THE SIMPLE CODE FOR ZEBRA

by

Dr. ir. W. L. VAN DER POEL

681.142 ZEBRA

I. INTRODUCTION

ZEBRA is an electronic digital computer logically designed in the Dr. Neher Laboratory of the Netherlands Postal and Telecommunications Services and technically developed and constructed by Standard Telephones and Cables Ltd, England. It has been the object of the designers to build a machine of a simple construction which is most reliable and requires a minimum of service, while at the same time a high speed and great flexibility of programming are maintained. It is quite obvious that sacrifices had to be made for this simplicity. Thus in general it is true that programming (drawing up a list of detailed instructions for the machine) is somewhat more difficult for ZEBRA than for most other machines. For example multiplication is not a built-in feature of the machine. Thus even the elementary arithmetic operation must be programmed. On the other hand the flexibility of the basic programming is very great and the speed which can be attained by skilled programmers is relatively high. Nevertheless, programming in the real code of the machine is still a job which requires a good deal of very special training.

To make an automatic computer more easily accessible to those who occasionally have calculations to make and who do not want to devote too much time to learning the real code of ZEBRA, another code called SIMPLE CODE has been developed.

Machines like ZEBRA are not only able to make calculations but they can also do all kinds of work which can be expressed in formal logical rules, such as translation of one code into another. In this way complicated operations can e.g. be expressed in Simple Code as one single instruction, while in the machine this operation is translated into a whole set of real code instructions. The code is translated behind the scenes by a very complicated programme, called an interpretive programme. In this article we shall not deal with the real code nor with any of the actual components of the machine, but we shall exclusively treat the Simple Code as if it describes the properties of quite another machine in which all the features of Simple Code have been built in. The user does not have to bother about the real machine and when he keeps strictly to the rules of Simple Code he can use the machine in a very effective way after only few hours of study. In view of the aforegoing it is hoped that many more people can and will use ZEBRA, and keep a better contact with the numerical results of their problems.

In some places of this article consideration will be given to the co-operation between real code and Simple Code, and for the sake of completeness something will be said about the background of real code, but occasional users can completely ignore these points as they will only be needed for very advanced programmers.

II. PROGRAMMING

Programming is drawing up a detailed list of instructions telling the computer what to do to solve a problem. This does not only include instructions for the effecting of the required arithmetical operations but also instructions to tell the machine how many numbers are to be read, how the lay-out of the printed results is to be made etc. This part of a problem is sometimes far more difficult to tackle than the actual arithmetic, but for the time being we shall first devote our time to arithmetic coding.

One of the first difficulties of using a computer (not only electronic computers but also mechanical desk calculators) is the problem of capacity of the counters. Many computers require that all numbers appearing in the calculation should lie in the range $-1 < a < +1$ and should have say 9 decimals precision. It is quite a difficult task to secure that all variables lie within this range. E.g. the number $\pi = 3.1415\cdot$ cannot be represented but must be expressed by giving $\pi - 3$ or $\frac{1}{10}\pi$. This difficulty can be overcome by using numbers in floating point. In this system all numbers will be written as

$$a \times 10^{b}$$

where $0.1 < |a| \leqq 1$ and b is an integer. a is called the mantissa and b the exponent.

E.g. 35.67	can be written as	$+0.3567 \times 10^{2}$
—0.0032	,, ,, ,, ,,	-0.3200×10^{-2}

In the next paragraph a short account of arithmetic in the floating system will be given.

A second difficulty of programming is the repetition of a process for a number of times together with keeping track of running indices. In real code this is often a point where mistakes can easily be made. Moreover, running indices require variable instructions, i.e. calculation with the instructions themselves, and this again is far beyond the scope of occasional users. In Simple Code special provisions have been made for repeating and counting, and also for making variable instructions. In no case are the instructions written down by the programmer variable and he never has to make calculations with instructions. Counting will be dealt with extensively in the respective chapters.

As no operations are done on instructions it was thought better to separate the two kinds of information. The Simple Code machine has a separate store for numbers (about 1200) and a separate store for instructions (also about 1200). This is especially useful for the beginner. Later on it will appear that in case of lack of capacity in the number store, the instruction store can also be used for numbers and inversely. The locations in the number store can each contain one number (floating). The locations are numbered from 0 onwards and the numbers of these locations are called the addresses of the respective locations. The location *n* itself will often be referred to as the address *n*. The locations in the instruction store need not be numbered as the instructions are put into this store from the beginning, and they are executed consecutively.

When the instruction store is used for numbers the absolute addresses in the instruction store will be written as *n*·; the point indicates that not address *n* in the number store is meant, but *n* in the instruction store. Cf. Chapter XIII.

An instruction consists of two parts: an operation part specifying the type of operation to be effected (addition, multiplication, etc.) and the address telling the machine where to find the operand in the store. The operation is denoted by a letter and the address follows the letter (e.g. *A*311 or *H*2). This sort of code is called a one-address-code.

III. FLOATING NUMBERS

All numbers processed by the arithmetic instructions are floating numbers in the machine. Some of the basic arithmetic rules for floating numbers will be outlined in this chapter.

A number can of course be written as follows:

$$35.79 = +\,0.003579 \times 10^4$$

In that case it is floating but non-normalised. Thus the available capacity of the mantissa will not be used completely. The general rule is that all numbers within the machine will be automatically normalised ($0.1 < |\text{mantissa}| \leq 1$). The appearance of an unnormalised number on paper certainly points to bad functioning of the machine or a mistake in the programme.

All numbers in the machine have a precision of 9 decimal digits in the mantissa and 3 in the exponent. When results are produced in floating form the number will be printed as ± 0.xxxxxxxxx ± xxx, in which the signed fraction denotes the mantissa and the signed integer denotes the exponent.

Adding floating numbers is effected in the following way. First the exponents must be made equal. Then the addition of mantissa can be effected.

$$\begin{array}{lrlll}
\text{E.g.} & 22.67 = & +\,0.226700000\ +2 & = & +\,0.226700000\ +2 \\
 & 3.31 = & +\,0.331000000\ +1 & = & \underline{+\,0.033100000\ +2} \\
 & & & & +\,0.259800000\ +2
\end{array}$$

In case that the difference between two exponents is more than 9, one of the mantissae is shifted over more than 9 digits so that nothing is left for the addition. Then the actual addition need not be done at all.

Another case arises in the following situation:

$$\begin{array}{lll}
+\,0.570000000 & +1 & (=5.7) \\
\underline{+\,0.740000000} & \underline{+1} & (=7.4) \\
+\,1.310000000 & +1 &
\end{array}$$

Now the mantissa is too large and the number must be shifted into the form

$$+\,0.131000000 \quad +2 \quad (=13.1)$$

As can be seen in this example, one digit of the precision is lost at the right hand side of this number.

Loss of precision is even more apparent in cases where one number is positive and the other is negative, or in the case of subtraction of two positive numbers.

$$\begin{array}{lrl}
\text{E.g.} & +\,0.632421111 & -3 \\
 & +\,0.632311111 & -3 \\
- & \overline{+\,0.000110000} & -3 \\
\text{normalised} & +\,0.110000\boxed{000} & -6
\end{array}$$

The 3 zeros on the right hand side are shifted into the register but they have no significance for the precision of the number. This loss of precision is one of the serious dangers of the floating number system. Only a mathematical treatment of the problem can reveal this loss of precision.

For many practical cases the difficulty does not arise at all and we shall not deal with the mathematical difficulties of floating arithmetic in the rest of this article any more.

A peculiar case is the subtraction of two equal numbers.

$$\begin{array}{rrr} & +0.632000000 & +3 \\ - & +0.632000000 & +3 \\ \hline & +0.000000000 & +3 \end{array}$$

The result can never be normalised. In other words: the number 0 cannot be represented in the floating system. In that case the machine will automatically supply the number $+1.000000000 \times 10^{-999}$, a very small number indeed, which plays the rôle of 0 in an effective way. The result of a subtraction of two equal numbers will always be $+10^{-999}$, never -10^{-999}. This is of importance for the test instruction. (Cf. Chapter VIII.)

Multiplication of two numbers in the floating system is done in the following way:

$$\begin{array}{rrr} +0.350000000 & +3 & (= \quad 350) \\ +0.440000000 & +2 & (= \quad 44) \\ \hline +0.154000000 & +5 & (= 15400) \end{array}$$

In words: The mantissae are multiplied while the exponents are added. The product of the mantissae could be smaller than 0.1. In that case the product is normalised.

$$\begin{array}{lrr} & +0.350000000 & +3 \\ & +0.220000000 & +2 \\ \hline & +0.077000000 & +5 \\ \text{normalised} & +0.770000000 & +4 \end{array}$$

Because in the multiplication a double length product is formed from two single length numbers, the normalisation does not result in a loss of precision. After the normalisation the product is rounded off to 9 decimals.

In case of a division there is the same difficulty before the actual division can be started. When the dividend is greater than the divisor, the quotient would be too large. Hence the dividend is first shifted to the right in that case.

E.g.

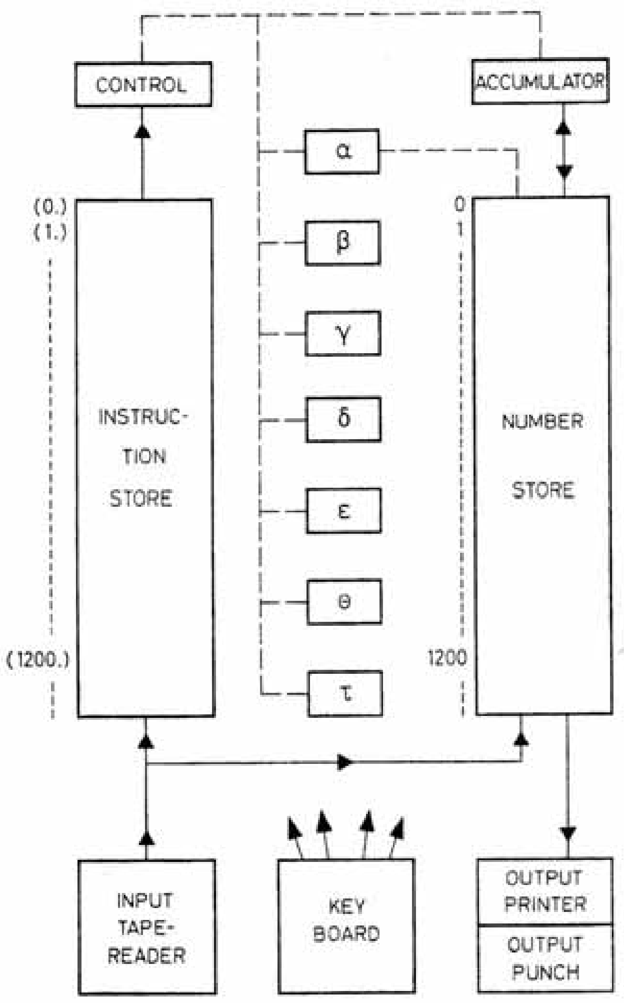
$$\frac{+0.300000000 \quad +5}{+0.200000000 \quad +2} = \frac{+0.030000000 \quad +6}{+0.200000000 \quad +2} =$$
$$= +0.150000000 \quad +4$$

The mantissae are divided, the exponents are subtracted.

Of course the user needs not know this because all actions are automatic. But it can give some more insight into the particular difficulties of calculation in floating point. A single instruction for one addition thus appears to be in reality a very complicated set of more elementary operations in real machine code.

IV. THE SIMPLE CODE MACHINE

When it operates with Simple Code, ZEBRA can be regarded as a completely different machine. We shall now enumerate its most important parts as they will play a rôle in the following chapters.



1. The number store containing about 1200 locations for floating numbers. The locations are numbered from 0—1200. They are called addresses.
2. The instruction store containing about 1200 locations for instructions. The locations need not be numbered but sometimes it is useful to refer to them as 0·—1200·.

3. The accumulator. This is a special location with a capacity of one number. In the accumulator all arithmetic operations are performed. It will be abbreviated by *A*. When we speak of the contents of the accumulator we shall write this as (*A*). In the same way the contents of location *n* or *n*· are denoted by (*n*) or (*n*·) respectively.
4. A number of small registers for counting and other special purposes. They are denoted as follows:

α	main counting register	All these registers contain integers, not floating numbers
β	second counting register	
γ	increment of second count	
δ	safety register for main counting register	
ε	register for count limit	
θ	return instruction from count cycle	These registers contain a jump instruction of the kind *Xp*
τ	return instruction from jump	

The manner in which all these registers can be used is stated in the following paragraphs.

5. The input tape reader, a photoelectric tape reader for reading data into the machine. This reader can accept tapes of the same kind as normal 5-hole teleprinter tape. Instructions as well as numerical information are fed into this reader.
6. The output printer for printing the results of a calculation. It is a normal 7 char/s teleprinter, used for slow output.
7. The output punch, required to punch the results of the machine in the form of teleprinter tape, which is done at 50 char/s. The resulting tape can be printed off-line. (I.e. on a separate printer, not directly connected to the machine.) This is used for fast output. The subsequent writing out on off-line printers does not delay the operation of the computer.
8. The control. This part is the central governing organ. Its action consists of taking consecutive instructions out of the store and executing them in the accumulator. Input and output are also under the direction of the control.
9. The keyboard. This is in fact a manual control of the operator on the machine.

There are the following keys:

Clear	: Stop the machine and put it in the position to receive a new problem or a new batch of data.
Start	: Start programme or input of numerical data or go on when the machine has stopped because of a stop instruction.
Key U1	: A key which can be given any meaning by programming a branch instruction asking for the position of the key.
Key U2	: The same as for key U1.
Key U3	: Not in use.
Key U4, U5, U6	Used for cutting open, a diagnostic aid for printing intermediate results necessary for error detection.
Key U6	: When put off, clearing and starting will make the machine read a new instruction tape. When put on, clearing and starting will make the machine re-begin its last programme, which is still in the machine. This is called restarting.

A telephone dial for giving small integers to the machine when the latter asks for them by stopping.

Many of these parts will be more thorougly discussed in the appropriate chapters.

V. THE DETAILED INSTRUCTIONS

Before discussing the detailed list of instructions a few general remarks are appropriate. The different types of instructions can be sub-divided into different classes and they will be dealt with in the same classes. They are:

a. Arithmetic instructions, which are all operations on the accumulator and which all use some address in the number store. The general address will be written as *n*. Addresses must never be written with zeros preceding the significant digits. Even in the extreme case of address zero nothing must be written!! This is very useful because it economises on writing symbols and also economises on the time for the machine for reading programmes. Therefore address zero is used as the most frequent working register.

   Examples: A5 but not A005

   A but not A0

   (Later on we shall see that for lack of different symbols new types of instructions are devised by adding one or more zeros. For example H is different from H0 but in both cases the address is 0 and is not written down.)

b. Input and output instructions. They govern the reading of numerical information from the input tape or the printing or punching of results on the output organs.

c. Control instructions, which can make a programme deviate from strict sequential working and jump to another instruction, conditional or unconditional. They do not refer to addresses because they are dealing with instructions in the (non-numbered) instruction store. Instead of it, they are referring to instructions which are given *labels*. These labels are denoted in the following explanatory notes by p to distinguish them from addresses n.

d. Input indications, which are not instructions in the strict sense but codes appearing on the tape (and not going in the store) to direct the instructions to go into the correct locations, to give labels, etc.

e. Special instructions or Z-instructions, which are meant for special operations on the accumulator such as log, sin, etc. They do not refer to a location in the store and their „address" is only used to indicate the type of special operation meant. They all have the form Zn.

f. Counting instructions, which can repeat a complete set of instructions a number of times.

VI. THE ARITHMETIC INSTRUCTIONS

The action of a few instructions will be described in a shorthand form.

instr.	action
Hn :	$(n) \rightarrow A$

in words

Take the number contained in location n of the number store to the accumulator. The contents of n will not be destroyed by reading it out, but the previous contents of A will be lost by reading in a new number.

instr.	action
An :	$(A) + (n) \rightarrow A$

in words

Add the contents of location n to the contents of the accumulator and place the result in the accumulator. This overwrites the addend. (n) is preserved.

instr.	action
Sn :	$(A) - (n) \rightarrow A$

in words

Subtract the contents of location n from the contents of the accumulator and place the result in the accumulator.

