	X X X X	X X X X	W2
W3	0 0 0 0	0 0 0 0	0 0	0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0	0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0	0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0	0 0	0 0 0 0	0 0 0 0	W4
W5	1 1 1 1	1 1 1 1	1 1	1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1	1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1	1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1	1 1	1 1 1 1	1 1 1 1	W6
W7	2 2 2 2	2 2 2 2	2 2	2 2	2 2 2 2	2 2 2 2	2 2 2 2	2 2 2 2	2 2	2 2	2 2 2 2	2 2 2 2	2 2 2 2	2 2 2 2	2 2	2 2	2 2 2 2	2 2 2 2	2 2 2 2	2 2 2 2	2 2	2 2	2 2 2 2	2 2 2 2	W8
W9	3 3 3 3	3 3 3 3	3 3	3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3	3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3	3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3	3 3	3 3 3 3	3 3 3 3	W10
W11	4 4 4 4	4 4 4 4	4 4	4 4	4 4 4 4	4 4 4 4	4 4 4 4	4 4 4 4	4 4	4 4	4 4 4 4	4 4 4 4	4 4 4 4	4 4 4 4	4 4	4 4	4 4 4 4	4 4 4 4	4 4 4 4	4 4 4 4	4 4	4 4	4 4 4 4	4 4 4 4	W12
		A			B			C			D			E			F			G			H		
W13	5 5 5 5	5 5 5 5	5 5	5 5	5 5 5 5	5 5 5 5	5 5 5 5	5 5 5 5	5 5	5 5	5 5 5 5	5 5 5 5	5 5 5 5	5 5 5 5	5 5	5 5	5 5 5 5	5 5 5 5	5 5 5 5	5 5 5 5	5 5	5 5	5 5 5 5	5 5 5 5	W14
W15	6 6 6 6	6 6 6 6	6 6	6 6	6 6 6 6	6 6 6 6	6 6 6 6	6 6 6 6	6 6	6 6	6 6 6 6	6 6 6 6	6 6 6 6	6 6 6 6	6 6	6 6	6 6 6 6	6 6 6 6	6 6 6 6	6 6 6 6	6 6	6 6	6 6 6 6	6 6 6 6	W16
W17	7 7 7 7	7 7 7 7	7 7	7 7	7 7 7 7	7 7 7 7	7 7 7 7	7 7 7 7	7 7	7 7	7 7 7 7	7 7 7 7	7 7 7 7	7 7 7 7	7 7	7 7	7 7 7 7	7 7 7 7	7 7 7 7	7 7 7 7	7 7	7 7	7 7 7 7	7 7 7 7	W18
W19	8 8 8 8	8 8 8 8	8 8	8 8	8 8 8 8	8 8 8 8	8 8 8 8	8 8 8 8	8 8	8 8	8 8 8 8	8 8 8 8	8 8 8 8	8 8 8 8	8 8	8 8	8 8 8 8	8 8 8 8	8 8 8 8	8 8 8 8	8 8	8 8	8 8 8 8	8 8 8 8	W20
W21	9 9 9 9	9 9 9 9	9 9	9 9	9 9 9 9	9 9 9 9	9 9 9 9	9 9 9 9	9 9	9 9	9 9 9 9	9 9 9 9	9 9 9 9	9 9 9 9	9 9	9 9	9 9 9 9	9 9 9 9	9 9 9 9	9 9 9 9	9 9	9 9	9 9 9 9	9 9 9 9	W22

IBM 831356 ORDVAC

Figure 4

The word labeled K represented by the first forty characters ("Y"s), in the first row, is for a "key-word", and the word labeled M, represented by the other forty characters ("Y"s), in the same row is for the "modifier word". The key word and the modifier word are explained in detail in Chapter VII.

We can now leave the punching and the transcribing of the routine for S_{100} to the student.

Polynomials. The functions called polynomials play important roles in machine computations. A polynomial of nth degree has the form: $f(x) = a_o + a_1 x + a_2 x^2 + \ldots\ldots + a_n x^n$, where $n = 1, 2, 3, \ldots\ldots$; and employing the summation notation, $f(x) = a_o + \sum_{k=1}^{n} a_k x^k$. The coding of loops for polynomials involves modification of addresses. We can use two different formulas.

a) the first formation formula is, $S_{i+1} = S_i + a_{i+1} x^{i+1}$,

where $S_i = a_o + \sum_{k=1}^{i} a_k x^k$.

Thus, $S_1 = a_o + a_1 x^1$

$S_2 = S_1 + a_2 x^2$

.

.

$S_n = S_{n-1} + a_n x^n$.

The derivation of the above formation formula is omitted, because it is almost self-evident.

b) The second formation formula requires a little change in the notation. Let the n-th degree polynomial be $S_{n+1} = f(x) = a_o x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$. The subscript n+1 does not refer to the degree, n, of the polynomial, but to the number of terms. The formation formula is as follows:

$S_{i+1} = xS_i + a_i$, where $S_i = xS_{i-1} + a_{i-1}$. Thus $S_1 = a_o$

$$S_2 = xS_1 + a_1$$

$$\cdot \quad \cdot \quad \cdot$$

$$\cdot \quad \cdot \quad \cdot$$

$$S_n = xS_{n-1} + a_{n-1}$$

$$S_{n+1} = xS_n + a_n.$$

The above formula is referred to as the "nesting procedure". The derivation of the nesting procedure is explained on the example of a 4-th degree polynomial, $S_5 = a_o x^4 + a_1 x^3 + a_2 x^2 + a_3 x + a_4$. Factor out and group as follows:

$$\begin{aligned} S_5 &= (a_o x^3 + a_1 x^2 + a_2 x + a_3)x + a_4, \\ &= ((a_o x^2 + a_1 x + a_2)x + a_3)x + a_4, \\ &= (((a_o x + a_1)x + a_2)x + a_3)x + a_4, \\ &= ((((a_o)x + a_1)x + a_2)x + a_3)x + a_4. \end{aligned}$$

If we call, $S_1 = a_o$; $S_2 = xa_o + a_1$; $S_3 = x(xa_o + a_1) + a_2$;

$S_4 = x(x(xa_o + a_1) + a_2) + a_3$;

then $S_2 = xS_1 + a_1$; $S_3 = xS_2 + a_2$; $S_4 = xS_3 + a_3$;

$S_5 = xS_4 + a_4$, and in every case $S_{i+1} = xS_i + a_i$, which is the formation formula for the nesting procedure.

Exercises. Program and code loops for the following computations, assuming that no scaling is necessary:

1. $S_{12} = \sum_{k=1}^{12} a_k x^k$;

2. $Q = \sum_{k=1}^{18} a_k x^{2k-2} \Big/ \sum_{k=1}^{83} b_k x^{2k-1}$

Testing an integer for evenness or oddness. We shall remind the student that in a machine number $e_o e_1 \ldots e_{39}$ the first bit, e_o, is the sign bit. When e_o is zero the machine interprets the number as a positive number, when e_o is one, the machine interprets this machine number as a negative number (See Chapter 1, Complements). Another thing which the student must bear in mind is that an even integer in binary representation has the last bit (the least significant bit) zero, and an odd integer in binary representation has the last bit one. For example, 5(10) = 101(2), and 6(10) = 110(2).

The machine representation of 5.2^{-3} is A = 0101 0000 ....0(36 zeros)

The machine representation of 6.2^{-3} is B = 0110 0000 ....0(37 zeros)

If we could shift A left 3 (multiply by 2^3) so that e_3 would replace e_o, then, after the shift, the first bit would be 1. This machine number the machine would interpret as a negative number. If we shift B in a similar way the first bit would be 0 and the machine would interpret this as a positive number. The above example illustrates a general rule for testing an integer by the machine for evenness or oddness, which is as follows: To test an integer M scaled by a factor 2^{-n} for evenness or oddness shift the machine number $M.2^{-n}$ left n (that is multiply it by 2^n) and test the shifted number as to whether it is positive or negative. If the shifted number, namely the original e_n (now in the e_o position) is zero (interpreted by the machine as indicating a positive number), then M is even; if the shifted number is one (interpreted as indicating a negative number), then M is odd. However, none of the shift orders already mentioned, can instruct the machine to replace e_o, that is, they shift only through e_1, leaving e_o unchanged (see Chapter IV, Shift Orders). In order to accomplish the left shift by n places of the scaled number $M.2^{-n}$ mentioned above, we shall introduce now a shift order that would instruct the machine to shift through e_o as well. Let the initial contents of R1 be $e_o e_1 \ldots e_{39}$, of R2 be $d_o d_1, \ldots d_{39}$.

Preliminary Symbol	Sexadecimal Symbol	(R1) after the shift
←S— (any)	74...	$e_1 e_2 \ldots e_{39} d_1$

The address is ignored

The order ←S— shifts left 1 the contents of R1 through e_o as well. Thus, e_o is replaced by e_1, e_1 by e_2, and so on. The last bit, e_{39}, is replaced by d_1 (from R2). Thus the shifting of a machine number, $2^{-n}M$, left n has to be done in two stages : shift left (n-1) by the conventional order ←— (n-1), which replaces e_1 by the last bit of the integer M (e_n); then shift left 1 by the order ←S— , which replaces e_o by e_1. The test for evenness or oddness is represented on a flow chart in a separate box shown in Figure 5.

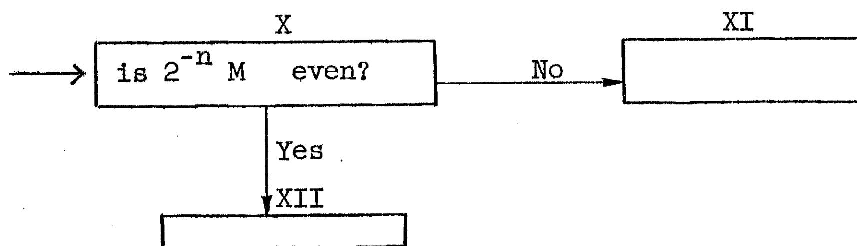


Figure 5

The operations in Box X, which tests 2^{-n} M (stored at D1) for evenness or oddness are coded as follows:

Sequence	Tape	Word	Order	R1	Description
X,1			+ D1	2^{-n} M	$(R1) = 2^{-n}M = e_o e_1 \ldots e_n \ldots e_{39}$
2			← (n-1)	$(2^{-n}M).2^{n-1}$	$(R1) = e_o e_n \ldots e_{39} 00.0$ (n-1 zeros)
3			←S—	$(2^{-n}M).2^{n}$	$(R1) = e_n \ldots e_{39} 0 \ldots 0$
4			C XII,1		If M is even, that is, if $(R1) \geqslant 0$ direct control to XII,1. If M is odd, that is, if $(R1) < 0$, direct control to XI,1.
X1,1			------		

Alternating Sums. A sum of the form $S_n = u_1 - u_2 + u_3 - u_4 - \ldots\ldots$, $n = 2, 3, 4, \ldots$, where the signs of the consecutive terms alternate from plus to minus and from minus to plus is called an "alternating sum". Introducing the sign changer, $(-1)^{k+1}$, k=1, 2, 3,.., we can write an alternating sum in a shorthand notation, $S_n = \sum_{k=1}^{n} (-1)^{k+1} u_k$. Several methods of

programming machine computation for alternating sums are explained in the following example.

<u>Example</u>: To program and to code a loop for $S_{10} = \sum_{k=1}^{10} (-1)^{k+1} a_k$, assuming that no scaling is necessary and all the terms $a_1, a_2, \ldots, a_{10}$ are positive numbers.

1. <u>Programming</u>.

<u>Method I</u>. The terms $a_1, a_2, \ldots a_{10}$ are stored as positive numbers. We shall use two formation formulas:

a) $S_{i+1} = S_i + a_{i+1}$, for i even, and

b) $S_{i+1} = S_i - a_{i+1}$, for i odd, where: $S_i = \sum_{k=1}^{i} (-1)^{k+1} a_k$,

$S_1 = a_1$

$S_2 = S_1 - a_2$

$S_3 = S_2 + a_3$

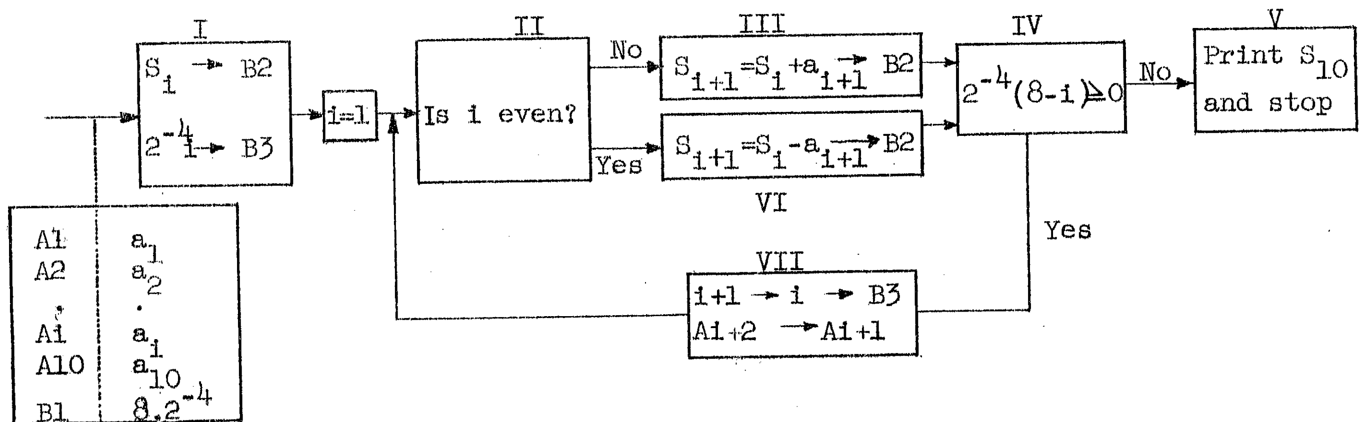
. . .

. . .

$S_{10} = S_9 - a_{10}$.

We shall test i for evenness or oddness; when i is even we shall apply formula b), when i is odd we shall apply formula a).

<u>Flow Chart</u>.



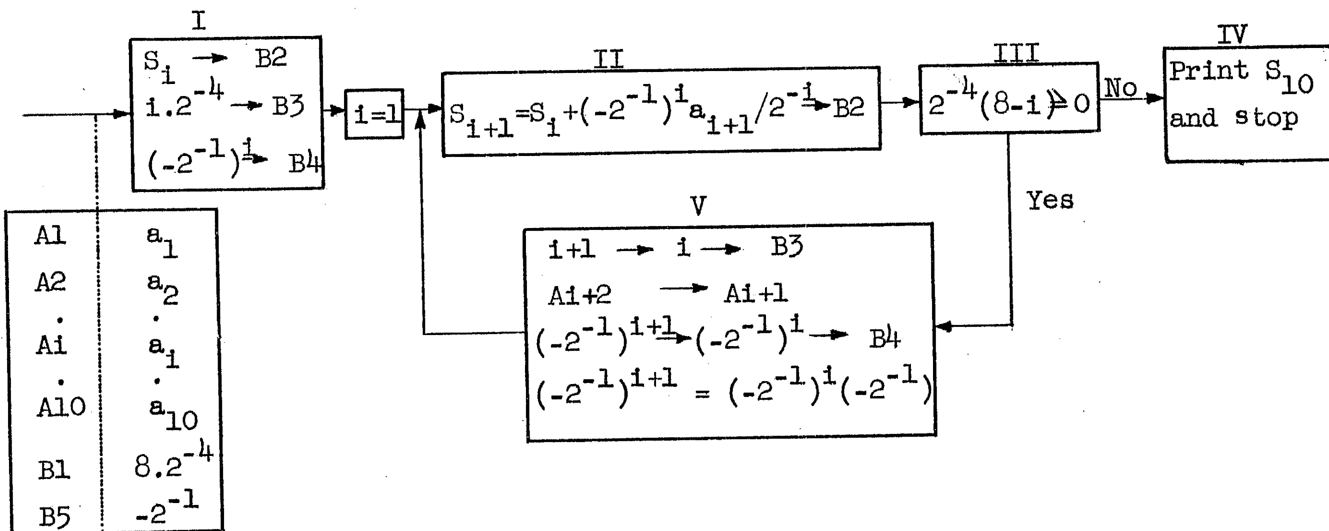
Method II. The terms a_1, a_2, ... a_{10} are stored as positive numbers. We shall use the formation formula $S_{i+1} = S_i + (-1)^i a_{i+1}$ where:

$$S_1 = a_1$$
$$S_2 = S_1 + (-1)^1 a_2$$
$$S_3 = S_2 + (-1)^2 a_3$$
$$\cdot$$
$$\cdot$$
$$S_n = S_{n-1} + (-1)^{n-1} a_n.$$

In every partial sum, S_{i+1}, the term a_{i+1} is multiplied by $(-1)^i$. The factor (-1) would be scaled by 2^{-1}; therefore we would multiply a_{i+1} by the factor $(-2^{-1})^i$, and afterwards divide the term a_{i+1} by 2^{-i} (or multiply by 2^i, that is, shift left i, see note d). Thus, the operations in the formation formula (by the Method II) would be

$$S_{i+1} = S_i + (-2^{-1})^i a_{i+1}/2^{-i}.$$

Flow Chart.



We shall show the preliminary coding of the Boxes II and V.

Seq.	Tape	Order	Symbol	R1	R2	Memory
II,1			$\lvert + \rvert$ B4	$\lvert(-2^{-1})^i\rvert = 2^{-i}$		
2			M T1			$2^{-i} \rightarrow$ T1
3			R B4		$(-2^{-1})^i$	
4			Xu [A2]	$(-2^{-1})^i a_{i+1}$		
5			÷ T1		$(-2^{-1})^i a_{i+1}/2^{-i} = (-1)^i a_{i+1}$	
6			A + II,8	$(-1)^i a_{i+1}$		
7			(+)B2	$S_i + (-1)^i a_{i+1} = S_{i+1}$		
8			M B2			$S_{i+1} \rightarrow$ B2
V,1			$\overrightarrow{1}$ 3	1.2^{-4}		
2			(+) B3	$2^{-4}(i+1)$		
3			M B3			$2^{-4}(i+1) \rightarrow$ B3
4			$\overrightarrow{1}$ 38	1.2^{-39}		
5			(+) II,4	$(II,4)+1.2^{-39}$		
6			M II,4			The order-pair II,3; and II,4 at its previous address
7			R B5		-2^{-1}	
8			Xu B4	$(-2^{-1})^i(-2^{-1})$		
9			M B4			$(-2^{-1})^{i+1} \rightarrow$ B4
10			U II,1	Transfer control to II,1.		

Notes: a) The orders II,1; II,2 form and store at T1 the absolute value of $(-2^{-1})^i$, that is $\lvert(-2^{-1})^i\rvert = 2^{-i}$, which we need later on as a divisor for a_{i+1}.

b) The preliminary storage of -2^{-1} (see Storage Box) is convenient, because replacing $(-2^{-1})^i$ by $(-2^{-1})^{i+1}$ in Box V involves multiplication by -2^{-1}.

c) The address-part of the order II,4 is [A2]; square brackets mark the modification. The orders V,4; V,5; and V,6 instruct the modification of the address-part of the order II,4 which we assumed to be the right order.

d) The alternative coding of Boxes II and V involves "modification of the amount of shift". The sexadecimal representation of the order $\leftarrow n$, which shifts left n the contents of R1 (multiplies (R1) by 2^n), is $18\overset{n}{\ldots}$ (see Chapter 4, Shift Orders). The three dots are filled not by an address but by the three digit sexadecimal equivalent of n. For example, the sexadecimal representation of $\leftarrow 5$ is 18005; the sexadecimal representation of $\leftarrow 38$ is 18026. The analysis carried out in explanation for address-modification can be carried out identically for modification of the amount of shift. To increase the amount of shift from n to n+1 we add to the instruction word 2^{-19} when a shift order is the left order or 2^{-39} when a shift order is the right order.

Using the order $\leftarrow n$ which instructs to multiply (R1) by 2^n, we can write the operations in Box II as follows:

II

$$S_{i+1} = S_i + (-2^{-1})^i a_{i+1}(2^i)$$

and the operations in the substitution Box V:

V

$$i+1 \longrightarrow i \longrightarrow B3$$
$$Ai+2 \longrightarrow Ai+1$$
$$(-2^{-1})^{i+1} \longrightarrow (-2^{-1})^i$$
$$2^{i+1} \longrightarrow 2^i$$

The preliminary coding for this alternative is:

Seq.	Tape	Order	Symbol	R1	R2	Memory
II,1			R B4		$(-2^{-1})^i$	
2			Xu[A2]	$(-2^{-1})^i a_{i+1}$		
3			$\leftarrow$[1]	$(-2^{-1})^i a_{i+1}(2^i)=(-1)^i a_{i+1}$		
4			(+) B2	$S_i + (-1)^i a_{i+1} = S_{i+1}$		
5			M B2			S_{i+1} B2
V,1			$\nrightarrow$ 3	1.2^{-4}		
2			(+) B3	$2^{-4}(i+1)$		
3			M B3			$2^{-4}(i+1) \longrightarrow$ B3
4			$\nrightarrow$ 38	1.2^{-39}		
5			(+) II,2	$(II, 2)+1.2^{-39}$		changed pair
6			M II,2			(II,1 and II,2) at its previous address
7			R B5		-2^{-1}	
8			Xu B4	$(-2^{-1})^i(-2^{-1})$		
9			M B4			$(-2^{-1})^{i+1} \longrightarrow$ B4
10			$\nrightarrow$ 18	1.2^{-19}		
11			(+) II,3	$(II, 3)+1.2^{-19}$		changed pair
12			M II,3			(II,3 and II,4) at its previous address
13			U II,1	Transfer control to II,1		

Notes: a) The orders V,4; V,5; V,6 modify the address of II,2; the orders V,10; V,11; V,12 modify the amount of shift in II,3. The amount of shift in the order II,3, [1] , is in square brackets to mark the subsequent modifications which will make it 2, 3, ...i.

The alternative coding of Boxes II and V results in eighteen orders, and in this respect it has no advantage over the first version, which also has eighteen orders. The advantage of the alternative version over the initial one consists in not having any division orders. When possible, division should be avoided for two reasons: 1) it introduces round-off error (see Chapter 2, Short List of Orders), and 2) it slows down the computation (indeed it takes more time than other arithmetic operations).

Method III. The terms a_2, a_4, a_6, a_8, a_{10}, are subtrahends. If we store them initially as negative numbers, that is if we store $-a_2$ at A2, $-a_4$ at A4, and so on, then we can compute S_{10} as a non-alternating sum, that is, $S_{10} = \sum_{k=1}^{10} a_k$.

Method IV. All terms are stored as positive numbers. If we add separately the minuends, $a_1 + a_3 + a_5 + a_7 + a_9 = \sum_{k=1}^{5} a_{2k-1} = M_5$, and separately the subtrahends, $a_2 + a_4 + a_6 + a_8 + a_{10} = \sum_{k=1}^{5} a_{2k} = N_5$, then, $S_{10} = M_5 - N_5$. The formation formula for M_5 is $M_{i+1} = M_i + a_{2i+1}$, $M_1 = a_1$, and for N_5 is $N_{i+1} = N_i + a_{2i+2}$, $N_1 = a_2$.

Flow Chart.

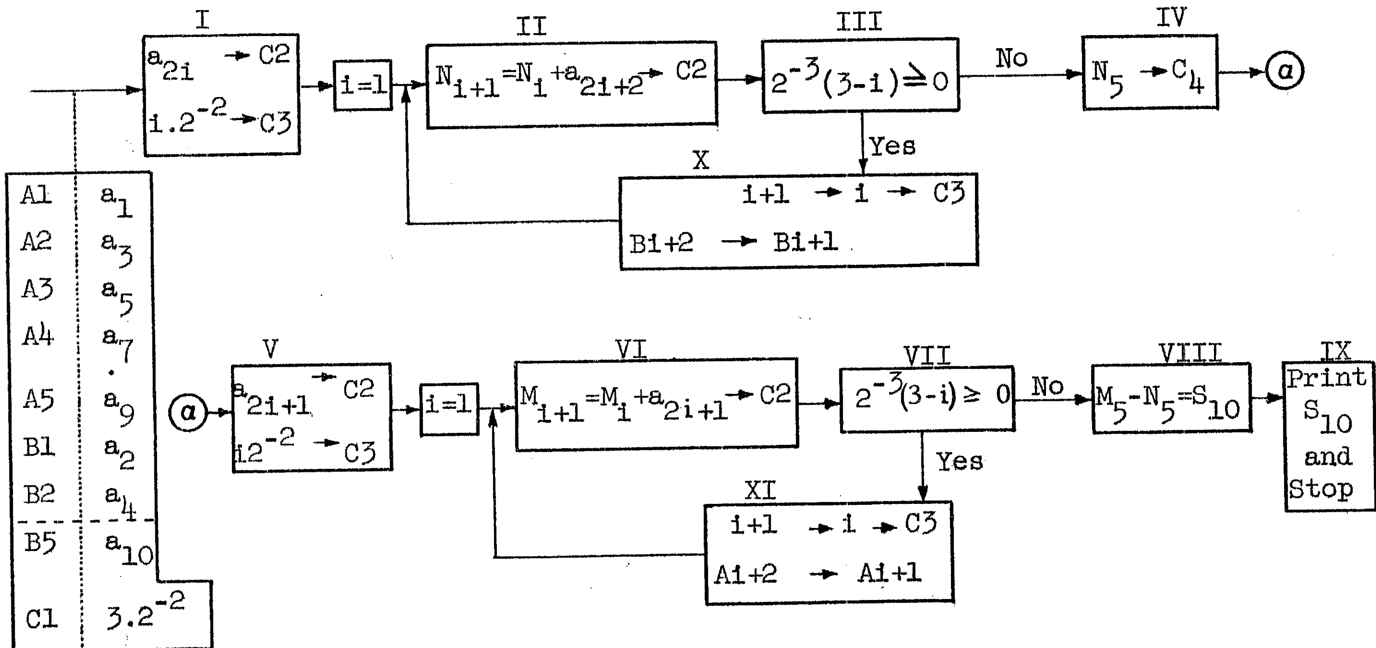
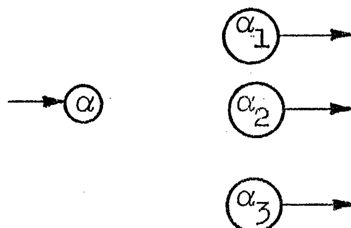


Figure 6

Notes: a) The operations in Boxes I, II, III, IV, and X compute N_5 and store it at C4. The operations in Boxes V, VI, VII, VIII, IX, and XI compute M_5, compute S_{10} and print S_{10}. As a rule, the computation of subtrahend (and divisor) precedes the computation of minuend (dividend).

b) The little circle enclosing "α", which immediately follows Box IV and another one like that which precedes Box V indicate a "remote connection". We cut the branch connecting Boxes IV and V, ending each part of the broken branch with a circle marked by "α" which indicates that Boxes IV and V are connected. Remote connections are useful when a connected flow chart would be too big for one sheet of paper. The remote connections used in our flow chart →(α) (α)→ are of the "fixed type". There are also cases necessitating several possible continuations.



These are called "variable remote connections".

c) Both loops, for N_5 and for M_5, involve modification of addresses.

2., 3., & 4. Coding, Assigning Addresses, Card Punching, and Transcribing for the Computation of S_{10}.

These, for each method, are left to the student as an exercise.

Choosing the method which is most appropriate for a given alternating sum.

For the sum $S_{10} = \sum_{k=1}^{10} (-1)^{k+1} a_k$ Method III of storing subtrahends as negative numbers is the simplest and the most appropriate. But for other sums that is not the case always. Take for example the sum

$S_n = \sum_{k=1}^{n} (-x)^{k+1}/(k-1)! = 1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots$ The formation formula

for this sum is $S_{i+1} = S_i + u_{i+1}$, where, $u_{i+1} = u_i(-x/i)$,

$$u_i = (-x)^{i-1}/(i-1)! \;, \quad S_i = \sum_{k=1}^{i} (-x)^{k-1}/(k-1)!$$

Thus, $S_1 = u_1 = 1$

$$S_2 = S_1 + u_1\,(-x/1) = S_1 + u_2$$

$$S_3 = S_2 + u_2\,(-x/2) = S_2 + u_3$$

$$\cdot \quad \cdot$$

$$S_n = S_{n-1} + u_{n-1}(-x/(n-1)) = S_{n-1} + u_n.$$

If we compute for a desired accuracy, that is for $\big||S_{i+1}| - |S_i|\big| < E$, the methods of adding separately minuends and subtrahends or of storing subtrahends as negative numbers are not practical because we do not know in advance how many terms will be needed. Indeed the storage of terms is completely unnecessary because each term u_{i+1} can be formed by multiplying the preceding term u_i by $(-x/i)$. A method similar to Method II would be the most appropriate in this case. Programming and coding of this problem is left as an exercise to the student.

Different problems call for different methods, and it would not be practical to give a general rule as to which method should be used. The best guide in this matter is a little experience.

Exercises.

1. Program and code the loop for $S_n = \sum_{k=1}^{n} (-1)^{k-1} x^{2k-2}/(2k-2)!$ with accuracy given by the condition that $\big||S_{i+1}| - |S_i|\big| < E$. Use Method II. (For the purpose of scaling assume that $n \leq 20$; $|x| < 1$).

2. Program and code the loop for $S_{30} = \sum_{k=1}^{30} (-1)^{k-1}/k$, using all four methods.

Selecting the greatest number from a set of n numbers.

To sort a set of n distinct numbers, $a_1, a_2, \ldots, a_n$, means to arrange them or to store them at consecutive addresses in decreasing or increasing magnitude. Assume that resulting from some machine computations we have one hundred distinct numbers, $a_1, a_2, \ldots, a_{100}$, stored respectively at the addresses A1, A2, ..., A100, and we want to instruct the machine to sort these numbers, printing them or storing them at B1, B2, ..., B100 in decreasing magnitude. An outline of instructions for the machine could be: select the greatest number from the set of one hundred numbers and print it, or store it at B1; then select the greatest number from the set of the remaining ninety-nine numbers and print it, or store it at B2; etc, Thus instructed, the machine could perform repeatedly operations of selecting and storing, or printing the greatest number. For this reason we shall begin from the program of operations of selecting the greatest number, which is a principal operation of sorting.

Example.

To select and print the greatest number from the set of one hundred distinct positive numbers $a_1, a_2, \ldots, a_{100}$, stored respectively at the addresses A1, A2, ..., A100.

Programming. An outline of a program to select the greatest number would be as follows: compare a_1 and a_2, select the greater of the two, rename it a_m, and store it at some special address, say B1; then compare a_m with a_3, select the greater of these two, and store it again at B1; then compare a_m from the preceding operation with a_4, select the greater of the two, and so on until all one hundred numbers are tested. The operation of comparing a_m with a_i, $i = 2, 3, 4, \ldots, 100$, has to be repeated 99 times, suggesting a loop. The machine will be instructed to test the difference $a_m - a_i$; if $a_m - a_i \geq 0$, then a_m would remain the greatest of all the tested numbers and will be left at B1; if, $a_m - a_i < 0$, then a_i (being the greater of the two) would be promoted to a_m and stored at B1, replacing the former a_m. The number stored at any given time at B1 would be the greatest number of all the numbers tested up to that time, the a_m.

After the 99th test, a_m will be the greatest of all the numbers; a_m renamed as a_M to distinguish it as the greatest of all, will then be printed. The numbers a_1, a_2, ..., a_{100} are initially stored in the memory, which implies that each of them in absolute value is less than one.

Flow Chart

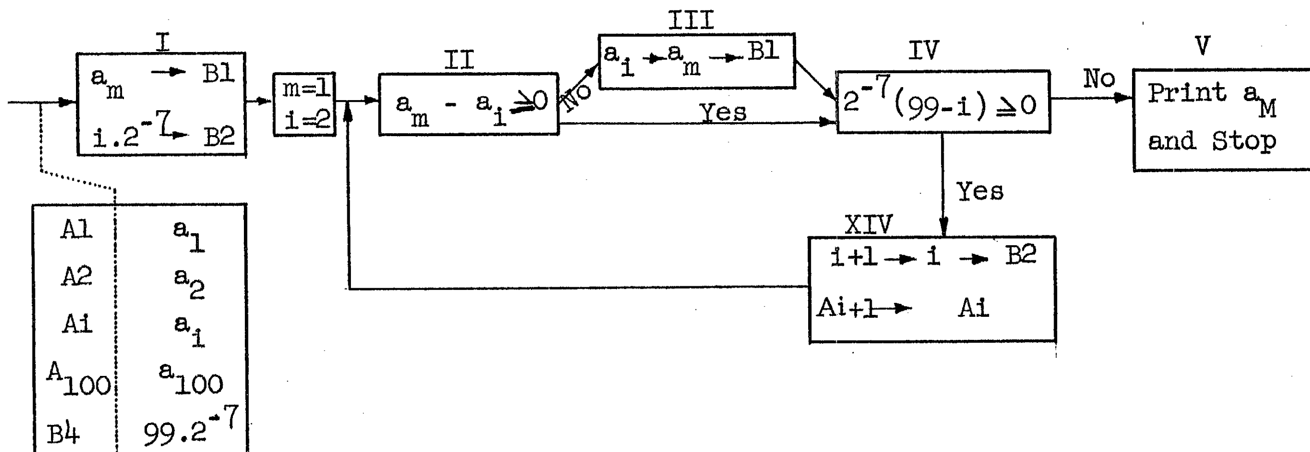


Figure 7

2., 3., and 4. Coding, Assigning Addresses, Punching, and Transcribing.

This is left to the student as an exercise.

Sorting.

Example: To sort and print one hundred positive, distinct numbers, a_1, a_2, ..., a_{100}, stored respectively at the addresses A1, A2, ..., A100.

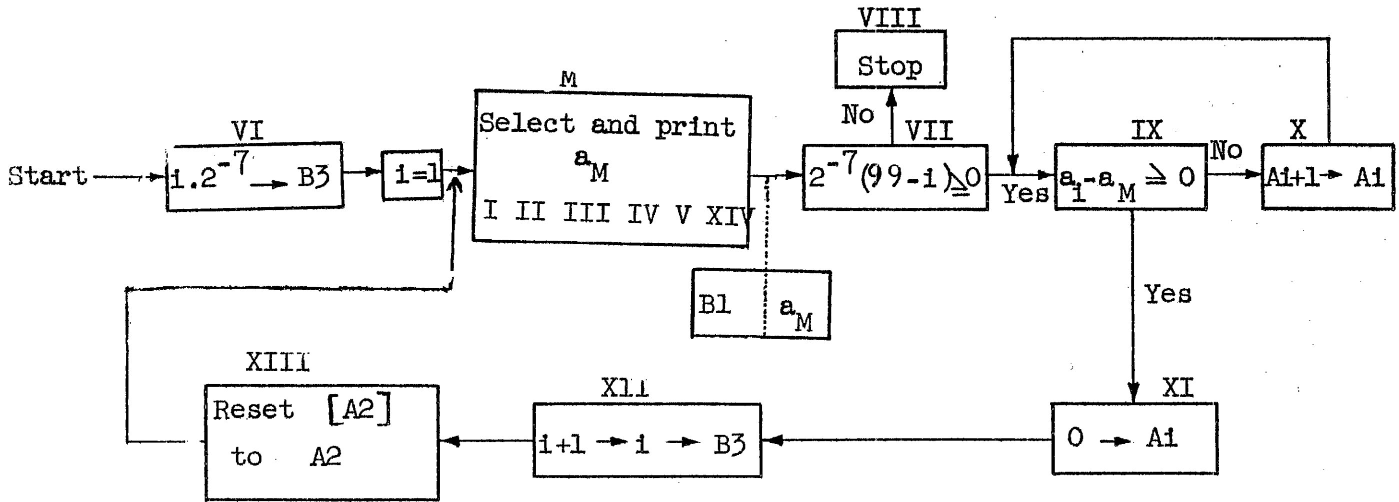
1. Programming. We shall repeat the outline of a program for the operations of sorting a_1, a_2, ..., a_{100}: select the greatest number from the set of one hundred and print it; then select the greatest number from the set of the remaining ninety-nine numbers and print it; etc. We know how in the first run the machine selected the greatest number from the initial set of one hundred. We shall consider now the second run of selecting the greatest number from the set of the remaining ninety-nine.

We must bear in mind that after the selection of a_M all the numbers, including a_M, are still stored in the memory at their initial addresses, A1, A2, ..., A100, because none of them was replaced by something else. In order to be able to select the greatest number of the remaining ninety-nine positive numbers we must first erase a_M (that is, replace it by 0) from its initial memory position (a_M is also stored at B1 which is not the initial memory position). The trouble is that we do not know where a_M was initially stored. Thus, we must instruct the machine to find the initial storage of a_M, to erase it, and after the erasure to direct the second run of operations which would select and print the greatest number of the remaining ninety-nine numbers. The course of operations is now obvious; selecting and printing the greatest number, erasing it from its initial memory position, selecting and printing the next greatest number, erasing it from its initial memory position, and so on.

To find the initial address of a_M we would instruct the machine thus: compare a_1 with a_M, if $a_1 - a_M < 0$, then a_1 is not the a_M, therefore continue comparing a_2 with a_M, a_3 with a_M, and so on, until at a certain step some a_i equals a_M. The address of this a_i is Ai, which would also be the address of a_M, because a_i is the a_M. Erase a_M at Ai (ordering: oM Ai; see List of Orders) and direct control to Box I (see Figure 7) for the next run of operations selecting a_M.

The flow chart for sorting $a_1, a_2, \ldots, a_{100}$ will contain the flow chart for selecting a_M from $a_1, a_2, \ldots, a_{100}$ with a little change, namely, in Box V the operation "Stop" will be erased. For the sake of compactness the whole flow chart for selecting a_M will be represented by a single box labeled M.

General Flow Chart for Sorting $a_1, a_2, \ldots, a_{100}$.



Flow Chart of Box M Selecting and Printing a_M is in Figure 7.

Notes:

a) The operations in the comparison Box IX compare a_i with a_M. If $a_i - a_M < 0$ control goes to the substitution Box X, where the operations replace a_i by a_{i+1} (by modifying the address), and control goes back to Box IX, where a_{i+1} is compared with a_M. The operations in Boxes IX and X will cycle as long as $a_i - a_M$ is less than zero. When for some a_i the difference $a_i - a_M = 0$, then control goes to the erasure Box XI, where the operations erase a_M from the address Ai (storing 0 at Ai) and transfer control to the counter in Box XII.

b) The operations which modify addresses and the operations of the counter are usually grouped in the same box, but in this problem we must have them apart, in Boxes X and XII, because address modifications are needed only as long as a_i's remain smaller than a_M, whereas i in Box VI can be replaced by i+1 only after the machine found the a_i which equals a_M and erased it from Ai.

c) The storage Box reminds the coder that the operations in Box M stored a_M at B1.

d) The preliminary coding of Boxes IX, X, XI. Coding of Boxes IX, X and XI requires an application of "extract orders", which are explained below.

e) Before the second, third, fourth, and so on, run of operations in M, all the addresses that were modified have to be reset to their initial values, which is performed in Box XIII. This demands that A2 be stored at say B6 which is most conveniently done in the storage Box of M.

Extract Orders. In order to code the operations in the erasure Box XI we shall get acquainted with the "extract orders", which instruct the machine to change the address-part of instruction words while they are stored in the memory. The operations of address modification change an address by moving the instruction word to the arithmetic unit and performing on it arithmetic operations. Extract orders change in a certain way the address-parts of instruction words without moving them from the memory and without performing arithmetic operations on them.

List of Extract Orders

Orders in this list refer to two instruction words. The first one consisting say of the left-order II, 1 and the right-order II,2 is stored at the address A1. The second one consisting say of the left-order X,3 and the right-order X,4 is held in the register R1. Thus, using sexadecimal representation, the initial contents of A1 is $a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10}$, and the initial contents of R1 is $r_1r_2r_3r_4r_5r_6r_7r_8r_9r_{10}$.

No.	Prelim Symbol	Sexadec. Symbol	The Contents after the execution of the order of R1(X,3 and X,4)	A1(II,1 and II,2)
1	E A1	90...	$r_1r_2r_3r_4r_5r_6r_7r_8r_9r_{10}$ (unchanged)	$a_1a_2r_3r_4r_5a_6a_7a_8a_9a_{10}$ The address-part of the left order is replaced by the address part of the left order from (R1).

No.	Prelim Symbol	Sexadec. Symbol	The Contents after the execution of the order of R1(X,3 and X,4)	A1(II,1 and II,2)
2	E' A1	50...	$r_1r_2r_3r_4r_5r_6r_7r_8r_9r_{10}$ (unchanged)	$a_1a_2a_3a_4a_5a_6a_7r_8r_9r_{10}$ The address-part of the right order is replaced by the address-part of the right order from (R1).
3	oE A1	60...	00000 00000 (Contents of R1 erased)	$a_1a_2$0 0 0 $a_6a_7a_8a_9a_{10}$ The address-part of the left order is replaced by 000.
4	oE' A1	70...	00000 00000	$a_1a_2a_3a_4a_5a_6a_7$0 0 0 The address-part of the right order is replaced by 000.

Notes:

a) In the case of Order No. 1 we say that the address from X,3 was "extracted" to II,1. In the case of Order No. 2 we say that the address from X,4 was extracted to II,2.

b) The extract orders like the transfer orders generally refer to instruction words (see this Chapter, Transfer Orders). As we do not know the addresses of instruction words in preliminary coding we must refer to the sequence number, which is a symbol for the contents of a given address. (See this Chapter, Transfer Orders.)

Examples:

1. Given two instruction words, (X,3 and X,4) and (II,1 and II,2) stored respectively at A1 and A2. In preliminary coding the addresses, A1 and A2 are not yet known. To extract the address from X,3 to II,1.

The preliminary coding of the operations which perform this extraction is as follows:

Seq.	Tape	Order	Symbol	R1	Memory
1			+ X,3	X,3 and X,4	
2			E,II,1		The address part of II,1 is replaced by the address-part of X,3.

2. Given a sequence of orders:

Seq.	Symbol	R1	Memory
II,1	+ A1	(A1)	
2	M A2	(A1) ⟶ A2	
3	oMB1	0	Contents of B1 erased
4	U X,1	Transfer control to X,1	

To insert orders between II,2 and II,3 that would extract the address part from II,1 (that is A1) to II,3 (that is to replace the address-part of II,3, B1, by the address part of II,1 which is A1).

Seq.	Symbol	R1	Memory
II,1	+ A1	(A1)	
2	M A2	(A1) ⟶ A2	
3	+ II,1	(II,1 and II,2) = {+A1 and M A2}	
4	E II,5		The address-part of II,5 (that is B1) is replaced by the address-part of II,1 (that is by A1).
5	oM [B1]	0	The address-part of II,5 is no longer B1, but A1; therefore the contents of A1 are erased after the execution of II,5.
6	U X,1	Transfer control to X,1.	