Example: A simple programme for calculating $a + b - c$

Suppose $(2)=a$, $(3)=b$, and $(4)=c$.

Also written

2	a
3	b
4	c

Then the programme runs as follows:

progr.	comment
$H2$	$(2) \rightarrow A$
$A3$	$(2) + (3) \rightarrow A$
$S4$	$(2) + (3) - (4) \rightarrow A$
	The result is left in A

To be able to put the result again in the store we have the instructions

instr.	action
Un :	$(A) \rightarrow n$

in words

Put the number contained in the accumulator into location n in the store, overwriting the previous contents of that location. (A) is preserved.

instr.	action
Tn :	$(A) \rightarrow n$
	$0 \rightarrow A$

in words

Put the number contained in the accumulator into location n. Then clear the accumulator by putting "0" (= very small number 1×10^{-999}) into the accumulator.

In the explanation of following instructions we shall not describe the action so much in detail as has been done above, because the same principles apply, i.e. reading out does not disturb a register, writing destroys the previous contents. The shorthand form of the description shall be retained, only supplemented by remarks when necessary.

Vn :	$(A) \times (n) \rightarrow A$	Positive multiplication
Nn :	$-(A) \times (n) \rightarrow A$	Negative multiplication
Dn :	$(A)\ /\ (n) \rightarrow A$	Division (only in positive variant)

By means of these instructions we shall code some examples.

$$\frac{ab + cd}{ef - gh} = k$$

```
H6  } e . f → A          It is often better
V7  }                    first to calculate
U   { Store in zero      the numerator
    { temporarily!
H8  } Form g . h
N9  }
A     Add ef
U     Store ef — gh → 0
H2  } ab
V3  }
U1    Store ab in 1
H4  } cd
V5  }
A1    ab + cd → A
D     Divide by ef — gh
U10   Result to 10
```

Suppose	
2	*a*
3	*b*
4	*c*
5	*d*
6	*e*
7	*f*
8	*g*
9	*h*
(10	*k*)

As formulae of the form $ab + cd$ are very frequent a special provision has been made for accumulative multiplication. This is effected by a pair of instructions of which the first is a *K*-instruction. In that case the next instruction must be a *V*-instruction or an *N*-instruction.

$$\boxed{\left.\begin{matrix}Kn\\Vm\end{matrix}\right\}\ (A) + (n) \times (m) \rightarrow A}$$

$$\boxed{\left.\begin{matrix}Kn\\Nm\end{matrix}\right\}\ (A) - (n) \times (m) \rightarrow A}$$

The *V* must immediately follow the *K*-instruction. The *V*- and *N*-instruction have a different meaning when used as second instruction in a *KV* or *KN* pair. *K* can only be used before a *V*- or an *N*-instruction. In no case it may precede another type of instruction. (Cf. Chapter VII for the use of *K* for input of numbers during input of instructions.)

Now our previous example can be written much shorter as:

```
H6     ef in the normal way
V7
K8     ef — gh by an
N9    accumulative multiplication
U      Store ef — gh → 0
H2  }
V3  }
K4  }  ab + cd
V5  }
D      Divide
U10    Store result in 10
```

Often squares appear in formulae. For additive squaring the following instruction exists:

$$\boxed{V0n\ :\ (A) + (n)^2 \rightarrow A}$$

Add the square of (n) to the accumulator. Notice that the 0 preceding the address does not belong to the address (because an address may not be preceded by zeros) but belongs to the type of operation.

$$\boxed{N0n\ :\ (A) - (n)^2 \rightarrow A}$$

Example: $(a^2 + b^2)^2 \rightarrow$ location 4

```
       | Because it is not known in      2 | a
H2     | advance that (A) is clear,      3 | b
V2     | the first multiplication
       | must be normal.
V03      a² + b²
U        (a² + b²) → 0
V        (a² + b²)²
U4
```

If this programme follows another piece of programming which we have ended with a store instruction, this can be done with *T*. In that case the example can be abbreviated to:

```
Tx       of previous piece
V02  }
V03  }   a² + b²
U    }
V    }   (a² + b²)²
U4
```

For doing multiplications with factors of 10 two special instructions exist.

$$\boxed{\begin{matrix}D0n\ :\ \text{Exponent } (A) + \text{exponent } (n) \rightarrow\\ \rightarrow \text{exponent } A\end{matrix}}$$

Add the exponent part of the number in location *n* to the exponent part of the accumulator.
The mantissa of *A* will be undisturbed, the mantissa of *n* will be disregarded.

Example: Multiply (4) by 10^6.
Suppose $(2) = 1 \times 10^6$.
*H*4 Take (4)
*D*02 Add 6 to the exponent
*U*4 Store $(4) \times 10^6 \rightarrow 4$

This instruction is quicker than a multiplication with 10^6 and preserves better the precision of the mantissa.

Another version of this instruction exists:

$$\boxed{\begin{matrix}D00000n\ :\ \text{exponent } (A) - \text{exponent } (n) \rightarrow\\ \rightarrow \text{exponent } A. \qquad \text{Mantissa undisturbed}\end{matrix}}$$

This instruction divides by a factor of 10 by subtracting something from the exponent.
Later on special applications of *D*0 and *D*00000 will be given in the count instructions.

VII. INPUT AND OUTPUT INSTRUCTIONS

To bring numbers into the machine, the programme must ask for these numbers by the following instructions.

Ln :	Read a number from the input tape and place it in location n.

The number is written on the tape by using the symbols +, —, 0 to 9 and decimal point. Numbers need not be written in floating form but can be written in fixed point form. They will be converted into floating point automatically. A sign must precede the number. Unsigned numbers are not permitted.

```
Examples:      + 35
              + 035
                + .3
               + 0.3
              — 37.568
         + 3000000
              — 0.000567891234567 0000
```

Only the 10 most significant digits (digits from the left starting at the first digit $\neq 0$) will be taken into account. The rest will be skipped. Thus in the last example we could have written with the same effect — 0.0005678912345. At the beginning of a number blank will be skipped until a sign + or — is seen. Then the actual reading starts. The end of the number will be marked by the sign of the next number on the tape. The last number to be read must also be followed by + or — to stop reading. The tape reader will see this next + or — and finish the number just read, but the tape reader will not make a step to the next symbol. Hence the sign can be reread on the next L-instruction.

Also the symbol Y can be given as an end symbol of a number. In that case the Y will be stepped over.

Sometimes a number must be given in floating form because its exponent is too large to be given in the form of the number of zeros of a fixed point number. In that case the exponent is written on the tape as given by the examples.

Examples:

+ 0.3E — 15 $(= 0.3 \times 10^{-15})$

— 35.79E + 20 $(= -35.79 \times 10^{+20} = -0.3579 \times 10^{22})$

For reading greater quantities of numbers the following instruction exists:

$L0n$:	Read numbers from tape and place them in n, $n+1$, $n+2$ etc. until a Y is encountered on the tape. The amount of numbers read is placed in δ.

With this instruction a whole string of numbers can be read in one instruction and counted at the same time in δ. (Cf. Chapter XI, counting instructions.)

When an L-instruction reads something else on the tape than numbers (e.g. instructions) the latter will be accepted as such but then the L-instruction will not return after reading and the machine will go on reading in instructions.

Input of numbers can not only be effected in machine code fixed point form, but also in teleprinter code, floating form. When the teleprinter code input routine (a programme belonging to the set of retrograde subroutines, cf. appendix 1) is in the machine, floating numbers can be taken in by L or $L0$. These numbers must have the form

$$\underbrace{\pm\ 0.\text{xxxxxxxxx}}_{\text{mantissa}}\quad \underbrace{\pm\ \text{xxx}}_{\text{exp}}$$

This is the same form as for floating output except that the mantissa may have less than 9 decimals and the exponent less than 3 decimals. The mantissa can also be ± 1.0000.

A number tape in teleprinter code must be preceded by T in teleprinter code. All carriage returns and line feeds will be skipped. After T all text will be skipped until a figure shift is encountered. After this the programme will search for the first + or — and will read the number. The end of the number will be given by the sign of the next number. A series of numbers read with $L0$ can be ended by blank tape (equivalent with Y in machine code).

A letter shift will correct that part of the number in which it appears. Hence a letter shift in the mantissa will only correct the mantissa which must be repeated in the correct form.

Blank after the mantissa without an exponent following will stop the tape.

Instead of beginning with T, teleprinter code tape may also begin with blank followed by the sign of the first number. There may be no symbol between the blank and the first sign. This is the exact form in which most output is produced. Hence output tapes in floating form can immediately be used for subsequent input into the machine.

Only +, —, ., 0 to 9 are used for numbers. Also E and Y are used for numbers. The rest is used for instruction and addressing.

The correction symbol # consists of all 5 holes on a tape and can be used to overpunch any combination (Cf. Appendix 4). The general rule for the use of the correction sign, whether in numbers or instructions, will be:

a correction sign on the tape *after* a number or an instruction will erase this word. The word must then be repeated correctly. This repetition may be preceded by an arbitrary number of blanks.

Example:

+ 34.56 punched as

$$\underbrace{+\ 34.57}_{\text{wrong number}}\quad\underbrace{\#}_{\text{corr.}}$$

(blank optional)

$$\underbrace{+\ 34.56}_{\text{correct number}}\quad\underbrace{+\ \text{etc.}}$$

The same holds for instruction

*V*0300 punched as *V*0200 # (blank optional) *V*0300, etc.

Output of number in floating form is done by:

Pn : Print the contents of location *n* in floating form on the teleprinter. Lay-out:
$\underbrace{\pm\ 0.\text{xxxxxxxxx}}_{\text{mantissa}}$ space $\underbrace{\pm\ \text{xxx}}_{\text{exp.}}$ space space
It does not destroy (*n*) nor (*A*).

In the exponents the non-significant zeros will be suppressed except the last one.

```
Example: + 0.345678901 +   0
         — 0.100000001 —  12
         + 1.000000000 — 999
                (smallest number representing 0).
```

The printer works at a speed of about 7 characters/second (150 ms/char.), hence printing of a number takes 2870 ms.

A faster way of output is by the punch actuated by

*P*0*n* : Punch the contents of location *n* in floating form on the punch. Does not destroy (*n*).

The code in which the number is punched is teleprinter code, not machine code. These tapes are primarily meant for reproduction on a separate printer, not directly attached to the machine (it takes 400 ms/number).

Apart from these *P* and *P*0 instr. to print directly from the store, there are a few special Z-instructions to do the following:

*Z*8	: Print (*A*) in floating form	No register is destroyed
*Z*22	: Punch (*A*) in floating form	
*Z*9	: Print a carriage return, line feed, figure shift	
*Z*23	: Punch a carriage return, line feed, figure shift	

The number of characters on a line of the printer is 69. Therefore no more than 3 floating numbers can be printed on one line. (The number of lines on the height of an *A*4 format sheet is also 69.)

Instructions to print in fixed-point form will be discussed in Chapter X.

Example: Read x, y, form $z = x^2 + y^2$, print z.

*L*2	Read x in 2
*L*3	Read y in 3
*H*2	$\left.\right\}$ x^2
*V*2	
*V*03	$x^2 + y^2$
*P*2	$\left.\right\}$ Print x
*P*3	Print y
*Z*8	Print $x^2 + y^2$, still in the acc.
*Z*9	Give carriage return, line feed

VIII. CONTROL INSTRUCTIONS

Of course, it is not possible to make a programme, only consisting of one sequence of instructions, each executed once. An automatic computer derives its power from the fact that it can repeat the same series of instructions over and over.

To make possible a break in the strictly sequential execution of instructions we have a jump instruction. But because instructions are not numbered we do not jump to an address but to an instruction specially provided with a label.

Xp : Jump to instruction, labelled *p* and proceed from there serially. Store a jump to the instruction following the present instruction in r. $p = 0(1)99$

The explanation of the second part of the description of *Xp* will be postponed to the instr. *X*0*p*.

Immediately in conjunction with using labels, assignment of labels must be discussed.

Qp : The instruction following this indication will be labelled *p*. $p = 1(1)99$

This *Qp* is not a true instruction but only an indication on the programme sheet and on the tape. In the machine it will not be executed as an instruction.

Example: Read two numbers, multiply them and print the result. Go on with a next set of numbers etc.

→*Q*6	*L*2	Read a_k
	*L*3	Read b_k
	*H*2	$a_k \cdot b_k$
	*V*3	
	*Z*8	Print prod.
	*Z*9	*Cr. lf*
—	*X*6	Return to *Q*6

Labels are restricted to the range 0 to 99 but they can be assigned in any desired order.

The machine is safeguarded against assigning a label twice. Of course two different points cannot be given the same name, the same label, because the machine would not know which point to take. The tape stops immediately during input of instructions when this situation occurs.

When jumps are referring to label p, but when indication Qp has never been given, this will only be detected when the programme has started its action and has arrived at this jump. Then the machine will stop.

Label 0 plays a special rôle. It is automatically and permanently assigned to the first instruction in the instruction store.

Hence the label Q with no "address" following will certainly stop the tape. But the instruction X without numerals following will jump to the first instr. of the programme. This explains that in the description of Xp the p could run from 0 to 99 but for Qp it can only have the values 1 to 99.

A label which has not been used before by an Xp or Qp is called cleared. Initially all labels 1 to 99 are cleared. In Xp there will be a label which at that moment is not cleared any more but not yet assigned. The Qp will then assign the label to the actual value.

It is irrelevant to the jump whether the label is issued before the jump or after. The whole programme is first read from the tape and put into the instruction store and only then the programme is started. Many jumps can jump to the same point.

The advantage of using labels instead of absolute addressing becomes clear when we think of the following situation. Suppose the instr. store was numbered and the jumps were absolute jumps.

```
     |  —    }
     |  —    } instr.
     |  —    }
     |  —    }
100· | X105·
-----+-------
101· |  —    {
102· |  —    {  new
103· |  —    {  instructions
104· |  —    {
105· |  —    {
```

Suppose that afterwards we see that between 102· and 103· a few instructions have been forgotten. Inserting then requires a re-addressing of the jump and a complete renumbering of all following instructions. This can be a very tedious job and it frequently gives rise to new errors.

Every jump has apart from the jumping actions also the task to place a jump in τ. This (τ) is used again in the following instruction:

$X0p$:	$(\tau) \rightarrow p$ Store the jump contained in τ into the location labelled p. Between the instr. $X0p$ and a preceding Xp instruction no other instructions may be executed (except $+\ 00n$ and $XRRR0p$).

Often a programme contains a part which performs a special action that has to be effected repeatedly in different parts of the programme. E.g. the calculation of $y = a_2x^2 + a_1x + a_0$ has to be effected in various places of the programme with different values of x. In such a case we make a sub-routine for the whole action in such a way that this sub-routine can be used by simply jumping to it.

E.g. The sub-routine for $y = a_2x^2 + a_1x + a_0 =$ $= (a_2x + a_1)x + a_0$ can read as follows:

```
                 (A) = x    Suppose 20 | a2
Q3 | X04                            21 | a1
   | U      place x in 0            22 | a0
   | H20 }
   | V   }  a2x
   | A21 }
   | V   }  (a2x + a1)x
   | A22
Q4 | X      This will be replaced by the return
   |        jump and the programme returns
   |        with (A) = y.
```

This sub-routine can be used by jumping to it with $X3$:

```
   | X3     Jump to sub-routine for calculation
   |        of y = a2x^2 + a1x + a0.
```

The jump will store in τ a return jump to the point following the instr. $X3$. As the machine does this completely automatically, no label need be given to that point. We shall say that every jump has remembered in τ the place where it came from in the form of a return jump.