Notes:

a) We assume here that the orders II,1 and II,2 make an order pair. If we know that the instruction word (II,1 and II,2) is stored say at T5, then the order II,3 would be + T5. This we know of course in the final coding and then we have to refer to the address.

b) To be able to extract the address from II,1 we have to move the instruction word (II,1 and II,2) to the register R1, which is performed by the order II,3.

c) II,4 replaces the address-part in the order which follows, that is, in the order II,5. The orders II,4 and II,5 are stored at consecutive addresses and will be executed one after another. The order II,4 will change the address-part of the order II,5 before the order II,5 will be executed. The whole routine is stored in the memory before the machine begins the computations. The orders which precede II,4 and the orders which follow II,4 are in the memory at the same time.

d) Now, the address-part of II,5 (which initially was B1) can be anything, because the preceding order, II,4, replaces whatever was there. The address B1, as in the case of address modification, is in square brackets to mark that it will be subsequently replaced.

e) Normally, the orders II,4 and II,5 cannot make an order pair, meaning that they cannot be in the same instruction word. The order II,4 replaces the address-part of II,5 in the memory. The order II,4 before being executed, would be moved from the memory to the order register R_3, and so would its companion, II,5. Hence the order II,5 cannot be a companion to the order II,4 if it is to be executed immediately after the order II,4. If the pairing of the orders would make II,4 and II,5 an order pair, we must introduce somewhere a "dummy order" to break the pair. A dummy order may be any order that does not change the routine. A transfer order is usually the most convenient for the purpose. For example, inserting an order U II, 3 (transfer control to II,3) between the order II,1 which is + A1 and the order II,3 which is M A2 will not affect the operation (A1) ⟶ A2.

f) The order II,3 moves to R1 the instruction word which we represented by either the sequence numbers of its left and right orders, that is, (II,1 and II,2), or the preliminary symbols of the two orders, that is, + A1 and M A2. In the case of address extraction the second alternative is preferable. In this example both representations are shown in the column for (R1).

Preliminary Coding of the Erasure Box XI. We are now able to code the erasure box XI for the problem of sorting $a_1, a_2, \ldots, a_{100}$. Box IX is also included because it is needed for reference.

Seq.	Tape	Order	Symbol	R1	Memory
IX,1			+ [A1] (+Ai)	$a_i = a_1$	
2			(-) B1	$a_i - a_M$	
3			C XI,1	Transfer control to XI,1 if $a_i - a_M \geq 0$	
XI,1			+ IX,1	(IX,1 and IX,2) = {+Ai and (-) B1}	
2			E XI,3		The address-part of XI,3 (which is A1) is replaced by the address-part of IX,1 (that is A1 is replaced by Ai).
3			oM [A1]		Contents of Ai are erased, because the order XI,2 replaced A1 by Ai.
4			U XII,1	Transfer control to XII,1	

Note:

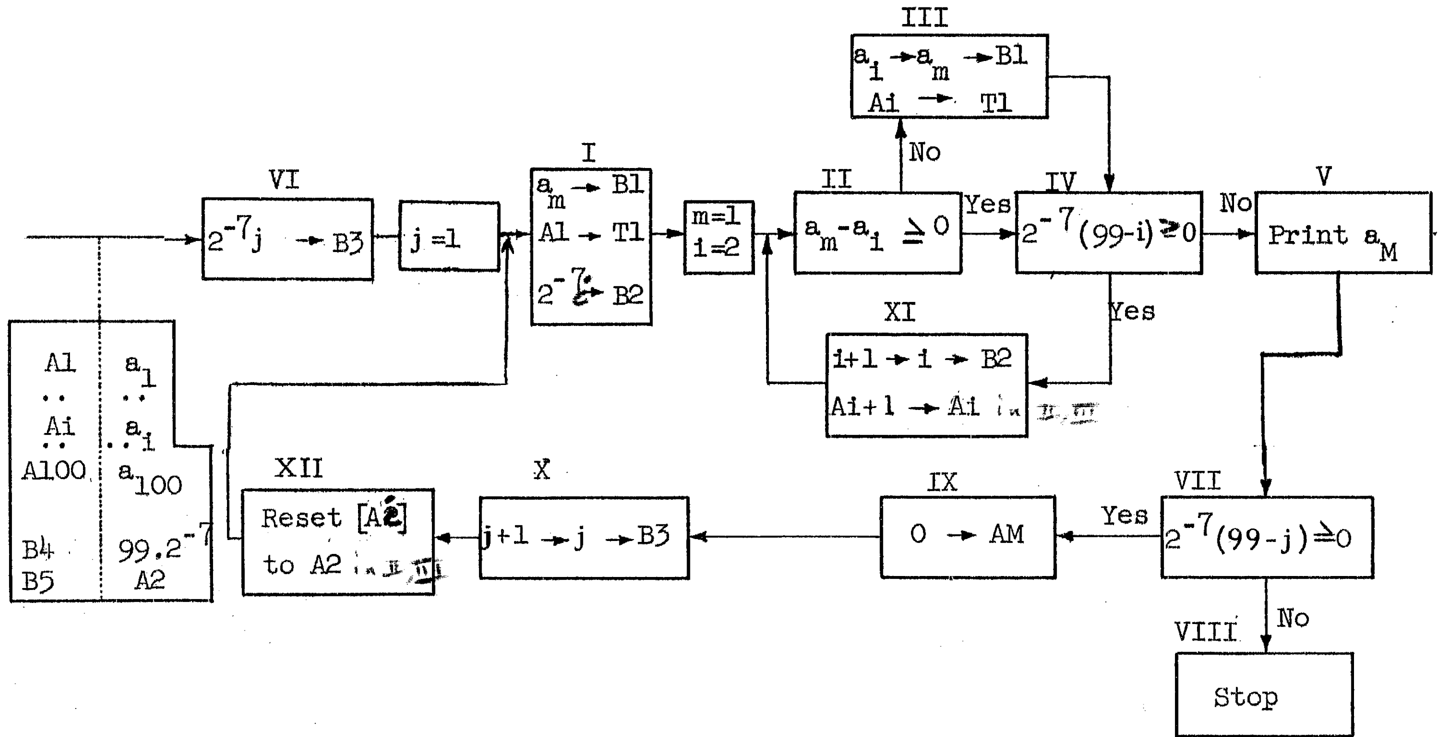
The address-part of the order IX,1, "A1", is in brackets to indicate subsequent modifications, meaning that in the first run it will be A1, in the second it will be A2; then A3, A4, ..., Ai. The greatest number a_M, which we want to erase, is stored at a certain Ai, which is found in Box IX when $a_i - a_M \geq 0$. This address Ai has to be extracted from IX,1 to XI,3, because the order XI,3 erases the contents of the address specified in its address-part.

2., 3., 4. Coding, Assigning Addresses, Punching and Transcribing.
This is left to the student as an exercise.

Alternative Program for Sorting $a_1, a_2, \ldots, a_{100}$.

In the alternative program for sorting $a_1, a_2, \ldots, a_{100}$ the loop of operations selecting the greatest number a_M has additional operations which track the initial address of a_M as well. The alternative program is explained in the flow chart.

Flow Chart.



Notes:

a) In the storage Box I we store not only the first a_m, which is initially a_1, but also its address, or rather the word which contains the address of a_1. Box III, which stores the greatest number at that time, a_m, stores also its address at T1. After ninety-nine runs which cycle either in Boxes II, III, IV, and XI or in Boxes II, IV, and XI, the memory position T1 would contain the address of a_M, which we shall call AM.

b) Compared with the first alternative which had three loops this alternative has only two loops. The small loop selects a_M and consists of Boxes II, III, IV and IX, and the large loop erases a_M, and contains the small loop and the Boxes V, VI, VII, IX, X, and I. After ninety-nine cycles in the small loop follows one run in the large loop.

Preliminary Coding of Boxes I, III, and IX.

The preliminary coding of Boxes I,III, and IX will explain the storing of the address of a_m, that is of Ai, and the erasure of a_M.

Seq.	Tape	Order	Symbol	R1	Memory
I,1			+[A1]	$a_m = a_1$	
2			M B1		$a_m \longrightarrow$ B1
3			+ I,1	I,1 and I,2; (+A1 and M B1)	
4			M T1		(I,1 and I,2) $\longrightarrow$ T1
5			$\to$ 5	$2^{-7}(i) = 2^{-7}(2)$	
6			M B2		$2^{-7}i \longrightarrow$ B2
III,1			+[A2](Ai)	a_i	
2			M B1		$a_i \longrightarrow$ B1
3			+ III,1	(III,1 and III,2) = + Ai and M B1	
4			M T1		(III,1 and III,2) $\longrightarrow$ T1
IX,1			+ T1	(III,1 and III,2) = +AM and M B1	
2			E IX,3		The address-part of III,1 replaced the address-part of IX,3, that is, AM replaced A1.
3			oM [A1]		a_M is erased from AM; the preceding order IX,2 replaced the address-part, A1, by AM.
4			U X, 1	Transfer control to X,1.	

Notes:

a) The order I,3 moves the instruction word (I,1 and I,2) to R1. The left order, I,1, whose preliminary symbol is +[A1], contains the address A1. The order I,4 stores this whole instruction word at T1. This instruction word remains also at its previous address because, as we know, moving a word from a memory position does not erase its contents.

b) The orders III,3 and III,4 perform similar operations on the instruction word (III,1 and III,2), which contains the address Ai.

c) The erasure Box IX was explained sufficiently before.

The question, which alternative is more advantageous, should be answered by the student in Exercise 3 at the end of this Chapter.

2., 3., 4. <u>Coding, Assigning Addresses, Punching and Transcribing.</u> These are left to the student as an exercise.

Summary. A basic part of nearly every flow chart is an "induction loop" in which a recursive routine is followed. A general set-up of a loop can be shown in a diagram as in Figure 8.

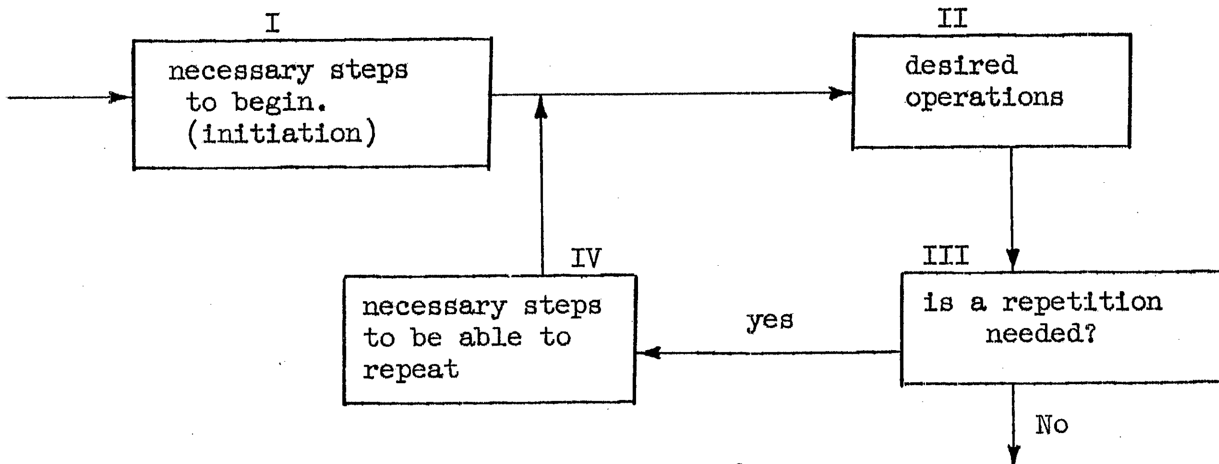


Figure 8.

Exercises:

1. The numbers 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.9, are stored at the following addresses:

0.3 at 010
0.8 at 011
0.0 at 012
0.6 at 013
0.9 at 014
0.1 at 015
0.5 at 016
0.2 at 017
0.4 at 018
0.7 at 019

Note: The student will store these numbers himself.

Prepare for machine computation the sorting of the above numbers and put the problem on the machine (print them out).

2. Assume that ORDVAC completes one operation of addition, subtraction storing, moving from the memory, transferring control, or comparing in 0.1 millisecond = 0.0001 of a second.
Multiplication or division in 1 millisecond = 0.001 of a second.
Printing one word (teletype) in 2 seconds.
Reading one binary IBM card in 1 second.

Find the time it will take ORDVAC to sort the numbers in Exercise 1.

3. Find the time it will take ORDVAC to sort $a_1, a_2, \ldots, a_{100}$ using each alternative. Deduce which alternative is more advantageous.

4. Assume that one hour of computations on ORDVAC costs $100.00. Find the cost of sorting one hundred numbers by the machine using each alternative.

CHAPTER VII

SUBROUTINES

A "routine" is a sequence of words, in machine language, designed to achieve some objective. A "subroutine" can be considered as a routine which is subordinate to a larger routine, subordinate in the sense that the objective of the subroutine is an integral part of the broad objective of the routine. Objectives of subroutines are usually of a general nature in that the objective is applicable to a wide class of problems. In most cases, the objective of a given subroutine is the computation of a given mathematical function such as $\sqrt{x}$, sin x, etc. We wish to emphasize that the following discussion on routines and subroutines is concerned with sequences of words which are designed for specific purposes, and which are "frequently used" or "often repeated". The distinction between routines and subroutines is made purely for descriptive purposes. The paramount purpose of routines and subroutines is to have available sequences of words which have been tested and can be applied in many situations with a minimum of effort.

Consider a routine whose objective is:

a. to compute $f(x) = x + \sqrt{x} + \sin x$, $0 \leq x \leq c$, $\Delta x = c/100$, and

b. to print x and f(x) in "decimal" form for each argument, x, in the specified range.

In order to obtain a routine that can accomplish this broad objective, it is necessary to design sub-sequences of words (in machine language) with the following particular objectives:

Sub-sequence 1, whose objective is to generate successive arguments, x;
" " 2, " " " " compute $u = \sqrt{x}$;
" " 3, " " " " compute $v = \sin x$;
" " 4, " " " " form the sum $y = x + u + v$;
" " 5, " " " " convert x and y to decimal form;
" " 6, " " " " print x and y in decimal form;
" " 7, " " " " determine when the objective is attained and to direct control according to existing plans.

We consider the combined sub-sequences as a routine. Sub-sequences "2", "3", and "5" are regarded as subroutines since their objectives are integral parts of the broad objective and general enough to be applicable to many other problems. Sub-sequences "2" and "3" are examples of computation subroutines whereas sub-sequence "5" is an example of a service subroutine. We classify sub-sequences "1", "4", "6", and "7" as special sequences because their respective objectives are either unique to this problem or can be constructed with a few words.

Subroutines, such as those for computing the values sin x, $\sqrt{x}$, etc, whose objectives are the computation of a function of a single variable, are coded assuming the following standard conditions:

I. When control is directed to the first order of the subroutines,

a) the specified argument is stored in R1;

b) the "return address" (abbreviated "R.A.") is stored as the right* address in R2; (the initial orders of the subroutine will move it from R2).

II. When control is directed to the "R.A." (return address) the result, (objective), is stored in R1.

The "return address" designates to what position control is to be directed after the subroutine has achieved its objective. When the objective requires more than one argument or produces multiple results, deviations from the standard conditions must be specified. For example, if the objective of a given subroutine produces two results, (such as sin x and cos x), then condition "II" above shows the deviation by indicating that the two results are stored in two specified positions.

To illustrate the design and characteristics of subroutines, we shall code an example of a subroutine whose objective is to compute, for a given "n", the sum of "n" numbers, a_i. For purposes of simplicity we shall assume that the following conditions have been satisfied when control is directed to the first order of the subroutine;

---

* In view of anticipated changes in control circuitry, this condition will be replaced by a condition which fixes the return address as the next in sequence with respect to the address of the order which directed control to the subroutine.

1. the numbers, a_i, are respectively stored in consecutive memory positions, Ai, i.e.,

   a_1 is stored in A1,
   a_2 is stored in A2,
   ⋮ ⋮
   a_n is stored in An;

2. each a_i and each partial sum, $S_i = a_1 + a_2 + a_3 + \ldots + a_i$ $(i=1,2,3,\ldots, n)$ is less than one in absolute value;
3. "n", the number of numbers to be summed, is recorded in R1, i.e. $2^{-39}(n)$ is in R1;
4. A1, the address of the first number, and R. A. (the return address) are specified in R2 as follows:

   2^{-19} A1 + 2^{-39} R.A.

   The above word is not only determined by the address of the first number, A1, and the return address, R.A., but in turn identifies A1 and R.A. separately. The scaling of these quantities as shown is merely to indicate their respective positions within the word.

The reader should observe that this subroutine can be considered as a function of two "variables", namely A1 and "n". In view of this fact we must deviate from standard conditions and require that these two "variables" be specified in some manner. Conditions 3. and 4. above is one means of specifying these "variables". Note that we have adhered to the standard condition that the R.A. be specified in the right address of R2.

The flow chart for this subroutine is shown in Figure 1.

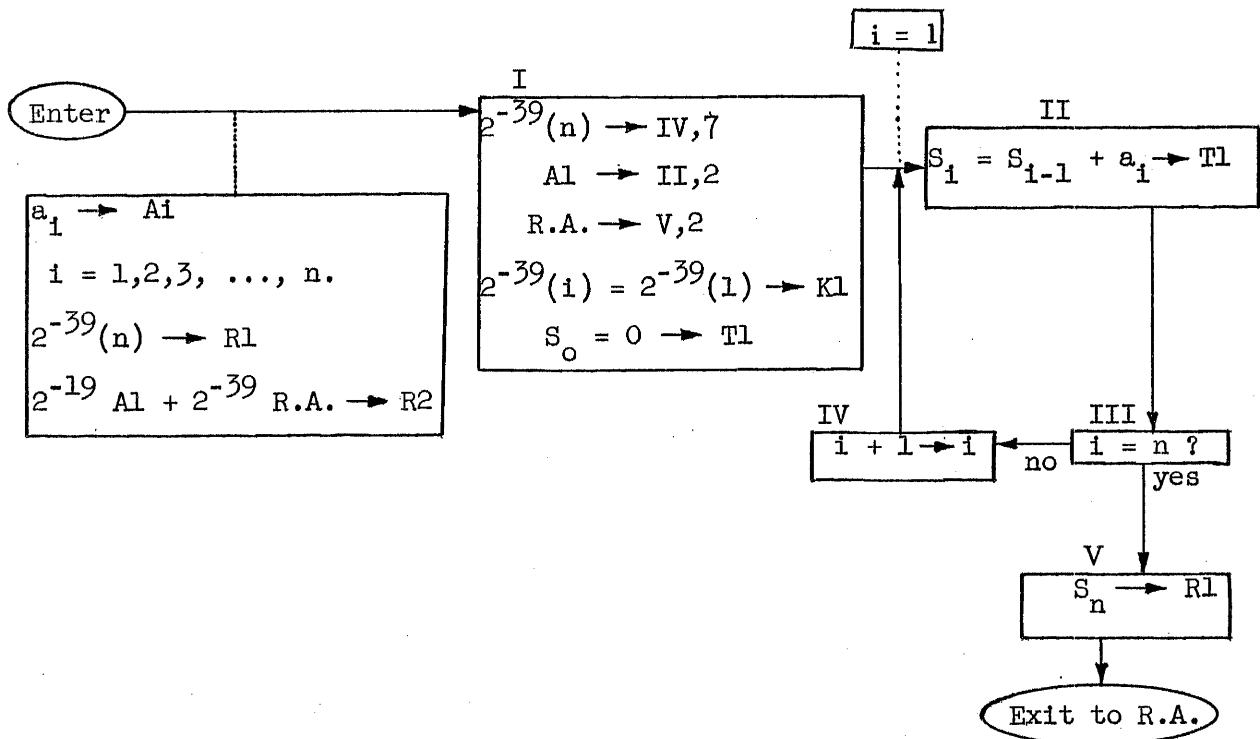


FIGURE 1

Notes on the flow chart:

The Box without a number states the standard conditions.

I. Box I is characterized as a set-up box. The orders corresponding to this box set (insert) addresses, parameters, and initial values which are necessary for the proper execution of the orders corresponding to the succeeding boxes. Control is then directed to the first order of Box II to begin the repetitive sequence. The parameter "n" is stored in IV,7 because, as the coding will show, this is a convenient place to store it.

II. The orders corresponding to Box II compute the ith partial sum, S_i, by adding the ith number a_i, to the previous sum, S_{i-1}. Control is then directed to the first order of Box III.

III. The orders corresponding to Box III are: first, to determine if the existing sum represents the desired final sum; next, control is directed accordingly. That is, if the existing sum represents the desired sum, ($i = n$), then control is directed to the first order of Box V. If the existing sum does not represent the desired sum, ($i < n$), then control is directed to the first order of Box IV.

IV. The orders corresponding to Box IV (substitution box) are: first, to generate the address of the "next" number to be added to the existing sum, (i.e., when the orders of Box II are being repeated); next, to record the number of terms, (i), that will have been summed when the comparison in Box III is made. Control is then directed to the first order of Box II to repeat the sequence.

V. The orders corresponding to Box V are first to store the desired sum in R1, and then to direct control to the specified return address.

The complete code for this example, shown in Figure 2 has arbitrarily been assigned memory positions 100 through 10F. This subroutine can be tested, recorded* on cards, and made available to anyone who desires to use it. All that is required of the user is that the four specified conditions be satisfied, and (5.), that the subroutine be stored in memory positions 100 through 10F. Condition (5.) does not appear too restrictive, especially since the user has already gained two advantages:

a. the amount of coding necessary to achieve his objective is decreased since it only requires a few words to satisfy the requirements of this subroutine.

b. since the amount of coding has been decreased, and since the subroutine has been tested, there is less chance for mistakes.

If a routine requires the use of two or more subroutines which have been coded for a common area of the memory, then it becomes necessary to move all except one of the subroutines from the common area to new areas of the memory. Clearly, no two subroutines can occupy the same memory positions at any given time. If the moving of subroutines to new areas is to be done manually by recoding each subroutine for each new area, then most of the advantages gained will have been sacrificed. Even though recoding a subroutine for a new area is just a matter of reassignment of addresses, manual recoding for new areas is impractical since there are too many possible areas. Hence, condition (5) is too restrictive. To

---

* Words can be recorded on many media such as coding sheets, paper tape, magnetic tape, paper cards, magnetic drums, etc.

FIGURE 2

ORDVAC Problem________

Sheet______

Seq.	Code	Word	Order	R_1		R_2		Mem.	Contents		Ord.	Contents		Description
					$2^{-39}(n)$	$2^{-19}A1$	$2^{-39}R.A.$							
I, 1	5010K	100	E' IV, 7								IV, 7	U II, 1	$2^{-39}(n)$	sets n
2	KN101		A + I, 3	$2^{-19}A1$	$2^{-39}R.A.$									
3	5010S	101	E' V, 2								V, 2	+ T1	U [R. A.]	sets R. A.
4	08014		→ 20		$2^{-19}A1$	$2^{-19}R.A.$	$2^{-39}A1$							
5	50104	102	E' II, 2								II, 2	+ T1	(+) [A1]	sets A1
6	K410J		+ K2		2^{-39}									
7	5010N	103	E' K1					K1	U II, 1	$2^{-39}(1)$				sets i = 1
8	3010F		oM T1					T1		0				Sets $S_o = 0$
II, 1	K410F	104	+ T1	S_{i-1}										
2	N4000		(+) [A_i]	S_i										
3	1010F	105	M T1					T1	S_i					
III, 1	K410N		+K1	U II, 1	$2^{-39}(i)$									
2	0410K	106	(-) IV, 7	$2^{-39}(i - n)$										
3	2010S		C V, 1											i = n?
IV, 1	K410J	107	+ K2		$2^{-39}(1)$									
2	N4104		(+) II, 2	+ T1	(+) A_i+1									
3	10104	108	M II, 2								II, 2	+ T1	(+) [A_{i+1}]	
4	K410J		+ K2		2^{-39}									i + 1 → i
5	N410N	109	(+) K1	U II, 1	$2^{-39}(i+1)$									
6	1010N		M K1					K1	U II, 1	$2^{-39}(i+1)$				
7	N0104	10K	U II, 1											
-	00000		[$2^{-39}(n)$]											
V, 1	K410F	10S	+ T1	S_n										S_n → R1
2	N0000		U [R. A.]											
K1	N0104	10N	U II, 1											
	00000		[$2^{-39}(i)$]											
K2	00000	10J	Constant											
	00001		$2^{-39}(1)$											
T1	00000	10F	~~Temporary~~											
	00000													

eliminate this restriction, we code subroutines using "relative addresses" so that recoding any subroutine for any given area of the memory can be automatically effected by an Input Routine. An Input Routine will be discussed in the next chapter.

A "relative address" is an address in or of an order, whose sexadecimal representation is a function of (related to) the area of the memory in which the order is stored. Addresses of instruction words are invariably relative.

In the example of the subroutine we coded for memory positions 100 through 10F, the sexadecimal representation of K1 was 10N. If the subroutine were coded for memory positions 200 through 20F, the sexadecimal representation of K1 would be 20N. Similarly, the address of orders II,1 and II,2 was 104, but if the subroutine were coded for memory positions beginning at 300, the address of orders II,1 and II,2 would be 304. Thus, the address assigned to K1 and the address assigned to the order pair, II,1 and II,2 are examples of relative addresses since the addresses assigned are dependent upon the area of the memory in which the subroutine is stored. Figure 3 shows the final code for the subroutine as coded for two different areas of the memory, one beginning at 100, the other beginning at 200.

	Beginning at 100.	SEQ.	Beginning at 200.	
100	5010K	I, 1	5020K	200
	KN101	2	KN201	
101	5010S	3	5020S	201
	08014	4	08014	
102	50104	5	50204	202
	K410J	6	K420J	
103	5010N	7	5020N	203
	3010F	8	3020F	
104	K410F	II, 1	K420F	204
	N4000	2	N4000	
105	1010F	3	1020F	205
	K4109	III, 1	K4209	
106	0410K	2	0420K	206
	2410S	3	2420S	
107	K410J	IV, 1	K420J	207
	N4104	2	N4204	
108	10104	3	10204	208
	K410J	4	K420J	
109	N410N	5	N420N	209
	1010N	6	1020N	
10K	N0104	7	N0204	20K
	00000	-	00000	
10S	K410F	V, 1	K420F	20S
	N0000	2	N0000	
10N	N0104	K1	N0204	20N
	00000		00000	
10J	00000	K2	00000	20J
	00001		00001	
10F	00000	T1	00000	20F
	00000		00000	

FIGURE 3

Observe that not all of the addresses are dependent upon the area of the memory in which the subroutine is stored.

To reduce the number of storage positions that are required when two or more subroutines are being used by a given routine, a specific area of the memory has been reserved for "temporary" positions and "frequently" used constants. These temporary positions and constants are used by all subroutines or routines as required.

If a subroutine has to be repeated its instruction words must be saved and remain stored in the same area of the memory after each execution of a subroutine. Memory positions which do not contain instruction words

and are used by a subroutine may have to be saved or may not. Temporary memory positions are those positions which are used only when the orders of a subroutine are executed and are not needed afterwards. In the example, T1 is such a position since its contents are of no need after the subroutine is executed. The proper execution of the subroutine is independent of the contents of T1 when control is directed to the first order of the subroutine. The position T1 can be used temporarily by any other subroutine. The address T1 (not the contents) is fixed because it is assigned in the reserved area of the memory and the address remains fixed regardless of the area of the memory in which a subroutine using T1 is stored.

Hence, we call T1 a "fixed" address, as distinguished from a "relative" address. The area reserved for temporary positions is 000 through 009 and 010 through 03L.

Some frequently used constants are: 0; 2^{-1}; 2^{-19}; etc. The following is a list of constants with the corresponding "fixed" memory addresses wherein they are stored.

zero,	0000000000, in 00K
dummy key word,	8000000000, in 00S
$2^{-19} + 2^{-39}$,	0000100001, in 00N
2^{-39},	0000000001, in 00J
2^{-1},	4000000000, in 00F
$1 - 2^{-39}$,	7LLLLLLLLL, in 00L.

The Input Routine stores these constants in the designated positions. They are available for use by all subroutines and routines. The addresses of these constants are included in the class of fixed addresses.

In shift orders, instead of an address we write the amount of shift (see shift orders, Chapter IV). The amount of shift in shift orders is independent of the area in the memory in which the order is stored. Consequently, the amount of shift in shift orders must be regarded as a fixed address.

To summarize, the addresses in and of the orders of subroutines can be divided into two classes:

a. relative addresses

b. fixed addresses.

This concept of relative and fixed addresses is not restricted to subroutines alone, it can be applied to routines as well.

To facilitate the use of subroutines, each of which can be automatically read in and recoded (by the Input Routine) for any area of the memory, the following procedure is employed:

I. All subroutines are coded for the area of the memory beginning at 000 and extending in sequence through as many positions as are necessary. Temporary positions are assigned fixed addresses in the reserved area, and the constants which are stored in the "fixed" positions are used when applicable.

II. Instead of representing each order by five sexadecimal characters, each order is represented by six sexadecimal characters as follows:

$$X_1X_2X_3X_4X_5X_6,$$

X_1X_2 are the two sexadecimal characters representing the order type;

$X_3X_4X_5X_6$ represent a four sexadecimal character address called "a pseudo address";

$X_4X_5X_6$ represents the sexadecimal address corresponding to the area of the memory beginning at 000;

X_3 indicates whether $X_4X_5X_6$ is relative or fixed;

X_3 = 0 implies that $X_4X_5X_6$ is fixed;

X_3 = 4 implies that $X_4X_5X_6$ is relative.

Hence, the range of relative addresses is 4000 through 4LLL.

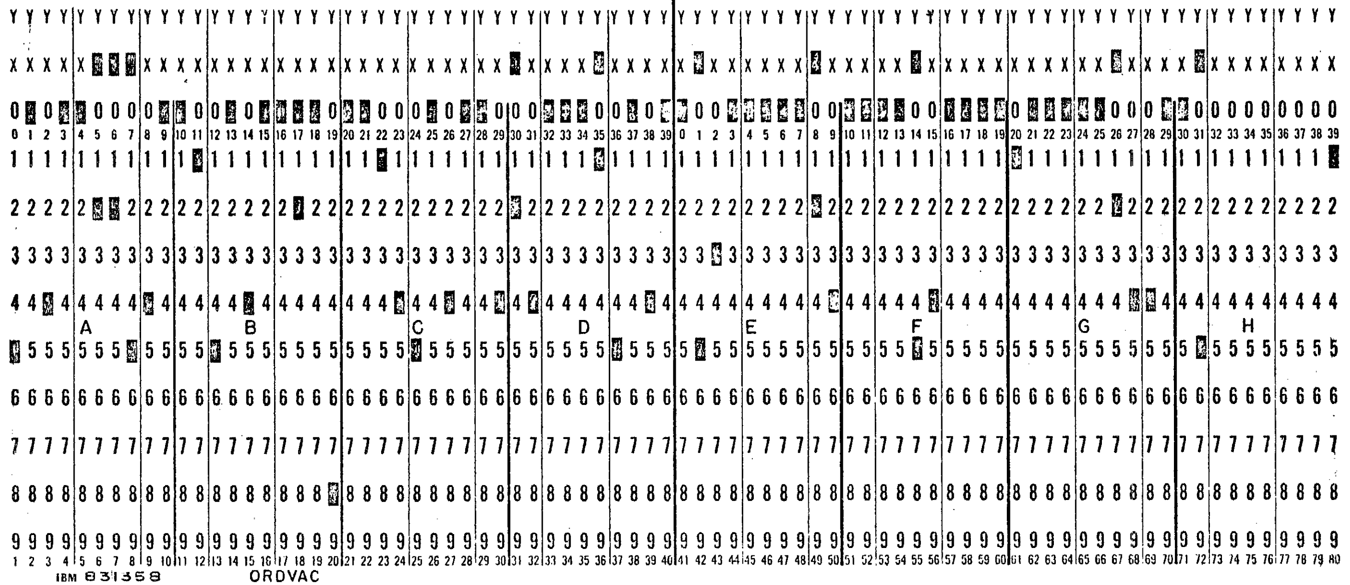
Thus, what was a ten sexadecimal character word is now represented by a twelve sexadecimal character word. For example, instead of writing the two orders I,3 and I,4 as 5000S 08014, we write these orders as 50400S 080014, where the "4" in the left order indicates that address "00S" is relative and the "0" following the "8" in the right order indicates that "014" is fixed. (The subroutine example has been recoded to conform to the above procedure. The complete coding is shown in Figure 4.)

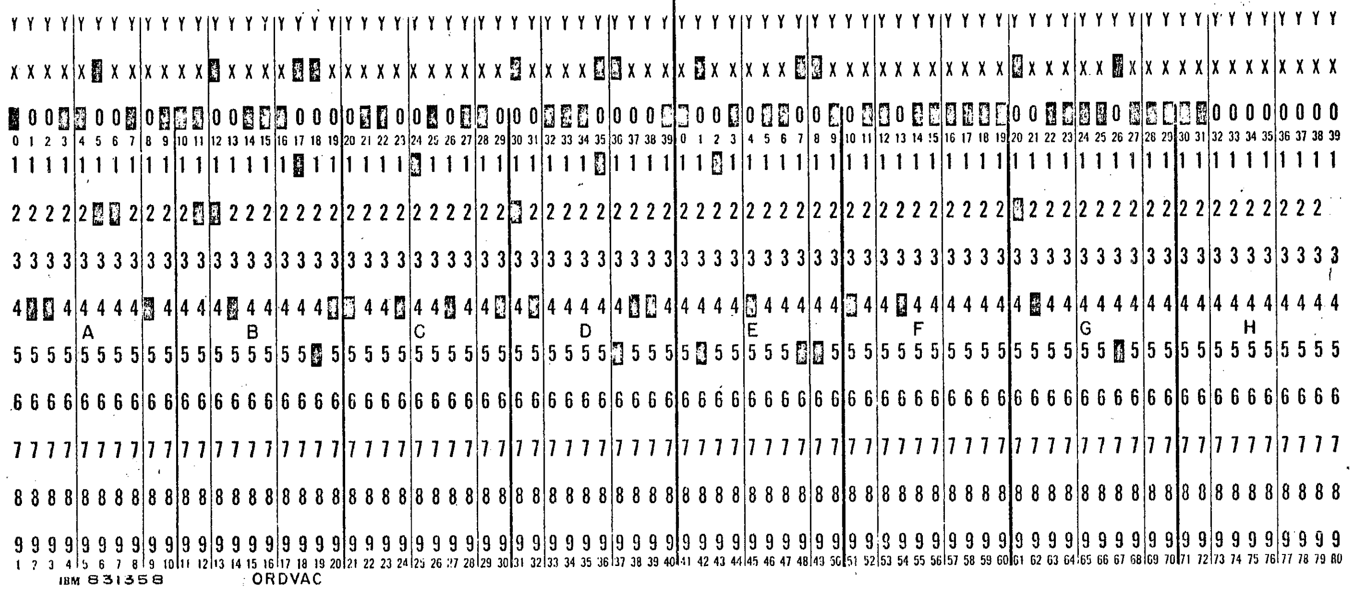
FIGURE 4.

ORDVAC Problem__________

Sheet_____

Seq.	Code	Word	Order	R_1		R_2		Mem.	Contents		Ord.	Contents		Description
					$2^{-39}(n)$	$2^{-19}A1$	$2^{-39}R.A.$							
I, 1	50400K	4000	E' IV, 7								IV,7	U II, 1	$2^{-39}(n)$	sets n
2	KN4001		A+ I, 3	$2^{-19}A1$	$2^{-39}R.A.$									
3	50400S	4001	E' V, 2								V, 2	+ T1	U [R.A.]	sets R.A.
4	080014		→ 20		$2^{-19}A1$	$2^{-19}R.A.$	$2^{-39}A1$							
5	504004	4002	E' II, 2								II, 2	+ T1	(+)[A1]	sets A1
6	K4000J		+ K2		2^{-39}									
7	50400N	4003	E' K1					K1	U II, 1	$2^{-39}(1)$				sets i = 1
8	300000		bM T1					T1		0				sets $S_o = 0$
II, 1	K40000	4004	+ T1	S_{i-1}										
2	N40000		(+) [A_i]	S_i										
3	100000	4005	M T1					T1	S_i					
III, 1	K4400N		+ K1	U II, 1	$2^{-39}(i)$									
2	04400K	4006	(-) IV, 7	$2^{-39}(i - n)$										
3	20400S		C V, 1											i = n?
IV, 1	K4000J	4007	+ K2		$2^{-39}(1)$									
2	N44004		(+) II, 2	+ T1	$(+)A_i+1$									
3	104004	4008	M II, 2								II, 2	+ T1	(+)[A_{i+1}]	
4	K4000J		+ K2		2^{-39}									i + 1 → i
5	N4400N	4009	(+) K1	U II, 1	$2^{-39}(i+1)$									
6	10400N		M K1					K1	U II, 1	$2^{-39}(i+1)$				
7	N04004	400K	U II, 1											
-	000000		[$2^{-39}(n)$]											
V, 1	K40000	400S	+ T1	S_n										S_n → R1
2	N00000		U [R.A.]											
K1	N04004	400N	U II, 1											
	000000		[$2^{-39}(i)$]											
T1		0000	Temporary											
K2		000J	Constant $2^{-39}(1)$											





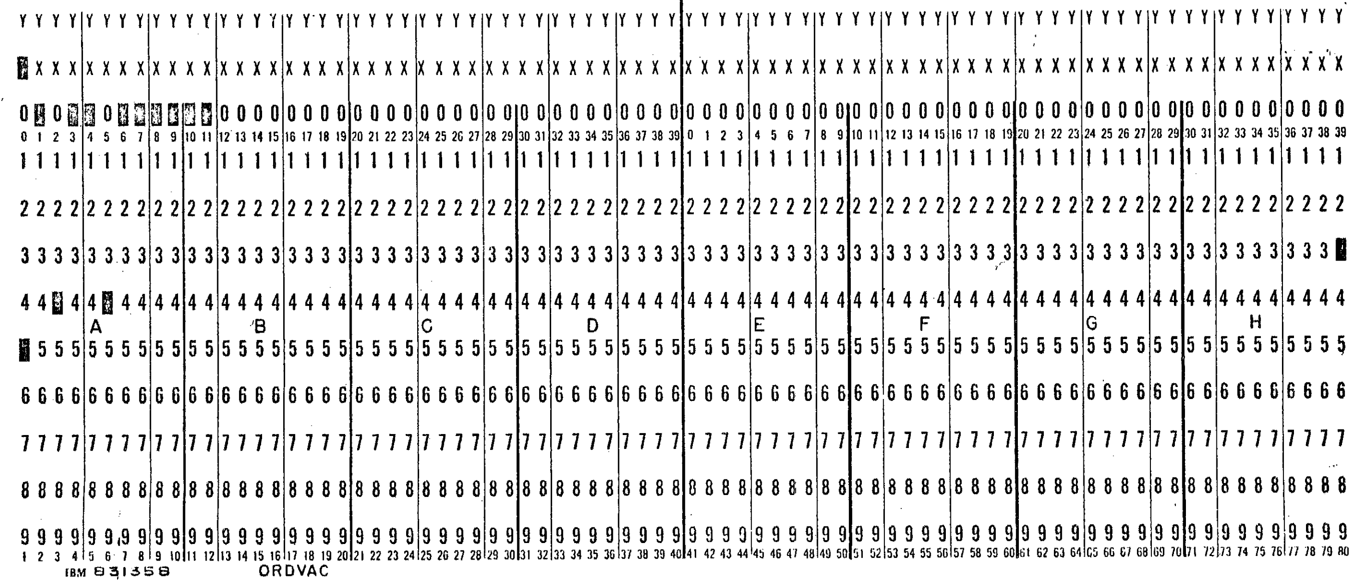


Figure 5

III. The set of words corresponding to this subroutine are recorded on cards using the pseudo address representation. The cards, shown in Figure 5, will be discussed in the next chapter.

The above procedure permits the Input Routine to recode any given subroutine for any "new" area of the memory by simply adding to each relative address the "new" address (specified by a key-word) of the first order of the subroutine.

For reference purposes, the following items are recorded and filed for each subroutine:

(1) A statement of the objective of the subroutine.

(2) A list of the equations or method employed;

(3) The list of conditions to be satisfied;

(4) a. the number of storage positions that the subroutine requires;
   b. the amount of machine time required to achieve the objective;
   c. the degree of accuracy to which the objective is attained, when applicable;

(5) A clear statement of special and/or unusual conditions;

(6) A flow chart;

(7) The detailed coding including a list of:
   a. numerical constants used and corresponding addresses of these constants;
   b. the addresses of temporary positions that are used.

(8) A test to insure actual machine operation.

(9) Two decks of cards, one is pseudo address form, (figure 5) the other in binary form. The binary form will be discussed in the next chapter.

Illustrations of items (1.), (2), (3), and (6) were given in the example. With regard to item (4),

a. thirteen memory positions are required; this number does not include the number of temporary positions or constants from the fixed area that are used;

b. the approximate machine time required to achieve the objective is computed by dividing the approximate number of orders to

be executed by the rate at which orders are executed. To determine the number of orders to be executed, an inspection of the flow chart indicates that:

1. the orders corresponding to Box I will be executed once;
2. the orders corresponding to Boxes II and III will be executed n times;
3. the orders corresponding to Box IV will be executed n-1 times;
4. the orders corresponding to Box V will be executed once.

The number of orders corresponding to each box is:

Box I. .... 8 orders,
Box II. .... 3 orders,
Box III. .... 3 orders,
Box IV. .... 7 orders,
Box V. .... 2 orders.

Therefore, the total number of orders that are necessary to achieve the objective is

$1(8) + n(3) + n(3) + (n-1)(7) + 1(2) = 13n + 3$ orders. Since the rate at which orders are executed is approximately 10,000 per second, the approximate machine time required is

$$\frac{13n + 3}{10,000} \text{ seconds, which is approximately } 1.3n \text{ milleseconds.}$$

Not all orders are executed at the 10,000 per sec. rate; orders such as multiplication and division are executed at approximately one tenth of this rate. Consequently, in computing the number of orders corresponding to any given box, order such as multiplication and division are weighted accordingly. A table of time estimates for various classes or orders is given in the Appendix (see page 246).

c. The degree of accuracy to which an objective (function) is obtained depends upon:

1. the numerical method that is used,
2. the error inherent in an argument (or arguments),
3. the amount of "round off" error generated in the process.[1]

---

1. BRL Report 816, "On the Study of Computational Errors." S. Gorn

Generally, the numerical methods used are approximating methods. For example, one can approximate the sin x by the well known infinite series expansion. Since only a finite number of the infinite number of terms are used, the method yields an approximation to this function. In order to compute the general term of such a series, multiplication and/or division are necessary. The machine operations of multiplication and division are not exact operations since a finite number of bits are used to represent a product or quotient. The machine operations "round-off" the products and quotients such that each product can be in error by as much as a half unit in the least significant bit, and each quotient can be in error by as much as a whole unit in the least significant bit. Hence, the "round-off error" is a function of the number of the multiplications and divisions that the numerical method necessitates. In many cases the computation of the maximum value of the total error (the combined 1., 2., and 3. above) is more complex than the numerical method itself. In such cases, experimental, probabilistic or statistical bounds for the errors are given. Estimates of the errors are beyond the scope of this text; it suffices to say that they do exist and error studies are being pursued extensively since the advent of high-speed digital computers. In special cases, special routines can be designed to compute approximate errors[2]. In our example an exact method was used involving exact operations; hence, if all ai are exact, the sum is exact.

The ideal subroutine would be that one which necessitates a minimum number of storage positions, a minimum amount of machine time, and maximizes the accuracy. Unfortunately, these ideal characteristics are incompatible in that one can increase accuracy by increasing time and the number of storage positions; similarly, one can decrease time by appropriate sacrifices in storage and accuracy. In some cases, it is practical to code a given subroutine to idealize one of the characteristics. For example, Figure 6 shows an alternative flow chart and code of a subroutine for obtaining the sum of n numbers. This code uses only nine words as compared to thirteen in the previous example (see page 137).

---

2. BRL Report 893, "Automatic Error Control", S. Gorn, R. Moore.

With regard to item (5.), it should be stated that for our example, the subroutine does not include a check to verify that the assumptions have been fulfilled. For example, if the absolute value of any partial sum exceeds unity, the subroutine does not detect this fact and hence will yield an erroneous result.

With regard to item (7.),

a. one constant was used, namely, $2^{-39}(1)$ which is stored in 00J.

b. one temporary position was used, namely T1 = 000.

Illustrations of items (8.) and (9.) will be given in the next chapter.

Exercise. Code a subroutine whose objective is to compute the sum,

$$S_n = p_1q_1 + p_2q_2 + p_3q_3 + \ldots. + p_nq_n$$

Assume that:

a. each p,q, and partial sum is less than one in absolute value;

b. p's and q's are stored respectively in binary form in consecutive positions, Pi and Qi, i.e.

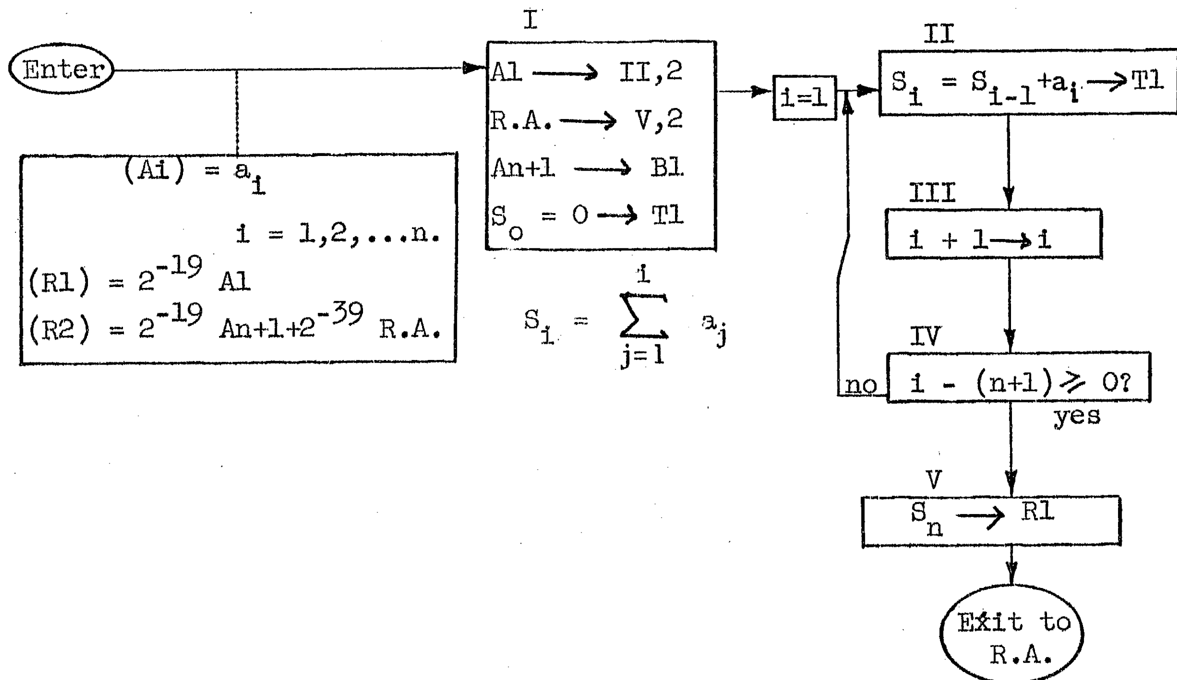
p_1 is in P1, q_1 is in Q1;

p_2 is in P2, q_2 is in Q2;

' ...... ' , ' ...... '

' ...... ' , ' ...... '

p_n is in Pn, q_n is in Qn.



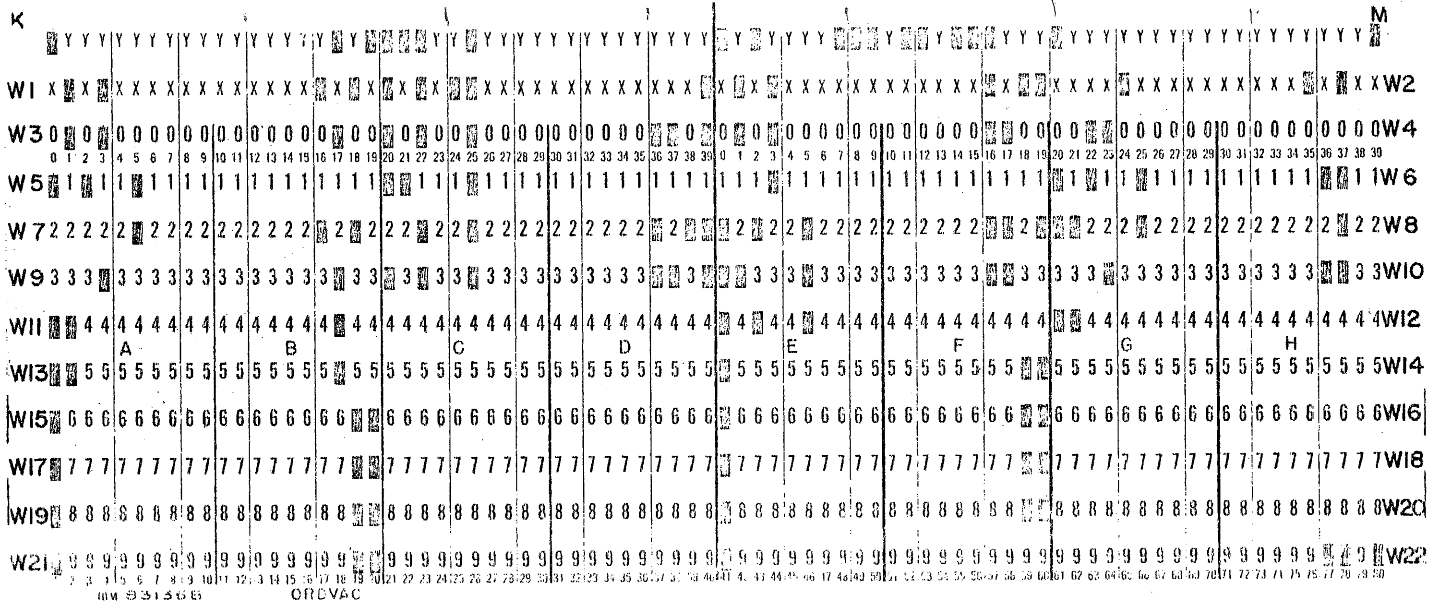
Seq.	Order Code	Address	Prelim. Order	Description
I, 1		4000	E II,2	A1 ⟶ II,2
2			A + I,3	2^{-19} An+1 + 2^{-39} R. A. ⟶ R1
3		4001	E' V,2	R.A. ⟶ V,2
4			E B1	An+1 ⟶ B1
5		4002	oM T1	S_o = 0 ⟶ T1
II,1			+ T1	S_{i-1} ⟶ R1
2		4003	(+) [Ai]	S_i ⟶ R1
3			M T1	S_i ⟶ T1
III,1		4004	+ B2	$2^{-19}(1) + 2^{-39}(1)$ ⟶ R1
2			(+) II,2	Ai+1 ⟶ R1 i + 1 ⟶ i
3		4005	E II,2	Ai+1 ⟶ II,2
IV, 1			(-) B1	(i) - (n+1) ⟶ R1
2		4006	C V, 1	(i) - (n+1) = 0 ?
3			U' II,1	transfer back to II,1
V, 1		4007	+ T1	S_n ⟶ R1
2			U [R.A.]	transfer to R.A.
B1		4008	(+) An+1 M T1	this word is used for comparison only
B2		000N	$2^{-19}(1) + 2^{-39}(1)$	This is a constant in the reserved area
T1		0000	S_i	this is a temporary position in the reserved area.

FIGURE 6
Alternate Flow Chart and Code of a subroutine to obtain the sum of "n" numbers.

CHAPTER VIII

Transcriber Routine and Input Routine

The nature of the Ordvac card input operation is such that words represented on cards in a form convenient for a coder are not of the form convenient for the card input operation. Words represented in a form convenient for the card input operation are said to be in "binary form". In this form of representation twenty-four words can be represented on a standard IBM card. The arrangement on the card of the twenty-four words in binary form is shown in Figure 1. We shall repeat the explanation of the word forms and their arrangement on IBM cards, which were given in Chapter III.



Binary Card Form

FIGURE 1

This card has twelve horizontal rows each labeled respectively across the entire row as "Y", "X", "0", "1", "2", ..., "9", and eighty vertical columns each labeled respectively at the bottom of the card as "1", "2", "3", ..., "80". Since there are eighty columns associated with any given row, eighty bits can be recorded in any given row. The small rectangular area defined by the intersection of a given row and column is called a "punching position". Punching positions 1 through 40

of any given row represent respectively the forty bits associated with one word, punching positions 41 through 80 represent correspondingly the forty bits associated with another word. A perforation (punch) in a punching position represents a binary "1", a non-punch represents a binary "0". Hence, in any given row we represent two words. Since the card input operation "reads" one complete row at a time, the ordering of the words alternates from the left (columns 1 through 40) to the right (columns 41 through 80) words of the card. For descriptive purposes we have labeled (in the extreme left and right margins) the words of the card in Figure 1. as W1, W2, W3, ..., W22. For Example, the word labeled W4 represents the forty bit word

0101000000 0000001100 0011000000 0000000000.

In sexadecimal form this word can be written as

5000N30000.

The manual preparation of words expressed in binary form is cumbersome; similarly, manually operated equipment is not adapted for recording words expressed in binary form. Hence the necessity for the two forms of representation:

1. Binary form for machine convenience
2. Sexadecimal (or decimal) form for coders' convenience.

Decimal form is here grouped with the sexadecimal form; the distinction between these two forms will be discussed in the next section.

The arrangement of words on a card in sexadecimal form is shown in Figure 2. Since a pseudo address is used to distinguish between relative and fixed addresses, words in sexadecimal form are represented by twelve sexadecimal characters as was explained in the preceding chapter. We use twelve successive columns to represent the twelve sexadecimal characters of a given word. The maximum number of words per card in this form is six. Columns 73 through 80 are unused. Sexadecimal characters less than "K" are represented by single punches in the corresponding rows, i.e. a sexadecimal "1" is represented by a punch in row "1" of a given column, a sexadecimal "2" is represented by a punch in the "2" row of a given column, etc. Sexadecimal characters greater than "9" are represented by double punches in the same column, that is, "K" is represented by punches

W1 W2 W3 W4 W5 W6

A B C D E F G H

IBM 831358 ORDVAC

W7 W8 W9 W10 W11 W12

A B C D E F G H

IBM 831358 ORDVAC

W13

A B C D E F G H

IBM 831358 ORDVAC

Fig. 2

in the "X" and "2" rows of a column, "S" is represented by punches in the "0" and "2" rows of a column, "N" is represented by punches in the "X" and "5" rows of a column, "J" is represented by punches in the "X" and "1" rows of a column, "F" is represented by punches in the "Y" and "6" rows of a column, "L" is represented by punches in the "X" and "3" rows of a column. (In the process of manually preparing cards, each double punch can be produced by pressing a single key of an alphabetical keyboard.) Again for descriptive purposes we have labeled (in the upper margin) the words of the cards of Figure 2 as W1, W2, W3, etc. For example, the word labeled W4, (columns 37 through 48), represents the twelve sexadecimal character word

50400N 300000.

The cards shown in Figure 2 represents the words of the subroutine example discussed in the preceding chapter.

Although the Transcriber Routine and Input Routine are two distinct routines, they are related and will be discussed as to their particular objectives and their relation to each other. Hereafter we will refer to the Transcriber Routine as the "Transcriber". The purpose of these routines is to "read into" the machine other routines or data represented on cards in a form convenient for coders and to store the given routine or data in any designated area of the memory in a form ready for execution or processing.

The objective of the Transcriber is to accept (read) words represented on cards in sexadecimal or decimal form and to produce cards with these same words represented in binary form. The binary cards are produced one after another during the action of the transcriber. After all of the binary cards have been produced and are being fed into the machine (in a manner that is described later) they are accepted by the Input Routine. Thus, the Transcriber is literally a "translator" in that it translates words represented in sexadecimal and decimal form to words represented in binary form.

To facilitate the use of subroutines, a pseudo address method was devised whereby the Transcriber could distinguish between relative and fixed addresses. Now in the same sense, it is desirable to devise some scheme whereby the Transcriber can distinguish between words represented in sexadecimal form and words represented in decimal form. This is desirable since it is convenient to represent datum numbers in decimal form. It is not only desirable to have the Transcriber recognized decimal datum numbers, but it is also convenient to have the Transcriber convert the decimal datum numbers to their binary equivalents. Other desirable facilities are:

a. the facility to store a word (or words) in a prescribed memory position (positions).

b. the facility to direct control to the first order of any given routine.

In order to indicate to the Transcriber which of many facilities is desired, special words, called "key words", (distinguishable from words to be stored in the memory), have been designed. Indeed a key word has been designed for each desired facility. One key word expresses one facility, another key word expresses another facility, etc. Key words are expressed in standard sexadecimal form, twelve characters per key word. An identifying characteristic of all key words is that the first character of each key word is always an "8" and the next three characters are always zero. The sixth and seventh characters of a key word identify a facility. When applicable, the last four characters of a key word represent a pseudo address associated with a facility.

Example 1. 800001 0002K0 is called a "10" type key word.
The facility associated with this key word is:
Direct control to the left order of memory position 2K0.

Example 2. 800000 0002K0 is called a "00" type key word.
The facility associated with this key word is:
Store the words following this key word in consecutive memory positions beginning at memory position 2K0, and modify each relative address of these words by amount 2K0.

In discussing the various key words and their corresponding facilities, we wish to emphasize that, in general, *the facilities are not carried out by the Transcriber*. Actually, the sexadecimal key words and the words associated with them are converted to their binary equivalents by the Transcriber. Then, at some later time when the deck of binary cards is being fed into the machine, the binary key words and the words associated with them are accepted by the Input Routine and the facility originally associated with the sexadecimal key word is carried out by the Input Routine. In this sense we merely wish to emphasize that even though we associate a facility with a Transcriber key word, we think in terms of a facility which will eventually be effected by the Input Routine. In essence, *the facilities are carried out by the Input Routine*.

Table of Transcriber Key Words and Associated Facilities

Type	Transcriber key word (Written in Sexadecimal form)	Facility (to be carried out by the Input Routine)
00	800000 00-$\overset{A}{\text{---}}$,	Store the following words in consecutive memory positions beginning at position "A", and modify each relative address of the words being stored by amount "A".
32	800003 20-$\overset{F}{\text{---}}$,	Store the following words in consecutive memory positions beginning at position "F" but continue to modify relative addresses by amount "A" of the previous "00" type key word
10	800001 00-$\overset{B}{\text{----}}$,	Direct control to the left order of memory position "B".
20	800002 000000,	Store the next word in the next consecutive memory position, and modify the relative addresses in the address portions of the word even though the word has the characteristics of a key word.
30	800003 00-$\overset{C}{\text{---}}$,	Convert the following decimal datum numbers to their binary equivalents and store the binary equivalents in consecutive memory positions beginning at position "C".

This is the one exception where all of the facility is not carried out by the Input Routine. The conversion is carried out by the Transcriber and the storing is carried out by the Input Routine.

As explained in Chapter III, decimal datum numbers are represented by a sign character and eleven decimal digits. The characters for the ten decimal digits are identical to the first ten sexadecimal characters, 0,1,2, ..., 9. A sexadecimal "K" represents a positive sign, a sexadecimal "S" represents a negative sign.

Example. +.32795480023 is represented as K32795480023,
-.32795480023 is represented as S32795480023.

This representation of decimal datum numbers is called standard decimal form.

When a coder plans the insertion of information into the machine, he decides in advance what facilities must be used. In listing the words of a routine to be inserted, the coder introduces key words, then the words to which the key word applies, then another key word, then the words to which that key word applies, etc.

The facility corresponding to the "10" type key word is effective immediately. The facility corresponding to the "00", "32", "30" types remains in effect until another key word is encountered. When a type "20" key word is recognized, its facility is effective only for the word immediately following the type "20" key word. Then, the facility previously in effect, (i.e. the facility in effect prior to the type "20" key word), again becomes effective until a new key word is encountered. Other available key words and their corresponding facilities will be given in chapters on the magnetic drum and floating point routines.

The objective of the Input Routine is:

a. to accept words represented on cards in binary form,

b. to store these words in designated memory positions, (or on the magnetic drum),

c. to modify relative addresses by prescribed amounts,

d. to direct control to a designated first order.

Hence, to store a given routine represented on cards in sexadecimal form, the Transcriber is used as an intermediate step to obtain the words in binary form convenient for the Input Routine. It was stated previously that the Transcriber could distinguish between relative and fixed addresses since the Transcriber had access to the pseudo addresses. However, the Input Routine modifies relative addresses, hence, the Input Routine must be able to distinguish between relative and fixed addresses. To indicate to the Input Routine whether addresses are relative or fixed, the Trancriber records a "modifier word" and a "modifier character" on each of the binary cards that it produces.

To illustrate the use of the "modifier word" and the "modifier character", the reader is referred to Figure 1, which shows a card produced by the Transcriber. The card shown in Figure 1 represents the subroutine example of the preceding chapter. The first word of the card (the left word of the "Y" row) is a binary key word. Expressed in sexadecimal form, this key word is

$$80005 \, \underline{i} \, \underline{\text{- - H -}}.$$

Normally, this "5" type key word is always the first word of every card produced by the Transcriber. The address, "H", of this word indicates to the Input Routine where the first word, W1, of this card is to be stored. The address "H" may be relative or fixed. The sexadecimal "i", the modifier character, indicates to the Input Routine whether the addresses in the words of the "X" row are relative or fixed. Since any row contains at most two words (at most four addresses), four bits (one sexadecimal character) are sufficient to indicate whether the four addresses of a row are relative or fixed. A binary "1" indicates that a corresponding address is relative, a binary "0" indicates that a corresponding address is fixed. The one to one correspondence between the four bits of "i" and the four addresses in the "X" row is:

reading from left to right,

the first bit of "i" is associated with the left address of W1
" second " " " " " " " right " " W1
" third " " " " " " " left " " W2
" fourth " " " " " " " right " " W2.

In Figure 1, "i" is 1110, which indicates that the only fixed address in the "X" row is the right address of W2. The three other addresses in the words in the "X" row are relative as indicated by the corresponding bits of "1". The "modifier word", (labeled "M" in the upper right margin), is always recorded in the right half of the "Y" row. The forty bits of this word reflect whether the forty addresses in the twenty words, W3 through W22, are relative or fixed. The one to one correspondence between the forty bits of the modifier word and the forty addresses in words W3 through W22 is as follows:

reading the bits of the modifier word from left to right,

the	first	bit	is	associated	with	the	left address	of W3,
"	second "	"	"	"	"	right	"	" W3,
"	third "	"	"	"	"	left	"	" W4,
"	fourth "	"	"	"	"	right	"	" W4,
	:							
	:							
the	fortieth	"	"	"	"	right	"	" W22.

The following is a list of key words associated with the Input Routine, with the corresponding facilities. In expressing these binary key words in sexadecimal form, (ten sexadecimal characters), the fifth sexadecimal character is used to identify the corresponding facility. These are the binary key words produced by the Transcriber. There is a similarity between the sexadecimal key words and the binary key words; however, there is no unique one to one correspondence between the two types.

Binary key words (expressed in 10 sexadecimal character form) Type		Facility
0	8000000-$\overset{A}{--}$-,	Same as the sexadecimal key word "00", store and modify.
1	8000100-$\overset{B}{--}$-,	Same as the sexadecimal key word "10", direct control to "B".

2	8000200000,	Same as the sexadecimal key word "20", store the next word in sequence even though it appears like a key word
3	8000300-C-,	Store the following words in consecutive memory positions beginning at position "C", but continue to modify relative addresses by amount "A" of the previous type "0" key word. Note that this facility differs from the facility associated with the "30" type key word.
5	80005i-H---,	Store the following words in consecutive positions beginning at memory position "H" and modify the addresses by amount "A" of the previous "0" type key word in accordance with the modifier word and the modifier character "i".

A type "0" or "1" key word always appears as the last word on a binary card.* That is, each time the Transcriber recognizes a "00" or "10" type key word, it produces a binary card with a "0" or "1" type key word as the last binary word of the card. If at the time of the recognition, the existing binary card being developed is not complete (full) the Transcriber will generate "dummy" key words to complete the card, except for the usual words in the "Y" row and the last word of the card. (Dummy key words have no effect; they are recognized by the Input Routine and immediately ignored.) This arrangement is made to allow a coder to make appropriate changes conveniently and to allow the storing of subroutines in sequence. One example in conjunction with the arrangement is: Suppose for some reason that one decides to change the location of a given subroutine after he has obtained the binary cards associated with the given subroutine. This implies the presence of a binary key word on a card preceding the subroutine, which indicated where, according to the former plan, the subroutine was to be stored. In accordance with the above arrangement, this particular key word will appear as the last word of a binary card. To effect a desired change, without changing any of the existing binary cards, one need only obtain another binary card reflecting the desired change. This card, containing the binary key

*However, the word W22 is not necessarily of either type; W22 usually is an order pair or a datum number just like other W's.

word reflecting the desired change, can be placed immediately behind the card containing the "old" key word. Since there will be no intervening words between the "old" and "new" key words, the facility associated with the "new" key word will supersede the facility associated with the "old" key word. Thus, the desired change will be effected. Other reasons for this arrangement of "0" or "1" type key words as the last words of a binary card, will become more apparent with experience.

The detailed description of the binary cards associated with the Input Routine is by no means simple; however, it should be borne in mind that the production of these cards is automatically carried out by the Transcriber. Futhermore, the detailed description of the binary cards, as to what they contain, and the arrangement of their contents, is given in the event one desires to analyze or check them for any reason whatsoever. Experience has shown that on many occasions coders refer to the binary cards for many reasons.

To illustrate the use of the Transcriber, the Input Routine, and the subroutine example of the preceding chapter, consider the following problem.

Example:

It is desired to compute and print the sum of fifty numbers. Each number and each partial sum is less than one in absolute value. The fifty numbers are recorded on IBM cards in standard decimal form.

Since a subroutine is available that can form this sum, we can immediately express a plan to achieve this objective in the form of a "rough" flow chart as follows

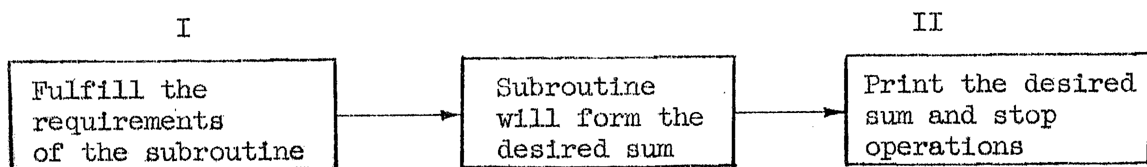


Figure 3

First, we will employ the facilities of the Transcriber and Input Routine to convert and store standard decimal numbers. In this respect (when control is directed to the first order of Box 1) we assume that the fifty numbers are stored respectively in memory positions A1 through A50. That is, we assume that the fifty numbers have been converted to their binary equivalents before they are stored in the designated memory positions. Next, directing our attention to Box I, a review of the requirements of the subroutine shows that before we direct control to the first order of the subroutine, we must provide instructions which insert a particular word in R1, and another particular word in R2. Here again, we assume that two particular words are stored respectively in memory positions B1, and B2. These words are:

$$(B1) = 2^{-39}n,$$

$$(B2) = 2^{-19}A1 + 2^{-39}R.A.$$

We can now append a box to the flow chart to indicate the conditions that will be assumed when control is directed to the first order of Box I

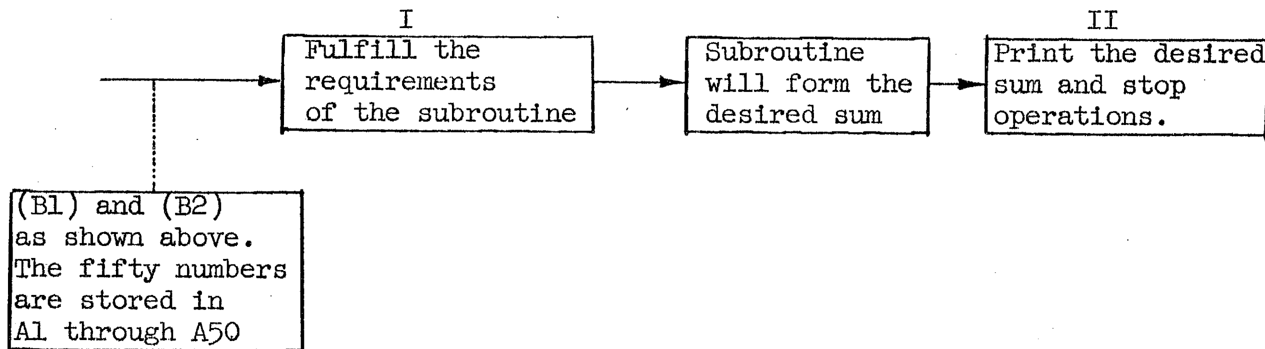


Figure 4

We can now write, in preliminary form, the orders of Box 1 to fulfill the requirements of the subroutine. These orders are as follows:

I, 1 + B1, (B1) ⟶ R1

, 2 R B2, (B2) ⟶ R2

, 3 U Sub., Control is directed to the first order of the subroutine. For this example, the "R.A." will be II,1. Since we know that the desired sum will be available in R1 when control is directed to the "R.A.", we can now write, in preliminary form, the orders of Box II to print the desired sum and stop operations. These orders are as follows:

II,1 M T1, the sum → T1

,2 R T1, the sum → R2

,3 P , Print the sum on the teleprinter

,4 Zx I,1, Stop operations; upon re-initiation control is directed to I,1 to repeat the process.

We now make the following sexadecimal storage assignments:

Let A1 be 100, then A50 is 131. Let T1 be 000. (The "100", "131", and "000" are sexadecimal addresses.)

Sequences I and II are consolidated with the following storage assignment

Seq.	Code	Word	Orders	Description
I,1	K40136		+B1	(B1) → R1
2	S40137	132	R B2	(B2) → R2
3	N00138	133	U Sub.	Control is directed to the first order of the subroutine.
-	000000		------	
II,1	100000	134	M T1	the sum → T1
2	S40000		R T1	the sum → R2
3	L40000	135	P	The sum is printed on the teleprinter
4	F00132		Zx I,1	Machine stops operations, repeats process if desired.
B1	000000	136	--------	
	000032		---n	
B2	000100	137	A1 ------	
	000134		R.A. ------	

Note that we have made all of the pseudo addresses reflect fixed addresses.

We now write the following key words:

800003000100, a card containing this key word will be followed by the cards containing the fifty numbers expressed in standard decimal form; (K or S and eleven decimal digits).

800000000132, a card containing this key word should be followed by the card containing words 132 through 137.

800000000138, a card containing this key word should be followed by the three cards corresponding to the subroutine. (This is not the standard procedure since all subroutines exist on cards in binary form. We will diverge from the standard procedure in this case to maintain simplicity.)

800001000132, a card containing this key word should follow the last card of the subroutine.

The next step would be the insertion of "20" type key words if needed. These cards, in the above given order, should be placed immediately behind the deck of cards corresponding to the Transcriber. This combined deck is then placed in the card reader. After the appropriate switches on the control panel are activated, the cards will automatically be read into the machine and corresponding binary cards will be produced. After the binary cards are produced, the machine stops operations. The binary cards are then placed immediately behind the deck of cards corresponding to the Input Routine, this combined deck is then placed in the card reader. Again, after activating the appropriate switches on the control panel, the cards will be automatically read into the machine. As soon as the Input Routine recognizes the "1" type key word, control will automatically be directed to the first order of the routine.[1] The routine will then be executed and the desired result will be printed on the teleprinter. The result will be printed in sexadecimal form since we made no provisions to have it reconverted to its decimal equivalent. We could have employed another subroutine for this purpose; but again for simplicity this was omitted.

If one desires to form the sum of fifty other numbers which satisfy the number size conditions, one need only "transcribe" the fifty decimal datum numbers. That is, one need only use the type "30" key word with the cards containing the fifty numbers to be summed, and the Transcriber. The binary cards thus produced can then replace the corresponding binary cards for the first set of 50 numbers that were originally included in the deck of binary cards that was used with the Input Routine. Thus, the original routine in binary card form can be used as often as desired, with appropriate changes if desired.

1. This assumes switch, S2, is in the stop disable (down) position.

Exercise 1. Construct the necessary key words which will permit one to obtain the sum of 500 numbers which satisfy the number size conditions. Assume that we wish to use the original binary card deck of the previous example. The appropriate changes are to be made through the use of key words and words associated with the key words.

Exercise 2. Assume that the number size conditions were not satisfied. That is, suppose that each number was merely known to be less than three in absolute value. Construct the necessary words and key words to modify the routine so that we could handle the case.

CHAPTER IX

CODE CHECKING

In previous chapters we discussed the various details that are necessary to prepare routines for machine execution. Since the details are so numerous, it is not surprising to find that users (or coders) make many mistakes. Some of the common mistakes are: to neglect to insert a needed instruction or datum number; to write an incorrect instruction or datum number; to direct control to an incorrect address; to forget to store a particular quantity; to assign conflicting storage positions; to use an incorrect key word; to fail to meet the requirements of a given subroutine; etc. As will become apparent with experience, there is no substitute for care and forethought. The process of detecting such mistakes in a given correctly formulated routine is called "code checking"[1].

Methods for detecting mistakes will vary with: experience, the complexity of a given routine, machine facilities for inspection and handling of mistakes, and existing routines designed specifically as checking aids. In what follows, we will be particularly concerned with the checking of a given routine. We assume that the formulation of the objective is correct, and hence, we will be concerned with the checking of the sequence of words which is supposed to represent the given formulation. Furthermore, we will assume that the machine, as apart from the code, is functioning properly. If one has reason to suspect that the machine is malfunctioning, it is customary to run the given routine two or three times, (time permitting), to determine if the machine results are consistent. If the machine produces inconsistent results under the same initial conditions, then the machine is obviously malfunctioning. Sometimes, machine malfunctions are determined by inspecting the contents of the arithmetic and control registers.

---

1. Some installations use the terms "debugging" or "tracing", we prefer the term "code checking" since it is indicative of the meaning.

Before a given routine is initially submitted to the machine for execution, one should be prepared with estimates sufficient to indicate approximate results to be expected from satisfactory operation of the given routine. In many cases we prepare "hand computations" so that results obtained from the machine can be compared with the results obtained by "hand computations". In some cases, anticipated results are obtained by other means. In any event, one must be able to determine if the results obtained from the machine are satisfactory or unsatisfactory. Hand computations which are performed in the same manner as the machine procedure enables one to check quantities of particular interest at various stages of the machine computation.

Having submitted a given routine to the machine for execution, we are immediately confronted with one of the following situations:

I. No results are produced;

II. Results that are produced are unsatisfactory;

III. Results that are produced are satisfactory.

I. If no results were produced, one should first determine if the Input Routine stored the given routine in the designated memory positions. This can be determined by employing the "Memory Print Out Routine". The objective of this routine is to print, (on cards in sexadecimal form), the contents of a designated range of consecutive memory positions. This routine can be stored in the same manner as any other routine or subroutine. Control should be directed to the first order of this routine at the time that control would ordinarily be directed to the first order of the routine being checked. The range of memory positions under consideration is the range of memory positions occupied by the routine being checked. The results printed by the Memory Print Out Routine will represent the existing "<u>internal</u>" routine as stored by the Input Routine. These printed results can be visually compared with the "<u>external</u>" routine, (handwritten copy), in this manner, discrepancies between the internal and external routines can be noted and appropriate corrections can be made. Of course it may happen that one is unable to obtain the results of the Memory Print Out Routine. In such a

case one should carefully examine the cards and key words associated with the routine that one is attempting to store.

Having verified that the given routine was stored correctly, one can now begin to analyze why no results were produced. Either

a. The machine stopped operation, or

b. The machine continued operation indefinitely, (cycled).

If the machine stopped operation, the control counter reflects the address of the order following the particular order that caused the machine to stop. The control register, R_3, exhibits the actual order that caused the machine to stop operation. An inspection of the control counter and control register might suffice to determine the mistake. For example, some of the reasons why the machine will stop operation are:

a1. a "print" order with no cards or tape in the corresponding output device;

a2. a "read" order with no cards or tape in the corresponding input device;

a3. a legitimate stop order, i.e., an instruction which tells the machine to stop operation;

a4. an undefined order.

If one cannot account for the particular order that caused the machine to stop operation, then it is advisable to employ the Code Checking Routine. The Code Checking Routine, hereafter called "code checker", will be discussed in an example. If a1 or a2 exists, this can easily be remedied by inserting the proper cards or tape in the appropriate device.

If case "b" exists, i.e., if the machine continued operation indefinitely and had to be stopped manually, one should attempt to determine which order or which sequence of orders the machine was repeating indefinitely. If, for example, the address displayed in the control counter remains constant, the machine is repeating an order or order pair in the memory position whose address appears in the control counter. An example of such an order pair is:

(P) = +T 1 U P.

In such a case, the address in the control counter remains fixed at P. The obvious mistake is that one did not intend to transfer control to P. Sometimes the process of manually starting and stopping the machine is sufficient to determine the addresses of some of the orders that are being repeated. In this process, the addresses in the control counter are observed each time the machine is halted. For example, this process might yield a series of addresses such as 2N3, 2S7, 2K1, 2KK, etc. This series of addresses is associated with some sequence and an inspection of the handwritten orders of this region might be sufficient to determine the difficulty. If the particular addresses are associated with a "legitimate" repetitive sequence, one should check the transfer orders, (conditional or unconditional), and the elements associated with the corresponding transfer orders. Again, if one is unable to determine the cause of the "cycle", it is advisable to employ the code checker.

II. If unsatisfactory results are produced, and if manual checks of initial data, subroutine requirements, logical flow connections, etc. fail to yield a reason for the unsatisfactory results, one again reverts to the code checker.

The objective of the code checker is to produce a detailed description of a given sequence of instructions that the machine is currently executing. This is to say that the code checker can produce a precise "blow by blow" description of all of the operations involved in a given sequence of instructions. The description thus produced is recorded on a deck of cards. The cards contain sexadecimal and decimal representations of the quantities used and obtained in the corresponding order. The cards are tabulated, thus giving a detailed description of each order that was checked. (A tabulator is an auxiliary device which prints on paper the characters corresponding to the perforations of a card.) This detailed description can be visually compared with the corresponding handwritten sequence of orders, (and the corresponding anticipated results). Apparent discrepancies can be noted and appropriate corrections can be made.

III. Even though a routine produces satisfactory results for a given case, this is not sufficient to assume that the routine will produce satisfactory results for all cases. This is particularly true if the case

being tested does not employ all of the options associated with a given routine. Sometimes, two, three, or even more tests are required to test the various options or extreme conditions associated with a given routine.

To illustrate the use of the Memory Print Out Routine and the code checker, we will "check" the subroutine example of Chapter VIII. The complete code is reproduced and shown in Figure 1, save for one exception. We have intentionally made one mistake in the coding of the subroutine to be checked. Assume that we are checking the subroutine for the first time and that we are unaware of the mistake. Before beginning the discussion of the test to check the subroutine, we wish to warn the reader in advance that the following discussion is lengthy and detailed. The discussion includes:

a. details of a test to check the subroutine;

b. details on the use of the Memory Print Out Routine;

c. details on the use of the code checker;

d. details on the analysis of the description produced by the code checker.

To test the subroutine, we will employ it to form the sum of 100 special numbers. Let the special numbers be: .0001, .0002, .0003, ..., .0098, .0099, .0100. We choose these special numbers for testing purposes because we know the exact sum of these numbers; hence, we know what result to expect if the subroutine is designed properly. We begin by assuming that the 100 numbers are recorded on cards in standard decimal form. Also, we can assume that we will construct the necessary key word to have the Transcriber and Input Routine respectively convert the decimal numbers to their binary equivalents and store them in designated memory positions, A1 through A100. The flow chart is as follows:

FIGURE 1.

ORDVAC Problem________

Sheet______

Seq.	Code	Word	Order	R_1		R_2		Mem.	Contents		Ord.	Contents		Description
					$2^{-39}(n)$	$2^{-19}A1$	$2^{-39}R.A.$							
I, 1	50400K	4000	E' IV, 7								IV,7	U II, 1	$2^{-39}(n)$	sets n
2	KN4001		A+ I, 3	$2^{-19}A1$	$2^{-39}R.A.$									
3	50400S	4001	E' V, 2								V, 2	+ T1	U [R.A.]	sets R.A.
4	080014		→ 20		$2^{-19}A1$	$2^{-19}R.A.$	$2^{-39}A1$							
5	504004	4002	E' II, 2								II, 2	+ T1	(+)[A1]	sets A1
6	K4000J		+ K2		2^{-39}									
7	50400N	4003	E'K1					K1	U II, 1	$2^{-39}(1)$				sets i = 1
8	300000		oM T1					T1	0					sets $S_o = 0$
II, 1	K40000	4004	+ T1	S_{i-1}										
2	K40000		+ $[A_i]$	S_i										
3	100000	4005	M T1					T1	S_i					
III, 1	K4400N		+ K1	U II, 1	$2^{-39}(i)$									
2	04400K	4006	(-) IV, 7	$2^{-39}(i - n)$										
3	20400S		C V, 1											i = n?
IV, 1	K4000J	4007	+ K2		$2^{-39}(1)$									
2	N44004		(+) II, 2	+ T1	$(+)A_i+1$									
3	104004	4008	M II, 2								II, 2	+ T1	$(+)[A_{i+1}]$	
4	K4000J		+ K2		2^{-39}									i + 1 → i
5	N4400N	4009	(+) K1	U II, 1	$2^{-39}(i+1)$									
6	10400N		M K1					K1	U II, 1	$2^{-39}(i+1)$				
7	N04004	400K	U II, 1											
-	000000		$[2^{-39}(n)]$											
V, 1	K40000	400S	+ T1	S_n										S_n → R1
2	N00000		U [R.A.]											
K1	N04004	400N	U II, 1											
	000000		$[2^{-39}(i)]$											
T1		0000	~~Temporary~~											
K2		000J	Constant $2^{-39}(1)$											

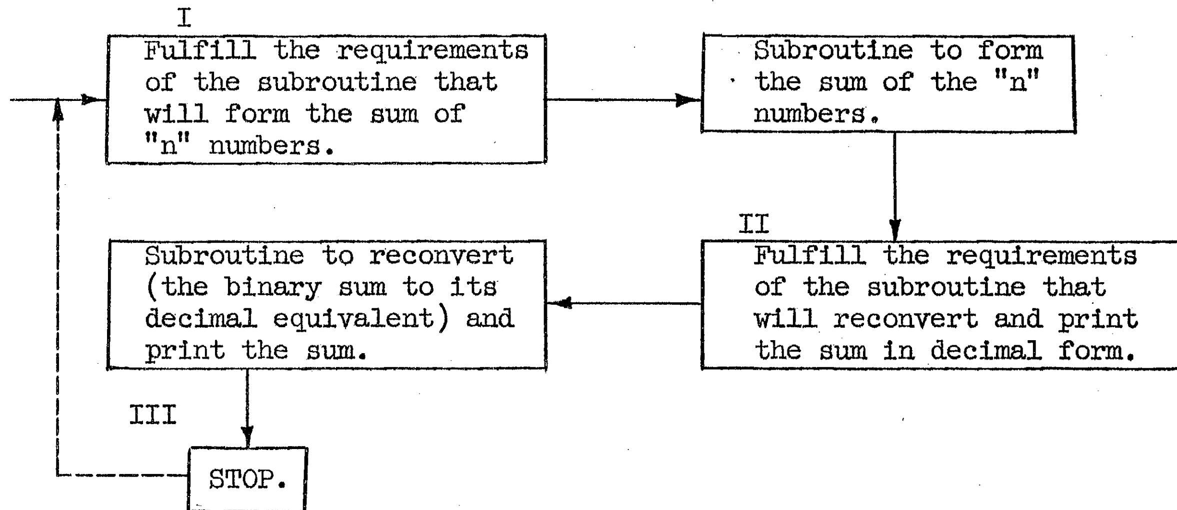


Figure 2

Notice, in Box II, that we are making provisions to have the sum reconverted and printed in decimal form. We have a standard subroutine available to do this. This subroutine requires eleven memory positions and the standard requirements are:

When control is directed to the first order of the subroutine,

a. the number to be reconverted must be located in R1;

b. the "R.A." must be located in the right address of R2.

The result is printed on the teleprinter before control is directed to the "R.A.". Thus, our plan indicates that we are going to employ two subroutines:

1. the subroutine to be checked, i.e. the subroutine for computing the sum of "n" numbers;
2. a subroutine for reconverting a binary number to its decimal equivalent before printing the decimal equivalent.

To fulfill the requirements of the summation subroutine, we write

```
I,1 + B1 }
         }  These orders fulfill the requirements of the
 ,2 R B2 }    summation subroutine.
 ,3 U Sub. Σ
 ,4 -----
```

B1 $2^{-39}n$

B2 2^{-19} A1 + 2^{-39} R. A.

(B1) and (B2) are the two words necessary to fulfill the requirements of the subroutine for forming the sum of "n" numbers, (subΣ). To fulfill the requirements of the reconversion and print subroutine we write:

II, 1 R B3
, 2 U Sub. Rv. & Print
B3 2^{-39} R.A.