The first instruction $X04$ of the sub-routine stores this return instr. from $\tau \rightarrow 4$ at the end of the sub-routine. Thus the action of the sub-routine takes place and at the end it returns to the instr. following $X3$. In this way $X3$ can be regarded as a single instruction with an arbitrary and possibly very complicated action. This $X3$ can appear many times in a programme and every time the whole sub-routine is executed. This is one of the most powerful features of programming. (For more details about sub-routines cf. Chapter XIII.)

The instruction $X0$ must be the first action of the sub-routine because most other instructions, especially of course another jump, will destroy the contents of τ.

Of course it is very undesirable to let a programme go on indefinitely in the example of forming $a_k \, . \, b_k$. Very often an action must stop according to a certain criterion. When the criterion is not yet fulfilled the programme must return and repeat the action, otherwise it must proceed. This is effected by:

Ep :	Jump to *p* only when the number in *A* is positive. Otherwise proceed to the next instruction.

and

E0p :	Jump to *p* only when the number in *A* is negative. Otherwise proceed to the next instruction.

The (A) can never be 0. In floating representation 0 is still a very small number. The 0 resulting from a subtraction of two equal numbers or resulting from a *T* instruction is always equal to $+1 \times 10^{-999}$ and hence positive.

Example: Read two numbers of which the first one is pos., then multiply them and print the result. Repeat until a first number which is neg., is found.

```
 ┌──→Q6 │ L2    Read a_k
 │      │ L3    Read b_k
 │      │ H2
 │ ┌────│ E07   Test if a_k is neg.
 │ │    │ - - -
 │ │    │ V3    Form a_k . b_k
 │ │    │ Z8    Print prod.
 │ │    │ Z9    cr, lf.
 └─┼────│ X6
   │    │ ───
   └─→Q7│ etc.
```

IX. INPUT INDICATIONS

Until now it has been supposed that the instructions in the example were already placed in the instruction store in some way. But something must be done to bring them in the store first. The general procedure runs as follows. The instructions together with all indications are written on paper. Then this is punched on the programme tape and fed into the machine. The input indications take care of the correct positioning of the programme in the store. When the complete programme has been read in, the last input indication starts the programme. During this work the programme can ask for numerical data with *L* instructions.

The first type of input indication is:

Y :	Clear all labels 1 to 99. Start input of instructions at the beginning of the instruction store. $0 \rightarrow \gamma$

The *Y* has this action only when it is immediately followed by the first instruction or in any case by an opening symbol *).

*) A symbol which begins a new item, a new instruction or a number, is called an opening symbol. They are *A, D, E, H, K, L, N, P, Q, S, T, U, V, X, Y, Z*, +, — . All other symbols are supplementary symbols. They are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, point, *R*, correction, *I*.

Secondly this *Y* must have been read as first symbol after the computer has been started from clear or it must have been read by an *L* instruction. In the middle of an instruction tape a *Y* will act as input indication for starting at the beginning of the store but it will not clear labels any more.

The *Y* may be preceded by an arbitrary number of blanks which are ignored by the tape reader.

Another type of input indication is:

Yp :	Start input of instructions beginning at label *p*. $p = 1(1)99$. Label *p* must be assigned already.

This input indication can only overwrite a piece of the programme because it can only start at a labelled location, a location that has been reached already before.

*Y*00 :	When read during the input of instructions: Start the execution of the programme from the point where the last *Y* or *Yp* started putting in instructions. When read as first item from a cleared start: Start the execution of programme at the first instruction in the store.

*Y*00 is generally used in conjunction with *Y p* in the form: *YpY*00. That *Yp* has never actually taken in any instructions, does not matter. The last point where input of instructions was started, was label *p* and there the programme will be started. It is general practice to use only *Y*00 when the piece of programming between the *Yp* and *Y*00 is short. Otherwise it is better to give in full *YpY*00 to avoid mistakes.

Example: Programme to calculate $e = \sum_{0}^{\infty} \frac{1}{k!}$

The programme can stop when the terms give no contribution any more.

	Y	Start input at beginning of instr. store. Clear labels
*Q*1	*L*1	Temporary programme. Read constant 1. Instr. is labelled 1
	L	Read following instr. !
	*Y*00	Start execution of temporary programme
	+ 1	Constant + 1 read by the *L*1 instr. and placed in 1
	Y	Closing symbol after the number + 1. This could also have been + #

	Y1	This Y1 is read by L and recognised as being not a number but an input indication. Begin input again at label 1 (beginning of instr. store), overwriting instr. $L1$ and L
	$H1$	
	$U2$	Place 1 as first sum $\frac{1}{0!}$ in 2
	$T3$	Place 1 also as first k ! in 3. Clear A
	$U4$	Place first $k = 0$ in 4. Will be augmented before use. All these actions are necessary to make the programme restartable. The restartability is a very important feature.
→Q2	$H4$ $A1$ $U4$	Augment k by 1
	$V3$ $U3$	Form next k !
	$H2$ $U5$	Place old sum in 5 for later test
	$H1$ $D3$	Form $\frac{1}{k!}$
	$A2$ $U2$	Add $\frac{1}{k!}$ to sum already formed
	$H5$ $S2$	Form $\sum \frac{1}{(k-1)!} - \sum \frac{1}{k!}$ This will be neg. as long as the terms give a contribution, but will be 0 (positive because 0 is in reality $+1 \times 10^{-999}$) as soon as the Σ's are equal
	$E02$	Return to label 2 as long as the terms still give a contribution
	Z9 $P2$	Give cr, lf and print the result
	$X3$	Make a loop stop. (We have not yet learned the stop instruction.)
Q3		The programme remains jumping to the same point.
	Y1Y00	Start execution of this programme at label 1

To be able to start input at any point in the store an input indication of the form Y0 exists:

$$\boxed{\text{Y0}n : \text{Start input of instructions at address } n \text{ in the number store}}$$

Remark that now n is an address in the number store, not a label. By means of this indication the number store can be used as extra space for instructions.

Example: Arrange a programme of 2000 instructions needing 200 working registers

	Y	
	—	
	— — —	instructions (about 1200, hence instr. store is full)
	—	
	$X50$	Jump to the other store
	Y0200	Start input at 200 of number store
→Q50	—	
	— — — —	Rest of instr.
	—	

Whether in the number store or in the instruction store, locations still can be labelled.

It is appropriate to discuss another facility in this place.

A point written after an address in the number store will designate a corresponding address in the instruction store.

In this way the instruction store becomes addressable by addresses $n\cdot$. By means of this facility we are able to keep a number of places free in the instruction store.

Example:

Y	Clear labels
— — —	Instructions
Y0200·	Start input 0Y instr. at 200·
— — — —	etc.

The point can be attached to all addresses in the number store. By means of this the instruction store can be used for numbers when the number store is too small. E.g. $A200\cdot$ means: add number from location 200· in the instruction store (henceforth called 200·.) (See Chapter XIII.)

All input indications Y have the following properties:
they restore the key-address to restart a programme in 0· with $U6 = 1$ and start;
they clear the γ-register (see count instructions, Chapter XI).

The Qp also belongs to the class of input indications. It was needed already in a previous chapter. The first single Y clears the labels, Qp issues the labels, Xp, $X0p$, Ep, $E0p$, Yp use labels. Individual labels can be cleared by:

$$\boxed{Q0p : \text{clear label } p}$$

It must not be given when a label has been used already by an Xp but has not yet been assigned by the corresponding *Qp*. In that case *Q0p* will clear the label before *Xp* has been adjusted to its proper value. Labels above 99 can be used when necessary, provided they are cleared initially by a corresponding *Q0p* (Cf. Appendix 2).

Sometimes an error is made while the tape is being punched. There is a way of correcting this on the punch by giving the combination correction on the tape after the word. This symbol consists of all 5 holes on the tape and will be denoted by the symbol # in written text. The symbol # will wipe out the preceding instruction or number. The complete correct number or instruction must be repeated after #. An arbitrary number of blanks may follow after #.

Examples:

*A*300# *A*30	In *A*300 one zero was punched too much. *A*30 was the correct word
*K*13# *H*13	*K*13 was punched instead of *H*13
+33.15# + 35.15	+33.15 was punched instead of +35.15

But not:

+33.15*Y*# +35.15*Y*	The *Y* has finished the number +33.15 already and the # comes too late.

Also:

+33.15#*Y* is wrong. The # has finished the number already and *Y* will not be read as closing symbol but as new opening symbol.

The symbol # can be used for making breakpoints in the tape. A breakpoint is a piece of blank tape often inserted between two sections which are rather independent or where a sub-routine must be inserted. This cannot be done by simply leaving blank after the last instruction of the first section. Suppose this last instruction would have been *A*1. By giving blanks after this *A*1 it would become *A*10000 .... 0 etc. By giving *A*1Z# the breakpoint can be realised. The Z will be corrected and blanks may be inserted. The tape does not stop. Of course it is irrelevant which opening symbol is given before #.

Word# :	the word is taken away and an arbitrary number of blanks may follow.

A way of making a stopping breakpoint is by giving:

QZ :	Stop input of tape. With the start key the tape can be restarted and goes on where it was stopped. An arbitrary number of blanks may follow QZ.

This combination can be used for making a breakpoint where a separate tape must be inserted. This tape is put into the tape reader on blank and is also ended with QZ.

The action of QZ can be understood by remembering that Q followed by an instruction (e.g. Z) will try to assign label 0. But this has been done already and therefore the tape stops. The Z only serves to indicate the end of Q. In fact any other opening letter will do. Also the Q may be *Qp* where *p* is an assigned label.

In a previous example the necessary numerical constant was fed into the machine during input of instruction by the help of a temporary programme. Often it is necessary to put in a few constants during input of programme. There is a way of doing this without a temporary programme by preceding the set of numbers by *K* and ending it with Y.

Example:

	Y0200	Prepare input of numbers in 200
200	*K*+1	Start input of numbers $+1 \rightarrow 200$
201	+2	$+2 \rightarrow 201$
202	+6 Y	$+6 \rightarrow 202$
203	instr.	The machine will go on putting instr. in 203 etc. unless a new input indication is given.

Also in the middle of instructions constants can be given.

Example:

```
      |  —
      |  —
      |  —
 ┌────|  X3         Jump over the set of constants
 │  ──┼──────────
 │ Q5 |  K—23       Put in constants —23, +32,
 │    |     +32     +17.6 into locations 0·5, 1·5, 2·5
 │    |     +17.6Y  (Cf. point facility, Chapter XIII).
 └→Q3 |  H1·5       Take const. + 32 → A
      |  etc.
```

The *K* is initially read as if it were a *K*-instruction, but as *K* can only be followed by a *V*- or an *N*-instruction it will be detected that numbers are following.

In fact the number will be read by the same piece of programme as for *L*0. Hence the same closing symbol Y. The first number will overwrite the *K* initially put in as instruction *K*.

There is a way to start the operation of simple code on any simple code address. This can be done by dialling a key address 44 or starting the machine with *X*44Z in normal code.

When the machine is started on 44 with *U*1 = 0 a start address in the instruction store can be dialled and the machine will start immediately.

When the machine is started on 44 with *U*1 = 1 a start address in the number store can be dialled.

When the machine is started on 44 with *U*2 = 1 a start label can be dialled.

X. SPECIAL INSTRUCTIONS

There are many useful actions that need not refer to an address. They are incorporated into Z-instructions. The "address" of a Z-instruction is not a label nor an address in the proper sense but is only an indication for the type of special operation wanted. A list of Z-instructions will be given below.

Of these Z-sub-routines Z, Z7, Z8, Z9, Z16, Z17, Z18, Z19, Z20, Z21, Z22, Z23, Z27 are permanently built into the system. All others are separate sub-routines of which only those, which are necessary, need be put into the computer. But in general they will all be kept in the store when there is room enough. In that case 1235 locations are available in the instruction store and 1235 in the number store. With none of the extra Z-instr. 1492 locations in each store are available. All Z-instructions below Z32 have a fixed meaning. There is room for special Z-programmes from Z32 to Z250 to be made by the programmer himself.

Z : Stop programme. Pressing the start key starts the programme on the next instruction. A second way of starting is by using the telephone dial. The dialled integer will be placed in α (and old $(\alpha) \rightarrow \delta$) and the next instruction will be executed.

The machine has two stop states. One state is attained after the clear button is pressed or after a mistake has been made in the programme. Then the machine can be restarted (when *U*6 is off) to read programme tape and it can also be restarted (when *U*6 is on) on the first instruction in the instruction store (on 0·). The other stop state is attained on a Z-instruction. The machine can now be restarted by pressing the start button or operating the dial.

$\boxed{Z1 : \sqrt{(A)} \rightarrow A}$ Take the square root of the number in *A* and put the result in *A*.

When $(A) < 0$ the machine skips the next instruction.

$\boxed{Z2 : \exp (A) \rightarrow A}$

$\boxed{Z3 : \ln (A) \rightarrow A}$ (A) must be pos. When $(A) < 0$ the machine stops on a clear stop.

$\boxed{Z4 : \sin (A) \rightarrow A}$ Angle in radians

When (A) is very large the sin and cos programme will cast off all whole revolutions of the angle, and the remaining precision in the fractional part can be very low or non-existent accordingly. The same holds for $e^{(A)}$ with Z2. This is one of the dangers of working in floating point.

$\boxed{Z5 : \cos (A) \rightarrow A}$ Angle in radians

$\boxed{Z6 : \arctan (A) \rightarrow A}$ The angle φ (in radians) will be $-\frac{1}{2}\pi < \varphi \leq \frac{1}{2}\pi$. For the other two quadrants the programme must make its own tests.

Example: Given a complex number $x + iy$. Determine modulus r and argument φ.
$(2) = x \quad (3) = y \quad (8) = \pi$

```
    | H2
    | V2
    | V03
    | Z1
    | U4    $\sqrt{x^2 + y^2} \rightarrow 4$
    | H3  }
    | D2  }  $\arctan \frac{y}{x} = \varphi$
    | Z6  }
    | U5    $\varphi \rightarrow 5$
    | H2
    | E11   Test $x$. If pos.: proceed
    | H5
    | E12   subtract
    | S8    $\pi$ when $\varphi$ is neg.
    | X13
    | ——
Q12 | A8
Q13 | U5    Add $\pi$ when $\varphi$ is pos.
Q11 | etc.
```

Z7 : Proceed to the next instruction when switch *U*1 on the keyboard is off. Skip the next instruction when *U*1 is on.

By means of this instruction a bifurcation can be made in the programme depending on an externally controlled switch. This is useful for suppressing intermediate results, and for making many other external decisions.

Example:

```
    | —
    | —
    | —
 ┌─ | Z7    Test U1
 ├→ | P3    Print (3) only when U1 is off
 │  |       $(U1 = 0)$
 └→ | etc.  Skip P3 when $U1 = 1$
    | —
    | —
```

Z8 : Print (A)

Z9 : Print carriage return, line feed, figure shift

(The figure shift is a teleprinter symbol for shifting the case from letters to figures. It is given as a precaution. As in Simple Code no letters can be printed, it is of no concern to us in this article.)

Z10 : $\log_{10}(A) \rightarrow A$ (A) must be pos. When $(A) < 0$ the machine stops on a clear stop.

Z11 : $\arccos(A) \rightarrow A$
Z12 : $\sinh(A) \rightarrow A$
Z13 : $\cosh(A) \rightarrow A$
Z14 : $\text{arcosh}(A) \rightarrow A$
Z15 : $\text{artanh}(A) \rightarrow A$

These sub-routines are of the so-called interpreted type and are much slower than all other sub-routines. Possibly they will be replaced by faster ones in the future.

Z16 : $2(A) \rightarrow A$
Z17 : $\frac{1}{2}(A) \rightarrow A$

These are faster than the corresponding multiplications by 2 and ½ and they do not require a constant.

Z18 : $|(A)| \rightarrow A$ Modulus of $(A) \rightarrow A$

Z19 : An instr. for changing from simple code to real machine code. This is of no concern for the purposes of this article. A similar instr. exists in real code to change over to simple code. See appendix 2.

Z20 : Will be treated together with the counting instructions.

Z21 : $-(A) \rightarrow A$

Z22 : Punch (A)

Z23 : Punch carr. ret., line feed, figure shift.

Z24 : Change "triple length" number into floating
Z25 : Change floating number into "triple length"

See App. 2

Z26 : Print or punch $|(\alpha)|$ spaces. Print when $(\alpha) < 0$, punch when $(\alpha) > 0$

Z27 : Proceed to next instr. when $U2 = 0$. Skip next instr. when $U2 = 1$

This is an instruction analogous to Z7 but now testing the switch $U2$.

Z28 : $1/(A) \rightarrow A$

Z29 :
Z30 :
Z31 :
Belong together and are used for printing/punching floating numbers in fixed point form.

With Z30 a lay-out pattern can be set up. This pattern is telling the instr. Z29 and Z31 how a number must be output. The pattern once set up will remain until the next Z30 instr. is encountered. Z30 sets the pattern to output sign when $(\delta) > 0$ and suppressing sign when $(\delta) < 0$. Before the decimal point $|(\delta)|$ digits will be output. This may be at most 10, at least 0. After the decimal point (α) digits will be output. This may be at most 9, at least 0. In case no digits after the point are output the point is also suppressed. Of course in no case does the precision of the floating number cover more than nine digits in total, hence it does not make much sense to print for example 7 digits before and 7 after the point unless the range of the numbers is greatly different. All numbers will be rounded off to the required number of places.

Thus:

Z30 : Set up pattern with $|(\delta)|$ digits before and (α) digits after the point. Suppress sign when $(\delta) < 0$.

Example:

```
+003 } Set pattern to 3 digits before
+004 } and 4 digits after point
Z30
```

Once a pattern is set numbers can be printed with

Z31 : Print (A) according to pattern and followed by two spaces.

When the number is too large to be printed, the integer part will be printed as ??? when it is $< 10^9$. When it is $> 10^9$ the complete number will be printed as $(\pm)$ TOO LARGE.

Z29 : Punch (A) according to pattern and followed by two spaces.

The same rules as for Z31 hold here.

XI. COUNTING INSTRUCTIONS

The α-count

For easy repetition of a set of instructions the count instructions are devised. A process which has to be repeated n times must be preceded by a prepare instruction which sets the count and must be ended with an instruction doing and testing the count. In its simplest forms these instructions are:

$+ 0n$: Prepare a count to n and

$+ m$: Count with m at a time. When n has not yet been reached, repeat the instructions starting at the instruction following the prepare instruction. When n has been reached, proceed.

For example: Repeat a set of instructions 10 times.
+ 010 : Set the count to 10

```
┌→  —  ┐
│   —  │
│   —  │ Instructions to be repeated 10 times
│   —  │
│   —  ┘
└─+ 1    Count with 1 until 10 is reached. Then
         go on.
```

The arrow indicates to where the counting instruction returns, when the required number of times has not yet been done.

Of course the above-mentioned description of the basic count instruction is too vague to serve as a rigorous definition. Therefore we shall first go a little bit further into the action of the count instructions.

Preparing a count is in fact doing three things:

a. setting the *running count* to zero as starting value,
b. setting the *count limit* to n,
c. remembering the return instruction to enable the counting instruction to find its way back to the beginning of the loop.

The three components of a count, namely: the running count, the count limit and the loop return instruction, are stored in the special counting registers mentioned already in Chapter IV.

α is used for the running count,
ε is used for the count limit,
θ is used for the loop return instruction. (Do not confuse the return instruction from a sub-routine stored in τ with the special loop return instruction stored in θ.)
δ is used for storing the previous (α).

Now a more accurate description of $+ 0n$ can be given:

$+ 0n$:	If $n \neq 0$: $(\alpha) \rightarrow \delta$ $0 \rightarrow \alpha$ $n \rightarrow \varepsilon$, loop return jumping to the next instruction $\rightarrow \theta$. Execute next instruction. If $n = 0$: skip next instruction and do nothing else.

In the description of the count order also reg. β and γ appear. They will be disregarded for the present.

$+ n$:	$(\alpha) + n \rightarrow \alpha$ $(\beta) + (\gamma) \rightarrow \beta$ When $(\alpha) \neq (\varepsilon)$: execute loop return in θ When $(\alpha) = (\varepsilon)$: $0 \rightarrow \gamma$ and proceed to next instr.

The register δ is used for preserving the old contents of α. It is called the safety register. β is not changed in general because any previous + instr. has cleared γ. α is not destroyed after the count is completed and can still be used.

The count limit must be reached exactly. A count prepared with + 021 and counted with + 2 will never go on but will repeat indefinitely. A count can be prepared to 20 and counted with 2 at a time. In 10 times the count will be ready.

Observe that n is not an address in the proper sense: it does not refer to a number store location but is used as the number itself. This is always a non negative integer. The registers α, β, γ, δ and ε can only contain integers, not floating numbers.

The case that in $+ 0n$ the $n = 0$ will be treated with the — 0 orders. As + 0 it does not make much sense as a preparation but often it is a useful instr. to skip one instruction. It does not change any of the count registers whatever and does not require a label otherwise needed when the skip is done by a jump.

It is not always known in advance how many times a process must be repeated. E.g. it can be the result of a calculation. Therefore the following prepare instruction is provided.

$— 0n$:	Prepare a count to (n) (instead of to n itself for $+ 0n$).

Now the n is a number address and (n) is a floating number. This floating number is converted into fixed point forms and rounded by adding $\frac{1}{2}$. Only the integer part of this is retained and used as count.

Written in shorthand this can be expressed by

$$[\,(n) + \tfrac{1}{2}\,]$$

where [ ] denotes the entier function.

One can also say that $[\,(n) + \frac{1}{2}\,]$ is the integer nearest to the floating number.

E.g. 2.99 is rounded up to 3.49, then .49 is dropped resulting in 3.

2.5 is also treated in the same way, giving $[\,3.0\,] = 3$

$[\,2.49 + \frac{1}{2}\,] = [\,2.99\,] = 2$

$[\,-2.49 + \frac{1}{2}\,] = [\,-1.99\,] = -2$! ! The next lower integer of — 1.99 is — 2

$[\,-2.51 + \frac{1}{2}\,] = [\,-2.01\,] = -3$

In general the counts are written as floating numbers to enable the programmer to effect arithmetical operations on them and in that case 2.ØØ × 2.ØØ can be 3.99 which is rounded to 4 on a — instruction.

The exact description of — 0 now becomes

$— 0n$:	If $[\,(n) + \frac{1}{2}\,] \neq 0$: $(\alpha) \rightarrow \delta$ $0 \rightarrow \alpha$ $[(n) + \frac{1}{2}] \rightarrow \varepsilon$, loop return to next instr. $\rightarrow \theta$. Execute next instruction. If $[\,(n) + \frac{1}{2}\,] = 0$: skip next instruction and do nothing else.

The same variant exists of $+ n$, the counting instruction.

$-n$: $(\alpha) + [(n) + \frac{1}{2}] \rightarrow \alpha \quad (\beta) + (\gamma) \rightarrow \beta$
When $(\alpha) \neq (\varepsilon)$: execute loop return in θ
When $(\alpha) = (\varepsilon)$: $0 \rightarrow \gamma$ and proceed to next instr.

Before giving examples a very powerful feature must be mentioned. Often a programme contains an instruction which must be varied according to the count. This can be done by adding a suffix R to the instr. to make it relative to the count.

$$\boxed{ARn = A\ n + (\alpha)}$$

ARn acts as if it were an A-instr. with an address n augmented by the present count (α). The instr. ARn is not modified in the store but the address n is augmented by (α) just before execution.

An R can be attached to the following types of instr. $H, A, S, U, T, V, N, K, D, V0, N0, X, E, L, L0, P, P0, E0$. No R can be attached to $D0, H0, U0$, all + and all —instr., $Y, Q, D00000$.

Example: Form the sum of the numbers in locations 200—300.

T	Clear accumulator
+ 0101	Prepare count to 101
$AR200$	Form variable instr. $A200+k$ where $k = 0(1)100$
+ 1	Count with 1

The same can be done with every other location instead of every consecutive location.

Example: Form the sum of the numbers in locations 200, 202, 204 . . . 400

T	Clear accumulator
+ 0202	Prepare count to 202
$AR200$	Add $(200 + 2k)$ where $k = 0(1)100$
+ 2	Count with 2 at a time

The prepare instruction — 0 serves for cases where the number of times is not known in advance. This shall be elucidated by means of a very important example.

We shall make a sub-routine for the calculation of a polynomial

$$y = a_n x^n + a_{n-1} x^{n-1} \ldots + a_0 = \sum_{k=0}^{n} a_k x^k$$

We assume that the degree will be in 100 and the coefficients in the order a_n, a_{n-1} . . . . etc. in location 101, 102 etc. x will be in the accumulator on entering the sub-routine.

The formula can be rewritten as:

$$y = (((((a_n x + a_{n-1})\, x + a_{n-2})\, x + a_{n-3})\, x + \ldots .\ a_1)\, x + a_0$$

Now the programme reads

Q2	$X03$	Place return instr. in location labelled 3. The sub-routine itself is labelled 2
	U	Place x in 0
	$H101$	Take $a_n \rightarrow Acc$
	— 0100	Prepare count to (100) times. This need not be known to the programmer
→	V	Multiply by x
	$AR102$	Add next coefficient
	+ 1	Count
Q3	Z	When ready: return to main routine. A dummy Z is placed here to keep free the place for the return instruction. In case the machine or the programmer fails to put the return instruction in its place, the machine stops.

The outward effect of this sub-routine is that the single instruction $X2$ forms $y = f(x) = \Sigma\, a_k x^k$ irrespective of degree and coefficients. Also compare this programme with the example stated in Chapter VIII. With as many instructions as given there a far more general programme has now been produced.

Sometimes it is necessary to count backwards. This can be accomplished by —instructions as in the following example.

Example: Calculate $y = \Sigma\, a_k x^k$ where $a_n = (100)$, $a_{n-1} = (99)$ etc.

U	Store x in 0	Suppose $(2) = -n$
$H100$	$a_n \rightarrow A$	$(3) = -1$
— 02	Prepare count to $(2) = -n$	$(A) = x$
V	Multiply with x	
$AR99$	Add next coefficient	
— 3	Count with —1 at a time	

As to the number of times, it does not matter whether counting is effected upward or downward, but for the variable instructions it differs a great deal.

It will be clear to the careful reader why the count must reach the count limit exactly. When the criterion would have been: repeat when $(\alpha) < (\varepsilon)$ and go on as soon as $(\alpha) \geqq (\varepsilon)$, then counts running backward would not have worked at all.

Now we shall give an example of a sub-routine doing a process p times where p will be given in the accumulator. p can be any number of times but also 0. In that case the process must be skipped altogether.

```
Q2 | X03              Place return instr. in 3
   | U                Store p = number of
   |                  times in 0. p must be
   |                  stored before a —0
   |                  instr. can set a count
   | — 0    p ≠ 0     Prepare count to p
   |                  times
   | + 0    p = 0     Do nothing but skip
   | X3               next instr.!
   |                  When p = 0 jump
   | —                over whole process
   | —                Instr. for doing process
   | —       loop
   | —
   | + 1
Q3 | Z                Return to main pro-
   |                  gramme
```

The instr. $+ 0$ has helped in jumping over the X3. It does not destroy any counts.

It is not necessary to finish a count cycle by applying a counting instruction. A count can also serve to make variable addresses.

Example: Look up the first positive number in the locations 100, 101 etc.

```
   | + 010000  Set dummy count to a high
   |           number
   | HR100     Take (100 + k)
   | E2        If number is pos., jump out
   | ———————   of the cycle
   | + 1       If number is neg., count and
   |           repeat
Q2 | etc.
```

The place where a series of numbers is starting in the store is not always known or sometimes the programme must be general enough to operate on several different sets of numbers in different places. In that case the count must be given a starting value and counted from there onwards instead of counting from 0 to n. The count must run from a to $a + n$. For this purpose the following instructions are available.

$$\boxed{+ 00n \;:\; (\alpha) \rightarrow \delta \qquad n \rightarrow \alpha}$$

In words: put previous (α) in safety in δ and put n in the count reg. .

In the same way the corresponding-order exists:

$$\boxed{- 00n \;:\; (\alpha) \rightarrow \delta \qquad [(n) + \tfrac{1}{2}] \rightarrow \alpha}$$

These orders are sufficient for setting the count itself, but they do nothing to the count limit or the loop return. Therefore a prepare instr. must follow. This prepares the count limit, puts $(\alpha) \rightarrow \delta$ and makes $(\alpha) = 0$. Now the following instr. puts the count in α.

Z20 : $(\alpha) + (\delta) \rightarrow \alpha \quad (\alpha) + (\delta) + (\varepsilon) \rightarrow \varepsilon$, loop return $\rightarrow \theta$ The instruction must immediately follow the prepare instruction to which it belongs.

As will be seen from the description the sum of α and δ is put in α. But after a preparation of a count α is cleared. Hence it simply means that $(\delta) \rightarrow \alpha$ and $(\delta) + (\varepsilon) \rightarrow \varepsilon$.

Example: Form sum of consecutive numbers

$$s = \sum_{k=0}^{n-1} a_k$$ where a_0 is contained in a location of which the address is given in 1. n is given in 2.

$$\text{Hence we can write } s = \sum_{k=0}^{n-1} (m + k)$$

$$a_k = (m + k),\ (1) = m$$

$$k = \text{running count},\ n = \text{count limit}$$

```
| — 001   m → α                        1 | m
| — 02    m → δ   0 → α   n → ε        2 | n
|         ret. instr. → θ
| Z20     m + 0 → α   m + n → ε
|         ret. instr. → θ
| AR      Executed as Am + k. Add term a_k
| + 1     Count
```

Remark that the loop is not closed to Z20 but to the instr. following Z20, because the latter has set a new return instr. For the above mentioned purpose Z20 must follow the —0 order.

There is one exception to this rule. A jump X may be between the $- 0$ and the Z20. It will only be used for programming tricks and normal programmes should never deviate from the rule.

There is another application of Z20. It adds (α) to (δ) and places the result in α. The addition is a fixed point algebraic addition of two integers. By its use something can be added to α.

Example: Add (n) to α

```
| — 00n   (α) → δ   [(n) + ½] → α
| Z20     (α) + (n) → α
```

The operation on ε makes no sense in this case, as Z20 is not given immediately after a $+ 0$ or $- 0$. Register ε is destroyed in this example as well as θ.

One of the applications of this facility is to recover (δ), as is for example needed with an *L*0 instruction, when the amount of numbers read must be counted.

Example: Read an unknown amount of numbers into location 100, 101 etc. The amount of numbers read must be put in α.

```
| +00     Clear α beforehand. It can not be done
|         afterwards as in that case δ will be lost
| L0100   Read a series of numbers in 100, 101 etc.
|         Count in δ
| Z20     (δ) + 0 → α
```

Later on we shall see how we can place (α) in the store again. The count of *L*0 could not have been kept in α otherwise it could never have been a relative instr. itself.

With the orders + 00 and — 00 we are now able to make the preparation for Z30, and the printing in fixed point form.

Example: Print (*A*) with sign, 2 digits before and 5 after the point

```
| + 002 }
| + 005 } Set up pattern
| Z30   }
| —
| —        other instr.
| —
| —
| Z31      Print (A) according to pattern,
|          previously set up.
```

Example: Print (A) in integer form without sign in 4 digits. Suppose (10) = — 4.

```
| — 0010 } Prepare pattern to 4 digits before
| + 00   } and 0 digits after the points. No
| Z30    } sign.  (δ) = — 4   (α) = 0
```

The β-count

Occasionally the β-count has been mentioned already in the text. It is applied in the double counts, counting cycles, within other counting cycles and in doing two counts at the same time.

The simplest case is a count within another count. Suppose we want to produce a table consisting of 20 blocks of 5 lines each. Every time the line count has reached 5, the block count must be increased. There is only one count register that can be tested for reaching the count limit and that is α. Hence for doing counts within counts, the outer count must be saved before the inner count can use the same register. For that purpose there exist the following instr.