Notice that we do not have to write an order to fulfill condition "a" of the reconversion and print subroutine. This condition is fulfilled by the summation subroutine in that the sum to be reconverted and printed is located in R1 when control is directed to the first order of the reconvert and print subroutine. The R.A. associated with the summation subroutine will be the address of II,1; the R.A. associated with the reconversion and print subroutine will be the address of III,1. These return addresses should be evident from an inspection of the flow chart. Referring to Box III we write:

III,1 Zu
,2 U I,1

Notice the order, "U I,1", following the stop order, "Zu". This order was inserted to provide for a repetition of the entire test, if desired. This is shown on the flow chart by the "broken line" extending from Box III to Box I.

Next, we consolidate the words of sequences I, II, and III, and assign the following sexadecimal addresses to the words of these sequences:

I,1	100	+ B1
,2		R B2
,3	101	U Sub. Σ
,4		-----
B1	102	
		$2^{-39}n$
B2	103	2^{-19} A1
		2^{-39} II,1

II,1	104	R B3
,2		U Sub.Rv. & Print
B3	105	
		2^{-39} III,1
III,1	106	Zu
,2		U I,1

Before we can write the final code for all of these words, we must decide on where we are going to store the two subroutines. The subroutine for computing the sum requires 13 memory positions; we will assign this subroutine to positions 107 through 113. The subroutine for reconverting and printing requires 11 positions, we will assign this subroutine to positions 114 through 11F. For the 100 numbers to be summed we will assign positions 200 through 263. The final code for sequences I, II, and III, using pseudo (four sexadecimal) addresses is:

I,1	K40102	100	+ B1
,2	S40103		R B2
,3	N00107	101	U Sub. $\sum$
,4	000000		-----
B1	000000	102	
	000064		2^{-39} n
B2	000200	103	2^{-19} A1
	000104		2^{-39} II,1
II,1	S40105	104	R B3
,2	N00114		U Sub.Rv.& Print
B3	000000	105	
	000106		2^{-39} III,1
III,1	000000	106	Zu
,2	N00100		U I,1

The raster showing the memory positions used for this example is shown in Figure 3. Notice that all pseudo addresses reflect fixed addresses. The following words, in the given order, will be punched on cards in standard form for use with the Transcriber.

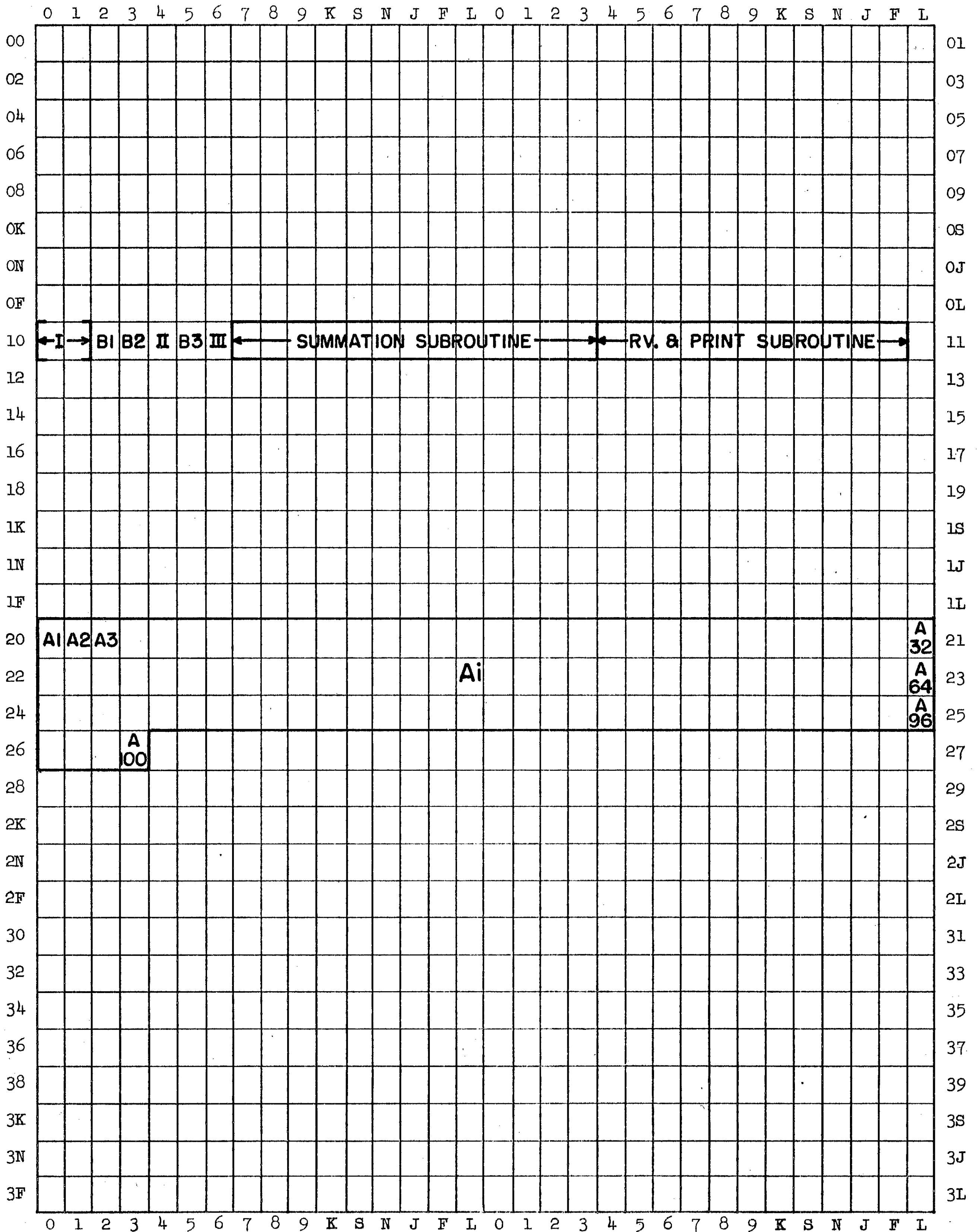


Figure 3

800000	000107	Key word to store summation subroutine
800000	000114	Key word to store Rv. & Print subroutine
800000	000100	Key word to store sequences I, II, and III
K40102	S40103	Sequences I, II, and III
N00107	000000	
000000	000064	
000200	000104	
S40105	N00114	
000000	000106	
000000	N00100	
800003	000200	Key word to convert and store the 100 numbers
K00010	000000	These are the 100 numbers to be summed, expressed in standard decimal form.
K00020	000000	
K00030	000000	
⋮	⋮	
K00990	000000	
K01000	000000	
800001	000LL1	Key word to complete a card and insure that the next key word (a "10" type) will appear alone on the next binary card. LL1 is the address of the first order of the Input Routine.
800001	000100	Key word to direct control to the first order of the test routine.

These cards will be transcribed, and corresponding binary cards will be produced. The first two binary cards produced will contain the key words for storing the two subroutines. This is in accordance with the arrangement discussed in the chapter on the Transcriber and Input Routines. The binary cards containing the two subroutines will be placed behind the cards containing the corresponding key words which designate where they are to be stored. These cards followed by the remaining binary cards will be placed behind the Input Routine. We are now ready to submit the routine to the machine for execution.

The cards are placed in the card reader, and after the activating of the appropriate switches on the control panel, the cards will be read and the machine will begin the operations. In less than one second after the cards are read, the teleprinter will begin to print the computed result. The result printed on the teleprinter is K010000000! The correct result is K505000000! Since we did not obtain the desired result, we can toggle S2 since we have made provisions to have the entire routine repeated. Before toggling S2 one should check the contents of R_3 to verify that the order that caused the machine to stop is the "Zu" order which we had stored in position 106. Further, the address in the control counter should be 107. Having verified the particular stop order, S2 can be toggled and again in less than one second the teleprinter should begin to print the computed result. Again the exhibited result is K010000000! This method of duplication is not sufficient to conclude that the machine is functioning properly. Notice that the routine was read into the machine only <u>once</u>. Indeed, the computation was performed twice, but the routine itself and the data were read only once. It is possible that the reader did not function properly. In this respect, the entire routine can be read into the machine and tried again. Again the exhibited result is K010000000! Under such circumstances, it is advisable to relinquish the use of the machine to the next candidate in line.

Obviously, there is a mistake somewhere. The problem now is to find the mistake. First, since the computed result looks familiar, it might be recognized as being the "last" of the 100 special numbers to be summed. This "clue" might be sufficient to determine the mistake in that such a result would be obtained if the sum <u>were not being accumulated</u>. This suggests a visual check of the (handwritten) sequence of words that were designed to accumulate the sum. If however, the incorrect result is not recognized, or if one is unable to detect the mistake, then it is advisable to determine if the <u>internal</u> routine corresponds to the <u>external</u> routine. To obtain the internal representation of the routine, we can employ the Memory Print Out Routine. Since the Memory Print Out Routine exists on cards in binary form, we need only transcribe a key word which will designate where we wish to store this routine. An inspection of the

raster will show that we can store this routine beginning at position 300. Further, since we will want to direct control to the first order of this routine, we must prepare another key word to do this. The following key words will enable us to accomplish these desired facilities:

1. 800000 000300 K.W. to store the routine at 300;
2. 800001 000300 K.W. to direct control to the first order of the Memory Print Out Routine.

Having obtained the two binary cards corresponding to these two key words, we insert the deck of binary cards corresponding to the Memory Print Out Routine between them. Next, we remove the last of the binary cards from the binary deck that we were using to test the given routine. (This is the card containing the key word that directs control to the first order of the test routine.) This card is replaced by the binary cards corresponding to the Memory Print Out Routine and the key words associated with it. To summarize, we have added the Memory Print Out Routine to the routine to be checked and control will be directed to the first order of the Memory Print Out Routine rather than to the first order of the test routine.

As yet, we have not indicated to the Memory Print Out Routine the specific range of memory positions whose contents are to be printed. That is, we have not fulfilled the requirements of the Memory Print Out Routine. The Memory Print Out Routine assumes that the "range" of memory positions, (whose contents are to be printed), is recorded on tape in the following form:

00 _ A _ 00 _ B _ .

This is a ten sexadecimal character word, where
"A" represents the address of the first memory position of the range;
"B" represents the address of the last memory position of the range.
It is assumed that the range is consecutive, i.e. A, A+1, A+2, ... B-1, B.
It is also assumed that the tape has been inserted in the tape input device when control is directed to the first order of the Memory Print Out Routine. For our example, we are interested in checking the "summation" subroutine and the sequence of words which fulfill the requirements of the

summation subroutine. The particular range of interest is included in the range 100 through 113. (Actually, this range includes sequences II and III, which we have added to the routine to obtain the result in decimal form). Hence we record on a paper tape the word

00100 00113,

and insert the tape in the tape input device. The binary cards are placed in the card reader, and after the activating of the appropriate switches on the control panel, the cards will be read. When control is directed to the first order of the Memory Print Out Routine, the tape will be read and the contents of the desired memory positions will be punched on cards. The contents of four consecutive memory positions will be recorded in sexadecimal form on each card. The contents of memory positions 100 through 103 will be recorded on the first card, the contents of memory positions 104 through 107 will be recorded on the second card, etc. The cards produced by the Memory Print Out Routine are tabulated, thus giving a typewritten copy of the internal routine as it appeared in the machine at the time that control would be directed to the first order of the routine. The typewritten copy thus obtained is shown in Figure 4.

```
K4102 S4103 N0107 00000 00000 00064 00200 00104
S4105 N0114 00000 00106 ......


                    .... N010S 00000
```

Figure 4

Tabulated (Typewritten) copy of the "internal routine".

The first word, K4102 S4013, represents the contents of position 100, the second word, N0107 00000, represents the contents of position 101, etc. A visual comparison of these words with the corresponding handwritten words (Figure 1) will show that the internal words correspond to the external words. The only difference is in the pseudo address character which does not appear in the machine, and hence does not appear in the internal words. In making a comparison between the internal and external words of the subroutine, one must account for the fact that the external words of the subroutine have addresses corresponding to the area of the memory beginning at 000, the internal words of the subroutine have addresses corresponding to

the area of the memory beginning at position 107. Hence, in making the comparison, one must expect a difference of 107 in each relative address. For example, the contents of 001 of the external routine is 50400K 080014. If 107 is added to the relative address, 00K, we obtain 111; thus, we should expect to find the contents of the corresponding internal word, (108), equal to 50111 08014. Notice that since "014" of the right order is fixed, it was not modified and hence appears as "014". Having verified that the internal routine represents the external routine, and if further visual checks fail to reveal any mistakes, it is advisable to employ the code checker.

The code checker is employed in a manner similar to any other routine or subroutine. That is, we first provide for storing the code checker in designated positions of the memory. Next, we must fulfill the necessary requirements of the code checker. The code checker has two general options, namely:

Option I. This option, called the "n" interval option, allows the user to specify "n" distinct (non-overlapping) intervals that are to be checked. That is, one can specify "n" intervals for which detailed information is desired. An interval is defined by two addresses, A and B, where A is the address of the first order of the interval to be checked and B is the address of the last order of the interval to be checked. In addition to specifying the addresses of the first and last orders of the interval, one must include whether the end of the interval is defined as the left or right order of B; checking will always begin at the left order of A. Thus, the general form of a word which specifies the addresses of the ith interval to be checked is as follows:

$$K_1 \; C_3 \; Ai \; C_2 \; 0 \; Bi$$

where:

Ai is the address of the first order of the ith interval;

Bi is the address of the last order of the ith interval;

$K_1 \leq 7$, indicates the number of times that detailed information is desired for the ith interval; if $K_1 = 0$, detailed information, relative to the ith interval. is printed everytime the order at A is encountered.

$C_2 = 0$ indicates that the left order of Bi is the last order of the interval;

$C_2 = 8$ indicates that the right order of Bi is the last order of the interval;

C_3 = 0 for i $\neq$ n;

C_3 = 8 for i = n; i. e;to indicate the last of the "n" intervals, C_3 = 8.

Ai and Bi must be addresses of orders which undergo no modification during the course of computation. That is, the start and stop orders of an interval should not contain variable addresses. The code checker assumes that the words defining the "n" intervals are recorded on paper tape in the form shown above, and that the tape is in the input device when control is directed to the first order of the routine to be checked. Thus, the words recorded on a tape for "n" intervals are:

K_1 0 A1 C_2 0 B1

K_2 0 A2 C_2 0 B2

K_i 0 Ai C_2 0 Bi

K_n 8 An C_2 0 Bn

Option II. This is called the "one" interval option. This option differs from Option I in two respects:

1) only "one" interval is specified;

2) this option allows one the facility to specify the number of times the given interval is to be traversed before checking begins, and the number of times that checking is desired.

The latter facility, 2) is desirable for intervals which are included in a repetitive sequence, since checking (i.e. printing of detailed information) is time consuming especially for long repetitive sequences. For Option II, the code checker requires that two words be recorded on paper tape and that the tape is in the input device when control is directed to the first order of the routine to be checked. The two words required by this option have the following form:

0 4 n 0 0 m

C_1 C_3 A C_2 0 B

where:

n = the number of times the interval is to be tranversed before checking begins;

m = the number of times checking is desired, (3 digits).

n and m are expressed in sexadecimal form;

The "4" in the first word of the pair is to allow the code checker to distinguish this option from Option I. $C_1 = 0$ indicates that the left order of A is the first order of the interval; $C_1 = 8$ indicates that the right order of A is the first order of the interval; C_2 has the same meaning as given in Option I. $C_3 = 0$ indicates that additional checking is desired after the checking of the current interval is complete.

$C_3 = 8$ indicates that no further checking is desired.

It is possible to change from Option II to Option I, however, the reverse is not possible. Under both options, the detailed information that will be printed is of the following form:

Order	Address of the order	(R1) Sexadecimal	(R2) Sexadecimal	Contents of the Memory Position Involved Sexadecimal	(R1) Dec.	(R2) Dec.	(Mem) Dec.	Blank
	00 _ _ _							
5 char.	3 char.	10 char.	10 char.	10 char.	10 char.	10 char.	10 char.	10 cols

The above information is printed after the order is executed. Some minor variations such as printing information on teletype, checking transfer of control orders only, etc, are available. The requirements for these variations are given in the local literature.

Now, to get back to the example under discussion, recall that we are interested in determining why the summation subroutine produced an unsatisfactory result. Assuming that we have been unable to determine any mistakes, we now resort to the facilities of the code checker. Our first concern is to select one of the two general options of the code checker. We shall employ Option II, i.e. the one interval option. Since the unsatisfactory result indicated that the summation was incorrect, we will want to define an interval which includes the "critical" orders of the summation subroutine. (See flow chart, Figure 1.) The critical orders of the summation subroutine are contained in sequences II, III and IV. The relative addresses spanning this critical interval are 004 and 00K. Since we are employing this subroutine beginning at position 107, the critical interval is defined as

004 + 107 = 10S and 00K + 107 = 111. Since we will be interested in detailed information the first time the orders in this interval are executed, we will set n = 0, implying that we want the interval "skipped" "zero" times before detailed information is produced. Further, since two times through the repetitive sequence should furnish enough information, we will set m = 2 implying that detailed information is desired only for the first and second passes through the critical interval. Notice that the critical interval will be traversed 100 times in order to obtain the desired result; however, it is not necessary to obtain detailed information for each repetition of the sequence since it is a repetition. Notice also that it is not necessary to obtain detailed information for the orders preceding this critical interval since any mistake in the preceding orders will be reflected when detailed information of the critical interval is obtained. Thus we construct the following two words to be recorded on tape:

1) 0 4 $\underbrace{000}_{n}$ 0 0 $\underbrace{002}_{m}$

2) 0 8 10S 0 0 111
C_1 C_3 A C_2 B

We can store the code checker beginning at memory position 300, (the same place where we had stored the Memory Print Out Routine.) This is advantageous since we need not prepare a new key word. We simply replace (in the "previous deck" we used when we employed the Memory Print Out Routine) the binary cards representing the Memory Print Out Routine by the binary cards representing the code checker. Further, we can replace the "last" binary card, of the "previous deck" which transferred control to the first order of the Memory Print Out Routine, by the "original" card which transfers control to the first order of our test routine. Under this arrangement, the code checker is read in "<u>last</u>". This arrangement is important since the routine to be checked must always be read in "<u>before</u>" the code checker. This is due to the fact that the code checker must "<u>modify</u>" orders of the routine to be checked in order to obtain control to begin the checking. We are now ready to employ the code checker.

The tape containing the words which define the interval to be checked is inserted in the tape input device. The binary cards are placed in the card reader and, after activating the proper switches on the control panel the routine is read in and the machine begins operations. The cards produced by the code checker are tabulated and the significant results are shown in Figure 5. (Others are also printed.)

Line (Not printed)	Order	Address	Sexadecimal (R1)	(R2)	(Memory) Position	Decimal (R1)	(R2)	(Memory) Position
1	K4000	10S		00104 00200		K0000 00000	K0000 00000	
2	K4200	10S				K0001 00000	K0001 00000	
3	10000	10N				K0001 00000	K0001 00000	
4	K4113	10N	N010S 00001		N010S 00001			
5	04111	10J	LLLLL LLL9J		N010S 00064			
6	24112	10J						
7	K400J	10F	00000 00001		00000 00001			
8	N410S	10F	K4000 00201		K4000 00200			
9	1010S	10L			K4000 00201			
10	K400J	10L	00000 00001		00000 00001			
11	N4113	110	N010S 00002		N010S 00001			
12	10113	110			N010S 00002			
13	N010S	111						
14	K4000	10S				K0001 00000	K0001 00000	
15	K4201	10S				K0002 00000	K0002 00000	
16	10000	10N				K0002 00000	K0002 00000	
17	K4113	10N	N010S 00002		N010S 00002			
18	04111	10J	LLLLL LLL9F		N010S 00064			
19	24112	10J						
20	K400J	10F	00000 00001		00000 00001			
21	N410S	10F	K4000 K4202		K4000 K4201			
22	1010S	10L			K4000 K4202			
23	K400J	10L	00000 00001		00000 00001			
24	N4113	110	N010S 00003		N010S 00002			
25	10113	110			N010S 00003			
26	N010S	111						

Detailed Description Produced by the Code Checker

Figure 5

The detailed description shown in Figure 5 consists of eight columns of numbers. The entries in the order column represent the actual orders that were checked by the code checker. The entries in the address column represent the respective addresses of the orders in the order column. The entries in columns three, four and five represent respectively the contents

of R1, the contents of R2, and the contents of the memory position involved in the order. The entries, expressed in sexadecimal form, represent the contents of R1, R2 and the memory position involved <u>after</u> the order was executed. The entries in columns six, seven and eight are the decimal equivalents written as K or S and 9 digits of the respective entries in columns three, four and five. The reason for the two representations, that is the sexadecimal and decimal, is that for some orders it is convenient to analyze the results in decimal form and for other orders it is convenient to analyze the results in sexadecimal form. The reasoning will become apparent when we analyze the detailed description shown in Figure 5.

To analyze the detailed description, one should have the handwritten copy of the routine being checked and a knowledge of the anticipated results of the orders of the routine being checked. In comparing the detailed description with the handwritten copy, one must account for the fact that the handwritten copy represents the subroutine as coded for memory positions beginning at 000 (or 4000). The detailed description represents the subroutine as coded for the area of the memory in which the subroutine was employed, namely the area of the memory beginning at 107. Thus, in making a comparison between the handwritten copy and the detailed description as produced by the code checker, we add 107 to each relative address of the handwritten copy. For example, when we defined the interval to be checked, we selected the repetitive sequences II, III and IV. The relative interval (interval associated with the handwritten copy) is 4004 through 400K. Since we stored this subroutine beginning at 107, the original orders associated with the relative interval, 4004 through 400K, will be stored and executed from positions 004 + 107 = 10S through 00K + 107 = 111. Thus, the detailed description is given in terms of the interval 10S through 111.

The entries in the address column represent the addresses of the sequence of orders that were checked. The sequence shown in the detailed description should correspond to the interval submitted to the code checker. An examination of this column shows that the sequence of orders checked was 10S, 10S, 10N, 10N, ... 111, and this same sequence was repeated. Thus, the entries in the address column show that the correct "sequence" of orders was checked the desired number of times. (Recall that we indicated that we

wanted this interval checked twice when we set m = 2.)

Consider the entries on the first line of the detailed description, K4000 10S 00000 00000 00104 00200 00000 00000 K0000 00000 K0000 00000. This represents the detailed information associated with the first order that was checked. The particular order was K4000, (column 2 of Figure 5). An examination of the handwritten copy shows that this order was designed to move the previous accumulated sum from temporary memory position T1 to R1. Initially, the previous accumulated sum should be zero. This was to be imposed by an order in a previous sequence. Therefore, as a result of this order, the contents of R1 should be exhibited as zero and the contents of the memory position involved, namely T1, should be zero. Hence, the results of this order are verified by examining the entries in columns three and five or columns six and eight. Columns three and five represent the sexadecimal equivalent of zero, columns six and eight represent the decimal equivalent of zero. Thus we conclude that this order is correct. In general, it is not necessary to check both the sexadecimal and decimal equivalents. For future checks of this particular order we will refer to the decimal entries in columns six and eight. Notice that the contents of R2 shown in column 4, i.e., (R2) in sexadecimal form, is 00104 00200. This represents the contents of R2 after the order K4000 was performed. Since the order executed, K4000, did not affect (R2), the entry shown there represents the result of the last order that affected (R2). An analysis of the last order that affected (R2) shows that the entry exhibited represents 2^{-19} R.A. + 2^{-39} A1, the result of the order I,4. In general, when analyzing the detailed description, only the entries of immediate interest are examined. The discussion of the contents of R2 was given to emphasize that the entries do represent the current contents of the registers and the memory position even if they were not affected by the current order.

Consider the entries on the second line of the detailed description:

			(R1)	Memory
K4200	10S	- -	K0001 00000	K0001 00000

This represents the detailed information associated with the second order that was checked. The particular order was K4200, (column 1 above). This order was the right order stored in memory position 10S, (column 2). An exami-

nation of the handwritten copy shows that this order was designed to add the jth number to the previous accumulated sum to obtain the jth partial sum. Notice that the address, 200, in this order, is the address of A1, the address of the first number to be added to the sum. Since the initial partial sum was zero, the new partial sum should be equal to the number stored in A1, i.e. a_1. A check of the test numbers shows that the first number, a_1, is .0001000000; hence, as a result of this order, the entries in columns six and eight should be .00010000000. An examination of these entries shows that these entries are correct. The next order was designed to store this new partial sum in T1, 000. An examination of the entry in column eight shows that this order was executed properly. Thus, the results of these first three orders correspond to the anticipated results.

For the remaining orders in this sequence, we will be interested in the results as represented in sexadecimal form; hence, we will examine the entries in columns three and five. Consider the entries on the fourth line of the detailed description:

K4113 10N N010S 00001 - - - - N010S 00001 - - - - - - - - - - - -

An examination of the handwritten copy shows that this order corresponds to the first order of sequence III, i.e. III, 1, + K1. This order was designed to move the counter, j, from K1 to R1. Since this is the first pass through the sequence, j = 1. Further, since j is scaled by 2^{-39}, the last sexadecimal character of the entries in columns three and five should be 1. An examination of these entries shows that the result is correct. One should not be disturbed by the fact that the entries contain more than the counter j, i.e., the entry N010S 00001 corresponds to the contents of K1, 113. The left characters, N010S, correspond to the order U II, 1, which is in accordance with the designed use of K1. An analysis of the entries on lines five thru thirteen of the detailed description, (that is, the remaining orders of the interval that was checked), will show that the entries correspond to the anticipated entries. Thus, the exhibited results of the first pass through the sequence indicate that these orders of the subroutine are doing the job they were designed to do.

Evidently, these results indicate that the interval that we selected to be checked is not responsible for the unsatisfactory result that was

obtained. Logically, then, one might conclude that the unsatisfactory result must be due to a mistake in one or more of the orders that follow the orders that have been checked. An examination of the handwritten copy will show that the orders that follow the interval that have not been checked correspond to the orders of Box V. These are the orders that store the final sum in R1 and transfer control to the R.A. These orders are followed by the reconversion and print orders. Since the reconversion and print subroutine has previously been checked, the probability of a mistake therein is small. One should check to see that the requirements of this subroutine have been fulfilled. A check of the orders of Box V will reveal that they are correct. Similarly, a check to see that the requirements of the reconversion and print subroutine have been fulfilled will also indicate that they are correct. A check of the initial data and key words will show that they are correct. Thus, we know that there is a mistake somewhere, as indicated by the unsatisfactory result, and yet, our efforts to determine the mistake have been in vain. . . . Perhaps by this time, the reader has observed the false reasoning in our analysis of the detailed description. The reader is referred to an earlier statement in the discussion which reads, "Thus, the <u>exhibited results</u> of the first pass through the sequence indicate that these orders of the subroutine are doing the job they were designed to do." Is it possible that the <u>exhibited results</u> are deceiving? In particular, can an <u>incorrect order</u> produce a <u>correct result</u>? The answer to both questions is yes! The following discussion will reveal the false reasoning we employed in the analysis of the detailed description.

Consider the entries on the fourteenth line of the detailed description;

K4000 10S - - - - - - - - - - - - - - - - - K0001 00000 - - - K0001 00000

The entries on this line correspond to the detailed description of order II,1 when it was executed the <u>second</u> time. As such, the design of the order is unchanged, i.e. this order was designed to move the previous accumulated sum from temporary memory position T1 to R1. We know the existing partial sum at this time is .00010 00000, consequently, the entries in columns six and eight of line fourteen should be .00010 00000. An examination of these entries shows that they are correct. Consider the entries on line fifteen:

K4201 10S - - - - - - - - - - - - - - - - - K0002 00000 - - - K0002 00000

The entries on this line correspond to the detailed description of order II,1 when it was executed the second time. This order was designed to add the jth number, (at this time $j = 2$), to the previous accumulated sum to obtain the jth partial sum. We know the address of the second number to be added to the sum; i.e. the address of a_2 is 201. This is verified by examining the address portion of the order, K4201. Further, we know that $a_2 = .00020\ 00000$, consequently the entry in column eight should be .00020 00000. An examination of this entry will show that it is correct. Now, if $a_1 = .00010\ 00000$, and $a_2 = .00020\ 00000$, then $a_1 + a_2 = .00030\ 00000$. This is the entry one should expect to find in column six. An examination of the entry in column six shows K0002 00000!! This is not the correct result. We expected K0003 00000 and the exhibited result is K0002 00000. Why the error? This particular order produced the correct result when it was executed the first time! At that time, the expected result was 00000 00000 + 00010 00000, i.e. $S_0 + a_1$, and the entry exhibited was as expected. The fact is, however, that the actual order performed was not an addition!! The order performed was simply an order to move the number from Aj to R1. Our false reasoning was due to the fact that

$$S_0 + a_1 = a_1, \text{ if and only if } S_0 \text{ is zero,}$$

and in the first execution of the order, S_0 was zero. That is, the order " + Aj ", is equivalent to the order "(+) Aj" if (R1) is zero at the time these orders are executed. Thus, our efforts have not been fruitless, the correct order corresponding to II,2 should be initially "N4000", not "K4000".

This correction should be made and the original test rerun. When the necessary test (or tests) produce the desired results the subroutine is considered checked. Generally, the entire checking procedure is repeated as often as is necessary. If there are mistakes in other intervals of the routine being checked, repeated applications of the code checker may be necessary. As can be readily seen from the previous discussion, the process of detecting mistakes can be tedious, time consuming and exasperating. Surely, the previous discussion accentuates the old proverb, "an ounce of prevention is worth a pound of cure".

Exercises:

1) Prepare the necessary (tape) words for use with the memory print out routine to obtain the sexadecimal print outs for ranges:

a. 21S thru 6K4
b. KNO thru JL3
c. 000 thru LLL.

2) Prepare the necessary tape words for use with the code checker to obtain a detailed description of the orders in the following intervals:

a. 10L thru 1J2, each time this interval is traversed
b. 3N3 thru 20K, the third, fourth and fifth time this interval is traversed
c. 688 thru 70L, the first three times this interval is traversed.

Assume that the above intervals begin with "left" orders and terminate with "right" orders.

CHAPTER X

"IBM IN" AND "IBM OUT" ORDERS;

"IBM IN" AND "IBM OUT" SUBROUTINES

Most problems to be solved by high-speed computers require a means of getting data into the machine, and similarly most problems require some means of getting data out of the machine. In view of this fact, certain logical orders are built in the machine to enable the coder to transfer data from (to) some external medium, (such as cards, tapes, etc.), to (from) some internal unit, (such as the arithmetic registers, the high-speed memory, etc.). Since these orders (for Ordvac) are primarily designed to transfer data which is represented in binary form, subroutines have been designed to facilitate the transfer of data which is represented in decimal form.

The purpose of this chapter is:

1. to describe the logical orders which are designed to transfer binary data from IBM cards to the arithmetic registers of the Ordvac, or from the arithmetic registers to IBM cards;
2. to describe and illustrate the use of two subroutines which are designed to transfer decimal data from IBM cards to the core memory, or from the core memory to IBM cards. We shall refer to these subroutines respectively as the "IBM IN" subroutine and the "IBM OUT" subroutine.

"IBM IN" ORDER.

The "IBM IN" ORDER transfers all 80 bits from one designated row of an IBM card, sending 40 bits to register R1 and 40 bits to register R^3. The description of the rows, columns, and punching positions of an IBM card was given in Chapter III. The first 40 bits of a row, i.e., the bits recorded in punching positions 1 through 40, are transferred to R1. The last 40 bits of the row, i.e., the bits recorded in punching positions 41 through 80, are transferred to R^3. A perforation, (punch), in a punching position is interpreted as a binary "one"; a non-punch is interpreted as a binary "zero".

Example:

Assume that the card shown in Figure 1 is in the card reader.

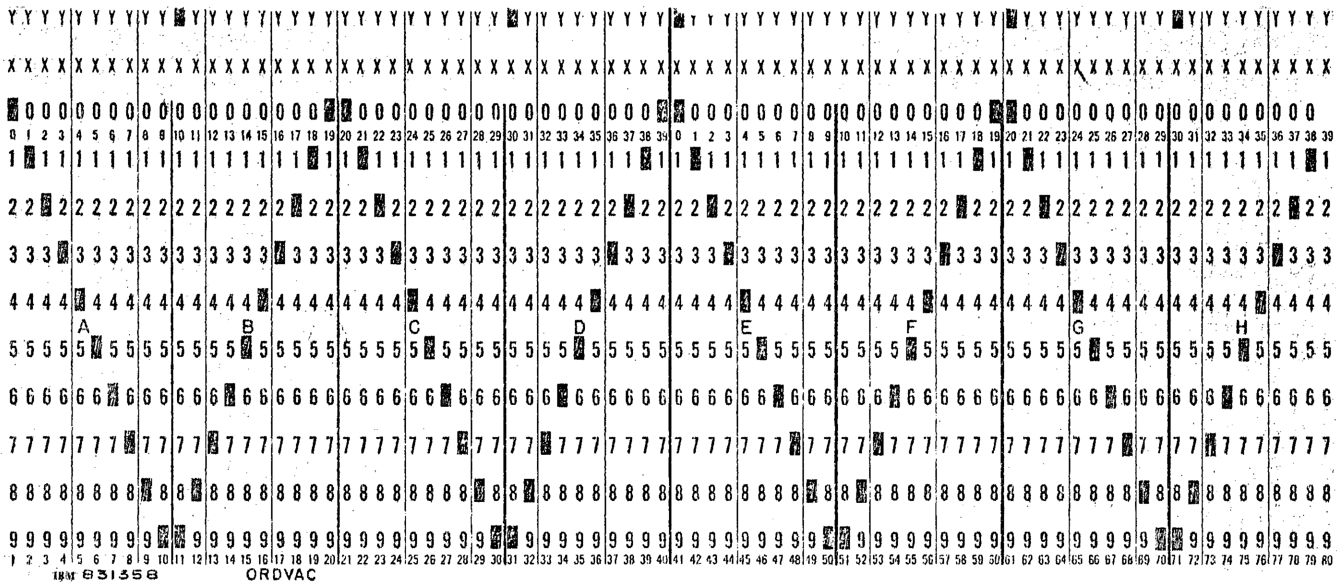


Figure 1

As a result of the first "IBM IN" ORDER, the contents of R1 and R^3 would be as follows:

$(R1)$ = 0000000000 1000000000 0000000000 1000000000

(R^3) = 1000000000 0000000000 1000000000 1000000000.

That is, the contents of the first row, (the "Y" row), would be transferred to R1 and R^3.

Since each "IBM IN" ORDER reads in one row of a card, twelve such orders are necessary to read in the twelve rows of a given card. Further, the first "IBM IN" ORDER, (the first of twelve with respect to a given card), will cause the twelve successive rows of the card to "pass by" a row reading mechanism at equal intervals of time. That is, the first row will be in position to be read at time "one"; the second row will be in position to be read at time "two"; etc; the twelfth row will be in position to be read at time "twelve". Therefore, it is important that twelve successive "IBM IN" ORDERS, with respect to a given card, be executed within definite time intervals. Since it takes a mechanical device a definite amount of time to move the "next" row of a card into

position to be read, a definite amount of time is available for computing between successive row reads. This is to say that during the time that a row is being moved into position to be read, the computer, (as apart from the card reader), may be used to execute orders different from the "IBM IN" Order. Indeed, a definite amount of time is necessary, (between successive row reads), to move the words that have been read into R1 and R^3 to designated positions in the core memory. In particular, the maximum amount of time that can be used to execute orders between successive "IBM IN" Orders is 7 milliseconds. This 7 millisecond maximum is based on a card reading rate of 84 cards per minute. If the 7 millisecond maximum is exceeded,then access to the next or succeeding rows of the card is lost, since the next or succeeding rows will have passed by the row reading mechanism. The number of rows that will have passed by the row reading mechanism is proportional to the amount of time exceeded. The fact that a row passes by the row reading mechanism does not imply that the row has been read, the row will be read if and only if an "IBM IN" Order is encountered just before or at the time the row is in position to be read. If a row is not in position to be read when an "IBM IN" Order is encountered, the computer waits until a row is in position before carrying out the "IBM IN" Order.

Similar to the 7 millisecond maximum time interval available between successive row reads, a minimum of 300 milliseconds is available between successive card reads i.e. between the last "IBM IN" Order associated with one card and the first "IBM IN" Order associated with the next card. This is to say that it takes 300 milliseconds to move the next card into position to be read.

Summary: The "IBM IN" Order reads one row of an IBM card into registers R1 and R^3. A row is interpreted as representing 80 bits. Twelve such orders are required to read the twelve rows of a card. A maximum of 7 milliseconds is available for executing orders between successive row reads. A minimum of 300 milliseconds is available for executing orders between successive card reads.

"IBM OUT" ORDER:

The "IBM OUT" Order is similar to the "IBM IN" Order. The "IBM OUT" Order records (punches) the contents of R1 and R2 in binary form on one row of an IBM card. The contents of R1 is recorded in the first 40 punching positions of the row. The contents of R2 is recorded in the last 40 punching positions of the row. Twelve such orders are necessary to record data in the twelve rows of a given card. A maximum of 6 milleseconds is available for executing orders between successive "IBM OUT" Orders. This maximum of 6 milliseconds is based on a card punching rate of 100 cards per minute. A minimum of 300 milliseconds is available between successive card recordings, (prints or punches).

EXAMPLE: Assume that the contents of R1 and R2 are as follows:

(R1) = 10101010101010101010. . . . . . .

(R2) = 10001000100010001 . . . . . .

In this case, as a result of an "IBM OUT" Order, the card shown in Figure 2 would be produced:

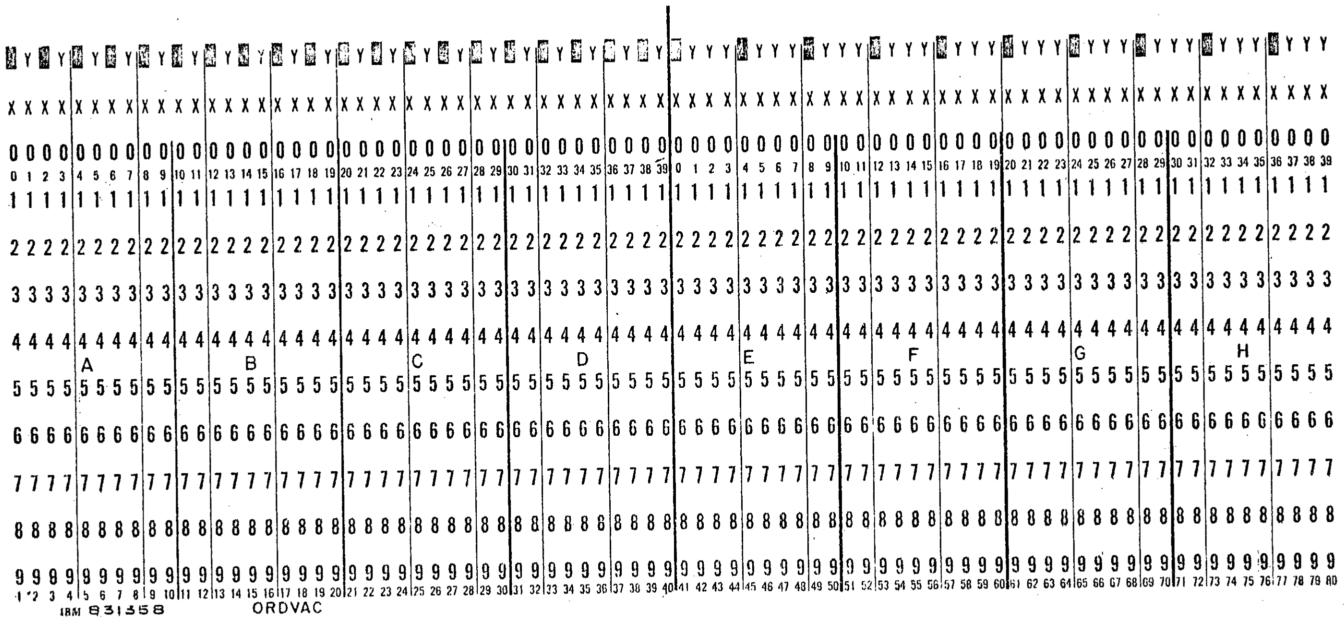


Figure 2

The "IBM IN" and "IBM OUT" orders are not described in the Appendix.

CARD "FORMAT" AND "FIELD" WORDS

In order to accommodate a wide class of problems, the IBM subroutines, (to be discussed presently), are designed to accept a "variable" number of

quantities (datum numbers) per card. Further, each quantity (datum number) may be represented by a "variable" number of decimal digits. The arrangement of quantities (datum numbers) on a card, as pertains to the order of the quantities and the corresponding number of columns that are used to represent each quantity, is called a card "format". A standard name that is used to refer to a group of successive columns of an IBM card is a "field". To indicate, to the subroutines, the specific card format that is being employed, "field" words are constructed which define the card format. That is, "field" words indicate the number of quantities represented on a card (cards) and the number of columns that are used to represent respective quantities.

A field word is composed of ten sexadecimal characters as follows:

$$S_1\ S_2\ S_3\ S_4\ S_5\ S_6\ S_7\ S_8\ S_9\ S_{10},$$

or more generally, S_i, $i = 1, 2, 3, \ldots, 10$. Being sexadecimal, each of these has (by definition) any one of 16 distinct values, 0, 1, 2, ..., 9, K, S, N, J, F, L. Each of these is to be interpreted as follows:

S_i = 1, 2, 3, ..., 9, K, S, defines the number of columns of the card that are used to represent the corresponding decimal quantities.

S_i = N, J, F, and L have special meanings. A signal to omit a certain number of columns (from one to fifteen) requires two sexadecimals, "Jx". Omit 3 columns is "J3"; omit 13 columns is "JJ". Note the distinction of the two "J's" in the last example. Omit 26 columns may be expressed in several ways such as: "JJJJ"; or "JNJF"; or "J9J9J8"; etc. When not preceded by "J" as above, the symbol S_i = "N" indicates the end of the card format. At present, the symbols S_i = "F", or "L" have no meaning to the "IBM IN" subroutine except when preceded by "J", the omit signal. The symbol S_i = "F" will be assigned a special meaning for use with the "IBM OUT" subroutine. S_i = "0" instructs the subroutines to continue interpreting the format with the next field word.

The following examples illustrate the construction and interpretation of field words.

Example 1. 78SNO 00000.

This field word defines the following card format:

the first quantity on the card is represented in columns 1 through 7;

" second " " " " " " " " 8 " 15;

" third " " " " " " " " 16 " 26;

Figure 3 shows a typical card corresponding to the format defined by the field word of Example 1.