*U*0*n* : Save count in *n*. (α) → β and → exponent part of *n*. (β) → α. (θ) and (ε) packed together → mantissa of *n*.

*H*0*n* : Restore count from *n*. (α) → β, exponent (*n*) → α, mantissa (*n*) unpacked → θ and ε.

Together with storing the three components of a count, also α and β are interchanged. The use of this will be clear from the examples.

Example: Make counts for 20 blocks of 5 lines

```
     | + 020   Prepare outer count to 20
 ┌─→ | —     }
 │   | —     }
 │   | —     } Other instr. in outer cycle
 │   | —     }
 │   | U02     Save outer count in 2
 │   | + 05    Prepare inner count to 5
 │ ┌→| —     }
 │ │ | —     }
 │ │ | —     } Process
 │ │ | —     }
 │ │ | —     }
 │ │ | Z9      Carr. ret., line feed
 │ └─| + 1     Count inner cycle
 │   | Z9      Give extra carr. ret.,
 │   |         line feed per block
 │   | H02     Bring back outer count in α
 │   | —     }
 │   | —     } Other instr. in outer cycle
 │   | —     }
 └───| + 1     Count outer cycle
```

As the outer count is not only saved in *n*, but also put in β, it can be used for the process in the inner cycle by the following relative instruction.

$\boxed{ARRn = An + (\beta)}$ An instruction with *RR* will be executed as having an address *n* + (β).

$\boxed{ARRRn = An + (\alpha) + (\beta)}$ The same, but then with an address *n* + (α) + (β). *RR* and *RRR* can be attached to instructions of the following types: *H, A, S, U, T, V, N, K, V*0, *N*0, *D, L, L*0, *P, P*0. They cannot be used after *X, X*0, *E* or *E*0, except *XRR* which exists.

In the case of *XRp, X0Rp, ERp* or *E0Rp* the meaning is that the location used is (α) places after the location labelled *p*. *XRRp* is jumping (β) places beyond label *p*.

The special instructions *XRRR*0 and — *RR* have a completely different meaning.

Examples of *RR* and *RRR* will be given in the matrix programme. The order in which the address, the *R*'s and the 0's are given, is irrelevant, provided that the 0's are preceding the address.

For example:

$$LRR021 = L0RR21 = L021RR = LR021R = \\ = L0R21R = L0R2R1 \text{ etc.}$$

Another use of the β-count is doing a simultaneous count. We then need a separate increment for the β-count. As can be seen from the description of the count instruction + and —, β is counted with (γ). The increment γ can be set with the instructions:

$+ 000n : n \rightarrow \gamma$	Set γ to n
$- 000n : [(n) + \frac{1}{2}] \rightarrow \gamma$	Set γ to (n)

Example:

Form the scalar product of two vectors a_k and b_k of n elements. The elements of both vectors will be placed in consecutive locations, vector a_k starting in a location of which the address is given in 2, vector b_k starting in a location of which the address is given in 3. n is given in 1.

We must form

$$P = \sum_{k=0}^{n-1} a_k b_k = \sum_{k=0}^{n-1} (p+k) \cdot (q+k)$$

```
       T       Clear acc. beforehand          1  | n
       — 003   q → α                          2  | p
       U0      Transfer q → β.                3  | q
               Otherwise U0 is a
               dummy save instr.
       — 002   p → α. Still (β) = q
       + 0001  Set increment γ of            p   | a₀
               β-count to 1              p + 1   | a₁
       — 01    Prepare count to n.       p + 2   | etc.
               p → δ  0 → α   n → ε
       Z20     p + k → α
               p + n → ε where               q   | b₀
               k = 0 (1) n — 1           q + 1   | b₁
                                                 | etc.
  ┌──→ KR      Take aₖ from p + k
  │            according to α count
  │    VRR     Multiply with bₖ from q + k
  │            according to β count
  └─── + 1     Count (α) + 1 → α. (β) + 1 → β.
       ↓       Test if count is ready
               (A) = P = Σ aₖ . bₖ
```

When the location of the vectors would have been fixed, everything would have been much simpler, but the unknown locations make two counts necessary.

Example: Take from a series of numbers a_k all positive numbers b_m and place them in consecutive locations. The initial address of a_k will be in 2. The initial address of b_m will be in 3. The cycle must be stopped when n items b_m have been reached.

```
          +0001        Set increment γ to 1         1 | n
          — 002        a → α                        2 | a
          U0           Interchange (α) and (β)      3 | b
          — 003        b → α   a → β
          — 01    }
          Z20     }    Prepare count for bₘ to n
 ┌─┬───→  HRR          Take aₖ = (a+k)  (β) = a+k
 │ │      E2    ──┐    If aₖ is pos., go to 2
 │ │      - - - - │
 │ └───   +     ←─┤    If neg., count in β but not in α
 ↑   Q2   UR    ←─┘    If pos., store aₖ as bₘ in
 │                     m (α) = b + m
 └─────   + 1          Count in α and in β with 1.
                       When α is counted n times,
                       leave process.
```

Example: Now the same problem, but with a counting cycle that stops when n items of a_k have been processed.

```
          — 003
          U0
          — 002             a → α   b → β
          — 01              Prepare for aₖ to n
          Z20
 ┌─┬───→  HR                Take aₖ = (a + k)
 │ │                        (α) = a + k
 │ │      E2       ──┐      Test aₖ
 │ │      - - - - -  │
 │ │      + 000    ←─┤      If neg., make (γ) = 0
 │ └───   + 1        │      Count with 1 in α and 0 in β
 ↑        X3         │      If ready: jump to 3
 │        ─────────  │
 │   Q2   + 0001   ←─┘      If pos., make (γ) = 1
 │        URR               Store bₘ  b + m
 │                          (β) = b + m
 └─────   + 1               Count
     Q3   etc.
```

Not only can arithmetic instructions be made relative to counts but also jumps can be made relative, which is done to make so-called multiple switches. Depending on the contents of α a jump of the form XRp can be made to jump to location $p+$ so many places more as (α) indicates. Often a relative jump is used to advantage at the beginning of a programme.

```
    | Y
    | + 00          Clear α beforehand
    | Z             Stop. Wait for dialling.
    |               Dialled number is placed in α
    | XR5   ──┐     Make relative jump to 5 etc.
Q5  | Xa    <─┤     Jump to a when 0 is dialled
    |         |     or when nothing is dialled
    |         |     but the start key pressed.
    |         |     That is the reason why α had
    |         |     to be cleared
    | Xb    <─┤     Jump to b when 1 is dialled
    | Xc    <─┤     Jump to c when 2 is dialled
    | —     <─┘
Qa  | —       }     Part a of programme
Qb  |         }     Part b of programme
    | etc.    }
```

Tests can also be made relative. In case they fail to jump, the next instruction is taken. Only when they are successful they make a relative jump. Notice that only a single *R* can be used with *X*0, *E*, *E*0, and a single and double *R* with *X*.

On the instr. *U*0*n* and *H*0*n* the count is put into safety and restored. Something will be said about the way the three components are stored in the respective registers.

Each floating number has a separate register in the machine for the mantissa and for the exponent. Although the exponent is not printed in more than 3 decimal digits, it is stored in the machine in about 9 digits. The mantissa occupies a full register. On storing a count with *U*0*n* the address part of the return jump is retained (it is in any case certain that a jump is meant). This leaves enough room for storing the count limit (ε) in the same register, but an upper limit is imposed upon the capacity of the count limit in that case. The count limit may not exceed 10^6. This is practically never the case.

The packed return address + count limit are stored in the mantissa part of a location. The count (α) is stored in the exponent part of the same register. The α has the complete exponent available.

Internally all counts are kept in 4-fold. This is of no concern to the programmer except in the case of storing counts, where it must be clearly remembered that also this 4-fold will be put into the exponent.

Example: $(\alpha) = 3$, then after storing with *U*0*n* the contents of *n* will have an exponent part of $4 \times 3 = 12$.

Later on we shall see some special applications for separating a number into its mantissa and exponent by using the *H*0*n* and *U*0*n* instructions for this purpose. Sometimes it can be troublesome to have the 4 fold in the exponent and therefore a special instruction exists:

$$\boxed{TRR0 \;:\; \tfrac{1}{4}\,(\alpha) \rightarrow \alpha}$$

This instruction is only to be used before *U*0*n* and it enables us to make really (α) into the exponent of a number.

In connection with this it must be remembered that calculations on the exponents can be done with *D*0*n* and *D*00000*n*.

Example: Read floating number from dial by dialling first the mantissa and then the exponent. *Z*7 indicates the sign of mantissa. Exponent always pos.

```
        | Z          Dial digits of mantissa,
        |            taken as integer
        | —RR        Store this in 0
        | H          Mantissa → A
┌──── + | Z7         Supply mantissa with sign
│ └──>  | +0
└────>  | Z21
        | D00000     Remove exponent part, leaving digits
        |            of mantissa behind the point
        | Z
        | TRR0       Make ¼ (α)
        | U0         Store this as exponent part in 0
        | D0         Supply mantissa with exponent
        | U          Store number in 0
```

The instruction — 00*n* transforms (*n*) into a fixed point number by taking $[\,(n) + \tfrac{1}{2}\,]$ and it places this in α. Often the reverse process will be required and a special instruction exists for this action:

$$\boxed{-RRn \;:\; \text{Make } (\alpha) \text{ floating and place this in } n}$$

The *RR* has nothing to do with relative instructions, as *R*'s cannot be used with counting instruction.

One of the most important applications of this instruction is the production of an argument from the count. Suppose that a table of a certain function $y = f(x)$ must be calculated for $x = 0(0.03)3$ (x running from 0 to 3 with increments of 0.03). The argument x might of course be calculated by adding 0.03 to the previous x in each cycle. But in that case an intolerably large accumulation of errors would occur, because 0.03 is not an exact number but a recurrent fraction of a precision of about 9 decimals (internally the machine works in the binary system). Moreover the printed argument with an error growing gradually larger and larger depending on whether 0.03 is represented by a number which is a little bit too large or too small, will produce unsightly tables. This can be prevented by calculating the argument from the count by — *RR*. As the count is kept as an integer which always is an exact number, there will be no accumulation of error in this case.

Example: Make a table with argument
$x = 0(0.03)3$.
There are 101 items to be printed.
Suppose $(1) = 0.03$.

```
┌→ | + 0101    Prepare count to 101
│  | — RR      Put the count $k$ in 0  $k = 0(1)101$
│  | H         Take $k$ as floating number in A
│  | V1        Form $x = 0.03\ k$  $x = 0(0.03)3$
│  | U         Store $x$ in 0
│  | ———— }
│  | ———— }    Calculate $f(x)$ etc.
└— | + 1
```

Another application is the separation of mantissa and exponent of a number.

Example: $(2) = a \times 10^b$. Put a in 3 as a floating number and put b in 4 as a floating number

```
| H2         Take $a \times 10^b$
| D000002    Subtract $b$ from exponent (divide
|            through $10^b$) leaving $a \times 10^0$
| U3         Store $a \to 3$
| H2      }
| D02     }  Double exponent
| U          Store number with $2b$ in 0
| D0         Form number with exponent $4b$
| U          Store $a \times 10^{4b}$
| H0         Take $a \times 10^{4b}$ in count registers as
|            if it were a count. Then $b \to \alpha$
|            (Remember that counts are kept in
|            4-fold)
| — RR4      Store $b$ floating → 4
```

A similar sort of operation is the separation of the integer part and fractional part of a number.

Example: $(2) = a = \overline{x,y}$ Put x in 3, y in 4

```
| H2         Take $\overline{x,y} \to A$
| — 002      Put $x$ in $\alpha$
| — RR3      Store $x$ in 3
| S3         Subtract $x$ from $\overline{x,y}$ leaving $0,y$
| U4         Store this in 4
```

It must be realised that now y has not necessarily the sign of x, because — 002 took the nearest integer value to the number $\overline{x,y}$. E.g. 4.7 will be split up into + 5 and — 0.3. To prevent this, $\frac{1}{2}$ must be first subtracted before applying — 00.

A single instruction belonging to the family of counting instructions still remains to be mentioned. It has appeared already that of the three components of a count, the count limit and the count itself can be set independently. The order + 0 (— 0) sets the count limit and + 00 (— 00) can separately set the count. Z20 can add something to the count. Only the loop return instruction is always coupled to setting the count limit or Z20. In some applications it is necessary to set the loop return instruction independently of setting the count.

$\boxed{\text{XRRR0}p \ : \ \text{Jump to label } p \text{ and put a jump, the loop return instruction, to the next instruction, in } \theta.}$

A characteristic example of application is the case where the first few cycles and the last few cycles of a loop process are different from the general cycle. Suppose a loop must be done n times, where the first two and the last two cycles are different.

$(5) = n - 2$

```
      | — 05       Prepare count to $n - 2$ times
      | —    }
      | —    }     Instr. of the first different process
      | —    }
      | XRRR02     Jump to the count instr. As count
      |            is not yet ready, it returns to the
      |            next instr.
      | —    }
      | —    }     Instr. of the second different
      | —    }     process
      | XRRR02     Jump to the count instr., count
      |            and return to the next instr.
 ┌→   | —    }
 │    | —    }     Instr. of the main cycle
 │    | —    }
 │    | —    }
 └Q2  | + 1        Count
      | —    }
      | —    }     Instr. of the penultimate different
      | —    }     process
      | —    }
      | XRRR02     Count.  As count is finished on
      |            leaving the main cycle, $(\alpha) \neq (\varepsilon)$
      |            on this occasion and instr. returns
      |            on next one
      | —    }
      | —    }     Instr. of last process
      | —    }
      | XRRR02     When necessary the count can still
      |            be augmented
      | etc.
```

Sometimes $XRRR0p$ is useful as a jump to a sub-routine. When no count has to be kept and the sub-routine itself does not use θ, the jump $XRRR0p$ will remember the return to the main programme in θ. The sub-routine can return with a count order. It does not matter how much it counts, provided that the count is not just finished. In that case the instr. following the + would be executed. In general (α) and (ε) will still contain the variables of the last finished count and a + 1 order will certainly return to the main programme.

An *XRRR*0 order does not destroy the return instr. in τ. Hence it can be used as one of the very few types of orders which may stand between an *Xp* and the corresponding *QpX*0*q*.

XII. TIMES OF EXECUTION

Until now we have not spoken about the times of execution of a programme in simple code. Very simple rules can be given.

For input of programme tapes and number tapes:
10 *ms* per symbol + 20 *ms* per complete instr. or number.

For execution of programmes:
30 *ms* per executed instruction.

For output of results:
On punch: 20 *ms* per symbol + 30 *ms* per number.
On printer: 150 *ms* per symbol + 30 *ms* per number.

These rules have been established by averaging the real operation times over a great number of programmes for practical problems.

For those who wish to make a more precise calculation of times a detailed list will be given below. All values will be rounded to an integer number of *ms*, as all times are averages. The exact operation time of an order depends on its exact location in the programme and on the location of the operand used.

Arithmetic instructions:

H	: 20 *ms*
A and *S*	: 40 *ms* (25 *ms* when one of the terms is $< 10^{-9} \times$ the other term)
U	: 20 *ms*
T	: 22 *ms*
V and *N*	: 35 *ms*
D	: 55 *ms*
K	: 20 *ms*
V after *K*, *N* after *K*, *V*0 and *N*0	: 16 *ms* + time of *A*
*D*0 and *D*00000	: 20 *ms*

Control instructions:

X	: 13 *ms*
E and *E*0	: 13 *ms*
*X*0	: 21 *ms*

Input and output instructions are completely determined by the times of the tape reader, punch and printer.

The tape reader reads 100 characters/sec.
The punch punches 50 characters/sec.
The printer prints 7 characters/sec.

L and *L*0	: 30 *ms* + 10 *ms* per character
P	: 30 *ms* + 150 *ms* per character
*P*0	: 30 *ms* + 20 *ms* per character

Count instructions:

+	: 15 *ms* when count is not ready; 23 *ms* when ready
+0	: 15 *ms*
+00	: 16 *ms*
+000	: 14 *ms*
—	: All instr. take 16 *ms* longer than the corresponding + instr. for the first two digits of (*n*). When (*n*) > 100 10 *ms* for each two digits more.
Z20	: 16 *ms*
*U*0	: 25 *ms*
*H*0	: 24 *ms*
—*RR*	: 45 *ms* for counts of 0 or 1 digit, 30 *ms* for each digit more. This is a very time-consuming instruction.

Relative instructions:

With *H, A, S, K, V, N, V*0*, N*0*, D, P, P*0 the first *R* takes 7 *ms* over the normal operation time. The second and third *R* each take 0.6 *ms* more.
With the instructions *U, T,* —*, X, X*0*, E, E*0*, L, L*0 each *R*, also the first one, takes 0.6 *ms*.

Z	: When dialling, the dial waits for 1.5 *s*. after the last digit before starting action
Z1	: 80 *ms*
Z2	: 120 *ms*
Z3	: 120 *ms*
Z4, Z5	: 100 *ms*
Z6	: 130 *ms*
Z7	: 15 *ms*
Z8, Z22	: Same as *P* and *P*0
Z9	: 470 *ms*
Z10	: 100 *ms*

Z11	Not exactly determined, but all between 200 *ms* and 500 *ms*
Z12	
Z13	
Z14	
Z15	
Z16	28 *ms*
Z17	19 *ms*
Z18	17 *ms*
Z21	14 *ms*
Z23	80 *ms*
Z24	30 *ms* + 10 *ms* × exponent of resulting number
Z25	Not yet known
Z26	Same as *P* or *P*0
Z27	15 *ms*
Z28	45 *ms*
Z29	50 *ms* + 20 *ms* per character
Z30	35 *ms*
Z31	50 *ms* + 150 *ms* per character

XIII. THE POINT FACILITY AND SUB-ROUTINES IN SIMPLE CODE

In IX the basic meaning of attaching a point after an address in the number store has been treated already.

> A point written behind a number address changes this address to the corresponding one in the instruction store.

One of the uses of this facility is the access to the instruction store for numbers (with the drawback that an absolute numbering system is now employed). With the input indication *Y*0*n* it was already possible to start input of instructions at address *n* of the number store. And with *Y*0*n*· it is possible to start input of instructions into the instruction store at the absolute address *n*· instead of consecutively from the beginning.

This point may only be attached to genuine number addresses, not to labels. Hence an *Xp*·, *Yp*·, *Zn*· may never appear. Although the address in *Zn* is treated as a number address on input, it is interpreted completely different on execution.

For multiple points and points after some + instructions see appendix 1.

Especially for self-contained sub-routines and easy assignment of space for working registers or constants a more elaborate facility is provided after point.

An instruction of the form *An*·*p* with a number address, a point and after the point a label has the following meaning:

> *An*·*p* is an instruction applying to a location lying *n* places after the location to which label *p* has been given. Label *p* must have been assigned already before using it with *n*·*p*

Suppose that we want to write a sub-routine for determining $y = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$, wherein the coefficients have a definite value. We want to make this sub-routine completely self-contained and general. That means that we must be able to place it anywhere in the store and that all working registers and constants are contained within the sub-routine.

To enable the programmer to jump to the sub-routine it must be given a label. This cannot be a fixed number because possibly another sub-routine uses the same label. Hence the label by which it is called in, must be given externally before the sub-routine tape is run in. On the other hand the coding within the sub-routine itself must be done with a fixed label, otherwise some instructions would have a form dependent upon the externally given label.

The way out is to give internally a label to the beginning (usually a low label), to let everything within the sub-routine be addressed relative to the label with *n*·*p* and to clear the internal label(s) at the end of the sub-routine tape. This makes the following convention necessary.

> When a programme contains a few sub-routines they must be run in before the main programme, so that they can use and clear labels, later used in the main programme. Only labels 2-9 shall be employed for this purpose.

Example: Make a sub-routine for $y = \sum_{k=0}^{5} a_k x^k$.

x will be taken to the sub-routine in *A*.
y is carried back in *A*.

	Qp		External label to be written by the user for every specific occasion
(0· 2)	Q2	*X*03	Internally used label
(1· 2)		*U*8· 2	Place return instr. in 3
(2· 2)		*H*9· 2	
(3· 2)		+ 05	
(4· 2)		*V*8· 2	
(5· 2)		*AR*10· 2	
(6· 2)		+ 1	
(7· 2)	Q3	Z	Return to main programme
(8· 2)		Z	Location for variable *x*
(9· 2)		*K* a_5	Start reading constants in 9· 2 etc.
(10· 2)		a_4	
(11· 2)		a_3	
(12· 2)		a_2	Constants placed in 9· 2 to 14· 2
(13· 2)		a_1	
(14· 2)		a_0	
		Y	End symbol to stop reading constants
		Q02 Q03	Clear labels 2 and 3
		A#	Make break point for enabling copying with blank between sections.

The make up of a complete programme needing two of this sort of sub-routines could be as follows:

```
        | Y              Start input and clear labels
        | Y05
        | K —      }
        |   —      }     Constants for main pro-
        |   —      }     gramme read into 5 etc.
        |   —      }
        | Y              Closing Y of series of numbers
        | Y0·            Start putting in instruction
        |                No labels are cleared as this is
        |                not necessary
  ┌──── | X1             Entrance point of programme
  │     |                is chosen at the beginning of
  │     |                the instr. store to make re-
  │     |                starting from a cleared stop
  │     |                possible with U6 = 1 and
  │     |                start. Jump to Q1
  │     | Q10 A #        Assign label 10 to first sub-
  │     | (blank)        routine and make breakpoint
  │     |                with blanks for easy copying
  │     |                in the sub-routine tape
  │     | 1st subr.      First sub-routine copied in.
  │     |                Completely self-contained.
  │     |                Only labelled 10 externally.
  │     |                Hence main-routine can call
  │     |                it in by X10
  │     | Q11 A #        Assign label 11 to second sub-
  │     |                routine
  │     | 2nd subr.      Second sub-routine can be
  │     |                called in with X11
  └→Q1  | —        }     Instructions of main pro-
        | —        }     gramme. Entrance is labelled
        | —        }     Q1, whereby Q2 and other
        | —        }     low numbered labels can be
        | X10      }     used freely
        | —        }
    Q2  | —        }
        | —        }
        | —        }     Sub-routines 1 and 2 are called
        | —        }     in by X10 and X11
        | —        }
        | X11      }
        | Y1Y00          Start execution of main pro-
        |                gramme at the beginning
```

In the previous example of making a general sub-routine it appeared that the programmer had to count the number of instructions of the sub-routine to be able to determine the places for the constants and working register.

With a somewhat different arrangement this could have been prevented at the cost of one more instr. in the following way:

```
(0·2)   Q2 | XRRR03       Jump over the constants
(1·2)      | Z            Keep free working space
(2·2)      | Ka5    }
(3·2)      |   a4   }
(4·2)      |   a3   }
(5·2)      |   a2   }     Constants
(6·2)      |   a1   }
(7·2)      |   a0   }
           |   Y          Closing Y
        Q3 | X04    }
           | U1·2   }
           | H2·2   }
           | + 05   }     Programme
           | V1·2   }
           | AR3·2  }
           | + 1    }
        Q4 | Z
           | Q02 Q03 Q04  Clear labels 2, 3 and 4
           | A #
```

A fine example of the application of sub-routines of the general type is the use of them for calculation with complex quantities. Suppose we want to make sub-routines for the complex equivalents of the orders *H, A, S, U, T, V, N* and *D*. The sub-routines will be given a fixed label for each type of operation to be called in. The address will be given in α.

Thus + 00*n* *X*10 will be the complex equivalent of *Hn*. Although being two instructions, they can be regarded as a single instruction with an address part of the form + 00*n* and an operation part of the form *X*10. The real and the imaginary components are supposed to be always in two consecutive locations of which the address of the first will be the "address" of the complex "location".

The other operations will be denoted by:

```
+ 00n X11 is equivalent to An   There will be a spe-
+ 00n X12 ,,     ,,      ,, Sn   cial "complex accu-
+ 00n X13 ,,     ,,      ,, Un   mulator" inside the
+ 00n X14 ,,     ,,      ,, Tn   sub-routines with
+ 00n X15 ,,     ,,      ,, Vn   address 1·10 and also
+ 00n X16 ,,     ,,      ,, Nn   a constant 1 + 0·i
+ 00n X17 ,,     ,,      ,, Dn   in 3·10
```

For the operations *KnVm* and *KnNm* we can preferably adopt a two address code

```
+ 00n + 00m X18 is equivalent to KnVm
+ 00n + 00m X19 ,,     ,,      ,, KnNm
```

All other operations can be effected in the normal Simple Code. Of course it is not necessary to prepare the address in α with *a* + 00 instruction, for a relative instruction can be realised as well, while the count is kept in α. Or preparation to a variable address can be effected with — 00*n* etc.

Sub-routines have been made for all normal functions of complex variables but they will not be treated here.

```
Q10
Q2        XRRR03    Jump to 3 over working reg.
(1·2)
(2·2)
          Y03·2     Complex accumulator
(3·2)     K + 1  }  Constant 1 + 0·i
(4·2)     + 0    }
          Y      }  Keep 5·2 and 6·2 free for working
          Y07·2  }  space
Q3                  Label 3 is issued, cleared and issued
→Q03      X03       again. Store return
          HR     }  Place real part of location
          U1·2   }  specified in α → 1·2
          HR1    }  Place imaginary part
Q4        U2·2   }  → 2·2
Q3        Z         Return
          ─────
Q11       X03       Place return instr.
Q7        H1·2      Add real part to
          AR        Re(Acc)
          U1·2
          H2·2   }  Add imaginary part to
          AR1    }  Im(Acc)
          X4        and use piece of Q10 programme
          ─────
Q12       X03       Place return instr.
Q8        H1·2
          SR        Subtr. real part from
          U1·2      Re(Acc)
          H2·2   }  Subtr. imaginary part from
          SR1    }  Im(Acc)
          X4        and use piece of Q10 programme
          ─────
Q13       X05       Place return instr.
          H1·2   }  Store Re(Acc) in (α)
          UR     }
          H2·2   }  Store Im(Acc) in (α) + 1
          UR1    }
Q5        Z         Label 5 can be cleared
─────     ─────
Q05       X03       Store return instr. in 3
Q14       X13       Use complete X13 instr. for storing
─────
          H4·2   }  For storing
          U1·2   }  Place 0 in Re(Acc) and
          X4     }  Im(Acc)
          ─────
Q15       X05       Store return instr.
          H1·2   }                 (A) = a + bi
          VR     }  Form           (n) = c + di
          K2·2   }  ac — bd
          NR1    }
          U5·2
          H1·2   }
          VR1    }
          K2·2   }  Form ad + bc
          VR     }
Q6        U2·2      Store this in 2·2 as Im(Acc)
          H5·2   }  Put Re(Acc) in its place
          U1·2   }
Q5        Z         Return
          ─────
```

```
Q16       X05       Place return instr.
          H1·2   }
          NR     }  Form — ac + bd
          K2·2   }
          VR1    }
          U5·2      Place this in 5·2
          H1·2   }
          NR1    }  Form — ad — bc
          K2·2   }
          NR     }
          X6        and finish action in part of sub-
          ─────     routine Q15
Q17       X05       Place return instr. in 5
Q05                 Label 5 can be cleared now
          HR     }
          VR     }  Form c² + d²
          V0R1   }
          U6·2      Place this in 6·2
          H1·2   }
          VR     }       ac + bd
          K2·2   }  Form ───────
          VR1    }       c² + d²
          D6·2   }
          U5·2      and store this in 5·2
          H2·2
          VR             bc — ad
          K1·2      Form ───────
          NR1            c² + d²
          D6·2
          X6        and finish action in Q15 programme
Q06       ─────     Label 6 can be cleared now
Q18       X03       Place return instr. in 3
          X5        Jump to special sub-routine for
          ─────     forming (n) × (m). Remark that
                    label 5 must still be issued again,
                    as the previous 5 has been cleared
          X7        Finish operation in addition part
          ─────
Q19       X03       Place return instr.
          X5        Form (n) × (m)
          ─────
          X8        Finish operation in subtraction part
          ─────
Q5        X06       Sub-routine for (n) × (m). Place
                    return instr.
          H03·2     (α) → β  0 → α. (Exp of 1 is 0!)
          Z20       (δ) → α. Hence (α) = n (β) = m
          HR     }
          VRR    }  Form Re(prod) of (n) × (m)
          KR1    }
          NRR1   }
          U5·2      and store this in 5·2
          HR     }
          VRR1   }  Form Im(prod)
          KR1    }
          VRR    }
          U6·2      and store this in 6·2
          + 005·2   Prepare now (α) = 5·2
Q6        Z         Return and the addition programme
                    will finish the addition of 5·2 — 6·2
                    to the "acc"
```

```
| Q02 Q03 Q04 Clear all temporary labels and
|             make breakpoint in tape
| Q05 Q06 Q07
| Q08 A #
```

With the help of this general sub-routine operations on complex numbers can be effected with a coding as simple as for real numbers, but the operation time is considerably longer.

Example: Read Z_1, Z_2 from tape and calculate

$$\frac{Z_1 - Z_2}{Z_1 + Z_2}$$

(Z_1 and Z_2 are complex numbers)

```
   | Y
   | X1             Jump to main programme
   | A #            Make breakpoint
   | Complex
   | sub-routine
Q1 | Z              Stop
   | L02            Read Z1 and Z2 in "2" and "4"
   |                resp. (in reality Re(Z1) → 2,
   |                Im(Z1) → 3,  Re (Z2) → 4,
   |                Im(Z2) → 5.
   | + 002 X10      Take Z1
   | + 004 X11      Add Z2
   | + 00 X13       Store Z1 + Z2 in 0
   | + 002 X10      Take Z1
   | + 004 X12      Subtract Z2
   | + 00 X17       Divide by Z1 + Z2
   | P1· 10  }
   | P2· 10  }      Print Re(Acc) and Im(Acc)
   | Z9             Carr. ret., line feed
   | X1             Return to start of programme for
   | ———————        next part of Z
   | Y1Y00          Start executing of programme
```

The labels after point have sometimes another useful application. When a large programme is made, several sets of working registers and sets of constants are used. It is not always known in advance how many working registers will be needed in a set, when the programmer starts writing the programme. Therefore it is very convenient to assign a label to the beginning of a set of working registers and number from there onwards with an address followed by this label.

Example:

Suppose there are three sets of working registers, one for the incidental intermediate results, one for a vector and one set for a matrix. We shall denote these registers by *n*· 5, *n*· 6 and *n*· 7. Thus the 3rd element of the vector is placed in 3· 6. It is not necessary to know beforehand where the labels will be placed exactly, but when the whole programme is finished the number of locations in each set is known.

Suppose it appears that 15 incidental working registers, 20 vector registers and 400 matrix registers are necessary. Then the following prefix to the programme will set the labels.

```
Y              Clear all labels
Y010 Q5        Start input at 10 of number store and
               assign label 5 to this location (In fact
               input is never started at 10)
Y015· 5 Q6     Start input at 15 locations after label 5
               and assign label 6 to this location
Y020· 6 Q7     Start input at 20 locations after label 6
               and assign label 7 to this location
Y0·            Start input of instructions at 0· in the
               normal way
—
—
—
—
—
A3· 6          Programme contains instr. making refer-
               ence to 3· 6
—
—
—
etc.
```

When some alteration is made, in the degree of the vector for example, only the indication Y020· 6 *Q7* need be changed and the tape can be run in again.