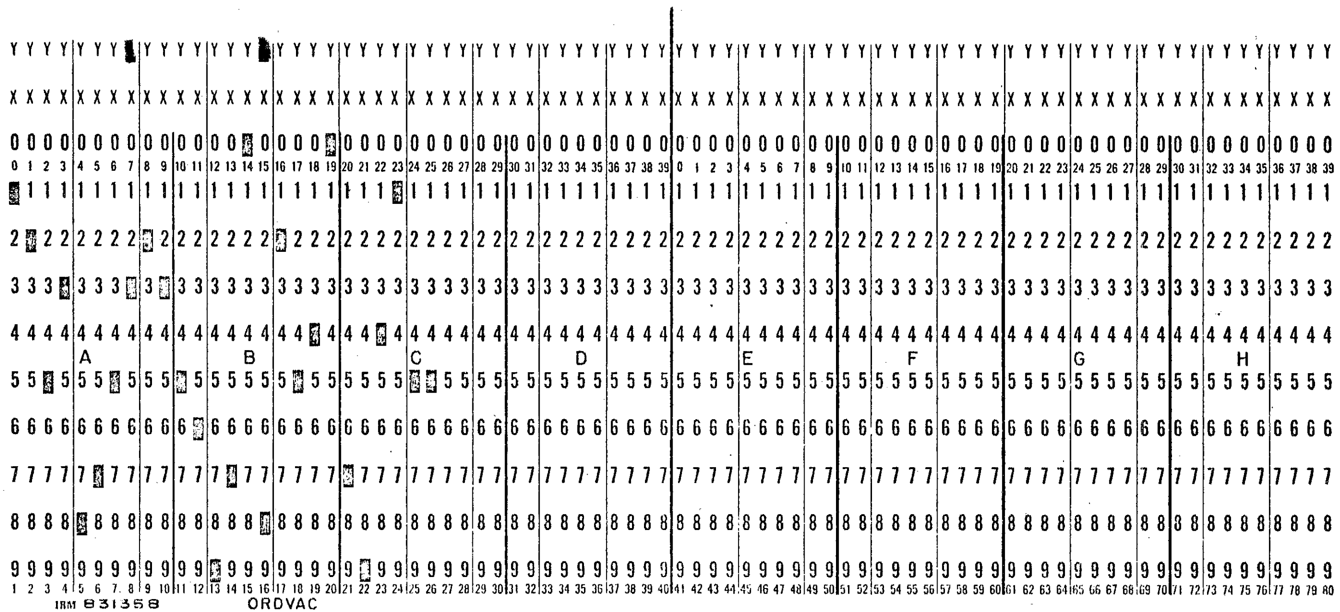


Figure 3

The second and third quantities on this card, "Y_1" and "Z_1", represent negative quantities. Negative quantities, such as these, are indicated by the presence of the "Y" punch in the first column of the field. The absence of the "Y" punch over the first column of a field indicates a positive quantity. Thus, the quantities on the card shown in Figure 3 are:

$$X_1 = + .1253875$$

$$Y_1 = - .32356970$$

$$Z_1 = - .82540794155.$$

Other options are available whereby one may represent negative quantities by the presence of an "X" punch instead of the "Y" punch, or one may use a separate column to define the sign of the quantity.

Example 2. 78SJ4 6N000.

This field word defines the following card format:

the first quantity on the card is represented in columns 1 through 7;
" second " " " " " " " " 8 " 15;
" third " " " " " " " " 16 " 26;

"J4" is a signal to ignore columns 27 through 30;
the fourth quantity on the card is represented in columns 31 through 36; the "N" indicates the end of the format.

Example 3. JLK85 JLJL5
5N000 00000.

This pair of field words defines the following card format:
columns 1 through 15 are to be ignored as indicated by "JL";
the first quantity on the card is represented in columns 16 through 25;
" second " " " " " " " " 26 " 33;
" third " " " " " " " " 34 " 38;

the thirty columns, 39 through 68, are to be ignored as indicated by "JLJL"
the fourth quantity on the card is represented in columns 69 through 73;
" fifth " " " " " " " " 74 " 78.

As can be seen from this example, more than one field word may be necessary to define a card format.

Example 4. 88888 88888
K5SN0 00000.

This pair of field words defines a format for thirteen quantities which are represented on two successive cards:
ten quantities, of eight digits each, are represented on the first card;
three quantities, of ten, five, and eleven digits respectively, are represented on the second card.
As can be seen from this example, a group of field words may define a format corresponding to more than one card.

Note that no examples were cited which illustrated the use of "$S_i = 0$" i.e. the signal to the subroutines to continue interpreting the format with the "next" field word. This character is used primarily to cancel a field word which had been established for checking purposes. For example, the first field word of Example 4 may have been employed for printing subsidiary quantities for checking purposes. The second field word may have been employed to define the format for the quantities to be printed during the "regular" run of operations. Having determined that the subsidiary quantities are no longer desired, one can "cancel" the first field word by replacing it with "00000 00000", or "08888 88888", etc.

EXERCISES:

Construct the field words corresponding to the following card formats:

a. For five quantities on one card, where:

x	is	to	be	represented	by	nine	decimal	digits	in	columns	1	through	9;
y	"	"	"	"	"	eight	"	"	"	"	10	"	17;
z	"	"	"	"	"	eleven	"	"	"	"	18	"	28;
u	"	"	"	"	"	"	"	"	"	"	29	"	39;
v	"	"	"	"	"	ten	"	"	"	"	40	"	49.

b. For the same set except that "v" is to be represented by seven decimal digits in columns 51 through 57, instead of in columns 40 through 49.

c. For twenty quantities represented on two cards, where each quantity is to be represented by five decimal digits: 16 quantities on the first card in columns 1 through 80; and 4 quantities on the second card in columns 1 through 20.

d. For the twenty quantities given in "c", except that the four quantities on the second card are to be represented in columns 31 through 50, instead of columns 1 through 20.

Let us now consider the meaning assigned to S_i = "F" in the "IBM OUT" subroutine.

In many problems it is desirable to have the output cards serially identified. That is, it is convenient to have recorded on each output card a number which can be taken to mean "this is the first card that was

produced, this is the second card that was produced, this is the "j^{th}" card that was produced", etc. In addition to the serial numbering of the output cards, it is often desirable to have a "code number", (or case number), recorded on each card so that each card can be identified or associated with a particular problem or a particular case of a general problem. For example, if one is running 100 cases of a given problem, where each case requires 500 output cards, it is convenient to be able to associate each output card with a particular case (one of the 100 cases) and the particular serial number (one of the 500). Since this identification facility is desirable, it is included in the "IBM OUT" subroutine and is recognized by the "IBM OUT" subroutine when S_i = "F" in a field word.

To effect this facility, the "IBM OUT" subroutine assumes that the code number, (or case number), is stored in the second word, W2, of the subroutine. Likewise, the subroutine assumes that the "unit" serial number, (or serial "increment"), is stored in the third word, W3, of the subroutine. As each card is being prepared for punching, the subroutine forms the sum (W2) + (W3) and stores this sum in W2. As each card is produced, the contents of W2 is recorded in the columns of the card defined by "Fq". That is, "F" is the signal to the subroutine that identification is desired, and "q" indicates the number of columns of the card that are to be used to represent the combined code and serial numbers, (W2). $(1 \leq q \leq 8)$. In the example cited above, where there were 100 cases, each of which necessitated 500 output cards, three columns would be sufficient to identify the case number and three columns would be sufficient to identify the serial number. That is, one could identify the cases by the integers 1, 2, 3, ..., 100; similarly, one could identify the serial numbers by the integers 1, 2, 3, ..., 500. Initially, one would be required to store the quantity $10^{-3}(1)$ in W2, and the quantity $10^{-6}(1)$ in W3. The quantity $10^{-3}(1)$ represents the initial case number, and the quantity $10^{-6}(1)$ represents the serial increment. The scaling of "j" by 10^{-3} and "i" by 10^{-6} actually "positions" the "j" and "i" in the combined sum, "$10^{-3}(j) + 10^{-6}(i)$". That is, the first three digits of (W2) represents "j" and the next three digits represent "i".

Example: If one stored:

$10^{-3}(1)$ in W2;

$10^{-6}(1)$ in W3,

"F6" in the field word defining the format;

then, the cards that are produced would have the following identification recorded in the columns defined by "F6":

the first	card	produced	would	contain	"001001"	in	the	designated columns;
the second	"	"	"	"	"001002"	"	"	" "
the third	"	"	"	"	"001003"	"	"	" "
etc.								
the "500th"	"	"	"	"	"001500"	"	"	" "

If one desired to have "j" and "i" separated by one blank column, the combined identification would require seven columns and the scaling of the serial increment would have to be 10^{-7}. To obtain identification for the second case, i.e. "j = 2" the coder would have to "reset" (W2) to $10^{-3}(2)$. This would have the effect of setting both "j = 2", and "i" = 0". The corresponding output cards would be identified as "j = 2" and "i" = 1, 2, 3, ..., 500.

In general, the identification recorded in the "q" columns of a card represents the first "q" digits of (W2), where

(W2) = (W2) + (W3).

Let us now take up the discussion of the two subroutines.

<u>"IBM IN" Subroutine (105 words with 2 FW's).</u>

The objective of the "IBM IN" subroutine is:

1. to <u>read</u> decimal quantities from "IBM" cards;
2. to <u>convert</u> the decimal quantities to corresponding binary equivalents;
3. to <u>store</u> the binary equivalents in designated memory positions.

The requirements for using the "IBM IN" subroutines are:

a. store the return address in the right-address position of R2;
b. store, in R1, a special two-part word of the form,

$$0\ 0\ \underbrace{X_1\ X_2\ X_3}_{\text{"M"}} \qquad 0\ 0\ \underbrace{X_4\ X_5\ X_6}_{\text{"P"}}$$

, where

"M", represented by the three sexadecimal characters $X_1X_2X_3$, specifies the number of decimal quantities that are to be read, converted and stored;

"P", represented by the three sexadecimal characters $X_4X_5X_6$, specifies the address where the first converted quantity is to be stored; the converted quantities are stored in consecutive memory positions, i.e. in P, P + 1, P + 2, etc.

c. specify the card format, i.e., store the field word (words) immediately following the last word of the subroutine proper.

"IBM OUT" Subroutine (128 words with 2 FW's).

The objective of the "IBM OUT" subroutine is:

1. to reconvert binary quantities to their corresponding decimal equivalents;
2. to record (punch) the decimal equivalents on "IBM" cards.

The requirements for using the "IBM OUT" subroutine are:

a. store the return address in the right-address position of R2;
b. store in R1 a special two-part word of the form,

00"M" 00"P", where

"M" represents the number of quantities that are to be reconverted and recorded;

"P" represents the address of the first quantity to be reconverted.

c. specify the card format, i.e. store the field word (words) immediately following the last word of the subroutine proper.

Note the similarity in the requirements for the "IBM IN" and the "IBM OUT" subroutines. The only significant difference in the two subroutines is that the "IBM OUT" subroutine includes the "identification facility.

To illustrate the use of the "IBM IN" and the "IBM OUT" subroutines consider the following problem:

Illustrative problem:

Given:

1. 30 sets of data, where each set is composed of sub-sets, $P_i = (x_i, y_i, z_i,)$ $i = 1, 2, 3, \ldots, 50$;
2. each sub-set is recorded on an IBM card as follows:

x	is	represented	by 7	decimal	digits	in	columns	11	through 17;
y	"	"	" 9	"	"	"	"	20	" 28;
z	"	"	" 11	"	"	"	"	32	" 42;

3. definitions of: $F(P_i)$, $G(P_i)$, and $H(P_i)$, hereafter referred to as

F_i, G_i, and H_i.

Wanted:

4. For each sub-set, P_i,
   a. compute: F_i, G_i, H_i;
   b. print (punch) in decimal form F_i, G_i, H_i, "set" number, (hereafter referred to as "j") and "serial" number, (hereafter referred to as "i"):

F_i	is	to	be	recorded	in	columns	1	through 7;
G_i	"	"	"	"	"	"	10	" 16;
H_i	"	"	"	"	"	"	19	" 25;
j	"	"	"	"	"	"	35	" 36;
i	"	"	"	"	"	"	39	" 40.

5. For each set
   a. compute $\sum F_i$, $\sum G_i$, $\sum H_i$;
   b. print (punch) in decimal form $\sum F_i$, $\sum G_i$, $\sum H_i$, "j", and "i" = 51

$\sum F_i$	is	to	be	recorded	in	columns	1	through 10;
$\sum G_i$	"	"	"	"	"	"	12	" 21;
$\sum H_i$	"	"	"	"	"	"	23	" 32;
j	"	"	"	"	"	"	35	" 36;
i=51"	"	"	"	"	"		39	" 40.

To specify the card format for each input card, (the cards containing the P_i), a field word is constructed according to the information given in "2". The field word specifying this format is:

JK7J2 9J3SN.

To specify the card format for each output card corresponding to each sub-set, P_i, field words are constructed according to the information given in "4b". The field words specifying this format are:

7J27J 27J90
F6N00 00000.

Notice the "F6" indicates that six columns are to be used for the combined identification of "j" and "i", namely columns 35 through 40. Since "4b" requires that "j" be recorded in columns 35 and 36, "j" must be scaled by 10^{-2}. Similarly, since "i" is to be recorded in columns 39 and 40, "i" must be scaled by 10^{-6}. Hence, to meet the identification requirements of the "IBM OUT" subroutine we will need the constants $10^{-2}(1)$ and $10^{-6}(1)$ to define the initial "j" and the serial increment "i".

To specify the format for each output card corresponding to each set, field words are constructed according to the information given in "5b". The field words specifying this format are:

KJ1KJ 1KJ20
F6N00 00000.

For future references, we will assume that the field words and constants are stored as follows:

JK7J2 9J3SN in B1
7J27J 27J90 in B2
F6N00 00000 in B3, B5
KJ1KJ 1KJ20 in B4
$10^{-2}(1)$ in B6
$10^{-6}(1)$ in B7.

A flow chart for this example is shown in Figure 4.

As is indicated on the flow chart, the "IBM IN" subroutine will be employed in Box IV; the "IBM OUT" subroutine will be employed in Boxes VI and VIII. We shall restrict our discussion to the orders corresponding to Boxes I, II, IV, VI, VIII, and X, as these are the boxes associated with the use of the "IBM IN" and "IBM OUT" subroutines.

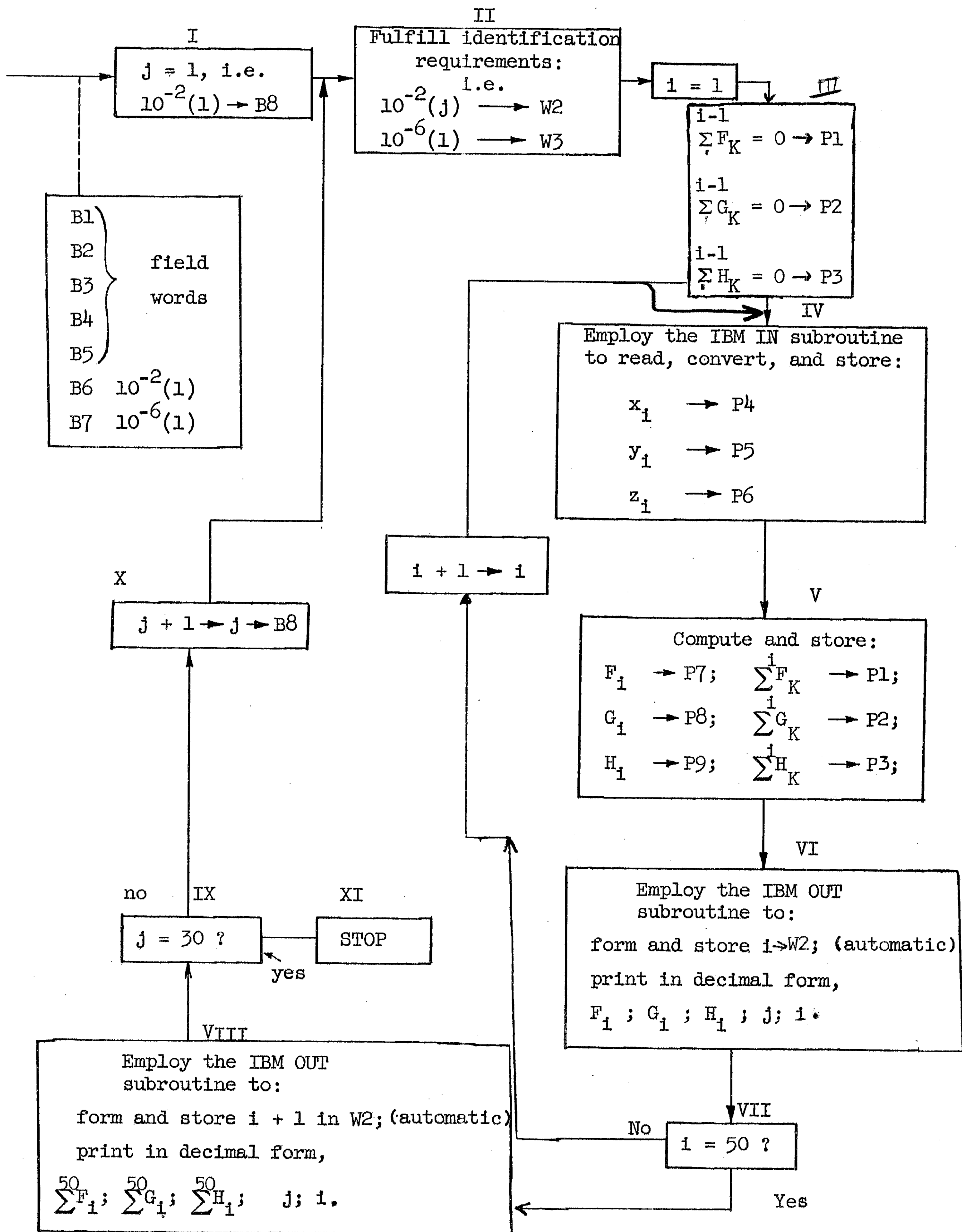


Figure 4

Assume that:

1. the field words and constants are stored in B1 through B7 as indicated on the flow chart;
2. the "IBM IN" subroutine is stored in positions K1 through K103,
3. the "IBM OUT" subroutine is stored in positions W1 through W126;

We shall construct the "special" instruction words and "return-addresses" for the respective subroutines as their needs arise.

Preliminary Order.		Description
I,1	+ B6	$10^{-2}(1) \rightarrow$ R1
,2	M B8	$j = 1$, i.e. $10^{-2}(j) = 10^{-2}(1) \rightarrow$ B8
		This pair of orders sets the initial "j" equal to one. Notice that "j" is stored in B8. The orders of Box II will refer to B8 to fulfill the identification requirement, i.e. to store the "set" number, "j" in W2.
II,1	+ B8	$10^{-2}(j) \rightarrow$ R1
,2	M W2	$10^{-2}(j) \rightarrow$ W2
,3	+ B7	$10^{-6}(1) = 10^{-6}(\Delta i) \rightarrow$ R1
,4	M W3	$10^{-6}(1) = 10^{-6}(\Delta i) \rightarrow$ W3
		These orders fulfill the requirements of the "IBM OUT" subroutine to record the identification on each output card. Notice the order II,2, which stores "j" in W2, actually sets "i" = 0, since the contents of W2 represents the combined identification.
IV,1	+ B1	F1, (field word No. 1), $\rightarrow$ R1
,2	M K104	F1 $\rightarrow$ K104
,3	R IV,6	return-address V,I $\rightarrow$ R2
,4	+ B9	special word $\rightarrow$ R1
,5	U IBM IN, i.e. to K1	transfer to "IBM IN" subroutine
,6	-- V,1	(this is a convenient place to store the return-address)
B9	00003 00...P4	(this is the special word which indicates that three quantities are to be read, converted, and stored, beginning at P4.)

Orders IV, 1, 2, 3 and 4 fulfill the requirements of the "IBM IN" subroutine

VI, 1 + B2	F2, (field word No. 2), ⟶ R1
, 2 M W127	F2 ⟶ W127
, 3 + B3	F3, (field word No. 3) ⟶ R1
, 4 M W128	F3 ⟶ W128
, 5 R VI,8	return-address VII,1 ⟶ R2
, 6 + B10	special word ⟶ R1
, 7 U IBM OUT, i.e. to W1	transfer to "IBM OUT" subroutine
, 8 --VII,1	(this is a convenient place to store the return-address)
B10 00003 / 00...P	(this is the special word which indicates that three quantities are to be reconverted and recorded, beginning from the quantity stored at P7.)

Orders VI, 1,2,3,4,5, and 6 fulfill the requirements of the "IBM OUT" subroutine.

VIII,1 + B4	F4, (field word No. 4), ⟶ R1
,2 M W127	F4 ⟶ W127
,3 + B5	F5, (field word No. 5), ⟶ R1
,4 M W128	F5 ⟶ W128
,5 R VIII,8	return-address IX,1 ⟶ R2
,6 + B11	special word ⟶ R1
,7U IBM OUT, i.e. to W1	transfer to "IBM OUT" subroutine
,8 -- IX,1	(this is a convenient place to store the return-address)
B11 00003 / 00...P1	(this is the special word which indicates that three quantities are to be reconverted and recorded, beginning from the quantity stored at P1.

Orders VIII,1,2,3,4,5 and 6 fulfill the requirements of the "IBM OUT" subroutine.

X, 1 + B6	$10^{-2}(1) \rightarrow$ R1
, 2 (+) B8	$10^{-2}(j + 1) \rightarrow$ R1

, 3 M B8	$10^{-2}(j + 1) \longrightarrow 10^{-2}(j) \longrightarrow$ B8
, 4 U II,1	transfer to II,1

The orders of Box X increase "j" by one and then control is directed to Box II to repeat the operations for the "set" corresponding to "j".

Before one can write the final code for the orders corresponding to the boxes which were coded, one must decide where the subroutines are to be stored. Once the subroutines have been assigned to specific positions of the memory, the addresses W1, W2, W3, W127, W128, K1 and K104, are defined explicitly. The essential statistics concerning the subroutines are listed at the end of this chapter.

<u>Exercises</u>: (refer to the illustrative problem)

1. Assume that one had 300 sets of data, (instead of 30), and that each set was composed of 80 sub-sets, P_i, (instead of 50).
   a. How many columns of a card would be necessary to represent the combined identification?
   b. How should one scale "j", the number of the set and "i", the serial increment?
   c. How should "j" and "i" be scaled if "j" is to be recorded in columns 38 through 40 and "i" is to be recorded in columns 35 through 36?
2. Suppose that x_i was represented on each input card in columns 44 through 50, (instead of columns 11 through 17). What changes would be required to adapt this format to the problem?
3. Consider the individual requirements of the subroutines. Which, if any, should be fulfilled "outside" of the loops?

Lengths of IBM IN and IBM OUT Subroutines: See also p. 287.

IBM IN	105 words	(4000 - 4068)
IBM OUT	128 words	(4000 - 407L)

Temporary Storage:

IBM IN	010 - 027 and 034 - 03L
IBM OUT	010 - 033 and 03F - 03L

Location of field words:

IBM IN	words 104 and 105	(4067 - 4068)
IBM OUT	words 127 and 128	(407F - 407L)

Location of code and serial identification numbers:

address of W2 is word 2 (4001) }
address of W3 is word 3 (4002) } IBM OUT only

Accuracy:

IBM IN — Maximum error is $2^{-39}(1)$, last bit is always a 1 except that zero is exact.

IBM OUT — Maximum error is $.5 \times 10^{-K} + 2^{-38}$, where K is the number of decimal digits printed.

Initially, the field words are set at KKKKK KKKNO which corresponds to eight fields of ten decimal digits each.

(W2) is initially 0.

(W3) is initially $10^{-10}(1)$.

CHAPTER XI

Floating-Point Routine

As we know from Chapter I, machines such as ORDVAC where the radix point is "fixed" within (or at either end of) the number representation, are called "fixed-point" machines. In ORDVAC the binary-point is fixed between the first and second bits of the 40 bit number representation.

x.xxxxxxxx xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx

Fixed binary-point.

Figure 1

Since the first bit in the number representation indicates the sign of the number, the statement that the binary-point is fixed (as shown in Figure 1) is equivalent to the assertion that numbers be less than one in absolute value. Since most computations involve numbers which do not satisfy this number size condition, the concept of scaling was introduced and discussed in Chapter IV. A number size analysis of each quantity that is to be used in computations, and the ensuing coding details that are necessary to maintain the number size condition can be and often are laborious. For some problems the number size analysis may be practically impossible.

The purpose of this chapter is to introduce the concept of "floating-point". The term "floating-point" is indicative of the fact that the radix point in the number representation may vary (or float), rather than remain fixed. In particular, we shall discuss a "Floating-Binary Routine", which is designed to eliminate the labor that is necessary to maintain the number size condition. Hereafter, we shall refer to the "Floating-Binary Routine" as the "OFB" (one-address floating-binary).

In coding for a fixed-point machine, a coder, being required to maintain the number size condition, keeps a record of the scale factor* associated with a corresponding variable. If for example, the variable x

*Unless stated otherwise, scale factor will always imply the quantity, $2^{\pm n}$, $n = 1,2,3...$, as explained in Chapter IV.

is scaled by 2^{-8}, each reference to the scaled variable is recorded (or listed) as $2^{-8}(x)$. In general, in coding for fixed-point machines, the record of scale factors is an external, *fixed* record. The record of scale factors is external in that it is not stored in the machine; the record is fixed in that the scale factors generally remain fixed as determined by a number size analysis. In contrast, in coding or floating-point machines (or routines) the record of scale factors is an *internal*, *variable* record. The record of scale factors is internal in that the record of scale factors is actually stored in the machine; the record is variable in that the scale factor associated with a corresponding quantity need not remain fixed but is allowed to vary (or float) as the quantity varies. The OFB maintains the number size condition by adjusting the record of scale factors as required.

Ordvac Floating-Binary Routine

Machine representation of a floating-binary number:

A machine floating-binary number is represented by 40 bits as shown in Figure 2 below.

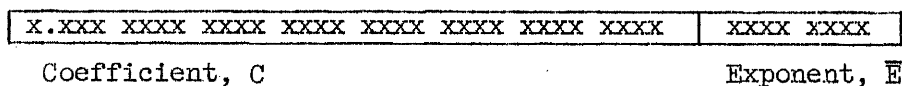


Figure 2

The first 32 bits represent what is called the "coefficient"; the last 8 bits represent what is called the "exponent". Thus, the machine representation of a floating binary number is composed of two parts, namely:

1. coefficient, C
2. exponent, $\bar{E}$.

The representation shown in Figure 2 represents the number N,

where $N = C.2^E$, and $E = \bar{E} - 128 = \bar{E} - 2^7$; $\bar{E} = E + 128 = E + 2^7$.

The coefficient, C, is a signed fixed-binary number, the binary point being fixed as shown in Figure 2. The first bit of the coefficient represents the sign of the floating binary number. Negative coefficients are represented as complements with respect to "2".

The exponent, $\bar{E}$, defines the scale factor associated with N, i.e. 2^{-E} N=C. The use of $\bar{E}$, rather than E, in the floating binary representation is a convenient means of "affixing" an algebraic sign to E. Since $E = \bar{E} - 128$,

E is negative if $\bar{E} < 128$;

E is positive if $\bar{E} > 128$;

E is zero if $\bar{E} = 128$.

Further, since 8 bits are used to represent $\bar{E}$,

$0 \leq \bar{E} \leq 255$, or $-128 \leq E \leq 127$.

E can be interpreted as defining the direction and magnitude that the fixed point should be shifted in order to obtain the location of the binary point in the unscaled representation of the number.

In order to define the number N uniquely in the floating binary form $C \cdot 2^{\bar{E}}$, the condition that $|C| \geq 1/2$, $N \neq 0$, is imposed. This unique form of representation is often referred to as the "normalized" form.

Examples:

Decimal Number	Machine Representation of Floating Binary Numbers								
	Coefficient, C							Exponent, $\bar{E}$	
0	0.000	0000	0000	0000	0000	0000	0000	0000	0000 0000
3.5	0.111	0000	0000	0000	0000	0000	0000	0000	1000 0010
-3.5	1.001	0000	0000	0000	0000	0000	0000	0000	1000 0010
.375	0.110	0000	0000	0000	0000	0000	0000	0000	0111 1111
-.375	1.010	0000	0000	0000	0000	0000	0000	0000	0111 1111
610.1	0.100	1100	0100	0011	0011	0011	0011	0011	1000 1010
-610.	1.011	0011	1100	0000	0000	0000	0000	0000	1000 1010

Exercises:

1) Write the floating-binary number representations for the following decimal numbers: 0.0625; -0.0625; 17.3; 1025.5.

2) What decimal numbers are represented by the following floating-binary numbers?

   a) 0100 0000 1000 0000 0000 0000 0000 0000 1000 0111

   b) 1011 1111 1000 0000 0000 0000 0000 0000 1000 0111

c) 0111 0000 0000 0000 0000 0000 0000 0000 1000 0000

d) 1101 0000 0000 0000 0000 0000 0000 0000 0111 1100

e) 0100 0000 0001 1000 0000 0000 0000 0000 1000 1011

Representation of floating-decimal numbers.

Since it is convenient for a coder to represent numbers in decimal form, a "floating-decimal" form is defined similar to the floating-binary form. The floating-decimal form is composed of two parts, a signed coefficient, C, and a signed exponent, E. The coefficient is represented by a sign and a number of decimal digits; the exponent is represented by a sign and generally two decimal digits. The general form of a floating-decimal number, $N = C \cdot 10^E$, is

$$\pm\, d_1\, d_2\, d_3 \,\ldots\, d_n \qquad \pm\, e_1\, e_2$$

Coefficient, C   Exponent, E

There are two essential differences between the floating-binary and floating-decimal forms:

1. The number of digits in the floating-decimal coefficient is not fixed as is the number of bits in the floating binary coefficient;
2. The floating-binary form uses $\overline{E}$ to represent the signed exponent, whereas the floating-decimal exponent is represented by a sign and two digits, $e_1\, e_2$.

In general, a coder need not represent numbers in floating-binary form. Special subroutines for use with the OFB have been designed to enable the coder to transfer floating-decimal data from (to) IBM cards to (from) the core memory. The requirements for using these subroutines are similar to the requirements for using the fixed-point "IBM IN" and "IBM OUT" subroutines. (See Chapter X.) The use of the OFB "IBM IN" and "IBM OUT" subroutines will be illustrated in an example at the end of this chapter.

Floating-Binary Routine, OFB.

The OFB is employed in a manner somewhat similar to general subroutines. That is, having met the requirements of the OFB, control is then directed to the OFB. However, the specific requirements for using the OFB differ from the conventional requirements for using subroutines. Floating-binary

operations are not interpreted directly by the control unit of the machine. The defined floating-binary operations are interpreted by the "control sequence" of the OFB. This control sequence is often referred to as a "monitor". The control sequence of the OFB is simply a sequence of instructions designed to interpret pseudo instructions. Having interpreted a pseudo instruction, the control sequence of the OFB then directs control to an appropriate OFB sequence which is designed specifically to carry out the operation associated with the corresponding pseudo instruction. After the appropriate OFB sequence carries out the defined operation, control is directed to the control sequence of the OFB and the next pseudo instruction is interpreted and carried out. The OFB continues operations in this manner until it encounters a pseudo instruction which indicates that control is to be directed to the machine control unit. The machine control unit then carries out the normal machine operations until an instruction is encountered which directs control to the OFB. Thus, the coder actually has at his command the equivalent of two "machines": one, a fixed-point, the other, a floating-point, (OFB). When a coder desires to have operations carried out in fixed point, he writes an instruction which is carried out by the machine control unit; when the coder desires to have operations carried out in floating point, he writes a pseudo instruction which is carried out by the control unit of the OFB. Actually, the OFB is designed to carry out both fixed and floating-point operations. The OFB is designed so as to minimize the number of transfers from and to the OFB.

OFB Arithmetic Operations.

The OFB arithmetic operations are designed to be similar to the fixed point operations. The OFB has a central accumulator, (similar to the arithmetic register, R1), which we shall refer to as "A". Since floating-binary numbers are composed of two parts, the OFB central accumulator, A, is composed of two registers, Ac and Ae, which are used to hold respectively the coefficient and the exponent of the floating-binary number. To obtain the sum, difference, product, or ratio of two floating-binary numbers, N_1 and N_2, the OFB first isolates the coefficient and exponent of each of the two numbers. That is, the component parts of N_1, C_1 and $\overline{E}_1$, are isolated. Similarly, the component parts of N_2, C_2 and $\overline{E}_2$, are isolated.

The OFB then forms the sum, difference, product or ratio of the floating-binary numbers, N_1 and N_2, by performing "fixed-point" operations on the components, C_1, $\overline{E}_1$, C_2, $\overline{E}_2$. For example, the product of N_1 and N_2 is obtained by performing the fixed-point operations to carry out the following steps:

1. isolate C_1 , $\overline{E}_1$, C_2 , $\overline{E}_2$;
2. form $C_1 \cdot C_2 = C_3$, (retain the 32 most significant bits of this product);
3. form $\overline{E}_3 = \overline{E}_1 + \overline{E}_2 - 128$, (8 bits);
4. combine C_3, (32 bits) and $\overline{E}_3$, (8 bits) ; call this N_3.

N_3 is the desired floating binary product.

Example:

Form the floating-binary product, $N_1 \cdot N_2 = N_3$, where

$N_1 = 7.5$

$N_2 = - 11.125.$

Expressed in floating-binary form,

N_1 = 0111 1000 0000 0000 0000 0000 0000 0000 1000 0011 ;

N_2 = 1010 0111 0000 0000 0000 0000 0000 0000 1000 0100 .

1. C_1 = 0111 1000 0000 0000 0000 0000 0000 0000 , $\overline{E}_1$ = 1000 0011 ;
   C_2 = 1010 0111 0000 0000 0000 0000 0000 0000 , $\overline{E}_2$ = 1000 0100 ;
2. $C_1 \cdot C_2 = C_3$ = 1010 1100 1001 0000 0000 0000 0000 0000 ;
3. $\overline{E}_3 = \overline{E}_1 + \overline{E}_2 - 128$ = 1000 0111 ;
4. N_3 = 1010 1100 1001 0000 0000 0000 0000 0000 1000 0111.

The decimal equivalent of N_3 is - 83.4375, the desired product.

The OFB actually performs more operations than are implied in steps "1" through "4". For example, before the product, N_3, is stored in a designated memory position, the OFB normalizes the product. During the process in which the product is being normalized, the OFB determines if the resulting exponent is in the tolerable range. If $E < 0$, the product is defined as zero. If $E > 255$, an address and an order pair are printed on the teleprinter, then the OFB directs control to a stop order to indicate that the tolerable bound of the exponent has been exceeded. The address that is printed on the teleprinter is the address of the order pair that is printed.

This information tells the coder that one of the two orders that were printed caused the exponent bound to be exceeded.

The operations for forming the sum, difference, or ratio of two floating-binary numbers are similar to the operations outlined for forming a product. The reader should observe that it requires many fixed-point operations to interpret and perform the floating-point operations. Consequently, the time required to perform the floating-point operations is approximately twelve to fifteen times as long as it takes to perform the corresponding fixed-point operations. The OFB performs approximately 650 floating-binary arithmetic operations per second. The above figures are estimates, the time required to perform a particular floating-binary operation varies, depending on the particular operation and the quantities that are used in the particular operation.

To illustrate the coding for OFB operations, consider first the orders that are required to form the sum of two fixed-point numbers, N_1 and N_2, which are stored respectively at P1 and P2. Assume that the sum, $N_1+N_2=N_3$, is to be stored in P3.

For fixed-point coding, one writes:

	Seq.	Order	Description
	a,1	+ P1	$N_1 \longrightarrow$ R1
(a)	a,2	(+) P2	$N_1 + N_2 = N_3 \longrightarrow$ R1
	a,3	M P3	$N_3 \longrightarrow$ P3

The coding to form and store the sum of two floating-binary numbers, N_1 and N_2, which are stored respectively in P1 and P2 is as follows:

	Seq.	Order	Description
	b,1	f + P1	$N_1 \longrightarrow$ A
(b)	b,2	f(+) P2	$N_1 + N_2 = N_3 \longrightarrow$ A
	b,3	f M P3	$N_3 \longrightarrow$ P3

The essential differences in "a", the fixed-point sequence, and "b", the floating binary sequence, are:

1. sequence "a" is interpreted and carried out by the machine control unit; sequence "b" is interpreted and carried out by the OFB.
2. in sequence "a", the numbers N_1 and N_2 are fixed-point numbers; in sequence "b", the numbers N_1 and N_2 are floating-binary numbers.

The "f +", "f (+)", and "f M" are the floating-binary orders which instruct the OFB to form and store the sum. Note that under the description in sequence "b", the result of each operation is recorded in the description column similar to the description recorded in sequence "a". The order "f + P_1" duplicates N_1 in the OFB accumulator, A. The order "f (+) P2" forms the sum, $N_1 + N_2 = N_3$, and stores this sum in the OFB accumulator, A. The order "f M P3" stores the sum, N_3, in P3. In connection with the "fM" order, the contents of the accumulator is normalized before it is stored in P3.

In order to have the OFB interpret and carry out the instructions of sequence "b", two requirements must be fulfilled:

(1) the address of the first instruction, "b,1", to be interpreted and carried out by the OFB must be specified;

(2) control must be directed to the OFB.

Since it is convenient, (and advantageous), to precede the first floating binary instruction by the orders which fulfill requirements (1) and (2), the following method is used:

Assume that orders "b,1" and "b,2" are stored in memory position K2. The orders which satisfy requirements (1) and (2) are:

Address	Seq.	Order
K1	r,1	+ K1
	r,2	U OFB

where K1 is the address of the order pair, "r,1" and "r,2", and K2 = K1 + 1. The first instruction to be interpreted by the OFB is defined to be the left order stored at K1 + 1 = K2. The advantage of this method is that it does not require memory space to store the address, K2. Thus, assuming "machine" control, the orders to fulfill the OFB requirements and the orders to form the sum in sequence "b" are as follows:

Address	Seq.	Order	Description
K1	r,1	+ K1	fulfill requirements (1) and (2)
	r,2	U OFB	
K2	b,1	f + P1	form and store the sum in P3.
	b,2	f (+) P2	
K3	b,3	f M P3	

The OFB is designed to interpret and perform 46 orders, (pseudo-instructions). The following table describes each of these orders. For reference purposes the orders are numbered from 1 through 46. The description of each order includes;

a. a verbal name of the order;

b. the preliminary (symbolic) form of the order;

c. the final (sexadecimal) form of the order;

d. a description of what the order is designed to do.

Unless otherwise stated, "P" in the preliminary form of the order represents the core memory address of the quantity to be used in the operation. The three dots "...", in the final form of the order represent the three sexadecimal characters which define "P" explicitly. Further, unless otherwise stated, it is assumed that the contents of "P" is a floating-binary number. In the description, "A" represents the OFB accumulator; "Ac" and "Ae" represent respectively the coefficient and exponent registers of the OFB accumulator, "A".

Table of OFB Orders

Arithmetic orders 1 through 10.

1. Floating clear add: f + P; 00...

   The contents of P is recorded in A; the coefficient is recorded in Ac, the exponent is recorded in Ae.

2. Floating clear subtract: f - P; 24...

   The negative of the contents of P is recorded in A; the negative coefficient is recorded in Ac; the exponent is recorded in Ae.

3. Floating add hold: f (+) P ; N4...

   The contents of P is added to the contents of A, the sum is recorded in A; the coefficient of the sum is recorded in Ac, the exponent of the sum is recorded in Ae.

4. Floating subtract hold: f (-); 04...

   The contents of P is subtracted from the contents of A, the difference is recorded in A; the coefficient of the difference is recorded in Ac, the exponent is recorded in Ae.

5. Floating multiply: f x P ; 68...

The contents of A is multiplied by the contents of P, the product is recorded in A; the coefficient is in Ac, the exponent is in Ae.

6. Floating divide: f ÷ P ; 78...

The contents of A is divided by the contents of P, the quotient is recorded in A; the coefficient is recorded in Ac, the exponent is in Ae.

7. Floating absolute value: f |+| P ; F4...

The absolute value of the contents of P is recorded in A, the coefficient is recorded in Ac; the exponent is in Ae.

8. Floating negative absolute value: f |-| P ; 64...

The negative of the absolute value of the contents of P is recorded in A, the coefficient is recorded in Ac; the exponent is in Ae.

9. Floating divide by 2^n: f$\longrightarrow$n; 08.$\overset{n}{\cdot\cdot}$.

The contents of A is divided by 2^n; i.e. the exponent in Ae is diminished by "n". The coefficient in Ac is unchanged.

10. Floating multiply by 2^n: f$\longleftarrow$ n; 18.$\overset{n}{\cdot\cdot}$.

The contents of A is multiplied by 2^n; i.e., the exponent in Ae is augmented by "n". The coefficient in Ac remains unchanged.

11. Floating store: f M P; 10...

The contents of A is normalized and stored in P. The normalized coefficient is retained in Ac, the adjusted exponent is retained in Ae.

Orders 12, 13, 14, 15, and the negative "branches" of orders 18 and 19 are transfer of control orders which direct the OFB to continue interpreting OFB orders.

Orders 16, 17, and the positive "branches" of orders 18 and 19 are transfer of control orders which instruct the OFB to direct control to the "machine" control; i.e., OFB control is relinquished.

12. Transfer to a left order: f U P; NN...

Control is directed to the left order of P. The contents of A is unchanged.

13. Transfer to a right order: f U' P; 1N...

Control is directed to the right order of P. The contents of A is unchanged.

14. Compare, transfer to the left order of P if (Ac) $\geq$ 0: f C P; 2N...
If the contents of Ac is negative, control is directed to the next order in sequence. If the contents of Ac $\geq$ 0, control is directed to the left order of P. The contents of A remains unchanged.

15. Compare, transfer to the right order of P if (Ac) $\geq$ 0: f C' P; 4 N...
If the contents of Ac is negative, control is directed to the next order in sequence. If the contents of Ac $\geq$ 0, control is directed to the right order of P. The contents of A remains unchanged.

16. Transfer to a left order: U P; NO...
Control is directed out of OFB to the left order of P.. The contents of A is unchanged.

17. Transfer to a right order: U'P; 14...
Control is directed out of OFB to the right order of P. The contents of A is unchanged.

18. Compare, transfer to the left order of P if (Ac) $\geq$ 0: CP; 20...
If the contents of Ac is negative, control is directed to the next order in sequence. If the contents of Ac $\geq$ 0, control is directed out of OFB to the left order of P. The contents of A is unchanged.

19. Compare, transfer to the right order of P if (Ac) $\geq$ 0: C'P; 40...
If the contents of Ac is negative, control is directed to the next order in sequence. If the contents of Ac $\geq$ 0, control is directed out of OFB to the right order of P. The contents of A is unchanged.

20. Stop order: Stop; LN ...
The OFB directs the machine to stop operations.

21. Dummy order: DN; F8...
This order is used for delay purposes when desired. It does not affect the contents of any of the registers or memory positions.

Conversion and Reconversion orders, 22 through 25.

22. Convert: (a number which is scaled by a power of ten to a number which is scaled by a power of two): C + P ; J8...
This order assumes that the number to be converted is stored in memory positions P and P + 1. It assumes that the number in P is

scaled by a power of ten. It is further assumed that the exponent of the power of ten (by which the number in P is scaled) is stored in P + 1, scaled by 10^{-2}. The OFB records the equivalent floating-binary number in A. In essence, this order transforms a binary number which is scaled by a power of ten to a binary number which is scaled by a power of two, a floating-binary number.