XIV. CUTTING OPEN WITH THE *I*-FACILITY

A somewhat more complicated programme is seldom completely correct the first time it is tried. We shall not discuss errors in punching the tape from the written documents. More often the programmer has made a slip in transcribing the formula into a list of instructions. Some errors are automatically signalled, as for example issuing a *Qp* twice or forgetting to give the corresponding *Qp* where *Xp* has been used already. Most errors are due to mistake in thought on the side of the programmer. When a programme gives wrong results it is often difficult to see what error has been made without knowing intermediate results. To make intermediate results visible the so-called *I*-facility is provided.

By placing an *I* after an instruction (e.g. *A20I*), the machine can be made to take away the instr. marked with *I* and to place a special instr. instead. When the programme is working it will print out a few intermediate results as soon as it encounters the special instruction. It prints the address of the location where the *I*-marked instruction is standing, it prints the contents of *A* before the instruction is executed and it prints (*A*) after the execution of the instruction. Also the machine can be made to print only the address. We thus say that the *I*-facility is working as tracer. This is very valuable when an error is supposed to be present in the correct looping and flow of procedure. Attaching an *I* to an instruction is called cutting-open on that instruction.

The rules for using *I* are:

I behind an instruction is disregarded completely during input when *U5* = 0. When *U5* = 1 all instr. marked with *I* are cut open.

During execution:

U5=1, *U4*=1, *U6*=1:	As soon as an instr. marked with *I* is encountered print: Carr. ret., line feed, address where instr. is located in the form n or n·, (*A*) before execution in floating form, (*A*) after execution in floating form and carr. ret., line feed.
U6=0:	Same as above but only print tracers: carr. ret., line feed, address of instr.
U4=0:	Do not print at all but retain the *I*-marks on the instructions. Printing can be resumed by putting *U4* = 1.
U5=0:	Print according to *U4* and *U6* as above. Restore the original instr. after printing (*A*) for the first time.

As many *I*'s may be attached as the store can contain (cf. small print of this chapter). *U5* determines whether the *I* will be accepted during input and it also determines whether *I* is taken away during execution.

The *I* must always be at the end of an instruction (e.g. *L0R200RI* but not *L0RI200R*). It can be corrected away by putting a correction after *I* (e.g. *L0RR200I* #). An *I* after *I* will also be seen as a correction.

There is a class of instructions to which *I* must be attached with care. These are all instructions placing a return instruction, namely: *Xp*, *XRRR0p*, + 0*n*, — 0*n*, Z*n*, Z20. When a simple jump *Xp* (and also *XRRR0p* used as jump only) are cut open, then only the contents of *A* are printed before execution of the jump. There is no come back after the instruction because it jumps away. But when with *Xp*, *XRRR0p* or Z*n* a sub-routine is called in, it comes back after return from the sub-routine and prints for the second time. This means that it still has not finished the *I* on *Xp* as long as the sub-routine has not returned. Within the sub-routine no other instruction may be cut open.

The same applies to + 0*n* placing a loop return. Within the loop no instruction may be cut open. And as the loop returns many times it will print (*A*) after the execution of + 0 also many times. To avoid mistakes it is better not to cut open + 0*n*, — 0*n* or Z20.

The instruction + or — is in effect a jump when it returns to the beginning of the loop. In that case *I* will only print (*A*) before the instruction. The last time when + or — leaves the loop, it will print (*A*) before and after the instruction.

Instructions cut open with *I* are taken away from the place where they belong and in that place a blocking instruction of special form is placed. The original instruction is placed in a list. The initial address of that list is normally 1000·, placed there by the initial Y followed by an opening symbol, which effects the clearing of the labels. All other input indications do not change anything to the list. Even temporary programmes may be executed without destroying the instruction, storing in the *I*-list, provided that they do not use the β-count. The store instruction for storing *I*-instruction in the *I*-list is kept in β.

It is not always possible to have the list of *I*-instruction, starting in 1000·, e.g. the programme can be too long for that. In that case a special input indication can be given of the form

Y0*nI* or Y0*n*·*I* :	Begin to store list of *I*-instructions at *n* or *n*· resp.

Such an indication must always be followed by a normal input indication as, when *U5* = 0, *I* would be disregarded.

Example: Start *I*-list at 700 of number store instead of at 1000·.

Y	Begin input at 0 and start list at 1000·. Clear labels
Y0700*I*	Begin *I*-list at 700
Y0·	Begin input of normal instr. at 0·.
—	
—	
—	Instr. of programme
AR3I	*AR*3 is stored at 700 instead of at 1000·.
—	
—	
—	
etc.	

In the library of sub-routines there exists a special programme for introducing *I*'s not previously written. It is a real machine code programme which must be run in somewhere in the real store (say at address *n*). After this programme has been started at *n*, absolute addresses in the instruction store may be dialled and these instr. will then be provided with an *I*. The use of this *I* and its removal follows the same rules as for a normal *I*.

XV. MATRIX SUB-ROUTINES

As illustrative material a set of sub-routines for matrix calculation will be given. In all these routines the essential difficulties lie in the handling of the count instructions.

The simpler sub-routines for reading, printing, or transporting a matrix will not be discussed. In all routines the matrix will be supposed to be square and stored row by row in consecutive locations. The place of the matrix is given by the address of the first location. The location of a_{ik} is $a + ni + k$.

Sub-routine for pre-multiplication of a vector by a matrix

Before entering the sub-routine, the address a of the matrix will be given in 0, the address b of the given vector in 1 and the address c of the resulting vector in 2. The degree of the matrix is taken to the sub-routine in A.

The operation can be described by

$$(c+i) = \sum_{k=0}^{n-1} (a+ni+k)\,(b+k) \qquad i = 0(1)n-1 \qquad \begin{array}{c|c} 0 & a \\ 1 & b \\ 2 & c \end{array}$$

```
Q2 | Y020·2Q3  Assign label 3 to working registers
   |           at the end
   | Y2        Start input at 2
   | X03       Place return instr.
   | T1·3      Place degree n → 1·3. Clear A
   | —00       a → α
   | U02·3     Interchange α and β
   | —002      c → α  (β) = a
   | —01·3  }  Prepare i-count to n
   | Z20    }  c → α c + n → ε  ret. instr. → θ
 → | U03·3     Store i-count in 3·3  a → α  c → β
   | U02·3     Interchange α and β.
   |           Later c + 1 → α    a + ni → β
   | +0001     1 → γ  as increment of β-count
   | —001      b → α
   | —01·3  }  Prepare k-count to n
   | Z20    }  b + k → α  a + ni + k → β
   |           k = 0(1)n—1
 → | KR     }  Form (b + k) (a + ni + k) and
   | VRR    }  add to sum in A
 — | +1        Count with 1 in α and β. When count
   |           is ready (β) = a + ni + n =
   |           = a + n (i + 1)
   | U02·3     Interchange counts a + n (i + 1) → α
   | H03·3     Bring back i-count   c + i → α
   |           a + n (i + 1) = β
   | TR        Store Σ in c + i     i = 0(1)n—1
 — | +1        Count i with 1 in α only
   | Z         = location 20·2 = 0·3. Place of
   |           return instr.
   |           = 1·3 working register for n
   |             2·3 } working registers
   |             3·3 } for storing counts
   | Y04·3     Keep working registers free
   | A ≠       Breakpoint for next programme
```

Sub-routine for transposition of a matrix

The operation can be described by

$$\begin{aligned} &(a + ni + k) \rightarrow a + nk + i \\ \text{and}\quad &(a + nk + i) \rightarrow a + ni + k \\ &\text{for } i = 1(1)n-1 \qquad k = 0(1)i-1 \end{aligned}$$

The elements for which $i = k$ are left undisturbed. This is the main diagonal. In this example we have an outer count from 0 to n and an inner count with a number of times, equal to the outer count. By a count $i = 1(1)n-1$ is meant a count running from 1 to $n-1$ with 1 at a time. For the limit n the count leaves the loop so that $i = n$ is not included in the process. a will be given in α and n in A when jumping to the sub-routine.

```
Q2  | XRRR03     Jump over the working registers
----+--------    and do not destroy (τ)
    |            1·2 used for n
    |            2·2 used for a
    |            3·2 used for storing a + ni count
    |            4·2 used for storing i, floating
    |            5·2 used for i-count
    |            6·2 working register
    | Y07·2      Leave working registers free and
    |            proceed input at 7·2
    | X04        Store return instr. in label 4
Q3  | —RR2·2     a → 2·2      (α) = a
    | U1·2       n → 1·2
    | U06·2      a → β
    | —01·2      Prepare i-count to n
    | —0001·2    n → γ
→Q5 | XRRR06     Jump to count instr. and return
    | --------   when not ready.  (α) = i
    |            (β) = a + ni    i = 1(1)n—1
    | —RR4·2     i → 4·2 floating
    | U05·2      i-count → 5·2  a + ni → α  i → β
    | U03·2      a + ni count → 3·2  i → α
    |            a + ni → β
    | —002·2   } Add a to α count
    | Z20      } Thus: a + i → α
    | U06·2      a + ni → α  a + i → β. Storing
    |            in 6·2 is not really necessary
    |          } Prepare k-count to i for inner cycle
    | —04·2    } ( ) = n  a + ni + k → α
    | Z20      } a + i + nk → β  k = 0(1)i—1
  → | HR       }
    | U6·2     } (a + ni + k) → 6·2 temporarily
    | HRR      }
    | UR       } (a + i + nk) → a + ni + k
    | H6·2     }
    | URR      } Old (a + ni + k) → a + i + nk
  — | +1         Count k with 1 in α, n in β
    | - - - - -
    | H03·2      a + ni → α
    | H05·2      i-count → α   a + ni → β
 —— | X5         Return to beginning of outer cycle.
    | --------   Counting is done by XRRR06
Q6  | +1         Count i. When not ready, return;
    |            when ready, proceed to next instr.
Q4  | Z          Return to main programme
----+--------
Q02 | Q03 Q04    Q05 Q06    Clear all used labels
    | A ≠        Break point
```

Sub-routine for the multiplication of two matrices

A multiplication of two matrices can be described by

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik}\, b_{kj}$$

Suppose that the address of the first location of a_{ik} is a, the address of the first location of the second factor is b and the address of the first location of the product is c. Then we can rewrite the process as:

$$\sum_{k=0}^{n-1} (a + ni + k)\ (b + j + nk) \rightarrow c + ni + j$$

for

$$i = 0(n)n^2 - n$$
$$j = 0(1)n - 1$$

We shall suppose that $(0) = a$, $(1) = b$, $(2) = c$ and $(A) = n$ on entering the sub-routine.

```
          Q2 | XRRR03   Jump over working registers and
     --------| ------   do not destroy τ              0 | a
             |          1·2 used for n                1 | b
             |          2·2 used for n^2              2 | c
             |          3·2 used for c-a-b-n-n^2
             |          4·2 used for storing(a+ni)count
             |          5·2 used for storing(b+j)count
             | Y06·2    Leave working registers free and
             |          proceed with input at 6·2
          Q3 | X04      Store return instr. in label 4
             | U1·2     n → 1·2
             | V1·2     Form n^2
             | U2·2     n^2 → 2·2
             | Z21      Form — n^2
             | S1·2    ⎫
             | S1      ⎪
             | S       ⎬ Form c-a-b-n-n^2
             | A2      ⎪
             | T3·2    ⎭ Store this → 3·2. Clear Acc
             | — 00     a → α
             | — 02·2  ⎫ Prepare i-count to a + ni → α
             | Z20     ⎭ i = 0(n)n^2 — n
 ┌─────────→ | U04·2    Store (a + ni) count → 4·2
 │           |          a + ni → β
 │           | — 001    b → α
 │           | — 01·2  ⎫ Prepare j-count to b + j → α
 │           | Z20     ⎭ j = 0(1)n — 1  (β) = a + ni
 │  ┌──────→ | U05·2    Store (b + j) count → 5·2
 │  │        |          a + ni → α     b + j → β
 │  │        | — 0001·2 n → γ
 │  │        | — 01·2  ⎫ Prepare k-count to a + ni + k → α
 │  │        | Z20     ⎭ b + j + nk → β  k = 0(1)n — 1
 │  │  ┌──→  | KR      ⎫ Form Σ (a+ni+k) (b + j + nk)
 │  │  │     | VRR     ⎬ When the count is ready
 │  │  └───  | + 1     ⎭ (α)=a + ni + n (β)=b + j + n^2
 │  │        | — 003·2 ⎫ Add (3·2) to α-count giving
 │  │        | Z20     ⎭ (α) = c-b-n^2 + ni
 │  │        | TRRR     Store Σ in location mentioned in
 │  │        |          (α) + (β) = c + ni + j
 │  │        | H04·2    a + ni → α
 │  │        | H05·2    (b + j) count → α  a + ni → β
 │  └─────── | + 1      Do j-count
 │           | H04·2    (a + ni) count → α
 └────────── | — 1·2    Do i-count with n at a time
          Q4 | Z        Return to main programme
         Q02 | Q03 Q04  Clear labels 2, 3 and 4
             | A #      Breakpoint
```

Sub-routine for the inversion of a matrix

The sub-routine treated here will be able to invert a matrix in the same locations as the matrix occupies without using any working registers outside the sub-routine. It is a straight forward elimination method derived from the Jordan process. No special care has been taken to safeguard the programme against ill-conditioned or degenerate matrices. In critical cases a better process could be devised, if for example the largest element of a row would be taken as pivotal element in the condensation.

The location of the first element of the matrix must be given in α, the degree of the matrix in A. The programme will return with *det* $[a]$ in the *Acc*.

The process used can be described as follows:

Be the given matrix a^{0}_{ij}. A sequence of intermediate matrices a^{k}_{ij} is formed. At last a^{n}_{ij} will be equal to $[a^{0}_{ij}]^{-1}$.

The pivotal element of the intermediate matrix k is called b_k. The leading element of each row is called c^{k}_{i}.