23. Reconvert: (a number which is scaled by a power of two, to a number which is scaled by a power of ten): RM P; JN. . .
This order does the reverse of Order 22. That is, it is assumed that the floating-binary number to be reconverted is in A. The OFB transforms this number to a number which is scaled by a power of ten. The coefficient is stored in P, the associated exponent of the power of ten (scaled by 10^{-2}) is stored in P + 1. In essence this order transforms a floating-binary number to an equivalent number which is scaled by a power of ten, a floating-decimal number.

24. Convert: (a fixed-binary number to a floating-binary number): C' + P; 98. . .
The OFB assumes that a fixed-binary number, which is less than one in absolute value, is stored in P. The OFB transforms the fixed-binary number to a floating-binary number; the floating-binary number is recorded in A. If the number in P does not represent a number which is less than one in absolute value, one can adjust the floating-binary which is produced by applying a floating multiply or divide by 2^n.
Example: Assume that the fixed binary number, x, scaled by 2^{-5}, is in P; i.e., 2^{-5} (x) is in P.
to obtain x in floating-binary, one writes:

,1	C' + P	$2^{-5}(x) \rightarrow A$
,2	$f \leftarrow 5$	$2^5 . 2^{-5}(x) = x \rightarrow A$

25. Reconvert: (a floating-binary number to a fixed-binary number): R'M P; 9N. . .
The OFB assumes that a floating-binary number, which is less than one in absolute value, is in A. The OFB stores the equivalent fixed point number in P. If the floating-binary number which is in

A is not less than one in absolute value, one can first apply an appropriate scale factor which will make the quantity less than one in absolute value.

Example: Assume that the floating-binary number x is in A. Assume that $|x| < 1000$, and that the floating-binary number 1000 is in P.

To obtain the fixed point number $10^{-3}(x)$ one writes:

,1 f ÷ P $10^{-3}(x)$ in floating-binary form → A;

,2 Rv' P $10^{-3}(x)$ in fixed point form → P.

26. Transfer to a subroutine: U* P ; SN. . .

The OFB subroutines assume that the normalized floating-binary argument is in A when control is directed to the first order of the subroutine. P represents the address of the first order of the subroutine. The result of the subroutine is recorded in A. After the subroutine is executed, the OFB executes the left order following the U* P order in sequence. For example, if the U* P order is in memory position X, then the order that the OFB commences with after the subroutine is executed is the left order in memory position X+1. (Hence, space wise, it is advantageous if the U* order is a right order).

Modification of addresses, orders 27, 28 and 29.

27. Modify a left address: $2^{-19}(+)$ P ; 84. . .

The address of the left order of P is advanced by one; i.e., $2^{-19}(1)$ is added to the contents of P. The result is recorded in P and in Ac.

28. Modify a right address: 2^{-39} (+) P ; 88. . .

The address of the right order of P is advanced by one; i.e., $2^{-39}(1)$ is added to the contents of P. The result is recorded in P and in Ac.

29. Modify both addresses: $(2^{-19} + 2^{-39})$ (+) P ; 8N. . .

The addresses of both orders in P are each advanced by one; i.e., $2^{-19}(1) + 2^{-39}(1)$ is added to the contents of P. The result is recorded in P and in Ac.

30. Polynomial evaluation: Polyn ; N8 . $\overset{W}{.}$.

This order instructs the OFB to evaluate the polynomial,

$y = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$.

W represents the address of a word which specifies:

1. "n", the degree of the polynomial;
2. A0, the address of the coefficient a_0;
3. a special address for OFB control, 067.

$\left.\right\}$ $(W) = 2^{-5}(n) \overset{A0}{---} 00067$

The OFB assumes the following standard conditions:

a. the argument, x, $|x| < 1$, is in memory position 016;
b. the coefficients are stored in consecutive positions, $A_0, A_1, \ldots A_n$
c. the result, y, and each partial sum is less than one in absolute value;

The floating-binary result, y, is recorded in A and control is directed to the next OFB order in sequence.

In essence, this order instructs the OFB to evaluate a polynomial in fixed point; then, assuming the result, y, is less than one in absolute value, the floating binary y is recorded in A.

31. Store a floating-binary zero: oM P ; 30...

Zero is recorded in A and in P.

32. Store the contents of the accumulator: sM P ; 70...

The contents of Ac and Ae are combined to form a 40 bit word, then this 40 bit word is recorded in P. This order differs from order 11, fM P, in that the contents of A is not normalized before it is recorded in P.

Orders 33 through 46 are similar to the ordinary machine orders. The description given in the Appendix apply to these orders except that the OFB refers to and records results in Ac rather than R1 or R2. These orders are included in the OFB group of defined operations to minimize the number of transfers of control from and to the OFB.

33. + P ; K8... ; (P) ⟶ Ac.
34. - P ; S8... ; - (P) ⟶ Ac.
35. (+) P ; 3N... ; (Ac) + (P) ⟶ Ac.
36. (-) P ; 5N... ; (Ac) - (P) ⟶ Ac.
37. x P ; 6N... ; (Ac) · (P) ⟶ Ac.
38. ÷ P ; 7N... ; (Ac) ÷ (P) ⟶ Ac.

39. M P ; 60... ; (Ac) $\longrightarrow$ P.

40. E P ; 90... ; the 12 bits e_8 through e_{19} of Ac replace the 12 corresponding bits of P.

41. E'P ; 50... ; the 12 bits e_{28} through e_{39} of Ac replace the 12 corresponding bits of P.

42. 1M P ; S0 ... ; $2^{-1}(1) \longrightarrow$ Ac and $2^{-1}(1) \longrightarrow$ P.

43. $\longleftarrow$ n ; 58.$\overset{n}{\cdots}$; 2^{n}(Ac) $\longrightarrow$ Ac.

44. $\longrightarrow$ n ; 48.$\overset{n}{\cdots}$; 2^{-n}(Ac) $\longrightarrow$ Ac.

45. $\xrightarrow{1}$ n ; 28.$\overset{n}{\cdots}$; $2^{-(n + 1)}(1) \longrightarrow$ Ac.

46. ←S— ; 74... ; (Ac) is shifted one place to the left through the sign position; i.e. $e_1 \longrightarrow e_0$, $e_2 \longrightarrow e_1$, $e_3 \longrightarrow e_2$, ..., $e_{39} \longrightarrow e_{38}$, $0 \longrightarrow e_{39}$.

Note that the floating point sexadecimal order types are the same as the sexadecimal order types defined for machine interpretation. The only exception is the order type for the "f + P" order, No. 1. The reason for the exception is to have the machine stop operations in the event the machine attempts to interpret the order "f + P" when it was desired to have the OFB interpret this order i.e. the order type for the "f + P" order is defined as "00" when it is to be interpreted by the OFB. If a coder mistakenly directs control such that the machine control attempts to interpret the "00" order, the machine stops operations as this is a machine order which causes the machine to stop operations.

Example:

Given two floating-binary numbers, N_1 and N_2, which are stored respectively in P1 and P2, assume machine control and write the orders to form and store:

the sum, $N_1 + N_2 = N_3$, in P3 ;
the difference, $N_1 - N_2 = N_4$, in P4 ;
the product, $N_1 \cdot N_2 = N_5$, in P5 ;
the quotient, $N_1/N_2 = N_6$, in P6.

Seq.	Order	Description
I,1	+ I,1	fulfill OFB requirements
,2	U OFB	
,3	f+P1	$N_1 \longrightarrow A$
,4	f(+) P2	$N_1 + N_2 = N_3 \longrightarrow A$
,5	fM P3	$N_3 \longrightarrow P3$
,6	f + P1	$N_1 \longrightarrow A$
,7	f(-) P2	$N_1 - N_2 = N_4 \longrightarrow A$
,8	fM P4	$N_4 \longrightarrow P4$
,9	f+P1	$N_1 \longrightarrow A$
,10	fx P2	$N_1 \cdot N_2 = N_5 \longrightarrow A$
,11	fM P5	$N_5 \longrightarrow P5$
,12	f+P1	$N_1 \longrightarrow A$
,13	f ÷ P2	$N_1/N_2 = N_6 \longrightarrow A$
,14	fM P6	$N_6 \longrightarrow P6$

The above example is given merely to illustrate the basic floating-binary arithmetic operations; a more general example is given at the end of this chapter.

The OFB is composed of a group of major sequences . . , the CONTROL SEQUENCE, an ARITHMETIC SEQUENCE, a FIXED POINT SEQUENCE, etc. Each sequence is dependent on the CONTROL SEQUENCE but most are independent of each other. Since each sequence is dependent on the CONTROL SEQUENCE, the CONTROL SEQUENCE is coded for the fixed area 040 through OSL. The OFB sequences, other than the CONTROL SEQUENCE, may be stored anywhere in the memory similar to general subroutines. The OFB is so designed that one need only store those sequences which one intends to employ. The number of

words in each sequence and the defined orders which are carried out in the respective sequences are given in Table 1 below.

OFB Sequence	Number of words	The defined orders in the sequence.
CONTROL	128	The CONTROL SEQUENCE interprets all 46 OFB orders and carries out the following orders: fU, fU', fC, fC', U, U', C, C', Stop, DN, sM, +, -, (+), (-), M.
FLOATING ARITHMETIC	66	f+, f-, f(+), f(-), fX, f÷, f|+|, f|-|, fM, f⟶, f⟵, oM.
CONVERT AND RECONVERT	62	Cv. and Rv.
CONVERT' AND RECONVERT'	11	Cv' and Rv'
SUBROUTINE	13	U*
POLYNOMIAL	11	POLYN.
Fixed Pt. Orders	14	E, E', x, ÷, M, ⟶, ⟵, ←S, 1→, 2^{-19}(+)M, 2^{-39}(+)M, $(2^{-19} + 2^{-39})$ (+)M.

TABLE 1

The OFB subroutines may be stored anywhere in the memory similar to the fixed point subroutines. In general, the OFB subroutines employ various OFB sequences and the temporary positions 010 through 03L. The number

of words in each OFB subroutine, and the OFB sequence or sequences which the respective subroutines employ are given in Table 2 below.

OFB SUBROUTINE	NUMBER OF WORDS	OFB SEQUENCES EMPLOYED
$\sqrt{x}$	31	None
e^x	50	Polyn, C', R', FLOATING ARITHMETIC
sin x and cos x	51	" " " "
sin x is recorded in A and 016; cos x is recorded in 017.		
ln x	41	C' R' FLOATING ARITHMETIC
F IBM IN	139*	Cv.
F IBM OUT (Sent)	211*	Rv.
Matrix Inversion and solution of systems of linear algebraic equations	62	FLOATING ARITHMETIC

(Statistics on other OFB subroutines are given in the local literature.)

TABLE 2

* This number includes two positions reserved for two field words.

The requirements for using the F IBM IN Subroutine are:

a. specify the card format; i.e. store field words immediately following the last word of the subroutine. (See Chapter X.)

A floating decimal datum number can be considered as two distinct numbers, namely coefficient and exponent. The coefficient may be represented by as many as eleven decimal digits, the exponent by two decimal digits. For example, the field word for F IBM IN subroutine, 526282K2S2 specifies the following format:

1st datum number in columns	1 - 7	(5 + 2 = 7)
2nd datum number in columns	8 - 15	(6 + 2 = 8)
3rd datum number in columns	16 - 25	(8 + 2 = 10)
4th datum number in columns	27 - 38	(K + 2 = 12)
5th datum number in columns	39 - 41	(S + 2 = 13)

b. store a special instruction word in A, (the OFB accumulator); this special instruction word has the following form:

n M 10 P , where

"n" expressed in sexadecimal form indicates the number of floating decimal quantities represented on each card;

"M" expressed in sexadecimal form indicates the total number of floating decimal quantities that are to be read, converted to floating binary, and stored;

"P" indicates the address where the first quantity is to be stored. (Successive quantities are stored in successive memory positions.)

c. Transfer to the F IBM IN Subroutine using U^* order.

The requirements for using the sine, cosine, square root, logarithm, and exponential subroutines are:

a. Store the normalized floating-binary argument, x, in the floating-binary accumulator, A.

b. Transfer to the desired subroutine using the U^* order.

When control is directed to the return address, the result of the subroutine is available in the OFB accumulator, A. The sine and cosine subroutine is one exception since this subroutine produces two results; the sin x is available in A and in memory position 016; the cos x is available in memory position 017.

The statistics and requirements for using various floating-binary subroutines are given in the Appendix.

Example employing the OFB and OFB subroutines.

Given: $F(x) = B \sin x - \frac{CD}{B} \cos x + De^{x} - E\sqrt{Bx}$,

where B, C, D, and E are constants, and

$$0 \leq x \leq 2A.$$

Wanted: 1. Compute $F(x_i)$, $i = 0, 1, 2, \ldots, 200$.

$$x_0 = 0,$$

$$\Delta x = \pi/10,$$

$$x_i = x_{i-1} + \Delta x = i\Delta x.$$

2. Print (punch) x_i and $F(x_i)$ on IBM cards, recording three successive x_i's and the three corresponding $F(x_i)$ on each IBM card as follows:

record x_{i-2} in columns 1 through 10;

record $F(x_{i-2})$ in columns 11 through 20;

record x_{i-1} in columns 21 through 30;

record $F(x_{i-1})$ in columns 31 through 40;

record x_i in columns 41 through 50;

record $F(x_i)$ in columns 51 through 60.

A detailed flow chart is shown in Figure 3; the corresponding preliminary code is given in Figure 4.

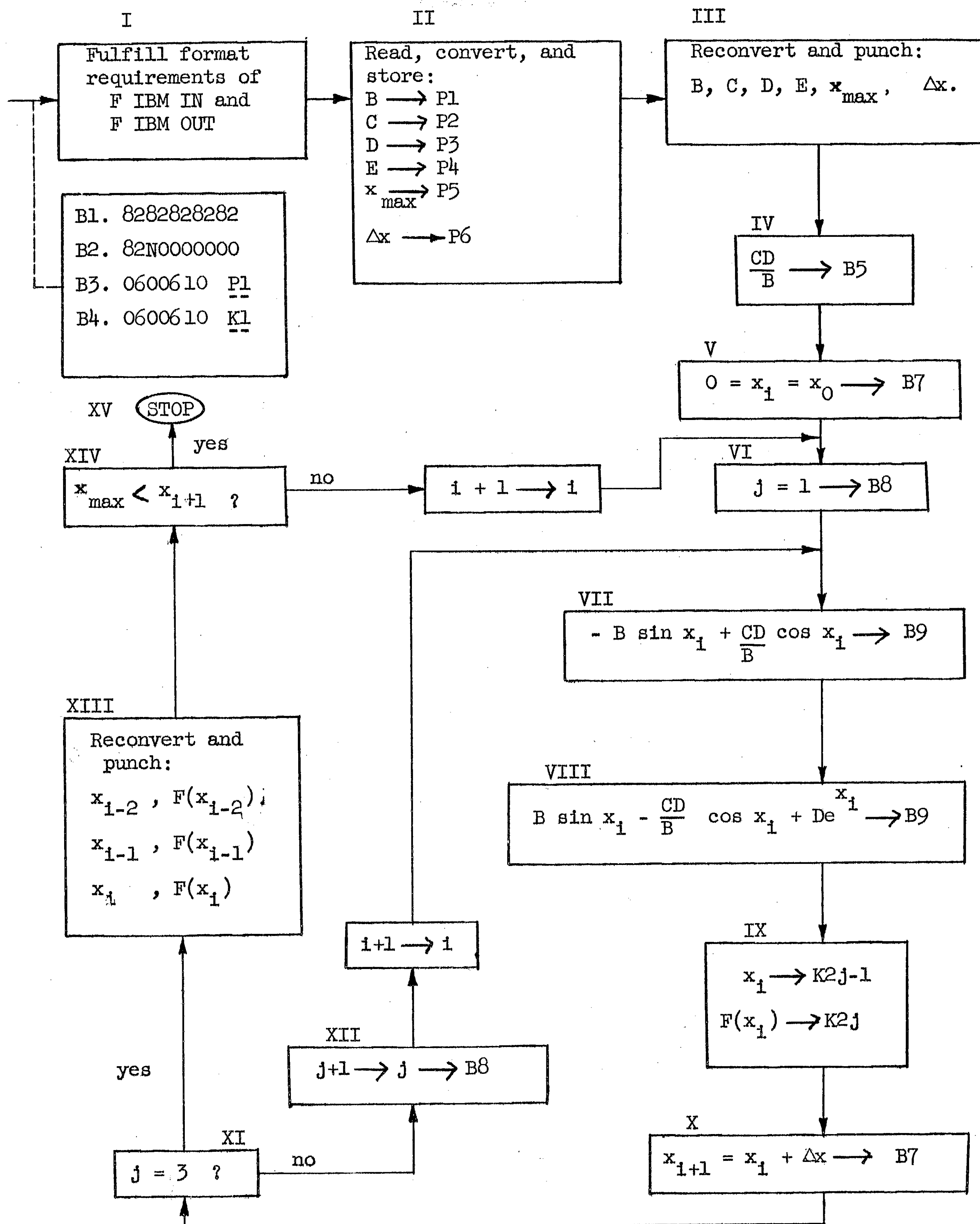


FIGURE 3

SEQUENCE	ORDER	DESCRIPTION		
I, 1	+ B1	FW1 ⟶ R1		
2	M G138	FW1 ⟶ G138	fulfill	F IBM IN
3	M W210	FW1 ⟶ W210	format	F IBM OUT
4	+ B2	FW2 ⟶ R1		
5	M G139	FW2 ⟶ G139	requirements	F IBM IN
6	M W211	FW2 ⟶ W211		F IBM OUT
II, 1	+ II,1			
2	U OFB	Transfer to OFB		
3	f + B3	06 006 00 P1 ⟶ A		
4	U* F IBM IN	Transfer to F IBM IN Sub.		
III, 1	f + B3	06 006 00 P1 ⟶ A		
2	U* F IBM OUT	Transfer to F IBM OUT Sub.		
IV, 1	f + P2	C ⟶ A		
2	f x P3	CD ⟶ A		
3	f ÷ P1	CD/B ⟶ A		
4	fM B5	CD/B ⟶ B5		
5	U' V, 1	Exit OFB Control		
V, 1	oM B7	$x_i = 0$ ⟶ B7		
VI, 1	⟶ 1	j=1, i.e. $2^{-2}(1)$ ⟶ R1		
2	M B8	j=1, i.e. $2^{-2}(1)$ ⟶ B8		
3	+ B4	06 006 00 K_1 ⟶ R1		
4	⟵ 20	00 K1 00000 ⟶ R1		
5	E IX, 2	$K_j = K_1$ ⟶ IX, 2		
6	U VII, 1	Transfer to VII,1		
VII, 1	+ VII,1			
2	U OFB	Transfer to OFB		
3	f + B7	x_i ⟶ A		
4	U* Sin Cos Sub.	sin x_i ⟶ A ; cos x_i ⟶ 017		
5	fx P1	B sin x_i ⟶ A		
6	fM B9	B sin x_i ⟶ B9		
7	f + 017	cos x_i ⟶ A		
8	fx B5	(CD/B) cos x_i ⟶ A		
9	f (-) B9	(-B sin x_i + (CD/B) cos x_i) ⟶ A		
10	fM B9	(-B sin x_i + (CD/B) cos x_i) ⟶ B9		
VIII, 1	f + B7	x_i ⟶ A		
2	U* e^x Sub.	Transfer to OFB e^{x_i} ⟶ A		
3	fx P3	De^{x_i} ⟶ A		
4	f(-) B9	(B sin x_i - (CD/B) cos x_i + De^{x_i}) ⟶ A		
5	fM B9	(B sin x_i - (CD/B) cos x_i + De^{x_i}) ⟶ B9		
IX, 1	f + B7	x_i ⟶ A		
2	fM [K2j-1]	x_i ⟶ K2j-1		
3	fx P1	B x_i ⟶ A		

FIGURE 4

SEQUENCE	ORDER	DESCRIPTION
IX, 4	fM B10	Normalized $Bx_i \longrightarrow$ B10
5	U* $\sqrt{\ }$ Sub	Transfer to $\sqrt{Sub}$; $\sqrt{Bx_i} \longrightarrow$ A
6	fx P4	E $\sqrt{Bx_i} \longrightarrow$ A
7	fM P2	E $\sqrt{Bx_i} \longrightarrow$ P2
8	2^{-19} (+) IX,2	$K(2j - 1) + 1 = K2j \longrightarrow$ A
9	E IX,12	$K2j \longrightarrow$ IX,12
10	f + B9	$B \sin x_i - (CD/B) \cos x_i + De^{x_i} \longrightarrow$ A
11	f (-) P2	$F(x_i) \longrightarrow$ A
12	fM [K2j]	$F(x_i) \longrightarrow$ K2i
X, 1	f + B7	$x_i \longrightarrow$ A
2	f(+) P6	$x_i + \Delta x = x_{i+1} \longrightarrow$ A
3	fM B7	$x_{i+1} \longrightarrow$ B7
4	U' XI, 1	Exit OFB Control
XI, 1	+ B8	$j \longrightarrow$ R1
2	C XII,1	$j = 3$? i.e. is $j < 0$?
3	U XIII,1	Transfer to XIII,1
XIII, 1	+ XIII,1	
2	U OFB	Transfer to OFB
3	f + B4	06 006 00 K1 $\longrightarrow$ A
4	U* F IBM OUT	Transfer to F IBM OUT Sub.
XIV, 1	f + P5	x max $\longrightarrow$ A
2	f (-) B7	x max $-x_{i+1} \longrightarrow$ A
3	C VI,1	$x_{max} < x_{i+1}$?
XV, 1 LN000	STOP	00000 would not work properly.
XII, 1	~~S~~	$j + 1 \longrightarrow$ R1
2	M B8	$j + 1 \longrightarrow j \longrightarrow$ B8
3	+ IX,2	$K2j \longrightarrow$ R1
4	(+) OON	$K2j + 1 = K(2j+1) \longrightarrow$ R1
5	E IX,2	$K2j+1 \longrightarrow$ IX,2
6	U VII,1	Transfer to VII,1

FIGURE 4

Notes on the flow chart and preliminary code.

The words, (B1, B2, B3, B4), shown in the storage box, will be used to fulfill the requirements of the F IBM IN and F IBM OUT subroutines. The words shown in B1 and B2 are the field words, (See Chapter X), which define the format for the six floating-decimal quantities. The signed coefficient of each quantity will be represented in the first eight columns of a field; the signed exponent of each quantity will be represented in the corresponding last two columns of the field.* Observe that the field words, (B1 and B2), will suffice for both the input and the output formats, since:

INPUT
- B will be recorded in columns 1 through 10;
- C will be recorded in columns 11 through 20;
- D will be recorded in columns 21 through 30;
- E will be recorded in columns 31 through 40;
- X_{max} will be recorded in columns 41 through 50;
- Δx will be recorded in columns 51 through 60.

OUTPUT
- x_{i-2} will be recorded in columns 1 through 10;
- $F(x_{i-2})$ will be recorded in columns 11 through 20;
- x_{i-1} will be recorded in columns 21 through 30;
- $F(x_{i-1})$ will be recorded in columns 31 through 40;
- x_i will be recorded in columns 41 through 50;
- $F(x_i)$ will be recorded in columns 51 through 60.

I. As indicated in Box I of the flow chart, the orders corresponding to Box I fulfill the format requirements of the F IBM IN and F IBM OUT subroutines. Note that these orders will be executed under machine control.

II. The orders corresponding to Box II, first direct control to the OFB, then a standard requirement of the F IBM IN subroutine is fulfilled. After the subroutine has read, converted, and stored the input quantities, OFB control is directed to the first order of Box III.

*This assumes that one is using the "double punch" option; i.e., where the sign is recorded over the first column of each field.

III. The orders corresponding to Box III first fulfill a standard requirement of the F IBM OUT subroutine, then control is directed to the subroutine. This "print out" of the input data which was just read is included to provide a means of checking the actual input data that was submitted. It suffices to say that this check costs little, (timewise and spacewise), yet provides an immediate visual check to verify that the input that was submitted was the desired input. After the subroutine has reconverted and punched the desired quantities, control is directed to the first order of Box IV.

IV. The first four orders corresponding to Box IV form and store $\frac{CD}{B}$ in B5. The order, IV,5 instructs the OFB to relinquish control to the machine control. Note that since order V,1 is a right order, the U' order is used to direct control to V,1.

V. The single order corresponding to Box V stores a zero in B7, thus defining the initial x, x_0, to be zero.

VI. Corresponding to Box VI, the initial value of j is defined as "1" and stored in B8. Since it is desired to record three sets of values on each output card, j will be used as a counter to determine whether three sets of values have been computed. Note that since Box IX indicates that the output quantities will be stored in K2j-1 and K2j, it is necessary to extract K1, (corresponding to j = 1), into the address portion of order IX,2. In Box IX, K2j will be computed as K2j-1 + 1. The last order of Box VI directs control to the first order of Box VII. Ordinarily, this last order is not necessary, but in view of the fact that the first orders of Box VII direct control to the OFB, it is necessary that the first order of Box VII be a left order. Note that the orders of Boxes V and VI are executed under machine control.

VII. In Box VII, the sine and cosine subroutine is employed to compute sin x_i and cos x_i. Then two terms of $F(x_i)$ are formed and stored in B9.

VIII. In Box VIII the exponential subroutine is employed to compute e^{x_i}. Then the third term of $F(x_i)$ is formed and added to the first two terms. Again the partial result is stored in B9.

IX. In Box IX, x_i is stored in K2j-1. Next, K2j is computed and stored in the address portion of IX,2 and IX,12. Then the square root subroutine is employed to compute the $\sqrt{Bx_i}$. Finally, the last term of $F(x_i)$ is computed and added to the previous terms. $F(x_i)$ is stored in K_{2j}.

X. In Box X the next x_i, i.e. x_{i+1}, is computed and restored in B7. The 1st order of Box X relinquishes OFB control and machine control commences with the first order of Box XI.

XI. The orders corresponding to Box XI determine whether three sets of values have been computed since the last three sets were recorded, i.e. whether j=3 or is less than 3. If three sets have been computed and stored, control is directed to the first order of Box XII. Observe that since the compare order is executed under machine control, it is necessary that the first order of both Boxes XII and XIII be executed under machine control.

XII. In Box XII the counter, j, is advanced by one and the corresponding K2j-1 is computed and stored in the address portion of order IX,2. Control is then directed to the first order of Box VII to begin the computation of the next $F(x_i)$.

XIII. In Box XIII control is directed to the OFB., then the F IBM OUT subroutine is employed to reconvert and punch the desired quantities.

XIV. The orders of Box XIV determine if the computations are complete, i.e. whether x_{max} is less than x_{i-1}. If x_{Max} is less than x_{i-1}, the computations are complete and control is directed to the stop order in Box XV. If x_{max} is not less than x_{i-1}, the computations are not complete and control is directed to the first order in Box VI. Observe that if the computations are complete, the OFB retains control and the stop order in Box XV is executed under OFB control. However, if the computations are not complete, the OFB relinquishes control and the first order of Box VI is executed under machine control as is desired.

Other options for using the OFB are available in the local literature. For example, an option where each OFB order is stored in a single memory

position, (rather than two OFB orders in a single memory position), is available. This option is so designed to increase the rate at which the OFB orders are performed. Also, a OFB Code Checker for use with the OFB is available. The requirements for using the OFB Code Checker are similar to the requirements for using the fixed-point Code Checker.

Exercises: (Refer to the illustrated example.)

1) Write the final code for the example.
2) Which sequences of the OFB are needed to perform the computations for this example?
3) Code the necessary changes to:
   a. Compute and store the entire table, x_i and $F(x_i)$, before printing. i.e. compute and store the 400 values before printing.
   b. Produce a set of tables, corresponding to sets of B, C, D and E.
   c. Change the increment x from $\frac{\pi}{100}$ to $\frac{\pi}{200}$.
   d. Change the increment x from $\frac{\pi}{100}$ to $\frac{\pi}{50}$.
   e. Change x_{max} from 2π to $\frac{3\pi}{2}$.

To provide a facility for storing floating-point constants, the Transcriber Routine includes the following key word.

800003 10 $\overset{H}{\text{----}}$.

This key word tells the Transcriber Routine to convert the floating decimal numbers, which immediately follow it, to their respective floating-binary equivalents. As a result of this key-word, the Transcriber produces the floating binary equivalents, preceded by the key word, 80003 0 $\overset{H}{\text{----}}$, which tells the Input Routine to store the floating-binary equivalents in consecutive positions beginning at position H. When using this facility, the floating decimal numbers must be represented by twenty-four characters, (24 columns of a card),

as shown below.

Coefficient, K/S } and eleven decimal digits,

Exponent, K/S } 2 digits and 9 zeros

Example: To store the floating-decimal contant, -13.0, in memory position 3N3 one writes

800003 1003N3
S13000 000000
K02000 000000

CHAPTER XII

The Ordvac Magnetic Drum

The Ordvac Magnetic Drum is an auxiliary storage device. The capacity of the drum is ten thousand thirty-two, (10032) words. We consider the drum to be an auxiliary storage device because the words stored on the drum are not directly accessible for arithmetic operations or most of the logical operations. In particular, we cannot perform any operation on the words stored on the drum other than the two logical operations of:

1. Transferring words from the drum to the high-speed (core) memory;
2. Transferring words to the drum from the high-speed (core) memory.

The drum is composed of two hundred nine, (209) "tracks" or "channels". Forty-eight (48) words can be recorded (stored) on each track. Words transferred to or from the drum are transferred in groups of forty-eight (48), i.e. <u>the contents of an entire track</u>.

Each track is identified by a corresponding "track number". Since there are 209 tracks, eight bits suffice to identify each track, i.e.,

the	eight	bits,	0000 0000,	identify	track	number	"zero";
"	"	"	0000 0001,	"	"	"	"one";
"	"	"	0000 0010,	"	"	"	"two";
'	'	'	' '	'	'	'	'
'	'	'	' '	'	'	'	'
"	"	"	1101 0000	"	"	"	"two hundred and eight".

Further, instead of identifying each track by the corresponding eight bits, each track is identified by two sexadecimal characters, the equivalent of the eight bits. Hence, the range of sexadecimal track numbers is " 00 " through " J0 ".

The identification of the two hundred and nine tracks, with their corresponding sexadecimal track numbers, is shown in Figure 1.

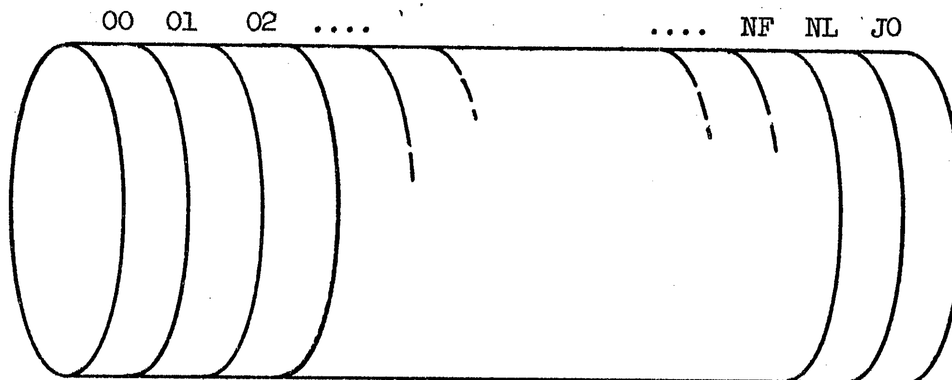
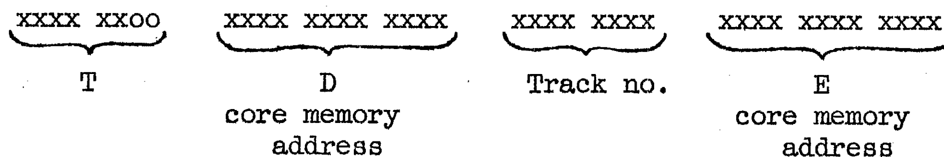


Figure 1

A drum order requires an entire word, (40 bits), and has the following form:



where, reading from left to right,

the first six bits, T, indicates whether the transfer of words is to be made from or to the track of the drum, the next two bits are unused;

T = 1101 01 indicates that the transfer of words is to be made from the core memory to a drum track;

T = 1101 00 indicates that the transfer of words is to be made from a drum track to the core memory;

The next twelve bits identify D, a core memory address, which identifies the first of forty-eight consecutive positions from or to which words are to be transferred;

The next eight bits identify the particular track from or to which forty-eight words will be transferred:

The last twelve bits identify E, a core memory address, which represents the address of the order to be performed after the drum order has been executed. Control is always directed to the left-order of E.

Transfer of words to the drum is effected by way of R2. Transfer of words from the drum is effected by way of R1. Each drum order will erase the previous contents of both R1 and R2.

To represent a drum order by ten sexadecimal characters, the two unused bits are combined with the six bits of T, giving two sexadecimal characters; D is represented by three sexadecimal characters; the track number is represented by two sexadecimal characters; and, E is represented by three sexadecimal characters.

Example 1. J02KS 003LF

This order will transfer the forty-eight words of track "00", (the first track on the drum), to memory positions 2KS through 2JK. Control will then be directed to the left-order of memory position 3LF. The words on track "00" remain intact, the previous contents of positions 2KS through 2JK are replaced by the forty-eight words of track "00".

Example 2. J4820 N2LKO

This order will transfer the forty-eight words from memory position 820 through 84L to track "N2", (track number 194). Control will then be directed to the left-order of memory position LKO. The contents of memory positions 820 through 84L remain intact, the previous contents of track "N2" are replaced by the contents of memory positions 820 through 84L.

A special key word, acceptable by the Transcriber, is designed to allow one to transfer words easily from cards to the drum, (or to transfer words from the drum to cards). We call this key word the "40" type key word. The "40" type key word, when used, must be followed by another word as shown below:

"40" type key word $800004\ 00\overset{n}{---}$

a drum order $J^{0}_{4}\overset{D}{----}\ \overset{q}{--}\ \overset{E}{----}$

This pair or words will always appear as the "last" pair on a binary card. The facilities associated with such pairs of words are:

For drum order type "J0",

Transfer the 48n words from "n" consecutive tracks of the drum, beginning from track "q", to memory positions beginning at D and extending through D + 48n - 1. Then, direct control to the left-order of memory position E.

For drum order type "J4",
Transfer the 48n consecutive words, from memory positions beginning at D and extending through D + 48n - 1, to the "n" consecutive tracks of the drum beginning at track "q" and extending through track "q + n - 1". Then direct control to the left-order of memory position E.

Correspondingly, the binary key word produced by the Transcriber and accepted by the Input Routine is called the "4" type key word. Written in sexadecimal form the drum binary key word is:

80004 00 -n-,

followed by

J_4^0 -D- -q- -E-.

The time required to transfer the forty-eight words of or to a track is approximately eighty (80) millesceconds. Since this transfer time is long as compared to the time required to transfer words within the core memory, it is advisable to arrange words on the drum so as to minimize the number of drum transfer orders in any given routine.

Exercises:

1. Construct an order that will transfer the forty-eight words of track one hundred and ninety-nine to memory positions beginning at position 5KJ. What is the address of the last memory position affected?
2. Construct an order that will transfer forty-eight words, beginning from memory position 903, to track seventy-seven. What is the address of the last word to be transferred?
3. Construct the necessary key words that will enable one to store 4800 binary numbers on tracks 100 through 199. Assume that the 4800 numbers are recorded on IBM cards in standard decimal form.

Tadeusz Leser
TADEUSZ LESER

Michael J. Romanelli
MICHAEL ROMANELLI

APPENDIX

Page

1) ORDVAC List of Orders. 239
2) Detailed Description of ORDVAC Orders 240
3) Decimal Representation of Powers 247
4) ORDVAC Code Checker 248
5) ORDVAC One Address Floating Binary Routine (OFB) 254
6) ORDVAC Floating Point (One Address) Code Checker 261
7) (OFB). Floating Point Square Root 265
8) (OFB). Floating Point Sin-Cos. 266
9) (OFB). Floating Point Exponential 267
10) (OFB). Floating Point Natural Logarithm 268
11) (OFB). Floating Point Arcsinx. 269
12) (OFB). Floating Point Arctangent 270
13) (OFB). Floating Point Arcsin-Cos. 271
14) (OFB). To Solve Normal Equations (SNE) and Matrix Inversion . . 272
15) (OFB). Floating Point IBM Input (IBMC). 273
16) (OFB). Floating Point Arccos 274
17) (OFB). Runge-Kutta-Gill System of Differential Equations . . . 275
18) (Fixed Point). Square Root. 276
19) (Fixed Point). Faster Square Root 277
20) (Fixed Point). Fastest Square Root 278
21) (Fixed Point). Sin and Cos 2 x 279
22) (Fixed Point). Arcsin-cos 280
23) (Fixed Point). Natural Logarithm 281
24) (Fixed Point). Arctangent 282
25) (Fixed Point). Runge-Kutta-Gill System of Differential Equations 284
26) New ORDVAC IBM Card Routine 285
27) Double Speed IBM Output Routine 289
28) Order Pair Routine (OP). 290
29) Printing Address Search 291
30) ORDVAC Drum and Memory IBM Print Out). 292
31) Sexadecimal Print. 294
32) Floating Decimal to Fixed Decimal Routine 295

APPENDIX

1.)

ORDVAC
LIST OF ORDERS

16 December 1955

Symbol	Sexadecimal Form	Symbol	Sexadecimal Form
+	K4	oE	LO
-	24	oE'	70
(+)	N4	M	10
(-)	04	oM	30
|+|	F4	1M	SO
|-|	64	⊕	88
[+]	84	(R)	F8
[-]	44	C	20
A+	KN	C'	40
A-	2N	U	NO
A(+)	NN	U'	14
A(-)	ON	oU	KO
A|+|	FN	oU'	34
A|-|	6N	Zx	FO
R	S4	Zu	OO }
			LN }
÷	78	T	94 028
X	68	P	L4 028
Xu	K8	IBM In	58 000
(X)	N8	IBM OUT	48 000
← n	18	From Drum to core	JO
←0— n	38	To Drum from core	J4
→ n	08	"A" +	8N
—1→ n	28		
←8—	74		
←16—	60		
←8T—	54		
E	90		
E'	50		

2.) DETAILED DESCRIPTION OF ORDVAC ORDERS

In the preliminary form of representation of orders, "P" represents the preliminary address of any one of the 4096 core memory positions. In the sexadecimal form of representation of orders, the three "dots", "...", represent the three sexadecimal characters of address "P". The description, of what the corresponding orders accomplish, includes only those positions and/or those registers involved or affected. That is, if the contents of a position or register (other than R^3) remains unchanged, this fact is omitted in the description. Further, the descriptions are valid only for those results which are less than one in absolute value.

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
1.	+ P	K4...	The contents of P is duplicated in R1. $(P) \longrightarrow R1.$
2.	(+) P	N4...	The contents of P is added to the contents of R1. The sum goes to R1. $(R1) + (P) \longrightarrow R1.$
3.	|+| P	F4..	The absolute value of the contents of P goes to R1. $\lvert(P)\rvert \longrightarrow R1.$
4.	[+] P	84...	The absolute value of the contents of P is added to the contents of R1. The sum goes to R1. $(R1) + \lvert(P)\rvert \longrightarrow R1.$
5.	- P	24...	The negative of the contents of P goes to R1. $-(P) \longrightarrow R1.$
6.	(-) P	04...	The contents of P is subtracted from the contents of R1. The difference goes to R1. $(R1) - (P) \longrightarrow R1.$
7.	|-| P	64...	The negative of the absolute value of the contents of P goes to R1. $-\lvert(P)\rvert \longrightarrow R1.$
8.	[-] P	44...	The absolute value of the contents of P is subtracted from the contents of R1. The difference goes to R1. $(R1) - \lvert(P)\rvert \longrightarrow R1.$

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
9.	÷ P	78...	The contents of R1 is divided by the contents of P. The resulting quotient, (a sign and 39 bits), goes to R2. The least significant bit of the quotient is always a "1". The remainder, shifted left one place, (i.e., twice the remainder), goes to R1. The sign of the remainder is the sign of the dividend, i.e. the sign of (R1). This order is referred to as "rounded division". $(R1) \div (P) \longrightarrow R2.$ 2 x the remainder $\longrightarrow$ R1.
10.	Xu P	K8...	The contents of R2 is multiplied by the contents of P. This order yields a 78 bit product. The sign and 39 most significant bits of the product go to R1; a "zero" (positive) sign and the 39 least significant bits go to R2. This order is referred to as "exact multiplication". $(R2) \cdot (P) \longrightarrow R1, R2.$
11.	X P	68...	This order yields the same results as the "Xu" order except that 2^{-40} is added to the 78 bit product. This order is referred to as "rounded multiplication": $(R2) \cdot (P) + 2^{-40} \longrightarrow R1, R2.$
12.	(X) P	N8...	This order yields the same result as the "Xu" order except that 2^{-39} times the previous contents of R1 is added to the 78 bit product. This order is convenient for multiple precision operations. $(R2) \cdot (P) + 2^{-39} (R1) \longrightarrow R1, R2.$
			In addition to the descriptions given for orders 1 through 12, $(P) \longrightarrow R3.$
13.	R P	S4...	The contents of P is duplicated in R2. $(P) \longrightarrow R2.$
14.	M P	10...	The contents of R1 is duplicated in P. This order is referred to as the "store order". $(R1) \longrightarrow P.$
15.	oM P	30...	The contents of R1 is deleted and the contents of P is deleted. $0 \longrightarrow R1$, and $0 \longrightarrow P.$
16.	1M P	S0	The contents of R1 is replaced by 2^{-1}. The contents of P is replaced by 2^{-1}. $2^{-1} \longrightarrow R1; \ 2^{-1} \longrightarrow P.$

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
17.	E' P	50	The 12 bits representing the address in the right order in P are replaced by the 12 corresponding bits in R1. This order is referred to as the "right extract" order. right address of (R1) ⟶ right address (P).
18	E P	90...	The 12 bits representing the address in the left order in P are replaced by the 12 corresponding bits in R1. This order is referred to as the "left extract" order. left address of (R1) ⟶ left address of (P).
19.	oE' P	70...	First, the contents of R1 is deleted, then the equivalent of the E' order is effected. 0 ⟶ R1, 0 ⟶ right address of (P).
20.	oE P	L0...	First, the contents of R1 is deleted, then the equivalent of the E order is effected. 0 ⟶ R1, 0 ⟶ left address of (P).
21.	C P	20...	If the contents of R1 ≥ 0, then control is directed to the left order of P; if the contents of R1 < 0, then control is directed to the next order in sequence. This order is referred to as a "compare order".
22.	C' P	40...	If the contents of R1 ≥ 0, then control is directed to the right order of P; if the contents of R1 < 0, then control is directed to the next order in sequence.
23.	U P	N0...	This order directs control to the left order of P. This order is referred to as a "transfer" (of control) order.
24.	oU P	K0...	First, the contents of R1 is deleted, then control is directed to the left order of P. 0 ⟶ R1, and control is directed to the left order of P.
25.	U' P	14...	This order directs control to the right order of P.
26.	oU' P	34...	First, the contents of R1 is deleted, then control is directed to the right order of P. 0 ⟶ R1, and control is directed to the right order of P.
27.	Zu ---	{ 00 ... { LN ...	This order causes the machine to stop operations. When started again, the machine begins with the next order in sequence. Notice that no address is required in this order.