The process is:

For $k = 0(1)n-1$: $a^{k}_{k,0} \rightarrow b_k$

For $j = 0(1)n-1$: $a^{k}_{k,j+1}/b_k \rightarrow a^{k+1}_{kj}$ where $a^{k}_{kn} = 1$

For $i = k+1(1)n-1$ and $i = 0(1)k-1$: $a^{k}_{i,0} \rightarrow c^{k}_{i}$

For $j = 0(1)n-2$: $a^{k}_{i,j+1} - c^{k}_{i}\ a^{k+1}_{kj} \rightarrow a^{k+1}_{ij}$

For $j = n-1$: $-c^{k}_{i}/b_k \rightarrow a^{k+1}_{i,n-1}$

During the outer cycle *det* $[a]$ is accumulated according to

$$det\ [a] = \prod_{k=0}^{n-1} b_k$$

The programme now reads

$\begin{array}{c|c} \alpha & a \\ A & n \end{array}$

```
Q2   L2·2        Temporary programme for reading constant 1
     L
     Y2Y00       Execute temporary programma and read 1 → 2·2
     Y2          Start again at label 2
     XRRR03      Jump over working registers and do not destroy (τ)
     ------
                 (1·2)  used for a
                 (2·2)  = 1
                 (3·2)  used for n
                 (4·2)        $b_k = a^k_{k,0}$
                 (5·2)        $n - 1$
                 (6·2)        $n^2$, later $(a + nk)$ count
                 (7·2)        $nk$ floating
                 (8·2)        $n(n - k - 1)$
                 (9·2)        $-c^k_i = -a^k_{i,0}$
                 (10·2)       $i$-count
                 (11·2)       $\Pi b_k$
     Y012·2      Leave working space free and proceed to input at 12·2
Q3   X04         Store return instr. in label 4
Q03  —RR1·2      $a \to 1·2$        Clear label 3
     U3·2        $n \to 3·2$
     H2·2        1 → 11·2 as starting value for the accumulation of det [a]
     U11·2
     H3·2        Form $n - 1$
     S2·2
     T5·2        $n - 1 \to 5·2$
     V03·2       Form $n^2$
     U6·2        $n^2 \to 6·2$
     +00
     U04·2       Clear $\beta$.  4·2 used as rubbish dump
     —001 2      $a \to \alpha$
     —06·2       Prepare outer count to $a + nk \to \alpha$      $k = 0(1)n - 1$
     Z20         $nk \to \beta$     ($nk = 0$ initially)
  →  +000        $0 \to \gamma$
     U06·2       Store $(a + nk)$count → 6·2   $a + nk \to \beta$   $nk \to \alpha$
     —RR7 2      $nk \to 7·2$ (floating)
     U04·2       $a + nk \to \alpha$      $nk \to \beta$
     S3·2        Form $n(n - k - 1)$
     U8·2        $n(n - k - 1) \to 8·2$
     HR          $a^k_{k,0} = b_k = (a + nk) \to 4·2$ as pivotal element
     U4·2
     V11·2       Accumulate $\Pi\ b_k$ in 11·2
     U11·2
     —05·2       Prepare count $a + nk + j \to \alpha$      $j = 0(1)n - 2$
     Z20
  →  HR1
     D4·2        Form $a^k_{k,j+1}/b_k \to a^{k+1}_{kj}$
     UR
Q3   +1          Label 3 is issued again
                 When finished, the count reads $(\alpha) = a + nk + n - 1$      $(b) = nk$
     H2·2
     D4·2        $1/b_k \to a^{k+1}_{kn}$
     UR
     —0001·2     $a \to \gamma$
     XRRR03      By executing $a + 1$ instr· $a + nk + n \to \alpha$      $a + nk \to \beta$
     ------      As $(\alpha) \neq$ count limit $a + nk + n - 1$ the + instr. returns to the instruction
                 following XRRR03
```

```
Q03   |           Clear label 3
      | — 08· 2   If $n(n-k-1) > 0$: prepare count $a+ni$ with $i = k+1(1)n-1$
      | - - - - - -
      | + 0       Skip next instr.
      | X5        If $n(n-k-1) = 0$: there is no count $i = k+1(1)n-1$
      | ______
Q3    | Z20       Form $a+nk+n+ni' = a+ni$   $i = k+1(1)n-1$ or form count $a+ni$
      |           $i = 0(1)k-1$ when coming from X3
      | HR      } Extract leading element of row $-c_i^k = -a_{i,0}^k$
      | Z21     }
      | U9· 2     Store $-c_i^k \rightarrow$ 9· 2
      | U010· 2   Store $(a+ni)$ count $\rightarrow$ 10· 2   $a+ni \rightarrow \beta$   $a+nk \rightarrow \alpha$
      | + 0001    $1 \rightarrow \gamma$
      | — 05· 2 } Prepare count $a+nk+j \rightarrow \alpha$   $a+ni+j \rightarrow \beta$
      | Z20     } with $j = 0(1)n-2$.   Count limit $= n-1$
      | HRR1    } Inner cycle
      | KR      }
      | V9· 2   } Form $a_{ij}^{k+1} = a_{i,j+1}^k - c_i^k\ a_{kj}^{k+1}$ and store this over old $a_{ij}^k = (a+ni+j)$
      | URR     }
      | + 1       $(\alpha)+1 \rightarrow \alpha$   $(\beta)+1 \rightarrow \beta$
      | - - - - - -
      | H9· 2   } Form for $j = n-1$: $-c_i^k / b_k$
      | D4· 2   }
      | URR       and store this over old element $a_{i,n-1}^k$
      | H06· 2    $a+nk \rightarrow \alpha$
      | H010· 2   $a+ni \rightarrow \alpha$   $a+nk \rightarrow \beta$
      | — 3· 2    $(\alpha)+n \rightarrow \alpha$. After completion of count $i = k+1(1)n-1$:
      | - - - - - $(\alpha) = a+n^2 > a+nk$
      |           After completion of count $i = 0(1)k-1$: $(\alpha) = a+nk$
      | U010· 2   $a+nk \rightarrow \alpha$   10· 2 is used as a dummy store
      | XRRR06    Test which $i$-count has ended on a + order
      | ______
      | U010· 2   $a+nk \rightarrow \beta$ again
Q5    | — 001 2   $a \rightarrow \alpha$
      | — 07· 2 }
      | - - - - - - } If count limit $nk > 0$: prepare $i$-count with $i = 0(1)k-1$ and re-enter $i$-cycle
      | X3      }
      | ______
      | + 0       If $nk = 0$: skip second $i$-count
Q6    | +         After completion of $i = k+1(1)n-1$ and $(\alpha) \neq$ count limit: return to instr.
      | - - - - - - following XRRR06
      |           After completion of $i = 0(1)k-1$ and $(\alpha) = a+nk$: next instr. is executed
      | — 007· 2  $nk \rightarrow \alpha$
      | H06· 2    $nk \rightarrow \beta$   restore $(a+nk)$ count in $\alpha$
      | — 0003· 2 $n \rightarrow \gamma$
      | H8· 2     $n(n-k-1) \rightarrow A$
      | — 3· 2    $a+n(k+1) \rightarrow \alpha$   $n(k+1) \rightarrow \beta$
      | - - - - - Execute and test $k$-count
      | H11· 2    $det\ [a] \rightarrow A$
Q4    | Z         Return to main programme
        ______
Q02   | Q03 Q04 Q05 Q06   Clear labels
      | A #       Break point
```

APPENDIX 1

THE PLACING OF SIMPLE CODE IN THE MACHINE STORE

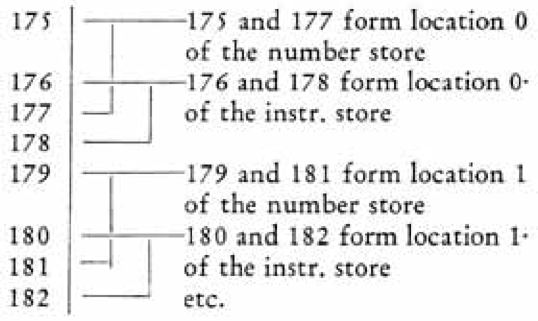
For advanced programmers it is necessary to know how the Simple Code interpretation programme is placed in the real machine code store, as well as where the object programme resulting from a source programme in Simple Code language is put. For all details about normal machine code the reader must be referred to the ZEBRA handbook, issued with every machine.

The Simple Code programme consists of several parts, i.e.:

a. The operation part, in which the different types of operations are executed.
b. The input part, by which instructions are taken in.
c. The *I*-part, which functionally belongs to the input part.
d. The *L*-part, in which all numbers are read.
e. The *P*-part, in which all output of floating numbers is effected. (*P*, *P*0, Z8, Z9, Z22, Z23).
f. The Z-list, in which the reference addresses for all Z-sub-routines are kept.

These 6 parts just fit into two blocks of the machine store. Normally they will be placed from 6144-7168. The placing of the 6 parts relative to each other is rather irrelevant except for the Z-list. This Z-list has been placed at the end of the two blocks so that in the next block 7168-7680 there is still room for more Z-instr. As this next block is filled with normal output programmes starting in 7368 there is space for about 250 more Z-orders. (Often the Print Store programme is kept on 7168 but this is not necessary.)

The Simple Code instruction store and number store are imbedded in the following way in the real machine store:


175
176
177
178
179
180
181
182
175 and 177 form location 0 of the number store
176 and 178 form location 0· of the instr. store
179 and 181 form location 1 of the number store
180 and 182 form location 1· of the instr. store
etc.

In the number store the first component is always the mantissa, the second component the exponent. Furthermore there is a list of labels. Label *p* is put into 75 + *p* in the machine store. Some machine addresses between 50 and 75 are used as live drum working registers for Simple Code.

It now follows how large the capacity will be when the basic Simple Code is in the machine. We find that there are [¼(6144 − 175)] = 1492 instr. addresses and 1492 number addresses. When this is not enough the whole Simple Code programme can be moved further back in the store. In that case normal input in block 7680-8192 can be removed altogether. (It is nowhere needed in Simple Code, except for teleprinter code input.) The standard output, normally in 7424-7680, can be moved up to 7936-8192 and Simple Code can ultimately moved up to 6944. Now the capacity in number and instruction store becomes [¼ (6944 − − 175)] = 1692.

When it is necessary to extend the list of labels over 100 it will be clear how this can be done. In fact the first component of 0 will be the same as 75 + 100. The easiest way to make a few more labels available is to use 1·, 2·, etc. of the instruction store.

1· = 176 and 178 in machine store =
= label 101 and 103
2· = 180 and 182 in machine store =
= label 105 and 107 etc.

These labels must be cleared separately with Q0101 etc. and can then be used in the normal way.

Until now only the basic Simple Code programme was considered without the functions. The standard function programmes such as for the Z1, Z2 etc. instr. and for teleprinter code input are organised in a special way. They all belong to the family of retrograde subroutines which are put into the machine, filling the store from back to front. In this way only the strictly necessary sub-routines could be fed in and a maximum free space is left for number and instruction store. Normally when there is room enough the following sub-routines are permanently left in the store in the following retrograde order. (The first routine mentioned is highest up in the store.)

Teleprinter code input:

Z1
Z2
Z3, Z10, Z24
Z4, Z5
Z6
Z11 to Z15
Z26

In general two complete blocks are reserved for them from 5120-6144. This leaves a capacity of number and instr. store of [¼ (5120 − 175)] = = 1271.

For the actual location and coding we can again best refer to the handbook of sub-routines.

The point facility can now be treated somewhat more fully than was previously possible. A point after the instr. adds unity (in machine code) to the address. Hence it shifts a number address $175 + 4n$, $177 + 4n$ into the corresponding instruction address $176 + 4n$, $178 + 4n$. It is possible to add more points. The effect will still be to shift up the address by unity. E.g. $H1\cdot\cdot$ will take 181 and 183 instead of 179 and 181. This is just the exponent of (1) as mantissa and the mantissa of (2) as exponent. In this way fixed point operations can be done on the mantissae of numbers by the *D*0 and *D*00000 instructions. In the same way a triple point can be used. $H\cdot\cdot\cdot\cdot = H1$

Something must still be said about the use of the fast access working registers in Simple Code.

During execution of a programme the short registers are occupied as follows:

(4) =	working register for miscellaneous purposes	
(5) =	ditto	Mantissa of (n) after *Kn* instr.
(6) =	Usually XK3BD-X003B2. Modifier for relative instr.	
(7) =	ALR for multiplication	
(8) =	Working register. Exponent of (n) after *Kn*.	Return instr. after *ALR*
(9) =	δ-register. Contains 4-fold	
(10) =	α-register. Contains 4-fold	
(11) =	β-register. Contains 4-fold	
(12) =	Accumulator	Mantissa
(13) =	Accumulator	Exponent
(14) =	Extraction instr. of the form *ACE* 176 + 4 *m* for extr. of next instr. from $m\cdot + 1$.	
(15) =	Usually XKCDE002. Modifier for extraction process.	

APPENDIX 2

LINKS BETWEEN SIMPLE CODE AND NORMAL CODE

It is not possible to do everything in Simple Code. For example printing text cannot be done in Simple Code. So the experienced programmer will sometimes use Simple Code and normal code together and he must be able to transfer the control from normal to Simple Code and back.

One way of entering Simple Code from normal code is by using X38P, in the following way:

X38P: Jump to Simple Code instruction in the location of which the address in the instruction store is mentioned in B.
A return instruction is kept to be used by Z19.

Example:

```
N
NKKBC } 140 → B
+ 140 }
X38P    Jump to instr. in Simple Code in
        instr. store address 140·
        From Simple Code we can return
        here.
```

The same instruction X38P can be used for jumping to a label p in Simple Code. In that case $-p$ must be put into B before using X38P.

> X38P: When $(B) = n > 0$: jump to address n in the instruction store.
> When $(B) = p < 0$: jump to label p.

The way to return from Simple Code to normal code is by Z19.

> Z19: Return to normal code on instruction following *X38P*.

The Z19 can only effect return to normal code when first normal code has entered Simple Code by an X38P instruction.

A way of transferring control from Simple Code to normal code without a previous jump of normal code to Simple Code can be effected by a specially made Zn, where $n > 32$. (All Zn with $n \leq 32$ have a fixed meaning.) The corresponding outlet in the Z-list must then be filled with a normal jump to the normal code programme. Such a transfer is very fast. As long as $(14) = \text{ACE}m'$ is not destroyed, the way back to interpretation in Simple Code can be made by X43P.

> X43P: Resume interpretation on Simple Code after last Simple Code instruction, which jumped to normal Code.

A third way of transferring control from Simple Code to normal code is by:

> UR0: Jump to machine code address n, when $(\alpha) = n\cdot \quad (\delta) \rightarrow \alpha$

Example:

```
+ 0020     20 → α
+ 004133 } Jump to machine code address
UR0      } 4133, but restore 20 → α
```

A return to Simple Code can be made by giving the instr. X43P. Control returns to the Simple Code instr. following UR0.

Of course UR0 is useful when the address in *a* is variable. When it is fixed another way of jumping to normal code is:

Q000*n*:	Jump to normal code address *n*. Return can again be made by X43P.

Not only the control must be transferred from normal code to Simple Code, but often also numbers must be carried to Simple Code. These numbers are in fixed point form in normal code, but must be converted to floating form in Simple Code. The following order is present for this conversion:

Z24:	Convert the number in short registers (12. 13 5) into a floating number to be placed in 12 and 13 (= *A* in Simple Code)

The number to be converted can have one length before the point in 12 and one length after the point in 13. We then say that in a double length number the point is in the middle. (5) does not matter in that case. Or it can be a double length number, point to the left. In that case the sections are placed in 13 and 5 and 12 must contain all zeros or all ones dependent on the sign + or − resp. In any case only the nine most significant digits will be converted to floating point and placed in the floating accumulators 12 and 13.

Example:

```
NE13         pre-instr.
NKKC       }
head of    } take head of number before point
number     }
NKKCE12    } and place this in 12
tail of    } place tail in 13
number     }
NKKBC      } jump to instr. 15 in instr. store
+ 15       } of SC
X38P       }
```

This instruction can then read:
(15·) = Z24: convert (12. 13 5) to floating → *A*
Z24 is a part of the log, ln sub-routine Z3, Z10.

The reverse operation is effected by:

Z25:	Convert (*A*) floating into a triple length number in (12. 13 5)

This instruction has not been built in yet. It will be a part of a new sub-routine for sinh and cosh.

In the same manner as Q000*n* transfers control to normal code address *n* as an instr. there exists an input indication:

Q00n:	Jump to machine code address *n* as soon as this indication is read during input

Note carefully the difference between Q000*n* being an instruction in the store and Q00*n* being an input indication.

APPENDIX 3

SIMPLE CODE 1½ LENGTH

A precision of 9 decimals in the mantissa is not always enough. As in the normal Simple Code (abbreviated SC) the exponent part of a number is only used for exponents between − 1000 and + 1000 there is still room for more precision by taking the tail of the mantissa and putting it into the exponent register. This has been done in Simple Code one and a half length (abbreviated SC 1½).

The code of SC 1½ is equal to the code of SC with the following exceptions:

a. For numbers on tape the reader will accept up to 17 significant digits instead of 10.

b. Numbers printed will appear as:
±0.XXXXXXXX XXXXXXXX Sp±XXX Sp Sp.
Internally the exponent *b* is always
$$-1024 \leqq b \leqq +1023$$
but the print part will automatically limit the exponent to ± 999.

c. Built in sub-routines are:
Z, Z7, Z8, Z9, Z16, Z17, Z18, Z19, Z20, Z21, Z22, Z23, Z27, Z28.
The sub-routines Z1 and Z26 are available in normal code and are consequently very fast but all other function sub-routines are interpreted Z-sub-routines, working in Simple Code themselves and are rather slow. No teleprinter code input, or output in fixed point form with Z29, Z30, Z31 is available.

d. The times of execution are about 1½ times as slow for *A*, *S*, *V*, *N*, and *V*0, *N*0. *D* is still slower. All other times are equal to the corresponding ones of SC. Of course input and output are slower, because there are more characters to read or print. The programme for execution of SC1½ is about 1350 instr. long. Thus without the function sub-routine for Z1 the capacity available in number and instruction store is about 1400 locations.