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
28.	Z P	FO...	If switch S_2 on the control panel is not in the "down" position, the machine stops operations and when started again, a "secondary transfer" of control operation is effected. If switch S_2 is in the "down" position, the machine does not stop operations, but continues with the "secondary transfer" of control operation.

A "secondary transfer" of control operation is associated with orders number 28 through 35 and order number 42. The order with which a "secondary transfer" is associated can be a left or a right order. If the order is a right order, the machine executes the right order and then control is directed to the left order of P. If the order is a left order, the machine executes this left order and the corresponding right order before control is directed to the left order of P. Further, if the corresponding right order is a transfer of control order, then the transfer of control to the left order of P is not effected.

Order No.	Prelim. Form	Sexadec. Form	Description
29	A + P	KN...	The contents of R2 is duplicated in R1. Then a secondary transfer is effected. $(R2) \longrightarrow R1.$
30.	A(+) P	NN...	The contents of R2 is added to the contents of R1. The sum goes to R1. Then a secondary transfer is effected. $(R1) + (R2) \longrightarrow R1.$
31.	A |+| P	FN	The absolute value of the contents of R2 goes to R1. Then a secondary transfer is effected. $\|(R2)\| \longrightarrow R1.$
32.	A - P	2N...	The negative of the contents of R2 goes to R1. Then a secondary transfer is effected. $-(R2) \longrightarrow R1.$
33.	A (-) P	ON...	The contents of R2 is subtracted from the contents of R1. The difference goes to R1. Then a secondary transfer is effected. $(R1) - (R2) \longrightarrow R1.$
34.	A |-| P	6N	The negative of the absolute value of the contents of R2 goes to R1. Then a secondary transfer is effected $-\|(R2)\| \longrightarrow R1.$

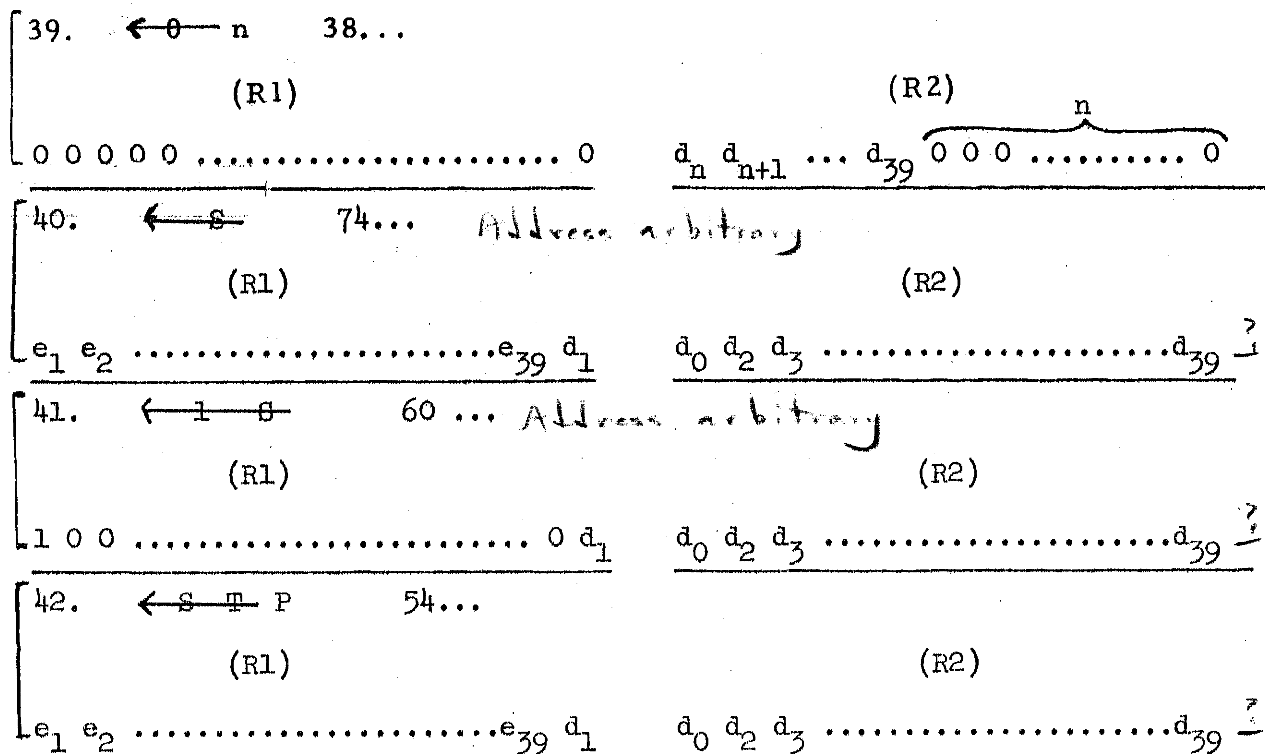
In addition to the descriptions given for orders 29 through 34,

$$(R2) \longrightarrow R^3.$$

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
35.	"A" + P	8N	The contents of R^3 is duplicated in R1. Then a secondary transfer is effected. $(R^3) \longrightarrow R1.$

Orders number 36 through 42 are referred to as "shift" orders. The three "dots", "...", in the sexadecimal representation of the corresponding orders, represent the sexadecimal equivalent of "n", the amount of the shift. If n = 0, the machine stops operations; "n" must be less than or equal to 63; if $n \geq 64$, the amount of shift is n - 64. For most cases, $n \leq 39$ is sufficient. Since the shift orders affect the "bit by bit" contents of R1 and R2, the respective descriptions are given by exhibiting the 40 bits of R1 and the 40 bits of R2. In the table, the contents of R1 and R2 are shown on the line under the preliminary and sexadecimal forms of the order. The results thus shown assume that the original contents of R1 and R2, (i.e. the (R1) and (R2) before the order is executed are respectively:

Contents of R1	Contents of R2
$e_0\ e_1\ e_2 \cdots\cdots e_{39}$	$d_0\ d_1\ d_2 \cdots\cdots d_{39}$
36. $\longrightarrow$ n 08...	
(R1)	(R2)
$\overbrace{e_0\ e_0\ e_0 \cdots e_0}^{n+1}\ e_1 \cdots\cdots e_{39-n}$	$d_0\ e_{40-n} \cdots e_{39}\ d_1\ d_2 \cdots d_{39-n}$
37. $\longleftarrow$ n 18...	
(R1)	(R2)
$e_0\ e_{n+1}\ e_{n+2} \cdots e_{39}\ \overbrace{00 \cdots\cdots 0}^{n}$	$d_n\ d_{n+1} \cdots d_{39}\ e_1\ e_2 \cdots\cdots e_n$
38. $\xrightarrow{1}$ n 28...	
(R1)	(R2)
$\overbrace{0\ 0\ 0 \cdots 0}^{n+1}\ 1\ 0\ 0 \cdots\cdots 0$	$d_0\ \overbrace{0\ 0 \cdots\cdots 0}^{n}\ d_1\ d_2 \cdots\cdots d_{39-n}$



Then a secondary transfer is effected (to P)

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
43.	T	94028	This order "reads" one word from teletype tape and stores this word in R1. It is assumed that the word recorded on the tape is expressed in sexadecimal form, i.e. ten sexadecimal characters. The binary equivalent, *40 bits, of the sexadecimal word is stored in R1. (R2) erased in the process. One word from tape → R1
44.	P	L4028	This order prints the contests of R2 on the teleprinter. The word is printed in sexadecimal form, ten sexadecimal characters, the equivalent of the 40 bits of R2. (R1) and (R2) erased in the process. (R2) is printed on the teleprinter.
45.	IBM IN	58000	See explanation in Chapter X.
46.	IBM Out	48000	See explanation in Chapter X.

Notice that orders 43 through 46 ~~do not~~ require ~~any~~ special addresses.

47.	From Drum	J0	See explanation in Chapter XII.
48.	To Drum	J4	See explanation in Chapter XII.

Order No.	Prelim. Form	Sexadec. Form	Description of what the corresponding orders accomplish
49.	(R) P	F8	This order is referred to as the "logical 'and' order". The contents of R2 and the contents of P are compared "bit by bit". If corresponding bits are "1's", the corresponding bit of the result is a one; otherwise, the corresponding bit of the result is "0". This is the equivalent of bit by bit multiplication without "carries". The result goes to R2.

Example: (R2) 1101 0011 0010 0101 0010 1111 0011 0101 1100 1010
(P) 1011 0101 1011 1100 1010 0111 0110 1001 0110 0111

1001 0001 0010 0100 0010 0111 0010 0001 0100 0010 ⟶ R2.

Order No.	Prelim. Form	Sexadec. Form	Description
50.	⊕ P	88...	This order is referred to as the "exclusive 'or' order". The contents of R1 and the contents of P are compared "bit by bit". If corresponding bits differ, then the corresponding bit of the result is a "1". This is the equivalent of bit by bit addition without 'carries'. The result goes to R1.

Example: (R1) 1101 0011 0010 0101 0010 1111 0011 0101 1100 1010
1011 0101 1011 1100 1010 0111 0110 1001 0110 0111

0110 0110 1001 1001 1000 1000 0101 1100 1010 1101 ⟶ R1.

Time Estimates for the Various Orders

Print one 10 digit word (teletype)	2 seconds
Read one 10 digit word (teletype)	.5 seconds
Read one IBM card	1.2 seconds
Print one IBM card	1.0 seconds
Multiplication or division	.001 second
Transfer 48 words to or from Magnetic Drum	.08 seconds
All other operations	.0001 seconds

Clarence R. White

3.)

Decimal Representation of 10^{EXP}, 2^{EXP}, 4^{EXP}, 8^{EXP}, 16^{EXP} for Integral Exponent

10^{EXP}		2^{EXP}	4^{EXP}	8^{EXP}	16^{EXP}
	1048576.	20	10		5
6	1000000.				
	524288.	19			
	262144.	18		6	
	131072.	17			
5	100000.				
	65536.	16	8		4
	32768.	15		5	
	16384.	14	7		
4	10000.				
	8192.	13			
	4096.	12	6	4	3
	2048.	11			
	1024.	10	5		
3	1000.				
	512.	9		3	
	256.	8	4		2
	128.	7			
2	100.				
	64.	6	3	2	
	32.	5			
	16.	4	2		
1	10.				
	8.	3		1	
	4.	2	1		
	2.	1			
0	1.	0	0	0	0
	.5	- 1			
	.25	- 2	- 1		
	.125	- 3		- 1	
- 1	.1				
	.0625	- 4	- 2		- 1
	.03125	- 5			
	.015625	- 6	- 3	- 2	
- 2	.01				
	.0078125	- 7			
	.00390625	- 8	- 4		- 2
	.001953125	- 9		- 3	
- 3	.001				
	.0009765625	- 10	- 5		
	.00048828125	- 11			
	.000244140625	- 12	- 6	- 4	- 3
	.0001220703125	- 13			
- 4	.0001				
	.00006103515625	- 14	- 7		
	.000030517578125	- 15		- 5	
	.0000152587890625	- 16	- 8		- 4
- 5	.00001				
	.00000762939453125	- 17			
	.000003814697265625	- 18	- 9	- 6	
	.0000019073486328125	- 19			
- 6	.000001				
	.00000095367431640625	- 20	- 10		- 5
	.000000476837158203125	- 21		- 7	
	.0000002384185791015625	- 22	- 11		
	.00000011920928955078125	- 23			
- 7	.0000001				

4.)

ORDVAC CODE CHECKER

L. W. Campbell and B. T. Wade

Title: ORDVAC Code Checker

Purpose: To print information about ORDVAC orders as they are performed.

Number of words: Permanent part : 12 words. (card P_1)

Monitor part : 135 (cards M1 to M14)

Write part : 288 for 29 to 52 intervals

Temporary Storage: None in memory; Tracks K6-SK on the drum.

Initial Requirements: Tape specifying intervals to be printed.

Result: Information about an order printed on a card (or teletypewriter).

Remarks: Halts (Zu 001) if attempt to go from n interval to 1 interval.
Halts (Zu 014) if did U' to an n interval start address. (Without monitoring the U' order).

Restrictions:
1) Drum orders must always be considered as left orders.
2) Contents of (R_2) is replaced by the drum order when a drum order is printed.
3) Monitor part (135 words) must not be stored at an address ending with 22, 62, K2 or F2.
4) Can't print within a subroutine for which printing has been automatically skipped and maintain control at the end of the subroutine.
5) The "start" orders of the program to be checked must be in the memory at the time the interval tape is read by the code checker.

Additional restrictions for n interval:
1) Can only start on a left order.
2) Whole word at the start address may not be referred to elsewhere in the program (except by transfer orders).
3) R^3 must not be important at the start address.
4) No. of intervals ≤ 52.

Additional restrictions for 1 interval:

1) Can't start on the right of a word that does a secondary transfer on the left unless the secondary transfer transfers to the next word (A+1) or the right order is a transfer order.

2) The half word at the start address may not be referred to elsewhere in the program (except by transfer orders).

3) Cannot start on the right of the same word that the previous interval stopped on. (This will cause the start to be by-passed n + 1 times instead of the usual n times).

December 1956

ORDVAC CODE CHECKER

5.) L. W. Campbell and B. T. Wade

This routine will print information about an ORDVAC order immediately after it has been performed. It is capable of printing at the rate of 100 orders per minute on IBM cards and uses only 12 memory positions for permanent storage.

A tape (or switch settings) controls the amount of printing. Printing may start or stop on almost any ORDVAC order and the number of times that the orders in a loop are printed is easily restricted. There are two somewhat different methods of controlling the printing: (1) the n interval method where all intervals that are to be printed are specified at the time the code checker is read into the machine or (2) the 1 interval method where one interval is printed before the next interval is read into the machine.

The n interval tape has the following form:

```
K c3 _ A _    c2 0 _ B _
. .    .       . .    .
. .    .       . .    .
. .    .       . .    .
. .    .       . .    .
K 8  _ A _   c2  0 _ B _
```

where: A = Address of order at which printing is to start.

B = Address of order at which printing is to stop.

(Printing will start at <u>any</u> A and stop when control reaches <u>any</u> B.)

C_2 = 0 specifies the left order at B.

C_2 = 8 specifies the right order at B.

$1 \leq K \leq 7$ specifies that printing is to start or continue at A K times only.

If K = 0, printing will start every time the order at A is performed.

C_3 = 0 specifies that more tape words and intervals follow.

C_3 = 8 specifies that this is the last word on the tape.

The 1 interval tape has two words for each interval in the following manner:

```
0 4 _ n _ 0 0 _ m _
c1 c3 _ A _ c2 0 _ B _
.  .    .    . .    .
.  .    .    . .    .
.  .    .    . .    .
0 4 _ n _ 0 0 _ m _
c1 8 _ A _ c2 0 _ B _
```

where: C_1 = 0 specifies the left order at A.

C_1 = 8 specifies the right order at A.

C_2, C_3, A, and B have the same meaning as for the n interval tape.

n = no. of times to pass A before printing starts at A. Printing starts on (n + 1)st time.

m = no. of times to stop printing at B before reading in the next interval. (The code treats m = 0 the same as m = 1).

$0 \leq n, m, \leq LLL$

(n and m must be specified in sexadecimal)

The two methods may be combined if the 1 interval method is used first and C_3 = 0 until the last of the n intervals is read from the tape.

The routine has three parts, each of which may be stored at any address ≥ 00K except for the second section which cannot be stored at any address ending with 22, 62, K2, or F2. The routine must be read in with the standard modifying input routine. A key word may be inserted in front of any or all of the three parts. If no key words are inserted, each part will immediately follow the preceding part. The first or permanent (P) part is 12 words long and it must be in the memory essentially all the time (whenever either a start or stop address is reached). This part is contained on one card.

The second part performs the monitoring and it normally is on the drum. It is called into the memory whenever a start address (A) is reached and

must remain there until a stop address (B) is reached. This second part is contained on cards labeled M1 to M14. The third part performs the printing and it is also normally on the drum. It is called into the memory whenever a start address (A) is reached and it remains there until either a stop address (B) is reached or a U* order (transfer to a subroutine) is performed. At the return from a subroutine, it is again called into the memory so that printing may resume. This third part is contained on cards labeled W1 to W20.

Whenever the second or third parts of the code checker are called into the memory from the drum, the contents of that memory space is first recorded on the drum. Whenever a stop (B) address is reached, the memory is reset to what it was at the start (A) address. In the case of the third part, this also happens at a U* order and at the return from the subroutine. Hence these two parts of the code checker may be stored in the same space as subroutines or anything that need not be in the memory while the code checker is actually being used. However, the second part may not be stored in the same space as any subroutine in which printing is automatically skipped. (The second part must remain in the memory to monitor and find the E' order.) The drum storage (tracks K6 - SK) must not be erased if the code checker is to do any printing afterwards.

Printing within subroutines may be automatically skipped by using a U* order (SN order or 34 for 9 bit code) to transfer to the subroutine, provided that the first E' order after the first secondary transfer within the subroutine sets the return address. The intervals should not include any start or stop addresses of orders that are done between the U* order and the order at the return address. (Starts and stop within such a subroutine will work, but control will be lost at the exit order.)

The routine changes the program code at all start (A) addresses but does not change it at any stop (B) addresses. For each n interval start, the whole word is replaced with:

(-) self
U code checker (4000)

hence, printing must start on the left of the word at A (unless the U' order to any start address is printed due to a previous start) and any start address

must not contain anything that is changed or referred to by the program (i.e. variable address, dummy address, constant, etc.) For the 1 interval case, only a half word in the program is changed to a transfer to the code checker, hence printing may start on either side of the start word and only that half word must not contain anything that is needed there by the program. The contents of these start addresses are replaced whenever printing is completed for that start. (At (K+1)st time A is reached for n interval or when B is reached m times for 1 interval). It is allowable to have two (or more) successive starts without stopping between them. It is allowable also to start and stop on the same order.

The contents of R_1, R_2, and R^3 are saved at all starts and are reset at all stops. (However R^3 is used for the n interval start).

Printing is normally done on IBM cards, one card for each order. Printing may be on teletypewriter if the card so labeled (W17) is removed. The information printed on a card is:

Field A.	B	C	D	E	F	G	H
order 00 Add.	(R_1)	(R_2) or drum ord.	(Memory)	(R_1)dec.	(R_2)dec.	(Memory) dec.	Blank

where fields A-D are sexadecimal and E-G are decimal with a K (or S) for + (or -). The decimal numbers are rounded (when possible) by .00000 00005.

There is an option that allows only orders that perform a transfer to be printed. This option may be used if card so labeled (W16) is removed.

All monitored halt orders (Zu and Zx) will be performed twice (as are all other orders except teletype and drum orders).

September 18, 1956

5.) ORDVAC One Address Floating Binary Routine (OFB)

L. Campbell and S. Lehman

General Description:

This is an interpretive routine that will interpret pseudo one address orders and perform floating binary arithmetic. It uses two memory positions as an accumulator that acts similarly to the machine's R_1 but is capable of containing a floating binary number. One memory position, which we shall call A_c, holds the coefficient of the floating binary number and the other memory position, which we shall call A_e, holds the binary exponent. Together, these two positions shall be called the accumulator or A. A_c is 010 and A_e is 011.

Numbers:

This routine normally uses the sign and the next 31 bits for the coefficient and the last eight bits for the exponent. These lengths can easily be changed by changing five constants in the control section of the routine. All exponents are held as the true exponent plus a bias of 2^{n-1} where n is the length of the exponent. The normal range of the exponent e is

$$-128 \leq e < 127$$

except for the exponent in A_e which has the range

$$-2^{46} \leq e_A < 2^{32}$$

and the bias is 128 or 80 in sexadecimal. All coefficients are kept normalized except the one in the accumulator. A normalized coefficient C is of the form

$$1/2 \leq |C| < 1$$

except zero which is a word of all zero bits.

Orders:

The pseudo orders use a six bit order type and an address that has the same length as the machine's address. There usually are two orders per word, but there is an optional method of coding only one order per word which accomplishes a gain of 12-20 per cent in speed.

Entrance:

Programs that may need to use the code checker and subroutines that need no code checking enter the routine at different places. Also, if the next group of orders that are to be performed is coded one order per word, then the entry point is different from that of two orders per word. In any case, the address of the word preceding the first pseudo order that is to be done must be in R_1 at the point of entrance. This is most easily accomplished by the following coding:

Programs		Subroutines					
2 orders/wd	1 order/wd	2 orders/wd			1 order/wd		
+ self	+ self	+ self	or	A + 056	+ self	or	A + 052
U 040	U 041	U 050		+ self	U 051		+ self

where self + 1 contains the first pseudo order that is to be done.

Description of the orders:

The memory contents as referred to in any order is here called N and N_c is the coefficient of N and N_e the exponent.

Fl. Pt. Arithmetic orders:

Final Code	Prelim Code	Description	A_c		A_e	Memory
00	f+	$N \rightarrow A$	+ N	00	00000 000 N_e	No change
24	f-	$-N \rightarrow A$	$-N_c$	00	00000 000 N_e	No change
F4	f|+|	$\|N\| \rightarrow A$	$\|N_c\|$	00	00000 000 N_e	No change
64	f|-|	$-\|N\| \rightarrow A$	$-\|N_c\|$	00	00000 000 N_e	No change
N4	f(+)	$A + N \rightarrow A$	$2^{-1}(A_c+N_c)$		(Larger of A_c & N) + 1	No change
04	f(-)	$A - N \rightarrow A$	$2^{-1}(A_c-N_c)$		" " "	No change
68	fX	$A \cdot N \rightarrow A$	$A_c \cdot N_c$		$A_e + N_e$ - Bias	No change
78	f÷	$A/N \rightarrow A$	$2^{-1}(A_c/N_c)$		$A_e - N_e$ + Bias+1	No change
10	fM	$A \rightarrow N$	A_e Normalized & rounded 00		00000 000 A_e	$A_c + A_e$
18	f← n	$2^n A \rightarrow A$	No change		$A_e + n$	No change
08	f→ n	$2^{-n} A \rightarrow A$	No change		$A_e - n$	No change
30	oM	0 A, N	00000 00000		00000 00000	00000 00000
70	sM	$A_c + A_e \rightarrow N$	No change		No change	$A_c + A_e$

Transfer Orders:

20	C	Transfer to the left machine order at N if $A_c \geq 0$. If $A_c < 0$, go on to next pseudo order.
40	C'	Transfer to the right machine order at N if $A_c \geq 0$. If $A_c < 0$, go on to next pseudo order.
NO	U	Transfer to left machine order at N.
14	U'	Transfer to right machine order at N.
2N	fC	Transfer to left pseudo order at N if $A_c \geq 0$. If $A_c < 0$, go on to next pseudo order.
4N	fC'	Transfer to right pseudo order at N if $A_c \geq 0$. If $A_c < 0$, go on to next pseudo order.
NN	fU	Transfer to left pseudo order at N.
1N	fU'	Transfer to right pseudo order at N.
SN	U*	Transfer to a subroutine at N and return to the next left pseudo order. It puts $A_c + A_e$ in R_1 as the argument for the subroutine and at the return from the subroutine, it puts the contents of R_1 in A_c and A_e before going on to the next left order. If the argument of a subroutine is a floating binary number, it must be normalized in A before giving this U* order. (Subroutines assume the argument in R_1, normalized if a fl. pt. no. begin and end in machine orders use the right address of R_2 or R^3 for the return address may use any pseudo order except U*).

The C, C', U, and U' orders provide the method of leaving the interpretive routine and transferring to machine orders. A_c is left in R_1 at the point of transfer. Note that the negative branch of all compare orders goes on to a pseudo order.

Fixed Point Orders:

All of these orders perform exactly as the machine does them with A_c being used as R_1 or R_2 and A_e is not changed.

K8	+	$N \rightarrow A_c$	SO	1M	$1/2 \rightarrow A_c$, N
S8	-	$-N \rightarrow A_c$	84	2^{-19}(+)M	$2^{-19} + N \rightarrow A_c$, N

3N	(+)	$N + A_c \longrightarrow A_c$
5N	(-)	$A_c - N \longrightarrow A_c$
6N	x	$A_c \cdot N \longrightarrow A_c$
7N	÷	$A_c / N \longrightarrow A_c$
60	M	$A_c \longrightarrow N$
90	E	Left Add. of $A_c \longrightarrow$ Left Add. of N
50	E'	Right " " " $\longrightarrow$ Right " " "

88	$2^{-39}(+)N$	$2^{-39} + N \quad A_c$, N
8N	$2^{-19} + 2^{-39}(+)M$	$2^{-19} + 2^{-39} + N \quad A_c$, N
58	$\longleftarrow$ n	$2^{n} A_c \longrightarrow A_c$
48	$\longrightarrow$ n	$2^{-n} A_c \longrightarrow A_c$
74	←~~6~~— n	$2^{1} A_c \longrightarrow A_c$
28	—~~1~~→ n	$2^{-n+1} \longrightarrow A_c$

Fixed Pt. conv. and Reconversion:

98 C' + Convert fixed pt. N ($-1 \leq N < 1$) to a floating binary number in A. The number in A is normalized and rounded.

9N R' M Reconvert the fl. pt. no. in A to fixed pt. and store it at N and also leave it in A_c. ($-1 \leq N < 1$) It does not check for spill and will store some residue if $A > 1$ initially. $N = 0$ if $|A| \leq 2^{-39}$ initially.

Fl. Decimal Conversion and Reconversion:

J8 C + Convert the floating decimal coeff. in N and the floating decimal exponent in N+1 to a fl. binary number in A. The number in N and N+1 must be in binary form with the exponent in N+1 scaled at 10^{-2}. (This scaling may easily be changed). The floating binary number in A is normalized and rounded.

JN RM Reconvert the fl. binary number in A to a floating decimal number and store the coeff. at N and the exponent at N + 1 scaled at 10^{-2}. The numbers at N and N+1 are in binary form. The number in A is destroyed in this reconversion process.

Miscellaneous Orders:

F8 DN Does nothing. No change in A, goes on to the next pseudo order.

LN H Unconditional Halt. Goes on to next pseudo order if toggled past.

N8 (N) P   Fixed pt. polyonomial evaluation. Evaluates

$$a_o + a_1 x + \dots + a_n x^n = y .$$

x is put in 016 before this order is given and N contains

$$2^{-3} n \quad A_o \quad x \; x \quad 0\,6\,7 \qquad a_o \quad A_i = i + A_o$$

The result y is left in A in floating binary but is not normalized. This section may also be used as a subroutine by having x in 016, the word at N in R_2 with 067 changed to any return address desired and transferring to 052. After completion of the subroutine y, in fixed point, in R_1, R_2 and A_c = 010.
$|x|$, $|y|$, $|a_i|$ and |partial sums| must be less than 1.

Storage Requirements:

The routine is coded in sections that, except for the control section, may be stored anywhere in the memory. If a certain section of orders are not used anywhere in a problem, that section need not be stored for that problem. All sections use 010-015 for temporary storage.

Control:

Fixed location: 040-OSL. This section must always be used to do any pseudo orders and it includes the following orders: all transfers except U*, Halt, DN, oM, sM, M, +, -, (+), (-). This section must be read into the machine before any of the other sections.

Floating Point Arithmetic:

66 words. Includes all of the floating arithmetic orders and is also used by any other orders that normalize numbers.

U*:

13 words includes only U* order.

C' + and R'M:

11 words.

C+ and RM:

62 words, includes both orders.

Fixed Point Orders:

14 words includes E, E', x, ÷, M, ←, →, ←S, 1→, 2^{-19}(+)M, 2^{-39}(+)M, 2^{-39}(+)M.

Polyonomial:

11 words.

Code Checking Features:

The routine will print two words on the teletypewriter and halt when it encounters certain difficulties. When it prints, the right address of the first word is the address of the order at which the routine encountered difficulty and the second word is the contents of that address. The second word contains the order that caused the trouble. The trouble will be one of the following:

1) Attempted a division by zero when $A_c \neq 0$. (Note: 0/0 = 0)
2) Exponent exceeded capacity when normalizing.
3) Attempted to use one order in a section that wasn't read into the machine after the control routine (see Storage Requirements above).
4) Attempted to use an unused pseudo order type.

After the printout, the machine will halt, and if toggled on the routine will insert the biggest possible exponent in A_e before going on to the next order.

Not all of the unused pseudo order types cause a printout, some of them halt without printing. A printout at this time, or anytime, may be obtained by transferring to the right side of 07K. If the counter says 0N0 at a halt, it may be a programmed halt with an LN order type.

The counter containing the address of the pseudo order being performed is the right address of 05K, or 06S for one order per word system.

One Order Per Word Coding:

Each order (half word) is assigned an address. The address N in the order is used as a right address but the fU or fC order

should be used to transfer to any one order per word. (fU' or fc' will work except when using the code checker).

The transcriber takes each half word as it is coded and places it on the right side of a word. It also "decodes" the order type and places a corresponding address in the left address of the word and adds in a 44 order type on the left. A 33 key word (see transcriber description) will start the transcriber to expanding half words to full words. Two half words are punched in each 12 col. field on a card in preparation for transcribing, but if the number of half words is uneven, an extra half word of 800000 must be added before a key word (32 or some other one) is given to change back to transcribing word for word.

Speed:

The average time to do a floating point arithmetic order is 2. ms. The routine does about 400 fl. pt. orders per second or about 500 per second if the one order per word method of coding is used.

September 1956

(6.) ORDVAC Floating Point (One Address) Code Checker

L. Campbell

This routine will print out information about the monitored one address floating point (OFB) orders as they are being performed. It has no effect on machine orders and will print only when floating point orders are being performed. The use of this code checker is almost identical with the use of the code checker that code checks the machine orders.

A tape (or switch settings) controls the amount of printing. Printing may start or stop on any floating point order and the number of times that the orders in a given loop are printed is easily restricted. There are two different methods of controlling the printing: (1) the n interval method where all intervals that are to be printed are specified at the time the code checker is read into the machine (2) the 1 interval method where one interval is printed before the next interval is read into the machine.

The n interval tape has the following form:

C_1+k	C_3	A		C_2	0	B
.	.	.		.	.	.
.	.	.		.	.	.
.	.	.		.	.	.
C_1+k	8	A		C_2	0	B

where:

A = Address of fl. pt. order at which printing is to start.

B = Address of fl. pt. order at which printing is to stop.

C_1 (or C_2) = 0 specifies the left order at A (or B)

C_1 (or C_2) = 8 specifies the right order at A (or B).

All one order per word orders must be specified as <u>right</u> orders.

$1 \leq k \leq 7$ specifies that printing is to start or continue at A k times only.

If k = 0, printing will start every time the order at A is performed.

C_3 = 0 specifies that more tape words and intervals follow.

C_3 = 8 specifies that all the tape has been read. (C_3 = 8 for the last word on the tape and 0 for all other words.)

The 1 interval tape has two words for each interval in the following manner:

0	4	n	0	0	m
C_1	C_3	A	C_2	0	B
.	.	. . .	.	.	. . .
.	.	. . .	.	.	. . .
.	.	. . .	.	.	. . .
0	4	n	0	0	m
C_1	8	A	C_2	0	B

where: C_1, C_2, and C_3 have the same meaning as above.

n = no. of times to pass A before printing starts at A. Printing starts on (n+1)st time.

m = no. of times to stop printing at B before read in the next interval. (The code treats m = 0 the same as m = 1).

$$0 \leq n, m \leq L\,L\,L$$

(m and n must be specified in sexadecimal)

The two methods may be combined if the 1 interval method is used first and C_3 = 0 until the last of the n intervals is read on the tape.

The m interval case adds n positions to the Control Section and the 1 interval case adds 3.

The routine has two parts, each of which may be stored anywhere. The first part shall be referred to as the Control Section and the second part as the Print Section. The routine must be read in with the standard modifying input routine. A key word at the beginning of the deck controls the storing of the Control Section and another key word card inserted between the sixth and seventh cards controls the storing of the Print Section. If no key word is inserted the Print Section will be stored at 4070. The Print Section is only used during an interval while printing is actually being done. This section may be stored in the same section of memory as subroutines or anything that is not needed in the memory while the code checker is printing. This section is normally on the drum, but when printing starts at an A address, the contents of the memory space is stored on the drum and the print section is called into the memory. The memory is reset every time that printing stops;

i.e., when entering a subroutine, when a B address is reached or when a transfer from floating point to machine orders is effected. The drum storage (tracks SS-N1) must not be erased by the program if the code checker is to print afterwards.

The routine does not change the program code in any manner but it does change the Control Section of the OFB and the U* section if it is being used. Hence the code checker must be read into the machine after the floating point (OFB) routine. It also uses the RM order section to reconvert numbers to fl. decimal, hence that section must be in the machine whenever printing is taking place.

Printing is normally done on IBM cards, one card for each order. Printing may be on teletypewriter if the 3rd, 4th and 5th cards from the end are removed. The information printed on a card is:

Field A	B	C	D	E - F	G - H
Order 00 Add.	$A_{coeff.}$	$A_{exp.}$	Memory	A in Fl. Dec.	Mem. in Fl. Dec.

where fields A - D are sexadecimal and E - H are decimal with a K (or S) for + (or -). The numbers are rounded (when possible) by approximately 00000 00005. The printing on the teletypewriter is two words per order:

Order 00 Add. A in Fl. Dec.

where A has K or S and six digit coefficient and K or S and two digit exponent.

If a decimal exponent of all 9's is printed, (in F or H) the decimal coefficient is the fixed point reconverted contents of the memory (or A). This occurs when numbers do not have the floating point form and are probably fixed nos. or orders. A zero in the memory will also be printed with all 9's for the exponent.

Orders in subroutines will never be printed if they are properly coded to use 050, 051, 052 or 056 as the entrance address to the OFB. In addition to this, printing always stops at an U* order and upon the return to the

next order after the U* order, the code checker resumes the same status as it had at the time of the U* order. This holds regardless of whether printing during the subroutine occurred or not.

The Control Section of the code checker must not be destroyed as long as the OFB is used. It may be erased if the Control Section (and U* section) is re-read into the machine.

Length of Routine:

Temporary storage: None

Control Section:

78 words (4000-404J) + n for n intervals

or, 78 words + 3 for 1 interval

Print Section:

144 words (4000-408L)

Drum Storage:

Tracks SS-N1 (must remain intact until all printing is completed).

November 16, 1956

7.) ORDVAC FLOATING POINT (OFB) CODE

Lloyd W. Campbell

Title: Floating Point Square Root

Purpose: Computes $x = +\sqrt{N}$

Type: Standard closed

Number of words: 31

Temporary storage: 010-014

Accuracy: At least nine significant decimal digits

Time: Average 6.8 ms Max. 8.3 ms Min. .64 ms (for N = 0)

Initial Requirements: Enter at first word
Initial Data: N in R_1
Return Address: Right Address of R_2 (or later R^3)

Result: $x = \sqrt{N}$ in R_1. x is normalized if N is normalized

Remarks: Halts if N < 0

Restrictions: Uses Control Section (constants) of OFB and $(1-2^{-39})$ at 00L

Description: Obtains initial approximation $x_o = .414 + .59375N$ and uses Newton-Raphson method of

$$x_{i+1} = \frac{N}{2x_i} + \frac{x_i}{2}$$

until

$$\left| \frac{N}{2x_i} - \frac{x_i}{2} \right| < \epsilon$$

ϵ at 401F is set at 2^{-16} which gives approximately the accuracy of $\epsilon^2 = 2^{-32}$. ϵ may be changed if more or less accuracy is desired.

x_i is the coeff. of $\sqrt{N}$ if N has an even exponent or $x_i \cdot \frac{1}{\sqrt{2}}$ is the coeff. of $\sqrt{N}$ if N has an odd exponent. The exponent of $x = \sqrt{N}$ is computed as

$$1/2 \text{ (Exp. of N + 1)}$$

November 16, 1956

8.) ORDVAC FLOATING POINT (OFB) CODE

Lloyd W. Campbell

Title: Floating Point sin-cos

Purpose: Computes sin x and cos x for given x.

Type: Standard closed

Number of words: 51

Temporary storage: 010-01K

Accuracy: Error < $5 \cdot 10^{-10}$ (error increases if x 2π)

Time: Average 23.7 ms Max. 28 ms Min. 21 ms.

Initial Requirements: Enter at first word.
Initial Data: x in R_1
Return Address: Right Address of R_2 (or later R^3)

Result: sin x in R_1 and 016
cos x in R_2 and 017

Remarks: If $x > 2^{41}$, sin x and cos x have no significance but result will probably be sin x = 0, cos x = 1.
If $x < 2^{-37}$, sin x = 0, cos x = 1.

Restrictions: Uses the Control Section, Floating Arith. Section and C'+ and R'M Section of the OFB floating point routine.
Uses $(1-2^{-39})$ at 00L.

Description: Computes $\frac{x}{2\pi}$ and discards the integer of the result.
Using the fractional part of $\frac{x}{2\pi}$, it computes $\bar{x}$ such that
$-1/4 \leq \bar{x} \leq 1/4$
and then $\sin x = \sin 2\pi \bar{x}$. The cos x is computed using the sin 2π y series and identity relations between the sin and cos.
The $\sin 2\pi \bar{x}$ is computed by using a chebyshev polynomial

$$\sin 2\pi \bar{x} = \bar{x} \sum_{i=0}^{5} 4^{2i+1} c_i \bar{x}^{2i}$$

where

c_0 = 1.5707 9632677
c_1 = -.64596 4095587
c_2 = .07969 2603718
c_3 = -.00468 1657795
c_4 = .00016 0254690
c_5 = .00000 3431829

December 1956

9.) ORDVAC FLOATING POINT (OFB) CODE

L. W. Campbell and R. H. Brunelle

Title: Floating Point Exponential

Purpose: Computes e^x for given x

Type: Standard closed

Number of words: 50

Temporary storage: 010-019

Accuracy: At least nine significant decimal digits.

Time: Average 13 ms. Max. 16 ms. Min. 1.1 ms.

Initial Requirements: Enter at first word
Initial Data: X in R_1
Return Address: Right Address of R_2

Result: e^x in R_1

Remarks: Halts at 402K if $x \geq 128$ (If toggled past this halt, $e^x = 2^{126}$).
Halts in OFB at 083 (after an erroneous error print)
if $88.8 < x < 128$. ($e^{88.7} \approx 10^{39}$ exceeds OFB range)
$e^o = 1$ exactly
$e^x = 0$ if $x < -88.8$

Description: Compute I and f where I is the integral part of $\frac{x}{\log_e 2}$ and

and f is the fractional part of $\frac{x}{\log_e 2}$.

If $1/2 \leq f < 1$, replace f with f - 1 and then if x > o replace I with I + 1.

Or if $-1 \leq f < -1/2$, replace f with f + 1 and then if x > o, replace I with I + 1.

If $-1/2 \leq f < 1/2$ and x < o, replace I with I + 1.

then

$$e^x = \left[1 + \frac{2f}{a - f + f^2 \left(b + \frac{c}{d + f^2}\right)} \right] 2^I$$

where $a = \frac{2}{\log_e 2}$ $\qquad c = 49/20\ a$

$b = .1\ a$ $\qquad d = 21/2\ a^2$

Restrictions: Uses Control and Floating Arithmetic sections of the OFB and the permanent constants at OOK-OOL.

November 29, 1956

10.)

ORDVAC FLOATING POINT (OFB) CODE

Lloyd W. Campbell

Title: Floating Point Natural Logarithm.

Purpose: Computes $\log_e x$ for given x.

Type: Standard closed.

Number of words: 41

Temporary storage: 010 - 019

Accuracy: At least nine significant decimal digits.

Time: Average 14.7 ms; maximum 20 ms; minimum 14 ms.

Initial Requirements: Enter at first word.
Initial Data: x in R_1
Return Address: Right Address of R_2

Result: $\log_e x$ in R_1 and 016.

Remarks: Halts (at 4001) if $x < 0$.

This routine must be changed if the length of the exponent is changed to something other than the standard eight bits.

Restrictions: Uses the Control and Floating Arithmetic sections of the OFB floating point routine.

Description: Let x_c = coefficient of x and x_e = exponent of x

Compute $\bar{x} = \frac{3/2\ x_c - 1}{3/2\ x_c + 1}$

and $\log_e 3/2 x_c = \bar{x} \sum_{i=o}^{4} Ci\ \bar{x}^{2i}$

where Ci are the coefficients of a Chebyshev polynomial

C_o = 2.00000 00000
C_1 = .66666 66169
C_2 = .40000 98961
C_3 = .28502 79895
C_4 = .24146 79584

then compute,

$$\log_e x = \log_e 3/2\ x_c \quad \log_e 2/3 + x_e \log_e 2$$

December 1956

11.) ORDVAC FLOATING POINT (OFB) CODE

L. W. Campbell

Title: Floating Point Arcsin

Purpose: Computes $\theta = \arcsin x$

Type: Standard closed.

Number of words: 64 + $\sqrt{x}$ (31 words) = 25 words.

Temporary storage: 010-01K

Accuracy: Error < $.10^{-10}$

Time: Average 24 ms. Max. 37 ms. Min. 5.5 ms.

Initial Requirements: Enter at first word

Initial Data: x in R_1

Return Address: Right Address of R_2

Result: θ in R_1 and 016. $-\pi/2 \leq \theta \leq \pi/2$; θ is in radians.

Remarks: If $1 < /x/ < 2$, routine assumes that $/x/ = 1$.

If $2 < /x/$, results will be arcsin $[x(\text{mod } 2)]$.

Restrictions: Uses Control, Floating Arithmetic and C' + &R'M sections of OFB. Floating Point $\sqrt{x}$ routine is used (if $/x/ > \sqrt{2}/2$) and must be stored in the memory immediately following this routine.

Description: Computes $\bar{x}$ in the following manner:

If $0 \leq /x/ \leq \sqrt{2}/2$, then $\bar{x} = - /x/$ and $\arcsin /x/ = /\arcsin \bar{x}/$

If $\sqrt{2}/2 \leq /x/ \leq 1$, then $\bar{x} = -(1 - x^2)^{1/2}$ and $\arcsin /x/ = /\pi/2 + \arcsin \bar{x}/$

$$\arcsin \bar{x} = \bar{x} \sum_{i=0}^{9} C_i (\bar{x})^{2i}$$

where C_i are the following Chebyshev coefficients:

C_o = .99999 99997 15 C_5 = .03717 68708 48

C_1 = .16666 67753 37 C_6 = -.04439 60736 91

C_2 = .07499 30572 86 C_7 = .16748 53422 27

C_3 = .04481 36827 32 C_8 = -.20271 07850 45

C_4 = .02826 84745 97 C_9 = .14636 21900 95

And finally, if $x < 0$, then $\arcsin x = - \arcsin /x/$.

12) ORDVAC FLOATING POINT (OFB) CODE

L. W. Campbell

Title: Floating Point Arctangent

Purpose: Computes θ = Arctan x

Type: Standard Closed

Number of words: 59

Temporary storage: 010-019

Accuracy: Error 5.10^{-10}

Time: Average 14.5 ms. Max. 18 ms Min. 2.5 ms.

Initial Requirements: Enter at first word

Initial Data: x in R_1

Return Address: Right Address of R_2

Result: θ in R_1 and 016. $-\pi/2 \le \theta \le \pi/2$; θ is in radians.

Restrictions: Uses the Control and Floating Arithmetic sections of the OFB.

Description: Computes $\bar{x}$ in the following manner:

If $0 \le /x/ \le \sqrt{2} - 1$ then $\bar{x} = /x/$ and $\arctan /x/ = \arctan \bar{x}$

If $\sqrt{2} - 1 < /x/ \le \sqrt{2} + 1$ then $\bar{x} = \frac{/x/ - 1}{/x/ + 1}$ and $\arctan /x/ = \pi/4 + \arctan \bar{x}$

If $\sqrt{2} + 1 < /x/$ then $\bar{x} = -\frac{1}{/x/}$ and $\arctan /x/ = \pi/2 + \arctan \bar{x}$

$$\arctan \bar{x} = \bar{x} \sum_{i=0}^{5} C_i (\bar{x})^{2i}$$

where C_i are the following Chebyshev coefficients:

C_0 = .99999 99993 92

C_1 = -.33333 30749 37

C_2 = .19998 21081 81

C_3 = -.14239 98333 75

C_4 = .10572 81793 18

C_5 = -.06033 25166 48

And finally, if $x < 0$, then $\arctan x = -\arctan /x/$.

December 1956

13.) ORDVAC FLOATING POINT (OFB) CODE

L. W. Campbell

Title: Floating Point arcsin-cos

Purpose: Computes Θ = arcsin-cos for given sin Θ and cos Θ.

Type: Standard closed

Number of words: 61

Temporary storage: 010-01K

Accuracy: Error $< 5.10^{-10}$

Time: Average 19.7 ms. Max. 25 ms. Min. 16.8 ms.

Initial Requirements: Enter at first word.

Initial Data: Left Address in R_1 is the address of sin Θ and the Right Address is the address of cos Θ

Return Address: Right Address of R_2

Result: Θ in R_1 and 016. $-\pi < \Theta \leq \pi$; Θ is in radians

Remarks: Cycles if 2 > sin Θ and cos $\Theta > \sqrt{2}/2 + 2^{-32}$

May cycle or get some sort of results (mod 2) if either or both sin and cos > 2.

Restrictions: Uses Control, Floating Arithmetic and C' + & R'M sections of OFB.

Description: Computes $\bar{x}$ in the following manner:

If $0 \leq /\sin\Theta/ \leq \sqrt{2}/2$, $\bar{x} = -/\sin\Theta/$ and $\Theta_1 = /\arcsin\bar{x}/$

Or if $/\sin\Theta/ > \sqrt{2}/2$ and $0 \leq /\cos\Theta/ \leq \sqrt{2}/2$, $\bar{x} = -/\cos\Theta/$ and $\Theta_1 = /\pi/2 + \arcsin\bar{x}/$

$$\arcsin\bar{x} = \bar{x}\sum_{i=0}^{9} C_i(\bar{x})^{2i}$$

where C_i are the following Chebyshev coefficients:

C_0 =	.99999 99997 15	C_5 =	.03717 68708 48
C_1 =	.16666 67753 37	C_6 =	-.04439 60736 91
C_2 =	.07499 30572 86	C_7 =	.16748 53422 27
C_3 =	.04481 36827 32	C_8 =	-.20271 07850 45
C_4 =	.02826 84745 97	C_9 =	.14636 21900 95

Then if $\cos\Theta \geq 0$, $\Theta_2 = \Theta_1$ or if $\cos\Theta \leq 0$, $\Theta_2 = \pi - \Theta_1$ and if $\sin\Theta \geq 0$, $\Theta = \Theta_2$ or if $\sin\Theta \leq 0$, $\Theta = -\Theta_2$.

January 1957

14.) ORDVAC FLOATING POINT (OFB) CODE

L. W. Campbell

Title: Solve Normal Equations (SNE) and Matrix Inversion.

Purpose: To solve a system of n equations in n unknowns and produce the inverse of the given coefficient matrix.

Number of words: 62

Temporary storage: 010-01S + n (n = size of matrix and $n \leq 36_{dec}$).

Accuracy: Depends on the "condition" of the system.

Time: Approx. Average $9n^3 + 12n^2$ ms (for n x n + 1 matrix).

Initial Requirements: Enter at first word.

Initial Data: x x _ A_{11} _ x x _ n _ in R_1 where A_{11} is the address of the first element of the matrix and n is the size of the square matrix A. The matrix is stored in consecutive memory positions in the order of first row, then b_1, second row, then b_2 etc. where the equations have the form Ax = b.

Return Address: Right Address of R_2

Result: The coefficient matrix A is replaced with its inverse and the column of b_K is replaced with the solution x_K.

Remarks: Will halt on a floating point division by zero if the variables need to be renumbered or if the matrix is singular.

Restrictions: Uses the Control and Floating Arithmetic sections of OFB. The size of the matrix is restricted to 36 unless provision is made for n temporary storage positions at some place other than the Standard 01N-03L.

Description: Uses Gauss elimination to solve the system of equations and performs the same row operations on the rows of the identity matrix to obtain the inverse. The identity matrix is not actually stored as such, the inverse matrix gradually replaces the original matrix.

December 1956

15.) ORDVAC FLOATING POINT (OFB) CODE

L. W. Campbell

Title: Floating Pt. IBM Input (IBMC)

Purpose: To read floating decimal numbers from cards and store the equivalent floating binary numbers in the memory.

Number of words: 34 words + IBMI (105 words) = 139 words.

Temporary storage: 010-03L

Initial Requirements: Enter at first word.

Initial Data: _N_ _M_ I X A_o _ _ in R_1

where N = no. of fl. pt. nos. on each card.

$N \leq 12$

M = total no. of fl. pt. nos. to be read

(N and M are sexadecimal nos.)

A_o = Initial store address for the M nos.

I = Increment for A_i (I = 0 is same as I = 1

Return Address: Right Address of R_2

Result: M floating binary numbers stored at A_o, A_I, A_{2I}, ...$A_{M - 1)I}$

Remarks: The format of the numbers on the card is controlled by the field words of the IBMI. These field words may be changed to any desired format except that each floating decimal number must be in two consecutive fields, i.e. coefficient is in one field and the exponent times 10^{-2} is in the next field.

Restrictions: Uses the Control, Fl. Arithmetic, and C+ and RM sections of the OFB.

IBMI must immediately follow this routine in the memory.

No. of fl. pt. nos. on each card (N) must be less than thirteen and the left half of the card must not have more than six of them.

Cannot print 000 unless I > 1.

16.) ORDVAC Floating Point (OFB) Code

L. W. Campbell — June 1957

Title: Floating Point Arccos

Purpose: Computes $\theta = \arccos x$

Number of words: 66 + $\sqrt{x}$ (31 words) = 97 words

Temporary storage: 010 - 01K

Accuracy: Error $< 5.10^{-10}$

Time: Average 24 ms. Max 37 ms. Min. 5.5 ms.

Initial Requirements: Enter at first word.
Initial Data: x in R_1.
Return Address: Right Address of R_2.

Result: θ in R_1 and 016. $0 \leq \theta \leq$; θ is in radians.

Remarks: If $1 < |x| < 2$, routine assumes that $|x| = 1$.

If $2 < |x|$, results will be arccos $[x(\bmod 2)]$.

Restrictions: Uses Control, Floating Arithmetic and C' + and R'm sections of OFB. OFB $\sqrt{x}$ routine is used (if $|x| > \sqrt{2}/2$) and must be stored in the memory immediately following this routine.

Description: Computes $\bar{x}$ in the following manner:

If $0 \leq |x| \leq \sqrt{2}/2$, then $\bar{x} = -|x|$ and $\arccos|x| = |\pi/2 + \arcsin \bar{x}|$

If $\sqrt{2}/2 \leq |x| \leq 1$, then $\bar{x} = -(1 - x^2)^{1/2}$ and $\arccos|x| = |\arcsin \bar{x}|$

$$\arcsin \bar{x} = \bar{x} \sum_{i=0}^{9} C_i (\bar{x})^{2i}$$

where C_i are the following coefficients

C_0 = .99999 99997 15 — C_5 = .03717 68708 48
C_1 = .16666 67753 37 — C_6 = -.04439 60736 91
C_2 = .07499 30572 86 — C_7 = .16748 53422 27
C_3 = .04481 36827 32 — C_8 = -.20271 07850 45
C_4 = .02826 84745 97 — C_9 = .14636 21900 95

And finally, if $x < 0$, $\arccos x = \pi - \arccos|x|$

April 1957

17.) ORDVAC OFB Routine (6 and 9 Bit Code)

Title: OFB RKG
Runge-Kutta-Gill Solution of Systems of First Order Differential Equations.

Purpose: Compute $y_i(t_o + \Delta t)$

Number of Words: 47 4000-402F

Temporary Storage: 016-01K These positions can be used by coder in D_o sequence.

Accuracy: $0(\Delta t)^5$

Time: Varies

Results: $y_i(t_o + \Delta t) \rightarrow Y_i (1 = 0,1,2,3, \ldots n)$

Remarks: ONE ADDRESS FLOATING POINT (OFB)

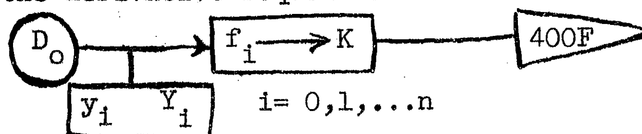
Restrictions: Assumes OFB in 040 and Arithmetic Orders.

Description: See General Description of RKG, but ignore references to scaling.

Requirements:

1. Store initial conditions $y_i(t_o)$ in Y_i($i = 0,1,2,3, \ldots n$)

2. Code the derivative sequence:



3. Enter at 4000 with the contents of the following registers as specified:

R1: $2^{-7}(n+1)$ _ B _ _ _ {Δt}

R2: _ _ _ D_o _ _ _ Exit Address _

B : _ _ Y_o _ K_o _ Q_o _

4. Exits and entrances are <u>not</u> under floating point control.

5. For successive steps enter at 4003 with contents of R1 and R2 immaterial.

6. Q_i must be set to zero initially.

NOTE: This program may be used for more than one system of equations by entering at 4000 with initial conditions whenever a change from one set of equations to another is made. Q_i must be reset to zero.

Programmer: John Wortman

9 bit code checked November 1956.

6 bit code checked April 1957.

April 1957

18.) ORDVAC Subroutine (6 Bit and 9 Bit Codes)

Code Number: 1 001 1

Title: Square Root

Purpose: Computes $\sqrt{x}$ for a given x

Number of Words: 9

Temporary Storage: 010, 011

Accuracy: $|(y^2 - x)| \leq 1.2^{-38}$

Time: 19.7 msec average (for 10 iterations)

Initial Requirements: Enter at first word.

2^{2n} x in R_1

Return Address in Right Address of R_2

Result: $2^n \sqrt{x}$ in R_1

Remarks: Halts if $x < 0$

Restrictions: None

Method: Newton-Raphson

$$y_{i+1} = \frac{x}{2y_i} + \frac{y_i}{2}$$

$$y_o = 1* = 1 - 2^{-39}$$

April 1957

19.) ORDVAC Subroutine (6 Bit Code)

Code Number: 1 001 2

Title: Faster Square Root

Purpose: Computes $\sqrt{x}$ for a given x

Number of Words: 26 words

Temporary Storage: 010-012

Accuracy: $|(y - \sqrt{x})| \leq 2^{-39}$

Time: 12.8 msec average

Initial Requirements: Enter at first word.

2^{2n} x in R_1

Return Address in Right Address of R_2

Result: $2^n \sqrt{x}$ in R_1

Remarks: Halts if $x < 0$

Restrictions: None

Method: Newton-Raphson

$$y_{i+1} = \frac{x'}{2y_i} + \frac{y_i}{2} \qquad x' = 2^{2k}x, \; 2^{-2} \leq x' < 1^*$$

$$y_o = \frac{2}{3}\left(x' - \frac{15}{16}\right) + 1^* \quad \text{for } x' < \frac{15}{16}$$

$$y_o = 1^* \quad \text{otherwise}$$

$$1^* = 1 - 2^{-39}$$

$$2^{-k}\sqrt{x'} = \sqrt{x}$$

Adapted by V. Woodward from 9 bit code by J. Wortman.

April 1957

20.)

ORDVAC Code (6 Bit)

L.W. Campbell

Code: 1 001 3

Title: Fastest Square Root

Purpose: Computes $x = +\sqrt{N}$ for given N. $(0 \leq N < 1)$

Number of Words: 41

Temporary Storage: 010-012

Accuracy: Error $< 2^{-38}$

Time: Average 9. ms. Max. 13.2 ms. Min. 3.8 ms(.8 ms if N = 0).

Initial Requirements: Enter at first word.
Initial Data: N in R_1
Return Address: Right Address of R_2

Result: $x = +\sqrt{N}$ in R_1

Remarks: Halts (at 4000) if $N < 0$.
If $N = 2^{2n} M$, result is $2^n \sqrt{M}$.
$\sqrt{1 - 2^{-39}} = 1 - 2^{-39}$

Restrictions: Uses permanent constants at 00N-00L.

Description: If N = 0, set $\sqrt{N} = 0$ and exit.
If $N \neq 0$, determine $\bar{N} = 2^{2s}N$ such that $1/4 \leq \bar{N} < 1$.
s = 0,1,2... or 19
If $1/4 \leq \bar{N} < 1/2$ compute initial approximation

$$x_o = .292 + .84375\bar{N}$$

or if $1/2 \leq \bar{N} < 1$ compute initial approximation

$$x_o = .414 + .59375\bar{N}$$

but if $x_o > 1$, use $x_o = 1 - 2^{-39}$

Use Newton-Rapshon method

$$x_{i+1} = \frac{\bar{N}}{2x_i} + \frac{x_i}{2}$$

until

$$\frac{N}{x_i} - x_i < 2^{-19} + 2^{-39}$$

Finaly use $2^{-s} x_{i+1} = \sqrt{N}$ unless $x_{i+1} \geq 1$. If $x_{i+1} \geq 1$ use $\sqrt{N} = 1 - 2^{-39}$

21.) ORDVAC Subroutine (6 Bit Code)

L. W. Campbell April 1957

Title: sin and cos $2\pi x$

Purpose: To compute sin $2\pi x$ and $2\pi x$ for given s.($-1 \leq x < 1$)

Number of Words: 77

Temporary storage: 010-014

Accuracy: Error $< 1.10^{-11}$

Time: Average 13.5 ms. Max. 15.6 ms. Min. 9.5 ms.

Initial Requirements: Enter at first word.

Initial Data: x in R_1 (x is obtained from an angle in radians by dividing by 2π or by dividing an angle in degrees by 360 and removing the scaling).

Return Address: Right Address of R_2

Result: 2^{-1} sin $2\pi x$ in R_1

2^{-1} cos $2\pi x$ in R_2

Restrictions: Uses permanent constants, zero at 00K and -1 at 00S.

Description: If $x < 0$, replace x with $x + 1$.

Compute $\bar{x} = (x - \pi/4)$ such that $-1/8 \leq \bar{x} \leq 1/8$ where n = 0,1,2,3 or 4.

Compute sin $\pi/2(4\bar{x})$ and cos $\pi/2(4\bar{x})$ by using series approximations that were obtained by using Chebyshev polynomials:

$$\sin \pi/2(4\bar{x}) = 4\bar{x} \sum_{i=0} C_i (4\bar{x})^{2i} \quad \text{and} \quad \cos \pi/2(4\bar{x}) = \sum_{i=1} D_i (4\bar{x})^{2i}$$

If $0 \leq 4\bar{x} < .07$

C_0 = 1.57079 632678	D_1 = .999999 999999
C_1 = -.645964 034300	D_2 = -1.23370 055011
C_2 = .079658 222095	D_3 = .253669 480321
	D_4 = -.020854 473715

If $.07 \leq 4\bar{x} \leq 1/2$

C_0 = 1.57079 632679	D_1 = 1.00000 000000
C_1 = -.645964 096155	D_2 = -1.23370 055013
C_2 = .079692 582744	D_3 = .253669 507157
C_3 = -.004681 266367	D_4 = -.020863 468006
C_4 = .000158 206524	D_5 = .000919 161752
	D_6 = -.000024 850988

Then if $3/8 < x \leq 7/8$, replace cos $\pi/2(4\bar{x})$ with -cos $\pi/2(4\bar{x})$ and if $1/8 < x \leq 3/8$ or if $5/8 < x \leq 7/8$, use sin $2\pi x$ = cos $\pi/2(4\bar{x})$ and cos $2\pi x$ = sin $\pi/2(4\bar{x})$.

22.)

ORDVAC Subroutine (6 Bit Code)

L. W. Campbell April 1957

Title: Arcsin-cos

Purpose: To compute Θ = arcsin-cos for given $2^{-1}\sin\Theta$ and $2^{-1}\cos\Theta$

Number of words: 68

Temporary storage: 010-015

Accuracy: Error $< 1.10^{-11}$

Time: Average 15.7 ms. Max. 17.6 ms. Min. 7.5 ms.

Initial Requirements: Enter at first Word.

Initial Data:

X X Add. of $2^{-1}\sin\Theta$ X X Add. of $2^{-1}\cos\Theta$ in R_1

Return Address: Right Address of R_2

Result: $2^{-2}\Theta$ in R_1 and 010. ($-\pi < \Theta \leq \pi$; Θ is in radians.)

Remarks: If it is desired that sin and cos not be scaled 2^{-1}, words at 4008 and 4032 may easily be changed so that no scaling is assumed on sin and cos. $\sqrt{2}/2 < \sin\Theta$ or $\sqrt{2}/2 \leq \cos\Theta$ is necessary to obtain a valid result.

Restrictions: None

Description: Determine $\bar{x}$ and Θ in the following manner:

If $0 \leq |\sin\Theta| \leq \sqrt{2}/2$, $\bar{x} = -|\sin\Theta|$ and $\Theta = |\arcsin\bar{x}|$

or if $\sqrt{2}/2 < |\sin\Theta|$ $\bar{x} = -|\cos\Theta|$ and $\Theta = |\pi/2 + \arcsin\bar{x}|$

$$\arcsin\bar{x} = \bar{x}\sum_{i=0} C_i(\bar{x})^{2i}$$

where C_i are the following coefficients which were obtained by using Chebyshev polynomials:

If $|\bar{x}| < .14$

C_0 = .999999 999966 C_2 = .074985 007367

C_1 = .166666 725638 C_3 = .045858 458705

or if $.14 \leq |\bar{x}|$

C_0 = .999999 999988 C_6 = .001021 611887

C_1 = .166666 670196 C_7 = .086081 512284

C_2 = .074999 672367 C_8 = -.195690 652747

C_3 = .044654 712853 C_9 = .387032 895891

C_4 = .030161 652364 C_{10} = -.393001 749282

C_5 = .024768 742062 C_{11} = .205529 674087

Then if $\cos\Theta < 0$ replace Θ with $\pi-\Theta$ and then if sin <0 replace Θ with $-\Theta$.

23.)

ORDVAC Subroutines (6 Bit Code)

April 1957

L. W. Campbell

Title: $\text{Log}_e\ x$

Purpose: Compute natural logarithm for given x.

Number of Words: 50

Temporary Storage: 010-014

Accuracy: At least ten significant digits. (if result is properly scaled)

Time: Average 14. ms. Max 20. ms. Min 12. ms.

Initial Requirements: Enter at first word.

Initial Data: $2^{-F_i}\ x$ in R_1 and $F_i \cdot 2^{-39}$ in the second word (4001) $F_i = 0$ unless it is changed. (Note: if x is scaled 2^{-2}, $F_i = +2$, if x is scaled by 2^4, $F_i = -4$.) $F_o \cdot 2^{-39}$ must be set in the third word (4002) so that result will be scaled $2^{-F_o} \log_e x$. $F_o = 0$ unless changed and $F_o \geq 0$ only.

Return Address: Right Address of R_2

Result: $2^{-F_o} \log_e x$ in R_1 ($F_o \cdot 2^{-39} \geq 0$ is at 4002)

Remarks: Halts (at 4005) if $x \leq 0$.
Halts (at 4016) if $F_o < 0$.

Restrictions: Uses permanent constants at 00K-00L.

Description: Normalize $2^{-F_i}\ x$ to obtain $2^{-\bar{F}_i}\ \bar{x}$ where $1/2 \leq \bar{x} < 1$.

Compute $\dfrac{3/2\ \bar{x} - 1}{3/2\ \bar{x} + 1} = y$

Then compute

$$\text{Log}_e\ 3/2\ \bar{x} = y \sum_{i=0}^{4} C_i\ y^{2i}$$

Where C_i are coefficients obtained by using Chebyshev polynomials

C_0 = 2.00000 000004

C_1 = .666666 616911

C_2 = .400009 896127

C_3 = .285027 989462

C_4 = .241467 958353

And finally

$$2^{-F_o} \log_e x = 2^{-F_o} (\text{Log}_e\ 3/2\ \bar{x} + \text{Log}_e\ 2/3 + \bar{F}_i \log_e 2)$$

24.) ORDVAC Subroutine (6 Bit Code)

L. W. Campbell May 1957

Title: Arctan x

Purpose: Computes θ = arctan x for given x ($-2^{39} \leq x < 2^{39}$)

Number of words: 73

Temporary storage: 010-015

Accuracy: Error $< 1.10^{-11}$

Time: Average 13.5 ms. Max. 15.8 ms. Min. 7.1 ms.

Initial Requirements: Enter at first word.

Initial Data: $2^{-s}x$ in R_1.

$s \cdot 2^{-39}$ set in the second word (4001).

$39 \geq s \geq 0$ only and s = 0 unless it is changed.

(If have $2^{-3}x$, s = 3).

Return Address: Right Address of R_2.

Result: $2^{-1}\,\theta$ in R_1 and 010. ($-\pi/2 \leq \theta \leq \pi/2$; θ is in radians)

Remarks.: Halt (at 4004) if $s < 0$.

Restrictions: Uses permanent constant -1 at 00S.

Description: Compute $\bar{x}$ in the following manner:

If $0 \leq |x| \leq \sqrt{2} - 1$ then $\bar{x} = |x|$ and $\arctan |x| = \arctan \bar{x}$

If $\sqrt{2} - 1 < |x| \leq \sqrt{2} + 1$ then $\bar{x} = \frac{|x| - 1}{|x| + 1}$ and $\arctan |\bar{x}| = \pi/4 + \arctan \bar{x}$

If $\sqrt{2} + 1 < |x|$ then $\bar{x} = -\frac{1}{|x|}$ and $\arctan |x| = \pi/2 + \arctan \bar{x}$

$$\arctan \bar{x} = \bar{x} \sum_{i=0} C_i (\bar{x})^{2i}$$

where C_i are coefficients obtained by using Chebyshev polynomials.

If $|\bar{x}| < .12$

$C_0 = .999999\ 999964$ $\quad C_2 = .199971\ 825718$

$C_1 = -.333333\ 252414$ $\quad C_3 = -.139709\ 382531$

If $.12 \leq |\bar{x}|$

$C_0 = .999999\ 999999$ $\quad C_4 = .111035\ 431503$

$C_1 = -.333333\ 332784$ $\quad C_5 = -.089933\ 027844$

$C_2 = .199999\ 931985$ $\quad C_6 = .069774\ 480166$

$C_3 = -.142853\ 916672$ $\quad C_7 = -.037710\ 838763$

and finally, if $x < 0$ replace θ(θ = arctan $|x|$) with $-\theta$.

April 1957

25.) ORDVAC Routine (6 Bit Code)

Title: Fixed Point RKG
Runge-Kutta-Gill Solution of Systems of First Order Differential Equations

Purpose: Compute $y_i(t_o + \Delta t)$

Number of Words: 99 (incl. preset box) 4000-4062

Temporary Storage: After initial entry, positions 4042+n thru 4062 are unused. 010 used in preset only.

Accuracy: $0(\Delta t)^5$

Time: Varies, function of n and derivative sequence.

Results: $y_i(t_o + \Delta t) \rightarrow Y_i$

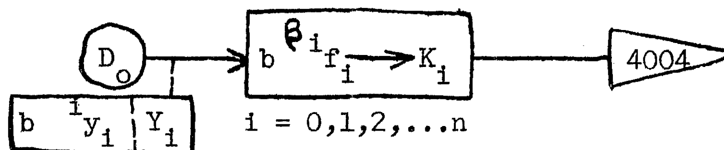
Remarks: FIXED POINT

Restrictions: $|s_i|$ should be minimized.

Description: See General Description of RKG.

Requirements:

1. Store initial conditions $b^{\alpha_i} y_i$ in Y_i
2. Store scaling indicators $\pm 2^{-39}(s_i)$ in S_i (non-compliments for $s_i < 0$)
3. Code the derivative sequence:



4. Enter at 4000 with the contents of the following registers as specified:

M1: $2^{-1}f$ _ Y_o _ K_o _ D_o

R1: 0 $\{b^m \Delta t\}$ $\{1/b\}$ _ S_o

R2: $2^{-7}n$ _ M_1 _ _ _ Exit Address

Set f = 2 if s_i varies

f = 1 if s_i = constant $\neq 0$ } for all i

f = 0 if $s_i = 0$

5. For successive steps transfer to 4000 with contents of R1 and R2 immaterial.

Programmer: Viola Woodward

9 bit code covered by other writeup containing General Description of RKG

April 11, 1956

26.)

ORDVAC IBM CARD ROUTINES

L. W. Campbell

These routines have been programmed according to the specifications that were stated at meetings of programmers on 20 January and 2 February 1956.

There are two separate routines, an Input routine and an Output routine and each may be stored as subroutines anywhere in the memory. They are used in essentially the same manner. They have arbitrary fields with the limitation of eleven decimal digits per number. Numbers are automatically converted (or reconverted) and the output routine automatically rounds each number by adding $.5 \times 10^{-p}$ where p is the number of decimal digits in the number to be printed. There is provision made for skipping arbitrary columns. The output routine will automatically print an identification number and/or a counter on each card if it is desired.

Entrance is made with a word in R_1 and the return address on the right of R_2 (or later in R_3.)

R_1	R_2
_N_ _M_ xx _Ao_	xxxxx xx _RA_

where M = the total number of numbers that are to be read (or printed) on the cards. Routine reads (or prints) enough cards to convert (or reconvert) M numbers. Ao = the initial address at which M numbers are to be stored (or printed from.) Numbers are stored in (or printed from) <u>consecutive</u> memory positions unless the address advance constant is changed before entering the routine. N = the number of numbers after which the next number starts a new card and starts using the field words from the beginning. If N = 0, it is ignored.

RA = Return Address.

M and N are both sexadecimal numbers.

The field words tell the routine how many columns are in each field. They also indicate how many columns to skip, when and where to print the identification and/or counter, and when to start a new card and start

using the field words from the beginning again. Each sexadecimal in a field word has a meaning as listed below. There may be as many field words as are desired but the storage space listed for the routine includes space for only two field words and these are the last two words. Adding more field words will lengthen the routine.

Meaning of sexadecimals in field words:

0 - Go to the next field word. (The routine also does this automatically after decoding all ten sexadecimals of any one field word.)

1 thru S - Convert (or Reconvert) this many decimal digits and store (or print) as a number. A number uses the same number of columns as decimal digits for double punched signs, but uses one additional column for single punched signs.

N - On output punch card as stands; on input ignore rest of card. Go back to first field word for next card.

Jq - Skip q columns where q is the next sexadecimal. $1 \leq q \leq L$

Fq - Not used by the Input Routine. Routine will Halt unless using no double punch option. The Output Routine will print the sum of C_1 and C_2 in q columns where q is the next sexadecimal. $1 \leq q \leq S$ The sum of C_1 and C_2 is restored in C_1 so that if C_2 contains 1×10^{-q}, the routine will automatically print a card counter on the cards. C_1 and C_2 are the second and third words of the output routine.

L - Not used by either routine. Both will Halt.

When the output routine has reconverted N numbers or M numbers, it will search the remainder of the card (if there is any left) for an F in the field words. If it finds an F, it will print the sum of C_1 and C_2 in the same columns as it did on the preceding cards.

Fields may go across the middle of a card and they may go from the end of one card onto the next card. The routines do <u>not</u> start the field words over at the beginning of each card unless told to do so by using N either in a field word or in the entrance word.

Signs: The input routine uses either an X or a Y punch as a minus sign. The output routine normally prints Y punches for minus signs but may be made to print X punches instead by using an option card. Nothing is printed for plus signs. The routines normally read and print double punched signs with the sign over the first decimal digit.

Options:

Both Routines:

No double punch: Signs are in separate columns.

Count cards: M specifies the total number of cards to be read or (printed) instead of total number of numbers.

Output Routine only:

X minus signs: prints X's instead of Y's for minus signs.

None of these options add to the length of the routine. Also available are cards that allow the field words and counter to be changed easily.

The input routine may easily be changed to give integer conversion.

The output routine will not reconvert - 1 correctly, it will give -66666 66666 6.

Both routines use the permanent constants stored at 00K - 00L.

Lengths of Routines:
Input: 105 words (4000-4068)
Output: 128 words (4000-407L)

Temporary Storage:
Input: 010-027 and 034-03L
Output: 010-033 and 03F-03L

Location of field words:
Input: words 104 and 105 (4067-4068)
Output: words 127 and 128 (407F-407L)

Location of Counter in output routine:
C_1 is word 2 (4001)
C_2 is word 3 (4002)

Location of Address advance constant (in right address only):
Input: word 89 (4058)
Output: word 98 (4061

Accuracy:

Input: Maximum error is 2^{-39} (Last bit is always a 1 except that zero is exact.)

Output: Maximum error is $.5 \times 10^{-K} + 2^{-38}$ where K is the number of decimal digits printed.

The field words are set at KKKKK KKKNO which is 8 fields of 10 decimal digits each. The counter in the output routine is set at $C_1 = 0$, $C_2 = 1 \times 10^{-10}$

The output routine described here is not a double speed routine.

This routine is available for both 6 bit and 9 bit order codes.

September 5, 1956

27.) DOUBLE SPEED IBM OUTPUT ROUTINE

L. W. Campbell

This routine is used the same as the IBM output routine described in write-up dated 11 April 1956 and it is intended that this routine should replace that routine. The advantage of this routine is that it will print cards at the maximum rate of the printer (100 cards/minute) where as the other routine will print about 50 cards/minute. The speed is gained only if programmers code so that more than one card is punched with one entrance to the routine. The speed is gained by doing the reconversion for the next card in between the rows of the card it is printing. It always prints <u>all</u> the cards each time it is entered, i.e. there is <u>not</u> one card left to print at the end of the problem.

The following three things are different than what was described for the slower routine:

Length:

144 words (4000-408L)

Temporary Storage:

010-03L and 4080-408S

Location of field words:

words 143 and 144 (408F-408L)

The timing is such that the routine will work properly unless a lot of <u>very</u> short fields are used. The number of fields included in any 8 consecutive columns should not be more than 3 to insure correct punching.

April 1957

28.) ORDVAC Code (6 Bit)

K. B. Betz and L. W. Campbell

Title: Order Pair Routine (OP)

Purpose: To perform any specified ORDVAC pair of orders a specified number of times and using a variable advance of the two addresses.

Number of Words: 9

Temporary storage: 010-011

Time: .6+n(.5 + order pair time) ms.

Initial Requirements: Enter at first word.
Initial Data: Order pair to be done in R_1

Left Add. Adv.	n	Right Add. Adv.	RA	in R_2

where n = number of times order pair is to be performed.
RA = Return address.

This routine is useful to clear a block of the memory or to move a block of data or code from one section of the memory to another section.

21 March 1957

29.)

PRINTING ADDRESS SEARCH

No. of Words: 21

Temporary Storage: 0010

Initial Requirements: Enter at first word.
Tape in tape reader with each word containing the address being searched for in both LEFT AND RIGHT address positions: 00xxx 00xxx

Result: Two words printed on teletype:

yyy00 00000 Address of word containing address being searched for.

----- ----- Contents of yyy

Remarks: If the Stop Disable switch is up, the machine will stop after the whole memory has been searched once.
If the switch is flipped to Start, the next word will be read from the tape and the memory searched for that address.

Note: There will be at least 2 pairs of prints for addresses within the routine itself.

USE 5-Key Binary Input.

This routine is a modification of the routine coded by Home' Reitwiesner.

30.)

ORDVAC DMO (DRUM AND MEMORY IBM PRINT OUT)

Lloyd Campbell

28 September 1955

This routine will print out the contents of specified tracks of the drum and specified portions of the memory on binary punched cards with necessary key words so that the deck of cards can be read back into the machine with a non-modifying input routine.

The routine reads a variable number of words from tape. The first group of words contains information about what tracks of the drum are to be printed. Each word consists of two half words of the form:

± _n_ Tk. No._

where n = number of consecutive tracks to be printed (n ≤ N6)

Tk. No. = The initial track number of the n tracks. Each half word must be positive except the last one which must be negative. If there is nothing to be printed from the drum, a tape word of 8000000000 will cause the drum print to be omitted.

The second group of words contain the addresses of the sections of the memory that are to be printed. Each word is of the form:

± x _A_ x x _B_

where A ≤ B ≤ LLL. The contents of the memory from A to B will be printed. Each such word must be positive except the last one which must be negative. If there is noting to be printed from the memory, a tape word of 80000 00000 will cause the memory printing to be omitted.

The last tape word is of the form:

x x _R A_ x x _C_

where R A = Return Address
C = The address that is to be printed in a transfer of control key word.

A card with a transfer of control key work on it is printed after the other printing is completed.

The routine assumes that it is to be executed at the time it is read into the machine. It must be read in with a modifying input routine.

The routine occupies no apparent memory space except 004 (or any one easily changed position) and, of course, a regular input routine. It uses tracks N6-J0 on the drum as temporary storage for the contents of the memory from 000 to 20L and this routine then uses that space to store itself and accomplish the printing. When finished printing, it restores the memory to what it held before this routine was read in (except for 004 and the input routine storage).

When printing the contents of the drum or of the memory, only the first of a group of consecutive positions that contain LLLLL LLLLL will be printed, i.e., "cleared" positions will not usually be printed.

Each card printed will have a store key word as its initial word. Unused portions of printed cards will be filled with 80000 00000 words.

When information that has been printed from the drum is to be read back into the drum, an input routine that includes recognition of the 80004 00 n key word must be used. The input routine must not use 020-19L of the memory as this is used to temporarily store the drum information.

It should be noted that if "cleared" tracks, or tracks that are partly "cleared", have been printed, they will not necessarily be "cleared" upon reading the information back into the machine. To be certain that cleared tracks will remain "cleared", it is necessary to insert a card that clears 020-19L of the memory after each 4 type key word. This will usually not be neceśary.*

Example 1: Tape:

00212 00K7F
801K0 00000
00040 00160
80200 002LL
00120 00200

This tape directs the printing of two tracks beginning at track 12, ten tracks beginning at 7F, and track no. K0. Then it directs the printing of the memory contents from 040 to 160 and from 200 to 2LL. Finally, it directs the printing of control card to 200 and then transfers control to address 120.

Example 2: Tape:

80000 00000
80100 003JF
00100 003L1

This tape directs the printing of nothing from the drum, the memory contents from 100 to 3JF, and a transfer of control card transferring to 3L1. After the printing is completed, control will be transferred to 100.

*"cleared" means containing LLLLL LLLLL on the drum or in the memory.

31.) ORDVAC Sexadecimal Print (6 Bit Code)

L. W. Campbell   August 1957

Title: Sexadecimal Print

Purpose: To print out the contents of specified memory positions on IBM cards in a form that can be tabulated on the 407 or 402 tabulators.

Number of Words: 90

Initial Requirements: Enter at first word.
Tape in tape reader with intervals to be printed specified:

0 0 A_1 0 0 B_1

0 0 A_2 0 0 B_2

etc.

8 x xxx x x R.A.*

To use as SUBROUTINE: Remove last card (card 6) from deck and transfer to first word with the following contents of R1 and R2:

x x A x x B in R_1

x x xxx x x R.A.* in R_2

Result: The first card printed will have the interval:

0 0 A 0 0 B in first 10 columns, rest of card blank.

The remaining cards will have contents of that interval in sexidecimal form, 8 ten digit fields per card.

If the 5th card is removed from the deck the sexadecimal cards will be printed with only 4 sexadecimal words per card - suitable to be printed on the 402 tabulator.

Remarks: Since punching a lot of 0's or L's (blank memory positions) in card tends to jam the printer, such fields will be printed with alternating columns blank:

L B L B L B L B L B or O B O B O B O B O B

where B = blank column

* R.A. = Return Address

32.)

FLOATING DECIMAL TO FIXED DECIMAL ROUTINE
(WITH DECIMAL POINTS PUNCHED ON CARDS)

J.V. Lanahan

I. It was assumed, when this program was written, that it would be applied to standard floating decimal output. It may be adapted easily to any other card format.

II. Card-punched decimal points are optional.

III. The program works as follows:

Given sets of n cards, each card punched with 6 or less floating decimal numbers the user will:

1. Punch a set of n cards (numbered 1 to n in columns 74 and 75) which will be placed on the front of the deck of cards to be converted. The first card will have punched in columns 2 and 3 the maximum "10" exponent which is associated with the first floating decimal number of the first card of the n cards to be converted. The maximum "10" exponents of the succeeding numbers on the card are to be punched in columns 11 and 12, 14 and 15, 23 and 24, 26 and 27, 35 and 36. The succeeding cards of the n cards will be punched similarly.
   On the first card, in addition to what was listed above, n will be punched in columns 47 and 48; and if there be an item on the first card of a set which the user would wish punched on each card (Time, for instance) the number (1 to 6) of the item should be punched in column 39.

2. To specify the location of decimal points to be punched on the output cards the user must place a binary punched card in the program. This card will have punched in it
   (i) in column 1 $\div$ y, x, 0. In column 41 $\div$ y, x, 0. In column 74 $\div$ 0.
   (ii) "1" punches to indicate the columns of the 1st of the n cards in which decimal points are to be punched; "2" punches to indicate the columns of the 2nd of the n cards in which decimal points are to be punched; etc. 7 decimal points per card must be indicated unless a change is made in the output-card format.

IV. The output is as follows:

Each number (including sign and decimal point if desired) uses ten columns of the card. The numbers will be punched in the same order as they appear on the input card. The identification item (see III) will be punched in columns 61 - 70. Each card will have punched in columns 71 and 72 its number (1 to n) in the set. The first digits of the numbers with the largest "10" exponent will be punched in the first column of the 10 column field (unless this column is occupied by a decimal point in which case the first digit appears in the second column) and numbers with smaller "10" exponents will be shifted right as necessary.

V. If no decimal points are to be punched one card (so indicated) must be removed from the program.

VI. If the user would change card formats he will need the following information:

1. This is a fixed point code.
2. The field words for the IBMI are in 271&280.
3. The field words for the IBMO are in 312&313.
4. The card format for the n instruction cards and the data cards is the same and is the format of the data cards + one field (containing the instruction card No.) The field containing the instruction card No. must be blank on the data cards.

DISTRIBUTION LIST

No. of Copies	Organization
3	Chief of Ordnance Department of the Army Washington 25, D. C. Attn: ORDTB - Bal Sec(2 cys) ORDTX-AR (1 cy)
10	British Joint Services Mission 1800 K Street, N. W. Washington 6, D. C. Attn: Mr. John Izzard, Reports Officer
4	Canadian Army Staff 2450 Mass. Ave., N. W. Washington 8, D. C.
3	Chief, Bureau of Ordnance Department of the Navy Washington 25, D. C. Attn: ReO
1	Commander Naval Proving Ground Dahlgren, Virginia
1	Commander Naval Ordnance Laboratory White Oak Silver Spring 19, Maryland
1	Commander Naval Ordnance Test Station China Lake, California Attn: Technical Library
1	Director Naval Research Laboratory Anacostia Station Washington 20, D. C.
1	Chief, Bureau of Aeronautics Department of the Navy Washington 25, D. C.
1	Commander U.S. Naval Air Missile Test Center Point Mugu, California
10	Director Armed Services Technical Information Agency Documents Service Center Knott Building Dayton 2, Ohio Attn: DSC - SD
1	National Bureau of Standards National Applied Mathematics Laboratory Computation Laboratory Washington 25, D. C. Attn: Franz L. Alt
1	Director National Security Agency Washington 25, D. C. Attn: R/D 36, Chief, Engineering Research Division
1	Commanding Officer Flight Determination Laboratory White Sands Proving Ground Las Cruces, New Mexico Attn: John Titus
1	Engineering Research Associates Division of Remington Rand, Inc. 1902 W. Minnehaha Avenue St. Paul, Minnesota
1	Harvard University Computation Laboratory Cambridge 38, Massachusetts Attn: Prof. H. Aiken
1	International Business Machines Corp. Engineering Laboratory Poughkeepsie, New York Attn: E. R. Lancaster, Advanced Engineering Education Dept.

DISTRIBUTION LIST

No. of Copies	Organization
1	Moore School of Electrical Engineering University of Pennsylvania Philadelphia, Pennsylvania
1	Oregon State College Department of Mathematics Corvallis, Oregon Attn: W. E. Milne
1	Princeton University Mathematics Department Princeton 1, New Jersey
1	Raytheon Manufacturing Co. P. O. Box 398 Bedford, Massachusetts
1	Remington Rand Univac Div. Sperry Rand Corp. 1900 W. Allegheny Ave. St. Paul, Minnesota Attn: Mr. Sam Howry, Systems Analysis
1	Stanford University Department of Mathematics Stanford, California Attn: Gabor Szego
1	University of California 942 Hilldale Avenue Berkeley, California Attn: D. H. Lehmer
1	R. F. Jackson University of Delaware Newark, Delaware
1	University of Illinois Department of Mathematics Urbana, Illinois Attn: A. H. Taub
1	Professor J. P. Nash University of Illinois Electronic Digital Computer Project Urbana, Illinois
1	Professor A. A. Bennett Department of Mathematics Brown University Providence 12, Rhode Island
1	Professor H. H. Goldstine Department of Mathematics Institute for Advanced Study Princeton, New Jersey
1	Dr. L. H. Thomas Watson Scientific Computing Lab. 612 W. 116th Street New York 27, New York
1	Chief Signal Officer Department of the Army Washington 25, D. C. Attn: Mr. G. H. McClurg SIGRD-6

Army--Aberdeen Proving Ground, Md--D